

Learning NSynth with LSTM and BLSTM

Subtitle: Project Two

Authors:

- ❑ Parinitha Nagaraja
- ❑ Qiaoran Li

Due Date: 12/12/2018

Professors: Zachary Butler

Class: CSCI 739 Machine Learning

School: Rochester Institute of Technology

Table of Content

Table of Content	0
Overview	1
Training	1
Data processing	1
Training Setup	1
Result	2
Discussion	5
Bibliography	6

Overview

Previously, we have explored feedforward neural network\fully connected network and convolutional network on the NSynth dataset. With fully connect model, we achieved a result of 45.752%. The convolutional network achieved an accuracy of 52.515%. This time, we will continue to explore the NSynth dataset, with yet another type of the neural network models - RNNs. Specifically, we will look into how LSTM and BLSTM perform on this dataset.

Section Training will contain the graphical representations of the network architectures used in both models, and the hyperparameters used to achieve the best result. Section Result described the training result both in a textual and graphical manner. Lastly, Section Discussion will conclude our findings and thoughts on the project itself.

Training

Data processing

This model uses a regular LSTM model which takes the first second of all NSynth dataset, each of which contains 160000 music frequencies. The batch size of this model is currently set to 32, let's just look into the 1 sample from the epoch. The sample has sample has the shape of 1*16000 across 1 second. We would like to have 1* 160 samples per time step, which means there will be 16000/160 = 100-time steps in the model. Assuming we want our input size is 100, depending on whether the network has the batch first setting on or not. The input tensor will process the following shape:

1. Batch first = T
Input(batch_size * sequence_length * input_size) = Input(32 * 100 * 160)
2. Batch first = F
Input(sequence_length * batch_size * input_size) = Input(100 * 32 * 160)

Training Setup

This section contains the details regarding network architectures and hyperparameters used to achieve the best result. See the following table for comparison.

	LSTM	BLSTM
Parameters used	LEARN_RATE = 0.0008 BATCH_SIZE = 32 EPOCHS = 20 <i># Network Parameters</i> INPUT_SIZE = 160 HIDDEN_SIZE = 128 NUM_LAYERS = 3	LEARN_RATE = 0.0008 BATCH_SIZE = 32 EPOCHS = 20 <i># Network Parameters</i> INPUT_SIZE = 160 HIDDEN_SIZE = 128 NUM_LAYERS = 3 bidirectional = True

Network architecture

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        # if use nn.RNN(), it hardly learns
        self.rnn = nn.LSTM(
            input_size=INPUT_SIZE,
            hidden_size=HIDDEN_SIZE,
            num_layers=NUM_LAYERS,
            batch_first=True,
        )

        self.out = nn.Linear(HIDDEN_SIZE, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        h0 = torch.zeros(NUM_LAYERS, x.size(0), HIDDEN_SIZE).to(device)
        c0 = torch.zeros(NUM_LAYERS, x.size(0), HIDDEN_SIZE).to(device)

        # None represents zero initial hidden state
        r_out, (h_n, h_c) = self.rnn(x, (h0, c0))

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        #out = F.softmax(out)
        return out
```

```
class BiRNN(nn.Module):
    def __init__(self):
        super(BiRNN, self).__init__()

        # if use nn.RNN(), it hardly learns
        self.rnn = nn.LSTM(
            input_size=INPUT_SIZE,
            hidden_size=HIDDEN_SIZE,
            num_layers=NUM_LAYERS,
            bidirectional=True,
            batch_first=True,
        )

        self.out = nn.Linear(HIDDEN_SIZE*2, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        h0 = torch.zeros(NUM_LAYERS*2, x.size(0), HIDDEN_SIZE).to(device)
        c0 = torch.zeros(NUM_LAYERS*2, x.size(0), HIDDEN_SIZE).to(device)

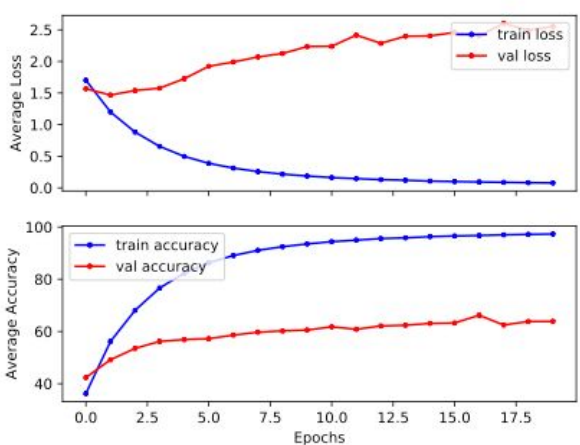
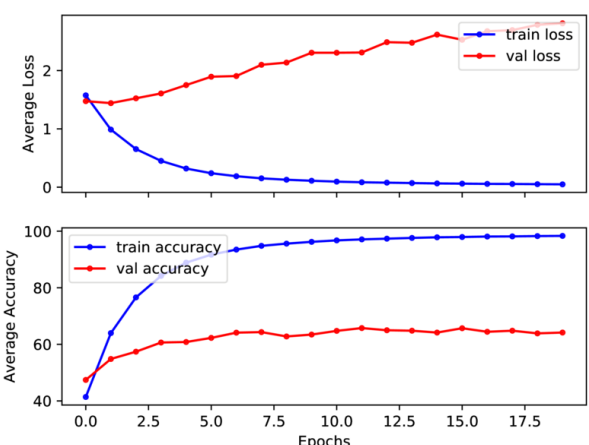
        # None represents zero initial hidden state
        r_out, (h_n, h_c) = self.rnn(x, (h0, c0))

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out
```

The RNN network architectures shown above are heavily inspired by Zhou's Pytorch Tutorial sites (Zhou, 2018).

Result

This section will discuss the training result with the aid of figures and descriptions. Similar to the training section, we will separately discuss the LSTM and BLSTM in the following subsections.

	LSTM	BLSTM
Loss over epoch	<p>LR = 0.0008, Momentum = 0.8, Epochs = 20</p> 	<p>LR = 0.0008, Epochs = 20</p> 
<p>Both LSTM and BLSTM started overfitting very quickly. Which is consistent with our findings in project 2. Which lead us to believe that a certain attribute within NSynth dataset makes it prone to overfitting.</p> <p>Strangely, we adopted the early dropping method, and realized that, even though the best model, at around epoch 2, should have better test accuracy than the last model. We found that the last models have better accuracy. So we are not sure if this is due to the way we visualized the validation loss.</p>		

If the model is indeed overfitted, given the time, we would like to try and increase the batch size or adapt regularization techniques to improve it.

Additionally, before settling on the 3-layer LSTM or BLSTM, we have tried using a 2-layer one. Surely, the 3-layer models have a slight increase in accuracy, at about 3%.

Last but not least, we had experimented with both SGD and Adam optimizers for the models, and realized that Adam is much better in this case. It allows for a dynamic momentum in proportion to the gradient magnitude, thus descending closer to the local/global minima. This suggests that NSynth loss gradient have some steep “valleys”.

1-D Visualizat ion

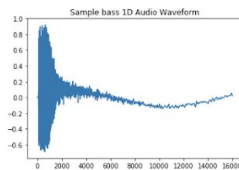


Fig. 2. Bass

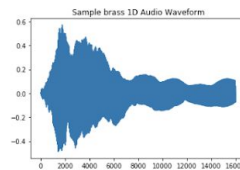


Fig. 3. Brass

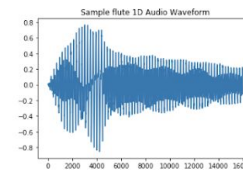


Fig. 4. Flute

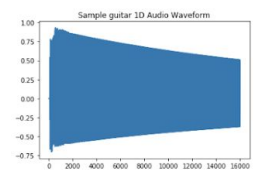


Fig. 5. Guitar

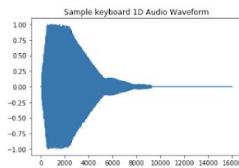


Fig. 6. Keyboard

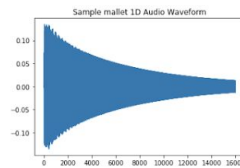


Fig. 7. Mallet

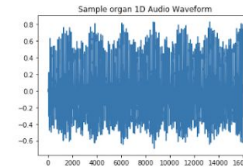


Fig. 8. Organ

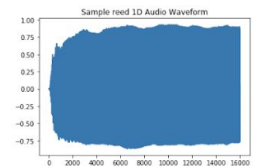


Fig. 9. Reed

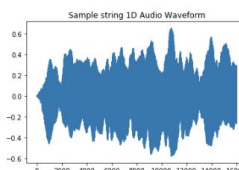


Fig. 10. String

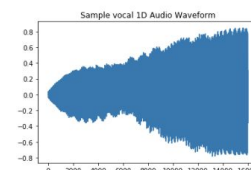
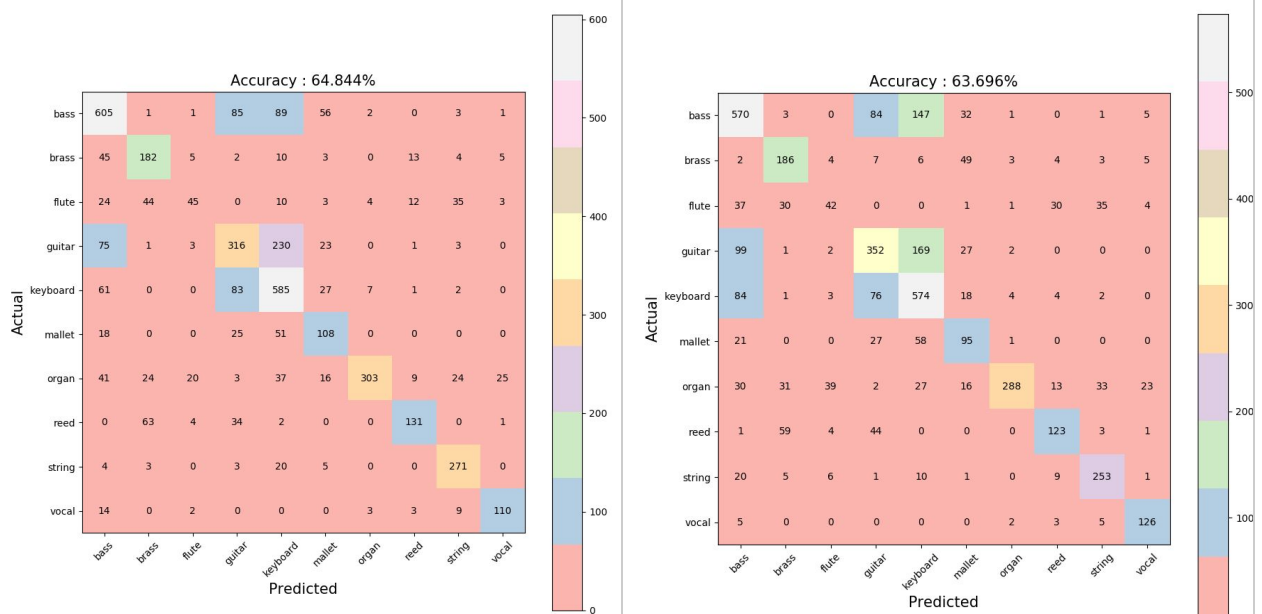


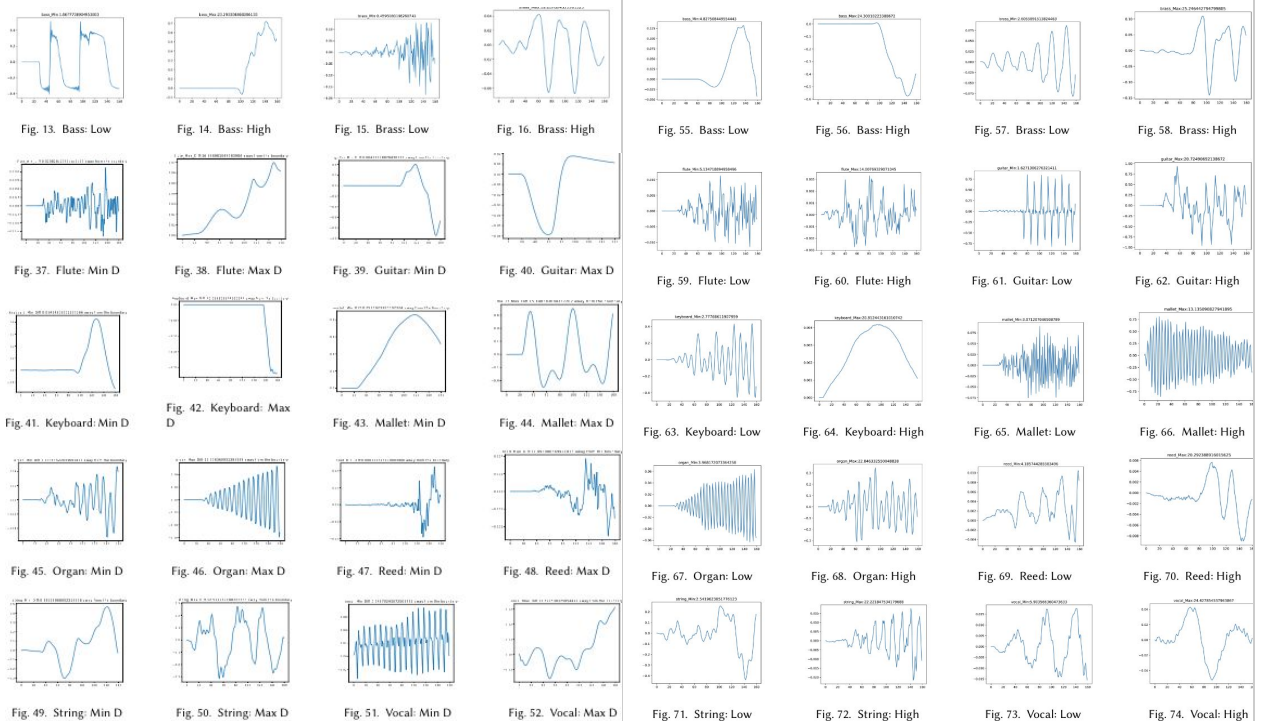
Fig. 11. Vocal

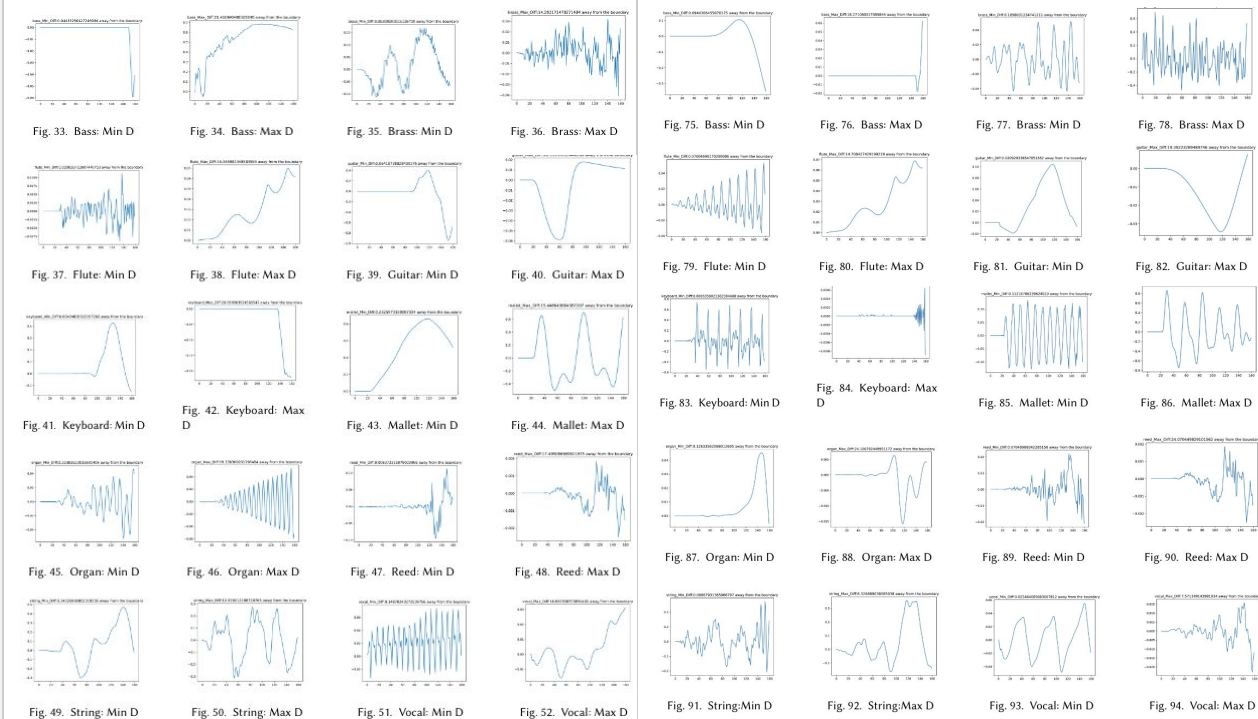
Confusion matrix



The LSTM achieved a higher accuracy than BLSTM. LSTM is at 64.844%, while BLSTM is at 63.696%. LSTM classified class String, Organ, and Bass more correctly when compared to the other classes. As of BLSTM, it classified class Guitar and vocal more correctly when compared to the other classes. The differences between the rest of the classes classification are quite similar within the two models. Based on the figure above, we can see that Both models seem to have a bit of a problem classifying bass, flute, guitar, and keyboard. BLSTM did a little worse in the class keyboard. We presume that this is caused by the BLSTM learning wrong correlations between the end of a keyboard sample and the beginning of a bass sample (bluesummers, 2017), since both classes have a small fade-in and fade-out transition.

Correct Class probability very high or very low



	<p><i>Note: if the above images are too small to view clearly, please see the original image in folder Test_Result.</i></p>	
<p>Incorrect class close to and not close to the decision boundary</p>	 <p>Fig. 33. Bass: Min D Fig. 34. Bass: Max D Fig. 35. Brass: Min D Fig. 36. Brass: Max D Fig. 75. Bass: Min D Fig. 76. Bass: Max D Fig. 77. Brass: Min D Fig. 78. Brass: Max D</p> <p>Fig. 37. Flute: Min D Fig. 38. Flute: Max D Fig. 39. Guitar: Min D Fig. 40. Guitar: Max D Fig. 79. Flute: Min D Fig. 80. Flute: Max D Fig. 81. Guitar: Min D Fig. 82. Guitar: Max D</p> <p>Fig. 41. Keyboard: Min D Fig. 42. Keyboard: Max D Fig. 43. Mallet: Min D Fig. 44. Mallet: Max D Fig. 83. Keyboard: Min D Fig. 84. Keyboard: Max D Fig. 85. Mallet: Min D Fig. 86. Mallet: Max D</p> <p>Fig. 45. Organ: Min D Fig. 46. Organ: Max D Fig. 47. Reed: Min D Fig. 48. Reed: Max D Fig. 87. Organ: Min D Fig. 88. Organ: Max D Fig. 89. Reed: Min D Fig. 90. Reed: Max D</p> <p>Fig. 49. String: Min D Fig. 50. String: Max D Fig. 51. Vocal: Min D Fig. 52. Vocal: Max D Fig. 91. String: Min D Fig. 92. String: Max D Fig. 93. Vocal: Min D Fig. 94. Vocal: Max D</p> <p><i>Note: if the above images are too small to view clearly, please see the original image in folder Test_Result.</i></p>	
<p>Training and testing times</p>	<p>4 am to 5:53 pm - 13:53 hours</p>	<p>4 am to 8:56 pm - 16:56 hours</p>
	<p>Granted we weren't able to use GPU to accelerate the training process, but our experience certainly shows that BLSTM takes a lot longer to train. Which make sense given its' complexity.</p>	

Discussion

What does the visualization tell us about the behavior of the LSTM and BLSTM classifiers, and their ability to discriminate between the instrument families?

Both LSTM and BLSTM are able to classify instrument families with lesser variety pretty well, such as String, Vocal, and so on. However, they struggle with similar looking samples, such as keyboard and brass. BLSTM especially struggle with samples that might have a symmetrical shape. The keyboard class is one such example, since the ending of it looks similar, in some case, as the beginning of a bass sample.

How do the LSTM and BLSTM compare in terms of speed and accuracy compared to your best network from project1?

BLSTM is a lot slower compared to LSTM, due to the additional complexity. BLSTM achieved 63.696%, which is about 1% less than LSTM, that achieved 64.844%. The difference is not significant but does prove the point that in machine learning, or a less complicated model sometimes performs better than a complex model. The right fit is the key.

What in your RNN network designs seems to have led to the results obtained? How might they be changed to improve the results?

Having an Adam optimizer in the gradient descent process definitely helped the model to learn better. Also, adding additional layers within the RNNs (from 2 layers to 3 layers) helped to improve the accuracy for about 3%. We read online that Tanh activation function is often used in RNNs, but adding it to the model did not change the result much. The loss over epoch figure does suggest that the model had overfitted, so given the opportunity, we'd like to adopt weight decay, bigger batch size to improve it. Also, having the entire 4 seconds as sample might yield a better result.

Additional Comments/Summary

In this project, we had to first familiarize ourselves with the RNNs, and how to reshape the tensors to match the network. Then we tweaked the parameters to improve its accuracy. Notably, changing the optimizer for gradient descent from SGD to Adam made the biggest difference. Which can be explained by the fact that Adam optimizer allows for a dynamic momentum in proportion to the gradient magnitude. Hence, when a model had steep slopes, Adam optimizer helps the model to descend further into the valley, aka the local or global minima.

Overall this project allowed us to have a better understanding of how RNNs such as LSTM and BLSTM work both theoretically and practically. When compared to project 1, the model did achieve a better accuracy. It theoretically makes sense since music notes are time-dependent data, naturally, RNNs should perform better. However, others have achieved similar or better accuracy in their convolutional models. It is possible that we haven't explored all the best parameter to really optimize the LSTM or the BLSTM models. Not to mention, we are currently studying 1 second of the audio file rather than 4. In the future, we'd like to look at increasing the number of layers and incorporate more seconds of the music notes.

Bibliography

bluesummers. (2017, 5 20). *What's the difference between a bidirectional LSTM and an LSTM?* . Retrieved from StackOverflow:

<https://stackoverflow.com/questions/43035827/whats-the-difference-between-a-bidirectional-lstm-and-an-lstm/44082853>

Zhou, M. (2018, 11 12). *PyTorch-Tutorial*. Retrieved from github:

https://github.com/MorvanZhou/PyTorch-Tutorial/blob/master/tutorial-contents/402_RNN_classifier.py