BeatGen - A Variational Model For Generating Drum Grooves

Matej Sojak, Jakob Lindermeir

***Abstract*—We propose a variational autoencoder which encodes single bars of a song into latent vectors as a model for music generation. A set of drum grooves of varying length is used as the data set. Latent vectors generated by the model can be used to interpolated between two given bars to create a smooth transition between two musical styles. The model can also be used to sample bars similar to one provided. The latent representation could be useful to model a song as a sequence of bars in further exploration.**

I. Introduction

This work was done as the final project of the course "Machine Learning and Data Mining" at the University of Valencia. During the course, the main aspect was using the machine learning technology for analytical purposes, such as classification. We would like to explore whether it is possible to use the taught techniques to support a creative process such as music composition.

We were influenced by previous investigations of the topic at different levels of professionality. The used dataset was gathered by a group of researchers at the Google Magenta project, which released a paper in 2019 detailing their data structure and a generative models [1]. An additional paper with another dataset, but a similar model as ours was also released in 2019 by members of the Magenta project [2]. We were also inspired in the way we represent the musical data by a blog post by Abraham Khan [3].

Most projects we looked at took one of two approaches: Either a model generating a sequence with a fixed length was proposed, or the model generates a song note by note. Our approach differs in that regard, because it aims to split the song into the bars it is composed of. This seems to be a reasonable decision especially when dealing with drum samples, because here the base structure of the bar often does not change much over the course of a song. Many other projects also only allow one note to be played at the same time. This is not viable in drum samples, since many instruments are often hit at once. Finally we consider how hard a given instrument is struck instead of just determining that is is hit, because we noticed that it made a big impact on the subjective hearing experience even if we do not change other aspects of the data.

This report is provided along with the code we wrote, which can also be found in a GitHub repository [1]. Additionally, sample output data from the model as well as pretrained model weights are included. We cannot provide the dataset itself directly due to licensing issues, but it is freely available for download. Python was used for the implementation of the model and data handling. The model was created in *Tensorflow* version 2 using the *Keras* API. Reading and writing of MIDI files was done using the *mido*. Most of the computation was done on the *Google Colaboratory* cloud computation platform using a GPU accelerated backend. The MIDI files were played using *Audacious*[2] with the *fluid* sound font.

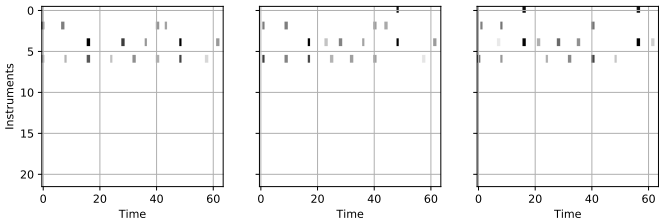

Fig. 1: A sample array representation, showing the first three bars of a song

II. Dataset

The Groove MIDI dataset[3] consists of a number of MIDI files, which are recorded performances of professional drummers on an electronic drum kit. The performances vary in length, style and speed. We only selected the recordings with a 4/4 time signature, which amount to 1139 recordings with 21695 bars in total.

A MIDI file consists of a sequences of messages encoded in a certain binary format. Two types of messages exist: Meta messages encode certain data, such as playback speed, time signature or the key the song is in. Note messages encode events if a note is turned on (e.g. a key on a keyboard is depressed) or off (e.g. the key is released). The pitch attribute of a note message specifies the frequency if the note. Since we are dealing with percussion data, these pitches do not have an intrinsic meaning, but are instead defined to represent a certain instrument. For example, a pitch of 36 represents the *Base drum 1* according to the general MIDI (GM) specification. The full table of the drum mapping can be found in the link at the footnote. We found that using the default setting of audacious provides satisfying playback results. Additionally, the force with which the key is pressed is recorded. This is called the velocity in the GM specification. Velocity values can range from 1 to 127. Each message also has a time attribute, which represents the time passed since the last message was received. Therefore a MIDI file has a relative time structure,

[1] BeatGen GitHub repo

[2] Audacious player

[3] Groove MIDI dataset

which we deemed unsuitable for machine learning purposes since it is difficult to normalize to a fixed input size.

We therefore converted the MIDI files into a three-dimensional array representation. Each element in the array represents an instrument played at a certain time in the song. The value of the array element corresponds to the velocity the note is played with. The first index describes the bar the note is in, while the second index indicates the timing of the note within the bar. The third index specifies the instrument, which is encoded in a dense integer representation ranging from 0 to the total number of instruments used, in our case 22. MIDI file analysis, parsing and writing functionality is provided by the functions in the module *MIDI.py*. A diagram of a sample representation can be seen in figure 1.

The function *analyse_filepack* provides statistics for a set of MIDI files and generates the integer encoding in the form of a dictionary. With this information, it is possible to convert a file using the function *parseFile*. The number of time steps in a bar can be set freely as a function parameter. In the files we used, a bar is 1920 MIDI ticks long, which means each note in a 4/4 time signature track is divided into 480 sub notes. At a tempo of 130 beats per second, this is far below the temporal resolution of the human ear and would also make the array representation big and sparse. After some experimentation, we decided of a temporal resolution of 64 steps per bar as a trade off between sound quality and array size. Therefore, the array representations have a shape of (number of bars, 64, 22) for each MIDI file. The function also normalizes the velocity information in the range [0,1]. Since we are dealing with drum data, the sound sample corresponding to an instrument is played entirely whenever a note_on message is received. Therefore the note_off messages carry no information and are discarded during conversion. The converted arrays are saved in the NumPy .npy format for convenient loading. Additional functions are documented within the module itself.

III. Model

The model we propose is termed a variational autoencoder (VAE) in literature. It is a variant of a classical autoencoder. The model aims to learn a meaning full latent representation of the dataset by modeling a multivariate normal distribution. The covariance matrix of this distribution is defined to be diagonal, because it is advantageous for all the latent dimensions to be independent of each other. The model consist of two functions: the encoder e and the decoder d. For each input sample, the encoder generates two vectors of dimension l. The first one describes the mean of the normal distribution, while the second one describes the variance. From this, a single vector z is sampled, which is passed to the decoder. The decoder maps back to the original input space, and aims to reconstruct the input to the encoder. A summary of the mappings for a rank two tensor input space is given below:

$$e : \mathbb{R}^{mxn} \to \mathbb{R}^l \times \mathbb{R}^l$$
$$x \mapsto (\mu, \sigma)$$

$$d : \mathbb{R}^l \to \mathbb{R}^{m \times n}$$
$$z \mapsto x'$$

Sampling from a normal distribution is not differentiable. To be able to apply optimization algorithms such as gradient decent to this model, a technique called the reparametrization trick has to be applied. Instead of sampling from the normal distribution directly, a standard normal distribution is sampled l times and the resulting vector is rescaled according to

$$z = \mu + \epsilon \times \sigma \tag{1}$$

where ϵ is the sample take from the standard normal distribution. To train the model using gradient descent, a loss function has to be provided. In VAEs, the loss consists of two parts. To ensure that the autoencoder tries to reconstruct the input, a reconstruction (RL) loss is computed.

$$Loss_{RL} = \Sigma_{i,j}(x_{ij} - d(e(x_{ij})))^2 \tag{2}$$

To prevent the model from spreading the learned distribution to far across the latent space, a regularization term called the *Kullback–Leibler divergence* is added. This is a measure of difference between the given diagonal normal distribution and a standard normal one. Therefore, high mean values or variances different from 1 are penalized.

$$Loss_{KL} = \frac{1}{2}\sum_{i=1}^{l}(\sigma_i^2 + \mu_i^2 - \ln(\sigma_i^2) - 1) \tag{3}$$

The total loss is then computed as the sum of the two losses:

$$Loss_{VAE} = Loss_{RL} + Loss_{KL} \tag{4}$$

The implementation of the VAE is adapted from a Keras example [4], which was adjusted to our needs. It can be found in the *models.py* module. The encoder model consists of three passes of a temporal convolution layer with a progressively increasing number of filters and temporal filter size 3 followed by temporal pooling with kernel size two. Afterwards, a two dense layers a used to generate the vectors describing the distribution of the desired dimension. The decoder is applying this architecture backwards: first a dense layer is used to generate an intermediate output, which is reshaped to an initial two-dimensional representation. This representation is fed into a upsampling layer followed by a convolutional one, which is repeated four times. A summary of the model can be obtained by setting the *debug* flag on instance creation.

IV. Results

The model and the learning algorithm can be adjusted by many different hyper parameters, which impact the model architecture and its learning behaviour. Since we are limited in time and resources, we cannot justify every decision we

[4] Keras VAE

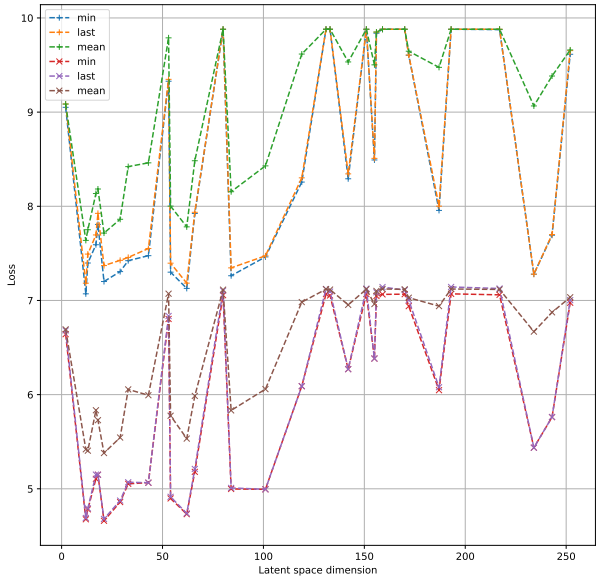

Fig. 2: Visualisation of different training curve characteristics as a function of the latent space dimensionality. The minimum, last and mean value of the reconstruction loss are shown for both the training and validation data set. the dotted lines between the point only serve as a visual guide and have no intrinsic meaning.

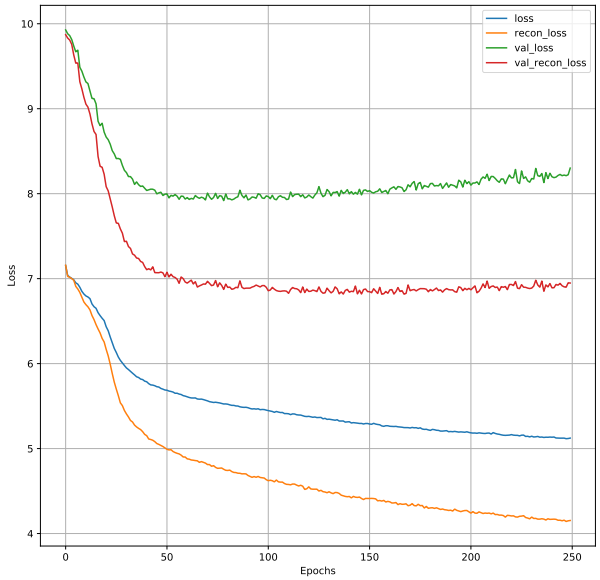

Fig. 3: Loss curve of a 16-dimensional latent space VAE. Reconstruction and total loss are show as a function of training epochs for both the training and validation set. The model converges approximately at episode 75.

made, but instead a lot of the found settings are the result of experimentation. A 10% of the data set is used for validation purposes. After we found a usable architecture, we focused of determining the optimal dimensionality of the latent space. A comparison of the training success for different latent space dimensionalities can be seen in figure 2. Notably the diagramms shows no general trend and overall chaotic behavior while

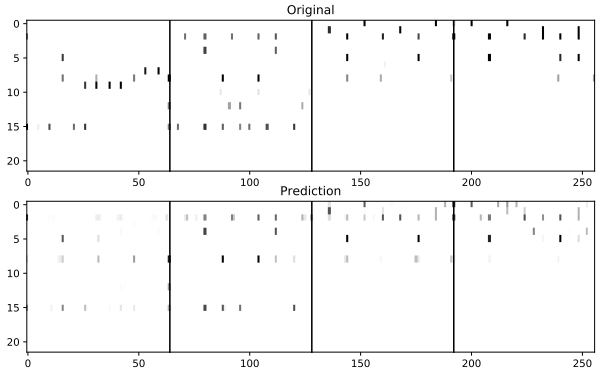

Fig. 4: This diagram shows four bars choosen randomly from the dataset as well as the reconstruction by the VAE

changing the dimensionality. The models were trained for 250 epochs each. We decided to use a 16-dimensional latent space for our model, because it produced reasonable results and can be trained faster than models with a higher dimensional latent space. A learning curve for this model can be seen in figure 3.

After successfully training the model, we thought of two ways for generating music with it. To create a smooth transition between two bars of percussion music, we encode both as latent vectors. We then generate intermediate latent vectors by linear interpolation. The initial latent vectors are pre- respectively appended and the entire sequence is then decoded back into the original space. Then the resulting array is written into a MIDI file and can be listened to. This functionality is provided by the *interpolate* method of the *VAE* class. An example MIDI file as well as MIDI files with loops of the original bars are included with the examples. it is also possible to sample the learned distribution around a given bar to produce different but similar bars. This is possible with the *sample* method of the *VAE* class.

V. Summary

The variational autoencoder proved to be a model capable of learning a meaningful latent representation of the dataset. From this, we we able to create new drum grooves by the means of interpolation between bars and sampling around a given input. Unfortunately we did not have enough time to look into a time evolution model using a recurrent neural network such as a LSTM, which could be done in a future project. We think this is an interesting topic which could lead to new impulses for the creative world.

References

[1] J. Gillick, A. Roberts, J. H. Engel, D. Eck, and D. Bamman, "Learning to groove with inverse sequence transformations," *CoRR*, vol. abs/1905.06118, 2019. [Online]. Available: http://arxiv.org/abs/1905.06118

[2] A. Roberts, J. H. Engel, C. Raffel, C. Hawthorne, and D. Eck, "A hierarchical latent vector model for learning long-term structure in music," *CoRR*, vol. abs/1803.05428, 2018. [Online]. Available: http://arxiv.org/abs/1803.05428

[3] A. Khan, "towardsdatascience.com," Dec 2018. [Online]. Available: https://towardsdatascience.com/generating-pokemon-inspired-music-from-neural-networks-bc240014132