Neural Networks – KIB.NNKI03

Single-Track MIDI Generation - A Comparison Between DCGAN and LSTM Networks

Robin Kock, s3670600   **Andrei Mihalachi, s3491862**

Annika Möller, s3585832   **Andrei Voinea, s3754243**

{r.kock, a.l.mihalachi, a.s.moller.chandiramani, c.a.voinea}@student.rug.nl

Faculty of Science and Engineering
University of Groningen
9747 AG Groningen, The Netherlands

Abstract

Music has been created by humans for generations, and it has recently become a possibility for machines to generate music to similar effect. In this paper, two different machine learning architectures are compared against each other in terms of the music they produce. The data set used is comprised of MIDI files that are filtered such that they contain only piano melodies. The first approach used in the present report is a Deep Convolutional Generative Adversarial Network (DCGAN), which is used to generate image representation of MIDI files, resulting in an original piece of music. The second is a Long Short-Term Memory network (LSTM), which uses a vector representation of MIDI files to generate music. The results of the two architectures were analysed in terms of music theory, giving light to several aspects in which they differ. Neither model produced coherent patterns that could typically be considered music, but the LSTM approach generated more complex arrangements.

Contents

I. Introduction

Music is represented by organized tones or signals in successions, temporal relationships and combinations, which can express various ideas and elicit an emotional response from a listener. Creating pleasant music is a daunting task even for humans, but previous work has shown that it is possible for a machine to produce musical patterns [1],[2],[3].

Neural network-based approaches used recently that can automate music creation employ Generative Adversarial Networks (GANs)[1],[2], Recurrent Neural Networks (RNNs)[3] or Convolutional Neural Networks (CNNs)[4]. These approaches, except [4], try to reproduce data that use the Musical Instrument Digital Interface (MIDI) standard.

GANs, introduced for the first time in 2014 [5], simultaneously train two models: a generator that captures the data distribution in a sample and a discriminator that tries to identify if the data (e.g. an image) came from the training sample. Therefore, by saving MIDI standard data in an image format, a GAN can be used to reproduce a musical tune. To this end, a Deep Convolutional GAN (DCGAN) was used to generate images from noise.

RNNs use a different approach to generate music. Instead of training a producer network and a discriminator network RNNs will try to predict a song from a sequence of given notes [6]. The network will be trained on many songs always trying to predict the next note. RNNs not only have long sequences of the song as input, but can also have some internal state. So, in principle they are able to remember where in the song they are, or other facts about the song. We use Long-Short Term Memory neural networks (LSTMs). These consist of normal perceptron layers and LSTM layers. These LSTM layers are connected to the input through an input gate, to the previous timestep and to the next layer through an outputgate. Finally they also have a forget gate to delete the current value [7].

In this report, the two previously mentioned approaches are compared on creating MIDI data. The goal is to generate a song of 106 notes that could be generally accepted as being musical. Since there is no real way of quantifying how well the networks achieve such a task, the performance will be assessed subjectively by listening to the generated songs.

II. Data

The data set used is represented by songs in MIDI format, spanning across a wide range of musical genres. Being compiled from several sources from the internet, the entire data set contains 77153 songs[8]. Each MIDI file is composed of several tracks, corresponding to each instrument or sound type found in the song. This data was filtered by track and instrument so that we could extract the part that was necessary for our project (the piano track). Each track can support up to 128 notes. Figure 1 is a representation of data using the MIDI standard.

After analysing all the songs in the data set, a large number of them did not conform to the MIDI file standard. Therefore, only 1405 songs were selected for pre-processing.

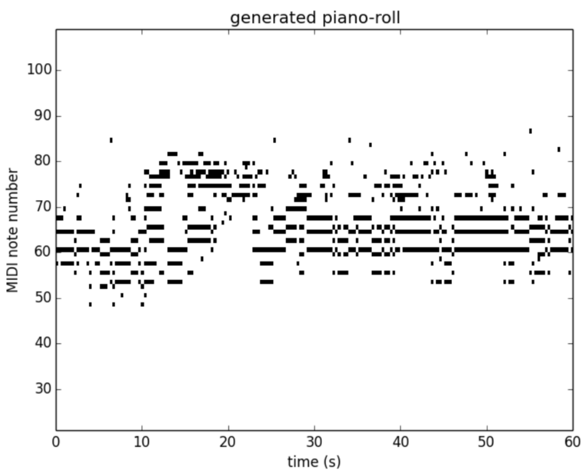

Fig. 1. An image representation of a MIDI file. On the *x-axis* is time, and on the *y-axis* is the note that is played (i.e the pitch). Each individual black bar represents a note that is played for a specific amount of time, and the combination of all of the notes is a song (or an excerpt of a song).

A. Pre-processing for the DCGAN

To train both the generator and the discriminator, a conversion tool was used to create images from a MIDI file [9]. The tool parses the file and extracts the notes of all instruments available. These notes are then placed in separate matrices for each instrument and are then saved as images. To avoid empty files (indicated by entirely black images), the images were only saved if the soundtrack had a piano track and the sum of all matrix elements was greater than 0.

From all valid MIDI files, 7407 images were generated that contained data only from the piano tracks. As we can observe in Figure 2, the notes were represented by white pixels. Therefore, each training image had a dimension of 106x106 pixels.

To ensure that the MIDI data represents a real world piano, the image starts at the 21st MIDI note (representing A_0) and ends at MIDI note 127 (representing G_9). Therefore, to improve performance during training and to avoid improper padding, we have limited each training image to a 106x106 pixels square.

B. Pre-processing for the LSTM

To train the LSTM network a similar method was used. Every MIDI file was analyzed and all tracks/channels that contained piano music were selected. The tracks where filtered by searching through the track name and the instrument name and only selecting the ones that contain piano. After this a matrix was created with 128 rows each row corresponding to one midi note. The columns of the matrix correspond to time steps in the piece of music, more specifically each column corresponds to one eighth of a note. Each value in the matrix is either one or zero depending on whether the note was pressed or not. We choose not to consider the velocity of each note to simplify the training. These matrices where saved temporarily and then later loaded for training.

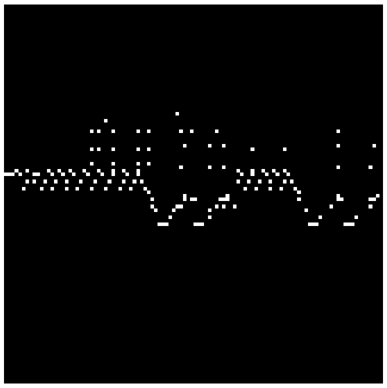

Fig. 2. Minute Waltz by Frédéric Chopin, the first 106 notes in the MIDI representation. The details of how to read the image can be found in Figure 1.

The most significant difference to the data generation for the DCGAN is that the LSTM can take any length of input and predict the next notes while the DCGAN only generates fixed length midi files.

III. Methods

The neural network approaches differed greatly both in the data used, the models employed to generate data and the hyper-parameters. Therefore, we will dedicate the following two sections to explaining how each was built.

A. Building the DCGAN

The data used to train was first converted to floating-point numbers, which would allow us to then normalize it in the [-1, 1] interval. Each image, therefore, had 127.5 subtracted first and then was divided by 127.5. The data set was then randomly shuffled using a TensorFlow method, having a buffer size of 10000 elements to ensure perfect shuffling. The data set was then split into batches of size 256 elements.

1) The Generator: The generator consists of a Sequential model built on the Keras API [10] in TensorFlow. The input layer was represented by a layer of 100 randomly generated values (in a normal distribution, $\mu = 0.0, \sigma = 1.0$). This input was then fully connected to a layer of size 53*53*256, followed by batch normalization and an activation using a leaky rectified linear unit (Leaky ReLU) function. We have chosen a size of 53x53 due to the irregular width and height of the original image. In this manner, we are able to gradually and accurately upsample the random seed input.

The batch normalization is performed using the Keras API, but the underlying algorithm is based on Sergey Ioffe's work [14]. Given x, some intermediate activations in a mini-batch of size k, we are able to calculate the batch's mean and variance

$$\mu_B = \frac{1}{k}\sum_{i=1}^{k} x_i$$

$$\sigma_B^2 = \frac{1}{k}\sum_{i=1}^{k}(x_i - \mu_B)^2$$

To compute a normalized x, we use a small factor

$$\varepsilon$$

that maintains the numerical stability in the following formula

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Finally $\hat{x}$ is transformed linearly through γ and β, which are learned parameters:

$$y_i = \gamma * \hat{x}_i + \beta$$

We are using ReLU as our activation function because it has been demonstrated that these can improve performance of deep models [15]. The Leaky variation is given by the following formula

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise} \end{cases}$$

The Leaky ReLU is therefore useful in our models since we want to maintain some random creativity for the generator and the possibility of the discriminator to correctly assess novel cases.

By reshaping the previous layer to (53, 53, 256), we were able to incrementally deconvolute the data through transposed convolution layers (as seen in Figure 3) to a shape of (106, 106, 1). Batch normalization and the Leaky ReLU activation function were applied to all the deconvolution layers. The bias vectors for each layer were removed to improve batch normalization speed

The output of the generator represents a one-channel image, which is forwarded to the discriminator for analysis. The loss function used is binary cross entropy and the optimizer employed is Adam, both being discussed in a later section.

2) The Discriminator: The discriminator is, again, created from a sequential model. To ensure that this model can identify a complex image created by the generator, we have partially mirrored its design. Therefore, the input image (produced by the generator) with a size of (106, 106, 1) is then passed to a convolutional layer, resulting in a shape of (53, 53, 32). The discriminator gradually upsamples the input using convolutional layers and applies a LearkyReLU function, followed by a Dropout layer (with a 30% chance of removing a neuron), until it achieved a shape of (53, 53, 128) (as seen in Figure 4). We do not use another convolutional layer to upsample the data to (53, 53, 256) due to memory limitations. After flattening the matrix to a vector of length 53*53*128, we fully connected it to a layer of size 1, which represents the output of the discriminator.

During training the dropout layers will set some connections to zero. In our case approximately 30% of all connections will be zero. The layer will also scale up all the other connections to account for the other ones. This reduces the potential for overfitting, because overfitting often results in some weights being very important and large. If

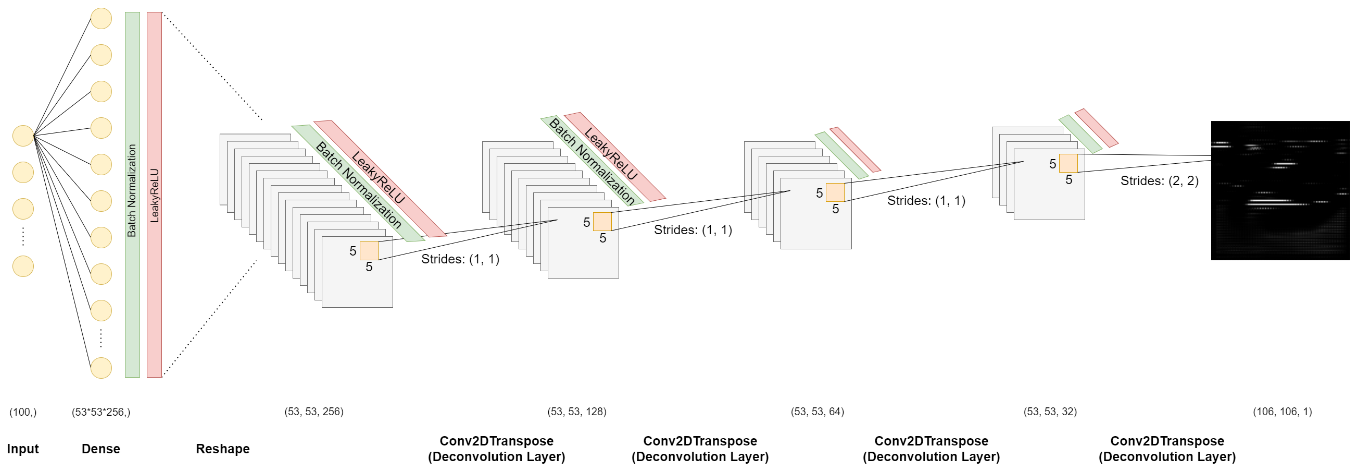

Fig. 3. An illustration of the Generator model. The numbers below each layer represent the shapes, while the text underneath represents the Keras layer types/transformations. The orange squares represent the kernels, which have a size of (5, 5) in our implementation.

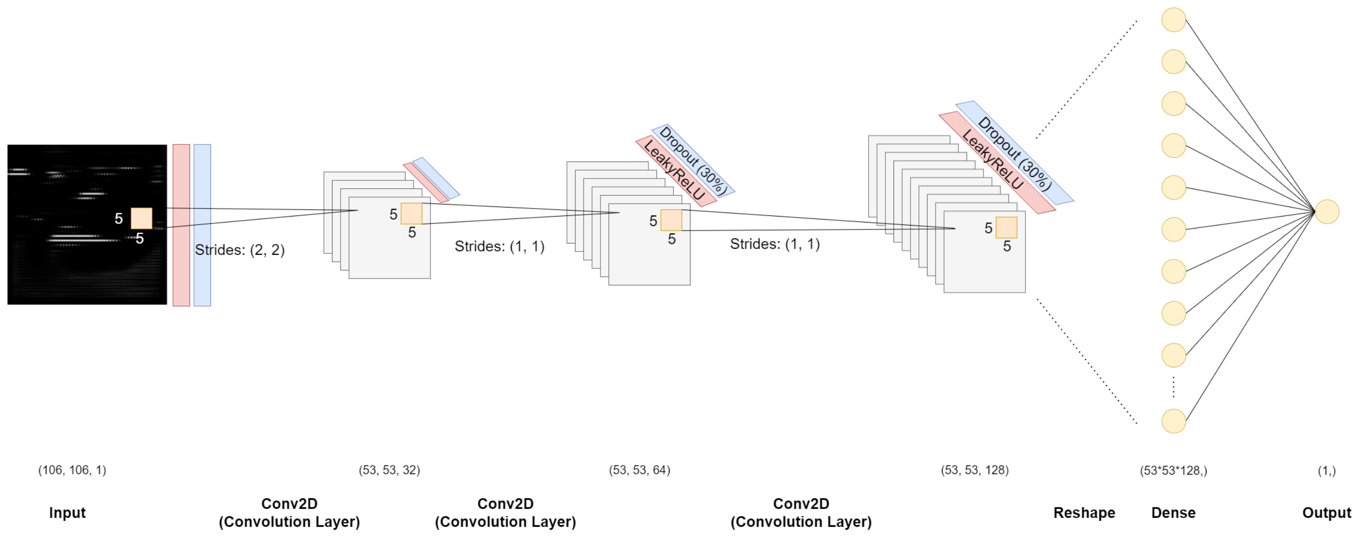

Fig. 4. An illustration of the Generator model. The numbers below each layer represent the shapes, while the text underneath represents the Keras layer types/transformations.

these weights will periodically be zero then overfitting is a lot harder.

Finally, the interaction of the two models can be seen in Figure 5, where the red and blue squares can be substituted with the respective models found in Figure 3 and 4.

As we can observe in Figure 3, the outputs of the generator sometimes contain noise that would not allow us to convert the images into a MIDI format. Therefore, before saving the data as images, a new matrix is created where we threshold each value using the formula:

$$f(x) = \begin{cases} 1 & \text{if } x > 0.2, \\ 0 & \text{otherwise} \end{cases}$$

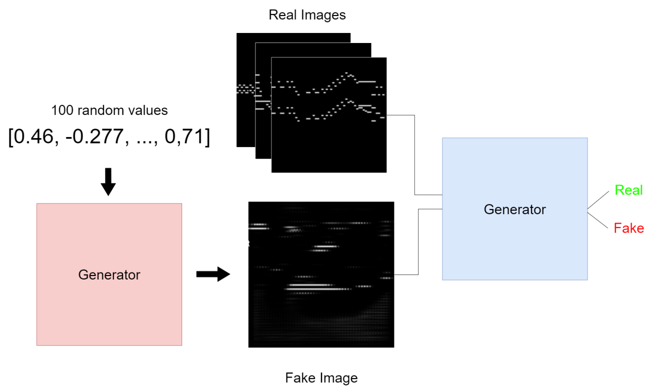

Fig. 5. An illustration showing the interaction between the generator and the discriminator. The array in the top left corner is generated using a generator that outputs values in a normal distribution, given the mean 0.0 and standard deviation 1.0.

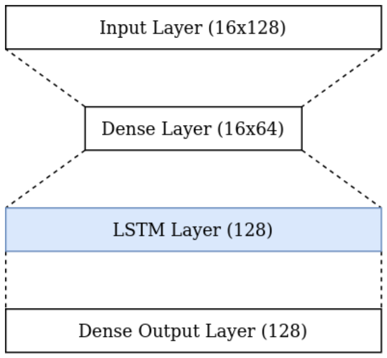

Fig. 6. An illustration of the LSTM model's architecture. Every layer contains its dimensions. The first number is the amount of timesteps per prediction. Finally, the recurrent layer is marked in blue.

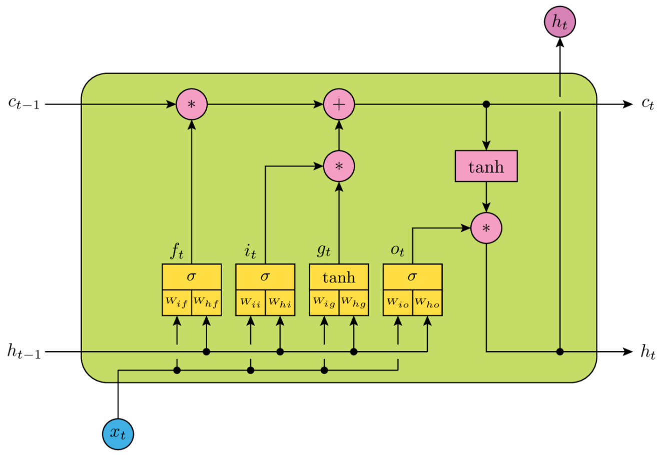

Fig. 7. Illustration of the inner workings of an LSTM cell. *Credit: Andre Holzner*

B. Building the LSTM

The LSTM was build using Keras [10] and its TensorFlow backend. The data was first pre-processed as described in the previous section using a program written in Rust. The resulting data contained 1405 songs and was uploaded to an online Jupyter Notebook environment. Along with the scripts to read the data, train the model, and export midi files. The rest of the training and evaluating was then done using the Jupyter notebook. This was done to gain access to the servers computing power.

1) Model Architecture: For the LSTM model we chose to use a relatively simple architecture. An overview of this architecture is in Figure 6. The first layer is the input layer. It has dimensions of 16*128. The first dimension is the amount of timesteps per prediction and the second dimension are the vectors containing the pressed notes. The second layer is a hidden fully-connected layer. This layer has dimensions of 16*64. The purpose of this layer is to reduce the input information a bit and abstract away from pure notes to something like chords.

The next layer is the LSTM layer, which contains 128 LSTM cells. The LSTM cells have three gates, each taking input from the previous activation of this cell (h_{t-1}) and the input to this layer (x_t). A weighted sum of these inputs to the gaits is computed and then a sigmoid activation function is applied. The three gates are the forget gate f_t, the input gate i_t and the output gate o_t. The input value g_t is computed the same way as the gates only using a *tanh* activation function. The memory value of this cell (c_t) is computed like this: $c_t = c_{t-1} * f_t + g_t * i_t$. And finally the output value of a cell is computed using this formula: $h_t = tanh(c_t) * o_t$. This entire computation is further illustrated in Figure 7.

The final layer of the LSTM model is another fully connected layer with 128 cells. This layer is the output layer and every node directly corresponds to a midi note. The model is trained using the binary cross-entropy loss function. This function will be explained later.

2) Training Procedure: The LSTM model was trained in 220 epochs. Every epoch all the training songs are shown to the network. Every song is split up into batches and each batch contains 64 training steps. Finally, every training step has an input and an output. The input consists of 16 time steps and the output of 1 time step. As already noted every time step is a vector of 128 values representing the midi notes. The model is initialized with random wights and the state of the LSTM cells are initialized with all zeroes.

At the beginning of each epoch the order of the songs is randomized. Then we train the model song by song. At the beginning of every song the states of the LSTM are reset to zero and the song is split up into batches according to the following pseudo code:

```
# ns containes all the time steps in
# the song

# tsc: time step count, this is the
# amount of time steps over all batches
tsc = len(ns) - 16

# this loop makes sure we have the
# right amount of time steps by appending
# time steps without notes
while tsc % 64 != 0:
    ns.append([0, 0, .., 0])
    tsc = len(ns) - 16

# in and out contain all the inputs and
# outputs to the model
in = [ns[i:i + time_len] for i in 0..tsc]
out = [ns[i + time_len] for i in 0..tsc]

# bs is the lsit of batches
bs = empyt_list()
# bc is the total amount of batches
bc = floor(tsc / 64)

# fill the batch array
for b in 0..bc:
    s = b * 64
    e = s + 64
    bs.append((in[s:e], out[s:e]))

return bs
```

These batches are than used to train the model batch by batch using the Adam optimizer. The Adam optimizer will be explained later with the binary cross-entropy loss function. Next we are going to explain some of the hyper-parameters.

The batch size of 64 is a trade-off between increased training stability and the amount of padded zeroes. As is written in the above pseudo code the final batch in a song gets padded with zeroes. These empty notes are harmful for training performance because the network learns to not output any notes. They are, however, necessary unless we use on-line learning with a batch size of 1. After some experimentation a batch size of 64 worked well. It is likely that this hyper-parameter could be tuned further for better performance.

In a similar manner the amount of time steps the model gets as input is a trade-off between the notes required before the model can produce a prediction and model simplicity. If this amount is great then the model can predict the next notes from the input alone, without remembering state in the LSTM nodes, but the model also needs at least that amount of notes to continue the song and is more likely to overfit the songs the songs it gets. We chose a value of 16 after some experimenting, but we are sure that this parameter could also be improved.

During our initial training tests we observed that the model learned quickly but then started to stagnate after some time. We determined that the learning rate was to high and implemented a dynamic mechanism to lower the learning rate. While training we keep two counters, one called l_+ and one called l_-. At the end of each epoch we check whether the loss value increased or decreased. Whenever the loss value increases l_+ gets incremented and l_- is reset to zero. Whenever the loss value decreased l_- is incremented. Then when l_- reaches 7 l_+ is reset and when l_+ reaches 5 the learning rate is decreased by 33%. This way if the loss does not decrease 5 times within a reasonable time frame the learning rate is decreased. We chose the values 5 and 7 to not decrease the learning rate because of some noise in training and they worked well, so we did not change them.

3) Testing procedure: The model was tested using separate testing data. However, since we are dealing with music testing data is not really a good indicator of generalization. The LSTM model is trying to predict the songs and produce music in that way. This is in a way a flawed approach since the problem of predicting music is ill-posed. The only real way to test if the model performs successfully is to run a user study, however, that is out of scope for this project.

To make the model predict songs and generate new songs we create a new model with a batch size of 1. Then we copy the weights from the old model. This new model then takes the start of a song and predicts the next notes. using the next notes this is repeated indefinitely. One last problem is converting the vector of floating point values to a vector of booleans to convert it back into midi files. For this we used a threshold of 0.05, but a different value might be better.

C. Adam optimizer and binary cross-entropy loss function

Both the LSTM model and DCGAN model use the Adam optimizer and a binary cross-entropy loss function. The Adam optimizer is an extension to stochastic gradient decent (SGD) [11]. The main change is that it keeps two exponential moving averages of the gradients per weight. The first average is of the gradient of that weight (m), and the second of the square of the gradient (v). The update rates of these moving averages are called β_1 and β_2 they are set to 0.9 and 0.999 respectively. These are the defaults suggested by the creators of Adam and worked well for us. The algorithm itself first computes m and v and then normalizes them like this: $\hat{m} = m/(1-\beta_1^t)$ and $\hat{v} = v/(1-\beta_2^t)$ where t is the current time step. The final weight update is computed like this: $-\alpha\hat{m}/(\sqrt{\hat{v}}+\varepsilon)$ where α is the learning rate and ε is a small constant to avoid dividing by zero. A big advantage of the Adam optimizer is that it works well for sparse weights and noisy input. We have sparse weights, since a lot of the outputs are zero and convectional networks also often contain sparse weights when selecting for specific features. And we also have noisy input, since the music we input is very diverse and finding a good generalization is close to impossible.

For the loss function both models use the binary cross-entropy loss function. This loss function is used for distributions over binary values. And since every individual note can either be on or off it makes sense to use this loss function. The formula for the loss is:

$$-\frac{1}{N} \cdot \sum_{i=1}^{N} y_i \cdot log(\hat{y}_i) + (1-y_i) \cdot log(1-\hat{y}_i)$$

In this formula y are the correct notes and $\hat{y}$ are the predicted notes. For the LSTM model we also tried the root mean squared loss function, but it gave us worse results.

IV. Results

Both of our models learn over time, since the loss of both models increases. However, the models do not necessarily generalize. This is very evident for the LSTM model. As seen in Figure 8 the training loss decreases for almost all epochs while the testing loss increases for a majority of all epochs. This is a strong indicator that our model does not generalize. As mentioned earlier it is very hard to generalize since generating music is an ill-formed problem. However, in the discussion section we will elaborate on methods to improve this.

The loss values of the generator and the discriminator models, as seen in Figure 9, indicate that both models adapt to one another. The discriminator quickly starts to generate realistic images, which is indicated by the gradual drop in the loss value around the 20th epoch. By training the models across a large number of epochs, we can observe that their loss values will converge around 1.0.

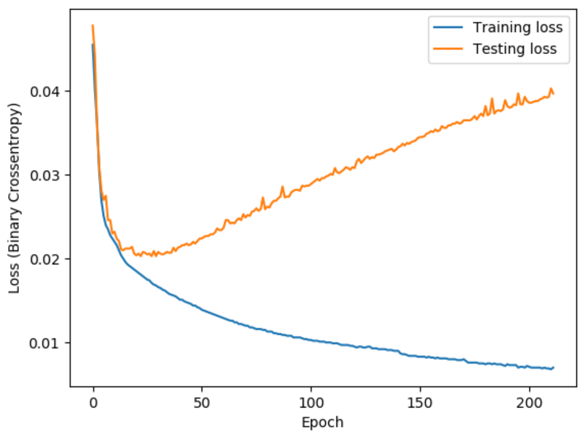

Fig. 8. The training and test loss of the LSTM model over 220 epochs.

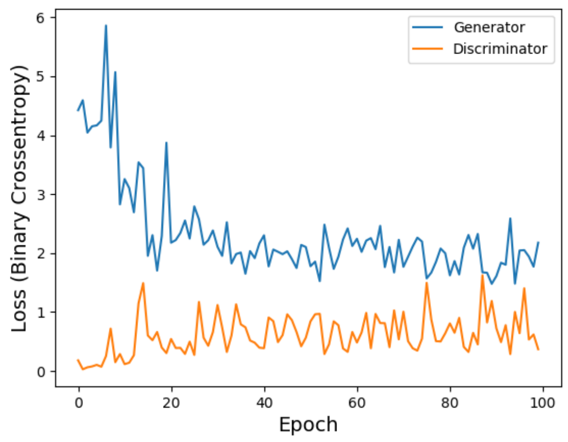

Fig. 9. The loss values of the Discriminator and the Generator across 116 epochs.

For the actual evaluation of our results we choose another method than looking at the loss of training and testing data. As mentioned in the Introduction, there is no conceivable way that we could quantify the success of our model, so we decided to analyze the results of our two neural networks simply by listening and comparing the music that they generate.

A. DCGAN

Upon first listening, the music that the GAN generates is quite obviously less typically 'musical' than that of the LSTM. It features many random pauses and oftentimes a single note is repeatedly played for a few seconds. The music does not adhere to the same key signature, instead switching between key signatures rapidly, which results in a rather atonal sound. There is some use of chords in the GAN-generated music, although individual, more monophonic melody lines are more present. The chords themselves tend to be major triads, which are some of the most commonly appearing chords in Western music [12]. This means that they are comprised of the first, third, and fifth notes of a scale (for example, C, E and G in a C major scale). Furthermore, when one note is played after the other, the interval they are played in is often a third or a fifth. Using the example of the C major scale, this would mean that a C and then an E is played, or a C and then a G. Again, these are the intervals are extremely common in Western music [12]. As mentioned before, the GAN-generated music mostly consist of individual melody lines. Sometimes there are two melody lines that overlap (with seemingly no relation to one another). It is also interesting to note that several of the pieces of generated music begin on the same note.

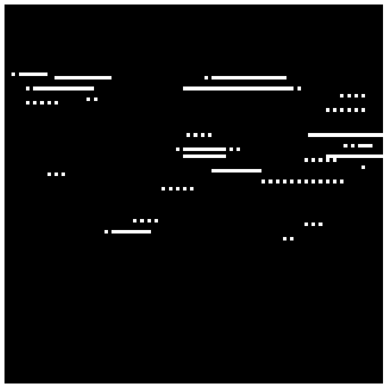

Fig. 10. Example of a song generated by the DCGAN.

B. LSTM

The music generated by the LSTM is more in line with what one would traditionally consider to be ”music”. It stays in the same key for a considerably large part of the track, which helps greatly in making it sound like an actual piece of music. The texture of this music is far more dense than the GAN-generated music, meaning that there are more notes being played at a time. The music generated by the LSTM involves both single melody lines and chords. The chords themselves are more varied than those generated by the GAN, and sound somewhat jazzy. This may be because there are 7th chords being used (which are often used in jazz and blues music [13]), which means that in a typical triad (for example C, E, and G), the seventh note of the scale is also added (in this case it would be a B flat). The rhythm of this music is much faster than the GAN-generated music, and sometimes features syncopated rhythms (also used frequently in jazz). Interestingly, repeated themes (or attempts at them) can sometimes be heard. Most music tends to use snippets of melody (themes) that are repeated throughout the song, which provides a sense of structure. It sounds like the LSTM tried to do this in some way, although not entirely successfully.

V. Discussion

Both models produced music-like MIDI files in the end. However, neither method produced remarkably good results. What surprised all of us was just how different both models

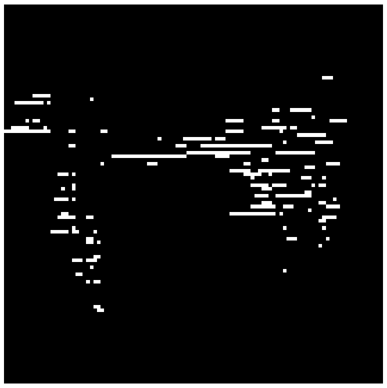
Fig. 11. Example of a song generated by the LSTM model.

were to train. They took different times to train and also required different optimizations and hyper-parameter tuning. This of course made sense, but we expected more experiences to carry over. The LSTM was initially very hard to get numerically stable, this was for many reasons, but mainly because of bad input data and too many layers. After we reduced the number of layers and added regularization the LSTM did not learn any complicated structures of the music. The predicted output often converged to some values for the notes and never changed. The main problems with the DC-GAN were of a very different nature. The models converged to an acceptable solution relatively quickly, however, before we even had time to properly train the model the machine would run out of memory. We had to simplify the model and implement the training differently to get it to work.

One thing which both models lack is a proper method to make the outputs discrete. This is less severe for the DCGAN, since the output is not fed back into the model. We chose to use a fixed threshold for both models, but ideally there would be a better solution where a threshold would be found dynamically or the fixed threshold would be incorporated into the training procedure.

As could be seen in the loss plot of the LSTM model (Figure 8) we overfit our data quite significantly. To avoid this many regularization methods can be used. We tried two methods while building the model but both resulted in the music being too simple and of worse quality. The first method we tried were dropout layers, and the second method was L2 regularization. Both of these methods significantly improved generalization, but the overall loss was higher and the music was worse. The solution to this problem most likely is a more complex model with more LSTM cells and some architecture changes, but this was out of scope for this project.

VI. References

[1] MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation (2017), Li-Chia Yang, Szu-Yu Chou, Yi-Hsuan Yang. Accessed: Jul. 2020. [Online]. Available: https://arxiv.org/abs/1703.10847

[2] MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment (2017), Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, Yi-Hsuan Yang. Accessed: Jul. 2020. [Online]. Available: https://arxiv.org/abs/1709.06298

[3] LSTM Based Music Generation (2019), Sanidhya Mangal, Rahul Modak, Poorva Joshi. Accessed: Jul. 2020. [Online]. Available: https://arxiv.org/abs/1908.01080

[4] WaveNet: A Generative Model for Raw Audio (2016), Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu. Accessed: Jul. 2020. [Online]. Available: https://arxiv.org/abs/1609.03499

[5] Generative Adversarial Networks (2014), Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. Accessed: Jul. 2020. [Online]. Available: https://arxiv.org/abs/1406.2661

[6] Illustrated Guide to Recurrent Neural Networks. (2018), Michael Phi. Accessed: Jul. 2020. [Online]. Available: https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9

[7] Long Short-term Memory. (1997), Sepp Hochreiter, Jürgen Schmidhuber. Accessed: Jul. 2020. [Online]. Available: https://www.researchgate.net/publication/13853244_Long_Short-term_Memory

[8] MIDI Dataset, Musical AI, May 2016.[Online].Available: https://composing.ai/dataset

[9] midi2img. (v1.0), Mathias Gatti. Accessed: Jul. 2020. [Online]. Available: https://github.com/mathigatti/midi2imgnn

[10] Keras. (2015), Chollet François. Accessed: Jul. 2020. [Online]. Available: https://keras.io/

[11] Adam: A Method for Stochastic Optimization. (2014), Diederik P. Kingma, Jimmy Ba. Accessed: Jul. 2020. [Online]. Available: https://arxiv.org/abs/1412.6980

[12] Michael Chanan, "Rise of the triad," in Musica practica: The social practice of Western music from Gregorian chant to postmodernism, Verso, London and New York: Verso, 1994, pp. 59-68.

[13] 7th Chords, Their Arpeggios, and Why We Use Them, Rowan Pattinson. Accessed: Jul. 2020. [Online]. Available: https://www.libertyparkmusic.com/jazz-harmony-7th-chords/

[14] Ioffe, S. and Szegedy, C., 2015. Batch Normalization: Accelerating Deep Network Training By Reducing Internal Covariate Shift. Available at: https://arxiv.org/abs/1502.03167.

[15] Glorot, X., Bordes, A. and Bengio, Y., 2011. Deep Sparse Rectifier Neural Networks. [online] PMLR. Available at: http://proceedings.mlr.press/v15/glorot11a.html.