Composing like a human:
Adapting generative networks to few-shot learning in the musical domain

Tudor Paisa
Student Number: 2019551
Administration Number: 315146
t.paisa@tilburguniversity.edu

THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE IN DATA SCIENCE AND SOCIETY,
AT THE SCHOOL OF HUMANITIES AND DIGITAL SCIENCES
OF TILBURG UNIVERSITY

Thesis Committee:
Dr. Menno van Zaanen
Dr. Martin Atzmueller

Tilburg University
School of Humanities and Digital Sciences
Department of Cognitive Science & Artificial Intelligence
Tilburg, The Netherlands
May 7, 2019

Abstract

Lorem Ipsum

Contents

1 A hazy shade of winter

Over the past sixty years, Artificial Neural Networks (ANNs) have experienced several popularity cycles noted in the literature by extensive publicity of over-inflated expectations with regards to the promises of connectionism (the field of science that tries to explain mental phenomena with the help of ANNs) (Knight, 2016; Minsky, Papert, & Bottou, 1988; Nilsson, 2009). More specifically, it promised that a system could learn to do anything it was programmed to do (Minsky et al., 1988). However, each wave of praise was followed by a mass of public disappointment, criticism, and funding cuts (e.g., Lighthill, 1972). These upturns and downturns of the field of connectionism - or more generally A.I. - are now known in the literature as the A.I. winters (Nilsson, 2009).

The recent revival of ANNs under the brand of Deep Learning (DL) has brought new ways to tackle Machine Learning problems - most notably in the context of classification (but also regression, although less popular; see LeCun, Bengio, and Hinton, 2015 for several examples). In addition, with the advent of chatbots (Dale, 2016), voice-activated personal assistants (Xiong et al., 2018), and general-purpose A.I. systems (Vinyals et al., 2017), DL introduced new frontiers in artificial intelligence research.

Broadly speaking, DL as a field, promises to solve tasks which are easily (intuitively) performed by people, but hard to formalize (e.g., recognizing faces or spoken word; Goodfellow, Bengio, Courville, & Bach, 2016). The solution comprises of enabling computers to learn from experience and understand the world through the discovery of hierarchical relationships between concepts (LeCun et al., 2015). This way, there is no need for manual input on the sort of knowledge that the computer needs (Goodfellow et al., 2016). A more simplistic and concise interpretation of DL would be to see it as a statistical technique for identifying patterns in sample data using large (deep) ANNs (Marcus, 2018).

The reasons for presenting these facts will be clarified bit-by-bit throughout this chapter (Section 1.3 provides a concrete explanation) however, a terse argumentation is that progress towards general A.I. systems is marked by scant explorations of ANNs outside of the classification task, and even more so when trying to overcome some of their shortcomings, such as being highly reliant on massive amounts of data.

1.1 Strange Fruit

Typically, in a neural network, data samples go through a set of input units (that might represent pixels, word embeddings, etc.), then through multiple hidden layers, each with a given number of nodes, and reaching the output units where the answer is given (Marcus, 2018). An overwhelming majority of experiments with ANNs involve discriminative models (Goodfellow et al., 2014), where the goal is to assign high-dimensional inputs to a single class label (such as an animal or piece of furniture; Krizhevsky, Sutskever, & Hinton, 2012; X. Zhang, Zhao, & LeCun, 2015). However, other possible applications of DL models include (but are not limited to): regression (Qiu, Zhang, Ren, Suganthan, & Amaratunga, 2014), data compression (Cheng, Sun, Takeuchi, & Katto, 2018), and generating new data points (Graves, 2013).

The last point mentioned above is particularly interesting not because it has gained a lot of traction in recent years, but because of its potential to assist the scientific community as well as the general population. Borrowing the analogy from Goodfellow (2016), one might wonder what is the value of studying generative models. If, for instance, we are dealing with images, such a framework would merely generate more images for us (something which the Internet as no shortage of). This sort of worldview, however, would only limit these systems from their full capacity; aside from the obvious use case, generative networks could also be used for simulating possible futures - such as for scheduling and planning (Finn & Levine, 2016) -, for providing predictions to missing data, or for enabling work on multi-modal outputs, where a single input may have more than one correct answer (Goodfellow, 2016).

Regardless, successes in generative models have been either scarce or unattractive, with Fully Visible Belief Networks (Frey, Hinton, & Dayan, 1996) being the most suitable (and popular) method for the longest run. Given the age and - alleged - popularity, a valid question to ask is why things haven't moved that much since then. As it turn out, this method requires generating one sample at a time (ramping up the cost of the process to $O(n)$). Coupled with the fact that the computation is done through a neural network, creating n samples requires a substantial amount of time (Goodfellow, 2016). This discussion is not exhaustive; for a comprehensive list of various generative approaches and their downsides, refer to Goodfellow (2016).

The discovery of Generative Adversarial Networks (GANs; Goodfellow et al., 2014) has leapfrogged the state of developments in this field. Besides countering the drawbacks outlined above, they have been particularly successful at a variety of tasks such as image super-resolution (Ledig et al., 2016, see Figure 1.1), style-based image generation (Karras, Laine, & Aila, 2018), or image-to-image translation (such as converting a satellite image into a map, or a sketch into a photorealistic image; Isola, Zhu, Zhou, & Efros, 2016, see Figure 1.2).

Similarly, and although not as popular, Recurrent Neural Networks (RNNs) have been successful at various generative tasks (Jenal, Savinov, Sattler, & Chaurasia, 2019; Sutskever, Martens, & Hinton, 2011). The reason is because RNNs contain a high-dimensional hidden state which allows them to remember and predict from previous inputs (Sutskever et al., 2011). This made them great candidates for predicting sequences (Graves, 2013) of text or characters, where the input at t_0 might be influenced by prior input at t_{-1}, t_{-2}, etc. (Fan,

(a) original (b) bicubic (c) SRResNet (d) SRGAN

Figure 1.1: Example of single-image super-resolution results that highlight the advantages of a GAN. The original high-resolution image (1.1a) has been downsampled to make a low-resolution image. The results of bicubic interpolation (1.1b) can be seen in the second image, followed by the results of SRResNet (1.1c), a neural network trained on mean squared error. Lastly, the results of SRGAN can be seen in the final image (1.1d). Figure adapted from Ledig et al. (2016).

Qian, Xie, & Soong, 2014). That being said, Long Short-Term Memory networks (LSTM; Hochreiter & Schmidhuber, 1997) - a variant of RNNs - have been purpose-built with memory cells which permit better storing and accessing of information. This design decision variant of the recurrent network good candidates for generating sequences with long-term dependencies (Graves, 2013).

Having said all of these, the majority of developments with generative models lie in the field of Computer Vision. But that is not to say that we can generate only images. Generative models have been applied with varying degrees of success also in Natural Language Processing - where the scope was to generate high-quality text with sufficient diversity (e.g., L. Chen et al., 2018; Yu, Zhang, Wang, & Yu, 2016). Less explored, but still successful, have been the efforts in generating music (e.g., Dong, Hsiao, Yang, & Yang, 2017; Mogren, 2016). Herein lies the scope of this paper. Given such minimal explorations into generating music, we evaluated the samples generated by a GAN (Mogren, 2016) and a LSTM network (Oore, Simon, Dieleman, Eck, & Simonyan, 2018) trained under the constraints imposed by few-shot learning (Section 1.2) against a baseline model proposed by Larochelle, Finn, and Ravi (2017). To reiterate, a full explanation of the purpose of the paper is provided in Section 1.3.

1.2 I want it all

DL models usually rely on a procedure called stochastic gradient descent (SGD) which implies computing an output and an error for a given input vector, followed by calculating the input's average gradient and adjusting the network's parameters in a manner that minimizes the error (LeCun et al., 2015). Ravi and Larochelle (2016) argue that the iterative nature of gradient-optimization algorithms does not allow them to perform well under the constraint of a set

Figure 1.2: Isola, Zhu, Zhou, and Efros (2016) coined the term image-to-image translation where an input image is transformed into a revisualization of the original input. This figures illustrates how an outline of a purse (leftmost column) is transformed into a colored photorealistic version of it (rightmost column). In the middle we have the ground truth: the real image of the purse. Figure adapted from Isola, Zhu, Zhou, and Efros (2016).

number of updates over few examples. In particular, when faced with non-convex optimization problems, DL algorithms do not guarantee convergence within a reasonable amount of time. In addition, they also emphasize that for each new problem (e.g., new dataset), the model needs to reinitialize with a new set of random parameters. Ultimately, this hurts the network's ability to converge to a good solution when constrained by a limited number of updates (Ravi & Larochelle, 2016). Thus they lay out two clear drawbacks of DL models: "data-hunger", and inability to perform separate tasks.

A clear outcome from solving these issues would be that DL models are easier to train (less data) as well as multifunctional. Nonetheless, this list is not exhaustive. There is also fact that humans have the ability to generalize after one (or few) example(s) of a given object (W.-Y. Chen, Liu, Kira, Wang, & Huang, 2018; Ravi & Larochelle, 2016; Vinyals, Blundell, Lillicrap, Kavukcuoglu, & Wierstra, 2016) - something which DL models tend to lack. Moreover, there are many fields where the data exhibits a large number of classes, with few examples per class. Bridging the gap between human-type learning and current learning architectures would allow a system to do a proper job at capturing and generalizing information from sparse data (Larochelle et al., 2017; Ravi & Larochelle, 2016). This has motivated a series of explorations in the direction of "few-shot" learning (i.e., learning a class from a few examples) or more broadly, into the field of "meta-learning" (learning to learn in hopes of generalizing to new tasks; W.-Y. Chen et al., 2018; Vinyals et al., 2016; R. Zhang, Che, Ghahramani, Bengio, & Song, 2018).

In spite of its proposed benefits, developments and experiments with few-shot learning are still scarce (Larochelle et al., 2017). Moreover, evaluations are largely concentrated on image data (see W.-Y. Chen et al., 2018; Clouâtre & De-

mers, 2019; Lake, Salakhutdinov, & Tenenbaum, 2019; Ravi & Larochelle, 2016; Vinyals et al., 2016). Having said that, few-shot experiments with generative networks are even less frequent; Clouâtre and Demers (2019), Dong et al. (2017) and R. Zhang et al. (2018) being some of the early (and few) explorers in this direction.

1.3 Composing like a human

At the time of writing, generative models and meta-learning are two exciting areas of research the intersection of which could be quite promising. However, to the knowledge of the researcher, no experiments which combine the both of them have been conducted in the musical domain. Thus, this paper examined the extent to which generative models can create novel and qualitative samples of music under the constraints posed by few-shot learning. In other words, it sought to evaluate the degree to which generative models can create novel and high-fidelity samples when training data is scarce. To avoid any ambiguity, novel is used here as new (i.e., not in the training set), and qualitative as consistent with common scales and structurally diverse (combines chord structures with single-note sequences and is not restricted to only a few notes).

This research will evaluate the generated samples of two state-of-the-art generative models (C-RNN-GAN of Mogren, 2016 and Performance-RNN of Oore et al., 2018) that have been trained under a meta-learning framework, Reptile (Nichol, Achiam, & Schulman, 2018), with a limited number of songs. The performances of these two models were compared to the baseline proposed by Larochelle et al. (2017) and to recorded performances played by expert piano players at the International e-Piano Competition (University of Minnesota, 2019).

In other words, this study sought to answer the following research questions:

- To what extent is the music created by a few-shot generative model comparable to the music of a generative model that is trained on the entire dataset?
- To what extent is the music created by a few-shot generative model comparable to real music?

The following chapters will discuss more about generative models, meta-learning, and some of the models and algorithms involved in this paper (Chapter 2), followed by a detailed description of the setup of the experiments (Chapter 3), and the results obtained from them (Chapter 4). Finally, Chapter 5 closes with a discussion of the results and limitations of the experimental setup, whereas Chapter 6 concludes the findings of this paper.

2 Related Work

This chapter takes a deep-dive into the fields of meta-learning and generative networks in order to highlight their current state of affairs. This will lay the groundwork for the experiments detailed in Chapter 3. Section 2.1 will present the principal network architecture of this paper, the recurrent neural network, and argument for its use as the foundation of a generative network. Section 2.2 will introduce Generative Adversarial Networks, a training method build specifically for the task, whereas Section 2.3 will formally introduce meta-learning and some state-of-the-art approaches.

2.1 Recurrent Neural Networks

In a standard ANN, the data traverses the network through the input layer, to the hidden layers, and finally the output layer; this setup is generally known as a feedforward neural network (Figure 2.1; Graves, 2012). Broadly speaking, this type of network is defined by the fact that its connections do not form cycles. If however we relax this condition, we arrive at a recurrent neural network (Figure 2.2a; Rumelhart, Hinton, & Williams, 1986): a family of neural networks specifically designed for processing sequential data (Goodfellow et al., 2016).

For transparency, it should be noted that it is possible to use a feedforward architecture, specifically a one-dimensional Convolutional Neural Network, with sequential data, where the convolution forms the basis of a time-delay neural network. Unfortunately, the output of such a system would be a function of a small number of neighbouring data points (such as the last and the next three notes in a song; Goodfellow et al., 2016). Despite that, a system that uses convolution would quickly reach its limitation in the musical domain. Here, knowledge of the notes played from the beginning of the song is needed, meaning that we need an architecture that is able to account for long-term dependencies - which brings us back to the RNN.

The information passes through the RNN in a similar manner to the feedforward network except that the activations which arrive at the hidden layer are from the current external input and hidden activations from the previous timestep. It should be noted that the parameters of a recurrent network (weights) are shared across different timesteps (Graves, 2012). This allows extending (and applying) the model to data points of different lengths whilst generalizing across all of them (Goodfellow et al., 2016). To place this into perspective, a feedforward network would require separate parameters for each

input feature, at each position.

Thus, a formal definition of the network's forward pass would be

$$h^t = f(h^{t-1}; x^t; \theta) \tag{2.1}$$

where h^t is the state of the hidden layer at time t, x^t is the input at time t, and θ the model parameters.

The backward pass in a RNN is similar to that of a feedforward network: given the partial derivatives of a differentiable loss function $\mathcal{L}$ we calculate the derivatives with respect to the network's weights (Graves, 2012). In this setting, a popular algorithm is called backpropagation through time (BPTT; Werbos, 1990) which consists of repeated applications of the chain rule. The difference between standard backpropagation and BPTT is that the loss function depends on the activation of the hidden layer through its influence on the output layer as well as its influence on the hidden layer at the next timestep (Graves, 2012).

Equation 2.1 can be drawn as in Figure 2.2a: input sequences x are mapped to a hidden layer h and an output y, with the state of the hidden layer at time t feeding back into itself with a time delay of 1. In other words, the state of h_t will influence the decision of the recurrent layer at time $t + 1$. Another way to illustrate this is to unfold the computational graph such that each component of the network is represented by different variables per step, as in Figure 2.2b.

The reason why these notions have been introduced is to be able to bring up the following point. As good as standard RNNs are - in theory - at accessing contextual information when mapping input sequences to output sequences, the range of contexts that can be accessed in a standard RNN architecture is in practice quite limited (Graves, 2012). Hochreiter and Schmidhuber (1997) outlines two problems caused by BPTT: the error signals flowing backwards in time tend to either (1) blow up or (2) vanish. As such, the Long Short-Term Memory network has been proposed, a recurrent architecture designed to overcome these issues (Hochreiter & Schmidhuber, 1997).

The LSTM is made of a network of recurrently connected subnetworks known as memory blocks, each with one or more self-connected memory cells and three multiplicative units: the input gate, the output gate, and the forget gate (Hochreiter & Schmidhuber, 1997). The three gates act as analogies for the read, write, reset computer operations and allow the memory cells to store and access information over long periods of time (Graves, 2012). See Figure 2.3 for an illustration of a LSTM memory block with a single cell.

Aside from being successful at a variety of classification tasks (e.g., Graves & Schmidhuber, 2005), this architecture has been successfully applied to generating new data points (such as creating sequences of text; Graves, 2013). In the context of generating musical performances, experiments with this neural architecture go as far as Eck and Schmidhuber (2002) however, a more noteworthy implementation is Performance-RNN (Oore et al., 2018); a three layer LSTM network with 512 cells each designed to generate classical compositions whilst maintaining the playing style of expert pianists. Considering its success, this model was re-implemented for the purposes of this paper's experiments (see Chapter 3).

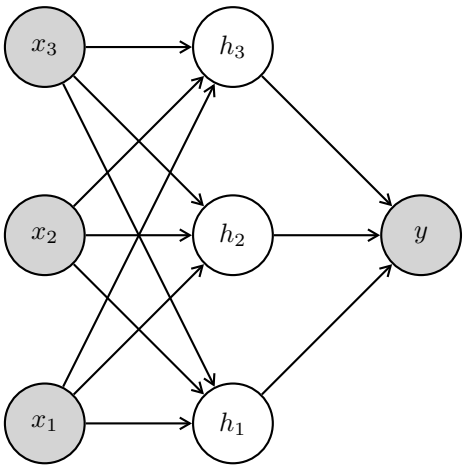


Figure 2.1: Example of a feedforward neural network with three nodes in the input layer (x_1, x_2, x_3), three nodes in the hidden layer (h_1, h_2, h_3), and an output layer a single node (y).

2.2 Generative Adversarial Networks

Nowadays, the most popular form of generating data is through the use of a Generative Adversarial Network (GAN; Goodfellow et al., 2014). The training procedure is set up as a game between two networks: a generator and a discriminator. The purpose of the former is to create samples that come from the same distribution as the training data, whereas the latter aims to determine which samples are real or fake (Goodfellow, 2016).

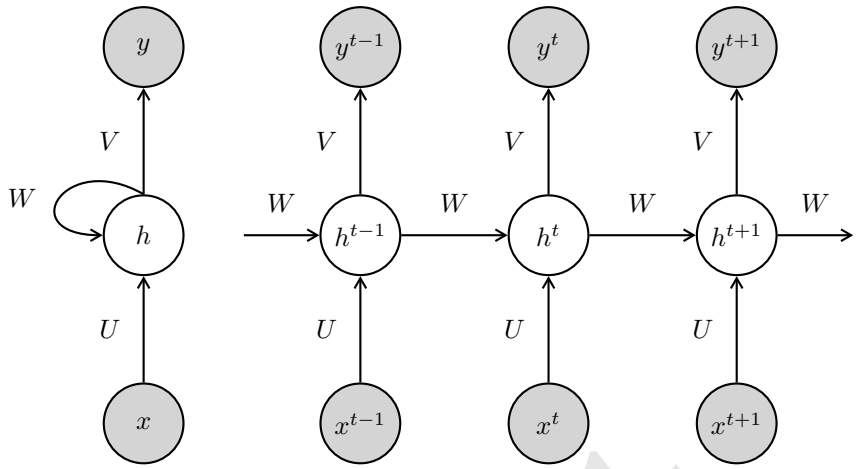
To make the generator's data distribution p_g imitate that of the real data p_{data}, we need to define a prior on input noise variables $p_z(z)$ and then represent a mapping to the data space $G(z;\theta_g)$ where G is a differentiable function represented by a network with parameters θ_g. The discriminator is another network with a mapping function $D(x;\theta_d)$ that outputs a single scalar. More concretely, $D(x)$ is the probability that x came from the real data distribution rather than that of the generator (Goodfellow, 2016). We train D to maximize the probability of correctly classifying the training data and samples from G (i.e., as "real" and "fake"), while simultaneously training G to minimize the dissimilarity between the two data distributions, *without actually looking at* p_{data}. In other words, G must minimize $\log(1 - D(G(z)))$ (Goodfellow et al., 2014).

To formalize, both models play a minimax game with a value function $V(G, D)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\ p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\ p_z(z)}[\log(1 - D(G(z)))] \tag{2.2}$$

where the solution is for G to recover the training data distribution and D to have a classification accuracy of 50% (Goodfellow et al., 2014).

Although new, GANs are a significant conceptual and technical innovation (Briot, Hadjeres, & Pachet, 2017) which can be seen in them revitalizing interest in generative models. As mentioned before, they have been proven successful at image super-resolution (Ledig et al., 2016), image-to-image trans-



(a) Folded (b) Unfolded

Figure 2.2: A typical view of an RNN (left) and an unfolded view of the same network (right). In both sub-figures each node is a layer of network units however, in the second representation these are shown for each timestep. The weighted connections from the input layer to the hidden one is labelled U, those from the hidden layer feeding back to itself are W, and those from the hidden layer to the output layer are labelled as V. Note that these weights are reused at each timestep. Bias has been omitted in this representation. Figure adapted from Graves (2012)

lation (Isola et al., 2016), or style-based image generation (Karras et al., 2018). Having said these, Mogren (2016) has shown that GANs do not belong only to the field of Computer Vision by successfully re-purposing this architecture to generate music. The model in question, C-RNN-GAN, features a completely unidirectional LSTM network with two layers, 350 units each as generator and a unidirectional LSTM network with a similar structure as discriminator.

2.3 Learning to learn

The rationale for investing time and energy into meta-learning is that we cannot capture the essence of learning simply by relying on a small number of algorithms, but rather a bunch of context-dependent learning strategies that would enable us to properly capture domain specific information (Schmidhuber, 1987). Given the complexities of each learning strategy, Schmidhuber (1987) argues that an obvious way to tackle this initiative is to give a system the ability to learn how to learn. A simplistic way to do this would be to train a system on different tasks (e.g, image classification, text classification, etc.). However, this approach entails training on a dataset that is much larger that what current hardware can withstand. Furthermore, this approach would stray away from the idea of building general-purpose A.I. architectures as generalizing over a task would require more data points that what a human would need. To be more concrete, a human is able to generalize a task/object after encountering one (or few) example(s) (W.-Y. Chen et al., 2018; Ravi & Larochelle,

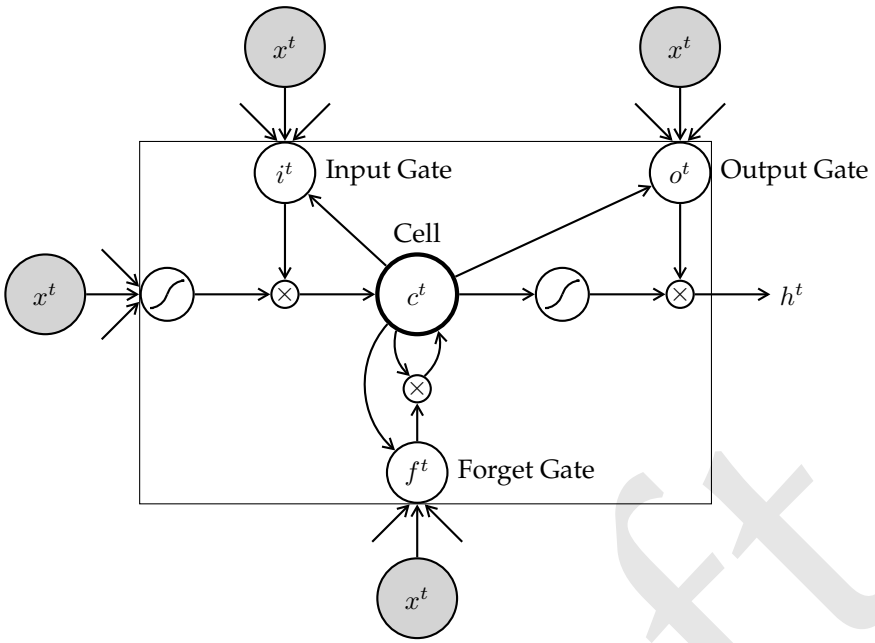


Figure 2.3: LSTM memory block with one cell. Here the gates are nonlinear summation units that collect activations from the inside and outside of the memory block and control the activation of the cell c^t via multiplications. The input and output gates (i^t, o^t) multiply the block's input and output activations respectively, while the forget gate (f^t) multiplies the previous hidden state. The input and output activation are typically tanh or sigmoid. The only outputs (h^t) from the block come from the output gate multiplication. Figure adapted from Graves (2012)

2016). Therefore, besides having to perform on a variety of tasks, a system would also need to excel at that using sparse amounts of data.

Although ideas of meta-learning can go as far as Schmidhuber's thesis in 1987, it seems that training a model using a handful of examples did not get much traction until Lake, Salakhutdinov, and Tenenbaum (2015) successfully designed a probabilistic learning framework capable of generalizing and learning a large number of concepts from a single class example (i.e., one-shot learning). For clarity, such a problem is typically defined as a n-shot k-way problem, where n stands for the number of examples per class, and k is the number of classes to be used in the training process.

Initially, it was thought that ANNs are incompatible with few-shot learning since the iterative nature of gradient-based optimization algorithms do not work well in the context of a set number of updates (Ravi & Larochelle, 2016) - and even more so when there are only a few examples to work with. Having said that, Rezende, Mohamed, Danihelka, Gregor, and Wierstra (2016) were among the first to provide a solution to the one-shot problem in the context of a neural network by implementing Bayesian reasoning into a deep network embedded within hierarchical latent variable models. Since that, numerous efforts were made to develop an algorithm that enables an ANN to

perform with strictly limited datasets. Probably the most popular algorithm at the time of writing is the Model-Agnostic Meta-Learning (MAML) algorithm (Finn, Abbeel, & Levine, 2017). One of MAML's core mechanics is that the parameters of the model are trained with a small number of gradient steps on with little training data to arrive at a good generalization on that task. However, this procedure requires calculating second-order derivatives which implies a substantial computational penalty - something which the authors did note. Therefore, they developed a first-order variant of MAML (FOMAML) which is able to perform similarly, if only with a small performance hit (Finn et al., 2017).

This seemed to be the case until Nichol et al. (2018) showed that FOMAML tends to be somewhat unstable. In response to that, they developed Reptile, a first-order meta-learning algorithm that rivals the performance of both MAML and FOMAML (Nichol et al., 2018). Just like MAML, it counteracts the shortcomings of gradient-optimization algorithms by learning the model's initial parameters in order to maximize performance on novel tasks, it allows for any type of network to be used, and does not place any restrictions on the type of loss function that can be used (Nichol et al., 2018). The difference between the two, however, lies in the way the parameters are updated.

Algorithm 1: Reptile (serial version)

Initialize θ, the vector of initial parameters
for *iteration* $= 1, 2, \ldots$ **do**
  Sample task τ, corresponding to loss L_τ on weight vectors $\tilde{\theta}$
  Compute $\tilde{\theta} = U_\tau^k(\theta)$, denoting k steps of SGD or Adam
  Update $\theta \leftarrow \theta + \epsilon(\tilde{\theta} - \theta)$
end

In essence, Reptile works (see Algorithm 1) as follows. A model with parameters θ is initialized, and a meta-loop (or meta-epochs) of i iterations starts. Within it, a task τ (e.g, image classification, text classification, etc.) is sampled, along with its corresponding loss function L_τ (for clarity, we assume that each type of task uses a different loss function). Then we compute the updated parameters $\tilde{\theta}$ in an inner-loop with k iterations on task τ (or in other words, once we sampled our task τ, we train our model k times on it using the θ as parameters so we can get $\tilde{\theta}$). Finally, we update our initial parameters θ using this formula: $\theta \leftarrow \theta + \epsilon(\tilde{\theta} - \theta)$. Then the algorithm continues to the next iteration of the meta-loop carrying over our updated parameters θ.

The explanation above is fairly verbose however, the algorithm itself is fairly simplistic and achieves similar performances to FOMAML (Nichol et al., 2018). This forms a compelling reason to use Reptile as the few-shot learning method of choice for this paper.

3 Methods

To reiterate from Section 1.3, the purpose of this paper is to establish (1) the extent to which the music generated under a few-shot learning algorithm is comparable to that of a model trained on the entire dataset, and (2) the extent to which the music from such a model is comparable to that of real musical performances. For these purposes, Performance-RNN and C-RNN-GAN were adapted to the Reptile training procedure (Algorithm 1) and evaluated against the (1) songs generated by the baseline proposed in Larochelle et al. (2017) - a single layer LSTM with 256 units - and (2) actual musical performances from the dataset (Section 3.1). For an in-depth explanation on the steps required to develop a model that is able to generate new data points refer to Section 3.3 and to Section 3.4 for details on the training procedure. The evaluation of the models' performances will be based on the number of statistically different bins (NDB; Richardson & Weiss, 2018) and several domain specific measurements adapted from Mogren (2016): polyphony, scale consistency, repetitions, tone span. Refer to Section 3.5 for more details on these metrics.

3.1 Dataset

The dataset for this research will be similar to that of (Oore et al., 2018) namely, the MAESTRO Dataset (Hawthorne et al., 2018). It consists of recorded MIDI data collected from each installment of the International Piano-e-Competition. All performances were done by piano experts on a Yamaha Disklavier, instrument which integrates a highly precise MIDI capture and playback system. The claim is that the recorded MIDI events are of sufficient fidelity to allow judges to remotely listen to the contestant's performance (also on a Disklavier).

The dataset contains MIDI recordings from nine years of the International Piano-e-Competition. This amounts to 1,184 piano performances, approximately 430 compositions, 6.18 million notes played, and approximately 172 hours of playback. There is also a recommended train/validation/test split (954, 105, 125 songs respectively) created on the following criteria (Hawthorne et al., 2018):

- No composition should appear in more than one split
- Training set is approx 80% of the dataset (in time), and the remaining is split equally on between the validation and test sets. Where possible, these proportions are true also within each composer.
- Popular compositions are in the training set

```
<note_on note=69 velocity=73 time=33>
<note_on note=71 velocity=78 time=15>
<control_change control=64 value=83 time=27>
<note_off note=69 velocity=0 time=6>
<note_off note=71 velocity=0 time=3>
```

Figure 3.1: Example of MIDI messages in the dataset, as printed by the Python library Mido (Bjørndalen, 2018). At 33 ticks since the last event, the player pressed an A4 (note 69), followed by a B4 (note 71), 15 ticks later. Next, the pedal (Control Change message with a value of 64) is pressed 27 ticks later, and the keys are released 6 and 3 tick later.

- The validation and test splits should maintain a variety of compositions

Moreover, each performance comes with additional metadata: the name of the composer, the title of the performance, the suggested train, validation, test splits, year of the performance, name of the file, and duration in seconds of the performance. Having said that, the dataset lacks clearly defined classes. The canonical composer could be used as a substitute however, the author strongly believes that there is a better way to categorize music: by genre.

Broadly speaking, Western classical music can be segregated into a number of genres based on the year of the composition and style of the piece. For this we can use the Concise Oxford Dictionary of Music (Kennedy & Bourne, 2007) which indicates to the following classical genres: baroque (1600 - 1750), classical (1750 - 1820), romanticism (1780 - 1910), modernism (1890 - 1930), and impressionism (circa 1890 - 1925). Granted, there are far more genres, but these are the ones that are covered by the dataset. As such, given the definitions of Kennedy and Bourne (2007) on said genres and artists, each individual piece from our dataset was manually assigned a genre. Concretely, the style of music was firstly determined by the period in which the composer has lived (in baroque, classical, etc.) or - if the time period of the genres would overlap (the case of modernism and impressionism) - the composer's primary style of music would be chosen according to the definition given by Kennedy and Bourne (2007). Said labelling can be found in Appendix A.

3.2 Data Processing

The dataset consists of MIDI files which would require some processing in order to be ready for a LSTM network. Here, the approach of Oore et al. (2018) was adopted where MIDI midi messages are transformed into a sequence of one-hot encoded events. More specifically, MIDI excerpts are represented as a sequence of events from a vocabulary of 416 different events - similar to Oore et al. (2018), but with one extra events in the vocabulary:

- 128 Note-On events: starts a new note (one for each of the 128 MIDI pitches)
- 128 Note-Off events: releases a note (one for each of the 128 MIDI pitches)

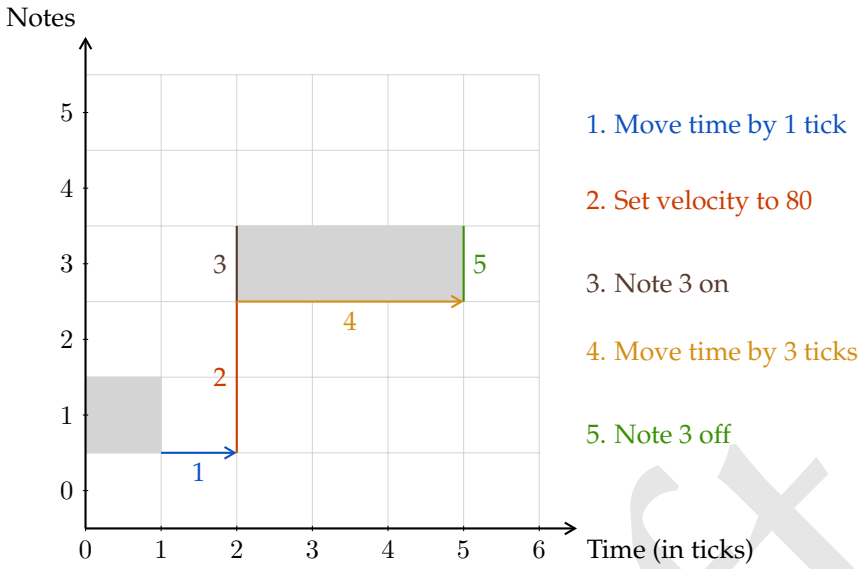


Figure 3.2: Example of how the data is represented. The progression illustrates how a piano roll of MIDI data is converted into a sequence of commands (right hand side) that belong to our events vocabulary. Figure adapted from Oore, Simon, Dieleman, Eck, and Simonyan (2018).

- 126 Time-Shift events: moves the time step forward somewhere between 0 ms and 1 second, in increments of 8 ms (Oore et al. (2018) ignored the 0ms forward step)
- 32 Velocity events: changes the velocity for each subsequent note, until the next velocity event
- 2 Pedal events: triggers or releases the pedal

A glance over the MIDI documentation (MIDI Association, 2019) shows how complex and granular the standard is. Luckily however, we do not need to implement all of the messages into our model in order to make it work. We only need three: `note_on`, `note_off`, and `pedal_on/off`. Figure 3.1 show an example of how the midi messages look like. It should be noted that in this figure, the time is represented in ticks, MIDI's smallest unit of time, who's value is usually a third of a beat (MIDI Association, 2019).

As it can be seen in Figure 3.1, each message contains multiple pieces of information such as the message type, the note/value of the message, the delta time (time elapsed since last message and the current message, measured in ticks), and velocity (if applicable). To one-hot encode this stream of messages, it is necessary to separate it so it contains only one type of information per input. Thus, the sequence can be further broken down such that the first message that the system encounters is the delta time, followed by the velocity, and the note/pedal event (and their respective values). An example of how the MIDI sequences are broken down into singular pieces of information can be found in Figure 3.2.

Having said these, it should be mentioned that the one-hot representation of MIDI messages has undergone some fine quantization, again, in a similar fashion to Oore et al. (2018). The delta times have been modified to correspond to a change of multiples of 8 ms (i.e., 0 ms, 8 ms, 16 ms, etc.) and the velocities have been partitioned into 32 steps as opposed to MIDI's 128. With regards to the time movements, Friberg, Bresin, and Sundberg (2006) note that a noticeable difference in temporally displacing a single tone in a sequence takes place from 10 ms and above. Thus, since our movements are quantized in steps of 8 ms, there should be no discernible difference for the human hearing. When it comes to the note velocities, it should be pointed to the reader that classical music features eight levels of common dynamic marking (from *ppp* to *fff*) which is why Oore et al. (2018) note that 32 steps are more than enough.

3.3 How to make music

Knowing what we know about RNNs and LSTMs from Section 2.1, one can devise a method where the network can generate a sequence of events. All that is needed is to present the system with an event to kickstart the generative process. Namely, an input x^t is fed to the network in order to output the next likely event x^{t+1}, which in turn is fed back to generate x^{t+2}. This process is repeated until we have a sequence of events of a desired length.

An alternative to this would be to define a noise vector z^t of an arbitrary size (e.g., 100), and a random input x^t selected from our events vocabulary. The noise vector is passed through a linear layer that output a tensor that would serve as the network's hidden state h^t at time t. It is then that x^t and h^t are fed into the LSTM layer to generate x^{t+1} and h^{t+1}. The reason for pre-defining the hidden state for the first input is to increase the number of possible outcomes coming from the network.

To be more precise, if we were to present the network with only x^t then the network would mark the input sequence as the beginning of the song. With that information, it is possible that the network would affix itself to following a specific pattern. For instance, if the first input would be a C#, the network could learn to always output E as the next note. Using a random hidden state is a measure that was placed to ensure some diversity in the process of generating samples (granted, it should be admitted the possibility of this scenario being unlikely, and that the regular method described at the beginning of this section would still work just as good).

Having said this, diversity is not the only reason for including a noise vector z into the training process. The reader might recall that this is part of the generation process in the framework of a GAN respectively, how the generator G would create samples. As such, all three models implemented in this paper feature the architecture of a GAN generator.

Moreover, in terms of generating the next event, Oore et al. (2018) have defined a parameter λ called temperature (Google Magenta, 2017). It is a value which uses the model's predicted events distribution to control the randomness of the output samples. In other words, let ϕ be the output of a LSTM network (for clarity, we ignore the hidden state h). Then, the next sequence x^{t+1} is defined as

$$x^{t+1} = X(\omega) \tag{3.1}$$

where

$$X = \text{softmax}(\phi/\lambda). \tag{3.2}$$

X is the predicted event distribution of the network's output divided by our temperature λ. Thus, x^{t+1} is actually a random sample from our defined categorical distribution X. In this paper we train our models with a temperature of 1.0, and generate samples with temperatures ranging from 0.8 to 1.2, with an increment of 0.1.

3.4 Training procedure

With the generative process outlined in the previous section, training the generative networks presented in this papers is somewhat straightforward. A batch of songs is sampled from the training set, the model generates as many songs as there are in the batch, and then compute the loss between the network's output and the real samples. This statement is essentially true for the baseline and Performance-RNN (in spite of the Reptile training procedure, inner-loop training is similar to the standard procedure). In contrast, the GAN model requires training two models simultaneously, where the generator's loss function is based on whether the discriminator is fooled by the fake samples (i.e., classifies them as real), and the discriminator's loss is based on whether it can distinguish between real and fake samples (it should be mentioned that both components of the GAN would use binary cross-entropy as their loss function; in contrast the baseline and Performance-RNN are based on cross entropy). However, there are still several items that we haven't touched upon.

Firstly, all three networks have been trained with teacher forcing, a procedure in which during training the model receives the ground truth y^t as input at time $t + 1$. Adopting this measure allows the model to create hidden-to-hidden connections and capture the necessary information from the history of input sequences (Goodfellow et al., 2016).

Secondly, Goodfellow et al. (2014) mention a couple issues that can arise when training a GAN. One is synchronizing the generator with the discriminator (it's possible that one of the models gets too strong, leaving the other unable to catch up), and the generator collapsing too many values of z to the same value of x (which would ultimately hurt the generator's diversity; this issue is known in the literature as mode collapse).

A solution to the issue of synchronicity would be to freeze the training of one of the models when the loss is below a predefined threshold (Mogren, 2016). In this paper, we stop training one of the models if its training loss is less than half the loss of the other's.

To combat mode collapse, Salimans et al. (2016) propose a technique called feature matching, where we change the objective of the generator to generate real data that matches the statistics of the real data. Since in the setting of a GAN showing the real data would be considered "cheating", we use the features of an intermediate layer of the discriminator. Specifically, we train the generator to match the expected value of those features (Salimans et al., 2016). Denoting $f(x)$ as the activations on an intermediate layer in the discriminator, the new objective for the generator becomes

$$\|\mathbb{E}_{x\ p_{data}} f(x) - \mathbb{E}_{z\ p_z(z)} f(G(z))\|_2^2 \tag{3.3}$$

which is commonly known as the squared error. However, to make the re-implementation of C-RNN-GAN true to the original, Mean Square Error (MSE) was used as the loss function when using feature matching.

It should be noted that now, the loss functions of both the generator and discriminator differ in scale, meaning that we can no longer compare them to freeze training of either models: the generator uses MSE, whereas the discriminator use binary cross-entropy. A workaround to this is to calculate the both losses, but use MSE for backpropagation and optimization, whilst binary cross-entropy is used to compare our models.

Moving forward, there is an aspect pertaining to the Reptile training framework which needs to be addressed: the training data. Normally, Reptile assumes that the training takes place in the n-shot k-way setting, meaning that our model is trained with k classes and n samples per class. Given that we have 954 samples in the training set, we adopted the same training procedure as Nichol et al. (2018) and Clouâtre and Demers (2019). Respectively, at each meta-loop we sample from our split n examples for each k class. For this paper, our $n = 1$ and $k = 5$ corresponding to a 5-way 1-shot experiment.

Another aspect that needs addressing is the number of epochs. All models were trained for 250 epochs each however, when looking at the way training is handled in Reptile (see Algorithm 1), this statement is inconclusive. Because the algorithm presents us with a meta-loop and an inner-loop, epochs is defined in this setting as the number of times parameters θ have been updated. Thus, the number of steps we have taken to compute $\tilde{\theta}$ is ignored. If this claim causes any skepticism about whether the models trained under Reptile would outperform simply because they "are trained much more", the reader should be reminded by the fact that the baseline is presented with 954 songs per epoch, whereas these models are trained with five.

With these in mind, it should be mentioned that neither of the three models was presented with a full song during training, but rather with each song split into fixed-size chunks. This decision was made to better accommodate the memory limits of the hardware (an Nvidia T4 GPU). A full explanation of the limitations is found in Appendix C. It suffices to say that our dataset won't fit in memory. As such, each song was split into chunks of 200 events (i.e., window size of 200), starting from the beginning and moving towards the end of the song at a rate of 50 events per chunk (i.e, stride size of 50). Thus, a song of, say, 800 events would be transformed into 13 chunks, which leads us to the topic of batch size.

At training, the Reptile models were presented with batches of 64 chunks, whereas the baseline was presented with batches of 2048 chunks. The rationale for the latter is that training of the baseline under a regimen of 64 batches would take approximately 250 hours until completion.

Finally, another aspect that needs addressing is the learning rate. The baseline and Reptile inner-loop training was conducted with an $\alpha = 0.001$, value borrowed from Oore et al. (2018). Since Reptile also dictates a learning rate for the meta-loop, this value was set to $\beta = 0.01$. The reason for choosing this value comes from the Reptile experiments done by Nichol et al. (2018), where the meta-loop learning rate tends to be ten times higher than the inner rate.

A summary of the parameters used in the training procedures is found in Table 3.1.

	Baseline	Performance RNN	C-RNN-GAN
Num. Layers	1	3	2
Num. Units	256	512	350
Dropout Rate	0.0	0.3	0.3
Learning Rate	0.001	0.001	0.001
Meta Rate	N/A	0.01	0.01
Batch Size	2048	64	64
Meta Epochs	250	250	250
Inner Epochs	N/A	3	3
Has bias vector	No	Yes	Yes
Gradient Clipped	No	Yes	Yes

Table 3.1: Summary of parameters used to construct and train the three models featured in this paper

3.5 Evaluation

Generally, DL models work by the principle of likelihood maximization, which simply says to choose the parameters that maximize the probability that the model assigns to the training data (Goodfellow, 2016). Formally, this means selecting the parameters that maximize $\prod_{i=1}^{N} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$:

$$\boldsymbol{\theta}^* = \arg\max \prod_{i=1}^{N} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{3.4}$$

However, calculating the product over many probabilities is prone to numerical problems such as underflow (Goodfellow, 2016). This is alleviated by calculating $\boldsymbol{\theta}^*$ in $\log$ space where the product is transformed into a sum:

$$\boldsymbol{\theta}^* = \arg\max \sum_{i=1}^{N} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{3.5}$$

That being said, calculating maximum likelihood can be thought of minimizing the Kullback-Leibler (KL) divergence: minimizing the dissimilarity between the data generating distribution p_{data} and the model p_{model}. Thus, $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} D_{KL}(p_{\text{data}}(\boldsymbol{x}) \| p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}))$. The KL divergence is given by:

$$DL_{KL} = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log p_{\text{data}}(\boldsymbol{x}) - \log p_{\text{model}}(\boldsymbol{x})]. \tag{3.6}$$

Having all these in mind, $\log p_{\text{data}}$ is a result of the data-generating process, and not the model. Therefore, a final simplification is applied where the maximum likelihood estimate would be calculated as:

$$\boldsymbol{\theta}^* = -\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log p_{\text{model}}(\boldsymbol{x})]. \tag{3.7}$$

Note that Eq. 3.5 is the same as Eq. 3.7. The reason why this is brought up, is because negative log-likelihood (NLL; Eq. 3.7) is used both as a loss function and as an evaluation metric (also for generative models; Borji, 2018; Yu et al., 2016).

However, as Borji (2018) notes, NLL is uninformative about the quality of the samples generated and it does not allow to answer whether the generative network is simply memorizing training examples. As such, this study will rely on other evaluation metrics when assessing generative models: the number of statistically different bins (NDB; Richardson & Weiss, 2018), polyphony, scale consistency, repetitions, and tone span (Mogren, 2016).

NDB is an evaluation metric specifically designed for generative models to measure the diversity of the generated samples. It follows the intuition that given two samples from the same distribution, the number of samples that fall into a bin (read as: class or cluster) should be the same (Richardson & Weiss, 2018). Let $I_B(\mathbf{x})$ be the indicator function for bin B. Then $I_B(\mathbf{x}) = 1$ if the sample falls into B, and zero otherwise. Let also $\{\mathbf{x}_i^p\}$ define N_p samples from a p distribution (e.g., test set samples) and $\{\mathbf{x}_i^q\}$ be the N_q samples from a q distribution (e.g., generated samples). If $p = q$ then the followings is also true:

$$\frac{1}{N_p}\sum_i I_B(\mathbf{x}_i^p) \approx \frac{1}{N_q}\sum_i I_B(\mathbf{x}_i^q) \tag{3.8}$$

The pooled sample proportion P_B is the proportion of samples (from the joined sets) that fall into B. The standard error for bin B is given by:

$$SE_B = \sqrt{P_B(1 - P_B)[1/N_p + 1/N_q]} \tag{3.9}$$

The test statistic is a z-score $z = \frac{P_B^p - P_B^q}{SE_B}$ where P_B^p and P_B^q are the proportions of each sample that fall into B. If z is smaller than a threshold (a significance level such as $\alpha = 0.05$), then the samples within that bin are statistically different. This whole process is repeated for each bin, and the number of statistically different bins is reported. The selection of bins is done through a K-means clustering performed on the $\mathbf{x}^p$ samples. Each of the generated samples $\mathbf{x}^q$ is by assigned to the nearest L_2 K centroid (Richardson & Weiss, 2018).

The more domain-specific evaluation measures are adapted from Mogren (2016). Polyphony measures how often a minimum of two tones are played simultaneously (when the start time is the same). It should be noted that this is a restrictive metric: it can give a low score to music where the two notes start at different times. Scale consistency is the fraction of notes that are part of a standard scale (e.g., major, minor, lydian, etc.; see Appendix B for a complete list with examples), and reporting the results for the best matching scale. Repetitions counts consecutive subsequences of notes, and tone span is the difference between the lowest note and highest note (counted in semi-tones; Mogren, 2016).

The choice of NDB comes from the fact that it is good at detecting overfitting (Borji, 2018), whereas the domain-specific metrics should favor models that generate high fidelity samples. In addition, these evaluation metrics have the benefit of being model-agnostic; they do not require a specific type of generative model.

3.6 Utilities

All of the experiments have been conducted in the programming language Python version 3.7.3 (van Rossum, 2019) on a Nvidia T4 GPU as provided by Google Colaboratory (Google, 2019). The models have been built entirely with PyTorch version 1.0.1 (Paszke et al., 2017), while the dataset has been processed with a combination of libraries: mido version 1.2.9 (Bjørndalen, 2018) for MIDI processing, numpy version 1.16.3 (van der Walt, Colbert, & Varoquaux, 2011) and pandas version 0.23.4 (McKinney, 2010) for processing arrays and data frames. Finally, scikit-learn version 0.20 (Pedregosa et al., 2011) was used for its implementation of K-means clustering. The PyTorch reimplementation of Performance-RNN made by Lee (2019) was used as an example for developing all three models featured in this paper.

4 Results

To establish the degree to which the music of a Reptile-based model is comparable to (1) a model trained on whole dataset and (2) songs from the dataset, a total of three models have been trained (see Table 3.1 for a list of training parameters). Training of the baseline finished after 27 hours, 9 hours for the Reptile adaptation of Performance-RNN, and 19 hours for the Reptile adaptation of C-RNN-GAN.

Afterwards, the three have been tasked to generate 125 songs of variable lengths - between 90% and 110% of the median length (34,222) of songs in the training set - with five different temperature settings ranging from 0.8 to 1.2. The generating process took approximately 1 hour per model per temperature. The resulting samples were then evaluated against the samples in the test set using the metrics described in Section 3.5: polyphony, repetitions, tone span, scale consistency, and number of statistically different bins ($N_{bins} = 20$; arbitrary choice).

The results in Table 4.1 underline several things about the trained models, the most important one being that all three lack the concept of motif (the presence of a dominant/recurring theme within a piece). The clear giveaway sign is the substantial lack of repetition in the generated samples. This is enforced by the larger levels of polyphony in the three models when compared to the test set. In addition, all models seems to be unrestrained with respect to using the whole gamut of notes they have available, but considering that a piano keyboard has 88 keys and the MIDI standard allows for 128 values (notes), it seems that our models might be heading into the right direction. When it comes to playing in a scale, all models show are equal across the board, with extremely minor variations. Having said these, the domain-specific evaluation metrics indicate that the models are fairly close to eachother but that should not change the fact that there is still room for improvement when compared to the real samples.

Having said that, the landscape changes a bit when looking at NDB, our other evaluation metric. With seven to ten bins statistically different (out of 20), our models seem to display a great deal of disparity form the test samples, even though they somewhat agree between themselves - in other words, they don't seem to be very different from each other.

With these result we have gathered enough information to answer our research questions, where we could claim that our C-RNN-GAN implementation produces some of the most best results (as it scores best on 3/4 domain-specific metrics). However, a deep-dive into the performance at training time paints a whole different picture.

Model	Temperature	Polyphony	Repetitions	Tone Span	Scale Consistency	NDB
Test Set	N/A	*0.4915*	*814.3280*	*66.9440*	*0.6827*	N/A
Baseline	0.8	0.5707	**388.0160**	78.4480	0.6770	11
Baseline	0.9	0.6054	373.0560	81.3120	0.6747	10
Baseline	1.0	0.6112	347.3600	83.0560	0.6728	10
Baseline	1.1	0.6292	334.4160	84.4400	0.6722	**7**
Baseline	1.2	0.6060	316.8160	85.0480	0.6726	9
Performance-RNN	0.8	0.5684	377.8240	78.8400	0.6760	9
Performance-RNN	0.9	0.5852	364.9680	81.4880	0.6757	8
Performance-RNN	1.0	0.6219	354.0800	83.2400	0.6730	9
Performance-RNN	1.1	0.6342	343.2240	84.4640	0.6721	9
Performance-RNN	1.2	0.6061	312.2480	85.2400	0.6723	10
C-RNN-GAN	0.8	**0.5505**	369.5760	**77.7920**	**0.6776**	9
C-RNN-GAN	0.9	0.5940	370.5040	81.4320	0.6749	9
C-RNN-GAN	1.0	0.6060	348.3520	83.2640	0.6733	8
C-RNN-GAN	1.1	0.6343	340.4160	84.6800	0.6730	11
C-RNN-GAN	1.2	0.6057	313.6080	85.0080	0.6726	10

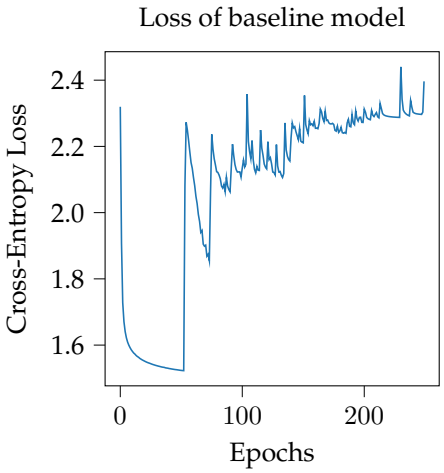
Table 4.1: Results of the evaluation process of the generated sample. For comparison, relevant statistics of the test samples have been included. The values which are closest to the test sample are in bold and highlighted in green.

When looking at Figure 4.1 we gain an insight on how well the training has gone. First and foremost, the loss of Performance-RNN is as one would expect: the model starts somewhat poorly, and then incrementally tunes itself to perform better (Figure 4.1b). The diminishing up-and-down motions are most likely indicative of the optimizer's carving its path to convergence to a (hopefully) global minima.

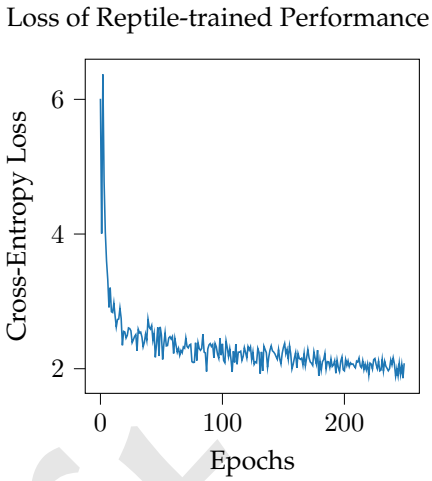
Figure 4.1a is somewhat perplexing at first glance. The loss drops quickly, only to jump back to its starting position after ≈50 epochs. A simple - and most likely true - explanation is that the learning rate is too big for the model. Considering the granularity with which all models have been trained, it could be that the model has escaped a good local minima due to the high learning rate. This should not be cause for any concern as further training would bring back the network to a similar position (it should probably reminded to the reader that the experiments in this papers are done with only 250 iterations, whereas more sophisticated DL experiments are somewhere in the order of thousands of iterations).

Finally, we have the losses of the GAN, a bewildering tableau with more questions than answers (Figure 4.1c). Firstly, it should be mentioned that GANs in and of themselves are somewhat finicky to train. Secondly, in a normal game of minimax, the loss of the generator and that of the discriminator should alternate adversarially: when one goes up, the other goes down. Here, they move together, to the point where both models reach a loss of zero - meaning that the generator is always able to fake the discriminator and the discriminator is always able to distinguish the fake samples from the real samples. This is likely due to the implementation of feature matching where instead of being trained in a competition, they are trained independently. However, this is the only explanation that I can come up with, and without other data with respect to the way the training progresses, it is difficult to say with certainty.

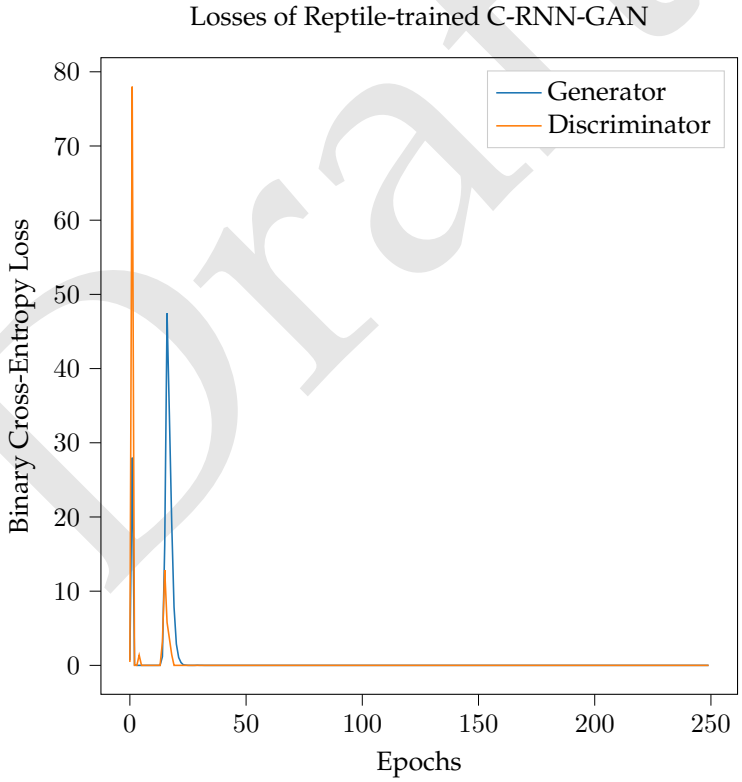
Having said that, whichever is the reason, does not change the evidence to the study's research questions. Respectively, our data suggests that models trained under Reptile are directly competitive to the baseline (a model trained on the entire dataset). However, this does not mean that they are "good enough". When compared to our test samples, the data points generated by the Reptile models fail to reach the same level of quality, a limitation likely imposed by the number of training iterations.



(a) Baseline loss at training time



(b) Performance-RNN loss at training time



(c) C-RNN-GAN loss at training time

Figure 4.1: Training losses of the three models featured in this paper. Note that the C-RNN-GAN reimplementation has a different loss function than the other two. Also to note, the loss of C-RNN-GAN is zero for both the generator and discriminator throughout most of the training time.

5 Discussion

6 Conclusion

Bibliography

Bjørndalen, O. M. (2018). Mido.

Borji, A. (2018). Pros and Cons of GAN Evaluation Measures. *arXiv:1802.03446 [cs]*. arXiv: 1802.03446 [cs]

Briot, J.-P., Hadjeres, G., & Pachet, F. (2017). Deep Learning Techniques for Music Generation - A Survey. *arXiv:1709.01620 [cs]*. arXiv: 1709.01620 [cs]

Chen, L., Dai, S., Tao, C., Shen, D., Gan, Z., Zhang, H., … Carin, L. (2018). Adversarial Text Generation via Feature-Mover's Distance. *arXiv:1809.06297 [cs]*. arXiv: 1809.06297 [cs]

Chen, W.-Y., Liu, Y.-C., Kira, Z., Wang, Y.-C. F., & Huang, J.-B. (2018). A Closer Look at Few-shot Classification.

Cheng, Z., Sun, H., Takeuchi, M., & Katto, J. (2018). Deep Convolutional AutoEncoder-based Lossy Image Compression. *arXiv:1804.09535 [cs]*. arXiv: 1804.09535 [cs]

Clouâtre, L., & Demers, M. (2019). FIGR: Few-shot Image Generation with Reptile. *arXiv:1901.02199 [cs, stat]*. arXiv: 1901.02199 [cs, stat]

Dale, R. (2016). The return of the chatbots. *Natural Language Engineering*, *22*(5), 811–817. doi:10.1017/S1351324916000243

Dong, H.-W., Hsiao, W.-Y., Yang, L.-C., & Yang, Y.-H. (2017). MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment. *arXiv:1709.06298 [cs, eess]*. arXiv: 1709.06298 [cs, eess]

Eck, D., & Schmidhuber, J. [Juergen]. (2002). *A First Look at Music Composition Using LSTM Recurrent Neural Networks*. Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale.

Fan, Y., Qian, Y., Xie, F.-L., & Soong, F. K. (2014). TTS synthesis with bidirectional LSTM based recurrent neural networks. In *INTERSPEECH*.

Finn, C., Abbeel, P., & Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *arXiv:1703.03400 [cs]*. arXiv: 1703.03400 [cs]

Finn, C., & Levine, S. (2016). Deep Visual Foresight for Planning Robot Motion. *arXiv:1610.00696 [cs]*. arXiv: 1610.00696 [cs]

Frey, B. J., Hinton, G. E., & Dayan, P. (1996). Does the Wake-sleep Algorithm Produce Good Density Estimators? In *Advances in Neural Information Processing Systems* (pp. 661–667). MIT Press.

Friberg, A., Bresin, R., & Sundberg, J. (2006). Overview of the KTH rule system for musical performance. *Advances in Cognitive Psychology*, *2*(2-3), 145–161.

Goodfellow, I. (2016). NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv:1701.00160 [cs]*. arXiv: 1701.00160 [cs]

Goodfellow, I., Bengio, Y., Courville, A., & Bach, F. (2016). *Deep Learning*. Cambridge, Massachusetts: The MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative Adversarial Networks. *arXiv:1406.2661 [cs, stat]*. arXiv: 1406.2661 [cs, stat]

Google. (2019). Colaboratory - Frequently Asked Questions. https://research.google.com/colaborator

Google Magenta. (2017). Performance RNN: Generating Music with Expressive Timing and Dynamics. https://magenta.tensorflow.org/performance-rnn.

Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Berlin Heidelberg: Springer-Verlag.

Graves, A. (2013). Generating Sequences With Recurrent Neural Networks. *arXiv:1308.0850 [cs]*. arXiv: 1308.0850 [cs]

Graves, A., & Schmidhuber, J. [Jürgen]. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*. IJCNN 2005, *18*(5), 602–610. doi:10.1016/j.neunet.2005.06.042

Hawthorne, C., Stasyuk, A., Roberts, A., Simon, I., Huang, C.-Z. A., Dieleman, S., … Eck, D. (2018). Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset. *arXiv:1810.12247 [cs, eess, stat]*. arXiv: 1810.12247 [cs, eess, stat]

Hochreiter, S., & Schmidhuber, J. [Jurgen]. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735. doi:10.1162/neco.1997.9.8.1735

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2016). Image-to-Image Translation with Conditional Adversarial Networks. *arXiv:1611.07004 [cs]*. arXiv: 1611.07004 [cs]

Jenal, A., Savinov, N., Sattler, T., & Chaurasia, G. (2019). RNN-based Generative Model for Fine-Grained Sketching. *arXiv:1901.03991 [cs]*. arXiv: 1901.03991 [cs]

Karras, T., Laine, S., & Aila, T. (2018). A Style-Based Generator Architecture for Generative Adversarial Networks. *arXiv:1812.04948 [cs, stat]*. arXiv: 1812.04948 [cs, stat]

Kennedy, M., & Bourne, J. (2007). *The Concise Oxford Dictionary of Music* (5th). Oxford: Oxford University Press.

Knight, W. (2016). AI winter isn't coming, says Baidu's Andrew Ng. https://www.technologyreview.co winter-isnt-coming/.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25* (pp. 1097–1105). Curran Associates, Inc.

Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, *350*(6266), 1332–1338. doi:10.1126/science.aab3050

Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2019). The Omniglot Challenge: A 3-Year Progress Report. *arXiv:1902.03477 [cs]*. arXiv: 1902.03477 [cs]

Larochelle, H., Finn, C., & Ravi, S. (2017). *Few-Shot Distribution Learning for Music Generation*. AI-ON.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. doi:10.1038/nature14539

Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., … Shi, W. (2016). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *arXiv:1609.04802 [cs, stat]*. arXiv: 1609.04802 [cs, stat]

Lee, Y. (2019). Event-based music generation with RNN in PyTorch. Contribute to djosix/Performance-RNN-PyTorch development by creating an account on GitHub.

Lighthill, J. (1972). Artificial Intelligence: A General Survey. In *Artificial Intelligence: A paper symposium*. Science Research Council.

Marcus, G. (2018). Deep Learning: A Critical Appraisal. *arXiv:1801.00631 [cs, stat]*. arXiv: 1801.00631 [cs, stat]

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference* (pp. 51–56).

MIDI Association. (2019). The Official MIDI Specifications. https://www.midi.org/midi/specifications.

Minsky, M., Papert, S. A., & Bottou, L. (1988). *Perceptrons* (Reissue edition). Cambridge, MA: MIT Press.

Mogren, O. (2016). C-RNN-GAN: Continuous recurrent neural networks with adversarial training. *arXiv:1611.09904 [cs]*. arXiv: 1611.09904 [cs]

Nichol, A., Achiam, J., & Schulman, J. (2018). On First-Order Meta-Learning Algorithms. *arXiv:1803.02999 [cs]*. arXiv: 1803.02999 [cs]

Nilsson, N. J. (2009). Speed Bumps. In *The Quest for Artificial Intelligence* (1 edition, pp. 381–409). Cambridge ; New York: Cambridge University Press.

Oore, S., Simon, I., Dieleman, S., Eck, D., & Simonyan, K. (2018). This Time with Feeling: Learning Expressive Musical Performance. *arXiv:1808.03715 [cs, eess]*. arXiv: 1808.03715 [cs, eess]

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., … Lerer, A. (2017). Automatic differentiation in PyTorch.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Qiu, X., Zhang, L., Ren, Y., Suganthan, P. N., & Amaratunga, G. (2014). Ensemble deep learning for regression and time series forecasting. In *2014 IEEE Symposium on Computational Intelligence in Ensemble Learning (CIEL)* (pp. 1–6). doi:10.1109/CIEL.2014.7015739

Ravi, S., & Larochelle, H. (2016). Optimization as a Model for Few-Shot Learning.

Rezende, D. J., Mohamed, S., Danihelka, I., Gregor, K., & Wierstra, D. (2016). One-Shot Generalization in Deep Generative Models. *arXiv:1603.05106 [cs, stat]*. arXiv: 1603.05106 [cs, stat]

Richardson, E., & Weiss, Y. (2018). On GANs and GMMs. *arXiv:1805.12462 [cs]*. arXiv: 1805.12462 [cs]

Rumelhart, D. E., Hinton, G. E., & Williams, R. (1986). Learning Representations by Back Propagating Errors. *Nature*, *323*, 533–536. doi:10.1038/323533a0

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved Techniques for Training GANs. *arXiv:1606.03498 [cs]*. arXiv: 1606.03498 [cs]

Schmidhuber, J. [Jürgen]. (1987). *Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook* (Diploma Thesis, Institut für Informatik, Technische Universität München, Munich).

Sutskever, I., Martens, J., & Hinton, G. (2011). Generating Text with Recurrent Neural Networks. In L. Getoor & T. Scheffer (Eds.), *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 1017–1024). ICML '11. ACM.

University of Minnesota. (2019). International Piano-e-Competition. http://www.piano-e-competition.com/.

van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, *13*(2), 22–30. doi:10.1109/MCSE.2011.37

van Rossum, G. (2019). Python. Python Software Foundation.

Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., & Wierstra, D. (2016). Matching Networks for One Shot Learning. *arXiv:1606.04080 [cs, stat]*. arXiv: 1606.04080 [cs, stat]

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., … Tsing, R. (2017). StarCraft II: A New Challenge for Reinforcement Learning. *arXiv:1708.04782 [cs]*. arXiv: 1708.04782 [cs]

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560. doi:10.1109/5.58337

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The Microsoft 2017 Conversational Speech Recognition System. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938). doi:10.1109/ICASSP.2018.8461870

Yu, L., Zhang, W., Wang, J., & Yu, Y. (2016). SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. *arXiv:1609.05473 [cs]*. arXiv: 1609.05473 [cs]

Zhang, R., Che, T., Ghahramani, Z., Bengio, Y., & Song, Y. (2018). MetaGAN: An Adversarial Approach to Few-Shot Learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31* (pp. 2371–2380). Curran Associates, Inc.

Zhang, X., Zhao, J., & LeCun, Y. (2015). Character-level Convolutional Networks for Text Classification. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 28* (pp. 649–657). Curran Associates, Inc.

A Composer labelling

Canonical Composer	Genre
Alban Berg	modernism
Alexander Scriabin	romanticism
Antonio Soler	baroque
Carl Maria von Weber	classical
Charles Gounod / Franz Liszt	classical
Claude Debussy	impressionist
César Franck	romanticism
Domenico Scarlatti	baroque
Edvard Grieg	romanticism
Felix Mendelssohn	romanticism
Felix Mendelssohn / Sergei Rachmaninoff	romanticism
Franz Liszt	romanticism
Franz Liszt / Camille Saint-Saëns	romanticism
Franz Liszt / Vladimir Horowitz	romanticism
Franz Schubert	classical
Franz Schubert / Franz Liszt	classical
Franz Schubert / Leopold Godowsky	classical
Fritz Kreisler / Sergei Rachmaninoff	modernism
Frédéric Chopin	romanticism
George Enescu	modernism
George Frideric Handel	baroque
Georges Bizet / Ferruccio Busoni	romanticism
Georges Bizet / Moritz Moszkowski	romanticism
Georges Bizet / Vladimir Horowitz	romanticism
Giuseppe Verdi / Franz Liszt	romanticism
Henry Purcell	baroque
Isaac Albéniz	romanticism
Isaac Albéniz / Leopold Godowsky	romanticism
Jean-Philippe Rameau	baroque
Johann Christian Fischer / Wolfgang Amadeus Mozart	classical
Johann Pachelbel	baroque
Johann Sebastian Bach	baroque
Johann Sebastian Bach / Egon Petri	baroque

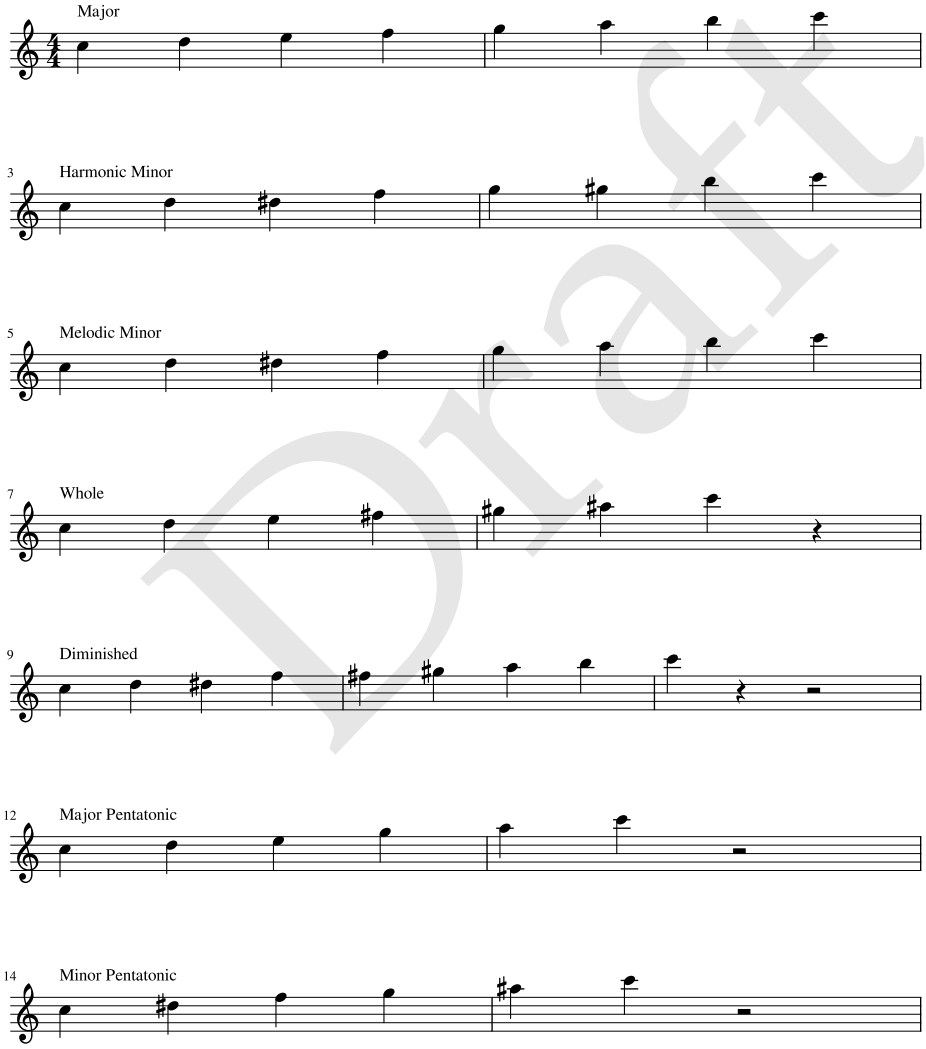
(continued)

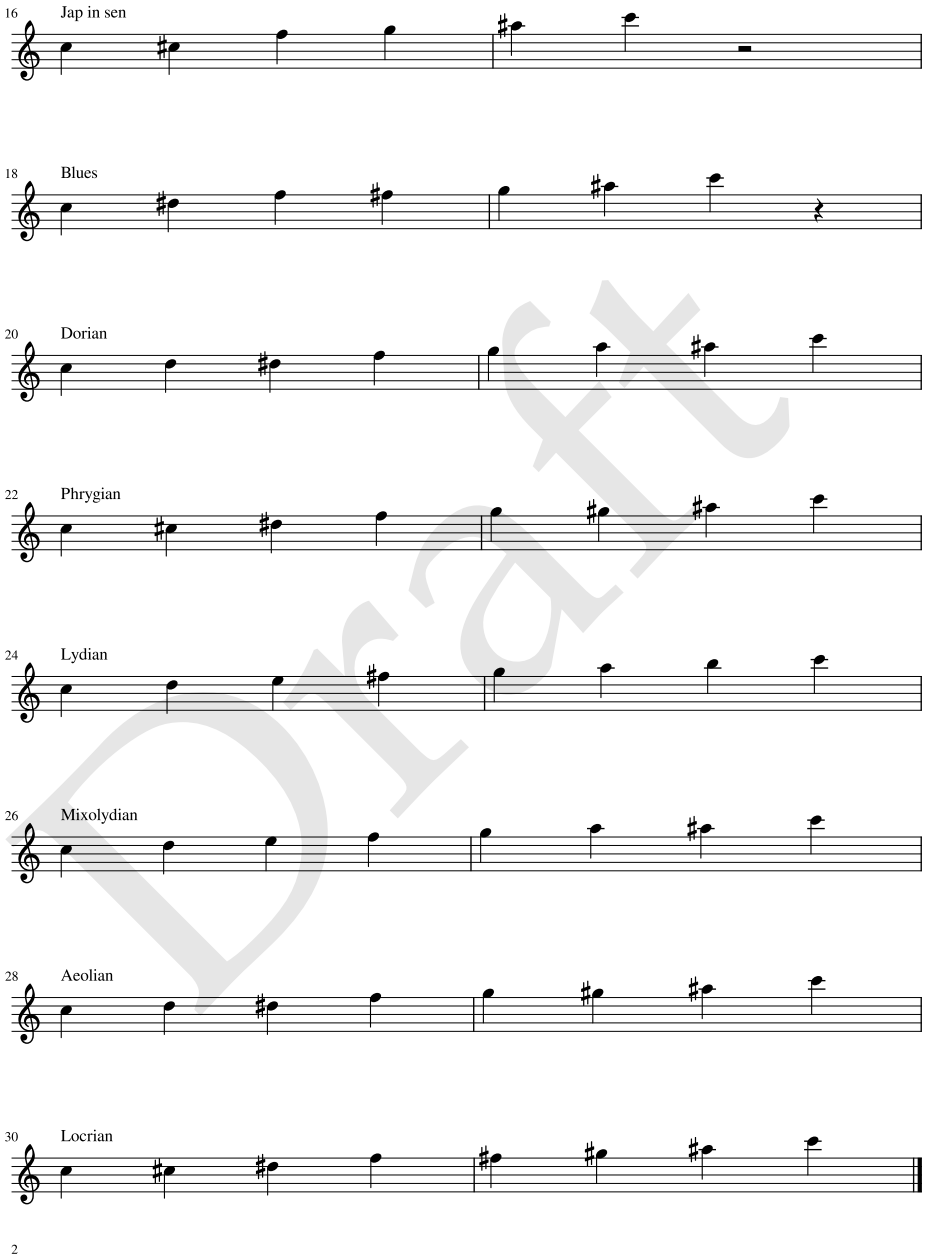
Canonical Composer	**Genre**
Johann Sebastian Bach / Ferruccio Busoni	baroque
Johann Sebastian Bach / Franz Liszt	baroque
Johann Sebastian Bach / Myra Hess	baroque
Johann Strauss / Alfred Grünfeld	romanticism
Johannes Brahms	romanticism
Joseph Haydn	classical
Leoš Janáček	romanticism
Ludwig van Beethoven	classical
Mikhail Glinka / Mily Balakirev	classical
Mily Balakirev	romanticism
Modest Mussorgsky	romanticism
Muzio Clementi	classical
Niccolò Paganini / Franz Liszt	classical
Nikolai Medtner	modernism
Nikolai Rimsky-Korsakov / Sergei Rachmaninoff	romanticism
Pyotr Ilyich Tchaikovsky	romanticism
Pyotr Ilyich Tchaikovsky / Mikhail Pletnev	romanticism
Pyotr Ilyich Tchaikovsky / Sergei Rachmaninoff	romanticism
Richard Wagner / Franz Liszt	romanticism
Robert Schumann	romanticism
Robert Schumann / Franz Liszt	romanticism
Sergei Rachmaninoff	romanticism
Sergei Rachmaninoff / György Cziffra	romanticism
Sergei Rachmaninoff / Vyacheslav Gryaznov	romanticism
Wolfgang Amadeus Mozart	classical

Table A.1: Genres assigned to the unique entries of composers in the MAESTRO dataset. Where two composers are present, indicates a collaboration or an adaptation of the latter's work made by the former

B Scales used in evaluation

All of the following examples have the note C as the root of the scale.




16
Jap in sen
18
Blues
20
Dorian
22
Phrygian
24
Lydian
26
Mixolydian
28
Aeolian
30
Locrian

C Memory Limitations

The length of our songs varies from 2,100 to 202,046 events. Coupled with the fact that our model is presented with one-hot encoded vectors, this translates into matrices of sizes 2,100 $\times$ 416 up to 202,046 $\times$ 416. A better illustration would be to assume that all training songs have a length equal to our mean length (44,827) and that we are training just the baseline. For a batch size of 64, this means we are dealing with a tensor of size 64 $\times$ 44,827 $\times$ 416 elements, where one element is 4 bytes. This translates into a tensor that is 4.77 gigabytes. To add on top of that, our training dictates generating a second tensor that is equal in the number of elements. Furthermore, PyTorch (Paszke et al., 2017), the machine learning library which is used in this paper, requires each element in our target tensor to be a double-precision floating point number - i.e., 8 bytes per element. By now we have exhausted the entire memory (12.5 gigabytes in Google Colaboratory) of our GPU leaving no space for any of our networks.

Granted, someone could make the argument to lower the batch size of the songs to 32. However, due to high variance in the length of our songs, we could risk a situation were all of the songs have a length of 100,000 and up (there are 68 such songs in the training set) leading to a tensor that is 5.32 gigabytes (single-precision floating point).

A batch size of 16 would be a more compelling alternative: 2.66 GB single-precision floating point with a 5.32 GB double-precision floating leaves 4.52 GB of space. However, if we account the size of the libraries imported into the training script and the hidden state tensors (batch size $\times$ vocabulary size) puts us dangerously close to the memory limits of the GPU.