INFORMS Journal on Computing
Vol. 18, No. 3, Summer 2006, pp. 321–338
ISSN 1091-9856 | EISSN 1526-5528 | 06 | 1803 | 0321

DOI 10.1287/ijoc.1050.0131

Recurrent Neural Networks for Music Computation

Judy A. Franklin
Computer Science Department, Smith College, Northampton, Massachusetts 01063, USA,
jfranklin@scinix.smith.edu

Some researchers in the computational sciences have considered music computation, including music reproduction and generation, as a dynamic system, i.e., a feedback process. The key element is that the state of the musical system depends on a history of past states. Recurrent (neural) networks have been deployed as models for learning musical processes. We first present a tutorial discussion of recurrent networks, covering those that have been used for music learning. Following this, we examine a thread of development of these recurrent networks for music computation that shows how more intricate music has been learned as the state of the art in recurrent networks improves. We present our findings that show that a long short-term memory recurrent network, with new representations that include music knowledge, can learn musical tasks, and can learn to reproduce long songs. Then, given a reharmonization of the chordal structure, it can generate an improvisation.

Key words: recurrent neural networks; computer music; music representation; LSTM
History: Accepted by Elaine Chew, Guest Editor of the Special Cluster on Music and Computation; received January 2004; revised July 2004, December 2004; accepted December 2004.

1. Introduction

Recurrent neural networks have been developed both by neural-network designers as well as process-control engineers. The state of the art of recurrent networks from their use as predictors and filters, to architectures of multiple nets, to the equivalence of recurrent network models with finite automata, push-down automata, and Turing machines, to limitations, evaluation, and stability, is described in Kolen and Kremer (2001) and Mandic and Chambers (2001).

The use of recurrent networks in music learning and composition parallels these efforts. We describe several specific recurrent networks that have been used for computer music. Our focus is on digital music at the pitch and duration level, not at the signal-processing level; i.e., we assume pitches and durations of notes are available when learning. These algorithms do not need to determine pitch from an acoustic signal and do not perform any frequency analysis. Rather, the focus is on whether recurrent networks can learn a long and cohesive composition and remember earlier motifs and structured song forms, as well as whether it can generate a new one. The type of recurrent network used affects its ability to learn music, but so does the representation of the inputs and outputs of the network. The choice of representation also depends on whether the network is learning from musical scores or from human performances. This is also a factor if the network is to be used for interactive playing, either during training or afterward. Finally, while the tempo can be varied externally, most work in using recurrent networks for music has assumed a fixed tempo, and the networks do not explicitly adapt to varying beats and tempos. There has been some focused work in using specialized networks to learn to recognize beat and tempo variations, a process called *entrainment* (see Desain et al. 1989, Large and Kolen 1994, and Allen and Dannenberg 1990).

In the next section, we describe several types of recurrent networks that have been used in music learning and composition programs. We include details of the algorithms, while also exploring possible limitations. This section may be read thoroughly, or skimmed before reading §3, where we describe how these networks have been used in past music systems. It is within this section that we begin to address issues of music representation. In §4 we describe our own work with the LSTM network and our music representations. We conclude in §5.

2. Neural Networks, Feedforward and Recurrent

2.1. Feedforward Networks

We first briefly describe non-recurrent, feedforward neural networks that consist of two or more layers of small processing units that are connected to the next layer by weighted connections. The output of each layer is fed forward through these connections to the

next layer, until the output layer is reached. This is called the *forward pass*. An error is formed at the output, and the error is passed back through the network in a backward pass, and the weights on the connections are incrementally adjusted. Through an iterative training procedure in which example inputs and the target outputs are presented to the network repeatedly, the network can learn a nonlinear function of the inputs and can also generalize and produce outputs for examples it has not seen before. Such networks are useful for pattern matching and classification and have been explored within the computer-music community to classify chords (Laden and Keefe 1991), to detect musical styles (Dannenberg et al. 1997), and to accomplish other tasks such as sound synthesis, pitch perception modeling, and learning to reproduce and to create melodies (Todd and Loy 1991, Griffith and Todd 1999).

As an example, suppose a network has three layers. The first layer is a set of numerical inputs, x_i, where the examples are presented. The inputs are generally multiplied by weights and are processed by the individual, generally nonlinear, processing units in the second layer. Each processing unit has its own set of connection weights. The weights may be labeled w_{ki}, denoting that input i is connected to unit k in the second layer by weight w_{ki}. Notice that the order of the subscripts is important. The output of unit k is calculated as

$$y_k(t) = f(net_k(t)) \tag{1}$$

where

$$net_k(t) = \sum_{i \in Inputs} w_{ki} x_i(t) \tag{2}$$

and often, the nonlinear sigmoid function

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

is used as the nonlinear output. It is monotonically increasing, with range from 0 to 1.

The outputs y_k of these second layer units are multiplied by another set of weights v_{jk} and the set of products $\{v_{jk} y_k\}$ becomes the set of inputs to processing unit j of the third layer. The third layer in this example network consists of one processing unit for every output the network must provide. If the network must output a pitch and a duration, it may be that the third layer will consist of two units, one for the pitch and one for the duration. There are many possible ways to represent values on the input and output of a network that, especially in the music domain, may reflect some domain structure, and these will be examined on a system-by-system basis in later sections. The network is trained by incrementally adjusting the weights on the connections so as to reduce some function of the network's output error E. This is the backward pass. Typically,

$$\Delta w_{ki} = -\alpha \frac{\partial E}{\partial w_{ki}} \tag{4}$$

and similarly

$$\Delta v_{jk} = -\alpha \frac{\partial E}{\partial v_{jk}} \tag{5}$$

with scalar learning rate α. The commonly used gradient-descent backpropagation algorithm (Rumelhart et al. 1986) propagates the error gradient back through the weights and nonlinear (but differentiable) functions in the processing units, using the chain rule to give general equations such as

$$\Delta w_{ki}(t) = \alpha \delta_k(t) x_i(t) \tag{6}$$

and

$$\Delta v_{jk}(t) = \alpha \delta_j(t) y_k(t) \tag{7}$$

where $\delta_k(t)$ and $\delta_j(t)$ are each a function of gradients multiplied by weights.

2.2. Feedback or Recurrent Networks

A recurrent network uses feedback from one or more of its units as input in choosing the next output. This means that values generated by units at time step $t-1$, say $y(t-1)$, are part of the inputs $x(t)$ used in selecting the next set of values $y(t)$. A network may be fully recurrent, i.e., all units are connected back to each other and to themselves, or some part of the network may be fed back in recurrent links. This section includes descriptions of several kinds of recurrent networks that have been specifically used in musical systems, ordered chronologically. The topology of each network is discussed, as is its forward pass to generate outputs, and its backward pass, to incrementally update the weights, taking into consideration the recurrence. The equations for the forward and backward passes are given. However, the derivations are left to the individual citations. In all cases, the derivations are instantiations of (sometimes-modified) gradient descent.

2.2.1. Jordan Networks. Jordan recurrent networks (Jordan 1986) include two types of recurrent links as shown in Figure 1. The first type is a link from the output layer back into the input layer to a set of input units, labeled *context units*. The network outputs depend not only on the external inputs, as in a feedforward net, but also on the outputs at the previous time step; i.e., $in(t) = out(t-1)$. The second type is the self-recurrent context unit. The self recurrence is from the input of the context unit back into the input, so the true input to a context unit, $unitin(t)$, is calculated

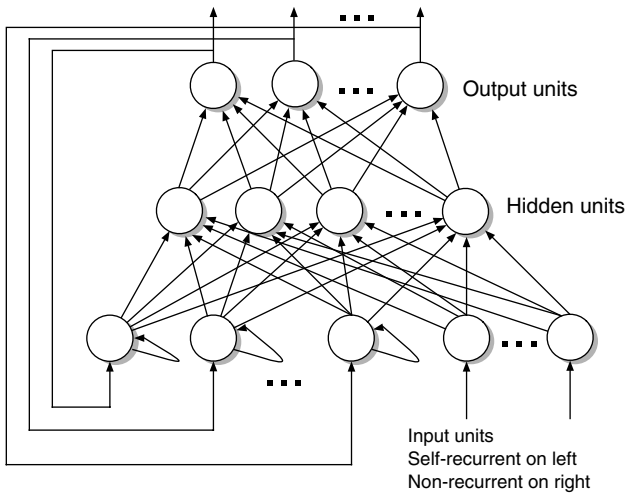

Figure 1 Jordan Recurrent Network Showing Input Context and Output Recurrence

as a combination of its past value $unitin(t-1)$ and of $in(t)$:

$$unitin(t) = \alpha unitin(t-1) + (1-\alpha)in(t) \tag{8}$$

with decay factor $0 < \alpha < 1$. The figure also shows non-recurrent external inputs. The recurrence on the context units provides a decaying history of the output over the most recent time steps. As in feedforward networks, the output units can be either linear or nonlinear functions of summed weighted inputs. The hidden units are nonlinear (sigmoid or hyperbolic tangent function).

Consider the problem of updating weights in a recurrent network. At each time step before the network is fully trained, the outputs are incorrect. However, the outputs and their incorrect values are being used as inputs to the network. How can the weight update equations be adjusted for these incorrect inputs? Williams and Zipser (1988) suggested teacher forcing, a method useable with Jordan networks. Since the *target* output is known during training, its value can be fed back to the input context units, rather than the actual output. In other words,

$$in(t) = out_{target}(t-1). \tag{9}$$

This means that the weight-update equations can be the feedforward network backpropagation equations, with $out_{target}(t-1)$ used as input to the context units. There are two drawbacks to this method. First, it is not useful for dealing with recurrence in hidden units, where the target output is not available. Second, the actual outputs may never be exactly equal to the targets. So when the network is used with new examples after learning, the actual output is fed back, and will include variations not present during training. Nonetheless, this is a useful method that has been used to train Jordan networks by both Todd (1991) and Franklin (2000) (see §§3.1 and 3.2).

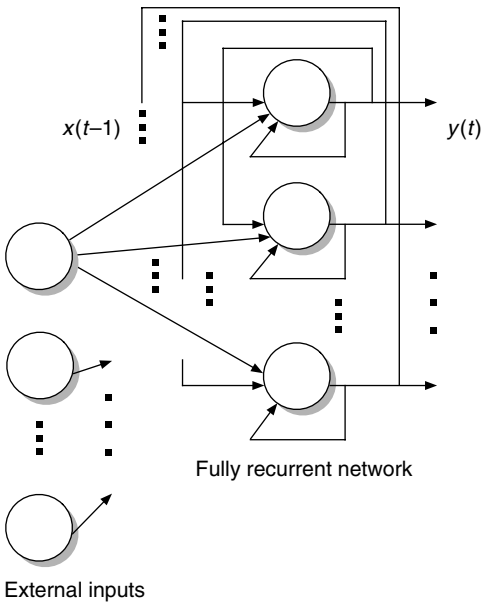

Figure 2 A Fully Recurrent Network, with External Inputs on Left

Notes. Network outputs are at right. The full set of inputs, including recurrent links, is shown as $x(t-1)$, and the network outputs are shown as the vector $y(t)$.

2.2.2. Backpropagation Through Time. Backpropagation through time (BPTT) is an algorithm that will work with a fully recurrent network as shown in Figure 2. It does not rely on teacher forcing. Suppose $x_i(t)$ is the set of all external inputs at time t (denoted as *Inputs*) plus the set of current outputs of all units (denoted as *Units*) in the network, $y_k(t)$ (Rumelhart et al. 1986, Campolucci 1998). For each unit k in the network, the output

$$y_k(t) = f_k(net_k(t)), \tag{10}$$

where f_k is a nonlinear function such as the sigmoid or hyperbolic tangent and, as we would expect from a forward pass similar to the feedforward network,

$$net_k(t) = \sum_{i \in Units \cup Inputs} w_{ki} x_i(t-1), \tag{11}$$

where the forward pass at time t depends on values at the previous time step $t-1$. Notice that the concept of network layers is eliminated by the full recurrence. BPTT is a batch algorithm where the feedforward pass is done over all examples in one sequence, and at each step in the sequence all errors are saved, along with all inputs to the units and all unit states.

Considering each unit k, the weight update for each weight w_{ki} connecting either unit i or external input i into unit k depends on summing terms over one whole sequence of time (compare to the simpler (6)),

$$\Delta w_{ki} = \alpha \sum_{\tau=t_0}^{t_1} \delta_k(\tau) x_i(\tau - 1), \tag{12}$$

where x_i is the ith input to the unit, and $\delta_k(\tau)$ is a function of derivatives and of errors at time τ and of *future* δ_is. All $\delta_l(\tau+1)$, for each unit l, are used to update each $\delta_k(\tau)$. We calculate the δ_k starting at the last time step $\tau = t_1$, and move its calculation back through time to step t_0:

$$\delta_k(\tau) = \begin{cases} f_k'(net_k(\tau))2e_k(\tau) & \tau = t_1 \\ f_k'(net_k(\tau))\left[2e_k(\tau) + \sum_{l \in Units} \delta_l(\tau+1)w_{lk}\right] & \\ & t_0 \le \tau < t_1. \end{cases} \quad (13)$$

This is the means by which errors are propagated back in time, from all units to each one unit. Conceptually, the network is unfolded and considered as a large many-layered feedforward network, with one layer per time step. $e_k(\tau)$ is the error between the desired or target output $y_d(\tau)$ and the unit k's actual output, $y_k(\tau)$:

$$e_k(\tau) = y_d(\tau) - y_k(\tau). \quad (14)$$

If the desired target is only presented at the end of the epoch, at $\tau = t_1$, $e_i(\tau)$ may only be nonzero at the end of the epoch. Also, e_k is only nonzero for units designated as output units for which targets are available. Any non-output, unit-weight updates are completely dependent on the time series of δ corrections. Once the δ_k are calculated for all τ, the weight update in (12) may be made.

In our early experiments with BPTT, we used only fully recurrent units within BPTT and designated one as the output unit that would be compared to the desired output at each step. Another option is to use the BPTT fully recurrent network as a nonlinear recurrent preprocessor to a standard nonlinear feedforward network. The feedforward network's outputs are compared to the target values; errors are formed and backpropagated through the feedforward network; and the error gradients from the feedforward net are passed back into the BPTT network as the errors $\{e_k\}$. The feedforward network can be implemented in batch mode, one batch per example sequence to be learned. Our experiments were more successful with this approach, and Mozer (1994) used this configuration in his CONCERT system (§3.3). It is possible to use truncated BPTT (Williams and Peng 1990) in an on-line manner, where only the most recent h values are used in the equations to compute the $\delta(\tau)$ values. While this method has been used in the control-engineering field, it has not been used for music applications.

2.2.3. Long Short-Term Memory (LSTM). The long short-term memory or LSTM network (Hochreiter and Schmidhuber 1997, Gers et al. 2000) is a significant departure from the other networks in that it employs a hidden layer of memory blocks that can be thought of as complex processing units, as shown in Figure 3. We will describe this network in more detail than the others, because it is more complex, and because it is the network we found to be most useful. The network uses a set of external inputs, provides a set of standard outputs, and contains the set of memory blocks. Rather than being one typical unit that sums its weighted inputs and passes them through a nonlinear sigmoid function, each memory block contains several units. Figure 4 shows a more detailed view of memory block j with n memory cells. First, there are one or more self-recurrent linear memory cells. Second, each unit contains three gating units that *are* typical sigmoid units, but are used in the unusual way of controlling access to the memory cells. One gate learns to control when the cell's outputs are passed out of the block, one learns to control when inputs are allowed to pass in to the cell, and a third one learns when it is appropriate to reset the memory cells. The lines leading out of the top of the block from the cells are the memory-block outputs that are fed into the output layer along with the outputs of all other memory blocks. The outputs of all blocks are also fed back recurrently to all of the memory blocks and are used to form net_{out}, net_{ϕ}, and net_{in}. The small black squares denote multiplication; e.g., $y^{in_j}(t)$ multiplies all of the $g(net_{c_j^v}(t))$.

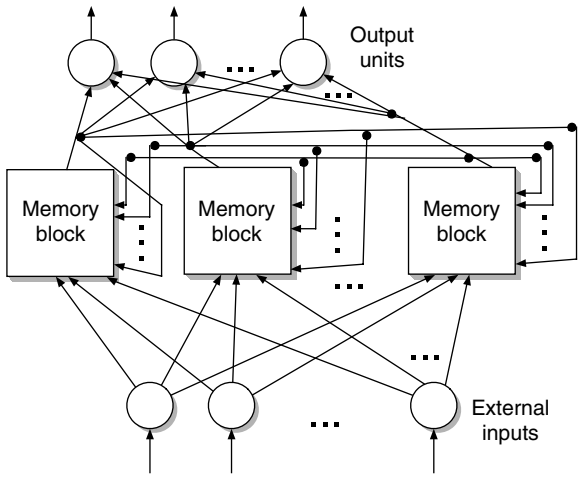

Figure 3 An LSTM Network with Recurrent Memory Blocks in the Hidden Layer Between the Input Layer and the Output Layer

LSTM's designers were driven by the desire to design a network that could overcome the vanishing-gradient problem (Hochreiter et al. 2001). Over time, as gradient information is passed backward to update weights whose values affect later outputs, the error/gradient information is continually decreased by weight-update scalar values that are typically less than one. Because of this, the gradient vanishes. Yet,

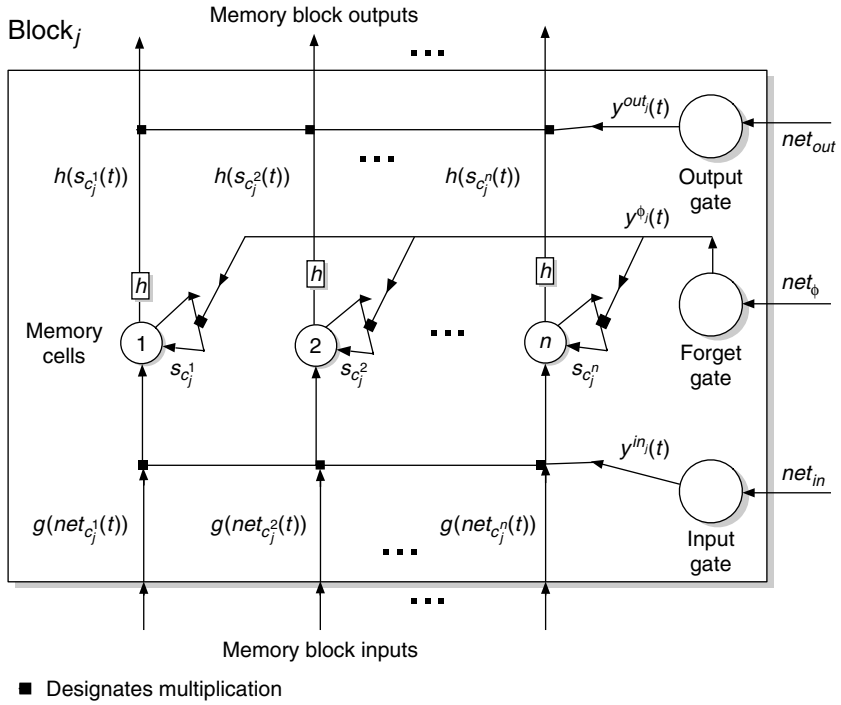

Figure 4 An LSTM Memory Block Showing n Memory Cells and Gates Learned by Nonlinear Units Receiving Either Inputs or Values from Recurrent Connections with Other Memory Blocks

the presence of an input value way back in time may be the best predictor of a value far forward in time. LSTM offers a mechanism where linear units can latch onto important data and store them without degradation for long periods of time, in order to decrease vanishing-gradient effects.

Referring again to Figure 4 and using the notation of Gers et al. (2000), c_j^v refers to the vth cell of memory block j. The memory-block inputs become inputs to each cell. For cell c_j^v, the inputs are multiplied by weights $w_{c_j^v m}$. These products are then summed to form $net_{c_j^v(t)}$, which is then passed through sigmoid function g, as shown at the bottom of Figure 4. The output of memory cell c_j^v is

$$s_{c_j^v}(t) = y^{\phi_j}(t)s_{c_j^v}(t-1) + y^{in_j}(t)g(net_{c_j^v(t)}), \qquad (15)$$

where $s_{c_j^v}(0) = 0$. By its role as multiplier in (15), the input gate output $y^{in_j}(t)$ is gating the entrance of new inputs, $g(net_{c_j^v(t)})$ into the cell. With a sigmoid output (see (3)), the value of $y^{in_j}(t)$ can swing between 0 and 1, allowing no access or complete access. Furthermore, each block j's forget-gate output $y^{\phi_j}(t)$ is gating the cell's own access to itself through its multiplication of $s_{c_j^v}(t-1)$ in (15), effectively resetting the cell when information it is storing is no longer needed. The original LSTM network did not include forget gates. Elimination of them is easily implemented by just setting $y^{\phi_j}(t)$ to be a constant 1. The cell's output $s_{c_j^v}(t)$ is passed through a sigmoid function, h, with range $[-1, 1]$, and then it may be passed on as an output of the memory block according to

$$y^{c_j^v}(t) = y^{out_j}(t)h(s_{c_j^v}(t)) \qquad (16)$$

where again we see gating in action. The output gate's output $y^{out_j}(t)$, ranging between 0 and 1, may allow $h(s_{c_j^v}(t))$ to pass out of the memory block, or it may inhibit it, by multiplying by 0. $y^{out_j}(t)$ is a sigmoid function of a weighted sum of inputs $net_{out_j}(t)$ ($y^{out_j}(t) = f(net_{out_j}(t))$) that are received via recurrent links from the memory blocks and from the external inputs to the network. Similarly, $y^{\phi_j}(t) = f(net_{\phi_j}(t))$ and $y^{in_j}(t) = f(net_{in_j}(t))$.

The weight updates for each block of the LSTM network are complex because of the use of the n memory cells and the three gates that control these n cells within each block. Furthermore, each output unit of the whole network has a set of weights used to multiply the values coming from the memory blocks. Each gate has a set of weights that it uses to multiply its inputs (recurrent inputs from all the memory blocks and also external inputs) and then pass through a sigmoid. Each cell has its own set of weights $w_{c_j^v m}$ used to calculate $net_{c_j^v}(t)$. We go through the steps of this calculation here.

Starting with the network-output units, the network's outputs $y^k(t)$ are the weighted sums $net_k(t)$, as in (2), passed through a sigmoid function f. The

output errors are passed back through the derivative of the sigmoid function f to obtain the error gradient

$$e_k(t) = f_k'(net_k(t))(t^k(t) - y^k(t)), \tag{17}$$

where $y^k(t)$ is the output of output unit k and $t^k(t)$ is its target (e.g., $t_k(t)$ may be the current target pitch). The weights connecting memory-block outputs to network outputs are updated using the errors

$$\Delta w_{km}(t) = \alpha e_k(t) h_m(t) y^{out_j}(t), \tag{18}$$

where $h_m(t) = h(s_{c_j^v}(t))$ for some block j and some cell c_j^v in that block and where $y^{out_j}(t)$ is the output gate output for the same block j. Compare this to (6), where now the input to the output unit is $h_m(t)y^{out_j}(t)$.

Inside memory block j, the output of each output gate, $y^{out_j}(t) = f_{out_j}(net_{out_j}(t))$, multiplies every $h(s_{c_j^v}(t))$ in that jth block and, therefore, the weight update for each output gate weight follows (6) as well but reflects those n products and their effects on the output gate's weight updates:

$$\Delta w_{out_{jm}}(t) = \alpha f_{out_j}'(net_{out_j}(t)) \sum_{k \in output\ units} e_k(t) \cdot \sum_{v=1}^{n} w_{kc_j^v} h(s_{c_j^v}(t)) x_m(t), \tag{19}$$

where $x_m(t)$ is the mth input to the output gate. This is the means by which the value $y^{out_j}(t)$ is learned. In other words, the network output errors are propagated back into the jth output gate, from each output unit through the weights connecting all of the cell outputs for block j to the output units.

The errors $e_k(t)$ are backpropagated further to obtain errors at the memory-cell level, according to

$$e_{s_{c_j^v}}(t) = y^{out_j}(t) h'(s_{c_j^v}(t)) \sum_{k \in output\ units} w_{kc_j^v} e_k(t). \tag{20}$$

The output gate's output $y^{out_j}(t)$ is simply a multiplier in this equation. Whereas its role in the computing of the outputs of the network in the forward pass is to determine if information from the cell is allowed out to the output units, its analogous role here in the backward pass is to allow or inhibit error information from flowing back through to the cell. If the cell contributed to the network output, it should also receive its share of the resulting error.

In order to update the weights $w_{c_j^v m}$ on the inputs to the cells and the weights $w_{\phi_j m}$ on the forget gate, as well as the weights $w_{in_j m}$ on the inputs to the input gate, these errors, $e_{s_{c_j^v}}$, must lastly be backpropagated through the memory cells. The cell weights $w_{c_j^v m}$ are updated according to how much they contributed to the error. The input and forget gates' weights, w_{in_j} and w_{ϕ_j} respectively, are updated depending on the sum of the errors of all the n cells (in their block j) that they gate. In other words,

$$\Delta w_{c_j^v m} = \alpha e_{s_{c_j^v}} \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}, \tag{21}$$

$$\Delta w_{\phi_j m} = \alpha \sum_{v=1}^{n} e_{s_{c_j^v}} \frac{\partial s_{c_j^v}(t)}{\partial w_{\phi_j m}}, \tag{22}$$

and

$$\Delta w_{in_j m}(t) = \alpha \sum_{v=1}^{n} e_{s_{c_j^v}} \frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}}, \tag{23}$$

where n is the number of cells in block j. Recalling from (15) that the memory cells are self-recurrent, these three partials are calculated using (15) as

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} y^{\phi_j}(t) + g'(net_{c_j^v}) y^{in_j}(t) x_m(t), \tag{24}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\phi_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\phi_j m}} y^{\phi_j}(t) + s_{c_j^v}(t-1) f_{\phi_j}'(net_{\phi_j}(t)) x_m(t), \tag{25}$$

and

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y^{\phi_j}(t) + g(net_{c_j^v}(t)) f_{in_j}'(net_{in_j}(t)) x_m(t). \tag{26}$$

Notice they all have the form

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{lm}} y^{\phi_j}(t) + \delta_l(t) x_m(t). \tag{27}$$

The only recursive weight-update equations are those involving the cell outputs $s_{c_j^v}$. The weight updates are actually estimates similar to the truncated backpropagation through time with $h = 1$ (as mentioned at the end of §2.2.2). The crucial element that leads to this network's success is the ability of the memory cell to "cache" error/gradient information for later use, as can be seen in (15) and (24–26).

In the configuration shown here, a single layer of nonlinear output units is attached to the output of the network. This could be a feedforward network with more than one hidden layer; equations are given in Hochreiter and Schmidhuber (1997). Also, a recurrent link may be added from the output layer to the inputs of the network, as is done in the simpler Jordan network. Eck and Schmidhuber (2002) take this approach in using this network for learning blues melodies, as we describe in §3.4.

3. Recurrent Networks for Music

Here we present several implementations of music systems that use the recurrent networks described in §2.2.

3.1. Using Jordan Networks—Melody Learning and Composition

Todd (1991) used a Jordan recurrent network (§2.2.1) in a system that can learn to reproduce songs. With the output of the network fed back to the input layer, and with a recurrent link on each input unit, the actual input is a decaying average of the most recent output values, providing a decaying memory of the melody.

How is this network used to reproduce a song? Todd's idea is to split time into 16th note fractions. Each iteration of the network produces the next 16th note fraction. During training, a song is given as a sequence of pitches, split into 16ths, to the network. The network must produce the next pitch on its output. One of the output units is called a Note Begin unit and is trained to output 1 if a new note is beginning. To output an eighth note of pitch E4, E4 is output for two iterations (two 16ths) and the *note-begin* is 1 for the first iteration and 0 for the second.

Todd uses one input for each pitch and one output for each pitch, in a "localist" representation of pitches, using 14 pitches in the key of C major, from D4 to C6. D4 is represented as 10000000000000, E4 as 01000000000000, F4 as 00100000000000, and so on. For example, to output D4 as an eighth note starting at time step t:

Step	Pitch Outputs	Note Begin Output
t	10000000000000	1
$t+1$	10000000000000	0

There are also several non-recurrent inputs called *plan inputs*. The Jordan network was originally designed to learn several plans, in the artificial-intelligence realm of planning, each one step by step. Here, the network learns several songs, pitch by pitch. The plan inputs indicate which song is being learned. The plan/song representation is similar to the pitch representation, with one input per plan/song. Thus if the network is being trained to learn song 1 of 3, the song inputs are 100, and they are 010 while learning song 2 of 3, and 001 for song 3.

In order to output a rest, all output units must be off, or below a threshold. Todd was able to train this network to learn melodies of up to 20 notes and rests that contain eighth, quarter, or dotted quarter notes, and to use one network to learn three melodies. New songs can be generated by the trained network either by varying and mixing the plan input values, or by introducing a new "seed" melody on the context inputs and recording the subsequent output.

3.2. Using Jordan Networks—CHIME

We use Todd's design (§3.1) as a basis for a two-phase learning system called CHIME (Franklin 2000) that, in phase 1, learns three 12-bar jazz melodies. The Jordan network is used with context and plan inputs. A range of two chromatic octaves is possible, leading to 24 context inputs and 24 outputs, where pitches are represented in the same type of localized representation (one bit or unit dedicated to each possible pitch). We too use a note-begin output unit but also add an explicit output unit for a rest because of the long rests in the learned melodies. An additional set of 12 inputs provides information about the underlying chords of the song. The 12 bits correspond to 12 chromatic pitches, four of which are 1, and eight of which are 0. The four "on" pitches are the chord tones. Chords are inverted to fit within the 12 inputs (i.e., no octaves are represented). For example, C7 is represented as 100010010010 (C, E, G, B-flat), and F7 is 100101000100 (F, A, C, E-flat inverted to C, E-flat, F, A).

Chords provide the harmonic structure of a song. Each individual chord provides a local context and chords change at perhaps a tenth or twentieth the rate at which notes change. The output units are trained with backpropagation, and the recurrence is managed by teacher forcing (Williams and Zipser 1988, Todd 1991).

In the second phase (Franklin 2002), more units are added to the Jordan network, and the output units are further trained via reinforcement learning to be able to improvise jazz. A scalar reinforcement value that indicates, numerically, how good or bad the output is, replaces the explicit error information on the output unit weight updates. The reinforcement value is generated by a set of rules for local in-time improvisation. This network learned to increase the reinforcement value over time, and an analysis of its improvisation shows that it not only generally heeds the improvisation rules but also employs parts of the original melodies learned in the first phase.

After both phases, the network could be used to "trade fours" with a human player. The human would improvise over four bars, and then the network would take the sequence of human notes and use it as its inputs to generate its responding four-bar improvisation.

Because there were several jazz-improvisation rules, we became concerned with the system's ability to learn the individual phenomena. It was difficult to discern this when analyzing its improvisations. Also, in the part of phase 1 in which the network learns to reproduce three songs, the songs' pitches and durations were never learned exactly. This was partly because of the limitations of rhythm created by restricting the timing to be one-sixteenth note per network iteration but also because of the limitations of the network itself. These concerns led us to our current study, as we will explain more in this paper.

3.3. Using BPTT—CONCERT

Mozer (1994) developed a system called CONCERT that is a recurrent network that can predict note-by-note and can also learn a somewhat coarser musical structure, at the phrase level with several notes per phrase. It uses a novel representation of pitch, duration, and chord that has a psychological, musical basis. Mozer's careful analysis of the behavior of the network for each task presented includes comparisons showing that the network is more general and concise than second and third-order probabilistic transition-table approaches.

CONCERT uses the backpropagation through time (BPTT) algorithm described in §2.2.2. The network is fully connected; each recurrent unit receives, in addition to the set of external inputs $x_j(n)$, the output of all of the recurrent units, including itself, at the last step $n-1$. Unlike Todd's architecture, n is not a time increment but rather a note increment. At each iteration of the network, the pitch, duration, and chord (if used) are outputs. Inputs are also pitch, duration, and current chord (if used), in a representation denoted PHCCCF described below.

The output layer in the network is non-recurrent; i.e., it is a feedforward layer attached to the outputs of all units in the recurrent network. This set of units is divided into three groups, providing the same pitch, duration, and chord configuration as is used in the PHCCCF input representation described below. The outputs of the final layer are treated as probabilities. A final layer that enables a probabilistic interpretation of the network outputs is useful for generating new compositions. A log-likelihood function involving the L2 norm of the actual vs. target outputs is minimized with BPTT training of the recurrent units.

3.3.1. PHCCCF Representation of Notes. Mozer uses a psychologically based representation of musical notes derived from Shepard (1987). In his first set of experiments, chords are not used. There are two sets of outputs (and two sets of inputs), one set for pitch and the other for duration. One pass through the network corresponds to a note.

Figure 5 shows the chromatic circle (CC) and the circle of fifths (CF), used with a linear octave value called *pitch height* (PH) for CONCERT's pitch representation. Six digits represent the angular position of a pitch on the CC and six more its angular position on the CF. C is represented as 000000 000000, C# as 000001 111110, D as 000011 111111, and so on. Mozer uses -1, 1 rather than 0, 1 because of implementation details. PH is represented as a single scalar input that maps the 48 pitch values between C1 and C5 to values between 1 and 20.

For chords, CONCERT uses a modified overlapping subharmonics representation of Laden and Keefe (1991). Each chord tone starts in Todd's 12-bit binary

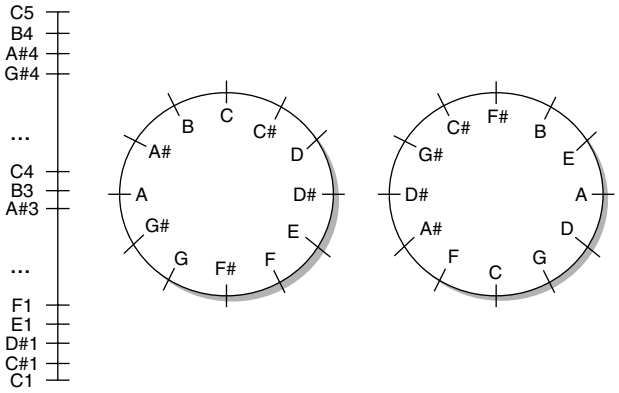

Figure 5 PHCCCF: Pitch Height, Chromatic Circle, Circle of Fifths Representation of Shepard and Mozer

Notes. Pitch position on the PH scale and on each circle CC and CF determines its representation.

representation, but five harmonics (integer multiples of the chord tone frequency) are added. The pitch C3 becomes C3, C4, G4, C5, E5. Both Laden and Keefe and subsequently Mozer use three-tone chords or triads only, because the harmonics of the 7th of the chord do not overlap with the triad harmonics. The C major triad chord: C3, E3, G3, with added harmonics, becomes C3, C4, G4, C5, E5, E3, E4, B4, E5, G#5, G3, G4, D4, G5, B5. The triad pitches and harmonics give an overlapping representation, where each overlapping pitch adds one to its corresponding input. Using the localized chord representation on a range of C3 through C7 requires 49 inputs. The C major triad is represented as

1000100100001001002000110102001100100000000000.

A 2 appears in the G4 and E5 positions, tones in which the C major triad harmonics overlap. In Mozer's implementation, the octave information is dropped, bringing the number of inputs back to 12 and introducing more overlap. Also, each overlapping pitch is weighted according to its harmonic number in the chord tone. C3 and its harmonics C3, C4, G4, C5, E5 contribute 1, 0.5, 0.25, 0.125, 0.0625 to their respective pitch inputs. In other words, $1+0.5+0.125$ is added to the input for C, 0.25 to the G input, and 0.0625 to the E input. An additional 13th chord input value is "on" if the chord is a tonic, subdominant, or dominant chord. This has its basis in human-perceived chord similarity but is also needed because only triads of chords are used. Furthermore, this assumes the song is written in one key throughout.

3.3.2. CONCERT's Duration Representation. Figure 6 shows the duration representation used in CONCERT. Analogously to PHCCCF, durations are represented as positions on three scales, where a quarter note is divided into 12 subdivisions. The angular positions on each of the mod 4/12 circle and the

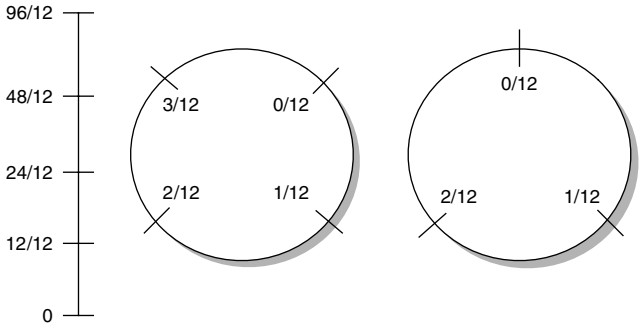

Figure 6 Duration Representation of Mozer: Duration Height, Mod 4/12 Circle, and Mod 3/12 Circle

mod 3/12 circles is determined by the remainder after first dividing by 12, then by dividing by 4 or 3, respectively. The duration height is the amount of the duration divided by 12. This duration scheme is more flexible than that of Todd's sixteenth notes (1/4th of a quarter note). Here, the smallest duration is 1/12th of a quarter note and, e.g., quarter and eighth note triplets can be represented.

3.3.3. CONCERT Results. In the first sets of experiments with CONCERT, only pitch and durations are learned. CONCERT was able to learn to reproduce diatonic scales and to predict the next note in the diatonic scale in a not-before-seen test set. Its performance was superior with the PHCCCF representation vs. the localized representation. One of the difficult tasks was a 21 note melody with an AABA phrase structure. The trouble was in predicting the first note of the melody in the third A. Mozer later combines the Jordan context units (8) with the fully recurrent units of BPTT to obtain an increase in performance.

Further experiments in composition are carried out first by training the network on Bach melodies and generating new Bach-like melodies. Secondly, harmonic structure is incorporated through chords inputs/outputs, and the network is trained on waltzes and then composes new waltzes, with their new corresponding chord structure.

3.4. LSTM for Blues Music

Eck and Schmidhuber (2002) describe research in using the LSTM recurrent learning network (§2.2.3) to learn and compose blues music. Their model of blues music is a standard 12-bar blues chord sequence over which music is composed/improvised. They successfully trained an LSTM network to learn a sequence of blues chords. Similarly to Todd, they split time into eighth-note increments, with one network iteration per eighth-note time slice. The network must be able to output a chord value for as many as eight time increments (for a whole-note chord) and then output the next chord in the sequence. Each chord has a duration of either eight or four time steps (whole-note or half-note durations). As with the Jordan network (§3.2), chords are represented as sets of three or four (triads or triads plus the seventh) simultaneous note values of 1 in a 12-note input representation, with non-chord note inputs set to 0. Chords are inverted to fit within one octave.

The network contains four cell blocks, each containing two cells. The cell blocks are fully connected to each other. The output layer that determines the next chord value is fully connected as well, to the cells blocks and to the input layer. This is a modified configuration of the one presented in §2.2.3. In addition to the forget gates, the whole network is reset if a large error occurs. During a reset, the weight values are retained, but all other values such as partial derivatives, activations (outputs), and cell states are set to 0. This enables the network to recover sooner and learn faster.

Biases were preset for the four memory blocks, at -0.5, -1.0, -1.5, and -2.0, enabling the blocks to enter into the initial computations one by one. The learning rate is small at 0.000001. They also use momentum, set at 0.9. This is sometimes used in feedforward networks as well and provides a decaying filter on the weight updates:

$$\Delta w(t) = 0.9w(t-1) + 0.1\frac{\partial E(t)}{\partial w(t)} \tag{28}$$

The outputs are considered probabilities of whether the corresponding note is on or off. The goal is to obtain an output of more that 0.5 for each note that is supposed to be on in a particular chord. All other outputs should be below 0.5. The outputs are treated as independent; the error function used for each is the cross-entropy objective function

$$E_k = -t_k \ln(y_k) - (1-t_k)\ln(1-y_k) \tag{29}$$

where y_k is the value of output unit k. $\partial E_k/\partial y_k$ takes the place of the error e_k in (14). This network is able to learn a 12-bar blues sequence of chords that is a total of 96 network (8th note) increments long.

A second experiment includes both learning melody and chords with two subnetworks containing, again, four cell blocks each. The output of the chord network is connected to the input of the melody network (but not vice versa). The authors themselves composed melodies over each of the 12 possible bars. Each melody is composed of eighth notes only, one note per iteration. Rests and other durations are not included. The network is trained on songs that are concatenations of these 1-bar melodies over the 12-bar blues chord sequence. The melody network is trained until the chords network has learned according to the criterion. In music-generation mode, the network can generate new melodies using this training.

4. LSTM for Jazz-Related Tasks, Long Melodies, and Human/MIDI Rhythms

Our work as described in §3.2 initially used the Jordan network with the localized binary pitch representation and time-sliced network iteration scheme for duration. We became interested in LSTM networks because of our desire for networks that have (1) better ability to learn a song exactly, (2) better ability to learn long songs/sequences, and (3) better ability to learn cause and effect over long time spans. Also in our previous work on reinforcement learning, we constructed a reinforcement function that rewarded several types of phenomena. We decided to study specific jazz-related tasks, to try to determine how difficult they are. To give this effort more depth, we considered how the network might generalize across different keys, and also what it might generate if given new inputs.

Furthermore, we more deeply examined note representations, driven by the desire to include more music knowledge in input and output representations and to give the networks more flexibility in rhythm so swing style can be incorporated.

In this section we first describe our work in developing a new pitch representation based on major and minor thirds. We have also devised an explicit duration representation that takes Mozer's modular representation further and allows even more flexibility. We describe results in comparing these new representations with localized and PHCCCF representations, using LSTM networks on short musical tasks. And we consider generalization issues. Finally, we show that an LSTM network can exactly learn a long song with an intricate rhythm, using these representations.

4.1. Circles-of-Thirds Representation

The circles-of-thirds representation is inspired by both the localized binary and CCCF representations, and Laden and Keefe's (1991) and Mozer's (1994) chord representations. It is also a recognition that the basic chord tones are created by the major and minor third intervals. It includes a pitch as well as a chord representation, and results in a seven-digit value for a pitch or a chord. Figure 7 shows the four circles of major thirds, a major third being four half steps between pitches, and the three circles of minor thirds, a minor third being three half steps. In the figure and in this discussion, we assume enharmonic equivalence. The top row is the set of circles of major thirds, each read counter-clockwise. E is a major third above C, G# is a major third above E, and C is a major third above G#. Similarly, on the second row, E-flat (assumed to be equivalent to D#) is a minor third above C, F# is a minor third above D# and so on.

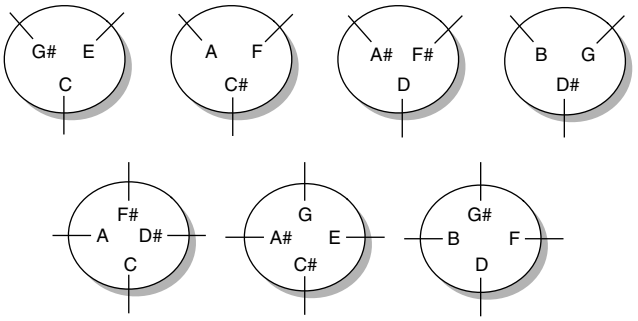

Figure 7 Circles-of-Thirds Pitch Representation

Notes. At top, circles of major thirds, at bottom, circles of minor thirds. A pitch is uniquely represented via these circles, assuming octave and enharmonic equivalence.

In our seven-bit representation of pitch, the first four bits indicate the circle of major thirds in which the pitch lies, and the second three bits, the circle of minor thirds. The index number of the circle the pitch lies in is encoded, unlike PHCCCF, in which it is the angular position on the circle. C's representation is 1000100, indicating major circle 1 and minor circle 1, and D's is 0010001, indicating major circle 3, and minor circle 3. D# is 0001100. Also unlike PHCCCF, pitches that are a half step apart (the minimum) do not have similar representations. A half step error in music can often sound out of place. In this representation, a pitch that has one bit out of place will still have either a common major or minor interval with the one intended. This may also make discoveries easier when this representation is used in reinforcement learning. In terms of neural computation itself, this is a concise representation that makes it easier to distinguish two different notes. That said, the PHCCCF does contain contrasting inputs, especially in its chromatic versus circle-of-fifths representations. It could very well be that a combination of PHCCCF and circles-of-thirds would be the best for very complex music computations. While the circles-of-thirds representation is not directly motivated by the work of Longuet-Higgins in characterizing musical intervals (Steedman 1994), this work may provide future guidance, especially if circles-of-thirds is combined with the PHCCCF representation.

The argument to have octave information as a separate input is a good one. The network is given the same pitch information independently of the octave, so it will not treat C3 as a completely different note than C4. Mozer stresses such similarities in his representation development. It also leads to a much more concise representation. Rather than using a single scalar pitch height as did Mozer, we currently include two single-bit inputs for octaves, one to indicate if the octave is C2 through B2 and the other to indicate if the octave is C4 through B4. If both bits are zero, the

default octave is C3 through B3. This octave information is needed for learning the long song, *Afro Blue* (§4.4), but not for the shorter musical tasks.

Chord progressions in jazz tunes include chords that differ in the seventh tone. Because the 7th chord tone is so important to jazz, our chords are the triad plus 7th (recall that Laden and Keefe ignore the 7th, as described in §3.3.1). We also include other chord tones in some experiments. Assuming the chord tones are the first, third, fifth, and seventh, using circles-of-thirds and no harmonics, we could represent the four chord tones as four separate pitches, each with a seven-bit representation for a total of 28 bits. However, it would be left up to the network to learn the relationship between chord tones. We borrowed from Laden and Keefe (1991) on overlapping chord tones as well as Mozer's (1994) more concise representation. The result is a representation for each chord that consists of seven values. No harmonics are included. Each value is the sum of the number of "on" bits from the circles-of-thirds representation for each note in the chord. For example, a C7 chord in a 28 bit circles-of-thirds representation is

1000100	1000010	0001010	0010010
C	E	G	B-flat

The overlapping representation is:

1000100	(C)
1000010	(E)
0001010	(G)
+0010010	(B-flat)
2011130	(C7 chord)

We in fact scale these values to lie between 0 and 1 since we have in our experience found networks to be more successful if their inputs are in the same range. The seven inputs for C7 are actually 0.6, 0, 0.3, 0.3, 0.3, 0.9, 0.

In other experiments, we use the C-major chord: C, E, G, B, represented as

1000100	1000010	0001010	0001001
C	E	G	B

The overlapping representation is:

1000100	(C)
1000010	(E)
0001010	(G)
0001001	(B)
2002121	(C major chord)

This we would scale to 1, 0, 0, 1, 0.5, 1, 0.5. We further discuss the chord representation later in the paper on an experiment by experiment basis. It is possible to represent embellished chords, such as altered chords (Berg 1990) with this representation. We anticipate a deeper study of this in the future.

4.2. Modular-Duration Representation

The vanishing-gradient problem is further exacerbated in configurations in which one iteration of the network corresponds to the minimal duration. A trade-off develops in which small note durations are desired; yet, the smaller the refinement of durations, the more network iterations are required to represent one duration. We focus on enabling the network to output the duration explicitly as does Mozer (1994), and we also extend Mozer's use of a modular representation. We are interested in moving beyond score-based durations and into learning human-like variations in duration that especially occur, and are encouraged, in jazz. Mozer refined a quarter note to 12 subdivisions, especially useful because 12 is divisible by 4 (to achieve 16th-note-level durations) and is divisible by 3 (to achieve quarter and eighth note triplets). We take this further by dividing quarter notes into 96 subdivisions, a standard called "ticks" in the Musical Instrument Digital Interface (MIDI) standard digital protocol (Messick 1988) and "clicks" in a music software package we use called Keykit (Thompson 2003). In a MIDI file the number of clock ticks per beat is specified at the beginning of the file. MIDI events are time stamped, relative to the previous MIDI event, in number of ticks.

Then a whole note, dotted half, half, quarter, dotted eighth, eighth, eighth triplet, sixteenth, sixteenth triplet, thirty-second, thirty-second triplet, sixty-fourth are 384, 288, 192, 96, 64, 48, 32, 24, 16, 12, 8, and 6 clicks, respectively (we also include 4, 3, 2, and 1). It has also been our experience that networks with a large number of inputs are less able to learn. We derived a modular duration representation of 16 bits. The 16th bit is 1 if the note duration divided by 384 is greater than or equal to 1, where $384 = 96 \times 4$, is the duration of a whole note. The 15th bit is 1 if the remainder after the duration is divided by 384 and then further divided by 288 is ≥ 1. The 14th bit is 1 if after dividing by 384, then 288, then 192 is ≥ 1, etc. Note that a dotted quarter can be represented as $96 + 48$.

As non-score examples, 55 is $48 + 6 + 1$, represented as 0000010000100001, and 289 is $288 + 1$, represented as 0100000000000001. Importantly, 289 is a dotted half note plus one click, an approximation to a dotted half note that could easily be played by a human performer and captured on MIDI input. With this representation we can represent any of the above standard score-notated durations, but we can also represent human-performed approximations or improvised durations. Also, in the future when we employ reinforcement learning, as mentioned with the circles-of-thirds pitch representation, it may provide an easy vehicle for exploration. However, one drawback of this method is that close numbers of ticks

may have radically different representations. This is a trade-off with the conciseness of the representation.

4.3. Results for Short Musical Tasks

We first experimented with the circles-of-thirds representation with three musical tasks and with an LSTM network (Franklin 2004a). The tasks are: (1) chord tones—given a dominant 7th chord as input, output in sequence the four chord tones; (2) chromatic lead-in—given each of 14 pairs of five-pitch sequences, output 1 at the end if the second, third, and fourth notes in the sequence are ordered chromatically and otherwise output 0; and (3) AABA melody—learn to reproduce one specific 32 note melody of the form AABA, given only the first note as input. This a memorization task. These are all pitch-sequence tasks and do not include durations. We found that the LSTM network can accurately learn the three short-sequence tasks. In our configuration, outputs are not fed back as inputs. The only recurrence is within the memory-block layer. We also tried several other kinds of recurrent networks with some limited success, but none were as successful as LSTM. Recurrent networks are nonlinear dynamic systems, many of which produce highly oscillatory, often unstable behavior. The clinching factor in our choice of LSTM is its consistent stability. We discuss these tasks further now, along with some follow-up generalization experiments.

Ignoring octaves, recall that both CCCF and the localized binary representations require 12 external inputs, and 12 output units if pitches are the outputs. The circles-of-thirds representation requires seven. There is a bias term used in LSTM that enables the blocks (specifically, the blocks' gates) to be "activated" one by one over time as it is learning. While we found -0.5 in the LSTM literature, we found -0.1 to work better for these tasks. The bias value of block 1 is 0, block 2 is -0.1, block 3 is -0.2, etc. In all experiments, we obtained better results with a lower learning rate on the output units than on the memory blocks. Also, including a direct link from input units to output units produced a much better rate of success. The number of iterations range from 10,000 to 15,000. We required precise outputs to be within 0.1 of the targets (which are always 0 or 1). When we first ran these experiments we were seeking a network that could exactly learn the specified output. We found that Jordan networks were unable to do this. However, when using recurrent networks for reinforcement learning and improvisation, we would like a component network that can provide exact riffs from known songs. Recall in our work with CHIME (§3.2) that a Jordan network first learned Sonny Rollins melodies in phase 1, and was further trained to improvise using reinforcement learning. We want the phase 1 network to learn these phase-1 melodies exactly. Secondly, most reinforcement learning techniques use some kind of predictive component that learns to attribute future rewards to current actions (or current rewards to past actions). Again, we are seeking precision here as we are developing techniques to combine these components with recurrent networks.

4.3.1. Chord Tones. Chord tones are pillars and cornerstones of jazz improvisation. If a chord is given as input, as part of a larger harmonic structure, say, an improvisor must be able to generate chord tones from that chord, to contribute to its larger improvisation. Dominant 7 chords are especially prevalent in jazz, and we needed to find out if the network could produce chord tones from the overlapping circles-of-thirds chord representation. In the chords task, each of the twelve dominant 7 chords, C7, C#7, etc. is presented, one at a time, as input for four increments. The network must first output the tonic, the third, the fifth, and then the (flat) seventh of the chord as output (e.g., input chord C7 for four increments, and output C, E, G, and B-flat). The chord-tones task is easy for an LSTM network containing ten memory blocks with one cell per memory block to generate the chord tones with 100% success. The learning rate for the seven output units is 0.2 and the memory block learning rate is 0.5. Note that, with just requiring the exact outputs in this way, these learning rates are much higher than reported by Eck and Schmidhuber (2002). In our generalization experiments, as Eck and Schmidhuber found, the learning rates had to be lower.

To see how well LSTM might generalize, we trained the LSTM network to generate chord tones for eight of the chords: C7, C#7, D7, D#7, E7, A7, A#7, and B7. The tonics for these eight chords are distributed evenly over the major and minor circles in Figure 7. After the network was trained to generate the four tones for these four chords, it was presented with the remaining four chords, F, F#, G, and G# in a test phase, with no training. With learning rates of 0.15 and 0.05 for the blocks and output units, respectively, with 15 blocks of two cells each, and after 12,000 epochs on the training set, Table 1 shows the target tone, and actual tone pairs. On the F chord, the output sequence is perfect, on F# all tones are correct except that it plays the tonic instead of the third (F# instead of A#). The G chord is also correct for three tones, but the tonic is missed; C# is output instead of the G. In the sequence of tones with G# as root, D is played instead of the tonic (D and G# share the same minor third circle), and then it settles onto the third, C. Considering the very small size of the training set, these are successful results.

4.3.2. Chromatic Lead In. Besides using chord tones in creating a melody, one effective technique of

Table 1 Target Tones and Actual Tones for Each Chord Tone Example

Chord	Tones
F	F, A, C, D#
	F, A, C, D#
F#	F#, A#, C#, E
	F#, F#, C#, E
G	G, B, D, F
	C#, B, D, F
G#	G#, B#, D#, F#
	D, B#, B#, B#

improvisation (Berg 1990) is to lead in to a chord tone with chromatic pitches just below or just above the chord tone. In this experiment, the network is given a set of seven pairs of sequences that it must classify. This is the kind of sub-evaluation that may need to occur in reinforcement learning. Each sequence contains five pitches. In the first sequence, the third note is a chromatic tie between the second note and the fourth note. Both the fourth and fifth notes are chord tones. The network should output a 0 at each time step, except the last, when the target is 1 if there is a chromatic lead in, and 0 otherwise. The positive sequences are from Berg (1990), and all occur over the Cma7 chord (with the 6th and 9th included as chord tones). There is no chord input to the network, however. The seven pairs of sequences, each sequence labeled with its correct final target, are:

c, d, d-, c, c 1 c, d, d, c, c 0
c, d, e-, e, e 1 c, d, d, e, e 0
g, g-, f, e, e 1 g, f, f, e, e 0
e, g, a-, a, a 1 e, g, g, a, a 0
a, b, b-, a, a 1 a, b, b, a, a 0
a, a, b-, b, b 1 a, a, a, b, b 0
d, e, e-, d, d 1 d, e, e, d, d 0

The chromatic lead in task turned out to be quite difficult. LSTM with circles-of-thirds learned this task with one cell each in ten memory blocks, using learning rates of 0.2 and 0.5. These learning rates are based on empirical studies over sets of experiments and reflect the best results for exact classification of all examples. This task is studied more in another paper (Franklin 2004b) that focuses on predicting the classification at the third iteration, rather than waiting until the end (a precursor to work in reinforcement learning).

4.3.3. AABA Melody. We start with the task of the network learning to reproduce exactly a melody that has an AABA form. The A form is an eight-pitch arpeggio over the Cma9 chord: C, D, E, G, G, E, D, C. The B form is an 8-pitch improvisation over the same chord, containing auxiliary pitches (Berg 1990): C, F, D#, E, F#, A, G#, F#. This produces a 32-note melody, presented as one example with 32 time steps to the network:

C, D, E, G, G, E, D, C
C, D, E, G, G, E, D, C
C, F, D#, E, F#, A, G#, F#
C, D, E, G, G, E, D, C

Recall that there is no feedback from the output pitches to input. The only external input is the representation for the C pitch, held constant for each of the 32 increments. Even a single AABA melody is a more difficult task than the chord tones task, requiring two cells in each of 15 blocks. The learning rates to learn one melody exactly are a rate of 0.05 on the output units and 0.15 on the blocks. This was an important experiment, needed to gain insight on how the network might work on much longer melodies.

We tried two new experiments with the AABA melody. First, we added two extra inputs that are bits 0, 1 when pitches from an A part are present on the input and that are 1, 0 when the B inputs are present. Thus the network receives "01" on these inputs for 16 increments, the "10" for eight increments, and then "01" for the remaining eight increments in the epoch. This dramatically decreased the number of epochs needed for learning to generate our chosen AABA melody; 8,000 epochs suffice, for 2 cells per block of the 15 blocks. Learning rates are 0.15 for the blocks and 0.05 for the output units. We also intentionally decreased the number of epochs used to train, in order to decrease the risk of overfitting.

We can also see what happens when the network is given a different input. For example, without any further training, the single note input C (the root of the underlying chord C major 7) may be replaced with E, the third of the underlying chord. The resulting melody is:

C, D, E, G, E, D, C, C
D, E, G, D, E, D, C, C
F, C, F#, A, F#, F#, F#, F#
C, D, E, G, G, E, D, C

This actually gives us a simple and direct way to cause the network to generate a rudimentary improvisation, and perhaps in the future to be one of several networks that can make a potential contribution to an overall improvisation. It produces a new melody, with variations on the original A, with repetition, and it replicates the final A of the form. The B part is distinct and has obvious substitutions of the original melody.

When the fifth, G, is substituted for the original C on input, without weight updates, the result is also a new melody. In these experiments, if the maximum of the outputs corresponding to the major circles is taken

as the index and the max of the three minor circles is taken as the minor circle index. We are careful to point this out here, since when the network learns the task exactly, the output representation is always clear, with one of the majors and one of the minors standing out, usually above 0.9 (on a scale from 0 to 1). When G was substituted, in one case the pitch could not be disambiguated. The two choices were F# or D, and F# was chosen (as one of the F# in the B part):

C, D, E, G, D, F#, C, C,
D, E, G, E, D, C, C, D,
F#, G, F#, F#, F, D, F#, D
C, D, D, E, G, A#, F#, C

Finally, the melody below results from B, the 7th of the C major chord, as input, and is quite similar to the one generated by the fifth:

C, D, E, G, D, F#, C, C
D, E, G, E, D, C, C, D,
D#, G, F#, F#, D, D, F#, D
C, D, D, E, G, A#, A#, C

We were also interested in discovering if the network could learn to generate transpositions of this melody. First, a single network was able to learn all 12 transposed melodies, given only the one underlying chord as input, for each 32-note melody. It was more successful when the major chord was given as input, rather than just the tonic, using the major-chord representation given at the end of §4.1.

Next, the single network was trained on eight of the twelve melodies and then tested on the remaining four, using the same tonics as for the chord tone train and test experiment in §4.3.1. This experiment was very hard. The network can learn to produce all of the eight transposed melodies on which it is trained, perfectly. Yet, on the test set, the best result was only a match of 9 of 32 of the notes. In order to achieve this, we had to make the learning rate much lower, and increase the number of epochs to 200,000. Also, we found that including other chord tones in the input chord, namely the second and the fourth, helped the network marginally in generalizing to the other four unseen transpositions. We note again that the training set is very small. In the future we will experiment with a larger set. Furthermore, in most jazz improvisation, the human or machine is given a harmonic structure that changes over the course of the song, and this rich context was not available to the network. We have used the underlying chord progressions as input in our previous work in CHIME and expect more capability with LSTM in the future. We do address this somewhat as well in the following section on learning a long melody.

4.4. Learning the Melody *Afro Blue*

We chose a jazz standard called *Afro Blue* composed by Mongo Santamaria and used both the circles-of-thirds and modular-duration representations with LSTM networks to learn this song. It is our experience that a very complex network with scores of inputs will not be able to learn a long intricate melody. So, inspired by Eck and Schmidhuber's (2002) use of separate networks to learn chords and melody, and also by Mozer's (1994) work in explicit learning of durations, we use two separate networks, one for melody and one for duration. We tested the network on two renditions of the song. One is a transcript of the "pure" score. The second is a MIDI-captured human performance. We first describe results for the human-played rendition. We used the same learning rates for all experiments: 0.05 for output units and 0.15 for blocks, values obtained from experiments on the short melody tasks of §4.3. Also, all networks have a direct link from inputs to outputs (as well as the links through the hidden layer of memory blocks), since all but one successful experiment on the short melody tasks required this link.

The human played melody has 101 notes (101 pitches, and 101 durations) while the original score contains 106 notes (including rests). Figure 8 shows an approximation to the musician's rendition of *Afro Blue* with the original underlying harmonic structure shown via the chord labels, as generated by Band in the Box music software (PG Music). It should be noted that software to generate a score from a MIDI file must necessarily make approximations that will "smooth" the intricate nuances of the actual note durations played. The song has a fairly complex coarse phrase structure. In the first experiments the network had only one single input. It is set to one whenever the current note is on the first beat of a bar. Otherwise, it is zero. The 101 pitches and 101 durations were learned exactly by a pitch and a duration network. The pitch network contains 15 blocks, with four cells each. When a rest appears, the seven target outputs are set to zero. The duration network contains 17 blocks, four cells each. Figure 9 shows LSTM's version, which is almost indistinguishable from Figure 8.

We also ran experiments where the current chord of the harmonic chord structure is presented as well as the one input indicating the first beat. The chord structure is AABB, repeated three times. The AABB form for the chords is:

—	F-7	—	G-7	—	Abmaj7 G-7	—	F-7	—
—	F-7	—	G-7	—	Abmaj7 G-7	—	F-7	—
—	Eb	—	Eb	—	Db Eb	—	F-7	—
—	Eb	—	Eb	—	Db Eb	—	F-7	—

The chords are represented in the overlapping circles-of-thirds representation. We compared this with

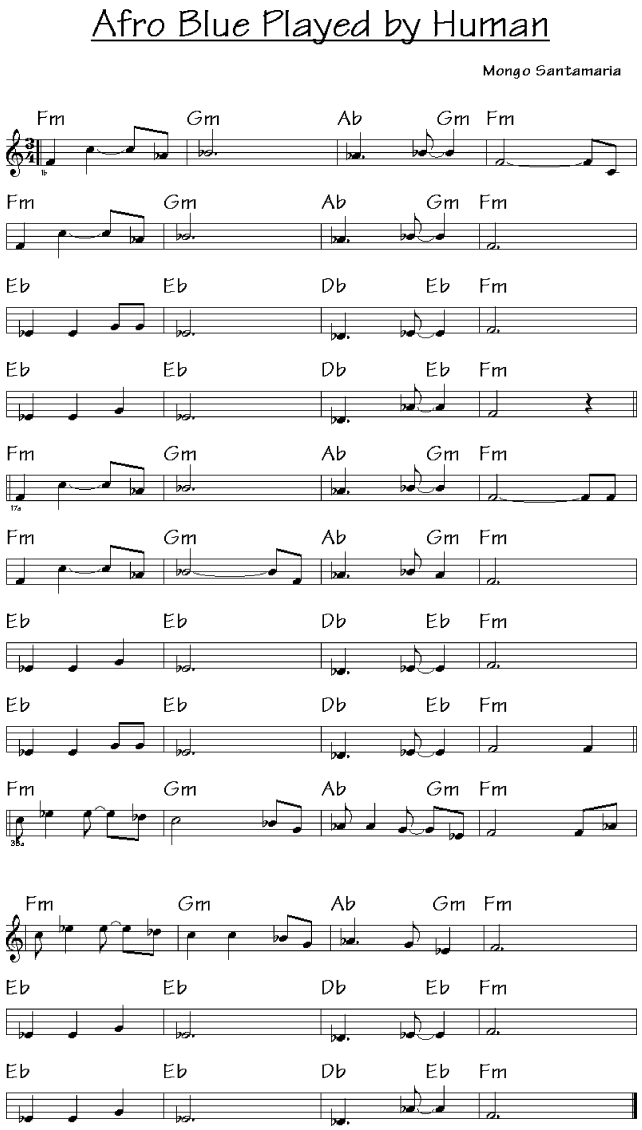

Figure 8 **Musician's Rendition of *Afro Blue***

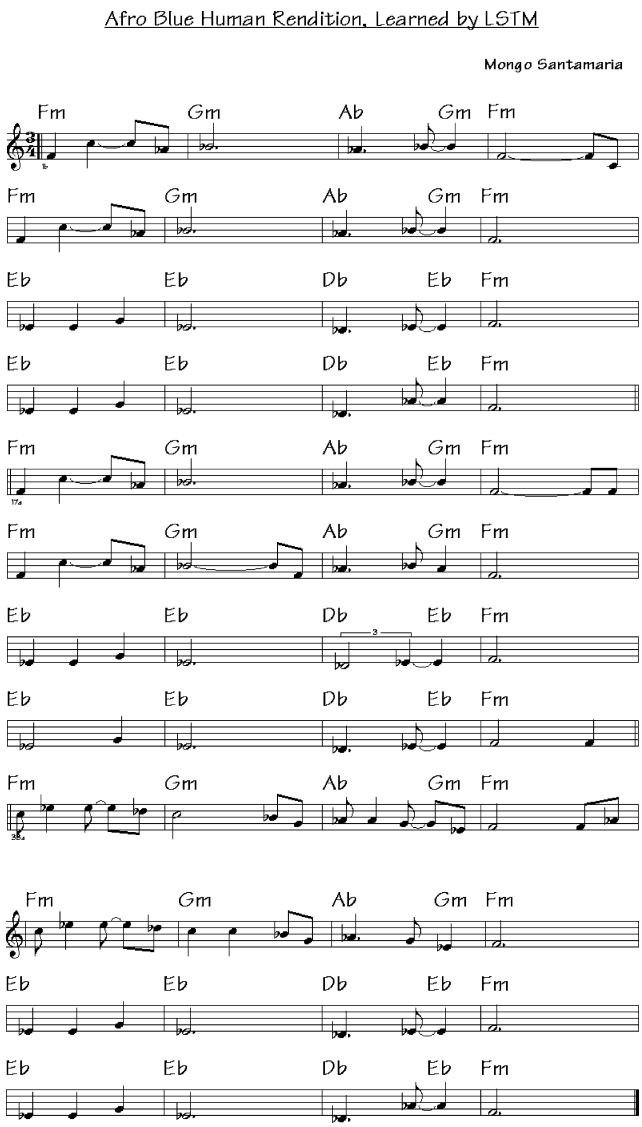

Figure 9 **LSTM's Version of the Human Rendition**

networks only receiving the one-bit beat input. In these comparisons, we ran networks for 15,000 epochs (presentations of the whole song), with a bias factor of −0.1, all with 18 memory blocks, each containing four memory cells. As can be seen in Table 2, both the pitch network and duration networks learn slightly more accurate outputs when given chord information in addition to beat information.

These are representative runs. The results do not vary significantly from one run to another, for a successful network configuration. From this we know that including chord information, and then using it for composing, is possible and will not diminish the networks' learning abilities.

It is not necessary to run the network for this many epochs to achieve useable results. For example, the maximum error during one run of the pitch network with beat input was 0.15 after 5,000 epochs. By comparing the outputs to a threshold of below 0.2 or above 0.8, the network outputs are correctly interpreted. We could even lower this to 0.4 and 0.6, respectively, and reduce the number of epochs required.

The LSTM network is able to learn the score-based form as well, although it was a more difficult task for the duration network. This initially surprised us, until we considered the redundancies in the score that could lead to temporal confusion for

Table 2 ***Afro Blue* from Human Rendition**

Type of network	Max output error
Duration: beat input	0.06
Duration: chords and beat inputs	0.03
Pitch: beat input	0.06
Pitch: chords and beat inputs	0.03

the network. The song's melody form is AABBAABBCCBB, with a slight variation at the end of the first A and at the end of the first C. The chord-changes form is AABBAABBAABB. However, the song's duration form is ABACDCDCACACDCDEFGHFGEFGHFGCDCDC. Consider the last four sub-phrases, DCDC. D is the duration sequence 96, 96, 96, 288, 144, 144, and C is the sequence of just 288. The equivalent in the human-performed durations are, for the same sub-phrase DCDC, 98, 129, 62, 291, 162, 132 then 282, and 96, 124, 65, 288, 161, 124, and then 289. These are quite different sequences, and also do not contain any exact repetition within them, as do the score's "pure" duration sub-phrases. We summarize the behavior of the two networks in Table 3, again with and without chord input. In these experiments, we started with 4,000 epochs and kept increasing by 1,000 until we found a number of epochs that produced all correct outputs. The pitch network is able to learn within the initial 4,000 epochs and results are the same whether chords are inputs or not. The duration network requires 6 cells per block (of 18 blocks). The chord information is very useful for the duration network, enabling it to learn to produce all correct outputs (with maximum error of 0.38) within 8,000 epochs. However, 8,000 epochs are not enough for it to learn without the chord information. At 15,000 epochs, the number used above in the human-rendition experiments, the duration network is able to learn to produce all correct outputs with just the one-bit beat input.

Now that the LSTM network can exactly learn the song *Afro Blue*, with the original chords as input, we can use some common reharmonizations, otherwise known as chord substitutions, to generate a new song. As can be seen in Figure 10, we have replaced the G-minor chord in the 2nd, 6th, 18th, 22nd, 34th, and 46th bars with the G half-diminished chord (G minor 7 flat 5); meaning where the network previously received the circles-of-thirds chord representation for G, Bb, D, and F, it now receives G, Bb, Db, and F. We have also replaced the F minor chord in the 4th, 8th, 20th, 24th, 36th, and 40th bars with the F half-diminished chord. Finally, we have replaced the E-flat major chord in bars 9, 13, 25, 29, 41, and 45 with the B-flat dominant seventh chord. The durations were not modified in this experiment. These are

Table 3 ***Afro Blue*** **from Score Rendition**

Experiment	Max output error	NumEpochs	Net config
Duration: beat inputs	0.09	15,000	6 cells, 18 blocks
Duration: chords and beat inputs	0.38	8,000	6 cells, 18 blocks
Pitch: beat input	0.08	4,000	4 cells, 18 blocks
Pitch: chords and beat inputs	0.08	4,000	4 cells, 18 blocks

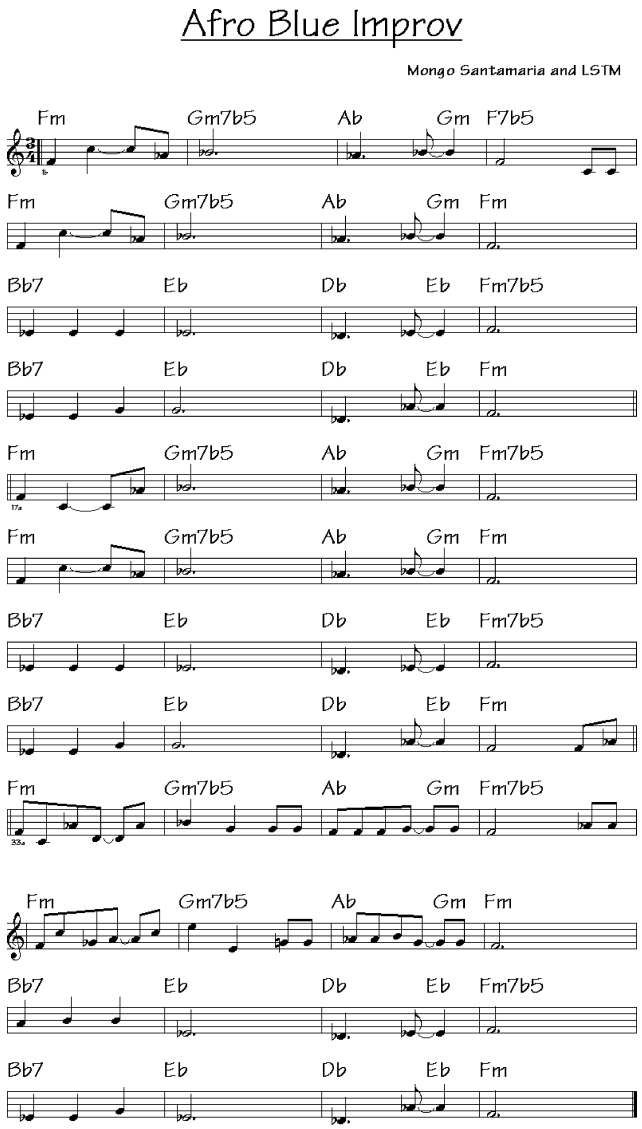

Figure 10 **LSTM's Improvisation Over Reharmonization of** ***Afro Blue***

common chord substitutions in jazz (Sabatella 1996, Berg 1990) and are numerous enough that they should affect the melody quite a bit. In fact, if we examine Figure 10, and compare it to Figure 9, we can see that the melody is very similar to the original with some changes. This is a common improvisation technique. In bars 9 and 25, over the first B-flat dominant 7 of those used instead of the E-flat major, the melody is changed to Eb, Eb, and Eb in both places. In the initial AABBAABB part of the form, the substitution of G half-diminished for G-minor does not change the melody. However, the melody does change over the CC part. A flat 9 (the G-flat note) is even "thrown in" over the F-minor chord starting the second C. And it is changed to a natural G in the next bar, over the Gm7b5 (i.e., the substituted H half-diminished). In the final AA part of the form, the first bar has been mod-

ified, but then the melody changes back to and stays as the original melody for the rest of the song.

While we are not pretending that the LSTM is suddenly understanding improvisation at a deep level here, it is able to produce something passable and interesting, just by knowing one song exactly, and by coping with several chord substitutions. We plan to develop this improvisational skill further, e.g., training the network by evaluating what it does with such chord substitutions, and adjusting its response. This will lead us to return to our work in reinforcement learning.

5. Conclusions

We presented several short musical tasks to LSTM networks and compared representations. Using our circles-of-thirds and modular-durations representations and the learning rates and configuration we found to be the most successful from these experiments, we trained dual pitch/duration LSTM networks to reproduce two versions of the song *Afro Blue*. One was human-rendered and one was the original score. While we have found a task that challenges a single LSTM network, we do not believe that any other recurrent networks we have used would be able to learn these songs. Having a dedicated network with one output iteration corresponding to the duration of an event makes it easier for the network to produce accurate sequences of event times. We contrast this with a network that produces one time unit's worth of output per iteration, and so must produce the same output over possibly many iterations to produce the duration of a single event. Another reason for the success of the network is its ability to learn relationships over long time spans. We have found the LSTM network to be a stable way to learn melodies in two ways; once it has learned, it retains that knowledge, and secondly, its learning process exhibits a low amount of oscillatory behavior, unlike the behavior of other recurrent neural networks.

We are looking forward to experimenting with a hierarchy or network of LSTM networks, that could learn the sub-phrase structures of a song. Our long-term goal is to return to experimentation with reinforcement learning for jazz improvisation with the improved representations and more able recurrent networks. One of our ideas is to use LSTM-based networks to learn to improvise in particular ways or over particular sets of chord sequences, and then use another network as a scheduler for these more local improvisors. Another idea is to have a reinforcement learning algorithm learn which networks to pass chords to and, perhaps, how to modify chords or learn to make chord substitutions to obtain desired effects.

Acknowledgments

This material is based on work supported by the National Science Foundation under Grants CDA-9720508 and IIS-0222541 and by Smith College. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. The author acknowledges the excellence of the John Payne Music Center, Brookline, MA for human jazz learning.

References

Allen, P., R. Dannenberg. 1990. Tracking musical beats in real time. *Internat. Comput. Music Conf.*, International Computer Music Association, San Francisco, CA, 140–143.

Berg, S. 1990. *Jazz Improvisation: The Goal-Note Method*, 2nd ed. Kendor Music, Inc., Delevan, NY.

Campolucci, P. 1998. A circuit theory approach to recurrent neural network architectures and learning methods. Ph.D. thesis, Dipartimento di Ingegneria Elettrica, Universitá Degli Studi di Bologna, Bologna, Italy.

Dannenberg, R., B. Thom, D. Watson. 1997. A machine learning approach to musical style recognition. *Proc. Internat. Comput. Music Conf.* ICMA, San Francisco, CA, 344–347.

Desain, P., H. Honing, K. de Rijk. 1989. A connectionist quantizer. *Proc. 1989 Internat. Comput. Music Conf.* ICMA, San Francisco, CA, 80–85.

Eck, D., J. Schmidhuber. 2002. Learning the long-term structure of the blues. *Proc. 2002 Internat. Conf. Artificial Neural Networks* (*ICANN*), 284–289.

Franklin, J. A. 2000. Multi-phase learning for jazz improvisation and interaction. *Proc. Eighth Biennial Connecticut College Sympos. Arts and Tech.* Connecticut College, New London, CT, 51–60.

Franklin, J. A. 2002. Learning and improvisation. T. G. Dietterich, S. Becker, Z. Ghahramani, eds. *Neural Information Processing Systems 14*. MIT Press, Cambridge, MA, 1377–1384.

Franklin, J. A. 2004a. Recurrent neural networks and pitch representations for music tasks. *Proc. 2004 Florida AI Research Sympos.* (*FLAIRS04*) *Special Track on AI and Music*. AAAI Press, Cambridge, MA, 33–37.

Franklin, J. A. 2004b. Predicting reinforcement of pitch sequences via LSTM and TD. *Proc. 2004 Internat. Comput. Music Conf.* ICMA, San Francisco, CA, 130–136.

Gers, F. A., J. Schmidhuber, F. Cummins. 2000. Learning to forget: Continual prediction with LSTM. *Neural Comput.* **12**(10) 2451–2471.

Griffith, N., P. Todd. 1999. *Musical Networks: Parallel Distributed Perception and Performance*. MIT Press, Cambridge, MA.

Hochreiter, S., Y. Bengio, P. Frasconi, J. Schmidhuber. 2001. Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Networks*. IEEE Press, New York.

Hochreiter, S., J. Schmidhuber. 1997. Long short-term memory. *Neural Comput.* **9**(8) 1735–1780.

Jordan, M. 1986. Attractor dynamics and parallelism in a connectionist sequential machine. *Proc. Eighth Annual Conf. Cognitive Sci. Soc.*, Amherst, MA, 531–546.

Kolen, J. F., S. C. Kremer. 2001. *A Field Guide to Dynamical Recurrent Networks*. IEEE Press, New York.

Laden, B., D. H. Keefe. 1991. The representation of pitch in a neural net model of chord classification. P. M. Todd, E. D. Loy, eds. *Music and Connectionism*. MIT Press, Cambridge, MA.

Large, E. W., J. F. Kolen. 1994. Resonance and the perception of musical meter. *Connection Sci.* **6**(2 & 3) 177–208.

Mandic, D. P., J. A. Chambers. 2001. *Recurrent Neural Networks for Prediction*. Wiley, West Sussex, UK.

Messick, P. 1988. *Maximum MIDI*. Manning Publications, Greenwich, CT.

Mozer, M. C. 1994. Neural network music composition by prediction: Exploring the benefits of psychophysical constraints and multiscale processing. *Connection Sci.* **6** 247–280.

PG Music. Band-in-a-Box. http://www.pgmusic.com/bandbox.htm.

Rumelhart, D., G. Hinton, R. Williams. 1986. Learning internal representations by error propagation. D. E. Rumelhart, J. L. McClelland, eds. *Parallel Distributed Processing 1*. MIT Press, Cambridge, MA, 318–362.

Sabatella, M. 1996. *A Whole Approach to Jazz Improvisation*. ADG Productions, Lawndale, CA.

Shepard, R. N. 1987. Toward a universal law of generalization for psychological science. *Science* **237** 1317–1323.

Steedman, M. 1994. The well-tempered computer. *Philos. Trans. Roy. Soc. Ser. A* **149** 115–131.

Thompson, T. 2003. Keykit programming language and graphical environment for MIDI. http://nosuch.com/tjt/.

Todd, P. M. 1991. A connectionist approach to algorithmic composition. P. M. Todd, E. D. Loy, eds. *Music and Connectionism*. MIT Press, Cambridge, MA.

Todd, P. M., E. D. Loy. 1991. *Music and Connectionism*. MIT Press, Cambridge, MA.

Williams, R. J., J. Peng. 1990. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Comput.* **2**(4) 490–501.

Williams, R. J., D. Zipser. 1988. A learning algorithm for continually running fully recurrent networks. Technical Report ICS-8805, Institute for Cognitive Science, University of California, San Diego, CA.