NESGAN

A Generative Autoencoder Network to Produce NES-Style Music

Ethan Schneider

SUNY Polytechnic Institute
Undergraduate Computer Science Capstone
Capstone Advisor: Dr. Michael J. Reale

Fall 2020

1 Introduction

This project is a generative adversarial network or GAN, that was trained to generate NES-style music, which was implemented using Tensorflow [5]. The way this works is that it takes in note data, pre-formatted from an NES Music Database [7] (or NESMDB), trains to that file to generate things that sound like it, and then output said data. As such, a way to export said data as listenable files was also needed. To implement this, we use the python package included in NESMDB [7, 6] to convert the data to .vgm files, then using VGMplay [2] to convert said files to .wav files.

This project was written to generate video-game style music to listen to. The choice was made to use NES music instead of modern-day music because music from the NES has is limited in size and scope due to the system limitations of the NES itself [7]. Also, music from the NES has a specific style, and it is interesting to see if the machine learning software could learn that style.

This is important as a whole because GANs have been used very often to generate visual data. GANs were initially implemented to generate photographic data [8], so generating is an interesting idea.

2 Related Work

There are many technologies used in this project that important to understand. Machine Learning as a whole is a still-evolving field in Computer Science. As such, it is important to understand the dataset and technologies used, alongside various related implementations of the data.

2.1 NES Music Data

The data as used in the Nintendo Entertainment System or NES is formatted into four channels. There are 2 pulse wave channels, a tringle wave channel, and a noise channel. On top of that, the NES also has an audio sampling channel which was ignored in the dataset provided.[7]

From the various data formats in the dataset, it was decided to use the Separated Score format, which presented the song data in an Nx4 NumPy array, where N is equal to the length of the song in notes. Each of the 4 noise channels also has a range the notes can appear in, the two pulse channels can appear as 0, or be between 32 and 108, the triangle channel can appear as 0, or be between 21 and 108, and the noise channel can be between 0 and 16. All of these channels store their notes as integers [7].

This data also comes with a rate in Hz, representing the playback rate of the notes, and a variable representing the number of note samples in the sequence. Although the number of samples changes from sequence to sequence, the rate remains the same. For the separated score dataset I used, the rate is set at 24 Hz, however, for the non-score formats, the rate is a higher 44.1 kHz. This is because the score formats were downsampled from the original .vgm format to be easier for the computer to handle in machine learning [7].

Included with the dataset is a python package used to transfer the data into various formats, between midi, the score formats, and the raw .vgm format. This package also installs VGMplay [2] alongside it to convert from .vgm to .wav, however, the pip install only works on UNIX machines, so to use the package, one has to install it manually [6].

2.2 GANs

GANs, or Generative Adversarial Networks, is a form of machine learning networks that take a dataset and train a generator on that dataset to generate works that appear similar to the dataset [8]. So that's the G in GAN, but the adversarial part is alongside training a generator to create works, another model called a discriminator is also trained to tell if a work is faked or not [8].

The generator itself generates data based on a random noise vector, which is then passed through various layers depending on how the network is set up, to eventually be reshaped into your training shape. The discriminator can then take this data, compare it to real data, and tell which one is real and which one is false. As the generator gets better and the discriminator gets better, theoretically you eventually reach a point where the generator produces data that is virtually indistinguishable from the original dataset [8].

The discriminator and the generator are trained at the same time in a training loop, with the discriminator being trained first and then the generator being trained. Besides, the discriminator may be trained for several sub-cycles in the dataset, to solve issues where the discriminator is performing worse than the generator [8].

2.3 LSTMs

LSTMs, or Long Short Term Memory, are layers that can learn data sequentially. Related to our dataset, they are very useful to machine learning training when dealing with data in a sequential form, such as sentences or music. This is because the LSTM layers can remember data over an arbitrary interval of time and make guesses on the data based on the current data and the remembered data [9].

In Tensorflow, LSTMs can also be passed into a Bidirectional Wrapper, which feeds the data simultaneously forward and backward through the model, essentially meaning the LSTM can make guesses on both the past information and also the future information on that particular data point [5, 10].

2.4 C-RNN-GAN

This is an implementation of a GAN to generate music based on MIDI files, those of classical music. This GAN generates music based on a simple system where The generator generates a random noise vector, which is then passed into several LSTM layers the output of each is passed into individual fully connected layers representing the notes, all of which are trained so the generator produces music. The discriminator then takes each note, passes them into a bidirectional LSTM, and finally takes those outputs into a fully connected layer to determine if the output [11].

A special thing about this network is that it uses feature matching, or instead of training the generator to avoid the discriminator detecting it as correct, instead the generator is trained to have the output of it match the discriminator's output of real data. This means that instead of having discriminator output that guesses if the model is correct, instead of the some metric that the generator tries to match [11].

2.5 Pokemon Music Generation

The last of the models that were researched, this paper presents several neural networks that were trained to single-instrument midi tracks of music from the Pokemon series of video games. In the paper, a simple LSTM network was presented, alongside a more complex GAN network [10].

A lot of inspiration was taken from this paper's GAN structure and used as a starting point to develop the GAN described below, however, A major difference is that the model described in this paper deals with multi-instrument score tracks, whereas the Pokemon GAN works with single-instrument MIDI files. Also, this dataset only has a training set, whereas the model below has options to use training, testing, and validation set, to avoid overfitting the data [10].

3 Method

The code of this project was split into several steps, first, the code needed to be formatted to be properly inputted into the network, next to the code of the network needed to be designed, and finally when the network generates results, said results need to be formatted back into the separated score format, and then they needed to be made into .wav files. For the most part, this codes is written in Python 3, however, the parts that implemented the nesmdb package[6] needed to be implemented in Python 2, as it's not converted for Python 3 yet. The functions in Python 3 and the GAN class are declared in functionDefs.py. Then, to run the class, various ipython notebooks are given. Loading the dataset, GAN training, and saving are done in the notebook GanTraining.ipynb. Besides, the file GanGenerate.ipynb will load a checkpoint from the GAN and generate files from that checkpoint. These files, describing the files below, can be found in the GitHub for the project [1]

[1] https://github.com/plaidScience/NESGAN-capstone

3.1 Preformat the Separated Score

The first issue that needed to be tackled is taking the dataset from its default form and making it into something the network can use. The data itself is stored individually in python compressed .pkl files, where there is one for each song. A function called `openScore()`, was implemented to extract a single .pkl file and return the NumPy array stored in the file alongside the stored number of samples and rate.

Building on top of that function, the function `openFolderSplit2()` was implemented to programmatically run the `openScore()` function for each file in a given folder. Options in the function were also made to normalize the data from the file between -1 and 1, split each file on a certain number of notes, and decide where the split files overlap, such as if you were to split the array [1, 2, 3, 4] into arrays of 2 notes, if the array overlapped you could therefore split the array into [1, 2], [2, 3], and [3, 4]. Then the function outputs one NumPy array of size NxKx4, where N is the number of given songs, and K is the size of each song, as given by the split size.

With the output of this function, we now have the data properly formatted for input into our Generative Adversarial Network.

3.2 The GAN

The GAN itself is defined as a python class, with an initialization function, `__init__()` where you declare the size of an individual song datapoint. The function will then create a discriminator and generator, with discriminator input and generator output scaled to the size of the individual data point. The discriminator also is configured to output accuracy alongside loss.

Also passed into the initialization function is a folder directory to append to the ./logs and ./checkpoints directory, as to differentiate from other logs and checkpoints. The init function then sets up the TensorBoard logging and the checkpoint saving.

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i + log(p_i) + (1 - y_i)log(p_i)]$$

Figure 1: Binary Cross Entropy Loss

The generator and discriminator's loss functions are also defined in the initialization function. They are both simple binary crossentopy losses, as defined by TensorFlow[5]. The functions of which are given in Figure 1. In addition, the generator and discriminator are both optimized with Adam optimizer, with a learning rate of 0.0002 and beta_i of 0.5 [5].

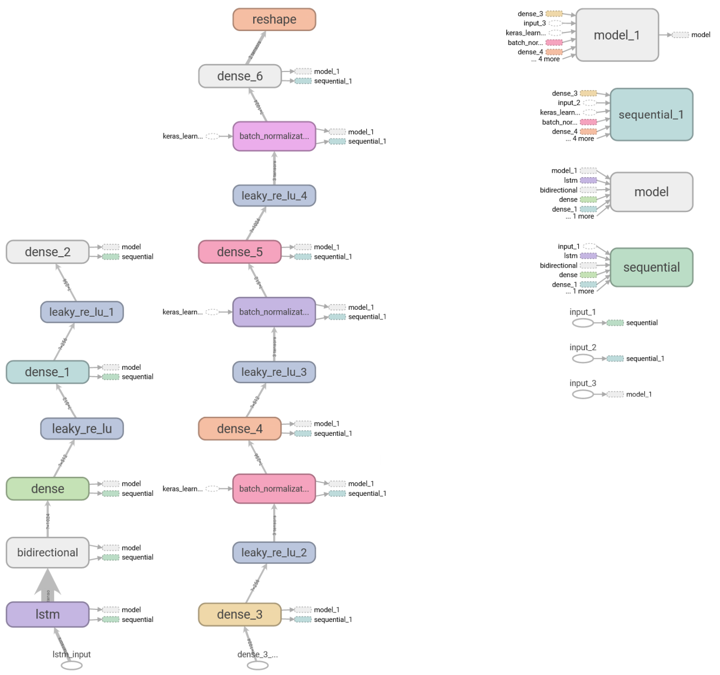


Figure 2: The graph of the Generator and Discriminator model layers, as given by TensorBoard

Next in the class are various functions creating the generator and discriminator. The full structure of each is given in Figure 2[2]. After that are the functions the class uses to train. The first of these that describes one step of training is called `train_step()` It takes a batch of songs and a batch size, generates some sequences to train the discriminator with, then it trains the discriminator on the inputted batch and generated batch, outputting the loss and the accuracy of each. The losses are averaged together, however, the accuracy of each is outputted separately. Then, the function will generate a new set of sequences and train the generator on that. After this function, there is a `test_step()` function, which takes all the same inputs of `train_step()` and gives all the same outputs, but instead, it tests and doesn't train, so it won't update the generator or discriminator. After this, there is a pretraining function for the generator and discriminator which were implemented but not used, called `pretrain_gen_step()` and `pretrain_disc_step()` respectively. These functions just run the same training that would be done in normal training, but only for the discriminator or generator. The generator pretraining does not need any batch passed in, it just needs the batch's size.

The next two functions, `writeLog()` and `writeTrainLog()`, just write train-

[2]The full graph of which can be found on the Experiment's TensorBoard

ing and testing logs to a log function, taking in the logs, the log names, and the function to write the log to. The `writeLog()` function for training also needs an ID to write the log to, in most cases it will be the epoch of the log. These functions were made by referencing several user-made functions on gitHub [3, 4][3][4]

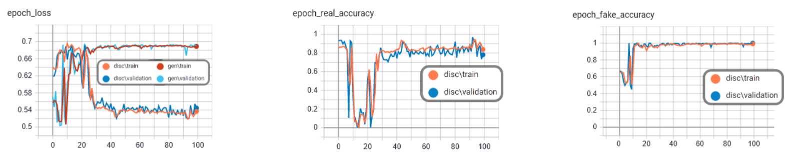
After that is the `train()` function itself. This function takes in a training set, a testing set, a validation set, the number of epochs to run for, the starting epoch to run on, (in case of loading from a checkpoint but still wanting to write out to the same log files,) an interval to sample the dataset too, and an interval to save the dataset to. This function runs for a certain number of epochs. For each of those epochs, it will divide the training set into batches and train on each of those batches, averaging each loss and accuracy after all the batches have run. Then if there is a validation set it does the same, but instead of training on the training set, it tests on the testing set. After that, it tests if the epoch is at the saving and logging interval, and saves and logs respectively the set if it is. Then, if there testing set, it will test it for each batch in the set and log the results. Finally, it will output the generator's and discriminator's training loss in a Matplotlib plot. There is also unused pretraining for the generator and discriminator that will just pretrain one for a set number of epochs. The way `train()` saves models references the TensorFlow tutorial on how to save models [1][5].

After this, there are some functions to generate a result from the GAN, which will output a generated set of notes, formatted with a rate of 24Hz and the number of samples equal to 4 times the output shape. These functions are implemented as `generate()` and `generateScaled()`, the second of which scales the music back to the default length for each channel.

3.3 Data Postformatting

Using these outputs, the data can then be formatted into either a CSV file or compressed into .pkl files. To compress it into a folder of CSV files, the function `saveFolderCSV()` is called, this will save the given scores into a folder, creating said folder if it does not exist. In the same way, you can call the function `saveFolder()` to save the seprsco as .pkl files into a folder.

Then, to utilize nesmdb's conversion functions, you load the folders in a Python 2 environment, call the nesmdb functions on each file, and they are saved in .vgm and .wav files. This is done in GenerateVGM.ipynb. The functions in this notebook are given an input and output folder, reading from the input folder and writing it out to the output folder.



(a) GAN Loss (b) Real Accuracy (c) Generated Accuracy

Figure 3: TensorBoard Output for Loss and Accuracy

6

4 Results

The results of the training are as seen above in Figure 3, as generated by TensorBoard,[7] These are results generated at the end of every epoch of training. As you can, see the accuracy of the generated images (3c) and the accuracy of the real images (3b) are both high at the end of the training, meaning that the training itself did not go well. Alongside that, the loss (3a) plateaus at around 1. This is a sign that the training did not go well.

Another metric from the results is the generated data from the generator. The Result of this can be found on the experiment's gitHub[8]. The generated .pkl files are found in the saved folder, and generated .csv files can be found in the savedCSV folder. Output .vgm and .wav files can be found in the savedVGM/ and savedVGMCSV/ folders for the .pkl and .csv results respectively. These files sound a bit like if you would take a noise machine and have it make NES music, another sign that the training did not go well.

5 Discussion

In this process as a whole, I learned a lot. I learned how to formulate a project idea and design a project based on said idea. Alongside that, I learned how GANs work. Machine Learning as a whole is very interesting to me, so it's nice that I was able to figure out how they work, especially as GANs are quickly becoming one of the biggest methods used in machine learning. Another thing I learned is how to schedule my time when working on a project by myself. I did have Dr. Reale help at the beginning give me a brief overview of how the project timescale should look, but after that a lot of the deadlines were self-imposed. That is something I feel that it is important I learned because having the ability to say to myself, "I need this done by then, then this done at this time," is a powerful tool for me to have for myself.

The construction and formulation of the GAN went very well. Once I had

[3]https://gist.github.com/joelthchao/ef6caa586b647c3c032a4f84d52e3a11
[4]https://gist.github.com/erenon/91f526302cd8e9d21b73f24c0f9c4bb8
[5]https://www.TensorFlow.org/tutorials/keras/save_and_load
[7]The Experiment's TensorBoard
[8]https://github.com/plaidScience/NESGAN-capstone

the basic format down, it was easy to add things such as TensorBoard callback, checkpoints, and even extend the training to also perform testing and verification. Another thing that went well is formatting the data. Once again, it took a bit to get a handle on how the dataset dealt with values, however, it became pretty easy to digest that information as things went on. Some things did go wrong in the process though, as I didn't realize right away how the sound data was formatted, and so I was passing illegal data out of the Network, which caused some issues when rendering the data, but I programmed the GAN in a way that made it very easy to tweak the output of that function.

Getting on to things went wrong, there was an issue where I could not train the GAN on my home computer that well, as the batch size was very low so it wasn't able to process the data very fast. I ended up solving this by using the school's big-bear computer, but it still was something that came up as an issue. Another issue that occurred was that the training ended up failing due to the discriminator being very good at telling what is a real song and what is a faked song, so eventually, there is a point where the generator can't get any better, so it just stops training to any meaningful extent. This could be solved with generator pretraining, where the generator is pretrained for some amount of epochs. It could also be solved by implementing training sub-cycles where the generator is trained to a greater extent in each training step. These all take more time to implement, which is a luxury I learned quickly runs out in these sorts of projects.

There are positives and negatives in my implementation of the GAN. In particular, To receive any audio output, you need to have two python environments, which is indeed a huge problem. However, the GAN was implemented as a class with each major part implemented as a function, so if you wanted to tweak it, you could just implement a class that inherits it and overrides anything you want to change. Some other positives are since all the functions are in an external class, the network's running is separate from the network's definition. This is something that I feel is a big problem, especially in python, where people implement a novel way of performing a task, but it's declared in the file that runs it. This means that someone has to re-declare it in a future file some time down the line if they want to reuse it. In my system, if I want the GAN to be used in multiple files, I just re-import the main function definition file, and that's it. However, this does have some flaws, as to a certain extent it does become unwieldy to step forward and back between the files changing things, but I feel that it's beneficial enough that I kept it the way it is.

If I were to redo this project, I would implement a way to load a more universal file format, like MIDI. The Separated Score files are interesting, but midi is used a whole lot in the music world, and having a way to process those files would be very useful for even expanding my dataset. Besides, I would tighten up the training loop. Currently, a lot is going on per-epoch and per-batch that is both repeated and could be made quickly that would end up exponentially increasing the training amount. Another thing I would redo is to find a better way to scale the data, as currently, I'm just scaling is based on the maximum value, and if I instead scale it based on the minimum and maximum, it could cause the GAN to produce more meaningful output.

6 Conclusion

In conclusion, this GAN needs a bit more work to produce better output, however, It is producing output, and it was implemented in a way that means that any additions can easily be added on by anyone else who wants to work on it in the future. Although the output is bad, it's still built on a stable foundation, which is a good thing.

6.1 Future Work

Some future work that could be made on the project is adding a way to easily pretrain on the dataset. Implemented with the generator, this would allow the generator to hopefully work on the data better and cause it to compete with the discriminator. Another thing to add on to is to modify the training steps to add some sub-cycles into training to hopefully also achieve the same benefit.

Something else to work on is also implementing a way in Python 3 to receive audio output, however, that requires either that you implement nesmdb in Python 3 or find a reasonable substitute. Alternatively, if one could find a way to transfer the score format directly to .wav, that would be important.

Another possible future work on this is extending it to other retro game console music, as NES isn't the only one, and they all have their limitations and specifications to deal with. Such as increased audio channels, a larger audio bit amount, and increased effects to add to audio. However, and implementation of the last one also requires an implementation of effects on the main model, which is a different issue in itself.

7 Appendix

All the code programmed for the project is included on the gitHub[9] however, the program still needs the prerequisite platforms and packages to run. As explained below, first you need to set up the two python environments, then you need to install the needed packages for each environment.

7.1 Setting up the Python Environment(s)

To be able to run the program, you will first need a python environment. You will need two python environments, a Python 3 environment to run the training and generation program, and a Python 2 environment to run the conversion software. I recommend using conda as your environment manager, as it is a powerful command-line tool to create, edit, and switch environments. To install conda and set up your environments:

[9] https://github.com/plaidScience/NESGAN-capstone

1. You first install it from the install page on the website[10] and follow the instructions on said site for the specific system you are running.

2. Then you need to run the '`conda create`' command on either your Anaconda Prompt if you are on Windows, or from your command line or terminal if you set conda up for your command line on Windows or if you are on macOS or Linux. This command is formatted like '`conda create -n ENVNAME python=VERSION`'.

   (a) To create the Python 3 environment, you could run '`conda create -n py3 python=3.6`'

   (b) To create the Python 2 environment, you could run '`conda create -n py2 python=2.7`'

7.2 Setting Up Training Packages

Now, setting up the Python 3 environment for training is very simple. The first thing to do is run the conda command '`conda activate py3`', where `py3` is your name for the Python 3 environment. Now that you are inside the environment, you can install the packages you need.

1. You will first need to install TensorFlow, this can be done by running the pip command '`pip install tensorflow`'

   (a) If you have a GPU that can be used for machine learning, you can also follow TensorFlow's guide[11] to set up training on your GPU.

2. Next, you can install various other packages needed for the programs. These are as follows: NumPy, MatPlotLib, and scikit-learn, and can be installed with the '`pip install`' command like so: '`pip install numpy matplotlib scikit-learn`'

3. Finally, the Jupyter package needs to be installed to run the notebooks. The command to install is '`pip install jupyter`'

   (a) I also recommend the nb-conda-kernels package, as it allows you to run Python 2 notebooks from your Python 3 environment using Jupyter-Notebook

Finally, you can run the command '`jupyter-notebook`', which launches the Jupyter Notebook and sets up a localhost:8888 tab for your local browser to run the Jupyter software on. After which, you can open and run the GanTraining or GanGenerate notebooks, to train a model or generate based on a saved model respectively.

[10]https://docs.conda.io/projects/conda/en/latest/user-guide/install/
[11]https://www.tensorflow.org/install/gpu

7.3 Setting Up Conversion Packages and Programs

Now, to set up your Python 2 environment, you have to then activate it using the conda command '`conda activate py2`', where `py2` is your name for the Python 2 environment, like how you activated the Python 3 environment in 7.2

1. Then, you need to install nesmdb, which on a Linux system can be installed with '`pip install nesmdb`', however, if you are not on a Linux system you can instead install the package from the Database's gitHub[12] and place the package folder in your project folder. If you are going this route, you will also need the NumPy and Subprocesses packages, installed with '`pip install numpy subprocess`'

2. After this, you need to install VGMPlay, which can be taken from it's vgmrips forum page [13]. When you unzip the file, the program will expect it's contents to be in a subfolder of the project folder called vgmplay/

   (a) To configure it for .wav output, you will need to change live 42 of the VGMPlay.ini file so that it reads `LogSound = 1` instead of `LogSound = 0`. This means that VGMPlay.exe will write out .wav files instead of playing the sounds of the .vgm files.

3. Finally, if you are not using nb-conda-kernels, you will also need Jupyter for this environment, installed the same way as in 7.2. If you are using nb-conda-kernels, you just need the iPython kernel in this environment, installed using '`conda install ipykernel`'

Finally, you can either run the `jupyter-notebook` command again in this environment or just run the `jupyter-notebook` command in your python 3 environment and run the GenerateVGM notebook.

7.4 Installation Summary

In Total, to the steps to install the prerequisite software are as follows:

1. Set up Python Environment
   (a) Install conda
   (b) Create a conda environment for Python 2 and Python 3
2. Set up Training Environment
   (a) Install Tensorflow
      i. Configure GPU Training (optional)
   (b) Install Numpy MatPlotLib, and Scikit-Learn

[12]https://github.com/chrisdonahue/nesmdb
[13]https://vgmrips.net/forum/viewtopic.php?t=112

   (c) Install Jupyter
      i. Install nb-conda-kernels (optinal)

3. Set up Conversion Environment
   (a) Install nesmdb and it's prereqisute packages
   (b) Install, configure, and place VGMPlay in it's correct folder
   (c) Install Jupyter or an IPython Kernel if you're using nb-conda-kernels

References

[1] Save and load models | TensorFlow Core.

[2] vgmrips/vgmplay, December 2020. original-date: 2014-11-08T15:49:06Z.

[3] 262588213843476. Keras uses TensorBoard Callback with train_on_batch.

[4] 262588213843476. train_on_batch_with_tensorboard.py.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Chris Donahue. chrisdonahue/nesmdb, December 2020. original-date: 2018-04-02T22:26:13Z.

[7] Chris Donahue, Huanru Henry Mao, and Julian McAuley. The nes music database: A multi-instrumental dataset with expressive performance attributes. In *ISMIR*, 2018.

[8] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. *arXiv:1406.2661 [cs, stat]*, June 2014. arXiv: 1406.2661.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[10] Abraham Khan. Generating Pokemon-Inspired Music from Neural Networks, December 2018.

[11] Olof Mogren. C-RNN-GAN: Continuous recurrent neural networks with adversarial training. *arXiv:1611.09904 [cs]*, November 2016. arXiv: 1611.09904.