To CAN or not to CAN?

Using a Creative GAN to generate new music

Hannah Köster, Marlena Napp, Jule Körner

March 2022

1 Introduction

It comes as no surprise that Computational Creativity has been of interest since the development of computers. It unites the "ability that some human individuals possess to create something that did not exist before" [3] with the deterministic instruction we give to the machine along with input data. The underlying quest of Computational Creativity therefore can be seen in better understanding creativity and evaluate how far we might be able to implement and replicate it [3].

One of the sub-fields of Computational Creativity is dedicated to the creation of music using deep learning techniques. There is a multitude of architectures serving different purposes within the creation of music with deep learning. For example, autoencoders can be used to create new musical content based on high-level features of the input and recurrent neural networks for sequence prediction to name only a couple of approaches [1]. Within the scope of this project, we were especially interested in generative adversarial networks (GANs) which can generate content that during training becomes hard to distinguish from the original data [1]. Since the development of GANs [7] the GAN architecture has been applied in various adaptations to different tasks. For example, the generation of Irish folk music [9], merging of genres with the architecture FusionGAN [4] or even transforming a piece of music from a source genre to a target genre e.g. from Jazz to Classic with the architecture CycleGAN [2].

We were interested to work with the paper by Tokui [10] which samples rhythm patterns between different electronic dance music styles with the architecture Creative GAN. We want to transfer this approach to generate not rhythm samples in electronic dance music but see if accompaniment samples in the intersection of the genres Pop, Jazz and Classic can also be generated with this architecture. This would mean that the architecture serves a more general purpose of creating new musical content of not only rhythm but also accompaniment. This architectural approach, CAN, is fitting for the purpose of genre fusion because in addition to creating new samples that are similar to the original samples the model is also trained to create samples that do not belong strictly to either one of the genre categories, thereby encouraging the creation of unique musical patterns. Tokui uses a GAN with Genre Ambiguity Loss or Creative GAN as they call it, inspired by Elgammal et al. 2017. In this GAN architecture, in addition to the first discriminator which is trained to discriminate real from fake samples, a second discriminator is

deployed which classifies the genre of the sample. Through the oppositely acting loss functions of the discriminators, the generator is enforced to create new samples that are as real to the originals but also not clearly categorizable to one genre.

As the code for the Creative GAN architecture is available on GitHub our main task was to find suitable data for our purpose, preprocess the data, adjust the model implementation to the newest TensorFlow version, train the model and evaluate the results with respect to our goal.

2 Data acquisition and preprocessing

2.1 Music datasets

Our constraints for selecting possible data sets were that they had to be sufficiently large (more than 200 songs) and in MIDI format as this format is most common when working with music data and can be easily transformed into sheet music for musicians to play.

The MIDI format (Musical Instruments Digital Interface) can contain various tracks played either simultaneously or successively. A track contains the notes, has an instrument assigned to it, which is indicated by the instrument index, and contains the information if the instrument is a drum. A note has the information of starting and end time, pitch, and velocity of the note. For our purpose, it was important that the MIDI file contained either a chord track (a track where a number of notes have a similar starting and end time) or some other form of accompaniment and no drums. Some exemplary MIDI file we were looking for would be with an instrument that can play chords as the Piano for example, and without a drum track. We ended up with a Pop Dataset[11] of 909 songs, a MIDI file collection of different genres containing also a Jazz Dataset [6] of 491 song and Classic Dataset of 292 songs. Given that not all of the samples contained a distinct accompaniment track that we could single out for chord extraction, we relied on the commonly found relation between melody and chords meaning that the melody line typically consists to a good amount of the tones found in the chords of the accompaniment.

2.2 Preprocessing

In order to acquire suitable samples to train our model with we needed some more preprocessing. We decided to use the libraries pretty_midi [8] and music21 [5]. As we aimed to create new accompaniment samples we used the chordify function from music21 to write the notes from all tracks of a MIDI file as chords in a single track. The reason for this is that we wanted to look for chord progressions rather than disassembled chords and melody. An example of how chordify transforms a sample can be seen in Figure 1. Furthermore, we used only samples in $\frac{4}{4}$ time signature to reduce the complexity of the samples.

After these first preprocessing steps, we needed to encode the MIDI information in a matrix format to train the model with the music data. The following encoding steps were inspired by the original paper and by the paper of Gino Brunner et al. [2]. We decided on a representation that indicates which note was played at which point in time (short: time x pitch). In a MIDI file, each note has a specific pitch, the possible pitch range goes from 0 to 127 in discrete steps. Furthermore, since each note can be of arbitrary length, we had to discretize time to enable matrix representation. As proposed by the mentioned papers, we tackled this issue by

re-sampling the songs in sixteenth notes. That means, by iterating the notes in the pretty_midi object we got the starting time of each note. We then calculated how many sixteenth notes correspond to this time.

For example, in a song that is in $\frac{4}{4}$ time signature and 120 bpm (beats per minute), there are 8 sixteenth notes per second. Taking a note that starts 4 seconds after the song has started corresponds to $4\ \text{sec} * 8\ \frac{\text{sixteenth}}{\text{sec}} = 32$ sixteenth notes. We did the same calculation for the end time of each note. In this way, we got a discrete-time representation for each note. After that, we marked each note start with a "1" and each end with a "3" in the matrix. The time steps in between got a "2" indicating that the note was held. Using this method, we transformed the whole song into a matrix representation.

Nonetheless, since each song might be different in length, the time dimension might vary in all songs. That is why we split the song matrix into smaller sub-matrices, such that each contained 64 sixteenth notes (correspond to 4 bars). We finally obtained several samples, all in the size 64x128 and ultimately labelled them with their respective genre.

After the preprocessing, we had obtained 5338 Pop samples, 13531 Jazz samples and 4464 Classic samples. In order to continue with the same amount of data from every genre, we took 4464 samples of each genre. The code for the preprocessing can be found here.

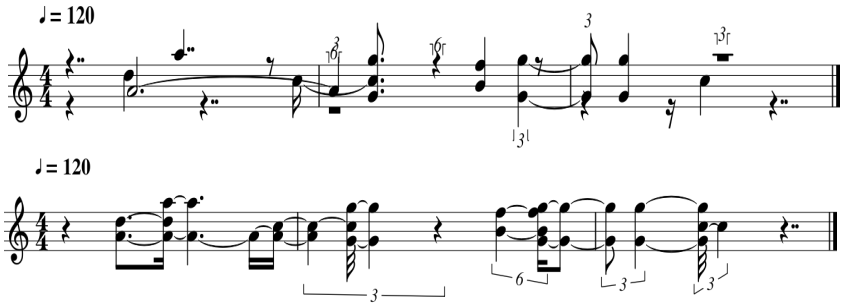

Figure 1: Classic genre note sequence before (upper row) and after (lower row) using Chordify

3 Model

3.1 Vanilla GAN

The Vanilla GAN is composed of two networks, the generator and the discriminator. The generative attribute of this architecture can be seen in the objective associated with it. Namely, generating new data based on the data that the model was trained with. The adversarial part is formed by the antagonistic interaction between the generator and the discriminator. More specifically, the generator's task is to generate data from random noise that is indistinguishable from the real data. The discriminator, on the contrary, tries to distinguish this generated fake data from real data. Furthermore, the loss functions reflect this antagonistic behaviour as well. During training, the generator loss penalizes the generator for producing samples that are classified as fake by the discriminator. Whereas the discriminator is penalized for each miss-classification (classifying fake samples as true and true ones as fake). In general, they use the same loss function. While

the generator minimizes, the discriminator maximizes the function during training.

3.2 Motivation for CAN

In order to be successful, the generator has to produce data that is as similar as possible to the real data. This leads to a generator that rather imitates the real data than comes up with something new that diverges from the underlying training data. Therefore, in the end, the network is not creative. In our case, this would mean that the trained network is able to produce music that is very similar to the three music genres of our training dataset. However, we intended to generate new music that cannot directly be traced back to only one of the three music genres. For this purpose, the paper by Tokui [10], on which our work is based, proposes an extended GAN framework, the Creative GAN (CAN).

3.3 Creative GAN (CAN)

In the Creative GAN, the generator is encouraged to generate data that diverges from the training data. This is done by adding a second discriminator. Next to the binary fake/true classification of the first discriminator (D_r), the second discriminator (D_c) determines the probability that a generated sample belongs to a music genre (multi-class classification). Thereby, during training, the generator does not only get feedback to adjust its weights such that its generations resemble the real data, but also that D_c is uncertain about the genre, leading to a uniformly distributed output. In order to get this feedback signal, the Vanilla Loss (1) is supplemented by the Genre Ambiguity Loss and the Genre Classification loss.

The resulting loss function, as proposed by Tokui [10], looks like this:

$$\min_G \max_D = log(D_r(x)) + log(1 - D_r(G(z))) \tag{1}$$

$$+ log(D_c(c = \hat{c}|x)) \tag{2}$$

$$- \sum_{k=1}^{K} (\frac{1}{K} log(D_c(c_k|G(z))) + (1 - \frac{1}{K} log(1 - D_c(c_k|G(z))))) \tag{3}$$

The goal of D_c is to classify each training sample with the correct genre by maximizing $log(D_c(c = \hat{c}|x))$ (Genre classification loss (2)).

The Genre Ambiguity Loss (3) equals the categorical cross-entropy loss, where K is the number of genres (in our case 3). A uniform distribution is taken as the target distribution. The generator is penalized when D_c classifies the generated samples with high probability. Meaning, D_c is very confident about the genre of the generated data. Hence, the generator aspires to minimize the cross-entropy with the uniform distribution by generating samples that cannot be attributed to one genre, thereby maximizing (3). This pushes the model to generate data that does not directly belong to one of the training genres and thus induces creativity. The structure of the Creative GAN can be seen in the following figure:

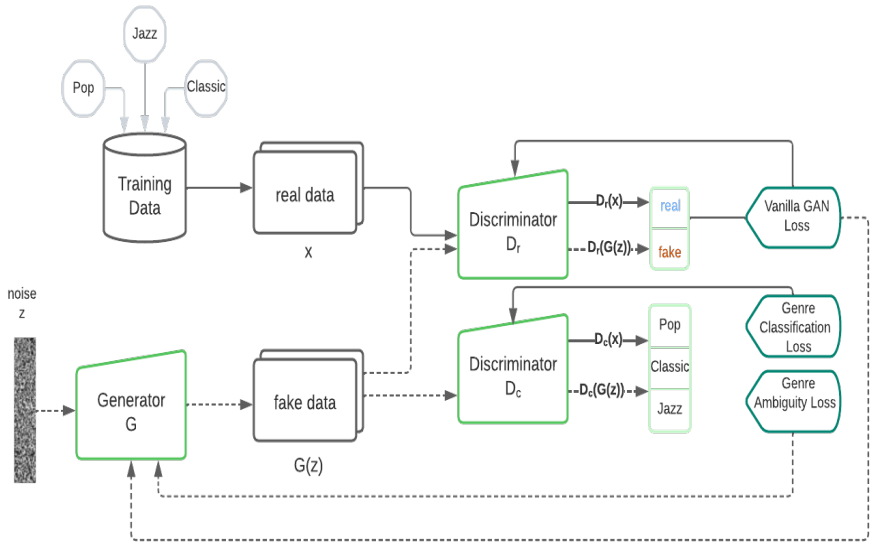

Figure 2: Creative GAN Architecture

4 Training

As with the model, the training loop is adapted from Tokui [10], with changes made to adjust for the slightly different model, logging in tensorboard, and using a newer TensorFlow version. Essentially, the training consists of two parts – training the discriminator and training the generator. For every time the generator was trained, the discriminator was trained three times. In order to keep training effective, the training of either of the two parts is suspended if the loss ratio becomes too large, indicating that one part overpowered the other. As soon as the loss ratio falls under the set threshold, which in our case was kept at 5, the training of the respective part resumes. The discriminator was trained on batches of real and generated or fake samples, with the D_r part receiving ones or zeros as true labels depending on whether the samples were fake or real, and the D_c part receiving the true genre labels of the real data for both the real and fake samples. To train the generator the whole GAN is used for training, with the discriminator part being fixed and not trained during this time. So, training the generator consists of generating noise, inputting it into the generator. The output of the generator is then used as input for the discriminator, with the goal of optimizing the generator loss - which we called Vanilla GAN loss in the CAN section and which stems from the D_r discriminator -, and genre ambiguity loss – which comes from the D_c discriminator.

For each training step of the discriminator and generator, the losses and corresponding accuracies are logged, and at the end of each epoch, a sample is generated. Furthermore, sheet music and audio of each sample are stored such that they can be viewed in the tensorboard. To enable the continuation of previous training temporary versions of the model are stored after each epoch and the model with the minimum loss during the current training run is also saved. These saved

models can then not only be used to continue training after a pause but can also be reloaded to generate samples without the need to retrain.

4.1 Hyperparameters

This architecture has quite a few hyperparameters that can be adjusted for the training. As previous studies provided seemingly well-working hyperparameters as good starting points, we decided to use the ones used in our original paper [10] as initial values, as our model is based on and very similar to theirs.

It should be noted that we did not split our data into separate training, validation, and test datasets, but instead used all available samples that met the criteria for the training. This was done as ultimately only the quality of the generated samples is important as opposed to the loss and accuracy metrics, which for GANs do not hold the same meaning as for other architectures and can be hard to interpret. The goal of this project is to generate accompaniment parts that are new and lie somewhere between the three genres of pop, classic, and jazz, and not to perfectly train the discriminator to work not only well on the training data but also generalize to unseen data.

5 Results

Training the model was very time-consuming, one epoch took over one hour. Due to the time constraints, we had for our project and some difficulties concerning Colab and Drive storage space that made it difficult to train for longer periods without human intervention, we were only able to train our model for 39 epochs. Nonetheless, despite this rather short training, we can summarize, that our model became better during training. From starting with samples that looked diffuse and unnatural, since all possible notes were displayed at the same time (Figure 3, the model learned quickly to generate note sequences that resemble ”real” note sequences. That means, the overall number of notes per sample was reduced leading to smaller chords but also many single sixteenth notes. In addition to that, the pitch range also became more natural. As it became clear in the Preprocessing section we allowed the maximal possible pitch range of 128. However, it is quite unusual that a regular piano plays very high or low pitches. Our model learned to capture the common pitch range in between, this indicates that it tries to learn the underlying data distribution in order to fool the discriminator D_r.

Furthermore, next to only learning to generate samples with the same sixteenth notes (Figure 4), it also learned to generate notes that are held (Figure 5). This shows that the training delivers more complex results the further the training progressed. But in general, there were only up to 4 different notes in one sample in Figure 6 you can see one of the most sophisticated samples that we have achieved:

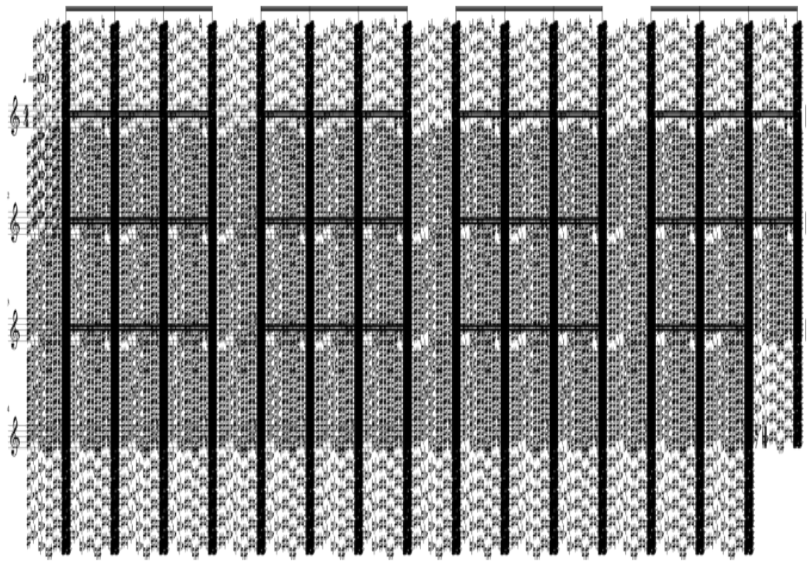

Figure 3: output before training

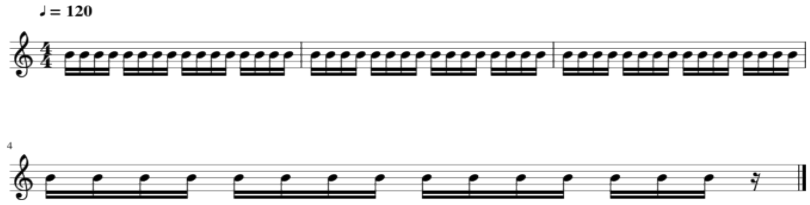

Figure 4: simple sixteenth notes in epoch 25

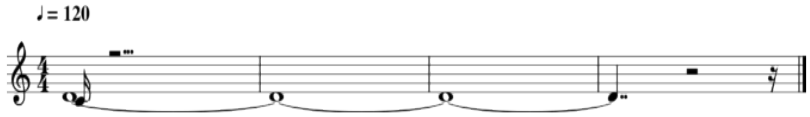

Figure 5: note held in epoch 32

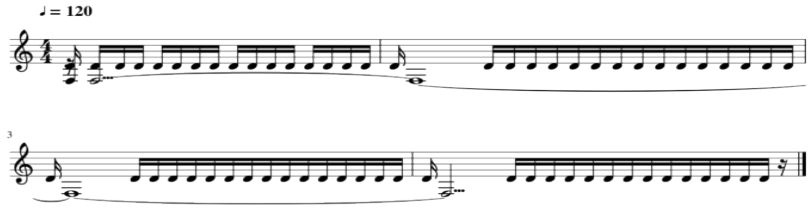

Figure 6: more complex note sequence in epoch 34

Unfortunately, the evaluation after training where we checked the performance of the trained generator revealed mode collapse. Thus, the trained generator only outputs samples that all look very similar. We therefore cannot produce a large variety of different music samples.

It can be said that we did not get our intended results, as we aimed to get samples that sound appealing. Nonetheless, due to the progress of the training that revealed also some good results, we are optimistic to get better results in the future. The reasons for some of the problems we faced will be made more explicit in the following section together with ideas to achieve better-sounding samples.

6 Discussion

As already mentioned, we did not get entirely satisfactory results, which likely is caused by insufficient training. As also already stated, we were unable to train for more epochs as training took a very long time. Reducing the number of trainable parameters of the network by adjusting some layer sizes might make training faster. In addition to that, minibatch learning could be used to speed up training without affecting the performance too much.

Due to the training being very time-consuming we initially planned to only adjust the hyperparameters of the batch size as we expected this to have the biggest effect. However, due to runtime issues, we did not have the chance to train multiple models in parallel. Furthermore, we did not have time left to run them in succession which is why we only have meaningful results for the batch size of 64. Thus, further tuning of the hyperparameters is left to be done in future studies. Most of the problems we had during training stemmed from running the training on Google Colab, as the GPU usage is limited, meaning that most of the training ran on slower CPU runtimes. Additionally, runtimes get disconnected after a while, especially if no activity is detected. So frequently storing the model to avoid losing progress is recommended. Nonetheless, saving to Drive also requires some care as free storage capacity is limited and easily reached when storing large datasets and models and full storage might cause models to not be saved completely. So, if running the training in Colab and saving the progress to Drive, one should regularly check whether the model is still training and make sure that there is enough storage space left, especially when letting the training run unattended for longer periods such as overnight.

When generating samples by using the stored models we noticed that the model suffers from mode collapse. This is a common problem of GANs and cannot entirely be avoided. Possible approaches that might help and could be used in future expansions are feature matching or minibatch discrimination. Furthermore, label-smoothing might also be a good idea. Initially, we also planned to use label smoothing as an additional training variant, but did not get to use it due to the mentioned time issues.

One thing that should be noted with respect to the data representation is that reappearing patterns or chord progressions in accompaniment can appear over different, non-fixed time scales. So using samples with a fixed time dimension (64 in our case) is probably not ideal. But obtaining suitable data for different time scales is very complicated, as the progressions or patterns would need to be detected and after that the songs needed to be split accordingly between the repetitions. However, detecting chords alone is not an easy task and is a research field of its own.

Although we did not yet achieve the expected results, we assume that with some adjustments and more training this architecture might be suitable not only for novel rhythm and accompaniment creation but also for the creation of new melodies and other types of musical content. Furthermore, it will be interesting to expand the architecture to other genres such as Rock, Folk or Latin music. An expansion of possible creations could be achieved by leaving out the $\frac{4}{4}$ time constraint which had been implemented in our project. This caused a quite drastic reduction in available samples and in retrospect was likely unnecessary, as we re-sampled the songs later to end up with uniform data. Future implementations should also consider that the full midi range of 128 (pitch dimension) is unlikely to be used. As this might render the training process more complex and slow we suggest sizing down pitch dimension to e.g. 64. Brunner et al. for example, propose a pitch dimension of 87 since a standard piano usually can play the pitches between 21 and 108 [2].

At last, we also want to put our work into a larger context of musical creativity and creation. With the rise of Music Streaming Platforms such as Spotify (entering the German market in 2012) and their incentive to match users with music they might like, a lot of effort and computational power has been put into the personalization of these suggestions. Already there is a lot to be said about the effects of these algorithms on artists and consumers. Small artists might be able to extend their reach faster through recommendations technologies. At the same time, users might be kept in a musical bubble, fed with new but similar content, having a hard time actually exploring outside their "musical comfort zone". With the increase of research on computational creativity, we also have to consider how artificially created musical content might affect artists and consumers. When recommendation algorithms expand their function from generating assorted playlists to generating musical content based on the favourite genres of a listener, we will have to reevaluate the value of music creation and creativity.

7 Conclusion

Although we expected the results to be more insightful into how genre-merging between our chosen genres could look like we as musicians are happy that the model at least outputs some playable notes. So to answer the question that we posed in the very beginning: The CAN can create useful samples, but needs some refinement with respect to preprocessing and optimization. Therefore, we conclude that our model itself trained well, but our goal of creating samples that represent the genres of Pop, Jazz and Classic has not been achieved fully. Nevertheless, we learned a lot about how to approach future projects. We think that future implementations of this model will likely result in better and more exciting samples when training the model for a longer time and avoiding mode collapse.

References

[1] Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. Deep learning techniques for music generation–a survey. *arXiv preprint arXiv:1709.01620*, 2017.

[2] Gino Brunner, Yuyi Wang, Roger Wattenhofer, and Sumu Zhao. Symbolic music genre

transfer with cyclegan. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 786–793, 2018.

[3] Filippo Carnovalini and Antonio Rodà. Computational creativity and music generation systems: An introduction to the state of the art. *Frontiers in Artificial Intelligence*, 3:14, 2020.

[4] Zhiqian Chen, Chih-Wei Wu, Yen-Cheng Lu, Alexander Lerch, and Chang-Tien Lu. Learning to fuse music genres with generative adversarial dual learning. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 817–822, 2017.

[5] Michael Scott Cuthbert and Christopher Ariza. music21: A toolkit for computer-aided musicology and symbolic music data.

[6] Lucas N Ferreira, Levi HS Lelis, and Jim Whitehead. Computer-generated music for tabletop role-playing games. 2020.

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[8] Colin Raffel and Daniel PW Ellis. Intuitive analysis, creation and manipulation of midi data with pretty_midi. In *15th international society for music information retrieval conference late breaking and demo papers*, pages 84–93, 2014.

[9] Bob Sturm and Oded Ben-Tal. Let's have another gan ainm: An experimental album of irish traditional music and computer-generated tunes, 2018.

[10] Nao Tokui. Can gan originate new electronic dance music genres?–generating novel rhythm patterns using gan with genre ambiguity loss. *arXiv preprint arXiv:2011.13062*, 2020.

[11] Ziyu Wang*, Ke Chen*, Junyan Jiang, Yiyi Zhang, Maoran Xu, Shuqi Dai, Guxian Bin, and Gus Xia. Pop909: A pop-song dataset for music arrangement generation. In *Proceedings of 21st International Conference on Music Information Retrieval, ISMIR*, 2020.