"ALEXANDRU-IOAN CUZA" UNIVERSITY OF IASI

FACULTY OF COMPUTER SCIENCE

BACHELOR'S THESIS

Music generation using recurrent neural networks

proposed by

Florin Grigoroșoaia

Session: July, 2019

Coordinator

Assoc. Prof. Ph.D. Dragoș-Teodor Gavriluț

″ALEXANDRU-IOAN CUZA″ UNIVERSITY OF IASI

FACULTY OF COMPUTER SCIENCE

Music generation using recurrent neural networks

Florin Grigoroșoaia

Session: July, 2019

Coordinator

Assoc. Prof. Ph.D. Dragoș-Teodor Gavriluț

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Music generation using recurrent neural networks**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea "Alexandru Ioan-Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Florin Grigoroșoaia**

Data: .... Semnătura: ....

ACORD PRIVIND PROPRIETATEA DREPTULUI DE AUTOR

Facultatea de Informatică este de acord ca drepturile de autor asupra programelor-calculator, în format executabil și sursă, să aparțină autorului prezentei lucrări, Grigoroșoaia Florin.

Încheierea acestui acord este necesară din următoarele motive:

Lucrarea de față este realizată folosind date confidențiale din cadrul companiei Bitdefender. Încheierea acestui acord este necesară deoarece codul sursă se află sub o clauză de confidențialitate între autorul lucrării și compania Bitdefender SRL.

Iași, Data ....

Absolvent **Florin Grigoroșoaia**

Decan **Iftene Adrian**: ....

Semnătura:
....

Contents

Abstract

Artificial intelligence technologies have been developed recently in almost each domain. The power of AI is present now in biology, security, automotive, having an impact on our daily life, aiming to raise its quality, help humans finish their tasks and creating new jobs. Reaching the most influential areas in the world, AI can be used also in media, such as news, movies and music. Artificial system can interact with music, learn its features, develop the ability to recognize or to find the genre of a song, but the most fascinating use would be to also generate it.

The idea of music generated by a computer could lead to the misunderstanding of AI's usage and the purpose of this thesis is to present a bound relation between human beings and an artificial system. The project aims to create a tool that can be used by artists to generate a song background.

Motivation

The number of people interested in artificial intelligence has raised exponentially, so did the number of artists that take into account the possibility to create a part of a song, a hit or even an entire album. A tool that has this feature could increase the chances of a gifted singer, but with less inspiration in creating the background of the song. In other words, singers could create their negative without a big amount of music theory knowledge, add their own voice and in the end obtain a song.

A human being can learn how to combine notes and frequencies or learn to play instruments, but this can require years of practice. A machine can learn all of these: different instruments, different music genres, even discover what feelings a song can send to people; and it can learn them in hours or days. Also, a real problem for artists and musicians is the lack of inspiration and the music consumers require something new every time, so adaptation to what they desire is required. For an artificial system the only problem is the input data; it needs to learn from the environment, as humans need too, but the advantage is that it can do it way faster and the information never disappears once it has been received. Having an entity that has music knowledge means that it can create based on what it has 'heard'.

The process of creating something entirely original is based, both for humans and artificial systems, on previous events, the environment and all information it has ever received. In other words, nothing new is entirely new; it is based on history. The human brain can retain data, such as a part of a song without us even noticing(when our attention is caught by a main activity, but in the background there is a radio playing). A piece of that information could be further used when creating a new song, even a tiny fragment without us knowing where it came from, but somewhere that data is stored in our brains. So does the artificial music generator work, it learns from a vast amount of information and uses insignificant pieces from each element.

Introduction

Artificial intelligence

Artificial intelligence is a sub-component of computer science that aims to create intelligent systems, that are modelled after human brain's intelligence.

Intelligence can be interpreted in many ways, several definitions being written throughout the years, Collins English Dictionary describes it as:

> *the ability to think, reason and understand instead of doing things automatically or by instinct.* [1]

Most importantly, intelligence can refer to the capability to learn, develop, acquire new skills, gain knowledge and the main action: to be able to front unprecedented situations. An intelligent system is capable to react to a new case and could also learn from it, by receiving a feedback from the environment whether the decision it had taken was a positive or negative one.

Machine learning

Machine learning is a field of Artificial intelligence with the purpose of creating these kind of systems, that have the ability to learn, to makes decisions and predictions. The difference by a simple artificial system and an intelligent one (created with the help of machine learning) is that the first one only makes decisions by following a pre-defined protocol:

[1] `https://www.psychologytoday.com/us/blog/hide-and-seek/201811/what-is-intelligence`

```
if first_situation_is_met:
    do first_action
elif second_situation_is_met:
    do second_action
else:
    do default_action
```

The second type of system described above learns from input data, which depends on context, but mainly is an enormous amount of information that is fed to a computer. This data is processed many times, each iteration various algorithms can be run and the system decides using an existent method how to adjust what is better to learn, with the knowledge of its target.

Artificial neural network

Artificial neural networks can be defined by a set of learning algorithms, inspired by the human brain, that aim to find and learn patterns from given input. An artificial neural network model consists of nodes, that are called neurons. A layer is compounded of a set of nodes and information circulates from layer to layer.

Deep learning

Deep learning is a component of machine learning that uses 'deep layers' in order to find more features of an input. Deep learning is based on artificial neural networks and some examples that are used in this category can be:

- deep neural networks
- deep belief neural networks
- recurrent neural networks
- convolutional neural networks

Recurrent neural network

Recurrent neural networks are a type of artificial neural networks that use memory (their internal state), taking as input the data they receive and also the memory, a previous state in time. They differ from simple neural networks(feedforward networks) by this new feature called memory. In the moment an output is generated, it is immediately saved 'in memory' and sent next as input to another step, creating a loop, allowing information to persist throughout the network.

Long short-term memory

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning.[2]

A long short-term memory network is a recurrent neural network so it also maintains the feature of memory, but it has the capability to learn long-term dependencies. In a LSTM model, information are sent to gates, that decide, using a sigmoid layer(with output between 0 and 1; 1 means keep this in memory, 0 the opposite) whether to keep it or not and further it is stored, if it is necessary, in a cell state. The cell is a container for data, that keeps track of a correspondence between input and output and defines decision gates.

Generative LSTM

A LSTM was initially created for predicting next sequences in data, but can be also used as a generative neural network.

Given a large corpus of sequence data, such as text documents, LSTM models can be designed to learn the general structural properties of the corpus, and when given a seed input, can generate new sequences that are representative of the original corpus. [3]

A recurrent neural network for music

The purpose of this thesis is to build a recurrent neural network model using a long short-term memory architecture that is able to generate music based on previous learned features. In the following chapter, details about the data that has been used to

[2]Gers, Felix (2001). "Long Short-Term Memory in Recurrent Neural Networks"

[3]`https://machinelearningmastery.com/\gentle-introduction-generative-long-short-term-memory-networks/`

train the models are provided. Furthermore, this data will have to be processed and reshaped so a special chapter for data processing comes after the details about data.

For developing this project, TensorFlow was mainly used, so a few details about the library and the configurations graph are introduced.

Later on, the model used will be presented along with the training, evaluation and generation steps.

Reaching the end, several pre-trained models are put-forward, in order to highlight the result of the training and to show differences between datasets and music genres.

As the idea of a music generator requires a practical use, a web application has been developed and the end focuses on how the application was implemented and how an artists can turn it to account.

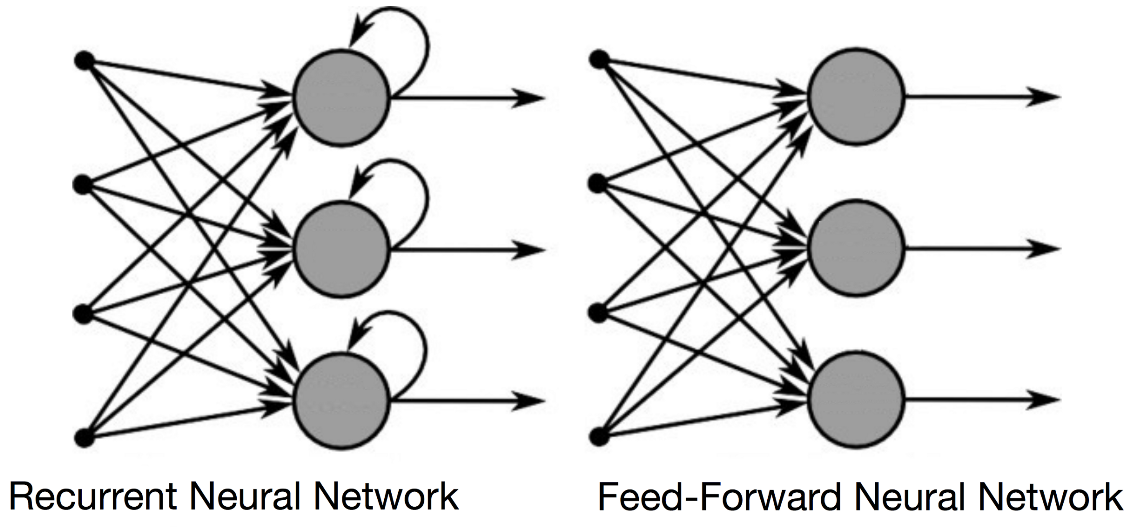

Figure 1: RNN and Feedforward NN

Personal contributions

During the development of the project, the personal contribution consists in an initial part of research about sound, how it is stored, MIDI files, generative neural networks and how they can be combined. In addition, the implementation of the project is the main contribution, writing the dataset creation, training and generation parts for the project.

Another important contribution is constituted by the training sessions, that required perfect previous development because this process takes a huge amount of time and mistakes only slows the training job by stopping, modifying code and starting again. This part required formerly data, so sets between five hundred and one thousand files had been selected, created and prepared for training. On top of all of these, the final contribution is the web application, that is a smooth level between the end user, that can be an artist, and the neural networks project.

Chapter 1

Dataset

1.1 Dataset

A machine learning algorithm requires data to gather knowledge. This represents the learning dataset used in machine learning algorithms and it is used in the process of learning, a process where an artificial model, which is a a data structure, iterates a huge number of times the dataset and accumulates information from each entry. Usually a dataset entry has a number of features, for example an image can be divided into a matrix of pixels and each one represents a feature; also the entry can be labeled, this means that there exists a mapping between each dataset entry and a value that represents a class the respective entry belongs to. As for the previous example, images can have labels such as digits, animals or other classes that represents what an image shows.

In order to reach to the dataset for the music generative recurrent neural network, the initial set for learning has to be presented and also the modification process it was passed through needs to be explained. A generative music model has to learn from actual songs to be able to create novel ones. A simple entry in the initial set is a MIDI file.

1.2 Midi

MIDI(Musical Instrument Digital Interface) is a communication protocol that allows musical instruments to exchange data between them and with computers. MIDI does not deliver actual audio signals, but instruction messages that contain details about the pitch, intensity. They can also transmit control signals for sound volume and vibrato. Midi is also a file format which stores and exchanges data. The advantages of a MIDI file

> *include small file size, ease of modification and manipulation and a wide choice of electronic instruments and synthesizer or digitally-sampled sounds. A MIDI recording of a performance on a keyboard could sound like a piano or other keyboard instrument; however, since MIDI records the messages and information about their notes and not the specific sounds, this recording could be changed to many other sounds, ranging from synthesized or sampled guitar or flute to full orchestra. A MIDI recording is not an audio signal, as with a sound recording made with a microphone.* [1]

1.3 Note sequences

Note sequences are actually protocol buffers,

> *Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data* [2],

a format easier to work with that MIDI files. The way data is serialized in a protocol buffer is specified by the user by creating pairs of key, value elements that represent protocol buffer messages. This is the way the user can communicate with a protocol buffer about how information is serialized. Protocol buffers are simpler, smaller and faster than XML, using easy object oriented representation for the user.

Note sequences are created using the protocol buffers from Google and define fields specific to music:

- Note: contains details about pitch, velocity, start time, end time, voice
- Time signature: contains details about time, numerator, denominator
- Key signature: contains details about time, key, mode
- Tempo: contains details about time and quarters per minute
- Pitch bend: contains details about time, bend, instrument, program
- Control change: contains details about time, quantized step, control number control value, instrument, program

[1] `https://www.midi.org/`
[2] `https://developers.google.com/protocol-buffers/`

- Part info: contains details about part
- Instrument info: contains details about instrument
- Source info: contains details about source, score and performance
- Text annotation: contains details about chord and beat
- Quantization info: contains details about steps per quarter, steps per second
- Subsequence info: contains details about start time offset, end time offset
- Section annotation: contains details about time and section id
- Section: contains details about time and section id
- Section group: contains details about section id and section group

1.4 Convert MIDI files to note sequences

The first step in building the dataset for training is to convert the initial set of files in MIDI format to a protocol buffer type, Note sequences. Having the set of MIDI files as a simple directory, this has to be recursively parsed and each entry needs to be converted to a protocol buffer. First of all the MIDI file is read with **pretty_midi**. The next step is to convert the MIDI file encoded as a string into a Note sequence. The header of the note sequence will contain the ticks per quarter and the encoding type which is MIDI.

All midi time signatures have to be collected from the file and added to the Note sequence object. Also, the key signatures from the MIDI files are moved into note sequence, specifying the mode: major or minor.

A further step is to append tempo changes of a MIDI file, that contain as fields a time and a quarter per minute.

By iterating all the instruments of a MIDI file, notes are gathered for each type and then added to the Note sequence object with all their information: instrument, program, start time, end time, pitch, velocity, a flag specifying whether it is a drum or not.

1.5 TensorFlow record

In order to programmatically load and use the current dataset, there is the need to keep all the note sequences inside a file format that can be easily parsed later.

1.5.1 TensorFlow

TensorFlow is the main library used for learning, evaluating and generating music and is

> *an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.* [3]

The version used for the project is TensorFlow 1.13.

1.5.2 TensorFlow record file

TensorFlow has a file format for storing binary records that is simply called - TFRecord. The advantage of storing the songs information compressed in such a file type is that data can be efficiently read because all the protocol buffers(note sequences) obtained from the first step(converting MIDI files to note sequences) are serialized and easily accessible from a TFRecord file.

In order to create a TFRecord file, TensorFlow provides a TFRecord Writer class that can be easily inherited and it requires only a write method to be overridden:

```
import tensorflow


class NoteSequenceTFRecordWriter(tensorflow.python_io.TFRecordWriter):
    def write(self, note_sequence):
        tensorflow.python_io.TFRecordWriter.write(
            self, note_sequence.SerializeToString())
```

From the snippet above, it can be seen that a TFRecordWriter contains binary data, so the note sequences have to be serialized first and then, the super method from TensorFlow TFRecordWriter can be used. To prepare the dataset for the next processing step, the list of all note sequences obtained from the initial MIDI files is iterated and the **write** method is called for each protocol buffer entry. Every binary record constructed

[3]https://www.tensorflow.org/

represents a chunk from the final container of the TFRecord. Each chunk is written to a local file with extension *.tfrecord* that acts as the respective container.

1.6 Used data

The first dataset used, for the first music category, jazz, is compounded of 926 MIDI files, with a total size of 36MB. All these files have been converted to Note sequences, resulting a TensorFlow record file that occupies 140MB. The time for this operation was one minute.

The second dataset belongs to classic music and has a total size of 29MB, containing 306 files, but with an average duration of 17 minutes, in comparison to the jazz MIDIs, that have an average duration of 3 minutes. They have been converted into Note sequences, with the result of a TensorFlow record file with the size equal to 95MB.

All these operations were run on a system with following configurations: *Intel Core i7-8700 CPU 3.20GHz, 16GB RAM, x64 architecture*.

Chapter 2

Data processing

2.1 Data processing

As described in the previous chapter "Dataset", the initial set compounded of MIDI files has to be processed before it can be used as a learning dataset for the recurrent neural networks model. As a first step, MIDI files were converted to an easier format, note sequences, and all the information in each MIDI entry has been passed to its correspondent protocol buffer.

In order to easily process big data, a data structure that takes an input type and converts it into an output type is required. Because the dataset will be submitted to many transforming operations, this data structure must be able to work in a chain of actions, sending the output of the latest executed to the following one. A concept that reminds of this workflow is the pipeline.

2.2 Pipeline

Pipeline is a technique of dividing a complex instruction into a number of linked phases. Each stage is linked to another one because step **i + 1** depends directly of the result of step **i**, but also depends indirectly of the output of step **i - 1**, **i - 1** providing input for **i**. Usually the first phase is the only one that does not require input from another phase(it might take it from a file) and the last one does not necessary have to send the output to another component of the pipeline(it can be written to a file).

Training data in machine learning can be time expensive, because the magnitude of the learning dataset can be high so quick conversion between different data types is mandatory, this is the reason a pipeline has to be introduced for the processing of note sequences(the protocol buffers obtained from initial MIDI files). Also, each intermediary data type can be easily sent through the pipeline and a physical(local save) instance does not have to be created. This would have brought time and hardware costs because each data type would have had to be serialized, written, loaded, deserialized, but using this data structure these kind of data can softly and rapidly flow through the pipeline. Having this process implemented, the problem left is to provide the input for the first phase of the pipeline and to store only the output of the last step. The next subsections present a detailed pipeline workflow, aiming to introduce the final state of the training dataset.

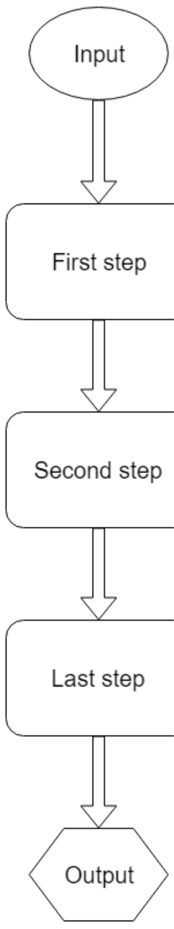

Figure 2.1: Pipeline

2.3 Base pipeline

Each pipeline type has to implement a Base pipeline class, but individually define the input and output types it requires.

The Base pipeline has three main fields:

- type of input
- type of output
- unique name

Also, each pipeline must implement the abstract method **transform(input)** of abstract **BasePipeline** class. This is an example of creating a new Pipeline type:

```
class CustomPipeline(BasePipeline):
    def __init__(self, input_type, output_type, name):
        super(BasePipeline, self).__init__(input_type,
                                           output_type,
                                           name)

    def transform(pipeline_input):
        ...
        return pipeline_output
```

2.4 Directed Acyclic Graph Pipeline

Knowing how to create custom pipelines, it means all pipelines for dataset processing can be created, but only a small detail has been left uncovered. Even if all instances of custom pipelines exist, they would have to be accessed separately, calling each **transform** method, taking care to preserve the correct execution order and defining the appropriate input and output types.

A workaround for this problem can arise if an external structure is aware of all data types and follows a defined flow. A structure as described has the form of a directed acyclic graph.

2.4.1 Directed Acyclic Graph

A graph can be defined as a data structure with a pair consisting in a set *V of vertices* and a set *E of edges*. The set of edges defines relations between pair of entries from the set of vertices. A directed graph is a graph defined previously with the mention that a relation from set *E* has a direction: taking *v1, v2* that belong to *V* the relation can be from *v1* to *v2* or from *v2* to *v1*; both relations can exist in a directed graph.

A directed acyclic graph is a directed graph defined as above where no relation circles exist. An examples of circle would be: taking *v1, v2, v3* that belong to *V*, there is a relation from *v1* to *v2*, a relation from *v2* to *v3* and a relation from *v3* to *v1*.

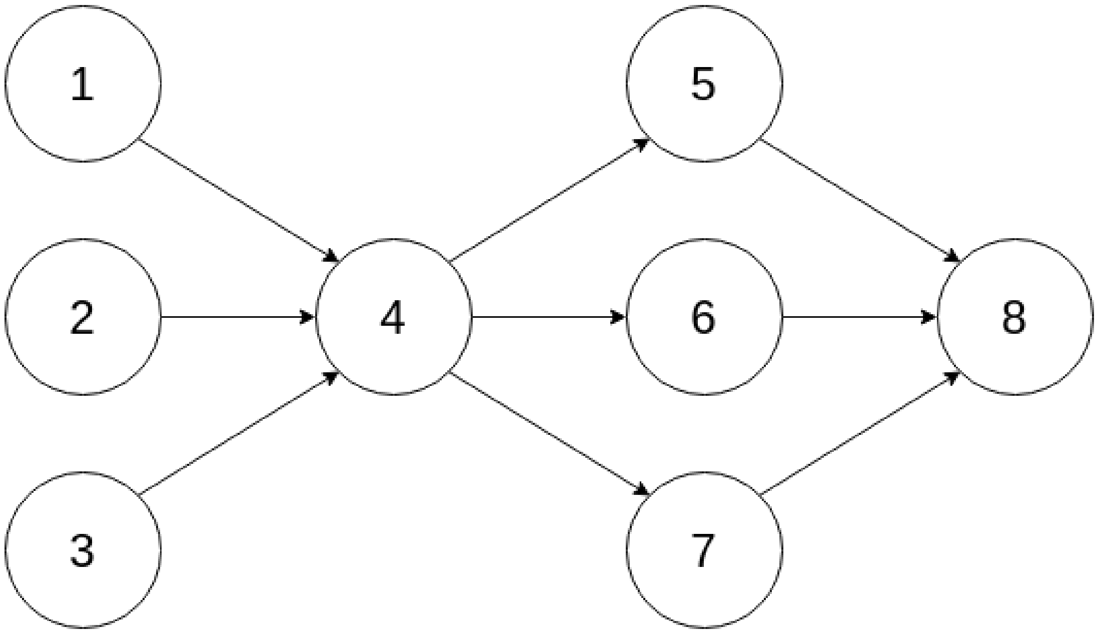

Figure 2.2: Direct Acyclic Graph

2.4.2 Directed Acyclic Graph Pipeline class

A structure as defined above can be implemented through a **DirectedAcyclicGraphPipeline** class that takes as input a config dictionary, where a pair is defined as follows: **(key, value)**: *(output_pipeline_instance, input_pipeline_instance)*. To get the pipeline started, a method that uses polymorphism and calls all the **transform** methods in correct order has to be defined. The whole pipeline workflow is presented in the snippet below:

```
pipelines_config = {
    first_intermediary_pipeline: input_pipeline,
    second_intermediary_pipeline: first_intermediary_pipeline,
    output_pipeline: second_intermediary_pipeline
}
dag_pipeline = DirectedAcyclicGraphPipeline(pipelines_config)
dag_pipeline.run_workflow()
```

2.5 DAG Input

The input for the Directed Acyclic Graph Pipeline is a container of note sequences, so for starting the workflow and creating the DAG Input, a TFRecord file is required. This is the file obtained after converting all MIDI files with a **TFRecordWriter**.

2.6 Partitioner

The next pipeline type is called a partitioner. The role of this pipeline is to randomly fragment the input into multiple output pieces. When creating the pipeline, a percentage value that represents the probability of the given input to belong to each of randomly created partitions can be sent. Also, the number of generated partitions can be set, with the mention that in this case, a list of probabilities that the input belongs to each partitions must be given. This pipeline can be used to create a set for each phase of the model: train, evaluation and testing.

2.7 Sustain pipeline

A sustain pedal or sustaining pedal (also called damper pedal, loud pedal, or open pedal) is the most commonly used pedal in a modern piano. It is typically the rightmost of two or three pedals. When pressed, the sustain pedal "sustains" all the damped strings on the piano by moving all the dampers away from the strings and allowing them to vibrate freely. All notes played will continue to sound until the vibration naturally ceases, or until the pedal is released. [1]

A sustain pipeline takes as input note sequences and applies sustain pedal control modifications. In the beginning, all *note on/off events* are sorted and sustained. Next, all events are iterated and notes that had been extended are ended and for an event with sustain on, end all previous notes that have the same pitch value.

2.8 Stretch pipeline

The stretch pipeline receives a Note sequence and creates one or many "stretched" Note sequences, depending on how many stretch factors the pipeline takes. Stretch factor:

[1] Edwin M. Ripin., "Sustaining pedal"., 2001

- value > 1.0: make the Note sequence slower by raising its length
- value < 1.0: make the Note sequence faster by lowering its length
- value = 1.0: do not stretch the Note sequence

Taking as input a Note sequence and a stretch factor, this pipeline:

- iterates the Note sequence's notes and multiplies the start time, end time with the stretch factor
- multiplies the total time of the sequence with the stretch factor
- iterates the Note sequence's event and multiplies each time with the stretch factor
- iterates the Note sequence's tempos and multiplies each quarter per minute with the stretch factor

For the current recurrent neural network model, the pipeline for evaluation receives a stretch factor equal to 1.0 and the one for training the Note sequences is stretched with the following stretch percentages: -2.5%, 0%, 5%, 2.5%, -5%.

2.9 Splitter

Splitter pipeline has the purpose to split a Note sequence at time intervals using a parameter hop seconds. This means that at each <hop seconds >interval the Note sequence will be split.

2.10 Quantizer

In digital music processing technology, quantization is the studio-software process of transforming performed musical notes, which may have some imprecision due to expressive performance, to an underlying musical representation that eliminates the imprecision. The process results in notes being set on beats and on exact fractions of beats. [2]

The next pipeline quantizes a Note sequence and takes as parameters the number of steps per quarter and the number of steps per second for the quantization operation(only one of these fields must be specified).

[2]Childs, G.W., IV (March 7, 2018). "A Music Producer's Guide to Quantizing MIDI"

2.11 Transposition pipeline

Transposition of a note means that to its pitch a parametrized amount will be added only if the pitch value belongs to a specified interval. This pipeline receives a list of amounts to perform transpositions and optionally a minimum and a maximum value for the described interval(default are the normal values for a MIDI pitch: 0 and 127).

For each amount value received, the Note sequence's notes are iterated and for each one, if the pitch values is between the minimum and maximum value given, the note is transposed with that amount(it is added that amount to the pitch, but with the mention that it can be negative). For each amount value, a new Note sequence is created. In the end the pipeline outputs many Note sequences with their notes transposed.

For the current recurrent neural network model, the pipeline for evaluation receives a 0 value (Note sequence is not transposed) and the one for training receives a list of transposition amounts: -3, -2, -1, 0, 1, 2, 3.

2.12 Extractor pipeline

Given a quantized Note sequence, this pipeline extracts sounds for given:

- start step: it starts extracting a sequence here
- minimum sound length: this is the minimum length of a sound in events, the shorter ones must be discarded
- maximum sound length: this is the maximum length of a sound in events, the longer ones must be truncated
- number of velocity bins
- split by instruments: if this is True, extract a sound for each instrument; if False, extract only one sound

The extractor pipeline outputs a list of extracted sounds.

2.13 Encoder pipeline

The encoder pipeline converts a list of sounds into a specific encoding. Data for the neural network model is encoded using an one-hot encoding method, that is required because most of the times machine learning algorithms cannot work directly with categorical data, which is often not numerical. This type of encoding is done by adding binary features for each unique data entry received.

2.14 DAG Output

The DAG Output pipeline is the last one in the chain of pipelines and its purpose is to take the result from Encoder pipeline and dump it to a desired file on disk.

2.15 Used data

For the first dataset, the Note sequences TensorFlow record file, with the size of 140MB was passed through the constructed DAG pipeline with the result of a new TF record having a larger size of 12GB. The time required by this operation was twenty minutes.

The classic dataset, which is expected to have more sounds extracted, because of the higher duration of each MIDI, has been processed by the pipeline and the final result for the model is a TensorFlow record file with a total size of 128GB, with a computing time of forty minutes.

All of these operations were executed on a system with following configurations: *Intel Core i7-8700 CPU 3.20GHz, 16GB RAM, x64 architecture*.

Chapter 3

TensorFlow Graph

3.1 TensorFlow Graph

TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow session to run parts of the graph across a set of local and remote devices. [1]

A TensorFlow graph is compounded of:

- structure: displays how single components are linked
  - nodes
  - edges
- collections: a container for storing metadata for further uses in TensorFlow applications: learning, evaluation, also generation

In the following sections the process of building the graph is presented handling all the modes the application is run in(training, evaluation or generation), but before entering into graph details, it is required to define a tensor.

3.2 Tensor

A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes. [2]

[1]`https://www.tensorflow.org/guide/graphs`
[2]`https://www.tensorflow.org/guide/tensors`

3.3 The input

In the case of training and evaluation, the TF record file is read using

```
import tensorflow

tensorflow.TFRecordReader()
```

and then the sequence features for inputs and labels are extracted parsing the read file. Input tensors are created using the previous sequence features and the length for an entry in the inputs sequence features. If in training mode, a shuffling operation is performed on these tensors. Finally, the respective tensor is sent as a parameter to a tensorflow function that creates batches of tensors of the same form as the argument. The chosen batch size for the model is 64, but this value can be easily changed and sent to the graph builder as a parameter.

The batch size is a hyper parameter for a neural network. Instead of process an entire set of samples, the algorithm can split the set into smaller batches and adjustments will be executed with less memory and faster because the weights are updated for each batch according to error computed for that particular batch.

For the case of generation, a TensorFlow placeholder, which is a substitute for a tensor, is created using the same batch size as for training.

3.4 Overfitting and used solution

Before describing the dropout method and the use of it in the music model, a few words about overfitting are necessary.

3.4.1 What is overfitting

Overfitting is the event when a model is trained particularly to the input data and does not generalize. This means the model learned the features of the dataset and can easily manage it well, but does not act well on fresh data, it actually learned 'by heart' instead of 'understanding' the data and discovering patterns.

3.4.2 Dropout

Dropout is a technique for reducing overfitting and has as main idea the reduction of some weights from each batch in the process of training. This is acquired simply by choosing a dropout factor and selecting that percentage of nodes to be invisible for this iteration. This means the training process executes this step only on the remained neurons, the invisible ones are omitted.

The dropout factor used for generation is 0 and for training and evaluation is 0.1.

3.5 The output

The output(for all run modes) is constructed with the help of a TensorFlow dynamic recurrent neural network using the inputs and a given cell, which is a Multi RNN Cell, built with layer sizes: [512, 512, 512], the dropout specified above. The basic cell for the previous described one is a TensorFlow BasicRnnCell.

3.6 Collection metadata

3.6.1 Train

The following metrics data has been added to the graph for the training mode:

- the loss is computed using softmax cross entropy
- perplexity is the exponent function computed on loss
- optimizer: Adam optimizer with a learning rate equal to 0.001
- event accuracy computed for correct predictions for a number of events

3.6.2 Evaluation

The following metrics data has been added to the graph for the evaluation mode:

- the loss is computed using softmax cross entropy
- metrics accuracy: using TensorFlow metrics accuracy
- metrics event accuracy: using TensorFlow metrics recall
- perplexity: exponent function computed on loss

3.6.3 Generation

The following data has been added to the graph for the generation mode:

- temperature is a TensorFlow placeholder
- softmax used with TensorFlow softmax
- inputs: computed as above

Chapter 4

Training and evaluation

4.1 Training the neural network

4.1.1 Neuron

First of all, a better explanation of what a neural network is and how the process of training takes place will be presented. As started in the introduction, an artificial neural network is inspired by the human brain, containing a set of neurons that 'communicate' and sent information. Approaching the theoretical part, an individual neuron is represented by a function that takes parameters and computes a final value using them.

Supposing the set of parameters is denoted by: $p_1, p_2, ...p_n$, each one of them has assigned a value describing the 'importance' of that parameter to the network. These values can be denoted by $w_1, w_2, ...w_n$, being called **weights** and they are used to calibrate the value of a corresponding parameter.

It can be said that a neuron fires when it has enough information, so it is ready to send it further. A simple activation function can be obtained by multiplying each p_i with the corresponding weight w_i and sum the results together. This is called an weighted sum. Next, if this value exceeds an established threshold value, the neuron fires. A popular function of this kind is called Rectified Linear Unit(ReLU): $relu(weighted\_sum) = max(weighted\_sum, 0)$. for the linear neuron described above. This is called an activation function. A final function for a simple neuron with the specified activation function looks this way: $\sum_{i=1}^{n} p_i * w_i$.

4.1.2 Training and loss function

A neural network is compounded of many neurons linked together, with neurons that take as input the output of previous neurons. By seeing the way they are connected, layers of neurons can be defined, each layer sending the output to next layer as input. Obviously, the first layer, which is the input layer, does not receive output from other neurons and the final layer outputs the result of the network and does not send the value to another neuron.

In order to discover how the neural network behaves and if it has learnt what it had been desired, a loss function can be used, that computes how far the output of the network is from the expected one.

The process of training tries to find the accurate values for the weights, in order to obtain correct output for the input data or to be as close as possible to it. That means that the loss function should be lower, so the target of the training, besides adjusting the weights, is to reduce the loss function.

As a short conclusion, the process of training aims to reduce the loss function, to get close to the real result by modifying the weights values.

4.1.3 TensorFlow training

The training TensorFlow app starts by initializing TensorFlow flags, such as:

- run directory
- path to TFRecord file containing sequence example records after pipeline processing
- number of steps to train
- evaluation flag: specifying whether to run also an evaluation job
- hyper parameters

The second step for preparing the training is to build the TensorFlow graph in training mode and then the training job can be started, using TensorFlow train method and specifying several details stored in TensorFlow flags and graph.

```
import tensorflow

for _ in range(number_of_iterations):
    tensorflow.contrib.training.train(...)
```

The used function for the training is Cross entropy.

4.1.4 Used data

The jazz dataset, that is a final TF record with the size of 12GB has been trained for 34 hours, with a total number of steps equal to 6000, whereas the classic dataset with the size of 128GB has been trained for 57 hours, for a total number of steps equal to 9000.

4.2 Evaluation

4.2.1 Neural network evaluation

In the terms of evaluation, in neural network the most important term is accuracy. An evaluation job represents the process of passing input data to the model and providing the result. The output is compared to the expected target and the number of fair computations are counted and divided by the total number of input entries. This final number is called accuracy.

4.2.2 TensorFlow evaluation

The addition for evaluation part besides training is that an evaluation job is running. The same workflow is followed as in the case of training, only that this time, the model is evaluated.

```
import tensorflow

tensorflow.contrib.training.evaluate_repeatedly(...)
```

Chapter 5

Generation model

5.1 Hyper parameters

Hyper parameters are variables that are sent to the network, defining its structure and influence the training and the final results.

Hidden layers

Hidden layers are those ones between input and output layers and they can increase the accuracy. The following is an example of a network with 4 hidden layers.

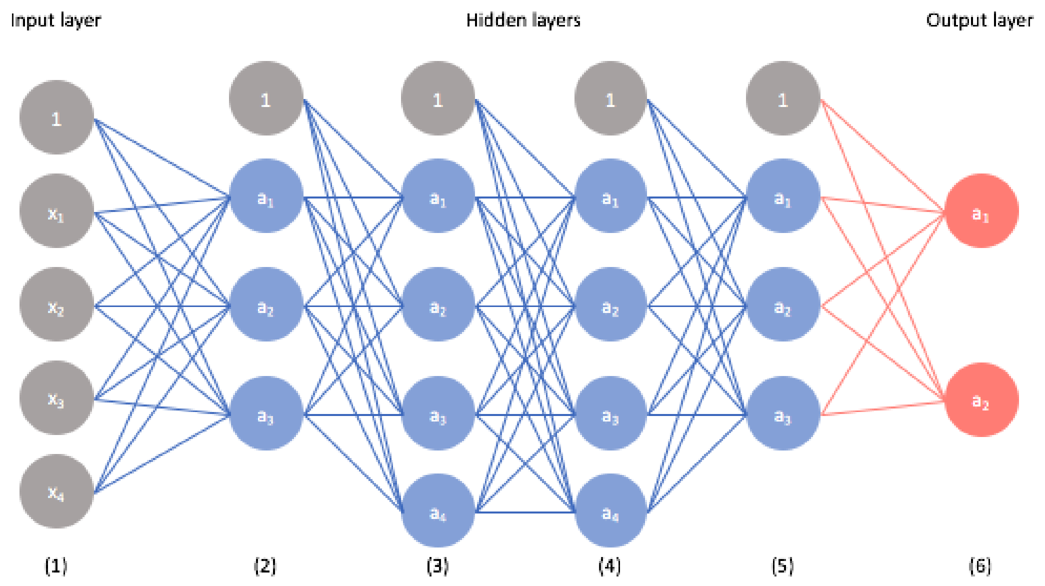

Figure 5.1: Hidden layers

Dropout

As described earlier in the thesis, dropout is a regularization technique that aims to reduce overfitting and to obtain better accuracy on a validation set, some input entries that have not been used for training, so are novel for the network.

Weights initialization

Before adjusting weights and find the accurate values in order to obtain the best output, they need to have some initial values. They can be initialized with a default value(as zero) or with randomly selected ones, from a normal distribution.

Learning rate

This value determines how fast a model learns, a low value slows the training and a higher one speeds the process. The difference is that learning slower determines a smooth way to converge, while a faster learning may lead to an early path to converge, but with the risk not leading to converge at all.

Epochs number

This hyper parameter it is actually the number of iterations for the training job. This is usually a large number, as used for the music generation model: five thousands, ten thousands.

Batch size

This is the size of a batch used for training. This number is usually small as chosen for the music generation model: 64.

5.2 Optimizers

Momentum

Momentum represents a variable learning rate and it can speed up the process of learning or also increase the accuracy. This technique determines the following direction of the gradient taking into account the previous decisions.

An optimization algorithm has the target to improve the results or in addition to speed up the process of a certain solution. For the case of neural networks, optimizers can increase the execution speed and improve accuracy.

Adagrad

Adaptive gradient has the purpose to adapt each learning rate to the size of the gradient. High gradients receive low learning rates, while lower gradients receive higher learning rates. This is done by dividing each gradient by a norm of past gradients. One advantage of Adagrad optimizer is that the features that appear rarely in the dataset are adjusted with a higher learning rate.

Rmsprop

Root mean square propagation is a optimizer built on Adagrad, but it does not use all the past gradients, only an exponential running average for them.

Adam

Adaptive moment estimation is built on Rmsprop and Momentum, only that it uses separate learning rates for each weight, divided by the norm of previous gradients and it uses a factor, similar to momentum, it builds on the previous gradients. This is the optimizer used for the music model.

5.3 Generation model

In order to create the generative recurrent neural network model, the TensorFlow graph can be used to define most of the specifications for the model, such as:

- layer sizes: [512, 512, 512]
- batch size: 64
- dropout factor: 0.1
- learning rate: 0.001

The next step is to collect all these details and hyper parameters and to define how the generation takes place. The main three methods that the generation model provides are described in the following sections, describing the generation case from big steps to smaller steps: generating events, generating steps, generating steps for batches.

5.4 Generating events

This is the first function the model calls, because it desires to generate an event sequence from a primer sequence received as input. It requires the length of the generated event sequence, that also counts the length of the primer one, also a temperature, that is used to divide the logits before computing the softmax function:

- temperature >1.0 increases randomness in generated events
- temperature <1.0 decreases randomness in generated events

The generative function needs a beam size, the number of steps per beam iteration and a beam factor used by beam search to generate event sequences.

The initial state and inputs of the model are retrieved from the TensorFlow graph, the existent input sequence is copied as the start of the final one and the new events are generated by a beam search that takes as arguments the above specified details and also a generative step function. The last argument is presented in the next chapter.

5.4.1 Beam search

In computer science, beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. [1]

For the music generative model, the beam search generates a new sequence and in the beginning the beam search algorithm contains a number of *beam size* instances of the primer sequence given as input.

Beam size, or beam width, is a parameter in the beam search algorithm which determines how many of the best partial solutions to evaluate. [2]

[1] https://hackernoon.com/beam-search-a-search-strategy-5d92fb7817f

[2] https://cs.stackexchange.com/questions/79309/beam-size-is-a-parameter-in-some-rnns-like-tensorflows-magenta-what-is-beam-si

In the generative recurrent neural network model, the beam search decides how many of the sequences should be passed to a decoder for each iteration of the algorithm, selecting the top *beam size* ones, using a score function.

The next step is to create a number(equal to the *branch factor* sent as parameter to beam search algorithm) of new event sequences for each existent sequence in the beam. Each new sequence corresponding to a starting sequence in the beam is extended with a number of steps equal to the corresponding parameter. A generated step for a new sequence can be retrieved from the next presented technique, generation for a single step.

The above described instructions, that are called **pruning** (selecting only a number of entries) and **branching**(generating a novel sequence for each selected one) are run by a specified number of iterations and finally, after the last step, the sequence with the best computed score is returned to the caller(the generating events method).

5.5 Generating steps

As described previously, in the process of beam search, for a new-build sequence, steps are generated and added to it one by one. This part of the generation takes as input the list of sequences and their present scores with the purpose of extending each sequence by one step computing a new score for every one. Because the model works on batches, the list is split by the *batch size* and the third generative method described in this chapter is used to modify each batch.

5.6 Generating steps for batch

This part of the generation extends with a single step a given batch of sequences. Using the encoder-decoder extension function that expands a sequence entry with a sample taken from the model(by basically making a random choice from the softmax probabilities), after computing the softmax function, the list of modified event sequences is returned.

Chapter 6

Generation

This chapter aims to present how the generation model is used by the final application and how a generator that works with all the sequences is created and used.

6.1 Initializing the music generation application

First of all, after the TensorFlow app is run, the TensorFlow flags are set, containing user specified data about:

- run directory
- input sequence: can be a midi, a list of pitches or a list of event values(no event, note off event or note on event)
- desired number of generated outputs
- output directory to store the generated sequences
- temperature: the randomness for new outputs
- beam size: used for beam search
- beam factor: used for beam search
- steps per iteration: used for beam search
- hyper parameters: such as learning rate, batch size, dropout factor

After the above input is configurated, a sequence generator is created. More details about how the generator works are described in the following section. All the arguments specified above are saved as run options for the generator.

The next step in generation app is to create the input sequence; depending on its type(midi file, pitches, event values), several transformations might be required because each of them is transformed differently into a note sequence. When the initial note sequence for training was created from a dataset of MIDI files, a conversion to protocol buffers was used. The same conversion can be applied in this case if the input is a MIDI file, but different transformations are used for pitches and event values.

The final step is to iterate a *number of generated outputs* and to call the method **generate** from the previously created generator. The generated sequences are transformed to MIDI files and saved in the requested location.

```
music_generator.add_options(details, hyper_parameters)
for _ in range(number_of_generated_outputs):
    generated_sequence = music_generator.generate(primer_sequence)
    MidiConvertor(generated_sequence).write_file(output_directory)
```

6.2 Sequence generator

The sequence generator is mainly used to apply a series of transformations on the primer sequence sent as parameter, same as some transformations have been utilized when creating the training dataset using a Direct Acyclic Graph. The sequence is trimmed, quantized and sounds are extracted from it before generated steps are added.

The last step of the generator is to call the model's generate method to create steps one by one until the desired number of steps is met.

Chapter 7

Application

7.1 Artificial neural network application architecture

In the following lines, a simple architecture of the application that trains and generates music is described.

TensorFlow Applications

- ApplicationDatasetCreator: has a DAG pipeline instance;
  run to create the dataset for training
- ApplicationTrainer: run to train a model
- ApplicationGenerator: run to generate music

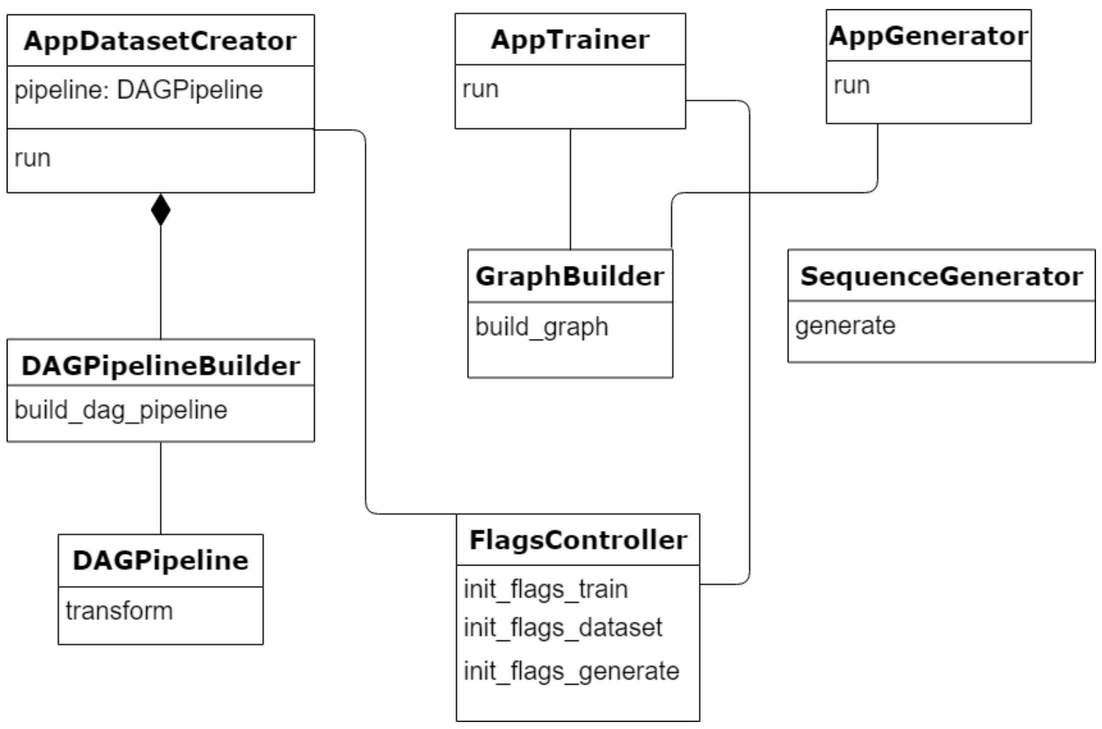

Figure 7.1: Applications diagram

Pipelines

- BasePipeline
- EncoderPipeline
- PartitionerPipeline
- ExtractorPipeline
- StretchPipeline
- SustainPipeline
- SplitterPipeline
- QuantizerPipeline
- TranspositionPipeline
- DAGPipelineBuilder

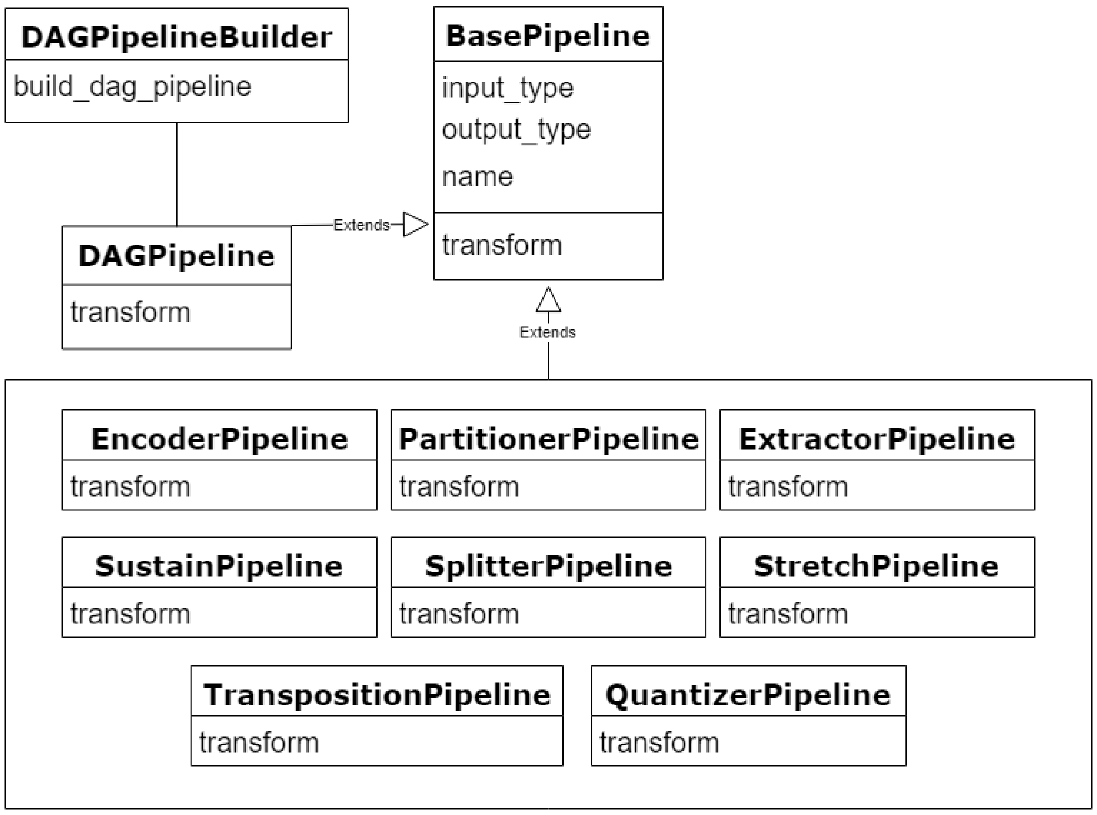

Figure 7.2: Pipelines diagram

Model

- AbstractModel: defines relations with TensorFlow session
- EventSequenceRnnModel: defines generative methods
- SounsRnnModel: defines a method for how the sound is generated

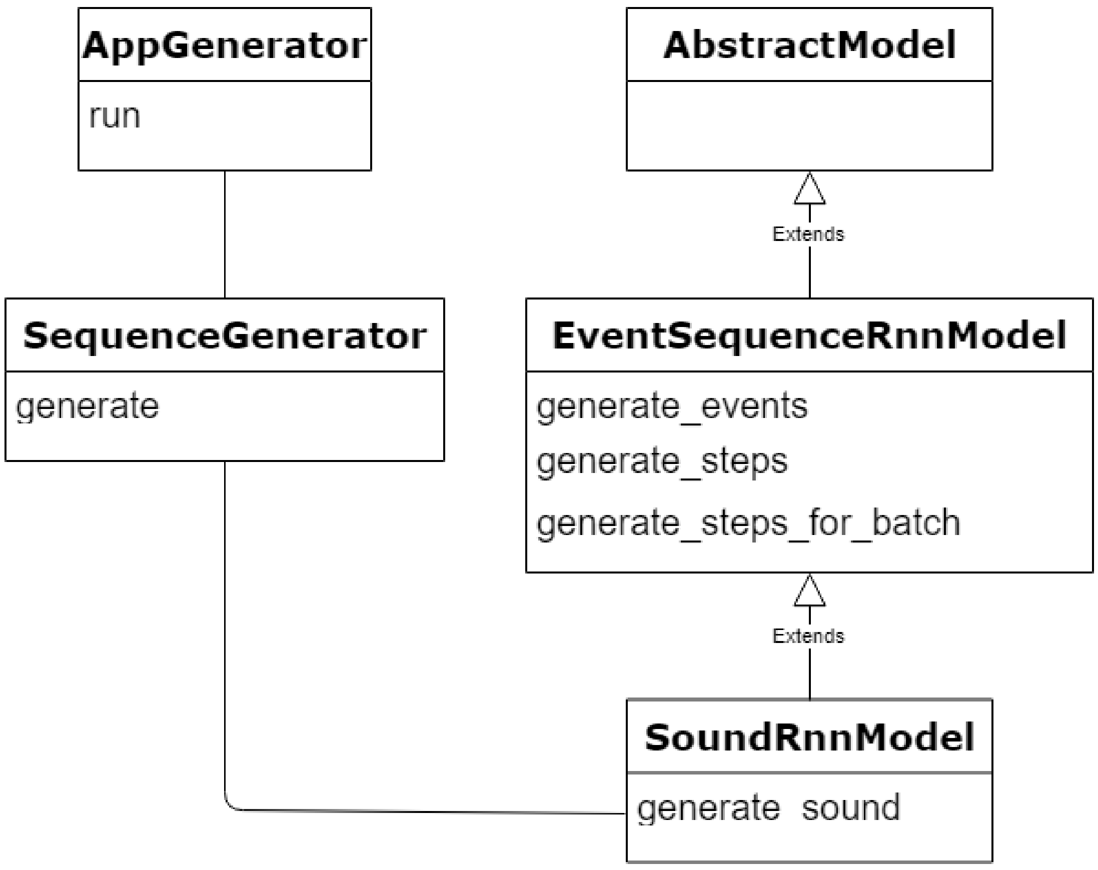

Figure 7.3: Model diagram

TensorFlow flags

- FlagsController: used to initialize TensorFlow flags

Generator

- SequenceGenerator: creates a music generator

TensorFlow graph

- GraphBuilder: build a TensorFlow graph for specified configuration

7.2 Web application

If an artist chooses to work with a music generation tool, the application needs to be easy to understand and to use, highlighting its features. A simple use case for the generative recurrent neural network is that an artist introduces a few musical notes as input for the tool and a continuation to those notes is fed back to him/her. The problem is this user does not intend to work with command lines, specify arguments such as paths, hyper parameters or so on and for sure, does not want to see code and understand it. The artist desires to work in his/her environment, using notes as **A2** or **F#4** and not the corresponding values for a MIDI format(45 respectively 66). A solution would be to offer the user a musical instrument and the possibility to play it. The scenario is simple: the artist plays an instrument, recording some musical notes and then the generated music for what it was played is sent back automatic.

The web application that highlights the use of the generative music network displays a piano, the user being able to interact with it and in the end, to press a **Generate** button.

7.2.1 Django

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source. [1]

7.2.2 Models

Django uses models to store information for an application. A model can be defined by a class, having the possibility to assign fields by specifying their type and a default value. Using django commands, these models structures are defined in a SQL database and corresponding instances can be defined by creating new objects of classes that represent models, objects that can be later inserted in the database.

The generator web application has 3 models. Their role is to create the interaction between the user and the instrument played, which is the piano. The models, defined as classes are the following:

- PianoKey: an abstract class for the following types; this defines each piano key on the instrument, with details about the key
- WhitePianoKey: a class that defines only white piano keys on the instrument; this class inherits the above one

[1] `https://www.djangoproject.com/`

- BlackPianoKey; a class that defines only black piano keys on the instrument; this class inherits the first one

On a regular piano and also the piano present on the web interface, there are 52 white piano keys, so 52 instances of the WhitePianoKey class, respectively 36 black piano keys with 36 corresponding objects instantiated from the class BlackPianoKey. Each key of the piano, which is actually an instance of PianoKey, is used to communicate with the front-end of the application, to send the details in the fields, to display the keys and also, to play the correct note corresponding to the pressed key.

7.2.3 Generation workflow

The workflow of generating music with the web application is quite simple. The artist has to sing his/her part using the interactive piano interface by pressing the keys, that are actually buttons and direct to a Django model. The sequence of notes introduced by the user is saved and updated each time he/she presses another piano key. This is stored as a list of values between 21 and 108, these representing MIDI numbers corresponding to musical notes. The mapping for a musical note is achieved in the back-end, where each model stores the note as **B4** and also the corresponding MIDI number: **71**. The pressed piano are stored directly as MIDI numbers, instead of musical notes, because the generative model accepts as a parameter a list of these type of values. The user starts the process when he/she presses the **Generate** button.

The next step is to call the generator, feed the model with the stored input and send back the output. The neural network creates as the final result a MIDI file, consisting in the input and the related generated sequence. The feedback for the web application after pressing the button is the novel sound that is played.

7.2.4 Controlling the output

As a first method to control the generated sequence, pressing the keys in a desired order, it is not the only possibility to modify the output of the model. The user has the possibility to select between genres of music, which actually means that trained models are switched. Also, the duration of the generated sequence can be specified.

Conclusions

Final results

The final application, that is integrated with a web application, offers an user the possibility to easily create music. The web application can be seen as an online instrument player, because the client can press the key and receive the correct note sound.

The project purpose, as stated in the beginning, is to create a way for humans, especially artists, to interact with an artificial system and to utilize its features. The system can be used by unprofessional individuals, by pressing notes in a random order, without the need of singing an actual song. The model succeeds to adapt to the input and generate a continuation for what has received because it had been trained on real data, songs created by professionals. Also, the application can interact with artists and real songs played by them at the virtual piano.

There are different trained models that are available to the end user, but not in their direct, physical form, instead the workaround is to allow him/her to select from a variety of genres. Two different runs of the generation, with the same input introduced to the piano, when distinct music genres are used, generate two outputs that stick to the genre, but also continue the sound from the initial notes.

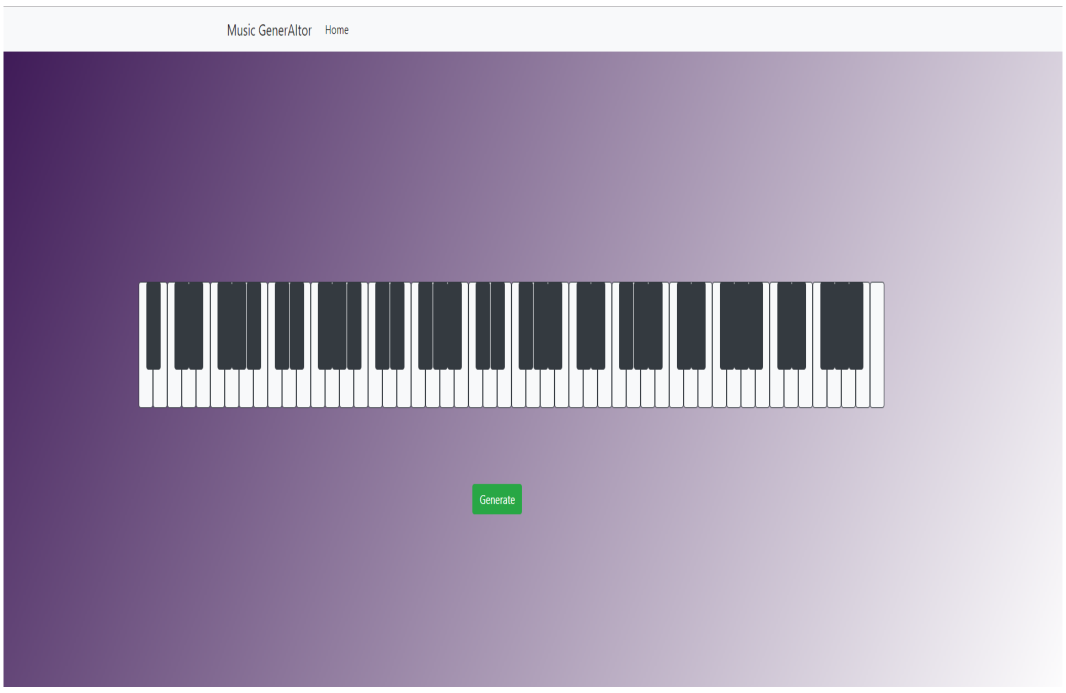

Figure 7.4: Web application

Further improvements

The model could be trained on more different and special datasets, with the target of allowing the user to play various instruments, the final result being sung by a chosen instrument. This feature would imply an additional selection applied to the dataset, that verifies for each MIDI file the instruments is sung by. The problem would be to find a huge amount of MIDI files played by a certain instrument, such as guitar.

Bibliography

- Swift Andrew (1997), *A brief Introduction to MIDI*
- Shuker Roy (1994), *Understanding popular music*
- Edwin M. Ripin (2001), *Sustaining pedal*
- Childs, G.W., IV (2019), *A Music Producer's Guide to Quantizing MIDI*
- Simon Price (2003), *Pro Tools: Using Beat Detective*
- LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey (2015), *Deep learning*
- Siegelmann, Hava T.; Sontag, Eduardo D. (1992), *On the Computational Power of Neural Nets.*
- Graves, A.; Liwicki, M.; Fernandez, S.; Bertolami, R.; Bunke, H.; Schmidhuber, J. (2009),
  "A Novel Connectionist System for Improved Unconstrained Handwriting Recognition"
- Reddy, D. Raj. (1977), *Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort. Department of Computer Science.*
- Norvig, Peter (1992), *Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP*
- Gers, Felix (2001), *Long Short-Term Memory in Recurrent Neural Networks*
- Judy A. Franklin (2008), *Jazz Melody Generation from Recurrent Network Learning of Several Human Melodies*
- Feynman Liang, Mark Gotham, Matthew Johnson, Jamie Shotton (2017), *AUTOMATIC STYLISTIC COMPOSITION OF BACH CHORALES WITH DEEP LSTM*
- `https://www.midi.org/`

- https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f
- https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/
- https://developers.google.com/protocol-buffers/
- https://www.tensorflow.org/
- https://www.psychologytoday.com/us/blog/hide-and-seek/201811/what-is-intelligence
- https://skymind.ai/wiki/neural-network
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://hackernoon.com/beam-search-a-search-strategy-5d92fb7817f
- (https://cs.stackexchange.com/questions/79309/beam-size-is-a-parameter-in-some-rnns-like-tensorflows-magenta-what-is-beam-si)
- (https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a)
- (https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f)

Annex