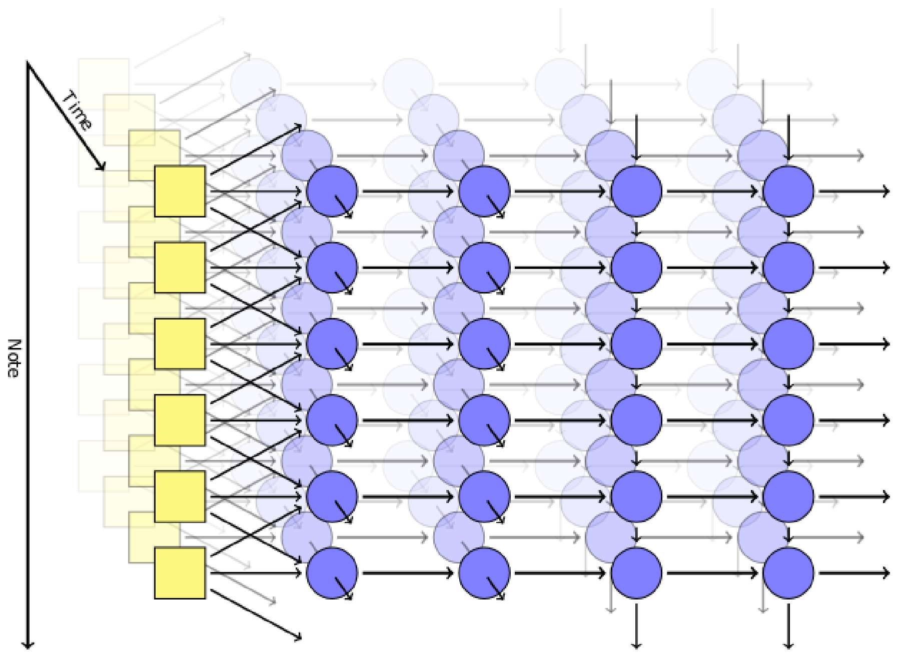
hexahedria

Daniel Johnson's personal fragment of the web

Composing Music With Recurrent Neural Networks



(Update: A paper based on this work has been accepted at EvoMusArt 2017! See here for more details.)

It's hard not to be blown away by the surprising power of neural networks these days. With enough training, so called "deep neural networks", with many nodes and hidden layers, can do impressively well on modeling and predicting all kinds of data. (If you don't know what I'm talking about, I recommend reading about recurrent character-level language models, Google Deep Dream, and neural Turing machines. Very cool stuff!) Now seems like as good a time as ever to experiment with what a neural network can do.

For a while now, I've been floating around vague ideas about writing a program to compose music. My original idea was based on a fractal decomposition of time and some sort of repetition mechanism, but after reading more about neural networks, I decided that they would be a better fit. So a few weeks ago, I got to work designing my network. And after training for a while, I am happy to report remarkable success!

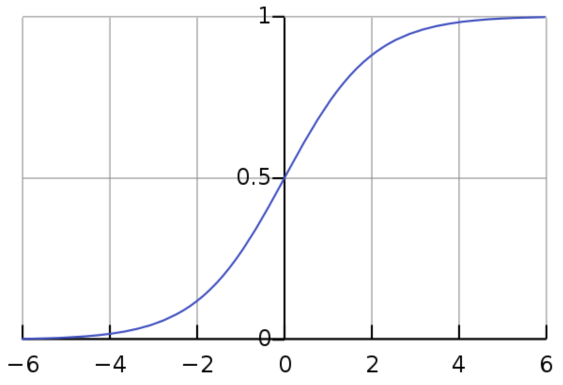
Here's a taste of things to come:

0:00 / 4:02

But first, some background about neural networks, and RNNs in particular. (If you already know all about neural networks, feel free to skip this part!)

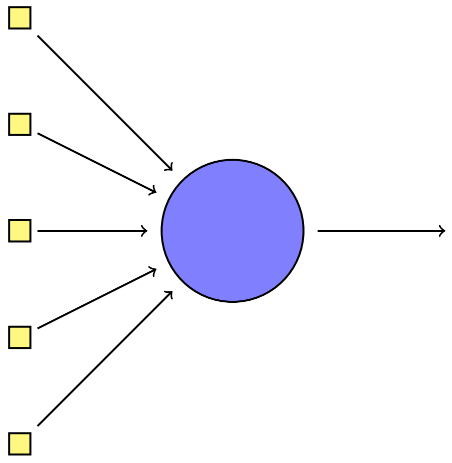
Feedforward Neural Networks:

A single node in a simple neural network takes some number of inputs, and then performs a weighted sum of those inputs, multiplying them each by some weight before adding them all together. Then, some constant (called "bias") is added, and the overall sum is then squashed into a range (usually -1 to 1 or 0 to 1) using a nonlinear activation function, such as a sigmoid function.

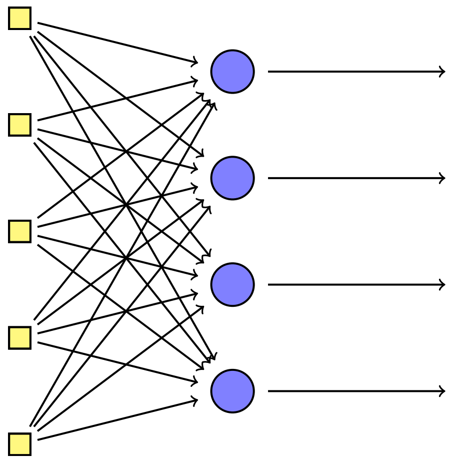


A sigmoid function.

We can visualize this node by drawing its inputs and single output as arrows, and denoting the weighted sum and activation by a circle:

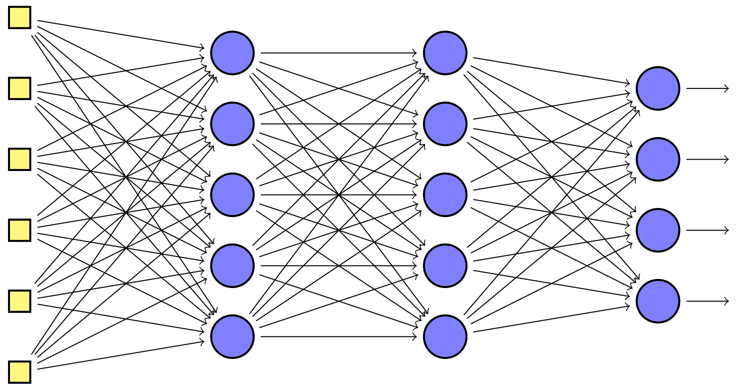


We can then take multiple nodes and feed them all the same inputs, but allow them to have different weights and biases. This is known as a layer.



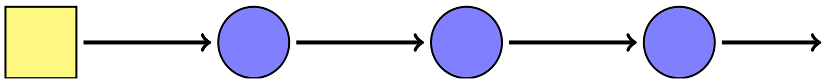
(Note: Because each node in the layer performs a weighted sum, but they all share the same inputs, we can calculate the outputs using a matrix multiplication, followed by elementwise activation! This is one reason why neural networks can be trained so effectively.)

Then we can connect multiple layers together:



and voila, we have a neural network. (Quick note on terminology: The set of inputs is called the “input layer”, the last layer of nodes is called the “output layer”, and all intermediate node layers are called “hidden layers”. Also, in case it isn’t clear, all the arrows from each node carry the same value, since each node has a single output value.)

For simplicity, we can visualize the layers as single objects, since that's how they are implemented most of the time:

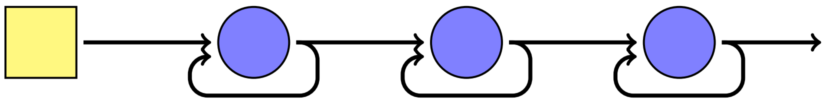


From this point on, when you see a single circle, that represents an entire layer of the network, and the arrows represent vectors of values.

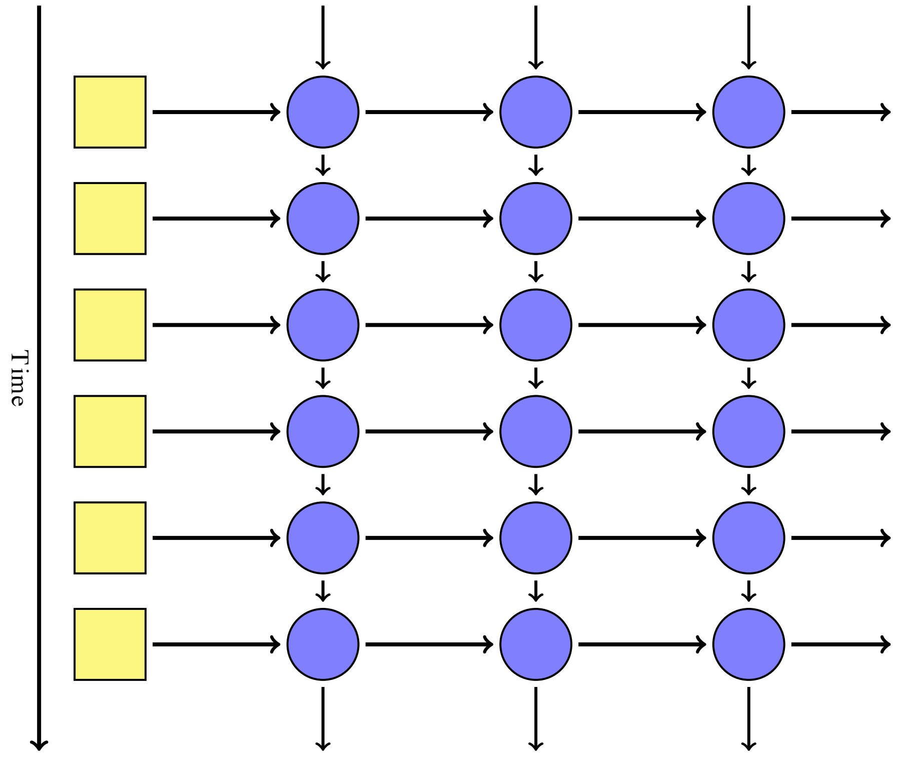
Recurrent Neural Networks

Notice that in the basic feedforward network, there is a single direction in which the information flows: from input to output. But in a recurrent neural network, this direction constraint does not exist. There are a lot of possible networks that can be classified as recurrent, but we will focus on one of the simplest and most practical.

Basically, what we can do is take the output of each hidden layer, and feed it back to itself as an additional input. Each node of the hidden layer receives both the list of inputs from the previous layer and the list of outputs of the current layer in the last time step. (So if the input layer has 5 values, and the hidden layer has 3 nodes, each hidden node receives as input a total of 5+3=8 values.)



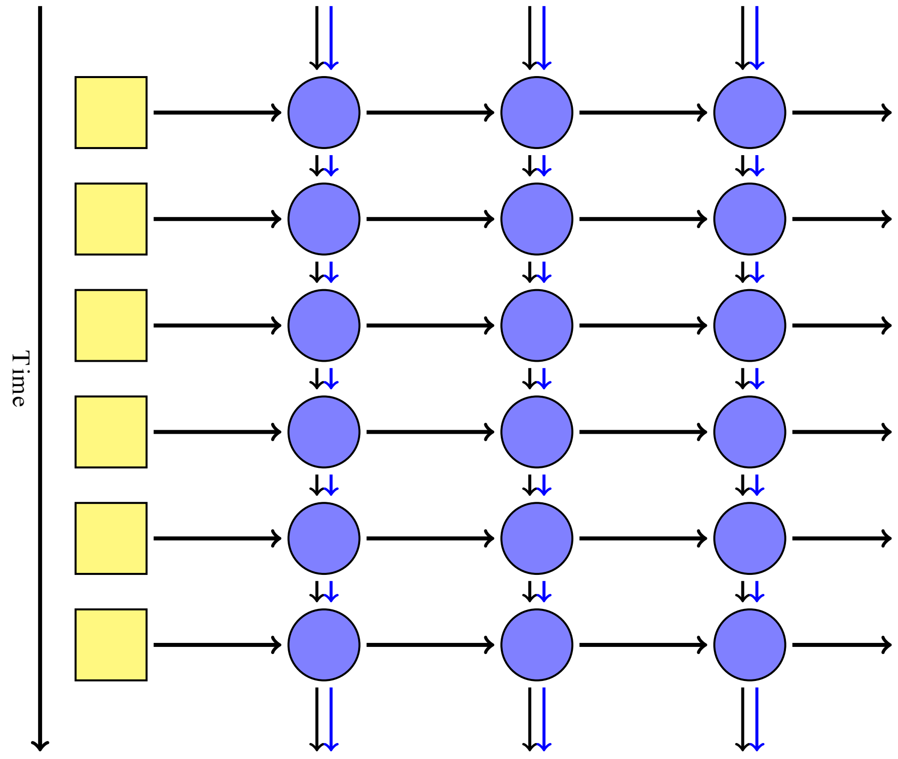
We can show this more clearly by unwrapping the network along the time axis:



In this representation, each horizontal line of layers is the network running at a single time step. Each hidden layer receives both input from the previous layer and input from itself one time step in the past.

The power of this is that it enables the network to have a simple version of memory, with very minimal overhead. This opens up the possibility of variable-length input and output: we can feed in inputs one-at-a-time, and let the network combine them using the state passed from each time step.

One problem with this is that the memory is very short-term. Any value that is output in one time step becomes input in the next, but unless that same value is output again, it is lost at the next tick. To solve this, we can use a Long Short-Term Memory (LSTM) node instead of a normal node. This introduces a "memory cell" value that is passed down for multiple time steps, and which can be added to or subtracted from at each tick. (I'm not going to go into all of the details, but you can read more about LSTMs in the original paper.)



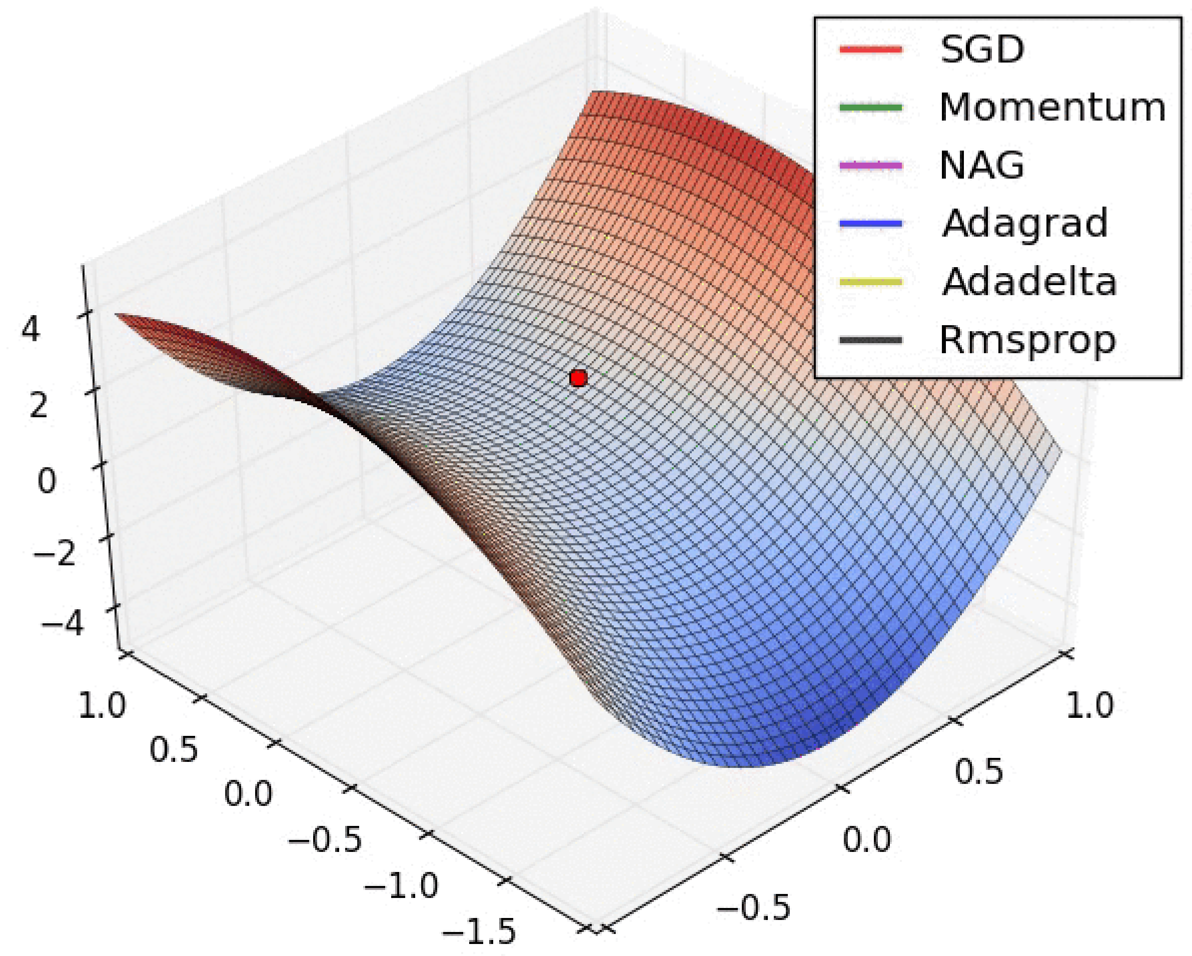
We can think of the memory cell data being sent in parallel with the activation output. Above, I have shown it using a blue arrow, and in my following diagrams I will omit it for simplicity.

Training Neural Networks

All those pretty pictures are nice, but how do we actually get the networks to output what we want? Well, the neural network behavior is determined by the set of weights and biases that each node has, so we need to adjust those to some correct value.

First, we need to define how good or bad any given output is, given the input. This value is called the *cost*. For example, if we were trying to use a neural network to model a mathematical function, the cost might be the difference between the function answer and the network output, squared. Or if we were trying to model the likelihood of letters appearing in a particular order, the cost might be one minus the probability of predicting the correct letter each time.

Once we have this cost value, we can use *backpropagation*. This boils down to calculating the gradient of the cost with respect to the weights (i.e the derivative of cost with respect to each weight for each node in each layer), and then using some optimization method to adjust the weights to reduce the cost. The bad news is that these optimization methods are often very complex. But the good news is that many of them are already implemented in libraries, so we can just feed our gradient to the right function and let it adjust our weights correctly. (If you're curious, and don't mind some math, some optimization methods include stochastic gradient descent, Hessian-free optimization, AdaGrad, and AdaDelta.)



Visualization of some commonly used optimization methods.

Can They Make Music?

All right, enough background. From here on, I'll be talking mostly about my own thought process and the design of my network architecture.

When I was starting to design my network's architecture, I naturally looked up how other people have approached this problem. A few existing methods are collected below:

- Bob Sturm uses a character-based model with an LSTM to generate a textual representation of a song (in abc notation). The network seems to only be able to play one note at a time, but achieves interesting temporal patterns.

- Doug Eck, in A First Look at Music Composition using LSTM Recurrent Neural Networks, uses LSTMs to do blues improvization. The sequences chosen all have the same set of chords, and the network has a single output node for each note, outputting the probability of that note being played at each time step. Results are promising in that it learns temporal structure, but is pretty restricted as to what it can output. Also, there is no distinction between playing a note and holding it, so the network cannot rearticulate held notes.
- Nicolas Boulanger-Lewandowski, in Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription, uses a network with two parts. There is an RNN to handle time dependency, which produces a set of outputs that are then used as the parameters for a restricted Boltzmann machine, which in turn models the conditional distribution of which notes should be played with which other notes. This model actually produces quite nice-sounding music, but does not seem to have a real sense of time, and only plays a couple of chords. (See here for the algorithm and sample output.)

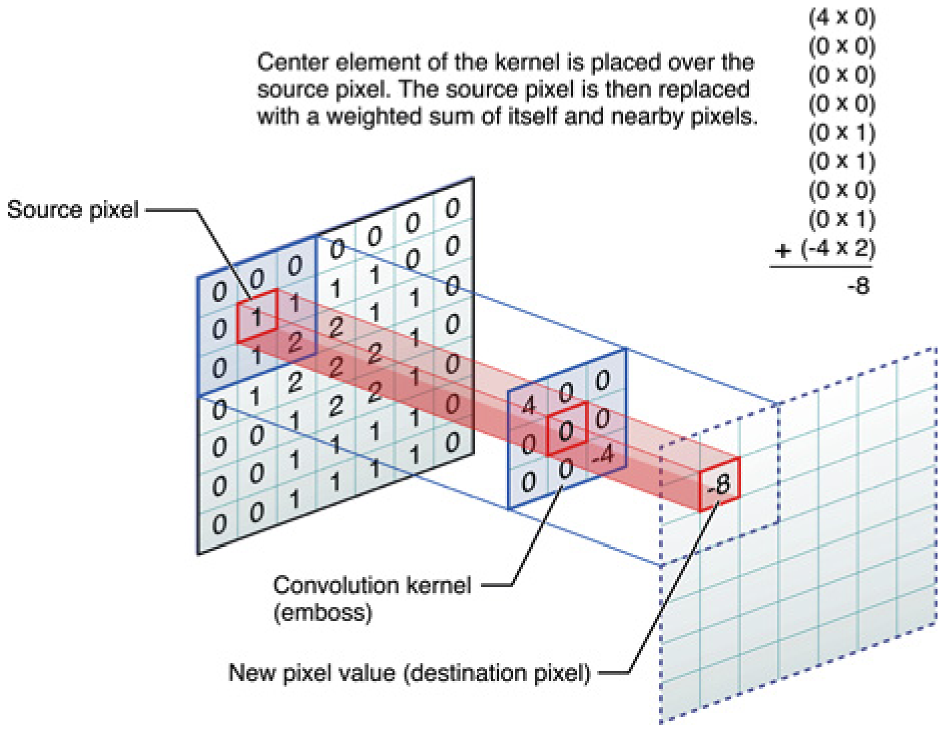
For my network design, there were a few properties I wanted it to have:

- Have some understanding of time signature: I wanted to give the neural network its current time in reference to a time signature, since most music is composed with a fixed time signature.
- Be time-invariant: I wanted the network to be able to compose indefinitely, so it needed to be identical for each time step.
- Be (mostly) note-invariant: Music can be freely transposed up and down, and it stays fundamentally the same. Thus, I wanted the structure of the neural network to be almost identical for each note.
- Allow multiple notes to be played simultaneously, and allow selection of coherent chords.
- Allow the same note to be repeated: playing C twice should be different than holding a single C for two beats.

I'll talk a bit more about the invariance properties, because I decided those were the most important. Most existing RNN-based music composition approaches are invariant in time, since each time step is a single iteration of the network. But they are in general not invariant in note. There is usually some specific output node that represents each note. So transposing everything up by, say, one whole step, would produce a completely different output. For most sequences, this is something you would want: "hello" is completely different from "ifmmp", which is just "transposed" one letter. But for music, you want to emphasize the relative relationships over the absolute positions: a C major chords sounds more like a D major chord than like a C minor chord, even though the C minor chord is closer with regard to absolute note positions.

There is one kind of neural network that is widely in use today that has this invariant property along multiple directions: convolutional neural networks for image recognition.

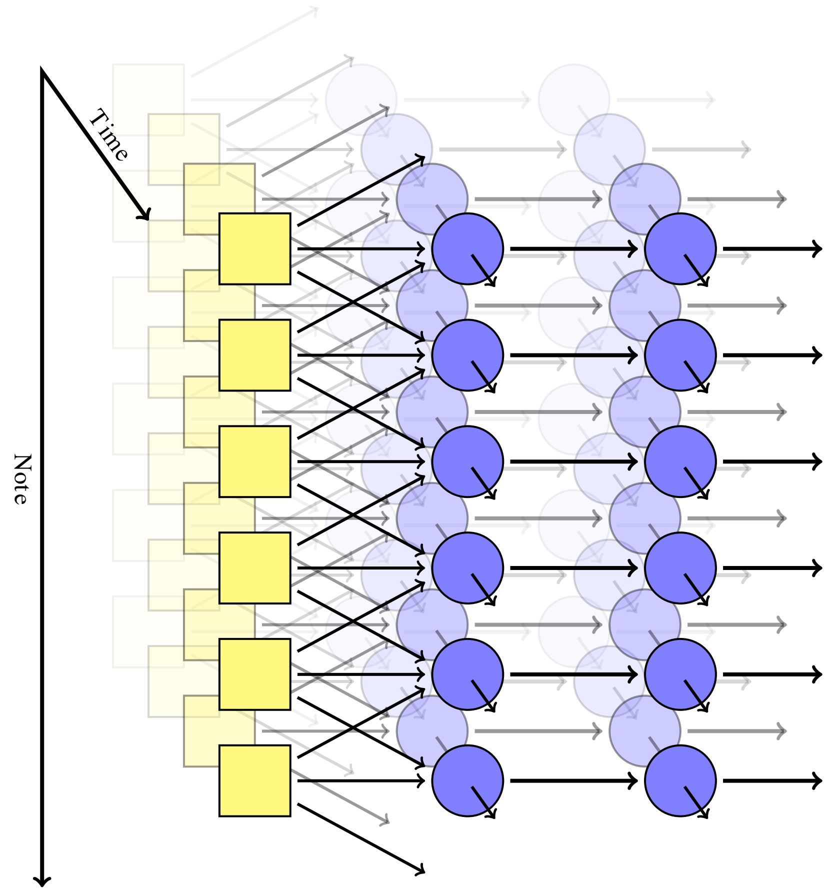
These work by basically learning a convolution kernel and then applying that same convolution kernel across every pixel of the input image.


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

How convolution works. Each pixel is replaced by a weighted sum of the surrounding pixels. The neural network has to learn the weights. Picture from developer.apple.com.

Hypothetically, what would happen if we replaced the convolution kernel with something else? Say, a recurrent neural network? Then each pixel would have its own neural network, which would take input from an area around the pixel. Each neural network would in turn have its own memory cells and recurrent connections across time.

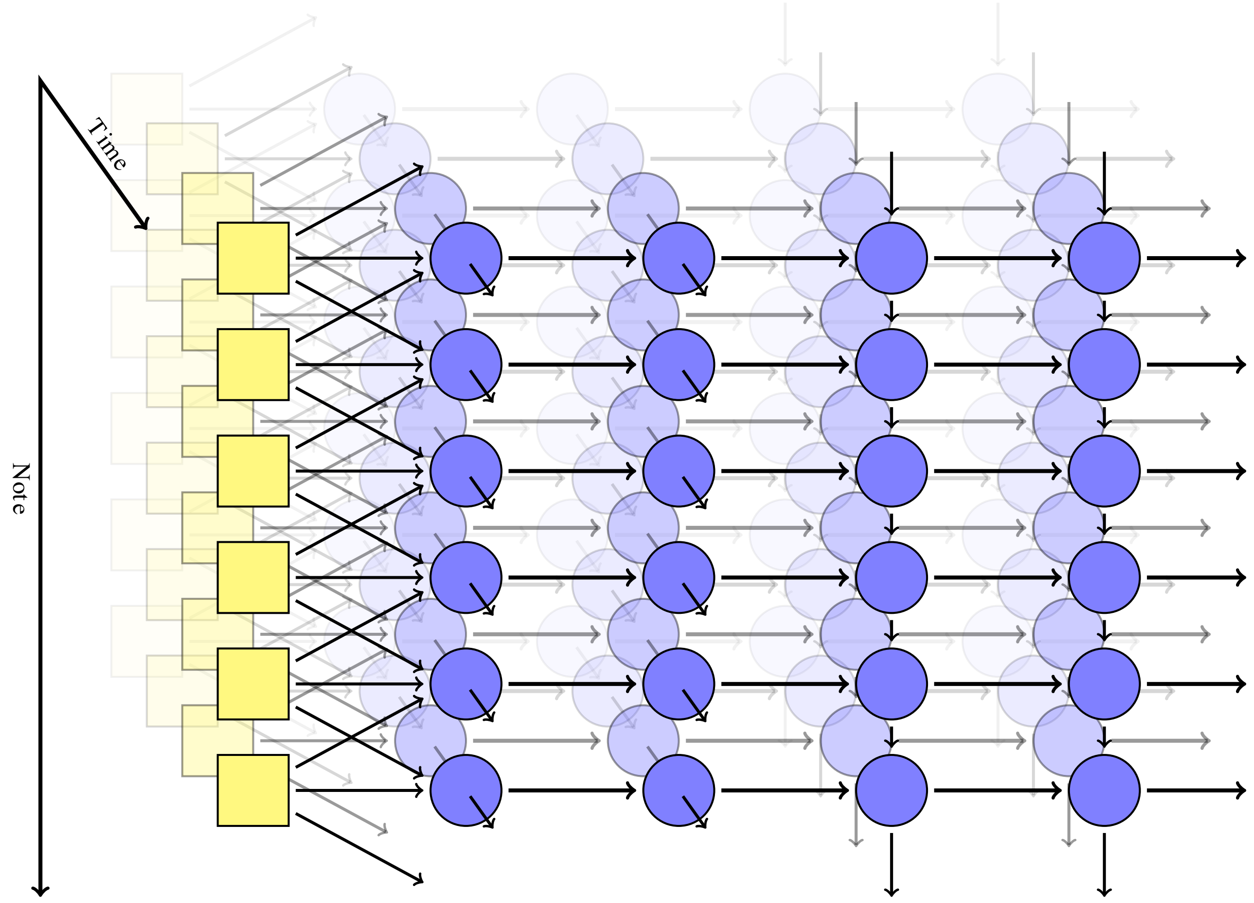
Now replace pixels with notes, and we have an idea for what we can do. If we make a stack of identical recurrent neural networks, one for each output note, and give each one a local neighborhood (for example, one octave above and below) around the note as its input, then we have a system that is invariant in both time and notes: the network can work with relative inputs in both directions.



Note: I have rotated the time axis here! Notice that time steps now come out of the page, as do the recurrent connections. You can think of each of the "flat" slices as a copy of the basic RNN picture from above. Also, I am showing each layer getting input from one note above and below. This is a simplification: the real network gets input from 12 notes (the number of half steps in an octave) in each direction.

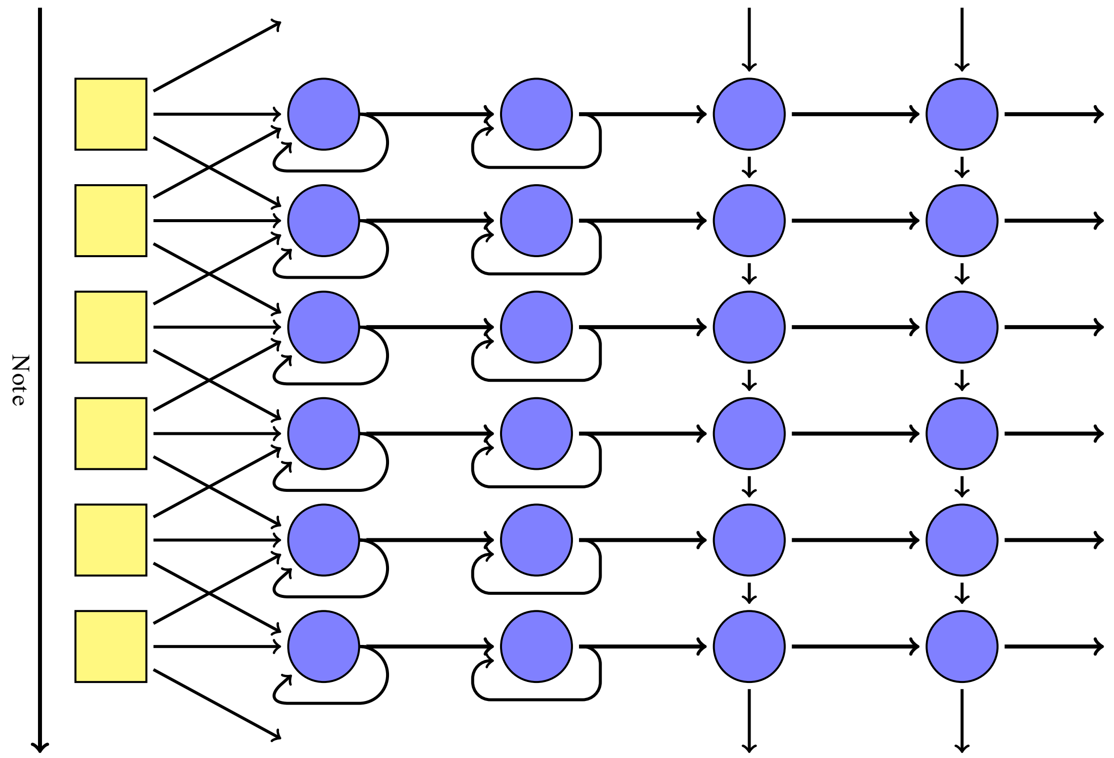
However, there is still a problem with this network. The recurrent connections allow patterns in time, but we have no mechanism to attain nice chords: each note's output is completely independent of every other note's output. Here we can draw inspiration from the RNN-RBM combination above: let the first part of our network deal with time, and let the second part create the nice chords. But an RBM gives a single conditional distribution of a bunch of outputs, which is incompatible with using one network per note.

The solution I decided to go with is something I am calling a "biaxial RNN". The idea is that we have two axes (and one pseudo-axis): there is the time axis and the note axis (and the direction-of-computation pseudo-axis). Each recurrent layer transforms inputs to outputs, and also sends recurrent connections along one of these axes. But there is no reason why they all have to send connections along the same axis!



Notice that the first two layers have connections across time steps, but are independent across notes. The last two layers, on the other hand, have connections between notes, but are independent between time steps. Together, this allows us to have patterns both in time and in note-space without sacrificing invariance!

It's a bit easier to see if we collapse one of the dimensions:



Now the time connections are shown as loops. It's important to remember that the loops are always delayed by one time step: the output at time *t* is part of the input at time *t*+1.

Input and Output Details

My network is based on this architectural idea, but of course the actual implementation is a bit more complex. First, we have the input to the first time-axis layer at each time step: (the number in brackets is the number of elements in the input vector that correspond to each part)

- **Position** [1]: The MIDI note value of the current note. Used to get a vague idea of how high or low a given note is, to allow for differences (like the concept that lower notes are typically chords, upper notes are typically melody).
- **Pitchclass** [12]: Will be 1 at the position of the current note, starting at A for 0 and increasing by 1 per half-step, and 0 for all the others. Used to allow selection of more common chords (i.e. it's more common to have a C major chord than an E-flat major chord)
- **Previous Vicinity** [50]: Gives context for surrounding notes in the last timestep, one octave in each direction. The value at index 2(i+12) is 1 if the note at offset i from current note was played last timestep, and 0 if it was not. The value at 2(i+12) + 1 is 1 if that note was *articulated* last timestep, and 0 if it was not. (So if you play a note and hold it, first timestep has 1 in both, second has it only in first. If you repeat a note, second will have 1 both times.)
- **Previous Context** [12]: Value at index i will be the number of times any note x where (x-i-pitchclass) mod 12 was played last timestep. Thus if current note is C and there were 2 E's last timestep, the value at index 4 (since E is 4 half steps above C) would be 2.
- **Beat** [4]: Essentially a binary representation of position within the measure, assuming 4/4 time. With each row being one of the beat inputs, and each column being a time step, it basically just repeats the following pattern:

  ```
  0101010101010101
  0011001100110011
  0000111100001111
  0000000011111111
  ```

  However, it is scaled to [-1, 1] instead of [0,1].</p>

Then there is the first hidden LSTM stack, which consists of LSTMs that have recurrent connections along the time-axis. The last time-axis layer outputs some note state that represents any time patterns. The second LSTM stack, which is recurrent along the note axis, then scans up from low notes to high notes. At each note-step (equivalent of time-steps) it gets as input

- the corresponding note-state vector from the previous LSTM stack

- a value (0 or 1) for whether the previous (half-step lower) note was chosen to be played (based on previous note-step, starts 0)
- a value (0 or 1) for whether the previous (half-step lower) note was chosen to be articulated (based on previous note-step, starts 0)

After the last LSTM, there is a simple, non-recurrent output layer that outputs 2 values:

- **Play Probability**, which is the probability that this note should be chosen to be played
- **Articulate Probability**, which is the probability the note is articulated, given that it is played. (This is only used to determine rearticulation for held notes.)

The model is implemented in Theano, a Python library that makes it easy to generate fast neural networks by compiling the network to GPU-optimized code and by automatically calculating gradients for you. The error messages can be a bit confusing (since exceptions tend to get thrown while it is running its generated code, not yours), but it's well worth the hassle. ## Using the Model

During training, we can feed in a randomly-selected batch of short music segments. We then take all of the output probabilities, and calculate the cross-entropy, which is a fancy way of saying we find the likelihood of generating the correct output given the output probabilities. After some manipulation using logarithms to make the probabilities not ridiculously tiny, followed by negating it so that it becomes a minimization problem, we plug that in as the cost into the AdaDelta optimizer and let it optimize our weights.

We can make training faster by taking advantage of the fact that we already know exactly which output we will choose at each time step. Basically, we can first batch all of the notes together and train the time-axis layers, and then we can reorder the output to batch all of the times together and train all the note-axis layers. This allows us to more effectively utilize the GPU, which is good at multiplying huge matrices.

To prevent our model from being overfit (which would mean learning specific parts of specific pieces instead of overall patterns and features), we can use something called *dropout*. Applying dropout essentially means randomly removing half of the hidden nodes from each layer during each training step. This prevents the nodes from gravitating toward fragile dependencies on each other and instead promotes specialization. (We can implement this by multiplying a mask with the outputs of each layer. Nodes are "removed" by zeroing their output in the given time step.)

During composition, we unfortunately cannot batch everything as effectively. At each time step, we have to first run the time-axis layers by one tick, and then run an entire recurrent sequence of the note-axis layers to determine what input to give to the time-axis layers at the next tick. This makes composition slower. In addition, we have to add a correction factor to account for the dropout during training. Practically, this means multiplying the output of each node by 0.5. This prevents the network from becoming overexcited due to the higher number of active nodes.

I trained the model using a g2.2xlarge Amazon Web Services instance. I was able to save money by using "spot instances", which are cheaper, ephemeral instances that can be shut down by Amazon and which are priced based on supply and demand. Prices fluctuated between $0.10 and $0.15 an hour for me, as opposed to $0.70 for a dedicated on-demand instance. My model used two hidden time-axis layers, each with 300 nodes, and two note-axis layers, with 100 and 50 nodes, respectively. I trained it using a dump of all of the midi files on the Classical Piano Midi Page, in batches of ten randomly-chosen 8-measure chunks at a time.

Warning about using spot instances: Be prepared for the system to go down without warning! (This happened to me twice over a few days of experimenting and training.) Make sure you are saving often during training, so that you do not lose too much data. Also, make sure to uncheck "Delete on Termination" so that Amazon doesn't wipe your instance memory when it shuts it down!

Results

Without further ado, here is a selection of some of the output of the network:

0:00 / 4:02 0:00 / 4:02

0:00 / 4:02 0:00 / 4:02

0:00 / 4:02 0:00 / 4:02

0:00 / 4:00 0:00 / 3:20

0:00 / 4:02 0:00 / 4:02

Sometimes it seems to get stuck playing the same chords for a really long time. It hasn't seemed to have learned how long to hold those. But overall, the output is pretty interesting. (I was tempted to remove some of the repetitive bits, but I decided that would be unfaithful to the point of the project, so I left them in.)
The source code for everything is available on GitHub. It's not super polished, but it should be possible to figure out what's going on.
You might also be interested in the discussions on Hacker News and Reddit.
If you want to replicate results yourself, you may be interested in the final training weights of my network.

02 Aug 2015
Tags: composition, music, neural network, project writeup, python, Theano

237 Comments **hexahedria** 1 **Login**