Peter M. Todd
Department of Psychology
Stanford University
Stanford, California 94305 USA
todd@psych.stanford.edu

A Connectionist Approach To Algorithmic Composition

With the advent of von Neumann-style computers, widespread exploration of new methods of music composition became possible. For the first time, complex sequences of carefully specified symbolic operations could be performed in a rapid fashion. Composers could develop algorithms embodying the compositional rules they were interested in and then use a computer to carry out these algorithms. In this way, composers could soon tell whether the results of their rules held artistic merit. This approach to algorithmic composition, based on the wedding between von Neumann computing machinery and rule-based software systems, has been prevalent for the past thirty years.

The arrival of a new paradigm for computing has made a different approach to algorithmic composition possible. This new computing paradigm is called *parallel distributed processing* (PDP), also known as *connectionism*. Computation is performed by a collection of several simple processing units connected in a network and acting in cooperation (Rumelhart and McClelland 1986). This is in stark contrast to the single powerful central processor used in the von Neumann architecture. One of the major features of the PDP approach is that it replaces strict rule-following behavior with regularity-learning and generalization (Dolson 1989). This fundamental shift allows the development of new algorithmic composition methods that rely on learning the structure of existing musical examples and generalizing from these learned structures to compose new pieces. These methods contrast greatly with the majority of older schemes that simply follow a previously assembled set of compositional rules, resulting in brittle systems typically unable to appropriately handle unexpected musical situations.

Computer Music Journal, Vol. 13, No. 4, Winter 1989,

To be sure, other algorithmic composition methods in the past have been based on abstracting certain features from musical examples and using these to create new compositions. Techniques such as Markov modeling with transition probability analysis (Jones 1981), Mathews' melody interpolation method (Mathews and Rosler 1968), and Cope's EMI system (Cope 1987) can all be placed in this category. However, the PDP computational paradigm provides a single powerful unifying approach within which to formulate a variety of algorithmic composition methods of this type. These new learning methods combine many of the features of the techniques listed above and add a variety of new capabilities. Perhaps most importantly, though, they yield different and interesting musical results.

This paper presents a particular type of PDP network for music composition applications. Various issues are discussed in designing the network, choosing the music representation used, training the network, and using it for composition. Comparisons are made to previous methods of algorithmic composition, and examples of the network's output are presented. This paper is intended to provide an indication of the power and range of PDP methods for algorithmic composition and to encourage others to begin exploring this new approach. Hence, rather than merely presenting a reduced compositional technique, alternative approaches and tangential ideas are included throughout as points of departure for further efforts.

A Network for Learning Musical Structure

Our new approach to algorithmic composition is first to create a network that can learn certain aspects of musical structure, second to give the network a selection of musical examples from which to learn those structural aspects, and third to let the network use what it has learned to construct

new pieces of music. We can satisfy the first step by designing a network that can exactly reproduce a given set of musical examples, because being able to reproduce the examples requires that the network has learned a great deal about their structure.

A network design that meets this music learning goal has been described in a previous paper by this author (Todd 1988). This network has been applied to both the task of algorithmic composition and the psychological modeling of certain aspects of human musical performance, such as tonal expectation (Bharucha and Todd 1989). This design is presented here. As in the original paper, I will restrict the musical domain to the relatively simple class of monophonic melodies. This restriction simplifies the nature of the network by avoiding certain problems associated with the representation of polyphony, which will be indicated later. However, the monophonic domain remains musically realistic and interesting, as the examples will show.

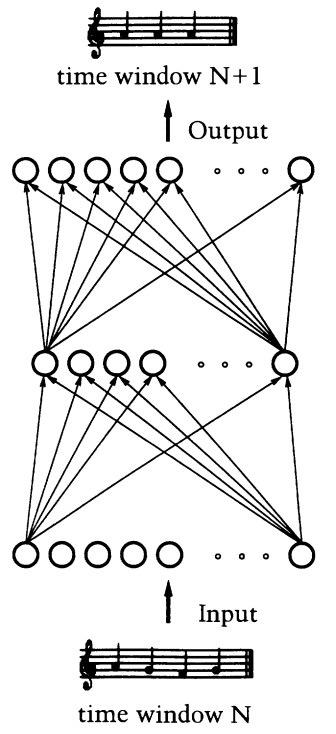


Fig. 1. A network design which can learn to associate time windows (e.g. measures) in a piece of music with the following time windows. Here, one measure as input produces the following measure as output. Circles represent individual units, lines represent directed connections between units, and arrows indicate the flow of activation through the network. Not all units or connections are shown.

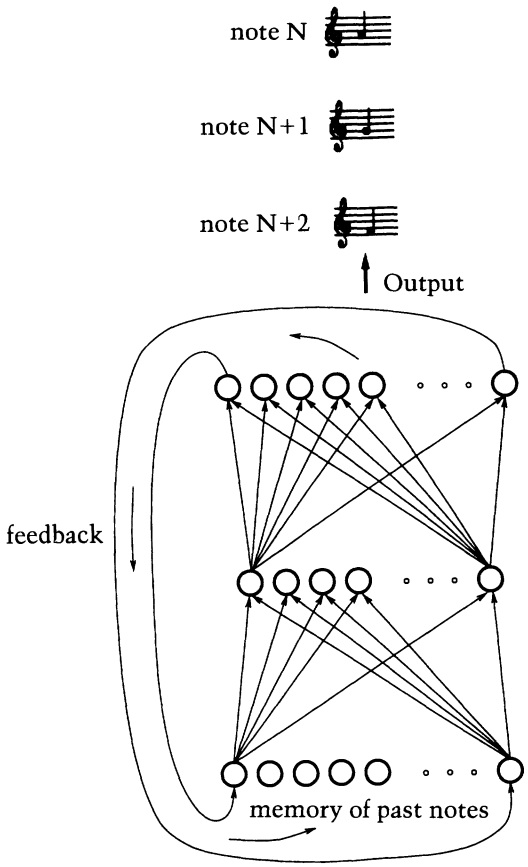
Network Design

Since music is fundamentally a temporal process, the first consideration in designing a network to learn melodies is how to represent time. One way time may be represented is by standard musical notation translated into an ordered spatial dimension. Thus, the common staff represents time flowing from left to right, marked off at regular intervals by measure bars. Music could be represented in a similar fashion in a PDP network, with a large chunk of time being processed simultaneously, in parallel, with different locations in time captured by different positions of processing units in the network. In the limiting case, the entire melody could be presented to the network simultaneously; alternatively, and requiring fewer input units, a sliding window of successive time-periods of fixed size could be used. This windowed approach is common in speech applications of various types, as in the NetTalk word-to-speech network (Sejnowski and Rosenberg 1987) and various phoneme recognition systems (Waibel et al. 1987).

In essence, the time-as-spatial-position representation converts the problem of learning music into the problem of learning spatial patterns. For example, learning a melody may consist of learning to associate each measure of the melody with the next one, as illustrated in Fig. 1. Thus when a particular measure is presented as the input to the network, the following measure will be produced as output. Learning to perform such pattern association is something at which PDP networks are quite good. Furthermore, networks are able to generalize to new patterns they have not previously learned, producing reasonable output in those cases as well. Thus, a new measure of music could be given as the input to a trained network, and it would produce as output its best guess at what would be a reasonable following measure. This generalizing behavior is the primary motivation for using PDP networks in a compositional context, since what we are interested in is exactly the generation of reasonable musical patterns in new situations.

While the spatial-position representation of time may be acceptable, it seems more intuitive to treat music as a sequential phenomenon, with notes

Fig. 2. A sequential network design which can learn to produce a sequence of notes, using a memory of the notes already produced. This memory is provided by the feedback connections shown, which channel produced notes back into the network.



being produced one after another in succession. This view calls for the use of a sequential network, which learns to produce a sequence of single notes rather than a set of notes simultaneously. In this case, time is represented by the relative position of a note in the sequence, rather than the spatial position of a note in a window of units. Where networks utilizing a spatial representation of time learn to associate a successive chunk of time with the previous chunk, sequential networks learn to produce the next note in a sequence based on some memory of past notes in the sequence. Thus, some memory of the past is needed in a sequential network, and this is provided by some sort of feedback connections that cycle current network activity back into the network for later use, as can be seen in Fig. 2.

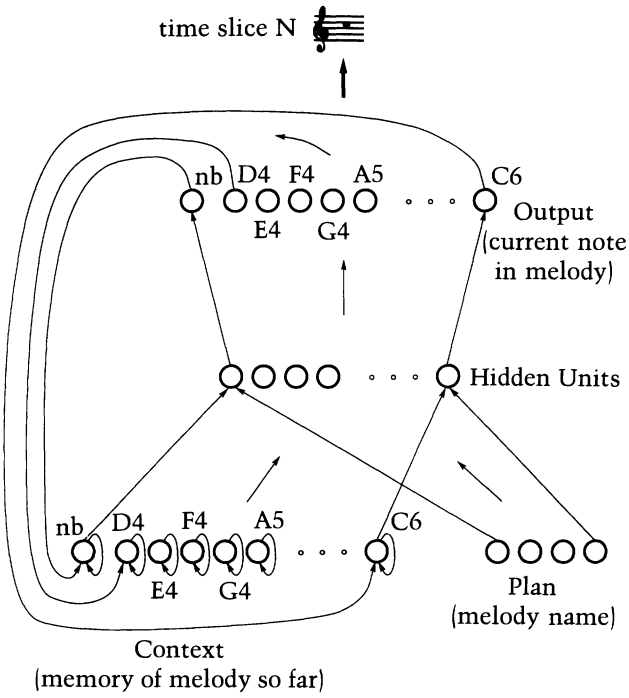
The learning phases of these two types of networks are very similar—both learn to associate certain output patterns with certain inputs by adjusting the weights on connections in the network. But their operation during production of melodies is quite different. Basically, the windowed-time pattern associator network produces a static output given its input: one window of time in produces one window of time out. The sequential network, on the other hand, cycles repeatedly to yield a sequence of successively produced outputs. Each of these outputs further influences the production of later outputs in the sequence via the network's feedback connections and its generalizing ability. This ongoing dynamic behavior has great implications for the sorts of sequences the network will produce, as will be seen later in this article.

Actually, the windowed-time and sequential-time approaches are not contradictory and may be combined to advantage. A sequential network that produces a sequence of time windows, rather than merely single notes, would learn a different set of associations and so make different generalizations during the composition phase. For the current discussion, though, a standard, single-event output sequential network design of the type first proposed by Jordan (1986a) has been used. A network of this type can learn to reproduce several monophonic melodies, thus capturing the important structural characteristics of a collection of pieces simultaneously. This makes it an ideal candidate for our purposes.

Jordan's sequential network design is essentially a typical, three-layer, feedforward network (Dolson 1989) with some modifications mostly in the first (input) layer, as shown in Fig. 3. One set of units in the first layer, called the *plan units*, indicate which sequence (of several possibilities) is being learned or produced. The units do this by having a fixed set of activations—the *plan*—turned on for the duration of the sequence. In effect the plan tells the network what to do by designating or naming the particular sequence being learned or produced.

The *context units* (also called *state units*) make up the remainder of the first layer. These units are so named because they maintain the memory of the sequence produced so far, which is the current con-

Fig. 3. The sequential network design used for compositional purposes in this paper. The current musical representation requires note-begin (nb) and pitch (D4-C6) units, as shown for both output and context; context units also have self-feedback connections. Each network output indicates the pitch at a certain time slice in the melody.



text or state that the network uses to produce the next element in the sequence. Each successive output of the network is entered into this memory by the feedback connections indicated from the output units to the context units.

A memory of more than just the single previous output is kept by having a self-feedback connection on each individual context unit, as shown in Fig. 3. These connections have a strength (weight) of less than 1.0, so that each context unit computes an exponentially decreasing sum of all of its previous inputs, which are the network's outputs. For example, if the self-feedback strength were 0.8, then a unit's memory would decrease proportionally by the amounts 0.8, 0.64, 0.51, 0.41, etc., as long as nothing new were entered into its memory. This connection strength cannot be greater than 1.0 or the activation values of the context units would explode exponentially.

The context units and plan units are all fully interconnected by a set of learned, weighted connections to the next layer of units, the *hidden units.* The hidden units are so named because they are neither at the network's input nor output, and so are in some sense buried inside the network. The hidden units combine the weighted information from the (fixed) plan units and the (evolving) context units, processing it via their logistic activation functions (Dolson 1989). They then pass on this processed information through the final set of weights to the output units. The output units then determine what the network will produce as the next element in the sequence. Each successive output is also finally passed along the feedback connections back to the context units, where they are added into the changing context. This in turn enables the computation of the following element in the sequence, and the cycle repeats.

The actual number of the various types of units used in the network depends on several factors. The number of plan units must be sufficient to specify different plans for all the different sequences to be learned. For example, we might want to use plans that have only one plan unit on at a time (i.e., with an activation of 1.0), while all the rest of the plan units are off (i.e., they have activations of 0.0). The particular plan unit that is on, for example the third or the fifth, specifies the sequence being processed (i.e., sequence number 3 or number 5). This type of plan is known as a *localist* representation, because each unit represents an entire entity (here an entire sequence) locally, by itself. If we wanted to learn N sequences for example, we would need N plan units to specify all of them in this way. On the other hand, a binary-coded plan representation would be more compact: in this case, we would need only $\log_2 N$ plan units to create N different plans. Thus plan 011 would specify sequence number 4 out of 8 possible, starting with 000. This is a *distributed* type of representation, because each entity is represented by a pattern of activation spread over several units at once.

The number of output units in the network depends on the representation of the sequence elements used, so it cannot be specified until this representation is settled. The number of context units depends on the type of memory desired. We will see below that having an equal number of output units and context units is useful. Finally, the number of hidden units depends on what the network must learn and cannot be exactly specified. If

there are too few hidden units, the network may lack the computational power to learn all it is intended to. It is a good idea, however, to keep the number of hidden units as small as possible, because this tends to enhance the network's ability to generalize to new situations (new inputs), which is usually important. This happens because the network is forced to squeeze its processing through a narrow channel of few hidden units, so that it must make use of only the most important aspects of the information it is processing. If we have too many hidden units, the network can use them essentially to memorize what it needs to do rather than extracting the important aspects that allow generalization from its training data. But at this point, finding the ideal number of hidden units to use is still a matter of trial and error.

What happens when the sequences to be learned and produced are melodies? In this case, each element of the sequence, and hence each output of the network, is a particular note, with at least some indication of pitch and duration. The plan units now represent the name of the particular melody being processed. Correspondingly, the context units store a representation of the sequence of notes produced so far for this melody. The output units code the current note of the melody. The hidden units have no special interpretation in this case other than to help compute each successive note output of the network.

In the musical sequential networks investigated so far, the relationships of the feedback connections from the output units to the context units have all been one-to-one. That is, for each output unit, there is a single corresponding context unit to which it is connected and which thus maintains an easily interpretable decaying memory of the output unit's recent activity. The weights on all these feedback connections are fixed at 1.0, so that they merely copy the activity of each output unit back to its particular context unit. The self-feedback connections on the context units, which allow the decaying memory of the output copies, have also had fixed weights. These weights, which are the same for all the context units, are usually set between 0.6 and 0.8. Values in this range give a relatively slow memory decay, so that the previous several sequence steps are available to help decide what the next output should be.

The self-feedback weight should not be too large, however, because the built-up memory values would then not change rapidly enough to differentiate the context among successive sequence steps. This differentiation is essential, since only the context changes from one step to the next in the network's input (remembering that the plan units are fixed). Also, the context units typically use a linear activation function, rather than a sigmoidal one. By not using the sigmoidal "squashing" activation function, the context units have increased dynamic range: i.e., their outputs can be anything instead of being restricted to the range from 0.0 to 1.0. This, too, helps to differentiate one context state from another. So the self-feedback weight is chosen to balance the desirability of a long memory trace against the disadvantage of having all the contexts blur together.

All of these feedback connections could be trained during learning rather than remaining fixed, but there are some reasons not to train them. Most importantly, it has not yet proved necessary. The networks investigated so far have learned sufficiently well without these extra degrees of freedom. The training process is also sped up by not including these additional weights that would need to be adjusted during learning. Finally, the fact that these weights are all the same and all unchanging lets us know exactly what the information at the context units will look like. They each compute the same type of decaying memory trace of their corresponding output unit. Knowing this can help us interpret just what the network is doing.

The issue of interpretation is also the main reason why another sequential network structure has not been used for these studies. The design described by Elman (1988) uses feedback connections from the hidden units, rather than the output units, to the context units. Since the hidden units typically compute some complicated, often uninterpretable function of their inputs, the memory kept in the context units will likely also be uninterpretable. This is in contrast to Jordan's design, where, as described earlier, each context unit keeps a memory of its corresponding output unit, which *is* inter-

pretable. [In general, interpretability seems like a good thing, although it is often not necessary. Certainly the Elman type of sequential network could be used for musical purposes; its behavior would simply be less analyzable. The extent to which this matters depends on just how deeply we wish to understand what is going on in the network.]

Melody Representation

Before the structure of the network can be finalized, we must decide upon the representation to use for the melodies to be learned and produced. This representation will determine the number and interpretation of the output units and of the corresponding context units. For the purposes of the current investigation, we will make several simplifying assumptions about the melodic sequences to be used, which affect the representation needed. One of these assumptions is that each monophonic melody is to be represented only as a sequence of notes, each with a certain pitch and duration. All other factors, such as tempo, loudness, and timbre, are ignored. Thus our representation need only capture two types of information: the pitch and duration of each note.

Pitch

To represent the pitch of a given note, two possibilities come to mind. Either the actual value of each pitch can be specified, or the relative transitions (intervals) between successive pitches can be used. In the first case, we would need output units corresponding to each of the possible pitches the network could produce, such as one unit for middle C, one unit for C♯, etc. The context units would then hold memories of these pitches. In the interval case, on the other hand, there would be output units corresponding to pitch changes of various sizes. So one output unit would designate a pitch change of +1 half step, another a change of −3 half steps, etc. To represent the melody A-B-C, then, the output from the actual-pitch network would be {A, B, C}, while the output from the pitch-interval network would be {A, +2, +1} (where the first pitch must be specified so we know from what value the intervals start).

The pitch-interval representation is appealing for several reasons. First, given a fixed number of output units, this representation is not restricted in the pitch range it can cover. The actual-pitch representation is so restricted. For instance, if we only had three output units, we could only represent three actual pitches. But if these output units corresponded to the pitch intervals −1, 0, and +1 half steps, we could cover any range of pitches in a melody by repeatedly going up or down, one half step at a time, at each successive note output. In this case the choice of interval sizes is necessarily limited to the number of output units.

Another advantage of the pitch-interval representation is that melodies are encoded in a more-or-less key-independent fashion. That is, aside from the initial specification of the starting actual pitch value, the network's output contains no indication of the key it is to be performed in (except, perhaps, for indications of key mode, major or minor, based on the intervals used). This key-independent learning is useful for letting the network discover common structure between melodies. If the network is to learn two different melodies in two different keys, then using an actual-pitch representation might obscure patterns of pitch movement present in both. For example, if one melody included the phrase C-G-E-F, and the other included the phrase F♯-C♯-A♯-B, the network would be likely to miss any connection between the two, since they use totally different output units. In a pitch-interval representation, though, these two phrases would be coded identically, allowing the network to see this commonality in the two melodies.

Key independence also allows transposition of an entire melody simply by changing the initial actual pitch (which need not even be produced by the network, but could be specified elsewhere). All of the intervals would remain unchanged, but the different starting pitch would put the rest of the sequence into a different key. In contrast, to transpose a melody in actual-pitch form, every single pitch output would need to be changed—that is, the network would have to be retrained. The ability to transpose

the network's output easily may be desirable in some situations.

The ease of transposition that the pitch-interval representation allows is also a major drawback. When an error is made in the production of a sequence of intervals, the rest of the melody will be transposed to a different key, relative to the segment before the error. This type of mistake is glaringly obvious when one hears it—as though someone were singing a melody in one key, and then suddenly switched to another. One error in the interval sequence will thus globally affect the performance of the whole melody. In contrast, mistakes in the output using the actual-pitch representation are purely local. Only the pitch of the individual wrong note will be altered; all the rest of the pitches in the sequence will retain their proper values, so that performance of the melody will be minimally affected.

Since the networks I first looked at were inclined to make errors, I chose the most error-free representation, using actual pitches. The output units thus each represent a single pitch, in a particular octave, as indicated in Fig. 3. Another simplifying assumption about the melodies to be learned now comes into play; all the melodies in the current set of examples have been transposed beforehand into the key of C and have no accidentals. In this way, we only need units representing the pitches in the key of C over the octave range required. By eliminating the need for extra output units for sharps and flats, we have made the network's learning task easier, at the cost of some restriction and preprocessing of the melodies to be learned.

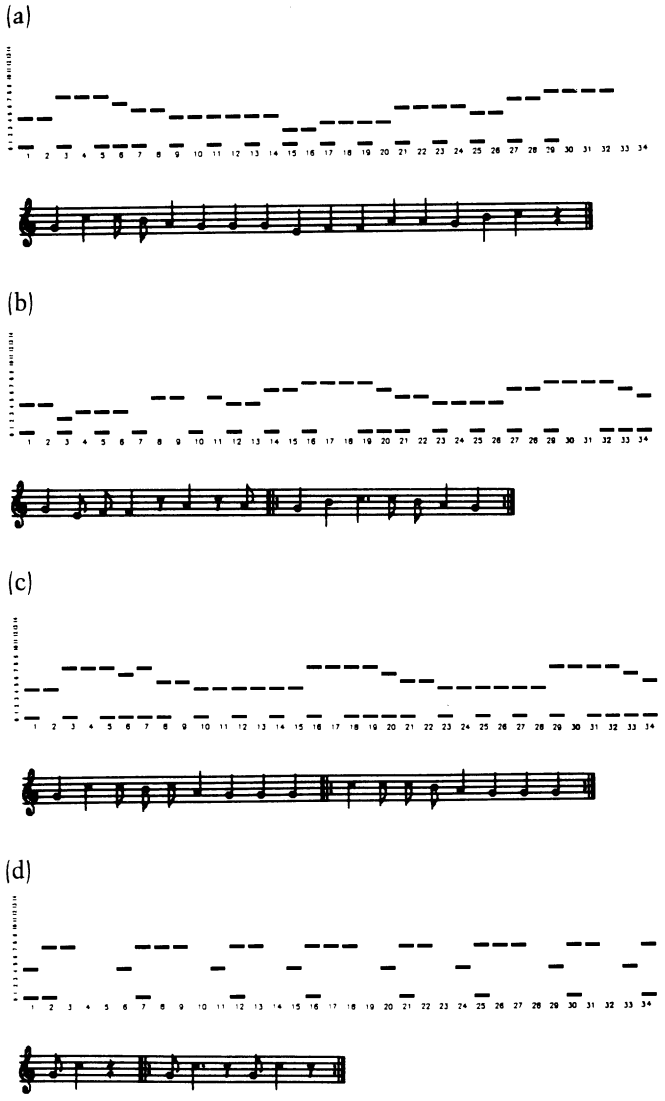
Note also that the output units use a localist representation of the actual pitch. As described above, a localist representation is one in which a single unit denotes an entire entity. In this case, only one unit is on at a time, and each unit itself designates a particular pitch: for example (with three output units), 001 for A, 010 for B, 100 for C. Rests are represented in this scheme by having none of the output pitch units on—000 in this example. This is opposed to a distributed representation—for example, a binary coding scheme such as 100 for A, 110 for C, 111 for D. The localist scheme has the advantage that each pitch representation is equally similar to every other one, in the sense that every pattern overlaps equally with the others (001 and 010 are the same in just one position, as are 001 and 100, and 010 and 100). In the binary-coded case, though, the representation of A is more similar to that of C than of D—100 and 110 are the same in two positions, while 100 and 111 are the same in only one position.

This difference would have an effect while training the network. For example, using the values just given, if a C is produced as output instead of an A, this would be a lesser mistake (since they are more similar) than producing a D for an A. As it learned, the network's knowledge of musical structure would begin to reflect this (probably) erroneous difference. Thus this distributed coding imposes a similarity-measure on the network's outputs that we probably do not want—there is no *a priori* reason to designate A and C as more similar than A and D. The localist pitch representation, which does not impose this differential similarity on the outputs, works better.

In addition, by using a localist pitch representation on the output units, the context units now become individual pitch memories. Since each context unit is connected to a single output unit, it maintains a memory only concerned with that output unit's own pitch. In particular, the context unit's activation level tells how recently and how often or long that pitch has been used in the current melody so far. This memory of pitch use seems like an appropriate thing to make use of in producing the subsequent notes of the melody.

It still seems unfortunate to have to abandon the pitch-interval representation completely. Its advantages could be gained and its drawbacks minimized by incorporating it into a hybrid pitch representation. For instance, intervals could be used for the majority of the network outputs, with the addition of calibrating actual pitches every so often. That is, at the start of every measure (or other suitable period), the actual pitch that the melody should be playing could be included in the network's output. This could be used to recalibrate the melody periodically, so that it remains in the proper key. A hybrid representation of this form could minimize the jarring effect of key changes after errors while retaining the advantages of intervals, such as the

Fig. 4. Network output using extrapolation from a single melody. In each case, both piano-roll-style output and common-practice music notation are shown. Network outputs for the first 34 time-slices are shown, with row 0 (bottom row) corresponding to the note-begin unit, and rows 1–14 corresponding to the pitch units, D4-C6. A black bar indicates the unit is on. Where the network output goes into a fixed loop, this is indicated by repeat bars in the music notation. (a) Melody 1, which the network is originally trained to produce with a plan of 1.0. (b) Extrapolation output using a plan of 0.0. (c) Extrapolation output using a plan of 2.0. (d) Extrapolation output using a plan of 3.0.



similarity of common pitch movement patterns in different keys.

Duration

The duration of notes in the melodic sequences must also be represented. As with the pitch representation, two clear alternatives present themselves. First, the duration could be specified in a separate pool of output (and context) units, alongside the pitch output units. The units could code for note duration in a localist fashion, with one unit designating a quarter-note, another a dotted eighth-note, etc. Or they could use a distributed representation, with for instance the number of units "on" (activation 1.0) representing the duration of the current note in sixteenth-notes. With the localist representation, the corresponding context units would hold a memory of the lengths of notes played recently in the melody; in the distributed case, the context units would be harder to analyze.

Alternatively, duration can be removed from explicit representation at the output units. Instead, the melody could be divided into equally spaced time slices of some fixed length, and each output in the sequence would correspond to the pitch during one time slice. Duration would then be captured by the number of successive outputs and hence the number of time slices a particular pitch stays on. This is equivalent to thinking of a melody as a function of pitch versus time (as in piano-roll notation), with the network giving the pitch value of this function at equally spaced intervals of time. I am using this time-slice representation for duration at present, in part because it simplifies the network's output—no separate note-duration units are needed. In addition, this representation allows the context units to capture potentially useful pitch-length information, as will be indicated below. The form of this representation can be seen in the example network output in Figs. 4–6.

The specific fixed length of the time slices to use should be the greatest common factor of the durations of all the notes in the melodies to be learned. This ensures that the duration of every note will be represented properly with a whole number of time slices. For example, if our network were only to learn the melody A-B-C with corresponding durations quarter-note, eighth-note, and dotted quarter-note, we would use time slices of eighth-note duration. The sequence the network would learn would then be {A, A, B, C, C, C}.

With this duration representation, the context units now not only capture what pitches were used recently in the melody, but also for how long. This is because the longer a given note's duration is, the more time slices its pitch will appear at the output,

Fig. 5. Network output using interpolation between two melodies. (a) Melody 1, trained with plan 1.0. (b) Interpolation output using a plan of 0.8. (c) Interpolation output using a plan of 0.7. (d) Interpolation output using a plan of 0.5; an additional 34 successive time-slices (68 total) are shown to indicate longer-term behavior. (e) Interpolation output using a plan of 0.2. (f) Melody 2, trained with plan 0.0.

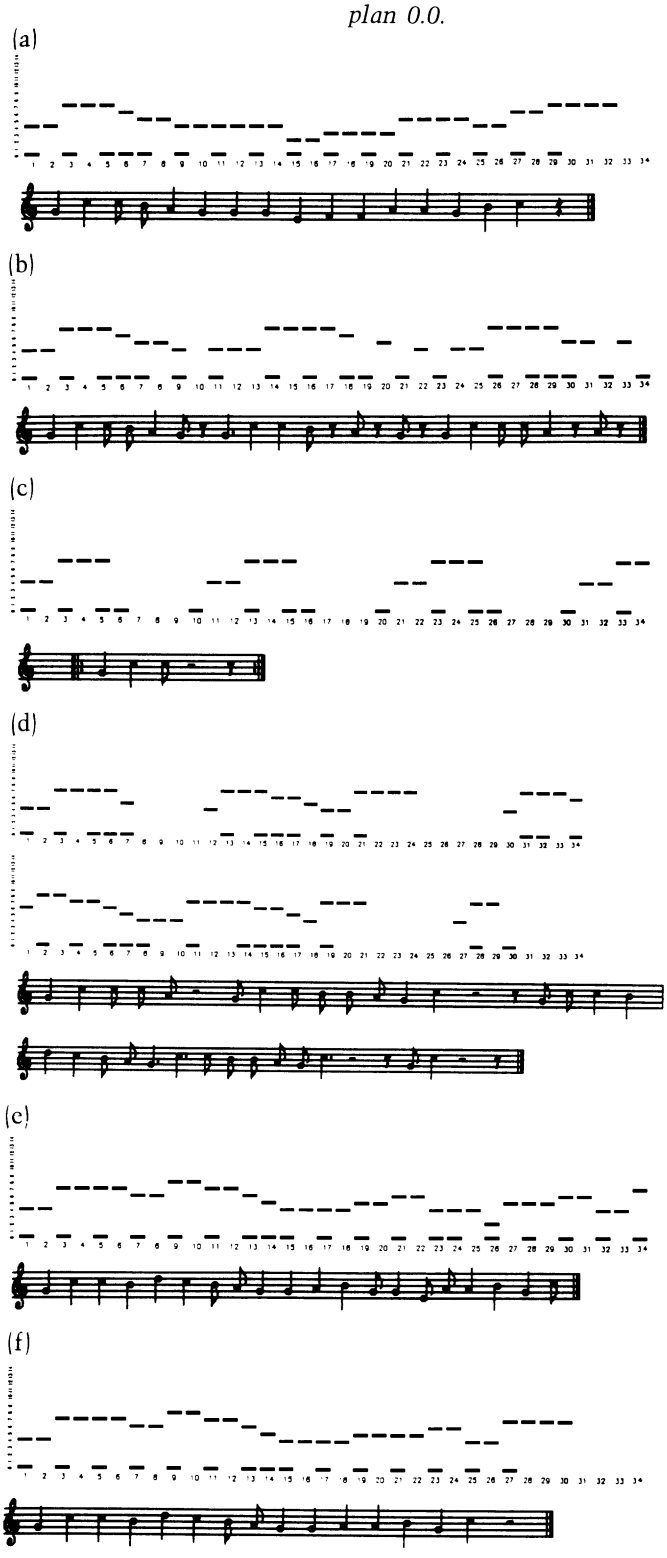
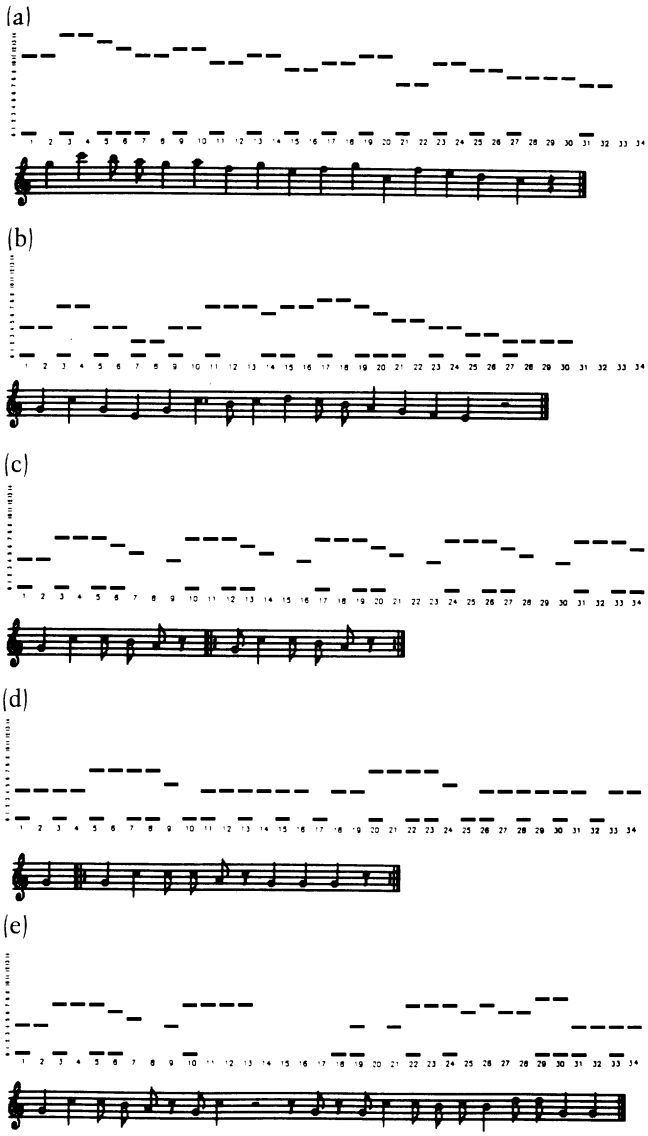


Fig. 6. Network output using altered melody space. (a) Melody 3, trained using plan vector (0.0, 1.0). (b) Melody 4, trained using plan vector (1.0, 1.0). (c) Interpolation output between melodies 1 and 2, incorporating training on 3 and 4, using plan vector (0.5, 0.0). (d) Interpolation output between melodies 1 and 2, trained with 8 hidden units, using a plan of 0.5. (e) Interpolation output between melodies 1 and 2, retrained with 15 hidden units, using a plan of 0.5.



and the greater the corresponding built-up context will be. Returning to the previous example, the context activation for pitch C will be greater than that for pitch B, since C was present for more time slices at the output. This combination of pitch and duration information seems like a useful type of context memory for melody production, which is another reason the current representation was chosen.

An additional piece of information must be included in the network's output when using time slices to represent duration. This is an indication of the time slices on which any notes begin. Without this indication, we would not be able to tell whether the network output {A, A} meant two notes of pitch A, each lasting one time slice, or one note of pitch A that is two time slices in duration. It is essential that our representation allows this distinction to be made, otherwise the network would be unable to deal with melodies with repeated notes of the same pitch.

For this reason, an additional note-begin marking unit is added to the output (and context), as shown in Fig. 3. This unit is on for those time slices in which a new note is begun and off when the time slice merely covers the continuation of a previously started note. The necessary inclusion of this note-begin unit makes the time-slice approach less clean than we might hope, but it still seems more succinct in some ways than having both pitch and duration units in the output. The tradeoff is that the number of output units is reduced when using time slices, but the number of steps in the sequence representation of any particular melody is increased. This is because there will usually be more time slices than notes, and for the duration-unit representation, only complete notes need to be produced at each sequence step.

The presence of the note-begin unit also complicates attempts at representing polyphonic sequences. In principle, it seems as though at each time slice, we should be able to turn on all the output pitch units that are present at the corresponding time in a polyphonic melody. Thus the pitches in chords and harmonic lines could all be represented simultaneously on the output units. The problem, though, is that for each pitch present, we need to be able to independently specify when the corresponding note began. This requires a separate note-begin marker for each pitch possible, which doubles the number of output units, or some other scheme to represent polyphonic onsets. This problem has not yet been solved reasonably, and so again all the examples that follow are monophonic.

Training and Running the Network

Once we have set up a sequential network with the proper number of plan, context, hidden, and output units, chosen a set of melodies for it to learn, and converted these into the appropriate representation, we are ready to train the network to produce these melodies. The steps to accomplish this could proceed as follows. First, the weights in the network would all be initialized to small random values for the beginning untrained state. Second, the particular plan for the first melody in the training set would be clamped onto the plan units (i.e., their activations would be set to the proper values). Next, the activations of the context units would be set to zero, so that we begin the sequence with a clean (empty) context. Then activity would be passed through the network, from the plan and context units (the zero-valued context units having no effect this first time), through the hidden units to the output units. There the final output values produced by the network would be compared to the desired target, namely the first time-slice pitch for the first melody, and the error between the two would be used to adjust the network weights by the method of back-propagation of error (Dolson 1989). Following this, the output values would be passed back along the feedback connections to be added into the current context, and then activity would be passed through the network again and once more compared to the next desired target.

We would repeat this process by cycling the outputs back to the context units, computing the next outputs and errors, and adjusting the weights accordingly for every time slice for the first melody. Then the context units would be set to zero again, the plan for the second melody clamped onto the plan units, and the whole process would be repeated

for the new melody. This would continue for all the melodies in the training set until the total summed error produced by the network for this set was below some threshold (i.e., until the network could produce these melodies more or less without mistake).

Note that the outputs of the network change over the course of training. At first, in the untrained state, the outputs produced will be quite different from the desired target values. As the network learns, however, the outputs will get closer and closer to the targets until they are sufficiently close and training can stop. This also means that the context values used by the network will change during training. The contexts computed for a given melody at the beginning of training will be very different from those computed at the end when the melody has been learned, because the contexts are being formed from the changing, fed-back output values. Thus the actual mapping task being learned by the network is not constant: its input values (at the context units) change during training, though its target values remain fixed. In essence the network is shooting at a moving target; as it adjusts its performance and hence its output and context values, what it is trying to learn changes as well. This complicates its learning task.

There is a way around this problem, however. Consider the network after it has successfully learned all the melodies in the training set. In this case, the outputs will all match the targets, and we could feed back the outputs or the targets to the context units with the same effect. This is in fact the actual final mapping we want the network to learn—from the correct outputs (the targets) via the context units to the next output. We may just as well set up this final mapping throughout the entire procedure as the one the network is to learn. This means that, instead of feeding back the output values during training, we should feed back the *target* values and construct the contexts using them. Now the mapping to be learned by the network will not change over the course of training (since the targets remain the same throughout), and the training phase will be much shorter.

One further adjustment can help the training process go more smoothly. Since we are using a localist representation of the monophonic melody pitches, only one output pitch unit will be on at a time, while all the rest are to be zero. In order to accommodate this trend, the network will first learn to turn off all the output units all the time, since this is quite close to the correct behavior and is easily achieved by lowering the biases to the outputs. If we let the network proceed this way initially, it could take a long time to correct this overzealous behavior. Therefore, we would like the impetus to turn the units off (when the target is zero) to be less strong than the impetus to turn them on (when the target is one). This way, the network is less likely to merely turn all the units off. We can accomplish this by using half the normal error whenever the target is zero: that is, in this case use $E = 0.5*(t - o)$, where E is the error at a particular output unit, t is that unit's target value, and o is its actual activation, instead of $E = (t - o)$ as usual (this error is then squared for further computational use). This is the strategy used for the simulations discussed here.

The length of the training process for a particular network and melody set, measured in the number of epochs (cycles through the entire training set), depends on the size of the network, the number of training melodies and their relatedness to each other, and (at least) two network training parameters—the learning rate and the momentum (Dolson 1989). Typically, low learning rates (on the order of 0.05) and high momenta (around 0.8) seem appropriate for the melody learning tasks investigated so far. To give approximate figures on the number of epochs needed in some different training situations, using these parameter values a network with 15 output units and 15 hidden units learned one short melody (34 time steps, as in the examples to come) in 2,000 epochs. To learn two such melodies, 5,500 epochs were required; for four, 8,500 epochs were needed. When fewer hidden units were used in the network—8 instead of 15—it took nearly 50,000 epochs to learn the same two melodies that took 5,500 epochs in the previous case, indicating the increased difficulty of squeezing the needed information into fewer weights. Such training times can translate into many hours of computational time, depending on the hardware used.

In contrast to the slow learning these networks often exhibit, their performance once trained is practically instantaneous. This is because no learning is being computed, and so cycling through a melody once to see how it comes out can go very quickly. During performance, we feed back the actual output values rather than the context values, so that if the network begins to produce different output values (due to our manipulations on it, as will be described shortly), these will affect the stored context and thus the outputs produced in the future. If the target values were fed back during performance, they would always tend to pull the network back toward producing the outputs it was trained to make.

When using the network to produce melodies, one more mechanism should be added to ensure proper behavior. Because we only want one output pitch unit on at a time with the present representation, we should process the final output values to make sure this is so, before the outputs are passed back to the context units or used elsewhere (such as to play the melody on a synthesizer). To accomplish this, we merely choose the output unit with the highest activation above 0.5 and set its new activation to 1.0, while all the others are set to 0.0. If no pitch units have activation above 0.5, this output is interpreted as a rest, and all activations are set to 0.0. By cleaning up the output activations in this way, we guarantee that the context values will accurately reflect just those pitches that were produced in the current sequence. Similarly, the note-begin unit's value is cleaned up, i.e., set to 0.0 or 1.0 depending on which side of 0.5 its original activation lies.

All of the simulations described hereafter were run using SunNet (Miyata 1987), a noncommercial back-propagation learning system implemented in C for use on Sun Microsystems computers. SunNet allows the writing of specialized network training and testing routines using its own internal programming language, giving great flexibility to the possible simulations. Most of the things described here could also be accomplished using *bp*, the back-propagation simulator available in McClelland and Rumelhart's (1988) *Explorations in Parallel Distributed Processing.*

Using the Network for Composition

Once we have trained a sequential network to produce one or more particular melodies, we can begin to use it to compose *new* melodies based on what it has learned. Because the plan vector is the major component of the network controlling which melody is produced, one way to produce new melodies is to give the network new plans. Depending on how many melodies the network was trained on, this will be done in different ways.

Unless otherwise stated, all of the network examples discussed in the following sections used 15 output units—1 note-begin unit and 14 pitch units—corresponding to the pitches in the key of C from D4 to C6, with no accidentals. They also used 15 context units, all with self-feedback connections of strength 0.7, and 15 hidden units. All of the original trained melodies used were 34 time slices in length, padded at the end with rests (as shown in the music notation). Each time slice corresponds to an eighth-note in duration, so that the original melodies are all approximately four measures long. For ease of discussion and interpretation, all the melodies produced by the networks in these examples are presented both in a modified piano-roll notation, with the pitch and the value of the note-begin unit at each time slice indicated individually, and in standard music notation.

Extrapolating from a Single Plan

If our network is trained on only one melody, then we have a single plan from which to extrapolate and choose new plans. For example, a network with a single plan unit was trained to produce melody 1 in Fig. 4a when given a plan of 1.0. Now we can instead give the network a different plan, by setting some other value onto its plan unit. If we let the network proceed with its normal performance, a new melody will be produced by first clearing the context units, then passing activation through the network, cleaning up and collecting the outputs, and cycling them back to the context units to repeat this process for a certain number of time slices.

As the examples in Figs. 4b–d show, a variety of behaviors are possible even in this simple situation. If we first use a plan of 0.0, we get the sequence shown in Fig. 4b, which quickly settles down into a repeating pattern that is 13 eighth-note time slices in length. This pattern, which first extends from time slice 12 to slice 24 and repeats thereafter, is constructed of chunks from various positions in the original melody—the first seven time slices match those at the end of melody 1 from slice 25 to 31, the next six match melody 1 from slice 5 to 10, and the interval from the final G to the beginning G matches the repeated Fs in melody 1 from slice 9 to 12. In essence, chunks of melody 1 that end with the same pitch that another chunk begins with have been spliced together to form the new melody. We will see this sort of behavior throughout the melodies composed by our networks, and the reason is relatively clear—when creating the new melody, the network continues to make the same sorts of transitions it did for the original trained melody (or melodies), occasionally switching to a different point in the original when the contexts (i.e., the pitches most recently played) more closely match that other point. This splicing-at-matching context results in new melodies that are smooth, mostly lacking sudden odd pitch intervals, but still interestingly different from the originals. At first glance, this is similar to behavior we would see in a Markov transition table process trained on this melody, but there are important differences, as will be described below.

The short melodic pattern in Fig. 4b can be seen to repeat indefinitely if we continue to cycle the network beyond the length of the original melody. Such repetition is common to the composed melodies in general. On an intuitive level, this can be explained by the fact that the network has spliced together the beginning and end of the phrase because this matches an interval (the repeated G quarter-notes) from the original melody 1. A more theoretical explanation for this behavior will be provided below.

Similar splicing and repetition can be seen in the sequence in Fig. 4c, produced with a plan of 2.0. In this case, the repeating phrase is also 13 time slices long and matches melody 1 from slice 3 to 14, with the addition of an extra eighth-note C as the second note. The interval from the end of the phrase (G) back to the beginning (C) matches the initial G-C interval in melody 1.

It is unclear where the additional eighth-note C in this melody arises exactly; the fact that there are four time slices worth of C matches the held C at the end of melody 1, but the extra eighth-note (caused by the note-begin unit coming on) is unprecedented. In fact, with most of these network-composed melodies, it is the rhythm (the pattern of note-begins) that is most inexplicable. This is probably because it is hardest for the network to learn when the note-begin unit should be on or off, as compared to which pitch unit should be on; the context is more useful for telling the next pitch than the next note-begin. Thus in the new melodies the pitch will often change without a corresponding note-begin, or a note-begin will be indicated during a rest (for example, see Fig. 4b, time slices 7 and 8). This shows that the network has not fully learned the relationships between pitch-changes or rests and note-begins. Perhaps with more training this relationship would be learned, or perhaps an alternate duration representation scheme, such as those described above, would solve this problem. For now, the melodies produced are interpreted as having note-begins whenever the network generates them and whenever the output changes pitch.

If we increase the plan value still further, to 3.0 for example, we get the interesting rhythmic pattern shown in Fig. 4d. Now we have the G-C interval from the beginning of melody 1 and the C-rest interval from the end of the melody. Because the context decays toward all zeros during the rest, and thus becomes closer to the initial all-zero context state, another G is generated. Interestingly, G-C-rest phrases of two different lengths are produced in this way, in alternation. The reason for this is again unclear; not everything produced by this composition method is predictable, but therein lies much of the usefulness and interest of this approach.

The network is not merely creating a transition-probability matrix and implementing a Markov process. First, unlike a Markov process, the network's behavior is deterministic: given a certain plan, the network will always produce the same sequence as

output. Secondly, though the network's next output is dependent on past outputs, like a Markov process, in this case the network uses a memory of its entire past (in the decaying context), rather than just one or two previous steps (that is, the network's state is essentially infinite). Thirdly, and perhaps most importantly, the network can generalize to produce reasonable ouputs for new contexts it has never encountered before. The behavior of a Markov process is undefined for states that are not in its transition matrix. The network, though, will do similar things for similar contexts, even when given a new context. This lets it produce new melodies quite different from those it was originally trained on. Finally, using different plans does not correspond to anything in the Markov process case, except perhaps for switching to whole new transition matrices. Thus we must conclude that the network is doing something else, a topic I will address shortly.

Interpolating Between Multiple Plans

If we train our original network with more than one melody, each having a different plan, then we can generate new melodies by specifying new plans interpolated between the trained ones. As expected, these interpolated melodies share features of the parent melodies between which they are interpolated, more or less in proportion to how similar their plans are to those of the originals.

The melodies in Fig. 5 show this effect. A network with a single plan unit was trained to produce melody 1 with a plan of 1.0 (Fig. 5a), while melody 2 was trained with a plan of 0.0 (Fig. 5f). With a plan of 0.8, the melody shown in Fig. 5b was produced, matching melody 1 primarily up to time slice 14, at which point it makes a transition back to melody 1's beginning; the same thing happens again at slice 26. Little influence of melody 2 can be seen. With a plan of 0.7 (Fig. 5c), though, we get a repeating rhythmic pattern consisting of the G-C transition that begins both melodies 1 and 2, followed by the rest that ends melody 2 after its final G-C transition. A plan of 0.5 (Fig. 5d)—halfway between the two original plans—produces phrases that are complex hybrids of both original melodies. Extra time slices are included in this case to show the behavior of the network beyond the original 34 time slices. Finally, a plan of 0.2 (Fig. 5e) yields a melody starting in a very similar manner to melody 2, but with traces of melody 1, especially the G-E transition at slice 25. Melody 2 does not even contain a E.

Note that even if we change the plan continuously between 0.0 and 1.0, the melodies generated will not change continuously. That is, all plans from 1.0 down to about 0.85 will generate melody 1; all plans from 0.85 down to about 0.73 will generate the melody in Fig. 5b; and so on down to plan 0.0 for melody 2. These discrete bands of different melodies are caused by the effect of the weights from the plan unit (or units when more than one is used) on the hidden units (see Fig. 3). These weights act as biases on the hidden units, causing them to compute different functions of the context values. The context values can be thought of as points in some higher-dimensional space, and the hidden units act as planes to cut this space up into regions for different outputs. Only when these planes have been shifted enough to put some of the context points in different regions will the output melodies be changed. The plans must change quite a bit before they shift the hidden unit planes sufficiently; thus we get bands in which the same melody is generated for different but nearby plans.

In interpolating between two melodies in this way, the network is not simply computing a weighted average of the pitches of the original two melodies, as Mathews did in his graphic score manipulation language GRIN, when he combined *The British Grenadiers* and *When Johnny Comes Marching Home* (Mathews and Rosler 1968). A violation of such averaging can be seen in the case of the E at time slice 26 in the interpolated melody in Fig. 5e, where both original melodies 1 and 2 have a G at this position. Moreover, the network is not computing any strict function of the particular pitches of the original melodies on a time-slice by time-slice basis; proof of this is seen again in the melody in Fig. 5e, which has a G at time slice 25 just before the E at slice 26, even though both melodies 1 and 2 have Fs at both time slices. Thus even though

the original melodies are the same at both of these positions, the new melody is different, indicating that something more than just the momentary pitches of the originals is going into its creation (as we would suspect from the previous discussion of spliced transitions).

Interpretations

One way to think about what the network actually is doing is in terms of constructing a complicated, higher-dimensional melody space, with each point in that space corresponding to a melody. The network learns to put one melody at a point identified by the first plan, another melody at a point identified by the second plan, and so on for all the melodies and plans in the training set. When it is given intermediate (or extrapolated) plans, the network then indexes some new point in the melody space between (or beyond) the original melodies. Depending on the structure of the melody space constructed, different new melodies will be produced.

The melody space can be altered in many ways. If we train a network on additional melodies, for instance, the new things it learns will change the space. For example, suppose we train a network with two plan units on melodies 1 and 2 from before and on melodies 3 and 4 as shown in Figs. 6a and 6b, with plans (0.0, 0.0), (1.0, 0.0), (0.0, 1.0), and (1.0, 1.0), respectively. Now if we interpolate between just melodies 1 and 2 as before, by using a plan of (0.5, 0.0), we get the result shown in Fig. 6c. This is quite different from the corresponding interpolated melody in Fig. 5d, when the network was trained only on melodies 1 and 2. Another way to alter the melody space and the resulting composed melodies is to use a different number of hidden units. If we train a network with only 8 hidden units (instead of the 15 used in all other cases) on melodies 1 and 2 with a single plan unit and then use a plan of 0.5 again to interpolate between them, we get the (different) pattern shown in Fig. 6d. Finally, simply retraining the same network design on the same melodies can give different melody spaces each time, due to the random starting point of each training session. Figure 6e shows the interpolation result of using plan 0.5 with the same network and melodies as in Fig. 5, but after going through a different training session. Again the new melody is different from all the others in this discussion.

Now let us go to the opposite extreme of analysis and consider the network's behavior during a single sequence at the time-slice by time-slice level, instead of the whole population of sequences. What the network has learned to do here is to associate particular patterns at the input layer, the context and plan units, with particular patterns at the output units. This pattern association is a type of rule-governed behavior; the network has learned rules of the form "these past pitches mean this pitch is next." Bharucha and Todd (1989) present an example of this behavior in the case of learning chord progressions. These rules emerge from the data the network is trained on. The more often particular sorts of transitions are seen in the training set, the stronger the corresponding pattern-association rule becomes. Furthermore, the nature of the network's computation lets these rules generalize to new situations and new contexts that did not occur in the training set. This generalization occurs based on the similarity of the new situations to prototype situations the network has learned about. This type of generalization is a crucial difference between the network's behavior and that of strict, rule-based compositional systems. Additionally, the rules the network develops are not really symbolic, as they are in most other compositional systems, but rather are stated at a lower descriptive level of vectors of past pitch activity.

Finally, from an intermediate perspective, we can consider the network as it produces an ongoing single sequence. Jordan (1986b) has shown that sequential networks of this type, being nonlinear dynamical systems, can develop attractor limit cycles, that is, repeating sequences that other sequences can fall into. The examples discussed previously showed several instances of this behavior. For instance, the repeating patterns shown in Figs. 4b–d and 5c are limit cycles—once the network begins producing one of these patterns (for the given plan), it will remain locked in this cycle. These limit cycles can be quite short, as in these instances, or

quite long, perhaps not emerging until the network has been run for many time slices beyond the length of the original trained melodies. Furthermore, similar sequences are likely to be sucked into these cycles: if we were to alter the initial context, it would only take a few time slices for the sequence to fall back into the original repeating pattern.

Other Methods for Generating New Melodies

The sequential networks presented here are not limited to plan interpolation and extrapolation for creating new melodies. Many other methods are possible. For example, the plan could be changed dynamically throughout the course of the generated melody so that the new sequence might begin close to one of the trained melodies and end close to another. The context could be altered in various ways, either before melody generation to provide a nonzero starting point, or during melody generation to provide more or less sudden shifts to different melodic points. The weights in the network could be changed randomly (or systematically) to alter the overall behavior of the network and probably make it more unpredictable and less likely to mimic the melodies in the training set. Additionally, the network could be treated in a probabilistic, nondeterministic way, by training the output units to have activation levels corresponding to their melodic strength in a particular context, and then actually picking the output to be used in a probabilistic manner based on these levels. In this case, the stronger a pitch fits a particular context, the more likely it is to be picked as the actual output, but other weaker candidates always have a chance of being chosen as well. This probabilistic behavior comes closer to that of a Markov process, but still remains distinct.

Applications and Further Directions

The sequential network described here can be a useful tool for the algorithmic composition of melodic sequences similar, in some ways, to a set of chosen melodies. This method has at least one great advantage over other, rule-based methods of algorithmic composition, which is that explicit rules need not be devised. Rather, the composer need only collect examples of the desired sorts of musical sequences and train the network on them. The training phase, while time-consuming, is automatic, and once done, the network can produce a large variety of melodies based on those it has learned. These new melodies, while incorporating important elements of the training set, remain more or less unpredictable and therefore musically interesting.

As it now stands, the place for this network method of composition is restricted to generating relatively short musical lines, high in local structure but lacking in overall global organization. Still, this can be very useful for generating new phrases based on previously composed examples, for coming up with new rhythmic patterns and interpolating between them, or, if the pitch units are instead interpreted as chord units, for coming up with new chord progressions using elements of old ones. This method can also find use as a means of overcoming composer's block, as suggested by David Cope as one of the applications of his EMI system (Cope 1987).

To extend the network's usefulness to a broader range of musical applications, several problems need to be overcome. First, the network's music representation should be improved to allow polyphony, as well as other musical parameters beyond just pitch and duration. A better way of handling rhythm and timing needs to be developed. Some method for letting the network learn variable-length pattern associations is needed so that it can insert ornamentations or delete passages from sequences. And most importantly, the length of sequences that the network can learn must be increased. This can be improved by the addition of other forms of context, such as a global clock or counter added to the inputs. But the real solution will lie in developing a method for letting the network learn the hierarchical organization of sequences, so that melodies need not be treated as simply one long flat string of notes with no higher-level structure. Approaches such as using the outputs of one slow-moving sequential network as the successive plans of another faster-paced sequence network have been suggested,

but the problem remains of learning how to divide the hierarchical structure between the networks involved.

Conclusions

The sequential network methods presented here can be profitable avenues for the exploration of new approaches to algorithmic composition. By training networks on selected melodies and then using the network's inherent, rule-like generalization ability and limit-cycle dynamics, new melodies similar to the original training examples can be generated. The variety of musical patterns that can be created under this one unifying approach indicate its potential for useful application. The possibilities for further work expanding the capabilities of this approach are virtually limitless.

Acknowledgments

This research was supported by a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation. I wish to thank Dave Rumelhart, Jamshed Bharucha, and Gareth Loy for their encouragement in this research, and Mary Ann Norris for her aid in preparing this paper.

References

Bharucha, J. J., and P. M. Todd. 1989. "Modeling the Perception of Tonal Structure with Neural Nets." *Computer Music Journal* 13(4):44–53.

Cope, D. 1987. "An Expert System for Computer-assisted Composition." *Computer Music Journal* 11(4):30–46.

Dolson, M. 1989. "Machine Tongues XII: Introduction to Neural Nets." *Computer Music Journal* 13(3):00–00.

Elman, J. L. 1988. "Finding Structure in Time." Technical Report 8801. La Jolla: University of California, Center for Research in Language.

Jones, K. 1981. "Compositional Applications of Stochastic Processes." *Computer Music Journal* 5(2):45–61.

Jordan, M. I. 1986a. "Serial Order: A Parallel Distributed Processing Approach." Technical Report ICS-8604. La Jolla: University of California, Institute for Cognitive Science.

Jordan, M. I. 1986b. "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine." *Proceedings of the Eighth Annual Conference of the Cognitive Science Society.* Hillsdale, N.J.: Erlbaum Associates.

Mathews, M. V., and L. Rosler. 1968. "Graphical Language for the Scores of Computer-generated Sounds." *Perspectives of New Music* 6:92–118.

McClelland, J. L., and D. E. Rumelhart. 1988. *Explorations in Parallel Distributed Processing.* Cambridge, Massachusetts: MIT Press.

Miyata, Y. 1987. "SunNet Version 5.2: A Tool for Constructing, Running, and Looking into a PDP Network in a Sun Graphics Window." Technical Report ICS-8708. La Jolla: University of California, Institute for Cognitive Science.

Rumelhart, D. E., and J. L. McClelland, eds. 1986. *Parallel Distibuted Processing: Explorations in the Microstructure of Cognition.* Cambridge, Massachusetts: MIT Press.

Sejnowski, T. J., and C. R. Rosenberg. 1987. "Parallel Networks that Learn to Pronounce English Text." *Complex Systems* 1:145–168.

Todd, P. M. 1988. "A Sequential Network Design for Musical Applications." In D. Touretzky, G. Hinton, and T. Sejnowski, eds. *Proceedings of the 1988 Connectionist Models Summer School.* Menlo Park, California: Morgan Kaufmann.

Waibel, A., et al. 1987. "Phoneme Recognition Using Time-Delay Neural Networks." Technical Report TR-I-0006. ATR Interpreting Telephony Research Laboratories.