5-4-2020

A Category-Theoretic Compositional Framework of Perceptron-Based Neural Networks plus an Architecture for Modeling Sequences Conditioned to Time-Structured Context: An Implementation of a Generative Model of Jazz Solo Improvisations

Rodrigo Ivan Castro Lopez Vaal

A CATEGORY-THEORETIC COMPOSITIONAL FRAMEWORK OF PERCEPTRON-BASED NEURAL NETWORKS PLUS AN ARCHITECTURE FOR MODELING SEQUENCES CONDITIONED TO TIME-STRUCTURED CONTEXT: AN IMPLEMENTATION OF A GENERATIVE MODEL OF JAZZ SOLO IMPROVISATIONS

by

RODRIGO CASTRO LOPEZ VAAL

Under the Direction of Mariana Montiel, PhD

ABSTRACT

This work introduces an algebraic graphical language of perceptrons, multilayer perceptrons, recurrent neural networks, and long short-term memory neural networks, via string diagrams of a suitable hypergraph category equipped with a concatenation diagram operation by means of a monoidal endofunctor. Using this language, we introduce a neural network architecture for modeling sequential data in which each sequence is subject to a specific context with a temporal structure, that is, each data point of a sequence is conditioned to a different past, present, and future context than the other points. As proof of concept, this architecture is implemented as a generative model of jazz solo improvisations.

INDEX WORDS: Deep learning, Neural networks, Category theory, String diagrams, Long short-term memory, Jazz solo generation.

A CATEGORY-THEORETIC COMPOSITIONAL FRAMEWORK OF PERCEPTRON-BASED NEURAL NETWORKS PLUS AN ARCHITECTURE FOR MODELING SEQUENCES CONDITIONED TO TIME-STRUCTURED CONTEXT: AN IMPLEMENTATION OF A GENERATIVE MODEL OF JAZZ SOLO IMPROVISATIONS

by

RODRIGO CASTRO LOPEZ VAAL

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2020

A CATEGORY-THEORETIC COMPOSITIONAL FRAMEWORK OF PERCEPTRON-BASED NEURAL NETWORKS PLUS AN ARCHITECTURE FOR MODELING SEQUENCES CONDITIONED TO TIME-STRUCTURED CONTEXT: AN IMPLEMENTATION OF A GENERATIVE MODEL OF JAZZ SOLO IMPROVISATIONS

by

RODRIGO CASTRO LOPEZ VAAL

Committee Chair: Mariana Montiel

Committee: Vladimir Bondarenko

Marina Arav

Martin Norgaard

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2020

DEDICATION

To my parents, Blanca and Felipe.

Everything you have done in your lives reverberates in mine in form of freedom, beauty, happiness, and immeasurable privilege. I am forever thankful.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my supervisor, Professor Mariana Montiel, without whom this journey would not have been possible in the first place. Thank you for your constant support and advice, and for believing in me throughout these years.

My appreciation also extends to my friends, Irina Ilioaea and Ben Sirb, from whom I have learned immensely, and to my peers Caroline Parnass, Shushan He, Husneara Rahman, Siang Ng, Guangming Jing, Aurelie Akossi, and Adam Kellam, for all those shared moments of joy and strugle during our mathematical pilgrimage.

I wish to extend my sincere thanks to the members of my committee, Professor Martin Norgaard, Professor Marina Arav, and Professor Vladimir Bondarenko, for setting aside time to review my work providing useful corrections, and for their flexibility during these times of global crisis. I am also grateful to Professor Jonathan Shihao Ji, Dr. Ulises Moya, Dr. James Fairbanks, and Evan Patterson for their generous and helpful suggestions.

Thanks should also go to Angel Gonzalez, John Phillips (R.I.P.), Sara Gregory, and Juan Carlos Martinez, for their greathearted assistance during my time in the US. This journey would have been much harder without their help.

Finally, I cannot begin to express my thanks to my parents, Blanca Lopez Vaal and Felipe Castro. Papás, I would not be who I am if it had not been for the freedom of thought you bestowed on me. Thank you for your unconditional love, support, and trust.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

- CT: Category Theory
- DL: Deep Learning
- FC: Fully Connected
- GSU: Georgia State University
- LSTM: Long Short-Term Memory
- NN: Neural Network
- RNN: Recurrent Neural Network

CHAPTER 1
INTRODUCTION

The goal of this dissertation is threefold: to present a formal diagrammatic language of neural networks, to introduce a neural network architecture to model sequences subject to time-structured context, and to provide an implementation of such an architecture as a jazz solo generative model.

In order to build a formal diagrammatic language that allows us to rigorously describe neural network architectures, we turn to the use of string diagrams (a.k.a. wiring diagrams), a graphical language for monoidal categories [26]. In recent years, category theory has seen a surge of applications framed in the context of monoidal categories (for example, see [1, 3]). Broadly speaking, one important strength of monoidal categories is that they naturally model compositionality, that is, smaller structures being composed together to produce larger structures. Strings diagrams provide a graphical framework that enables us to describe the composition of structures in a way that is both precise and mathematically justified. In chapter 3, we construct a monoidal category suited to model the morphisms that constitute different types of neural networks (except convolutional), and use their associated string diagram representations to describe some of the most fundamental neural network architectures. For the reader interested in the mathematical foundations of monoidal categories and string diagrams, in chapter 2 we provide the concepts and definitions that are (strictly) necessary to construct our desired category in chapter 3.

Along with constructing the diagrams of different neural networks, we also provide in-

troductory concepts that intend to give a general idea and motivation of neural networks to the reader unfamiliar with deep learning. A particular emphasis is given to familiarising the reader with long short-term memory (LSTM) neural networks.

Once we have presented LSTMs and built a graphical notation to describe neural network architectures, in chapter 4 we use these tools to introduce a novel neural network architecture that aims to model sequences subject to a context that has a structure in time. Long short-term memory networks have proved to be an effective tool when dealing with sequential data [28] and the architecture introduced in this manuscript makes use of LSTM networks to model data that consists of two aspects: a sequence of signals in which each signal has a certain duration in time, and a sequence of events that conditions the sequence of signals. The sequence of events is regarded as a time-structured context to which the sequence of signals is subject. In other words, the sequential structure of the signals is correlated to the sequential and time structure of the events. Our architecture aims to implicitly model such a correlation. As proof of concept, in chapter 5, we present an implementation of this architecture as a generative model of jazz solo improvisations.

Deep learning applications to music are vast [7], and the use of LSTMs to generate jazz improvisations is not new [10, 15]. The main purpose of our implementation on jazz music is to demonstrate that the architecture proposed in chapter 4 is trainable (the loss function that compares a prediction with its ground truth can be minimized smoothly and consistently), and that new sequences in the style of the corpus can be generated. However, we also believe that the architecture would be a strong first step towards developing a thorough

musical AI project, since its novel time-dependent connection mechanism accounts for the conditioning of a solo phrase to a chord progression. The desired learning task of our model is to take a chord progression (chosen by the user) and generate a jazz phrase that "works well" harmonically with the progression, and that the generated phrase resembles the stylistic patterns and note choices seen in the training corpus. In the appendix of this manuscript we provide a complete Python implementation of this model using the deep-learning framework PyTorch.

Finally, in chapter 6, we discuss the potential reach of our contributions, and possible lines of research that could be carried out further.

CHAPTER 2
A GRAPHICAL LANGUAGE OF MONOIDAL CATEGORIES

2.1 Preliminaries

In order to understand the formal background in which the diagrams used in this work are rooted, namely, string diagrams of monoidal categories, we first need to review the basic definitions of a category, functor, and natural transformation. For a more in-depth and detailed introduction see [22, 27].

Definition 2.1.1. A **category** $\mathcal{C}$ consists of

- a collection $\mathcal{C}_0$ of objects;
- a collection $\mathcal{C}_1$ of arrows (or morphisms);
- two assignments $s, t : \mathcal{C}_1 \to \mathcal{C}_0$ which attach two objects $s(f), t(f) \in \mathcal{C}_0$ to an arrow $f \in \mathcal{C}_1$ to specify its *source* (domain) and *target* (codomain) respectively;
- a partial composition $\mathcal{C}_1 \times \mathcal{C}_1 \to \mathcal{C}_1$ which assigns, to any pair of arrows $f, g \in \mathcal{C}_1$ such that $t(f) = s(g)$, their sequential composite arrow $f; g$ (traditionally written $g \circ f$);
- an assignment $\mathrm{id}_{\_} : \mathcal{C}_0 \to \mathcal{C}_1$ which assigns to each object A the identity arrow $\mathrm{id}_A : A \to A$;

such that the following properties are satisfied:

(i) source and target are respected by composition: $s(f; g) = s(f)$, $t(f; g) = t(g)$;

(ii) source and target are respected by identities: $s(\mathrm{id}_A) = A$, $t(\mathrm{id}_A) = A$;

(*iii*) composition is associative: $(f;g);h = f;(g;h)$ whenever $t(f) = s(g)$ and $t(g) = s(h)$;

(*iv*) composition satisfies the left and right unit laws: if $s(f) = A$ and $t(f) = B$, then $f;\mathrm{id}_B = f = \mathrm{id}_A;f$.

We will often write $A \in \mathcal{C}$ instead of $A \in \mathcal{C}_0$ when it is clear from context that we are talking about an object. Also, we will usually write $f : A \to B$ or $A \xrightarrow{f} B$ for an arrow $f \in \mathcal{C}_1$ to state that $s(f) = A$ and $t(f) = B$.

Definition 2.1.2. For categories $\mathcal{C}$ and $\mathcal{D}$, a **functor** $F : \mathcal{C} \to \mathcal{D}$ is a category morphism sending each object $A \in \mathcal{C}$ to an object $F(A) \in \mathcal{D}$, and each arrow $f : A \to B$ of $\mathcal{C}$ to an arrow $F(f) : F(A) \to F(B)$ of $\mathcal{D}$, such that

(*i*) F preserves composition: $F(f;g) = F(f);F(g)$ whenever the left side is well defined;

(*ii*) F preserves identity morphisms: for each object $A \in \mathcal{C}$, $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$.

Definition 2.1.3. Given categories $\mathcal{C}, \mathcal{D}$ and functors $F : \mathcal{C} \to \mathcal{D}$, $G : \mathcal{C} \to \mathcal{D}$, a **natural transformation** $\tau : F \to G$ consists of a $\mathcal{D}$-morphism $\tau_A : F(A) \to G(A)$ for every object $A \in \mathcal{C}$ such that, for every arrow $A \xrightarrow{f} B$ of $\mathcal{C}$, the following diagram commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{\tau_A} & G(A) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow{\tau_B} & G(B) \end{array} .$$

A **natural isomorphism** is a natural transformation each of whose morphisms τ_A, τ_B in the diagram above are isomorphisms in $\mathcal{D}$.

Definition 2.1.4. Given categories $\mathcal{C}$ and $\mathcal{D}$, the **product category** $\mathcal{C} \times \mathcal{D}$ is the category whose objects are all ordered pairs (C, D) with $C \in \mathcal{C}_0$, $D \in \mathcal{D}_0$, and in which an arrow $(f, g) : (C, D) \to (C', D')$ of $\mathcal{C} \times \mathcal{D}$ is a pair of arrows $f : C \to C'$ of $\mathcal{C}$, and $g : D \to D'$ of $\mathcal{D}$. The identity of an object (C, D) is $(\mathrm{id}_C, \mathrm{id}_D)$. If $(f', g') : (C', D') \to (C'', D'')$ is another arrow of $\mathcal{C} \times \mathcal{D}$, the composition $(f, g); (f', g')$ is defined as

$$(f, g); (f', g') = (f; f', g; g') : (C, D) \to (C'', D'') \ .$$

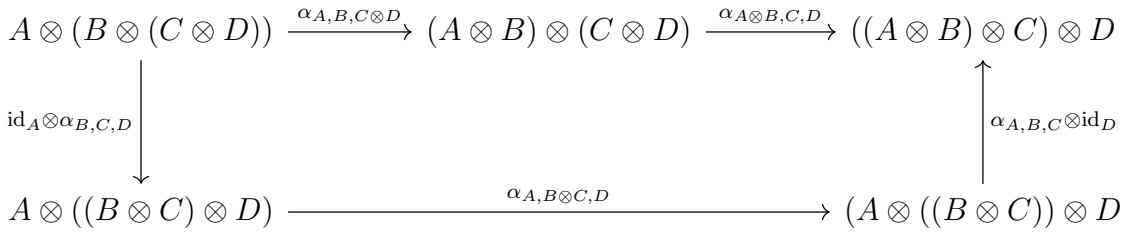
Definition 2.1.5. A **terminal object** in a category $\mathcal{C}$ is an object 1 satisfying the universal property that for every other object $A \in \mathcal{C}$ there exists one and only one morphism $A \xrightarrow{!} 1$. The terminal object of any category, if it exists, is unique up to unique isomorphism.

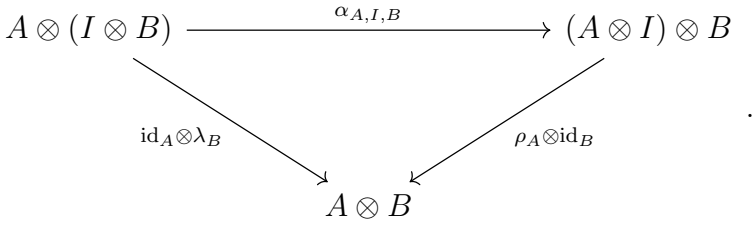
Definition 2.1.6. If a category $\mathcal{C}$ has a terminal object 1, a **global element** x of an object $A \in \mathcal{C}$ is a morphism $x : 1 \to A$.

When the category has sets for objects and set functions for arrows, any sigleton $\{*\}$ is a terminal object 1, and $\mathrm{Hom}(1, A)$ is in a one-to-one correspondence with the elements of a set A. Hence the name global *element*.

2.2 Monoidal Categories

Definition 2.2.1. A **monoidal category** consists of a category $\mathcal{C}$, a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, a distinguished object $I \in \mathcal{C}$, and natural isomorphisms $\alpha_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C$, $\lambda_A : I \otimes A \to A$, $\rho_A : A \otimes I \to A$, such that $\lambda_I = \rho_I$ and the following diagrams commute:





.

The bifunctor $\otimes$, often referred to as the **monoidal product**, assigns to each pair of objects $A, B \in \mathcal{C}$ an object $A \otimes B$ of $\mathcal{C}$, and to each pair of arrows $f : A \to C$, $g : B \to D$ an arrow $f \otimes g : A \otimes B \to C \otimes D$ of $\mathcal{C}$. We call I the **monoidal unit**, $\alpha = \alpha_{\_,\_,\_}$ the **associator**, and λ, ρ the **left** and **right unitor** respectively. For simplicity, we shall denote the monoidal category $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$ just by $(\mathcal{C}, \otimes, I)$.

A **strict monoidal category** is a monoidal category in which the natural isomorphisms α, λ, ρ are all identity morphisms. We will assume all monoidal categories in this work to be strict, so we may drop the parentheses sometimes and write $A \otimes B \otimes C$ without ambiguity.

Definition 2.2.2. A **symmetric monoidal category** is a monoidal category equipped with natural isomorphisms $\sigma_{A,B} : A \otimes B \to B \otimes A$ such that $\sigma_{B,A}\sigma_{A,B} = 1_{A\otimes B}$ and the

following diagram commutes

$$\begin{array}{ccccc} A \otimes (B \otimes C) & \xrightarrow{\mathrm{id}_A \otimes \sigma_{B,C}} & A \otimes (C \otimes B) & \xrightarrow{\alpha_{A,C,B}} & (A \otimes C) \otimes B \\ \downarrow{\scriptstyle \alpha_{A,B,C}} & & & & \downarrow{\scriptstyle \sigma_{A,C} \otimes \mathrm{id}_B} \\ (A \otimes B) \otimes C & \xrightarrow{\sigma_{A \otimes B,C}} & C \otimes (A \otimes B) & \xrightarrow{\alpha_{C,A,B}} & (C \otimes A) \otimes B \end{array} .$$

The morphism $\sigma_{\_,\_}$ is commonly known as the **braiding** of the monoidal category.

Definition 2.2.3. A **lax monoidal functor** between two monoidal categories $(\mathcal{C}, \otimes_{\mathcal{C}}, I_{\mathcal{C}})$ and $(\mathcal{D}, \otimes_{\mathcal{D}}, I_{\mathcal{D}})$ consists of a functor $F : \mathcal{C} \to \mathcal{D}$, a morphism $\psi : I_{\mathcal{D}} \to F(I_{\mathcal{C}})$, and natural transformations $\varphi_{A,B} : F(A) \otimes_{\mathcal{D}} F(B) \to F(A \otimes_{\mathcal{C}} B)$, such that the following diagrams commute

$$\begin{array}{ccc} F(A) \otimes_{\mathcal{D}} (F(B) \otimes_{\mathcal{D}} F(C)) & \xrightarrow{\alpha^{\mathcal{D}}} & (F(A) \otimes_{\mathcal{D}} F(B)) \otimes_{\mathcal{D}} F(C) \\ \downarrow{\scriptstyle \mathrm{id} \otimes \varphi} & & \downarrow{\scriptstyle \varphi \otimes \mathrm{id}} \\ F(A) \otimes_{\mathcal{D}} F(B \otimes_{\mathcal{C}} C) & & F(A \otimes_{\mathcal{C}} B) \otimes_{\mathcal{D}} F(C) \\ \downarrow{\scriptstyle \varphi} & & \downarrow{\scriptstyle \varphi} \\ F(A \otimes_{\mathcal{C}} (B \otimes_{\mathcal{C}} C)) & \xrightarrow{F(\alpha)} & F((A \otimes_{\mathcal{C}} B) \otimes_{\mathcal{C}} C) \end{array} \tag{2.1}$$

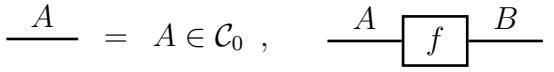
$$\begin{array}{ccc} I_{\mathcal{D}} \otimes_{\mathcal{D}} F(A) & \xrightarrow{\psi \otimes \mathrm{id}} & F(I_{\mathcal{C}}) \otimes_{\mathcal{D}} F(A) \\ \downarrow{\scriptstyle \lambda^{\mathcal{D}}} & & \downarrow{\scriptstyle \varphi} \\ F(A) & \xleftarrow{F(\lambda^{\mathcal{C}})} & F(I_{\mathcal{C}} \otimes_{\mathcal{C}} A) \end{array} \tag{2.2}$$

$$\begin{array}{ccc} F(A) \otimes_{\mathcal{D}} I_{\mathcal{D}} & \xrightarrow{\mathrm{id}\otimes\psi} & F(A) \otimes_{\mathcal{D}} F(I_{\mathcal{C}}) \\ \Big\downarrow{\scriptstyle \rho^{\mathcal{D}}} & & \Big\downarrow{\scriptstyle \varphi} \\ F(A) & \xleftarrow{F(\rho^{\mathcal{C}})} & F(A \otimes_{\mathcal{C}} I_{\mathcal{C}}) \end{array} \tag{2.3}$$

where $\alpha^{\mathcal{C}}, \lambda^{\mathcal{C}}, \rho^{\mathcal{C}}$ and $\alpha^{\mathcal{D}}, \lambda^{\mathcal{D}}, \rho^{\mathcal{D}}$ denote the associator, left unitor, and right unitor of $\mathcal{C}$ and $\mathcal{D}$ respectively. We say that F is a **strong monoidal functor** if ψ and φ are isomorphisms, and a **strict monoidal functor** if ψ and φ are identities.

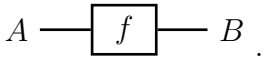
2.3 String Diagrams

A category $\mathcal{C}$ can be regarded as a directed multigraph G with a rule on how to compose paths. The nodes of G are the objects of the category, and the edges are the arrows (morphisms) from one object to another. Each node has a loop edge corresponding to the identity arrow. The associative composition operation of arrows of the category, together with the coherence laws of how it must behave, constitutes the rule on how to compose paths in the graph. Roughly speaking, string diagrams correspond to the notion of the line graph of G, in which the nodes become the edges and the edges become the nodes. The nodes of the line graph are the morphisms of the category, and the edges are the objects. For this reason, we will depict an object $A \in \mathcal{C}_0$ as a string (or edge, wire), and a morphism $f \in \mathcal{C}_1$ as a node (or box) connected to the strings of its domain on the left and its codomain on the right:

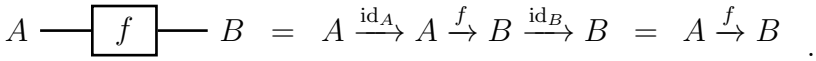


In order to omit drawing the direction of the strings, we will read the diagrams from left to right (we will see that we only require the beginning and the ending of a string to *travel* from left to right). The identity of an object will be depicted as

Alternatively, sometimes we might tag a string with the name of the object on the sides rather than in the top as in

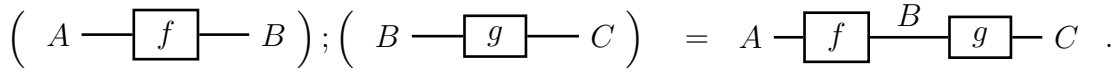


This is justified by the depiction of the identity and the coherence laws of the category, since

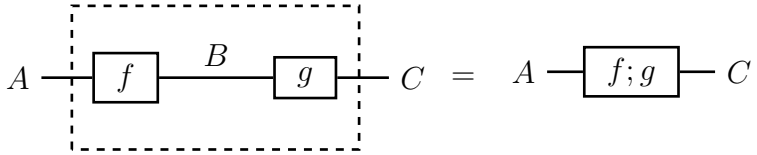


Moreover, the identity will allow us to extend the length of a string arbitrarily, as if we were attaching as many identities to a string as needed. Also, this will allow us to *slide* boxes along strings.

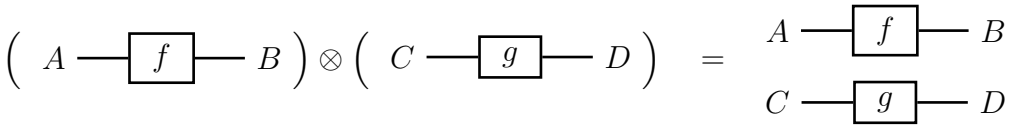
The sequential composition of morphisms f and g will be drawn as connecting the common wires $t(f)$ and $s(g)$:



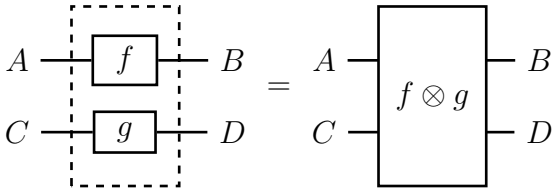
Given that $f;g$ is itself a morphism of the category, we will allow ourselves to *black-box* the composition of f and g as a single box connecting the domain of f to the codomain of g:



Similarly, the monoidal product of a monoidal category $(\mathcal{C}, \otimes, I)$ will be depicted as a parallel composition (juxtaposition) of diagrams in the vertical direction, for example



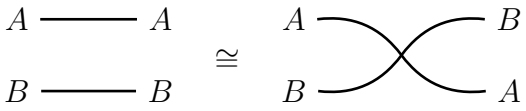
As we did with sequential composition, we can represent the morphism $f \otimes g$ as a single box with domain $A \otimes C$ and codomain $B \otimes D$:



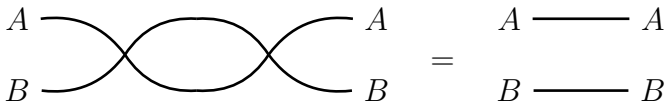
As we see, string diagrams provide a friendly depiction of morphisms connecting inputs (the incoming wires) to outputs (the outgoing wires). Furthermore, we can observe from the last diagram that the structure of a monoidal category allows us to start talking about boxes that connect multiple inputs to multiple outputs, that is, hyperedges in the graph G which we mentioned at the beginning of this section. To further expand the reach and form of these hyperedges, in [11, 17] the notion of hypergraph categories is introduced by equipping each object of a monoidal category with the structure of a special commutative Frobenius monoid, as we will see in the next section.

Lastly, the fact that the braiding $\sigma_{A,B}$ of a symmetric monoidal category is a natural

isomorphism $A \otimes B \to B \otimes A$, is represented in diagrams as



whereas the fact that $\sigma_{B,A}\sigma_{A,B} = 1_{A\otimes B}$, is given by



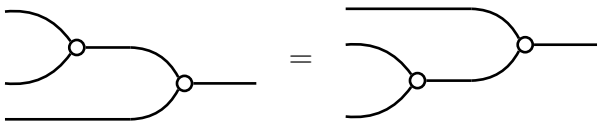
.

2.4 Hypergraph Categories and Hopf Algebras

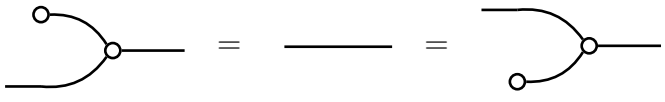
Definition 2.4.1. A **monoid object** (M, μ, ν) in a symmetric monoidal category $(C, \otimes, I)$ is an object $M \in \mathcal{C}$ equipped with two morphisms

$\mu : M \otimes M \to M$ $\qquad$ $\nu : I \to M$

satisfying the associative law

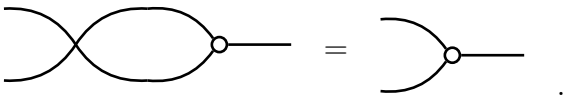


and the left and right unit laws



.

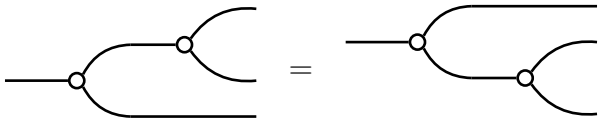
A monoid is **commutative** if it also satisfies



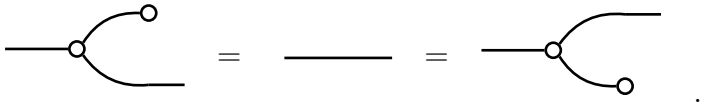
Definition 2.4.2. A **comonoid object** (M, μ, ν) in a symmetric monoidal category $(C, \otimes, I)$ is an object $M \in \mathcal{C}$ equipped with two morphisms

$\delta : M \to M \otimes M$ $\epsilon : M \to I$

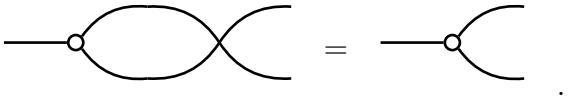
satisfying the coassociative law



and the left and right counit laws

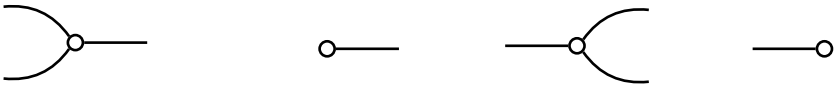


A **cocommutative comonoid** also satisfies



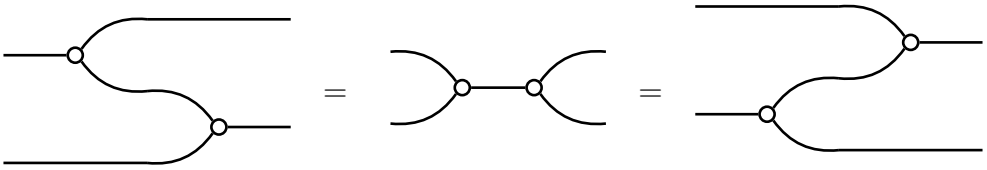
A **Frobenius monoid** consists of a commutative monoid and cocommutative comonoid on the same object that interact with each other obeying the so-called Frobenius and special axioms. They are also known as commutative separable algebras [25].

Definition 2.4.3. Given a symmetric monoidal category $(\mathcal{C}, \otimes, I)$, a **special commutative Frobenius monoid** $(A, \mu_F, \nu_F, \delta_F, \epsilon_F)$ consists of an object A together with the morphisms

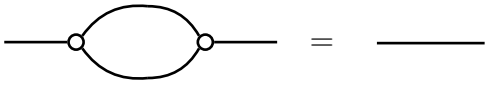


$\mu : A \otimes A \to A$ $\quad$ $\nu : I \to A$ $\quad$ $\delta : A \to A \otimes A$ $\quad$ $\epsilon : A \to I$

such that (A, μ_F, ν_F) is a commutative monoid, and $(A, \delta_F, \epsilon_F)$ is a cocommutative comonoid obeying the Frobenius axiom

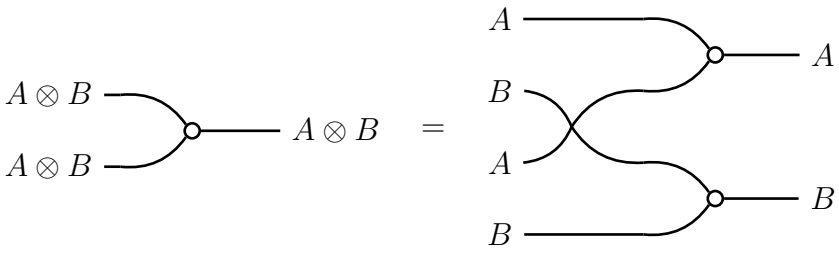


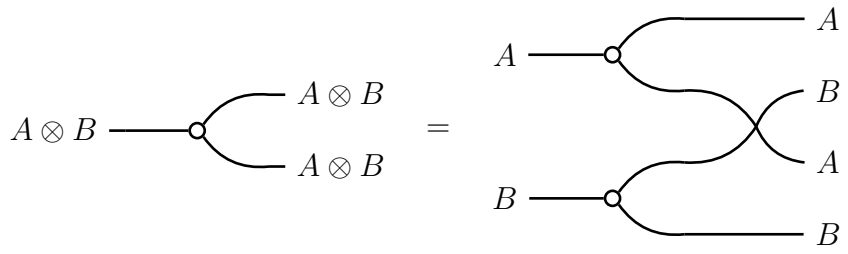
and the special axiom



.

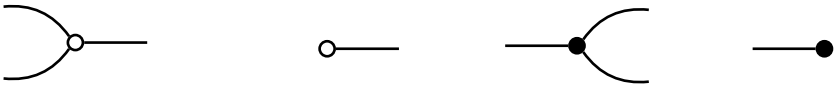
Definition 2.4.4. A **hypergraph category** is a symmetric monoidal category $(C, \otimes, I)$ in which every object is equipped as a special commutative Frobenius monoid such that it interacts with the monoidal product satisfying





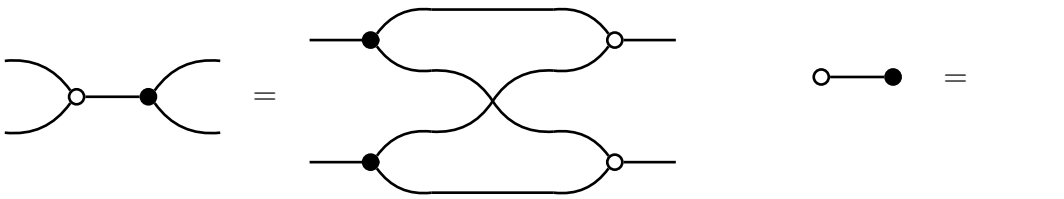
Hypergraph categories are also known as well-supported compact closed categories [8].

Now, suppose we have an object with a monoid structure (A, μ, ν) and a comonoid structure (A, δ, ϵ) that are not dual of each other, say



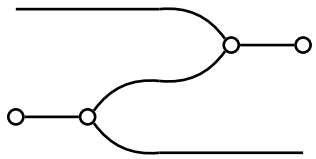
$\mu : A \otimes A \to A$ $\quad$ $\nu : I \to A$ $\quad$ $\delta : A \to A \otimes A$ $\quad$ $\epsilon : A \to I$

(we will use the notation for δ and ϵ in chapter 3 for a specific comonoid structure). In [4], the authors describe in detail how a pair of a monoid and a comonoid such as the ones described above interact with each other. In particular, the bimonoid $(A, \mu, \nu, \delta, \epsilon)$ in a symmetric monoidal category $(\mathcal{C}, \otimes, I)$ forms what is called a Hopf algebra, and it satisfies

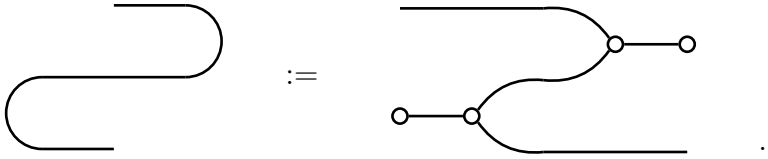


Analogous laws are satisfied for the dual bialgebra. For a full description of intearcting Hopf algebras see [4].

Hypergraph categories have a useful feature: they are self-dual and compact closed. One consequence of these facts is that the wires in a hypergraph category can also *flow* from right to left. We won't go over the deatils here, a more extended explanation can be found in [11, 12, 17]. For our purposes, it will suffice to have the property that the middle part of wire can flow from right to left. For example, consider the diagram



Observe that the connecting wire on the left (the input) is the same as the connecting wire on the right (the output). Thus, this diagram can be thought of as a single string whose endpoints flow from left to right, but whose middle part flow in the opposite direction:



As mentioned in the previous section, hypergraph categories provide more power and flexibility to build any kind of hyperedges from multiple inputs to multiple outputs. We will use these properties to construct a suitable symmetric monoidal category equipped with the structure we need in order to describe the neural network concepts that will be used in chapter 3.

CHAPTER 3

A STRING-DIAGRAMMATIC APPROACH OF PERCEPTRON-BASED NEURAL NETWORKS

This chapter covers some of the basics of neural network (NN) architectures framing them as string diagrams of a hypergraph category equipped with extra structure. Although nowadays there are numerous types of NN architectures, we will focus on three of the most fundamental ones, namely, multilayer perceptron, recurrent, and long short-term memory. Then, we will use some of these architectures to build a new one that models our sequence problem described in the next chapter. We begin by building a suitable category and diagram notation through which these architectures will be described.

3.1 The Categorical Construction

Consider the symmetric monoidal category $(\mathbf{Euc}, \otimes, I)$ (see definition 2.2.2), where the objects of **Euc** are euclidean spaces regarded as affine spaces, and whose arrows are transformations between Euclidean spaces (these can be linear, affine, nonlinear, etc.). The associative product $\otimes$ will function as an abstract monoidal operation that allows us to consider the juxtaposition of any number of independent transformations simultaneously, that is, the coexistence of diagrams in parallel. In consequence, we shall regard the unit I as the empty diagram.

In addition, we will equip every object $A \in \mathbf{Euc}$ with a special Frobenius monoid structure (see definition 2.4.3) given by a *copy* map δ, a *discard* map ϵ, a *co-copy* map μ, and an

initialize map ν, denoted by

$\delta : A \to A \otimes A$ $\epsilon : A \to I$ $\mu : A \otimes A \to A$ $\nu : I \to A$

providing $(\mathbf{Euc}, \otimes, I)$ with the structure of a hypergraph category (see definition 2.4.4).

Moreover, every Euclidean space is naturally a monoid object (see definition 2.4.1) under vector addition, and also under the Hadamard product (element-wise multiplication). In fact, every object is a ring under these two operations, but for our purposes we only require the monoid structures of each object. We will denote the monoid structure of an object $M \in \mathbf{Euc}$ under **vector addition** by

$\mu_+ : M \otimes M \to M$ $\nu_+ : I \to M$

whence is the zero vector, and the monoid structure of $M \in \mathbf{Euc}$ under the **Hadamard product** by

$\mu_\bullet : M \otimes M \to M$ $\nu_\bullet : I \to M$

whence is the all-ones vector.

There is one additional structure with which we need to furnish our category, and that is the concatenation of vectors of points of any two Euclidean spaces. Vector concatenation

is a fundamental operation in deep learning because it allows us to pass a number of vectors to a network without having to change its arrangement of operations. This is useful when developing a deep-learning framework because it enables conditioning a network to any number of vectors without having to rewrite the modules of the framework used to construct the network, it only requires to change the input size.

Given two row vectors $a \in \mathbb{R}^k, b \in \mathbb{R}^l$, we denote their (horizontal) concatenation by $a \sqcup b = [a \ \ b] \in \mathbb{R}^{k+l}$. In order to mesh this operation with our diagrammatic formalism, we introduce a **concatenation** operation as the strong monoidal functor $\sqcup : (\mathbf{Euc}, \otimes, I) \to (\mathbf{Euc}, \otimes, I)$ (see definition 2.2.3) given by the morphisms $\varphi_{A,B} : A \otimes B \to A \sqcup B$ and $\psi : I \to [\ ]$, where $[\ ]$ denotes the empty vector, satisfying

$$\begin{array}{ccc}
A \otimes (B \otimes C) & \xrightarrow{\alpha} & (A \otimes B) \otimes C \\
\downarrow{\scriptstyle \mathrm{id}\otimes\varphi} & & \downarrow{\scriptstyle \varphi\otimes\mathrm{id}} \\
A \otimes (B \sqcup C) & & (A \sqcup B) \otimes C \\
\downarrow{\scriptstyle \varphi} & & \downarrow{\scriptstyle \varphi} \\
A \sqcup (B \sqcup C)) & \xrightarrow{\sqcup(\alpha)} & (A \sqcup B) \sqcup C
\end{array} \tag{3.1}$$

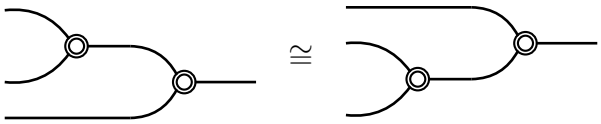
$$\begin{array}{ccc}
I \otimes A & \xrightarrow{\psi\otimes\mathrm{id}} & [\ ] \otimes A \\
\downarrow{\scriptstyle \lambda} & & \downarrow{\scriptstyle \varphi} \\
A & \xleftarrow{\sqcup(\lambda)} & [\ ] \sqcup A
\end{array} \qquad
\begin{array}{ccc}
A \otimes I & \xrightarrow{\mathrm{id}\otimes\psi} & A \otimes [\ ] \\
\downarrow{\scriptstyle \rho} & & \downarrow{\scriptstyle \varphi} \\
A & \xleftarrow{\sqcup(\rho)} & A \sqcup [\ ]
\end{array} \tag{3.2}$$

where α denotes the associator of $(\mathbf{Euc}, \otimes, I)$, and λ, ρ are its left and right unitor respec-

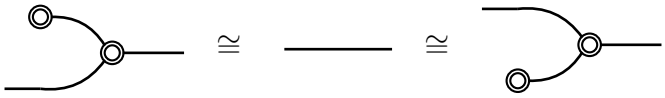
tively. If we let φ and ψ be denoted by

$$\varphi_{A,B} : A \otimes B \to A \sqcup B \qquad \psi : I \to [\ ] \ ,$$

then the commutative diagrams above translate into string diagrams as



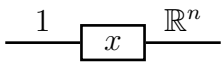
and



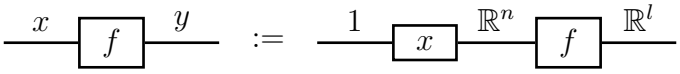
respectively, which clearly correspond to an associativity law, and left and right unit unit laws of a monoid. This is to be expected since, just as the monoidal product, concatenation induces a free monoid over **Euc**. For this reason, the author has decided to denote φ and ψ in an analogous way to the way we denoted a monoid object in definition 2.4.1, with the difference that the white circle is double-lined. This is to emphasise the fact that we are not depicting a monoid structure of an object, but rather a free monoid structure of the category itself.

Finally, to help the interpretability of diagrams, we will usually tag input and output strings by the name of global elements of the given spaces rather than by the spaces' names

themselves, in other words, the strings will carry the names of the points of the Euclidean space to which it belongs. For example, suppose $x \in \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}^l$, and $f(x) = y \in \mathbb{R}^l$. Strictly speaking, in string diagrams x is depicted as the global element morphism



where 1 is the Euclidean space with just one point. However, we will allow the following notation



to better distinguish inputs or outputs that are instances of the same object. Moreover, any operation on a point $x \in \mathbb{R}^n$ will be assumed to be an operation on its associated vector over the standard basis.

3.2 Affine Transformations and Training

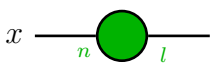
Among all the transformations from one affine space to another, affine transformations are at the very heart of neural network architectures (convolutions are equally important, but we will save their formulation for subsequent work). These transformations, encoded by a *weight* matrix W and a *bias* vector b, carry the trainable parameters with respect to which an objective function $\mathcal{L}$ is minimized. In supervised learning, a dataset usually consists of pairs of points (x, y) in which y serves as a *label* or *target* associated to the point x. In general, the goal of a machine learning model, such as a neural network, is to take a point x as an input and to generate a *prediction* $\hat{y}$ as an output such that, in average, $\mathcal{L}(\hat{y}, y)$ meets

an optimizing criterion for all points (x, y) in the dataset. Choosing a convenient vector representation for x, y, and $\hat{y}$, as well as an effective loss function that makes sense for a given purpose, is of crucial importance to training a model. Such techniques constitute a vast research area and are not the focus of this work, we refer the reader to [13] for further details.

There are various methods by which a function can be minimized, but the most used by far in deep learning are algorithms that are gradient-based. Such algorithms depend on the gradient of the loss function with respect to the parameters of all affine transformations involved in the architecture, and they update such parameters by some gradient step criterion. Given the great relevance of affine transformations that harbor trainable parameters of an architecture, we have decided to dedicate a special kind of shape for the nodes representing these transformations. Trainable affine transformations will be depicted as colored circle nodes, with the option of writing in color the sizes of the input and output spaces right below the incoming and outgoing wires respectively. For example, let $\bigcirc : \mathbb{R}^n \to \mathbb{R}^l$ be the affine transformation given by

$$\hat{y} = xW + b \tag{3.3}$$

with x, y seen as row vectors (a convection in deep learning). This morphism will be depicted as the string diagram



. (3.4)

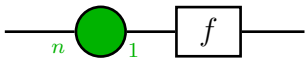
Clearly, the size of the matrix W should be $n \times l$, while b should be a row vector of size

$1 \times l$. Training the architecture (3.4) with $l = 1$ is known as *linear regression* (trained on a suitable dataset where the targets are scalars).

The idea of using colored circles to represent these matrix operations was inspired by tensor network diagrams [5, 6], although our adoption here differs a little. To our knowledge, this is not a standard practice in category theory, we just found it visually helpful when designing architectures.

3.3 Perceptron and Fully Connected Layer

A **perceptron** architecture is an affine transformation $\bigcirc : \mathbb{R}^n \to \mathbb{R}$ followed by a nonlinear function $f : \mathbb{R} \to \mathbb{R}$, usually referred to as *activation* function:



. (3.5)

Training this architecture with f being a **sigmoid** function (a bounded, differentiable, real-valued function whose domain is all real numbers and its first derivative is non-negative everywhere) is known as *logistic regression*. Often, the term *sigmoid* is reserved for the specific function

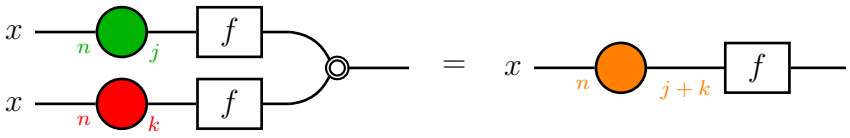
$$\sigma(z) = \frac{1}{1 + e^{-z}} \ . \tag{3.6}$$

We will follow the same convention. When applied to a vector $x = [x_i]$, the sigmoid function acts element-wise: $\sigma(x) = [\sigma(x_i)]$.

Furthermore, the architecture 3.5 is also what in deep learning is known as a *neuron*. Concatenating l different neurons with the same element-wise activation function constitutes

what it is known as a *layer* of size l. To simplify the representation of stacked (concatenated) neurons, we will make use of the following observation.

Observation 3.3.1. *Given a pair of transformations* $\bigcirc : \mathbb{R}^n \to \mathbb{R}^j$, $\bigcirc : \mathbb{R}^n \to \mathbb{R}^k$, *and an element-wise function* f, *then*



(3.7)

for some transformation $\bigcirc : \mathbb{R}^n \to \mathbb{R}^{j+k}$.

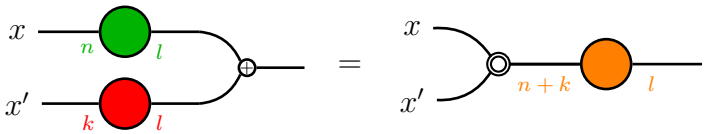
Proof. Let $x_{1\times n} \in \mathbb{R}^n$, $W_{n\times(j+k)} = [W_{n\times j} \;\; W_{n\times k}] \in \mathbb{R}^{n\times(j+k)}$, $b_{1\times(j+k)} = [b_{1\times j} \;\; b_{1\times k}] \in \mathbb{R}^{j+k}$, and let f be an element-wise function. Then we observe that

$$\begin{aligned}
[f\left(x_{1\times n}W_{n\times j} + b_{1\times j}\right) \quad f\left(x_{1\times n}W_{n\times k} + b_{1\times k}\right)] &= f\left([x_{1\times n}W_{n\times j} + b_{1\times j} \quad x_{1\times n}W_{n\times k} + b_{1\times k}]\right) \\
&= f\left(x_{1\times n}[W_{n\times j} \;\; W_{n\times k}] + [b_{1\times j} \;\; b_{1\times k}]\right) \\
&= f\left(x_{1\times n}W_{n\times(j+k)} + b_{1\times(j+k)}\right) \quad ,
\end{aligned}$$

which is indeed an affine transformation $\bigcirc : \mathbb{R}^n \to \mathbb{R}^{j+k}$ followed by f. □

Another useful property for re-writing affine transformations to our convenience is the following.

Observation 3.3.2. *Given a pair of transformations* $\bigcirc : \mathbb{R}^n \to \mathbb{R}^l$, $\bigcirc : \mathbb{R}^k \to \mathbb{R}^l$, *there exists a transformation* $\bigcirc : \mathbb{R}^{n+k} \to \mathbb{R}^l$ *such that*



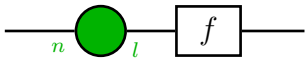
(3.8)

Proof. Similar to the proof above, let $W_{(n+k)\times l} = \begin{bmatrix} W_{n\times l} \\ W_{k\times l} \end{bmatrix}$, $b_{1\times(l)} = b_{1\times l} + b_{1\times l}$. Then, for two vectors $x_{1\times n} \in \mathbb{R}^n$ and $x'_{1\times k} \in \mathbb{R}^k$, we have

$$\begin{aligned}
(x_{1\times n}W_{n\times l} + b_{1\times l}) + \left(x'_{1\times k}W_{k\times l} + b_{1\times l}\right) &= \left(x_{1\times n}W_{n\times l} + x'_{1\times k}W_{k\times l}\right) + (b_{1\times l} + b_{1\times l}) \\
&= \begin{bmatrix} x_{1\times n} & x'_{1\times k} \end{bmatrix} \begin{bmatrix} W_{n\times l} \\ W_{k\times l} \end{bmatrix} + (b_{1\times l} + b_{1\times l}) \\
&= \begin{bmatrix} x_{1\times n} & x'_{1\times k} \end{bmatrix} W_{(n+k)\times l} + b_{1\times(l)} \ .
\end{aligned}$$

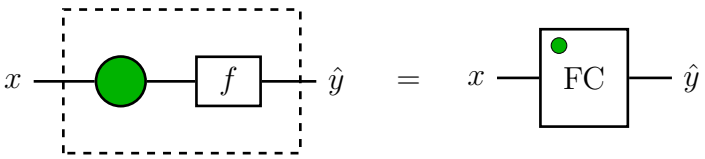
□

By observation 3.3.1, therefore, a layer of l stacked neurons can also be seen as the output of the transformation



. (3.9)

For this reason, the architecture (3.9) is also called a **fully connected** (FC) layer. Fully connected layers are found somewhere in almost all neural network architectures, even in many convolutional networks in which at least one fully connected layer is usually attached as the last layer. We will dedicate to these types of morphisms their own blackboxed node FC together with a colored circle mark in the left upper corner corresponding to the affine transformation they harbor:



(3.10)

Drawing a colored circle in the left upper corner of the box, as in the right hand of the

equation above, is a stylistic choice of the author. To our knowledge, this is not a standard notation in category theory. The main reasons for this choice are, first, to visually keep track of what morphisms contain trainable parameters and whether or not they share the same parameters (*shared weights*), and secondly, in upcoming diagrams, to also keep track of how many different affine transformations a morphism contains. Moreover, whenever it is not of much relevance, we will omit the indices indicating the sizes of the input and output spaces of a transformation, just as we did in the diagrams above. Such indices are usually determined by the form of the dataset, or they are a hyperparameter choice of the architect.

In upcoming architectures (when we deal with timesteps), it will come in handy to have black-boxed node version, denoted as $\boxed{\text{FCs}}$, of several instances of a given FC layer in parallel as shown in figure 3.1, which can also be written in a compressed way as

$$x^{\langle 0\rangle} \otimes \cdots \otimes x^{\langle T\rangle} \longrightarrow \boxed{\text{FCs}} \longrightarrow \hat{y}^{\langle 0\rangle} \otimes \cdots \otimes \hat{y}^{\langle T\rangle} \quad . \tag{3.11}$$

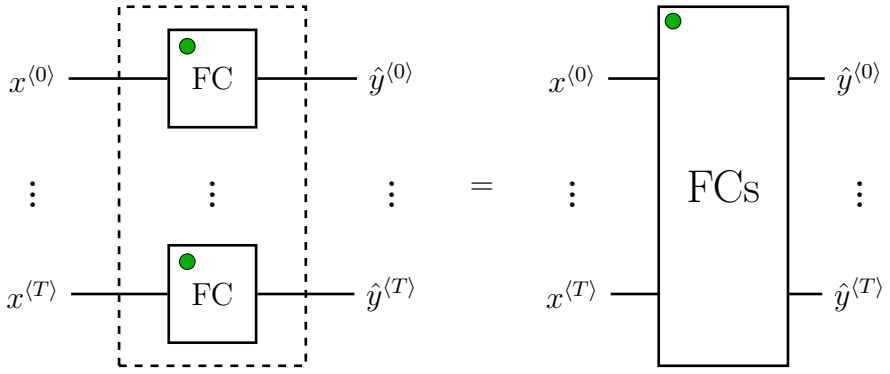
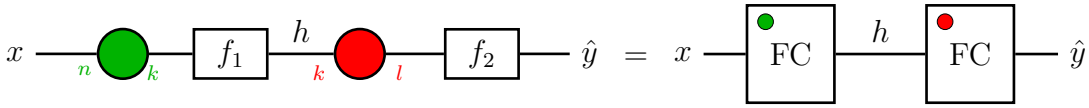


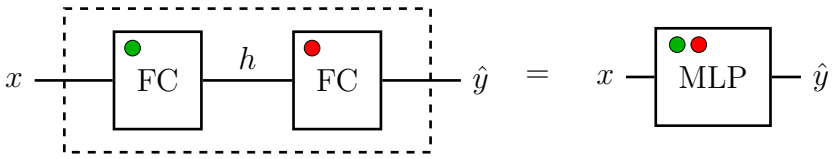
Figure 3.1: *Black-boxing T instances of one fully connected layer.*

3.4 Multilayer Perceptron

Consider the architecture of two fully connected layers composed sequentially, say



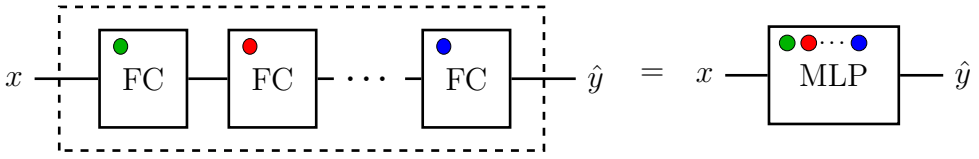
where h is called *hidden layer* or *hidden state*. This architecture is called a **multilayer perceptron**, and we will denote it with a node named MLP as in



(3.12)

Some people refer to the space to which h belongs as the *latent space*. The study of the properties of this space and how it is created during training is an active research area and it has broad applications such as autoencoders, non-linear dimensionality reduction, word embeddings, etc. In general, we can regard h in diagram 3.12 as a vector of some space that summarizes information of x relevant to the prediction $\hat{y}$.

In the case of 3.12 above, we say that the MLP has one hidden layer. However, a multilayer perceptron can consist of more than one hidden layer. This is the case when we connect sequentially three or more fully connected layers, as in



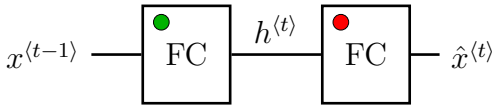
(3.13)

Here, each colored circle represents a different trainable affine transformation, which are represented in the node in the right as well. We refer to the number of hidden layers as the *depth* of the network (which equals the number of fully connected layers minus 1). The use of deeper networks is what inspired the name *deep* learning.

A fundamental and well-known result in machine learning is the fact that there exists an integer k and two non-polynomial locally bounded piece-wise continuous functions f_1, f_2 such that the morphism 3.12 can approximate any function $\mathbb{R}^n \to \mathbb{R}^l$ to any degree of accuracy. This is known as the universal approximation property [18, 19]. The fact that multilayer perceptrons are universal aproximators is what makes neural networks so powerful, and with the advance of computing power and data collection, they have risen over many other machine learning techniques in the last decade. The fact that they satisfy the universal approximation theorem does not directly imply that the choice of k, f_1, and f_2 is known for a given desired accuracy, however, many advances have been made in this direction [9, 20, 21].

3.5 Recurrent Neural Network

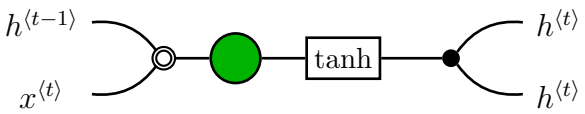
The idea of building a recurrent neural network (RNN) was motivated by the desire of modeling sequence problems while taking advantage of the power an MLP. Let us consider a particular problem as a case study. Suppose we have sequences of points of the form $x^{\langle 0\rangle}, x^{\langle 1\rangle}, \ldots, x^{\langle T\rangle}$, and that for any timestep $0 < t \leq T$, we would like to model the conditional probability $P(x^{\langle t\rangle}|x^{\langle t-1\rangle}, x^{\langle t-2\rangle}, \ldots, x^{\langle 0\rangle})$. One thing we could do is to train an MLP that maps $x^{\langle t-1\rangle}$ to a prediction $\hat{x}^{\langle t\rangle}$ (that will be compared to the target $x^{\langle t\rangle}$), say



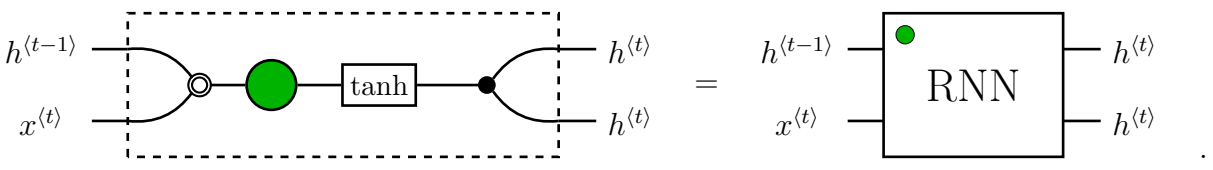
.

However, with this construction we can only model $P(x^{\langle t\rangle}|x^{\langle t-1\rangle})$. The next best thing we could do is to concatenate all previous vector points $x^{\langle t-1\rangle} \sqcup x^{\langle t-2\rangle} \sqcup \cdots \sqcup x^{\langle 0\rangle}$, pass it to the network, and have a prediction $\hat{x}^{\langle t\rangle}$. However, we observe that as t progresses, the conditioning becomes larger and larger, so the input size of the network would not be consistent, so we could not use the exact same transformations for every timestep. An elegant solution to this dilemma is to make use of the hidden state $h^{\langle t\rangle}$ by concatenating it to the next timestep.

An **RNN unit** is a morphism of the form



which takes as input the concatenation of $h^{\langle t-1\rangle}$ and $x^{\langle t\rangle}$, and outputs two copies of $h^{\langle t\rangle}$. The hyperbolic tangent is the standard activation function in these networks, but any other activation function can be used. We will denote these type of units by a node RNN ,

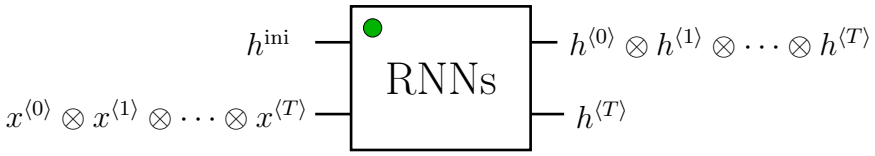


.

In our sequence prediction example, one copy of $h^{\langle t\rangle}$ can be passed to a FC layer to generate

a prediction, and the other copy is passed to the RNN unit of the next timestep creating the feedback diagram shown in figure 3.2. With this modification, now $h^{\langle t \rangle}$ (ideally) contains a summarized information of $x^{\langle t \rangle}$ as well as all previous points $x^{\langle t-1 \rangle}, x^{\langle t-2 \rangle}, \ldots, x^{\langle 0 \rangle}$), and, in principle, our conditioning problem is solved.

The diagram in figure 3.2 is what is known as a (forward) **recurrent neural network**. We can black-box the entire diagram as one single node named RNNs as in figure 3.3, which can be written also in a compressed way as



(3.14)

Here, h^{ini} is called the *initial hidden state*, and it is mainly used to condition the whole

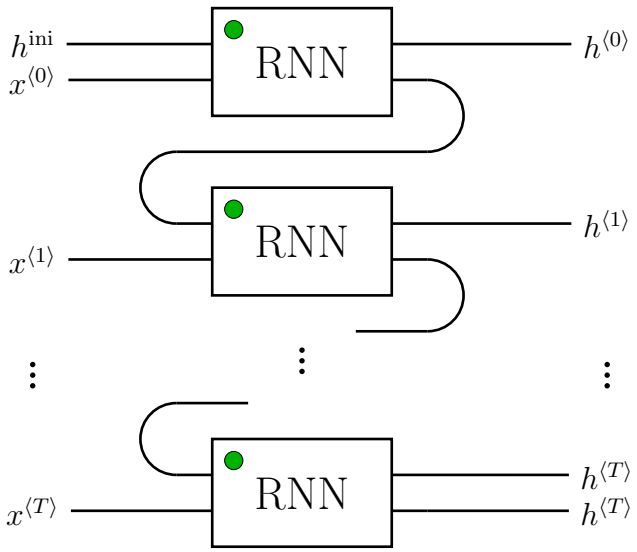


Figure 3.2: *T recurrent units connected forming an RNN.*

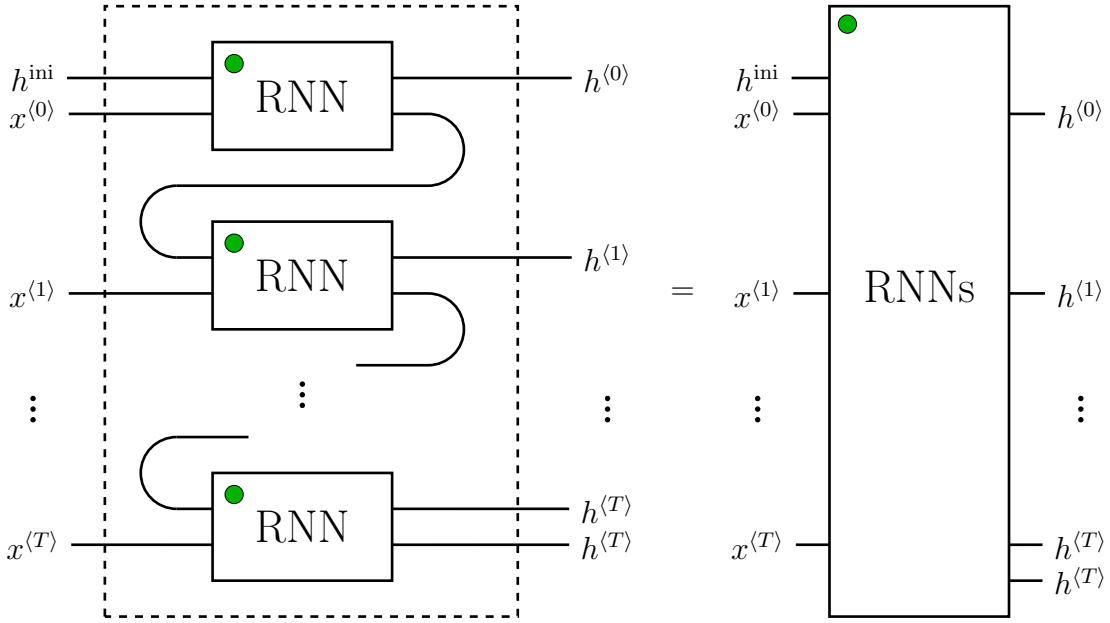
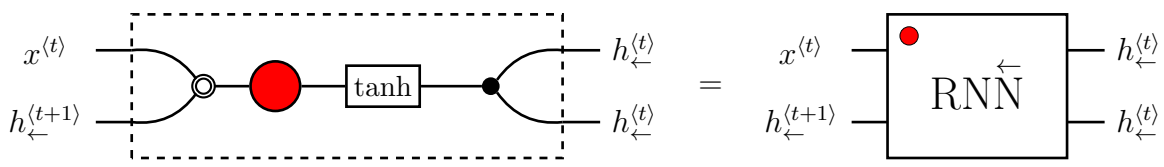


Figure 3.3: *Black-boxing a recurrent neural network into a single node.*

recurrent network to some previous information. For example, it could be the last hidden state $h^{\langle \tilde{T} \rangle}$ encoded by another network, or it could also be just a vector of zeros if such conditioning is not needed.

Now, let us modify our case study and suppose that instead we would like to model $P(x^{\langle t \rangle} | x^{\langle t+1 \rangle}, x^{\langle t+2 \rangle}, \ldots, x^{\langle T \rangle})$ for $0 \leq t < T$. Following the same reasoning above, we can emulate a similar formulation by creating a **backward RNN unit**, which we will denote as a node $\boxed{\text{RN}\overleftarrow{\text{N}}}$, given by



which takes as inputs the point $x^{\langle t \rangle}$ and the hidden state of the following time-step denoted by $h_{\overleftarrow{}}^{\langle t+1 \rangle}$. Using this new cell we can construct a **backward recurrent neural network** defined by the diagram architecture shown in figure 3.4. We will denote a backward RNN by a single node $\boxed{\overleftarrow{\text{RNN}}\text{s}}$ as in

$$\begin{array}{l} x^{\langle 0 \rangle} \otimes x^{\langle 1 \rangle} \otimes \cdots \otimes x^{\langle T \rangle} \\ h_{\overleftarrow{}}^{\text{ini}} \end{array} \longrightarrow \boxed{\overleftarrow{\text{RNN}}\text{s}} \longrightarrow \begin{array}{l} h_{\overleftarrow{}}^{\langle 0 \rangle} \\ h_{\overleftarrow{}}^{\langle 0 \rangle} \otimes h_{\overleftarrow{}}^{\langle 1 \rangle} \otimes \cdots \otimes h_{\overleftarrow{}}^{\langle T \rangle} \end{array} \tag{3.15}$$

that instead of modeling the conditional dependence of previous points, it models the conditional dependence of following points. We could achieve the same morphism by just reversing the order of the original sequence and pass it to a (forward) recurrent network, however, the main usefulness of a backward RNN is that with it we can construct a **bidirectional recurrent neural network** by combining 3.14 and 3.15 as in the diagram

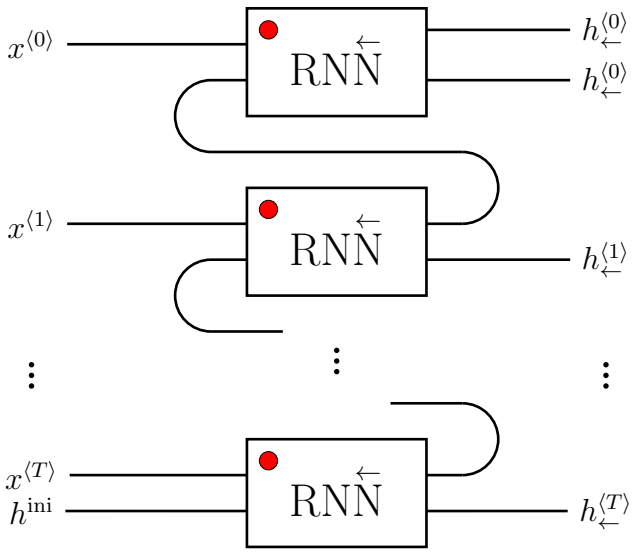
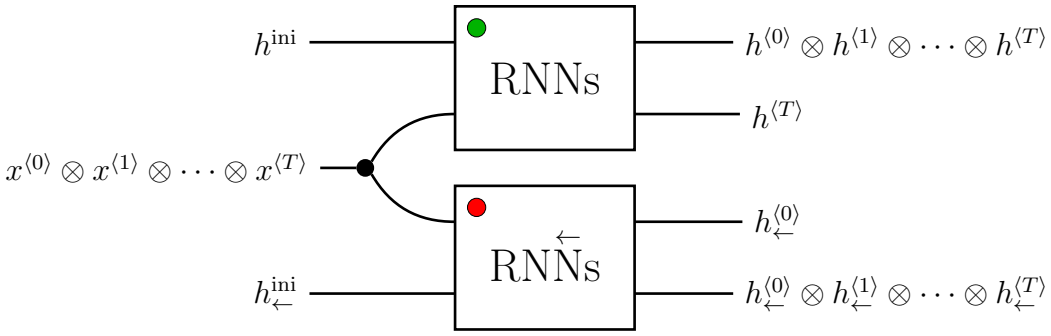
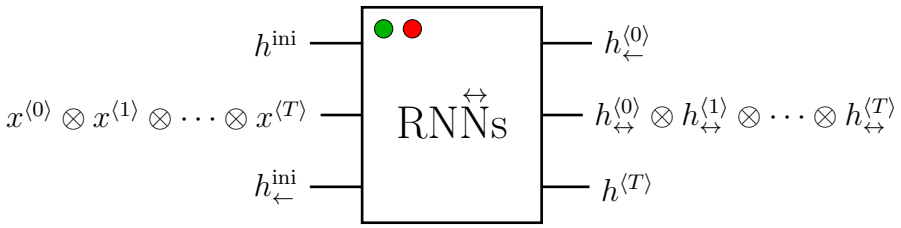


Figure 3.4: *Diagram of a backward RNN.*



Before black-boxing the diagram above, let us rearrange the outgoing wires in a way that will be more aligned with how deep-learning frameworks organize the ouputs of a bidirectional RNN. Besides braiding the wires however we want, we will concatenate each $h^{\langle i\rangle}$ and $h^{\langle i\rangle}_{\leftarrow}$ for $i = 0 \dots T$. Let $h^{\langle i\rangle}_{\leftrightarrow} = h^{\langle i\rangle} \sqcup h^{\langle i\rangle}_{\leftarrow}$, then, it is possible to rearrange the outgoing wires and black-box the diagram above into a single node denoted by $\boxed{\overleftrightarrow{\text{RNNs}}}$ as in the diagram below



In the above architecture, a hidden state $h^{\langle t\rangle}_{\leftrightarrow}$ encodes information about $x^{\langle t\rangle}$, previous points in the order $x^{\langle 0\rangle}, \dots, x^{\langle t-1\rangle}$, and following points in the order $x^{\langle t+1\rangle}, \dots, x^{\langle T\rangle}$.

3.6 Long Short-Term Memory Neural Network

In spite of the usefulness of RNNs, they are considered to be hard to train when T is large, due to vanishing gradients [2]. One improvement to RNNs are the so-called long short-term memory (LSTM) neural networks, whose recurrent unit involves a more sophisticated com-

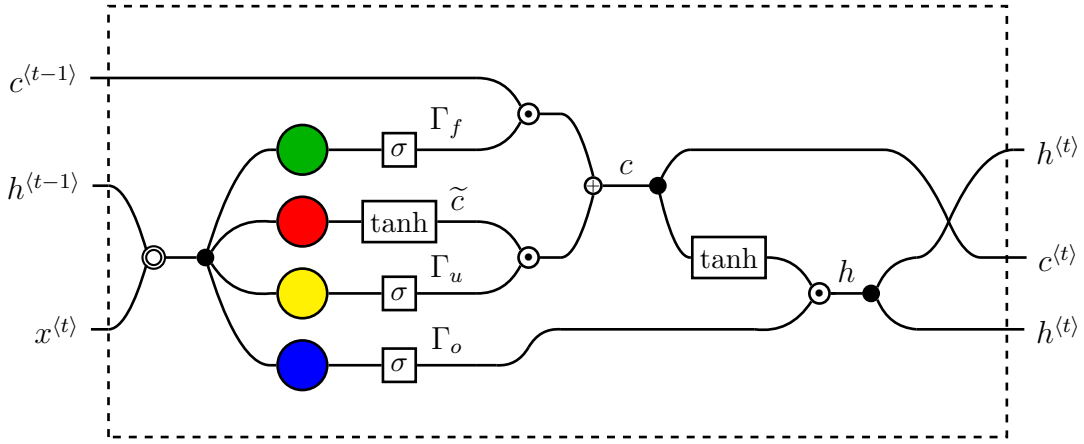
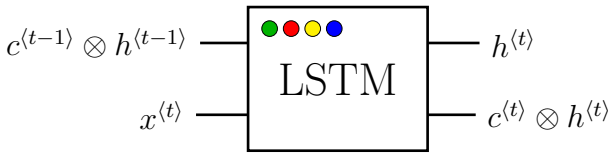


Figure 3.5: *Internal diagram of an LSTM unit.*

position of morphisms which is more robust to vanishing gradients, and, as a consequence, more efficient handling long time dependencies [14]. An **LSTM unit** is given by the diagram shown in figure 3.5, where c is referred to as the *memory cell*, Γ_f, Γ_u, and Γ_o are the *forget*, *update*, and *output* gate respectively, and $\widetilde{c}$ is sometimes called the *candidate* cell. For more details on the conception of such unit, see [14].

From a black-boxed perspective, LSTM units connect to each other in the same way RNNs do. Let us denote the LSTM unit above by a single node $\boxed{\text{LSTM}}$ as follows



.

Then, in analogy with a (forward) RNN in figure 3.2, we construct a (forward) **long short-term memory** neural network which has the form shown in figure 3.6, for some sequence

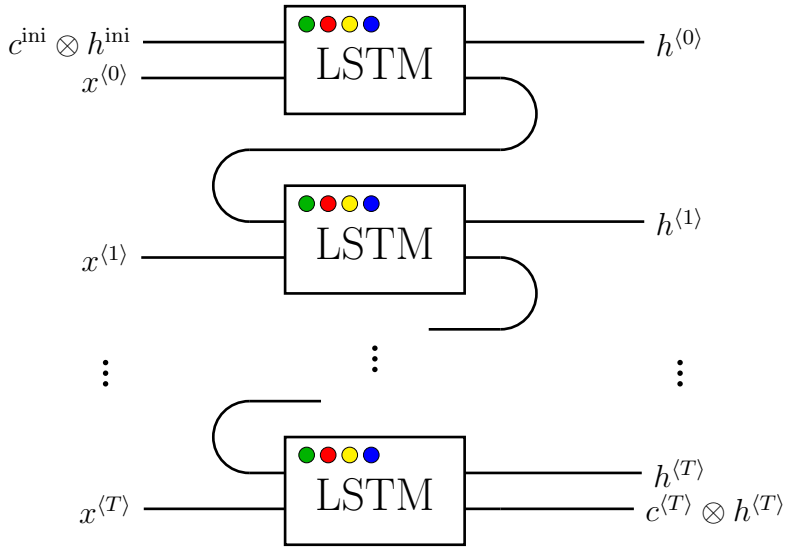
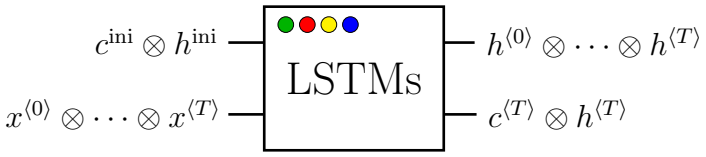


Figure 3.6: *Recurrent connections of an LSTM.*

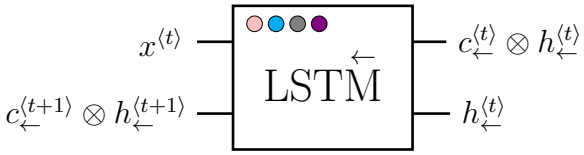
$x^{\langle 0 \rangle}, x^{\langle 1 \rangle}, \ldots, x^{\langle T \rangle}$. If we compare figure 3.6 to figure 3.2, we observe that inputs, outputs, and feedback connections are completely analogous in both diagrams, with the only difference that an LSTM requires an initial memory cell state $c^{\langle \text{ini} \rangle}$, and outputs an extra string $c^{\langle T \rangle}$. Furthermore, we can black-box the diagram in figure 3.6 and represent it by the single node



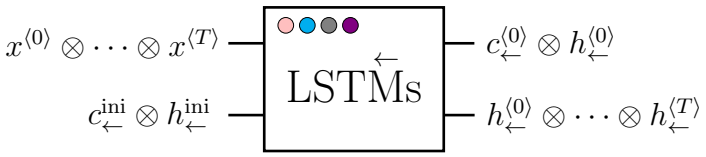
(3.16)

.

Similarly, with the same composition of morphisms of a LSTM unit (up to rearrangement of the input and output wires), we can conceive a **backward LSTM unit** as a node

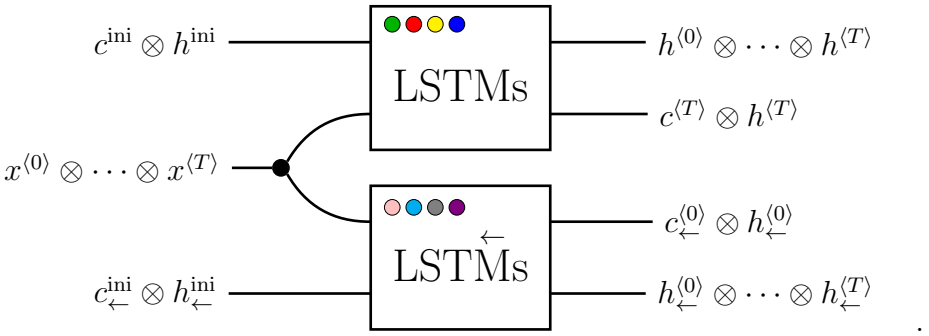


that allows us to construct, in analogy to (3.15), a **backward LSTM** as the morphism

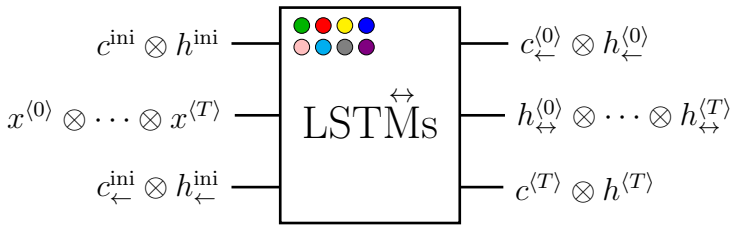


(3.17)

With (3.16) and (3.17) we can construct a **bidirectional LSTM** as follows



By letting $h_{\leftrightarrow}^{\langle i \rangle} = h^{\langle i \rangle} \sqcup h_{\leftarrow}^{\langle i \rangle}$ for $i = 0, \ldots, T$, and rearranging the outgoing wires accordingly, we are able to denote a bidirectional LSTM by the single node



(3.18)

3.7 Array Programming

We conclude this chapter with a discussion on an alternative interpretation of what a wire could also represent in our diagrams.

Given a *batch* of m training examples $(x^{(i)}, y^{(i)})$, where $x^{(i)}$ and $x^{(i)}$ are row vectors, it is a common technique in machine learning to stack $x^{(i)}$ as rows of a matrix X, and the targets $y^{(i)}$ as rows of a matrix Y. Feeding these matrices to a given neural network and a loss function, as opposed to feeding each $(x^{(i)}, y^{(i)})$ one by one using a for-loop, favors computational efficiency due to the fact matrix multiplications are done faster than for-loops in most programming languages that support vector arrays. This technique is known in machine learning as *vectorization* or *array programming*.

The reader can verify that all the diagrams and transformations we have constructed so far can also be regarded as diagrams whose wires are matrices of the form described above. To build such a perspective, we just need to make a few minor adjustments. Firstly, we need to agree that concatenating two matrices is done by rows. Secondly, any activation function that depends on more than one entry of a vector (e.g. Softmax) should be applied row by row. Lastly, in an affine transformation encoded by a matrix $W_{n\times l}$ and a vector $b_{1\times l}$, such as the one described by equation 3.3, we need to *broadcast* the vector b into having a size $m \times l$. This simply means taking m copies of b and stack them as rows. Then, the vectorized version of equation 3.3 would look like

$$\hat{Y}_{m\times l} = X_{m\times n} W_{n\times l} + B_{m\times l} \ , \tag{3.19}$$

where B is b broadcasted. Besides making a formal adjustment on what the new objects of the category are, the adjustments above are all we need in order to read the diagrams in the context of array programming.

However, we consider that array programming relates more to the subject of computational efficiency than to the algebra of morphisms of neural networks, therefore we have preferred to treat our wires as just vectors, since our main goal is to describe the compositional nature of these morphisms.

CHAPTER 4

MODELING SEQUENCES SUBJECT TO A TIME-STRUCTURED CONTEXT

In this chapter we present a dynamic neural network architecture for modeling sequences that have a conditioning context with a certain temporal structure, and the sequence points are not necessarily equally spaced in time. To illustrate the nature of this problem, let us imagine a hypothetical system that produces a sequence of signals in which each signal, besides having some other attributes, has a certain duration in time. Let us say that, every time a signal ends, the system has to decide what signal to produce next and how long it should last. At the same time, there exists a surrounding sequence of events (a context with temporal structure) that the system has to also take into account before deciding what signal to produce next. Each event has a duration in time, and the order in which these events happen is relevant to the system's decision. Ideally, when making a decision in a given point of time, the system should take into consideration not only what previous signals it has produced, but also the past, present, and future (if available) events and their order. Let us formulate the problem more precisely.

4.1 Formulation of the Problem

Consider a sequence $E = (e^{\langle 0 \rangle}, \ldots, e^{\langle T \rangle})$ of $T + 1$ events, and a sequence $S = (s^{\langle 0 \rangle}, \ldots, s^{\langle T' \rangle})$ of $T' + 1$ signals. Let $l^{\langle k \rangle}$ denote the duration in time of the signal $s^{\langle k \rangle}$. We assume that the event durations are dicretized and not drawn from a continuous set (the architecture presented in this chapter can be adapted for the continuous case as well, but we will reserve

that exposition for future reports). Without loss of generality, let us assume that the duration of each event is 1 time unit. To achieve this, first we would make a time transformation such that every event duration is an integer. Then, if an event has a duration of $m > 1$ time units, we will split that event into m succeeding identical events of 1 time unit each. The duration $l^{\langle k \rangle}$ of a signal should be adjusted to the time transformation above. We believe this transformation leads to an easier implementation.

Under these assumptions, a training example is a tuple (E, S) with $E = (e^{\langle 0 \rangle}, \ldots, e^{\langle T \rangle})$ and $S = (s^{\langle 0 \rangle}, \ldots, s^{\langle T^{'} \rangle})$, where each event $e^{\langle k \rangle}$ has a duration of 1 time unit, and a signal $s^{\langle k \rangle}$ has a duration of $l^{\langle k \rangle}$ time units. For implementation purposes, we will fix T in every training example (so all sequences have the same duration in time), and we will allow $T^{'}$ to be variable (the dual approach of fixing $T^{'}$ and allowing T to be variable is also possible). Thus, a dataset for us is a set $X = \{(E^{(i)}, S^{(i)}) \mid i = 0, \ldots, m-1\}$ of m training examples, where $E^{(i)} = (e^{\langle 0 \rangle (i)}, \ldots, e^{\langle T \rangle (i)})$ is a sequence of $T + 1$ events, and $S^{(i)} = (s^{\langle 0 \rangle (i)}, \ldots, s^{\langle T_i^{'} \rangle (i)})$ is a sequence of $T_i^{'} + 1$ signals, where, in principle, we can have $T_i^{'} \neq T_j^{'}$ for $i \neq j$. We should note that the sequences $E^{(i)}$ and $S^{(i)}$ must have the same total duration in time, that is,

$$T + 1 = \sum_{k=0}^{T_i^{'}} l^{\langle k \rangle}$$

for all $i = 0, \ldots, m-1$. To simplify our presentation, whenever we refer to a single training example we shall drop the example index everywhere and replace $T_i^{'}$ with just $T^{'}$.

To have a visual idea of how a tuple (E, S) of sequences are intertwined in time, let us consider two short and simple examples with $T = 3$ (so all sequences have a duration of 4 time units). Events and signals will have just two attributes: a letter and a duration in time

(upper case letters for events, lower case letters for signals).

- Example 1: Let $E = (A, A, C, B)$, and let $S = (b, a, c)$, with $l^{\langle 0 \rangle} = 2$, $l^{\langle 1 \rangle} = 1/2$, and $l^{\langle 2 \rangle} = 3/2$, as shown in figure 4.1.

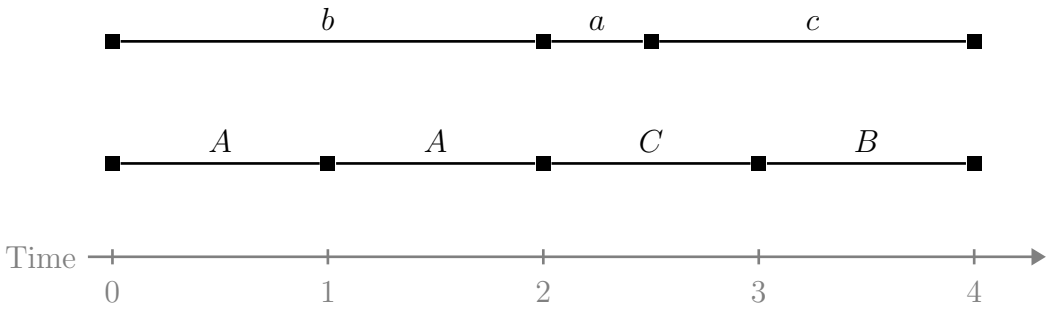


Figure 4.1: *Example 1 of time-structure relation between E and S.*

- Example 2: Let $E = (B, B, C, C)$, and let $S = (d, e, c, c, a, b)$, with $l^{\langle 0 \rangle} = 1/2$, $l^{\langle 1 \rangle} = 1/2$, and $l^{\langle 2 \rangle} = 1$, $l^{\langle 2 \rangle} = 1/2$, $l^{\langle 4 \rangle} = 1/2$, and $l^{\langle 5 \rangle} = 1$, as shown in figure 4.2.

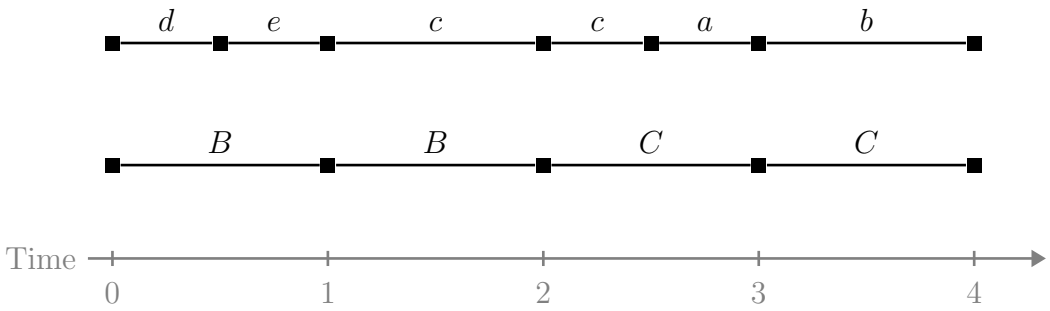


Figure 4.2: *Example 2 of time-structure relation between E and S.*

Naturally, events and signals can have more than two attributes, but the time attribute is the most important one for our formulation, since it is the one responsible for a relative time

relation and conditional dependence between the sequences. This is the motivation behind building an architecture whose connections vary from one training example to the other, which we will present in a subsequent section.

4.2 Relative Event Conditional Dependence

Coming back to the hypothetical system we imagined at the beginning of this chapter, if it was that system the one that produced the sequence of signals in figure 4.1, the information used in its decision process would be as follows. Let t denote time, and the event $e^{\langle k \rangle}$ is considered to be occurring in the time interval $[k, k+1)$.

- At $t = 0$: {previous decisions=$None$, past events=$None$, current event=A, future events=(A, C, B) } $\Rightarrow$ Decision: $s^{\langle 0 \rangle} = b$ with $l^{\langle 0 \rangle} = 2$.
- At $t = 2$: {previous decisions= ($s^{\langle 0 \rangle} = b$ with $l^{\langle 0 \rangle} = 2$), past events=$(A, A)$, current event=$C$, future events=$B$ } $\Rightarrow$ Decision: $s^{\langle 1 \rangle} = a$ with $l^{\langle 1 \rangle} = 1/2$.
- At $t = 5/2$: {previous decisions= ($s^{\langle 0 \rangle} = b$ with $l^{\langle 0 \rangle} = 2$, $s^{\langle 1 \rangle} = a$ with $l^{\langle 1 \rangle} = 1/2$), past events=$(A, A)$, current event=$C$, future events=$B$ } $\Rightarrow$ Decision: $s^{\langle 2 \rangle} = c$ with $l^{\langle 2 \rangle} = 3/2$.

Observe that each time t at which a decision is made, t equals the sum of the time durations of all previous signals generated up to that point. Moreover, at each decision, the sequences of past and future events, as well as the current event, are potentially different. This is what we refer to by relative event conditional dependence. Let us present a more formal definition, but first, we will need an integer that locates the index current event.

Definition 4.2.1. Consider a sequence $S = (s^{\langle 0 \rangle}, \ldots, s^{\langle T^{'} \rangle})$ of points $s^{\langle k \rangle}$ called signals, each with an associated duration in time $l^{\langle k \rangle}$. Let

$$l_k = \left\lfloor \sum_{j=0}^{k} l^{\langle j \rangle} \right\rfloor, \ \ 0 \leq k \leq T^{'} \tag{4.1}$$

be known as the **dynamic index** after k time-steps. For convenience, we define $l_{-1} = 0$.

The dynamic index at a time-step k equals the integer part of the total time duration of all the signals $(s^{\langle 0 \rangle}, \ldots, s^{\langle k \rangle})$. This index allows to express the relative event conditional dependence at each time-step as follows.

Definition 4.2.2. Given a tuple (E, S) with S as described above, and $E = (e^{\langle 0 \rangle}, \ldots, e^{\langle T \rangle})$ a sequence of points $e^{\langle k \rangle}$ called events each with an associated duration of 1 time unit. By **relative event conditional dependence** of a signal $s^{\langle k+1 \rangle}$ of S with respect to E, we refer to the conditional dependence of $s^{\langle k+1 \rangle}$ to the set consisting of

- the *current* event $e^{\langle l_k \rangle}$;
- the sequence of all *past* events $(e^{\langle 0 \rangle}, \ldots, e^{\langle l_k - 1 \rangle})$;
- the sequence of all *future* events $(e^{\langle l_k + 1 \rangle}, \ldots, e^{\langle T \rangle})$,

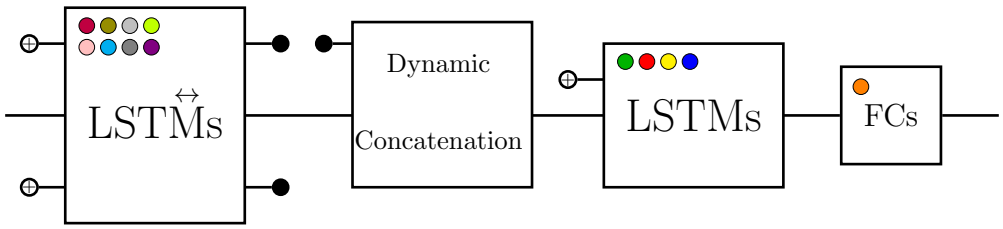
where l_k is the dynamic index.

The main goal of this chapter is to model sequences of signals of a dataset subject to a relative event conditional dependence, with the purpose of predicting or generating new sequences of signals given a particular sequence of events in a way that is aligned with the distribution of the dataset.

4.3 Proposed Architecture

Consider a dataset $X = \{(E^{(i)}, S^{(i)}) \mid i = 0, \ldots, m-1\}$ as described above, in which each point of $S^{(i)}$ has a relative event conditional dependence to $E^{(i)}$. We introduce an architecture consisting of two LSTM neural networks dynamically connected (i.e. the associated string diagram between LSTMs changes from one training example to the other), followed by a fully connected layer at each time-step. The dynamic connection between the LSTMs consists of a concatenation mechanism that aims to account for the relative event conditional dependence in each training example, as it depends directly on the sequence of dynamic indices $(l_0, \ldots l_{T_i-1})$ of each training example.

During training, the architecture can be drawn as the diagram

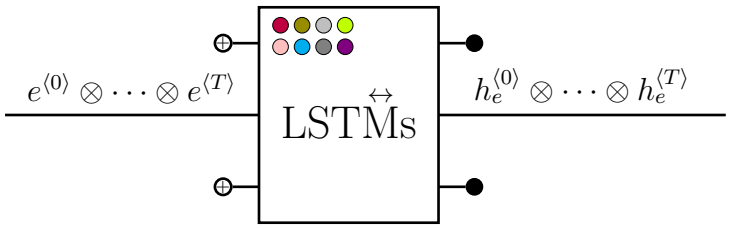


(4.2)

We make the distinction "during training" because during prediction or generation (once the weights are trained), the concatenation mechanism cannot be executed completely before the LSTMs node since it will depend on the time duration of every signal generated. This is possible during training because we know all time durations before hand, as we explain below.

Let us consider a single training example (E, S). The four nodes of the architecture displayed above are as follows.

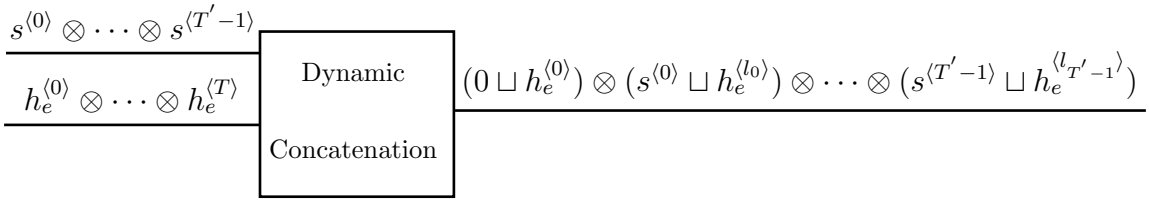
1. A **bidirectional LSTM** network (see diagram 3.18) takes a sequence $e^{\langle 0 \rangle} \otimes \cdots \otimes e^{\langle T \rangle}$ with zero-vectors as initial hidden states, and returns the (bidirectional) hidden states $h_e^{\langle 0 \rangle} \otimes \cdots \otimes h_e^{\langle T \rangle}$ and extra copies of the first and last hidden state that we will discard:



.

Ideally, the network will force each hidden state $h_e^{\langle j \rangle}$ to encode relevant information about event $e^{\langle j \rangle}$, the sequence of previous events $(e^{\langle 0 \rangle}, \ldots, e^{\langle j-1 \rangle})$, and the sequence of following events $(e^{\langle j+1 \rangle}, \ldots, e^{\langle T \rangle})$.

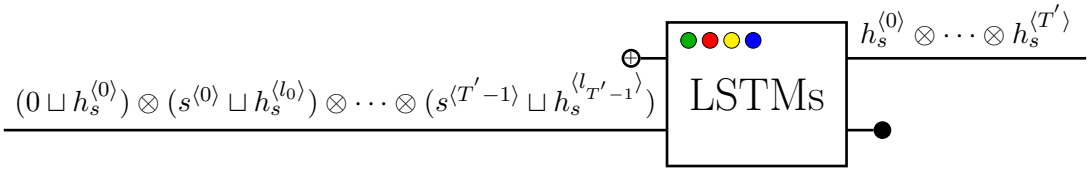
2. A **concatenation mechanism** takes a sequence $s^{\langle 0 \rangle} \otimes \cdots \otimes s^{\langle T^{'}-1 \rangle}$ and the hidden states $h_e^{\langle 0 \rangle} \otimes \cdots \otimes h_e^{\langle T \rangle}$, and concatenates to each signal $s^{\langle k \rangle}$ the hidden state $h_e^{\langle l_k \rangle}$ for $0 \leq k \leq T^{'} - 1$, where l_k is the dynamic index (see definition 4.2.1). It also generates the concatenation of a vector of zeros of the same size as $s^{\langle k \rangle}$ with the state $h_e^{\langle 0 \rangle}$:



.

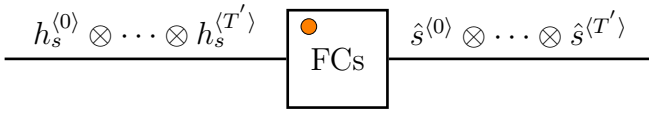
3. A **forward LSTM** network (see diagram 3.16) takes the output of the concatenation mechanism and returns hidden states $h_s^{\langle 0 \rangle} \otimes \cdots \otimes h_s^{\langle T^{'} \rangle}$. It takes as initial hidden state a

zero-vector, and we discard the extra copy of the last hidden state:



Ideally, each hidden state $h_s^{\langle k \rangle}$ will encode relevant information about the sequence of previous signals $s^{\langle 0 \rangle}, \ldots, s^{\langle k-1 \rangle}$, and, through $h_e^{\langle l_{k-1} \rangle}$, information about the proper relative event conditioning.

4. A **fully connected** layer (see diagram 3.1) takes every hidden state $h_s^{\langle k \rangle}$ and returns a signal prediction $\hat{s}^{\langle k \rangle}$:



.

Finally, to train the weights of the network, we minimize a loss function that compares the prediction $(\hat{s}^{\langle 0 \rangle}, \ldots, \hat{s}^{\langle T' \rangle})$ to the ground truth $(s^{\langle 0 \rangle}, \ldots, s^{\langle T' \rangle})$ for each training example, a standard method in statistical learning.

Let us illustrate how the concatenation mechanism looks like by examining two examples analogous to the examples in section 4.1. Again, let $T = 3$. Our goal is to illustrate what hidden states $h_e^{\langle k \rangle}$ are passed to the second LSTM network in order to account for the relative event conditional dependence. Recall that we are considering the architecture during training.

- Example 1: $l^{\langle 0 \rangle} = 2$, $l^{\langle 1 \rangle} = 1/2$, $l^{\langle 2 \rangle} = 3/2$ (see figure 4.3). Observe that at $t = 0$ there are no previous signals, hence the zero-vector concatenated to $h_e^{\langle 0 \rangle}$ in order to generate a first signal prediction $\hat{s}^{\langle 0 \rangle}$. The dynamic indices are $l_0 = \lfloor l^{\langle 0 \rangle} \rfloor = 2$, $l_1 = \lfloor l^{\langle 0 \rangle} + l^{\langle 1 \rangle} \rfloor = 2$, so $h_e^{\langle l_0 \rangle} = h_e^{\langle 2 \rangle}$, $h_e^{\langle l_1 \rangle} = h_e^{\langle 2 \rangle}$.

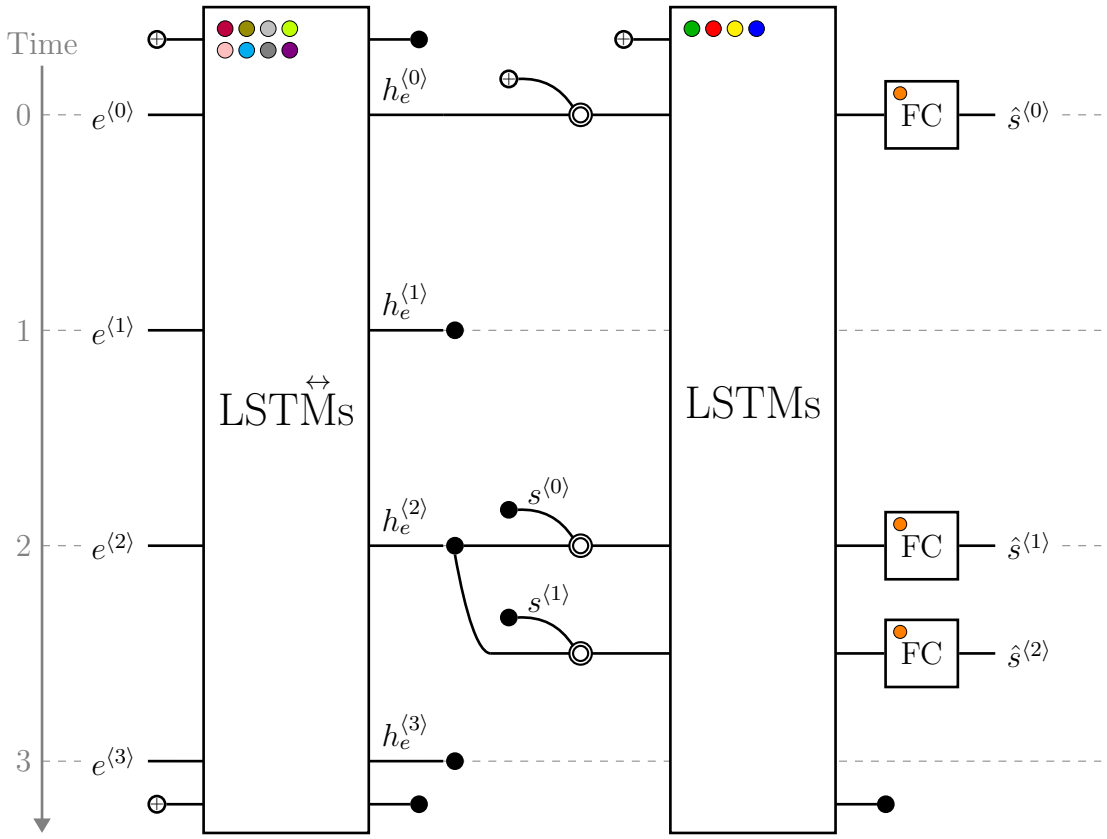


Figure 4.3: *Dynamic concatenation example 1.*

- Example 2: $l^{\langle 0 \rangle} = 1/2$, $l^{\langle 1 \rangle} = 1/2$, $l^{\langle 2 \rangle} = 1$, $l^{\langle 2 \rangle} = 1/2$, $l^{\langle 4 \rangle} = 1/2$, $l^{\langle 5 \rangle} = 1$ (see figure 4.4). The dynamic indices are $l_0 = \lfloor 1/2 \rfloor = 0$, $l_1 = \lfloor 1/2 + 1/2 \rfloor = 1$, $l_2 = \lfloor 1/2 + 1/2 + 1 \rfloor = 2$, $l_3 = \lfloor 1/2 + 1/2 + 1 + 1/2 \rfloor = 2$, $l_4 = \lfloor 1/2 + 1/2 + 1 + 1/2 + 1/2 \rfloor = 3$. Thus, $h_e^{\langle l_0 \rangle} = h_e^{\langle 0 \rangle}$,

$h_e^{\langle l_1 \rangle} = h_e^{\langle 1 \rangle}$, $h_e^{\langle l_2 \rangle} = h_e^{\langle 2 \rangle}$, $h_e^{\langle l_3 \rangle} = h_e^{\langle 2 \rangle}$, $h_e^{\langle l_4 \rangle} = h_e^{\langle 3 \rangle}$.

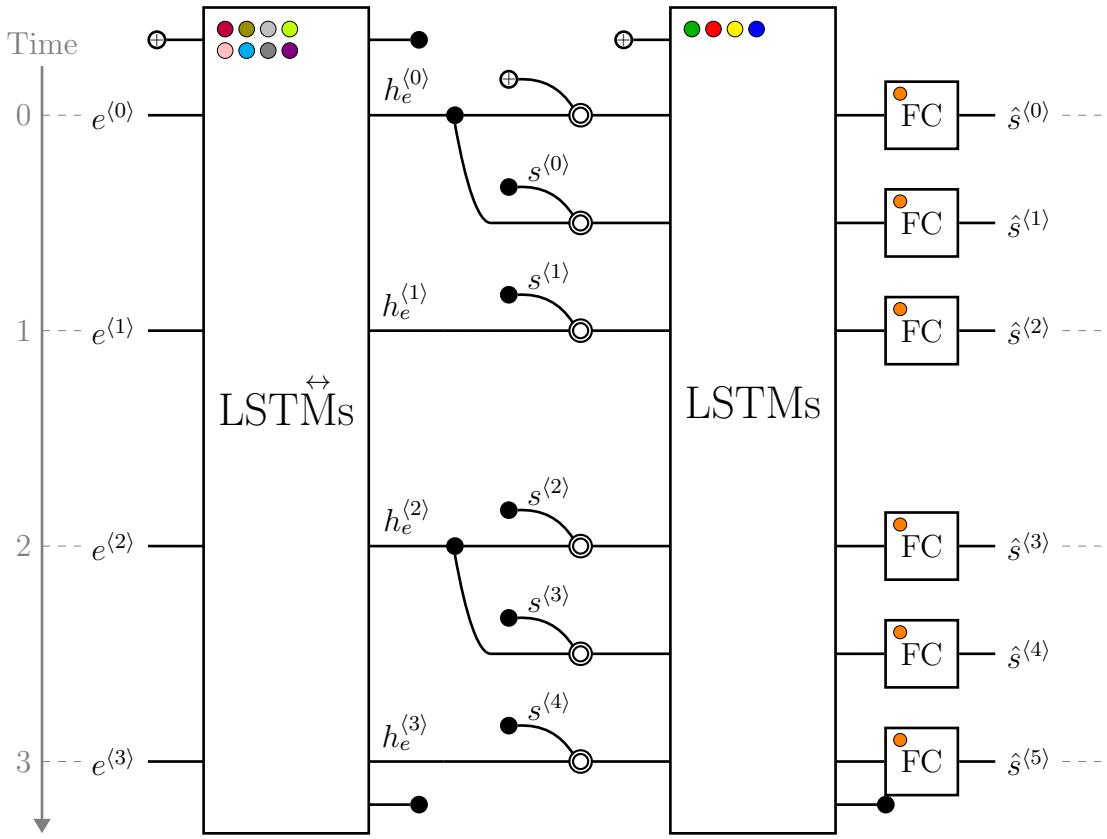


Figure 4.4: *Dynamic concatenation example 2.*

During prediction/generation, once the model is trained, the predictions $\hat{s}^{\langle k \rangle}$ should replace $s^{\langle k \rangle}$ as the inputs of the second LSTM, and each duration $l^{\langle k \rangle}$ should be now the time duration of the prediction $\hat{s}^{\langle k \rangle}$. This is the reason why, during prediction/generation, the concatenation node cannot be written completely before the second LSTM, since we need to "wait" for the next predicted signal to know its duration in time, in order to concatenate the adequate hidden state $h_e^{\langle l_k \rangle}$ in the next time-step.

Summarizing, the architecture is designed to account for the following dependence. A prediction $\hat{s}^{\langle k \rangle}$ is obtained from the hidden state $h_s^{\langle k+1 \rangle}$. In turn, the hidden state $h_s^{\langle k+1 \rangle}$ is obtained from $h_s^{\langle k \rangle}$, $s^{\langle k \rangle}$, and $h_e^{\langle l_k \rangle}$.

- $h_s^{\langle k \rangle}$ and $s^{\langle k \rangle}$ account for the sequence of previous signals $(s^{\langle 0 \rangle}, \ldots, s^{\langle k \rangle})$. (The state $h_s^{\langle k \rangle}$ also encodes some information about previous events, something that can be used to implement a variation of the architecture, as explained in the next section.)
- $h_e^{\langle l_k \rangle}$ accounts for the relative event conditional dependence.

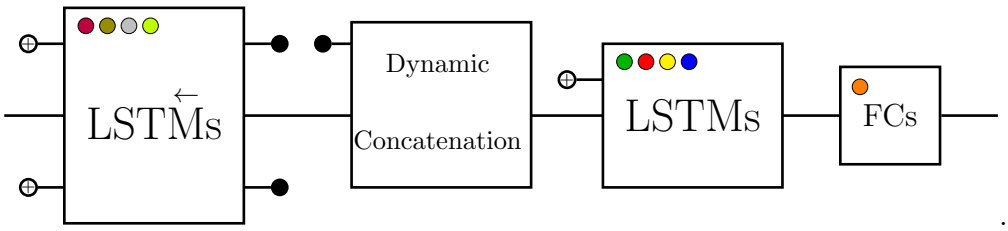
This is the motivation behind having designed the presented architecture, in order to model the sequence problem described at the beginning of the chapter.

4.4 Variations

In this section we explore some possible small variations of the proposed architecture above that can also model our sequence problem.

4.4.1 A Backward LSTM instead of Bidirectional

As we mentioned in the previous section, if we observe the connectivity of the architecture, a hidden state $h_s^{\langle k \rangle}$ also encodes information about the event structure due to the concatenation of the states $h_e^{\langle l_0 \rangle}, \ldots, h_e^{\langle l_{k-1} \rangle}$ in the previous time-steps, at least in a less direct way. We could take advantage of this fact and replace the bidirectional LSTM encoding the event structure by a backward LSTM (see diagram 3.17), modifying the architecture as

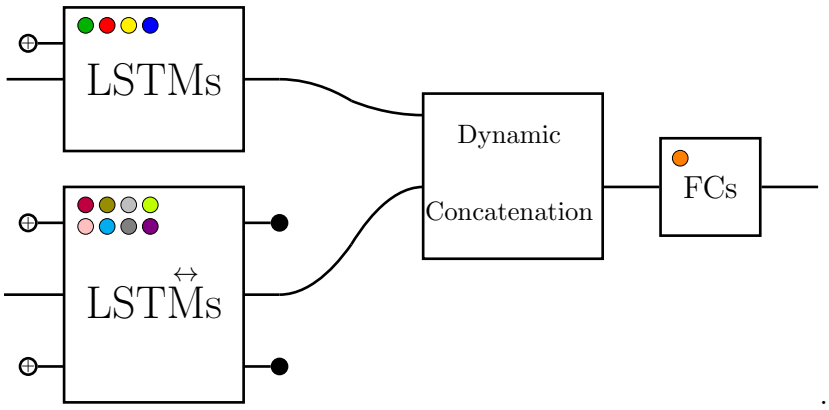


Here, a hidden state $h_e^{\langle l_k \rangle}$ encodes information only about the current event $e^{\langle l_k \rangle}$ and following events, but the concatenation mechanisms and structure of the second LSTM makes a hidden state $h_s^{\langle k \rangle}$ encode information about past events as well.

In addition to the implementation of the original proposed architecture trained on jazz solos, a version of the variation presented above was implemented as well, yielding equally promising results.

4.4.2 Concatenation after both LSTMs

Another possibility is to first have both LSTMs and then concatenate their hidden states to be passed to the fully connected layer, as given by the diagram

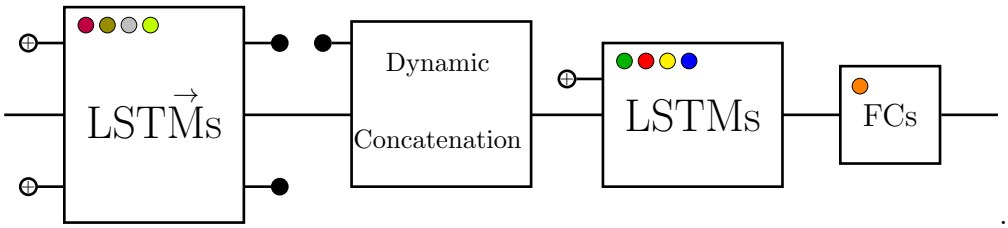


Here, the (forward) LSTM encodes only the sequential information of S without any conditioning, while the bidirectional LSTM encodes E. Then, the hidden state $h_s^{\langle k \rangle}$ is concatenated with $h_e^{\langle l_k \rangle}$ to properly condition the production of the prediction $\hat{s}^{\langle k+1 \rangle}$.

This variation has not been implemented by the author, but we suspect that similar results could be achieved.

4.4.3 Online Prediction

An online prediction (in real time) could be trained if we relax the conditioning to be only current and previous events. This scenario could be possible whenever we would like to predict signals in real time as new events occur. We can tackle this variation of the problem by replacing the bidirectional LSTM by a (forward) LSTM, as in



During prediction/generation, the event LSTM could process events as they come, and again, the concatenation would have to be executed one at a time. This variation is convenient when future events are unknown.

4.5 Trainability

Some of the methods most commonly used to train the weights of a neural network are gradient-based optimization algorithms. In general, a loss function comparing a prediction

against the ground truth (and usually some additional penalties such as regularization) is minimized with respect to all trainable weights by using some version of a gradient-descent step minimizer. For this, we first need to be able to calculate the gradient of the loss function with respect to the trainable weights by means of the chain rule (a.k.a. back-propagation). The diagram of our architecture changes from example to example due to the dynamic concatenation mechanism which depends on the set of signal durations $\{l_s^{\langle k\rangle(i)} \mid k = 0, \ldots, T_i\}$ of each example i. This implies that the loss function, and hence, its derivatives with respect to the weights, will have a different explicit form for each example, making every gradient step different in form. However, the set of trainable weights does not change from example to example (shared weights), so, in principle, we should be able to update the same set of weighs for all examples with some degree of consistency. Nevertheless, it is not always clear if the value of a loss function associated to such a dynamic architecture can actually be decreased consistently achieving a sufficiently small value. For this reason, in the next chapter we present a prototype of this architecture on jazz music data as an experiment and as proof of concept. Early results show that a loss function can indeed be decreased consistently and smoothly, with promising musical results.

4.6 Vectorizing the Concatenation Mechanism

When training our network, executing the dynamic concatenation mechanism for each example and back-propagating through it can be very costly computationally, specially if it is implemented using for-loops. For this reason, it would be desirable to vectorize (see section

3.7) this mechanism and perform it by means of matrix multiplications. Fortunately, during training, we have all the durations of the signals in each training example, so we know before hand the indices of the hidden states that should be concatenated to the signals. Then, in practice, we can vectorize the concatenation mechanism for each example as follows.

As we saw in section 4.3, the concatenation mechanism takes $s^{\langle 0\rangle} \otimes \cdots \otimes s^{\langle T'-1\rangle}$ and $h_e^{\langle 0\rangle} \otimes \cdots \otimes h_e^{\langle T\rangle}$ and yields the concatenations $(0 \sqcup h_s^{\langle 0\rangle}) \otimes (s^{\langle 0\rangle} \sqcup h_s^{\langle l_0\rangle}) \otimes \cdots \otimes (s^{\langle T'-1\rangle} \sqcup h_s^{\langle l_{T'-1}\rangle})$. Let the the hidden states and the signals be arranged as rows of the matrices H_e and S_{in}, respectively, and let the first row of S_{in} be all zeros, that is,

$$H_e = \begin{bmatrix} - \; h_e^{\langle 0\rangle} \; - \\ \vdots \\ - \; h_e^{\langle T\rangle} \; - \end{bmatrix} \quad , \quad S_{\text{in}} = \begin{bmatrix} 0 \\ - \quad s^{\langle 0\rangle} \quad - \\ \vdots \\ - \; s^{\langle T'-1\rangle} \; - \end{bmatrix} \quad .$$

Recall that $T^{'}$ can be different for each training example, so the number of rows of S_{in} will be variable, while the number of rows of H_e is fixed for all examples. Our goal is to output the matrix

$$\begin{bmatrix} - \quad 0 \sqcup h_s^{\langle 0\rangle} \quad - \\ - \quad s^{\langle 0\rangle} \sqcup h_e^{\langle l_0\rangle} \quad - \\ \vdots \\ - \; s^{\langle T^{'}-1\rangle} \sqcup h_e^{\langle l_{T'-1}\rangle} \; - \end{bmatrix} \quad ,$$

which can also be written as the matrix

$$\begin{bmatrix} - \quad 0 \sqcup h_e^{\langle 0\rangle} \quad - \\ - \quad s^{\langle 0\rangle} \sqcup h_e^{\langle l_0\rangle} \quad - \\ \vdots \\ - \; s^{\langle T^{'}-1\rangle} \sqcup h_e^{\langle l_{T'-1}\rangle} \; - \end{bmatrix} = \begin{bmatrix} 0 \quad - \quad h_e^{\langle 0\rangle} \quad - \\ - \quad s^{\langle 0\rangle} \quad - - \quad h_e^{\langle l_0\rangle} \quad - \\ \vdots \\ - \; s^{\langle T'-1\rangle} \; - - \; h_e^{\langle l_{T'-1}\rangle} \; - \end{bmatrix} = \begin{bmatrix} S_{\text{in}} & H_l \end{bmatrix}$$

where

$$H_l = \begin{bmatrix} - & h_e^{\langle 0 \rangle} & - \\ - & h_e^{\langle l_0 \rangle} & - \\ & \vdots & \\ - & h_e^{\langle l_{T'-1} \rangle} & - \end{bmatrix} .$$

One way of getting this output via matrix multiplications is to construct a matrix L_S for each training example such that $L_S H_e = H_l$, and then concatenate $L_S H_e$ to S_{in}. We could have all matrices L_S constructed and stored before training begins, and use them as the dynamic concatenation mechanism after each pass through the bidirectional LSTM (the one that outputs H_e). The matrices L_S that we need are the following.

Definition 4.6.1. Let $\{l_0, l_1, \ldots, l_{T'}\}$ be the set of dynamic indices of a sequence S of signals subject to a sequence E of $T+1$ events. The **placing matrix** L_S associated to S is the matrix given by

$$(L_S)_{ij} = \begin{cases} 1 & \text{if} \quad i = j = 0; \\ 1 & \text{if} \quad j = l_{i-1},\ i \geq 1; \\ 0 & \text{otherwise;} \end{cases}$$

where $0 \leq i \leq T'$, $0 \leq j \leq l_{T'-1}$.

Claim: $L_S H_e = H_l$ if L_S is the placing matrix.

Proof: Observe that L_S is a $(0,1)$-matrix in which each row has exactly one 1, therefore, multiplication from the left by L_S selects out the j-th row of H_e. The first row ($i = 0$) of L_S has its 1 in the first position, so the first row $h_e^{\langle 0 \rangle}$ of H_e will always be selected to be the first row of H_l. For $i \geq 1$, L_S will select out the (l_{i-1})-th row of H_e, as desired. $\square$

Let us construct the placing matrix for the second example in section 4.3, where we have the dynamic indices $l_0 = 0$, $l_1 = 1$, $l_2 = 2$, $l_3 = 2$, $l_4 = 3$. Thus, the placing matrix is given

$$L_S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

Then, we have that

$$L_S H_e = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_e^{\langle 0 \rangle} \\ h_e^{\langle 1 \rangle} \\ h_e^{\langle 2 \rangle} \\ h_e^{\langle 3 \rangle} \end{bmatrix} = \begin{bmatrix} h_e^{\langle 0 \rangle} \\ h_e^{\langle 0 \rangle} \\ h_e^{\langle 1 \rangle} \\ h_e^{\langle 2 \rangle} \\ h_e^{\langle 2 \rangle} \\ h_e^{\langle 3 \rangle} \end{bmatrix} ,$$

which indeed corresponds to the concatenation connections in figure 4.4.

Each placing matrix depends on the set of time durations of a sequence of signals, and since during training we know the time durations of the sequences in the dataset, we are able to construct these matrices before we start training. We found that implementing the concatenation mechanism as a for-loop or as if-statements slowed down the training considerably. By using the approach of placing matrices, we were able to decrease the training time by roughly 86% in the implementation presented in the next chapter.

CHAPTER 5
A GENERATIVE MODEL OF JAZZ SOLO IMPROVISATIONS

5.1 Introduction

The goal of this chapter is to present a prototype example of a potential use that the architecture proposed in section 4 could have. Here, we present an implementation trained over a corpus of jazz solo transcriptions. The full Python/PyTorch code is included in the appendix of this manuscript.

The corpus consists of 48 Charlie Parker solo transcriptions. The transcriptions are in the form of lead sheets: they contain the solo notes and the chord symbols of the harmonic progression. Figure 5.1 shows the first seven measures of a solo transcription in lead sheet format.

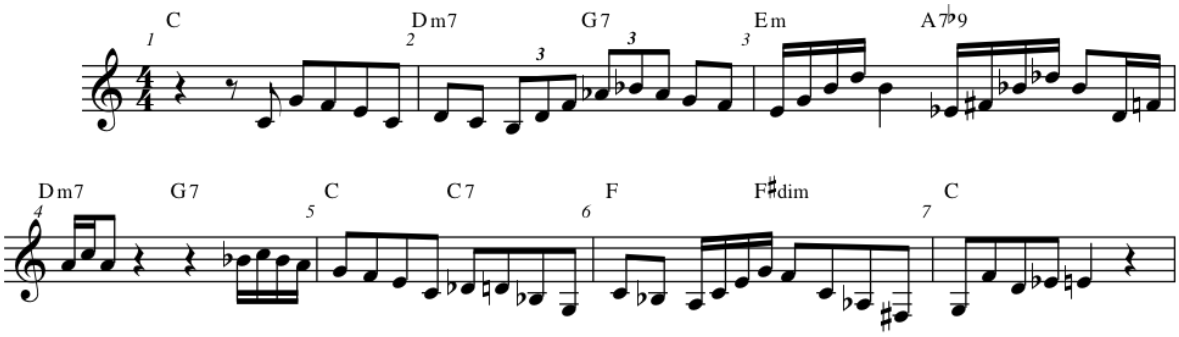


Figure 5.1: *Lead sheet of a solo transcription.*

These type of transcriptions are usually created by a human who, by listening to a recording of Parker playing together with other musicians, annotates the notes played by Parker during his solo. The transcriber also deduces the chord progression by listening to the

harmony played by one or some of the instruments, and/or it might be a musical piece with a well known chord progression that the transcriber already knows or to which she has access. Sometimes, because of several reasons such as poor recording, instruments overlapping too much, etc., there cannot be a 100% confidence about the accuracy of the transcribed notes as compared to the actual notes played. However, such ambiguities are usually very few and/or not too significant.

Moreover, when a jazz band plays a piece, it is understood that the chord progression is a shared mental scheme that the musicians will navigate together through time, whether because the chord progression was explicitly stated, or because they share a strong common background and tradition that allows them to do so. This mental scheme serves as a central harmonic structure that the musicians deviate from and gravitate towards in their improvisations. The annotation of the chord symbols in a solo transcription intends to serve as a representation of such a mental structure. This structure admits an infinite number of possible interpretations in the sound space, as long as they fall close enough to the accepted practices of the style.

Hence, it is worth having in mind that a solo transcription is not an embodiment of a solo interpretation in the sound space, but a symbolic description of such an interpretation, projected from a sound space onto a symbolic space of mental schemes. Here, our implementation is trained on and generates music in the symbolic musical space.

5.2 Information about the Corpus

As mentioned above, the corpus consists of 48 Charlie Parker solo transcriptions. There is a total of 2479 measures of music (not counting measures with only rests). The shortest uninterrupted solo improvisation is 4 measures long (probably from trading 4's with other musician), while the longest is 131 measures long. The lowest pitch found in the corpus is 34 (given as a midi pitch number), while the highest is 80, although pitches 35 and 36 are not present. There is a total of 41 distinct time durations; in ascending order and given as a fraction of a beat, the set of time durations types is {1/12, 1/8, 1/6, 1/5, 1/4, 1/3, 3/8, 5/12, 1/2, 7/12, 5/8, 2/3, 17/24, 3/4, 5/6, 7/8, 11/12, 1, 7/6, 6/5, 5/4, 4/3, 11/8, 3/2, 19/12, 5/3, 7/4, 11/6, 15/8, 3/2, 19/12, 5/3, 7/4, 11/6, 15/8, 2, 7/3, 5/2, 8/3, 17/6, 3, 27/8, 7/2, 4, 9/2, 5, 11/2}.

5.3 Conditioning Context of a Solo Improvisation

When improvising, there are several factors that might influence a musician's solo, such as the melody of the piece, other musicians' executions and ideas, real-time interactions with other musicians, etc. One factor that is of great importance is the chord progression. In tonal jazz styles, the chord progression influences a musician's note choices in the sense that her melodic line, contrasted with the chord harmonies, produces a harmonic color that she intended to produce. With this in mind, a chord progression can be seen as a sequence of events that condition a sequence of signals (the notes and rests). The time duration of every chord symbol in the Charlie Parker corpus is an integer, so a chord that has a duration

$m > 1$ can be broken down into m consecutive identical chords of duration 1.

A note symbol has two main attributes: a pitch and a duration in time. There are some extra attributes that a note can have, such as volume, articulation, and others, but for our model we will only require pitch and time duration. The time duration attribute allows us to regard an instrument that produces only one note at a time (such as an alto sax) as a system that produces a sequence of signals. A rest, which also has a time duration, can be regarded as a signal as well. Thus, each solo transcription in our corpus can be framed as a sequence of signals conditioned to a context comprised of a sequence of events, such as the problem formulated in section 4.1.

The pitch of a note will be given as a midi pitch (an integer from 0 to 127). We will assign to rests the integer 128, which we will regard as the silent pitch. The time duration of a note will be given in quarter length, that is, the number of beats that a it lasts. For example, consider the phrase shown in figure 5.2. Each chord symbol has a duration of 2 beats. The first note F, has pitch 65 and duration 2; the second note A has pitch 69 and duration 1/2, and the third note G has pitch 67 and duration 3/2. Therefore we can regard this music phrase as the sequence of signals $S = (s^{\langle 0 \rangle}, s^{\langle 1 \rangle}, s^{\langle 1 \rangle}) = (65, 69, 67)$ with $l^{\langle 0 \rangle} = 2, l^{\langle 1 \rangle} = 1/2, l^{\langle 2 \rangle} = 3/2$, conditioned to the sequence of events $E = (e^{\langle 0 \rangle}, e^{\langle 1 \rangle}, e^{\langle 2 \rangle}, e^{\langle 3 \rangle}) = (\text{Dm7}, \text{Dm7}, \text{G7}, \text{G7})$. Figure

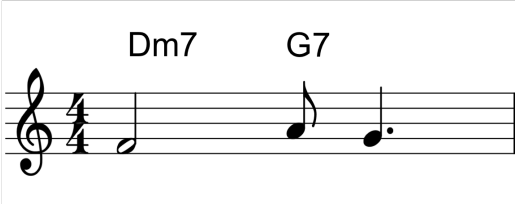


Figure 5.2: *Music phrase example I.*

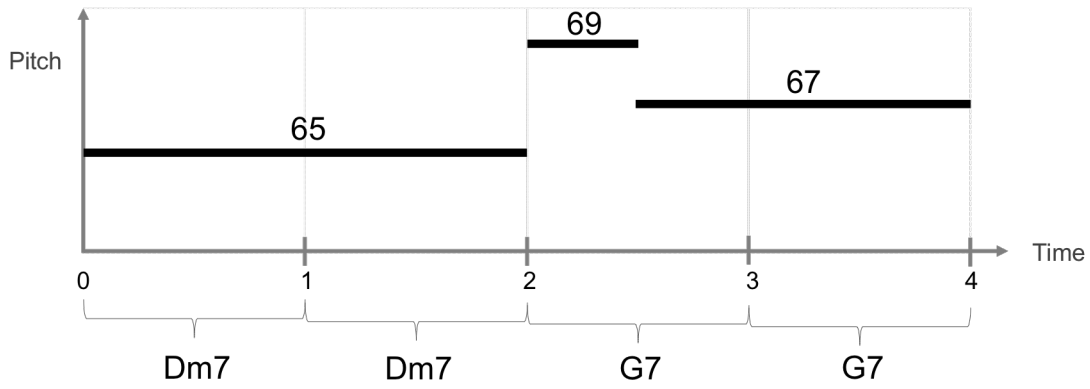


Figure 5.3: *Music phrase example 1 as signals and events.*

5.3 shows a depiction of (E, S) in a similar format to the examples displayed in figures 4.1 and 4.2 in section 4.1.

Observe that the sequence of durations $(l^{\langle 0\rangle}, l^{\langle 1\rangle}, l^{\langle 2\rangle})$ is the same as in example 1 in sections 4.1 (figure 4.1) and 4.3. Therefore, if our architecture were to be trained on this music example, its string diagram would look like the diagram in figure 4.3.

A music phrase that would induce a string diagram like the one in figure 4.4 is shown in figure 5.4. In this example we have that $E = (\mathrm{G7}, \mathrm{G7}, \mathrm{C}, \mathrm{C})$ and $S = (77, 74, 71, 128, 65, 67)$ with $l^{\langle 0\rangle} = 1/2$, $l^{\langle 1\rangle} = 1/2$, $l^{\langle 2\rangle} = 1$, $l^{\langle 2\rangle} = 1/2$, $l^{\langle 4\rangle} = 1/2$, $l^{\langle 5\rangle} = 1$. This example has the same sequence of time durations as example 2 in sections 4.1 (figure 4.2) and 4.3.

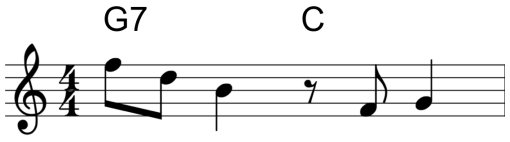


Figure 5.4: *Music phrase example II.*

5.4 Constructing the Dataset

Having framed notes as signals and chords as conditioning events, we would like to use the corpus of solo transcriptions to construct a suitable dataset that our neural network architecture from section 4.3 can ingest. Recall from section 4.1 that a dataset for our architecture is a set $X = \{(E^{(i)}, S^{(i)}) \mid i = 0, \ldots, m-1\}$ where $E^{(i)} = (e^{\langle 0 \rangle (i)}, \ldots, e^{\langle T \rangle (i)})$, $S^{(i)} = (s^{\langle 0 \rangle (i)}, \ldots, s^{\langle T_i' \rangle (i)})$, T is fixed, T_i' is variable, and each event has a duration of 1 time unit.

We begin by fixing the number of events in each training example (or equivalently, the time duration of each training example). Since the shortest solo phrase in our corpus is 4 measures long (16 beats), we have chosen $T = 15$. Bear in mind that this choice is flexible and somewhat arbitrary, any $T < 15$ is also possible, and $T > 15$ is possible by simply disregarding solos with a duration less than $T + 1$ beats. (We experimented with $T = 3, 7$ and 15, with the first two yielding results that adhere better to the chord harmony, and the later yielding more satisfying results in terms of melodic contour.)

After choosing $T = 15$, we decided that any four consecutive full measures would constitute a training example. For example, in one solo transcription with a duration greater than 4 measures, measures 1-4 comprise one training example, measures 2-5 comprise another training example, and so on. In our code we named these solo snips *training windows*. Figure 5.5 shows an example of how these windows are selected. There is a total of 2143 training windows in the Parker corpus. The number of notes/rests in each training window varies from 8 to 81.

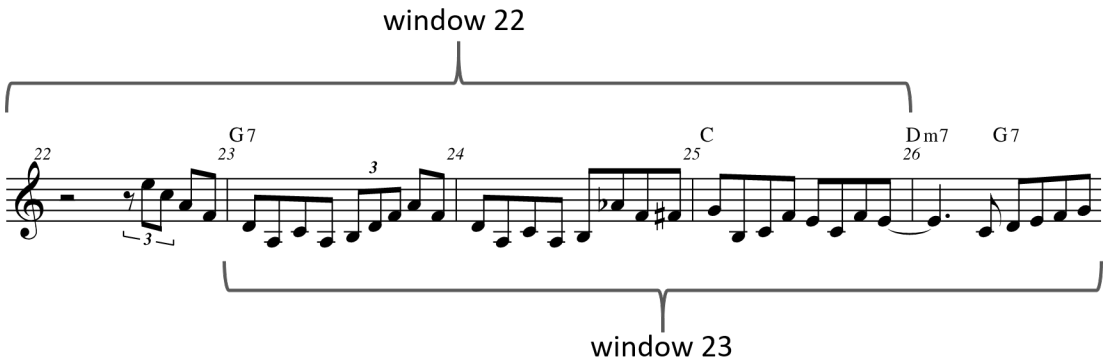


Figure 5.5: *Example of selection of training windows.*

Next, we choose a vector representation for the notes and the chords. A standard practice in deep learning is the use of the vector representation known as one-hot encoding, which consists of representing each data point as a $(0, 1)$-vector with exactly one 1. Each axis (or entry) represents a class, so each vector has a 1 in the entry corresponding to the class to which it belongs. This vector representation orthogonalizes (and normalizes) the different classes of data points. We will use a similar representation, with the difference that we will allow more than one 1 in a vector, so the vector classes will be orthogonal but not normal. (We will use the word "class" in the sense of machine learning classification, and not in the sense of equivalence classes.)

A vector representing a note will have two 1's, one indicating the pitch and one indicating the duration type (and zeros elsewhere). In order to keep the model small, we won't consider all 128 possible pitches and all possible time durations available in music notation. Instead, we will consider only the range of pitches from the lowest to the highest pitch used in the corpus, and the set of time duration types found in the corpus. Note vectors will have a size

of 1×89:

- a 1 in the first $80-34+1 = 47$ entries indicates the pitch, where the entry 0 corresponds to the pitch 34, and the entry 46 corresponds to the pitch 80;
- a 1 in entry 47 indicates pitch 128 or rest;
- a 1 in the other 41 remaining entries indicates the time duration type.

For example, the vector representation of a note with pitch 69 and duration 1/2 would look like the vector in figure 5.6.

A vector representing a chord will have 24 entries (from 0 to 23). The first 12 entries will correspond to the note letter of the root of the chord. The last 12 entries will correspond to the letters of the chord tones (including the root). There can only be one 1 in the first 12 entries, since there is only one root, but the last 12 entries accept any number of 1's. For example, the chord G7 has for root the note letter G, and chord tones G, B, D, F; figure 5.7 shows its vector representation. In the jazz tradition, the chord tone letters associated to

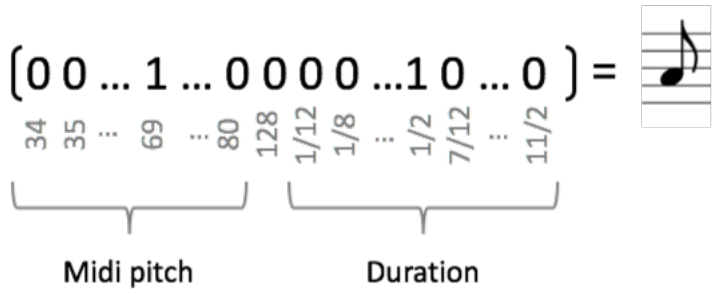


Figure 5.6: *Example of the vector representation of a note.*

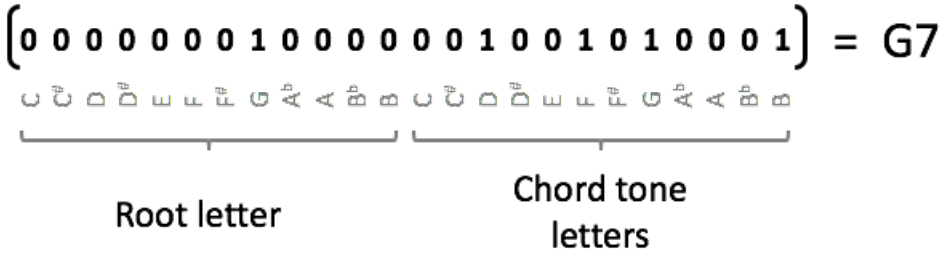


Figure 5.7: *Example of the vector representation of a chord.*

a chord symbol is somewhat flexible (but still logical). For our python implementation, we used the chord tones that the python library Music21 assigns to chord symbols.

The Pre-processing Module in the appendix of this manuscript takes a directory with lead-sheet solo transcription in xml format, and returns a 3-dimensional number array of events, a list of matrices of signals, a list of duration types, the minimum pitch, and the maximum pitch. The array of events has size $2143 \times 16 \times 24$, which are the number of training windows, the number of events (chords) in each example, and the size of each chord vector, respectively. The list of matrices of signals has 2143 matrices S_i of size $T_i^{'} \times 89$, where $T_i^{'}$ is the number of signals (notes and rests) in the i-th training window. The module also prints out information about the corpus, such as the information in section 5.2.

5.5 Training the Neural Network

Once we build the dataset, we are set to train the architecture 4.2. The Training Module in the appendix contains a python code implementation of the training algorithm. Before beginning the training algorithm, we use the deep-learning framework PyTorch to define

the architecture using its neural network modules, as well as the loss function, optimizer, regularization, and scheduler. One advantage that PyTorch offers is that its automatic differentiation module handles dynamic architectures, such as ours, extremely well [24]. From a general perspective, the training process is as follows.

1. Feed the dataset to the network to compute the predictions $\{\hat{s}^{\langle k \rangle (i)}\}_{k=0,\ldots,T_i'}^{i=0,\ldots,m-1}$ for all training windows, where m is the number of training windows.
2. Pass the predictions and ground truths $\{\hat{s}^{\langle k \rangle (i)}, s^{\langle k \rangle (i)}\}_{k=0,\ldots,T_i'}^{i=0,\ldots,m-1}$ to a loss function that compares them.
3. Find the gradient of the loss function with respect to all trainable parameters of the network.
4. Update all trainable parameters by performing a gradient step towards a minimizer of the loss function.
5. Repeat until the optimization criterion is met.

Now, there are many choices available for the activation functions of the network, hidden state sizes, loss function, optimizer, learning rate, etc. Below, we share some of the settings used when training the network on the Parker corpus.

- The activation functions inside the LSTMs are the ones by default, as described in chapter 3.
- The activation function of the fully connected layer (the last node of the architecture) is the Sigmoid function.

– The loss function is the Binary Cross Entropy, which, without taking into account the regularization term, is the function

$$\mathcal{L} = -\frac{1}{m}\sum_{i=0}^{m-1}\sum_{k=0}^{T_i'}\left[s^{\langle k\rangle(i)}\log\hat{s}^{\langle k\rangle(i)} + (1-s^{\langle k\rangle(i)})\log\left(1-\hat{s}^{\langle k\rangle(i)}\right)\right] \quad ,$$

where 1 is a vector of all 1's.

– The Adam optimizer as the gradient optimization algorithm (see [16]).

Since our implementation was trained on a personal computer, there was no systematic search for good values of the hyper-parameters such as the learning rate and hidden state sizes. Rather, we experimented with a few and selected the ones that yielded promising musical results. Figure 5.8 shows two examples of the graph of the loss function versus the number of gradient steps. The graph on the left corresponds to a learning rate of 0.0002, while the one in the right corresponds to a learning rate of 0.0005. Both trainings used a cosine scheduler, a weight decay of 1e-8, hidden sizes of 48 and 128 for the first and second LSTM respectively, and the same seed for the random initialization of the network weights.

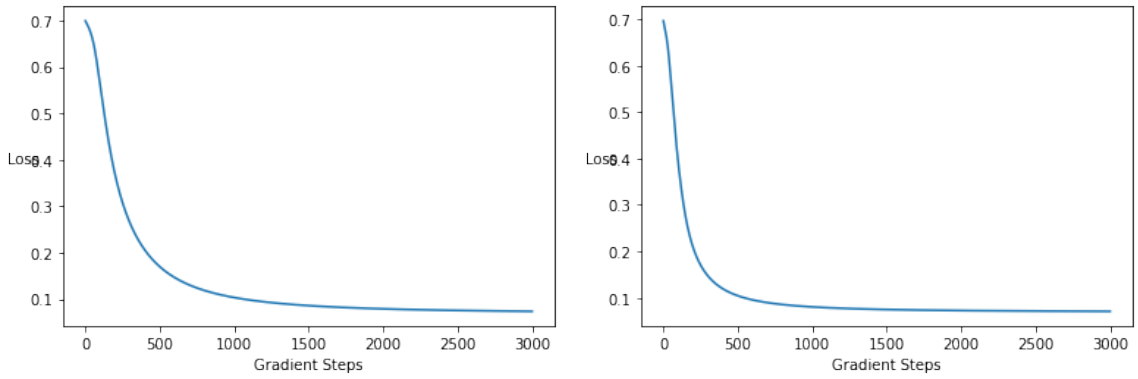


Figure 5.8: *Loss function values during training with randomly initialized weights.*

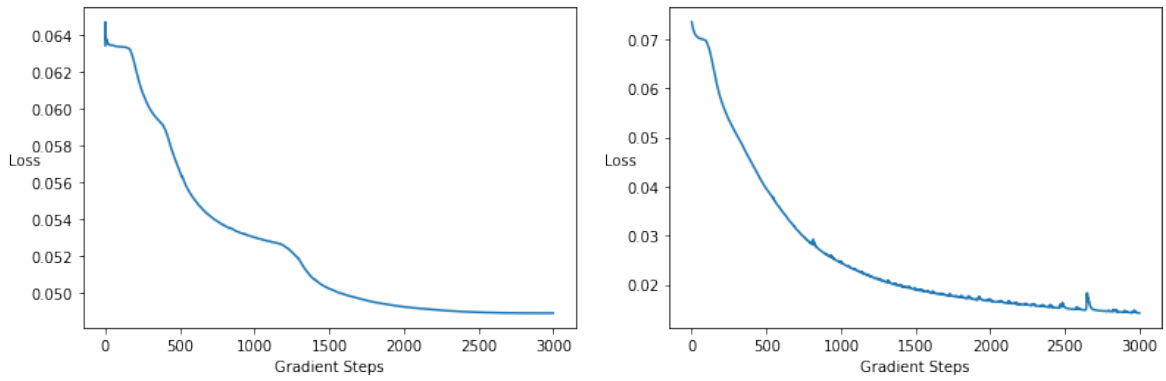


Figure 5.9: *Loss function values during training with pre-trained weights.*

Due to limited computing power, after the first 3000 gradient steps, we stored the weights and started a new iteration initializing the weights as the previously stored values (so the weights are not randomly initialized this time). Figure 5.9 shows two examples of such trainings. In the graph on the left the same cosine scheduler was used, while in the graph on the right we swithed to a step LR scheduler (every 300 steps by a factor of 0.5). This process can be repeated many times in the attempt to keep decreasing the value of the loss function.

As we can see, the learning curves seem smooth enough and the value of the function can be consistenly minimized, as desired.

5.6 Generating New Solos

Once the weights of the network are trained, we can use the architecture to generate new sequences of signals subject to sequences of events. The goal is to pass to the network a chord progression of our choosing and produce a sequence of notes/rests subject to that chord progression. The Generative Module in the appendix contains the python code implementation

of the process described below.

There are two main modifications we need to make to our architecture in order to convert it into a generative model. First, we modify the activation function of the fully connected layer: instead of the Sigmoid function, we apply a Softmax function to the first 48 entries of the vector (corresponding to pitch/rest), and another Softmax function to the last 41 entries (corresponding to the time duration type). For a vector $x = [x_1, \ldots, x_n]$, the Softmax of x is defined as the vector

$$\text{Softmax}(x) = \left[\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \right] .$$

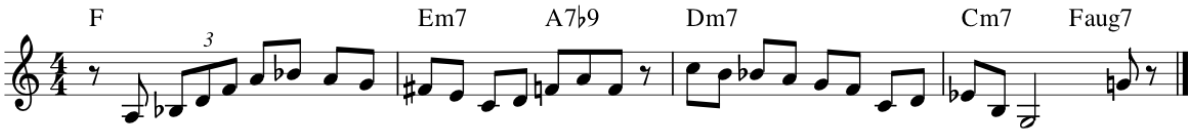
By doing this, we will create categorical probability distributions for the pitch/rest and the time duration so we can then sample from them, instead of just selecting the most likely values for each attribute. This improves the variability of the sequences generated.

The second modification is the concatenation mechanism. During generation, there is no ground truth for the signals, hence, as mentioned in section 4.3, the concatenation mechanism cannot be executed all before the second LSTM (since we don't know before-hand the set of dynamic indices of the whole sequence to be generated). Instead, for every signal we generate, we get its duration to compute the dynamic index, and then in the next time-step concatenate the corresponding hidden state from the first LSTM.

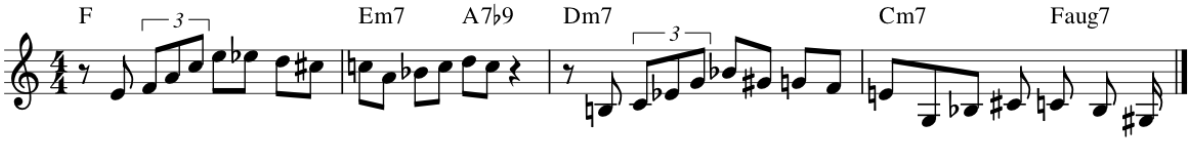
In order to generate a new sequence of notes, we must provide a chord progression as conditioning context that will be ingested by the network in vector form. The solo phrase generated will hopefully "work well" with the chord progression. For example, we passed

to a trained network the chord progression | F | Em7 A7b9 | Dm7 | Cm7 Faug7 |, which corresponds to the sequence of events (F, F, F, F, Em7, Em7, A7b9, A7b9, Dm7, Dm7, Dm7, Dm7, Cm7, Cm7, Faug7, Faug7). We should note that during generation, the length of the provided chord progression can be any integer, not necessarily equal to the length used during training. Below are some of the generated phrases we obtained; they were generated with the same trained weights, they are different thanks to the sampling process.

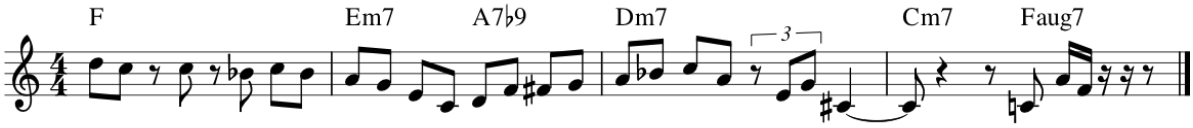
Generated phrase 1



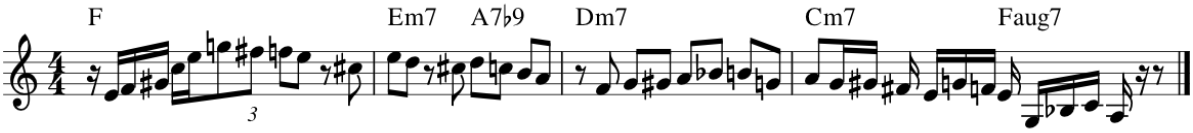
Generated phrase 2



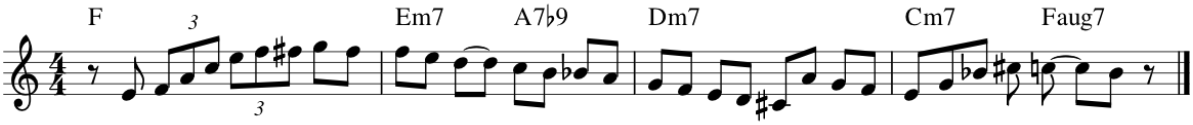
Generated phrase 3



Generated phrase 4

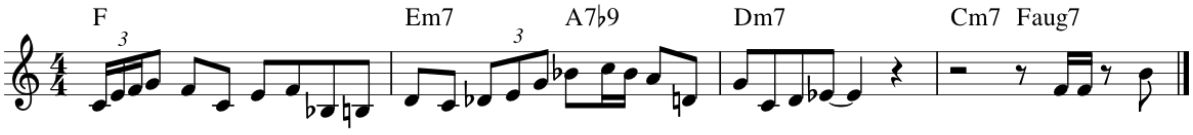


Generated phrase 5

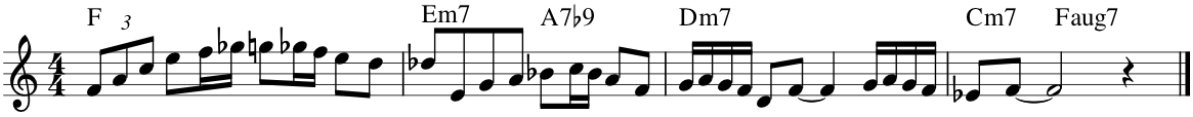


For the reader to compare, below are original phrases played by Charlie Parker over the same chord progression. Notice that some passages are very similar, but no generated phrase is equal to an original one.

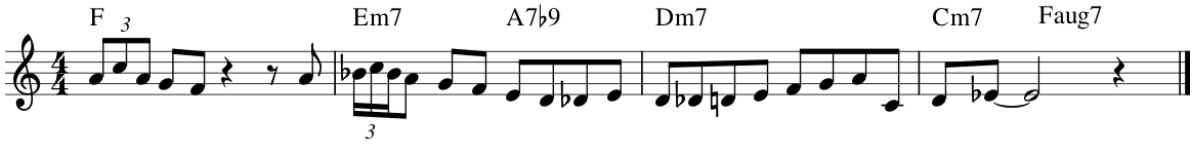
Original phrase 1



Original phrase 2



Original phrase 3



5.7 Further Improvements

As mentioned before, the main goal of the implementation presented in this chapter is to serve as a prototype experiment of the architecture presented in section 4.3, and to show that it is trainable, as suggested by the learning curves in figures 5.8 and 5.9. However, we believe that this implementation could also serve as a first step towards a more complete and thorough jazz-solo AI project. Below we provide a list of some ideas that could improve the quality of the generated solos and the overall reach of the model.

1. Build chord embeddings. The vector representation that we are using for the chords (figure 5.7) does not completely account for chords whose harmonic function is more similar than others. For example, consider the vectors C7, Em7b5, and Cm7. The vector C7 differs from both Em7b5 and Cm7 by just one 1. If we regard the vectors as points of a 24-dimensional space, the distance (C7,Em7b5) is the same as the distance (C7,Cm7), however, the harmonic functionality of C7 and Em7b5 should probably be more similar than those of C7 and Cm7. We would like to infer such similarity from the training corpus, since different musicians will approach this similarity differently. One way of tackling this issue is to build vector embeddings for chords. See [23] for more details.

2. Include the downbeat/upbeat information. We know that in jazz styles like be-bop, it is important to distinguish between notes played in the downbeat and notes played in the upbeat. Usually, notes played in the downbeat are chord tones (or available extensions), and passing notes and non-chord tones are usually played in the upbeats. Therefore, it would make sense to include this attribute in the vector representation of each note. In our experiments we noticed that the longer a phrase gets, the more the note generation loses structural sense of time. We think it would be helpful if at each time-step we pass to the network information about the offset (modulo 1) that the next signal will have.

3. Focusing more on the current chord and neighboring chords. We believe that many jazz musicians would agree that, when improvising a solo, they do not necessarily

think of the whole chord progression at all times. Sometimes is enough to know only a few following chords. We could mimic this idea in our architecture by putting more emphasis in the current chord and the next one, or even perhaps by restricting the reach of back-propagation in the LSTM that processes the events to just a few time-steps.

4. Larger training corpus. The dataset that we used is probably somewhat small by today's deep-learning standards. Training on a considerably larger corpus could probably help the network generalize better to unseen chord progressions.

5. Larger network. Provided that enough computing power is available, increasing the depth of the network and the size of the hidden states would improve the learning power of the network. However, if the dataset is still small, this might not be of much help.

6. Systematic hyper-parameter search. Again, provided that enough computing power is available, different models corresponding to different sets of hyper-parameters can be trained simultaneously. Then, the best models would be selected to be cross-validated and tested. Usually, the hyper-parameter search is a random search or a grid search.

7. Train longer. Training a model for a longer period of time would be desirable, specially if a systematic hyper-parameter search is used. This also requires access to more computing power.

CHAPTER 6
CONCLUSIONS

This dissertation describes the development of a string-diagrammatic language of (non-convolutional) neural networks that allows us to describe architectures in a rigorous way. To our knowledge, in the majority of the literature regarding neural networks, the depiction of neural network architectures is not rooted in any formal theory. With the language proposed in this work, the depiction of a neural networks is no longer just a mere sketch, but a mathematical body whose algebraic compositional structure is well accounted for. By continuing to extend this language to more types of neural networks, most importantly, convolutional neural networks, we could open new possibilities on how neural networks are built in practice. For example, the diagrams 3.1, 3.14, 3.16, 3.18 are in one-to-one correspondence with PyTorch's neural network modules, their inputs and outputs are the same, as well as the internal structure. This means that, in principle, one could "code" an architecture by putting together nodes and strings instead of using written syntax. Although it is not our goal, this illustrates the convenience of having a formal and coherent graphical language.

In addition, using the graphical language above, we introduced a novel neural network architecture suited for modeling systems that generate signals with a duration in time, conditioned to a context in the form of a sequence of events. The main novelty of this architecture consists of a concatenation mechanism that depends on the time-relative dependence of the sequence of signals with respect to the sequence of events, and a technique to its vectorized implementation.

One important question when first designing the architecture above was to investigate if an architecture with such a highly dynamic computation graph (diagram) would be trainable. The fact that the diagram of the architecture changes from one training example to the other (not only in the number of time-steps, but also in the connectivity between LSTMs), implies that the gradient of the loss function with respect to the trainable parameters is also different for each training example. Nevertheless, due to the fact that the network shares the same weights for each example, we can update all the weights of the network in each gradient hoping that, in average, the value of the loss function can reach a small enough value. However, although this makes sense theoretically, we tested this assumption in practice by developing a prototype implementation of the architecture trained on a corpus of 48 jazz solo transcriptions.

Lastly, we presented the main details of the implementation of the architecture as a generative model of jazz solo improvisations, and included the Python code. We framed a solo improvisation based on a chord progression as a sequence of signals conditioned to a sequence of events, and built a suitable dataset out of a corpus of solo transcriptions. With this implementation we answered our initial question of trainability of the architecture, and, in spite of having limited computing power, we were able to train a small architecture well enough to produce musical results that resemble the corpus. Since we consider that this implementation can be taken further as a large musical AI project, we provided some potential additions that can be implemented that could improve the power of the musical model.

REFERENCES

[1] John C Baez and Jason Erbele. Categories in control. *Theory and Applications of Categories*, 30(24):836–881, 2015.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[3] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. The calculus of signal flow diagrams i: linear relations on streams. *Information and Computation*, 252:2–29, 2017.

[4] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Interacting hopf algebras. *Journal of Pure and Applied Algebra*, 221(1):144–184, 2017.

[5] Tai-Danae Bradley, E Miles Stoudenmire, and John Terilla. Modeling sequences with quantum states: A look under the hood. *arXiv preprint arXiv:1910.07425*, 2019.

[6] Jacob C Bridgeman and Christopher T Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of Physics A: Mathematical and Theoretical*, 50(22):223001, 2017.

[7] Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. *Deep learning techniques for music generation*, volume 10. Springer, 2019.

[8] Aurelio Carboni. Matrices, relations, and group representations. *Journal of Algebra*, 136(2):497–529, 1991.

[9] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang.

Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org, 2017.

[10] Douglas Eck and Juergen Schmidhuber. Finding temporal structure in music: Blues improvisation with lstm recurrent networks. In *Proceedings of the 12th IEEE workshop on neural networks for signal processing*, pages 747–756. IEEE, 2002.

[11] Brendan Fong. Decorated cospans. *Theory and Applications of Categories*, 30(33):1096–1120, 2015.

[12] Brendan Fong. The algebra of open and interconnected systems. *arXiv preprint arXiv:1609.05382*, 2016.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] Daniel D Johnson, Robert M Keller, and Nicholas Weintraut. Learning to create jazz melodies using a product of experts. In *ICCC*, pages 151–158, 2017.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Aleks Kissinger. Finite matrices are complete for (dagger-) hypergraph categories. *arXiv preprint arXiv:1406.5942*, 2014.

[18] Anastasis Kratsios. The universal approximation property: Characterizations, existence, and a canonical topology for deep-learning. *arXiv preprint arXiv:1910.03344*,

2019.

[19] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[20] Yang Li and Shihao Ji. L_0-arm: Network sparsification via stochastic binary optimization. *The European Conference on Machine Learning (ECML)*, 2019.

[21] Yang Li and Shihao Ji. Neural plasticity networks. *arXiv preprint arXiv:1908.08118*, 2019.

[22] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[23] Sephora Madjiheurem, Lizhen Qu, and Christian Walder. Chord2vec: Learning musical chord embeddings. In *Proceedings of the constructive machine learning workshop at 30th conference on neural information processing systems (NIPS2016), Barcelona, Spain*, 2016.

[24] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[25] Robert Rosebrugh, Nicoletta Sabadini, and Robert FC Walters. Generic commutative separable algebras and cospans of graphs. *Theory and applications of categories*, 15(6):164–177, 2005.

[26] Peter Selinger. A survey of graphical languages for monoidal categories. In *New struc-*

tures for physics, pages 289–355. Springer, 2010.

[27] Harold Simmons. *An introduction to category theory*. Cambridge University Press, 2011.

[28] Kamilya Smagulova and Alex Pappachen James. A survey on lstm memristive neural network architectures and applications. *The European Physical Journal Special Topics*, 228(10):2313–2324, 2019.

APPENDIX

Pre-processing Module

```
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 11 11:02:10 2019

@author: Rodrigo Castro
"""

import os
from music21 import *
from fractions import Fraction
from random import shuffle
import numpy as np
import torch
from torch.autograd import Variable


#---------------------------------- HELPER FUNCTIONS
    -------------------------------------#

def createSoloDict(directory_in_str):

    print('Creating a dictonary of all solos...')
    directory = os.fsencode(directory_in_str)
    solo_dict = {}
    for file in os.listdir(directory):
         filename = os.fsdecode(file)
         if filename.endswith('.xml') or filename.endswith('.musicxml'):
             solo_name            = os.path.splitext(filename)[0]
             path                 = str(directory_in_str + '/' + filename
    )
             parsed_solo          = converter.parse(path)
             solo_dict[solo_name] = parsed_solo
             continue
         else:
             continue

    return solo_dict


def inspect_corpus(solo_dict, beats_per_measure=4):

    print('GETTING SOME INFORMATION ABOUT YOUR CORPUS: ')

    all_solo_durations = []
```

```
43      total_measures      = 0
44      all_note_durations = set()
45      soloist_range       = set()
46      never_plays         = set()
47      tonalities          = []
48      tonality_count      = set()
49      beat_types          = set()
50      for key in solo_dict.keys():
51          score                     = solo_dict[key]
52          score_duration_beats      = score.quarterLength
53          score_duration_measures = score_duration_beats /
    beats_per_measure
54          total_measures          += int(score_duration_measures)
55          tonality                  = score.analyze('key')
56
57          all_solo_durations.append(score_duration_measures)
58          tonalities.append(tonality)
59
60          #print(str(key) +' is ' + str(int(score_duration_measures)) + '
    measures long;')
61
62          if score_duration_beats % beats_per_measure != 0:
63              print(f'-Warning: {key} has a measure that is not {
    beats_per_measure} beats long:')
64              for measure_idx in range (1 , int(score_duration_measures +
    1) ):
65                  if score.measure(measure_idx, indicesNotNumbers = True).
    quarterLength % beats_per_measure != 0:
66                      print(f'The problem is in measure # {measure_idx + 1}
    ')
67          if len(score.getElementsByClass('Part')) > 1 :
68              print(f'*Warning: it seems that there is more than one
    instrument part in the file {key}')
69
70          for pitch in score.pitches:
71              soloist_range.add(pitch.midi)
72
73          for item in score.recurse().getElementsByClass(['Note','Rest']).
    stripTies():
74              all_note_durations.add(Fraction(item.quarterLength))
75              beat_types.add(Fraction(item.offset%1))
76              if item.quarterLength == Fraction(0, 1):
77                  print(f'There is a note with no length: {(key,item,item.
    activeSite,item.offset)}')
78
79      min_pitch, max_pitch       = int(min(soloist_range)), int(max(
    soloist_range))
80      min_duration, max_duration = int(min(all_solo_durations)), int(max(
    all_solo_durations))
81
82      for pitch in range(min_pitch, max_pitch + 1):
```

```
83          if (pitch in soloist_range) == False:
84              never_plays.add(pitch)
85
86      durations_list = list(all_note_durations)
87      durations_list.sort()
88
89      for tone in tonalities:
90          counter = tonalities.count(tone)
91          tonality_count.add((counter,tone))
92
93      print(f'*There are {int(len(solo_dict))} files in the corpus.')
94      print(f'*The shortest solo in the corpus has {min_duration} measures.
        ')
95      print(f'*The longest solo in the corpus has {max_duration} measures.'
        )
96      print(f'*There are in total {total_measures} measures of music.')
97      print(f'*The lowest midi pitch played in the corpus is {min_pitch},
        while the highest midi pitch is' + \
98            f'{max_pitch}. However, the soloist(s) never played the midi
        pitches {never_plays}.')
99      print(f'*There are {len(durations_list)} different note/rest
        durations in the corpus.')
100     #print(tonality_count)
101     #print(durations_list)
102     #print(beat_types)
103
104     return min_pitch, max_pitch, durations_list
105
106
107 def clean_chords(solo_dict, beats_per_measure=4):
108     print('Fixing measures with no explicit chord symbol assigned...')
109     for key in solo_dict.keys():
110         score = solo_dict[key]
111         harmony.realizeChordSymbolDurations(score)
112         score_duration_beats    = int(score.quarterLength)
113         score_duration_measures = int(score_duration_beats /
        beats_per_measure)
114         for measure_idx in range(0, score_duration_measures):
115             measure_chords = score.measure(measure_idx, indicesNotNumbers
            =True).recurse().getElementsByClass('ChordSymbol')
116             if [c for c in measure_chords] == []:
117                 #print('Fixing chord in measure ' + str(measure_idx+1) +
                ' in ' + str(key))
118                 last_chord      = score.measure(measure_idx-1,
                indicesNotNumbers=True).recurse().getElementsByClass('ChordSymbol')
                [-1]
119                 last_chord_name = last_chord.figure
120                 try:
121                     missing_chord = harmony.ChordSymbol(last_chord_name)
122                 except:
123                     missing_chord_root       = last_chord.root().name
```

```
124                     missing_chord_pitches    = [p.name for p in
    last_chord.pitches]
125                     missing_chord            = harmony.ChordSymbol(root=
    missing_chord_root)
126                     missing_chord.pitchNames = missing_chord_pitches
127                 missing_chord.quarterLength = beats_per_measure
128 
129                 score.parts[0].measure(measure_idx, indicesNotNumbers=
    True).insert(0, missing_chord)
130                 if last_chord.quarterLength % beats_per_measure == 0:
131                     score.parts[0].measure(measure_idx-1,
    indicesNotNumbers=True).getElementsByClass('ChordSymbol')[-1].
    quarterLength \
132                     = beats_per_measure
133                 else:
134                     score.parts[0].measure(measure_idx-1,
    indicesNotNumbers=True).getElementsByClass('ChordSymbol')[-1].
    quarterLength \
135                     = last_chord.quarterLength % beats_per_measure
136 
137 
138 def parse_dict(solo_dict, durations_list,min_pitch, max_pitch,
    window_size=4, \
139                beats_per_measure=4, transpose=False):
140     all_windows = []
141     dict_len = len(solo_dict)
142     for key_idx, key in enumerate(solo_dict.keys()):
143         score                    = solo_dict[key]
144         score_duration_measures = len(score.recurse().getElementsByClass(
    'Measure'))
145         last_window_idx          = score_duration_measures - window_size
146         print(f'Splitting {key}, score {key_idx+1}/{dict_len}')
147         for window_idx in range(0, last_window_idx):
148             window = score.measures( window_idx , window_idx +
    window_size , indicesNotNumbers=True)
149             if window.quarterLength != beats_per_measure*window_size :
150                  print(f'Window {(key,window_idx)} has quarter length {
    window.quarterLength}')
151 
152             all_windows.append(window)
153             if transpose==True:
154                 print(f'Processing and transposing window {window_idx
    +1}/{last_window_idx}')
155                 for interval in range(-5,7):
156                     if interval == 0:
157                         continue
158                     #print('Transposing ' + str(interval) + ' half steps
    ...')
159                     transposed_window = window.transpose(interval)
160                     min_pitch += -5
161                     max_pitch += 6
```

```
162                     all_windows.append(transposed_window)
163     shuffle(all_windows)
164
165     num_windows          = len(all_windows)
166     chord_vect_size      = 24
167     pitch_vect_size      = max_pitch - min_pitch + 2   #include rest as a
        pitch
168     duration_vect_size   = len(durations_list)
169     note_vect_size       = pitch_vect_size + duration_vect_size
170     progression_matrices = []
171     melody_matrices      = []
172     for count, window in enumerate(all_windows):
173         print('Encoding window ' + str(count+1) + '/' + str(num_windows))
174         harmony.realizeChordSymbolDurations(window)
175         window_chords = window.recurse().getElementsByClass('ChordSymbol'
    )
176         window_notes  = window.recurse().getElementsByClass(['Note','Rest
    ']).stripTies()
177
178         progression_matrix = np.zeros([0, chord_vect_size])
179         for chord in window_chords:
180             chord_vector   = np.zeros([1, chord_vect_size])
181             chord_duration = int(chord.quarterLength)
182             root_idx       = chord.root().midi % 12
183             chord_pitches    = [p.midi for p in chord.pitches]
184             chord_vector[0,root_idx] = 1
185             for i, chord_pitch in enumerate(chord_pitches):
186                 chord_pitch_idx = chord_pitch % 12
187                 chord_vector[0, 12 + chord_pitch_idx] = 1
188             for j in range(chord_duration):
189                 progression_matrix = np.append(progression_matrix,
    chord_vector, axis=0)
190         progression_matrices.append(progression_matrix)
191
192         melody_matrix = np.zeros([0, note_vect_size])
193         for note in window_notes:
194             pitch_vector    = np.zeros([1, pitch_vect_size])
195             duration_vector = np.zeros([1, duration_vect_size])
196             if note.isRest:
197                 pitch_idx = pitch_vect_size - 1
198             else:
199                 pitch_idx = note.pitch.midi - min_pitch
200             pitch_vector[0, pitch_idx] = 1
201             duration = Fraction(note.quarterLength)
202             if duration in durations_list:
203                 duration_idx = durations_list.index(duration)
204             else:
205                 raise ValueError('The duration ' + str(duration) + ' is
    not in durations_list!')
206             duration_vector[0, duration_idx] = 1
207             note_vector   = np.append(pitch_vector, duration_vector, axis
```

```
    =1)
208             melody_matrix = np.append(melody_matrix, note_vector, axis=0)
209         melody_matrices.append(melody_matrix)
210 
211 
212         #check for problems in the length of the chord progressions:
213         '''
214         for idx,window in enumerate(progression_matrices):
215             if window.shape != (16,24):
216                 print(idx,window.shape)
217                 for chord in all_windows[idx].recurse().
    getElementsByClass('ChordSymbol'):
218                     print(chord.quarterLength,chord)
219         '''
220 
221 
222         #check for problems in the length of the note sequences:
223         '''
224         for i,window in enumerate(all_windows):
225             window_notes  = window.parts[0].recurse().getElementsByClass
    (['Note','Rest']).stripTies()
226             if window_notes.quarterLength != 16:
227                 print(i,window_notes.quarterLength)
228         '''
229 
230     return progression_matrices, melody_matrices
231 
232 
233 def matrices2tensors(progression_matrices, melody_matrices):
234     event_data = np.stack(progression_matrices)
235     E = Variable(torch.from_numpy(event_data))
236     E = E.type(torch.FloatTensor)
237 
238     S=[]
239     for window in melody_matrices:
240         window = np.stack(window)
241         window = Variable(torch.from_numpy(window))
242         window = window.type(torch.FloatTensor)
243         S.append(window)
244 
245     return E, S, durations_list
246 
247 
248 
249 #------------------------------ PUBLIC FUNCTION
    -------------------------------------#
250 
251 
252 def build_dataset( directory_in_str, filename, beats_per_measure=4,
    transpose=False):
253     solo_dict = createSoloDict(directory_in_str)
```

```
254     min_pitch , max_pitch , durations_list = inspect_corpus(solo_dict)
255     clean_chords(solo_dict, beats_per_measure)
256     progression_matrices, melody_matrices = \
257     parse_dict(solo_dict, durations_list,min_pitch, max_pitch,
      window_size=4, \
258                 beats_per_measure=4,transpose=False)
259     E, S, durations_list = matrices2tensors(progression_matrices,
      melody_windows)
260     Training_data = [ E, S, durations_list, min_pitch , max_pitch]
261     torch.save(Training_data, filename)
262
263 #build_dataset('Divided_solos', 'Parker_Dataset.pt', beats_per_measure=4)
264 #build_dataset('Divided_solos', 'Parker_Dataset_allKeys.pt',
      beats_per_measure=4, transpose=True)
```

Training Module

```
1
2 # coding: utf-8
3
4 # <a href="https://colab.research.google.com/github/irodcast/dynamic_solo
      /blob/master/train_model_2.ipynb" target="_parent"><img src="https://
      colab.research.google.com/assets/colab-badge.svg" alt="Open In Colab
      "/></a>
5
6 # In[ ]:
7
8
9 import numpy as np
10 import torch
11 from torch.autograd import Variable
12 import torch.nn.functional as F
13 import matplotlib.pyplot as plt
14 import time
15
16 print(f'PyTorch version: {torch.__version__}')
17
18 if torch.cuda.is_available()==True:
19     use_cuda = True
20     print(f'GPU available: {torch.cuda.get_device_name(0)} ({torch.cuda.
      device_count()} count)')
21
22
23 # In[ ]:
24
25
26 def default_device():
27     if torch.cuda.is_available()==True:
28         dflt_device = torch.device('cuda')
```

```
29      else:
30          dflt_device = torch.device('cpu')
31
32      return dflt_device
33
34
35  def load_data(dir_in_str):
36      E, S_pre, durations_list, min_pitch, max_pitch = torch.load(
        dir_in_str)
37      E = E.to(device=dflt_device)
38      S = []
39      for tensor in S_pre:
40          S.append(tensor.to(device=dflt_device))
41
42      return E, S, durations_list, min_pitch, max_pitch
43
44
45  def dimensions(E,S):
46      num_event_examples, num_events , event_emb_size  = E.shape
47      num_seq_examples = len(S)
48      signal_emb_size = S[0].size(1)
49
50      dims = [num_event_examples, num_events , event_emb_size,
        num_seq_examples, signal_emb_size ]
51
52      return dims
53
54
55  def prepare_data(S):
56      S_pre_input = []
57      first_row = torch.zeros(1,signal_emb_size).to(device=dflt_device)
58      for tensor in S:
59          expanded_tensor = torch.cat((first_row, tensor), dim=0)
60          new_tensor = expanded_tensor[:-1, :]
61          S_pre_input.append(new_tensor)
62
63      conditioning_idxs_vectors = []
64      for tensor in S:
65          conditioning_indices = torch.zeros(tensor.shape[0], 1).to(device=
        dflt_device)
66          cumulative_duration = 0
67          for row in range(0, tensor.shape[0]-1):
68              vector = tensor[row, :]
69              pitch_idx, rhythm_idx = list((vector != 0).nonzero())
70              pitch_idx, rhythm_idx = int(pitch_idx), int(rhythm_idx)
71              duration_type_idx = rhythm_idx - rhythm_idx_ini
72              duration_type = durations_list[duration_type_idx]
73              cumulative_duration += duration_type
74              conditioning_indices[row+1] = int(cumulative_duration)
75          conditioning_idxs_vectors.append(conditioning_indices)
76
```

```
77      lengths_list = []
78      for tensor in S:
79          lengths_list.append(tensor.shape[0])
80
81      S_padded = torch.nn.utils.rnn.pad_sequence(S, batch_first=True)
82      S_packed = torch.nn.utils.rnn.pack_padded_sequence(S_padded,
    batch_first=True, lengths=lengths_list, enforce_sorted=False)
83
84      return S_packed, S_padded, S_pre_input, lengths_list,
    conditioning_idxs_vectors
85
86
87  def create_placing_matrices(conditioning_idxs_vectors, num_events):
88      placing_conditioning_matrices = []
89      for vector in conditioning_idxs_vectors:
90          placing_matrix = torch.zeros(vector.shape[0], num_events).to(
    device=dflt_device)
91          for i in range(vector.shape[0]):
92              placing_matrix[i, int(vector[i])] = 1
93          placing_conditioning_matrices.append(placing_matrix)
94
95      return placing_conditioning_matrices
96
97
98  def concatenate_conditioning(S_pre_input,
    encoded_conditioning,
    placing_conditioning_matrices, lengths_list):
99      S_conditioned = []
100     for idx, tensor in enumerate(S_pre_input):
101       placing_matrix = placing_conditioning_matrices[idx]
102       dynamic_conditioning = torch.mm(placing_matrix,
    encoded_conditioning[idx,:,:])
103       concatenated_input = torch.cat((tensor, dynamic_conditioning), dim
    =1)
104       S_conditioned.append(concatenated_input)
105     S_input = torch.nn.utils.rnn.pad_sequence(S_conditioned, batch_first=
    True)
106     S_input = torch.nn.utils.rnn.pack_padded_sequence(S_input,
    batch_first=True, lengths=lengths_list, enforce_sorted=False)
107
108     return S_input
109
110
111 # In[ ]:
112
113
114 class event_net(torch.nn.Module):
115
116     def __init__(self, event_emb_size, event_hidden_size,
    event_output_size,                  num_event_layers,
    num_event_examples, num_directions):
```

```
117         super(event_net, self).__init__()
118 
119         self.event_emb_size     = event_emb_size
120         self.event_hidden_size  = event_hidden_size
121         self.event_output_size  = event_output_size
122         self.num_event_layers   = num_event_layers
123         self.num_event_examples = num_event_examples
124         self.num_directions     = num_directions
125 
126         self.event_lstm   = torch.nn.LSTM(self.event_emb_size, self.
    event_hidden_size,                                          self.
    num_event_layers, batch_first=True, bidirectional=True)
127         self.event_linear = torch.nn.Linear(self.event_hidden_size*
    num_directions, self.event_output_size)
128 
129         self.initHidden  = self.init_hidden()
130 
131     def init_hidden(self):
132         h_ini = (torch.zeros(self.num_event_layers*num_directions, self.
    num_event_examples, self.event_hidden_size),              torch.zeros(
    self.num_event_layers*num_directions, self.num_event_examples, self.
    event_hidden_size) )
133 
134     def forward(self, Events):
135         event_lstm_out, event_hidden = self.event_lstm(Events, self.
    initHidden)
136         linear_output = self.event_linear(event_lstm_out*num_event_layers
    )
137         event_output = torch.sigmoid(linear_output)
138 
139         return event_output #event_lstm_out
140 
141 
142 class signal_net(torch.nn.Module):
143     def __init__(self, signal_emb_size, conditioning_size,
    signal_hidden_size,                 signal_output_size,
    num_signal_layers, num_signal_examples):
144         super(signal_net, self).__init__()
145 
146         self.signal_emb_size     = signal_emb_size
147         self.conditioning_size   = conditioning_size
148         self.signal_hidden_size  = signal_hidden_size
149         self.signal_output_size  = signal_output_size
150         self.num_signal_layers   = num_signal_layers
151         self.num_signal_examples = num_signal_examples
152 
153         self.signal_lstm   = torch.nn.LSTM(self.signal_emb_size+self.
    conditioning_size, self.signal_hidden_size,
                self.num_signal_layers, batch_first=True)
154         self.signal_linear = torch.nn.Linear(self.signal_hidden_size,
    self.signal_output_size)
```

```
155
156     def forward(self, S_input, prev_hidden):
157         signal_lstm_out, signal_hidden = self.signal_lstm(S_input,
        prev_hidden)
158         signal_linear_output = self.signal_linear(signal_lstm_out.data)
159         #signal_output = torch.sigmoid(signal_linear_output)
160
161         return signal_linear_output, signal_hidden
162
163
164 # In[ ]:
165
166
167 dflt_device = default_device()
168
169 E, S, durations_list, min_pitch, max_pitch = load_data('
        Parker_Dataset_unshuffled.pt')
170 #E, S = E[0:6,:,:], S[0:6]
171
172 num_event_examples, num_events , event_emb_size, num_signal_examples,
        signal_emb_size = dimensions(E,S)
173 rhythm_idx_ini = max_pitch - min_pitch + 1 + True
174
175 S_packed, S_padded, S_pre_input, lengths_list, conditioning_idxs_vectors
        = prepare_data(S)
176 placing_conditioning_matrices = create_placing_matrices(
        conditioning_idxs_vectors, num_events)
177
178
179 # In[ ]:
180
181
182 torch.manual_seed(12)
183
184 #Choose dimensions for event LSTM
185 num_event_layers = 1
186 event_hidden_size = 32
187 num_directions = 2
188 event_output_size = 48
189
190 #Choose dimensions for signal LSTM
191 num_signal_layers  = 1
192 signal_hidden_size = 128
193 signal_output_size = 89
194 conditioning_size  = event_output_size
195
196 #Create 1st LSTM
197 event_forward_pass = event_net(event_emb_size, event_hidden_size,
        event_output_size,                  num_event_layers,
        num_event_examples, num_directions)
198 event_forward_pass = event_forward_pass.to(device=dflt_device)
```

```
199
200 #Create 2nd LSTM
201 signal_forward_pass = signal_net(signal_emb_size, conditioning_size,
        signal_hidden_size,                  signal_output_size,
        num_signal_layers, num_signal_examples)
202 signal_forward_pass = signal_forward_pass.to(device=dflt_device)
203 signal_h_ini = (torch.zeros(num_signal_layers, num_signal_examples,
        signal_hidden_size).to(device=dflt_device),              torch.zeros(
        num_signal_layers, num_signal_examples, signal_hidden_size).to(device=
        dflt_device) )
204
205 weights = list(event_forward_pass.parameters()) + list(
        signal_forward_pass.parameters())
206
207 #Number of parameters
208 num_event_parameters = sum([p.numel() for p in event_forward_pass.
        parameters()])
209 print(f'Number of parameters in LSTM of events: {num_event_parameters}')
210
211 num_signal_parameters = sum([p.numel() for p in signal_forward_pass.
        parameters()])
212 print(f'Number of parameters in LSTM of signals: {num_signal_parameters}'
        )
213
214 print(f'Total number of parameters: {num_event_parameters+
        num_signal_parameters}')
215
216
217 # In[ ]:
218
219
220 LR          = 0.05
221 epochs      = 3000
222 WeightDecay = 1e-8
223 Momentum    = 0.9
224
225 loss_func    = torch.nn.BCEWithLogitsLoss()
226 #loss_func    = torch.nn.MSELoss()
227 optimizer    = torch.optim.Adam(weights, lr=LR, betas=(0.9, 0.999), eps=1
        e-8, weight_decay = WeightDecay )
228 #optimizer    = torch.optim.RMSprop(weights,lr=LR, alpha=0.99, eps=1e-8,
        weight_decay = WeightDecay, momentum = Momentum, centered=True)
229 #scheduler    = torch.optim.lr_scheduler.StepLR(optimizer, step_size=300,
         gamma=0.5, last_epoch=-1)
230 scheduler    = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
        T_max=epochs, eta_min=0.000001, last_epoch=-1)
231
232 loss_hist  = []
233 for epoch in range(1, epochs+1):
234   t = time.time()
235   optimizer.zero_grad()
```

```
236   encoded_conditioning = event_forward_pass(E)
237   S_input = concatenate_conditioning(S_pre_input, encoded_conditioning,
       placing_conditioning_matrices, lengths_list)
238   S_hat, _ = signal_forward_pass(S_input, signal_h_ini)
239   Loss = loss_func(S_hat, S_packed.data)
240   Loss.backward()
241   optimizer.step()
242   loss_hist.append(Loss.item())
243   scheduler.step()
244   #if epoch%200==0:
245   print(f'Epoch: {epoch}, Loss: {Loss}  (Learning rate: {scheduler.get_lr
       ()}, Time: {round(time.time()-t,4)}s')
246
247 plt.plot(loss_hist[:])
248 plt.xlabel('Gradient Steps')
249 vert_label=plt.ylabel('Loss')
250 vert_label.set_rotation(0)
251
252
253 # In[ ]:
254
255
256 hyperparameters = {'num_event_layers':num_event_layers, '
      event_hidden_size':event_hidden_size,                          '
      num_directions':num_directions, 'event_output_size':event_output_size,
                              'num_signal_layers':num_signal_layers, '
      signal_hidden_size':signal_hidden_size,
257                         'signal_output_size':signal_output_size, '
      conditioning_size':event_output_size,\
258                          'LR': LR, 'epochs':epochs, 'WeightDecay':
      WeightDecay, 'Momentum':Momentum }
259 model_parameters = [hyperparameters, event_forward_pass.state_dict(),
      signal_forward_pass.state_dict()]
260
261 torch.save(model_parameters, 'model_parameters.pt')
```

Generative Module

```
1
2 # coding: utf-8
3
4 # In[1]:
5
6
7 import numpy as np
8 import torch
9 from torch.autograd import Variable
10 import torch.nn.functional as F
11 from music21 import *
```

```
12
13 print(torch.__version__)
14
15
16 # In[2]:
17
18
19 def default_device():
20     if torch.cuda.is_available()==True:
21         dflt_device = torch.device('cuda')
22     else:
23         dflt_device = torch.device('cpu')
24
25     return dflt_device
26
27
28 def dimensions(E,S):
29     num_event_examples, num_events , event_emb_size  = E.shape
30     num_seq_examples = len(S)
31     signal_emb_size = S[0].size(1)
32
33     dims = [num_event_examples, num_events , event_emb_size,
    num_seq_examples, signal_emb_size ]
34
35     return dims
36
37
38 def net_generate(e, sample=False):
39     event_steps = e.shape[0]
40     encoded_conditioning = event_forward_pass(e.view(1, event_steps, -1))
41     #signal_h_ini = (torch.zeros(num_signal_layers, num_signal_examples,
    signal_hidden_size).to(device=dflt_device),\
42                 #torch.zeros(num_signal_layers, num_signal_examples,
    signal_hidden_size).to(device=dflt_device) )
43     #hidden = (signal_h_ini[0][0,0,:].view(num_signal_layers, 1,
    signal_hidden_size), \
44             #signal_h_ini[1][0,0,:].view(num_signal_layers, 1,
    signal_hidden_size))
45     hidden = (torch.zeros(num_signal_layers, 1, signal_hidden_size).to(
    device=dflt_device),              torch.zeros(num_signal_layers, 1,
    signal_hidden_size).to(device=dflt_device) )
46     signal_prev = torch.zeros(1, signal_emb_size).to(device=dflt_device)
47     prediction_list     = []
48     raw_prediction_list = []
49     cumulative_duration = 0
50     while cumulative_duration <= float(event_steps-1):
51         dynamic_idx = int(cumulative_duration)
52         conditioning = encoded_conditioning[0, dynamic_idx, :].view(1,-1)
53         signal_input = torch.cat((signal_prev, conditioning), dim=1)
54         signal_input = signal_input.view(1,1,-1)
55         s_hat_pre, hidden = signal_forward_pass(signal_input, hidden )
```

```
56          #s_hat = torch.sigmoid(s_hat)  #smoothens the future prob dist
57          s_hat_pitch  = F.softmax( s_hat_pre[0, 0, 0:rhythm_idx_ini], dim
    = 0 )
58          s_hat_rhythm = F.softmax( s_hat_pre[0, 0, rhythm_idx_ini:], dim =
     0 )
59
60          raw_s_hat = torch.cat ((s_hat_pitch, s_hat_rhythm), dim = 0).view
    (1,-1)
61          raw_prediction_list.append(raw_s_hat)
62
63          if sample == False:
64              note_max , note_argmax = s_hat_pitch.max(0)
65              rhythm_max , rhythm_argmax = s_hat_rhythm.max(0)
66
67          if sample == True:
68              note_prob_dist = torch.distributions.Categorical(s_hat_pitch)
69              note_argmax = int(note_prob_dist.sample())
70              rhythm_prob_dist = torch.distributions.Categorical(
    s_hat_rhythm)
71              rhythm_argmax = int(rhythm_prob_dist.sample())
72              #rhythm_max , rhythm_argmax = s_hat_rhythm.max(0)
73
74          s_hat = torch.zeros(1, signal_emb_size)
75          s_hat[0, int(note_argmax)] = 1
76          s_hat[0, int(rhythm_idx_ini + int(rhythm_argmax))] = 1
77          prediction_list.append(s_hat)
78
79          #pitch_idx, rhythm_idx = list((s_hat != 0).nonzero())
80          #pitch_idx, rhythm_idx = int(pitch_idx), int(rhythm_idx)
81          #duration_type_idx = rhythm_argmax - rhythm_idx_ini
82          duration_type = durations_list[rhythm_argmax]
83          cumulative_duration += duration_type
84
85          signal_prev  = s_hat.to(device=dflt_device)
86
87      prediction = torch.cat(prediction_list)
88      raw_prediction = torch.cat(raw_prediction_list)
89
90      return prediction , raw_prediction
91
92
93  def vect2note(vector):
94      note_embedding_size = signal_emb_size
95      assert np.shape(vector) == (note_embedding_size,)
96      duration_idx = int(np.argwhere(vector[rhythm_idx_ini:]))
97      duration = durations_list[duration_idx]
98
99      if vector[rhythm_idx_ini-1] == 1:
100         nota = note.Rest()
101         nota.quarterLength = duration
102     else:
```

```
103         height = int(np.argwhere(vector[:rhythm_idx_ini-1])) + min_pitch
104         nota = note.Note()
105         nota.pitch.midi = height
106         nota.quarterLength = duration
107 
108     return nota
109 
110 
111 def matrix2melody(melodyMatrix):
112     m,n = melodyMatrix.shape
113     melodyStream = stream.Stream()
114 
115     #To impose a time and key signature:
116     melodyStream.timeSignature = meter.TimeSignature('4/4')
117     melodyStream.keySignature = key.Key('C')
118 
119     for i in range(n):
120         vector = melodyMatrix[:,i]
121         nota = vect2note(vector)
122         melodyStream.append(nota)
123         melody = melodyStream.stripTies()
124 
125     return melody
126 
127 
128 def enter_progression(event_emb_size, beats_per_measure=4):
129 
130     chord_vect_size = event_emb_size
131     progression_matrix  = np.zeros([0, chord_vect_size])
132     progression_symbols = []
133     chord_list = []
134 
135     print('How many bars long is your progression? (enter an integer): ')
136     bars = int(input())
137     total_beats = beats_per_measure * bars
138     for beat in range(0, total_beats):
139         progression_symbols.append('?')
140 
141     progression_display = ''
142     progression_display_length = total_beats + bars + 1
143     temp_display = progression_symbols[:]
144     for i in range(0, progression_display_length, 5):
145         temp_display.insert(i,'|')
146 
147     for i in range(len(temp_display)):
148         progression_display += temp_display[i] + ' '
149 
150     print('Progression status: ' + progression_display )
151 
152     counter = 0
153     while True:
```

```
154         chord_vector = np.zeros([1, chord_vect_size])
155 
156         print('Enter a chord:')
157         chord_name = str(input())
158 
159         print('How many beats of that chord?')
160         chord_duration = int(input())
161         total_duration = counter + chord_duration
162         if total_duration > total_beats:
163             break
164         else:
165             chord = harmony.ChordSymbol(chord_name)
166             chord.quarterLength = chord_duration
167             chord_list.append(chord)
168 
169             root_idx       = chord.root().midi % 12
170             chord_pitches  = [p.midi for p in chord.pitches]
171             chord_vector[0,root_idx] = 1
172             for i, chord_pitch in enumerate(chord_pitches):
173                 chord_pitch_idx = chord_pitch % 12
174                 chord_vector[0, 12 + chord_pitch_idx] = 1
175             for j in range(chord_duration):
176                 progression_matrix = np.append(progression_matrix,
    chord_vector, axis=0)
177 
178             for i in range(counter, total_duration):
179                 progression_symbols[i] = chord_name
180             temp_display = progression_symbols[:]
181 
182             for i in range(0, progression_display_length, 5):
183                 temp_display.insert(i,'|')
184 
185             progression_display = ''
186             for i in range(len(temp_display)):
187                 progression_display += temp_display[i] + ' '
188 
189             print('Progression: ' + progression_display)
190             counter = total_duration
191             if counter == total_beats:
192                 break
193             else:
194                 continue
195 
196     chord_matrix = torch.from_numpy(progression_matrix)
197     chord_matrix = chord_matrix.type(torch.FloatTensor)
198 
199     return progression_display , chord_matrix , chord_list
200 
201 
202 def predict_new(chord_matrix, chord_list, sample=False):
203 
```

```
204     solo_prediction, raw_prediction = net_generate(chord_matrix, sample)
205     solo_prediction = solo_prediction.transpose(0,1)
206     raw_prediction = raw_prediction.transpose(0,1)
207     solo_prediction = solo_prediction.numpy()
208     raw_prediction = raw_prediction.detach().numpy()
209 
210     solo = matrix2melody(solo_prediction)
211 
212     chord_sounds = stream.Stream()
213     for acorde in chord_list:
214         acorde.writeAsChord = True
215         chord_sounds.append(acorde)
216 
217     solo_with_chords = stream.Score()
218     solo_with_chords.insert(0, solo)
219     solo_with_chords.insert(0, chord_sounds)
220 
221     return solo_with_chords, solo_prediction , raw_prediction
222 
223 
224 # In[3]:
225 
226 
227 class event_net(torch.nn.Module):
228 
229     def __init__(self, event_emb_size, event_hidden_size,
    event_output_size,                  num_event_layers,
    num_event_examples, num_directions):
230         super(event_net, self).__init__()
231 
232         self.event_emb_size     = event_emb_size
233         self.event_hidden_size  = event_hidden_size
234         self.event_output_size  = event_output_size
235         self.num_event_layers   = num_event_layers
236         self.num_event_examples = num_event_examples
237         self.num_directions     = num_directions
238 
239         self.event_lstm   = torch.nn.LSTM(self.event_emb_size, self.
    event_hidden_size,                                      self.
    num_event_layers, batch_first=True, bidirectional=True)
240         self.event_linear = torch.nn.Linear(self.event_hidden_size*
    num_directions, self.event_output_size)
241 
242         self.initHidden  = self.init_hidden()
243 
244     def init_hidden(self):
245         h_ini = (torch.zeros(self.num_event_layers*num_directions, self.
    num_event_examples, self.event_hidden_size),              torch.zeros(
    self.num_event_layers*num_directions, self.num_event_examples, self.
    event_hidden_size) )
246 
```

```
247     def forward(self, Events):
248         event_lstm_out, event_hidden = self.event_lstm(Events, self.
    initHidden)
249         linear_output = self.event_linear(event_lstm_out*num_event_layers
    )
250         event_output = torch.sigmoid(linear_output)
251 
252         return event_output #event_lstm_out
253 
254 
255 class signal_net(torch.nn.Module):
256     def __init__(self, signal_emb_size, conditioning_size,
    signal_hidden_size,                   signal_output_size,
    num_signal_layers, num_signal_examples):
257         super(signal_net, self).__init__()
258 
259         self.signal_emb_size     = signal_emb_size
260         self.conditioning_size   = conditioning_size
261         self.signal_hidden_size  = signal_hidden_size
262         self.signal_output_size  = signal_output_size
263         self.num_signal_layers   = num_signal_layers
264         self.num_signal_examples = num_signal_examples
265 
266         self.signal_lstm   = torch.nn.LSTM(self.signal_emb_size+self.
    conditioning_size, self.signal_hidden_size,
                self.num_signal_layers, batch_first=True)
267         self.signal_linear = torch.nn.Linear(self.signal_hidden_size,
    self.signal_output_size)
268 
269     def forward(self, S_input, prev_hidden):
270         signal_lstm_out, signal_hidden = self.signal_lstm(S_input,
    prev_hidden)
271         signal_linear_output = self.signal_linear(signal_lstm_out.data)
272         #signal_output = torch.sigmoid(signal_linear_output)
273 
274         return signal_linear_output, signal_hidden
275 
276 
277 # In[31]:
278 
279 
280 dflt_device = default_device()
281 
282 E, S, durations_list, min_pitch, max_pitch = torch.load('Parker_Dataset.
    pt')
283 num_event_examples, num_events , event_emb_size, num_signal_examples,
    signal_emb_size = dimensions(E,S)
284 rhythm_idx_ini = max_pitch - min_pitch + 1 + True
285 
286 hyperparameters, event_forward_pass_parameters,
    signal_forward_pass_parameters = torch.load('model_parameters.pt',
```

```
    map_location=torch.device('cpu'))

num_event_layers   = hyperparameters['num_event_layers']
event_hidden_size  = hyperparameters['event_hidden_size']
num_directions     = hyperparameters['num_directions']
event_output_size  = hyperparameters['event_output_size']
num_signal_layers  = hyperparameters['num_signal_layers']
signal_hidden_size = hyperparameters['signal_hidden_size']
signal_output_size = hyperparameters['signal_output_size']
conditioning_size  = hyperparameters['conditioning_size']

event_forward_pass  = event_net(event_emb_size, event_hidden_size,
    event_output_size,                      num_event_layers,
    num_event_examples, num_directions)
signal_forward_pass = signal_net(signal_emb_size, conditioning_size,
    signal_hidden_size,                       signal_output_size,
    num_signal_layers, num_signal_examples)
event_forward_pass  = event_forward_pass.to(device=dflt_device)
signal_forward_pass = signal_forward_pass.to(device=dflt_device)

event_forward_pass.load_state_dict(event_forward_pass_parameters)
signal_forward_pass.load_state_dict(signal_forward_pass_parameters)


# In[33]:


progression_display , chord_matrix , chord_list = enter_progression(
    event_emb_size, beats_per_measure=4)


# In[37]:


#chord_matrix = E[1,:,:]
solo, solo_prediction , raw_prediction = predict_new(chord_matrix,
    chord_list, sample=True)
solo.show()
solo.show('midi')


# In[38]:


solo.write('xml', 'SoloSnips/generated_solo.xml')
```