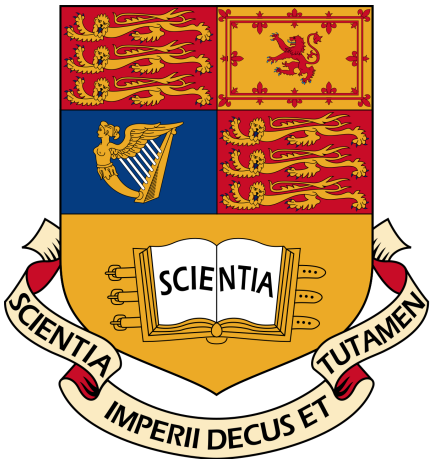
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2018



Project Title:	**Comper: A Collaborative Musical Accompaniment System using Deep Latent Vector Models**
Student:	**Chan Jun Shern**
CID:	**00861477**
Course:	**EEE4**
Project Supervisor:	**Professor Yiannis Demiris**
Second Marker:	**Dr Javier Barria**

""If I have seen further, it is only by standing on the shoulders of giants.""

Isaac Newton

This quote, made famous in a 1675 letter from Isaac Newton to Robert Hooke (Newton, 1675), captures beautifully the nature of scientific research in an age of collaboration. The work we do builds upon the work of those who came before us, as well as the support from those who stand around us. It is my hope that in writing this, I may offer a shoulder to all who come after me.

IMPERIAL COLLEGE LONDON

Abstract

Faculty of Engineering
Department of Electrical and Electronic Engineering

Electrical and Electronic Engineering (MEng)

Comper: A Collaborative Musical Accompaniment System using Deep Latent Vector Models

by CHAN Jun Shern

The field of Artificial Intelligence (AI) has made great advances in the past decade, with systems surpassing humans in many tasks previously believed to be impossible for AI. One of the great challenges that remain is to create AI capable of creativity; in particular, music is one such field which remains largely unconquered by AI systems. This project provides an evaluation of the current state-of-the-art in relevant technologies, and concludes that although there has been substantial prior work in algorithmic composition, few systems possess the versatility to participate in live collaborative music-making.

To this end, the project takes a deep learning approach using a variational autoencoder to learn a semantically-meaningful latent vector representation for musical phrases based on a dataset of music. It is shown that the system is capable of generating original and realistic musical content by sampling from this latent space. Furthermore, by using recurrent neural networks to predict next-step musical accompaniments, we are able to create a duet-like system which produces musically-relevant accompaniments given input from a human musician on a digital instrument.

Finally, the project applies these techniques to a real-time musical accompaniment system named Comper, which provides a unique and novel way for human musicians to interact and collaborate with musical AI.

Acknowledgements

Firstly, I would like to thank my supervisor Professor Yiannis Demiris for all his guidance and time spent trading ideas with me especially during the early days of the project. Without his feedback and constant encouragement throughout the year, this project would not have been possible.

I am grateful also to Mason Bretan, for his willingness to help and give advice to a stranger halfway across the world. Many of the ideas in this project were inspired by his work, and his advice during our discussions helped to inform the direction of my project when I did not know where to start.

Next, I would like to thank Tobias Fischer, for working with me to overcome the roadblocks I faced and for giving me much needed perspective when I most needed it.

Finally, to my family and my friends I will always be grateful. To my friends for making the days lighter, and to my family for their love and for raising me to enjoy the wonders of art and science in tandem.

Contents

List of Figures

Chapter 1

Introduction

1.1 Goals

The main goal of project is to produce a musical system capable of collaborating with a human musician in real-time. Each phase of work was led by different guiding objectives which have also evolved following the insights gained from the work as it unfolded. Broadly, the key goals which have guided the development of the project are:

1. Investigate the current state-of-the-art in artificial intelligence (AI) for music composition, and **how these techniques can be applied** to a real-time musical accompaniment task
2. Investigate the potential of **latent vector models** to capture meaningful representations of music, and their potential to be used a generative manner for music composition
3. Implement a **system for musical collaboration** in a multi-agent environment

1.2 Scope

A fully autonomous robotic musician would encompass a large breadth of technologies spanning different fields. Given limited time and resources for this project, we decide to focus on the key challenge of **algorithmic composition**. For other aspects, we either leave out the tasks due to being less critical to our chosen evaluation of the system, or opt for specialized experimental conditions which make the tasks simpler to handle.

In particular, we simplify the problem of machine listening by using Musical Instrument Digital Interface (MIDI) -compatible instruments which produce symbolic data instead of raw audio. In terms of producing sound, we simply play the synthesized MIDI output through ordinary computer speakers which avoids the difficulties of musical mechatronics.

Furthermore, music itself has a wide variety of styles, which is not only difficult to implement for, but also troublesome to evaluate the success of. As such, we decide to focus on developing accompaniments for Western music, in particular Western Pop and Rock genres, and generate accompaniments in the voice of keyboard-based instruments (which also have a distinct style from say, a guitar or saxophone).

1.3 Overview

Chapter 2 sets the scene for our problem by diving into previous work that has been done in the field of machine-generated music. It begins with a brief look at the interplay between technology and music, touching on the different ways people have used technology to enhance the composition, performance and consumption of music. Then, it provides a summary of algorithmic composition techniques, noting that currently, the approaches which have come closest to achieving human-level machine accompaniment have come from the field of deep learning. Finally, an introduction to deep learning is provided to provide the necessary technical background for the techniques used in the following sections.

Chapter 3 refines the objectives of the project in a more concrete manner, laying out the definition and requirements of a collaborative musical system, which we will then revisit in Chapter 7 to evaluate the the success of the project.

Chapter 4 and Chapter 5 detail the main bulk of the work covered in the project. **Chapter 4** covers the design of the accompaniment algorithm from problem formulation to experimentation and testing of various deep learning models, showing the results from each experiment and discussing the relative strengths and weaknesses of each model. This finally leads into a decision on which models are suitable for the task of musical accompaniment, and simultaneously inspires an interesting form of interaction which guides the development of the final system.

Chapter 5 documents the implementation of the whole project, pointing out the technical difficulties at each step and describing how they were tackled in detail, including all the hardware and software used or produced for the project. This chapter covers implementation details of both Chapter 4's experimental work and the final product used for demonstration including the composer program and user interface, which is called Comper.

Chapter 6, Chapter 7 and Chapter 8 evaluate the outcomes of the project in different terms. **Chapter 6** tests the output of Comper in terms of musical quality, and discusses shortcomings based on those tests. **Chapter 7** looks back at the overarching requirements set out in Chapter 3 to see how well the project achieves its primary objectives. **Chapter 8** takes a bird's eye view of the situation, evaluating the success of the project as a whole, and the limitations of the resulting system.

Finally, **Chapter 9** wraps up the project with closing remarks, while also discussing any considerations for future work.

Chapter 2

Background

2.1 Machine Musicianship

Machine Musicianship is an ongoing field of research which studies how technology can emulate the role of human musicians by carrying out musical tasks. There is a broader definition of musical technologies which includes instruments and tools developed to be used by musicians in their workflow; in general a better understanding of the topics in machine musicianship can aid the development of new tools and new ways of thinking for music creation and consumption.

A Survey of Robotic Musicianship (Bretan and Weinberg, 2016) gives a definition for Robotic Musicianship, which draws a helpful distinction between musical mechatronics and machine musicianship. Musical mechatronics refers to the electromechanical control of musical instruments by robots to produce music (such as controlling the fingers of a robot to press the keys on a piano), whereas machine musicianship relates to a range of topics including human-robot interaction and collaboration, robot understanding of contextual cues and machine listening, and algorithmic composition.

The following sections give a brief introduction to topics in a number of fields related to machine musicianship, beginning with examples of possible human-machine interaction situations, leading into how machines can interpret the musical and contextual cues given by a co-performer, then concluding our survey with examples of work in AI composition.

We begin by discussing some of the different ways that machines and humans can interact in musical settings.

2.1.1 The Role of Machines in Music

The foremost applications of technology in music in today's music industry are as tools which aid and extend composition and performance. This is seen in the evolution of musical instruments (for example the birth of electric guitars and digital pianos from their acoustic counterparts), peripheral tools which improve a musician's workflow (software such as digital audio workstations), and even new ways of creating and remixing music (sample-based electronic music and DJ-ing).

These tools are important developments in the music industry, providing new ways to augment human ability and new styles of music as direct consequence of the medium of creation. As technology becomes more sophisticated, music technology can shift from being static tools to intelligent decision-making systems, which may hold their own as musicians or interact with musicians in a synergistic manner. For the purpose of this project, we choose to focus on this latter application, looking at how machines can take on collaborative roles in music creation.

Human-machine collaboration for musical tasks can be applied to many different types of interactions. The first, most obvious example is that of a "robotic bandmate". This type of interaction generally comprises a multi-agent situation where a human plays one instrument and the robot plays another, and they collaborate to produce a musical piece using different voices which complement each other.

This is certainly not the only type of interaction possible. Shared-control systems can be created for multiple agents sharing a single instrument in the same voice, for example in musical prostheses which could be used to augment a human player's performance by providing assistance in either physical or musical ability, or both. Bretan et al produced such a system in their prosthetic arm designed for an amputee drummer, using electromyography to sense the muscle intent of an amputee (Bretan et al., 2016). The system produces a drum beat using an interesting shared-control system where the prosthesis is not just a static tool, but a dynamic contributor.

The simplest form of collaboration between multiple musicians is through harmony. Harmonic composition comes with an extensive set of rules governing the use of notes and chords in particular combinations (Christensen, 2006). The difficulty with composing using only harmonic rules is that following basic harmony rules results in very rigid and predictable compositions; for more interesting compositions, more complex harmony rules are available but are not well-defined enough to constitute a programmable set of instructions. At best, composer systems based purely on harmonic rules can only produce what has been taught to them by their creator, so are limited in their ability to generate novel melodies and musical directions (this is a known characteristic of knowledge-based systems, as we will see in 2.1.4).

An alternative form of musical collaboration is that of "call-and-response" (also known by different names such as "antiphony" in classical music and "trading bars" in jazz), where one musician leads with a phrase of music, and the collaborator responds with a musically suitable reply. Pachet's Continuator (Pachet, 2003) uses this mechanism to interesting effect, resulting in a interesting performance which passes their version of a musical Turing test.

In a general musical context, musicians are expected to adapt naturally and switch between different roles or interaction styles based on cues from their partners - either one of or a combination of auditory (musical or verbal) and visual (body language) cues. Herein lies one of the greatest difficulties of the field - understanding of multimodal cues is a complicated task. Pan et al tackles this in their study of "initiative exchange" using visual and auditory (decrease in volume and head-nodding) communication between a robot and human musician (Pan, Kim, and Suzuki, 2010). However, the fact that there are no universal agreed rules for this kind of nonverbal communication means that current methods such as that of Pan et al can only be used to recognize a limited set of predetermined signals, which is not sufficient for real-world conditions.

Instead of (or as a supplement to) detecting cues from the interacting party, many human-robot interaction systems have taken the approach of attempting to predict, or anticipate, the next state of other agents. Hoffman et al (Hoffman, 2010) studied the use of anticipation for a variety of collaborative tasks, and found that having a model for anticipation can improve the fluency of the collaboration. Hoffman's work uses a different anticipatory model for each type of task, but similar ideas could be applied to musical tasks by predicting the next segment of music generated by a partner, then adapting one's own role to suit the anticipated role. Sarabia et al provide a reasonable implementation of this, using a synchronized grammars

framework to model the actions of interacting musical agents, with a robot anticipating its partner agent's next actions (Sarabia, Lee, and Demiris, 2015).

2.1.2 Music Information Retrieval

In the context of machine musicianship, machine listening is about how a system listens to music, and music information retrieval is about how to interpret that music. In a collaborative music environment, correct interpretation of music provides essential context for composing an appropriate accompaniment. Depending on the composition strategy, interpretation may be required at different levels of abstraction.

Typical listener systems begin with raw audio signals and extract features from this to build a higher-level interpretation. The target level of abstraction depends on the application of the system, but common targets of music information retrieval can be divided into ***harmonic information*** (pitch, chord, key detection), ***temporal information*** (beat, tempo, time signature detection), and other information which could include notions such as ***metadata*** (mood, genre, role) and ***mathematical representations***.

We note that music information retrieval in performed music does not only have to depend on audio data. For example, (Cicconet, Bretan, and Weinberg, 2012) used a vision system to synchronize drum strokes between a human and robot drummer, and Nisbet et al (Nisbet and Green, 2017) has had success using depth vision to capture both pitch and velocity information from a piano performance. However, the high potential for visual occlusion makes this unlikely to be a reliable information stream in a live performance setting.

Harmonic Information

One of the foremost properties of music is pitch. However, extracting pitch information from audio is a difficult task - (Derrien, 2014) has had success in pitch detection of monophonic audio signals from an acoustic guitar using frequency-domain signal processing, but the large majority of popular music makes use of polyphony with multiple voices and instruments which is far more difficult and has not yet been solved completely (Collins, 2007).

Worth mentioning is the task of score-following, which successfully employs polyphonic note tracking given a score of music which is being followed. For example, (Arzt, Widmer, and Dixon, 2012) uses dynamic time warping for accurate and robust real-time score-following. These systems can be useful for collaborative performance applications, but allow little room for improvisation which is an important facet of performed music.

That said, one would be right to point out that human musicians don't have perfect polyphonic listening either, but can still compose effective accompaniments. Many human musicians are able to interpret the current chord of the music by listening, and use this to inform the context of their compositions. This kind of audio-to-chord information retrieval has been studied as well, for example in (Lee, 2006) which recognizes chords by matching frequency spectrum chromagrams, or Pitch Class Profiles (PCP), from piano recordings to empirically-learned PCP templates of chords.

An alternative route to music information retrieval is to skip the initial step of processing raw audio waveforms, by starting directly from symbolic representations

of music. This was made possible in recent decades with the advent of digital instruments and the popular ***Music Instrument Digital Interface (MIDI)*** communications protocol (Association, 1983). MIDI provides a digital symbolic representation of music which is used by most digital instruments nowadays. This means that by using a digital piano connected to a computer, we are able to get MIDI messages which directly describe information about music being played in terms of pitch, velocity and much more. A quick primer to the MIDI protocol and its standard messages is provided in Appendix A.

After notes and chords, the remaining harmonic information of immediate interest for the application of composing accompaniments is the notion of musical key. If we can assume knowledge about the pitches of notes in a given musical piece, this is a relatively straightforward task which can be solved by matching a histogram of pitch frequencies against a PCP template of keys such as the ones provided in (Krumhansl, 2001), though as we will see in Appendix B this method is not fully robust especially in small-window predictions.

Temporal Information

Rhythm is another crucial property of music which robotic musicians must strive to understand. Of greatest interest to us are the notions of beat and tempo: beat is generally defined to be a unit of time in music - a single count, more intuitively defined by the times where we tap our feet to music. On the other hand, tempo refers to the pace of a given musical piece, usually measured in beats per minute (BPM) (Rafii, 2012).

To build up an understanding of rhythm (here used to refer to general timing information of music), there are several common tasks which must be solved: onset detection, tempo estimation, and beat tracking. This is not an exhaustive list of challenges in rhythm, but provide a good foundation towards the goal of composing accompaniments.

Onset detection is simple in the context of discrete musical notes - the onset of a note is the start time of that note. This is straightforward enough when the actual onset of notes are high in amplitude and velocity, such as the sound of a kick drum, in which case the onset can be easily found by inspecting bursts and peaks in the energy spectrum or envelope of the signal amplitude. However, not all notes are triggered equally, and a variety of methods have been developed to improve robustness of onset detection, usually based either on signal features or probabilistic models (Bello et al., 2005).

As before, it is also possible to circumvent the issue of onset detection by using digital MIDI instruments which provide note onset information in terms of MIDI message timestamps.

The next challenge is global tempo estimation, which is of interest in constant-tempo musical pieces. Ground-breaking work by Foote et al in 2001 made use of self-similarity in music to visually identify rhythmic patterns (Foote and Cooper, 2001), and since then the most successful results in tempo estimation generally exploit periodicity in music, such as Bock et al's top-performing estimator which exploits self-similarity using resonating comb-filters and neural networks (Böck, Krebs, and Widmer, 2015).

Traditionally, beat-tracking algorithms often begin with tempo estimation, which forms a prior for obtaining beat times - one such example is the work of Dixon et al (Dixon, 2001). However, recent state-of-the-art beat detection results have been obtained directly from audio, using a kind of neural network called bi-directional Long

Short-Term Memory (LSTM) recurrent neural networks (RNN) to perform frame-by-frame beat classification (Böck, Krebs, and Widmer, 2014).

Metadata and Non-traditional Representations

A big class of music information retrieval which we have neglected to mention so far is that of metadata retrieval. Metadata comprises a wide range of higher-level semantic information related to music such as the mood or genre of a piece, or identifiers such as audio fingerprints, with applications in content-based music retrieval and auto-tagging of music (often useful for organizing large collections of songs). Schedl et al give a good overview (Schedl, Gómez, and Urbano, 2014) of semantic information retrieval and annotation of music, but we will not explore these techniques in detail here since although some metadata could provide useful context, they are of less direct relevance given current implementations of accompaniment composition.

In a slight departure from traditional notions of music information, recent works in the field of music information retrieval have used ideas from mathematics and information theory to extract "good" representations of music information depending on what we need the information for. An exciting example of such work is that of (Bretan, Weinberg, and Heck, 2016), which demonstrate the use of neural network-based autoencoders to generate embeddings of MIDI music inputs, then generating new MIDI music from the embeddings which successfully retains the style of the original music. Another exciting result from deep learning is StyleNet (Malik and Ek, 2017), which uses bi-directional LSTM RNNs to transfer expression onto a (previously expressionless) musical/MIDI score. These works show that neural networks are capable of capturing and applying the nuances of expression in music, which has remained one of the more elusive objectives in the field of machine listening for music information retrieval.

2.1.3 Musical Mechatronics

For completeness, it is wise to discuss the importance of embodiment in a musical collaborator, although this is beyond the scope of this project. In general, human-machine collaboration in music composition does not require physical embodiment of the machine. Two examples of projects in human-machine duets, The Continuator (Pachet, 2003) and AI Duet (*AI Duet by Yotam Mann*), use AI agents which are not embodied physically - both achieve musical collaboration purely through communication of music information (digital and auditory) between the human player and the AI software, and found great success in mainstream popular media with their approach.

However, there is evidence that a human-robot collaboration system could benefit from a physically embodied robot. Kidd et al demonstrate in ("Effect of a robot on user perceptions") that a physically present robot improves the engagement scores of humans who interact with it. In a musical context, Hoffman et al (Hoffman and Weinberg, 2010b) find that a physically embodied robot improves synchronization in particular when playing low tempo music. For live performances as well, it is clear that a physically embodied performer adds value to the performance, which we know from the difference in performance value of a live concert versus a radio playback.

Bretan and Weinberg's Survey of Robotic Musicianship (Bretan and Weinberg, 2016) takes the argument of embodiment further, with the idea that *"there is artistic potential in the non-human characteristics of machines including increased precision, freedom of physical design, and the ability to perform fast computations."* Interestingly, this suggests that traditional music has been shaped in certain ways due to human anatomy, so in designing a physical robot with non-human morphology to play music, we might discover a new creative avenue which stands apart from human-produced music.

Decades of research in human-robot interaction suggest useful considerations for the design of a musical robot. Michalowski et al created a dancing robot named Keepon (Michalowski, Sabanovic, and Kozima, 2007), with a simple creature-like morphology consisting of just a head and body, but with enough degrees of freedom to bob its head in multiple directions. The robots Shimi ("Chronicles of a Robotic Musical Companion.") and Shimon (Hoffman and Weinberg, 2010a) use foot-tapping and head-bobbing respectively to display engagement with music. Studies into each of these robots agree that friendly physical design and context-aware gestures can improve interaction and engagement with humans, and in music, simple actions such as bobbing to a beat can result in improved user engagement.

2.1.4 Algorithmic Composition

Algorithmic music constitutes any kind of music generated following an algorithm. Composers have used algorithmic techniques for music composition long before the invention of computers: the Greek philosopher Pythagoras studied the relationship between harmony and mathematics, and sought mathematical systems for composing music (Simoni, 2003); Mozart made famous the concept of 'dice music' in the 18th century, which involved assembling a number of musical fragments in random order to create new compositions (Hedges, 1978).

Similar ideas were brought forward to modern times in algorithmic composition using computers. Mathematical ideas emerged in music composition through the use of ***fractals***, with interesting, strongly-patterned results like that seen in the work by Hsu et al (Hsü and Hsü, 1990; Hsü and Hsü, 1991). While interesting, mathematically-structured techniques by themselves have been limited to applications in very niche musical settings.

Given the wealth of generative rules and structure in music theory, a natural approach to composition is through the use of ***rule-based grammar systems***, which originate from the field of Natural Language Processing but can similarly be used to formulate syntactical rules in music. These systems such as McCormack et al's (McCormack, 1996) yield promising results especially for classical music composition, in which the rules of music theory are rigorously defined, but are less effective for contemporary music (though grammar-based systems do exist for specific subsets such as composing drum patterns (Kitani and Koike, 2010)). Clearly, the difficulty lies in formulating effective grammars which enforce good musical structure yet are flexible enough to produce interesting ideas - Gillick et al tackle this challenge with their system which automatically learns appropriate grammars from jazz music (Gillick, Tang, and Keller, 2010).

Genetic algorithms are another class of techniques which have had success in composing music. Typically, the algorithm begins with a large initial population of musical compositions (either created randomly or with some hand-crafted heuristics as a good starting point). Every composition in the population is evaluated with a fitness score, based on expert knowledge of desirable properties according to music

theory (Jacob, 1995; Özcan and Erçal, 2007) or given interactively by user feedback (Biles, 1994), and selected for reproduction based on fitness. A new generation of children compositions are born inheriting properties from their parents, with some added mutation so that improvements are possible from generation to generation, and eventually a generation may be produced with high fitness scores and therefore (hopefully) good musical properties. However, the downside of evolutionary techniques is that they require a heavy amount of human intervention for appropriate initialization, fitness evaluations and rules of mutation. Furthermore, the necessity to breed many generations means that such methods generally cannot be used for real-time improvisation.

Around the same time that expert knowledge-based systems were being explored, other strong contenders appeared using ***probabilistic approaches***. These often made use of Markov models, which learned note-to-note or phrase-to-phrase state transition probabilities from expert music to stochastically generate new pieces of music. Markov generation proved to be effective in mimicking the style of human musicians, as demonstrated by Nikolaidis et al's composer system which generated compositions in the style of jazz masters John Coltrane and Thelonious Monk (Nikolaidis and Weinberg, 2010).

Pachet's The Continuator (Pachet, 2003) also made use of a Markovian approach with a duet-system which passed a 'musical Turing test' in 2003; however Pachet acknowledged a shortcoming of Markov chains in their lack of controllability, and published his solution to the problem many years later with ***constraint-satisfaction algorithms*** (Pachet and Roy, 2011). At the cutting edge of this technology, Pachet et al recently released a full 15-song album "Hello World" (Records, 2017) composed for mainstream music, using his system FlowComposer (Pachet and Roy, 2014) to generate compositions with human-led direction.

Most recently, ***deep learning*** has taken the spotlight in AI composition. Leading the charge, the Magenta project (*Magenta*) from the Google Brain team have actively pursued the goal of creating art, including music, exclusively using neural networks as the meat of their algorithms. The basic principle of neural networks is that you can train a biologically-inspired learning model using thousands of samples of music, and it will learn patterns from that training dataset so that you can use the model to predict and generate new sequences. An early problem with using neural networks, usually Recurrent Neural Networks (RNN), for music composition was that resultant compositions lacked long-term structure, but this changed with the introduction of Long Short-Term Memory (LSTM) RNNs which allow the networks to retain information over longer sequences (Graves, 2013). Perhaps the best example of state-of-the-art AI music is from Performance-RNN (Simon and Oore, 2017), an LSTM-RNN which impressively captures expressiveness from a MIDI dataset of piano performances, to generate MIDI music which sound indistinguishable from human playing.

Another exciting result of neural networks comes from Bretan et al's work which uses a form of neural networks called autoencoders to learn an 'embedding' - the autoencoder's internal representation of information - of a given musical input. The low-dimensional embedding is used as a musically meaningful fingerprint for chunks of music, such that when used in conjunction with an LSTM-RNN it is possible to pick and use semantically relevant chunks which can be used as building blocks to compose musical output (Bretan, Weinberg, and Heck, 2016). This work was also developed into a "call-and-response" system where the AI accompaniment is able to use this music-fingerprinting system to playback musical responses from

a library of recorded musical phrases in response to a musical prompt (Bretan et al., 2017a).

Further in the use of autoencoders and embedding spaces, the Magenta team's MusicVAE system (Roberts et al., 2018a) has broken ground on using arithmetic transformations within the embedding space to tweak meaningful properties of music such as note density, adding a degree of control to which has previously been so elusive for AI composition systems. This work has also been extended recently to allow composition of multitrack measures, showing unprecedented success in AI composition of multitrack music supporting various instrument voices such as piano, drums, guitar and bass (Simon et al., 2018). This work is currently one of the few systems capable of context-aware composition, as the model supports chord-conditioned generation, such that users can specify a chord progression and have the model compose over those chords.

2.2 Deep Learning Tutorial

Machines are very good at following rules, and a great part of the success of the industrial revolution up to the current digital age has happened thanks to machines and programs which follow simple rules, and can be put to work at a large scale.

However, the tasks which these rule-following machines can solve have traditionally been limited by our own abilities to describe those rules. For example, a large number of tasks which humans handle on a daily basis are tasks which we do not understand well enough to describe, or are too tedious to list out as rules for machines to follow. These tasks include comprehension of visual and auditory information, natural language processing, and indeed, music comprehension and composition - all of which have seen an upheaval in performance by machines since the advent of machine learning.

Deep learning techniques often have an advantage over other approaches in that they are able to model highly complex behaviors simply by observing a large amount of data - an important step in machine learning known as **training**. By training on the appropriate dataset and formulating the right objectives to optimize, sufficiently 'deep' models will be able to learn arbitrarily complex functions, limited only by the time it takes to train and the amount of compute resources needed.

As seen previously, many state-of-the-art results in machine musicianship have been achieved using deep learning. In the following sections, we will study how deep learning works to achieve these outstanding results.

2.2.1 A Brief Introduction to Neural Networks

Deep learning is a sub-field under the broader umbrella of **machine learning**. The goal of machine learning is to develop a mathematical function to model a set of training data, and the hope is that through extrapolation the model is able to produce the correct output values when evaluated on new input values.

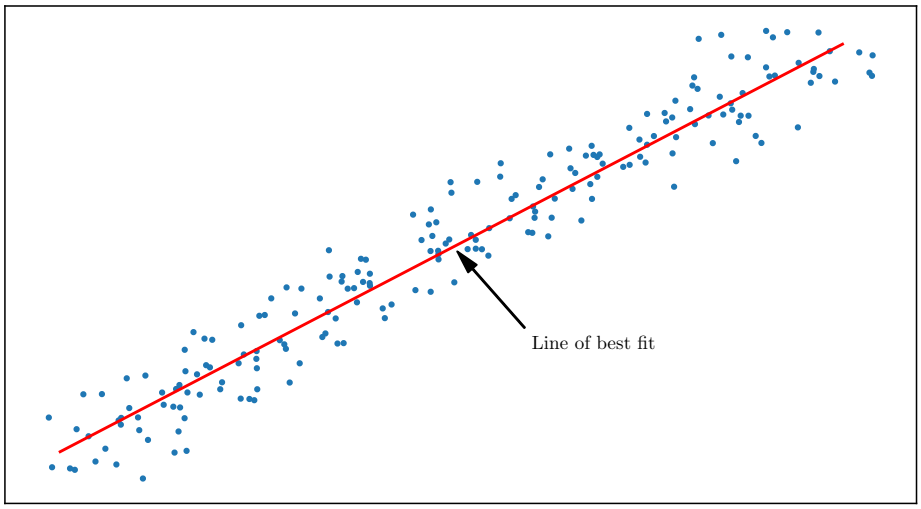


FIGURE 2.1: Linear regression to find a line of best fit in the presence of noisy data.

A hopefully familiar example of this is the linear regression problem (say in 2D), where we desire to find the equation of a line which best describes some sample of data points. If we know with certainty the coordinates of two points (x_1, y_1) and

(x_2, y_2) which perfectly represent the whole dataset without noise effects, we can use the closed-form solution for solving the line equation for example by first calculating the gradient as $m = \frac{y_2 - y_1}{x_2 - x_1}$ and the y-intercept $c = y_1 - mx_1$, then the line equation is simply $y = mx + c$.

The problem here is that in most cases, the dataset will consist of many noisy points (see Figure 2.1), in which case the simple closed-form solution is not enough. There are other mathematical techniques which can approximate the solution for these overdetermined systems, for instance least-squares methods (*Least Squares Fitting*). However, as we go forward seeking far more complex non-linear functions to model noisy real-life datasets, a more general solution would come in handy. This is the promise of machine learning, that we can use a general algorithm and let the algorithm figure out the model which best fits our problem.

Key Ingredients for Machine Learning

At first sight, it sounds like with machine learning, we can feed the dataset we wish to model into the algorithm, and it will figure out the best model for us for free. Of course, things are never that easy, and it turns out that the bulk of the work in training machine learning models lies in problem formulation.

Every machine learning setup needs to have the following ingredients:

1. **Dataset**: This consists of input data x and output data y (optional, depending on whether we are applying supervised or unsupervised learning).

2. **Model**: The model is the function f_θ which maps the input to the output. Earlier, we said that the whole point of machine learning is that it helps us learn the function of best fit, so we don't need to know the function beforehand. The trick here is that we need to put in a placeholder function at the beginning, which starts out with wrong parameters but will be molded through the training process into a model which achieves our objectives. The parameters θ of the function are known as **weights**, and it is by updating these weights during each training step, that the model is able to learn and improve.

3. **Loss function**: This is a user-defined function $J(\theta)$ which calculates the error, or loss, of the model-produced output for given weights, evaluated against the training dataset. This loss function describes, in an upside-down way, the objective that the algorithm should aim to achieve; ideally the loss will be zero when the model is perfectly tuned. Alternatively, we can also think of a reward function, which would be the loss function inverted, which our algorithm needs to maximize. However, by convention most machine learning algorithms are set up to minimize losses instead of maximize rewards.

The Magic of Gradient Descent

Recap: We now know that there is a model, which starts out with some incorrect parameters, and there is a process called training, which uses the dataset to update the parameters in a way that the model gets better and better on each training step.

So how does training work? How does the algorithm know how to update the model parameters in such a way that the model improves instead of getting worse? The magic lies in a technique called **gradient descent**.

Let us go back to the example of linear regression. We can formulate our loss function $J(\theta)$ for function parameters θ as the sum of squared differences between the predicted output y_{pred} and true output values y_{true}, given an input x and $y_{pred} = f_\theta(x)$:

$$J(\theta) = \sum(y_{true} - y_{pred})^2$$

We can plot this loss function against the weight parameter θ, as in Figure 2.2.

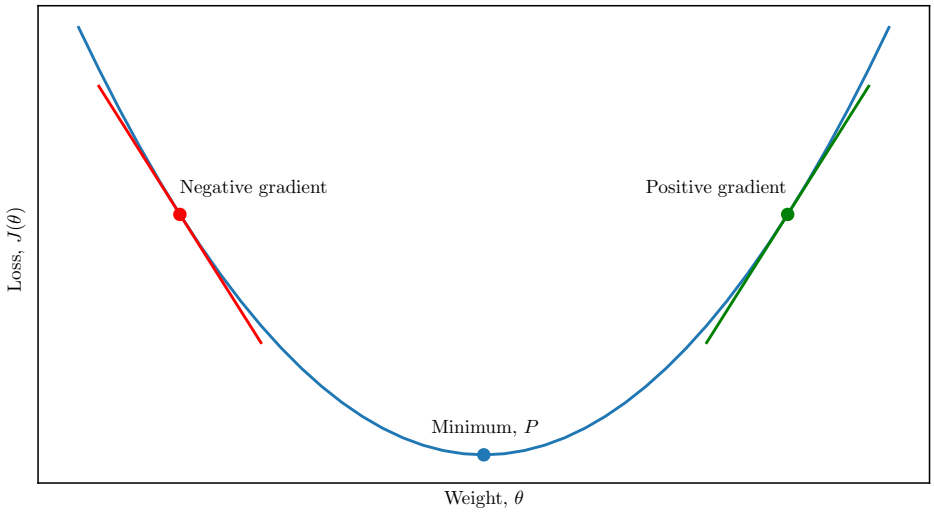


FIGURE 2.2: Plot of loss function against varying weight parameter.

At the start of the algorithm with arbitrarily initialized weights, we might begin anywhere on the curve. Our goal is to end up at the minimum point of the curve at point P, where the loss is minimized. Without prior knowledge of the precise shape of the curve and our position on it, how do we ensure that we can update the weights in such a way that the function moves towards the minimum point?

Consider the gradient of the loss function with respect to a parameter θ (for simplicity, we consider only one weight but this also works using partial derivatives when there are multiple weights):

$$J(\theta) = 2\sum(y_{true} - f_\theta(x))(x)$$

This gradient contains information about the direction of the slope of the loss function. In other words, we can use the gradient of the loss function to tell us if our current position on the loss curve is sloping upwards or downwards. If we wish to move our position downwards, then all we need to do is to update our weights in the opposite direction of the gradient evaluated at our current position, with a step size of η (called the **learning rate**):

$$\theta_{new} = \theta_{prev} - \eta \frac{dJ(\theta)}{d\theta}$$

Applying this step for all weights in the function using all points in the dataset, we are able to slowly move the function down the loss curve, eventually minimizing the loss function and achieving our target (if the loss function correctly describes the target). This method of updating weights using the gradient of the loss function is called gradient descent, and it is the key idea behind almost all of supervised machine learning.

It turns out that this method of calculating the gradient of the loss function and updating the weights against the direction of the gradient, always works to move the function in the direction of steepest descent regardless of what the function is.

There are some issues to be careful of, including the problem of getting stuck in a local instead of global minimum, and taking too large steps resulting in overshooting the minimum, but the general principles hold true. This means that so long as we have a model to train, and a loss function which is differentiable, we can always use gradient descent to train our model.

Going Deep: Modeling Complex Non-linear Functions

So far, we have only talked about what machine learning is. Where does deep learning come into the picture?

Thinking back to the ingredients for machine learning we discussed, it is not obvious what kind of placeholder function we should use to model arbitrarily complex non-linear functions. Mathematically, we know that many different kinds of functions exist: linear, quadratic, cubic, exponential, and the list goes on. How do we decide which kind of function we should use, and how can we make sure that the function is high-order enough to effectively model our problem?

This is where deep learning comes in. The ideas behind deep learning started from McCulloch and Pitts (McCulloch and Pitts, 1943) with their idea that we can model complex functions as a network of simpler functions. However, it was not until 1986 that deep learning really started to take off, thanks to Hinton et al's seminal paper describing a learning procedure called **backpropagation** which can be thought of as a way to apply gradient descent to a chain of nested functions (Rumelhart, Hinton, and Williams, 1986).

Today's neural networks are made up of a graph of functions called **nodes**. Each node takes in an arbitrary number of inputs and maps it to an output value, through a series of two operations: a simple linear function, followed by a non-linear activation function.

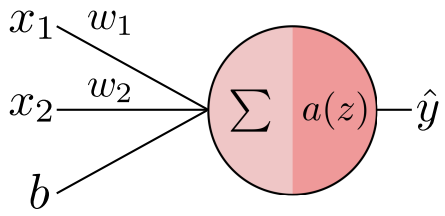


FIGURE 2.3: Diagram of a single neuron. As seen in this diagram, the bias term b can be thought of as an additional input parameter with unit weight.

The **linear function** is simply the equation $z(x) = wx + b$, and is governed by a single weight parameter w in combination with a bias term b for that node. In the case of multiple inputs, each input x_i is treated with its own weight and summed together, but the bias term is applied only once for the whole node. In general:

$$z(x) = \sum w_i x_i + b$$

The non-linear function $a(z)$, also known as the **activation function**, can be thought of as a method to control the output range of the linear function. One example of a common activation function used in neural networks is the sigmoid function $a(z) = \frac{1}{1+e^{-z}}$, which maps its input to an output range between 0 and 1. The final output of a node can be written as:

$$\hat{y}(x) = a(\sum w_i x_i + b)$$

By chaining the output of each node into the input of another node, we can use a network of nodes to perform increasingly complex mathematical functions simply by adding more nodes (see Figure 2.4). A node is also called a neuron, hence these networks of neurons are popularly referred to as **Artificial Neural Networks (ANN)**, due to parallels between this sort of learning network and the neural networks in the human brain (McCulloch and Pitts, 1943).

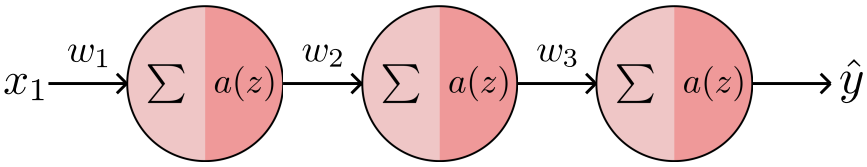


FIGURE 2.4: Diagram of a chain of three neurons in series with a single input. The bias terms have been left out for brevity.

Using a chain of functions instead of just a single function, it becomes more complicated to perform gradient descent and update the weights of each function correctly, but the core algorithm remains the same:

1. **Forward-propagation**: We start at the inputs, and propagate their values forward through the functions to calculate the output values at the end.

2. **Evaluate loss**: We compare the output values to the true output values, and compute the error between the true and prediction outputs.

3. **Back-propagation**: We obtain the gradient for this loss, and update the weights for the functions closest to the output first. We can apply the chain rule (*Chain Rule*) to recursively update the weights for all functions throughout the network, from the output functions back to the input functions. This concludes a single pass of gradient descent, then we can start again from forward-propagation and repeat until the loss reaches an acceptable value.

The design of precisely what neural network architecture to use (how to chain one neuron to the next) is still under heavy research, but different types of architectures have been shown to be useful in different contexts - this is what we will cover in the following sections.

2.2.2 Dense Networks

A common class of neural network architectures are **dense networks**, also known as **fully-connected networks**. In general, instead of just chaining a series of nodes, it is desirable to use nodes in parallel. The nodes in parallel are known as a **layer**.

Now consider a layer where all the nodes in it has connections to every single node in the layer before it. This makes sense because it allows the network full flexibility in making use of whichever connections it needs, and the network itself will discern which connections are less important through the training process. This characteristic of connecting all nodes in neighboring layers makes the network "fully connected" or "densely connected", hence the name.

The following diagram shows an example of a simple three-layer network, which are each densely connected to the next layer. The first and last layers in a network are known as the input and output layers respectively, and any layers in between are called hidden layers.

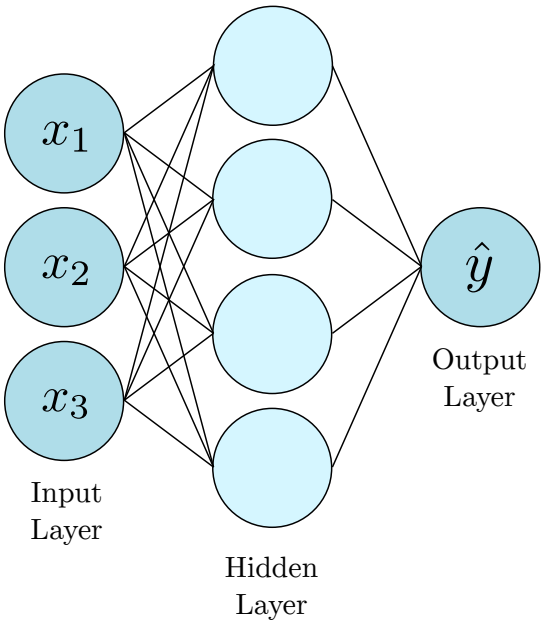


FIGURE 2.5: Diagram of a three-layer dense network. In contrast to previous representations, the input and output variables x_i and y are represented as nodes in the network as well.

2.2.3 Convolutional Networks

One big issue with fully connected networks is that especially when we have a large number of input dimensions, the number of weight parameters and computation required to carry out back-propagation for the weight terms of all nodes becomes computationally infeasible. For example, image classification is a standard problem in computer vision, and we typically have to deal with image dimensions of 512x512 and above. At 512x512, we have a 262,144-dimensional input. Assuming we wish to connect this input to a dense layer with 1000 nodes, we end up with $262144 \times 1000 = 262,144,200$ different weights to train. Clearly, there needs to be a better way. **Convolutional Neural Networks (CNN)** were developed to overcome this problem.

In the simple dense layer, we thought of each layer as a single column of nodes, which provides a convenient and accurate depiction. In the case of dense layers, the ordering and layout of nodes within a layer makes no difference.

But in the case of 2D convolutional networks, we incorporate a spatial dimension into the network, laying out nodes in a grid which lies parallel to the input image. By laying out the nodes in space, we can define a **local receptive field** for each neuron. This way, each neuron is only responsible for the inputs local to its own position, depending on the defined receptive field.

To compute the forward propagation value of each node, the node applies a weight to the value of each of its inputs and sums them up as before. A convenient way to think of this weighted computation of inputs is to look at it as a dot product between two matrices: one matrix containing the subset of inputs within the receptive field, and the other matrix containing the weights for each of those inputs. We call this second matrix containing the weight values a **filter**.

To apply this operation across the whole image space, for convolutional layers the convention is for all neurons to share the same weight parameters on each filter. This design choice was inspired by the observation that in image data (as well as

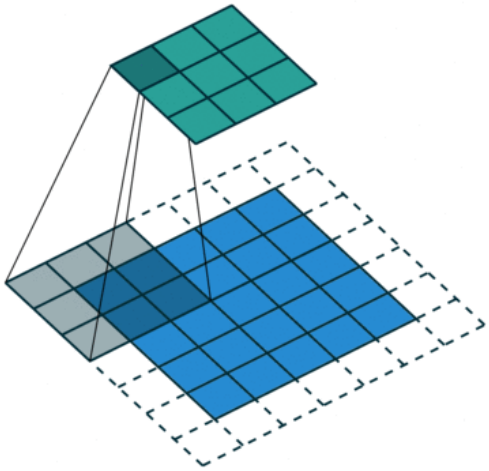


FIGURE 2.6: Diagram of a convolution filter with shape (3,3) taking strides of size (2,2), over an input with (1,1) padding. The top layer is the output of the convolution layer, the bottom layer is the input layer, and the shadow is the convolution filter (receptive field). *Image source: (Dumoulin and Visin, 2016) - see* `https://github.com/vdumoulin/conv_arithmetic` *for intuitive animations of CNNs.*

other data), certain features tend to be repeated throughout the image space, so that useful learned parameters can be shared and applied across the whole image.

An alternative way to think of the shared-parameter filter is that instead of thinking of it as many filters sharing the same parameters, we can think of it as a single filter being shifted across the whole image space. The operations of shifting the filter and calculating its dot product at different positions is similar to the mathematical **convolution** (*Chain Rule*) operation, hence giving its name (though the **cross-correlation** is in fact a closer match (*Chain Rule*)).

Considering that each neuron often has a receptive field size greater than one input, if we use the same number of nodes in the convolutional layer as its input layer, there will likely be significant overlap in between the receptive fields of adjacent nodes. As such, for convolutional layers we often want to decide on a **stride length**, which is the distance (in number of pixels) between the receptive field centers of adjacent neurons. When considering stride lengths greater than one, the resulting convolutional layer will certainly have fewer neurons than input pixels. This is often a desirable property of convolutional layers, and many networks use convolutional layers to reduce the dimensionality of the input data to simplify subsequent computations.

However, parameter-sharing is quite a heavy constraint; if every neuron shares the same filter, it is unlikely that this one filter will be able to capture all of the important details from the whole image. To overcome this problem, we can expand the dimensions of our convolutional layer even further: Instead of having just one flat 2D grid of nodes, we can stack multiple grids, each learning its own filter parameters (but usually the same receptive field size).

2.2.4 Recurrent Networks

Up to this point, we have networks which can look at a snapshot of input data, and produce an output after a single feed-forward pass through the network. However,

in the real world it is often not enough to look at just one snapshot in time to make decisions; intelligent systems benefit from using past information especially when modeling datasets which are sequential in nature.

To achieve this, we consider the idea of a neuron with feedback: Instead of neurons which can only feed information to other neurons, we can connect the neuron's output to itself as one of its own inputs (see Figure 2.7). Using this feedback mechanism, in the next iteration, the neuron can take its own past behavior into account. Networks which make use of these feedback mechanisms are called **Recurrent Neural Networks (RNN)**.

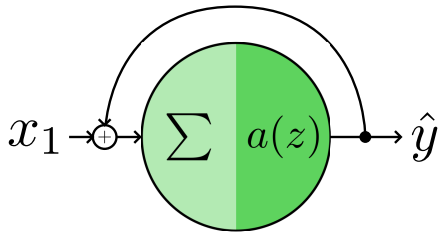


FIGURE 2.7: Diagram of a single recurrent neuron. The output is fed back as one of the node inputs.

RNNs can be configured in many ways, but broadly we can imagine four different use-cases relating to sequence data (Karpathy, 2015):

1. *Single input, sequence output*
2. *Sequence input, single output*
3. *Sequence input, sequence output* in sync; each input produces an output
4. *Sequence input, sequence output* out of sync; full input sequence is processed before the output sequence is produced

Many variations of RNNs have been tested, but one innovation in particular deserves some attention due to its successful use in algorithmic composition: **Long Short-Term Memory (LSTM)** RNNs (Graves, 2013).

The main issue with vanilla RNNs is that on every iteration, the weights get updated based on the most recent inputs and outputs. This means that RNNs suffer from short-term memory loss - the effect of older inputs eventually get overwritten in the weight parameters, and this is a problem when modeling long sequences such as music.

LSTMs are able to overcome this short-term memory problem using a special variation on typical RNN cells. An LSTM cell extends the standard recurrent neuron with built-in memory mechanisms called **cell states**, which are passed from one iteration to the next. Each neuron's output is calculated as a function of the current input and the cell state of the previous iteration, which acts as the cell's memory.

The cell states of an LSTM interact with the neuron's output but they are not the same thing, unlike in standard RNNs where the cell state is just the output of the cell in the previous iteration. Instead, the cell states of the LSTM are updated based on the previous cell state and previous input, and exactly how the cell state update occurs is what makes LSTMs so different from plain RNNs.

The cell state is regulated by gates, which are in themselves neural layers which may or may not alter the cell state at a given iteration, depending on the feed-forward value of that gate (which learns to make alter-the-cell-state-or-not decisions through the training phase, just like any other part of the network) (Olah, 2015).

There exist hundreds of variants of RNNs, depending on the connections and neural architectures even within each RNN cell. But in general, the general working principles of all variants are based on these ideas, and RNNs have been proven to be remarkably effective in many applications - some may even say, unreasonably effective (Karpathy, 2015).

2.2.5 Autoencoders and Latent Spaces

One particularly interesting type of neural network architecture is called the **autoencoder**. The simplest type of autoencoders are simply densely connected autoencoder networks, such as the one in Figure 2.8.

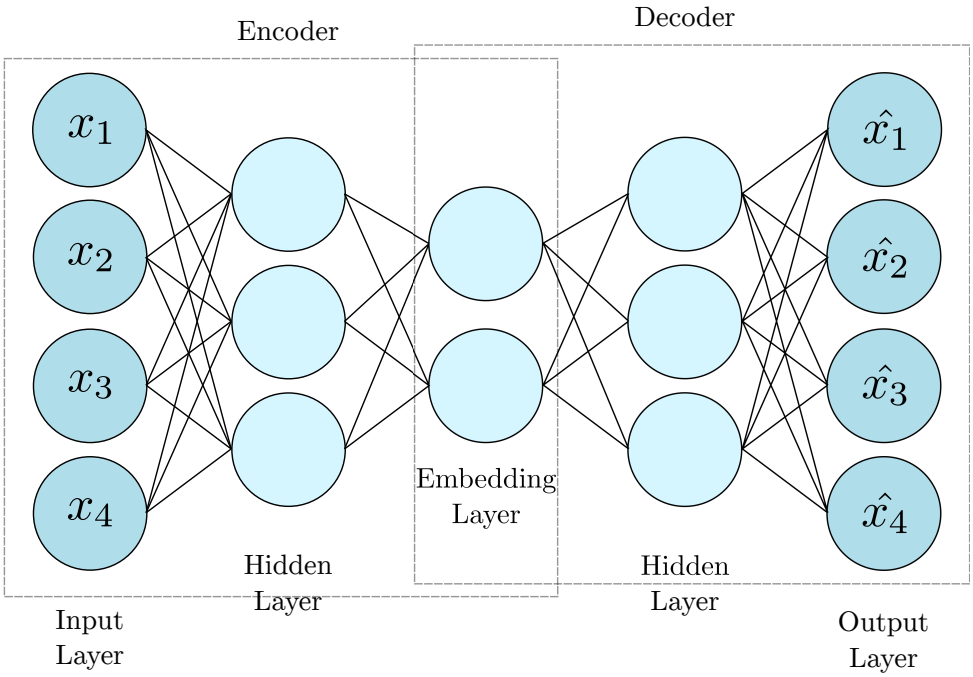


FIGURE 2.8: Diagram of a dense autoencoder network. The network aims to recreate the input at its output after passing the data through a bottleneck embedding layer.

This network works in the same way as the fully-connected networks we have already seen, except for some key constraints which all autoencoders share:

1. The input and output dimensions are identical.
2. The middlemost hidden layer has a dimension much smaller than the input and output layers. This layer is known as the **embedding layer**.
3. The network from the input layer to the embedding layer is (as much as possible) symmetrical to the network from the embedding layer to the output layer. The first half of the network is known as the **encoder**, and the second half is known as the **decoder**.

Apart from the network architecture, the dataset for autoencoders also have a special characteristic: The target output is the same as the input.

This may seem like a strange and redundant network to have. Why do we need a neural network to answer a question which we already have the answer for?

To understand the usefulness of this technique, consider what the network is doing. The network forces the input data through various layers, down to a embedding layer which has much a much smaller dimension that the input. This part of the feed-forward process is called encoding - the encoder portion of the network encodes the high-dimensional data into a lower dimensional space. Next, the network uses its decoder portion to decode the low dimensional data back into the original dimensions, trying to recreate the input as closely as possible.

This is a problem of information compression and dimensionality reduction: In order to recreate its input at the output, the autoencoder has to figure out a way to fit that large amount of data into a considerably smaller bottleneck. To do this, during the training phase the encoder learns to dissect a given input and pick out the essence of its information, and the decoder learns to recreate a likely reconstruction simply based on that internal representation it has in the embedding.

The first practical application of this encoding and decoding process is that, during the encoding process the autoencoder only picks out the important information, and discards all the noise surrounding the data. This means that at the output, the autoencoder can actually produce a version of the input with noise removed (example in Figure 2.9). Autoencoders trained for the purpose of removing noise are called **denoising autoencoders**, and are trained with noisy inputs and evaluated on noiseless outputs.

FIGURE 2.9: Example of denoising autoencoder applied to MNIST handwritten digit images. Top row: Autoencoder input. Bottom row: Autoencoder output. *(Image from The Keras Blog (Chollet, 2016))*

On the decoder side, the autoencoder is also learning something useful: How to recreate a realistic sample from the training distribution simply given the essence of that sample in the form of embedding layer outputs. We can give a name to this "essence" at the embedding layer output, and call it the **embedding vector** or the **latent vector**.

This is where things get interesting for applications in creative AI. We can think of the decoder as an artist, who takes an idea (the latent vector) and turns it into a piece of art (the decoder output). If we had a way to generate good latent vectors, we would be able to generate new pieces of art by passing latent vectors into the decoder. For this reason, the decoder is also sometimes called the **generator**.

For this purpose, researchers have developed a class of autoencoders called **Variational Autoencoders (VAE)**. VAEs are the same as classic autoencoders, except for an additional constraint: The distribution of latent vectors produced by the encoder is encouraged to follow a Gaussian distribution.

The motivation behind this constraint is that if the distribution of real samples map to a Gaussian distribution in latent vector space, we can then have some certainty in saying that if we randomly sample a point from the Gaussian distribution in latent space, we are likely to pick a latent vector which the decoder can use to produce a sample which mimics samples from the input distribution. This gives

powerful generative abilities to the neural network; if for example we train a VAE on a dataset of art paintings, we can simply sample the Gaussian distribution and feed it into the decoder to produce a new painting in the same style!

To enforce the latent distribution constraint on the VAE, we reformulate the loss function to optimize for the **Evidence Lower Bound (ELBO)** (Yang, 2017) objective. Essentially, we incorporate a new term into our loss function, which measures the **Kullback-Leibler divergence** $D(P(a)||P(b))$ (*Kullback-Leibler Distance*) between the distribution of our generated latent vectors and the standard Gaussian distribution. By training the network to minimize this loss, we are able to enforce similarity to the Gaussian distribution (Doersch, 2016).

$$\log P(X|z) - D(Q(z|X)||P(z))$$

However, a good latent space for generation purposes can be tricky to learn, and we wish for our latent space to have the following properties (Roberts et al., 2018a):

- **Expression**: Any real example can be mapped to some point in the latent space and reconstructed from it.
- **Realism**: Any point in this space represents some realistic example, including ones not in the training set.
- **Smoothness**: Examples from nearby points in latent space have similar qualities to one another.

Notably, Google's Project Magenta has had success using latent spaces in art for creating new drawings (Ha and Eck, 2017), audio samples (Engel et al., 2017), and even composing new music (Roberts et al., 2018a).

Given a latent space with those properties, we can do even more than just random sampling to generate new data: we can apply transformations within the latent space, which can be used to alter meaningful properties of the data.

Some examples of latent space transformations include latent interpolation, where we obtain a meaningful "average" of two data inputs by calculating the average of their latent space representations and decoding that average latent vector, and latent arithmetic, where we can add or subtract properties from data by adding or subtracting property vectors in latent space. One great example of this is from (Ha and Eck, 2017), where drawings were added and subtracted in the latent space, and the decoded output produced shows that the latent arithmetic produced clearly interpretable results (shown in Figure 2.10.

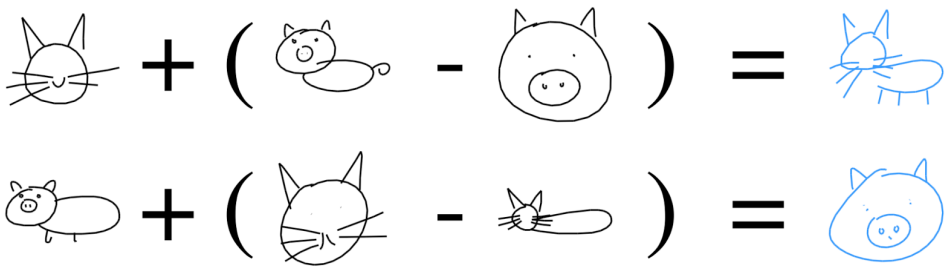


FIGURE 2.10: Example of latent space arithmetic applied to sketches, used to generate new sketches from latent combinations of other sketches. *(Image from Google AI Blog (Ha, 2017))*

2.2.6 Hybrid Networks

Of course, we are not limited to just one type of network architecture. Oftentimes for complicated problems, we wish to take advantage of the strengths of different types of networks. For example, if we are working with video, we know that video is essentially just a sequence of images - so why not join CNNs (good for images) with RNNs (good for sequences)? In the context of this project, we might wish to use latent space transformations to manipulate musical data - in which case, we can use a mix of autoencoders and RNNs (since music is sequential).

Indeed, the proposed approach (outlined in Chapter 4) explores the performance of different types of hybrid networks applied to music collaboration tasks.

2.3 Summary

Historically, the technological pipeline of an AI musician has been crafted by painstakingly engineering each interaction of our robots with music - we see this especially clearly in our analysis of machine listening, where systems for interpreting music at different levels of abstraction are balanced precariously upon each other, with varying levels of success depending on the methods of evaluation.

On its own, AI music composition as a field has a long history and, while it cannot yet be considered mature technology, it has found moderate levels of success in real applications, with companies making use of the technology to release full pop albums, generate movie soundtracks, and more. We find that systems which arbitrarily produce compositions that sound "good" turn out to be achievable, helped by a wealth of music theory knowledge in the field as well as the intrinsically plagiarizable nature of music which machines can exploit by copying phrases and styles from human experts.

However, as in other fields of robot musicianship, systems which produce seemingly human-level results have only succeeded with extensive hand-holding, and are often narrow-minded in their abilities. Machine understanding of musical context and the flexibility required for general musical collaboration turn out to be still far from the reach of current techniques.

Thankfully, the field has not stayed stagnant despite its long history, largely thanks to breakthroughs in machine learning techniques in recent times. We see a trend in neural network-based approaches outperforming previous states-of-the-art, and these techniques appear better-poised to capture the nuances of music with less hand-tuning.

Chapter 3

Requirements Capture

This chapter pertains to the overall requirements of the project. We offer suggestions as to what determines the success of the project, and outline tangible requirements where possible. In cases where satisfaction of a requirement cannot be discretized into a true or false answer, we discuss options for benchmarking to measure how well that requirement is fulfilled.

The project requirements are approached from two different perspectives: Expectations for a musical collaborator, and technical requirements for the system.

3.1 How to Be A Good Band Member

The notion of 'good' music accompaniment is a highly subjective matter, as is the notion of measuring the value of any artistic work. For the purposes of our project, we describe benchmarks for accompaniment quality and methods for measuring if the system has reached that benchmark. These benchmarks are cumulative, such that higher benchmarks require satisfaction of that benchmark's own requirements as well as the previous requirements.

3.1.1 The Bare Minimum: "Don't Step On My Toes"

- The accompaniment system should not obstruct the experience of the musician or listeners, beyond any limitations imposed by the scope of the project.
- The musical output from the accompaniment should (within a certain degree of tolerance) obey the following basic principles of music theory:
  - Stay in key: Stays within the same key as input music (from the human player)
  - Stay on beat: Note onsets should occur at quantized time intervals according to the beat and the most common musical beat-fractions (whole-, half-, quarter-, eighth-, sixteenth-, and triplet-beats)

3.1.2 Added Value: "You're Pretty Good, Kid"

- The musical output should provide accompaniments in the form of fills, basslines, harmonies or melodies in ways that are musically pleasant in the ears of the observing audience. (Needs listener evaluation)

3.1.3 Thought Leadership: "That's A Great Idea, Let's Go With That!"

- The collaboration system should be capable of producing original and interesting ideas relevant to the current musical context

- The collaboration system should be able to make decisions about the progression of the music and lead the transition into next movements

3.2 Technical Requirements and System Design

In addition to requirements on the form of musical output, there are particular requirements that the design should meet in order to achieve its intended use.

3.2.1 Real-time Performance

- The system plays music in real-time
- The system responds to current information in real-time with minimal lag from perception to reaction
- The system should never be late for band practice

3.2.2 System Prerequisites

- The system does not require any custom hardware or high-cost equipment
- The system can be installed on ordinary systems

3.2.3 Interactions and Interface

- Users do not require any technical knowledge to operate the system
- The interaction the between the system and the user should feel natural and intuitive to the user
- The system encourages a synergistic collaboration which encourages new forms of music composition or performance

Chapter 4

Algorithm Design

This chapter describes the development of the algorithm used for the accompaniment generation task.

As discussed previously in Chapter 2, we choose in this project to focus on deep learning models for learning musical accompaniments, though a small sample of alternative algorithms have also been implemented for comparison. Those alternatives are described in Appendix B, and brought forward for comparison against our chosen model at the end of this chapter.

During actual development, exploration of ideas and models happened iteratively, guided at each step by conclusions and hypotheses suggested by each new result. Thus, the development work carried out did not follow the order presented here; this documentation is the cleaned-up version offered to readers with the benefit of hindsight.

All decisions and directions of algorithm development have been developed within the context of this project (unless otherwise cited). At every step of the design process, thoughts and considerations are provided, noting limitations and constraints where applicable.

Finally, the chapter concludes with a summary of all the algorithms which have been explored, and a reasoned decision on which model will be used in the implementation of Comper (detailed in Chapter 5.

4.1 Problem Overview

The objective of our algorithm is to predict a **unit** of musical accompaniments for the next time step $t+1$, given a sequence of units up to current time t.

Careful selection of the size of each time step Δt is important, because too small a Δt will require predictions to be made at a high frequency which may not be achievable for real-time applications. On the other hand, we also do not want Δt to be too large, because Δt also represents the worst-case time-lag in the algorithm's response to new information.

For this project, a Δt of 4-beats is chosen, as it is the standard duration for a measure in $\frac{4}{4}$ musical time. Equivalently, this implies that the duration of a single unit of music we use is 4-beats long. We note that other choices of Δt may be worth exploring, but this is not explored here due to time limitations.

Now, we can more concretely describe the problem we wish to solve, which is to predict the next 4-beat accompaniment unit, given a sequence of player units input_t and accompaniment units comp_t:

$$\ldots, (\text{input}_{t-2}, \text{comp}_{t-2}), (\text{input}_{t-1}, \text{comp}_{t-1}), (\text{input}_t, \text{comp}_t) \xRightarrow{\text{Model}} \text{comp}_{t+1}$$

Before considering any models for making predictions, we first need to concretely define what makes up each unit of music. This is the topic of our next section.

4.2 Datasets

The first question to address in our model design is what data to use. The importance of good training data for deep learning cannot be understated, and obtaining the right dataset is often the most difficult step in practical deep learning.

There are two main considerations in selecting a dataset: The information content, and the data representation.

The **information content** of our training data is dependent on whichever data source we use. We need to be careful that the distribution of our training dataset closely matches the distribution of the test (or deployment) dataset which our system will be deployed on. For example, if we want our system to be used for jazz music accompaniment, we must train the model on jazz music, and not classical or any other genre of music.

The **data representation** is how we present the data to the model. In general, the choice of data representation is malleable even after we have decided on a data source; so long as the information content is the same it is usually possible to convert from one representation to another.

4.2.1 Musical Information

Below, we discuss the information fields which are required for a musical accompaniment system.

Conceptually, we choose to describe music as a sequence of musical notes. A **musical note** is the fundamental building block of music used to make up songs, and any music we aim to create with our system can hence be described fully as a list of musical notes.

A musical note consists of the following properties:

- **Pitch**: This field describes the tone or frequency of a musical note.
- **Velocity**: This field describes the strength at which a note is produced. It is related and often proportional to the volume of the note, but the velocity also captures other information about how the timbre of an instrument may change as it is struck harder or softer.
- **Time**: This field is used to describe both the onset and duration of the note, which can equivalently be called the attack and release times.

There are also further information fields relevant to music such as expression and sound effects, but the three fields discussed here are often the only musical information registered by digital instruments as well, so for the scope of this project this is deemed sufficient.

4.2.2 Representations

As mentioned before, it is also important to carefully choose the representation in which musical information is presented to the network. In theory, any representation which captures the same information can be used to train the model, but data representations can aid or impede model learning in important ways. Furthermore, this representation is coupled tightly with the choice of network architecture itself, since certain architectures work best with certain representations.

Representation Design

Several design choices can be made with regards to data representation:

1. **Continuous / Discrete**: For each of the information fields, we can choose to represent the data as continuous information or discrete categories. For instance, to represent pitch, we can think of two compelling representations which are attractive in their own ways - as a continuous frequency value, or as discrete notes which precludes many of the in-between frequencies which are less commonly used in Western music. This choice also makes a difference in terms of the problem statement: in the case of discrete information we are looking at a classification problem, whereas in the case of continuous information it is a regression problem.

2. **Dimensionality**: The dimensionality of the input data has direct consequences for the size of the network, which can get more difficult to train as the network size grows (since larger networks require more computational time and more data for training).

3. **Completeness**: Depending on the intended scope of the problem, we may sacrifice some relevant information from the dataset to make learning easier. Almost all symbolic representations of music are lossy due to quantization effects and difficulty in representing precise expressions in sound. However, some representations are less complete than others: for example, pianoroll matrix representations are quantized to a fixed temporal resolution of ticks-per-beat, whereas sequences of note-events can encode arbitrary time durations between subsequent events which may capture more nuanced expressions in rhythm.

4. **Conciseness**: When using a dataset crafted for other applications, there are often information fields present which may not be relevant. In such cases, it is desirable to convert the dataset to a representation which ignores these fields entirely to cut out noise effects. Ideally, we can pick a representation which captures exactly all the information which is required for learning, and nothing more. This not only helps to reduce the noise effects of irrelevant information, but could also allow a simpler representation which reduces the dimensionality of the data.

5. **Availability**: In certain cases, certain data representations will be difficult to obtain datasets for, and difficult to convert into from other representations due to missing or incompatible data fields. For longer-term projects it may be possible to record data in whichever representation we want, but due to the volume of data required for successful deep learning it is usually easier to use existing datasets.

These properties are not mutually independent; for example we often increase the dimensionality of the problem when we aim for more completeness; or we lose some completeness if we discretize the dataset for classification instead of regression.

Representation Options

In the following, we consider alternative representations which can be used. These representations do not cover the full spectrum of possibilities, but are presented as useful case studies to illustrate the pros and cons of various representations.

1. **Sequence of note events**

   Perhaps the most natural representation of music is to simply record a list of all note events that occur. To illustrate this, Table 4.1 shows one way to use note events to represent the a song.

offset (s)	pitch (MIDI)	velocity (0-1)	duration (s)
0.0	60	0.7	2.0
1.0	65	0.8	1.0
0.0	72	1.0	1.0
1.0	71	0.5	1.0
⋮	⋮	⋮	⋮

   TABLE 4.1: Example sequence of note events. Each row represents a single event, separated in time from the previous event by the `offset`.

   In the suggested representation, the first parameter `offset` represents the time (in beats) from the previous note event. This is a slightly unintuitive idea, but it is an essential one to allow us to lay out the timing of notes. An alternative to `offset` would be to use absolute time values, where each note event has a `time` parameter which counts the time from the start of the song.

   Also, note that we can alternatively discard the duration parameter by including note-off events (this requires no special handling, it would simply be another event with velocity 0).

2. **Pianoroll matrix**

   The pianoroll matrix is a pitch- and time-quantized representation of music as a 2D matrix. One dimension of the matrix represents time, the other dimension represents pitch, and the elements of the matrix are velocity values.

   The pianoroll matrix lends itself well to visualization, in that the matrix values can directly be mapped to a intensity plot as seen in Figure 4.1.

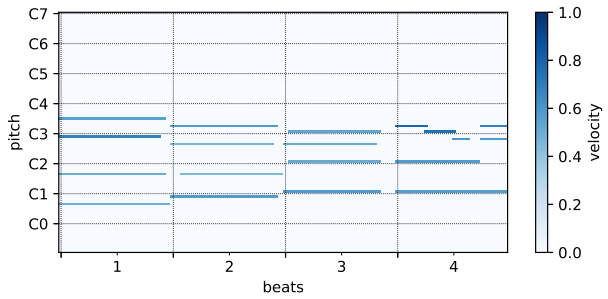


FIGURE 4.1: Intensity plot of a pianoroll matrix for an example 4-beat measure of music. The y-axis denotes 127 discrete pitches separated into 12-pitch octaves, the x-axis denotes time at a resolution of 24-ticks per beat, and intensity denotes velocity.

3. **Onsets matrix**

   A simplified alternative to the pianoroll matrix is the onsets matrix. This is an almost identical representation as the pianoroll matrix, but instead of representing the full note, we capture only the **onset** of each note.

   One advantage of this representation over the full pianoroll matrix is conciseness. In the full pianoroll representation, all intermediate values between the note start and note end hold no informational value but would still need to be learned by the model for syntactical correctness. In the onsets-only representation, all of these values can be discarded.

   One example of an onsets matrix can be seen in Figure 4.2.

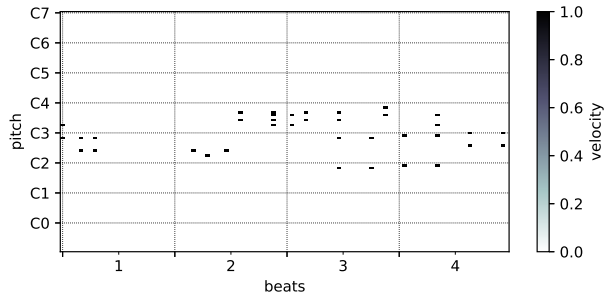


FIGURE 4.2: Intensity plot of a binarized onsets matrix for an example 4-beat measure of music. This is similar to the pianoroll matrix in 4.1, except it only records onsets and the velocity value can only be 1 or 0.

   However, this particular representation is lossy, in that it ignores the release event of each note. To mitigate overwhelming sustain and wetness of playback without note off events, this representation should be limited to playback with instrument voices which by nature are primarily onset-driven and have low sustain, such as the marimba.

   Alternative versions of the matrix which capture all note changes including release events are possible, but are less elegant (note-off events would need to be represented by special tokens in the matrix to differentiate them from no-event values; this would require symbolic interpretation of the matrix instead of the purely numerical representation currently).

4. **Sequence of pitch words**

   The idea of representing music as a sequence of **words** is inspired by work in the field of Natural Language Processing (NLP). In NLP applications such as Machine Translation, the most successful models represent language as a sequence of words from a fixed **vocabulary**.

   One way to achieve this is to start with a quantized time representation. The song is simply divided into equal-length time **ticks**, so that each pitch word only needs to carry pitch and velocity information.

   Then, to tokenize each each tick of musical information, the pitch and velocity information will need to be quantized further so that it can be sorted into a vocabulary of repeatable words.

   This idea is an attractive one especially given its success for NLP applications, but the main problem is that musical words have a far larger typical vocabulary than words in language - and the spread of vocabulary usage can differ

greatly from song to song, whereas in a language such as English, the majority of text and speech can be represented by 10,000 words or so. This is an important difference given that most NLP models which build upon word sequences rely on observing the context of repeated words throughout the corpus. If in the case of music, words are repeated infrequently, such methods will require an immense corpus of training data to work well.

5. **Sequence of monophonic pitch words**

   One other way to attain higher word repetition is to limit our accompaniment scope to monophonic music. By limiting the scope to monophonic voices, there can only be a single active pitch at each time tick. Hence, using binary-quantized velocity with 128 discrete pitches, there can only be 128 possible active words, plus one word to represent the silent case. Hence the vocabulary size will reduce significantly to 129 possible words, which simplifies the problem to a 129-class classification problem.

   However, this is a significantly lossy and limiting representation which will be insufficient for most songs (since most songs are not monophonic).

4.2.3 Data Sources

Deep learning models typically require on the order of 100,000s and upwards of datapoints for training. The number of datapoints required depends on the network architecture and the complexity of the problem it seeks to model, but in general the types of problems that are interesting enough to warrant deep learning will require a large amount of samples to learn from. For comparison, Google Magenta's MusicVAE (Roberts et al., 2018a) and PerformanceRNN (Simon and Oore, 2017) models were trained on 1.5 million MIDI songs scraped from the web, and recordings of 1400 real piano performances respectively.

Some music datasets popular for machine learning tasks include:

1. **The Lakh MIDI Dataset** (Raffel, 2016a)

   176,581 MIDI files scraped from various sources on the internet, consisting of music from different genres (primarily Western music). (Raffel, 2016b)

2. **The Lakh Pianoroll Dataset** (Dong et al., 2017)

   Derived from the Lakh MIDI Dataset (Raffel, 2016a), the Lakh Pianoroll Dataset contains all the same songs post-processed into pianoroll matrices. The Lakh Pianoroll Dataset website (Dong et al., 2018) hosts various versions of their dataset. For example, the `LPD-5-cleansed` version contains 21,425 pianoroll matrices which have been matched to entries in the Million Song Dataset (MSD), have $\frac{4}{4}$ musical time signature, and have multi-track instruments merged into 5 common categories (Drums, Piano, Guitar, Bass, Strings).

3. **JSB Chorales** (Boulanger-Lewandowski, Bengio, and Vincent, 2012)

   A collection of 60 of Johann Sebastian Bach's chorales (classical music).

4. **Yamaha e-Piano Competition Dataset** (*Signature MIDI Collection* 2013)

   A website hosting a collection of high-quality MIDI recordings for piano performances of selected classical pieces of over 60 famous classical musicians.

Of the various data sources considered, the Lakh Pianoroll Dataset was finally selected as the data source for this project due to having a large corpus of music from Western genres (which is the target defined in the project scope) and offering a cleaned dataset, LPD-5, which allows us to extract piano tracks from the files with minimal data cleaning work.

4.2.4 Data Preparation Procedure

Data Preprocessing

Generally, MIDI velocities lie in a range between 0-127, but the pianoroll matrices from LPD-5 have some velocity values up to 500 presumably due to the merging of multiple instrument tracks. Hence, we first clip the velocity values between 0-127, then normalize the values between 0-1 which is a more typical range for the output of neural networks.

To reduce the dimensionality of the data, we "crop" the matrices to user-definable parameters `MIN_PITCH` and `MAX_PITCH`. This reduces the pitch space from 128 to something smaller, since in most songs there will not be notes at the extreme ends of the MIDI pitch spectrum (for comparison, standard pianos only have 88 pitches). Various pitch ranges were tested, finally settling on a range of 96 pitches from MIDI number 13 (C^{-2}) to 108 (C^{7}). This final pitch range was chosen as a compromise between completeness of the representation and suitability for the chosen model architecture (discussed further in 4.4.2).

Data Augmentation

Melody and harmony in music are portable in the sense that so long as relative intervals are maintained, the starting pitch, or the key of the music, can be transposed freely. In that sense, using for instance a 127-pitch system, each song in our dataset can actually be transposed into 127 equally valid songs. However, most songs tend remain around the center of the keyboard, so we are only interested in teaching our model the concept of **key-invariance**, for which we need to represent each song in 12 different keys (the number of semitones in an octave).

Transposition is achieved quite simply in a pianoroll representation. For a key shift of 1 semitone, we simply shift all data in the pianoroll matrix up by one row (along the pitch axis).

We perform data augmentation on the dataset by shifting each pianoroll between -5 and +6 semitones, to create 12[1] copies of each song in different keys (including the original key).

Training and Testing Datasets

An common problem with deep learning is that models may **overfit** the data that it trains on, and not be able to generalize to new data.

To check if the model is overfitting, it is common practice to **validate** the model by evaluating on a separate test dataset which was not used during training.

Throughout the experiments, all datasets are separated with a 90:10 split for training and testing datasets respectively. This split is carried out at the pianoroll (song) level, to make sure that the test dataset and training dataset are clearly separated.

[1] In the data preparation script, an option `NUM_TRANSPOSITIONS` is provided to allow for fewer transpositions to be made, to save memory or computation time.

Final Datasets

To train our deep learning models, the dataset needs to be prepared in exactly the right **shape** to match the input layer of the model we wish to train.

In the following sections of this chapter, we describe various different model architectures which require training data in different input shapes and feeding configurations. As such, we also need to prepare different datasets for each different model architecture.

The main datasets produced are the following, all derived from the LPD-5 dataset (described in 4.2.3):

1. **Pianoroll Sequence Dataset**

   The first dataset caters to sequential models which take a sequence of w units as inputs and produce a single unit as output:

   $$(\text{input}_{i-w}, \text{comp}_{i-w}), \ldots, (\text{input}_{i-1}, \text{comp}_{i-1}), (\text{input}_i, \text{comp}_i) \xRightarrow{\text{Model}} \text{comp}_{i+1}$$

   The API we use for training sequence data (more information in Chapter 5) does not require datasets which explicitly contain (input, output) pairs of datapoints. Instead, we can provide two aligned sequences of input and comp units, and the training process can extract the (input, output) pairs from the sequences on-the-fly.

   To obtain the two aligned sequences, we split each pianoroll from our dataset into the two complementary pianorolls, `input` and `comp`, which serve as accompaniments for each other. This is done in a naive way, using the `PARTITION_NOTE`[2] as the splitting point to divide the pianoroll along the pitch axis into left- and right-accompaniments (as in left and right hand of a piano player). This approach was inspired by (Bretan, 2017).

   After obtaining the left- and right-accompaniments for each pianoroll, we concatenate all accompaniment pianorolls into two long sequences, taking care to align the sequences and pad the end of each song with empty units so that the end of one song does not lead to a prediction of the next song.

   Next, both long sequences are duplicated and concatenated with accompaniments swapped. This is to ensure that the model learns to compose both left and right accompaniments.

   Finally, the entire long sequences are chopped into 4-beat long pianoroll units, which are 96 ticks long given a beat resolution of 24 ticks per beat.

2. **Pianoroll Individual Units Dataset**

   This dataset is used for training autoencoders (discussed in 4.5) which, given an input unit, aim to predict the identical unit at its output.

   The dataset is produced in a similar way as the Pianoroll Sequence Dataset is produced, except that there is no separation into pairs of accompaniments - the full pianoroll is used.

[2] For this project, the `PARTITION_NOTE` was chosen to be constant at MIDI pitch number 60 (middle C), though a more sophisticated approach to split the pianorolls adaptively could be possible, for example by using the mean pitch of a given song.

An additional 'noisy' version the dataset is produced for training denoising autoencoders. The noise added to this dataset is that for every unit, a random 30% of the pitches are dropped (set to zero).

3. **Onsets Sequence Dataset**

   This dataset is nearly identical to the Pianoroll Sequence Dataset, except that it uses an binarized onsets matrix representation. For the binarization, all note onsets with velocities greater than 0.1 are set to 1.0, and all other values are set to 0.

   All other characteristics of the onsets matrix are as described in the previous section 4.2.2.

4. **Onsets Individual Units Dataset**

   This dataset is nearly identical to the Pianoroll Individual Units Dataset, except that it uses a binarized onsets matrix representation, just like the Onsets Sequence Dataset.

   The resulting matrices for this dataset are:

5. **Bassline Sequence Dataset**

   This dataset address a different approach than the others by predicting a binarized monophonic bassline instead of a full accompaniment. The advantage of this approach is that we can treat each unit as 96-tick sequence of `NUM_PITCHES`-dimensional classification problems, which might be easier to solve than 96x96-dimensional regression.

   The monophonic bassline we aim to predict here is simply the lowest activated pitch of each tick instance of the pianoroll, binarized to either 0 or 1. All other elements in the pianoroll are set to 0.

   The objective of this dataset is to do softmax classification to generate a sequence of output ticks. When performing classification, it is easiest to represent the output data as a **one-hot encoding (OHE)**. The OHE is a vector representation of a class label, which each element corresponds to one class label. All elements of the vector are zeros except at the position corresponding to its label, which is 1.

   Conveniently, each column of the binarized monophonic pianoroll is already an almost-one-hot encoding, except for the fact that we are missing the one-hot-encodings for two special tokens: The start of sequence token, and the empty token.

   Sacrificing two rows of pitch information (inconsequential since pitch size is chosen to be arbitrarily larger most music anyway) and the first column of tick information (also inconsequential since a single tick has smaller temporal resolution than can be discerned by humans), we can use

   - Pitch 0: Empty token
   - Pitch -1 (the highest pitch): Start of sequence token

   Empty columns can then simply be replaced with the `[1, 0, 0, 0, ...]` OHE vector, and the first column of each unit replaced with `[..., 0, 0, 0, 1]` for the start token. Care is taken during output playback to ignore these special tokens, replacing them simply with empty vectors.

4.3 Metrics and Custom Losses

In this section, we formulate the model's objective(s). The objective is defined in terms of the **loss function**, which the model will learn to minimize.

For prediction tasks in machine learning, a **loss function** $J(\theta)$ describes the error of a predicted output $\hat{y}$ given the **ground truth** y in the training data, for a model with parameters θ. For our chosen representations, the output and ground truth are matrices, hence the loss functions are defined across matrices with M elements.

The typical formulation of a loss function measures some numerical distance between the prediction and the ground truth, which is an appropriate loss formulation when the training data is the absolute correct answer; however, in the case of music composition there is never just one correct answer. There can always be multiple correct answers, so in this section we also explore other metrics which can be used as alternatives or complements to the numerical distance formulations.

Furthermore, the ideas developed in this section also provide useful **metrics** which can be used to compare and evaluate different models in the next sections.

4.3.1 Target Metrics

The target metrics we consider can be categorized based on four main objectives: **distance to ground truth** (how well the output matches the training data), **musical properties** (how well the output complies with rules of music theory), **syntactical properties** (how well the output obeys syntactical rules), and **latent variable properties** (how well the model learns latent variables with desired properties).

All of the metrics defined below are formulated as losses, meaning that the smaller the value of the metric, the smaller the error. This is chosen by design so that the model learns to minimize the metrics.

Distance to Ground Truth

For distance-to-ground truth metrics, we define distances between each predicted value in the output. For example, when the output is a pianoroll matrix, the loss is calculated for each element in the matrix and the overall loss of the model is taken as a combination (either sum or average) of those individual losses.

Each of the metrics in this section are fairly common mathematical operations, so instead of reinventing the wheel, we use the equivalent loss functions provided by the Keras deep learning library during implementation (See Chapter 5).

1. **Mean Squared Error**

   Mean squared error simply takes the difference between values and squares that difference so that only the absolute difference is considered (taking the square is preferred over the absolute value is preferred since squaring is a *differentiable* operation - see 4.3.2).

   $$J(\theta) = \frac{1}{M}\sum_{i=1}^{M}(\hat{y}_i - y_i)^2$$

   While this particular formulation includes the normalizing factor $\frac{1}{M}$, whether or not we choose to include the factor does not actually change the training objective since the number of values M is a constant with regards to the model parameters. However, the factor does affect the scale of the loss value which

is important to consider when combining multiple loss objectives as in section 4.3.3.

2. **Binary Crossentropy**

   Binary cross-entropy measures the performance of a classification model whose output is a probability value between 0 and 1. Given an output which predicts the class to be 1 with a confidence of p, the logic can be illustrated as a simple function:

```
binary_cross_entropy(y_true, p):
    if y_true == 1:
        return -log(p)
    else:
        return -log(1 - p)
```

   Equivalently, we can use the formula (applied across all M predictions):

$$J(\theta) = -\sum_{i=1}^{M}(y_i \log p + (1 - y_i) \log(1 - p))$$

   Since the cross-entropy loss is meant for binary classifications, it is not designed for regression problems like predicting a pianoroll matrix (where the values are also between 0 and 1, but we would like to be able to predict intermediate values with high confidence instead of predicting 1 or 0 with low confidence to get intermediate values). That said, we also tested binary crossentropy loss for training pianoroll matrix regression models and it produces surprisingly good results (see 4.4 and 4.5).

3. **Cosine Distance**

   Cosine distance is an inversion of **cosine proximity**, which measures the similarity between two vectors. Geometrically, cosine proximity measures the angle between two vectors, so by inverting that formulation, the cosine proximity gives a measure which evaluates to 0 when the vectors point in the same direction and 1 when the vectors are orthogonal.

   Applied to matrices, the cosine distance is inversely proportional to the elementwise product between two matrices:

$$J(\theta) = 1 - \frac{\sum_{i=1}^{M} y_i \hat{y}_i}{\sqrt{\sum_{i=1}^{M} y_i^2}\sqrt{\sum_{i=1}^{M} \hat{y}_i^2}}$$

Musical Properties

1. **Pitch Class Distance**

   In music one of the most important ideas is the concept of musical **key**. As a rule of thumb, the key of a song dictates which pitches sound good and which pitches sound bad in the context of that song. This is a simplification but it is a good heuristic for musical quality.

   A further fact to note is that key signatures are octave-invariant. For example, in the key of C Major, it is equally valid to play the note C^6 (MIDI 60) as it is to

play C^7 (MIDI 72). Therefore, we want our metric to also be octave-invariant. For that property, we compare **pitch classes** instead of the absolute pitch of notes.

In our dataset, we do not have key information attached with each pianoroll. Therefore, one way to encourage the model to produce accompaniments in the same key as the given input is to encourage it to use the same **pitch classes** which are used by the input.

The following are two methods which measure the difference in pitch classes of two pianorolls:

(a) **Pitch Class Intersection-Over-Union**

We can calculate the difference between pitch classes of two pianorolls A and B by creating a set S of all active pitch classes for each pianoroll. Then, we calculate the distance as:

$$J(\theta) = 1 - \frac{\text{count}(S_A \cap S_B)}{\text{count}(S_A \cup S_B)}$$

(b) **Pitch Class Histogram Distance**

An alternative method is to create a pitch class histogram representation of each pianoroll, then calculate the distance between the two histograms. This method is similar to Bag-of-Words (BoW) methods commonly used in computer vision and other machine learning fields.

To create a pitch class histogram, we simply take the sum of note velocities along each pitch class, and normalize histogram values by dividing by the sum of all velocities.

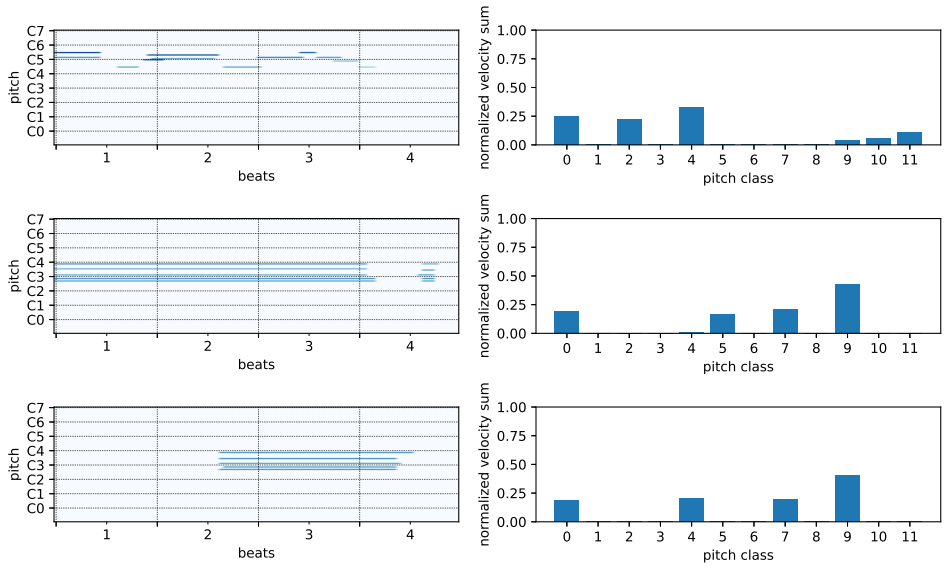


FIGURE 4.3: Three example pianoroll units (*Left*) and their respective pitch histograms (*Right*). The histogram values are calculated by taking the sum of note velocities along each pitch class, then normalizing by dividing by the sum of all velocities.

Then, the distance (either Mean Squared Error or Cosine Distance) between the pitch histograms H for the true and predicted pianorolls can be used as a loss function (shown here using Mean Squared Error):

$$J(\theta) = \frac{1}{M}\sum_{i=1}^{M}(\hat{H}_i - H_i)^2$$

2. **Rhythm Loss**

   In general, we prefer notes that start at particular beat-divisions. Most common would be $\frac{1}{1}$ beats, then $\frac{1}{2}$ beats, then $\frac{1}{4}$ beats, and so on.

   Being slightly off these beats by a tick or two is acceptable (even desirable, for a less robotic musical performance), but in general we want to encourage note onsets close to these common beat-divisions.

   Therefore, we can use a metric that imposes:

   - Reward for onsets occurring within some tolerance σ from $\frac{1}{2}$-beat marks (ie. at 24-tick resolution, occuring at 0, 11, 23...)
   - Impartial to onsets occurring at $\frac{1}{4}$-beat marks
   - Penalizes onsets occurring at all other ticks

   To implement scoring of each onset according to the rules above, we define a score mask, a matrix of similar shape to the pianoroll we wish to evaluate, populated with values corresponding to the score at each position (shown in Figure 4.4).

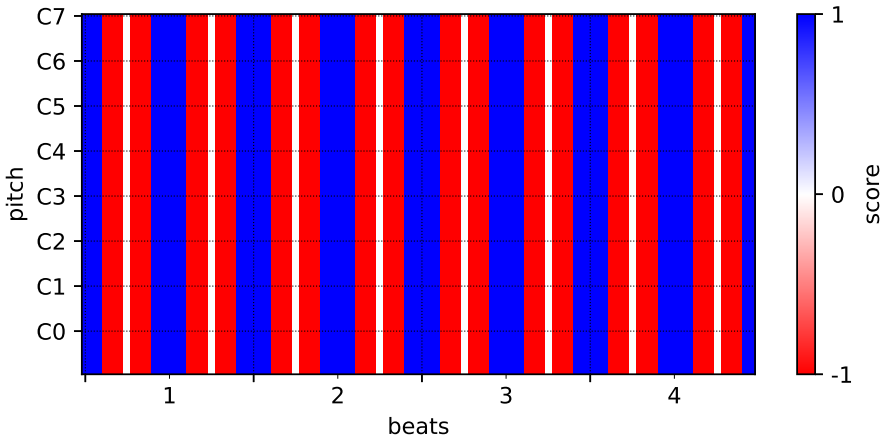


FIGURE 4.4: Intensity plot of an example mask for scoring pianoroll onsets. This mask rewards onsets within $\sigma = 2$ ticks from $\frac{1}{2}$-beat marks (*blue*), is impartial to onsets at $\frac{1}{4}$-beat marks (*white*), and penalizes all other onsets (*red*).

Using this score mask with values s_i, the loss can then be calculated as the dot product between the score mask and the binarized onsets $\hat{o}_i$ of the predicted pianoroll:

$$J(\theta) = \sum_{i=1}^{M} s_i \hat{o}_i$$

Note that this metric is different from the other metrics we have seen so far in that it is a measure of musical quality for a single pianoroll, instead of a measure of distance between two pianorolls.

3. **Onsets Distance**

   This metric is a variation of the previous Rhythm Loss metric, the key difference being that it measures reconstruction quality instead of musical quality. In other words, it measures the distance between two pianorolls instead of the quality of one pianoroll.

   This is done by comparing the **note onsets** between two pianoroll matrices along the time axis only, disregarding which pitch the onset occurs on. This way, we can encourage the model to strike chords or notes in sync with the input (again, ignoring pitch and melodic content).

   For a given matrix, we can define an onsets vector O as the pianoroll onsets matrix summed along the pitch axis. The resultant vector O will have T elements, corresponding to the number of ticks. Then, to calculate the loss between two pianoroll matrices we can take the Mean Squared Distance between their onset vectors O and $\hat{O}$:

$$J(\theta) = \sum_{i=1}^{T} (O_i - \hat{O}_i)^2$$

Syntactical Properties

1. **Pianoroll Smoothness Loss**

   A syntactical requirement of the MIDI playback format is that each note can only have a single onset velocity, which cannot change until after that note has been released. Therefore, the pianoroll matrix representation expects all velocity values for a single note to remain constant until the note has been released, after which the velocity becomes 0.

   This turns out to be a problem when estimating velocities using regression, since the model predicts continuous values between 0-1, and most of the time the resulting predictions are somewhere in-between.

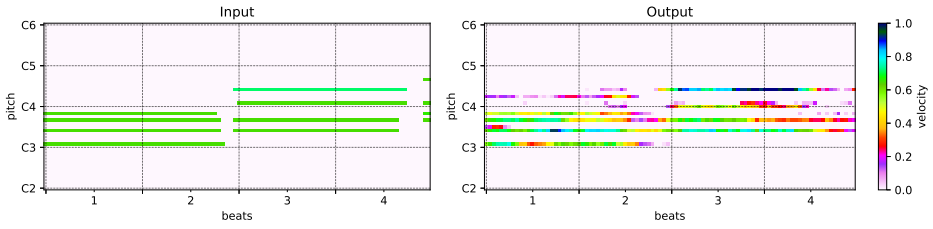


FIGURE 4.5: A true pianoroll unit (*Left*) and its reconstruction through a convolutional autoencoder (*Right*). The velocities within each note have significant variance and unclear onset/release boundaries, which becomes a problem during playback. One way to combat this issue is to design a loss function which encourages **smoothness**.

   This issue can be mitigated by various means of quantization and output post-processing, but fundamentally the continuous values are a difficult issue to solve unless the model learns to produce clear cut-offs between notes and constant values within notes.

   To solve this problem, we develop a loss function which aims to penalize changes along the tick-axis:

```
smoothness_loss(pianoroll_A, pianoroll_B):
    # Pianorolls have shape (pitches, ticks)
    changes_matrix = pianoroll_A[:, 1:end] - pianoroll_A[:, 0:end-1]

    number_of_changes = count(absolute(changes_matrix) > epsilon)
    # (epsilon is the smallest acceptable tremor in velocity)
    return number_of_changes
```

This formulation penalizes all velocity changes equally, which is not exactly what we want but the expectation is that since the number of velocity changes corresponding to note events is very small compared to overall number of changes, the model should minimize all other velocity tremors and allow the note events to pass when considered in tandem with other model objectives.

Latent Variable Properties

1. **Kullback-Leibler Divergence against Gaussian Distribution**

   As discussed in Chapter 2, a desirable property of generative VAE models is that the latent variables of the model closely follow the Gaussian distribution.

   This loss function aims to encourage similarity between the model's latent distribution and the Gaussian distribution, by penalizing the Kullback-Leibler Divergence $D(P(a)||P(b))$ between the two distributions $Q(z|X)$ and $P(z)$.

   $$J(\theta) = \log P(X|z) - D(Q(z|X)||P(z))$$

   Importantly, we note that this loss is different from all the others in that it does not evaluate a property of a single matrix, but of the distribution of latent variables learned by the model.

4.3.2 Differentiable Approximations for Backpropagation

In Chapter 2, we saw that the training process for neural networks involves backpropagation, which uses gradients descent to iteratively tune the weights of the model based on the gradients of the loss function.

Therefore, a necessary condition for loss functions to be used in training neural networks is for the functions to be **differentiable** (*Differentiable*). Unfortunately, not all programming paradigms are differentiable (for example, conditional statements and loop constructs are not differentiable), so to use the previous discussed metrics as loss functions in model training, we need to reformulate some of the metrics.

The reformulation work was done by using a limited set of differentiable functions (mostly matrix operations) defined in the Keras deep learning library. The full code for the custom final differentiable loss functions is provided in Appendix C.

Due to the limited set of operations which can be used, not all of the metrics defined previously can be used as loss functions. However, they are still useful as evaluation metrics, as we will see in Chapter 6.

The metrics which were successfully converted to differentiable loss functions are: Pitch Class Histogram Distance, Onsets Loss, Pianoroll Smoothness Loss, and Kullback-Leibler Divergence against Gaussian Distribution. Additionally, all of the losses provided by the Keras library (including Mean Squared Distance, Binary Crossentropy and Cosine Distance) can also be used as loss functions.

4.3.3 Loss Functions In Action

To demonstrate the effectiveness of the loss functions, we test each loss function with a Convolutional Autoencoder model (see Listing 4.1) which attempts to recreate the input pianoroll matrix at its output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         (None, 96, 96, 1)         0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 8, 96, 50)         650
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 1, 24, 100)        160100
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 1, 6, 200)         80200
_________________________________________________________________
flatten_9 (Flatten)          (None, 1200)              0
_________________________________________________________________
dense_40 (Dense)             (None, 500)               600500
_________________________________________________________________
batch_normalization_41 (Batc (None, 500)               2000
_________________________________________________________________
dense_41 (Dense)             (None, 50)                25050
_________________________________________________________________
dense_42 (Dense)             (None, 500)               25500
_________________________________________________________________
batch_normalization_42 (Batc (None, 500)               2000
_________________________________________________________________
dense_43 (Dense)             (None, 1200)              601200
_________________________________________________________________
reshape_9 (Reshape)          (None, 1, 6, 200)         0
_________________________________________________________________
conv2d_transpose_25 (Conv2DT (None, 1, 24, 100)        80100
_________________________________________________________________
conv2d_transpose_26 (Conv2DT (None, 8, 96, 50)         160050
_________________________________________________________________
conv2d_transpose_27 (Conv2DT (None, 96, 96, 1)         601
=================================================================
Total params: 1,737,951
Trainable params: 1,735,951
Non-trainable params: 2,000
```

LISTING 4.1: Convolutional autoencoder model used when testing loss functions. The model has a latent dimension of 50, and is trained for 50 epochs on about 20,000 units. See 4.5 for architecture design

Individual Loss Functions

To see the effect of each loss function clearly, we train the model on each loss function individually and evaluate the output of the model.

One exception which we leave out of this discussion is the Kullback-Leibler Divergence (KLD) loss objective regarding the distribution of latent variables. The evaluation of the KLD does not make sense unless in the context of a Variational Autoencoder, which requires a tweak to the model architecture. Furthermore, interpreting the impact of the KLD term deserves a much deeper discussion, which we will defer to Section 4.5.

For all other loss functions, the results are given in the following figures; we show that the outputs for all our models actually succeed in enforcing their respective objectives in a clearly interpretable way.

1. **Mean Squared Error** - See Figure 4.6
2. **Binary Crossentropy** - See Figure 4.7
3. **Cosine Distance** - See Figure 4.8
4. **Pitch Class Histogram Distance** - See Figure 4.9
5. **Onsets Distance** - See Figure 4.10
6. **Pianoroll Smoothness Loss** - See Figure 4.11

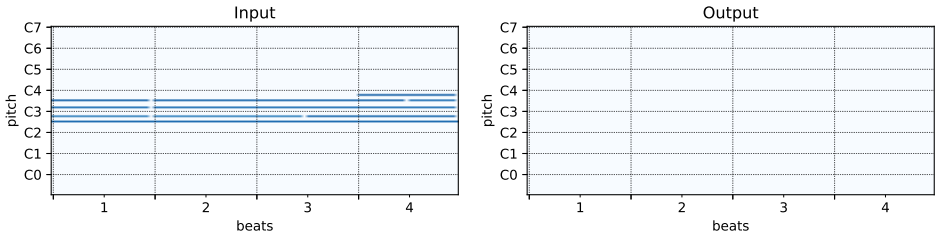


FIGURE 4.6: An input pianoroll unit (*Left*) and its reconstruction from a model trained with Mean Squared Error (MSE) Loss (*Right*). The model learns to predict empty outputs since pianoroll matrices are sparse and the MSE between sparse and empty matrices are low.

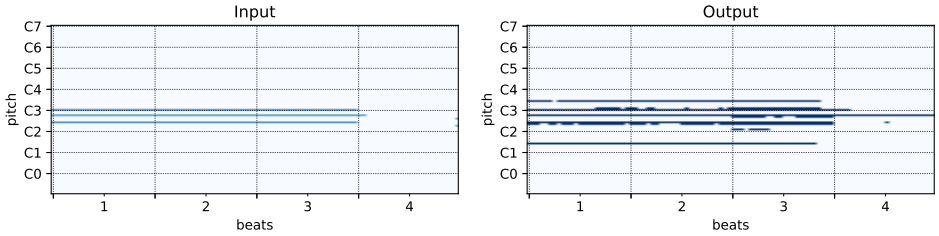


FIGURE 4.7: An input pianoroll unit (*Left*) and its reconstruction from a model trained with Binary Crossentropy (BXE) Loss (*Right*). The BXE-trained model learns to produce matrices which are visually similar, but there are a high number of artifacts in the output which are musically displeasing.

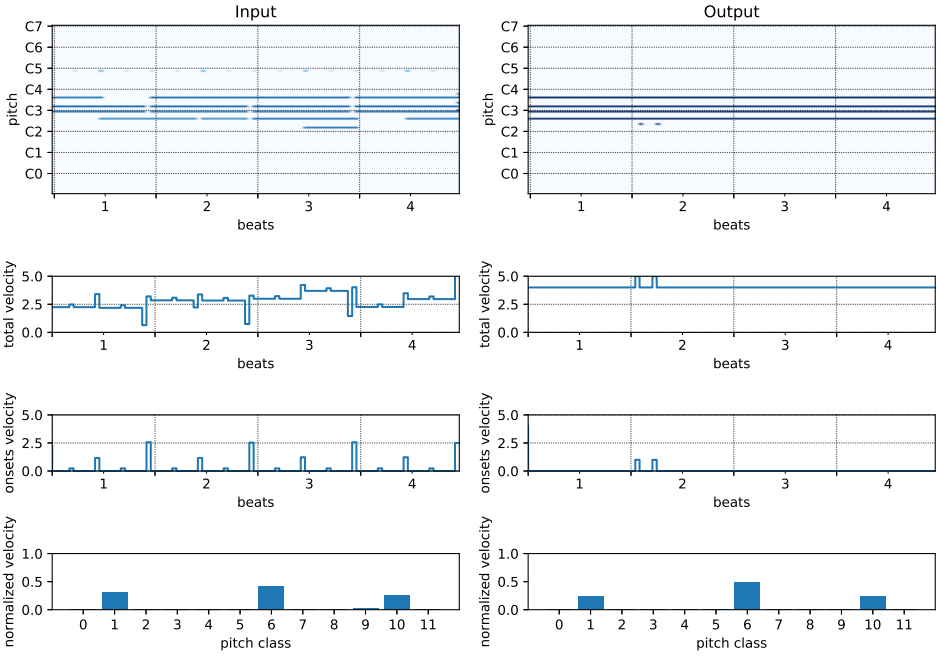


FIGURE 4.8: An input pianoroll unit (*Left*) and its reconstruction from a model trained with Cosine Distance (*Right*). This model does well in producing similar-key outputs, but fails to capture the correct note onsets and releases.

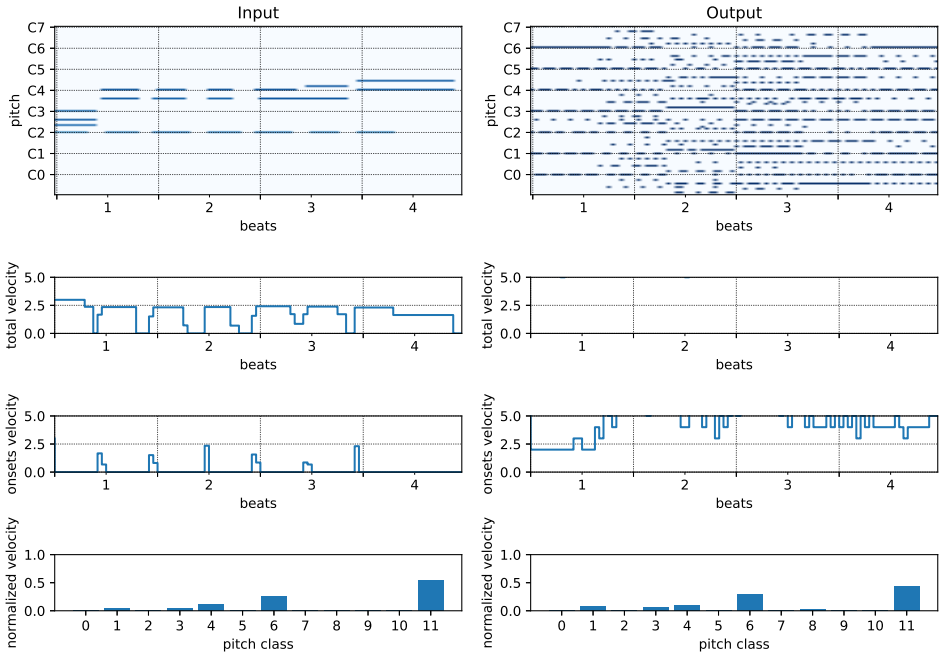


FIGURE 4.9: An input pianoroll unit (*Left*) and its reconstruction from a model trained with Pitch Class Histogram Distance (*Right*). This model fulfills its objective of mimicking the input's pitch class histogram, but ignores all other features of the music.

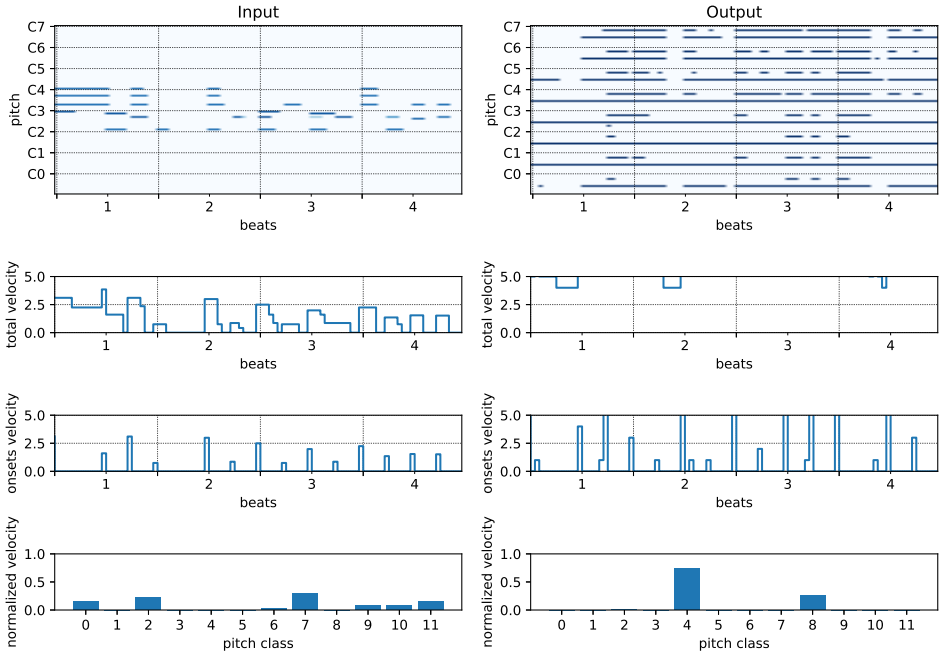


FIGURE 4.10: An input pianoroll unit (*Left*) and its reconstruction from a model trained with Onsets Distance (*Right*). This model fulfills its objective of mimicking the input's onsets in time, but ignores all other features of the music.

Combining Loss Functions

In the previous section, we saw clearly how each of the loss functions affected the model in a clear and interpretable way. However, ultimately the goal of the model

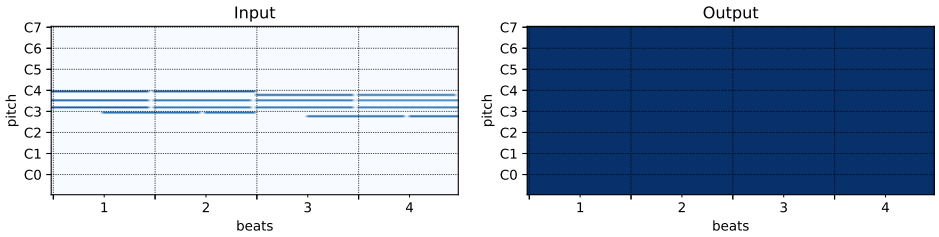


FIGURE 4.11: An input pianoroll unit (*Left*) and its reconstruction from a model trained with Pianoroll Smoothness Loss (*Right*). The constant output demonstrates that the model successfully learns to minimize tremors along the time axis.

is to combine the characteristics targeted by all of those loss functions, not just one characteristic at a time.

To achieve this combination of desired characteristics, we can formulate a loss function which jointly penalizes all the loss functions, shown in Listing 4.2.

```
def combined_loss(input_x, output_x):
    # Distance to ground truth
    mse_loss = coeff_mse * losses.mean_squared_error(K.flatten(input_x), K.flatten(output_x))
    xent_loss = coeff_xent * metrics.binary_crossentropy(K.flatten(input_x), K.flatten(output_x))
    cos_loss = coeff_cos * losses.cosine_proximity(K.flatten(input_x), K.flatten(output_x))
    # Musical properties
    pitch_loss = coeff_pitch * custom_loss.pitch_histogram_distance(input_x, output_x)
    onset_loss = coeff_onset * custom_loss.onset_distance(input_x, output_x)
    # Syntactical properties
    smooth_loss = coeff_smooth * custom_loss.smoothness_loss(output_x)
    # Latent variable distribution loss
    kl_loss = coeff_kl * custom_loss.kl_divergence(z_mean, z_log_var)

    return mse_loss + xent_loss + cos_loss + pitch_loss + onset_loss + smooth_loss + kl_loss
```

LISTING 4.2: Custom loss function using a combination of all loss terms.

The question that remains is how to choose the respective coefficients of the respective loss components, which weighs their relative importances. By changing the coefficients of the loss components, we change the shape of the loss function's hyperplane and gradients, thereby influencing the learning process of the model.

When designing custom loss functions, it is generally desirable to use the simplest function which describes the objective, since complicated loss functions tend to have complicated hypersurfaces which makes it likely that the learning process will plateau after getting stuck in a **saddle point**, never reaching the global minimum (Dauphin et al., 2014) (See Figures 4.12 and 4.17). Thus, it is not a good idea to use every single one of the loss components; for instance, each of the Mean Squared Error, Cross Entropy and Cosine Distance are alternative ways to measure the same reconstruction objective, so we will likely want to choose one of the terms instead of all of them.

The general approach taken in testing the combined functions follows the general outline:

- Choose one of Mean Squared Error, Cross Entropy or Cosine Distance, since at least one of these terms are necessary for reconstructing the input in detail.
- If the latent space properties of the Variational Autoencoder are desired, must include KL-Divergence term. Otherwise, it can be omitted.
- Apply additional loss functions to encourage musical or syntactical properties as required.

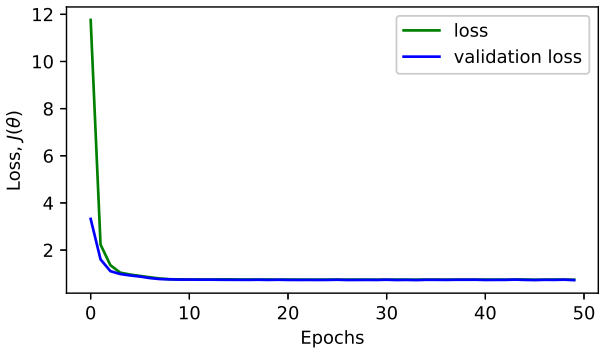


FIGURE 4.12: The training and validation losses for 50 epochs a model training using a combination of all loss terms. We see that the loss plateaus quickly and is unable to improve further after getting stuck in a saddle point.

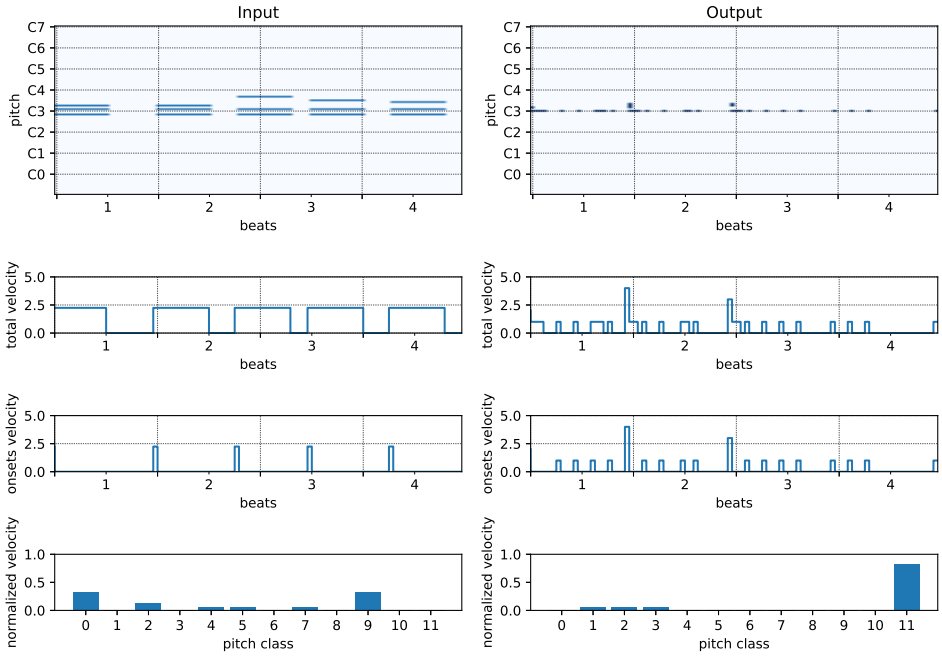


FIGURE 4.13: An input pianoroll unit (*Left*) and the output from the same model from Figure 4.12 (*Right*). This model fails to satisfy any of its objectives due to the complexity of the loss function.

The specific combination of loss components which works best also depends on the model architecture we are training. Thus, the final loss function will be discussed in the context of each model we evaluate in the following sections.

4.4 End-to-end Approach

At first glance, the problem of predicting new musical units suggests a direct **end-to-end model** which takes as input a sequence of past pianoroll units, and predicts a new pianoroll unit. Such a model is considered end-to-end because it attempts to solve the problem with no intermediate steps.

In this section, we look at several neural network architectures for solving the problem end-to-end, and evaluate the results achieved by each model.

4.4.1 Approach Overview

Before diving into specific model architectures, we can take a high-level overview of the general approach, illustrated in Figure 4.14.

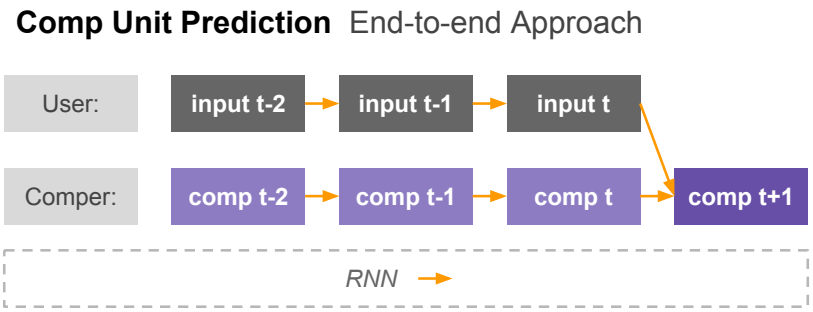


FIGURE 4.14: High-level diagram illustrates our end-to-end approach to predicting compositions. Two streams of music called **input** and **comp** are treated separately based on their source, and handled in segments called **units**. The two streams of units are fed into an Recurrent Neural Network (RNN) which predicts a new comp unit.

We establish the following implementation details that hold true for all of our end-to-end models:

- **Two input streams**: The main difference between accompaniment and composition using sequence models is that for accompaniment, you generally do not want the model to continue all past inputs - otherwise the model will keep "hijacking" the player's melodies. To encourage this separation between the roles of player and accompanier, we provide two input streams to the model: **input** and **comp**, and the model is trained to produce only comp units.
- **Fixed-window inputs**: For simplicity, we choose to use a **fixed window** of 4 units from each of the input and comp sequences as input. This is fixed for all our experiments on predicting sequences of units[3].
- **Inference**: During online prediction, we append each predicted comp unit to the end of the comp input sequence, so that the model makes predictions based on its past predictions as well as the input sequence. Note that at the start of inference, we need to pad empty units into the model to fill up the fixed window.

[3]Other window sizes were also tested (and it is even possible to have variable length windows), but in our experiments, changing these parameters did not appear to have a significant effect on output quality. Furthermore, the choice of 4 units is also a reasonable number of measures for a human musician to be able to start improvising from.

4.4.2 Dense and Convolutional Recurrent Neural Networks

Since we are taking a sequence of units as input, we use Recurrent Neural Networks (RNN) to predict the output. However, since each input unit is high-dimensional (96 pitches $\times$ 96 ticks $=$ 9216 elements), it is a good idea to place a layer(s) between the input and the RNN to reduce the dimensionality of the layers.

Dense Recurrent Neural Networks

We begin with an architecture which uses **fully-connected layers** between the input layer and the RNN layer.

The main disadvantage of using a fully-connected layer between the input and RNN layers is that such a network will need to train a large amount of weights on all its connections. A Keras summary[4] of the model used is shown in Listing 4.3, with 74,935,016 trainable parameters. Such a large number of parameters is almost certainly going to **overfit** unless the problem complexity and dataset are unusually large.

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_5 (InputLayer)         (None, 4, 96, 96, 1) 0
_________________________________________________________________________________________________
input_6 (InputLayer)         (None, 4, 96, 96, 1) 0
_________________________________________________________________________________________________
time_distributed_5 (TimeDistrib (None, 4, 9216)       0           input_5[0][0]
_________________________________________________________________________________________________
time_distributed_6 (TimeDistrib (None, 4, 9216)       0           input_6[0][0]
_________________________________________________________________________________________________
lstm_5 (LSTM)                (None, 800)               32054400    time_distributed_5[0][0]
_________________________________________________________________________________________________
lstm_6 (LSTM)                (None, 800)               32054400    time_distributed_6[0][0]
_________________________________________________________________________________________________
concatenate_3 (Concatenate)  (None, 1600)              0           lstm_5[0][0]
                                                                   lstm_6[0][0]
_________________________________________________________________________________________________
dense_5 (Dense)              (None, 1000)              1601000     concatenate_3[0][0]
_________________________________________________________________________________________________
dropout_3 (Dropout)          (None, 1000)              0           dense_5[0][0]
_________________________________________________________________________________________________
dense_6 (Dense)              (None, 9216)              9225216     dropout_3[0][0]
_________________________________________________________________________________________________
reshape_3 (Reshape)          (None, 96, 96, 1)         0           dense_6[0][0]
=================================================================================================
Total params: 74,935,016
Trainable params: 74,935,016
Non-trainable params: 0
```

LISTING 4.3: Architecture of end-to-end dense RNN. Note the number of parameters: 74,935,016. Such a large number of parameters almost guarantees overfitting.

In our case, the model unsurprisingly overfits to the data - this is seen clearly in Figure 4.15. On testing, the model predicts pitch-perfect accompaniments for the training dataset, but is unable to generalize to make good predictions on the validation dataset.

Convolutional Recurrent Neural Networks

To help with this problem of large input dimensionality, we apply some convolution layers between the input and the RNN layers. At the output as well, the low dimensional output is brought back up to the appropriate matrix dimensions using de-convolution layers (can be thought of as inverse operations to the convolution).

[4] Layer dimensions in Keras follow the convention `[BATCH, (TIME), DATA, ..., (DATA), (CHANNEL)]`, where the dimensions in round brackets are optional depending on the type of layer. For more information about how to read Keras summaries, see 5.1.3

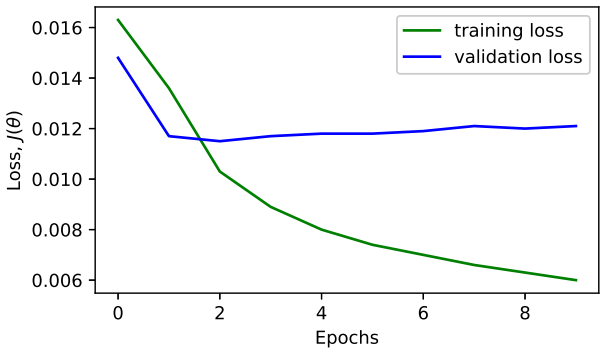


FIGURE 4.15: The training and validation loss when training a dense end-to-end model. The network is **overfitting**, seen in increasingly poor validation performance as the training loss improves.

As seen in Listing 4.4, this architecture produces a much smaller and number of trainable weights despite appearing slightly more complicated.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================
input_1 (InputLayer)         (None, 4, 96, 96, 1) 0
_________________________________________________________________
input_2 (InputLayer)         (None, 4, 96, 96, 1) 0
_________________________________________________________________
time_distributed_1 (TimeDistrib (None, 4, 8, 96, 50) 650       input_1[0][0]
_________________________________________________________________
time_distributed_7 (TimeDistrib (None, 4, 8, 96, 50) 650       input_2[0][0]
_________________________________________________________________
time_distributed_2 (TimeDistrib (None, 4, 1, 24, 100 160100    time_distributed_1[0][0]
_________________________________________________________________
time_distributed_8 (TimeDistrib (None, 4, 1, 24, 100 160100    time_distributed_7[0][0]
_________________________________________________________________
time_distributed_3 (TimeDistrib (None, 4, 1, 6, 200) 80200     time_distributed_2[0][0]
_________________________________________________________________
time_distributed_9 (TimeDistrib (None, 4, 1, 6, 200) 80200     time_distributed_8[0][0]
_________________________________________________________________
time_distributed_4 (TimeDistrib (None, 4, 1200)      0         time_distributed_3[0][0]
_________________________________________________________________
time_distributed_10 (TimeDistri (None, 4, 1200)      0         time_distributed_9[0][0]
_________________________________________________________________
time_distributed_5 (TimeDistrib (None, 4, 1000)      1201000   time_distributed_4[0][0]
_________________________________________________________________
time_distributed_11 (TimeDistri (None, 4, 1000)      1201000   time_distributed_10[0][0]
_________________________________________________________________
time_distributed_6 (TimeDistrib (None, 4, 1000)      4000      time_distributed_5[0][0]
_________________________________________________________________
time_distributed_12 (TimeDistri (None, 4, 1000)      4000      time_distributed_11[0][0]
_________________________________________________________________
lstm_1 (LSTM)                (None, 20)                81680     time_distributed_6[0][0]
_________________________________________________________________
lstm_2 (LSTM)                (None, 20)                81680     time_distributed_12[0][0]
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 40)                0         lstm_1[0][0]
                                                                 lstm_2[0][0]
_________________________________________________________________
dense_3 (Dense)              (None, 1000)              41000     concatenate_1[0][0]
_________________________________________________________________
batch_normalization_3 (BatchNor (None, 1000)           4000      dense_3[0][0]
_________________________________________________________________
dense_4 (Dense)              (None, 1200)              1201200   batch_normalization_3[0][0]
_________________________________________________________________
batch_normalization_4 (BatchNor (None, 1200)           4800      dense_4[0][0]
_________________________________________________________________
reshape_1 (Reshape)          (None, 1, 6, 200)         0         batch_normalization_4[0][0]
_________________________________________________________________
conv2d_transpose_1 (Conv2DTrans (None, 1, 24, 100)     80100     reshape_1[0][0]
_________________________________________________________________
conv2d_transpose_2 (Conv2DTrans (None, 8, 96, 50)      160050    conv2d_transpose_1[0][0]
_________________________________________________________________
conv2d_transpose_3 (Conv2DTrans (None, 96, 96, 1)      601       conv2d_transpose_2[0][0]
=================================================================
Total params: 4,547,011
Trainable params: 4,538,611
Non-trainable params: 8,400
_________________________________________________________________
```

LISTING 4.4: Architecture of a end-to-end convolutional RNN model. The model size is much more manageable than the dense model.

We train the model for 100 epochs on a dataset of 1,000 unique pianorolls. Figure 4.16 shows an example of our prediction.

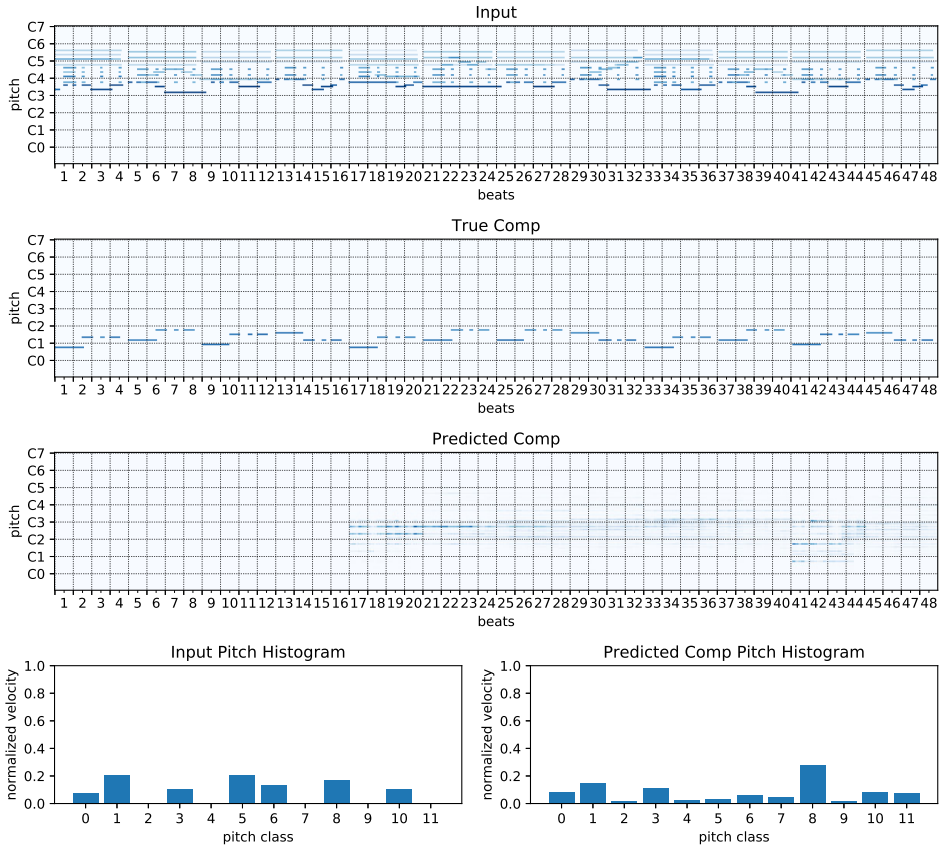


FIGURE 4.16: A sequence of input units, along with the true and predicted accompaniments from the convolutional LSTM. The predicted accompaniments begin after 4 units (4 units $\times$ 4 beat-per-unit = 16 beats) since that is the size of the model's input window. The predicted accompaniments are poorly-formed but the similar input and prediction pitch histograms indicate that accompaniments are being generated in the correct key.

4.4.3 Sequence-to-sequence Models

Seeing the model struggle with predicting accompaniments with good syntactical form, we might consider reframing the problem by classification instead of regression. Indeed, classification techniques have actually had more success in deep learning for music composition (Roberts et al., 2018a) (Simon and Oore, 2017).

We use a sequence-to-sequence model, specifically an **LSTM encoder-decoder** with teacher forcing, inspired by similar techniques in Natural Language Processing for machine translation (Chollet, 2017).

The general approach is as follows:

1. **Encoder**: We use an LSTM to encode the sequence of past inputs, and another LSTM to encode the sequence of accompaniments. Convolutional layers are added between each input and the

2. **Merge**: Having two streams of information from two LSTMs, we merge[5] the output states of both LSTMs, then pass that hidden layer to the decoder.

3. **Decoder**: The decoder consists of a third LSTM which uses the hidden layer from the merge step as the LSTM's initial state. The output LSTM predicts pitch tokens one tick at a time (predicting output[...t+1] given output:[...t]). The final output of the network is a dense layer with the same number of nodes as the number of classes (possible tokens). The output layer has a softmax activation which converts the output values to probabilities for each possible class. On inference from the model, the highest probability class is selected at each time step.

This model is slightly different from previous models which predict whole units at once - the decoder only predicts a single time tick, so during inference we need to make predictions from the decoder `NUM_TICKS = 96` times to produce a full unit.

The full architecture of the model is shown in Listing 4.5.

```
1
2  Layer (type)                 Output Shape              Param #     Connected to
3  ==================================================================================================
4  input_4 (InputLayer)         (None, 4, 96, 96, 1) 0
5  __________________________________________________________________________________________________
6  input_5 (InputLayer)         (None, 4, 96, 96, 1) 0
7  __________________________________________________________________________________________________
8  time_distributed_11 (TimeDistri (None, 4, 8, 96, 50) 650         input_4[0][0]
9  __________________________________________________________________________________________________
10 time_distributed_16 (TimeDistri (None, 4, 8, 96, 50) 650         input_5[0][0]
11 __________________________________________________________________________________________________
12 time_distributed_12 (TimeDistri (None, 4, 1, 24, 200 320200      time_distributed_11[0][0]
13 __________________________________________________________________________________________________
14 time_distributed_17 (TimeDistri (None, 4, 1, 24, 200 320200      time_distributed_16[0][0]
15 __________________________________________________________________________________________________
16 time_distributed_13 (TimeDistri (None, 4, 1, 6, 500) 400500      time_distributed_12[0][0]
17 __________________________________________________________________________________________________
18 time_distributed_18 (TimeDistri (None, 4, 1, 6, 500) 400500      time_distributed_17[0][0]
19 __________________________________________________________________________________________________
20 time_distributed_14 (TimeDistri (None, 4, 3000)      0           time_distributed_13[0][0]
21 __________________________________________________________________________________________________
22 time_distributed_19 (TimeDistri (None, 4, 3000)      0           time_distributed_18[0][0]
23 __________________________________________________________________________________________________
24 time_distributed_15 (TimeDistri (None, 4, 3000)      12000       time_distributed_14[0][0]
25 __________________________________________________________________________________________________
26 time_distributed_20 (TimeDistri (None, 4, 3000)      12000       time_distributed_19[0][0]
27 __________________________________________________________________________________________________
28 lstm_4 (LSTM)                 [(None, 100), (None, 1240400     time_distributed_15[0][0]
29 __________________________________________________________________________________________________
30 lstm_5 (LSTM)                 [(None, 100), (None, 1240400     time_distributed_20[0][0]
31 __________________________________________________________________________________________________
32 input_6 (InputLayer)          (None, None, 96)     0
33 __________________________________________________________________________________________________
34 merge_3 (Merge)               (None, 100)          0           lstm_4[0][1]
35                                                               lstm_5[0][1]
36 __________________________________________________________________________________________________
37 merge_4 (Merge)               (None, 100)          0           lstm_4[0][2]
38                                                               lstm_5[0][2]
39 __________________________________________________________________________________________________
40 lstm_6 (LSTM)                 [(None, None, 100),  78800       input_6[0][0]
41                                                               merge_3[0][0]
42                                                               merge_4[0][0]
43 __________________________________________________________________________________________________
44 dense_2 (Dense)               (None, None, 96)     9696        lstm_6[0][0]
45 ==================================================================================================
46 Total params: 4,035,996
47 Trainable params: 4,023,996
48 Non-trainable params: 12,000
49 __________________________________________________________________________________________________
```

LISTING 4.5: Architecture of an end-to-end RNN encoder-decoder model. The **encoder** portion of the model uses a convolutional LSTM to encode a 4-unit window of input data into a single hidden state. The **decoder** portion uses that hidden state as the initial state of its LSTM, then predicts a new token one tick at a time.

The output from the LSTM encoder-decoder is shown below:

[5]There are several possible ways to merge the output states [h1, c1] and [h2, c2]. Current method simply applies [h1+h2, c1+c2].

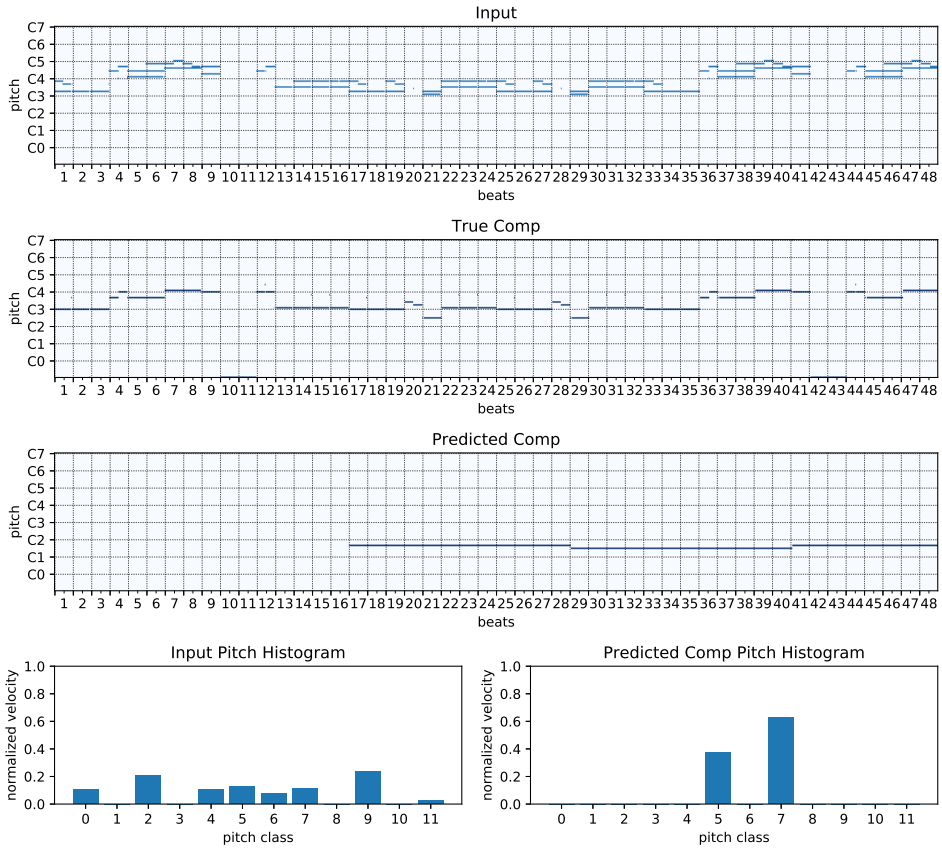


FIGURE 4.17: A sequence of input units, along with the true and predicted accompaniments from the LSTM encoder-decoder. The model produces musically-pleasing (in-key, as seen in the pitch histograms) accompaniments, but tends to make constant predictions throughout each unit.

In general, the bassline predictions from the LSTM encoder-decoder are very different from the ground truth, but sounds musically-pleasing (by being in key, or even in harmony with the input).

However, the model tends to make predictions which hold the same note throughout each unit. One possible explanation for this is due to the tick-by-tick inference method of the model: At a temporal resolution of 24 ticks-per-beat, loosely assuming that the most common note duration is 1 beat long, then the likelihood of the next tick having the same note as the current tick is $\frac{23}{24} \approx 96\%$. Thus, at each step the decoder is likely to repeat the same token.

One possible way to solve the issue of holding a constant note is by reducing the temporal resolution. For example, (Roberts et al., 2018a) uses a resolution of 4 ticks per beat, and have had success in using this technique for composing music. However, at 4 ticks-per-beat we are restricted to 16th-note rhythms, so for this project we seek to find another approach which does not make such a lossy compromise.

4.5 Latent Space Approach

In the previous section, we saw how end-to-end models can be used to predict accompaniments to musical inputs. However, the general conclusion from those experiments is that while end-to-end models can easily learn to fit the training data, generalization to other inputs is poor. When faced with new input from the validation set or in live testing situations, the model makes conservative predictions which are not musically interesting (for example, the convolutional RNN model predicting scattered low-velocity notes or the RNN encoder-decoder predicting sparse, long-duration notes).

Furthermore, because the end-to-end models are by definition direct input-to-output models, we have limited avenues to isolate shortcomings in the system and make focused improvements, for example to focus on input interpretation or output composition. It is also difficult to iterate on versions of end-to-end models, because training end-to-end models requires a long time and large amounts of data.

A potential solution to these problems lies in the **latent space approach**, which breaks the problem into two portions: Learning to interpret and construct musical units, and learning to predict new accompaniments.

4.5.1 Approach Overview

The motivation behind this approach comes from prior work on latent spaces, which demonstrate that it is possible to create musically-meaningful latent spaces which can be used as a "palette" for generating music (Roberts et al., 2018b) (Roberts et al., 2018a) (Simon et al., 2018).

The way this is done is by using a **variational autoencoder (VAE)** model to learn an **encoder** and **decoder** which act as an interpreter and composer respectively, to map between musical units and latent variables.

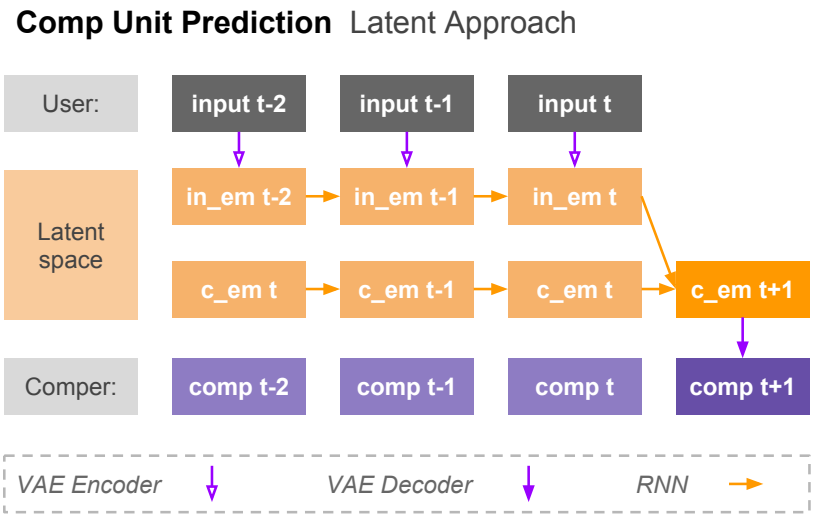


FIGURE 4.18: High-level diagram illustrates our latent approach to predicting accompaniments. As before, we treat input and comp units in separate streams. An intermediate step is added where each unit is first **encoded** into a latent variable (also called an **embedding**) using a Variational Autoencoder (VAE). The two streams of embeddings are fed into an RNN which predicts a new comp embedding. Finally, the embedding is **decoded** back into a musical unit.

Applied to our objective of music accompaniment, we can then conceive a model outlined as follows:

1. **Encoder**: Given a sequence of input and comp units, we can use an encoder to encode the inputs into a sequence of latent variables.
2. **Latent Variable RNN**: The sequence of input and comp latent variables from the previous step can be fed into two RNNs (like in the end-to-end approach), and used to predict the next comp latent variable.
3. **Latent Space Manipulation** (*Optional*): Given a meaningful latent space, we can apply simple transformations to the predicted latent variable to manipulate its musical properties.
4. **Decoder**: Finally, we use the decoder to decode the predicted latent variable into a musical unit for playback.

With this approach, there is the additional option of latent space manipulation, something which was not possible with the previous end-to-end models. This offers an additional degree of control and interactivity to the model, which is an important advantage especially in accompaniment tasks, given the interactive nature of musical collaboration.

The success of this approach hinges of learning a latent space with good **realism** (ability to construct realistic units from any point in the space) and **smoothness** (similar properties between nearby units), otherwise decoding the predicted latent variable will produce poor results.

4.5.2 Baseline: Autoencoder

We begin with a basic autoencoder. As seen previously in 4.4.2, a dense network is likely to be very parameter-heavy, which leads to difficulty in generalization. Thus, we use a convolutional autoencoder. To design our network, we need to consider the following:

- **Size of latent dimension**: Ultimately, the goal of the autoencoder is to learn the encoder and decoder for the embedding layer. The specific dimension of that embedding layer is less important for plain autoencoders, but for variational autoencoders it is important to keep the latent dimension as small as possible to reduce the degrees of freedom of the latent variable to encourage latent properties such as realism (see 2.2.5 for background). Therefore, we design the autoencoder using the smallest latent dimension which allows good reconstruction.
- **Number and type of layers**: In an autoencoder, each layer serves or encode or decode information from one layer to the next. Since the encoder and decoder are symmetric, for design purposes we only need to consider one half, for example the encoder. The goal of the encoder is to reduce the dimensionality of the input to the latent dimension, while retaining as much meaningful information as possible. The number of dimensions in each layer is loosely related to the amount of information that can pass through that layer, hence it is important not to drop dimensions too quickly between layers. For example, to go from a 10,000-dimensional input to a 2-dimensional embedding layer, it is better to go through multiple layers, gradually reducing the size of each layer.

On the other hand, using too many layers can also lead to over-complication of the model and overfitting.

- **Shape of convolutional filters**: Using valid padding and strides without overlap, convolution layers have the effect of reducing subsequent layer dimensions. This is a desirable effect for the encoder portion of our model, so we pick filter shapes according to the size of the consequent layer that we want to have.

Bearing the above considerations in mind (also with inspiration from (Bretan et al., 2017b)), the architecture seen in Listing 4.6 was developed.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 96, 96, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 96, 50)         650
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 24, 100)        160100
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 6, 200)         80200
_________________________________________________________________
flatten_1 (Flatten)          (None, 1200)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               600500
_________________________________________________________________
batch_normalization_1 (Batch (None, 500)               2000
_________________________________________________________________
dense_2 (Dense)              (None, 50)                25050
_________________________________________________________________
dense_3 (Dense)              (None, 500)               25500
_________________________________________________________________
batch_normalization_2 (Batch (None, 500)               2000
_________________________________________________________________
dense_4 (Dense)              (None, 1200)              601200
_________________________________________________________________
reshape_1 (Reshape)          (None, 1, 6, 200)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 1, 24, 100)        80100
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 8, 96, 50)         160050
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 96, 96, 1)         601
=================================================================
Total params: 1,737,951
Trainable params: 1,735,951
Non-trainable params: 2,000
_________________________________________________________________
```

LISTING 4.6: Architecture of plain autoencoder model. The **encoder** portion of the model has three convolutional and a dense hidden layer before the embedding layer with 50 dimensions. The **decoder** has an inverted version of the encoder architecture.

Features worth noting include:

- The **latent dimension** is 50; this is a significant drop in dimensionality from the 9216 input dimensions.
- The encoder uses **three convolutional layers** followed by a **dense hidden layer**, and finally the **dense embedding layer**.
- The **filter size** of the three convolutional layers is guided by musical heuristics. For example, we expect that features across octaves or beats may be reusable through filter parameter sharing, therefore we shape the filters according to octave and beat-division sizes and stride the filters without overlap.
- The model is trained with simple **binary crossentropy** as the loss objective.
- The model is trained for 50 epochs on $\approx 200,000$ pianoroll units.
- Trained using the **Adam optimizer**, with a constant learning rate of 0.002

Finally, we can test the autoencoder: Figure 4.19 shows an example of an input and reconstructed pianoroll matrix. The autoencoder succeeds at encoding and decoding the pianoroll matrix, with significant dimensionality reduction. There are some details lost and artefacts added but the overall reconstruction quality is promising.

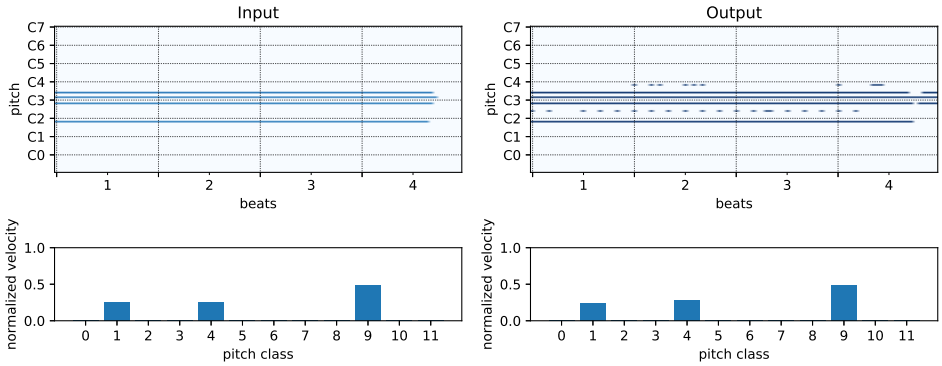


FIGURE 4.19: Input and reconstructed pianoroll units from the plain autoencoder. The model successfully reconstructs the input, with some small distortion.

4.5.3 Denoising Autoencoder

Work from (Bretan et al., 2017b) shows that denoising autoencoders are able to learn more meaningful representations, by forcing the model to pick out the most important features of the input amidst noise.

Therefore, we train the same autoencoder model for the task of denoising input data, where the input data are pianoroll units with 30% of the notes dropped and the model aims to predict the full units (noisy dataset described in 4.2.4). All training parameters are the same as when training the plain autoencoder, but more iteration are required to reach a similar level of quality (this particular example uses 100 epochs).

Figure 4.20 shows the results of the denoising autoencoder. The output retains most of the input notes, and even adds some notes of its own which have good harmonic relationship with the other notes. This is an exciting result - we see that using something as simple as a denoising autoencoder, the model is able to emulate basic understanding of harmony.

4.5.4 Variational Autoencoder

Building off the autoencoder model we had previously, we need to make a small change to the model to make it a variational autoencoder (VAE). Instead of having a single dense hidden layer at the end of the encoder, we branch into two hidden layers which respectively represent the mean and variance of all the latent variables. Then, the embedding layer is sampled from this distribution.

The sampling process for the VAE's embedding layer using Keras deep learning library is shown in Listing 4.7, and the full architecture of the VAE model is shown in Listing 4.8.

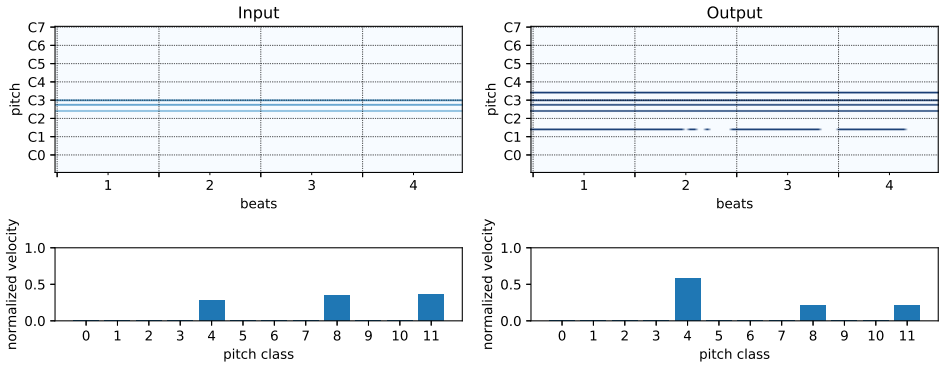


FIGURE 4.20: Input and reconstructed pianoroll units from the denoising autoencoder. The model successfully reconstructs the input, and adds some additional notes of its own. As can be seen in the pitch histograms, none of the additional notes violate the key of the input.

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim), mean=0., stddev=1.0)
    return z_mean + K.exp(z_log_var / 2) * epsilon

z_mean = Dense(latent_dim)(x)
z_log_var = Dense(latent_dim)(x)
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_var])
```

LISTING 4.7: Keras code illustrates the sampling procedure to produce the embedding layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_4 (InputLayer)         (None, 96, 96, 1)         0         
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 8, 96, 50)         650       
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 1, 24, 100)        160100    
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 1, 6, 200)         80200     
_________________________________________________________________
flatten_4 (Flatten)          (None, 1200)              0         
_________________________________________________________________
dense_14 (Dense)             (None, 500)               600500    
_________________________________________________________________
batch_normalization_16 (BatchNo (None, 500)            2000      
_________________________________________________________________
dense_15 (Dense)             (None, 50)                25050     
_________________________________________________________________
dense_16 (Dense)             (None, 50)                25050     
_________________________________________________________________
lambda_2 (Lambda)            (None, 50)                0         
                                                                 
_________________________________________________________________
dense_17 (Dense)             (None, 500)               25500     
_________________________________________________________________
batch_normalization_17 (BatchNo (None, 500)            2000      
_________________________________________________________________
dense_18 (Dense)             (None, 1200)              601200    
_________________________________________________________________
reshape_4 (Reshape)          (None, 1, 6, 200)         0         
_________________________________________________________________
conv2d_transpose_10 (Conv2DTran (None, 1, 24, 100)     80100     
_________________________________________________________________
conv2d_transpose_11 (Conv2DTran (None, 8, 96, 50)      160050    
_________________________________________________________________
conv2d_transpose_12 (Conv2DTran (None, 96, 96, 1)      601       
=================================================================
Total params: 1,763,001
Trainable params: 1,761,001
Non-trainable params: 2,000
_________________________________________________________________
```

LISTING 4.8: Architecture of the VAE model. The embedding layer of the model is sampled using a Lambda layer from two Dense layers representing the mean and variance of the latent variable distribution.

Full Pianoroll Units

Using the variational autoencoder model we defined, we train our model to reconstruct full pianoroll units, noting the following differences from the previous autoencoder model:

- 200,000 units from the Pianoroll Individual Units Dataset (4.2.4), using noisy input to learn denoising
- Loss function: Binary-crossentropy with KL-divergence term

An example of an input pianoroll and its reconstruction through the model is shown in Figure 4.21. We observe that the autoencoder is able to reconstruct the output, but the "blurred pianoroll" problem is present.

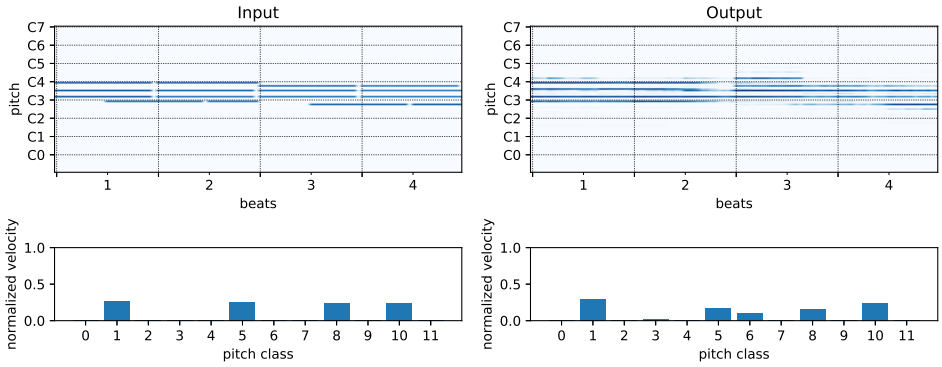


FIGURE 4.21: Input and reconstructed pianoroll units from the denoising variational autoencoder. The output successfully captures the essence of the input and produces notes in the right pitches, but the notes of the pianoroll appear "blurred" and the note boundaries are not well-defined.

The "blurred pianoroll" reconstruction issue turns out to be a big problem in playback - since there are no well-defined note boundaries, note onsets and releases are lost in the current representation. Several different model iterations including trying the Pianoroll Smoothness Loss term (4.3.2) were tested, but none were able to eliminate the problem to an acceptable level.

Nearest-Neighbor Unit Selection

To address the issue of blurred reconstruction, we explore a alternative for reconstruction of latent variables: Selecting a similar unit from a corpus of ground-truth units using nearest-neighbor selection on latent variables. This method is called **unit selection**, and is inspired by similar work from (Bretan, 2017).

No separate training process is required, we simply use the same variational autoencoder.

For nearest-neighbor selection, we follow the following procedure:

1. Prepare a corpus of 200,000 unique units and compute their latent representations using the trained encoder.
2. Given an input pianoroll unit, encode it to obtain its latent representation.

3. Use a fast nearest-neighbor search technique (such as KD-Trees) to find the unit in the corpus with the smallest latent space mean-squared distance from our input unit.

Nearest-neighbor selection within the latent space provides two main benefits: Firstly, within a meaningful latent space, the nearest-neighbor of a given unit is more likely to have similar musical properties than if selection is done at the matrix-level. Secondly, the small latent dimension allows nearest-neighbor computation to be done quickly enough for real-time performance.

An example of an input unit, nearest-neighbor, and decoded unit are shown for comparison in Figure 4.22.

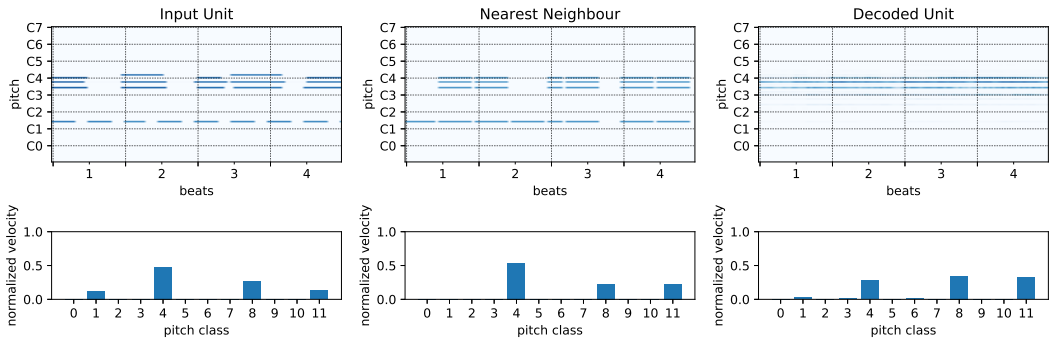


FIGURE 4.22: Input, nearest neighbor, and decoder-reconstructed pianoroll units using the denoising variational autoencoder. The input unit (*Left*) is encoded into a latent variable, which is reconstructed using two alternative methods: Unit selection and decoder-reconstruction. The unit selection output (*Middle*) retains similar musical qualities including pitch and rhythmic style. The decoded output (*Right*) retains pitch properties but suffers from blurring, whereas the unit selection output produces music with perfect realism.

We observe that the unit selection technique is able to produce output units which share similar properties as the input. While this technique does not provide a perfect reconstruction of the input, it guarantees that the output will be well-formed and musically-well defined.

As such, unit selection can be a useful technique and a practical way to get good musical outputs "for free", but it comes with some limitations: The output is limited to whatever units are in the corpus and, the entire corpus needs to be carried with the model for it to work. This can be inconvenient especially when using a large corpus of units, due to the memory uptake.

Onsets Units

Considering that the "blurred pianorolls" problem is related to the difficulty of predicting each note trail entirely from onset to release, we might also consider an alternative representation which represents notes more concisely. Therefore, we also train the model for predicting onset matrices, as an alternative to the full pianoroll representation:

- 200,000 units from the Onsets Individual Units Dataset (4.2.4), using noisy input to learn denoising
- Loss function: Cosine Distance with KL-divergence term

- Trained using the Adam optimizer, with a constant learning rate of 0.002

For onsets prediction, we need to take special care at the output of the model since the output matrix is a probability matrix indicating probability of onsets, not an onsets matrix. To obtain the onsets matrix, we apply a simple threshold binarization to the output probability matrix, setting elements with probability greater than 0.5 to 1, and all other elements to 0.

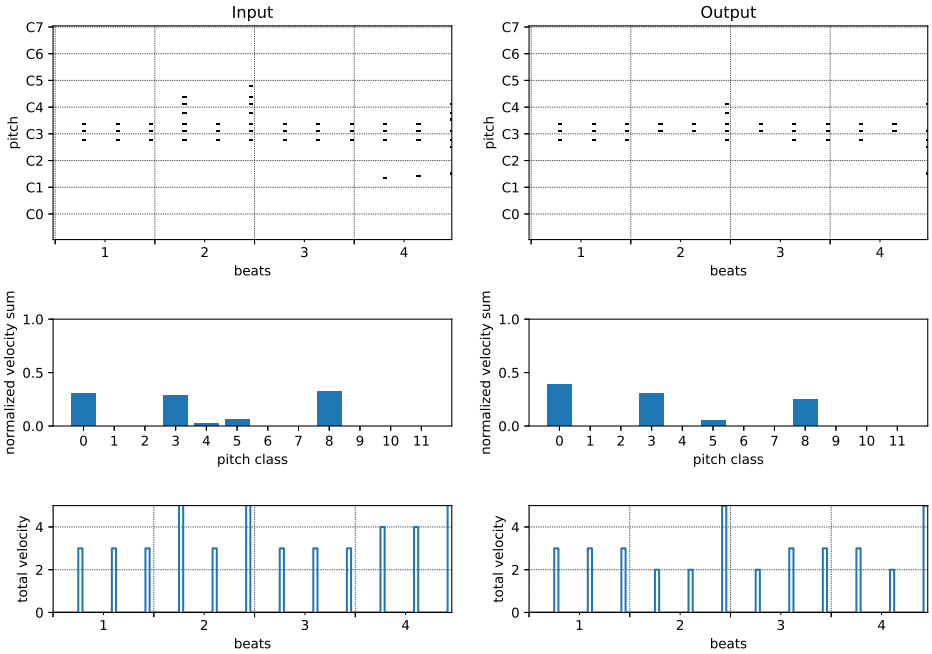


FIGURE 4.23: Input and output onsets units using the denoising variational autoencoder trained on onsets matrices. The output unit (*Right*) is shown after thresholding, to convert the predicted probabilities matrix into binarized onsets. We see that the output closely recreates the input, and there are no issues with blurred notes nor noticeable artefacts in the output.

An example of an input pianoroll and its binarized reconstruction through the model is shown in Figure 4.23. The autoencoder produces a realistic and well-formed reconstruction with no artefacts, with near-perfect similarity to the input.

We conclude that the onsets-only representation works best with our current approach, and use it as the primary representation in the following experiments.

4.5.5 Latent Spaces

At the introduction to the latent space approach as well as in Chapter 2's discussion on latent spaces, we looked at the importance of three properties for latent spaces: **expressionism**, **realism**, and **smoothness**. Additionally, we briefly mentioned the possibility of **latent space manipulation**. In the following, we perform various experiments to test these properties of the latent space.

Visualization

To begin, it might be interesting to visualize the latent variables within the VAE latent space compared to latent variables in a plain AE latent space. For the VAE

latent space, we expect latent variables to be clustered tightly around the origin due to our KL-divergence which enforces a distribution with mean = 0 and standard deviation = 1. For the AE latent space, the latent variables are expected to be more disorganized.

Since it is difficult to visualize a latent space of 50 dimensions, for visualization purposes we train additional VAE and AE models with a 2-dimensional latent space. Of course, visualization of these 2D latent spaces do not necessarily indicate a similar result for the 50-dimensional latent space, but such a visualization is still helpful to get an idea of what our models are learning.

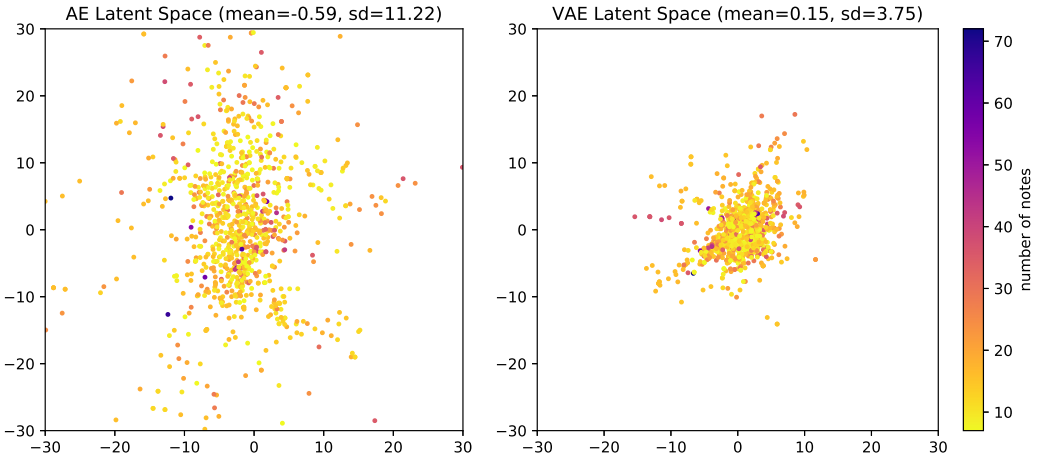


FIGURE 4.24: Visualization of 1000 units from the test dataset, encoded in the 2D latent spaces of an AE and VAE. As expected, the VAE latent space has points scattered similarly to a Gaussian distribution, whereas the AE latent variables are sparsely scattered. For interest, we plot the color of each point according to the note density of that unit but there are no obvious signs of clustering by note density.

Figure 4.24 shows a plot of the latent encodings for 1000 units from our test dataset, trained on AE and VAE models. We see that indeed, the VAE latent space has latent variables distributed like a standard Gaussian, whereas the AE latent space has variables scattered much more widely.

This confirms our expectation about the distribution of latent variables.

Reconstruction

Expressionism relates to the property that any real unit can be encoded to a variable in the latent space, and that latent variable can be decoded back to the same unit. Since this is exactly what the variational autoencoder achieved in reconstructing units from the test dataset, we consider that this property is satisfied.

Random sampling

For **realism**, any random latent variable sampled from the latent space should be able to be decoded into realistic musical units.

To test this property, we simply sample from the normal distribution to create latent variables, and use the decoder portion of our VAE to generate units from each sampled latent variable. Figure 4.25 shows these results.

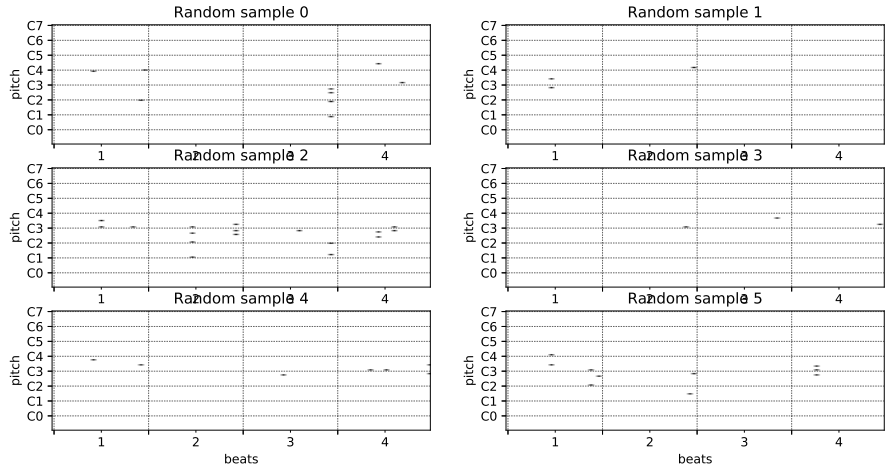


FIGURE 4.25: Units generated by random sampling of variables in the latent space of the variational autoencoder. Visually and musically, the units appear realistic.

We note that testing realism quantitatively requires a more nuanced analysis which falls beyond the scope of this project. Visually, we observe (and can confirm this through audio playback of the units) that the units sampled from the variational autoencoder's latent space possess good musical quality.

For comparison, we also construct randomly sampled latent variables from a plain autoencoder. This produces a majority of empty binarized onsets matrices, due to the low-confidence probability matrices yielded from the latent space with relatively poor realism.

Interpolation

Smoothness measures the similarity in meaningful properties between units whose latent representations are nearby in the latent space. One way we can see this is by interpolation between two variables in the latent space, and observing their decoded units. If the latent space is smooth, we should observe that musical properties of the interpolated units should appear to vary smoothly between the two endpoint units.

By defining an interpolating variable α which is 0 at the beginning of the interpolation and 1 at the end, we can define an interpolated latent variable z_α between two latent variables z_0 and z_1 as:

$$z_\alpha = z_0 * (1 - \alpha) + z_1 * \alpha$$

Figure 4.26 shows the results of a 5-point interpolation including the two endpoints. It is seen that the pitch histograms and onset velocity profiles of all units vary smoothly in accordance with the interpolating variable. This indicates that the latent space is indeed smooth.

Latent arithmetic

The idea behind latent arithmetic is that there are latent vectors within the latent space which represent meaningful properties in the decoded space. Then, we can add and subtract latent property vectors to a given latent variable, and get a new

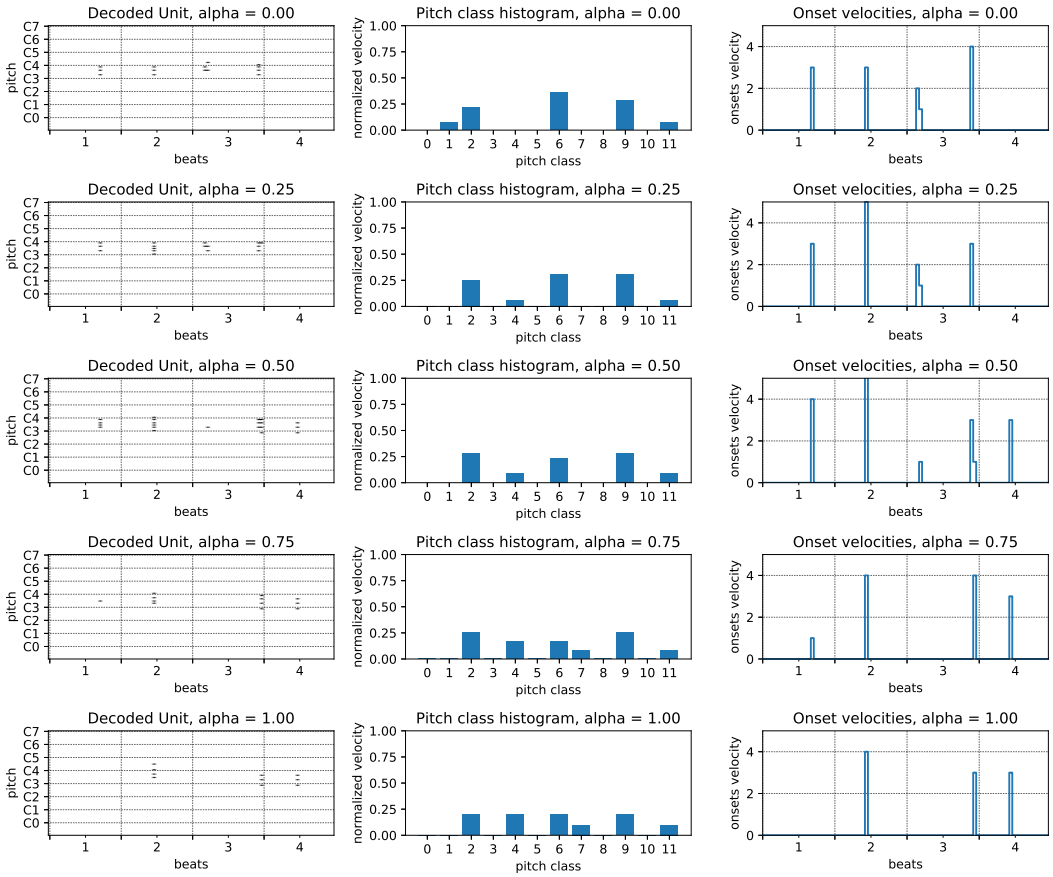


FIGURE 4.26: Units generated from interpolated variables in the latent space. The pitch histograms and onset velocity profiles of the units are seen to vary smoothly with the latent variables.

latent variable which when decoded, is a version of the original sample but with that property changed. For example, some useful musical properties we could possibly manipulate using latent arithmetic include: Average note velocity, note density, pitch position (higher or lower octaves), musical forwardness (melody/background), and more.

In this project, we apply latent arithmetic for **note density**, but in further work this approach can also be applied to other musical properties.

The process used for obtaining a property vector for latent arithmetic is as follows:

- Calculate the average note density of all units in the dataset (note density in the case of binarized onsets matrices is simply the mean of the matrix elements).
- Randomly select 1,000 units with above average note density, and 1,000 units with below average note density.
- Arbitrarily pair up the low density units with high density units, and take the difference between every pair (high - low).
- Then, the property vector for **positive note density** is the mean of all (high - low) pair differences.

To apply the property vector to a given unit, we simply add the latent property vector to the latent representation of that unit, and decode the transformed latent variable.

$$z_{transformed} = z_{original} + z_{property}$$

The results of applying a positive density vector to a unit is shown in Figure 4.27. We see that the latent space transformation successfully changes that property of the unit (in this case, increasing the note density of the unit). Importantly, the transformation does not affect other properties of the unit - for example, the pitch histogram of the unit remains mostly unchanged.

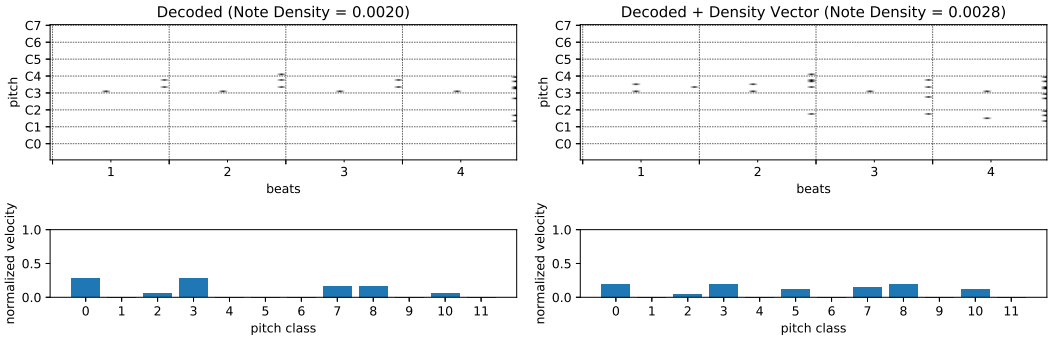


FIGURE 4.27: A decoded latent variable before and after applying a positive density vector. It is seen that the note density of the unit increases from 0.0020 to 0.0028. It is noted that the application of the property vector does not significantly change the pitch histogram or other important properties of the unit.

4.5.6 Latent Variable Recurrent Neural Network

Finally, we can attempt accompaniment predictions within the latent space, using our VAE encoder and decoder to map from real units to latent variables. To predict latent variables, we employ a two-stream RNN network similar to what we did in 4.4, except that our input and output are latent variables instead of pianoroll matrices. The model architecture shown in Listing 4.9.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================
input_7 (InputLayer)         (None, 4, 50)             0
_________________________________________________________________
input_8 (InputLayer)         (None, 4, 50)             0
_________________________________________________________________
lstm_7 (LSTM)                (None, 200)               200800    input_7[0][0]
_________________________________________________________________
lstm_8 (LSTM)                (None, 200)               200800    input_8[0][0]
_________________________________________________________________
concatenate_4 (Concatenate)  (None, 400)               0         lstm_7[0][0]
                                                                 lstm_8[0][0]
_________________________________________________________________
dense_7 (Dense)              (None, 500)               200500    concatenate_4[0][0]
_________________________________________________________________
dense_8 (Dense)              (None, 50)                25050     dense_7[0][0]
=================================================================
Total params: 627,150
Trainable params: 627,150
Non-trainable params: 0
_________________________________________________________________
```

LISTING 4.9: Architecture of the latent variable RNN. The model takes as input two sequences of latent variables to keep the input and comp variables separate, feeding them through two LSTMs before concatenating and passing a dense layer before the final output layer, which predicts a single latent variable.

Unfortunately, upon training it becomes clear that the model is overfitting and not learning to generalize to new data. This is seen in Figure 4.28.

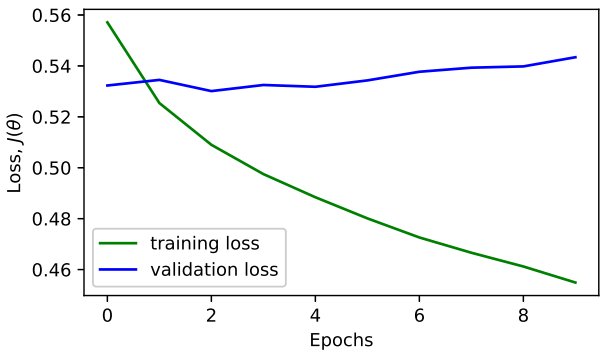


FIGURE 4.28: The training and validation loss when training a fully-connected end-to-end model. The network has too many weights to train which leads to **overfitting**; this is illustrated by increasingly poor validation set performance as the training loss improves.

Figure 4.29 shows the model predicting accompaniment for data previously seen in the training dataset, and Figure 4.30 shows the same model's accompaniments for new data from the validation set. As usual, the predictions start at the 17th beat, after a 4-unit window.

The outputs verify this phenomenon: The accompaniments generated for the training data have good quality and closely resemble the ground truth accompaniment, whereas the model predicts only empty units for new data from the validation set.

A further observation is that from the plot of onset probabilities in Figure 4.30, we see that the model is actually predicting empty units with high confidence (all elements have near-zero probability of being onset) rather than making low-confidence predictions (which is what we had from the end-to-end convolutional RNN in Section 4.4).

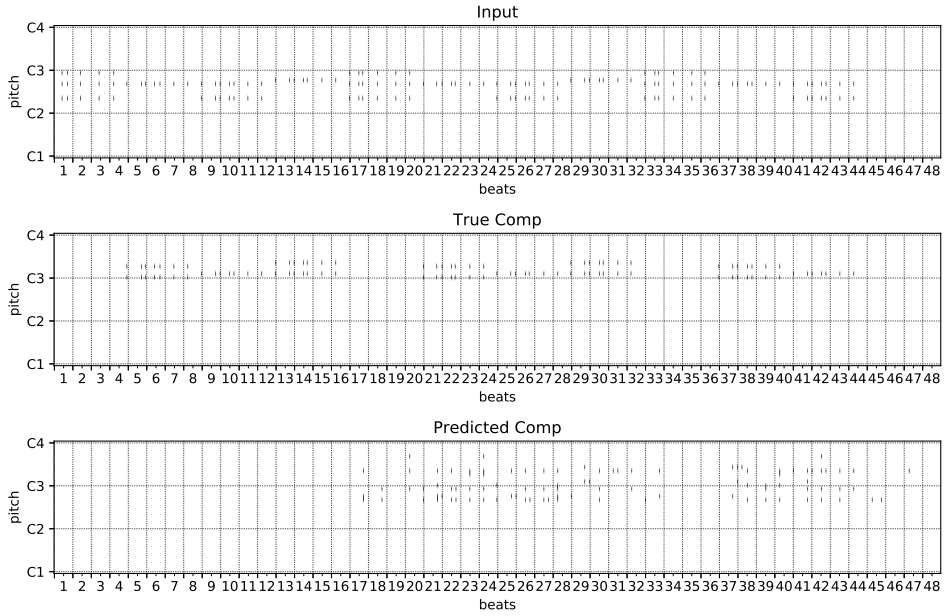


FIGURE 4.29: Input from the training dataset, ground truth accompaniment, and predicted accompaniment using the latent-variable RNN. We see that the model is able to make good accompaniments which closely resemble the ground truth when predicting for the training data.

4.6 Summary

In this chapter, we have explored a large variety of models for predicting musical accompaniments. The main architectures considered are:

1. *(End-to-end approach)* Convolutional RNN
2. *(End-to-end approach)* End-to-end RNN Encoder-Decoder
3. *(Latent space approach)* VAE + Latent Variable RNN

The Convolutional RNN and Latent Variable RNN use a similar approach, but the latter breaks up the architecture into two models for targeted training. In our experiments, both of these approaches suffer from overfitting, so they do not generalize well to new data, resulting in poor output.

Of the three approaches, the RNN Encoder-Decoder was the only model capable of producing a reasonable accompaniment when evaluated on test data. We attribute this success to the fact that the RNN Encoder-Decoder model is a classification model rather than a regression model, which the other models are. However, this approach also comes with the constraint that the predicted accompaniments are limited to monophonic music only.

Additionally, in the development of variational autoencoders we looked at two different methods for generating music from a latent space: Decoder Reconstruction and Unit Selection. Both of these techniques are capable of generating realistic musical units, and furthermore we discover that through the latent space we can apply interesting transformations to musical units.

The ability to generate music from a latent space could lead to interesting applications for music accompaniment. In particular, we can imagine a **call-and-response**

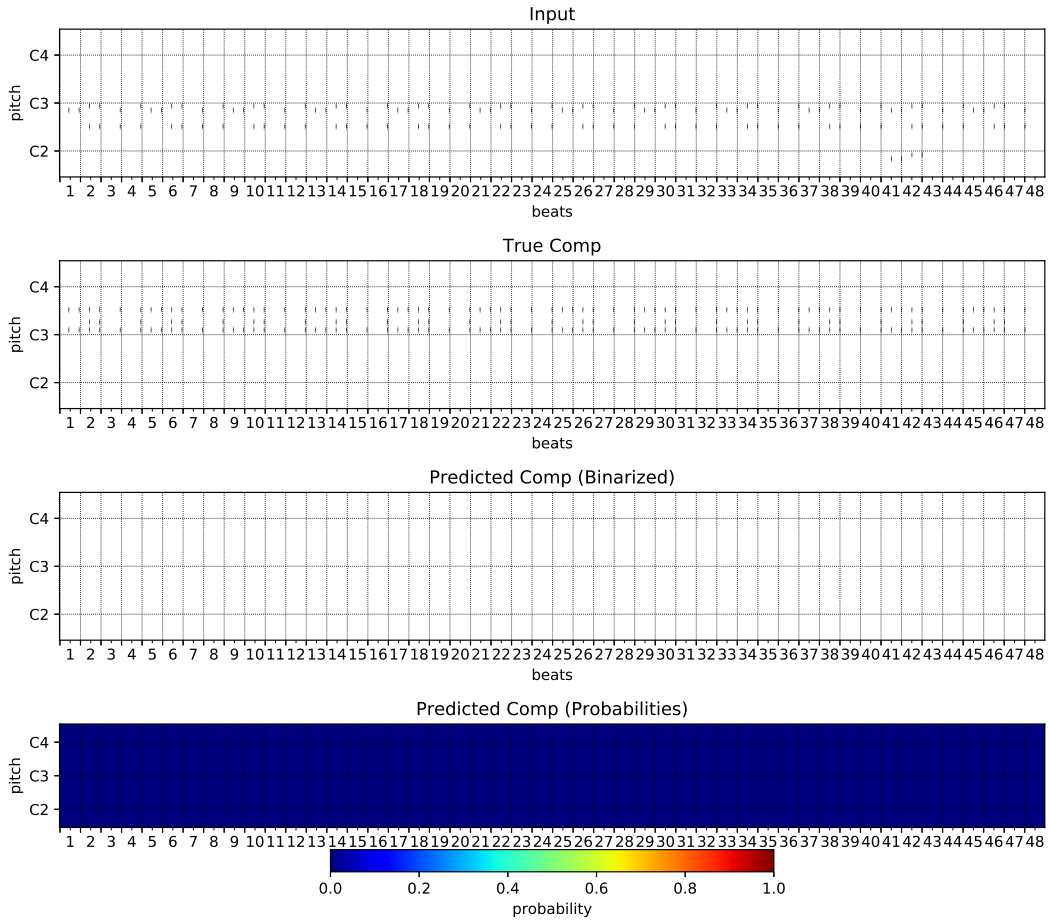


FIGURE 4.30: Input from the validation dataset, ground truth accompaniment, predicted accompaniment, and prediction onset probabilities using the latent-variable RNN. We see that on the test data, the model is unable to make good predictions, and gets stuck predicting empty units for any new inputs. Furthermore, the plot of onset probabilities (*bottom*) shows that the model is predicting "no onset" with high confidence (near zero probabilities).

interaction between a human musician and the system, where the system interprets a human's input through the latent space and returns a response in a turn-based interaction.

Chapter 5

Implementation

The whole of the project work has been split into two main bodies: Model development and interface development. Development on the two bodies of work are largely separate, with different development environments and tools, yet was carried out in tandem, with the results from one informing the direction of the other.

This chapter describes the implementation work done on both sides, and how they tie together in the end to produce the resulting accompaniment system, Comper.

A number of software products which arise from the work will be made available publicly, through two separate repositories on the online code-sharing platform GitHub (*Build software better, together*):

1. **comper** | **https://github.com/JunShern/comper**: Contains all work and documentation for model development and training including code, procedures and experiment results.
2. **comper-ui** | **https://github.com/JunShern/comper-ui**: Contains the user-interface of Comper which can be used to run the live accompaniment demos.

5.1 Model Development

In this section, we discuss the platform and tools which were used during the algorithm development phase.

5.1.1 Platform

For the purpose of training deep learning models efficiently, it is important to have a machine with strong GPU capability. In this project, we use a machine instance running on Google Cloud Platform (*Google Cloud Computing, Hosting Services and API*) with the following specifications:

- **CPU**: 6 vCPUs
- **GPU**: 1 x NVIDIA Tesla K80
- **RAM**: 30 GB
- **Disk**: 500 GB standard persistent disk

The machine is accessed via SSH through a local laptop. The local machine and cloud instance are used hand-in-hand, with most development work being done locally, then models are trained and tested on the cloud instance.

GitHub is used to sync development between the local and cloud machines, complying standard `git` practices for version control. Larger files such as datasets or saved models exceed the memory limit of GitHub repositories, so are left out of the version control flow (by use of a `.gitignore` file). Instead, Secure File Transfer Protocol (SFTP) is used to transfer these larger files between the local and cloud machines (*Transferring Files to Instances | Compute Engine Documentation | Google Cloud* 2018).

5.1.2 Organizing and Managing Experiments

Perhaps an underestimated facet of deep learning research is the difficulty in managing large volumes of experiments, especially when experimentation is carried out in an exploratory nature. Each experiment to test the performance of a new model needs to record all variables and training parameters. Furthermore, all work on algorithm design and model experiments carried out throughout the project was done with reproducibility in mind.

To help on the fronts of experiment logging as well as documentation for reproduction, we use a development environment called Jupyter Notebooks (*Project Jupyter* 2018). Jupyter Notebooks is an open-source web application which provides an environment for creating documents which include live code, equations, visualizations and narrative text. These features allow the notebooks are to be used simultaneously as the development environment as well as as the documentation or experiment logbook, which is what is done for this project.

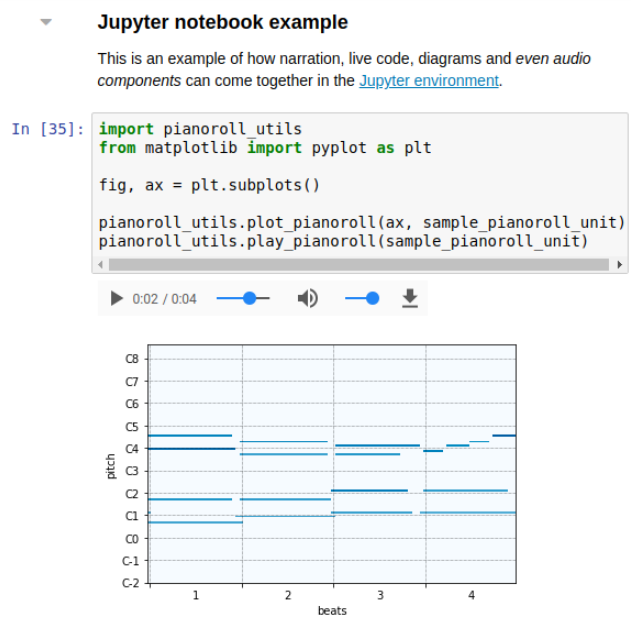


FIGURE 5.1: A screenshot of a Jupyter Notebook. Also seen in the picture are the audio playback and visualization tools developed in part for this project.

We provide 5 Jupyter Notebooks which include experimental procedure, code, results and narration in full detail (all notebooks are available on the GitHub repository):

- **Data Preparation** | `data_prep.ipynb`: This notebook includes full documentation of the procedure for preparing the dataset used in the experiments, including runnable code which can be used to generate new datasets.

- **End-to-end Models** | `end-to-end.ipynb`: This notebook includes the code for reproducing the end-to-end models discussed in 4.4, as well as preamble and postmortem analyses for each experiment.
- **Autoencoder Models** | `autoencoder_models.ipynb`: This notebook includes the code for reproducing the autoencoder models (including variational autoencoders) discussed in 4.5, as well as preamble and postmortem analyses for each experiment.
- **Latent Variable Sequence Models** | `latent_rnns.ipynb`: This notebook includes the code for reproducing the latent space experiments discussed in 4.5, as well as preamble and postmortem analyses for each experiment.
- **Miscellaneous** | `misc_playground.ipynb`: This notebook includes the code and narration for miscellaneous topics not directly belonging to any of the previous notebooks including loss function analyses, troubleshooting live demo models, and producing graphs for this report.

Since all of the notebooks share common links to datasets and user-defined variables (such as the pianoroll pitch range or beats-per-unit), all of the above notebooks depend on a standalone declarations file `imports_and_user_variables.py` which contains all the necessary definitions and import statements to be shared across all notebooks.

5.1.3 Software and APIs

The entirety of the model development software has been developed using the Python programming language (*Welcome to Python.org*), chosen for the strong ecosystem of packages and libraries for deep learning in Python.

To facilitate reproducibility and ease-of-setup on new environments, there is a `requirements.txt` file which can be used with the Python `pip` package manager (*pip*) to install the required Python libraries including the correct versions used during the experiments.

In the following sections, we describe the software tools used for model development, including open-source libraries as well as custom software.

Machine Learning

The backbone of our data representation relies on `numpy`, an open-source library which provides features for complex numerical operations in Python including data structures for multi-dimensional arrays, fast matrix operations and much more. Therefore, the pianoroll matrix representations used in our work are all numpy arrays.

For deep learning, we use a deep learning framework called Keras (*Keras: The Python Deep Learning library*). The Keras API provides access to a library called Tensorflow (Abadi et al., 2015), and is built in a way that allows developers to deal with high-level constructs of deep learning such as convolutional layers or recurrent layers instead of individual neurons. The primary data structure in Keras are called tensors, which are compatible with numpy arrays.

A Keras-generated summary of the dense end-to-end pianoroll prediction model architecture (previously seen in 4.4) is shown in Listing 5.1. This is a typical way to visualize model architectures in Keras, and is our chosen method of visualization throughout this report due to simplicity and conciseness. The summary is mostly self-explanatory but worth mentioning is the Output Shape, which shows the output

dimensions of each layer. Layer dimensions in Keras follow the convention `[BATCH, (TIME), DATA, ..., (DATA), (CHANNEL)]`, where the dimensions in round brackets are optional depending on the type of layer. For example, recurrent layers require the TIME dimension and convolutional layers require the CHANNEL dimension. This convention is not strictly enforced by the API but we follow this convention for our experiments.

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_5 (InputLayer)         (None, 4, 96, 96, 1) 0
_________________________________________________________________________________________________
input_6 (InputLayer)         (None, 4, 96, 96, 1) 0
_________________________________________________________________________________________________
time_distributed_5 (TimeDistrib (None, 4, 9216)      0           input_5[0][0]
_________________________________________________________________________________________________
time_distributed_6 (TimeDistrib (None, 4, 9216)      0           input_6[0][0]
_________________________________________________________________________________________________
lstm_5 (LSTM)                (None, 800)           32054400    time_distributed_5[0][0]
_________________________________________________________________________________________________
lstm_6 (LSTM)                (None, 800)           32054400    time_distributed_6[0][0]
_________________________________________________________________________________________________
concatenate_3 (Concatenate)  (None, 1600)          0           lstm_5[0][0]
                                                               lstm_6[0][0]
_________________________________________________________________________________________________
dense_5 (Dense)              (None, 1000)          1601000     concatenate_3[0][0]
_________________________________________________________________________________________________
dropout_3 (Dropout)          (None, 1000)          0           dense_5[0][0]
_________________________________________________________________________________________________
dense_6 (Dense)              (None, 9216)          9225216     dropout_3[0][0]
_________________________________________________________________________________________________
reshape_3 (Reshape)          (None, 96, 96, 1)     0           dense_6[0][0]
=================================================================================================
Total params: 74,935,016
Trainable params: 74,935,016
Non-trainable params: 0
```

LISTING 5.1: Architecture of an end-to-end dense RNN. The shape of each layer is preceded by None, since the first dimension corresponds to a variable **batch size** of training data. Also note that the last dimension of the InputLayer is 1, corresponding to a single-channel pianoroll matrix.

Additionally, the Keras API allows us to train and save models on disk for later use, which is important to allow us to be able to use our trained models in the online system.

Datasets on RAM vs Datasets on Disk

A special consideration necessary for our project is the consideration of large training datasets. We have established that the chosen data structure for our musical units from the dataset is numpy arrays. This is a sufficient approach when dealing with datasets which can fit entirely in the machine's RAM. However, when we load up a large dataset comprising millions of datapoints, it is likely that the memory requirement of that dataset will exceed the available RAM on the computer, so it will not be possible to load all the data at once for training.

We provide a quick analysis of the expected memory footprint of our dataset:

- **Number of units**: 2,000,000 (approximately 100 units per song, with 20,000 songs in the Lakh Pianoroll Dataset)
- **Size per unit**: 36864 bytes (using a 96x96 numpy matrix of 16-bit floats)
- **Total size**: 74GB (2,000,000 * 37kB)

For a single dataset, we expect less because many songs don't have pianorolls, and many pianorolls have empty units. Also, in this project we choose to train on a subset of songs rather than all 20,000 songs, for easier training given limited time.

That said, we often wish to have more than one dataset loaded at a time, for example both noisy and clean versions of the dataset when training a denoising autoencoder. These multi-gigabytes of data far exceeds RAM memory on standard machines (typical laptops have 4-8GB of RAM) and even cloud machines (the current setup has 30GB of RAM).

The solution to this problem is to save dataset variables on disk and load the data into RAM dynamically, based on which portion of the data is currently being used. We use a library called h5py (*HDF5 for Python*) to help manage dynamic-loading of variables to and from disk in the HDF5 binary data format (*HDF Home* 2018). Using h5py, we are able to use numpy-style arrays which are stored on disk rather than in RAM.

Music and MIDI handling

The core representation of music in our experiments is the pianoroll matrix. However, the most well-supported format in digital music is the MIDI format, which we need to use in order to playback our results from digital synthesizers. We use the mido library (*Mido - MIDI objects for Python*) to provide MIDI functionality such as data structures for MIDI messages, routing of MIDI messages to and from external ports (including MIDI instruments and MIDI synthesizers), and saving streams of messages as MIDI files.

To bridge the divide between the our numpy pianoroll matrices and sequences of mido MIDI messages, we develop a function to convert from pianoroll matrices to MIDI event sequences (adapted from a similar function in the pypianoroll library (Dong et al., 2017)). A simplified implementation of the function is shown in Listing 5.2.

```
def pianoroll_2_events(pianoroll, min_pitch=0, max_pitch=127, is_onsets_matrix=False):
    """
    Takes an input pianoroll of shape (NUM_PITCHES, NUM_TICKS)
    and returns a list of quantized events. Adjacent nonzero values
    of the same pitch will be considered a single note with their
    mean as its velocity.
    """
    num_pitches = pianoroll.shape[0]
    num_ticks = pianoroll.shape[1]

    events = [[] for _ in range(num_ticks)] # Each tick gets a list to store events
    clipped = pianoroll.astype(int)
    binarized = clipped.astype(bool)
    padded = np.pad(binarized, ((0, 0), (1, 1)), 'constant')
    diff = np.diff(padded.astype(int), axis=1)

    for p in range(num_pitches):
        pitch = min_pitch + p
        note_ons = np.nonzero(diff[p,:] > 0)[0]
        note_offs = np.nonzero(diff[p,:] < 0)[0]
        for idx, note_on in enumerate(note_ons):
            velocity = np.mean(clipped[p, note_on:note_offs[idx]])
            # Create message events
            on_msg = mido.Message('note_on', note=pitch, velocity=int(velocity), time=0)
            events[note_ons[idx]].append(on_msg)
            if note_offs[idx] < num_ticks and not is_onsets_matrix:
                off_msg = mido.Message('note_on', note=pitch, velocity=0, time=0)
                events[note_offs[idx]].append(off_msg)
    return events
```

LISTING 5.2: Function to convert from a pianoroll matrix representation to a list of MIDI events.

It is often desirable during development to be able to playback pianoroll matrices as audio clips. We achieve this in our experiments by first saving the list of MIDI events as a MIDI file using mido, then using the command-line tool TiMidity++ (*TiMidity++*) to synthesize that MIDI file as a `.wav` file. The Jupyter environment can provide direct playback of that `.wav` file in-place, using the browser-native HTML `<audio>` element. This entire functionality is wrapped up into a single function for

convenient playback of pianoroll matrices. An example of this audio playback functionality was seen previously in Figure 5.1.

Visualization

To visualize data, we use Matplotlib (*Matplotlib*), which provides functionality for plotting heatmaps, histograms, bar plots, line plots, scatter plots and many more, with good support for numpy arrays.

A number of visualizations are repeated frequently throughout the project, for example visualizing the pianoroll matrix, pitch histograms and others. We develop custom functions for these common visualizations, with initial code based on the pypianoroll library (Dong et al., 2017) but extended to suit our needs.

Custom Modules

To facilitate code reuse, we develop two modules which provide common access to common operations and functions throughout the project:

1. `pianoroll_utils.py`: Utility functions bundling common operations for pianoroll matrices, including:
    - *Pianoroll manipulation* - Pitch range cropping, padding cropped matrices, key transposition
    - *Data representation conversions* - Extract pianoroll units from full-song pianorolls; convert pianoroll units to onsets matrix, convert pianoroll to MIDI events
    - *Visualization and playback* - Pianoroll visualization, pitch class histogram visualization, onsets/velocity visualization, audio playback
2. `custom_loss.py`: Implementations for all loss functions and metrics as described in 4.3.

5.2 Interface Development: The Making of Comper

After having trained our models, we have all the pieces we need to create all the online accompaniment system, Comper[1]. This section details the implementation of the Comper prototype, which provides access to two different interaction modes, **Call-and-Response** and **Accompaniment**, using the accompaniment models trained in Chapter 4. The result is a final accompaniment system which can be used by a musician.

5.2.1 Overview

At a very high-level, the interactions of the proposed system can be illustrated using the block diagram as in Figure 5.2.

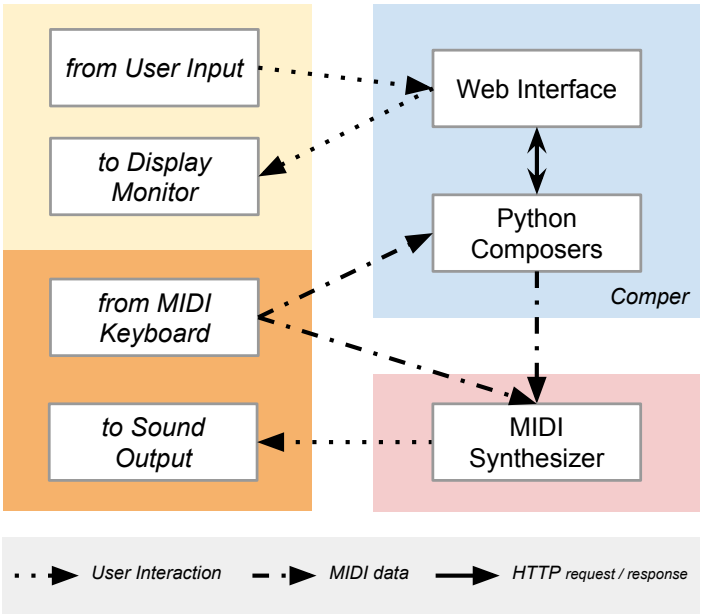


FIGURE 5.2: High-level overview. There are two agents, the user (*orange*) and Comper (*blue*). Additionally, there is a MIDI synthesizer (*pink*) which is needed to convert Comper's MIDI output to sound.

The system consists of two agents - a human musician (orange/light orange portion of Figure 5.2) and Comper (blue portion of Figure 5.2). Additionally, there is a MIDI Synthesizer (pink portion of Figure 5.2) which is not a part of Comper, but is a required component to convert Comper's MIDI output to sound.

A typical user flow is as follows: The user plays a MIDI keyboard, which is physically connected to Comper via USB. Comper interprets MIDI messages from the keyboard and its internal composer program generates an appropriate musical response via the MIDI synthesizer, while also providing visual feedback of the interaction through its web interface. Optionally, the user can choose to alter Comper's interaction modes and settings through the web interface.

[1]The name 'Comper' comes from the term 'comping' which means 'to accompany' or 'to complement'. The term is often used in jazz music to describe improvising accompaniments.

5.2.2 Hardware

Comper is designed to run on any typical computer - it has been designed and tested on a Lenovo Ideapad laptop with no dedicated GPU and four vCPUs on Intel Core i5 processors.

For music collaboration, the user needs to have a MIDI instrument (such as a MIDI keyboard) which can connect to the Comper-installed laptop via USB. The current system has been tested with an M-Audio Keystation 61 MIDI keyboard.

5.2.3 MIDI Interface

The MIDI API (Association, 1983) is a protocol for digital communication of symbolic music data, and is well-specified across multiple platforms including Windows, Mac and Linux. Using this API, MIDI applications are able to stream MIDI messages to and from other MIDI applications and class-compliant USB MIDI devices (such as hardware MIDI keyboards and controllers). More information about MIDI and its specifications is provided in Appendix A.

Of direct relevance, the JACK Audio Connection Kit (*JACK Audio Connection Kit*) is a cross-platform API which provides real-time inter-application routing of audio and MIDI streams. This is convenient because instead of specifying connections to particular input and output MIDI ports from each application node, all JACK-compliant applications can simply advertise their own MIDI input or output ports, and a JACK-router application such as QjackCtl on Linux (*QjackCtl: JACK Audio Connection Kit - Qt GUI Interface*) can be used to make or break connections between any or all available ports on-the-fly, from a single interface.

Therefore, the Comper system also relies on JACK via QjackCtl to route MIDI messages from the MIDI keyboard to Comper, and from Comper to the MIDI synthesizer. An example of MIDI routing using QjackCtl is shown in Figure 5.3.

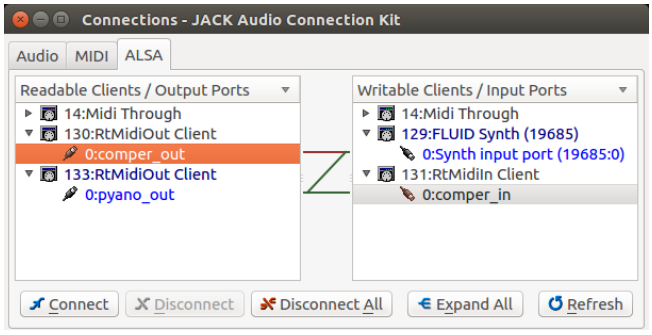


FIGURE 5.3: An example of routing MIDI streams on QjackCtl's user interface. This particular example shows connections from a MIDI instrument (Pyano) feeding into the input of Comper, and both outputs of Pyano and Comper are sent to the software synthesizer (Fluidsynth) to generate sounds from the MIDI messages.

5.2.4 Software

The software side of the Comper prototype currently follows a *partial* **client-server architecture** to allow for flexible user interactions. Due to the length and complexity of the code, it is not helpful to include all relevant snippets in this report; instead, the

full code for Comper is made available online at https://github.com/JunShern/comper-ui. The main software nodes of the system are as follows:

1. **Server / Composer program**

   This Python program runs the majority of Comper's core capabilities including receiving MIDI messages, composing a response, and sending the response out to the synthesizer. On top of that, this program also acts as a server for Comper's user interface, sending composition information to the user interface for visualization and waiting on peripheral user interactions from the interface.

2. **Client / User interface**

   This is a web-application which produces visual feedback on compositions and interactions using information sent by the server, and provides a user interface for selecting the composer program's interaction modes and settings.

3. **MIDI synthesizer**

   This is not a part of Comper, but is essential for converting Comper's MIDI output to audible music. This can be any software MIDI synthesizer - the system has been tested with FluidSynth (*FluidSynth*), which works well.

Composer program

Comper's composer program was designed to be an extendable interface for developing composer algorithms. It is developed in Python, using the Python Mido library (*Mido - MIDI objects for Python*) to provide access to the MIDI functionalities including data structures for MIDI messages, and routing to and from other MIDI ports.

The core functionality of Comper is encapsulated in the base Composer class, which simply does the following:

- Provide access to input and output MIDI ports
- Offers a dummy callback `register_player_note` method which is called upon receipt of incoming MIDI messages
- Offers a dummy `generate_comp` method which is run repeatedly in its own thread
- Provides standard methods and data structures to record incoming MIDI messages, record outgoing MIDI messages, calculate timing information for new MIDI messages, and keep track of current note (ON/OFF) states

As such, it is simple to extend this class to build a new type of composer - by inheriting the Composer base class, new composer implementations only need to override the `register_player_note` and `generate_comp` methods to handle new notes and produce new notes appropriately. A skeleton of the Composer base class is shown in Listing 5.3.

```
class Composer(object):
    """
    Base class for possible composer types, implements:
    - causal memory states of music
    - callback function for new notes
    - empty comping function
    """
    def __init__(self):
        self.input_messages = collections.deque(maxlen=MEMORY_LENGTH)
        self.comp_messages = collections.deque(maxlen=MEMORY_LENGTH)
        self.active_notes = []
        self.previous_event_time = 0

    def get_deltatime(self):
        """
        Delta time gives the time since the previous MIDI event
        """

    def register_player_note(self, msg, precision=None):
        """
        Registers the deltatime of a note_on/note_off message,
        and keeps track of all messages
        """

    def add_to_input_memory(self, msg):
        """
        Stores note_on/note_off messages in input memory,
        and updates the currently active notes
        """

    def add_to_comp_memory(self, msg):
        """
        Stores generated messages in comp memory
        """

    def generate_comp(self, _):
        """
        Dummy function - should be overriden in child classes
        """

    def exit(self):
        return
```

LISTING 5.3: Composer class skeleton including method prototypes and docstrings.

Based on the results of Chapter 4, we define two main accompaniment modes:

1. **Unit Call-and-Response**

   This interaction mode is inspired by (Bretan, 2017), and entails a turn-based interaction between the musician and Comper where the musican plays for one unit and Comper plays for the next unit, and then the interaction repeats.

   To implement this, we define a subclass of Composer called UnitLooper, which overrides `generate_comp` to last for one full unit per call. During each call, `generate_comp` either stays silent for the user to play their turn, or plays back a predicted comp pianoroll if it is Comper's turn. During the user's turn, `register_play_note` is used to update an input pianoroll which captures what the user has played during this turn. At the end of the turn, the input pianoroll is fed into the **variational autoencoder (decoder reconstruction)** model from 4.5, which is used to generate a new comp pianoroll for playback during Comper's next turn.

   A fixed drum pattern is played throughout the entire interaction, to indicate the tempo that the user must follow.

   An abridged version of the overriden `generate_comp` method for the Call-and-Response interaction is shown in Listing 5.4.

```
def generate_comp(self, outport):
    """
    Simultaneously carries out three goals:
    1. Acts as a metronome by playing drum sounds at each beat
    2. Plays back recorded user input in a loop
    3. Plays back the generated accompaniment
    """

    # Empty the input
    self.input_pianoroll = np.zeros((self.num_pitches, self.num_ticks))
    self.input_events = [[] for _ in range(self.num_ticks)] # Each tick gets a list to store events

    DRUM_CHANNEL = 9
    HI_HAT, BASS, SNARE = (42, 36, 38)
    beat_sounds = [(BASS, HI_HAT), (HI_HAT,), (HI_HAT,), (HI_HAT,)]
    # Loop through every tick in every beat
    for beat in range(self.beats_per_bar):
        # Play drum sounds at each beat
        for sound in beat_sounds[beat]:
            msg = mido.Message('note_on', note=sound, velocity=100, time=0.)
            msg = msg.copy(channel=DRUM_CHANNEL)
            outport.send(msg)
        # Play recorded messages and wait at each tick
        for tick in range(self.ticks_per_beat):
            self.current_tick = beat*self.ticks_per_beat + tick
            if self.loopcount % 2 == 0:
                for msg in self.input_events[self.current_tick]:
                    outport.send(msg.copy(channel=COMP_CHANNEL, time=0))
            else:
                for msg in self.comp_events[self.current_tick]:
                    outport.send(msg.copy(channel=COMP_CHANNEL, time=0))
            time.sleep(self.seconds_per_tick)

    # Predict comp events for the next unit
    if np.sum(self.input_pianoroll) > 0:
        self.comp_pianoroll = self.unit_predictor.get_comp_pianoroll(self.input_pianoroll)
        self.comp_events = pianoroll_utils.pianoroll_2_events(self.comp_pianoroll)
    else:
        self.comp_events = [[] for _ in range(self.num_ticks)] # Each tick gets a list to store events
    self.loopcount += 1
```

LISTING 5.4: Unit Call-and-Response method generate_comp which either waits or plays an accompaniment for 4-beats. At the end of the unit, the method calls unit_predictor.get_comp_pianoroll(), which returns a new comp unit using a trained VAE model for playback during the next turn.

2. **Unit Accompanier**

The Unit Accompanier mode similarly handles units as the Unit Call-and-Response mode does, but instead of being turns-based, both the user and Comper are simultaneously in play.

As before, `register_play_note` is used to capture the user's notes in an input pianoroll. At the end of each unit, the window of the past 4 input and comp units are fed into the **RNN encoder-decoder** model from 4.4 to generate the next comp unit. The predicted comp unit is then played during the next unit using `generate_comp`. An abridged version of the `generate_comp` method for the Unit Accompanier is shown in Listing 5.5.

Note that at the start of the interaction, we have to pad the window with empty input and comp units so that we feed the right number of units into the prediction model.

```
def generate_comp(self, outport):
    """
    Simultaneously carries out two goals:
    1. Acts as a metronome by playing drum sounds at each beat
    2. Plays back the generated accompaniment
    """
    # Empty the input
    self.input_pianoroll = np.zeros((self.num_pitches, self.num_ticks))
    self.input_events = [[] for _ in range(self.num_ticks)] # Each tick gets a list to store events

    DRUM_CHANNEL = 9
    HI_HAT, BASS, SNARE = (42, 36, 38)
    beat_sounds = [(BASS, HI_HAT), (HI_HAT,), (HI_HAT,), (HI_HAT,)]
    # Loop through every tick in every beat
    for beat in range(self.beats_per_bar):
        # Play drum sounds at each beat
        for sound in beat_sounds[beat]:
            msg = mido.Message('note_on', note=sound, velocity=100, time=0.)
            msg = msg.copy(channel=DRUM_CHANNEL)
            outport.send(msg)
        # Play recorded messages and wait at each tick
        for tick in range(self.ticks_per_beat):
            self.current_tick = beat*self.ticks_per_beat + tick
            for msg in self.comp_events[self.current_tick]:
                outport.send(msg.copy(channel=COMP_CHANNEL, time=0))
            time.sleep(self.seconds_per_tick)

    # Predict comp events for the next unit
    self.comp_pianoroll = self.unit_predictor.get_comp_pianoroll(self.input_pianoroll)
    self.comp_events = pianoroll_utils.pianoroll_2_events(self.comp_pianoroll)
    self.rec_count += 1
```

LISTING 5.5: Unit Accompaniment method generate_comp which plays an accompaniment for 4-beats. At the end of the unit, the method calls unit_predictor.get_comp_pianoroll(), which returns a new comp unit using a trained RNN Encoder-Decoder model for playback during the next turn.

Using this framework, any of the composer models developed in Chapter 4 are available for use either via Unit Call-and-Response or Unit Accompanier mode.

Client-Server Interface

The client-server architecture is a popular paradigm for developing web applications. Simply put, the client is the front-end application that runs on the user's browser, and the server is a program that connects to the client and runs outside of the browser, fetching information or otherwise responding to what the client requests.

In our case, the **server** is a Python program which runs both the composer program discussion previously, and communicates with the client via HTTP requests to update the client's visualizations or change the composer state based on user input from the client.

On the other hand, the **client** is a web application which serves as the user interface to Comper, using browser-based technologies (HTML5, Javascript and CSS). The reason we choose to use the browser as a tool for building our user interface is because browsers offer strong cross-platform graphical user interface (GUI) functionality, and there is scope for making the project more portable and accessible online in future work (See 9.1).

Importantly, even though we are using web technologies to build our interface, all of our programs are run locally - nothing is sent over the Internet. This is important to minimize latency in interactions between the client and the server.

A bare-bones prototype of this client-server interface has been completed, and currently provides the following interactions:

- **MIDI message visualization**: The current version provides checkbox selectors to choose from available MIDI inputs, and sends a HTTP GET request to receive MIDI messages from the Composer. When the client receives this, it

prints the MIDI messages to the console and displays the current state (ON/OFF) of all 127 possible MIDI note pitches. Each MIDI channel is mapped to a different color for ease of recognizing which input the messages correspond to (each player's instrument can be broadcasted on a different MIDI channel from 1-16). The client requests a stream of MIDI messages from an arbitrary number of MIDI inputs via the server, prints these to the screen, and visualizes the states of current notes.

- **Composer class selection**: The client provides a dropdown selector which allows the user to choose from one of the available Composer classes. On the press of a button, the client sends a HTTP POST request to the server which stops the current Composer program and starts it again with the new user-selected Composer.

Finally, we can see a screenshot of Comper in action in Figure 5.4.

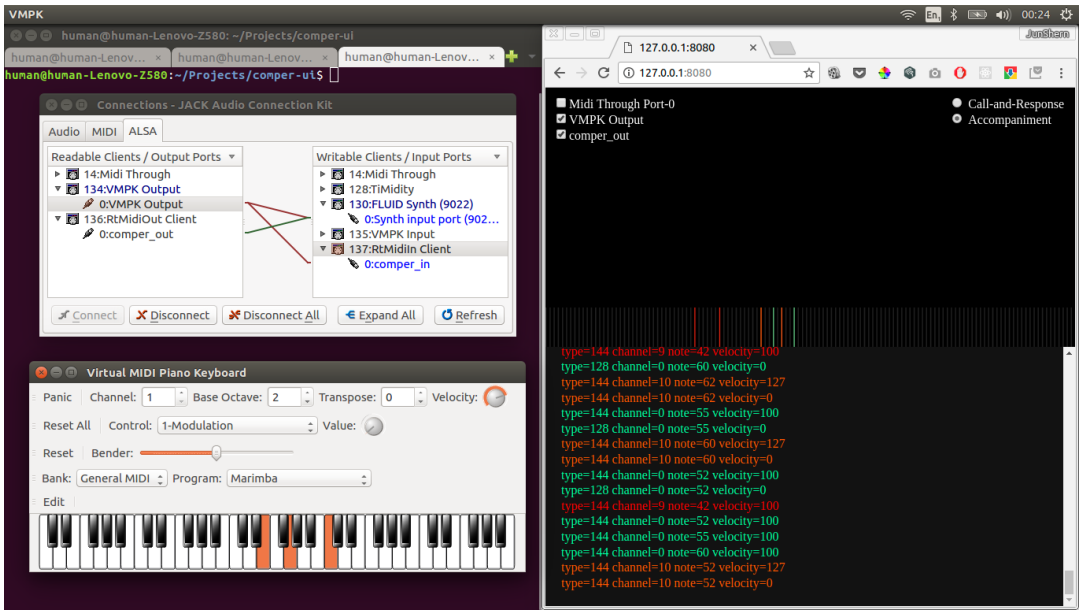


FIGURE 5.4: Comper in action. The connections into the system are shown in QjackCtl in the top left window. On the bottom left is a virtual piano keyboard, which the user plays as an input to Comper. The window on the right shows Comper's user interface running in a browser, set to Accompaniment mode and visualizing inputs from the user (*turquoise*), Comper itself (*orange*), as well as the drum notes for the keeping in time with the unit's beats (*red*).

Chapter 6

Testing

In this section, we put Comper to the test and evaluate the quality of its output in a live demo setting.

As discussed previously, there are two main interaction modes for Comper: **Call-and-response** and **Accompaniment**. Both of these interactions are tested individually but using a common evaluation procedure:

1. A human musician is selected to participate in the live demo, and asked to play a set of 50 units on the input MIDI keyboard, either by taking turns or in one continuous song depending on the interaction mode.
2. Comper accompanies the human musician
3. The entire interaction is recorded, including both input and comp units.
4. The quality of the output units is evaluated for pitch and tempo properties. (Analysis provided in the next section)

Figure 6.1 and Figure 6.2 show 5 randomly-selected pairs of units from the recorded test session.

6.1 Evaluating Musical Quality

To measure the musical quality Comper's output units, we use two of the metrics previously defined in 4.3:

- **Pitch Class Histogram Distance**: This is used to measure the similarity in pitch classes (therefore **key**) between each input and output unit pair. Returns a distance value between 0 and 1, where 0 is maximum similarity and 1 is completely orthogonal.
- **Rhythm Loss**: This is used to measure the rhythmic quality of Comper's output units on its own in terms of whether the notes are struck at ordinary beat-intervals or not. Returns a score between -1 and 1, where -1 is perfectly on-beat and 1 is perfectly off-beat.

Both metrics are calculated for all 50 unit-pairs of each interaction. The mean results across all units for each interaction are shown in Table 6.1.

Interaction Style	Mean Pitch Histogram Distance	Mean Rhythm Loss
Call-and-Response	0.43	0.13
Accompaniment	0.72	-1.00

TABLE 6.1: Mean evaluation metrics for 50 units of live interaction with a human musician on each of Comper's two interaction styles. Pitch Histogram Distance is measured between 0 and 1, where 0 is maximum similarity and 1 is completely orthogonal; Rhythm Loss is measured between -1 and 1, where -1 is perfectly on-beat and 1 is completely off beat.

6.2 Analysis

The first thing to observe from these tests is that the two interactions are different in large ways.

The Call-and-Response interaction uses the VAE decoder reconstruction model which depends on the onsets representation, and is therefore carried out using a marimba voice with polyphony but only have onsets (no distinct note release time).

On the other hand, the Accompaniment interaction uses the RNN Encoder-Decoder model which depends on the one-hot encoded pianoroll matrix representation, and so is carried out using a piano voice but is limited to monophonic bassline accompaniment.

These differences affect the results of the metrics in very important ways. In particular, for the Accompaniment interaction the metrics are not fully suitable since each unit only produces a single note throughout the whole unit. For instance, since every note produced by Comper during Accompaniment mode is produced exactly at the start of each unit, this is judged by the Rhythm Loss metric as 100% on-beat (-1). This is true by definition, but it fails to capture the fact that playing a single note at the start of each measure is musically boring. Furthermore, in the case of single-note units, the Pitch Class Histogram Distance is always going to judge that unit harshly although that one note may be a harmonically suitable accompaniment for that input. This can also be considered a failing of the model for producing only a single note per unit, but the metrics do not fully capture the values or shortcomings of the Accompaniment units.

The chosen metrics are more suited for the Call-and-Response interaction. As we can see in Figure 6.1, the notes of response units are generally in a similar pitch class as those of the input units, which is appropriately captured by the Pitch Class Histogram Distance score of 0.43. In terms of rhythm, we see that the response units do not seem to follow ordinary beat-divisions, and this is also reflected appropriately in the poor Rhythm Loss of 0.13.

Ultimately, the test results from Table 6.1 show that Comper performs better pitch-wise during Call-and-Response mode, but follows the beat better during Accompaniment mode.

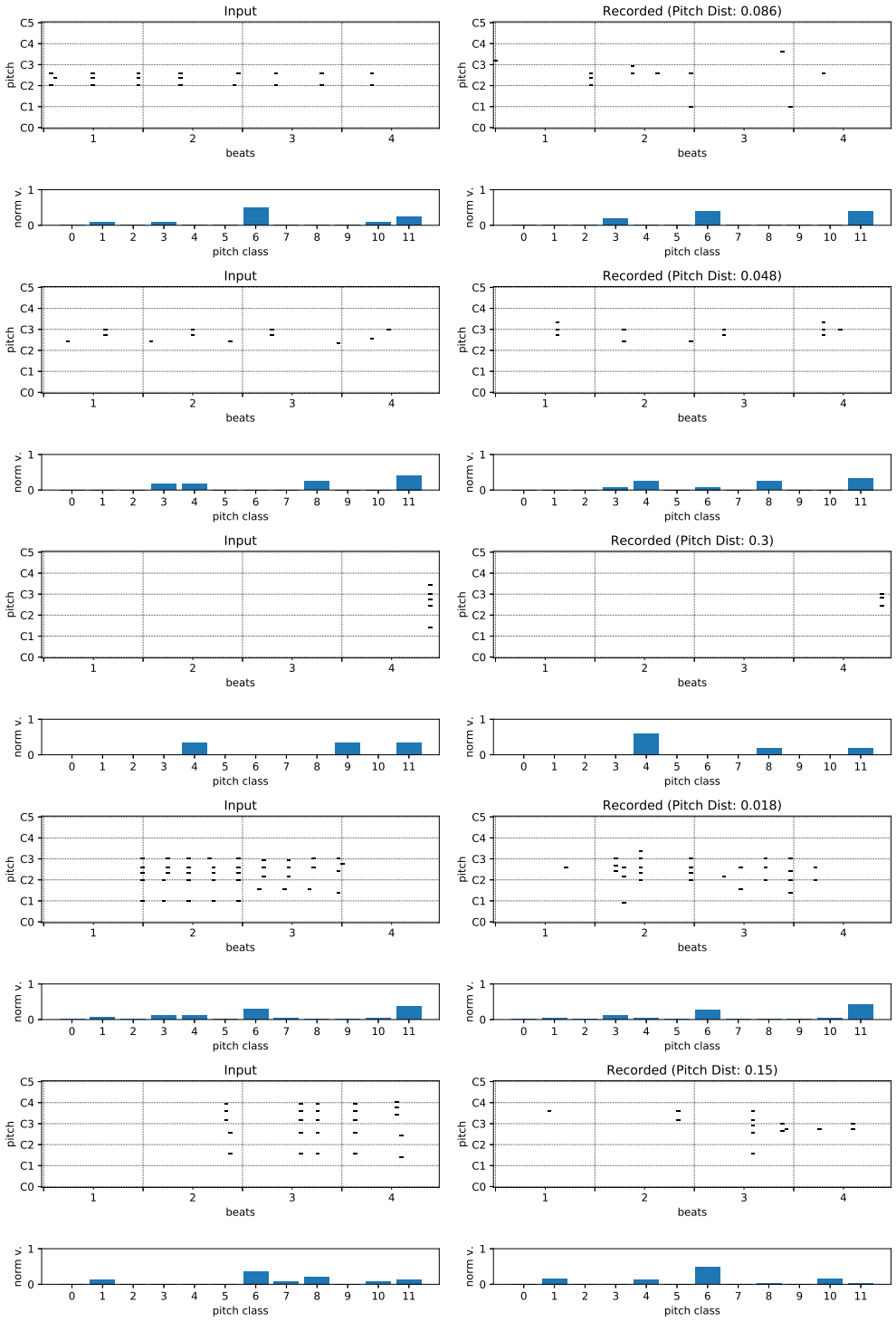


FIGURE 6.1: Examples of input and output units for a Call-and-Response interaction, recorded directly from Comper through the composer program. This interaction is turn-based, so the musician plays an input (*Left*) which is then interpreted by Comper which plays its response (*Right*) during the next unit. We see that Comper's response units share similar pitch qualities to the input while adding its own original twist.

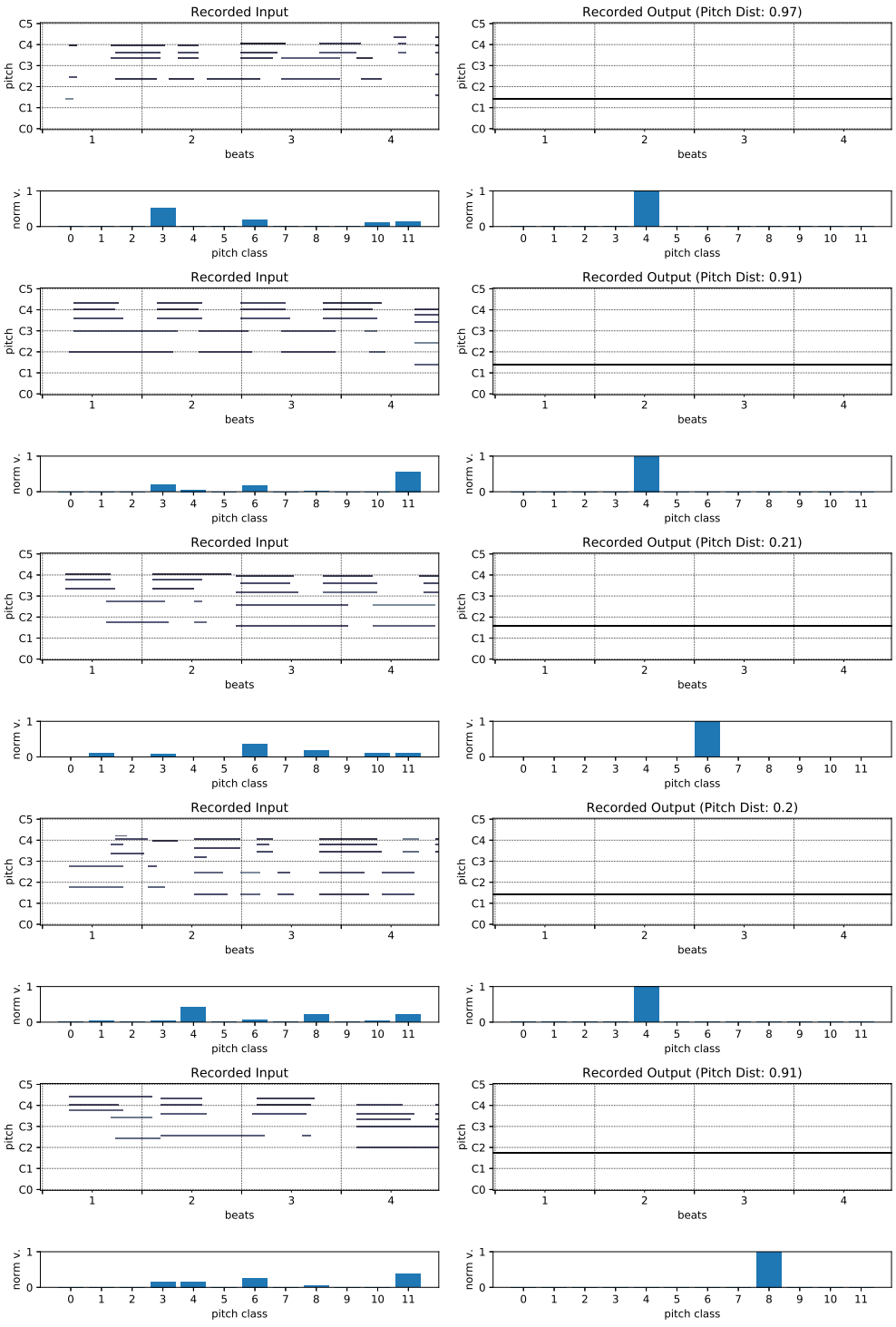


FIGURE 6.2: Examples of input and output units for a Accompaniment interaction, recorded directly from Comper through the composer program. For this interaction, the musician's input (*Left*) is played at the same time as Comper's accompaniment (*Right*). We see that Comper's response units remain constant throughout each unit, and responsiveness to the input is not obvious.

Chapter 7

Results

In this chapter, we evaluate how well the system satisfies the initial requirements discussed in Chapter 3. To accomplish this assessment, the requirements from before are simply laid out systematically and addressed one-by-one:

7.1 How to Be A Good Band Member

7.1.1 The Bare Minimum: "Don't Step On My Toes"

- *The accompaniment system should not obstruct the experience of the musician or listeners, beyond any limitations imposed by the scope of the project.*

  There is unfortunately an additional constraint imposed by the current system that was not set by the scope of the project: The music in the interaction has to follow a fixed tempo, since the system depends on beat-aligned units. Besides this, there are no further obstructions.

- *The musical output from the accompaniment should (within a certain degree of tolerance) obey the following basic principles of music theory:*

  - *Stay in key: Stays within the same key as input music (from the human player)*
    As seen in Chapter 6, both interaction modes of Comper are able to respond to key changes in the input. However, there are occasional mistakes with notes being struck outside of the correct key, and these stick out very clearly to a listener.
  - *Stay on beat: Note onsets should occur at quantized time intervals according to the beat and the most common musical beat-fractions (whole-, half-, quarter-, eighth-, sixteenth-, and triplet-beats)*
    Again, Chapter 6 validates that both of Comper's interaction modes are capable of following the beat, but perfect beat-following is not always achieved and results vary depending on current input as well.

7.1.2 Added Value: "You're Pretty Good, Kid"

- *The musical output should provide accompaniments in the form of fills, basslines, harmonies or melodies in ways that are musically pleasant in the ears of the observing audience.* ***(Needs listener evaluation)***

  Due to time constraints, we were not able to test this with listener trials. However, simple observation suggests that although the accompaniment provided by the system satisfies the previous benchmark, it would fail to impress an average human listener.

7.1.3 Thought Leadership: "That's A Great Idea, Let's Go With That!"

- *The collaboration system should be capable of producing original and interesting ideas relevant to the current musical context*

  Since the system is unable to impress at the previous benchmark, by the cumulative definition of our benchmarks it also cannot satisfy this third benchmark.

- *The collaboration system should be able to make decisions about the progression of the music and lead the transition into next movements*

  Since the system is unable to impress at the previous benchmark, by the cumulative definition of our benchmarks it also cannot satisfy this third benchmark.

7.2 Technical Requirements and System Design

7.2.1 Real-time Performance

- *The system plays music in real-time*

  The bottleneck in real-time performance of our system lies in prediction of new units at the boundary between each unit.

  We can measure this bottleneck by measuring the time taken for a single unit prediction, and include all types of models for the sake of comparison. To get an accurate estimate, we perform each measurement as the "wall clock time" on a standard laptop with 1000 repetitions and obtain the average. The measurements are shown in Table 7.1.

Model	Unit Prediction Time (s)
End-to-end Convolutional RNN	0.0190
End-to-end RNN Encoder-Decoder	0.1576
VAE (Decoder Reconstruction)	0.0066
VAE (Unit Selection)	0.0464
Encoder : Latent RNN : Decoder	0.0112

TABLE 7.1: Time taken to predict a single unit for different accompaniment models. Comper's system currently uses the RNN Encoder-Decoder and the VAE (Decoder Reconstruction) models for its accompaniments; prediction delay with the VAE is negligible, however the RNN Encoder-Decoder's prediction time of 160ms causes a slightly perceptible delay during live testing.

  The results show that for Comper's two interaction modes, Call-and-Response should have no perceptible lag but Accompaniment mode has on average a 160ms pause between every 4-beat measure. In future versions, it is possible to remove the delay completely by starting the prediction slightly before the end of each unit, so that the next unit will be ready by the start of the next measure.

- *The system responds to current information in real-time with minimal lag from perception to reaction*

  As discussed in 5.2.4, the online accompaniment system perceives information in real-time and directly records this information. However, there is a lag

between perception and reaction which lasts for 2 beats on average (the system only responds to new information after the current 4-beat unit has ended, which on average will happen 2 beats later).

- *The system should never be late for band practice*

  Yes.

7.2.2 System Prerequisites

- *The system does not require any custom hardware or high-cost equipment*

  Yes. The equipment required for this system are standard for a typical digital musician: Ordinary laptop and a MIDI instrument.

- *The system can be installed on ordinary systems*

  The system is built with fully cross-platform components; however, it has currently only been tested on and developed for a Debian Linux (Ubuntu) environment, so porting to other operating systems may require extra work.

7.2.3 Interactions and Interface

- *Users do not require any technical knowledge to operate the system*

  Currently, the installation process requires working knowledge of installing packages in Linux environments. Beyond that, running the program simply requires following the documentation to start the program from a command line. Actual operation of the system after it has started does not require any technical knowledge.

- *The interaction the between the system and the user should feel natural and intuitive to the user*

  Without going into user trials, a basic observation is that the interaction with the system is similar to playing music with a metronome, since the user needs to follow the pianoroll beats.

- *The system encourages a synergistic collaboration which encourages new forms of music composition or performance*

  The system actually encourages a new form of music collaboration through the call-and-response interaction mode. This type of interaction happens in ordinary music as well, but the environment of performing call-and-response with Comper is different from human-to-human interactions, which may be an interesting opportunity for creativity.

Chapter 8

Evaluation

8.1 Summary and Evaluation

An ongoing struggle throughout this project was the uncertainty in addressing a problem in relatively unconquered territory. At this time of writing, there is still no other work which has demonstrated success in live music accompaniment by an AI at the level which this project aims for. To figure out the best approach to take, we worked through many options from first principles to evaluate their potential.

During initial exploration (See Appendix B) we looked at accompaniment models like Markov Chains and saw how they were able to produce realistic-sounding output by regurgitating previous inputs, but were difficult to control or use for producing original new ideas.

Neural network models seemed to hold more promise in their approach, especially given recent success in related work (eg. algorithmic composition) and adjacent fields (eg. natural language processing). This led to a decision to explore deep learning models for music accompaniment; however once again it was unclear what type of model would work best so we took a broad approach, trying different problem formulations and models to solve them.

Specifically, we started with different ways to frame the problem in terms of data representation, which in itself already did not have a clear winner. We looked at different kinds of models: We saw how end-to-end models were difficult to train and impossible to piece apart for troubleshooting; nevertheless, we were able to design an end-to-end RNN Encoder-Decoder which produces reasonable musical accompaniments alongside input musical units.

We also looked at latent spaces as a potentially interesting avenue for new types of interactions and exploration in music. The use of latent variables for sequence prediction ultimately did not produce satisfying accompaniments for our validation dataset, but the intermediate results of latent spaces were interesting enough to suggest a different type of human-machine interaction: musical call-and-response.

On the user interface side of things, the software platform developed for handling multiple composer classes enables reuse with different composer models with ease and flexibility. The Comper user interface as well is easy to use, though the user interface design was not developed to its full potential in this prototype since that is not the focus of the project. Furthermore, the design of the system as well and the chosen accompaniment models are seen to be able to produce accompaniments to musical inputs in real-time. Overall, the design of the Comper platform fulfills its requirements.

Aside from the system design itself, the test procedure used for evaluating Comper could be better-developed with more time. Currently the system is tested with live input from a human musician, which satisfies experimental validity (the test closely resembles real-life use) but lacks repeatability since each play-through will

have differences in quality. Future iterations could have a recorded input fed into the system for improved repeatability.

Finally, the metrics used for evaluating music quality from test results (Pitch Class Histogram Distance, Rhythm Loss) are sufficient to measure basic musical properties, but going forward in the field there will be a need to develop more sophisticated methods of evaluating musical quality in a more comprehensive way.

The overall objective and scope of the project has been ambitious for the provided time frame; this diminished the depth of analysis in each covered topic but the overall problem has been tackled with creative solutions from beginning to end. Ultimately, the project was successful in producing a real-time musical accompaniment system, albeit with some limitations in the resulting system.

8.2 Limitations

The first, most obvious limitation of our approach is that it relies on a fixed tempo. This limitation tied to the chosen representation using fixed-tick pianorolls. Given a fixed-tick representation and also fixed units, it is very difficult to change the model to be adaptive to changes in tempo.

Furthermore, the onsets-only representation which was shown to produce the best results in our experiments is itself a particularly limiting data representation, since it uses binarized velocities and cannot explicitly model note-release events.

We note that in general, the current data representations we have explored in the course of this project do not appear to yield the intuitiveness and flexibility of humans' own understanding of music - perhaps a different representation of music information will be necessary to overcome the current limitations.

Further out, the choice of symbolic music itself is a limiting factor; the choice of using symbolic music and the MIDI format places constraints on the pitch frequency, timbre and expression of the output.

Finally, looking at the broader task of accompaniment and music collaboration, we note that the current approach still only tackles a small subset of the problem. Our focus on interaction through purely musical content ignores a large amount of contextual cues that are typically present in human collaboration; this is a much larger problem scope that our approach does not even consider.

Chapter 9

Conclusions and Further Work

9.1 Further Work

The majority of Comper has been designed with future extension in mind.

The modularity of the composers program allows for new accompaniment models to be trained and plugged in, which would be useful when models with better accompaniment qualities are developed. The number of possibilities when it comes to model variations are endless: we can dig much deeper into each approach we have seen so far (for example, there is still a lot of potential and further testing to be done with latent spaces), as well as branch out into new models using completely different ideas.

On the front end, the use of a web interface for Comper's front-end was an intentional choice leading into a possible fully portable, web-based version of Comper. The current implementation has high overhead in setup including installation, but also in running the server, client and routing MIDI connections manually. It is conceivable that with further development, we might be able to move into a fully client-side web application, so that interested users would not need to install anything, but can just head to the Comper website and use the application online. This is possible because of new tools like Tensorflow.js which provides support for running deep learning models directly in the browser, and also the fact that most major browsers now also provide direct access to MIDI hardware through native support of the Web MIDI API (*Web MIDI API*) or browser extensions.

Finally, two main design principles driving software development of this project are portability and reusability. Where possible, each software node has been developed using cross-platform languages and open-source libraries with trivial licensing, and its interfaces and architectures are designed such that the nodes might be useful in other contexts. The entirety of the project has been made available online through the two repositories on GitHub (`https://github.com/JunShern/comper` and `https://github.com/JunShern/comper-ui`), where some documentation exists and will be extended even after this project ends. The hope is that the work produced by this project will provide a useful jumping-off point for other researchers in the field, since progress in this area is still young and there is a lot of future work to cover.

9.2 Conclusion

Based on the limitations of the current approach and other approaches we have seen so far, it is likely that any accompaniment systems that truly reach the flexibility and reactiveness of human musicians are a long way off. The technologies we have seen all seek to solve one part of the problem, but the field of machine musicianship

is immense, and covers a large subset of human abilities which form part of the grander vision of strong AI (Kurzweil, 2010).

To quote Nicholas M. Collins in his thesis, *"I do not claim to cover the building of a complete artificial musician that can match a human musician in all their accomplishments and quirks. Such an undertaking would require addressing the same immense problems, of biological complexity, neural architecture, culture- specific knowledge and training, as have obstructed artificial intelligence research."* (Collins, 2007)

Nevertheless, this project has been successful in exploring and evaluating a large breadth of accompaniment strategies, and applied cutting-edge technologies in novel ways to create a minimal but functional prototype demonstrating live music accompaniment by an AI named Comper.

Appendix A

MIDI - Musical Instrument Digital Interface

The Musical Instrument Digital Interface (MIDI) specification defines a standard for digital representation, storage, and transfer of symbolic music information. Here, we describe only a summarized introduction to the standard with enough key points to grasp the role of MIDI in our project.[1] The MIDI specification includes the following:

- A specification for **MIDI messages**, which encodes information about musical content or peripheral performance states.
- A specification for **Standard MIDI Files (SMF)**, which store MIDI messages with additional metadata in the `.midi` format.
- **General MIDI (GM)** specifications, which detail playback information such as instrument mappings, to encourage consistent playback of messages across different implementations of MIDI synthesizers.

Additionally, operating systems and the Universal Serial Bus (USB) standards have been extended to support an API for **inter-application routing** of MIDI messages (supported across all major operating systems), and a specification for **USB-MIDI** devices which allows inter-device MIDI communication without the need for additional drivers

It is important to realize that on its own, MIDI messages do not contain any raw audio information. Hence, to obtain sound, MIDI messages must always be synthesized through a MIDI synthesizer, which knows how to synthesize the correct sounds by following the GM specification.

There exist many MIDI messages to specify various performance and musical states, but the most common MIDI messages are the *Note On* and *Note Off* event messages. Each *Note On* or *Note Off* message carries the following information:

1. *Channel Number* - The channel on which to broadcast the note, from 1-16.
2. *Note Number* - Corresponds to one of 127 possible musical notes across 10 different octaves. MIDI note number 60 is defined as "Middle C", and all other note numbers are relative. The full table of note numbers and their respective note names can be found at (*MIDI Note Numbers for Different Octaves*).
3. *Velocity* - The strength at which the note is sounded.
4. *Time* - Time interval (in seconds) between the previous note and the current note, also known as *deltatime*.

[1] Please refer to (Association, 1983) for the original specification and (*Specs*) for up-to-date documentation and tutorials.

Appendix B

Non-Deep Learning Models

During initial exploration, various "naive" accompaniment models were produced to provide insight into their potential

Ultimately, it was decided that these models produced results that were not original enough to be interesting, but they are documented here for comparison to the main approaches covered in the report.

Each of the models are produced as classes in Comper's composer program (See 5.2.4), which allows composer algorithms to be developed incrementally from 'dummy' composers to more sophisticated ones, each building upon functionalities of previous classes:

1. **RandomMemoryDurationless**

   This composer simply records the `note` (pitch) property of each message which has been played by the human player, and plays back a random sample from those notes at a fixed interval (every 0.5 seconds).

   Evaluation: Dummy composer, built as an exploratory step to confirm that the framework works as expected. Resulting compositions are monophonic - single notes are produced at a fixed rate with no sense of melody (but retains pitch range and key of the input as expected, since it samples notes directly from the input).

2. **RandomMemory**

   This composer extends `RandomMemoryDurationless` by sampling entire messages from the input history instead of just pitch. The messages are played back in immediate succession, with inter-note time intervals determined by the `deltatime` property of each note.

   Evaluation: Sampling entire messages produces a far better result than sampling only pitches. The key difference is due to having a notion of timing now, which not only provides a more interesting rhythm, but also switches the composer from monophonic to polyphonic output (since in MIDI there is no notion of chords; a chord is simply a sequence of NOTE-ON messages separated by small `deltatime` properties, such that the notes are all triggered at almost the same time).

3. **Arpeggiator**

   This composer exploits knowledge of currently-held notes, by playing back each of the currently-held notes in sequence at a fixed rate. For example, if the user holds a C-major triad chord `C-E-G`, Arpeggiator will produce a repeated pattern of notes in succession `C, E, G, C, E, G, C, E, G, ...` until the user releases those notes.

Evaluation: `Arpeggiator` is inspired by the arpeggiator feature found in many electronic music tools. It is not a real 'composer' in the sense that it simply warps the current input to create its output, but it is the first context-aware composer in our project since it actually produces an output based on what the user is currently playing.

4. **MarkovDurationless**

   `MarkovDurationless` is similar to `RandomMemoryDurationless`, in that it plays back a note from the user's history at a constant rate. However, the note is not (completely) randomly chosen this time, instead the composer builds a Markov chain: a table of states and transition states based on the sequence of input notes.

   For example, if the user plays a sequence `C, D, E, C, F, G, A, C` then given a starting note `C`, the MarkovDurationless composer will stochastically select a note which the user has previously transitioned from `C` to - in this case either `D` or `F`. The following note will then be chosen based on the historical next-states of the most recent composer-played note, and so on.

   Evaluation: This composer produces a marginally better output than a random sequence from `RandomMemory`. The sequence of notes produced tend to be more coherent since they follow (at an immediate-local level) the melodic structure of what the user has played before, but the lack of timing information means that the output is still monophonic and uninteresting.

5. **MarkovQuantizeDuration**

   `MarkovQuantizeDuration` extends `MarkovDurationless` by sampling a whole message instead of just the note. An additional quantization step was required to quantize each message's `deltatime`, since the exact recorded `deltatime` of each note is a near-continuous value (eg. 0.12342525... seconds), so the chances of there being an exact match in the recorded state transition table is vanishingly small. Quantization is applied by rounding each `deltatime` to 2 decimal places (eg. 0.12342525... to 0.12 seconds), and new notes are generated based on historical transitions as before.

   Evaluation: This composer produces the best output so far. Given consistent input of musical chords and notes, the composer is able to produce an output which sounds similar in style to the input sequence, which is to be expected.

   There remain several problems with this approach, however:

   - The system lacks long-term coherence, since it only follows sequences at a note-to-note transition level. This could be improved using N-grams (N-most recent message sequence) as states instead of individual messages.
   - The system plays from a very limited repertoire of knowledge. This is because the Markov model currently only trains on online input. To produce more 'intelligent' and diverse compositions, the algorithm should be extended to include an offline training phase where the model can be trained on a database of many MIDI songs.
   - The system has no notion of current player context, since the output generation is based on the composer's own notes and the entire history of the player input is treated equally. This leads to the composer generating a composition all on its own, which does not complement the player's current music.

Appendix C

Custom Loss Functions

All of the following functions depend on the Keras Backend library, which are import using: `from keras import backend as K`

```python
def get_pitch_class_histogram(pianorolls_batch):
    """
    Given a batch of pianoroll matrices, return a Tensor of shape (NUM_BATCHES, 12)
    with each 12 elements in a batch being the sum of velocities for that pitch class,
    normalized between zero and one.
    """
    pianorolls_batch_tensor = K.cast(K.squeeze(pianorolls_batch, -1), 'float32')
    # Get sum of velocities for each pitch class
    pitch_velocities = K.sum(pianorolls_batch_tensor, axis=2) # sum along tick axis
    pitch_class_velocities = K.sum(K.reshape(pitch_velocities, (-1, 8, 12)), axis=1) # Separated into octaves
    # Normalize against total note velocities in this pianoroll
    total_velocities = K.sum(pitch_class_velocities, axis=1, keepdims=True)
    norm_pitch_class_velocities = pitch_class_velocities / K.clip(total_velocities, K.epsilon(), None) # Protect against 0-division
    return norm_pitch_class_velocities

def pitch_histogram_distance(pianorolls_batch_1, pianorolls_batch_2, distance_metric='cosine'):
    """
    Calculates the pitch class histograms for two batches of pianoroll matrices,
    then using cosine proximity to calculate the distance between histograms.
    Returns a distance value between 0 and 1, where 0 is maximum similarity and
    1 is completely orthogonal.
    """
    hist1 = get_pitch_class_histogram(pianorolls_batch_1)
    hist2 = get_pitch_class_histogram(pianorolls_batch_2)
    if distance_metric == 'cosine':
        distance = 1 + losses.cosine_proximity(hist1, hist2)
    elif distance_metric == 'mse':
        distance = losses.mean_squared_error(hist1, hist2)
    return distance
```

LISTING C.1: Functions to calculate the pitch class histogram distance between two batches of pianoroll matrices

```python
def onset_distance(pianorolls_batch_1, pianorolls_batch_2):
    """
    Given two batches of pianoroll matrices, return the difference
    between their note onset matrices.
    """
    onsets_1 = get_note_onsets_time_only(pianorolls_batch_1)
    onsets_2 = get_note_onsets_time_only(pianorolls_batch_2)
    return losses.mean_squared_error(onsets_1, onsets_2)

def get_note_onsets_time_only(pianorolls_batch):
    """
    Given a batch of pianorolls (NUM_BATCHES, NUM_PITCHES, NUM_TICKS, 1),
    return a matrix of shape (NUM_BATCHES, NUM_TICKS), containing the
    total onset velocities at each tick (if no onsets, value will be 0).
    """
    pianorolls_batch = K.cast(pianorolls_batch, 'float32')
    # Pad along time axis
    padded = K.spatial_2d_padding(pianorolls_batch, padding=((0, 0), (1, 0)), data_format="channels_last")
    # Remove channel axis
    padded_no_channel = K.squeeze(padded, 3)
    # Take difference along time axis
    diff = padded_no_channel[:,:,1:] - padded_no_channel[:,:,:-1]
    clipped_diff = K.clip(diff, 0, None) # Ignore negative (note-off) values
    # Sum along pitch axis
    time_only = K.sum(clipped_diff, axis=1)
    return time_only
```

LISTING C.2: Functions to calculate the onsets distance between two batches of pianoroll matrices

```
def smoothness_loss(pianorolls_batch):
    EPSILON = 0.01
    # Remove channel axis
    pianorolls_batch = K.squeeze(pianorolls_batch, axis=3)
    # Take difference along time axis
    diff = pianorolls_batch[:,:,1:] - pianorolls_batch[:,:,:-1]
    clipped_diff = K.clip(diff, 0, EPSILON)
    return K.mean(clipped_diff)
```

LISTING C.3: Function to calculate the smoothness loss for a batch of pianoroll matrices

```
def sampling(args):
    # This function is used to sample a latent variable z during model training
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim), mean=0., stddev=1.0)
    return z_mean + K.exp(z_log_var / 2) * epsilon

def kl_loss(input_x, output_x):
    kl_loss = -0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
    return kl_loss
```

LISTING C.4: Functions to calculate the Kullback-Leibler Divergence of a latent variable z against the Standard Gaussian Distribution.

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Arzt, Andreas, Gerhard Widmer, and Simon Dixon (2012). "Adaptive distance normalization for real-time music tracking". In: *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European*. IEEE, pp. 2689–2693.

Association, International MIDI et al. (1983). "MIDI musical instrument digital interface specification 1.0". In: *Los Angeles*.

Association, MIDI. *Specs*. https://www.midi.org/specifications. Accessed: 2018-01-27.

Bello, Juan Pablo et al. (2005). "A tutorial on onset detection in music signals". In: *IEEE Transactions on speech and audio processing* 13.5, pp. 1035–1047.

Biles, John A (1994). "GenJam: A genetic algorithm for generating jazz solos". In:

Böck, Sebastian, Florian Krebs, and Gerhard Widmer (2014). "A Multi-model Approach to Beat Tracking Considering Heterogeneous Music Styles." In:

— (2015). "Accurate Tempo Estimation Based on Recurrent Neural Networks and Resonating Comb Filters." In: *ISMIR*, pp. 625–631.

Boulanger-Lewandowski, Nicolas, Yoshua Bengio, and Pascal Vincent (2012). "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription". In: *arXiv preprint arXiv:1206.6392*.

Bretan, Mason and Gil Weinberg. "Chronicles of a Robotic Musical Companion." In:

— (2016). "A survey of robotic musicianship". In: *Communications of the ACM* 59.5, pp. 100–109.

Bretan, Mason, Gil Weinberg, and Larry Heck (2016). "A Unit Selection Methodology for Music Generation Using Deep Neural Networks". In: *arXiv preprint arXiv:1612.03789*.

Bretan, Mason et al. (2016). "A Robotic Prosthesis for an Amputee Drummer". In: *arXiv preprint arXiv:1612.04391*.

Bretan, Mason et al. (2017a). "Deep Music: Towards Musical Dialogue." In: *AAAI*, pp. 5081–5082.

Bretan, Mason et al. (2017b). "Learning and Evaluating Musical Features with Deep Autoencoders". In: *arXiv preprint arXiv:1706.04486*.

Bretan, Peter Mason (2017). "Towards An Embodied Musical Mind: Generative Algorithms for Robotic Musicians". PhD thesis. Georgia Institute of Technology.

Build software better, together. URL: https://github.com/.

Chollet, Francois. *Keras: The Python Deep Learning library*. URL: https://keras.io/.

— (2016). *Building Autoencoders in Keras*. URL: https://blog.keras.io/building-autoencoders-in-keras.html.

— (2017). *A ten-minute introduction to sequence-to-sequence learning in Keras*. URL: https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html.

Christensen, Thomas (2006). *The Cambridge history of Western music theory*. Cambridge University Press.

Cicconet, Marcelo, Mason Bretan, and Gil Weinberg (2012). "Visual cues-based anticipation for percussionist-robot interaction". In: *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*. ACM, pp. 117–118.

Collins, Nicholas M (2007). "Towards autonomous agents for live computer music: Realtime machine listening and interactive music systems". PhD thesis. University of Cambridge.

Dauphin, Yann N et al. (2014). "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in neural information processing systems*, pp. 2933–2941.

Derrien, Olivier (2014). "A very low latency pitch tracker for audio to MIDI conversion". In: *17th International Conference on Digital Audio Effects (DAFx-14)*.

Dixon, Simon (2001). "Automatic extraction of tempo and beat from expressive performances". In: *Journal of New Music Research* 30.1, pp. 39–58.

Doersch, Carl (2016). "Tutorial on variational autoencoders". In: *arXiv preprint arXiv:1606.05908*.

Dong, Hao-Wen et al. (2017). "MuseGAN: Symbolic-domain music generation and accompaniment with multi-track sequential generative adversarial networks". In: *arXiv preprint arXiv:1709.06298*.

Dong, Hao-Wen et al. (2018). *Lakh Pianoroll Dataset*. URL: https://salu133445.github.io/lakh-pianoroll-dataset/.

Dumoulin, Vincent and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *ArXiv e-prints*. eprint: 1603.07285.

Engel, Jesse et al. (2017). "Neural audio synthesis of musical notes with wavenet autoencoders". In: *arXiv preprint arXiv:1704.01279*.

FluidSynth. *FluidSynth*. URL: http://www.fluidsynth.org/.

Foote, Jonathan and Matthew L Cooper (2001). "Visualizing Musical Structure and Rhythm via Self-Similarity." In:

Gillick, Jon, Kevin Tang, and Robert M Keller (2010). "Machine learning of jazz grammars". In: *Computer Music Journal* 34.3, pp. 56–66.

Google Cloud Computing, Hosting Services and API. URL: https://cloud.google.com/.

Graves, Alex (2013). "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850*.

Ha, David (2017). *Teaching Machines to Draw*. URL: https://ai.googleblog.com/2017/04/teaching-machines-to-draw.html.

Ha, David and Douglas Eck (2017). "A neural representation of sketch drawings". In: *arXiv preprint arXiv:1704.03477*.

HDF Home (2018). URL: https://www.hdfgroup.org/.

HDF5 for Python. URL: https://www.h5py.org/.

Hedges, Stephen A (1978). "Dice music in the eighteenth century". In: *Music & Letters* 59.2, pp. 180–187.

Hoffman, Guy (2010). "Anticipation in Human-Robot Interaction." In:

Hoffman, Guy and Gil Weinberg (2010a). "Gesture-based human-robot jazz improvisation". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, pp. 582–587.

— (2010b). "Synchronization in human-robot musicianship". In: *RO-MAN, 2010 IEEE*. IEEE, pp. 718–724.

Hsü, Kenneth Jinghwa and Andreas J Hsü (1990). "Fractal geometry of music." In: *Proceedings of the National Academy of Sciences* 87.3, pp. 938–941.

Hsü, Kenneth Jinghwa and Andrew Hsü (1991). "Self-similarity of the" 1/f noise" called music." In: *Proceedings of the National Academy of Sciences* 88.8, pp. 3507–3509.

JACK Audio Connection Kit. http://www.jackaudio.org/. Accessed: 2018-01-27.

Jacob, Bruce (1995). "Composing with genetic algorithms". In:

Karpathy, Andrej (2015). *The Unreasonable Effectiveness of Recurrent Neural Networks*. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Kidd, Cory D and Cynthia Breazeal. "Effect of a robot on user perceptions". In: *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*. Vol. 4. IEEE, pp. 3559–3564.

Kitani, Kris Makoto and Hideki Koike (2010). "ImprovGenerator: Online Grammatical Induction for On-the-Fly Improvisation Accompaniment." In:

Krumhansl, Carol L (2001). *Cognitive foundations of musical pitch*. Oxford University Press.

Kurzweil, Ray (2010). *The singularity is near*. Gerald Duckworth & Co.

Lee, Kyogu (2006). "Automatic Chord Recognition from Audio Using Enhanced Pitch Class Profile." In:

Magenta. https://magenta.tensorflow.org/. Accessed: 2018-01-27.

Malik, Iman and Carl Henrik Ek (2017). "Neural translation of musical style". In: *arXiv preprint arXiv:1708.03535*.

Mann, Yotam. *AI Duet by Yotam Mann*. https://experiments.withgoogle.com/ai/ai-duet. Accessed: 2018-01-27.

Matplotlib. URL: https://matplotlib.org/.

McCormack, Jon (1996). "Grammar based music composition". In: *Complex systems* 96, pp. 321–336.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4.

Michalowski, Marek P, Selma Sabanovic, and Hideki Kozima (2007). "A dancing robot for rhythmic social interaction". In: *Human-Robot Interaction (HRI), 2007 2nd ACM/IEEE International Conference on*. IEEE, pp. 89–96.

MIDI Note Numbers for Different Octaves. http://www.electronics.dit.ie/staff/tscarff/Music_technology/midi/midi_note_numbers_for_octaves.htm. Accessed: 2018-01-27.

Mido - MIDI objects for Python. https://mido.readthedocs.io/. Accessed: 2018-01-27.

Newton, Isaac (1675). *Isaac Newton letter to Robert Hooke*.

Nikolaidis, Ryan and Gil Weinberg (2010). "Playing with the masters: A model for improvisatory musical interaction between robots and humans". In: *RO-MAN, 2010 IEEE*. IEEE, pp. 712–717.

Nisbet, Albert and Richard Green (2017). "Capture of Dynamic Piano Performance with Depth Vision". In:

Olah, Christopher (2015). *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Özcan, Ender and Türker Erçal (2007). "A genetic algorithm for generating improvised music". In: *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, pp. 266–277.

Pachet, Francois (2003). "The continuator: Musical interaction with style". In: *Journal of New Music Research* 32.3, pp. 333–341.

Pachet, François and Pierre Roy (2011). "Markov constraints: steerable generation of Markov sequences". In: *Constraints* 16.2, pp. 148–172.

— (2014). "Imitative leadsheet generation with user constraints". In: *Proceedings of the Twenty-first European Conference on Artificial Intelligence*. IOS Press, pp. 1077–1078.

Pan, Ye, Min-Gyu Kim, and Kenji Suzuki (2010). "A Robot Musician Interacting with a Human Partner through Initiative Exchange." In: *NIME*, pp. 166–169.

pip. URL: https://pypi.org/project/pip/.
Project Jupyter (2018). URL: http://jupyter.org/.
QjackCtl: JACK Audio Connection Kit - Qt GUI Interface. https://qjackctl.sourceforge.io/. Accessed: 2018-01-27.
Raffel, Colin (2016a). *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching*. Columbia University.
Raffel, Collin (2016b). *The Lakh MIDI Dataset v0.1*. URL: http://colinraffel.com/projects/lmd/.
Rafii, Zafar (2012). *Rhythm Analysis in Music*. URL: http://www.cs.northwestern.edu/~pardo/courses/eecs352/lectures/MPM12-rhythm.pdf.
Records, Flow (2017). *hello world: the first album composed with ai*. URL: https://www.helloworldalbum.net/.
Roberts, Adam et al. (2018a). "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music". In: eprint: arXiv:1803.05428.
Roberts, Adam et al. (2018b). "Learning Latent Representations of Music to Generate Interactive Musical Palettes". In:
Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088, pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.
Sarabia, Miguel, Kyuhwa Lee, and Yiannis Demiris (2015). "Towards a synchronised Grammars framework for adaptive musical human-robot collaboration". In: *Robot and Human Interactive Communication (RO-MAN), 2015 24th IEEE International Symposium on*. IEEE, pp. 715–721.
Schedl, Markus, Emilia Gómez, Julián Urbano, et al. (2014). "Music information retrieval: Recent developments and applications". In: *Foundations and Trends® in Information Retrieval* 8.2-3, pp. 127–261.
Signature MIDI Collection (2013). URL: http://www.yamahaden.com/midi-files.
Simon, Ian and Sageev Oore (2017). *Performance RNN: Generating Music with Expressive Timing and Dynamics*. https://magenta.tensorflow.org/performance-rnn. Blog.
Simon, Ian et al. (2018). *Learning a Latent Space of Multitrack Measures*. eprint: arXiv:1806.00195.
Simoni, Mary (2003). "Algorithmic composition: a gentle introduction to music composition using common LISP and common music". In: *SPO Scholarly Monograph Series*.
TiMidity++. URL: http://timidity.sourceforge.net/.
Transferring Files to Instances | Compute Engine Documentation | Google Cloud (2018). URL: https://cloud.google.com/compute/docs/instances/transfer-files#filebrowser.
w3c. *Web MIDI API*. https://www.w3.org/TR/webmidi/. Accessed: 2018-01-27.
Weisstein, Eric W. *Chain Rule*. URL: http://mathworld.wolfram.com/ChainRule.html.
— *Chain Rule*. URL: http://mathworld.wolfram.com/Convolution.html.
— *Chain Rule*. URL: http://mathworld.wolfram.com/Cross-Correlation.html.
— *Differentiable*. URL: http://mathworld.wolfram.com/Differentiable.html.
— *Kullback-Leibler Distance*. URL: http://mathworld.wolfram.com/Kullback-LeiblerDistance.html.
— *Least Squares Fitting*. URL: http://mathworld.wolfram.com/LeastSquaresFitting.html.
Welcome to Python.org. URL: https://www.python.org/.
Yang, Xitong (2017). "Understanding the Variational Lower Bound". In: