1.1 Project Outcome

The undertaken task of researching and implementing music composition using neural networks and GAs was a challenge but a great exercise in learning. The functional goal of composing music using NN was achieved. The GA method was studied only as a learning goal.

Music composition using NNs is similar to the task of language modeling. This is because music is very similar to human language, with similar structure and grammar as a language. A considerable amount of time and effort was spent in researching how NNs can be used for the task of music composition. Recurrent Neural Networks (RNNs) and Long Short Term Memory (LSTM) NNs were studied. The LSTM method was chosen because of its capability of maintaining context of long-term dependencies in an input sequence.

1.2 Report on Requirements

The functional goal of composing music using neural networks was achieved. MIDI output files were generated based on a seed key note. The generated music was satisfactorily pleasant to hear.

The task of designing a LSTM neural network to compose music was challenging and did not leave much time for exploring the GA method for music composition. The GA method was treated as a learning goal.

1.3 Neural Network Theory, Design and Tools

1.3.1 Theory:

For the task of music composition, we need to design a neural network that can maintain the context of musical notes generated over a period of time since musical structure consists of definite relationships between notes across time. A Long Short Term Memory (LSTM) neural network is suited for this purpose. Before going into the design of a LSTM Neural Network, it is essential to understand basic Recurrent Neural Networks (RNN) and why they are not best suited for long term dependencies in sequences.

A RNN is as shown below:

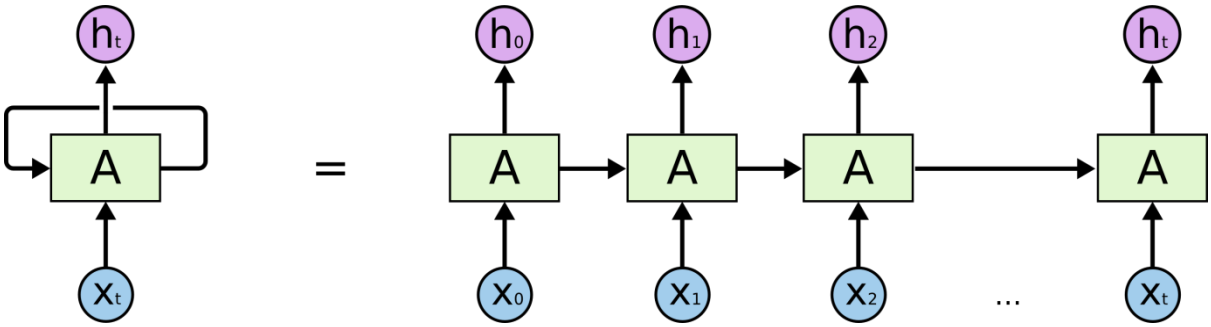

Figure 1

(Picture from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

The basic idea behind such neural networks is a feedback loop in the hidden layer that allows information to persist across time steps. As seen on the left side of figure (1) above is a part of the hidden layer that accepts input X_t and outputs a value h_t. On the right side of figure (1), we see this loop unrolled over multiple time steps. At each time step, the cell 'A' receives the input X_t while also maintaining a hidden state, which it updates based on the input. This can be explained by the equations below:

```
rnn = RNN( )
y = RNN.step( )

class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

The above steps are the forward pass of a simple RNN. 'x' is the input, the 'W's are the weights and y is the output. As we can see, the hidden state always maintains context of all inputs that have passed forward. This allows RNNs to handle dependencies in input. Multiple layers can also be stacked to improve performance. This doesn't always work well though and in fact, RNNs are not very good at remembering long term dependencies. They also suffer from the vanishing and exploding gradients problem. This is where LSTMs come in.

At the core of a LSTM NN is the cell in its hidden layer that consists of three gates. The gates are composed out of a sigmoid neural net layer and a pointwise multiplication operation. They behave like control structures that regulate how much information persists in a cell and how much goes out. The picture below shows the basic idea behind LSTM NNs:

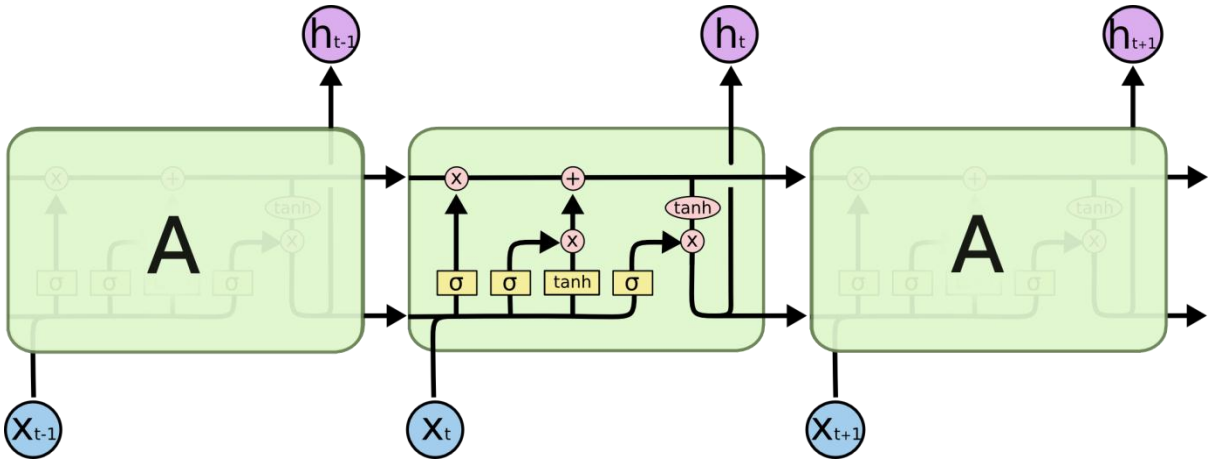

Figure 2

(Picture from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The equations below show how the internal state is updated at the forget layer (f_t), the cell layer C_t and the output layer h_t :

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

As we can see, LSTMs are much more sophisticated than RNNs.

1.3.2 Tools and Design:

For the purposes of composing music, we design a NN based on LSTM units. The input consists of a 1-hot encoded vector that represents a musical note. The output is the next predicted note (vector of probabilities). The next input to the NN is the note that was just predicted. This allows us to generate a related sequence of musical notes. Please refer to section 1.3 and figure (1) in the design document for details on this.

The TensorFlow machine learning platform was used to implement and execute the designed neural network. The availability of a LSTM neural network library and the sequence-to-sequence model for decoding sequences made TensorFlow a viable choice.

The ‘abc2midi’ tool (available publicly) was used to convert music generated in the abc notation to a MIDI file for listening purposes.

The basic design of the LSTM model that was used is as shown below:

lstmModel:

```
rnn_size = 128 ; num_layers = 2;  batchSize = 1;  seqLen  = 500
learningRate = 0.002 ;  decayRate = 0.97

// The core lstm
lstmLayer = rnn_cell.BasicLSTMCell(rnn_size)
lstmOp = rnn_cell.MultiRNNCell([lstmLayer] * num_layers)
```

```
inputData = tf.placeHolder(int, [batch_size, sequence length])
targetOutput = tf.placeHolder(int, [batch_size, sequence length])

// For output logit calculations
softmax_w = tf.get_variable(“softmax_w”, [rnn_size, vocab_size])
softmax_b = tf.get_variable(“softmax_b”, [vocab_size])

embedding = tf.get_variable("embedding", [vocab_size, rnn_size])
inputs = tf.split(1, seq_length,   tf.nn.embedding_lookup(embedding, inputData))

// During training provide correct input at all time steps
output = seq2seq.rnn_decoder(inputs, self.initial_state, lstmOp,
            loop_function=getPrev if predict else None, scope='rnnlm')
 // Output probabilities between 0 to 1 based on softmax function
 logits = tf.nn_xw_plus_b(output, softmax_w,  softmax_b)

 loss = seq2seq.sequence_loss_by_example([logits], [tf.reshape(targetOutput [-1])],
        [tf.ones([batch_size * seq_len]), vocab_size)

cost = tf.reduce_sum(loss) // set cost function

 //Set gradient optimizer
 tvars = tf.trainableVariables()
 grads = tf.clip_by_global_norm(tf.gradients(cost, tvars))
 optimizer = tf.train.AdamOptimizer(learningRate)
trainingOp = optimizer.apply_gradients(zip(grads, tvars))
```

As seen above, the model consisted of two hidden LSTM layers, each of size 128. The ‘AdamOptimizer’ was used to optimize and train the model. Also of note in the model is the value of ‘seqLen’ which represents the length of a sequence the model is trained on in one go. This is set so that it can fit in an entire melody. The value used was 500. In cases where a melody is not 500 characters long, the melody is padded with the value ‘z’ which is a musical rest in abc notation.

The model was then trained on an input dataset consisting of 1700 tunes in the abc music format (please refer to design documentation section 1.3 for details). An excerpt from this training set is as shown below:

```
K: Eminor
|:E2 F2 GD|A2 B2 d2|e4 ec|g2 e2 (3edB|^M
d2 B2 G2|dd B2 A2|A2 B2 e2|d2 A2 AG|^M
AG E3 E|A2 B2 BE|A2 B3 G|dd B2 A2|^M
G3 A ED|G2 A2 B2|1A6-|A3 FG F:|2A6|d2 e2 ge||^M
a4 ab|a2 g2 B2|g3 e ga|g2 e2 cB|^M
cB g3 B|BA f2 ad-|d4 dB|de g2 ea-|^M
a4 ab|a2 g2 b2|{g}e6|e2 d2 cB|^M
cB- B2 BG|d2 c2 B2|BA- A4|c2 B2 G2||z
```

During training, checkpoints of the model are saved at regular intervals. After training was completed, the model was sampled with a key. Examples are shown below:

Sample 1:

Seed text: “K: Amajor”

Output:

lstm_amajor.mid

Sample 2:

Seed text: “K: Eminor”

Output:

lstm_eminor.mid

Other Resources Used:

The python code for sampling from a trained model and reading batch input from the training data was used from the following publicly available code:

https://github.com/sherjilozair/char-rnn-tensorflow/blob/master/sample.py

https://github.com/sherjilozair/char-rnn-tensorflow/blob/master/utils.py

1.4 Conclusion

The task of composing music using neural networks was challenging and a great learning experience. A significant amount of time was spent on researching neural networks and strategies on how to use them for composing music. The idea of basing music composition as a character level language modeling task worked out very well. As can be observed from the attached output files, the music generated was of very good quality and also significantly different between the two outputs to conclude that the neural network did indeed learn and model the music composition process.

Future work could include experimenting with different sizes for the hidden layer and the number of hidden layers in the NN. It may also be worthwhile experimenting with different optimizers and different learning rates. Training the neural network on different kinds of data sets and observing the how the output differs based on the training set would also be something to consider.