Transcribing Musical Notes with a Convolutional Encoder-Decoder Neural Network

Robert Simpson

April 2023

Contents

1 Introduction

1.1 Abstract

This project will explore using deep learning to transcribe musical notes to sheet music. During the process of this, I will collect and pre-process my own data, ultimately creating a new dataset of audio files and corresponding sheet music images. I will then be creating a convolutional encoder-decoder network that will learn features from these audio files and generate their corresponding sheet music image based on this.

1.2 Introduction

The aim of this project will be to create a model that can produce human-readable sheet music images from audio files of the format .wav. Transcription of audio into sheet music falls under the field of Music Information Retrieval (MIR). The key here is that the model needs to not only be able to detect the note (or frequency) at which the sound is being played, but also learn how to convert these frequencies into sheet music format. This project will demonstrate that this is possible using an encoder-decoder model. At present, I am unable to find other music-related work using this exact method, which is why I want to demonstrate that this is a plausible approach in hopes this can be expanded on in the future. I will be comparing the use of different loss functions (L1Smooth, MSE, SSIM) to see which provides the best reconstruction. Different variations of my proposed model will be evaluated quantitatively using the evaluation metric Peak Signal-to-Noise Ratio (PSNR). As well as this, a basic human eye test can be used to evaluate the model qualitatively. If the sheet music is readable by a human, then this can be considered a satisfactory reconstruction. To train and test this model, I will be creating a new dataset from scratch. This dataset will comprise of audio files for each of the 88 notes on a piano as well as their corresponding sheet music images. By using software called Arturia V, I will be able to produce notes from lots of different types of pianos. In doing this, I want to show that the model can learn useful latent representations regardless of the tone of the audio. Thus, making my model more robust as well as allowing me to easily expand the dataset.

1.3 Related Work

1.3.1 Image Synthesis

A key concept which underpins my work is the problem of image synthesis. In this section I would like to discuss some other works concerning image synthesis. Reed et al. (2016) implement a GAN for image synthesis from text. In their problem they learn text encodings which differs from mine as I will be learning audio encodings. The generator part of the GAN must not only learn to generate plausible images but also match images with the correct text description. The discriminator then learns to reject fake looking images and incorrect pairings of image and text. The work of Ramesh et al. (2022) shows a more recent example of text-conditional image generation. They use a contrastive model known as CLIP (Radford et al., 2021) to generate an image embedding based on a text caption. A decoder then generates an image conditioned on this image embedding. This more modern approach produces much higher quality images than the previous paper discussed.

1.3.2 Cross-Modal Models

Since my proposed model will involve cross modality (Ngiam et al., 2011) (audio and image), I would also like to discuss some other cross-modal works in deep learning. Jin et al. (2021) implement a model that aims to match corresponding videos and music. This is done by first extracting features from the audio and video modalities using convolutional layers, max pooling and then ReLU. In this part of the model, they also use an attention mechanism based on the work by Xu et al. (2017) to reduce redundant data. They then use an embedding network to map the feature vectors into an embedding space. This then allows the videos and music to be compared and as such, facilitates cross-modality. We can see another application of cross-modality with the work by Spurr et al. (n.d.).Here, they learn a cross-modal latent space of RGB, 2D and 3D images of hand poses. The aim here being to allow for a posterior estimate of a hand pose to be generated across any of these three modalities. In this case, this is done by using a Variational Autoencoder (Kingma and Welling, 2013) which encodes data as a distribution over the latent space instead of a single point unlike my model.

1.3.3 Convolutional Encoder-decoder Models

As I am using a convolutional encoder-decoder model, I would also like to discuss work in the Deep Learning field using such models. Ronneberger et al. (2015) first proposed this convolutional encoder-decoder style network with their network U-Net. The main idea is that the image is encoded into a latent variable but is then upsampled back to an image. This was initially proposed to easily segment biomedical images. Image segmentation is an issue that, when overcome, can be very useful in a wide range of applications such as scene understanding or autonomous driving. Mao et al. (n.d.) implement a convolutional encoder-decoder for image restoration. Unlike the previous paper discussed, they include symmetric skip connections in their architecture. This is inspired by highway networks and deep residual networks and involves skipping layers of a network instead of always feeding the output of a layer into the neighbouring layer. In their paper, the connections are symmetric as they connect convolutional layers in the encoder to their mirrored deconvolutional layer in the decoder. They do this as, for their problem, they want to retain lots of image details for a high-quality restoration. Without the skip connections, image details can be lost as the network gets deeper. The feature maps passed by the skip connections will carry more detail and thus allow for better restoration during the deconvolution stage. As my model is not strictly for image restoration and involves translation between modalities, I will not be using skip connections like this.

1.3.4 Deep Learning and Music/Audio Work

Finally, my project looks at applying deep learning to music and audio-based problems and thus, I would like to discuss to other works involving deep learning and music/audio. The work of Arandjelović and Zisserman (2017) looks to use deep learning to learn audio-visual correspondence. This means, does an example sound correspond to an example image (i.e does the sound come from the image)? They do this via a vision subnetwork and an audio subnetwork which both reduce their respective data to $1\times1\times512$ vectors. These vectors are then concatenated into a $1\times1\times1024$ vector which is passed through a fusion network to see if they correspond or not. The audio subnetwork used in this paper helped influence my choice of architecture as it showed me that we can convert audio to spectrograms and then learn features of the audio by feeding these spectrograms into a convolutional network. Moreover, we still retain useful information if we reduce the dimensions of these spectrograms using pooling and encode them into latent variables. Another influential paper for me in the field was Dorfer

et al. (2018). This paper looks at a similar issue I am trying to address in audio and sheet music cross-modality. However, they take the different approach of formulating a retrieval problem. In this case, they learn an embedded space of sheet music and audio and force corresponding data samples to have a minimal cosine distance. This means, if you want to query the embedded space with an audio sample, the sheet music image with the smallest cosine distance to this query will be selected as the corresponding image. In contrast to this, my project will look to take an audio sample as input and reconstruct a new sheet music image based on the learned features, thus making it a generative model instead of a retrieval-based model.

As previously mentioned, I am presently unable to find any work that tackles this particular problem with the approach I am choosing to use and thus there is no existing model/work to directly improve on.

1.4 Background

In this section, I would like to discuss some background knowledge related to the problem that will be useful for the reader in later sections.

1.4.1 Musical Background

Since undertaking this project requires some musical knowledge, I will discuss a few concepts relevant to the problem. Firstly, I would like to discuss musical notes. A musical note in sheet music represents the pitch and duration of a sound. The pitch of a note is directly related to the frequency of the sound waves which produce it. This is important as I will later introduce a concept called spectrograms which aim to represent such frequencies but in image format. On a standard piano, there are 88 of these notes with the lowest being an A-1 and the highest being C7. Each of these notes can be represented in sheet music format and, as such, my dataset consists of 88 sheet music images. Figure 1.4.1.1 shows the sheet music representation of the note A#0. The symbol on the far left of the image is known as a bass clef. There are two types of clefs to represent all the keys on a piano. The other is known as a treble clef (this can be seen on the far left of Figure 1.4.1.2). The clef tells us what side of the piano the note is on. A bass clef tells us it is a note played in the left hand (so on the left side of the piano) and a treble clef tells us that it is a note played in the right hand (on the right side of the piano). Also, in Figure 1.4.1.1, a small 8 can be seen under the bass clef. This tells the pianist that the note is in the left hand and is an octave down. Similarly, in Figure 1.4.1.2, a small 8 can be seen above the treble clef. This tells the pianist that the note is in the right hand and is an octave up. We can also see a C symbol in both figures. This represents the time signature which in this case is common time or 4/4. The rightmost symbols in both figures are the notes. Thanks to the clef, we know the area of the piano in which our note is in. Now, the position of the note head (the elliptical part of the symbol) within the horizontal lines (staves) tells us what specific note to play. You will also notice the inverted hash symbol in Figure 1.4.1.1. This is known as a sharp symbol.

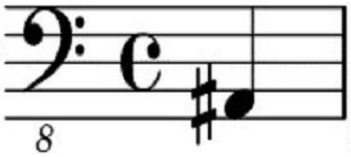

Figure 1.1: A#0 sheet music image

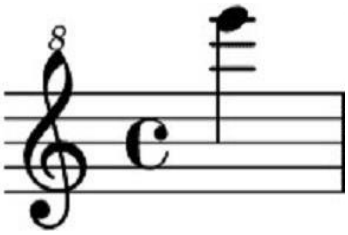

Figure 1.2: F6 sheet music image

1.4.2 Convolutional Neural Networks

As part of the methodology of this project, I will be implementing a variation of a Convolutional Neural Network. As such, I will discuss the key ideas behind this type of network. A CNN usually consists of multiple blocks of the following: first a convolutional layer, then a pooling layer (average pooling or max pooling) and finally an activation layer (usually ReLU). The role of the convolutional layer is to extract features like edges and colour. This is done by "convolving" a 2D array of weights (called a kernel or filter) with the input. Convolving is the process of sliding the kernel across the input and calculating the dot product of the kernel weights and the values of the input (for example the pixel values of an image). The output of this layer is known as a feature map. We can pass this feature map to a pooling layer. A pooling layer is used to reduce the dimensions of our data as it is fed forward through different blocks. This also eases the computational burden when processing the data. We can apply max pooling which takes the max value of the values covered by the pooling kernel (usually a 2×2 or 4×4 kernel) or average pooling which instead averages all the values. Evidently, using a 2×2 kernel takes 4 values and only outputs 1 and using a 4×4 kernel takes 16 values and only outputs 1. This is how pooling reduces the dimensions of data. After the pooling layer has reduced the dimensions, the data is then passed to an activation layer. The activation layer is important as this is what adds non-linearity to the network which would otherwise only be a linear model. Non-linearity allows us to develop more complex representations and functions that a simple linear model cannot.

1.4.3 Mel Spectrograms

As mentioned earlier, I will be implementing a variation of a Convolutional Neural Network. However, CNNs are used for images but the input data for my problem is an audio file. So how do we use audio data with CNNs? This is where Mel spectrograms come in. Spectrograms provide a 'snapshot' of an audio wave. A spectrogram is produced by segmenting the audio signal and then applying a Fourier Transform to each segment to determine the frequencies contained in that segment. These Fourier Transforms are then combined into a single plot to give us a spectrogram. Spectrograms are plotted with frequency (y-axis) against time (x-axis) and different colours are used to indicate the amplitude of each frequency. A brighter colour means higher energy. A Mel Spectrogram makes two alterations to a normal spectrogram. First it uses the Mel Scale instead of frequency and second, it uses the Decibel Scale instead of amplitude. This gives a more accurate representation of the way humans perceive sound and produces much better spectrograms. Figure 1.4.3.1 shows a Mel Spectrogram created from a data sample from my dataset. The reason why these visual representations of audio are so important to this project is that audio data can be converted into these spectrograms and then fed into the CNN architecture since they are images. This is why spectrograms are used for most Deep Learning projects

that involve audio data.

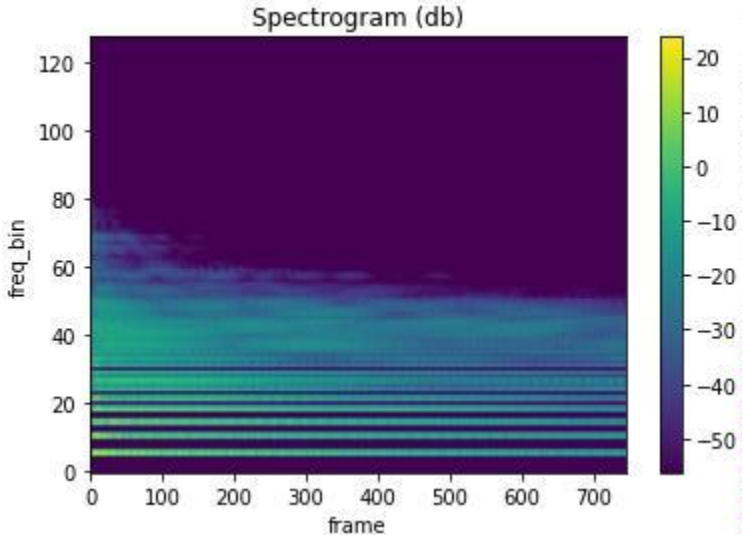

Figure 1.3: Mel spectrogram of audio file from my dataset

1.4.4 Musical Instrument Digital Interface (MIDI)

MIDI allows us to transmit information such as musical notes, timings, and pitch information from a MIDI controller (for example a digital keyboard or a synthesizer) to a computer. However, MIDI does not transmit sound so you cannot "play" a MIDI file. Consequently, we must use VSTs if we want to listen to an audio representation of our MIDI file.

1.4.5 Virtual Studio Technology (VST)

VST instruments emulate the sound of real instruments. They receive information via MIDI and output digital audio. We can make use of this within a Digital Audio Workstation (DAW) which allows us to create or manipulate MIDI files and then export to an audio file using our chosen VST.

2 Methodology

2.1 Dataset

To undertake this project, I required a dataset of sheet music images and audio samples. There were no existing datasets suitable for the aims of this project and thus I needed to create a new dataset from scratch. This is one of the major contributions of this project to the field due to the time and effort required to produce such a dataset. The dataset consists of .wav audio files and sheet music images all pre-processed and distributed as equally as possible into a test and a train set. A training set is the set of samples from a dataset used by the model to learn from and update its parameters. A testing set is used to evaluate the performance of a model. The samples in a testing set are not used in the learning process and therefore are "unseen". Below I will describe the features of the dataset and the process undertaken to create it.

2.1.1 Audio Files

I first needed to create some audio files. To do this I used a VST called Arturia V and a DAW called Soundbridge. Soundbridge allowed me to create a MIDI representation of each of the 88 keys on a piano that are all the same length. This is vital as the dimensions of all my data must be the same. Arturia allows for the corresponding audio representations of these MIDI representations to be produced. A key part of why I chose Arturia was that it has large collection of different types of pianos (for example a German Grand, a Japanese Grand, an Upright etc.). I theorised that this would allow me to build a large dataset with all unique data samples as, although the samples would have common semantic meanings (for example an A1 note played on a German Grand is the same semantically as an A1 note played on a Japanese Grand), the actual audio data would have slightly different features due to the different tones produced by different instruments. This is important because, if my dataset was simply made up of duplicated sounds from one instrument, then, when it comes to making a test and train split, I would have no distinct unseen data for the model to be tested on. It is important for test data to be unseen (not used by the model to update parameters during the training process) as this tests the model's ability to generalise. If a model learns to fit the training data too well, this will lead to what is known as overfitting. This is when a model's performance on the training data keeps improving but the performance on the testing data starts to get worse. We need unseen data to test our model to make sure it can generalise and is not overfitting. See Figure 2.1.1.1 for an example loss plot where overfitting has occurred. A model that performs well in training but extremely poor in testing is useless to us. Using Arturia allows me to produce enough distinct samples to train the model to a good standard of performance whilst also allowing me to create unseen samples to test the model's ability to generalise. Figure 2.1.1.2 and Figure 2.1.1.3 show examples of audio samples from two different instruments but with the same semantic meanings. We can see that the spectrograms have some subtle differences. I also theorised that this would improve the robustness of the model. If the model trains using lots of different representations of a note, then, in theory, the model should be able to recognise the note being played no matter the piano it is played on. This is a great quality for a model to have as, in a real-world application, the model would be used for a range of pianos and so it needs to learn to deal with this. The process of data augmentation uses similar logic, and I will discuss this later in this section. The step-by-step process I took to create new audio samples was as follows. First, I select the instrument I want to use for this new set of data I am creating. I then select the note that I want the audio for by creating a MIDI track and inserting a MIDI block in the location of the note I want. This tells Soundbridge the note that I want and the duration I want it for. Once

I've done this, I can "export loop" which basically allows me to export the MIDI track to a .wav file. The audio will use the instrument I selected from my VST. This process is repeated 88 times (once for each piano key) per instrument I choose to use. After I have got the audio for all 88 notes, I then must add each of the new audio files as a new record in my annotations file. Evidently, all these steps make for a time-consuming process (especially when thousands of data samples are needed), and this is why the dataset forms a major part of my project and is a key contribution to the field.

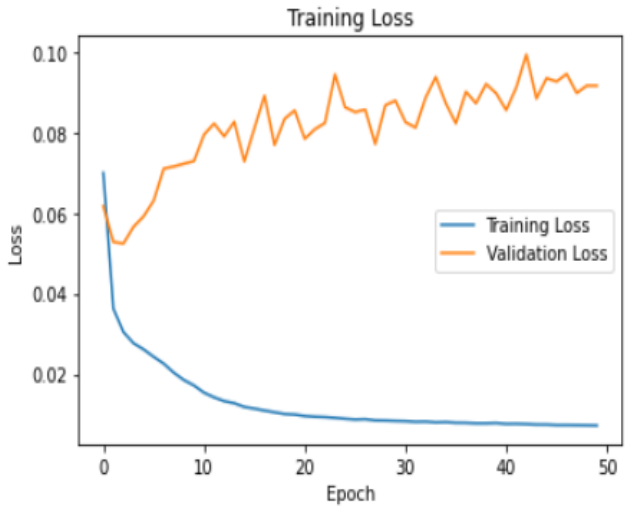

Figure 2.1: Example of overfitting model loss plot

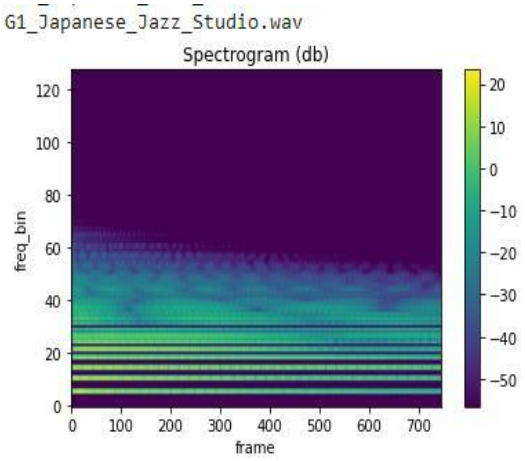

Figure 2.2: Mel spectrogram of G1 note played on Japanese Jazz Piano

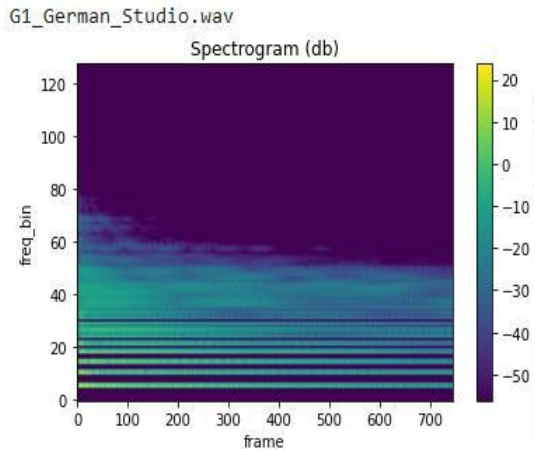

Figure 2.3: Mel spectrogram of G1 note played on German Grand Piano

2.1.2 Sheet Music Images

Next, since the aim of my project is to go from audio to sheet music images, I needed some sheet music images for my ground truth. As previously mentioned, there are 88 keys on a piano and thus I will need 88 sheet music images if I want to represent all the keys. To achieve this, I used the Melobytes MIDI to sheet music tool online. When creating my audio samples, I made sure to also export the MIDI representations for this very reason. I only needed to do this once though (i.e for one instrument) as the MIDI representations stay the same regardless of the instrument. An alternative to Melobytes would have been to use MuseScore which would allow me to manually create the sheet music. This would be preferred normally, however, for this project I chose to use Melobytes to save some time and since the general aim of this project can still be achieved and later revisited with a more refined dataset if needed. The Melobytes tool outputs a full A4 page but I only needed a small segment of sheet music since I am only considering one note. This meant the next step was to crop the image to the smallest possible dimensions whilst retaining all the key features. Another important factor was that I needed all the images to have the same dimensions. This is because, I need to specify what dimensions I want my model to output in the final layer. Since some of the images have details that can be quite high or low in the image (see Figure 2.1.2.1 and Figure 2.1.2.2), I needed dimensions that accounted for this. I decided to set dimensions for all my images to 162×300 (height x width). To set these dimensions, I used the resize tool in paint to manually set them to 162×300 after cropping them. Again, this could be quite a time-consuming process even with using Melobytes. An important note about the semantic meanings of piano keys is that some keys can be called a sharp or a flat depending on musical context. Since my project only deals with single notes, there is no musical context and thus I only use sharps here.

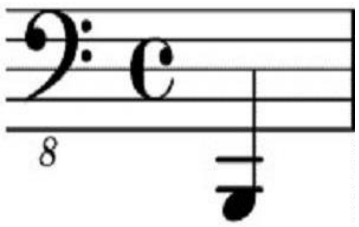

Figure 2.4: B-1 sheet music image

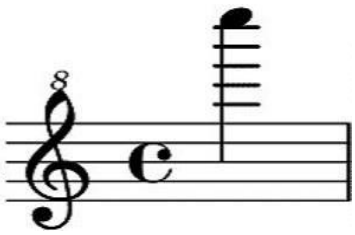

Figure 2.5: C7 sheet music image

2.1.3 Dataset Distribution

With any AI/machine learning dataset, we need our data to be split into a testing set and a training set. As mentioned previously, a testing set contains the samples we want to be "unseen" (unseen meaning not used to update the model parameters), whereas a training set contains the samples we want our model to use to learn from (update parameters) during the training stage. As such, we need a larger training set than testing set as the model will perform better with lots of training samples. Figure 2.1.3.1 shows my test and train split. Common practice in machine learning is to have a 20:80 split and, as you can see from my figure, I have 2464 audio samples in total with 516 (around 20.9%) being in the testing set and 1948 being in the training set. Another important point when distributing a dataset is to make sure your testing set is representative of the dataset. In my scenario for example, I could have a 20:80 split as shown but maybe the notes C2 and G#4 only appear twice in the testing set whereas the notes F1 and D3 appear ten times. This would be problematic as it would mean there

is more of a bias towards the C2 and G#4 in training. Then, when it comes to evaluating the model on the testing set, the model would perform poorly on the notes F1 and D3 (due to a lack of representation in the training set) but the performance on these notes would have a larger weight towards the overall testing performance as they have more representation in the testing set than other notes. To avoid this, I made sure to go through by hand and distribute the notes into testing and training sets as equally as possible to maximise model performance. Figure 2.1.3.2 shows the distribution of notes in the training set and Figure 2.1.3.3 shows the distribution of notes in the testing set. You see that most of the notes in the training set are represented by 22 audio samples, and most of the notes in the testing set are represented by 6 audio samples. There are a few outliers but not enough to affect the model performance.

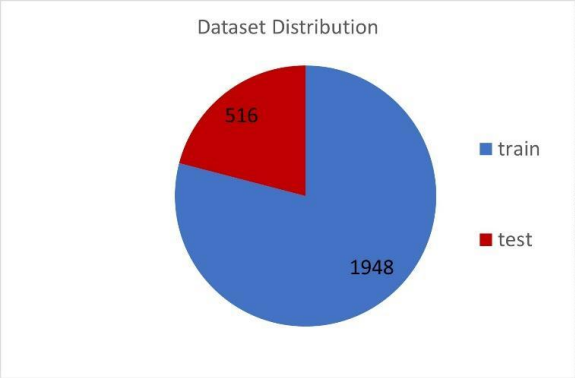

Figure 2.6: Dataset training and testing distribution

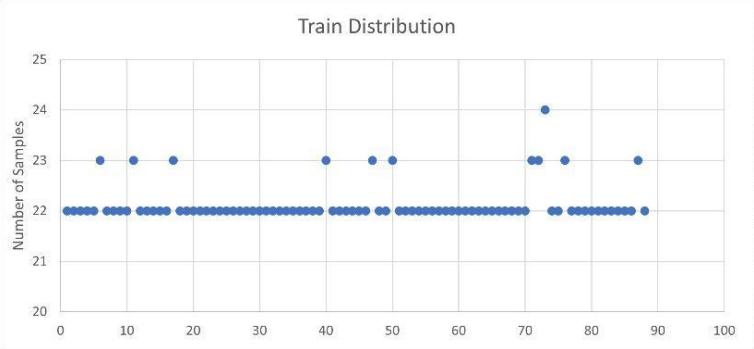

Figure 2.7: Distribution of notes in training set

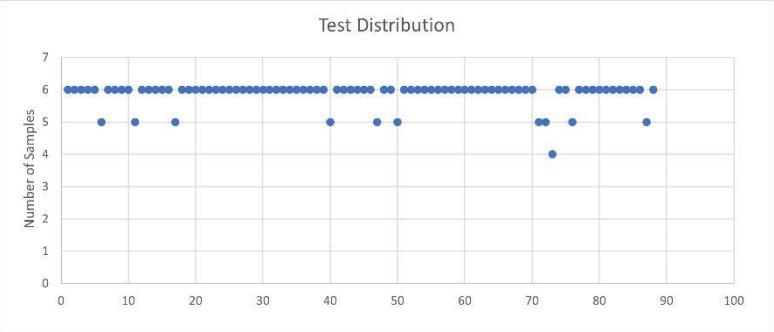

Figure 2.8: Distribution of notes in testing set

2.1.4 Data Augmentation

One final aspect of my dataset I would like to discuss is the use of data augmentation. Data augmentation is common practice in machine learning and is used to make our make our models more robust and to expand our datasets. Example augmentations on an image would be blurring, pixel dropout and rotations. For my project, I augmented my audio data by simply applying some white noise to the signals. This doubled my number of audio samples and meant my model would be robust to noisy/imperfect audio.

2.2 Software

For this project, I chose to use Python as my programming language. Python is popular due to its easier learning curve and readability (as it is a high-level language) compared to some lower-level languages like C. However, the main reason Python is the most popular language for Artificial Intelligence and Machine Learning projects is due to its extensive library ecosystem. There are a great range of libraries to facilitate deep learning, data processing, data visualization, data analysis and much more. This is perfect for a project like mine and thus was the obvious choice. Below I will discuss some of the key libraries and software I took advantage of during this project.

2.2.1 PyTorch

For my project I chose to use the PyTorch library over the alternative for deep learning which is TensorFlow. One reason for this is that PyTorch is much more intuitive than TensorFlow and doesn't have as steep of a learning curve. This was an important factor as I was relatively new to coding deep learning models and thus, I needed a library that I could get to grips with quickly so I could start implementing and improving my ideas as early into the project as possible. Another reason for choosing PyTorch was its popularity in the research community. As mentioned earlier, PyTorch is much more intuitive than TensorFlow and, as such, it is easier to start implementing new deep learning ideas. This makes it popular in the research community as researchers want to be able to implement new ideas as quickly as possible. As a result of this, the corresponding code for a lot of published papers is written using PyTorch and therefore, if I wanted to use previous work in the field for guidance/ideas, it was useful for me to implement my own model in PyTorch too. PyTorch provides us with all the key tools for creating a deep learning model. Firstly, the dataset class provides us with the blueprint

for implementing our custom dataset using overriding. As well as this, PyTorch gives us access to all the common loss functions used in deep learning such as L1, MSE, L1Smooth etc. Finally, and most importantly, PyTorch has implementations of all sorts of network layers (such as convolutional, batch normalisation and pooling) that can be used to construct a network.

2.2.2 Torchaudio

Torchaudio is a library for audio and signal processing with Pytorch. Since my project uses audio and deep learning, Torchaudio was an important library for me to use. I used Torchaudio for processes such as loading audio samples, resampling audio samples, and transforming audio samples to mel spectrograms.

2.2.3 Matplotlib

Matplotlib is a useful Python library for machine learning and artificial intelligence as it allows us to visualize data. In my case, I used Matplotlib create loss plots for my testing and training sets after training my model (see Figure 2.2.3.1 for an example loss plot). Visualizing loss values at each epoch using loss curves is very important in machine learning as it allows us to see if our model has converged at an optimum or not. If the model has converged, we should see a gradient of near 0 and this tells us which epoch to stop training at. If the model has not converged, then we know we can increase the number of epochs as we have not found the optimum yet.

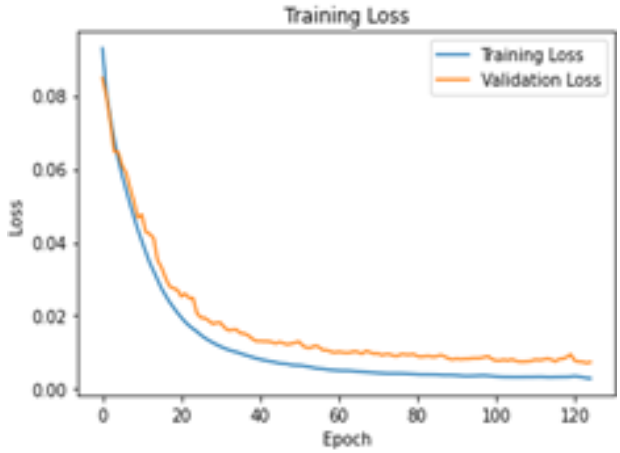

Figure 2.9: Example loss plot in which the model performs well(generlises well and doesn't overfit)

2.2.4 Google Colab

One important factor to consider when undertaking a deep learning project is the computational resources required to train a deep neural network. My personal laptop would not have been enough to train the network I needed to implement for my problem in a reasonable amount of time. This is where Google Colab Comes in. Not only does Google Colab allow you to write and execute arbitrary Python code through your browser but it also gives you access to powerful cloud GPUs to run processes on. This is especially important when it comes to deep learning as the training stage can take hours or

even days in some extreme cases with large datasets and deep networks. Having access to powerful GPUs such as the ones available through Colab allow me to drastically decrease these training times. This is important as I need to be able to see the results of my training runs as quickly as possible so I can make improvements.

2.3 Network Architecture

Next, I needed to design the network I was going to be using to solve my proposed problem. As mentioned previously, I wanted to use a convolutional encoder-decoder model for my project as this approach has not been applied to such a problem before. A convolutional encoder works by learning features and reducing the height and width of the input image whilst increasing the channels. This encodes the image into what is known as a latent representation. A latent representation is a simplified version of the original image that can still represent the important features. From here, a convolutional decoder learns to take this latent representation and reconstruct it back to the original image as accurately as possible (thus, a convolutional encoder-decoder). A decoder follows a mirrored architecture of the encoder. Figure 2.3.5.1 shows my encoder architecture and Figure 2.3.5.2 shows my decoder architecture. In the following section, I would like to discuss, in more detail, this architecture and why each of the chosen layers are used.

2.3.1 Convolutional 2D

The first layer to discuss is the Conv2D layer. This is the layer that detects features throughout the image like edges. The use of this layer is what makes my model a "convolutional" encoder-decoder. Conv2D layers use filters (or kernels) which slide across the input matrix calculating the dot product (which is how features are extracted). Different features are extracted based on what weights the filters use. We do not need to specify these weights as they are learned during the training process. One thing we do need to specify is the kernel size. Common choices for this are 5×5 and 3×3. I ultimately chose 5×5 for my optimal model (the reason for this will be discussed in the results/evaluation section). This is also why I reshape my encoder output data before feeding it into the decoder as a 5×5 kernel is initially too big for the input data to the decoder otherwise.

2.3.2 Batch Normalisation 2D

Another layer I used is the BatchNorm2D layer. Ioffe and Szegedy (2015) introduced batch normalisation as a method of combatting a phenomenon known as internal covariate shift. They explain how this is caused by the distribution of each layer's inputs changing during training as the parameters of the previous layers change. Batch normalisation solves this issue by normalising the output of the previous layer which is done by subtracting the empirical mean over the batch divided by the empirical standard deviation. As well as combatting internal covariate shift, batch normalisation has some other benefits. These include, faster convergence, providing some regularisation (batch normalisation adds a little bit of noise which consequently improves generalisation) and allowing activation functions that may not have been viable in deep networks to be usable (because the values passed into these functions are now regulated thanks to batch normalisation).

2.3.3 Leaky ReLU

In each convolutional block it is important to include an activation layer to ensure nonlinearity in the network. The activation function I chose to use was Leaky ReLU. Initially, I used ReLU. ReLU works by simply returning 0 if it receives any negative value, or x if x is a positive value. However, when using ReLU, I was achieving undesirable results. When researching ways to improve my model, I came across DiscoGAN (Kim et al., 2017). DiscoGAN is a model which aims to discover cross-domain relations. For example, take an image of a brown bag as input and output a shoe of the same colour. Even though a Generative Adversarial Network has a completely different architecture to an encoder-decoder, I took interest in this work since it aimed to take an input image of an object and output an image of a completely different object. Since my work aims to take an input image (spectrogram) and output a completely different image (sheet music), I wondered if there was anything within the DiscoGAN architecture that would benefit me. I noticed they used Leaky ReLU in their architecture instead of standard ReLU. I decided to try this in my architecture, and I found a drastic increase in results. So why might this have been? Well Leaky ReLU works by again setting positive inputs x to x, however, instead of setting all negative values to 0, negative values are multiplied by some constant which is pre-defined. In my case I used 0.2 as this is what as used in the DiscoGAN architecture. This likely led to an improvement in my model performance as Leaky ReLU helps to avoid the "dying ReLU" problem (which occurs with standard ReLU). The dying ReLU problem is when a neuron doesn't activate due to a negative bias and therefore is considered "dead". This is likely what was happening with my model and thus Leaky ReLU improved the performance and became my activation function of choice.

2.3.4 Max Pooling 2D

The next layer I used was MaxPool2D. This layer is simply used to reduce the dimensions of the input data. Max pooling does this by sliding a kernel across the input (usually of size 2x2 or 4x4) which takes the max value out of all the values covered by the kernel. For a 2x2 kernel, this reduces 4 values to 1, and for a 4x4 kernel this reduces 16 values to 1. This was required in my architecture as an encoder reduces the dimensions of its input data to encode it into a latent variable.

2.3.5 Upsample

Since I need to increase the data dimensions in the decoder, I use upsampling layers to mirror the max pooling layers. I chose to do this via the nearest neighbour method which upsamples by making the new values duplicates of their neighbour.

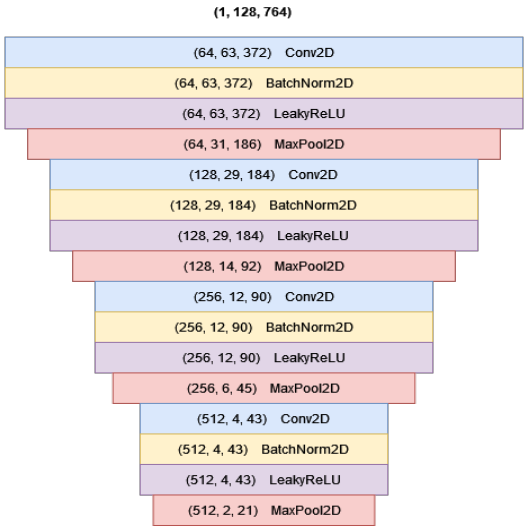

Figure 2.10: My encoder architecture

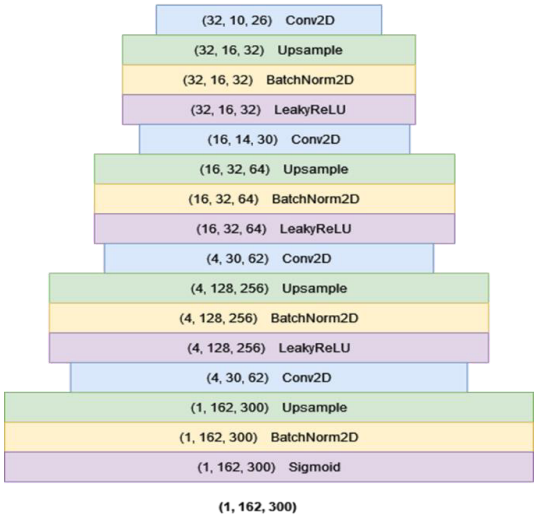

Figure 2.11: My decoder architecture

2.4 Training Implementation

Now that I had an initial architecture design, I next needed to implement the script to train the model. In this section I will discuss the key aspects of this part of the methodology. This code can be found at: https://git.cs.bham.ac.uk/projects-2022-23/rxs161 which contains different architectures, saved model weights and the training and testing scripts.

2.4.1 Dataset and DataLoader

First, I needed to implement a class to handle my custom dataset. This can be done by inheriting from and overriding the existing dataset class within PyTorch. The custom class I created has 5 attributes, an annotations attribute which is the annotations file I created for the dataset, an audio_dir attribute which is the directory of the audio files, two transformations attributes (one is to transform audio samples to mel spectrograms, and one is to transform images to tensors), and finally a target sample rate that I want all audio files to have. I then needed to implement some methods to ensure that my dataset data is in the correct format when being fed into the network. The first method I implemented was a resample_if_necessary method. This method simply takes an input audio signal and resamples it to the target sample rate if needed. The next method I implemented was a mix_down_if_necessary method. This method mixes an input signal down to mono (1 channel) if needed (which is important as the model will expect all the data to be 1 channel). The final two methods get_audio_sample_path and get_audio_sample_label get the path of the current audio sample and its image label respectively. All these methods are used by the __getitem__ method which loads and returns an audio sample and its corresponding image label from the dataset in the correct format. Now that I had a custom dataset class, I could instantiate a train dataset and test dataset object (the difference being that the training instance uses the training annotations, and the testing instance uses the testing annotations). I separated the training and testing data into two separate dataset instances as this made it much easier when it came to implementing the training and testing loops. Once we have our dataset objects,

we can use PyTorch's DataLoader iterable. DataLoader retrieves samples from our dataset object in batches of a size which is specified by us (I will discuss batch sizes further in a later section). We can then iterate through the DataLoader to retrieve batches of our data to be used in training or testing.

2.4.2 Training Loop

Now that I had a custom dataset class, I could start to implement the code to train my model. The training part is split into two functions, one function called train_one_epoch which trains the model for one run through of the dataset, and another function called train which simply loops the train_one_epoch function n times (where n is the number of epochs). Within the train_one_epoch function, I loop through the training DataLoader to load the batches of samples from the training set. I then feed the data batch by batch into the model and get predictions from the model. A loss value is then computed by using some loss function to compare the model predictions to their target images (I will discuss choice of loss function in a later section). Based on this loss value I then backpropagate and perform a single optimization step to update the model parameters (choice of optimizer will also be discussed in a later section). During this process, I also collect a running loss total which is then divided by the number of DataLoader iterations to get an average loss value across the batches. This entire process is undertaken during one epoch. If the model has a quality dataset, good architecture and optimal hyperparameters, we should see the average loss decrease rapidly from epoch to epoch until the rate at which the loss decreases starts to slow - which tells us that the model is starting to converge to an optimum. We plot a loss curve to help us visualize this and spot at what epoch the model is converging. Refer to Figure 2.2.3.1 to see an example of this visualization.

2.4.3 Testing Loop

The testing loop follows the same process as the training loop but with a couple of differences. Firstly, the testing loop uses the testing DataLoader as we want to use our testing set to evaluate the model performance and not our training set. This is because, the training performance doesn't represent the model's ability to generalise and perform well on "unseen" data as the model has updated its parameters based on this data. To ensure the testing set remains "unseen" for future epochs, we exclude the backpropagation and optimizer steps taken in the training loop. This stops the model updating its parameters based on the loss values calculated on the testing set. We still calculate a running loss across batches and calculate an average loss for the testing set. If we then plot the testing loss values and training loss values on the same graph, we can get an idea of how the model is performing. If the model is performing well, the testing plot should follow a similar shape to the training plot but with values slightly above the training loss values (again refer back to Figure 2.2.3.1 to see an example of this). If the model is performing poorly, the model would have a larger discrepancy between the training and testing loss values. Finally, the model could also be overfitting. This occurs when the testing loss starts to increase as the training loss is still increasing (refer to Figure 2.1.1.1 to see an example of this).

2.4.4 Loss Functions

As mentioned previously, to update model parameters, we must first calculate the loss/error between the model predictions and the ground truth. There are multiple loss functions that can be used to do this. A loss function is just some function that takes a model prediction and the corresponding target (what we want the model to predict) and outputs a loss value based on some calculations that aim to

evaluate the similarity of the prediction and target. For my project I considered three loss functions – Mean Squared Error (MSE), L1 Smooth, and Structural Similarity Index Measure (SSIM). MSE works by calculating the difference between the predicted output and the target and then squaring this difference:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{1}$$

$$(Where\ n = batch\ size,\ Y_i = i^{th}\ prediction\ in\ batch\ and\ \hat{Y} = i^{th}\ target\ in\ batch)$$

The aim of MSE is to punish large differences between predictions and targets more. For example, if $Y_i - \hat{Y}_i$ is 2, then, with MSE we get a value of 4. However if $Y_i - \hat{Y}_i$ is 5 then we get a value of 25. A downside of this is that one really poor prediction can have a big influence on the average loss value for the batch. L1 Smooth aims to combine MSE and standard L1 (which works the same as MSE except it doesn't square the differences). It does this by setting a value beta which determines whether we use L1 loss or MSE. If the difference $Y_i - \hat{Y}_i$ is below beta, we use MSE, but if it is greater than beta, we use L1. This stops large loss values having a big influence on the average loss but then still allows smaller loss values to take advantage of the benefits of MSE. Finally, SSIM is an evaluation metric used to measure the similarity between two images. Instead of just looking at the difference between pixel values like MSE or L1, SSIM looks at 3 key features when determining image similarity: luminance, contrast, and structure. The SSIM expression is as follows:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{2}$$

The motivation behind using an SSIM-based loss is that MSE and L1 Smooth just reduce loss by reducing the pixel differences even though a low total pixel difference doesn't necessarily mean an accurate reconstruction. This is why, in theory, a loss function that decreases loss in proportion to increasing SSIM, should force the model to learn to predict similar images to the target images. As SSIM by default is not a loss function but an evaluation metric, I needed a custom implementation of an SSIM-based loss function. I found a GitHub repository (psyrocloud, 2020) which did just this. I just had to make a slight alteration to the code to make it work for single channel images. I will discuss the loss function that I chose for my optimal model in the results/evaluation section.

2.4.5 Optimizer

An optimizer is the algorithm that updates our model parameters after the model has backpropagated. For my project, I chose to use the most popular optimizer – the Adam optimizer (Kingma and Ba, 2014). The reason for the popularity of the Adam optimizer is that it combines the advantages of multiple optimizers (AdaGrad, RMSProp and Gradient Descent with Momentum). AdaGrad aims to update faster along sparse features (features where lots of the values are 0) as normal gradient descent will update slowly on sparse features even if they are very relevant to the problem. RMSProp is a variant of AdaGrad which aims to stop the learning rate decaying too aggressively and ultimately leading to very small updates on dense features. This is done by introducing a hyperparameter, beta, which slows the growth of the value of the accumulated gradient norm square (this is the value that causes the learning rate to decay in AdaGrad). Momentum is a concept that allows the model to make updates quicker if it is confident it is going in the correct direction. It does this by constantly updating a velocity value which is based on accumulating previous gradients. As such, the model gives more

weight to recent parameter updates. This makes sense since recent updates are more useful to us than initial parameter updates. Adam works by combining the idea of momentum with the idea of aiming to update faster on sparse features without decaying the learning rate for dense features too quickly. This is what makes it the popular choice for me and others.

2.4.6 Hyperparameters

Finally, before training a model, we need to decide on the hyperparameters. These include batch size, learning rate, and epochs. The batch size decides how many samples are processed before updating the model parameters. A batch size of 1 is what is known as stochastic gradient descent, a batch size greater than 1 but less than the size of the dataset is known as mini-batch gradient descent, and a batch size equal to the size of the training set is what is known as gradient descent. Since I was using a batch normalisation, a batch size of 1 was not a viable option. Also, a larger batch size gives a more accurate representation of the current loss value of the model meaning we get more accurate parameter updates. A downside of large batch sizes is it means more data loaded onto your GPU and thus a large batch size may not always be feasible. Another consideration is that a batch size too large doesn't allow for many parameter updates per epoch. If you have 1000 training samples and use batch sizes of 100, you only get 10 parameter updates over the course of an epoch. It may be better to scale your batch size with the size of your dataset. For my dataset of 1948 training samples, I found a batch size of 32 to give the optimal results. The learning rate determines how quickly the model learns. A large learning rate causes bigger parameter updates, and a smaller learning rate causes smaller parameter updates. If the learning rate is too large for the problem, then the model will converge too quickly to a suboptimal solution. If the learning rate is too small, the model may get stuck and never converge at all. There is no exact science to determining what learning rate to use, you can only experiment and make observations. For example, an unstable loss curve would suggest the need to decrease the learning rate as the model is constantly making major parameter updates. For my project, I tested learning rates such as 0.1, 0.01, 0.00001 but I found 0.001 to be the optimal learning rate. Finally, epochs determine how many times your model goes through your whole dataset before concluding training. Like learning rate, this is a value where you can only really decide on the best value by observing the behaviour of your model using loss curves. If your model concludes training before convergence (loss curves have flattened), it is good practice to increase your number of epochs to allow for convergence. However, it is important to not set your number of epochs too high as this could cause the model to start overfitting.

3 Results

When evaluating the results of this project, I made sure to conduct multiple tests with different conditions to find the optimal model. I first needed to find the best loss function, and thus kept a constant architecture and constant hyperparameters whilst changing the loss function and evaluating the results. I then needed to find the optimal kernel sizes for my convolutional layers since 3×3 and 5×5 were the standard, but I didn't know which were best for my problem. I fixed my loss function to the best one and trained the model with different kernel sizes to evaluate which was the best. After a training run, the current weights of the model are saved. To test the model, I load the model with those saved weights and feed the testing set into the model. I get a PSNR value for each of the predictions and calculate an average.

3.1 Evaluation Metrics

Before showing the results I got, I would first like to discuss the evaluation metrics that can be used for such a problem. I have previously mentioned SSIM as a quantitative evaluation metric, however, I chose not to use this as it wouldn't make sense to evaluate the SSIM-based loss function with SSIM as an SSIM-based loss function would obviously see the best results in this case. An unbiased alternative to this is what is known as Peak Signal-to-Noise Ratio (PSNR). This works by calculating the ratio of the maximum value of the signal to the noise that distorts it. PSNR is represented by the following expression:

$$PSNR = 20 \cdot log_{10}(MAX_I) - 10 \cdot log_{10}(MSE) \tag{3}$$

A higher PSNR value represents a better-quality image. A qualitative way to evaluate the results would be through a simple eye test. If the sheet music is readable by a human, then this can be considered a successful reconstruction. We can combine this qualitative approach with the quantitative one to work out a PSNR value that produce images that are quality enough to be human readable. Through analysing my results, I found that a value of around 20 PSNR produced a reasonable quality human readable image. Figure 3.1.1 shows an example 20 PSNR reconstruction. This is worth considering as I evaluate my results.

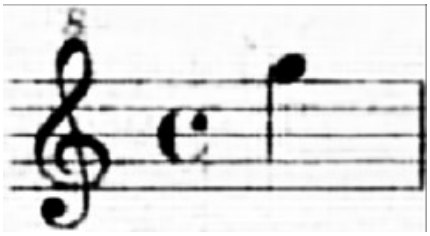

Figure 3.1: 20 PSNR Sheet music image

3.2 Loss Functions

Table 1 shows the average PSNR of the testing set with each loss function. Figure 3.2.1 shows an A5 note produced with L1 Smooth, Figure 3.2.2 shows the same note produced with MSE, and finally, Figure 3.2.3 shows the same note produced with SSIM. (Figure 3.2.4 shows the ground truth image

for this note). From these results we can see that the L1 Smooth loss function performs the best. The possible reasons for this are that, firstly, L1 Smooth combines the advantages of L1 and MSE loss and thus it performs better than using MSE alone. Secondly, L1 Smooth likely performs better than the SSIM-based loss as SSIM is more concerned with structural similarities than individual pixel intensities. This can lead to poorer reconstructions of smaller details. For example, in Figure 3.2.3 the model has struggled to give a detailed reconstruction of the octave symbol on top of the clef, whereas, in Figure 3.2.1, we can see a clear 8 shape.

Table 1: Average PSNR of model trained with each loss function

Loss Function	Avg. Testing PSNR
SSIM	17.98
MSE	18.61
L1 Smooth	19.14

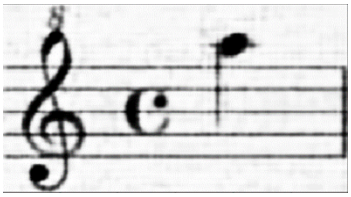

Figure 3.2: A5 note predicted by model with L1 Smooth Loss

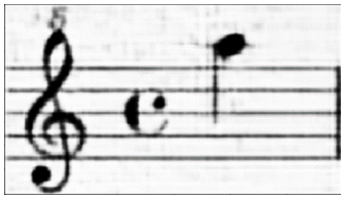

Figure 3.3: A5 note predicted by model with MSE loss

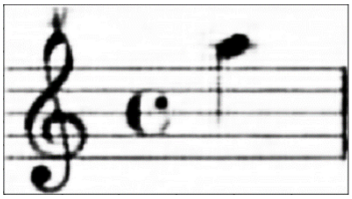

Figure 3.4: A5 note predicted by model with SSIM-based loss

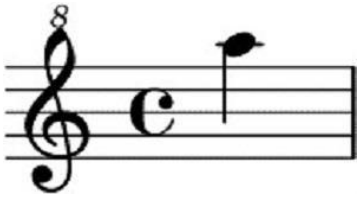

Figure 3.5: Ground truth A5 sheet music image

3.3 Kernel Sizes

Table 2 shows the average PSNR of the testing set with different convolutional kernel sizes. Figure 3.3.1 shows an A5 note produced with 5×5 kernels, Figure 3.3.2 shows an A5 note produced with mixed kernels, and Figure 3.3.3 shows an A5 note produced with 3×3 Kernels. (Refer back to Figure 3.2.4 for ground truth image). We can see that using 5×5 kernels yields the best results, with 3×3 yielding the worst. Firstly, this could be because 5×5 kernels require more parameters and thus gives the model more expressive power. Alternatively, it could be that a 3×3 kernel retains more details (9 pixels become 1 instead of 25) and thus is more likely to retain noise.

Table 2: Average PSNR of model trained with different convolutional kernel dimensions

Kernel Size	Avg. Testing PSNR
3x3	15.23
5x5	19.14
Mixed (3x3 and 5x5)	16.58

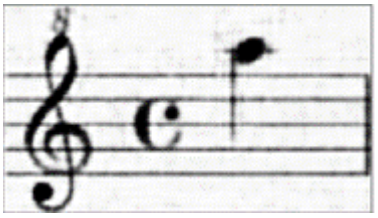

Figure 3.6: A5 note predicted by model using 5x5 convolutional kernels

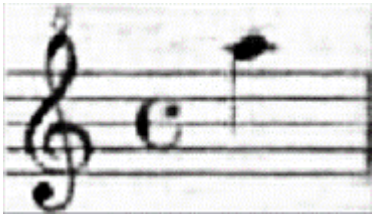

Figure 3.7: A5 note predicted by model using mixed convolutional kernels

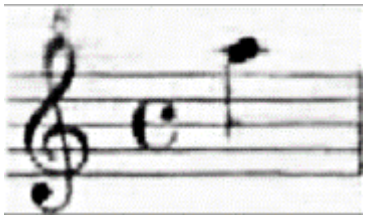

Figure 3.8: A5 note predicted by model using 3x3 convolutional kernels

3.4 Loss Plots

Figure 3.4.1 shows the loss plot for the model which was trained using L1 Smooth loss, Figure 3.4.2 shows the loss plot for the model which was trained using MSE loss, and Figure 3.4.3 shows the loss plot for the model which was trained using SSIM-based loss. We can see that, in all the training runs, no overfitting occurred, and the loss curves follow the shapes we expect. This means that the model has learnt meaningful latent representations of the audio samples that allow them to be decoded into sheet music as well as allowing for generalisation. Also, it tells us that the testing set is representative of the whole dataset.

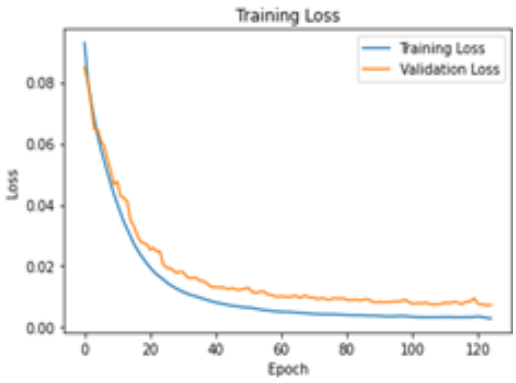

Figure 3.9: L1 Smooth loss plot

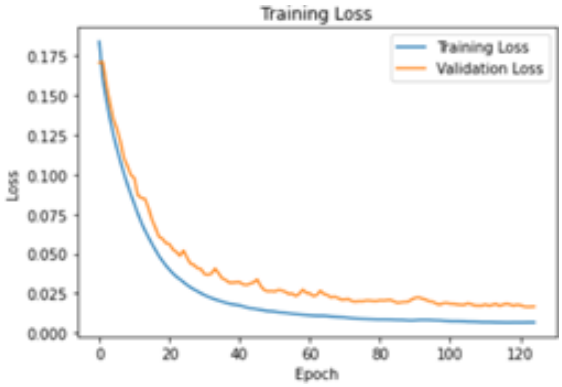

Figure 3.10: MSE loss plot

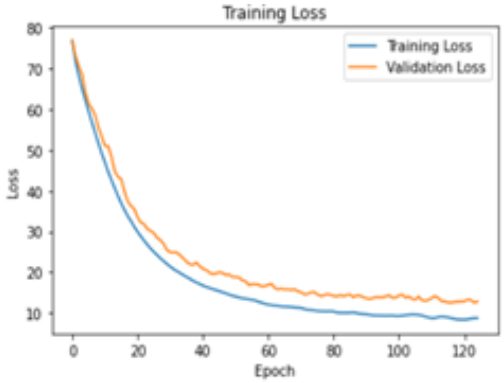

Figure 3.11: SSIM-based loss plot

3.5 Evaluation of Results

From the results presented, we can see that the optimal loss function to use for this problem was L1 Smooth, and the optimal convolutional kernel dimensions were 5×5. This combination produced the best average PSNR on the test set (19.14). This is very close to the target average PSNR of 20 which I mentioned earlier. We can also see, from the optimal example presented in Figure 3.3.1, the model is able to produce human readable reconstructions. However, upon examining all the predictions from the model on the test set, I started to notice a trend. It appeared the model performed worse on the lower and higher frequency notes (the keys on the far left and far right of the piano). Figure 3.5.1 shows an example poor reconstruction of a high frequency note (the top image) and a low frequency note (the bottom image). To better visualize this, I decided to feed the test set into the model again, but instead of taking an average over the whole test set, I took the average PSNR of each individual note and created a scatter plot. Figure 3.5.2 shows each point on the scatter plot lined up with its corresponding piano key. We can see the highest average PSNR at about 23 PSNR in the right of the middle section of the piano. The lowest average appears to be around 12/13 PSNR towards the left side of the piano. However, the spread-out nature of this plot makes it hard to visualize the trend of the data, therefore I made another plot with the data points more densely packed. Now, from Figure 3.5.3, the trend is clear. It seems the model is performing its worst on the low frequency notes but then performing well towards the middle of the piano, with averages above 20 PSNR, before a slight drop off as it reaches the high frequency notes (although the PSNR of the high frequency notes doesn't quite drop to the levels of the low frequency notes). This backs up the conclusions I made with my qualitative observations of the dataset. It is also interesting to consider how good the average PSNR across the whole training set is considering how low some of the average values are for some of the notes. If I could increase these averages in the poor performing areas of the piano, then the overall average would be considerably more successful. So why does the model exhibit such behaviour? Well due to time restrictions and the scope of the project, I haven't been able to fully investigate, however, I do have some ideas. Since the model learns from the Mel spectrograms of the audio files, I decided to look at the spectrograms for different frequencies of notes. Figure 3.5.4 shows the spectrograms of two low frequency notes with noise applied to the signal and Figure 3.5.5 shows the spectrograms of two high frequency notes with noise applied signal. If we compare the spectrograms, even though there is noise present, the high frequency spectrograms have clear unique features present on the left side of the

image. In contrast, it is much more difficult to spot any clear unique features on the spectrograms of the low notes. I have theorised that this could maybe be the cause of the poor performance exhibited by the model on the low notes. If the model struggles to extract features from the noisy versions of the low notes, then this would make half the training samples for the low notes basically redundant (since half the data is audio without noise and the other half is audio with noise). This of course is just a theory and, due to time restrictions, I haven't had time to collect evidence to back this theory. I will discuss in the future works section my ideas for how to investigate this.

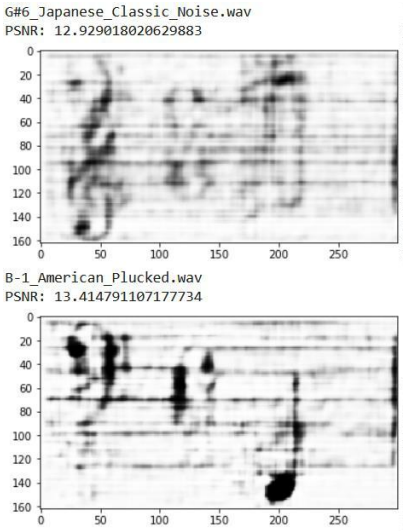

Figure 3.12: Poor reconstructions of low and high notes

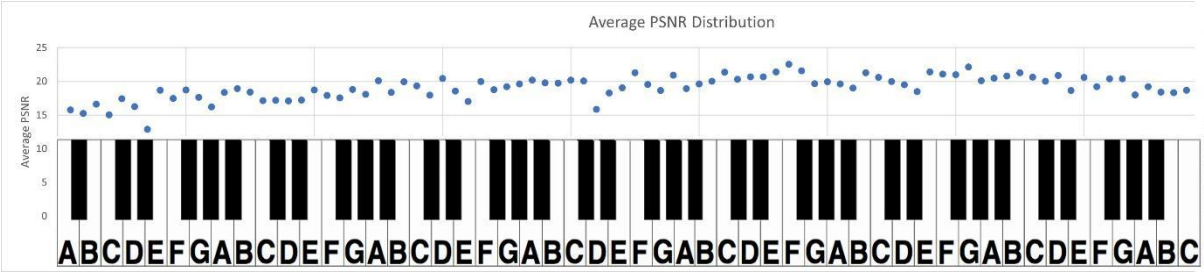

Figure 3.13: Average PSNR of each note

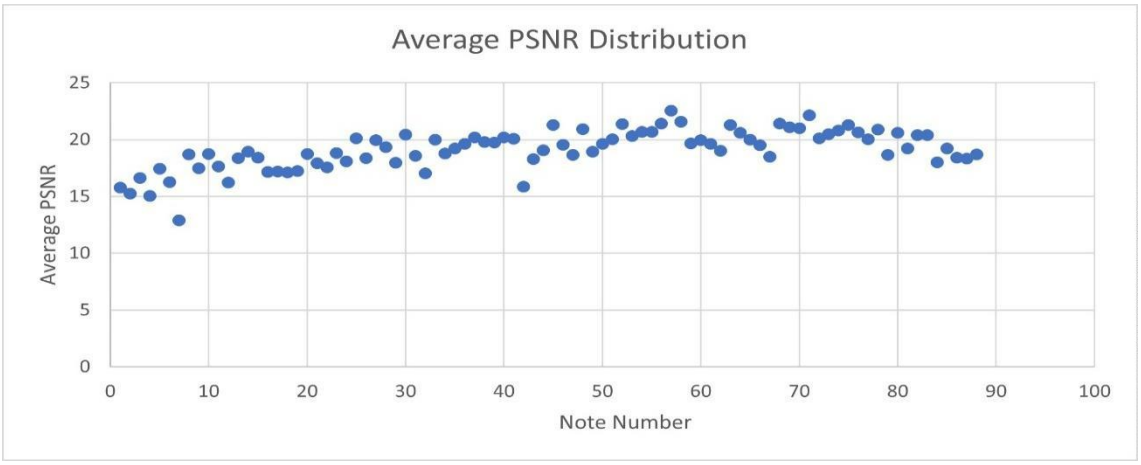

Figure 3.14: More densley packed scatter plot to show trend of average PSNR of each note

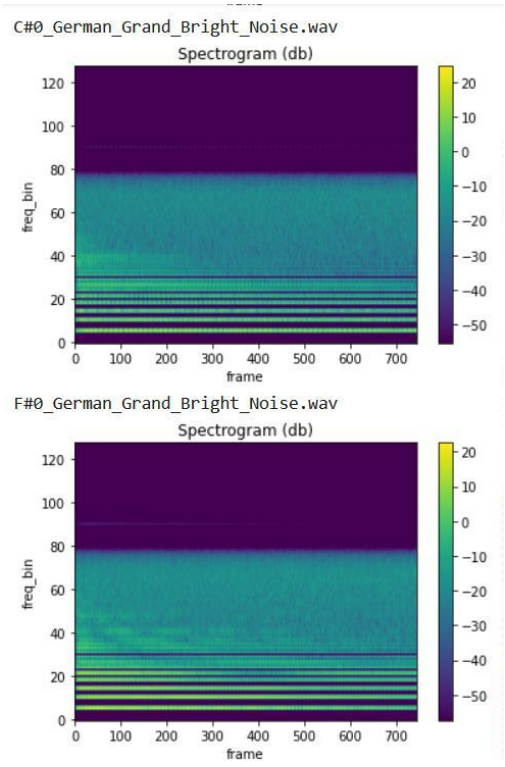

Figure 3.15: Mel spectrograms of low notes with noise

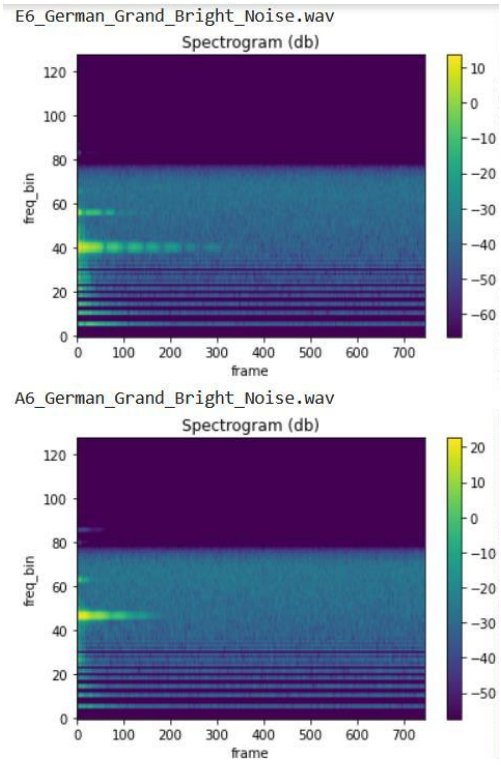

Figure 3.16: Mel spectrograms of high notes with noise

4 Conclusion

4.1 Discussion

4.1.1 Project Success

After evaluating the results, I would conclude that the project has been a success. I have showed that we can take a generative approach to audio-to-sheet music transcription using a convolutional encoder-decoder. The model was able to learn unique features from the mel spectrograms of the audio samples and generate images with an average PSNR of 19.14. This is more impressive given the fact that, as discussed earlier, the lower notes had very low PSNR scores which would have impacted this overall average value greatly. Despite the success, there are still lots of avenues to follow after this project to achieve even more impressive goals. I will discuss these in the future work section.

4.1.2 Review of Software Used

Firstly, for the dataset creation part of the project, I thought the software used was adequate. Arturia provided me with enough different piano sounds that I was able to collect enough data samples to get some meaningful results. For future works, other software could be explored but I wouldn't make this a priority. Similarly, SoundBridge provided me with everything I needed for my dataset creation. Since SoundBridge is free, there may be paid alternatives that could provide a better experience and speed up the process, but I was able to achieve everything I wanted to with SoundBridge. Python proved a good choice of programming language as I made use of a range of its libraries during the project. I was able to pick PyTorch up quite quickly where I may have struggled with TensforFlow. Its intuitiveness really aided me in fast implementations and debugging. Coupling this with Torchaudio to apply the required transformations to my dataset was a seamless process. Then, visualizing results using Matplotlib was of great help during the project to evaluate the current performance of the model and workout the best course of action to take (i.e. using loss plots). Finally, Google Colab was very useful to me for speeding up training times as well as running sections of code using the cell-based IDE to isolate errors and build up my training script step-by-step.

4.2 Future Work

4.2.1 Low and High Notes

The first step when considering future work would be to improve the reconstructions of low and high frequency notes. Due to my aforementioned theory, when applying noise to low notes, I would maybe consider applying less noise in comparison to notes around the middle of the piano. It may not be feasible to apply the same level of noise to all frequencies due to the reasons I presented earlier. I could do this and evaluate the results again to see if any improvements are made.

4.2.2 Sheet to Audio

Another future work would be to try reverse the model and go from sheet music images to audio. This would be quite easy to implement in terms of the model architecture, however, the more difficult part is with the data collection. For this problem, we would need lots of sheet music image samples and the only way to do this would be to take the images used in the ground truth for this project and apply lots and lots of augmentations like blurring, rotation, pixel dropout etc. It was much easier to collect

more audio data as I could just collect 88 notes on a new instrument. The same logic would apply to this problem though. We generate a spectrogram from a sheet music image and then convert this spectrogram to an audio file.

4.2.3 Longer Audio

Another avenue to go down would be to transcribe longer audio. The logical next step from here would be to do this with scales (Figure 4.2.4.1 shows a C major scale). One way to do this would be to maybe apply a sliding window approach. The model could slide across the spectrogram and transcribe the sheet music image note by note. The model would then stitch these segments together to create the scale.

4.2.4 Dataset and Conference Submission

Finally, due to this project's uniqueness and possible usefulness, I intend to submit this work to a leading conference in Machine Learning (with discussion and confirmation from my supervisor) and release the dataset to the community to facilitate other works and allow others to refine and expand the dataset to produce even better models which build on the success achieved by this project.

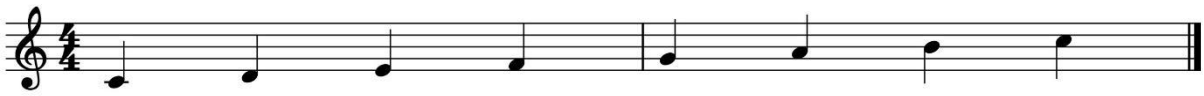

Figure 4.1: C major scale