

Neural Audio Codec Specialised for Speech Audio Compression

Analysis	3
Background	3
The Problem	3
How the problem was researched	3
SoundStream	3
Encodec paper	4
Target User	4
Interview	4
Other Existing Solutions	5
Neural based methods	5
Lyra Codec	5
Encodec	6
Conventional methods	6
Opus	6
Description of PESQ Score	7
Datasets	7
Objectives	8
Documented Design	10
Overall Design	10
The Inference System	11
Encoded file structure	12
User Interface Design	14
Inference system class diagram	16
The Model System	17
The Trainer	17
The Loss System	18
General Structure of the Loss System	18
Discriminator Loss	18
Feature Loss	19
Whisper Loss	19
Reconstruction Loss	19
Quantization Loss	20
Loss Balancing	20
Dataset Loading	22
Network Structure	23
Vector Quantization	26
Forward pass pseudocode	26
Frozen K-Means Pseudocode	26
Residual Vector Quantization	27

Object Oriented Approach	28
Complete Class Diagram	30
Technical Solution	33
Models	33
Algorithms	33
Coding Styles	34
Solution	35
Testing	68
Model Testing	68
Residual Vector Quantization	68
Discriminator Model	70
Moving Average Tests	71
Inference Testing	72
File Testing	72
Inference Testing	73
GUI Tests	74
Model Selection and Stats	74
Encode Decode Widget	75
Full System	76
Evaluation	78
Evaluation Of Objectives	78
User Feedback	82
End User Feedback	82
Reflection on End User Feedback	82
Final Feedback	84
Looking to the future	85
GUI	85
The Model	85
What else could this be used for	86

Analysis

Background

With the recent rise of podcasting thousands of hours of spoken audio are transmitted per day across the internet, and stored in computers across the globe. These audio files are normally encoded with conventional audio codecs, which attempt lossy compression of the sound in a reversible manner, however these codecs are generally unaware of the content of the audio. For example, they cannot distinguish between a spoken word, and the buzz of a fly in the background, this means that much of the bandwidth is taken up through transmission of unnecessary data. Neural network based codecs attempt to help that problem. Where in conventional codecs an algorithm is predefined, in a neural codec the structure of the network is defined, and the weights and biases are learned. This allows significant improvements in perceived quality, with negligible change, or even a decrease, in bitrate.

The Problem

Whilst some neural based audio codecs exist none exist which provide an end-user friendly interface to interact with. This project will: 1. Train a neural based vocoder. 2. Provide an easy to use interface for interacting with this network.

How the problem was researched

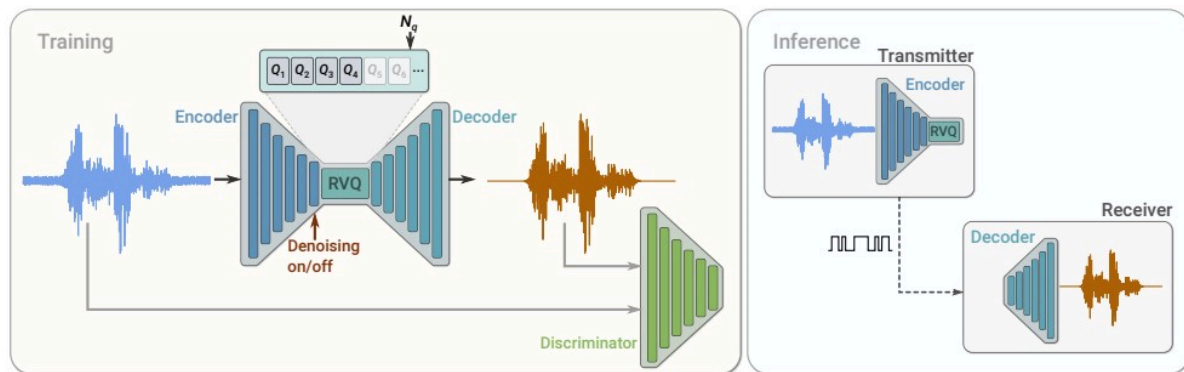
I have researched this problem by looking into some similar problems such as SoundStream and Encodec, these papers by Google and Facebook respectively both suggest methods of encoding and decoding speech efficiently based on a SeaNet architecture, my end project is likely to be a mixture between the architectures suggested in these two papers. Below are the key points from each of these.

SoundStream

<https://arxiv.org/pdf/2107.03312.pdf>

This paper introduces the idea of using a fully convolutional encoder/decoder network for the compression of audio. It further introduces the idea of residual vector quantization as a way to significantly improve the speed of the network whilst maintaining the quality of the result.

The reasons for this modified quantization follow as, due to this reasoning, my product will have to use residual vector quantization. Vector Quantization very quickly becomes infeasible as we increase the bitrate due to the rapidly increasing size of the codebook required. For example, if targeting a bitrate of 3 kbps, with an input sampling rate of 16000 kHz, the size of the codebook would be 2^{50} , which isn't feasible to store or search. This problem is dealt with by residual vector quantization, instead of using one large codebook we split this into multiple quantization layers each applied sequentially to the error of the previous layer. For example 5 codebooks with size 2^{10} . This is much more feasible to store and manipulate.



The above image details the network which sound stream uses, this project is likely to use a similar network structure, however denoising isn't a key needed aspect for this project, so will not be needed.

Encodec paper

<https://arxiv.org/pdf/2210.13438.pdf>

This paper builds on the soundstream paper introducing the idea of variable bandwidth without having to switch between models by "cutting" out residual vector quantization layers. This paper also introduces multiscale discriminators allowing much higher audio quality through an adversarial style of network. Discriminators work by training to decide whether a given input is from the network or non-encoded input, this provides a form of perceptual loss where the network attempts to make the output "sound" like the inputs, however not necessarily keep the content the same. Further to improve reliability of training a loss balancer is used in encodec, this means that the correct weights are always applied to all of the individual losses.

Target User

Jacob Liu who is an avid podcaster who's running out of disk space on his computer.

Interview

What is important to you in a speech codec?

Good quality, as in not overwhelmingly high file size, can run on a computer. fast.

What's more important, emotion or content?

Of course content is more important, however a level of emotion is also needed to get a point across.

What do you prefer, decreased audio quality, or increased file size?

A balance of the two, it would be great to have an option between the two, either increased file size or increased audio quality.

How would it be easiest for you to interact with this product?

I don't always have an internet connection during my podcasts, so it would be useful if it is able to run offline on my Windows laptop.

What potential problems can you foresee with the use of my program?

The content of words has to be maintained, along with "comfort" of listening, the resultant audio can't be too difficult to listen to.

What is the longest podcast that you might want to encode?

Up to 15 minutes.

Other Existing Solutions

Neural based methods

Lyra Codec

Lyra codec is a google designed and implemented codec designed for use in android phones, this codec was initially released in 2020 and hasn't gained widespread use, however it is used in the Google Duo platform. Google has provided an android API which allows for easy compression of speech, however there are no applications for compressing common file formats, such as mp3 and wav files.

Pros	Cons
Speech specialised.	Outdated network structure, structures which allow increased quality now exist. Only provides an android API, difficult to use on other platforms.

Lyra Codec reinforces to me the importance of ease of use, and of using an up to date model structure to ensure that the best possible performance is achieved.

Encodec

<https://github.com/facebookresearch/encodec>

Encodec is based on the Encodec paper, which is based on the SoundStream paper. This is a new network architecture using residual vector quantization, leading to a significant increase in resultant quality. Models trained using this structure are unspecialised, meaning that they will work for speech, but also for music, and most other audio meaning that . Due to using residual vector quantization this model can "cut off" quantizers, significantly reducing the resultant file size, whilst maintaining as much performance as possible.

Pros	Cons
Variable compression ratio.	Hard to interact with - only through the command line. Specialised towards music, lowering performance on spoken audio. Fails to deal with large file sizes.

From this you can see the importance of being able to vary the compression quality to deal with different requirements, such as maintaining speech quality.

Conventional methods

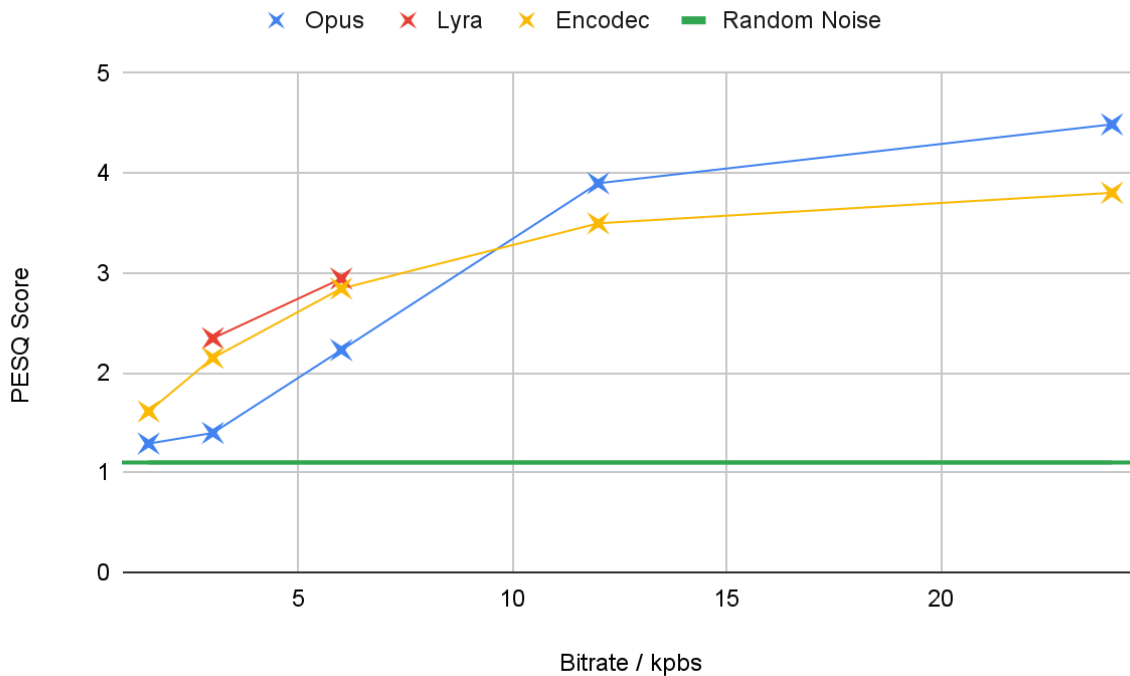
Opus

Opus is a conventional audio codec which was standardised by the Internet Engineering Task Force (IETF), in 2012, this can transmit speech at bit rates as low as 12 kbits/s. However, bitrates can vary.

Pros	Cons
------	------

Widely used, so online applications exist to convert between common file types and opus.

Very low quality when at low bitrates



Description of PESQ Score

Perceptual Evaluation of Speech Quality (PESQ) scores are a measure of the quality of an audio calculated in an objective manner. It is commonly used in the telecommunications industry to detect where audio quality is most severely degraded, with a PESQ score under 2.5 being considered as the level where considerable effort is required to understand the output, and a PESQ score of 1 being akin to random noise. It can be measured in 2 ways, with reference, and without reference, however for these purposes with reference is the most effective method of evaluation as it produces more reliable results.

Datasets

Name	Common Voice	LibriSpeech	Google AudioSet
Length	3,438 hours	1000 hours	5,800 hours
Source	90,000 random speakers	Audio books only	Youtube videos

Licence	Mozilla Public License	CC BY 4.0	CC BY-SA 4.0
---------	------------------------	-----------	--------------

The common voice dataset is ideal for our use, it has both a public licence and a large variety of speakers. Whilst google AudioSet is a varied dataset with a significant number of hours and speakers it is too unwieldy as it would first have to be filtered for non-speech content, and downloading the amount of videos required isn't feasible on my internet connection. The LibriSpeech dataset will also be useful for testing, however overfitting (the model just learning the data) is likely to happen, so likely won't be good enough for the final models.

Objectives

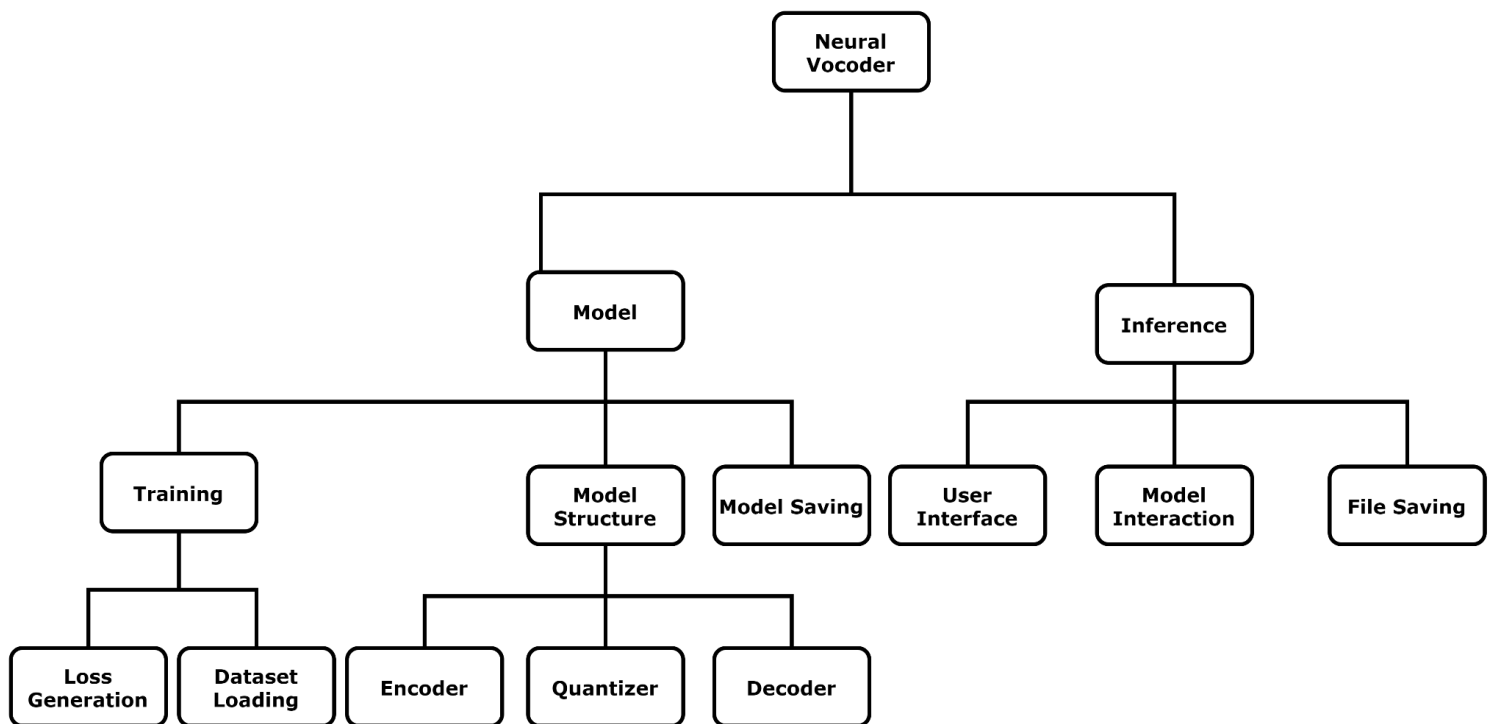
1. The program to encode speech using pytorch
 - 1.1. A network to compress audio to an intermediary form
 - 1.2. A quantization layer
 - 1.2.1. The layer should be able to encode vectors into ids
 - 1.2.2. The layer should be able to efficiently deal with "dead" codebook codes.
2. The program to decode the encoded form
 - 2.1. A dequantization layer
 - 2.1.1. The layer should be able to accurately undo vector quantization.
 - 2.1.2. The layer should be able to deal with variable sized quantization layers.
 - 2.2. A network to decompress audio from the intermediary form into the original form
 - 2.2.1. Resultant audio has to maintain the content of speech
 - 2.2.2. The resultant audio should have a PESQ score of above 1.5 at 3 kbps.
3. The network needs to be trained
 - 3.1. Multiple loss functions need to be designed
 - 3.1.1. Perceptual loss based on existing speech to text models
 - 3.1.2. Perceptual loss based on a separate discriminator model.
 - 3.1.3. Non-perceptual loss based on difference between original and output.
 - 3.2. Trained model needs to be saved
 - 3.2.1. Includes network hyperparameters
 - 3.2.2. Includes the parameters of encoder, decoder and quantizer
 - 3.3. Loss progress needs to be displayed
 - 3.3.1. Training progress should be output to TensorBoard.
4. A PyQT based GUI for a user to interact with the network

- 4.1. The user needs to be able to select a audio file and then output an encoded form
 - 4.1.1. The user should be able to select a model.
 - 4.1.2. Input files should be able to be of wav, mp3 audio formats
 - 4.1.3. Needs to be able to deal with audio files of greater than 15 minutes in length.
- 4.2. The user needs to be able to select an encoded file and output a audio file.
 - 4.2.1. The user should also be able to select an output file type from mp3, wav.
 - 4.2.2. The user needs to select an output file location and name.
 - 4.2.3. A file format for the encoded file needs to be designed.
 - 4.2.3.1. Metadata needs to be saved to prevent errors where differing models are used.
- 4.3. Application needs to be able to run on target customers computer
 - 4.3.1. No need to use a GPU for inference
 - 4.3.2. Supports windows

Documented Design

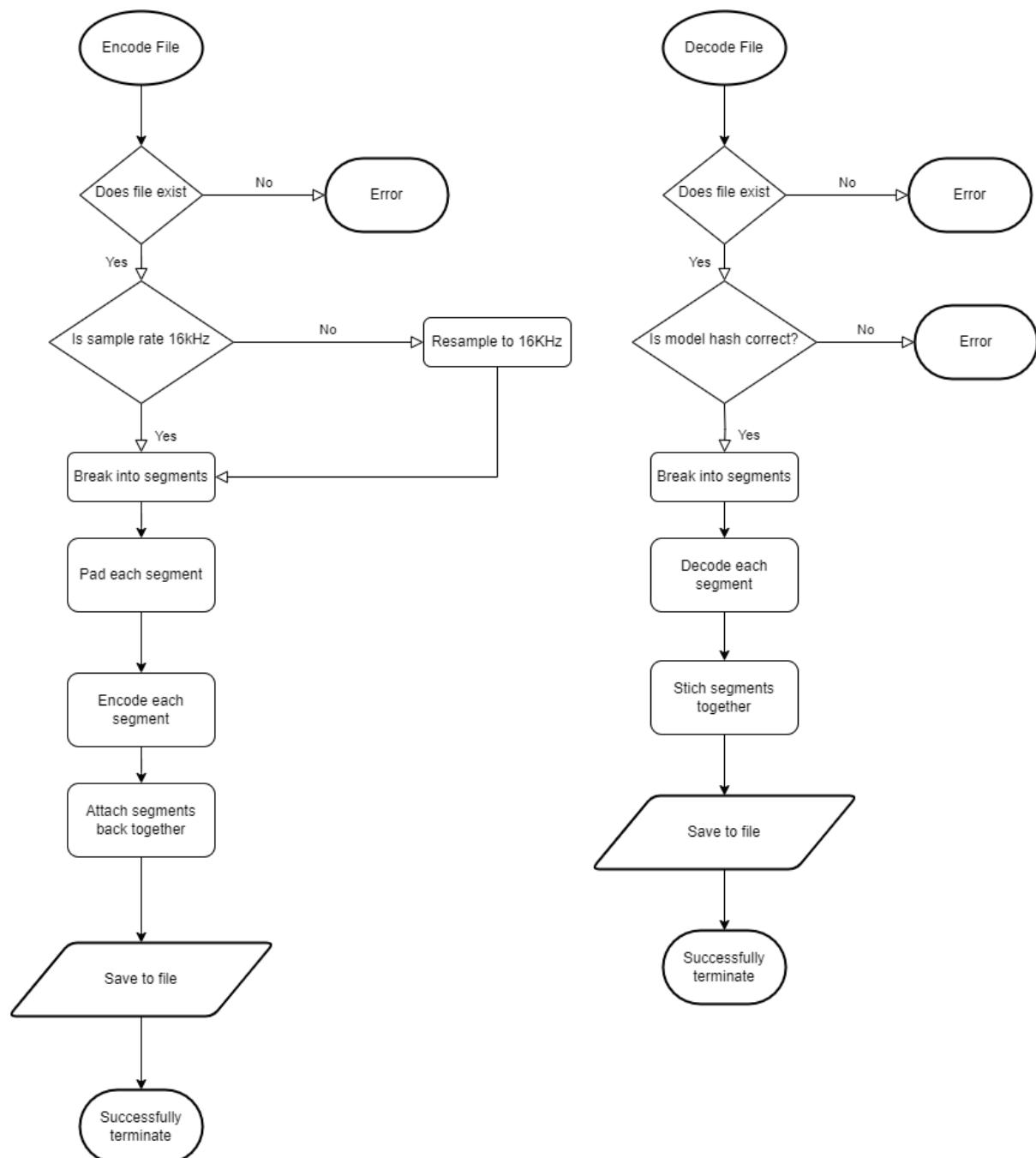
Overall Design

This project can be split into two main sections, the model and the inference system. Whilst the inference system relies on the model, it should only rely on the interface provided of the model. Below is a structure chart representing the different parts I have broken this project down into.



The Inference System

Inference is the process of allowing the user to use the model which of course depends on the model for function, however only interacts through the Model interface allowing for the components inside the model to be switched out when required. The inference system contains three main parts, the user interface, interaction with the model and finally saving the generated files.



This flowchart represents the steps required for our inference system to carry out once the user has instructed us to carry out inference. The structure of both encoding and decoding is

mostly similar. Each of these functions will also throw an error when there is a problem, this can then be caught by the user interface and displayed to the user.

Encoded file structure

Saved Encoded File:

Type	Description
UInt	Magic Number set to 0x12121212
UInt	Length - length of data in frames
UByte	Codebooks - the number of codebooks per frame
UByte	Bit Depth - bit depth of each codebook
16 Bytes	Model hash
UByte	Padding - set to 0
Frame[Length]	Frame array

Frame:

Type	Description
NBit[Codebooks]	Bit depth is given by bit depth from further up in the file.

Types:

UInt - unsigned 32 bit number

UByte - unsigned 8 bit number

NBit - unsigned number with a variable bit depth

A class diagram representing the file saving part of this program:

File	FileReader	FileWriter
+ length: int + n_codebooks: int + bit_depth: int + data: list[list[int]] + model_hash: int + write(str) + read(str): File	# file: File # bytes: list[Byte] # font_pointer: int + read_bit(): int + read_byte(): int + read_n_bits(int): int + read_short(): int + read_32_bit(): int	# file: File # bytes: list[Byte] # font_pointer: int + write_bit(): int + write_byte(): int + write_n_bits(int): int + write_short(): int + write_32_bit(): int

The FileReader and FileWriter act as wrappers to Python's IO library, they allow for the writing and reading of more complicated data types such as shorts, where Python's IO library only works on a byte level.

Another complication with this project is that we want to be able to save a n bit length number efficiently, so we have to operate a "queue" like structure broken up into bytes which can then be saved, with the bytes variable in both the FileReader and FileWriter, and the front_pointer.

The File class represents an "encoded file", it contains all of the required information about a file, allowing for this data to be easily passed around the program later on. It also has the write and read function which interact with FileWriter and FileReader respectively to save its data to a file.

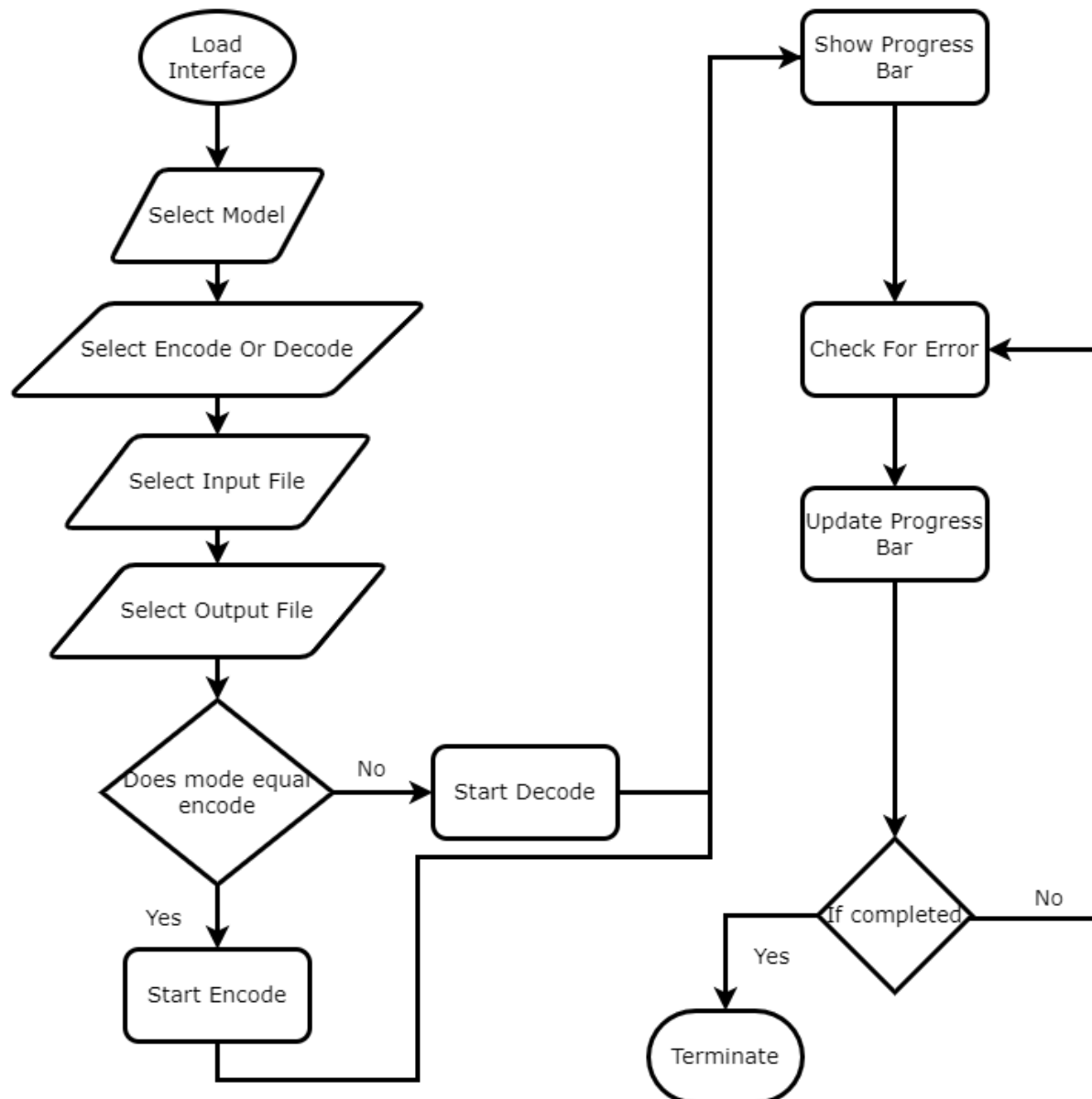
For example here is the pseudo code for reading one bit:

1. if at end of file: error
2. $\text{index_in_byte} \leftarrow \text{front_pointer} \bmod 8$
3. $\text{byte_index} \leftarrow \text{front_pointer} \text{ intdiv } 8$
4. $\text{mask} \leftarrow 0b10000000 \text{ shift right by index_in_byte}$
5. $\text{value} \leftarrow (\text{bytes}[\text{byte_index}] \text{ AND mask})$
6. $\text{value} \leftarrow \text{value} \text{ shift right by } 7 - \text{index_in_byte}$
7. Increment front pointer

The process of writing a bit is the reverse of reading a bit. For dealing with n bits we could just run the above pseudocode many times, however that wouldn't be efficient, so instead we define a custom read_n_bits() function, this function deals with each byte the integer is spread across, allowing us to significantly reduce the calls to read_bit().

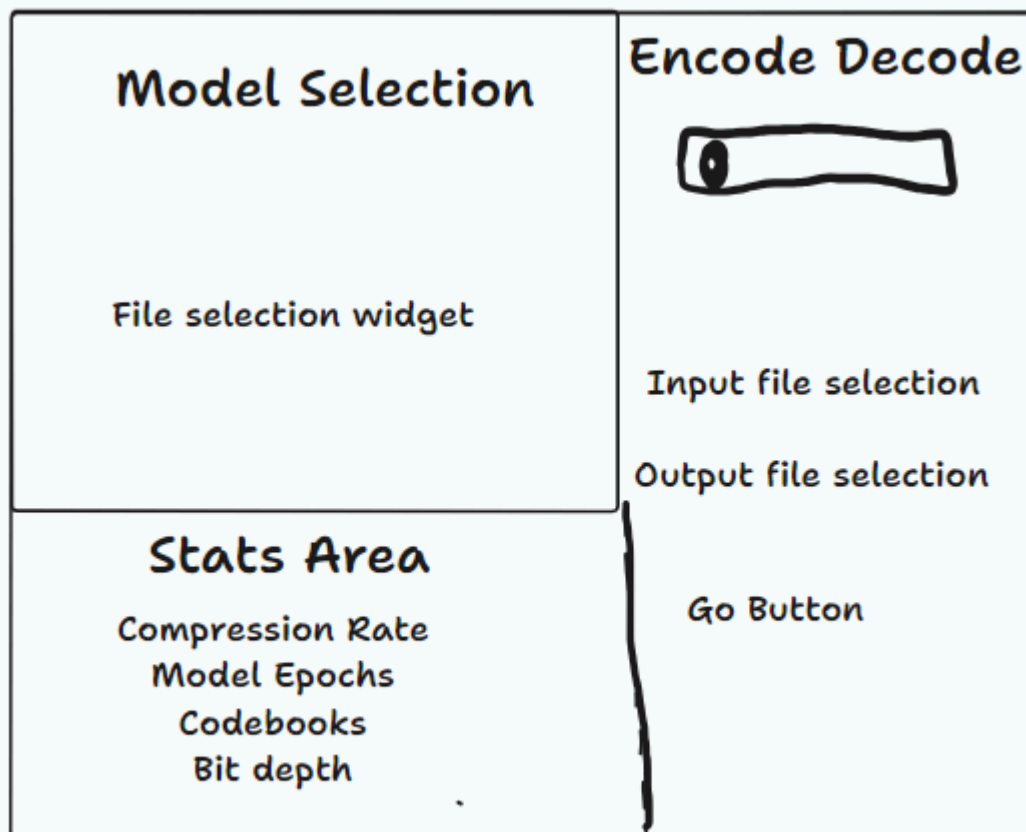
User Interface Design

The user interface was designed with functionality in mind, however it needs to be able to deal with crashes and make intuitive sense for the end user. The user flow looks like this:

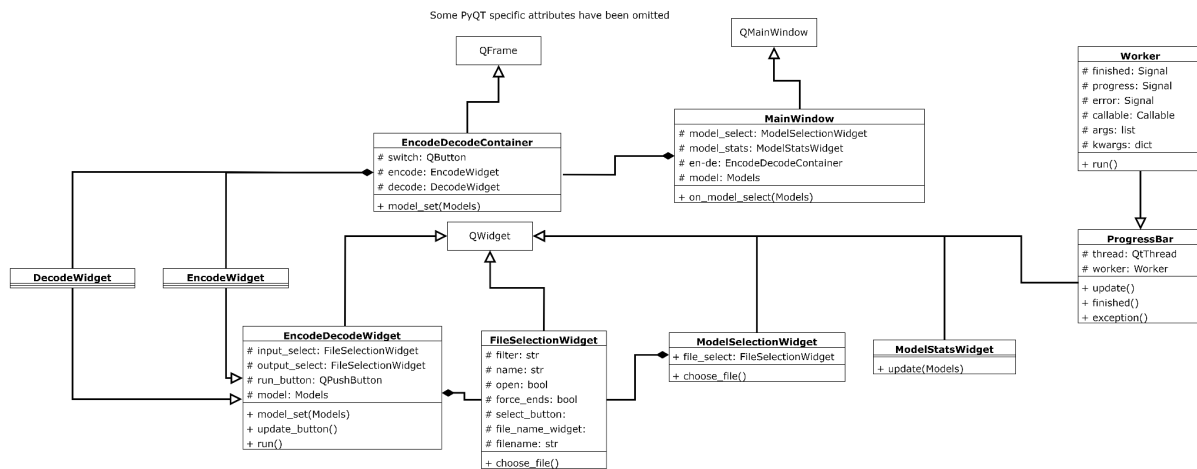


The user first selects the model, selects whether to encode or decode, and finally the input and output files, we can then start the process of encoding or decoding, however with decoding we'll first check whether the correct model is being used using a hash of the parameters of the model.

The user interface is implemented using PyQt6, with the main window subdivided into 3 "sub areas", which are model selection, model statistics, and the encode/decode widget. The encode/decode widget is a widget which can be switched between encode and decode allowing the user to switch between modes at the click of a button. An initial sketch design of the user interface looks like:



Inference system class diagram



Most classes are children of the QWidget class, this represents all of the displayed areas. The DecodeWidget and EncodeWidget are children of the EncodeDecodeWidget, with all functionality defined there and each of these children classes only supplying a slightly modified initialisation function.

As we don't want the UI to freeze whilst encoding/decoding we also need to have the processing being done on a separate thread, this is done through the Worker class, where a worker is a PyQt object, allowing us to make use of PYQT's signals and slots. This enables the progress bar, as the progress signal emits the current percentage progress of the task. The progress bar serves as a way to give feedback to the user what the current state of progress is, as processing can take several minutes.

The Model System

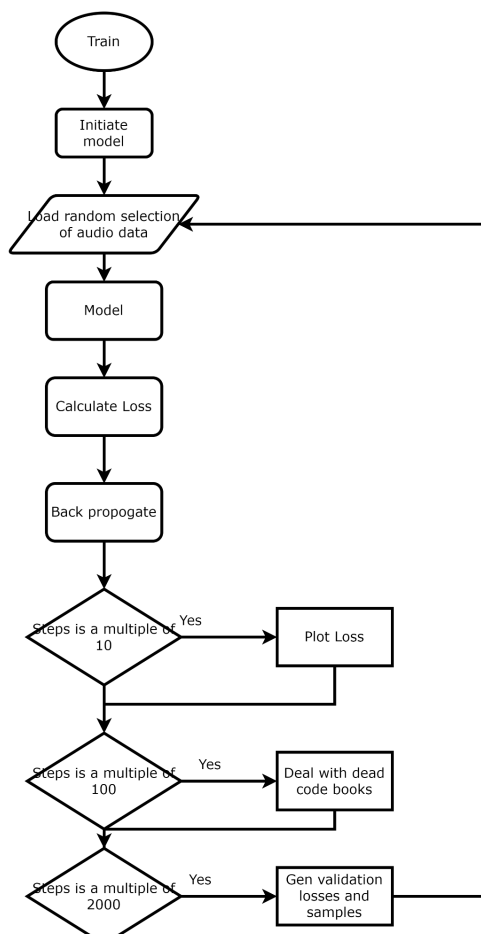
The model system can be further subdivided into two main parts, network structure and training. Whilst training relies to an extent on the model structure, this split is fairly hard and these two areas should only both interact in the Trainer class.

The Trainer

Trainer
<pre># learning_rate: float # discrim_learning_rate: float # betas: (float, float) # train_loader: DataLoader # valid_loader: DataLoader # loss_gen: LossGenerator # steps: int # model: Model # optimizer: Optimizer</pre>
<pre>+ run_epoch(int) + gen_valid_losses() + save_model() + gen_samples()</pre>

The trainer is the class which manages training, not only does it define the optimizers, it manages the training batches, and the logging of losses at a certain point in time. It even allows us to generate samples from a model as we train so that we can audibly check on training progress.

In `run_epoch` we first get data from the data loaders, then run the model on said data, generate the loss, back propagate, and then apply the changes. Once we've done this we carry out steps depending on what step number we're on. This is the main function of the training which is run repeatedly until training has finished.



The overall flow of the training looks like this, with some steps which are always carried out, such as back propagation, and some steps which are only carried out rarely as they are computationally expensive such as validation losses and making samples.

The Loss System

An important feature of the training of every machine learning system is the loss system. This assigns a number to how "well" the model is performing. This number can then be used for backpropagation, where each parameter of the model (each weight and bias) is modified to decrease this number. We care about multiple different factors when it comes to the quality of the resulting model, so multiple loss functions are needed. For this reason we also need a loss balancer, a method of making it so that all of the losses are optimised in proportions of our choice, as without this some "easier" areas to optimise would be improved whilst others left behind.

General Structure of the Loss System

The loss system is composed of two main parts, the deterministic element, and a discriminator element; these two are then combined to create the overall loss.

A discriminator is used to attempt to make the model sound as similar to human speech as possible. This is a different model which is trained to determine whether the given input audio was generated by our sound compression model, or is just the raw audio with no compression. We can then make our model aim to trick this classifier, and that significantly increases the perceived quality of the model. In effect the deterministic side to the loss functions is just "hinting" the model in a direction to go to minimise this loss.

Discriminator Loss

The equation used to determine discriminator loss for the classifier follows this, with x representing the inputs, y representing the outputs, and K representing the number of inputs.

$$L_{dc} = \frac{1}{K} \sum_k \max(0, 1 - D(x_k)) + \frac{1}{K} \sum_k \max(0, 1 + D(y_k))$$

This can be split up into two parts at the plus sign, the first half trains the classifier on what is real, optimising towards 1, and the second half on what has been generated, optimising towards -1.

The equation for training the generator on using the discriminator is similar, however doesn't take into account the original (x) values, and rewards for the opposite of the classifier loss.

$$L_{dg} = \frac{1}{K} \sum_k \max(0, 1 - D(y_k))$$

Feature Loss

Feature loss uses the same discriminator model as mentioned above, however it compares the internal layers of the discriminator instead of just its output, it is defined as followed:

$$L_f = \frac{1}{L} \sum_l \|D^l(x) - D^l(y)\|_1$$

Whisper Loss

This is the only speech specific loss function that will be implemented, this will use a model called Whisper, created and trained by OpenAI to determine how similar the content of the speech is to the original. For this we will use only the encoder side of the model, this translates input audio into a latent space which can then be used by the whisper models decode to translate audio to text, however as we don't need text we can just intercept it half way through this process. We want to minimise:

$$L_w = \|W(x) - W(y)\|_1 + \beta \|W(x) - W(y)\|_2$$

Where W represents the whisper model. However this unfortunately isn't quite as simple as just plugging in the correct numbers as the whisper model expects a context length of 48000 frames, whereas we can be training our model on any number of frames, therefore we have to divide our training batch into appropriately sized batches to run through the whisper model, as our context length is likely to be significantly less than whispers this will also have to involve grouping the examples in our training batch. We can do this through first determining a length which is a factor of 48000 that we can convert all of our examples to, and then calculating the amount of padding needed to add to either side to get each example to that size, finally appending all tensors, and dividing at the 48000 mark, fortunately these last two steps can be done easily and efficiently using the torch.view function.

Reconstruction Loss

Reconstruction loss consists of two factors, time loss, looking at the time dimension, and frequency loss looking at the frequency dimension, these two are weighted and then summed, resulting in an equation which look like:

$$L_r = L_{rt} + \beta L_{rf}$$

Time loss is the much less significant part of this, it is simply defined as the L1 loss between the inputs and outputs of our model, ie:

$$L_{rt} = \|x - y\|_1$$

Frequency loss on the other hand is defined similarly, however done on the frequency domain, so we first have to run the inputs and outputs through a MelSpectrogram, we use a list of different spectrograms to reduce the chance of a frequency not being picked up by this.

$$L_{rf} = \frac{1}{K} \sum_k \|S_k(x) - S_k(y)\|_1$$

Quantization Loss

The final type of loss is quantization loss, this is calculated by the quantization layer as the difference between the outputs and the inputs of that layer. This is used for training the quantizer and the quantizer only, and shouldn't have any effect on the rest of the system.

Loss Balancing

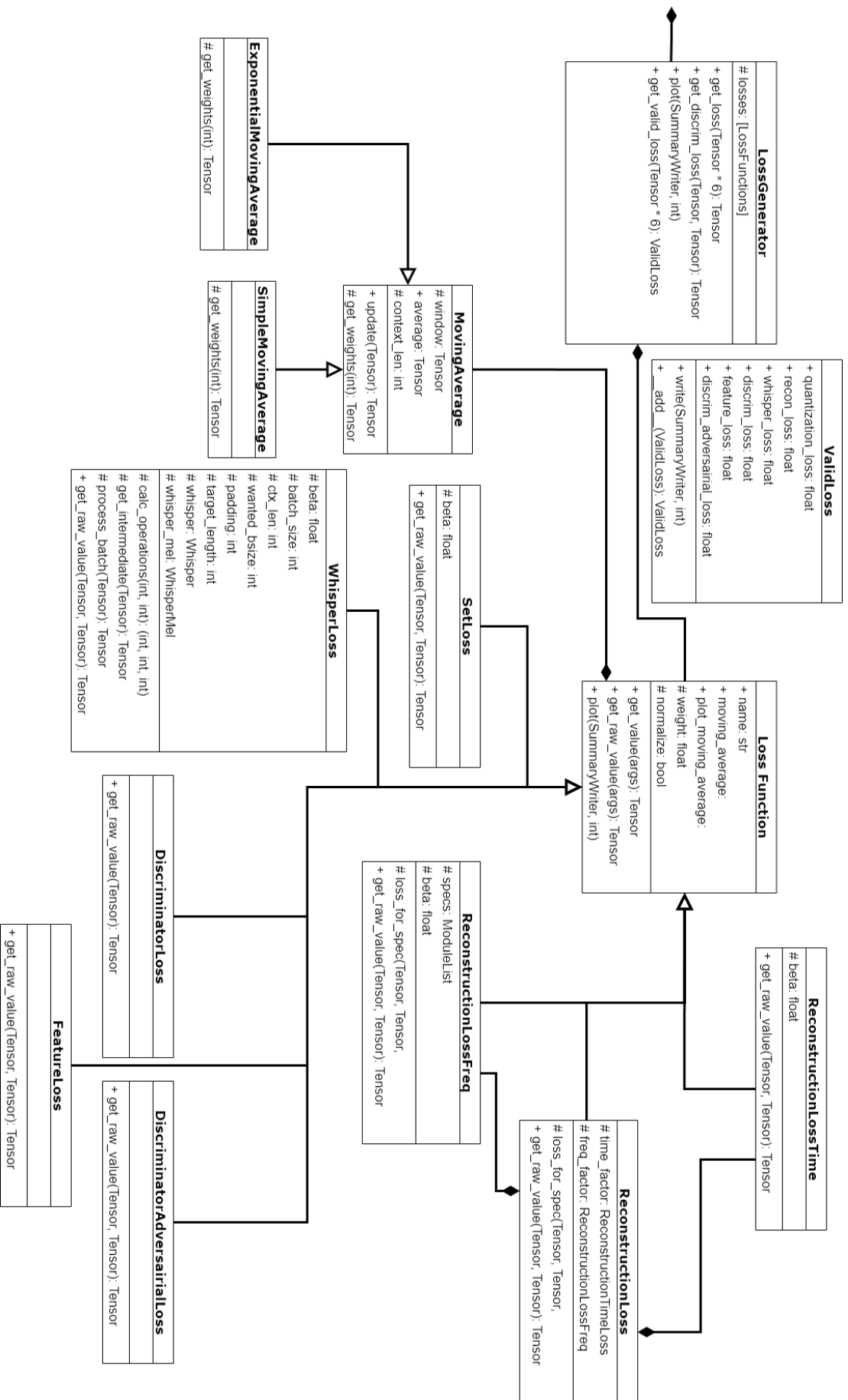
The method of loss balancing used is attempting to make each individual loss be close to one. To do this we divide each loss by the exponential moving average of each loss, i.e. following the equation:

$$\tilde{l}_i = \lambda_i \cdot \frac{l_i}{\langle l_i \rangle}$$

To calculate the total loss we then use:

$$L_G = \sum_i \tilde{l}_i$$

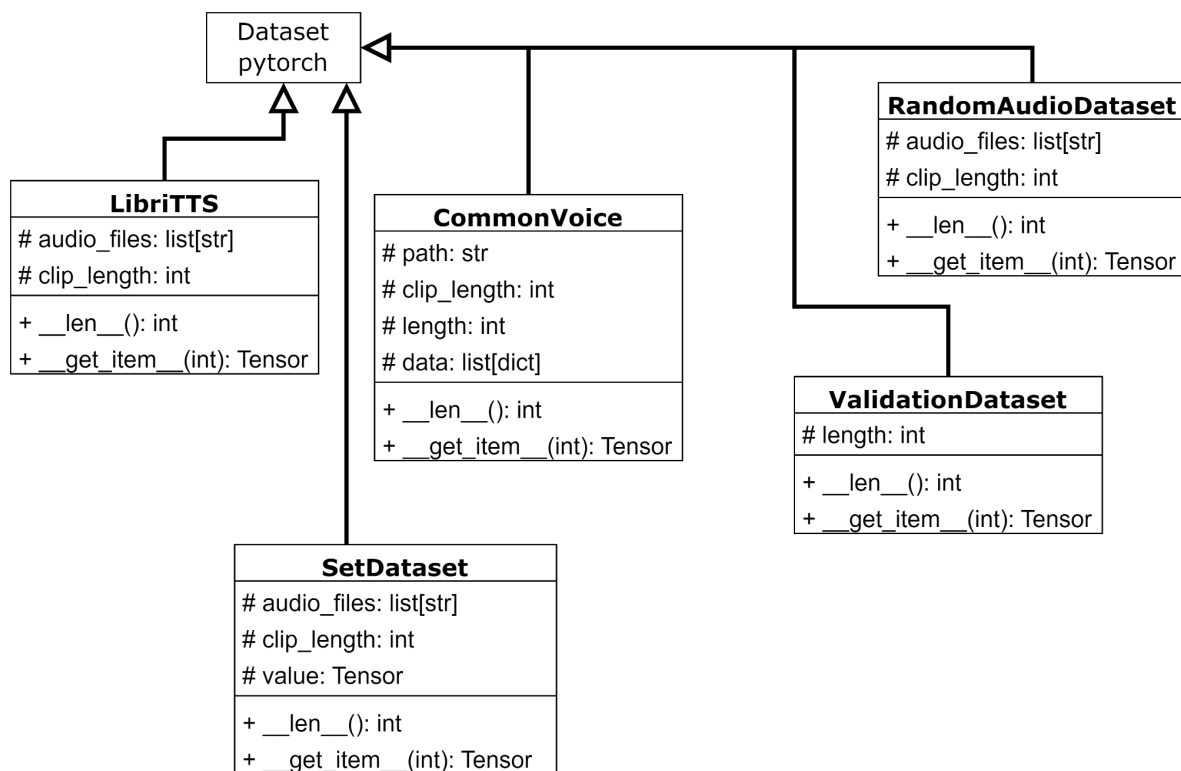
Following is a diagram of the classes used for the loss system. It consists mainly of the children of two classes, LossFunction, and MovingAverage, with LossGenerator wrapping this whole system into a simple interface. There is a child class for each loss function defined above with parent LossFunction, two moving average classes, representing 2 different types of moving average, simple moving average and exponential moving average. The LossFunction class implements the loss balancing method written above, with an abstract method get_raw_value, which is left for each child class to implement.



Dataset Loading

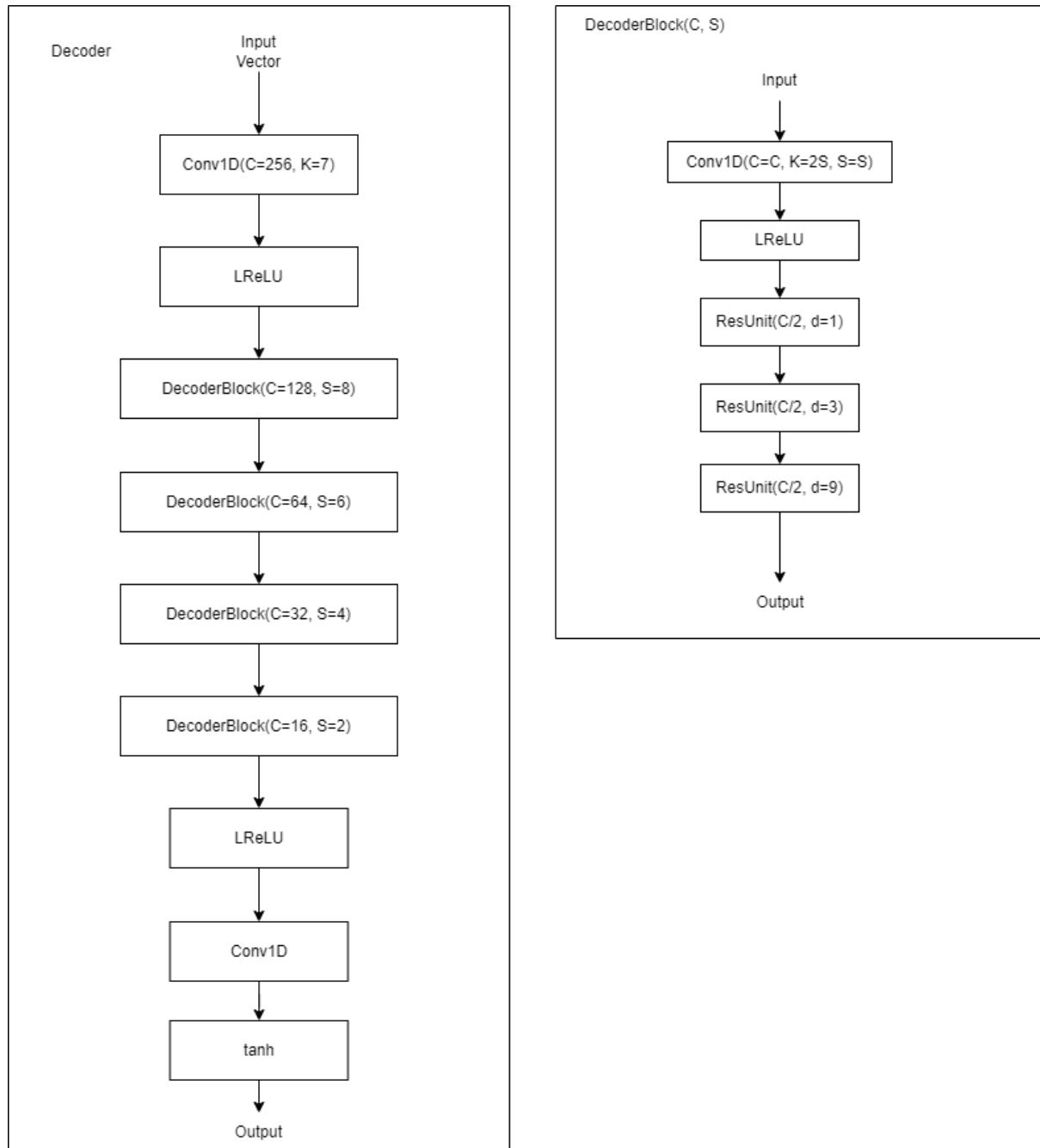
This project supports training from multiple datasets, with some datasets leading to higher quality at specific tasks and some being more general. Each dataset comes with a slightly different format, however to reduce computation whilst carrying out training each dataset is stored on disk as a directory of Waveform Audio File Format (wav) files. Supported datasets for full training runs are LibriTTS and CommonVoice. When reading LibriTTS there are no useful annotations, those have been removed from the saved dataset, and just a list of files is generated, however CommonVoice provides a useful index of files, so we use that instead. ValidateSpeechDataset works similarly to LibriTTS, however is much shorter and has a wide variety of excerpts which aren't included in the training datasets to allow for effective validation. For testing purposes there is also a RandomAudioDataset which generates completely random audio, and SetAudioDataset, which always returns a certain segment of audio.

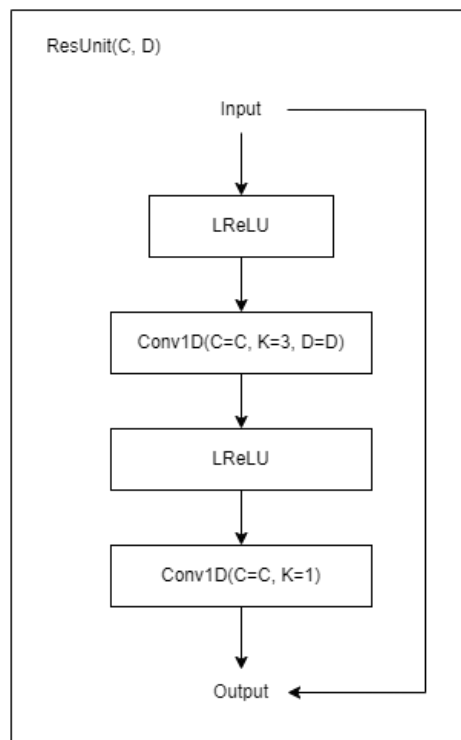
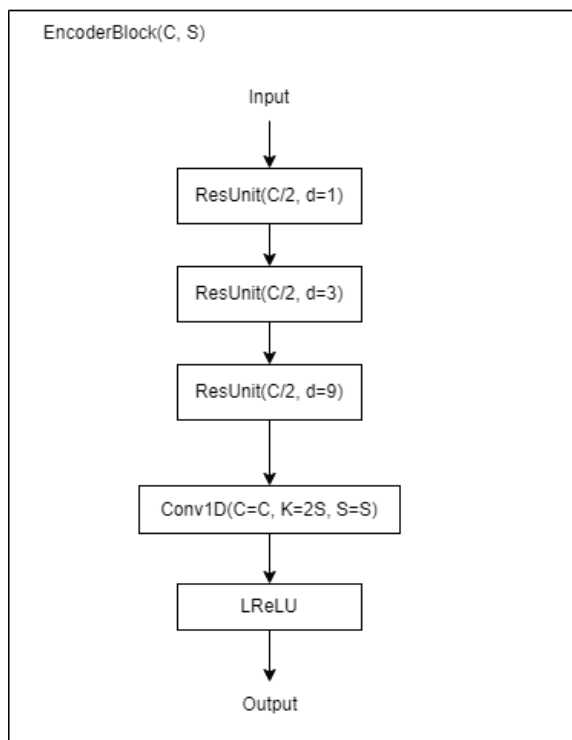
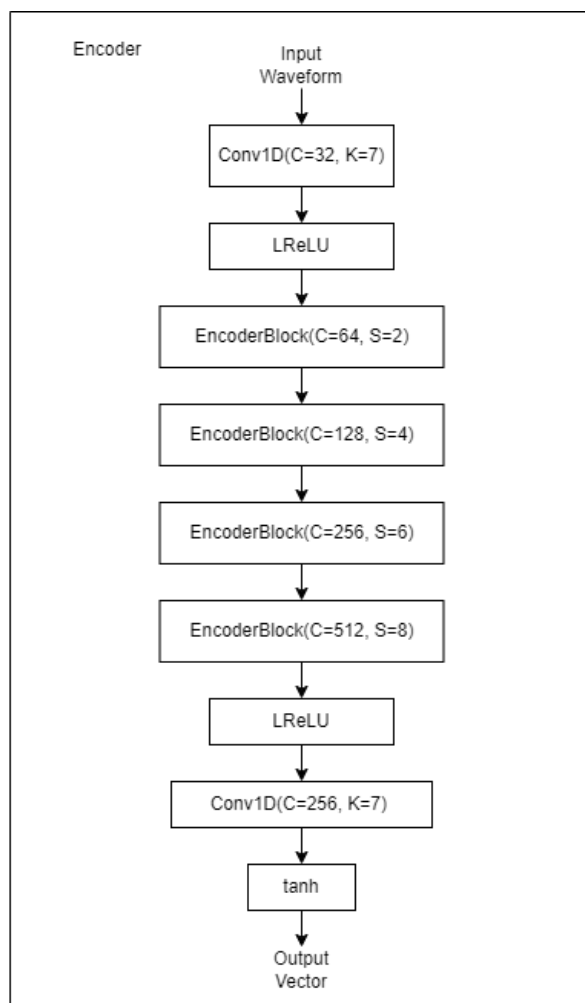
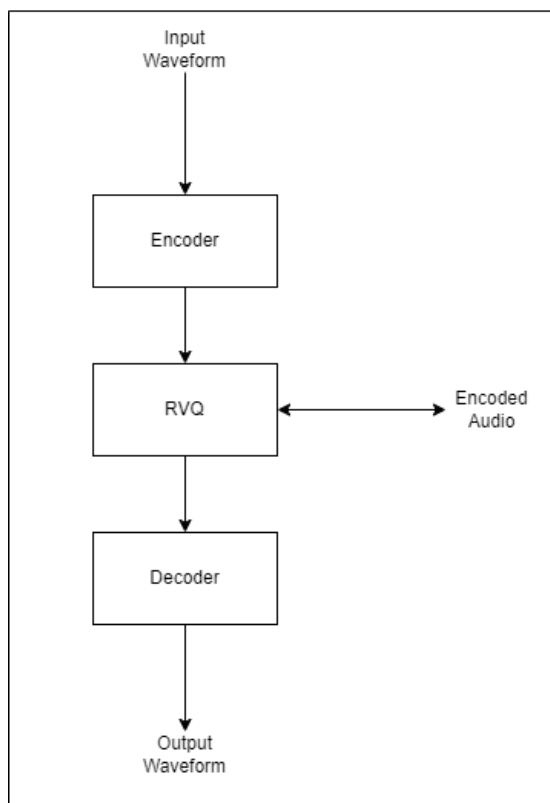
The datasets are organised into classes, with each dataset getting its own class, all of these classes extend the pytorch Dataset class as this provides multithreading and allows for the use of a PytorchDataloader when training. The class diagram is as follows:

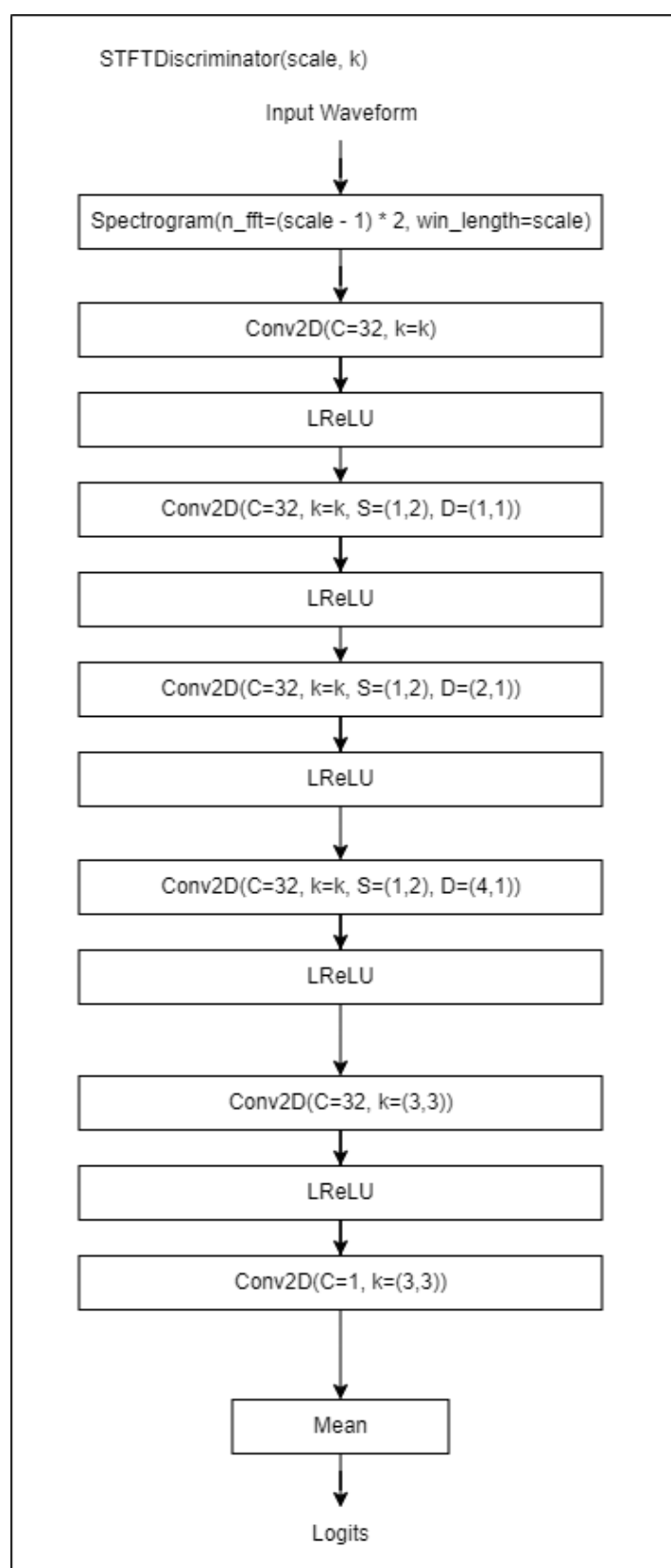


Network Structure

Below is the structure of each neural network. The arrows represent the flow of data from one layer to another, when multiple arrows combine into the same box, the average of the data held by the two arrows is taken.







Most of the layers represent pytorch layers of the same name, except where the layer defined otherwise in the diagrams, custom layers added are the Vector Quantization layers, Encoder/Decoder blocks, and ResUnits. These layers are either made up of other layers,

which is detailed above, or, in the case of Vector Quantization, are completely custom, with how they work detailed below.

Vector Quantization

Forward pass pseudocode

Inputs: `xs`, `codebook_vectors`

Output: Embedded Vectors

1. Get distance from each `x` to each `codebook_vector`
2. For each `x`, get the index of the minimum
3. Create a new values tensor, store in it the `codebook_vectors[x_index]` for each `x`
4. Calculate the loss by calculating the mean squared error loss between the inputs and the outputs.
5. Detach the change in `x` from the gradient graph.

This pseudocode has encoded vectors inputted, and outputs the embedded vectors, it does this through a minimum distance algorithm, each point is assigned to the point which is the least "different" to it. This difference is calculated using the Euclidean distance between the two vectors in n dimensional space. This serves the purpose of giving us the best representation of the input vector using only our codebook vectors, when encoding we'd save this, however as we want to do the forward pass we then immediately go into decoding (line 3 onwards), where we use this index in our codebook to generate an output hopefully as similar to the original encoded vectors as possible for decoding. As we don't want the codebook vectors being trained by any loss functions except the quantization loss we finally calculate this loss, and then make sure that the change of the vector is removed from the gradient graph.

Frozen K-Means Pseudocode

K-Means clustering is the common algorithm used for grouping vectors into similar areas. We use a similar algorithm, however "freeze" some of our codebook vectors to allow for only "non-dead" vectors to be changed.

Inputs: `cached`, `codebook_vectors`, frozen vectors - where `cached` is the past 5 steps of data

Outputs: updated `codebook_vectors`

1. Generate a list of unique vectors in cache.
2. For each unfrozen vector in codebooks assign a random starting location selected from the unique cache vectors.
3. Repeat for `k_means_iters` times:

- 3.1. get distances between each vector in cache and all codebook vectors
- 3.2. Assign each vector in cache to the nearest codebook vector.
- 3.3. Count up the number of usages of each codebook vector
- 3.4. If the number of usages == 0, assign the codebook vector a random choice from the cache.
- 3.5. If the number of usages != 0:
 - 3.5.1. Calculate the mean of all of the vectors assigned to each centroid
 - 3.5.2. Set the codebook_vector for each centroid to the centroid mean.

The k-means algorithm works by repeatedly generating a list of "closest" points to each centroid (encoding vector), and then moving the centroid to the mean of those points; it does this repeatedly for a number of iterations, in our case likely between 5-10. This makes it so that the centroid more accurately represents the points which have it as their closest centroid. In the case where the centroid has no points the centroid is assigned to a random point. In the algorithm above some of these vectors are also frozen, this means that they won't be moved, we do this to any vectors which have been used more than a certain number of times since this algorithm was last run, as the decoder has been trained to expect those specific values, and changing them causes a significant degradation in performance.

Residual Vector Quantization

Residual Vector Quantization is an algorithm used for vector encoding which allows for a significant increase in the quality of data stored, without a significant increase in memory requirements trying to store millions of unique vectors.

Inputs: xs, quantizers

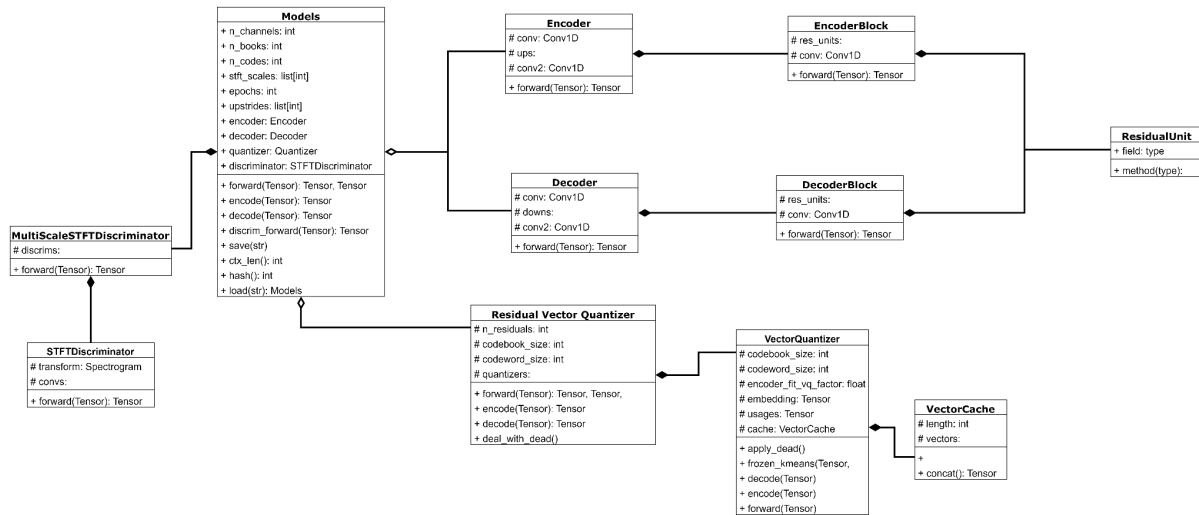
Outputs: ys

1. residual \leftarrow xs.copy()
2. loss_sum \leftarrow 0
3. indices \leftarrow 0
4. For quantizer in quantizers:
 - 1.1. encoded \leftarrow quantizer(residual)
 - 1.2. residual \leftarrow residual - quantizer
 - 1.3. loss_sum \leftarrow loss_sum + quantizer.loss
 - 1.4. Concatenate quantizer.indices to indices

This algorithm works by first carrying out vector quantization as above, then, to improve the accuracy of the encoded vectors, we take the residual (the difference between the original and encoded), and carry the same process out on that, we repeat this for multiple quantizers which the aim of best representing the original data into a list of codebook indexes. A similar way to achieve the same end effect is to have very large codebooks in vector quantization, however this massively increases the memory requirement. For example instead of having two layers with 1024 codebook vectors, you could have one with over one million.

Object Oriented Approach

This is all divided into classes, with each network layer having its own class, however some of these classes are supplied by pytorch, in the diagram below we only show the classes which have been written for this project, not those in the standard pytorch library.



On the very left hand side of this diagram is the discriminator, which is described in more detail in the loss section. The rest of this diagram shows the generator model with the residual vector quantization and vector quantization classes holding the algorithm detailed just above this.

Models
+ n_channels: int + n_books: int + n_codes: int + stft_scales: list[int] + epochs: int + upstrides: list[int] + encoder: Encoder + decoder: Decoder + quantizer: Quantizer + discriminator: STFTDiscriminator
+ forward(Tensor): Tensor, Tensor + encode(Tensor): Tensor + decode(Tensor): Tensor + discrim_forward(Tensor): Tensor + save(str) + ctx_len(): int + hash(): int + load(str): Models

Looking at the models class in more detail, this holds the configuration, and objects required for the model system. For example the quantizer, encoder and decoder.

For this reason this class has functions such as encode and decode which allow for easy interaction with the model. This is the interface through which all interaction with the model should happen.

We also need to be able to save all of the data in this class to allow us to read it back at a later time to either continue training, or use it for inference.

When doing this there's a few key elements that need to be saved, the model parameters (n_channels, n_books, n_codes, upstrides, and

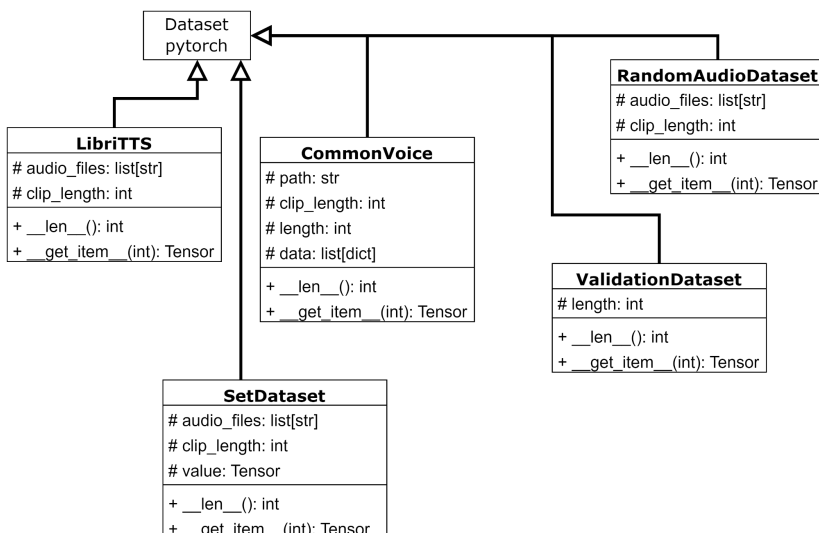
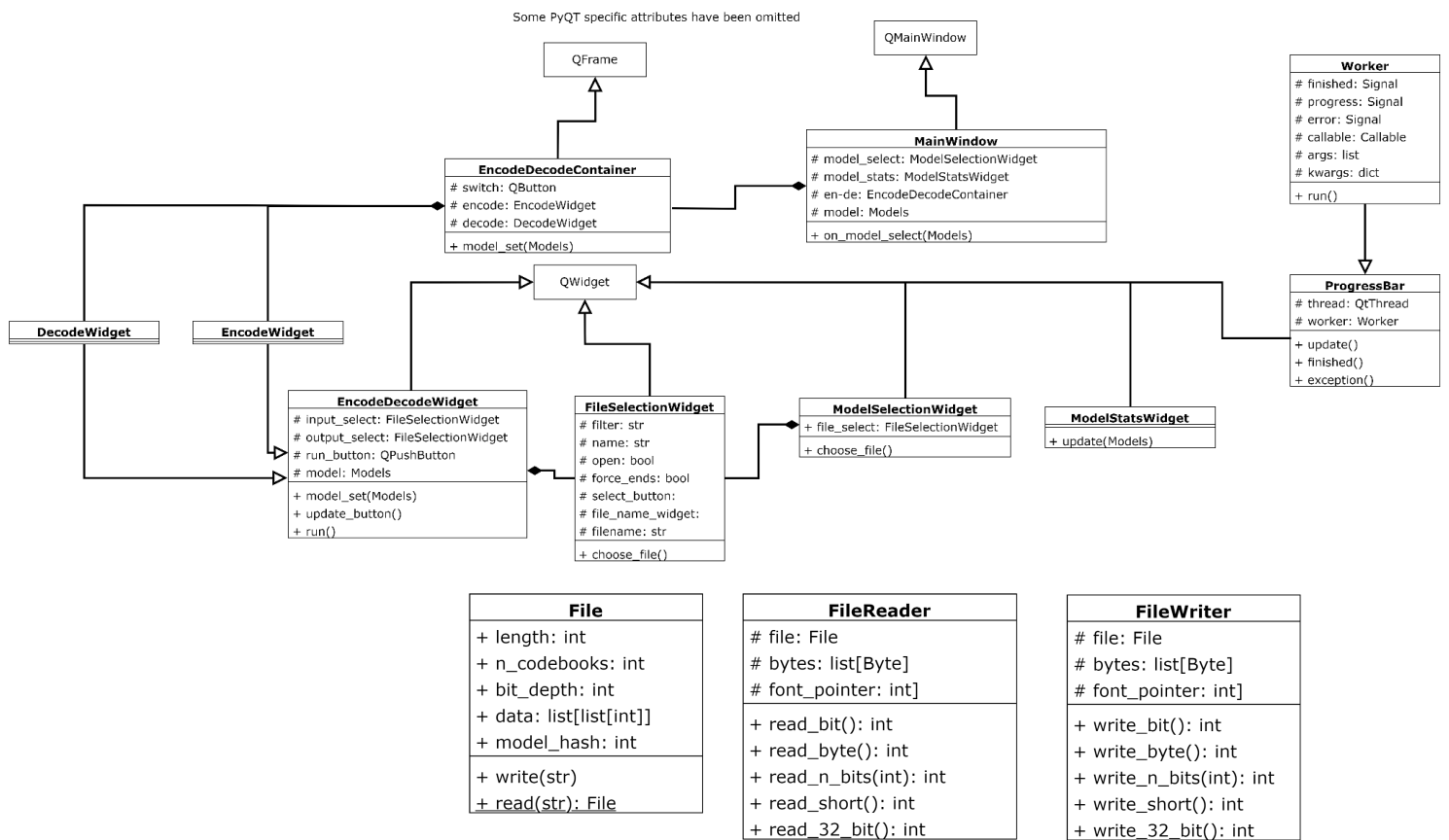
epochs), then for data integrity purposes, a hash of the model parameters, and finally the model parameters themselves, this is done using pytorch's save method, which allows for the easy writing and reading of dictionaries from files.

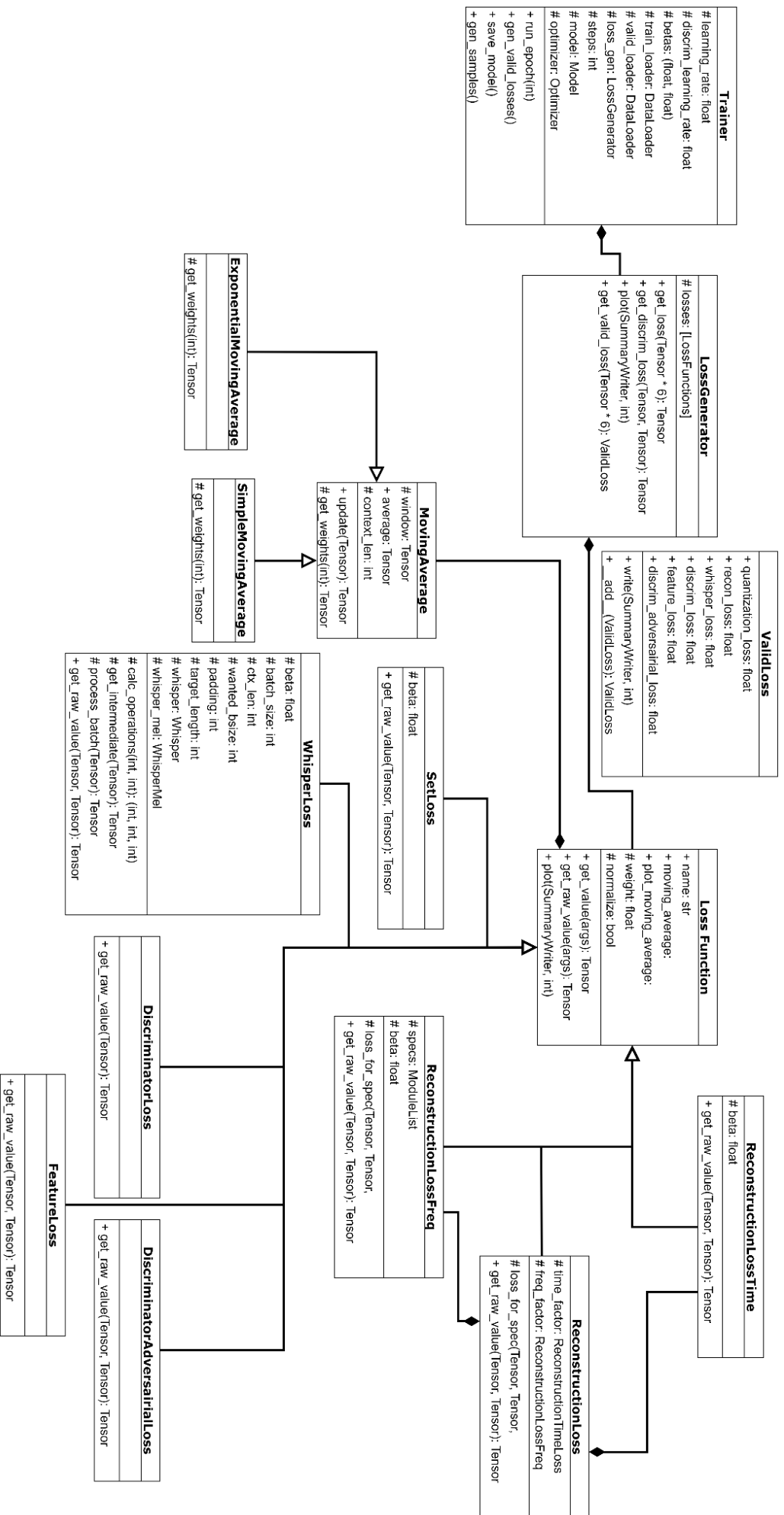
VectorCache
length: int # vectors: list[Tensor]
+ add_vector(Tensor) + concat(): Tensor

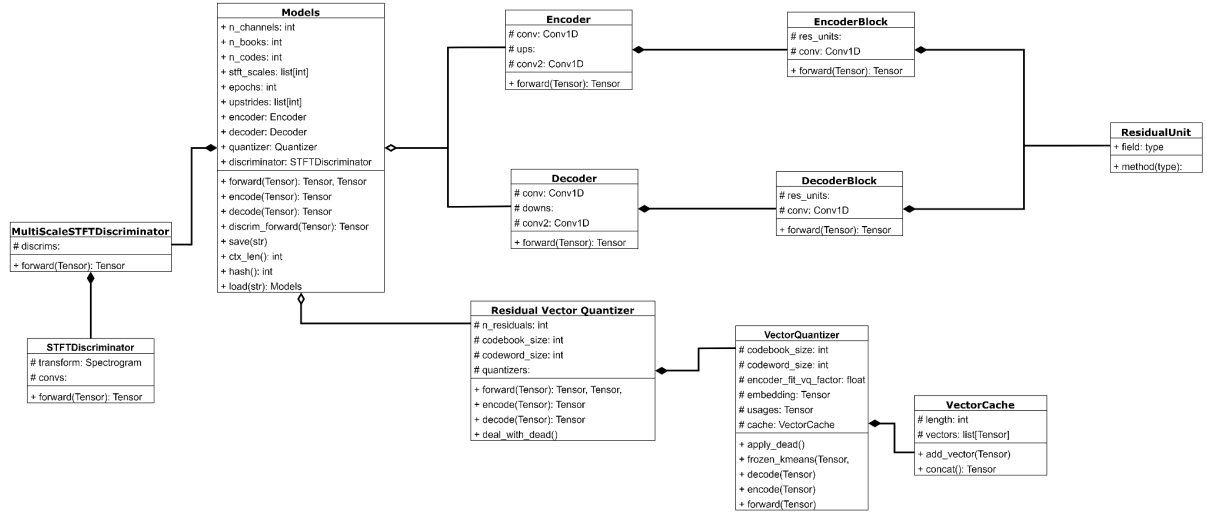
Vector cache acts as a very simple queue, keeping at least n vectors in a list. The only difference between this cache and a queue is the concat function, which concatenates all of the Tensors in the queue together. For example if the queue size is 5, and is made of NxM Tensors, the output will be of shape 5NxM.

Complete Class Diagram

Each class diagram has been explained individually in its corresponding section, however here's a complete diagram representing the whole system, it is split across multiple pages.







Technical Solution

Models

Complex scientific/mathematical/robotics/control/business model

A complex mathematical model is used in this solution, with most data stored as tensors, which then have mathematical operations applied to them.

Complex user-defined use of object-oriented programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces

A complex OOP model is used throughout this solution, however is most evident in how this solution deals with loss functions, each loss function is in its own class which all extend the *LossFunction* class.

Tensor Data Model

Throughout the model portion of this solution most data is modelled as Tensors, this is a common machine learning way of representing data as multidimensional matrices of data which can then be treated either as arrays or matrices.

Algorithms

Advanced Matrix Operations

Consistently throughout Residual Vector Quantization is advanced matrix calculations, these are used especially during the frozen kmeans algorithm for efficiency.

Queue Operations

The file reader and file writer objects function as a queue when reading or writing data to the filesystem.

User defined algorithms

The file reader and writer objects use user defined algorithms to be able to write/read any bit length integers to a stream of bytes.

Dynamic Generation of Objects

Objects are generated dynamically based on the model file which is when running the inference program.

Hashing

Hashing is used for validating that the model has been loaded correctly and has remained identical to when it was saved. It is also used for validating that the same model file used for encoding is used for decoding.

Coding Styles

Modules with appropriate interfaces

All modules have appropriate interfaces as set out in the design section of this document.

Loosely coupled modules

All modules are loosely coupled with interaction only through the provided interfaces, for example the inference module only interacts with the network module through the Models interface.

Cohesive Modules

Each module consists of only the code for a specific function throughout this entire NEA with each file having a set purpose and only doing things dedicated to that purpose.

Defensive Programming

User inputs are always checked for validity, model files are checked for corruption and assert statements are used in training code to ensure that bugs can't go unnoticed.

Solution

model/	36
main.py	36
train.py	37
models.py	39
vq.py	43
datasets.py	46
utils.py	48
model/loss/	49
loss.py	49
discriminator_model.py	52
moving_average.py	54
model/analysis/	55
dead_analysis.py	55
inference/	56
main.py	56
inference.py	62
file_structure.py	64
datasets/	68
process_dataset.sh	68

model/

The training system is in the model directory and is written in a way that allows for a developer to easily modify the network being created, however it is intended to be used by programmers, for this reason settings need to be changed in code (namely in the main.py file), and we prefer errors to crash the program so they can be identified and fixed as otherwise pytorch can give unintended results.

main.py

	<pre>import torch from model import datasets import models from torch.utils.data import DataLoader import vq import model.train as train from model.loss.loss import LossGenerator</pre>
This function is designed to be edited for each training run, it is	<pre>def main(): batch_size = 32 device = "cuda" if torch.cuda.is_available() else "cpu"</pre>

where the model parameters are set, and things are loaded, it then keeps looping over running epochs to train the model

```
#m = models.Models(192, 4, 1024, upstrides=[2,4,6,8], device=device)
m = models.Models.load("logs-t/epoch17/models.saved", device=device)
context_length = m.ctx_len*32

train_data = datasets.CommonVoice(context_length)
valid_loader = datasets.CommonVoice(context_length, "test", length = 100)

train_dataloader = DataLoader(train_data, batch_size=batch_size,
shuffle=True, num_workers=8)
valid_loader = DataLoader(valid_loader, batch_size=batch_size,
num_workers=8)

loss_gen = LossGenerator(context_length, batch_size, device=device)

trainer = train.Trainer(m, train_dataloader, valid_loader, loss_gen,
device=device, learning_rate=0.00005)
while True:
    trainer.run_epoch()
    trainer.save_model(f"logs-t/epoch{m.epochs}/model.saved")
    m.epochs += 1
    print(f"Epoch {m.epochs} starting")

if __name__ == "__main__":
    main()
```

train.py

```
import torch
import itertools
from models import *
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm
import torchaudio
import os
from loss import loss
```

```
class Trainer:
    def __init__(self, models: Models, train_loader: DataLoader, valid_loader: DataLoader,
loss_gen: loss.LossGenerator, device="cpu", learning_rate=0.0002, betas=[0.5, 0.9],
discrim_learning_rate=0.0002, gamma=0.99) -> None:
        self.models = models
        self.learning_rate = learning_rate
        self.discrim_learning_rate = discrim_learning_rate
        self.betas = betas
        self.train_loader = train_loader
        self.valid_loader = valid_loader
        self.loss_gen = loss_gen
        self.device = device

        self.model_optimizer = torch.optim.Adam(itertools.chain(models.encoder.parameters(),
models.decoder.parameters(), models.quantizer.parameters()), lr=learning_rate, betas=betas)
        self.discriminator_optimizer = torch.optim.Adam(models.discriminator.parameters(),
lr=discrim_learning_rate, betas=betas)
```

```

        self.scheduler_model = torch.optim.lr_scheduler.ExponentialLR(self.model_optimizer,
gamma=gamma)
        self.scheduler_discrim =
torch.optim.lr_scheduler.ExponentialLR(self.discriminator_optimizer, gamma=gamma)

        self.writer = SummaryWriter(log_dir="logs-t/")

        self.steps = 1 # start steps at 1 so that we don't run logging on first step

```

```

def run_epoch(self):
    self.models.train()
    for x in tqdm(self.train_loader):
        self.models.train() # can't be too careful
        x = x.to(self.device)
        y, q_loss = self.models(x)
        discrim_x, feature_x = self.models.discrim_forward(x)
        discrim_y, feature_y = self.models.discrim_forward(y)

        self.model_optimizer.zero_grad()
        loss = self.loss_gen.get_loss(x, y, discrim_y, feature_x, feature_y, q_loss)
        self.writer.add_scalar("test/main", loss.item(), self.steps)
        loss.backward(retain_graph=True)

        self.model_optimizer.step()

        self.discriminator_optimizer.zero_grad()
        discrim_x, _ = self.models.discrim_forward(x.detach())
        discrim_y, _ = self.models.discrim_forward(y.detach())
        discrim_loss = self.loss_gen.get_discrim_loss(discrim_x, discrim_y)

        self.writer.add_scalar("test/discrim", discrim_loss.item(), self.steps)
        discrim_loss.backward()
        if self.steps % 3 < 2:
            self.discriminator_optimizer.step()

        del x, discrim_x, discrim_y, discrim_loss, loss, feature_x, feature_y, y, q_loss # we
delete them here so that when we deal with validation we don't have them taking up vram (normally
pytorch deletes them when they go out of scope, however this isn't reliable).

        if self.steps % 100 == 0:
            with torch.no_grad():
                self.models.quantizer.deal_with_dead()

        if self.steps % 10 == 0:
            self.loss_gen.plot(self.writer, self.steps)
        if self.steps % 2000 == 0:
            self.gen_samples(f"epoch{self.models.epochs}")
        if self.steps % 5000 == 0:
            self.gen_valid_losses()
        self.steps += 1

    self.scheduler_model.step()
    self.scheduler_discrim.step()

```

```

@torch.no_grad()
def gen_valid_losses(self):
    self.models.eval()
    total_loss = loss.ValidLoss(0,0,0,0,0,0)
    for x in self.valid_loader:

```

```

        x = x.to(self.device)
        y, quant_loss = self.models(x)
        discrim_x, feat_x = self.models.discrim_forward(x.detach())
        discrim_y, feat_y = self.models.discrim_forward(y.detach())

        total_loss += self.loss_gen.get_valid_loss(x, y, discrim_x, discrim_y, feat_x, feat_y,
quant_loss)

        total_loss.write(self.writer, self.steps)
        self.models.train()

    def save_model(self, folder_name):
        self.models.save(folder_name)

    def gen_samples(self, folder_name):
        self.models.eval()
        os.makedirs(f"samples/{folder_name}", exist_ok = True)
        for i, case in enumerate(self.valid_loader):
            if i > 5:
                break
            output, _ = self.models(case.to("cuda"))
            torchaudio.save(f"samples/{folder_name}/{i}-clean.wav", case[0], sample_rate=16000)
            torchaudio.save(f"samples/{folder_name}/{i}-encoded.wav", output.cpu()[0],
sample_rate=16000)
            self.models.train()

```

models.py

This file implements the model, both defining the Models class, a wrapper for the rest of the model, and each of the layers of the model excluding vector quantization.

```

from torch import nn
import torch.nn.functional as F
from torch.nn.utils import weight_norm
from model.loss.discriminator_model import MultiScaleSTFTDiscriminator
from model import vq
import math
import os
import torch
import gzip
import hashlib
import json
import itertools, functools
import io
LEAKY_RELU = 0.2
INIT_MEAN = 0.0
INIT_STD = 0.01

class Models(nn.Module):
    def __init__(self, n_channels, nbooks, ncodes, epochs=0, sftf_scales=[1024, 512, 256],
upstrides=[2,4,6,8], device="cpu", discrim=True) -> None: # improve type hints here, probably
make a separate quantizer class
        super().__init__()
        self.n_channels = n_channels
        self.nbooks = nbooks
        self.ncodes = ncodes

```

```

self.stft_scales = stft_scales
self.epochs = epochs
self.upstrides = upstrides

self.encoder = Encoder(n_channels, upstrides=upstrides).to(device)
self.quantizer = vq.RVQ(nbooks, ncodes, n_channels).to(device)
self.decoder = Decoder(n_channels, upstrides=upstrides).to(device)
if discrim:
    self.discriminator = MultiScaleSTFTDiscriminator(scales=stft_scales).to(device)
else:
    self.discriminator = None

def forward(self, x):
    """ return in the form y, q_loss """
    encoded = self.encoder(x)

    encoded = torch.transpose(encoded, -1, -2)
    after_q, _, q_loss = self.quantizer(encoded)
    after_q = torch.transpose(after_q, -1, -2)
    y = self.decoder(after_q)
    return y, q_loss

def encode(self, x):
    encoded = self.encoder(x)

    encoded = torch.transpose(encoded, -1, -2)
    output = self.quantizer.encode(encoded)
    return output

def decode(self, x):
    after_q = self.quantizer.decode(x)
    after_q = torch.transpose(after_q, -1, -2)
    y = self.decoder(after_q)
    return y

def discrim_forward(self, x):
    return self.discriminator(x)

def save(self, file_name):
    os.makedirs(f"{''.join(file_name.split('/')[:-1])}", exist_ok = True)
    buffer = io.BytesIO()
    torch.save(self.state_dict(), buffer)
    buffer.seek(0)
    model_files = buffer.read()
    model_hash = hashlib.md5(model_files).digest()

    params = {
        "n_channels": self.n_channels,
        "ncodes": self.ncodes,
        "nbooks": self.nbooks,
        "upstrides": self.upstrides,
        "epochs": self.epochs
    }

    result = {
        "model_hash": model_hash,
        "params": params,
        "models": model_files
    }

    torch.save(result, f"{file_name}")

```

```

@property
def ctx_len(self):
    return functools.reduce(lambda x, y : x*y, self.upstrides)

@property
def hash(self):
    """ Really slow, shouldn't be used frequently """
    buffer = io.BytesIO()
    torch.save(self.state_dict(), buffer)
    buffer.seek(0)
    model_files = buffer.read()
    model_hash = hashlib.md5(model_files).digest()
    hash_as_int = int.from_bytes(model_hash, byteorder="big", signed=False)
    return hash_as_int

def load(folder_name, device="cpu"):
    try:
        m = torch.load(folder_name)
    except IOError:
        raise Exception("File doesn't exist")

    if hashlib.md5(m["models"]).digest() != m["model_hash"]:
        raise Exception("Invalid hash")

    model_statedict = torch.load(io.BytesIO(m["models"]))
    params = m["params"]
    models = Models(params["n_channels"], params["nbooks"], params["ncodes"],
epochs=params["epochs"], device=device)
    models.load_state_dict(model_statedict)
    return models

```

```

def get_padding(kernel_size, dilation=1):
    return ((kernel_size-1) * dilation) // 2

# paper https://arxiv.org/pdf/2009.02095.pdf
class ResidualUnit(nn.Module):
    """ Residual Unit, input and output are same dimension """
    def __init__(self, nChannels, dilation) -> None:
        super().__init__()
        self.conv1 = weight_norm(nn.Conv1d(nChannels, nChannels, kernel_size=3,
padding=get_padding(3, dilation), dilation=dilation))
        self.conv2 = weight_norm(nn.Conv1d(nChannels, nChannels, kernel_size=1))
        nn.init.normal_(self.conv1.weight, INIT_MEAN, INIT_STD)
        nn.init.normal_(self.conv2.weight, INIT_MEAN, INIT_STD)

    def forward(self, x):
        xt = F.leaky_relu(x, LEAKY_RELU)
        xt = self.conv1(xt)
        xt = F.leaky_relu(xt, LEAKY_RELU)
        xt = self.conv2(xt)
        return xt + x

```

```

class EncoderBlock(nn.Module):
    """ Encoder block, requires input size to be nOutChannels / 2 """
    def __init__(self, nOutChannels, stride) -> None:
        super().__init__()
        self.res_units = nn.ModuleList([
            ResidualUnit(int(nOutChannels/2), dilation=1), # in N out N
            ResidualUnit(int(nOutChannels/2), dilation=3), # in N out N
            ResidualUnit(int(nOutChannels/2), dilation=9) # in N out N
        ])

```



```

        self.conv = weight_norm(nn.Conv1d(int(nOutChannels/2), nOutChannels,
kernel_size=2*stride, stride=stride, padding=get_padding(2*stride))) # in N out 2N

```

```

def forward(self, x):
    for unit in self.res_units:
        x = unit(x)

    x = self.conv(x)
    x = F.leaky_relu(x, LEAKY_RELU)
    return x

```

```

class DecoderBlock(nn.Module):
    """ Decoder block, requires input size to be nOutChannels / 2 """
    def __init__(self, nOutChannels, stride) -> None:
        super().__init__()
        self.res_units = nn.ModuleList([
            ResidualUnit(int(nOutChannels), dilation=1), # in N out N
            ResidualUnit(int(nOutChannels), dilation=3), # in N out N
            ResidualUnit(int(nOutChannels), dilation=9) # in N out N
        ])
        self.conv = weight_norm(nn.ConvTranspose1d(nOutChannels*2, nOutChannels,
kernel_size=2*stride, stride=stride, padding=math.ceil((stride) / 2))) # in 2N out N

    def forward(self, x):
        x = self.conv(x)
        x = F.leaky_relu(x, LEAKY_RELU)

        for unit in self.res_units:
            x = unit(x)

        return x

```

```

class Encoder(nn.Module):
    def __init__(self, endChannels, upstrides=[2,4,6,8], base_width=32) -> None:
        super().__init__()
        self.conv = weight_norm(nn.Conv1d(1, base_width, kernel_size=7, padding=get_padding(7)))
        self.ups = nn.ModuleList()

        multiplier = 1
        for i in range(len(upstrides)):
            multiplier *= 2
            self.ups.append(EncoderBlock(base_width * multiplier, stride=upstrides[i]))

        self.conv2 = weight_norm(nn.Conv1d(base_width*multiplier, endChannels, kernel_size=7,
padding=get_padding(7)))
        nn.init.normal_(self.conv.weight, INIT_MEAN, INIT_STD)
        nn.init.normal_(self.conv2.weight, INIT_MEAN, INIT_STD)

    def forward(self, x):
        x = self.conv(x)
        x = F.leaky_relu(x, LEAKY_RELU)

        for unit in self.ups:
            x = unit(x)

        x = F.leaky_relu(x, LEAKY_RELU)
        x = self.conv2(x)
        y = F.tanh(x)
        return x

```

```

class Decoder(nn.Module):
    def __init__(self, endChannels, upstrides=[2,4,6,8], base_width=256) -> None:

```

```

        super().__init__()

        self.ups = nn.ModuleList()
        downstrides = upstrides[::-1]
        self.conv = weight_norm(nn.Conv1d(endChannels, base_width, kernel_size=7,
padding=get_padding(7)))

        for i in range(len(downstrides)):
            self.ups.append(DecoderBlock(base_width//(2**(i+1)), stride=downstrides[i]))

        self.conv2 = weight_norm(nn.Conv1d(base_width//(2**(len(downstrides))), 1,
kernel_size=7, padding=get_padding(7)))
        nn.init.normal_(self.conv.weight, INIT_MEAN, INIT_STD)
        nn.init.normal_(self.conv2.weight, INIT_MEAN, INIT_STD)

    def forward(self, x):
        x = self.conv(x)
        x = F.leaky_relu(x, LEAKY_RELU)

        for unit in self.ups:
            x = unit(x)

        x = F.leaky_relu(x, LEAKY_RELU)
        x = self.conv2(x)
        x = F.tanh(x)
        return x

```

vq.py

This file implements residual vector quantization, providing the classes VectorCache, VQ and RVQ, due to this file involving difficult to understand code explanations of code are included next the program. To understand lots of this file a basic understanding of embeddings in neural networks is required, embeddings serve as mappings between discrete integers and vectors. When conventionally used these are one way, always from discrete to vectors, however in our case instead we can also go the other way using the most "similar" discrete translation.

To do this embeddings use something called one_hot encoding, where the input is a list of discrete things to encode, and the output is a tensor where each row has a 1 in the place of the discrete thing, and zeros in all other entries, this can then be matrix multiplied with the codebook, a list of vectors where the first vector represents the first thing, and second vector the second thing etc, giving a list of things.

Here we have imports and the implementation of a very basic queue, the efficiency of this queue doesn't matter as it's

```

from torch import nn
import torch.nn.functional as F
import random
import torch

```

<p>not storing masses of data and is rarely used.</p>	<pre>class VectorCache(): def __init__(self, length) -> None: self._length = length self._vectors = [] # if I need to I could implement a proper circular queue here def add_vector(self, new_vector: torch.Tensor): self._vectors.append(new_vector) if len(self._vectors) > self._length: self._vectors.pop(0) def concat(self) -> torch.Tensor: return torch.cat(self._vectors, dim=0)</pre>
<p>This is the initialisation of the VQ class, this class provides vector quantization.</p>	<pre>class VQ(nn.Module): def __init__(self, codebook_size, codeword_size, n=1, beta=0.2) -> None: super().__init__() self.codebook_size = codebook_size self.codeword_size = codeword_size self.encoder_fit_vq_factor = beta self.embedding = nn.Parameter(torch.zeros((codebook_size, codeword_size))) self.embedding.data.uniform_(-1.0 / (n**2), 1.0 / (n**2)) usages = torch.zeros((codebook_size), requires_grad=False) self.register_buffer('usages', usages) self.cache = VectorCache(20)</pre>
<p>This apply dead function goes through the list of usages and decides on which embedding vectors aren't used and runs frozen kmeans on them. It then clears out the usages by setting them all to zero.</p>	<pre>def apply_dead(self): indices_of_dead = self.usages == 0 # WARNING this doesn't actually give a list of indices, but a list of true / false values indices_of_alive = self.usages > 0 self.frozen_kmeans(self.cache.concat(), indices_of_alive) self.usages[:] = torch.zeros((self.codebook_size), requires_grad=False)</pre>
<p>This implements the frozen_kmeans algorithm from the documented design section.</p> <p>Most of this section is commented however how we calculate the mean for each codebook isn't entirely clear. Instead of using an iterative solution as suggested in the design, which wouldn't be able to be efficiently calculated on a gpu, we use matrix multiplication, first we use pytorch's one_hot and multiplied with the</p>	<pre>@torch.no_grad() def frozen_kmeans(self, cached, frozen_state, kmeans_iters=5): frozen_length = torch.sum(frozen_state).to(torch.int32) # shorthand for count boolean unfrozen_length = frozen_state.shape[0] - frozen_length unique_vecs = cached.unique(dim=0) random_idx = torch.randperm(unique_vecs.size(dim=0))[:unfrozen_length] self.embedding[torch.logical_not(frozen_state)] = cached[random_idx] ## Initialisaation complete move on to algorithm for _ in range(kmeans_iters): distances = torch.sum(cached**2, dim=1, keepdim=True) + torch.sum(self.embedding, dim=1) - 2.0 * torch.matmul(cached, self.embedding.t()) # calculate the distances from each centroid to the value dists, indexes = torch.min(distances, dim=1) counts = torch.bincount(indexes,</pre>

<p>cached entries, this creates a tensor where each column represents the cache entries for each centroid. These columns can then be summed, and divided by the number of cache entries to get the average position of each point.</p>	<pre> minlength=fronzen_state.shape[0]) # count the number in each bin counts[fronzen_state] = -1 # make sure to mark as frozen ### First deal with those with no nearby points ### # These have to be dealt with separately as otherwise would get NaN # get the indices of the points furthest away from their centroid indices = torch.sort(dists).indices[:torch.sum(counts==0)] # torch.sum(Tensor[bool]) is short hand for count number of trues self.embedding[counts == 0] = cached[indices] ### Then deal with the means ### oh = F.one_hot(indexes, num_classes=fronzen_state.shape[0]).to(torch.float32).t() amount_of_each = torch.sum(oh, dim=1) total_of_each = torch.matmul(oh, cached) result_means = total_of_each / torch.unsqueeze(amount_of_each, 1) # be careful with result means, where counts = 0 result means > 0 self.embedding[counts > 0] = result_means[counts > 0] </pre>
<p>Here we act as a regular embedding (as detailed above) for decoding, and use the distance to the embedding vector for going the other way,</p>	<pre> def decode(self, indexes): one_hot = F.one_hot(indexes, num_classes=self.codebook_size).float() values = torch.matmul(one_hot, self.embedding) return values def forward(self, x): self.cache.add_vector(x.reshape(-1, self.codeword_size).detach()) dist = torch.sum(x**2, dim=-1, keepdim=True) + torch.sum(self.embedding**2, dim=1) - 2.0 * torch.matmul(x, self.embedding.t()) # x^2 + y^2 - 2xy vals, indexes = torch.min(dist, dim=-1) # this isn't differentiable, so need to add in loss later (passthrough loss) # Two lines below are just normal style embedding one_hot = F.one_hot(indexes, num_classes=self.codebook_size).float() values = torch.matmul(one_hot, self.embedding) loss1 = F.mse_loss(x.detach(), values) # only train the embedding loss2 = F.mse_loss(x, values.detach()) # only train the encoder values = x + (values - x).detach() if self.training: self.usages = self.usages + torch.sum(one_hot.reshape(-1, self.codebook_size), dim=-2) return values, indexes, loss1 + loss2 * self.encoder_fit_vq_factor </pre>
<p>Residual Vector Quantization as detailed in Documented Design</p>	<pre> class RVQ(torch.nn.Module): def __init__(self, n_residuals, codebook_size, codeword_size, bypass_factor=0.5) -> None: super().__init__() </pre>

```

self.n_residals = n_residuals
self.codebook_size = codebook_size
self.codeword_size = codeword_size
self.quantizers = nn.ModuleList([VQ(self.codebook_size,
self.codeword_size) for i in range(n_residuals)])

self.bypass_factor = bypass_factor

def forward(self, x):
    y_hat = torch.zeros_like(x)
    residual = x
    loss = 0
    indices = None
    for i, q in enumerate(self.quantizers):
        q_values, q_i, l = q(residual) # in this we don't actually
care about the indices
        residual = residual - q_values # inplace operators mess
with pytorch, so we can't use them
        y_hat = y_hat + q_values # see above
        loss = loss+l # see above
        q_i = q_i.unsqueeze(-1)
        indices = q_i if indices == None else torch.cat((indices,
q_i), dim=-1)

    loss = loss / len(self.quantizers)

    y_hat = x + (y_hat - x).detach() # maintains gradients in x, but
removes the ones we don't want in y_hat

    return y_hat, indices, loss

def encode(self, data):
    y_hat, indices, loss = self.forward(data)
    return indices

def decode(self, indices):
    result = 0.0
    for i, q in enumerate(self.quantizers):
        result += q.decode(indices[... , i])
    return result

def initialise(self, x):
    residual = x
    for q in self.quantizers:
        q.initialise(residual)
        q_values, _, _ = q(residual)
        residual = residual - q_values

def deal_with_dead(self):
    for q in self.quantizers:
        q.apply_dead()

```

datasets.py

```

import torch
import torchaudio

```

```

from torch.utils.data import Dataset
import glob
import random
import matplotlib.pyplot as plt
import math
from model import utils
from torchaudio.functional import resample

```

```

def make_length(audio, length):
    if audio.shape[1] > length:
        start = random.randint(0, audio.shape[1] - length - 1)
        return audio[:, start:start+length]
    else:
        padding = torch.zeros(1, length)
        padding[:, :audio.shape[1]] = audio
        return padding

```

```

class LibriTTS(Dataset):
    def __init__(self, clip_length, length=None):
        self._audio_files = glob.glob("datasets/speech_train/*.wav")
        random.Random(12321).shuffle(self._audio_files) # make them appear in a random order,
        set seed for reproducibility
        if length != None:
            self._audio_files = self._audio_files[:length]
            # We have to do the above otherwise it is likely we train on one speaker for a bit, and
            then move on to another, etc, possibly not generalizing the model then
            self._clip_length = clip_length

    def __len__(self):
        return len(self._audio_files)

    def __getitem__(self, index):
        sound, sample_rate = torchaudio.load(self._audio_files[index % len(self._audio_files)])
        assert sound.shape[0] == 1, "Only mono audio allowed, no stereo"
        assert sample_rate == 16000, "Sample rate of file isn't 16 kHz"
        sound = utils.norm(sound) # Want to make sure that we normalize over the whole clip, if
        we only normalize over our sample, silences may just become noise
        return make_length(sound, self._clip_length)

```

```

class CommonVoice(Dataset):
    def __init__(self, clip_length, mode: str="train", path:
str="/mnt/d/datasets/CommonVoice/cv-corpus-14.0-2023-06-23/en", length: int=-1) -> None:
        super().__init__()
        self._path = path
        self._clip_length = clip_length
        self._length = length

        with open(f"{path}/{mode}.tsv") as f:
            lines = f.readlines()

            titles = lines[0].split("\t")
            self._data = [{titles[j]: d for j, d in enumerate(line.split("\t"))} for line in
lines[1:]]
            if self._length != -1 and self._length < len(self._data):
                self._data = random.choices(self._data, k=self._length)

    def __len__(self):
        return len(self._data)

    def __getitem__(self, index) -> torch.Tensor:
        sound, sample_rate = torchaudio.load(f"{self._path}/clips/{self._data[index]['path']}")

```

```

if sample_rate != 16000:
    sound = resample(sound, sample_rate, 16000)
sound = utils.norm(sound)

return make_length(sound, self._clip_length)

```

```

class RandomAudioDataset(Dataset):
    def __init__(self, clip_length, length):
        self._clip_length = clip_length
        self._length = length

    def __len__(self):
        return self._length

    def __getitem__(self, index):
        return torch.randn((1, self._clip_length))

```

```

class SetAudioDataset(Dataset):
    def __init__(self, clip_length, length, value):
        self._clip_length = clip_length
        self._length = length
        self._value = value.unsqueeze(0)

    def __len__(self):
        return self._length

    def __getitem__(self, index):
        return self._value

```

```

class ValidateSpeechDataset(Dataset):
    def __init__(self):
        """ all will be multiples of clip_length """
        self._audio_files = glob.glob("datasets/speech_valid/*.wav")
        random.Random(12321).shuffle(self._audio_files) # make them appear in a random order,
set seed for reproducibility
        # We have to do the above otherwise it is likely we train on one speaker for a bit, and
then move on to another, etc, possibly not generalizing the model then
        self._audio_files = self._audio_files[:5]

    def __len__(self):
        return len(self._audio_files)

    def __getitem__(self, index):
        sound, sample_rate = torchaudio.load(self._audio_files[index % len(self._audio_files)])
        sound = utils.norm(sound)
        assert sound.shape[0] == 1 # this should always be the case, but if files have been
messed with crash
        assert sample_rate == 16000

        padding = torch.zeros(1, 16000 * 30)
        padding[:, :sound.shape[1]] = sound
        return padding

```

utils.py

This file contains random utilities used throughout the development of this project, whilst neither of these functions are used elsewhere in the final solution they are incredibly useful for examining what the model is doing, and figuring out how to improve it.

```

import torch
import matplotlib.pyplot as plt

def plot_waveform(waveform, file="matplotlib.png"):
    waveform = waveform.to("cpu").numpy()[0]
    print(waveform.shape)
    ax = plt.subplot()
    ax.set_ylim(bottom=-1, top=1)
    ax.plot(torch.arange(0, waveform.shape[0]), waveform)
    plt.savefig(file) # developing using wsl, but want to work on normal linux and windows, so
not messing with backends to make show() work

def plot_spectrograms(spectrogram1, spectrogram2, file="matplotlib.png"):
    plt.figure()

    f, axarr = plt.subplots(1, 2)
    axarr[0].imshow(spectrogram1.log2()[0, :, :500].to("cpu").numpy(), aspect='auto')
    axarr[1].imshow(spectrogram2.log2()[0, :, :500].to("cpu").numpy(), aspect='auto')
    f.savefig(file) # developing using wsl, but want to work on normal linux, so not messing
with backends to make show() work
    plt.close()

def norm(tensor):
    l, _ = torch.max(torch.abs(tensor), dim=-1)
    return (tensor / l) * 0.80

```

model/loss/

loss.py

```

import whisper
import torch
import torchaudio
from model.datasets import *
from model.datasets import torch
from model.loss import moving_average
import torch.nn.functional as F
from torch.utils.tensorboard import SummaryWriter

class Loss(torch.nn.Module):
    def __init__(self, name: str, weight: float, normalize: bool=True) -> None:
        super().__init__()
        self.name = name
        self.moving_average = moving_average.EMA(1000, beta=0.999)
        self.plot_average = moving_average.SMA(25) # should always be the same as the plot
interval, need to figure out a way to make this so
        self._weight = weight
        self._normalize = normalize

    def get_value(self, *args) -> torch.Tensor:
        raw = self.get_raw_value(*args)
        self.plot_average.update(raw)

```



```

        #return raw
        if self._normalize:
            return self._weight * (raw / (self.moving_average.update(raw.item()))) # Reduce it
            slightly cause it gives better reliability
        else:
            return self._weight * raw

    def forward(self, *args) -> torch.Tensor:
        return self.get_value(*args)

    def get_raw_value(self, *args) -> torch.Tensor:
        raise NotImplemented

    def plot(self, writer: SummaryWriter, steps: int) -> None:
        writer.add_scalar(f"train_loss/{self.name}", self.plot_average.average, steps)

```

```

class ReconstructionLossFreq(Loss):
    def __init__(self, weight, beta=1) -> None:
        super().__init__("frequency loss", weight)
        self._specs = torch.nn.ModuleList([torchaudio.transforms.MelSpectrogram(16000, n_mels=80,
n_fft=2 ** (i+1), win_length=2**i, hop_length=2 ** (i-2), f_max=8000, f_min=0) for i in range(9,
10)]) # see if the values here are reasonable, could well be way off
        self._beta = beta

    def loss_for_spec(self, x, y, spec) -> torch.Tensor:
        xs, ys = spec(x), spec(y)
        return F.l1_loss(xs, ys) + self._beta * F.mse_loss(xs, ys)

    def get_raw_value(self, x, y):
        total = 0.0
        for spec in self._specs: # can't use list comprehension here as pytorch then plays
tricks
            total = total + self.loss_for_spec(x, y, spec)
        return total / len(self._specs)

```

```

class ReconstructionLossTime(Loss): # This makes silence actually silent
    def __init__(self, weight, **args) -> None:
        super().__init__("time loss", weight, **args)

    def get_raw_value(self, x, y):
        return F.l1_loss(x, y)

```

```

class ReconstructionLoss(Loss):
    def __init__(self, time_weight, freq_weight, beta=1) -> None:
        super().__init__("Reconstruction Loss", 1, normalize=False)
        self.time_factor = ReconstructionLossTime(time_weight)
        self.freq_factor = ReconstructionLossFreq(freq_weight)

    def get_raw_value(self, x, y):
        #return self.loss_for_spec(x, y, self.spec) / (750 * 4 * 4)
        return self.time_factor(x, y) + self.freq_factor(x, y)

    def plot(self, writer, steps):
        self.time_factor.plot(writer, steps)
        self.freq_factor.plot(writer, steps)

```

```

class SetLoss(Loss): # used for testing
    def get_raw_value(self, raw_value):
        return raw_value

```

```

class WhisperMel(torch.nn.Module):
    def __init__(self) -> None:

```

```

    super().__init__()
    self._spectrogram = torchaudio.transforms.Spectrogram(n_fft=400, hop_length=160)
    self.register_buffer("_mel_filters", torchaudio.functional.melscale_fbanks(n_freqs=201,
f_max=8000, f_min=0, n_mels=80, sample_rate=16000, norm="slaney", mel_scale="slaney"))

    def forward(self, x):
        # steps here taken from whisper.log_mel_spectrogram in order to maintain compatability
(not have to retrain whisper), but this is much faster
        spec = self._spectrogram(x)[..., :-1]
        mel = (spec.transpose(-1, -2) @ self._mel_filters).transpose(-1, -2)

        log = torch.clamp_min(mel, 1e-10).log10() # clamp to prevent divide by 0
        log = torch.maximum(log, log.max() - 8.0)
        log = (log + 4.0) / 4.0
        return log

class WhisperLoss(Loss):
    WHISPER_EXPECTED = 480000
    WHISPER_EXPECTED_FACTORS = [7500, 8000, 9600, 10000, 12000, 15000, 16000, 19200, 20000,
24000, 30000, 32000, 40000, 48000, 60000, 80000, 96000, 120000, 160000, 240000, 480000] #
precomputed for performance, low omitted as they will never be used
    MIN_PADDING = 2000

    def __init__(self, context_length, batch_size, weight, beta=1.) -> None:
        super().__init__("Whisper Loss", weight)
        self._beta = beta
        self._batch_size = batch_size
        self._ctx_len = context_length

        self._wanted_bsize, self._padding, self._target_length =
self.calc_operations(context_length, batch_size)

        self.whisper = whisper.load_model("tiny.en")
        self.melspec = WhisperMel()

    def calc_operations(self, ctx_len, batch_size):
        candidates = [i for i in WhisperLoss.WHISPER_EXPECTED_FACTORS if i > ctx_len +
WhisperLoss.MIN_PADDING]
        aimed_length = candidates[0] # just take the first, will be the most efficient
        padding = aimed_length - ctx_len

        # stage 2, find the wanted batch size (we will add padding until the required batch size
= wanted)
        amount_per_batch = int(WhisperLoss.WHISPER_EXPECTED / aimed_length)
        whisper_batch_size = math.ceil(batch_size / amount_per_batch)
        wanted_batch_size = whisper_batch_size * amount_per_batch
        return (wanted_batch_size, padding, aimed_length)

    def get_intermediate(self, x):
        return self.whisper.encoder(x)

    def process_batch(self, x):
        x = torch.nn.functional.pad(x, (0, self._padding))

        if x.shape[0] != self._wanted_bsize: # make up the dimensions to
            t = torch.zeros(self._wanted_bsize, 1, self._target_length).to(x.device)
            t[:x.shape[0], ...] = x
            x = t

        return x.view(-1, WhisperLoss.WHISPER_EXPECTED)

    def get_raw_value(self, x, y) -> torch.Tensor:

```

```

if self.training:
    xp, yp = self.process_batch(x), self.process_batch(y)
else:
    xp, yp = torch.squeeze(x, 1), torch.squeeze(y, 1)

xspec, yspec = self.melspec(xp), self.melspec(yp)
xw, yw = self.get_intermediate(xspec), self.get_intermediate(yspec)

return F.l1_loss(xw, yw) + self._beta * F.mse_loss(xw, yw)

```

```

class DiscriminatorLoss(Loss):
    def __init__(self, weight) -> None:
        super().__init__("Discriminator Loss", weight)
        self.register_buffer("_empty", torch.tensor([0.]))

    def get_raw_value(self, discrim_y) -> torch.Tensor:
        values = torch.maximum(1-discrim_y, self._empty)
        return torch.mean(values)

```

```

class DiscriminatorAdversarialLoss(Loss):
    def __init__(self, weight) -> None:
        super().__init__("Discriminator Adversarial Loss", weight, normalize=False)
        self.register_buffer("_empty", torch.tensor([0.]))

    def get_raw_value(self, discrim_x, discrim_y) -> torch.Tensor: # get value as we don't want
to apply weight balancing
        xs = torch.maximum(1-discrim_x, self._empty)
        ys = torch.maximum(1+discrim_y, self._empty)
        l = torch.mean(xs) + torch.mean(ys)
        self.plot_average.update(l) # keep updating moving average anyway for logging purposes
        return l

```

```

class FeatureLoss(Loss):
    def __init__(self, weight) -> None:
        super().__init__("Discriminator Feature Loss", weight)

    def get_raw_value(self, feat_x: list[list[torch.Tensor]], feat_y: list[list[torch.Tensor]])
-> torch.Tensor: # lists are both of the form discrim, layer, tensor<b, x, y>
        total = 0.
        assert len(feat_x) == len(feat_y) # these dims should be the same, not prod code either
so better to hard crash

        for disc_x, disc_y in zip(feat_x, feat_y):
            assert len(disc_x) == len(disc_y) # these dims should be the same, not prod code
either so better to hard crash
            disc_total = 0.
            for l_x, l_y in zip(disc_x, disc_y):
                disc_total += F.l1_loss(l_x, l_y)

        total += disc_total / len(disc_x)
        return total / len(feat_x) # the above is just F.l1_loss(feat_x, feat_y), but we're using
lists for some stuff not tensors so have to do it manually

```

discriminator_model.py

```

import torch
from torch.nn.utils import weight_norm

```

```

from torch import nn
import torchaudio
from model import datasets
from torch.utils.data import DataLoader
import math
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir="logs_test/")
LEAKY_RELU = 0.2
INIT_MEAN = 0.0
INIT_STD = 0.01

def get_padding_nd(kernel_sizes: list[int], dilations: list[int]) -> list[int]:
    return [math.floor((kernel_size-1) * dilation / 2) for kernel_size, dilation in
zip(kernel_sizes, dilations)]

class STFTDiscriminator(torch.nn.Module):
    """ A combination between soundstream and encdec """
    def __init__(self,
        scale,
        kernel_size=(3,8)) -> None:
        super().__init__()
        self._transform = torchaudio.transforms.Spectrogram(n_fft=(scale - 1) * 2,
win_length=scale, normalized=False, power=None, center=False, pad_mode=None)

        self._convs = nn.ModuleList([
            weight_norm(nn.Conv2d(in_channels=1, out_channels=32, kernel_size=kernel_size,
padding=get_padding_nd(kernel_size, (1,1)))),
            weight_norm(nn.Conv2d(in_channels=32, out_channels=32, kernel_size=kernel_size,
stride=(1,2), dilation=(1,1), padding=get_padding_nd(kernel_size, (1,1)))),
            weight_norm(nn.Conv2d(in_channels=32, out_channels=32, kernel_size=kernel_size,
stride=(1,2), dilation=(2,1), padding=get_padding_nd(kernel_size, (2,1)))),
            weight_norm(nn.Conv2d(in_channels=32, out_channels=32, kernel_size=kernel_size,
stride=(1,2), dilation=(4,1), padding=get_padding_nd(kernel_size, (4,1)))),
            weight_norm(nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(3,3),
padding=get_padding_nd((3,3), (1,1)))),
            weight_norm(nn.Conv2d(in_channels=32, out_channels=1, kernel_size=(3,3),
padding=get_padding_nd((3,3), (1,1))))
        ])

        for conv in self._convs:
            nn.init.normal_(conv.weight, INIT_MEAN, INIT_STD)

    def forward(self, x):
        spec: torch.Tensor = self._transform(x)
        spec = torch.cat([spec.real, spec.imag], dim=-1)
        spec = spec.transpose(-1, -2)
        internal_activations = [] # will be in form L B X Y

        for conv in self._convs:
            spec = conv(spec)
            spec = torch.nn.functional.leaky_relu(spec, LEAKY_RELU)
            internal_activations.append(spec)

        logits = torch.nn.functional.tanh(spec)

        return logits.view(logits.shape[0], -1).mean(dim=1), internal_activations

class MultiScaleSTFTDiscriminator(torch.nn.Module):
    def __init__(self, scales = [2048, 1024, 512, 256, 128]) -> None:
        super().__init__()
        self._discrims = torch.nn.ModuleList([STFTDiscriminator(scale) for scale in scales])

```

```

def forward(self, x):
    logit, disc_feature = self._discrims[0](x)
    result = logit.unsqueeze(0).transpose(0, 1)

    features = [disc_feature] # need to use lists :( due to differing lengths of dimensions,
    too hard to exclude at the other end

    for discrim in self._discrims[1:]:
        logit, disc_feature = discrim(x)
        result = torch.concat((logit.unsqueeze(0).transpose(0, 1), result), dim=1)
        features.append(disc_feature)
    return result, features # D B L X Y to B D L X Y, keep batch first (convention)

```

moving_average.py

```

from torch import nn
import torch

class MovingAverage(nn.Module):
    def __init__(self, context_len: int, intial_val=1) -> None:
        super().__init__()
        self.register_buffer("_window", torch.full(torch.empty(context_len, dtype=torch.float32),
        intial_val))
        self.average = intial_val
        self._context_len = context_len

        weights = self.get_weights(context_len)
        self.register_buffer("_weights", weights)
        self._weight_sum = torch.sum(weights)
        self._length = 0

    @torch.no_grad()
    def update(self, new_value) -> torch.Tensor:
        self._length += 1
        self._window = torch.roll(self._window, 1)
        self._window[0] = new_value

        s = self._weight_sum if self._length >= self._context_len else
        torch.sum(self._weights[:self._length])
        result = torch.sum(self._window[:self._length] * self._weights[:self._length]) / s
        self.average = result
        return result

    def get_weights(self, context_len) -> torch.Tensor:
        raise NotImplemented

class EMA(MovingAverage):
    """
    Exponential Moving Average

    The weight of each element decreases with the form 1/n. IDK if this is actually exponential
    moving average, however it was the first thing
    I found and I'm about to run out of mobile data.
    """

```

```

"""
def __init__(self, *args, beta, **kwargs) -> None: # the parameters here do a bit of
trolling at points, when bugs look here
    self._beta = beta
    super().__init__(*args, **kwargs)

def get_weights(self, context_len) -> torch.Tensor:
    weights = torch.linspace(0, context_len-1, context_len)
    weights = self._beta ** weights
    return weights
class SMA(MovingAverage):
    """
    Simple moving average

    The weight of all elements in the context length is equal

    This could probably be implemented more efficiently, however this is pretty clean.
    Only thing is a useless elementwise multiply (we don't care about slow initialisation), but
    that is at training only.
    """

    def get_weights(self, context_len) -> torch.Tensor:
        weights = torch.ones(context_len)
        return weights

```

model/analysis/

dead_analysis.py

This file was used to justify the need of dealing with dead codewords, and then for testing that the solution effectively uses the codeword spread.

```

import sys, os

from model.datasets import *
import torch
import utils
import matplotlib.pyplot as plt
import itertools
from torchaudio import transforms
import torch.nn.functional as F
from torch import nn
import models
import torch
import vq
from model.loss import loss_functions
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm
writer = SummaryWriter(log_dir="logs/")

context_length = 240 * 5
batch_size = 64
TENSORBOARD_INTERVAL = 25

```

```

VALID_SAVE_INTERVAL = 100
device = "cuda" if torch.cuda.is_available() else "cpu"

train_data = LibriTTS(context_length)
valid_loader = ValidateSpeechDataset(48)

train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_loader, batch_size=batch_size)

encoder = models.Encoder(256).to(device)
quantizer = vq.RVQ(8, 1024, 256).to(device)
decoder = models.Decoder(256).to(device)

encoder.load_state_dict(torch.load("logs/encoder.state"))
decoder.load_state_dict(torch.load("logs/decoder.state"))
quantizer.load_state_dict(torch.load("logs/quantizer.state"))

encoder.eval()
decoder.eval()
quantizer.eval()
quantize_train = None
data = torch.zeros((8, 1024), dtype=torch.float32).to(device)
with torch.no_grad():
    for i, truth in enumerate(tqdm(train_dataloader)):
        truth = truth.to(device)
        outputs = encoder(truth)
        quantizer_loss = 0

        outputs = torch.transpose(outputs, 1, 2) # BCT -> BTC

        _, indices, _ = quantizer(outputs)
        indices = torch.flatten(indices, end_dim=-2).t()
        o = torch.nn.functional.one_hot(indices, num_classes=1024)
        counts = torch.sum(o, dim=1)
        data = data + counts
        # indices = torch.flatten(indices).cpu().detach().numpy()

print(data.shape)
for i in range(8):
    length = torch.sum(data[i])
    for x in range(100):
        print(i, x, len((data[i] < (x / 100) * length).nonzero(as_tuple=True)[0]))

    tk = torch.topk(data[i], 100)
    print(torch.sum(data[i]).item())
    print(torch.sum(tk.values).item() / torch.sum(data[i]).item())

```

inference/

main.py

```

import typing
from PyQt6.QtCore import QObject
from PyQt6.QtWidgets import QApplication, QWidget, QPushButton, QMainWindow, QGridLayout,
QHBoxLayout, QLabel, QFileDialog, QVBoxLayout, QFrame, QSlider, QProgressBar, QMessageBox

```

```

from PyQt6 import QtCore, QtWidgets
import sys
from model import models
import inference
import math
import time
from typing import Callable
from threading import Thread
import traceback

```

```

class Worker(QtCore.QObject):
    finished = QtCore.pyqtSignal()
    error = QtCore.pyqtSignal(Exception)
    progress = QtCore.pyqtSignal(float)
    def __init__(self, callable, *args, **kwargs) -> None:
        super().__init__()
        self.callable = callable
        self.args = args
        self.kwargs = kwargs

    def run(self):
        self.kwargs["progress_callback"] = self.progress.emit
        try:
            self.callable(*self.args, **self.kwargs)
        except Exception as e:
            traceback.print_exception(e)
            self.error.emit(e)
        self.finished.emit()

```

```

class ProgressBar(QWidget):

    def __init__(self, callable, args, main_window) -> None:
        super().__init__()
        self.main_window = main_window
        self.pbar = QProgressBar()
        layout = QVBoxLayout()

        layout.addWidget(self.pbar)
        self.setLayout(layout)

        self._thread = QtCore.QThread()
        self.worker = Worker(callable, *args)

        self.worker.moveToThread(self._thread)

        self._thread.started.connect(self.worker.run)
        self.worker.progress.connect(self.update)
        self.worker.finished.connect(self.finished)
        self.worker.error.connect(self.exception)

        # house keeping
        self.worker.finished.connect(self._thread.quit)
        self.worker.finished.connect(self.worker.deleteLater)
        self._thread.finished.connect(self._thread.deleteLater)

        self._thread.start()
        self.show()

```



```

        self.main_window.setDisabled(True)

def update(self, amount):
    self.pbar.setValue(int(100 * amount))

def finished(self):
    self.main_window.setDisabled(False)
    self.close()

def exception(self, e):
    if isinstance(e, inference.InvalidHashException):
        button = QMessageBox.critical(
            None,
            "Error",
            "This file was encoded with a different model, please select the correct model",
            buttons=QMessageBox.StandardButton.Ok
        )
    else:
        button = QMessageBox.critical(
            None,
            "Error",
            "An error has occurred, try again",
            buttons=QMessageBox.StandardButton.Ok
        )

```

```

class FileSelectionWidget(QFrame):
    file_chosen = QtCore.pyqtSignal(str)
    def __init__(self, name: str, filter: str, open=True) -> None:
        super().__init__()
        self.filter = filter
        self.name = name
        self.open = open

        self.setObjectName("FileSelectionWidget")

        self.setStyleSheet("""
            FileSelectionWidget {
                background-color: #b9b9b9;
                color: white;
                font-size: 20px;
                margin: 2px;
                border-radius: 5px;
            }""")

        layout = QHBoxLayout()
        layout.setContentsMargins(3, 3, 3, 3)
        self.select_button = QPushButton("Select")
        self.select_button.clicked.connect(self.choose_file)
        layout.addWidget(self.select_button)

        self._file_name = QLabel("No file selected")
        layout.addWidget(self._file_name)

        self.main_widget = QWidget()
        self.setLayout(layout)

```

```

        self.setMaximumHeight(50)
        self.filename = ""

    def choose_file(self):
        if self.open:
            fname = QFileDialog.getOpenFileName(self, 'Open file',
                                                '', self.filter)[0]
        else:
            fname = QFileDialog.getSaveFileName(self, 'Save file',
                                                '', self.filter)[0]

        self.filename = fname
        self._file_name.setText(fname if fname else "No file selected")
        self.file_chosen.emit(fname)

```

```

class ModelSelectionWidget(QWidget):
    updated = QtCore.pyqtSignal(models.Models)

    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        layout.addWidget(QLabel("Model Selection"))
        self.file_select = FileSelectionWidget("Model File", "Model files (.saved)")
        self.file_select.file_chosen.connect(self.model_update)

        layout.addWidget(self.file_select)
        self.setLayout(layout)

    def model_update(self):
        try:
            m = models.Models.load(self.file_select.filename)
            self.updated.emit(m)
        except Exception as e:
            button = QMessageBox.critical(
                None,
                "Error",
                f"Failed to load model due to: {e}",
                buttons=QMessageBox.StandardButton.Ok
            )

```

```

class ModelStatsWidget(QWidget):
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        layout.addWidget(QLabel("Model Stats"))
        self.epochs = QLabel("Epochs: ?")
        self.ncodes = QLabel("Codes: ?")
        self.nbooks = QLabel("Codebooks: ?")
        self.bitrate = QLabel("Bitrate: ?")

        layout.addWidget(self.epochs)
        layout.addWidget(self.ncodes)
        layout.addWidget(self.nbooks)
        layout.addWidget(self.bitrate)
        self.setLayout(layout)

    def update(self, models: models.Models):

```

```

self.epochs.setText(f"Epochs: {models.epochs}")
self.ncodes.setText(f"Codebooks: {models.ncodes}")
self.nbooks.setText(f"Books: {models.nbooks}")

bitrate = 16000 / models.ctx_len * models.nbooks * math.log2(models.ncodes) / 1000
self.bitrate.setText(f"Bitrate: {bitrate:.2f} kbps")

```

```

class EncodeDecodeWidget(QWidget):
    def __init__(self, name, input_filter, output_filter, main_window):
        super().__init__()
        self.main_window = main_window
        layout = QVBoxLayout()
        layout.addWidget(QLabel(name))
        self.input_select = FileSelectionWidget("Input", input_filter)
        self.output_select = FileSelectionWidget("Output", output_filter, open=False)

        self.input_select.file_chosen.connect(self.update_button)
        self.output_select.file_chosen.connect(self.update_button)

        self.run_button = QPushButton("Run")
        self.run_button.setDisabled(True)
        self.run_button.clicked.connect(self.run)
        layout.addWidget(self.input_select)
        layout.addWidget(self.output_select)
        layout.addWidget(self.run_button)
        self.setLayout(layout)

        self.model = None

    def model_set(self, model):
        self.model = model
        self.update_button()

    def update_button(self):
        if self.model != None and self.input_select.filename != "" and self.output_select.filename != "":
            self.run_button.setDisabled(False)
        else:
            self.run_button.setDisabled(True)

    def run(self):
        pass

```

```

class EncodeWidget(EncodeDecodeWidget):
    def __init__(self, main_window):
        super().__init__("Encode", "Audio files (*.wav)", "Audio files (*.sc)", ".sc", main_window)

    def run(self):
        self.progress = ProgressBar(inference.wav_to_sc, (self.input_select.filename, self.output_select.filename, self.model), self.main_window)
        self.progress.show()
        # progress has can't be a local variable as otherwise gc becomes greedy and "eats" it before it's been displayed

```

```

class DecodeWidget(EncodeDecodeWidget):
    def __init__(self, main_window):

```

```

        super().__init__("Decode", "Audio files (*.sc)", "Audio files (*.wav)", ".wav",
main_window)

```

```

    def run(self):
        self.progress = ProgressBar(inference.sc_to_wav, (self.input_select.filename,
self.output_select.filename, self.model), self.main_window)
        self.progress.show()

```

```

class EncodeDecodeContainer(QFrame):
    def __init__(self, main_window):
        super().__init__()

        self.switch = QPushButton("Mode")
        self.switch.setCheckable(True)
        self.switch.clicked.connect(self.switch_ec)

        self.encode = EncodeWidget(main_window)
        self.decode = DecodeWidget(main_window)

        layout = QVBoxLayout()
        layout.addWidget(self.switch)
        layout.addWidget(self.encode)
        layout.addWidget(self.decode)

        self.decode.setVisible(False)

        self.setLayout(layout)

    def switch_ec(self, value):
        self.decode.setVisible(value)
        self.encode.setVisible(not value)

    def model_set(self, model):
        self.encode.model_set(model)
        self.decode.model_set(model)

```

```

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("My App")

        self.model_select = ModelSelectionWidget()
        self.model_stats = ModelStatsWidget()
        self.encode_decode = EncodeDecodeContainer(main_window=self)

        layout = QGridLayout()
        layout.addWidget(self.model_select, 0, 0, 2, 2)
        layout.addWidget(self.model_stats, 2, 0, 2, 2)
        layout.addWidget(self.encode_decode, 0, 2, 4, 2)

        self.model_select.updated.connect(self.model_stats.update)
        self.model_select.updated.connect(self.on_model_select)
        self.model_select.updated.connect(self.encode_decode.model_set)

        self.model = None

```

```

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
        self.setFixedSize(600, 300)

    def on_model_select(self, model):
        self.model = model

app = QApplication(sys.argv)

window = MainWindow()
window.show() # IMPORTANT!!!! Windows are hidden by default.

# Start the event loop.
app.exec()

```

inference.py

```

import torch
from model import models
import torch
import file_structure
from model.utils import norm
import torchaudio
import traceback
from typing import Callable

PADDING_LEN = 10
SEGMENT_LEN = 290

def split_with_padding(data, split_len: int, padding_len: int):
    current = 0
    while current < len(data):
        yield data[max(current - padding_len, 0): current + split_len + padding_len]
        current += split_len + padding_len

class InvalidHashException(Exception):
    pass

@torch.inference_mode()
def sc_to_wav(path: str, output_path: str, model: models.Models, progress_callback:
Callable[[float], None] = None):
    model.eval()
    f = file_structure.File.read(path)

    if f.model_hash != model.hash:
        raise InvalidHashException

    indices = torch.tensor(f.data)

```

```

result_wav = None

segment_count = math.ceil( indices.shape[0] / ((PADDING_LEN + SEGMENT_LEN)))
for i, segment in enumerate(split_with_padding(indices, SEGMENT_LEN, PADDING_LEN)):
    segment_wave = model.decode(segment.unsqueeze(0)).squeeze(0)
    if result_wav == None:
        result_wav = segment_wave[..., :(SEGMENT_LEN+int(PADDING_LEN / 2)) * model.ctx_len]
    else:
        without_padding = segment_wave[..., (int(PADDING_LEN / 2)) * model.ctx_len :
(SEGMENT_LEN+PADDING_LEN+int(PADDING_LEN / 2)) * model.ctx_len]
        result_wav = torch.concat((result_wav, without_padding), dim=1)

    if progress_callback != None:
        progress_callback(i / segment_count)

torchaudio.save(output_path, result_wav.cpu(), sample_rate=16000)

if progress_callback != None:
    progress_callback(1)
return result_wav

@torch.inference_mode()
def wav_to_sc(path: str, output_path: str, model: models.Models, progress_callback:
Callable[[float], None] = None):
    model.eval()
    sound, sample_rate = torchaudio.load(path)
    audio_data = norm(sound)

    if sample_rate != 16000:
        audio_data = torchaudio.functional.resample(sound, orig_freq=sample_rate, new_freq=16000)

    codebooks = None

    segment_count = math.ceil( audio_data.shape[1] / ((PADDING_LEN + SEGMENT_LEN) *
model.ctx_len))
    for i, segment in enumerate(split_with_padding(audio_data.squeeze(0), SEGMENT_LEN *
model.ctx_len, PADDING_LEN * model.ctx_len)):
        segment_books = model.encode(segment.unsqueeze(0)).squeeze(0)
        if codebooks == None:
            codebooks = segment_books[:SEGMENT_LEN+int(PADDING_LEN / 2)]
        else:
            without_padding = segment_books[int(PADDING_LEN / 2) :
SEGMENT_LEN+PADDING_LEN+int(PADDING_LEN / 2)]
            codebooks = torch.concat((codebooks, without_padding), dim=0)

        if progress_callback != None:
            progress_callback(i / segment_count)

    f = file_structure.File(codebooks, model_hash=model.hash,
data_bit_depth=math.ceil(math.log2(model.ncodes)), n_codebooks=model.nbooks)
    f.write(output_path)
    if progress_callback != None:
        progress_callback(1)
    return f

@torch.no_grad()
def wav_to_sc_short(path: str, output_path: str, model: models.Models):

```

```

        """ This does no clever stuff to reduce memory usage, ONLY intended as a test for if result
        is the same """
        try:
            sound, sample_rate = torchaudio.load(path)
            sound = norm(sound)

            if sample_rate != 16000:
                sound = torchaudio.functional.resample(sound, orig_freq=sample_rate, new_freq=16000)

            audio_data = sound.unsqueeze(0)
            codebooks = model.encode(audio_data)

            f = file_structure.File(codebooks.squeeze(0), model_hash=model.hash,
            data_bit_depth=math.ceil(math.log2(model.ncodes)), n_codebooks=model.nbooks)
            f.write(output_path)
            return f
        except Exception: # absolutely any exception in here and we just return failed
            traceback.print_exc()
            return False

```

```

@torch.no_grad()
def sc_to_wav_short(path: str, output_path: str, model: models.Models, progress_callback:
Callable[[float], None] = None):
    """ This does no clever stuff to reduce memory usage, ONLY intended as a test for if result
    is the same """
    try:
        f = file_structure.File.read(path)

        indices = torch.tensor(f.data).unsqueeze(0)

        out = model.decode(indices).squeeze(0)
        torchaudio.save(output_path, out.cpu(), sample_rate=16000)

        return out
    except Exception:
        traceback.print_exc()
        return False # we have failed

```

file_structure.py

```

from operator import ior
import functools
from typing import Iterator
import timeit
import math

```

```

class File:
    MAGIC_NUM = 0x12121212 # TODO think of a more meaningful magic number, has to be 8 hex
    digits
    @staticmethod
    def read(filepath):

```

```

reader = FileReader(filepath)
num = reader.read_32_bit()
if num != File.MAGIC_NUM: # check that the file is actually the correct type
    print(hex(num))
    raise InvalidMagicNumberException

length = reader.read_32_bit()
n_codebooks = reader.read_byte()
data_bit_depth = reader.read_byte()

model_hash = reader.read_n_bits(128)

reader.read_byte() # padding

data = []
for _ in range(length):
    sample = []
    for _ in range(n_codebooks):
        sample.append(reader.read_n_bits(n_bits=data_bit_depth))
    data.append(sample)
reader.close()
return File(data, model_hash=model_hash, data_bit_depth=data_bit_depth, length=length,
n_codebooks=n_codebooks)

def __init__(self, data: Iterator[Iterator[int]], data_bit_depth: int, model_hash: str,
length=None, n_codebooks=None) -> None:
    self.length = length if length != None else len(data)
    self.n_codebooks = n_codebooks if n_codebooks != None else len(data[0])
    self.data_bit_depth = data_bit_depth
    self.data = data
    self.model_hash = model_hash

def write(self, filepath):
    writer = FileWriter(filepath)

    writer.write_32_bit(File.MAGIC_NUM)

    writer.write_32_bit(self.length)
    writer.write_byte(self.n_codebooks)
    writer.write_byte(self.data_bit_depth)
    writer.write_n_bits(self.model_hash, 128)

    writer.write_byte(0) # write a padding byte, not really needed, but makes analysis easier

    x = 0
    for sample in self.data:
        for codebook in sample:
            writer.write_n_bits(codebook, self.data_bit_depth)

    writer.close()

class InvalidMagicNumberException(Exception):
    pass

class EOFError(Exception):
    pass

```



```

class FileReader(): # A java DataInputStream inspired reader, you can probably see my java
origins coming in through here, but we work on the level of bits, not bytes
    def __init__(self, path: str) -> None:
        self._file = open(path, mode="rb")
        self._bytes = [int.from_bytes([byte], "big") for byte in self._file.read()]
        self._front_pointer = 0 # The index of the next bit to be read

    def read_bit(self):
        if self._front_pointer // 8 == len(self._bytes):
            return EOFError
        index_in_byte = self._front_pointer % 8
        mask = 0b10000000 >> index_in_byte
        bit = (self._bytes[self._front_pointer // 8] & mask) >> (7-index_in_byte)
        self._front_pointer += 1
        return bit

    def read_byte(self):
        return self.read_n_bits(8)

    def read_n_bits(self, n_bits):
        if (self._front_pointer + n_bits - 1) // 8 == len(self._bytes):
            raise EOFError
        f_in_byte = (self._front_pointer) % 8

        if f_in_byte == 0 and n_bits >= 8: # this doesn't always help, but can sometimes bring
around great benefits
            val = self._bytes[self._front_pointer // 8]
            self._front_pointer += 8
            if n_bits > 8:
                return val << (n_bits - 8) | self.read_n_bits(n_bits-8)
            else:
                return val << (n_bits - 8)

        fit = 8-f_in_byte

        space_after = max(fit-n_bits, 0)
        space_before = 0 # max(8 - f_in_byte - n_bits, 0)
        # we want to be 0, unless all can fit in remainging byte

        m1 = (1 << fit) - 1 # eliminate before
        m2 = ~((1 << space_after) - 1) # eliminate after
        mask = m1 & m2
        val = ((self._bytes[self._front_pointer // 8] & mask) >> space_after) << (space_before)
        self._front_pointer += min(fit, n_bits)
        remaining = n_bits - fit
        if remaining > 0:
            return (val << remaining) + self.read_n_bits(remaining)
        else:
            return val

    def read_short(self):
        b1 = self.read_byte()
        b2 = self.read_byte()
        return (b1 << 8) + b2

    def read_32_bit(self):
        s1 = self.read_short()

```

```

        s2 = self.read_short()
        return (s1 << 16) + s2

    def close(self):
        self._file.close()

class FileWriter():
    def __init__(self, path: str) -> None:
        self.file = open(path, mode="wb")
        self.bytes = []
        self.front_pointer = -1

    def write_bit(self, bit):
        bit &= 1 # just in case everything else has gone wrong
        self.front_pointer += 1
        if len(self.bytes) <= (self.front_pointer // 8): # we need to be careful here incase
            someone has messed with self.front_pointer, and it hasn't necessarily incremented how we'd expect
            self.bytes.append(0)

        index_in_byte = self.front_pointer % 8
        self.bytes[self.front_pointer // 8] |= (bit << (7-index_in_byte)) # use or here to
        incase there is any existing data

    def write_byte(self, byte):
        self.write_n_bits(byte, n_bits=8)

    def write_short(self, short):
        self.write_byte(short >> 8)
        self.write_byte(short & 0xFF)

    def write_32_bit(self, integer): # java typing is so much better, not really sure how to
        handle signing, so for now, we don't
        self.write_short(integer >> 16)
        self.write_short(integer & 0xFFFF)

    def write_n_bits(self, to_write, n_bits): # this looks like a pain, and can probably be
        further improved in the future, but can't deny 2x speed improvement
        # print(self.front_pointer, n_bits)
        if (self.front_pointer + n_bits) // 8 >= len(self.bytes):
            self.bytes.append(0)

        f_in_byte = (self.front_pointer + 1) % 8

        if f_in_byte == 0 and n_bits >= 8: # the benefit of this changes dependant on bitdepth,
            ranging from 3x improvement, to none
            self.bytes[(self.front_pointer+1) // 8] = to_write >> (n_bits - 8)
            self.front_pointer += 8

        if n_bits > 8:
            self.write_n_bits(to_write, n_bits-8)
            return

        fit = 8 - f_in_byte
        space_after = max(fit - n_bits, 0) # how much space do we have after
        space_before = max(f_in_byte + (n_bits - 8), 0)

```

```

m1 = (1 << fit) - 1 # eliminate before
m2 = ~((1 << space_after) - 1) # eliminate after

mask = m1 & m2
result = ((to_write << space_after) >> space_before) & mask
self.bytes[(self.front_pointer+1) // 8] |= result

self.front_pointer += min(fit, n_bits)

remaining = n_bits - fit
if remaining > 0:
    self.write_n_bits(to_write, remaining)

def close(self):
    self.file.write(bytes(byte & 0xFF for byte in self.bytes))
    self.file.close()

```

datasets/

process_dataset.sh

This file is used to process datasets before training can be done, LibriSpeech datasets come as .flac files in several thousands of folders depending on speaker, so these will have to be converted over to wav files and moved into a single folder before training can begin.

```

#!/bin/bash
folder=LibriSpeech/train-clean-360

for file in $(find "$folder" -type f -iname "*.flac")
do
    name=$(basename "$file" .flac)
    dir=$(dirname "$file")
    if [ ! -f "speech_train/$name.wav" ]; then
        ffmpeg -i "$file" "speech_train/$name".wav -hide_banner -loglevel error
        echo "$name"
    fi
    rm $file
done

```

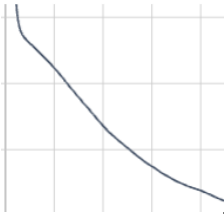
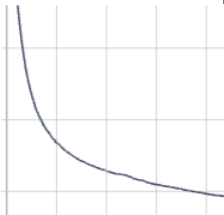
Testing

Due to significant parts of this project not being user facing, a significant amount of testing is done using unit tests and a few full system tests. Lots of the tests below have code with them that is used to generate the result, in many cases this code will give a pass or a fail, however in others looking at a graph, or seeing a general trend is required, in this case I have provided the values output.

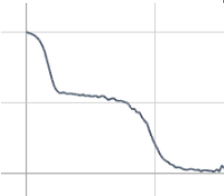
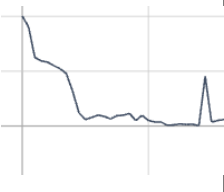
Model Testing

Residual Vector Quantization

Test Number	Test	Expected Result	Actual Result	Pass / Fail
1.01	kmeans clustering should reduce quantization loss	Average quantization loss before is less than after	Loss Before: 87.5 Loss After: 81.0	Pass
	<pre>import torch import vq import tqdm point_len = 5 quantizer = vq.VQ(1024, point_len).to("cuda") with torch.no_grad(): after = [] before = [] # generate a random array of vectors of length x for i in tqdm.tqdm(range(100)): # tqdm is a library used for console progress bars points = torch.normal(0,20,(20000, point_len)).to("cuda") before.append(quantizer(points)[2]) quantizer.frozen_kmeans(points, torch.Tensor([False] * 1024).to(torch.bool), kmeans_iters=1) x = quantizer(points)[2] after.append(x) print(sum(before) / len(before)) print(sum(after) / len(after))</pre>			
1.02	Vector Cache concat concatenates vectors correctly	The first input is returned concatenated	[1] then [2] correctly goes to [1, 2]	Pass

		with the second		
	<pre> cache = vq.VectorCache(3) cache.add_vector(torch.Tensor([1])) self.assertTrue(torch.equal(cache.concat(), torch.Tensor([1]))) cache.add_vector(torch.Tensor([2])) self.assertTrue(torch.equal(cache.concat(), torch.Tensor([1, 2]))) </pre>			
1.03	Vector Cache is limited to the correct length	Only the last 3 vectors given are saved	When given the numbers 0-49 individually, [47, 48, 49] are returned	Pass
	<pre> cache = vq.VectorCache(3) for i in range(50): cache.add_vector(torch.Tensor([i])) self.assertTrue(torch.equal(cache.concat(), torch.Tensor([47, 48, 49]))) </pre>			
1.04	Vector Quantizer learns data	Quantization loss decreases over iterations		Pass
1.05	Residual Vector Quantizer learns data	Quantization loss decreases over iterations		Pass
	<pre> residual = False x = vq.RVQ(5, 5, 5) if residual else vq.VQ(5, 5) optimizer = torch.optim.SGD(x.parameters(), lr=0.01, momentum=0.9) writer = SummaryWriter(log_dir="logs_test/") data = torch.normal(mean=2, std=2, size=(200, 5)) for i in tqdm.tqdm(range(1000)): values, indexes, loss = x(data) # print(loss) writer.add_scalar("loss", loss, i) optimizer.zero_grad() loss.backward() optimizer.step() </pre>			

Discriminator Model

2.01	The discriminator model discriminates between noise and speech	The value of loss decreases	Loss vs Steps graph 	Pass
2.02	The multiscale discriminator model discriminates between noise and speech	The value of loss decreases		Pass
<pre> multiscale = False loader = DataLoader(datasets.LibriTTS(240*8), 20) random_loader = DataLoader(datasets.RandomAudioDataset(240*8, 100), 20) discrim = MultiScaleSTFTDiscriminator([512, 256]) if multiscale else STFTDiscriminator(256) optim = torch.optim.Adam(discrim.parameters(), lr=0.0005, betas=[0.5, 0.9]) discrim.train() i = 0 while True: for _, (actual, random) in enumerate(zip(loader, random_loader)): x = discrim(actual)[0] y = discrim(random)[0] a = 1-x # we want x to be high b = 1+y # we want y to be low a[a<0] = 0 # if x already above 1 we want to ignore in the loss b[b<0] = 0 # if y already below -1 we want to ignore in the loss loss = torch.mean(a) loss += torch.mean(b) optim.zero_grad() loss.backward() optim.step() print(i, loss) writer.add_scalar("loss", loss.item(), i) i += 1 </pre>				

Moving Average Tests

3.01	Simple Moving Average works	The below code passes	The below code passes	Pass
	<pre>def test_simple(self): simple_average = SMA(100) self.assertEqual(simple_average.average, 1, "Initial average should be 1") self.assertEqual(simple_average.update(100), 1.99) self.assertEqual(simple_average.update(100), 2.98) for _ in range(100): simple_average.update(100) self.assertEqual(simple_average.average, 100)</pre>			
3.02	Exponential Moving Average works	The below code passes	The below code passes	Pass
	<pre>def test_exponential(self): exponential_average = EMA(100) self.assertEqual(exponential_average.average, 1, "Initial average should be 1") self.assertAlmostEqual(exponential_average.update(100).item(), 20.08, places=2) self.assertAlmostEqual(exponential_average.update(100).item(), 29.63, places=2) for _ in range(100): exponential_average.update(100) self.assertEqual(exponential_average.average, 100)</pre>			
3.03	Setting the initial value works	The below code passes	The below code passes	Pass
	<pre>def test_initial_value(self): self.assertEqual(EMA(10, initial_val=10).average, 10) sma = SMA(10, initial_val=10) self.assertTrue(torch.all(sma._window == 10)) # check that the window is all initialised correctly</pre>			

Inference Testing

File Testing

4.01	File writer and reader works for bits	Read is the same as written	Read is the same as written	Pass
	<pre>def test_bit(self): writer = file_structure.FileWriter("/tmp/x.test") order = [random.randrange(0, 2) for i in range(1000)] for i in order: writer.write_bit(i) writer.close() reader = file_structure.FileReader("/tmp/x.test") for i in order: self.assertEqual(reader.read_bit(), i) reader.close()</pre>			
4.02	File writer and reader works for bytes	Read is the same as written	Read is the same as written	Pass
	<pre>def test_byte(self): writer = file_structure.FileWriter("/tmp/x.test") order = [random.randrange(0, 256) for i in range(1000)] for i in order: writer.write_byte(i) writer.close() reader = file_structure.FileReader("/tmp/x.test") for i in order: self.assertEqual(reader.read_byte(), i) reader.close()</pre>			
4.03	File writer and reader works for any length numbers	Read is the same as written	Read is the same as written	Pass
	<pre>def test_nbit(self): writer = file_structure.FileWriter("/tmp/Y.test") order = [(bit_depth, random.randrange(0, 2 ** (bit_depth - 1))) for bit_depth in [random.randrange(1, 50) for i in range(100000)]] # don't particularly like the random here, but enough iterations mean likelihood of false working is very low for i in order:</pre>			

	<pre> writer.write_n_bits(i[1], i[0]) writer.close() reader = file_structure.FileReader("/tmp/Y.test") for i in order: self.assertEqual(reader.read_n_bits(i[0]), i[1]) reader.close() </pre>
--	--

Inference Testing

5.01	Test Encode doesn't fail	wav_to_sc returns True	wav_to_sc returns True	Pass
	<pre> def test_aencode(self): model = models.Models.load("logs-t/epoch46/models.saved") input = "samples/epoch0/0-clean.wav" output = "test/a.sc" out = inference.wav_to_sc(input, output, model) self.assertTrue(out) </pre>			
5.02	Test memory saving encode has same result to a normal encode	The resultant encoding is equal	The resultant encoding is equal	Pass
	<pre> def test_encodeshortlong(self): model = models.Models.load("logs-t/epoch46/models.saved") input = "samples/epoch0/0-clean.wav" output = "test/a.sc" long = inference.wav_to_sc(input, output, model) short = inference.wav_to_sc_short(input, output, model) same = torch.eq(long.data, short.data) self.assertTrue(torch.all(same), "All equal") </pre>			
5.03	Test decode doesn't fail	sc_to_wav returns a Tensor	sc_to_wav returns a tensor	pass
	<pre> def test_decode(self): model = models.Models.load("logs-t/epoch46/models.saved") input = "test/a.sc" output = "test/tmp.wav" out = inference.sc_to_wav(input, output, model) self.assertIsInstance(out, torch.Tensor) </pre>			
5.04	Memory saving decode has similar result to normal decode	The results are similar	l1 loss between the two is almost 0	pass

	<pre> def test_decodeshortlong(self): model = models.Models.load("logs-t/epoch46/models.saved") output = "samples/epoch0/0-clean.wav" input = "test/a.sc" long = inference.sc_to_wav(input, output, model) short = inference.sc_to_wav_short(input, output, model) ll_diff = torch.nn.functional.l1_loss(long, short) self.assertAlmostEqual(ll_diff.item(), 0) </pre>			
5.05	Padding split function works as intended	The list is split as intended	[[1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8], [8, 9, 10]]	Pass
	<pre> def test_paddingsplit(self): d = [1,2,3,4,5,6,7,8,9,10] a = inference.split_with_padding(d, 1, 1) result = [[1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8], [8, 9, 10]] self.assertEqual(list(a), result) </pre>			

GUI Tests

Due to this testing being visual I have created a video as evidence for the completion of these tests:

<https://youtu.be/xOBi-f0OOwE>

Model Selection and Stats

6.01	Model selection warns when hash is invalid	A message appears warning that the hash is invalid	Video at 0:00	Pass
6.02	Model selection warns when using outdated model file	A message appears warning that the model file is invalid	Video at 0:11	Pass
6.03	Model selection successfully opens valid file	No errors when opening valid file	Video at 0:20	Pass
6.04	Model stats shows correct stats	For high quality bitrate should be 3.3 kbps	Video at 0:31	Pass

6.05	Model stats update	When model changed to low quality the codebook count goes to 4	Video at 0:38	Pass
------	--------------------	--	---------------	------

Encode Decode Widget

7.01	Encode Decode switch works	Clicking the switch should switch to decode mode, then encode mode	Video at 0:46	Pass
7.02	Selecting input and output works	The inputs and outputs should appear in the boxes	Video at 0:52	Pass
7.03	Widgets save values	When a value has been entered as the input of encode it should remain the input of encode	Video at 1:06	Pass
7.04	The run button enables when all required are selected	Once a model, an input and an output have been selected the run button stops being greyed out.	Video at 1:11	Pass
7.05	Run button starts encoding.	A progress bar should appear and the main window is greyed out.	Video at 1:18	Pass
7.06	Run button starts decoding	A progress bar should appear and the main window is greyed out.	Video at 1:27	Pass
7.07	A variety of audio file input types work	.m4a, .mp3, .flac, .wav	Video at 1:34	Pass

		should all be able to be encoded without error.		
7.08	Invalid audio files don't cause the program to crash	An error should be shown and the user should be returned to the main window.	Video at 1:46	Pass
7.09	Invalid saved encoded files shouldn't cause the program to crash	An error should be shown to the user and the user should be returned to the main window.	Video at 1:50	Pass
7.10	Using the wrong model notifies the user	An error appears when the wrong model is used	Video at 1:55	Pass

Full System

8.01	The high quality model works	After audio is encoded and decoded it should be recognisable	Video at 2:05	Pass
8.02	The low quality model works	After audio is encoded and decoded it should be recognisable	Video at 2:10	Pass
8.03	The PESQ score is acceptable	The average PESQ score should be above 1.5 at 3 kbps	Actual: 1.81	Pass
	<pre> from pesq import pesq import glob from tqdm import tqdm import random import torch from model.models import Models </pre>			

	<pre> import torchaudio model = Models.load("model_saves/highqual.saved", device="cuda") audio_files = random.sample(glob.glob("datasets/speech_train/*.wav"), 500) values = [] with torch.no_grad(): for file in tqdm(audio_files): # inefficient to not batch, but this is only # iterating over 500, it's fine ref, rate = torchaudio.load(file) ref = ref.to("cuda") res = model.forward(ref)[0] torchaudio.save("evaluation/tmp/test.wav", res.cpu(), rate) values.append(pesq(rate, ref.squeeze(0).cpu().numpy(), res.squeeze(0).cpu().numpy(), 'nb')) print(sum(values) / len(values)) </pre>			
8.04	The trainer works	Loss graphs should decrease for non discriminator losses	All graphs look as they should	Pass
				

Evaluation

Evaluation Of Objectives

1. The program to encode speech using pytorch

1.1. A network to compress audio to an intermediary form

This objective has been achieved with the program using convolutional layers in a ResNet structure to encode audio into an intermediary form of 256 vectors representing 382 sound samples.

1.2. A quantization layer

1.2.1. The layer should be able to encode vectors into ids

The layer can encode vectors into ids as shown in the testing section.

1.2.2. The layer should be able to efficiently deal with "dead" codebook codes.

The model can efficiently deal with unused codebook codes using

2. A program to decode the encoded form

2.1. A dequantization layer

This was successful, however I ended up integrating it into the quantization layer instead of having them as two separate layers as that allows for less time to be taken during training.

2.1.1. The model should be able to accurately undo vector quantization

This objective was successful with the average quantization loss (the mean difference between the original and the output numbers), is below $1e-5$, which is low enough to not cause significant effect.

2.1.2. The layer should be able to deal with variable sized quantization layers.

Changing the size of the quantization layer is easy with just changing one number in one place allowing for increasing the number of codebooks or increasing the number of codewords per codebook.

2.2. A network to decompress audio from the intermediary form into the original form.

2.2.1. The resultant audio has to maintain the content of speech

As demonstrated in the testing section the high quality model maintains the words spoken which are easy to understand.

2.2.2. The resultant audio should have a PESQ score of above 1.5 at 3 kbps

The resultant audio of the high quality model has a PESQ score of 1.9 which is above 1.5. I would also expect this to continue to increase as more training goes on. Possibly an extra loss function looking at smoothness would help reduce some of the pops which also harm this score so it could be even higher.

3. The network needs to be trained

3.1. Multiple loss functions need to be designed

3.1.1. Perceptual loss based on existing speech to text models.

The whisper loss is incredibly effective, it is highly weighted in training and means that even on small training runs with very low quality models the content of the speech is generally discernible.

3.1.2. Perceptual loss based on a separate discriminator model

The separate discriminator model does work, and is also significantly weighted, however it does make the entire system much more unstable. Something to consider for future versions is diagnosing what aspect makes them unstable and modifying the model to try and make them less so.

3.1.3. Non-perceptual loss based on difference between the original and output

Whilst this does work it ended up acting instead of as the main training goal a "hint" as to how to best optimise the other objectives. For this reason it was generally weighted fairly low. It was implemented as the sum of the differences between the original and output waveform, and the sum of the differences between the original and output spectrum.

3.2. The trained model needs to be saved.

3.2.1. Includes network hyperparameters

All important hyperparameters related to using the model were saved, such as the codebook size and amount of codewords allowing for the network structure to be fully rebuilt for the parameters to be loaded into.

- 3.2.2. Includes the parameters of the encoder, decoder and quantizer
This was achieved using pytorch's built-in save model functionality, and then writing to a custom file. Using this allowed us to make use of pytorch's built-in compression, and wasn't a worry as file size of the model files isn't a massive issue. However in the future it would be worth considering a custom file writer allowing for smaller model files.

- 3.3. Loss progress needs to be displayed

- 3.3.1. Training progress should be output to TensorBoard

Training progress is output to TensorBoard, with graphs of all different loss functions and validation loss graphed and displayed for a user to show training progress. Some screenshots of the TensorBoard dashboard can be seen in the testing section.

4. A PyQt based GUI for a user to interact with the network

- 4.1. The user needs to be able to select a audio file and then output an encoded form

- 4.1.1. The user should be able to select a model

The user can select a model through selecting a model file, this means that the user can use models which have been pretrained, or use their own custom trained model.

- 4.1.2. Input files should be able to be of wav, mp3 audio formats

Input files can be of any of the following formats: .wav, .mp3, .m4a, and .flac. This objective was completed using pytorch audio's load function.

- 4.1.3. Needs to be able to deal with audio files of greater than 15 minutes in length

Memory saving measures implemented in inference.py mean that the model should be able to deal with hours of audio.

- 4.2. The user needs to be able to select and encoded file and output an audio file

- 4.2.1. The user should be able to select an output file format from .wav, .mp3.

The user can select from the following formats: .wav, .mp3, .m4a, and .flac. This can be done by just making the output file path have one of these endings.

4.2.2. The user needs to select an output file location and name

This can be done using the output select box and will only allow the selection of valid types, the run button is disabled until the user has selected an output file location.

4.2.3. A file format for the encoded file needs to be designed

4.2.3.1. Metadata needs to be saved to prevent errors where differing models are used.

Metadata is saved including a model hash allowing for the unique identification of a single model and the user to be notified if they are using the wrong model.

4.3. Application needs to be able to run on target customers computer

4.3.1. No need to use a GPU for inference

Whilst a GPU does increase the speed of inference, not having one doesn't mean that the application becomes unusable, in all of the tests above the application is run in CPU only mode.

4.3.2. Supports windows

Despite having been developed using WSL, and all of the tests being done using this, the application works on windows.

User Feedback

End User Feedback

After completing this project I asked my end user Jacob Liu his thoughts on the project. He responded as follows:

The project overall works well, with all key areas working cleanly, however there is a few improvements that I think should be made:

- The mode switch should be a toggle instead of a button.
- The inference system should be able to support files in stereo, not just mono.

Reflection on End User Feedback

Having received this feedback I set about fixing these problems.

- The mode switch should be a toggle instead of a button.

This UI/UX problem whilst simple in nature was slightly more difficult to fix than expected due to the lack of a toggle widget in python, so I had to implement a custom toggle widget. The updated code looks as follows:

```
class ToggleButton(QWidget):
    clicked = QtCore.pyqtSignal(bool)

    def __init__(self, text) -> None:
        super().__init__()
        self.current_val = False
        self.setFixedHeight(30)
        self.setFixedWidth(100)

    def mousePressEvent(self, event: QtGui.QMouseEvent):
        if event.button() is Qt.MouseButton.LeftButton:
            self.current_val = not self.current_val
            self.clicked.emit(self.current_val)
            self.update()

    def paintEvent(self, e):
        painter = QtGui.QPainter(self)
        brush = QtGui.QBrush()
        width = painter.device().width()
        height = painter.device().height() - 2
        painter.drawRoundedRect(0, 0, width, height, height/2, height/2)

        brush.setColor(QtGui.QColor("blue")) # configure brush for rendering of circle
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        painter.setBrush(brush)
        if self.current_val:
            painter.drawEllipse(5, 5, height - 10, height - 10)
```

```

else:
    painter.drawEllipse(width - (height - 10) - 5, 5, height - 10, height - 10)

class EncodeDecodeContainer(QFrame):
    def __init__(self, main_window):
        super().__init__()

        self.label = QLabel("Mode: ")
        self.switch = ToggleButton()
        self.switch.clicked.connect(self.switch_ec)
        l1 = QHBoxLayout()
        l1.addWidget(self.label)
        l1.addWidget(self.switch)

        self.encode = EncodeWidget(main_window)
        self.decode = DecodeWidget(main_window)

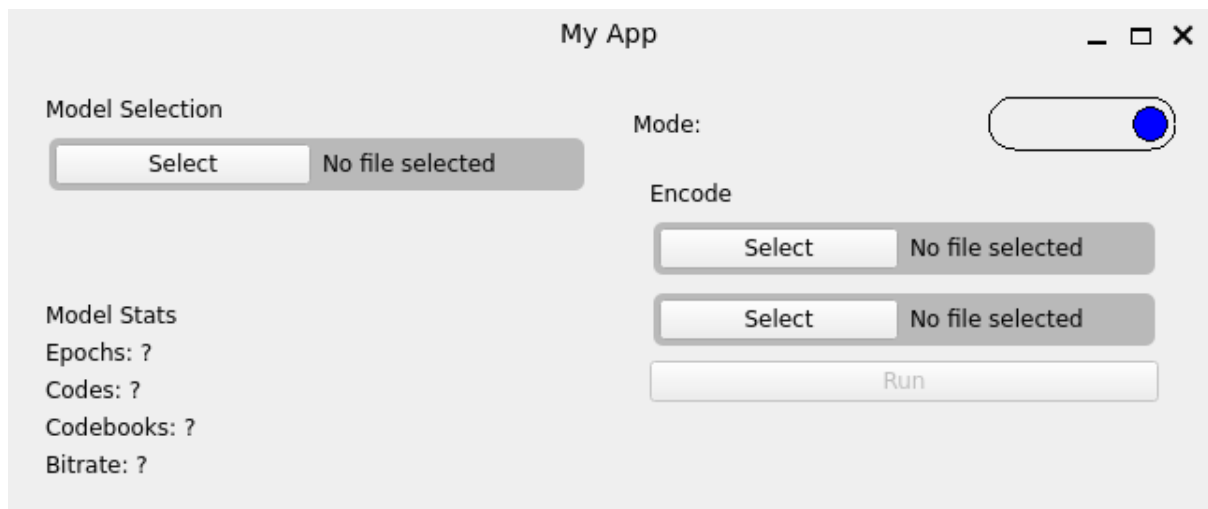
        layout = QVBoxLayout()
        layout.addWidget(self.switch, alignment=QtCore.Qt.AlignmentFlag.AlignHCenter)
        layout.addWidget(self.encode, alignment=QtCore.Qt.AlignmentFlag.AlignTop)
        layout.addWidget(self.decode, alignment=QtCore.Qt.AlignmentFlag.AlignTop)

        self.decode.setVisible(False)

        self.setLayout(layout)

```

The GUI now looks like:



- The inference system should be able to support files in stereo, not just mono.

This is something I overlooked in my initial objectives, as whilst some audio is in mono, most professionally produced audio, including podcasts, is recorded in stereo, although generally has the same data going to each track. To fix this I've added a step to the inference pipeline where before converting the audio to 16 kHz, the audio is first converted from n channels to mono by taking an average across each channel at each time interval. Below is the additional code needed to do this:

```
if sound.shape[0] != 1:  
    sound = (sound.sum(axis=0) / 2).unsqueeze(0)
```

Final Feedback

Having made these changes I asked my user again for feedback, he said:

The GUI now looks a lot better and the application works well.

Looking to the future

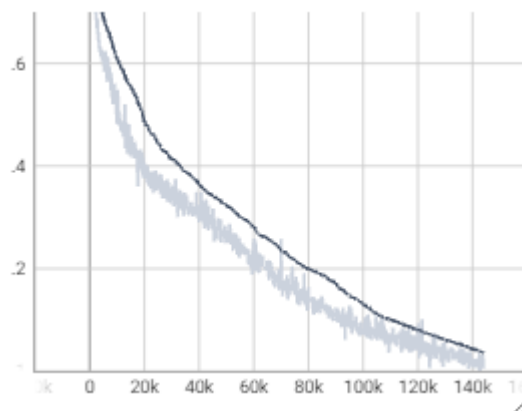
Whilst I am pleased with my program and that the solution is sufficient for the end users needs, there are many upgrades which I think can be made to the system.

GUI

The current GUI is very functional, it does what it needs to do with very little extra. One thing to do in the future is improve the looks of this GUI and add new features such as using a "pre built" model without having to select a file. This would simplify its use by a user. To also allow the encoding of all files in a directory would significantly speed up the process of switching over to this system.

The Model

The model, whilst understandable, has clear room for improvement. A significant portion of this improvement can come from using the same training program and model structure, just training the model for longer. The evidence for this can be found by looking at the loss graphs. Below is a graph of whisper loss over time for part of the low quality model training:



Looking at this graph there continues to be a significant downward trend even when reaching the end of this training run. This suggests that more training will continue to improve performance significantly, something that would be something to do in the future. This could also be helped by investigating some of the bottlenecks during training and improving the performance of, for example the discriminators, which at the moment significantly slow training.

Improved hyperparameter tuning would also help these new models as it would mean that they reach optimal performance with less training. Throughout this project I haven't had time to do proper tests involving changing the learning rate of both the discriminator and the main model as these tests take multiple days each. Better tuning of the learning rate could significantly reduce the time taken to train, and therefore improve the quality of the overall model.

What else could this be used for

Compressing audio in this form is a useful task, however ideas developed in this project can be used for other (possibly more interesting) tasks. It has been demonstrated that the Encodec codec can be used with a transformer model for high fidelity audio generation, this project would likely be able to decrease the training times, and increase the fidelity of the final product.