Machine Learning Applications for Audio Processing

Noel Gomariz Kühne

***Abstract*—The rapid development of deep learning techniques has lead to a huge expansion in its domain of application, from image recognition and computer vision to stock market prediction and natural science applications. Audio processing and generation is not far behind in development and there are plenty of tasks, such as speech recognition and generation, where deep learning has proven to be invaluable for its performance. More over, the study of audio oriented machine learning is of great interest because it can be directly related to the study of time-series and sequential data in general. In this document we will mainly explore two kind of networks, autoencoders and recurrent networks, applied to the study of audio related tasks, specifically audio denoising and melody generation.**

***Keywords*—** sample rate, frequency domain, short-time Fourier transform, white noise, spectrogram, deep convolutional autoencoder, variational autoencoder, long short term memory.

1. INTRODUCTION

Different types of deep learning approaches can be followed when aiming to solve problems regarding audio data. For example, a standard feed forward neural network (FFNN) could be trained using each audio sample as a feature. Nevertheless, audio recordings consist of thousands of samples per second, and the number of parameters needed to train such a FFNN would be too large to obtain good results with regular computation resources. Other structures as recurrent neural networks (RNNs) are more efficient and are actually used in applications such as speech or music generation.

When dealing with noise-corrupted data, one of the most popular strategies is to opt for autoencoders. This architectures can reduce the amount of features needed to process a data sample so that it is easier to train a model. From this start point, other structures such as deep convolutional autoencoders (DCAE) or variational autoencoders (VAE) have been developed intending to maximize the performance of the task in hand. Both DCAEs and VAEs will employed to attempt the task of audio denoising, resulting in a reconstruction of a clean recording from a corrupted one.

Melody generation, in the other hand, is the perfect challenge for a RNN. Music is characterized by the relevance of its time structure: for example, a musical phrase can be repeated over and over in a piece, with slight variations, yielding in a rich, meaningful and connected composition. The key feature that makes RNNs the perfect option to learn how to generate music is that they are the only deep learning structure which count with an internal memory. We will study the performance of long short term memory networks (LSTM), training them with sequences of melodies to then produce a new composition.

This deep learning architectures will be explored in Section 2, emphasizing the intuition needed to know why they work for a specific problem. The data sets to study will be presented in Section 3, together with the preprocessing needed for each problem. Lastly, in Section 4 we will explain the training process and the results obtained.

2. DEEP LEARNING ARCHITECTURES FOR AUDIO PROCESSING

A. *Autoencoders and variational autoencoders*

An autoencoder (AE) is a deep learning structure that has the same number of input and output nodes. It consists of two parts: the encoder, in charge of compressing the input into a lower (or higher, if desired) dimensional space, called the latent space; and the decoder, responsible of returning an point from the latent space to its original shape. In essence, an autoencoder is capable of learning the most relevant features of an input to then reconstruct it from the latent space. As an analogue, we have the principal component analysis algorithm, which performs a similar job as the encoder. Nevertheless, the advantage of autoencoders is that they are capable of learning *non-linear features* of the input data.

On the other hand, a deep convolutional autoencoder (DCAE) is an autoencoder whose encoder and decoder are composed by several blocks of convolutional layers, being the decoder a mirror reflection of the encoder (Figure 1). A convolutional layer performs a set of operations over an input matrix $\mathbf{x}$: first, a convolution itself, $(\mathbf{x} \star \mathbf{g})[i, j] = \sum_{m,n} \mathbf{x}[m, n]\mathbf{g}[i - m, j - n]$; then, a point-wise activation function can be applied to the result of the convolution; lastly, a pooling stage further reduces the dimension of the convolution result.

The implementation of the DCAE architecture is implicit in the class **Autoencoder**[2], which wraps the functionalities of Keras. The arguments of an Autoencoder object are: the input shape, a tuple containing the image width, height and number of channels; the parameters related to each convolutional layer, being those the number of filters, the

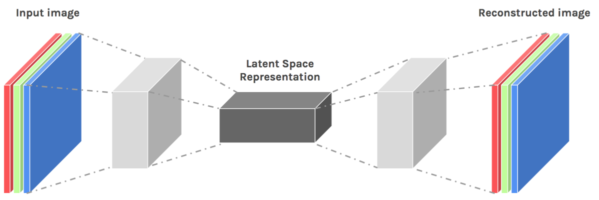

Figure 1: Deep convolutional autoencoder unfolded into input image, convolutional layers, latent space and reconstructed output.

size of the kernel and the stride; and lastly the dimension of the latent space. With this parameters, the encoder is built by adding each convolutional block. A block is made of:

- Convolutional layer, where the kernels and the pooling are applied.
- ReLU activation layer.
- Batch Normalization layer, which scales the data inside each mini-batch.

When all the blocks are added, the last step to finish the encoder is to add the *bottleneck*. This is the term given to the layers leading to the latent representation. First, the data is flattened, and then connected to a Dense layer which dimension is that of the latent space.

The decoder is a mirror reflection of the encoder: first, a Dense layer connected to the latent representation whose output is reshaped into the shape it had before the Flatten layer; next, the same convolutional blocks as in the encoder but in reversed order are added, with the exception that in the last block we introduce a *Sigmoid* activation, which leads to the reconstruction of the sample fed into the encoder.

DCAEs are interesting because of the variability in the tasks they can perform. Our hope is that the model can learn to distinguish the most relevant features from an image and separate them from random noise, in order to eliminate white noise from signals. Nevertheless, DCAEs have their limitations: when a DCAE has compressed the data through the convolutional layers, it stores the output of the encoder in the latent space in the form of a single point. Also, AEs usually take as loss function the Mean Squared Error (MSE). This two characteristics make the points in the latent space be quite spread out and not symmetrically distributed around the origin. Thus, passing a new sample through the encoder could lead to a point in the latent space which does not relate to the training data. To illustrate this, in Figure 2 we reproduced the latent representation of ~ 600 data points sampled from a DCAE with a two-dimensional latent space trained using the MNIST data set, and as we can see, the points seem to be uniformly distributed.

An alternative to DCAEs are variational autoencoders (VAEs), which can be more reliable when it comes to produce new samples from the latent space of a trained model. VAEs add two modifications to vanilla autoencoders. First, a new way of encoding the data points in the latent space

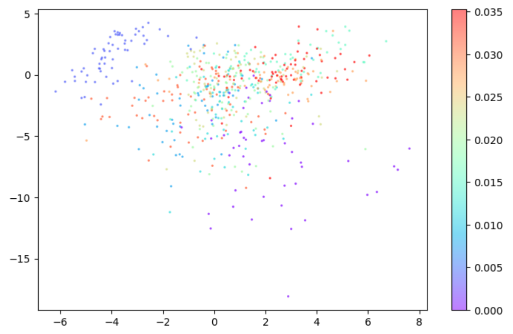

Figure 2: Points in the latent space of an autoencoder trained with the MNIST data set.

is introduced: now, each sample is represented in the latent space not by a point, but by a multivariate gaussian distribution. With this variation, we reduce the probability of selecting a meaningless point from the latent space. The other implementation that distinguishes a variational autoencoder deals with the loss function. To the MSE we add the so called Kullback-Leibler divergence:

$$D_{KL} = \frac{1}{2}\sum_{i}(1 + log(\sigma_i^2) - \mu_i^2 - \sigma_i^2) \tag{1}$$

where μ and σ are the mean vector and variance of the probability distribution. This term penalizes observations where the mean vector and variance differ from the standard multivariate normal distribution, which is centered in the origin of the latent space. The last modification to the loss function is the so called reconstruction loss weight, a multiplicative factor of the MSE that stands for the relative significance of the reconstruction error and the Kullback-Leibler divergence. With all this adjustments, the loss function of a VAE is

$$C = \alpha \cdot MSE + D_{KL} \tag{2}$$

The implementation of a **VAE**[3] object follows the strategy of the Autoencoder implementation. The steps to follow are the same until the last convolutional block is added. Then, instead of a bottleneck leading to the latent space, we have two Dense layers: one for the mean vector of the distribution and the other for the variance. With the output of those layers we sample a point from the corresponding distribution.

B. *Long short term memory networks*

RNNs are a kind of deep learning network that is specially designed to deal with sequential data. In contrast with a simple FFNN, which only process the current input, a RNN keeps track of the current input and the recent past. The internal functioning of a RNN can be sketched as a recurrent relation: the state of the system at each time step takes as input the previous state of the network and the current input[4]: $h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$ where θ are the **shared parameters** of the network. This means that there are 3 kind of connections in the RNN: a connection between the hidden states, denoted as $\mathbf{W}$, a connection between the input and the hidden state, $\mathbf{U}$ and a connection between the hidden state and the output produced at each time step, $\mathbf{V}$. This connections are weight matrices shared through all the iterations, characteristic that reduces the number of parameters of the network. Assuming a task where the output is distributed in categories, the forward propagation in a RNN is described by the equations:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} \tag{3}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}) \tag{4}$$
$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \tag{5}$$
$$\hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)}) \tag{6}$$

where $\mathbf{b}$ and $\mathbf{c}$ are the biases assigned to the input and output connections respectively, and we have taken as activation function the hyperbolic tangent.

The fact that RNNs can keep track of the past inputs of a sequence is not enough for them to be effective to be trained with time series, as RNNs present two major inconveniences: exploding and vanishing gradients. Vanishing gradients is specially difficult to deal with, and it can lead to the sudden stop of the training. LSTM networks were developed in order to address this problem. They introduce the idea of gated cells, which basically decide if a piece of information is stored or deleted[5]. A LSTM unit features three gates: input, output and forget. The algorithm in charge of optimizing the weights associated to the gated cells as well as to the connections between hidden states, inputs and outputs is called back propagation through time[6].

3. Preprocessing audio data

A. *Converting audio to images*

As shown in figure 1, AEs accept two-dimensional image-like inputs. However, audio recordings are one-dimensional time series. Thus, to train such architecture with audio recordings, we need a unique way to relate a signal to an image. This is achieved thanks to the short time Fourier transform (STFT) algorithm (see Section 6). Through the STFT, we can obtain a two-dimensional map of the intensity of a signal in relation to the frequency spectrum and time, called a *spectrogram*. As an example, Figure 3 shows the spectrogram of a middle C played on a piano. Higher amplitudes are represented by purple and white regions, and as we see, the higher amplitude frequencies are distributed around the base note and its harmonics (integer multiples of the base frequency).

The data set we chose to train both the DCAE and the VAE is the free spoken digits data set (FSDD), which consists of 3000 audios of spoken digits from 0 to 9 by 6 different people. We also created a new version of the data set, but with added white noise composed with sound waves in the range of frequencies between 0 and 20000 Hz, corresponding to the human hearing spectrum (Figure 4).

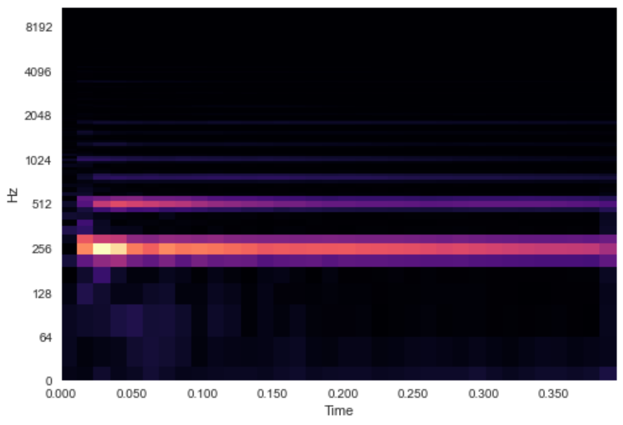

Figure 3: Spectrogram of a single note. Highly present frequencies point the base note and its harmonics.

To preprocess the audio files and convert them to spectrograms, we made use of the functionalities of the library Librosa, specially designed to deal with signal-like data. The steps followed during the preprocessing stage were:

1) Obtain the STFT of both the original recordings and their corrupted versions. We chose to use a frame size and hop length of 512 and 256 pixels respectively. Also, in order to have the same shape for all images, we set up a fixed duration to load from the signals, equal to 0.55 seconds. If a recording exceeds the duration, it is trimmed down. If in the other hand it does not reach it, we apply padding to the signal, meaning that we fill the missing samples with zero amplitude.
2) The result from STFT is a complex-valued matrix. Thus, it is necessary to take the module of each entry to obtain a map of the intensity.
3) Human brain perceives sound in a logarithmic way, so it is common to convert spectrograms from power to dB.
4) Scale all the spectrograms, mapping the values between 0 and 1. The minimum and maximum values of each spectrogram were stored so that we could apply the inverse transformation to recover the signal.
5) Randomly split the data into train and test. We kept a 20% of the spectrograms for testing purposes.

B. *Preprocessing songs*

The data used to train a LSTM network to generate melody will be obtained from the *KernScores database* (http://kern.ccarh.org/). This database is a library of songs encoded in **kern representation[7], a system that allows to represent pitch and duration, as well as other music score-related information. Since we are interested in producing only melody, we are going to train our

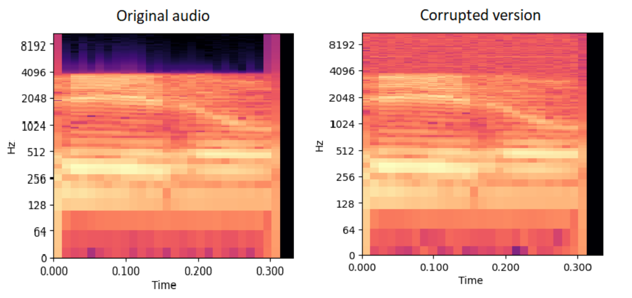

Figure 4: Logarithmic spectrograms of a sample from the FSDD set and its corrupted version with added white noise. The main features of the original image are distinguishable since the amplitude of the noise is not too high.

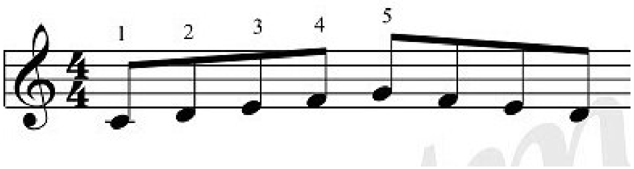

Figure 5: Simple music score to show the chosen encoding.

LSTM model using the monophonic songs section of the database, specifically the files corresponding to compositions from Franz Schubert and Stephen Foster. The main goal of the preprocessing stage is to cast a set of songs into a single integer-like series that contains all the notes and rests included in the set of songs.

The songs we are going to process are all in different keys, meaning that all the 12 notes of the chromatic scale are present in the set. Nevertheless, we only want to train the model with notes corresponding to a certain configuration of accidentals, so that the melody produced by the network belongs either to a major or minor key. Thus, we need to transpose all the songs to two keys, one being the minor relative of the other. For simplicity, we will choose C Major and A Minor, as they don't use any accidentals.

With all the songs transposed to C Major and A Minor, we need to encode the pitch and duration of a note in such a way that it can be fed into a network. We will adopt the midi convention about pitch, where each note is assigned an integer number. For example, the middle C of a piano is characterized by the number 60. For simplicity, to define the duration of a note we will just consider whole, half, quarter, triplet, eighth and sixteenth notes and their combinations, and reject songs where notes of other duration appear. The quarter note is assigned a duration of 1, while the whole note is equivalent to 4 and the sixteenth note to 0.25. Thus, to encode a note, we will create a list of length equal to the number of sixteenth notes it holds. The first element of the list will be the midi representation of the pitch, and the following will just be underscores. For example, the first four notes of the fragment in figure 5 is represented by the encoding [60, '_', 62, '_', 64, '_', 65, '_']. An encoded song will be constituted by the concatenation of the lists of each note.

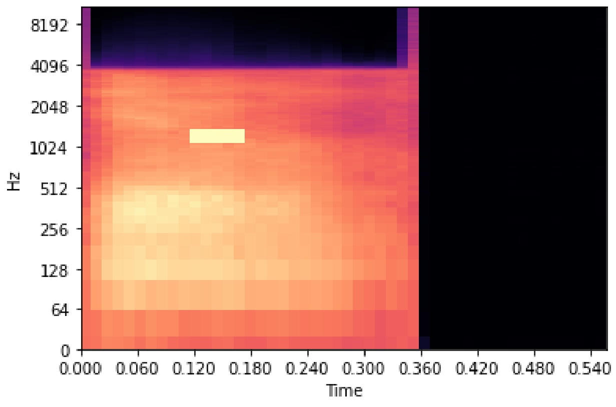

Figure 6: Example of a denoised image using a DCAE trained on the FSDD set. An unexpected saturated region appears, due to the separation between the points in the latent space.

Once all the songs are encoded, the next stage is to join all of them in a single list, from which we will generate sequences to be fed into the network for training. When that is done, we will map all the symbols in the list to integers, getting rid of the underscores that way and obtaining inputs that can be interpreted by the network. Lastly, we will create the sequences that will constitute the inputs and targets of the network. Given a sequence length, an input will be a slice of the list of that length, and its label will be the immediate following item in the list.

4. Training and Results

A. Denoising autoencoders

We trained both DCAE and VAE using as input data $\mathbf{x}$ the corrupted spectrograms and as labels the clean ones, $\mathbf{y}$. With the same number of convolutional layers, we found a better performance from the DCAE in terms of the MSE computed over the test set. Nevertheless, when displaying the denoised images produced by the DCAE, we encountered that most of them presented unwanted regions of time-frequency saturated in amplitude (Figure 6).

Thus, we chose to run several training stages of a VAE, to find the model with the best settings. In order to find the optimal set of hyperparameters (η, B, E) = (learning rate, batch size, number of epochs)[1] and structure of the model, we performed several training stages and then calculated the MSE of the test data. First we tried 4 different architectures, all with the same hyperparameters ($\eta = 0.0005$, $B = 32$, $E = 50$) but changing the depth of the encoder/decoder and the dimension of the latent space,

[1]The reconstruction weight can also be taken to be a hyperparameter. Nevertheless, for simplicity, we will keep it constant through all the calculations, $\alpha = 50000$, as in previous runs we concluded it is not a crucial feature of the model, though it is crucial that it is a high number, to prioritize the reconstruction of the image.

Conv. Layers	Latent dimension	MSE
2	32	0.0018
2	64	0.0016
3	32	0.002
3	64	0.0019
4	32	0.0015

Table 1: Results of the test on different VAE architectures. The models with depth 2 were assigned 100 and 50 filters. For depth 3: 150, 100 and 50 filters. For the model with depth 4: 150, 100, 50 and 50 filters. The size of the kernel and the stride were: 3 pixels and 2 for all layers in the models with depth of 2 and 3; and 5 pixels and strides of 1, 2, 2 and 1 for the model with 4 layers.

Learning rate	Batch Size	Epochs	MSE
0.003 47	20	57	0.0015
0.001 57	29	50	0.0015
0.002 02	47	54	0.0015
0.000 56	37	102	0.0015
0.002 05	48	63	0.0019
0.000 16	33	77	0.0017

Table 2: MSE computed over the test data with a VAE with 2 convolutional layers and a latent space dimension of 64.

to then choose the one that gave the best test MSE and train it with different sets of hyperparameters.

The exploration through the different architectures shows a slight improvement of the MSE for the model with four convolutional layers. Nevertheless, given that the model with 2 layers and latent space of 64 dimensions is computationally cheaper to train and the MSE does not suffer too much, that will be the one chosen to train with various hyperparameters. We performed a random grid search letting them be in the intervals $\eta \in [0.00005, 0.001]$, $B \in [15, 50]$, $E \in [60, 140]$. The results are found in Table 2

From the analysis we can deduce that a batch size too big is not desirable for the training. Although is not presented in the results, we observed that a learning rate too high is not among the best options. A possible explanation is that, in the first epochs, both the reconstruction error and the Kullback-Leibler divergence take considerably high values, sometimes even enough to produce overflow. It seems that the set of hyperparameters that gives the best ratio between computation time and performance lies near the values $\eta^\star \sim 0.002$, $B^\star \sim 35$, $E \sim 60$.

The last test of the variational autoencoder is to actually check if the denoised spectrograms are similar to the original ones. A last model was trained with the parameters picked above in order to get an ideal reconstruction. Figure 7 shows two examples of denoised spectrograms.

About the recovering of audio from the denoised spectrograms, there is a major inconvenient which we cannot skip with this implementation of autoencoder: the STFT is an

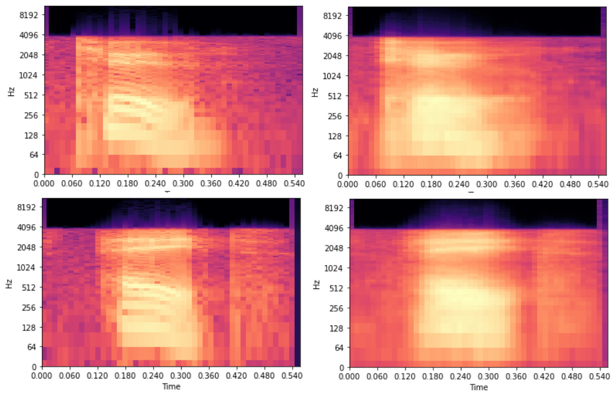

Figure 7: Examples of spectrograms denoised with our final model of variational autoencoder. On the left side, the spectrograms extracted from the original/clean recordings. On the right, the reconstruction obtained from the corrupted/noisy spectrograms.

invertible transformation, which means that we can convert a **complex-valued** spectrogram back to signal. Since, in order to feed it into the autoencoder, we take the module of the spectrogram matrix, we miss the information about the phase which was implicit in the complex values. A way to recover part of the phase information is the Griffin and Lim algorithm[8] (GLA), an iterative process given by the following steps:

1) Add a random phase layer to the real spectrogram.
2) Compute the inverse STFT.
3) From the signal obtained in 2) perform a STFT to generate a complex-valued spectrogram.
4) Replace the magnitude values of the spectrogram produced in 3) with the values of the original spectrogram.
5) Iterate from 2) until the desired accuracy is obtained.

Each iteration of the GLA improves a little bit the phase information of the signal, but it will never be as precise as that of the spectrogram before the module was taken, and this loss of information leads to a low-quality reconstruction of the audio.

B. LSTM

The final length of the data set used to train the LSTM model, depreciating songs where notes of unacceptable duration appear, is of 58 songs, which translates into 25000 training sequences after splitting the data into train and test, leaving 20% for testing. Even though for a generative task like this the most relevant evaluations are related not to accuracy or error, but to the network learning the concepts of key and harmony, we will also check how different configurations of hyperparameters affect the prediction over the test set. The values to vary will be the number of internal nodes of the LSTM layer, n, learning rate, batch size and number of epochs. A random search between the values $n \in [100, 300]$, $\eta \in [0.0001, 0.01]$, $B \in [16, 64]$, $E \in [50, 100]$ was conducted, whose results

Nodes	Learning rate	Batch Size	Epochs	A
256	0.0010	57	70	0.83
133	0.0061	22	74	0.72
184	0.0028	62	52	0.85
191	0.0086	34	57	0.75
207	0.0008	47	86	0.85
179	0.0024	25	51	0.76

Table 3: Results of the accuracy of the prediction over the test set for randomly selected values of the hyperparameters.

are shown in Table 3, aiming to maximize the accuracy score, A.

$$A = \frac{\sum_{i=0}^{n-1} \delta(\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i})}{n} \tag{7}$$

where $\delta(\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i}) = 1$ if $\mathbf{y_i} = \hat{\mathbf{y}}_\mathbf{i}$ and 0 otherwise.

Given the results for the accuracy score, we will use the model with 184 hidden nodes to produce a new melody, since it is the one that gave the best time-performance trade-off. To generate a new melody, the model starts from a seed of our choosing, and predicts the next value of the sequence. Then the seed is updated including the value generated and the process is repeated the desired number of steps, where each step corresponds to the generation of an event of duration equal to 0.25. Figure 8 shows the composition of the model. The seed fed into the LSTM is the part corresponding to the first bar. The most remarkable features about the generated melody are that almost all the notes belong to the key, C Major, and that the sequence ends in the tonic of the key. This ensures that the network has successfully learned the concept of tonality and basic harmony.

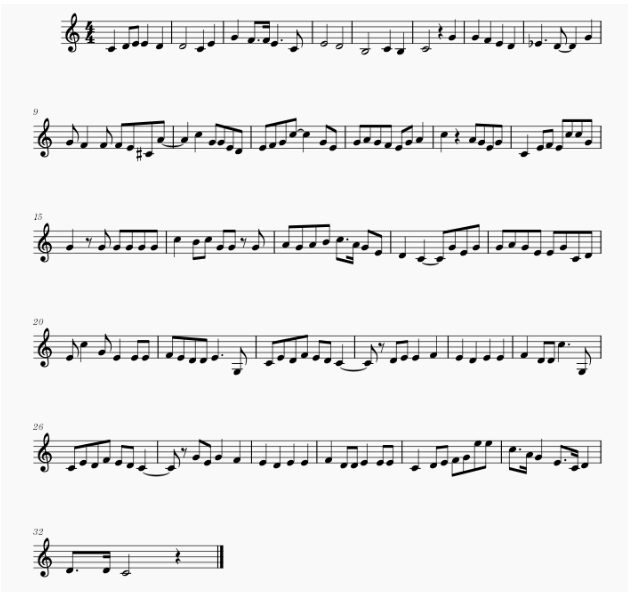
Figure 8: Score of the melody generated by the LSTM model.

5. Conclusion

The journey through the world of audio-oriented machine learning applications has left some takeaway points to keep in mind. The first aspect we would like to emphasize is how important the preprocessing stage is. Around a 60% of the programming effort was dedicated to the analysis and preparation of the chosen data, fact that makes us value the relevance of this step.

In relation to the denoising variational autoencoder, it is encouraging to observe the results of the denoised spectrograms. The reconstructed images recreate the dominant regions of the original spectrograms, even the subtle details in the high frequency spectrum. The approach of autoencoders is interesting not only in the music or audio related work, but it could also be taken as a start point for other any other field dealing with signal or image manipulation. As an example, we find specially interesting the field of single channel separation[?], where an autoencoder can be trained to distinguish a complex signal as it could be a single instrument in a mixed recording.

For music generation, on the other hand, it is surprising how well an artificial model could recreate concepts as key and harmony, that are so deeply connected to human perception and culture. Further work can be done from this point, for example, training a multiphonic network that can be fed with chords (https://www.kaggle.com/karnikakapoor/music-generation-lstm) to produce more complex compositions.

6. Appendix: short time Fourier transform

Suppose we have a discrete signal, as could be an audio recording, consisting of N samples $x(t_n) \equiv x_n,\ n = 0, 1, ..., N-1$. The Fourier transform of the signal is obtained by the discrete Fourier transform (DFT) algorithm.

$$\hat{x}_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi n \frac{k}{N}} \tag{8}$$

$\hat{x}_k$ gives information about the frequency distribution of the whole signal. Thus, when we compute the DFT of the signal, we lose all the information regarding time-dependence. To train a machine learning model, we would like to keep information about both frequency and time evolution. This is achieved with the STFT algorithm. The idea behind STFT is to divide the audio into small sections (windowing), which duration is determined by the *frame size*. A usual choice of windowing function is the step function, with a width equal to the frame size (equation 9).

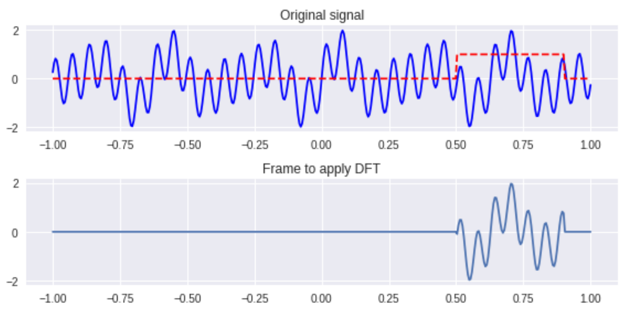

Figure 9: Example of windowing with the step function

$$w(n;f) = \begin{cases} 1 & if \quad n \in (0, f) \\ 0 & if \quad n \notin (0, f) \end{cases} \tag{9}$$

where f is the frame size. Now, we could divide our signal into N/f sections and apply the DFT to each one of them individually. Nevertheless, it is common to divide the signal in a way that the sections overlap, that is, the displacement taken for each section is not equal to the frame size, but to another parameter called *hop length*, represented by the letter H and that satisfies $H < f$.

Now we are ready to define the short time Fourier transform as a two-dimensional map of the frequency distribution for each time frame, called a spectrogram:

$$S(m,k) = \sum_{n=0}^{N-1} x_n w(n - mH; f) e^{-i2\pi n \frac{k}{N}} \tag{10}$$

It is specially relevant to be careful in the way we choose the frame size: a frame size too small would lead to a good resolution in the image, but it would not be able to distinguish low-frequency dependencies. On the opposite side, choosing the frame size to be too large will make us lose information about the time evolution of the signal.

REFERENCES

[1] Keiron O'Shea, Ryan Nash. **An Introduction to Convolutional Neural Networks (pages 5, 6. 2015).**

[2] Valerio Velardo. Generating sound with neural networks (2021, April 27). https://www.youtube.com/playlist?list=PL-wATfeyAMNpEyENTc-tVH5tfLGKtSWPp

[3] Chollet, F. (2020, May 3). Variational AutoEncoder. Keras. https://keras.io/examples/generative/vae/.

[4] Goodfellow et al, Deep Learning, MIT Press, 2016, Section 10.2

[5] Goodfellow et al, Deep Learning, MIT Press, 2016, Section 10.10.1

[6] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, Jurgen Schmidhuber. **LSTM: A Search Space Odyssey.** (page 2, 2017).

[7] The Humdrum Toolkit for Computational Music Analysis. https://www.humdrum.org/index.html.

[8] Nathanaël Perraudin, Peter Balazs, Peter L. Søndergaard, **A Fast Griffin-Lim Algorithm** (Page 2, 2013).

[9] Emad M. Grais, Mark D. Plumbley. **Single channel audio source separation using convolutional denoising autoencoders** (2017).