Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Adam Elhadi Gafar
20/09/2023

An investigation into Neural Style Transfer and its applications for creative expression in digital media

Project supervisor: Adam Prügel-Bennett
Second examiner: Geoff Merrett

A project report submitted for the award of
Computer Science with Artificial Intelligence (MEng)

Abstract

Neural Style Transfer approaches comprise methods by which digital media, typically images or video, are transformed using the "style" of reference media. The target media is generated through deep-learning based techniques which embed the style of the reference but keep the target's constituent features and structure recognisably similar – allowing for such examples as real-world photographs rendered in the signature style of a given artist, say.

With recent advances in neural-network derived speech processing, coupled with novel approaches to audio generation, the following project intends to explore the application of Neural Style Transfer techniques to generate samples of audio in which speech may be synthesised in the style of a given speaker – i.e. speech would be generated in some target speaker's voice in which they would appear to "imitate" the speech and mannerisms in recitation of the speaker's script in a reference audio sample.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

Acknowledgements

I would like to thank my project supervisors Jonathon Hare (original) and Adam Prügel-Bennett (current) for their guidance, little as it could be due to the complications in producing this project.

Table of Contents

Introduction

Problem

With NST having been explored in literature thoroughly with synthesising images in the visual domain, and with results being easily appreciable due to texture (style) application to input images, the problem arises of how to apply this to audio. Concretely, in reproducing reference speech with different identifiable speakers, the applicability of NST is dependent on the viability of feature extraction and the evaluation of textural application in reproduction within the audio domain. This would involve reframing audio data in a visual analogue – such that style can be analysed and applied to the synthesis of new data, ensuring preservation of the constituent content.

Goals

Firstly, a reversible method of processing the data to be operated upon in a machine learning context must be appraised and utilised throughout the NST procedure. With the data transformed in such a way, a convolutional model must be designed and implemented with an architecture containing layers that can sufficiently represent feature embeddings corresponding to style and content. A labelled dataset of speech recordings would thus be required to train this model.

With the creation of such a model as a prerequisite, the NST approach would utilise its feature representations to perform the synthesis of new data; appropriate deviations from the typical visual-domain implementation would be required to ensure the expected outcome is of the expected form (the reproduction of content reference audio in the "voice" of the speaker of the style reference audio). Data generated would need to be reprocessed from the machine learning-usable format back to listenable waveforms conforming to the parameters of, at least, those of the pre-import content reference.

Scope

Whilst the initial brief broadly proposed the investigation of NST techniques to multiple media domains (such as video), the scope has since been restricted to audio, and further to just speech – as opposed to other forms such as music. This is due to the prerequisite of the feature-extraction model implementation, which would involve impractically complex architectures and training to be able to encompass the multitude of types of audio media in existence. With this goal of creating, designing, and training a neural network for the required embeddings, the development of such a model with the computational resources available is likely to take days if not weeks – making exceeding this level of complexity an intractable proposition given the scope of this project.

Furthermore, though additions and alterations to the NST algorithm will be necessary, advanced techniques for generating audio such as using *Generative Adversarial Networks* and the like would also likely be outside the scope of what would be realistic given computational and project time constraints.

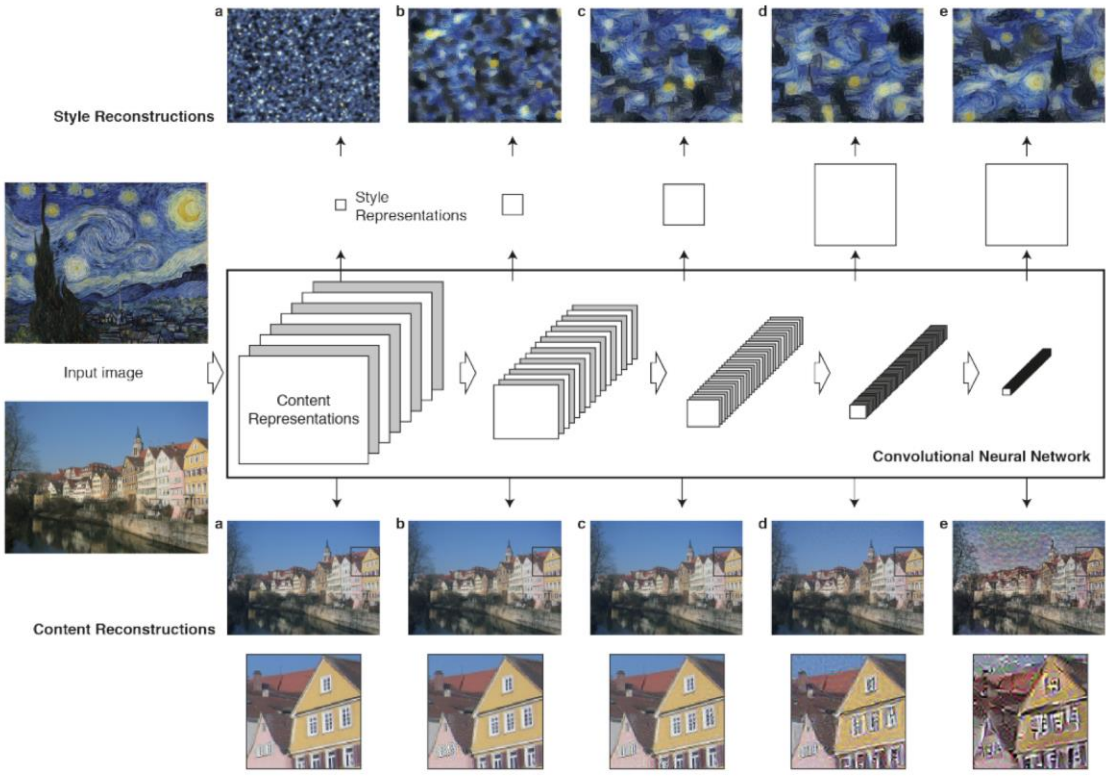
Literature Review

Neural Style Transfer

The codification of Neural Style Transfer techniques can be attributed to the seminal paper released by *Gatys et al.* (1). The first paper of its type, the authors sought to leverage the rise of advanced *Deep Neural Networks* (DNNs) in the image processing field for a new application: attempting the synthesis of images, with the visually-perceived artistic style of one applied to the conserved content of the other – this was achieved with *"high perceptual quality"* (p. 1).

Concretely, through the consideration of the convolutional layers within object-recognition DNNs to be extracting "features", representations of content and style can be extracted. Through reconstructions of source images through each convolutional layers' feature maps, the authors were able to demonstrate (Fig 1) that later layers in a network represented content well (the visual objects or scenery of an image); correlations between filter responses at each layer captured the "style" (p. 2) – this could also be considered the extraction of a textural representation (2).

Figure 1: Gatys et al. (1 p. 3); style and content extraction (representation via reconstruction) from feature layer responses in a CNN



The disentanglement (relative, as mutual similarities are impossible to eliminate (1 p. 6)) of the style and content representation allows for them to form the basis of two respective metrics – a loss function to minimise these can then be calculated and used to synthesise a new image emphasising either of the aforementioned metrics for the desired visual appeal (p. 6).

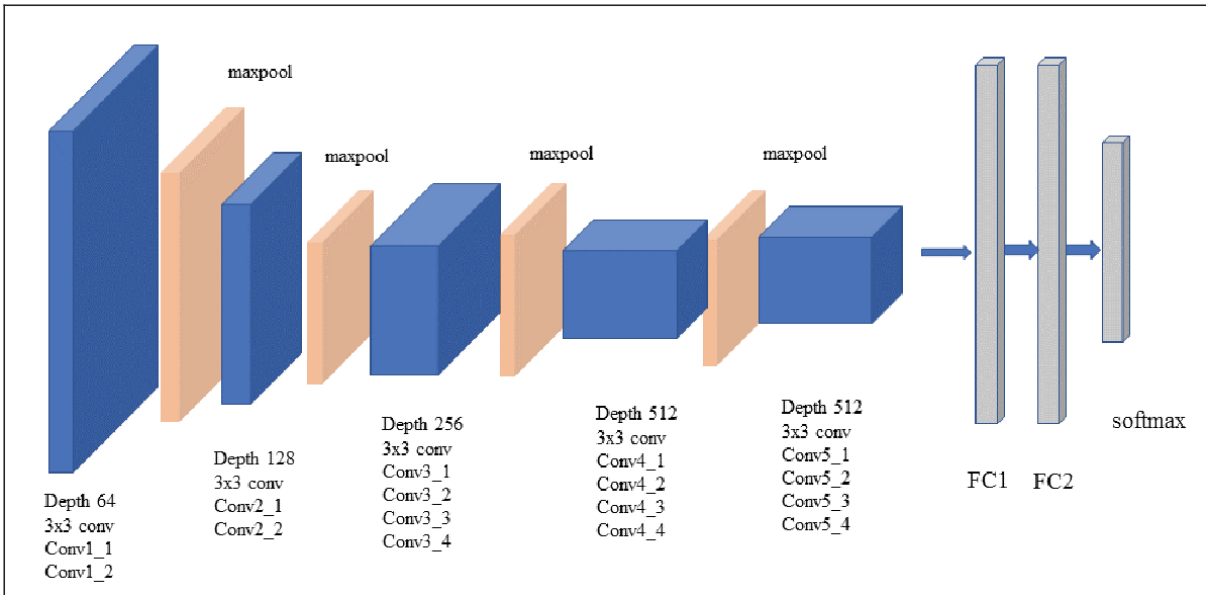
To generate the novel image, a white-noise image is transformed using backpropagation until feature responses at the desired convolutional layers are sufficiently matched between the reference and synthesised image (p. 10). For content loss, a squared-error loss function of the difference in activations of desired filters between the two images is used (p. 10). For minimising style (alternatively "texture") loss, the *Gram matrices* of the activations in the desired layers for the two images – most simply, a feature matrix multiplied by its transpose – have their mean-squared distance minimised. It is of note that in literature the reasons for the Gram matrix giving a textural representation were not well understood at the time, with *Yanghao et al.* proposing theoretical equivalence to *Maximum Mean Discrepancy* as an explanation in 2017 (3).

The total loss function is thus defined (constants α and β define the ratio between content and style loss emphasis, and the symbols $\vec{p}$, $\vec{a}$, and $\vec{x}$ represent the content reference, style reference, and generated images respectively) (1 p. 12):

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

In implementation, the approaches of *Gatys et al.* and *Luan et al.* in the similar paper *Deep Photo Style Transfer* (4) used a selection of certain convolutional layers from the pretrained *VGG-19* image classification network (5) (Fig 2): `conv4_2` for the content representation; `conv1_1`, `conv2_1`, `conv3_1`, `conv4_1` and `conv5_1` for the style representation. (4 p. 4)

Figure 2: VGG-19 Architecture, source: https://towardsdatascience.com/extract-features-visualize-filters-and-feature-maps-in-vgg16-and-vgg19-cnn-models-d2da6333edd0



Design Strategy

The following concerns outline the considerations for designing the ultimately realised approach:

Data and feature extraction

A suitable method for transforming audio wave files into a format on which machine learning methods can be executed is required; such that the degree to which the problem can be modelled similarly to an image-based NST problem can be evaluated. The most likely representation would be spectrographic, such as by performing a Short-Time Fourier Transform (STFT) on the data.

Given the processed data, the embeddings for content and style would need to be extracted. Within the visual domain, pretrained classifier convolutional layers are usually sufficient to provide these, but there is no guarantee of an audio classifier (particularly when less complex than these models when self-implemented) performing to the same standard. It is possible that separate architectures would be required to generate appropriate representations of audio style and content.

Audio synthesis and style transfer

With the goal of generating novel audio that minimises the loss of the style and reference targets, the method of synthesis may not necessarily be the same as those outlined in traditional NST approaches. Concretely, whilst residual noise from initialising the output randomly might be a visual non-issue, output audio from such a method might introduce deleterious audio artifacts if not accounted for and minimised. It may thus be preferable to initialise using copies of either of the target waveforms instead, depending on experimental results.

For the execution of the style transfer algorithm, further experimentation will be required to arrive at the appropriate model layers and weightings to use for style and content, and the gradient descent may need to be adjusted (e.g. the optimiser used, further audio processing at each step) to ensure convergence and maximise the quality of results.

Model design

A Convolutional Neural Network is the most likely architecture to be used – the design will need to account for the way in which input will be processed; the appropriate tensor sizes and kernels for the output of each layer; the layer arrangement; and any fully connected output layers to classification required for training. At minimum, the model will need to be able to classify the voices of the input dataset for its embeddings to represent and content style as required; a need for the network (or a separate one) to phonetically or lexicographically process audio may arise, depending on experimental results, for the content representation.

Evaluation, testing, and training

For the classification of speaker identity as outlined in the model design, a labelled dataset of input audio and respective speaker labels would be necessary. Evaluation of

training to this end would be reasonably trivial – simply minimise loss of label classification and maximise accuracy on a split training set. If phonetic or lexical analysis of the data is required, the dataset would need to also include textual information (e.g. transcripts) corresponding to each waveform.

This approach is logical as the classifier would evaluate whether output audio would come from the target speaker (to train its relevant style embeddings), though whether this is sufficient to also generate content embeddings may need to be answered by further modelling based on whether output audio would contain the same phonetic content as input.

Evaluation of the output of the Neural Style Algorithm itself is somewhat difficult, as whilst the style and content loss values are trivial to quantify, the ultimate output quality is ultimately subjective to the listener (without advanced techniques to evaluate audio that would likely cause scope creep).

Implementation

The ultimate design arrived at through experimentation and different implementation approaches (see next chapter) is outlined below:

Tools, libraries, and frameworks

Due to pre-existing familiarity and competency in the context of machine learning, the language chosen for the project was *Python 3*, with all libraries and dependencies existing within a package environment managed by the *Anaconda* distribution. For audio processing and analysis, the *librosa* package (6) was used; for machine learning functionality, the *PyTorch* (7) library was chosen due to its relative ease of use and CUDA/graphics functionality; *Matplotlib* (8) provided the graphical tools for visualising output; and *NumPy* was required for array and matrix manipulation and passing data to and from the listed libraries.

Voice classifier

Dataset

Ultimately, it was decided that the model used would classify the identity of different speakers within a dataset – to this end, the *Centre for Speech Technology Voice Cloning Toolkit* (VCTK) (9) corpus was selected to provide the classifier's training data. This is a labelled dataset of 43873 entries (108 speakers reading around 400 sentences, mostly selected from newspaper text); each item contains the waveform (.flac format), the audio sample rate (48000Hz), a textual transcript, a speaker identifier string, and a sentence identifier string.

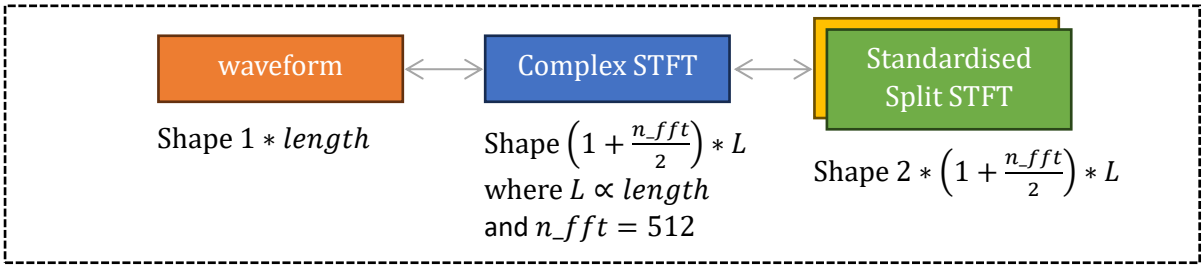
To prepare the data for the model, a modification of the dataset that reduced each item to the audio and its speaker identifier was created. Rather than a waveform, the audio is represented by a split float representation of the real and imaginary parts of the complex values returned by passing it through the *librosa* STFT function. The matrix shape of the data is thus $\mathbf{2} * \mathbf{257} * \boldsymbol{L}$, where $\boldsymbol{L}$ is proportional to the length of the original waveform and the **257** features correspond to the value (`n_ftt`) of **512** – chosen due to suitability for speech – as the signal length for the windows on which the discrete Fourier transforms are performed. Iterating across this dataset to calculate the

standard deviation and mean for each of the features (each shaped $\mathbf{2} * \mathbf{257}$), the data was then rescaled to standardised values where:

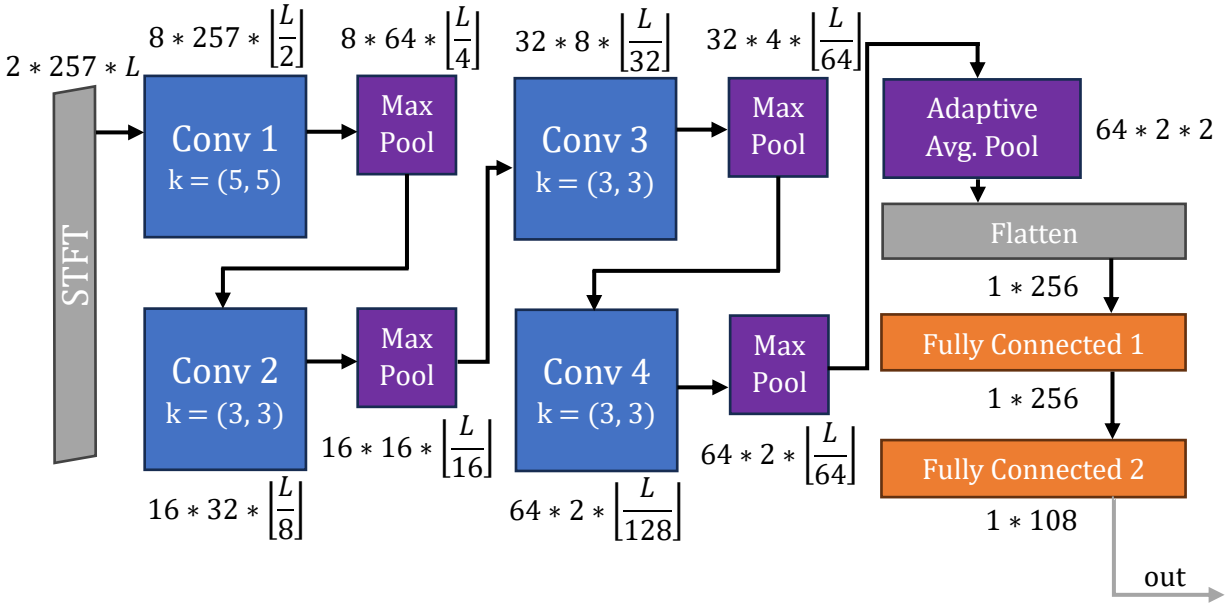
$$x_s = \frac{x - \mu_{feature}}{\sigma_{feature}}$$

Further iteration across the dataset was necessary to recreate the labels in training – with the number of unique speakers calculated as 108, the labels were added to a dictionary from which incrementing one-hot label vectors were created.

The new dataset was saved as binary representations of each item (represented as a list) to a new directory, ready to be used in training.



Network architecture



The network first passes the STFT data through four convolutional layers; the first with a $\mathbf{5} * \mathbf{5}$, `stride = 2` kernel and channel sizes $(\mathbf{2}, \mathbf{8})$ – the 2 input channels treating the real and imaginary parts of the input akin to 3 channels for RGB values in an image classifier; and the next three with a $\mathbf{3} * \mathbf{3}$, `stride = 2` kernel and channel sizes $(\boldsymbol{p}, \mathbf{2}\boldsymbol{p})$, where p is the number of output channels of the previous layer, thus ending in **64** output channels. Each layer is max-pooled with a $\mathbf{2} * \mathbf{2}$, `stride = 2` kernel.

As the output of the convolutional layers is of variable length due to the differing audio file durations, an adaptive average pool (kernel: $\mathbf{2} * \mathbf{2}$, `stride = 2`) reduces the output to a fixed-size feature matrix, which is then flattened to give **256** averaged features.

The features are finally passed to two fully connected layers of sizes $\mathbf{256 * 256}$ and $\mathbf{256 * 108}$, with the output layer producing a SoftMax-activated vector of probabilities for each of the labels. All other layers are activated with ReLU, allowing the weights of all layers to be initialised randomly using *He/Kaiming* initialisation (10).

The structure of the network as described is illustrated in the diagram at the beginning of this section, and the following module printout. Note the diagram's modules are annotated with the shape of their **output** tensor.

```
VCTKClassifier(
  (activation): ReLU()
  (conv1): Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2),
padding=(1, 1))
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
  (conv3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
  (conv4): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  (adaptive_pooling): AdaptiveAvgPool2d(output_size=(2, 2))
  (conv_layers): Sequential(
    (0): Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2),
padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
    (3): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
    (6): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
    (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=256, out_features=256,
bias=True)
  (fc2): Linear(in_features=256, out_features=108,
bias=True)
)
```

Style transfer

Loss calculation

The style loss is calculated through calculating the mean squared error loss of the Gram matrices of the input and reference (target) feature maps; the Gram matrix is calculated through first reshaping the feature vectors into a matrix of size ($\boldsymbol{bf} * \boldsymbol{mn}$) where b is the batch size, f is the number of feature maps, and $\boldsymbol{m} * \boldsymbol{n}$ is the shape of each feature map. This feature matrix is then multiplied by its transpose, and finally normalised through dividing by the total number of elements of the feature matrix ($\boldsymbol{bfmn}$).

The content loss, much more simply, is simply the mean square error loss between input and target features.

Total loss is thus calculated with the following formula:

$$\mathcal{L}_{total}(s_c, s_s, s_i) = \alpha\mathcal{L}_{content}(s_c, s_i) + \beta\mathcal{L}_{style}(s_s, s_i)$$

Where s_c is the content STFT, s_s is the style STFT, and s_i is the input STFT. α and β are weighting constants, which are $\mathbf{1}$ and $\mathbf{1000}$ respectively.

Audio transformations

To process the data, multiple functions had to be designed to transform the audio:

- `to_stft()` and `get_complex_stft()`
  The first of these transforms an input waveform into a STFT, then splits the complex values into a **2 * 257 * L** matrix of floats representing the separated real and imaginary parts of its data. The second function recombines this representation into a valid complex STFT for non-ML audio processing functions.
- `normalise_stft()` and `un_normalise_stft()`
  Due to training on a standardised training set, these functions apply and reverse the standardisation function to any input STFTs (using the custom VCTK-based dataset's mean and standard deviation).

Model generation

The classifier model is imported with its weights trained on the VCTK dataset, and both loss functions are implemented as *PyTorch* modules. Thus, a new model is generated with these modules attached to the appropriate convolutional layers of the model, with the loss calculation in their forward function, and accessible through two lists (the loss is derived using the output of the model for target and input features calculated up to the point of their insertion). The chosen convolutional layers to attach content and style loss modules to were the third and fourth, and first and second, respectively, as reflected in the following printout of the generated model's modules:

```
Sequential(
  (conv1): Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2),
padding=(1, 1))
  (sty_loss1): StyleLoss()
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
  (sty_loss2): StyleLoss()
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  (conv3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
  (con_loss3): ContentLoss()
  (relu3): ReLU()
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  (conv4): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
  (con_loss4): ContentLoss()
  (relu4): ReLU()
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
)
```

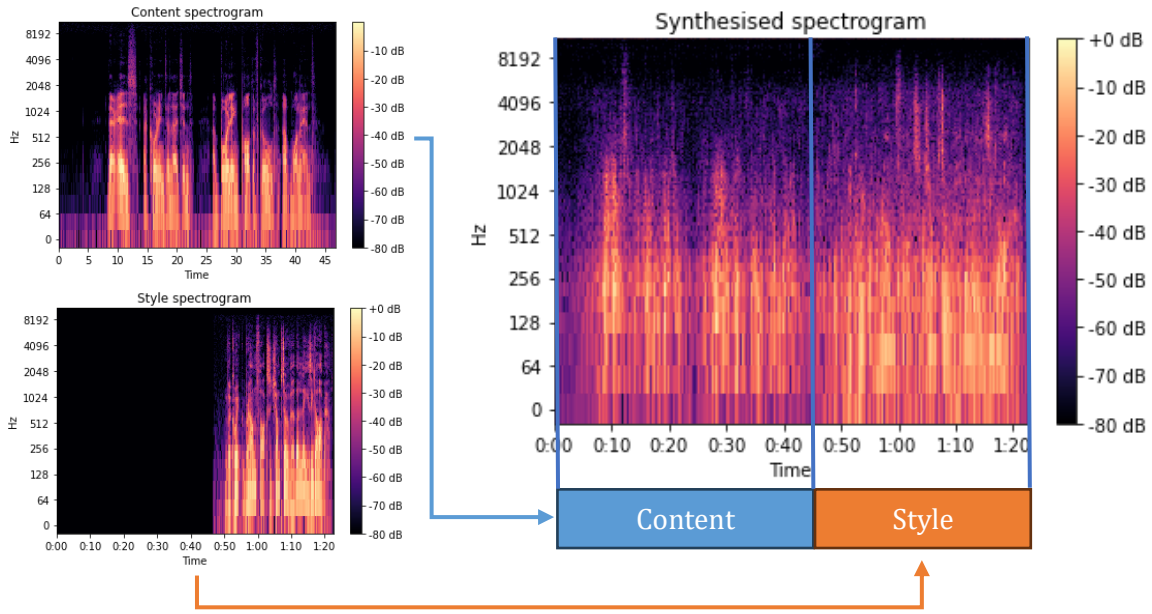
NST algorithm implementation

After loading the style and content audio waveforms, the former is then frameshifted by the latter's length with a prefix of padded 0s in its array and then copied to initialise the output audio – this reduces the influence of the style audio's content by placing it further ahead in the resultant waveform.

The three waveforms are then transformed to VCTK-standardised STFTs, with the *Limited-memory Boyden-Fletcher-Goldfarb-Shanno* algorithm (LBFGS) used as the optimiser to perform gradient descent on the output STFT to minimise the total loss function (using $content\ weighting = \mathbf{1}$ and $style\ weighting = \mathbf{1000}$) over a certain number of steps ($[\mathbf{5000}\,..\,\mathbf{10000}]$).

Due to the zero-padding applied to the style audio, which the output is a copy of, the algorithm thus "generates" a style-textured representation of the content audio in the gap within the padded prefix – avoiding overlap with the style audio's content. When the resultant STFT is transformed back to a waveform, the output is trimmed to the length

of the content to thus leave only the desired style-transferred content that was generated. This phenomenon is illustrated in the following diagram:



Experimentation and Results

All machine-learning operations were performed on a single CUDA-enabled device (*NVIDIA GeForce* RTX 3080). In the following chapter, much of the performed experimentation and changes in the implementation of the design are discussed, with corresponding data and results shown.

Initial classifier

Dataset and training

Whilst VCTK was always the dataset used to train the classifier, the form the dataset took changed much until its final custom design. Initially, the dataset was left as-is, with the STFTs generated for each item accessed during training. However, this resulted in excessively long training times, so the STFTs were instead calculated across the entire dataset with the new items saved to a directory – whilst this took several hours, subsequent training epochs using these items instead were much swifter. This dataset was then split randomly into a separate training and testing set with the ratio **80 : 20**.

Using the *Stochastic Gradient Descent* optimiser and the *Cross Entropy Loss* function, training then began to minimise the loss between model output and the matrices of each batch's one-hot label matrix. The batch size chosen was **32** as it was the highest power of 2 size that would not result in a crash due to memory requirements. However, no matter the learning rate chosen, there was no convergence. Even after increasing the model architecture (see next section), convergence was slow – only one epoch over several hours – and accuracy achieved was low (**~13%**).

Table 2: SGD Optimiser

```
# running training
torch.cuda.empty_cache()
train(train_loader, model, loss_fn, o
```

```
loss: 4.414498  [   32/35098]
loss: 4.210673  [ 3232/35098]
loss: 3.713645  [ 6432/35098]
loss: 3.921886  [ 9632/35098]
loss: 3.908557  [12832/35098]
loss: 3.534714  [16032/35098]
loss: 3.482207  [19232/35098]
loss: 3.430902  [22432/35098]
loss: 3.795104  [25632/35098]
loss: 3.562178  [28832/35098]
loss: 3.274305  [32032/35098]
```

```
# eval with test
test(test_loader, model, loss_fn)
```

```
275
Test Error:
 Accuracy: 13.6%, Avg loss: 3.380477
```

Table 2: Adam Optimiser

```
# running training
torch.cuda.empty_cache()
train(train_loader, model, loss_fn, o
```

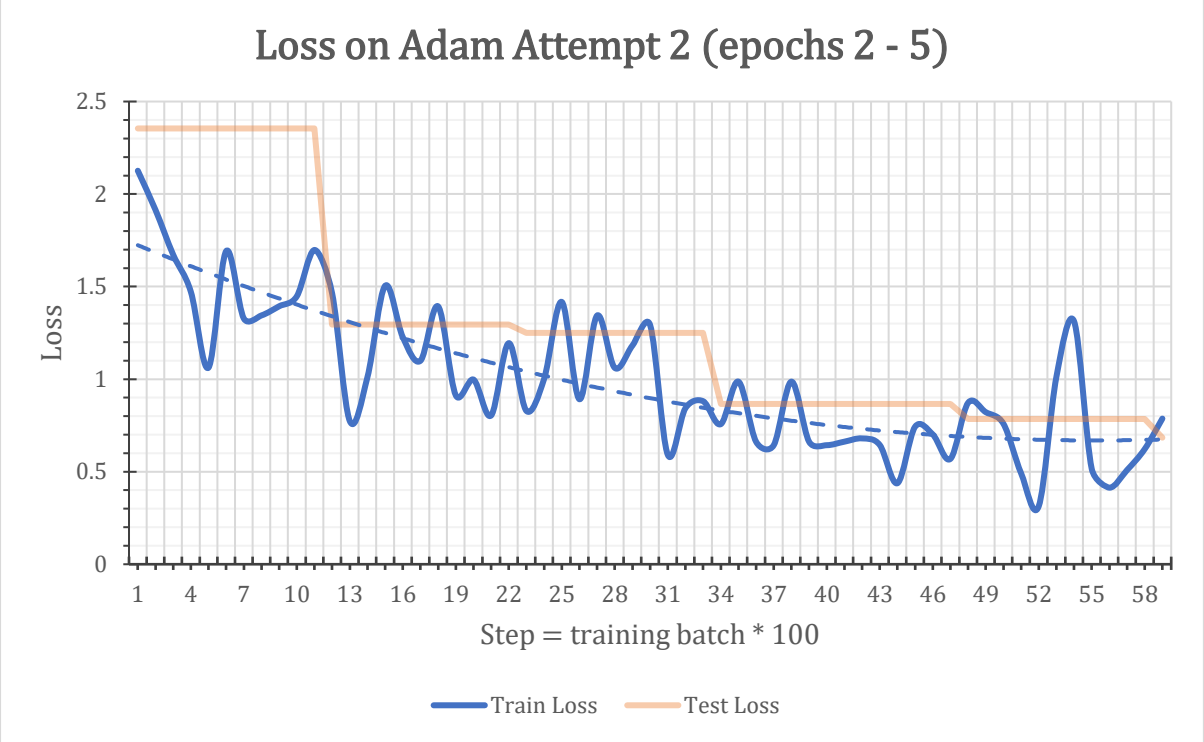
```
loss: 7.612947  [   32/35098]
loss: 4.355151  [ 3232/35098]
loss: 3.938634  [ 6432/35098]
loss: 3.956719  [ 9632/35098]
loss: 3.506454  [12832/35098]
loss: 3.311431  [16032/35098]
loss: 2.777250  [19232/35098]
loss: 2.792869  [22432/35098]
loss: 3.234783  [25632/35098]
loss: 2.657797  [28832/35098]
loss: 2.126100  [32032/35098]
```

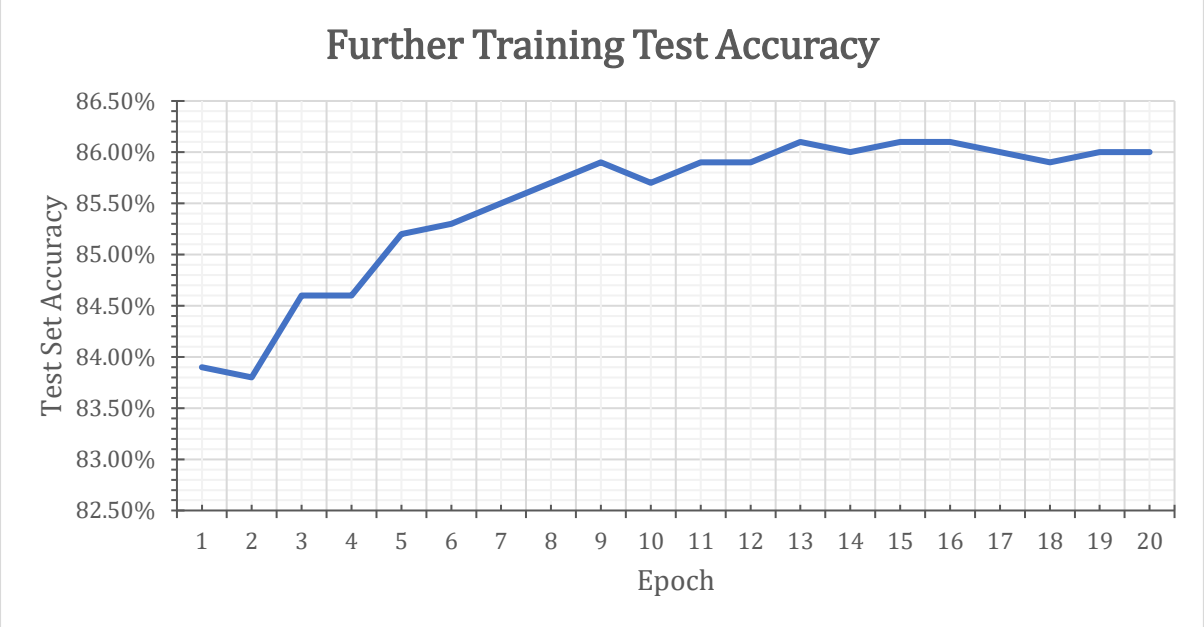
```
# eval with test
test(test_loader, model, loss_fn)
```

```
275
Test Error:
 Accuracy: 32.5%, Avg loss: 2.354607
```

This was addressed through three changes that came about through repeated experimentation over many days: first, the optimiser was changed to the Adam algorithm (11). Though noisily convergent, the speed and accuracy of this compared with SGD was larger – particularly with He/Kaiming weight initialisation, the second

change – this pushed accuracy to ~**30**% over the epoch. Finally, the realisation was made that the training set would have to be standardised – the process of doing so took days, but resulted in significant improvements to both speed and convergence (to ~**60%** accuracy over one epoch). Thus, training went ahead with up to **5** epochs; with different learning rates applied until the best performance for these 5 was seen ($\boldsymbol{lr} \approx [\mathbf{0.001}, \mathbf{0.005}]$), a further **20** were run to try and maximise test accuracy with the model.





Model architecture

Initially, the model had only four layers. The first two were convolutional layers, with kernel sizes of ($\mathbf{3} * \mathbf{1}$) and channel sizes ($\mathbf{1}, \mathbf{32}$) and ($\mathbf{32}, \mathbf{64}$) respectively, and their

outputs max-pooled with kernel ($\mathbf{2} * \mathbf{2}$, `stride = 2`). Due to the differing audio input durations, input STFTs also had varying lengths. Thus, each batch during training was sorted by size and then zero-padded (trailing) to all match the length of the first item. To give the same number of features across all batches for the classification to work, the output of the second convolutional layer was then adaptive-average pooled with kernel ($\mathbf{3} * \mathbf{3}$). The following two fully connected layers had dimensions ($\mathbf{576} * \mathbf{256}$) and ($\mathbf{256} * \mathbf{108}$). Leaky ReLU was initially the activation function for each layer (except the SoftMax output layer), but eventually ReLU was chosen due to slightly faster convergence. Unfortunately, this model was responsible for the poor first training attempt mentioned in the previous section – the model complexity was too low to achieve any meaningful convergence (test set accuracy was less than $\mathbf{5}\%$). The module

```
VoiceCNN(
  (activation): ReLU()
  (conv1): Conv2d(1, 32, kernel_size=(3, 1), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 1), stride=(1, 1),
padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  (adaptive_pooling): AdaptiveAvgPool2d(output_size=(3, 3))
  (fc1): Linear(in_features=576, out_features=256,
bias=True)
  (fc2): Linear(in_features=256, out_features=108,
bias=True)
)
```

print readout is shown below:

The model complexity was increased for the final iteration of the architecture, for which considerations were made for a balance between increased number of features against the computational power required to train them. For a more detailed description of the final model architecture, refer to the previous chapter – the model architecture for this classifier differed only in that the first convolutional layer had one channel (refer to next section for why this changed).

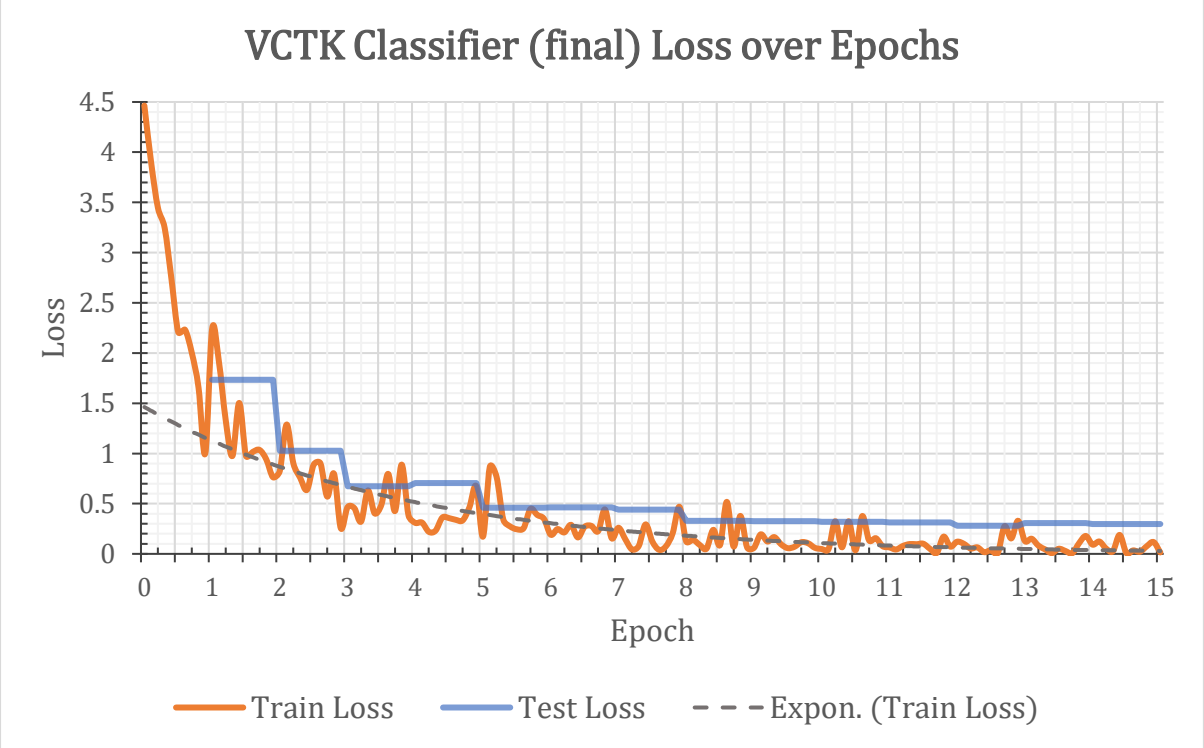
Final classifier

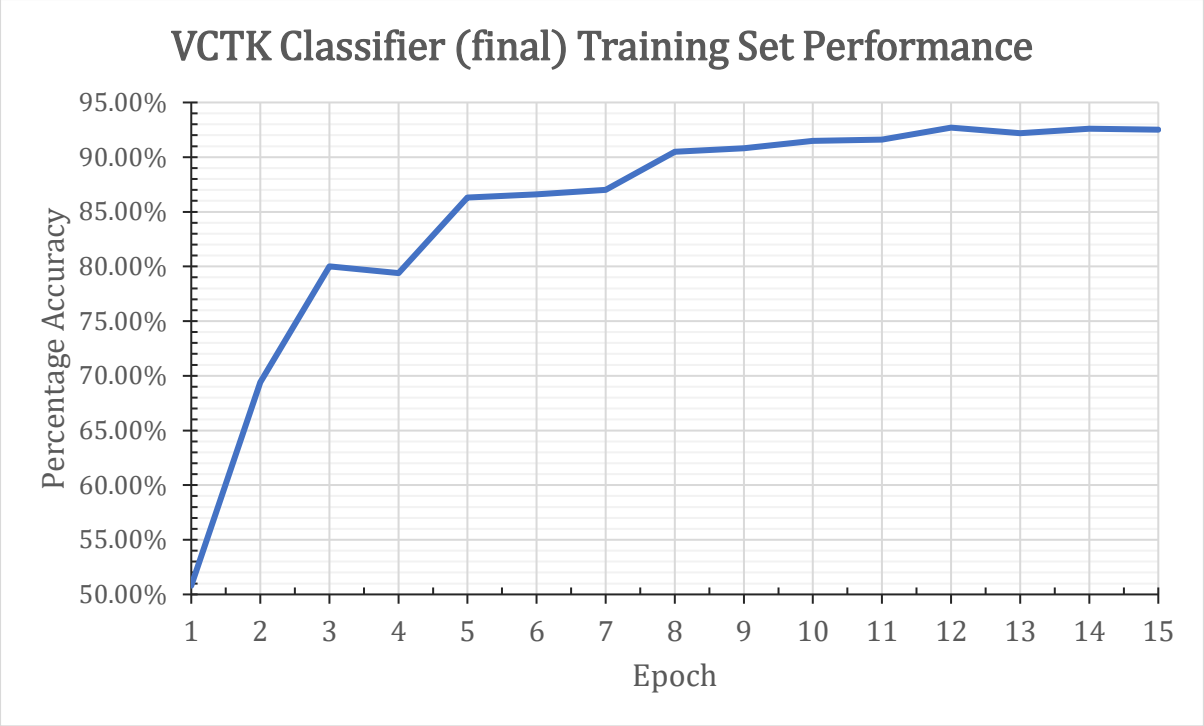
With the classifier seemingly finished, implementation of the Neural Style Transfer algorithm (see next section) began, but then at its gradient descent stage it was realised the functions to process each STFT data's complex values were non-revertible – an "STFT" was generated, but there was no way to return this to a waveform without inserting random phase data to reconstruct a complex-valued STFT. This was due to a misunderstanding of the ease to which gradient descent could be performed on the waveform directly using gradients calculated from their transformed data during the process of NST – it was much more practical to operate on the STFTs directly.

With no other choice to progress, training had to restart from the beginning. The model architecture itself was barely changed from its previous iteration – the solution to create

an invertible STFT representation was to simply split the real and imaginary parts into two channels and adjust the first convolutional layer accordingly (though this significantly increased training time to several days to complete about 20 epochs). Other than the time spent training, a large amount of effort had to be expended to recreate the custom VCTK dataset using this split, two-channel representation: accessing and converting each item to STFTs had a similarly long duration to creating the initial custom data set: the means and standard variances had to be recalculated for the real and imaginary split data separately; the standardised STFT representations were then saved to a new directory.

The training strategy and data processing stages were largely the same for this classifier. However, thanks to the more accurate STFT representation, convergence was faster, with a higher test set accuracy achieved over the same number of epochs. This is a relative comparison, as due to the doubling of the size of each training example and the number of channels in the first convolutional layer, each epoch took more than twice as long to complete. Additional changes introduced were a learning rate scheduler ($\mathbf{0.8} * \boldsymbol{lr}$ after each epoch; $\boldsymbol{lr_{init}} = \mathbf{0.005}$) to allow for a higher initial learning rate to speed convergence, and weight decay regularisation ($\mathbf{1} * \mathbf{10^{-6}}$). The graphs below show the first $\mathbf{15}$ epochs of training this classifier; though $\mathbf{10}$ more were performed, the change in loss was minimal and this likely caused overfit – the model weights at each epoch were saved to allow for avoidance of overfit models and experimentation during NST implementation.





Style transfer

After the final classifier was obtained, different permutations of the NST implementation were attempted. With the Gram matrix function written to accept both possible input dimensions, numerous layers of the model were used as the content and style layers. Many arrangements of the convolutional layer assignments were attempted, with some experiments even including the fully-connected layers. Eventually, the best results were found to be obtained using `[conv1, conv2]` for the style layers and `[conv3, conv4]` for content.

The reference audio used to generate novel STFTs via NST were either randomly chosen examples from the VCTK set, self-recorded clips, or other clips found online (little experimental difference was found from using clips from within the training set and without). All feature short phrases, words, or sentences selected as auditorily distinct enough to produce more easily recognised output. These audio files can be found in the archive under the directory `*/Code/audioinpex/`. It was chosen, at first, to initialise the output audio as copies of the content reference.

Initial results were of poor quality. To improve the output, several strategies were employed. Firstly, the number of steps for the algorithm was increased (typical values were in the magnitudes of $\mathbf{10}, \mathbf{000/100}, \mathbf{000/1}, \mathbf{000}, \mathbf{000}$). Attempts were also made with other optimisers than LBFGS, such as Adam – whilst this optimiser was faster, it would noisily converge (if it did at all). Changes to the style/content weights would have noticeable differences, but most were lateral and would not meaningfully help with achieving the desired output (to sound like the content audio "spoken by" the style reference speaker).

Results gained by this stage were mostly of slightly garbled replications of the content audio. Instead, the audio was then initialised from random values. Unfortunately, the

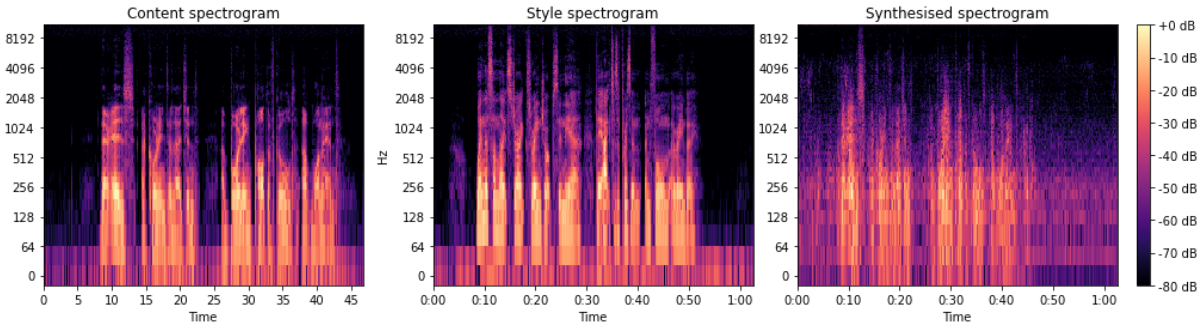
auditory noise generated by these attempts was overwhelming, with the gradient descent unable to override it even given a generous number of steps. With clamping of the values at each stage, the results were mostly quiet, slightly coloured brown noise with no recognisable speech.

Finally, attempts were made to initialise from the style audio – unintuitively, this gave the best results. It was noted that outputs from this stage in experimentation sounded like voices speaking the same sentences "overlaid" on top of one another. This is when the idea to zero-pad the style audio emerged – if the content could be generated in the zero-padded area of the output (itself a copy of this style audio, meaning overlap from the style audio's content would be localised to outside this region), it would be texturally regenerated using the learned style, if the weighting was of an appropriate value. This method bore fruit: with the style content trimmed after the content generation, the signal that was previously heard "overlaid" was isolated. Whilst audibly not of the highest quality (the goal of having the output naturally sound like the style speaker was unfortunately not attained), a certain similarity of the voices generated to the style was clear – regenerating outputs using the same content and different styles showed this to be true. Using this method, $\mathbf{5,000}$ to $\mathbf{10,000}$ steps with a weight of $\mathbf{1000}$ for style and $\mathbf{1}$ for content were found to be the best parameters.

Using a simple print STFT method taken from the *librosa* website (12), the experimental results from different stages are illustrated below. The directory to access saved results is `*\Code\results\` in the archive; *Microsoft Excel* spreadsheets are also included with loss values and intermediary spectrograms.

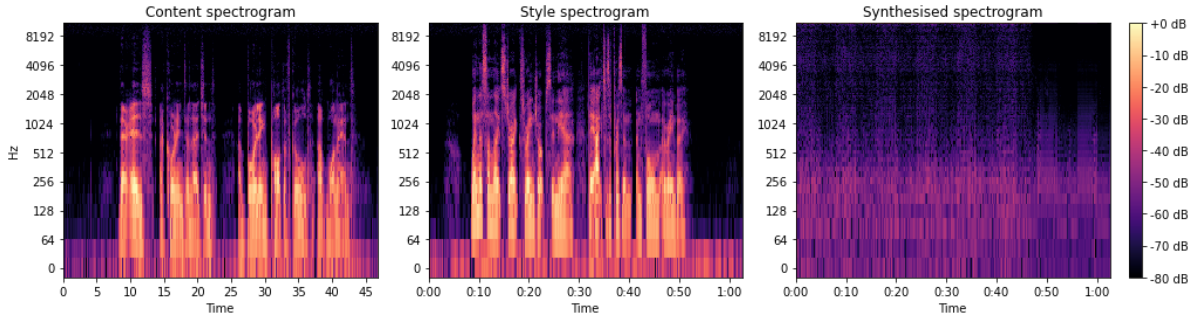
Early attempt 1

- *Parameters:*
  - Output initialised from content; using LGBFS optimiser; content layers `[conv4]`, style layers `[conv1, conv2, conv3]`; $\mathbf{100,000}$ steps; content/style weight $\mathbf{0 : 1 * 10^9}$.
  - Content and style filenames, respective: `p304_009_mic2.flac | p225_006_mic2.flac`
- *Results:*
  - *Description:*
    Reproduction of content with "style"-coloured content overlaid.
  - Output filename:
    `cw0_sw1e9_cc4_sc1c2c3_LBFGS_s100k_content.wav`
  - Spectrograms of content, style, and output audios respectively:



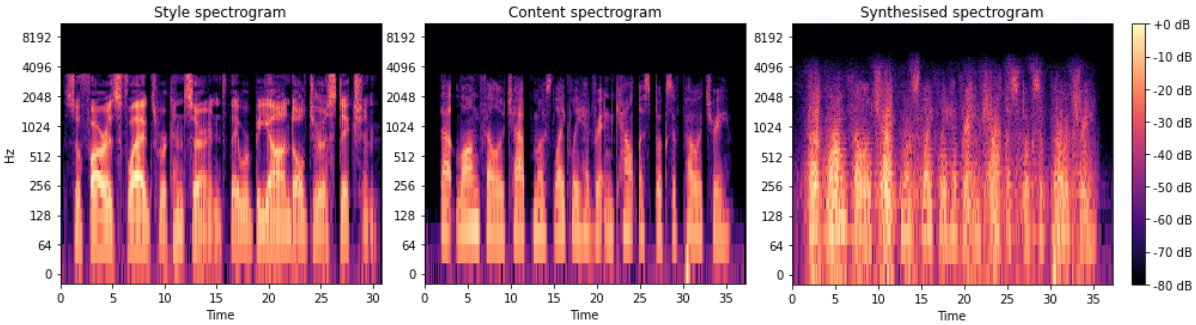
Early attempt 2

- *Parameters:*
  - Output initialised randomly; using LGBFS optimiser; content layers `[conv1]`, style layers `[conv1, conv2, conv3, conv4]`; $\mathbf{50,000}$ steps; content/style weight $\mathbf{100 : 1 * 10^7}$.
  - Content and style filenames, respective:
    `p304_009_mic2.flac | p225_006_mic2.flac`
- *Results:*
  - *Description:*
    Loud noise with slight hints of speech.
  - Output filename:
    `cw1e2_sw1e7_cc1_sc1c2c3c4_LBFGS_s50k_rand.wav`
  - Spectrograms of content, style, and output audios respectively:



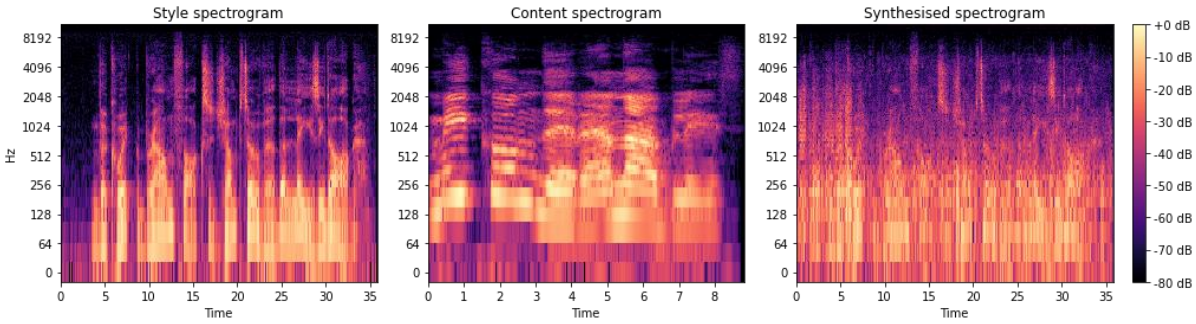
Early attempt 3

- *Parameters:*
  - Output initialised from style; using LGBFS optimiser; content layers `[conv1]`, style layers `[conv1, conv2, conv3]`; $\mathbf{10,000}$ steps; content/style weight $\mathbf{1 : 1}$.
  - Content and style filenames, respective:
    `p225_006_mic2.flac | p304_009_mic2.flac`
- *Results:*
  - *Description:*
    Reproduction of content with noisy "style"-coloured content overlaid.
  - Output filename:
    `cw1_sw1_cc1_sc1c2c3_LBFGS_s10k_style.wav`
  - Spectrograms of content, style, and output audios respectively:

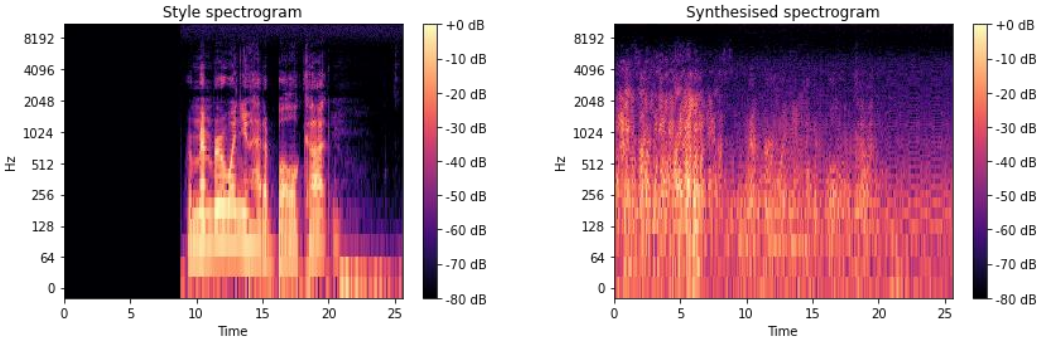


Later attempts

- *Parameters:*
  - Output initialised from style; using LGBFS optimiser; content layers `[conv3, conv4]`, style layers `[conv1, conv2]`; **10**, **000** steps; content/style weight **1 : 1000**.
    Note: these parameters are final; all subsequent attempts are the same.
  - Content and style filenames, respective:
    `human_female_greetings_08.wav | fox.wav`
- *Results:*
  - *Description:*
    Style-textured reproduction of content, with style content overlay.
  - Output filename:
    `cw1_sw1e3_sc1c2_cc3c4_LBFGS_s10k_style.wav`
  - Spectrograms of content, style, and output audios respectively:

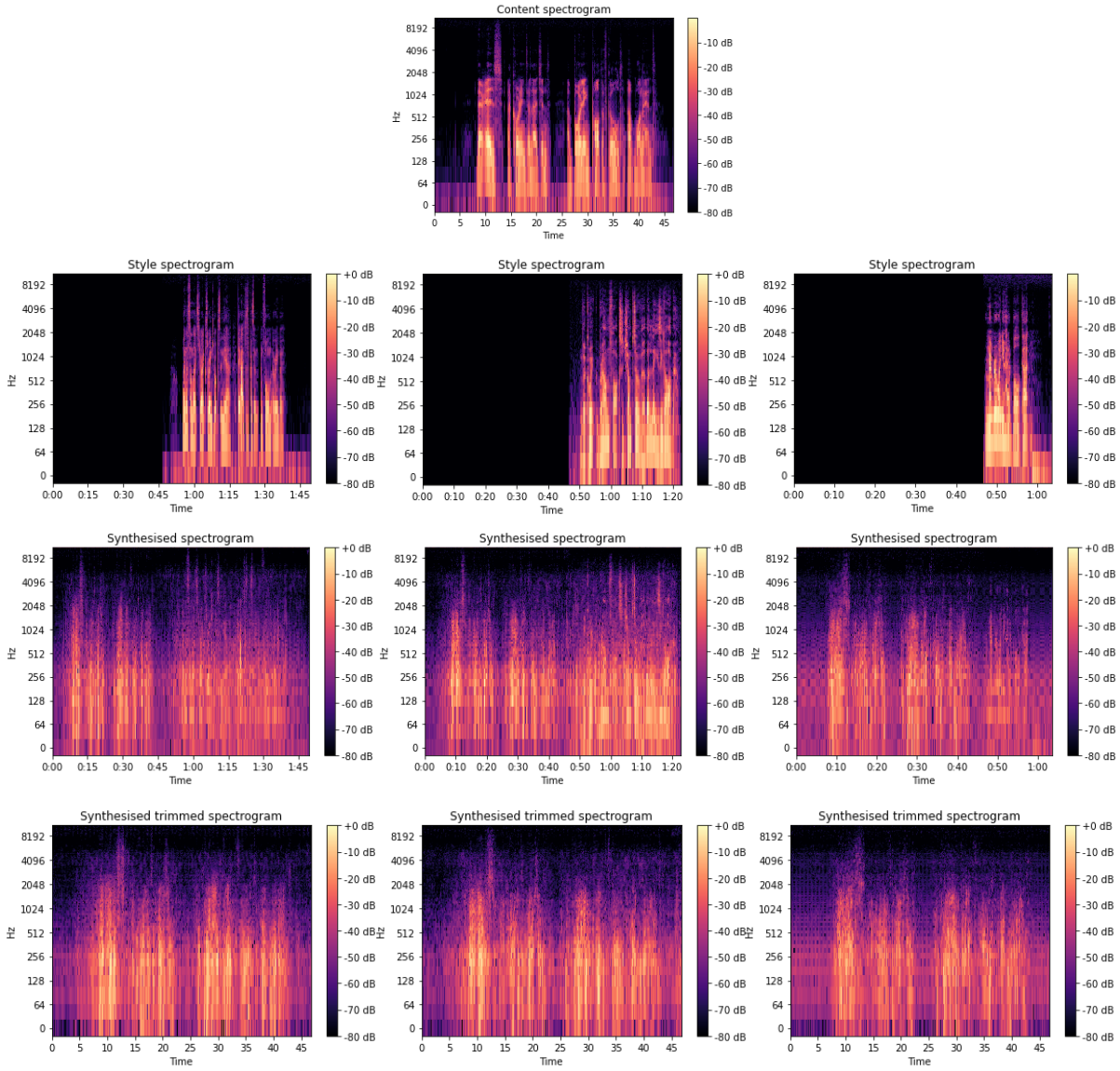


  - This is where the zero-pad frameshift idea was applied, the frameshifted version of this result avoids the style content overlay and is recorded at filename:
    `fs_cw1_sw1e3_cc1c2_sc3c4_LBFGS_s10k_style.wav`
  - Using a different style audio (`sample.wav`), a result was produced with a different "voice", proving the efficacy of the NST algorithm in producing the same content with different styles.
    - Output filename:
      `cw1_sw1e3_sc1c2_cc3c4_diff_style.wav`
    - Spectrograms of style, and output audios respectively (as content is the same); the zero-pad can be observed in style:



With the algorithm parameters finalised and good results achieved, the synthesised audio was finally trimmed to the content length (as there was trailing style audio content). The following spectrograms show the same content (`p304_009_mic2.flac`) but with different styles, with the results saved as

`304009_[style_name].wav`. The order of each row is: content, style, output, output trimmed.



Further results can be found under `*\Code\results\further\`, with names typically similar to `content_[content_name]_style_[style_name].wav`.

Progress and Project Management

The following GANTT chart details progress toward the project's completion.

		Pre	July				August					September		
Task No.	**Tasks**	**N/A**	1-9	10-16	17-23	24-31	1-6	7-13	14-20	21-31	1-10	11-17	18-24	25-30
1	**Project Brief**													
2	**Interim Report**													
3	**Literature Review and Research**													
4	**Design and Approach Plan**													
5	**Classifier**													
5.1	Initial Implementation													
5.2	Recreating Dataset													
5.3	Final Implementation													
6	**Neural Style Transfer**													
6.1	Initial Implementation													
6.2	Final Implementation													
6.3	Experimentation and Tuning													
6.4	Final Project Report													

Key:		
		Major disruption or no progress
		Minor disruption or invalid progress
		Good progress

Due to health issues and retaking Year 3, meaningful project progress began around June – July of this year. The minor disruption progress bars relate to prioritising module referral exams around the time, as well as medical flare-ups. These bars can also mean invalid – as described before, major issues with the classifier required the retraining of the model after the recreation of its dataset. The major disruptions cited were a large health issue that precluded meaningful progress until it was resolved.

Due to most progress taking place outside of the semester, the project was entirely developed without outside help or university resources, further delaying progress – many errors made with implementation and time spent learning how to do so could have been reduced with assistance within the university (such as supervisor meetings),

which was sorely lacked. Indeed, the training of the models themselves also took long due to using a single Personal Computer rather than the university's computational resources.

Whilst extensions were provided due to the aforementioned health issues and having to retake Third Year, this project is likely to be submitted slightly after the last extension given (particularly due to having to reimplement much of it).

Conclusions, Evaluation, Further Work, and Ideas

In conclusion, it is clear that Neural Style Transfer techniques are applicable to the audio domain, at least as far as speech is concerned. Through a voice classifier, the ability to generate content audio references with different styles, and thus in the voice of the style reference speaker, was achieved. However, the quality of the output audio was not to the standard of audible voice replication, with noise frequencies throughout and the voices only sounding slightly human, let alone resembling that of the style reference. In principle, this goal is likely achievable with a more complex classifier that encodes more detailed speech embeddings – due to the constraint of training the classifier from scratch, reaching this level of performance during this project appears to have been impractical.

Further ideas for improving performance would include elimination of non-audible frequencies (such as through using Mel-scale spectrograms instead of a naïve STFT spectrogram), or other more advanced audio-processing techniques. With regards to the chosen implementation, further performance could have been achieved through tuning of hyperparameters for the classifier, and the layers and weights for content and style during NST, using branch and bound or some other combinatorial optimisation strategy (another approach that was untenable due to the prohibitive time required to compute and implement).

With regards to evaluating the efficacy of NST in achieving life-like voices, perhaps a deep pretrained classifier, or speech recognition models for content loss, could have been used to see if more advanced speech embeddings would lend themselves to producing higher quality output given the same NST implementation. Other methods for audio synthesis could also be compared to NST, such as using *Generative Adversarial Networks*.

The classifier architecture itself may have had insufficient features as well: the decision to allow for variable length input required inputs to be pooled to an average length after the convolutional layers; had a set pad or trim length been specified this would be unnecessary and the fully connected layers would have worked directly with the output from the last convolutional layer – perhaps leading to more consistent and detailed feature embeddings. Usage of a larger dataset (or multiple different sets) and augmenting the dataset with noise could also have led the classifier to be more suited to voice feature extraction.

With regard to the NST algorithm itself, this may have been tweaked with more audio processing techniques to remove noise and smooth the spectrograms during gradient descent, though the prerequisite expertise in this domain was lacking to implement this.

References

1. *Image Style Transfer Using Convolutional Neural Networks.* **Gatys, L. A., A.S., Ecker and M., Bethge.** Las Vegas, NV, USA : IEEE Computer Society, 2016. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) . pp. 2414-2423.

2. *A Neural Algorithm of Artistic Style.* **Gatys, Leon A, Ecker, Alexander S and Bethge, Matthias.** s.l. : arXiv/cs.CV, 2015, Vol. 1508.06576.

3. *Demystifying Neural Style Transfer.* **Li, Yanghao, et al.** 2017, arXiv/cs.CV, Vol. 1701.01036.

4. *Deep Photo Style Transfer.* **Luan, Fujun, et al.** 2017, arXiv/cs.CV, Vol. 1703.07511.

5. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* **Simonyan, Karen and Zisserman, Andrew.** s.l. : arXiv/cs.CV, 2015, Vol. 1409.1556.

6. **McFee, Brian.** librosa/librosa: 0.10.1. *zenodo.* [Online] 2023. https://zenodo.org/record/8252662.

7. **Paszke, Adam, et al.** PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32.* s.l. : Curran Associates, Inc., 2019.

8. *Matplotlib: A 2D graphics environment.* **Hunter, J. D.** 3, s.l. : IEEE COMPUTER SOC, 2007, Computing in Science & Engineering, Vol. 9, pp. 90-95. doi:10.1109/MCSE.2007.55.

9. **Yamagishi, Junichi, Veaux, Christophe and MacDonald, Kirsten.** CSTR VCTK Corpus: English Multi-speaker Corpus for CSTR Voice Cloning Toolkit (version 0.92). *Edinburgh Datashare.* [Online] University of Edinburgh. The Centre for Speech Technology Research (CSTR), 13 November 2019. https://datashare.ed.ac.uk/handle/10283/3443.

10. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* **He, Kaiming, et al.** s.l. : arXiv/cs.CV, 2015, Vol. 1502.01852.

11. *Adam: A Method for Stochastic Optimization.* **P. Kingma, Diederik and Ba, Jimmy.** 2017 : arXiv/cs.LG, Vol. 1412.6980.

12. **librosa.stft.** ***librosa.*** **[Online] https://librosa.org/doc/main/generated/librosa.stft.html.**

Appendix A: Original Project Brief

An investigation into **Neural Style Transfer** and its applications for creative expression in digital media

Student Adam Gafar aeg3g18@soton.ac.uk
Supervisor Dr Jonathon Hare jsh2@ecs.soton.ac.uk

Project Briefing

Problems

Neural Style Transfer has arisen contemporarily as a means of altering articles of media in such a way that they are made to adopt the "style" of those supplied as reference. [1] Such a feat is made possible using deep learning algorithms for recognising objects (visually) and structure – allowing content to be preserved throughout their transformation.

Style Transfer has numerous creative and analytical applications, with extensive research in the field of images allowing for the promulgation of utilities for stylising artwork and photos. Research has continued into other forms of media, such as video and more recently audio (with such examples as changing instruments or genres of music).

Goals

The goal of this project is to conduct an investigation into the creative capabilities of Neural Style Transfer, beginning with the use and evaluation of current tools, libraries, and models to test their applications on digital media.

Armed with this understanding, the ultimate goal would be to train and build the architecture of a new model (or models), or otherwise produce applications with the intent to achieve style transfer within specific domains and toward specific purposes (e.g. voice/instrument transfer in audio).

Scope

This project is likely to use common machine learning libraries and APIs such as *TensorFlow*, *CUDA*, *Keras* etc. [2] Pretrained models for Style Transfer and more generally those for *Generative Adversarial Networks* (GAN) will also be used and analysed to eventually fuel attempts to train and architect new style transfer/generative models using convolutional neural networks and other deep learning methods.

The complexity, scale, and number of attempts in generating aforementioned models is limited by the projects' time constraints and personal unfamiliarity with the field; it is likely that even these will be trained using publicly available datasets. Thus, the scope of the project in the first semester of this year will be limited to dealing strictly with visual data (i.e. images), with the potential aim to branch into other areas such as video and audio within the second semester.

References

1. Gatys, L. A., Ecker, A. S. & Bethge, M., 2015. A Neural Algorithm of Artistic Style. *arXiv CoRR*, Volume abs/1508.06576.
2. TensorFlow, 2018. *Neural Style Transfer: Creating Art with Deep Learning using tf.keras and eager execution.* [Online] Available at: https://medium.com/tensorflow/neural-style-transfer-creating-art-with-deep-learning-using-tf-keras-and-eager-execution-7d541ac31398 [Accessed 16/10/2020]

Appendix B: Code

Note: imports, printing functions, sanity checks, and some outputs are removed; refer to corresponding files in archive for full code. These code extracts are taken from the corresponding files under `*/Code/` in the archive.

reversible stft dataset creation.ipynb

The following code was used to generate the custom VCTK-based dataset.

```
# convert audio from VCTK to STFT and stack real/imaginary values to 2 len
array
def stft_vctk(example):
    #512 is the recommended length for windowed signal (speech)
    sample_rate = example[1]
    audio = np.squeeze(example[0].numpy())
    audio_stft = librosa.stft(audio, n_fft = 512)

    # processing
    real = np.real(audio_stft)
    imag = np.imag(audio_stft)
    audio_stft_sep = np.stack([real, imag])

    return audio_stft_sep, sample_rate

# subclass that implements STFT conversion when getting items
class STFT_Dataset(datasets.VCTK_092):
    def __init__(self, path):
        super().__init__(root = path)

    def _load_audio(self, file_path):
        s, sr = stft_vctk(torchaudio.load(file_path))
        return (s, sr)

    def __len__(self):
        return super().__len__()
    def __getitem__(self, idx):
        # return wav file, label of speaker
        return super().__getitem__(idx)[0], super().__getitem__(idx)[3]

# initialise dataset
stft_data = STFT_Dataset("VCTK")

# get means
feature_mean_sums = np.zeros((stft_data.__getitem__(0)[0].shape[0],
stft_data.__getitem__(0)[0].shape[1]))

for x in range(stft_data.__len__()):
    example = stft_data.__getitem__(x)[0]
    feature_mean_sums = np.add(feature_mean_sums, np.mean(example, axis = 2))

#get means, save to file
dataset_feature_mean = feature_mean_sums / stft_data.__len__()
np.save("mean_comp", dataset_feature_mean)
```

```
# get variances
feature_variance_sums = np.zeros((stft_data.__getitem__(0)[0].shape[0],
stft_data.__getitem__(0)[0].shape[1]))

# collect squared sums of (data - mean) across dataset
for x in range(stft_data.__len__()):
    example = stft_data.__getitem__(x)[0]
    example = example - dataset_feature_mean[:, :, np.newaxis]
    sq_sum = np.mean(example ** 2, axis = 2)
    feature_variance_sums = np.add(feature_variance_sums, sq_sum)

# get standard deviation, save to file
std = np.sqrt(feature_variance_sums / stft_data.__len__())
np.save("std_comp", std)

# normalise then use pickle to save to folder
def normalise(stft, mean, std):
    std = np.where(std == 0, 1, std)
    stft = (stft - mean[:, :, np.newaxis]) / std[:, :, np.newaxis]
    return stft

# write to folder
for x in range(stft_data.__len__()):
    item = stft_data.__getitem__(x)
    item_norm_stft = normalise(item[0], dataset_feature_mean, std)
    item_label = item[1]

    with open("STFTcomp/stftc" + str(x), "wb") as file:
        pickle.dump([item_norm_stft, item_label], file)
```

VCTK classifier complex.ipynb

The following code was used to generate and train the final voice classifier model.

```
# custom VCTK-based dataset definition
class STFTC_Dataset(Dataset):
    def __init__(self, root: str):
        self.root_path = root + "/"

    def __len__(self):
        # hard code as length will not change and to prevent needing to access
original dataset
        return 43873

    # load items from directory
    def __getitem__(self, idx):
        with open(self.root_path + "stftc" + str(idx), "rb") as file:
            pickled_list = pickle.load(file)

        target_input, target_label = pickled_list
        return torch.from_numpy(target_input).float(), target_label

# initialise dataset
stftc_data = STFTC_Dataset("STFTcomp")
```

```
# split to test/train
split_ratio = 0.8
train_size = int(split_ratio * stftc_data.__len__())
test_size = stftc_data.__len__() - train_size

#device_gen = torch.Generator(device = device)
train_dataset, test_dataset = random_split(stftc_data, [train_size,
test_size])

"""

dataset parameters

*VCTK Structure:*
(0: waveform; 1: sample rate; 2: text transcript; 3: person identifier; 4:
text identifier)

*STFT VCTK structure:*
(0: waveform; 1: person identifier)

"""

# generate label dictionary from original VCTK dataset
label_temp_vctk = datasets.VCTK_092(root = "VCTK")

label_loader = iter(DataLoader(label_temp_vctk, shuffle = False))
dict_label = 0
label_dict = {}
while True:
   try:
       item = next(label_loader)
       if item[3] not in label_dict.keys():
           label_dict[item[3]] = dict_label
           dict_label = dict_label + 1
   except StopIteration:
       break

# using the above code to generate, hard code the label dictionary once done:
label_dict = {...}


"""
Defining model
"""
# model to classify between the different speakers
# we will treat the STFT matrix as if it is a 2d image: remember the columns
are the audio signal windows with frequency components

class VoiceCNN(nn.Module):
    def __init__(self):
        super(VoiceCNN, self).__init__()

        #activation
        self.activation = nn.ReLU()

        #CONV LAYERS
```

```
        #interpret complex numbers as two channels
        self.conv1 = nn.Conv2d(in_channels = 2, out_channels = 8, kernel_size
= (5, 5), stride = 2, padding = 1)
        nn.init.kaiming_normal_(self.conv1.weight, mode = 'fan_in',
nonlinearity = 'relu') # initialise weights

        self.conv2 = nn.Conv2d(in_channels = 8, out_channels = 16, kernel_size
= (3, 3), stride = 2, padding = 1)
        nn.init.kaiming_normal_(self.conv2.weight, mode = 'fan_in',
nonlinearity = 'relu')

        self.conv3 = nn.Conv2d(in_channels = 16, out_channels = 32,
kernel_size = (3, 3), stride = 2, padding = 1)
        nn.init.kaiming_normal_(self.conv3.weight, mode = 'fan_in',
nonlinearity = 'relu')

        self.conv4 = nn.Conv2d(in_channels = 32, out_channels = 64,
kernel_size = (3, 3), stride = 2, padding = 1)
        nn.init.kaiming_normal_(self.conv4.weight, mode = 'fan_in',
nonlinearity = 'relu')

        #POOLING

        self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)

        pool_h, pool_w = 2, 2
        self.adaptive_pooling = nn.AdaptiveAvgPool2d((pool_h, pool_w))

        # define conv block
        self.conv_layers = nn.Sequential(
            self.conv1,
            self.activation,
            self.pool,
            self.conv2,
            self.activation,
            self.pool,
            self.conv3,
            self.activation,
            self.pool,
            self.conv4,
            self.activation,
            self.pool
        )

        #FULLY CONNECTED LAYERS

        self.fc1 = nn.Linear(pool_h * pool_w * self.conv4.out_channels, 256)
        nn.init.kaiming_normal_(self.fc1.weight, mode = 'fan_in', nonlinearity
= 'relu')
        self.fc2 = nn.Linear(256, 108)
        nn.init.kaiming_normal_(self.fc2.weight, mode = 'fan_in', nonlinearity
= 'relu')

    def forward(self, x):
        # through conv layers
```

```
        out = self.conv_layers(x)

        # adaptive pooling to standardise feature map size
        out = self.adaptive_pooling(out)

        # flatten feature maps
        out = out.view(out.size(0), -1)

        # through connected layers
        out = self.activation(self.fc1(out))
        out = self.fc2(out)
        return out
"""
Loading data
"""
# hyperparam
set_batch_size = 32

# custom collate function as input STFTs are of differing lengths
def collate_pad(batch):
    # sort by length in descending order
    batch.sort(key = lambda x: x[0].shape[2], reverse = True)
    max_length = batch[0][0].shape[2]

    # pad all inputs to same length
    padded_X = []
    for item in batch:
        tensor, label = item
        pad_size = max_length - tensor.shape[2]
        padded = F.pad(tensor, (0, pad_size))
        padded_X.append(padded)

    # return padded X in a stack
    X = torch.stack(padded_X)

    # convert labels to one-hot stack
    dict_labels = torch.tensor([label_dict[(item[1],)] for item in batch])
    y = F.one_hot(dict_labels, len(label_dict)).float()

    return X, y

# dataloaders
train_loader = DataLoader(train_dataset, batch_size = set_batch_size, shuffle
= True, collate_fn = collate_pad)
test_loader = DataLoader(test_dataset, batch_size = set_batch_size, shuffle =
False, collate_fn = collate_pad)

"""
Training model
"""
# def training func
def train(dataloader, model, loss_fn, optimiser):
    length = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
```

```
        # pred error
        hx = model(X)
        loss = loss_fn(hx, y)

        # backprop
        loss.backward()
        optimiser.step()
        optimiser.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{length:>5d}]")

# def test func
def test(dataloader, model, loss_fn):
    length = len(dataloader.dataset)
    batch_length = len(dataloader)
    model.eval()

    test_loss, total_correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)

            hx = model(X)
            test_loss += loss_fn(hx, y).item()

            total_correct += (hx.argmax(1) ==
y.argmax(1)).type(torch.float).sum().item()
        test_loss /= batch_length
        total_correct /= length
        print(f"Test Error: \n Accuracy: {(100 * total_correct):>0.1f}%, Avg
loss: {test_loss:>8f} \n")

"""
Run training
"""
# hyperparams
learning_rate = 0.002
weight_decay = 1e-6
epochs = 10

# loss and optimiser
loss_fn = nn.CrossEntropyLoss()
optimiser = torch.optim.Adam(model.parameters(), lr = learning_rate,
weight_decay = weight_decay)

scheduler = torch.optim.lr_scheduler.ExponentialLR(optimiser, gamma = 0.8)

for epoch in range(epochs):
    print("\nEpoch " + str(epoch) +":")
    train(train_loader, model, loss_fn, optimiser)
    scheduler.step()
    test(test_loader, model, loss_fn)
```

```
    torch.save(model.state_dict(), "comp_model_params" + "_ep_" + str(epoch) +
".pth")
```

style transfer complex.ipynb

The following code was used to implement and run style transfer on audio files.

```
# class definition
class VCTKClassifier(nn.Module):
    [...]
# initialise model instance, load pretrained weights
VCTKClassifier_model = VCTKClassifier().to(device)
VCTKClassifier_model.load_state_dict(torch.load("comp_model_params_ep_9.pth"))
VCTKClassifier_model.eval()

## style transfer loss functions

### style loss
def gram_matrix(input_featuremaps):
    if input_featuremaps.dim() == 4:
        batch_size, n_featuremaps, f_m, f_n = input_featuremaps.size()

        # stack flattened feature vectors into a matrix
        feature_vecs = input_featuremaps.view(batch_size * n_featuremaps, f_m
* f_n)

    elif input_featuremaps.dim() == 2:
        feature_vecs = input_featuremaps
        batch_size = input_featuremaps.size()[0] / 2
        n_featuremaps = 2
        f_m = input_featuremaps.size()[1]
        f_n = 1
        #print(input_featuremaps.size())
        #print(batch_size * n_featuremaps * f_m * f_n)

    # multiply feature vec matrix by its transpose for gram matrix
    G_mat = torch.mm(feature_vecs, feature_vecs.t())
    #print(G_mat.size())

    # normalise matrix by total number of elements in feature maps
    G_mat = torch.div(G_mat, batch_size * n_featuremaps * f_m * f_n)

    return G_mat

# style loss module
class StyleLoss(nn.Module):
    def __init__(self, style_featuremaps):
        super(StyleLoss, self).__init__()

        # detach() to remove from backprop tree (remove from dynamic gradient
computation)
        self.style_G_mat = gram_matrix(style_featuremaps).detach()

    def forward(self, x):
        x_G_mat = gram_matrix(x)
```

```
        self.loss = F.mse_loss(x_G_mat, self.style_G_mat)
        return x

### content loss
# content loss module
class ContentLoss(nn.Module):
    def __init__(self, content):
        super(ContentLoss, self).__init__()

        # detach() to remove from backprop tree (remove from dynamic gradient
computation)
        self.content = content.detach()

    def forward(self, x):
        self.loss = F.mse_loss(x, self.content)
        return x

## running style transfer
### audio processing functions
# load waveform array
def load_audio(filepath):
    audio, sr = librosa.load(filepath, sr = 48000)
    audio = np.squeeze(audio)
    return audio

# standardise an STFT with given mean/std
def normalise_stft(stft, mean, std):
    std = np.where(std == 0, 1, std)
    stft = (stft - mean[:, :, np.newaxis]) / std[:, :, np.newaxis]
    return stft

# return from standardised STFT to un-standardised data
def un_normalise_stft(stft, mean, std):
    std = np.where(std == 0, 1, std)
    stft = (stft * std[:, :, np.newaxis]) + mean[:, :, np.newaxis]
    return stft

# return from split representation to 2D complex-valued matrix
def get_complex_stft(stft):
    real = stft[0]
    comp = stft[1]
    return stft[0] + 1j * stft[1]

# convert waveform to STFT; split complex values into stack of len 2
def to_stft(audio):
    #512 is the recommended length for windowed signal (speech)
    audio_stft = librosa.stft(audio, n_fft = 512)

    # processing
    real = np.real(audio_stft)
    imag = np.imag(audio_stft)
    audio_stft_sep = np.stack([real, imag])

    return audio_stft_sep
```

```
# function taken from source:
https://librosa.org/doc/main/generated/librosa.stft.html
def print_stft(stft, title):
    fig, ax = plt.subplots()
    img = librosa.display.specshow(librosa.amplitude_to_db(stft,
                                                           ref=np.max),
                                   y_axis='log', x_axis='time', ax=ax)
    ax.set_title(title + " spectrogram")
    fig.colorbar(img, ax=ax, format="%+2.0f dB")
    fig

### get style and content losses from model(s)
def get_style_content_loss_model(cnn, style_audio, content_audio):
    content_layers = ["conv3", "conv4"]
    # idea: use it on adaptive pooled?
    style_layers = ["conv1", "conv2"]

    # model for style and content loss
    sc_model = nn.Sequential().to(device)

    ind = 0

    # pad the shorter audio with 0s
    style_len = style_audio.shape[0]
    content_len = content_audio.shape[0]

    if style_len > content_len:
        content_audio = np.pad(content_audio, (0, style_len - content_len),
"constant")
    else:
        style_audio = np.pad(style_audio, (0, content_len - style_len),
"constant")

    # turn audio into stft and normalise by VCTK mean, std
    mean = np.load("mean_comp.npy")
    std = np.load("std_comp.npy")
    style_stft = torch.from_numpy(normalise_stft(to_stft(style_audio), mean,
std)).float()
    content_stft = torch.from_numpy(normalise_stft(to_stft(content_audio),
mean, std)).float()

    style_stft = style_stft[None, :, :, :].to(device)
    content_stft = content_stft[None, :, :, :].to(device)

    content_losses = []
    style_losses = []

    # traverse to find conv layers
    for layer in cnn.children():
        # find the sequential block with the convolutional layer stack
        if isinstance(layer, nn.Sequential):
            for seq_layer in layer:
                # label the layers, add to sc model
                if isinstance(seq_layer, nn.Conv2d):
                    ind = ind + 1
                    layer_name = "conv" + str(ind)
```

```
                elif isinstance(seq_layer, nn.ReLU):
                    layer_name = "relu" + str(ind)
                elif isinstance(seq_layer, nn.MaxPool2d):
                    layer_name = "pool" + str(ind)

                sc_model.add_module(layer_name, seq_layer)

                if layer_name in content_layers:
                    content_target = sc_model(content_stft).detach()
                    content_loss = ContentLoss(content_target)
                    sc_model.add_module("con_loss" + str(ind), content_loss)
                    content_losses.append(content_loss)

                if layer_name in style_layers:
                    style_target = sc_model(style_stft).detach()
                    style_loss = StyleLoss(style_target)
                    sc_model.add_module("sty_loss" + str(ind), style_loss)
                    style_losses.append(style_loss)


    return sc_model, style_losses, content_losses

### generating new audio/gradient descent
def audio_style_transfer(cnn, style_audio, content_audio, synth_audio, steps,
                         style_weight, content_weight):
    # normalisation parameters
    mean = np.load("mean_comp.npy")
    std = np.load("std_comp.npy")

    # obtain model with loss/content modules
    sc_model, style_losses, content_losses =
get_style_content_loss_model(VCTKClassifier_model, style_audio, content_audio)

    # synth audio of len content audio; if style audio longer then pad synth
to same length
    synth_audio_len = synth_audio.shape[0]
    style_audio_len = style_audio.shape[0]
    content_audio_len = content_audio.shape[0]

    if style_audio_len > synth_audio_len:
        synth_audio = np.pad(synth_audio, (0, style_audio_len -
synth_audio_len), "constant")
    elif content_audio_len > synth_audio_len:
        synth_audio = np.pad(synth_audio, (0, content_audio_len -
synth_audio_len), "constant")

    # use requires grad to specify the input is to be adjusted, not the model
    # tensor must be contiguous for optimiser to work
    synth_stft = torch.from_numpy(normalise_stft(to_stft(synth_audio), mean,
std)).float().contiguous()
    synth_stft = synth_stft[None, :, :, :].to(device)
    synth_stft.requires_grad_(True)

    sc_model.eval()
    sc_model.requires_grad_(False)
```

```
    # optimiser for gradient descent on input
    optimiser = torch.optim.LBFGS([synth_stft])

    global step_no
    step_no = 0
    while step_no <= steps:
        def closure():
            optimiser.zero_grad()
            sc_model(synth_stft)

            # get style/content losses
            style_loss_total = 0
            content_loss_total = 0

            for module in style_losses:
                style_loss_total = style_loss_total + module.loss
            for module in content_losses:
                content_loss_total = content_loss_total + module.loss

            # apply weighting
            style_loss_total = style_loss_total * style_weight
            content_loss_total = content_loss_total * content_weight

            total_loss = style_loss_total + content_loss_total
            total_loss.backward()

            global step_no
            step_no = step_no + 1
            if step_no % (steps / 10) == 0:
                step_mes = "Step " + str(step_no) + "/" + str(steps)
                print("\n" + step_mes + ":")
                print("Style Loss: " + str(style_loss_total.item()))
                print("Content Loss: " + str(content_loss_total.item()))

                # to visualise convergence, print spectrogram
                stft_copy = synth_stft.detach().clone().squeeze()
                stft_copy =
get_complex_stft(un_normalise_stft(stft_copy.cpu().numpy(), mean, std))
                print_stft(stft_copy, step_mes)

            return style_loss_total + content_loss_total

        optimiser.step(closure)

    return synth_stft

## run algorithm on data
# get style audio
# get content audio
style_audio = load_audio("audioinpex\\human_female_greetings_08.wav")
content_audio = load_audio("audioinpex\\girl.wav")

#frameshift idea
style_audio = np.pad(style_audio, (content_audio.shape[0], 0), "constant")

style_weight = 1e3
```

```
content_weight = 1
steps = 5000

# clone content for synth audio
#synth_audio = np.random.randn(content_audio.shape[0])
#synth_audio = np.copy(content_audio)
synth_audio = np.copy(style_audio)
synth_stft = audio_style_transfer(VCTKClassifier_model, style_audio,
content_audio, synth_audio, steps, style_weight, content_weight)

# un-standardise STFT
mean = np.load("mean_comp.npy")
std = np.load("std_comp.npy")

synth_stft = synth_stft.detach().squeeze().cpu().numpy()
synth_stft_unnorm_complex = get_complex_stft(un_normalise_stft(synth_stft,
mean, std))

# display reference and generated audio
print_stft(librosa.stft(style_audio, n_fft = 512), "Style")
print_stft(librosa.stft(content_audio, n_fft = 512), "Content")
print_stft(synth_stft_unnorm_complex, "Synthesised")

# generate output waveform and trim to original length
output_audio = librosa.istft(synth_stft_unnorm_complex, n_fft = 512)
if (output_audio.shape[0] > content_audio.shape[0]):
    output_audio = output_audio[:content_audio.shape[0]]

# display stft of trimmed output waveform
print_stft(librosa.stft(output_audio, n_fft = 512), "Synthesised trimmed")
#write output to file
sf.write("output.wav", output_audio, 48000, subtype='PCM_24')
```

Appendix C: Data

Data can be found in `*\Progress\` and `*\Code\results\` in the archive.

Attempt 1: *(initial training attempt, crashed)*

```
In [10]: # running training
         torch.cuda.empty_cache()
         train(train_loader, model, loss_fn, optimiser)

loss: 4.689397  [   32/35098]
loss: 4.679688  [ 3232/35098]
loss: 4.682794  [ 6432/35098]
loss: 4.688493  [ 9632/35098]
loss: 4.685208  [12832/35098]
loss: 4.681238  [16032/35098]
loss: 4.694326  [19232/35098]
loss: 4.673812  [22432/35098]
loss: 4.686839  [25632/35098]

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [10], in <cell line: 3>()
      1 # running training
      2 torch.cuda.empty_cache()
----> 3 train(train_loader, model, loss_fn, optimiser)

Input In [9], in train(dataloader, model, loss_fn, optimiser)
     40 model.train()
     41 for batch, (X, y) in enumerate(dataloader):
---> 42     X, y = X.to(device), y.to(device)
     44     # pred error
     45     hx = model(X)

RuntimeError: CUDA error: out of memory
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

Attempt 2: *(initial training attempt, took many hours too long)*

```
In [*]: # running training
        torch.cuda.empty_cache()
        train(train_loader, model, loss_fn, optimiser)

loss: 4.688584  [   32/35098]
loss: 4.676201  [ 3232/35098]
loss: 4.686811  [ 6432/35098]
```

Attempt 3: *(initial training attempt, took many hours too long, barely converged)*

```
# running training
torch.cuda.empty_cache()
train(train_loader, model, loss_fn, optimiser)
```

```
loss: 5.248116  [   32/35098]
loss: 4.590794  [ 3232/35098]
loss: 4.760403  [ 6432/35098]
loss: 4.603220  [ 9632/35098]
loss: 4.592437  [12832/35098]
loss: 4.652910  [16032/35098]
loss: 4.609207  [19232/35098]
loss: 4.553506  [22432/35098]
loss: 4.601806  [25632/35098]
loss: 4.500363  [28832/35098]
loss: 4.544326  [32032/35098]
```

Attempt 6: *(intermediate training attempt, few hours to complete, slight convergence)*

```
# running training
torch.cuda.empty_cache()
train(train_loader, model, loss_fn, optimiser)
```

```
loss: 4.414498  [   32/35098]
loss: 4.210673  [ 3232/35098]
loss: 3.713645  [ 6432/35098]
loss: 3.921886  [ 9632/35098]
loss: 3.908557  [12832/35098]
loss: 3.534714  [16032/35098]
loss: 3.482207  [19232/35098]
loss: 3.430902  [22432/35098]
loss: 3.795104  [25632/35098]
loss: 3.562178  [28832/35098]
loss: 3.274305  [32032/35098]
```

```
# eval with test
test(test_loader, model, loss_fn)
```

```
275
Test Error:
 Accuracy: 13.6%, Avg loss: 3.380477
```

Attempt 7: *(intermediate training attempt, few hours to complete, moderate convergence)*

In [26]:
```
# eval with test
test(test_loader, model, loss_fn)
```

```
275
Test Error:
 Accuracy: 29.4%, Avg loss: 2.635858
```

In [*]:
```
torch.save(model.state_dict(), "modelparamsinitial.pth")
for x in range(10):
    train(train_loader, model, loss_fn, optimiser)

torch.save(model.state_dict(), "modelparamsafter.pth")
```

```
loss: 2.903501  [   32/35098]
loss: 2.797290  [ 3232/35098]
loss: 2.806016  [ 6432/35098]
loss: 2.380155  [ 9632/35098]
loss: 2.670480  [12832/35098]
loss: 2.909770  [16032/35098]
loss: 2.681043  [19232/35098]
loss: 2.041160  [22432/35098]
loss: 2.929291  [25632/35098]
loss: 2.459074  [28832/35098]
loss: 2.506557  [32032/35098]
loss: 2.371437  [   32/35098]
loss: 2.980277  [ 3232/35098]
loss: 2.523675  [ 6432/35098]
loss: 2.572237  [ 9632/35098]
loss: 2.595480  [12832/35098]
loss: 2.356660  [16032/35098]
loss: 2.752448  [19232/35098]
loss: 2.519759  [22432/35098]
loss: 2.593661  [25632/35098]
loss: 2.941199  [28832/35098]
loss: 2.433182  [32032/35098]
```

Adam Attempt 1: *(advanced training attempt, used Adam optimiser, few hours to complete, moderate convergence)*

In [22]:
```
# running training
torch.cuda.empty_cache()
train(train_loader, model, loss_fn, optimiser)
```

```
loss: 7.612947  [   32/35098]
loss: 4.355151  [ 3232/35098]
loss: 3.938634  [ 6432/35098]
loss: 3.956719  [ 9632/35098]
loss: 3.506454  [12832/35098]
loss: 3.311431  [16032/35098]
loss: 2.777250  [19232/35098]
loss: 2.792869  [22432/35098]
loss: 3.234783  [25632/35098]
loss: 2.657797  [28832/35098]
loss: 2.126100  [32032/35098]
```

In [23]:
```
# eval with test
test(test_loader, model, loss_fn)
```

```
275
Test Error:
 Accuracy: 32.5%, Avg loss: 2.354607
```

Adam Attempt 2: *(advanced training attempt, several days to complete, good convergence)*

In [*]:
```
for epoch in range(epochs):
    train(train_loader, model, loss_fn, optimiser)
    scheduler.step()
    test(test_loader, model, loss_fn)
```

```
loss: 1.914374  [   32/35098]
loss: 1.672573  [ 3232/35098]
loss: 1.472868  [ 6432/35098]
loss: 1.062121  [ 9632/35098]
loss: 1.691667  [12832/35098]
loss: 1.327792  [16032/35098]
loss: 1.344281  [19232/35098]
loss: 1.393734  [22432/35098]
loss: 1.447721  [25632/35098]
loss: 1.697969  [28832/35098]
loss: 1.463234  [32032/35098]
275
Test Error:
 Accuracy: 60.9%, Avg loss: 1.294166

loss: 0.773767  [   32/35098]
```

Loss over additional epochs of ***Adam Attempt 2****:*

Train Loss	Test Loss
7.612947	
4.355151	
3.938634	
3.956719	
3.506454	
3.311431	
2.77725	
2.792869	
3.234783	
2.657797	
2.1261	2.3546
1.914374	2.3546
1.672573	2.3546
1.472868	2.3546
1.062121	2.3546
1.691667	2.3546
1.327792	2.3546
1.344281	2.3546
1.393734	2.3546
1.447721	2.3546
1.697969	2.3546
1.463234	1.2942
0.773767	1.2942
1.011691	1.2942
1.504476	1.2942
1.226812	1.2942
1.09937	1.2942
1.392967	1.2942
0.912616	1.2942
0.997804	1.2942
0.803686	1.2942
1.194622	1.2942
0.828288	1.2495
1.001601	1.2495
1.417289	1.2495
0.891374	1.2495
1.343767	1.2495
1.060934	1.2495
1.1814	1.2495
1.292734	1.2495
0.591323	1.2495
0.843128	1.2495
0.880918	1.2495
0.756832	0.8655
0.986685	0.8655
0.661843	0.8655
0.643247	0.8655
0.986685	0.8655
0.661843	0.8655
0.643247	0.8655
0.662337	0.8655
0.679901	0.8655
0.644361	0.8655
0.437678	0.8655
0.742945	0.8655
0.701635	0.8655
0.56937	0.8655
0.872201	0.7844
0.821767	0.7844
0.759427	0.7844
0.492039	0.7844
0.313642	0.7844
1.012631	0.7844
1.312413	0.7844
0.515619	0.7844
0.413681	0.7844
0.508875	0.7844
0.622293	0.7844
0.786971	0.6839

Test accuracy over even further epochs of ***Adam Attempt 2****:*

83.90%	85.30%	85.90%	86.10%
83.80%	85.50%	85.90%	86.00%
84.60%	85.70%	86.10%	85.90%
84.60%	85.90%	86.00%	86.00%
85.20%	85.70%	86.10%	86.00%

Graphs for these values can be found in the chapter **Experimentation and Results** *under the* Initial classifier *subheading.*

Final: *(final training attempt, several days to complete, best convergence)*

Epoch	Train Loss	Test Loss	Test Accuracy
0	4.464233		
0.1	3.897186		
0.2	3.446777		
0.3	3.244422		
0.4	2.736082		
0.5	2.207885		
0.6	2.23249		
0.7	2.003462		
0.8	1.654311		
0.9	1.001155		
1	2.238773	1.733902	50.80%
1.1	1.91746	1.733902	50.80%
1.2	1.336861	1.733902	50.80%
1.3	0.975038	1.733902	50.80%
1.4	1.503572	1.733902	50.80%
1.5	0.975773	1.733902	50.80%
1.6	1.015292	1.733902	50.80%
1.7	1.035469	1.733902	50.80%
1.8	0.941359	1.733902	50.80%
1.9	0.762376	1.733902	50.80%
2	0.826016	1.025941	69.40%
2.1	1.288123	1.025941	69.40%
2.2	0.901037	1.025941	69.40%
2.3	0.758784	1.025941	69.40%
2.4	0.637047	1.025941	69.40%
2.5	0.888524	1.025941	69.40%
2.6	0.901081	1.025941	69.40%
2.7	0.57021	1.025941	69.40%
2.8	0.797797	1.025941	69.40%
2.9	0.25591	1.025941	69.40%
3	0.465191	0.672488	80.00%
3.1	0.455571	0.672488	80.00%
3.2	0.321948	0.672488	80.00%
3.3	0.631597	0.672488	80.00%
3.4	0.403104	0.672488	80.00%
3.5	0.490049	0.672488	80.00%
3.6	0.796708	0.672488	80.00%
3.7	0.425197	0.672488	80.00%
3.8	0.888902	0.672488	80.00%
3.9	0.38343	0.672488	80.00%
4	0.308931	0.703675	79.40%
4.1	0.311559	0.703675	79.40%
4.2	0.219801	0.703675	79.40%
4.3	0.231765	0.703675	79.40%
4.4	0.362111	0.703675	79.40%
4.5	0.355739	0.703675	79.40%
4.6	0.338853	0.703675	79.40%
4.7	0.333431	0.703675	79.40%
4.8	0.464725	0.703675	79.40%
4.9	0.673368	0.703675	79.40%
5	0.171932	0.458431	86.30%
5.1	0.864804	0.458431	86.30%
5.2	0.767665	0.458431	86.30%
5.3	0.341065	0.458431	86.30%
5.4	0.273837	0.458431	86.30%
5.5	0.24504	0.458431	86.30%
5.6	0.255948	0.458431	86.30%
5.7	0.447391	0.458431	86.30%
5.8	0.387948	0.458431	86.30%
5.9	0.349256	0.458431	86.30%
6	0.190572	0.462848	86.60%
6.1	0.247172	0.462848	86.60%
6.2	0.213811	0.462848	86.60%
6.3	0.289042	0.462848	86.60%
6.4	0.163893	0.462848	86.60%
6.5	0.269561	0.462848	86.60%
6.6	0.278807	0.462848	86.60%
6.7	0.225892	0.462848	86.60%
6.8	0.442743	0.462848	86.60%
6.9	0.15553	0.462848	86.60%
7	0.260344	0.441864	87.00%
7.1	0.14701	0.441864	87.00%
7.2	0.043669	0.441864	87.00%
7.3	0.079303	0.441864	87.00%
7.4	0.293366	0.441864	87.00%
7.5	0.123277	0.441864	87.00%
7.6	0.040464	0.441864	87.00%
7.7	0.074694	0.441864	87.00%
7.8	0.204868	0.441864	87.00%
7.9	0.469014	0.441864	87.00%
8	0.12411	0.329428	90.50%
8.1	0.140698	0.329428	90.50%
8.2	0.08915	0.329428	90.50%
8.3	0.051721	0.329428	90.50%

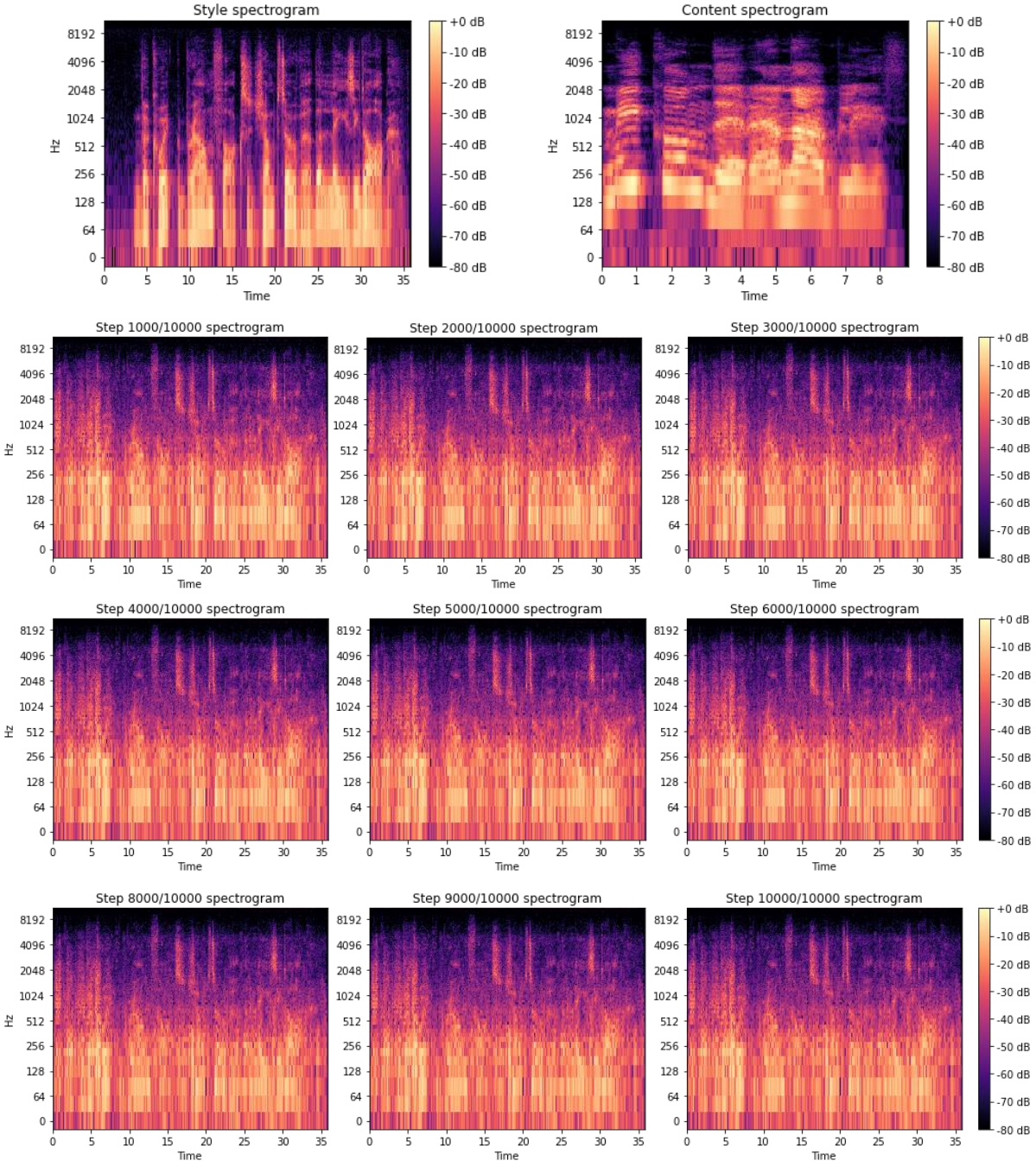
8.4	0.239793	0.329428	90.50%
8.5	0.088695	0.329428	90.50%
8.6	0.518129	0.329428	90.50%
8.7	0.073184	0.329428	90.50%
8.8	0.37804	0.329428	90.50%
8.9	0.063579	0.329428	90.50%
9	0.059045	0.327231	90.80%
9.1	0.194458	0.327231	90.80%
9.2	0.122812	0.327231	90.80%
9.3	0.167095	0.327231	90.80%
9.4	0.095671	0.327231	90.80%
9.5	0.056747	0.327231	90.80%
9.6	0.071057	0.327231	90.80%
9.7	0.116496	0.327231	90.80%
9.8	0.105806	0.327231	90.80%
9.9	0.060436	0.327231	90.80%
10	0.049054	0.32092	91.50%
10.1	0.044189	0.32092	91.50%
10.2	0.324905	0.32092	91.50%
10.3	0.065156	0.32092	91.50%
10.4	0.324975	0.32092	91.50%
10.5	0.034988	0.32092	91.50%
10.6	0.376625	0.32092	91.50%
10.7	0.13392	0.32092	91.50%
10.8	0.156443	0.32092	91.50%
10.9	0.076614	0.32092	91.50%
11	0.064789	0.313472	91.60%
11.1	0.043908	0.313472	91.60%
11.2	0.07908	0.313472	91.60%
11.3	0.097967	0.313472	91.60%
11.4	0.095977	0.313472	91.60%
11.5	0.101201	0.313472	91.60%
11.6	0.050455	0.313472	91.60%
11.7	0.009205	0.313472	91.60%
11.8	0.171665	0.313472	91.60%
11.9	0.072175	0.313472	91.60%
12	0.12138	0.280167	92.70%
12.1	0.100038	0.280167	92.70%
12.2	0.053585	0.280167	92.70%
12.3	0.064831	0.280167	92.70%
12.4	0.017189	0.280167	92.70%
12.5	0.035993	0.280167	92.70%
12.6	0.007201	0.280167	92.70%
12.7	0.275467	0.280167	92.70%
12.8	0.154035	0.280167	92.70%
12.9	0.330044	0.280167	92.70%
13	0.128332	0.308213	92.20%
13.1	0.150948	0.308213	92.20%
13.2	0.082132	0.308213	92.20%
13.3	0.044512	0.308213	92.20%
13.4	0.007346	0.308213	92.20%
13.5	0.051002	0.308213	92.20%
13.6	0.028699	0.308213	92.20%
13.7	0.00583	0.308213	92.20%
13.8	0.105635	0.308213	92.20%
13.9	0.178368	0.308213	92.20%
14	0.09206	0.297419	92.60%
14.1	0.123048	0.297419	92.60%
14.2	0.053325	0.297419	92.60%
14.3	0.02525	0.297419	92.60%
14.4	0.188893	0.297419	92.60%
14.5	0.007521	0.297419	92.60%
14.6	0.034638	0.297419	92.60%
14.7	0.021064	0.297419	92.60%
14.8	0.076027	0.297419	92.60%
14.9	0.117988	0.297419	92.60%
15	0.014459	0.295943	92.50%

Graphs for these values can be found in the chapter **Experimentation and Results** *under the* Final classifier *subheading.*

Example NST attempt: *(mostly decreasing losses proportional to weights, refer to* `*\Code\results\` *in archive for more results of this type)*

Data taken from `cw1_sw1e3_sc1c2_cc3c4_LBFGS_s10k_style.xlsx`.

Step	Style Loss	Content Loss
1000	0.031178	1.033086
2000	0.013681	0.847565
3000	0.008017	0.71012
4000	0.006245	0.662413
5000	0.005116	0.616003
6000	0.004495	0.595337
7000	0.004056	0.577607
8000	0.003769	0.569168
9000	0.003548	0.56209
10000	0.003218	0.53793



Appendix D: Archive directory contents and description

***/**

- **Code/**
  - `comp_model_params_ep_9.pth`
  - `comp_model_params_further_ep9.pth`

    Content description:
    Final classifier parameters used to import the model to the NST algorithm; experimentation alternated between the two (it is suspected the second is overfit).

  - `mean.npy`
  - `mean_comp.npy`
  - `std.npy`
  - `std_comp.npy`

    Content description:
    Matrices of the means and standard deviations of the STFTs across each feature of the custom VCTK-based dataset; these were recalculated across the split components of the complex values of the final dataset (the non-comp versions are from the initial, non-reversible STFT dataset).

  - `reversible stft dataset creation.ipynb`
  - `VCTK classifier complex.ipynb`
  - `style transfer complex.ipynb`

    Content description:
    Jupyter notebooks containing the final code for generating the custom dataset; defining and training the classifier, and performing style transfer, respectively. Abbreviated versions of these files are detailed under **Appendix B: Code**.

  - **audioinpex/**
    - `[name].wav`/`[name].flac`

      Content description:
      10 examples of audio files that were used in experimentation and producing the results in performing the style transfer algorithm.

  - **classifier params/**
    - `[name].pth`

      Content description:
      All parameters/model weights for the classifier generated during training. If the filename contains "comp", it is for the final classifier. Otherwise, it is for the initial classifier.

  - **old code and backups/**
    - `audio_style_transfer func backup.ipynb`
    - `create stft data.ipynb`
    - `VCTK classifier real.ipynb`

      Content description:
      Selections of old code that was deemed necessary to place in the archive to demonstrate progress, in order: the old style transfer implementation for the initial classifier (required random phase data to reconstruct); the old custom dataset creation (with the non-reversible STFT processing); and the file from which the initial classifier was defined and trained.

└ **`results/`**

Content description:

This archive contains a selection of the waveforms (wave files) and losses/spectrograms (spreadsheets) generated during experimentation on the NST implementation, some of them being described under the Style Transfer *subheading of the* **Experimentation and Results** *chapter:*

- └ `cw0_sw1e9_cc4_sc1c2c3_LBFGS_s100k_content.wav`
- └ `cw0_sw1e9_cc4_sc1c2c3_LBFGS_s100k_content.xlsx`

  Content description:

  *Results using content weight **0**, style weight **1e9**; content layers [**conv4**], style layers [**conv1, conv2, conv3**]; **LBFGS** optimiser; **100,000** steps; initialised from content audio.*

  ***Style** audio:* `../audioinpex/p225_006_mic2.flac`

  ***Content** audio:* `../audioinpex/p304_009_mic2.flac`

- └ `cw1e2_sw1e7_cc1_sc1c2c3c4_LBFGS_s50k_rand.wav`
- └ `cw1e2_sw1e7_cc1_sc1c2c3c4_LBFGS_s50k_rand.xlsx`

  Content description:

  *Results using content weight **100**, style weight **1e7**; content layers [**conv1**], style layers [**conv1, conv2, conv3, conv4**]; **LBFGS** optimiser; **50,000** steps; initialised from random noise.*

  ***Style** audio:* `../audioinpex/p225_006_mic2.flac`

  ***Content** audio:* `../audioinpex/p304_009_mic2.flac`

- └ `cw1_sw1_cc1_sc1c2c3_LBFGS_s10k_style.wav`
- └ `cw1_sw1_cc1_sc1c2c3_LBFGS_s10k_style.xlsx`

  Content description:

  *Results using content weight **1**, style weight **1**; content layers [**conv1**], style layers [**conv1, conv2, conv3**]; **LBFGS** optimiser; **10,000** steps; initialised from style audio.*

  ***Style** audio:* `../audioinpex/boy.wav`

  ***Content** audio:* `../audioinpex/girl.wav`

- └ `cw1_sw1e3_sc1c2_cc3c4_LBFGS_s10k_style.wav`
- └ `cw1_sw1e3_sc1c2_cc3c4_LBFGS_s10k_style.xlsx`

  Content description:

  *Results using content weight **1**, style weight **1000**; content layers [**conv3, conv4**], style layers [**conv1, conv2**]; **LBFGS** optimiser; **10,000** steps; initialised from style audio.*

  ***Style** audio:* `../audioinpex/fox.wav`

  ***Content** audio:* `../audioinpex/human_female_greetings_08.wav`

- └ `fs_cw1_sw1e3_sc1c2_cc3c4_LBFGS_s10k_style.wav`
- └ `rv_fs_cw1_sw1e3_cc1c2_sc3c4_LBFGS_s10k_style.wav`
- └ `cw1_sw1e3_sc1c2_cc3c4_diff_style.wav`
- └ `cw1_sw1e3_sc1c2_cc3c4_diff_style.xlsx`

  Content description:

  *Same parameters to generate as directly previous files. However, they differ in that their style audio was **zero-pad frameshifted**; the second has the **style** and **content** audio files reversed; and the third*

uses `../audioinpex/sample.wav` *as its* ***style*** *instead. The spreadsheet details the loss and spectrograms in generating the third wave file.*

- `cw1_sw1e3_cc3c4_sc1c2c3c4fc1.wav`

  Content description:
  Same parameters to generate as directly previous files.
  Style *audio:* `../audioinpex/girl.wav`
  Content *audio:* `../audioinpex/fox.wav`

- `304009_s225006.wav`
- `304009_fox.wav`
- `304009.xlsx`

  Content description:
  Same parameters to generate as directly previous files, with a spreadsheet showing the different spectrograms given the same content audio.
  Style *audios:* `../audioinpex/p225_006_mic2.flac`
  `../audioinpex/fox.wav`
  Content *audio:* `../audioinpex/ p304_009_mic2.flac`

- **`further/`**
  - `[content_name]_[style_name].wav`

    Content description:
    Further example outputs using the final parameters listed previously. Names for style and content reference audio used are in the output filename.

- **`STFTcomp/`**
  - `stftc0`

    Content description:
    The first item of the final custom VCTK-derived STFT dataset. Contains a list with the standardised $2 * 257 * L$ *array of the STFT, and a string label identifying the speaker. This was saved and is loadable using the* pickle *library. The actual directory contains 43873 examples.*

- **`Progress/`**
  - `Adam Attempt 1.png`
  - `Adam Attempt 2 Graph.png`
  - `Adam Attempt 2 Refining.png`
  - `Adam Attempt 2.png`

    Content description:
    Screenshots of loss graphs and terminal outputs for advanced Adam training on the initial classifier.

  - `Attempt 1.png`
  - `Attempt 2.png`
  - `Attempt 3.png`
  - `Attempt 6.png`
  - `Attempt 7 loss.png`

    Content description:
    Screenshots of loss graphs and terminal outputs for advanced Adam training on the initial classifier.

└ `Adam Attempt 2.xlsx`
└ `VCTK Final Graphs.xlsx`

Content description:
Data and resultant graphs for loss and test set accuracy for the final attempts at the initial classifier, and final classifier respectively.

Appendix E: Word Count

Total word count excluding appendices: **6672.**