Music generating with recurrent neural networks

Chao Long
c3long@eng.ucsd.edu

Eli Uc
euc@eng.ucsd.edu

Haakon Hukkelaas
haakohu@stud.ntnu.no

Peter Greer
pbgreerb@eng.ucsd.edu

Stian Rikstad Hanssen
hanssen.stian@gmail.com

Abstract

In previous assignment we have looked at standard neural networks and convolutional networks, where these networks are constrained by a fixed input size and output size. The exciting part and benefit of recurrent networks is their ability to operate over a arbitrary sequence of networks. This can be sequences in the input, output or over both. Throughout this assignment we will look at the power of the recurrent net, specifically LSTM units, to process and generate data in an sequential manner.

1 Introduction

Recurrent Neural Networks(RNN) offers to include temporal information about the world in it's prediction process[6]. This feature is exploited in a variety of applications for it's ability to produce intelligent behavior based on past events. However, these networks frequently suffer from the vanishing weight problem during back propagation. The Long Short Term Memory networks solve this and will be explored in this assignment.[5]

In this assignment we will explore the power of Recurrent Neural Networks by implementing a network to generate music files with the use of LSTM units. The dataset consists of 1124 unique songs in ABC format [1], where the mean length of each song is 433 characters. We will start with a simple 1-layer network with 100 hidden units, then experiment with the network structure, optimizers, and the use of dropout.

2 Music generation

2.1 Discussion

With the help of standard Pytorch tutorials[4] we first implemented an LSTM network using 100 hidden units, 1 hidden layer and a set batch size of 15. At first for training we used the strategy of picking subsequences at random from the training set with a dynamic sequence length. However, after some poorly formatted generations we switched and found better results only shuffling the training data once during the train-validation split and using a fixed sequence length of 25. With this setup a single epoch for us was an entire run through the training set, training on sequences from the start to finish of our training set in order. With this setup we found our first success in generating music files.

At first we evaluated the loss for our data at the end of every epoch, but we noticed that we ended up spending a significant amount of training time on loss evaluation to the point it was limiting our ability to test. Our solution to this was to evaluate loss on our data only every 10 epochs. We found that this still allowed us to converge before overfitting became a serious issue and significantly cut

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

down on our networks training time, even though we usually trained for 10-20 more epochs than necessary. To detect convergence we used early stopping when we saw a consecutive increase in the average validation loss two checks in a row.

When we started testing we used the Pytorch Adam optimizer, but eventually switched to standard gradient descent. We found that SGD paired with Nesterov momentum allowed us to converge much quicker than our previous setup and we felt more comfortable with the functionality of the gradient descent algorithm. After a bit of testing we landed on the optimizer parameters of a 0.01 learning rate with a 0.9 momentum value.

Upon increasing the number of hidden layers to 2 we expected to see a significant decrease in the converged average loss, but this was not the case. We tried multiple runs, but based on our graphs we could see very clearly that our LSTM model consistently converged with an average validation loss of $\sim$2.1 and accuracy of $\sim$40% with no meaningful variation. Most importantly there was no distinguishable change in the quality of our generated music samples. We viewed this outcome as evidence that the patterns present in our input data were mostly linear in nature and for this reason we gave up on trying to increase the number of hidden layers further. Eventually we did perform a 3 hidden layer test but found no distinguishable improvement as expected.

The last modification we tried was a switch from LSTM to using GRU and we immediately saw an improvement in both average loss and accuracy. The validation average loss hit 2.0 for the first time in our tests, and accuracy for validation increased by $\sim$2% on average. We looked for evidence that GRU was giving us faster convergence than LSTM, which had been suggested to us as what to expect, but found very little evidence to support that in our results and any change in convergence time was negligible. Still, due to the improvements to loss and accuracy we decided to stick with GRU for our final network.

2.2 Hyperparameters and model details

Our final model structure was the following. An expected input tensor of shape (batch size x sequence length x vocab size). We treated the "<start>" and "<end>" strings as special characters, and a total of 93 unique characters appeared in the input text so our vocab size was 95. The input was taken in as one hot encoding, meaning each of the batch size x sequence length vectors in the tensor was filled with nothing but zeros and 1 one to denote what character was sent in. For all our tests we used a batch size of 15 and sequence length of 25. After our failed attempts at dynamic sequence length we decided not to mess with the sequence length too much and we ran into performance issues when making the batch size too large or too small. The input was fed into a single layer of 100 GRU hidden units. After testing with more or units or layers we saw no benefit in changing these values. Lastly our gradient descent optimizer used a learning rate of 0.01 with a Nesterov momentum value of 0.9.

2.3 Music Samples

```
X:63
T:Lasse Here's af the Stainn (Coodorair
R:polka
H:Aerchiag also #67
K:G
A2FA AGE2|~D3E FD~E2|d>EA/E -190
M:6/8
D2Dd ecdB | AG~D2 dBdE |: (3FAG AFDD |
efbf gage|dfgf gfec|dBAF F2EF|GFED GEDE|GFE F2D|EFEFEFE|: FABA B3c2 | Bcde f2dc | AAce f2ef ||
```

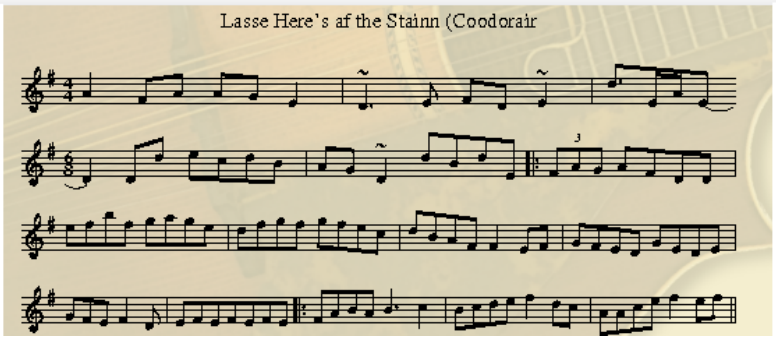

Figure 1: T1-1

```
X:41
T:High Stoup: Ive merl of Aley 't Mornpy pas Molsoo, #84
Z:id:hn-reel-46
M:C|
K:G
GEDE FABA|BFGE GcEE|DGBd eABA|FDEF EDGB|AFAB AFFA|B^FFE FEDF|
ABcB c>EFE|~F3F FAAF|F2DF ABcd|1 F2GE DD (3DDD :|2 DGFD FEDE|
GGEG EDGB|
DGAB BABd||
```

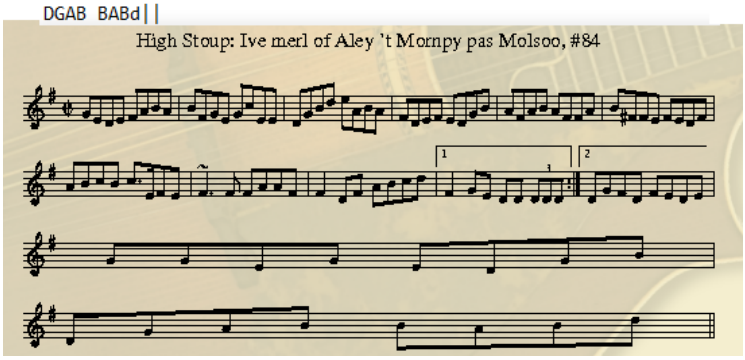

Figure 2: T1-2

Figure 3: Music generated with temperature = 1

```
X:97
T:Mary (1903-1998.
Z:id:hn-reel-52
M:C|
K:D
B2 gB (3EFE|F2AF GEDB,|
G2Bd BAGB|ABAG FDDF|EDEF GEDE||
|:GEGB GEEG|ABcA BA~A2|BAGE FEDC|
GB~B/ BAGA|BA~A2 eAFA|B2ef g2af|g2ed BA~B2|defg edBA|GE~E2 FG~B2|e2fe dcBA|GBAG FDD2:|
|: G2AB cBAG | FGAB cded | edcB ABcd | ecAG FEDE | FEFD DEFD | AGFG AGED :|
```

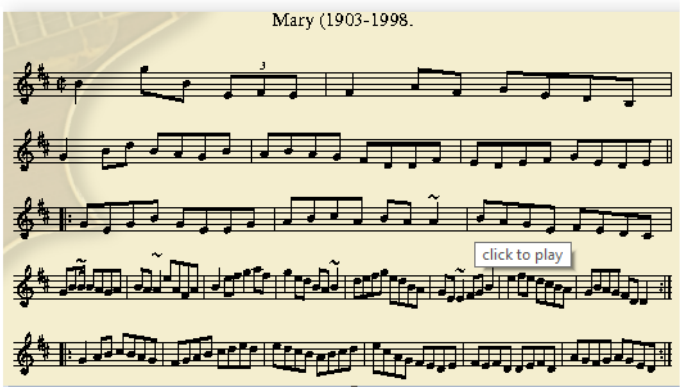

Figure 4: T0.5-1

```
X:29
T:Banrinan Tand of Casion al Bar
T:March in G, #14
Z:id:hn-reel-39
M:C|
K:D
FA AB/A/|GA BA||
|:fg fe/f/|gfed BAFA||
P:Variations:
A2AF ABcA|BAGE FDD2:|
|:BA~A2 BAFA|1 BA~A2 Bege | d2cd edcc | d2dB G2ed |
(3cde ~f3e | fafe defg | a2ag afeA | dBAF GED2 | GEAB BBBB | ABGB BABc | B2Bc BAGB :|2 GABA GBdB||
```

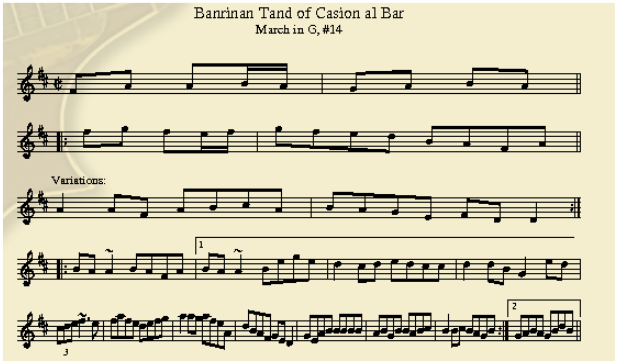

Figure 5: T0.5-2

Figure 6: Music generated with temperature = 0.5

```
:64
K:C"aledoz fapehrar.
O:Johy A EC'C2Do,>M
P:csremoban's.! w:ikD #1038
Z:Daveas
C:Orines
N:bqunpe's&BTeci'adesc.stap
DFryan,13
D:CEr1Bd efgg|a2be pedLetUre.m. HfMo, Wamd.fr MicpolecIxthF,G,K:EBEatAiphet plSeen
Z:id:hnarquieB\
O:Nancyr-log, Pork+!:6/8=1805)
Z:Fk|quD-ej
M:C|
K:D
(8D CG+,|f6^G4|4FAdc Bdgd:|cEezeg]~g>G~B/2A|1,F7
D|
G3/Ad/B/c/d_~c>B^A>_GE>DF:|DO
```

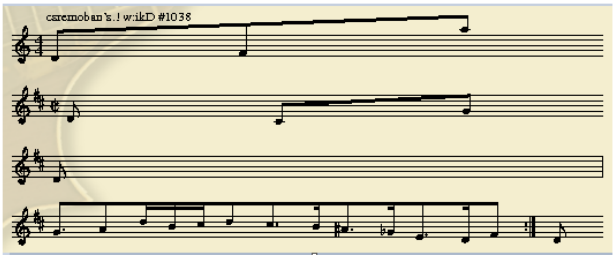

Figure 7: T2-1

```
X:6m
=foa t: ~fdHe)!fif.bg.c.8|=cBAG B_^^FGAG|Z>CoVes GnEylet.c. Bkonir-su
K:en:va AQ)
R:polka-810501912029
S:Hnay H\'s part,
T:i^)k: ir Go Snebe, plastgare, #7
W:batto Saea
D:Bit-Horvo:eatl.dhre.d B.Tr
Z:ile|
xas En\*
M:C|
K:G
ED B, Gqhe Staillan _at"miGMish sombsbo wximer #76U.
H:rifee/rapk#.
fan:tyduje/2|
```

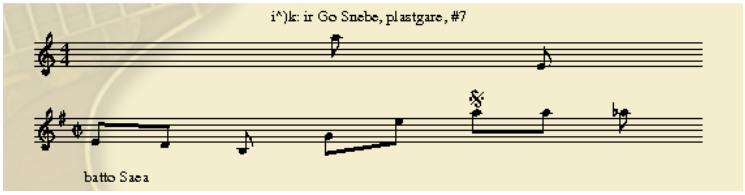

Figure 8: T2-2

Figure 9: Music generated with temperature = 2

2.4 Plots

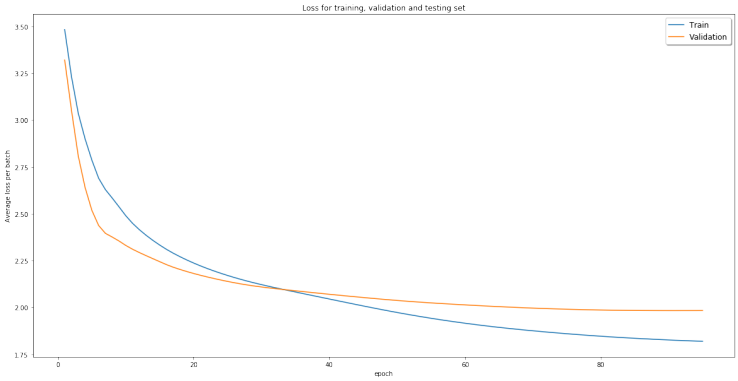

Figure 10: Cross entropy loss of our best model with loss calculated after every epoch

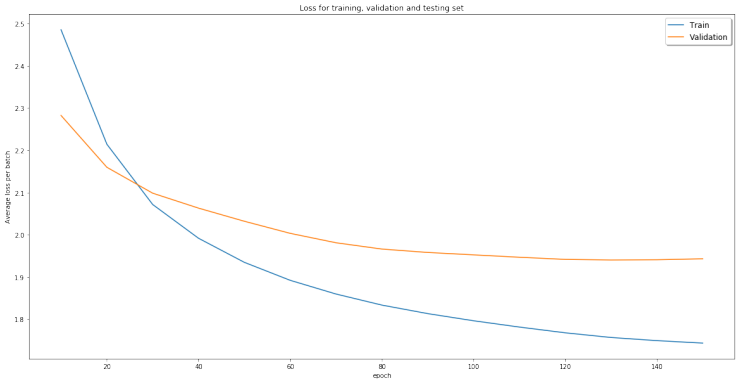

Figure 11: Cross entropy loss of our best model with loss calculated after every 10 epochs

2.4.1 Discussion

The plots in figures 10 and 11 represent the average loss calculated per batch of strings, which in this case was 375 characters. As we can see for the first 30 or so epochs the validation loss was slightly lower than training, but after that point the RNN began to recognize patterns specific to the training set better than those in the validation set. This is unsurprising due to the nature of training and overfitting, and is exactly what we predicted what we would see. In both these figures as well as all our other tests the network tended to reach peak generalization around 90 epochs at which point validation loss flatlines until our early stopping condition forces convergence.

We added in two Loss plots so that we could display our reasoning behind increasing the number of epochs between validation check for early stopping. As we can see both figures 10 and 11 converged to roughly the same results. A validation loss of 2.0 and a training loss of 1.75. This is to be expected because both graphs plot the exact same training setup, and even though we shuffled our input data into training and validation randomly we should still expect a very similar loss curve between tests. The two noticeable differences are that Figure 10 has a much steeper drop off and converges at epoch 93, where Figure 11 converges at epoch 150. The steep drop-off can easily be explained by the fact that 11 does not actually track validation loss until the 10th epoch and therefore it misses the initial sharp loss in validation error shown in Figure 10. The longer convergence time can be explained by the fact that our early stopping condition was tested 10 times as ofter for Figure 10, and we can see that both plots had effectively plateaued by the 90th epoch. For these reasons we decided increasing the number of epochs per validation check to 10 was acceptable to speed up future tests.

3 Changing number of hidden units

3.1 Plots

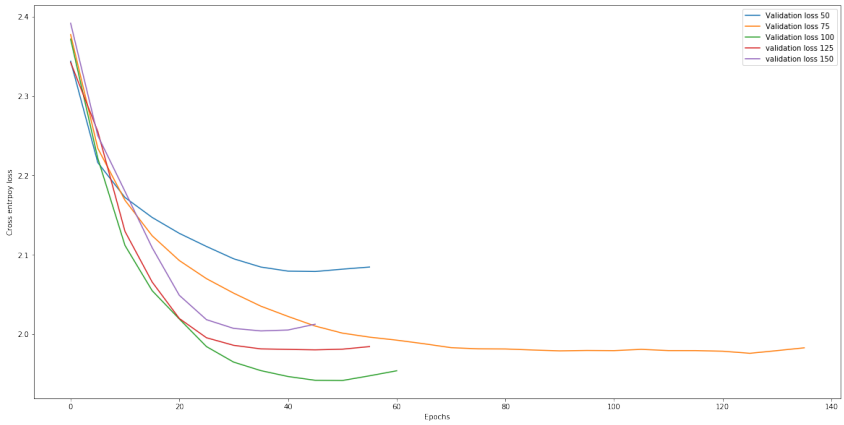

Figure 12: Validation loss for the RNN with one hidden layer with 50,75,100 and 125 hidden units.

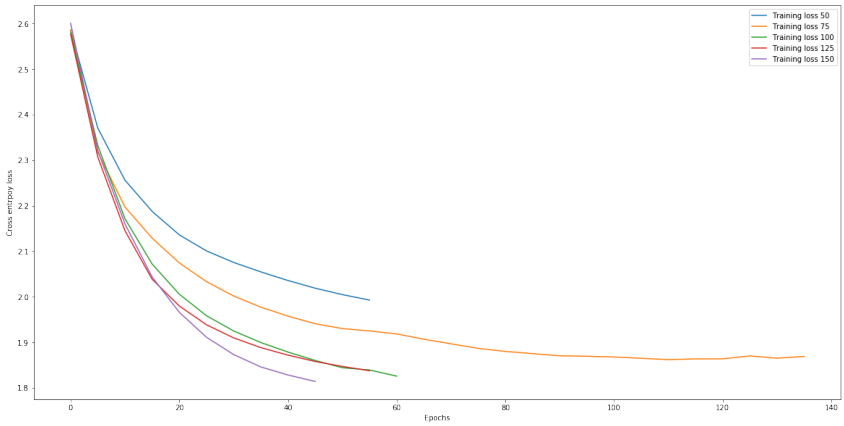

Figure 13: Training loss for the RNN with one hidden layer with 50, 75, 100 and 125 hidden units.

3.2 Discussion

Figure 12 and Figure 13 shows the validation and training loss for four different models with one LSTM layer with 50, 75, 100 and 125 hidden units. One thing to observe from the graph is that the model with 100 hidden units generalizes best, where it has the best validation loss in the end. We notice that the model with 125 hidden units has about the same training loss as the unit with 100 hidden units, however a significantly higher validation loss than the model with 100 units. This is a clear sign of overfitting, and can be a sign that the single layer with 125 hidden units is a too complex model for our problem. This is also confirmed by noting that the model with 150 hidden units is clearly performing better in terms of training loss, however is performing worse then the model with both 125 and 100 units.

Also an interesting find is that the model with 75 hidden units has almost double the convergence rate as the other models. This is with the same hyperparameters, and the only change is the amount of hidden units. We did three consecutive runs with 75 hidden units and observed the same result each time.

4 Dropout

By listening to generated music for each dropout model, we cannot get much more differences. However, from Figure 15 we can see it costs more epochs to converge while we use dropout methods. Thus the dropout methods decrease the training speed here. This is reasonable since dropout method increases the complexity of training by randomly choosing hidden units to be marked out in every epoch. We can also observe the model with dropout(p=0.1) needs more epochs(190) to converge than p=0.2 and p=0.3. Since we just tried three different dropout methods, we cannot get more general result.
From Figure 15 we can see the original model without dropout leads less validation loss and dropout methods lead more oscillations since the curve for our original model is more smooth. Thus dropout does not improve the result here.

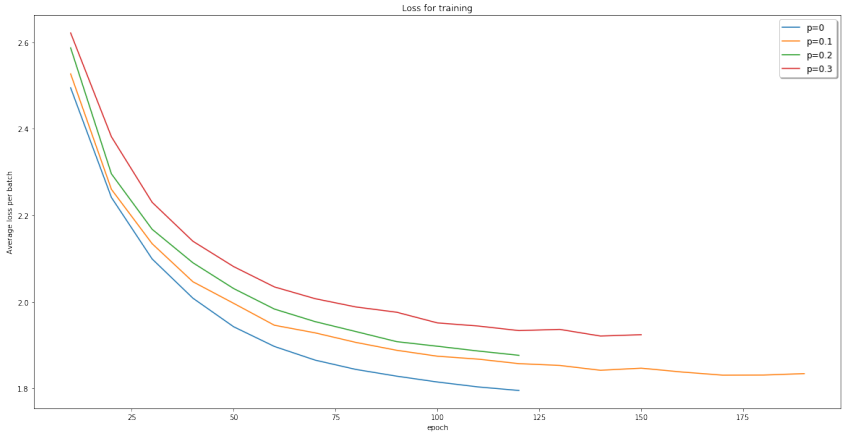

Figure 14: Training loss for dropout

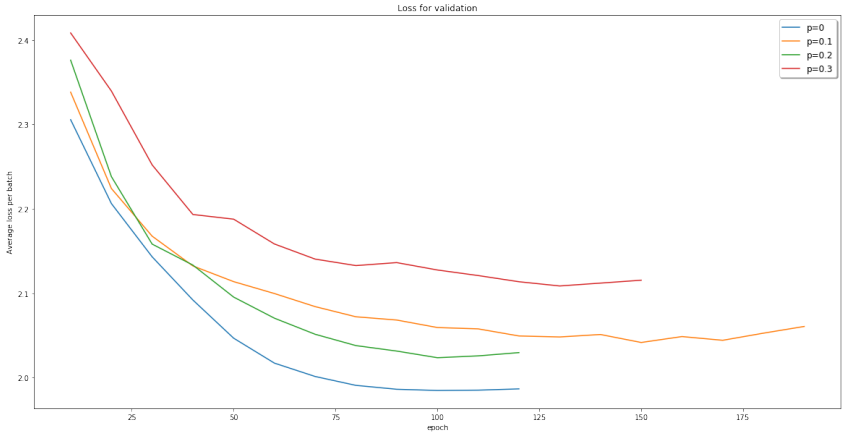

Figure 15: Validation loss for dropout

5 Optimizer exploration [5pts]

To further evaluate the network, different optimizers can be used in the model. The optimizers SGD, Adagrad and RMSProp will be tested and performance by loss is evaluated.

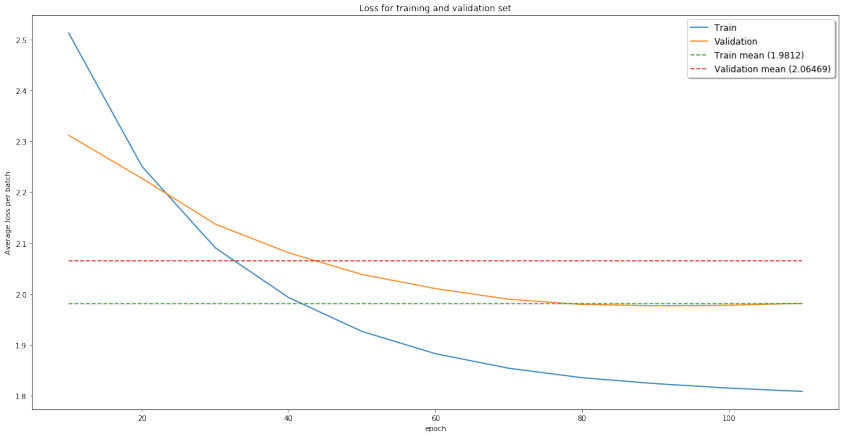

Figure 16: Training and validation loss for SGD

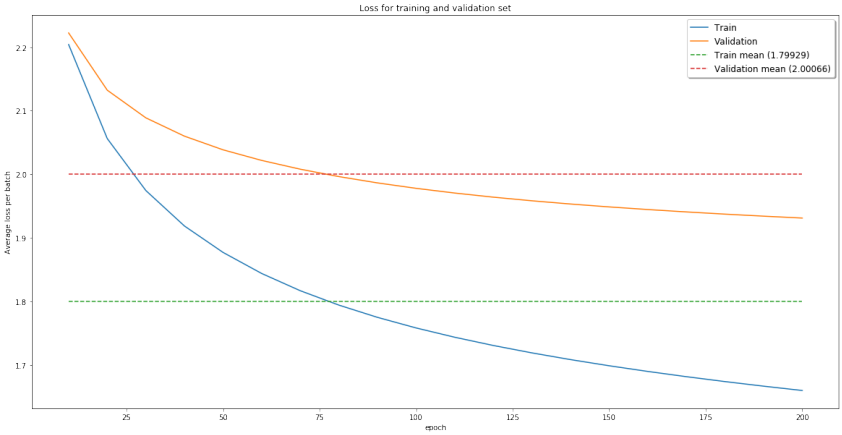

Figure 17: Training and validation loss for Adagrad

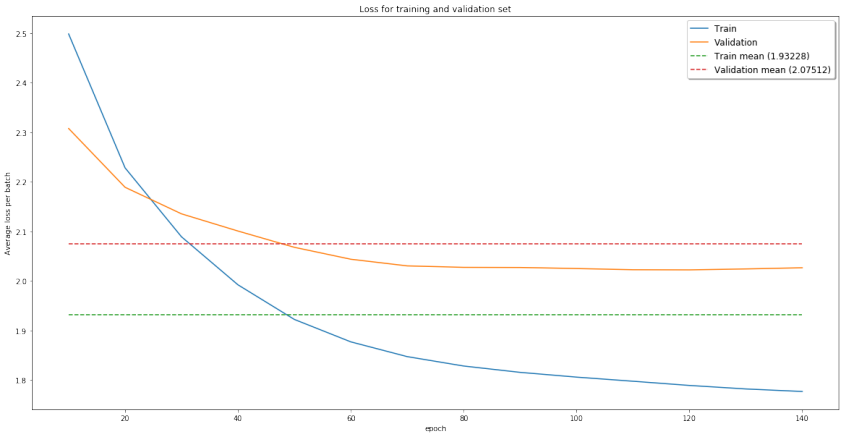

Figure 18: Training and validation loss for RMSProp

As seen in figure 16, 17 and 18, the three optimizers perform fairly similar. Adagrad performs the best both in training and validation based on the mean. It also achieves the lowest loss with 1.65950656 and 1.9310095 for training and validation respectively. Though RMSProp achieves

lower performance in these tests, an advantage of this optimizer is that it converges much quicker than Adagrad. RMSProp is stopped at 140 epochs, while Adagrad never achieved early stopping and ended at 200 epoch, the set max. However, RMSProp is still slower than SGD, which we favored for its quick learning.

It may be possible a different set of parameters could achieve different results. In this comparison both Adagrad and RMSProp use PyTorch's default parameters [2][3]. SGM have the same parameters used in section 2.3.

Adagrad may do best because the learning rate is adapted with component-wise gradient. Considering the variation in text when it comes to length, order and how uniform the text is, such adaptive learning rate may have an advantage.

6 Feature evaluation [5pts]

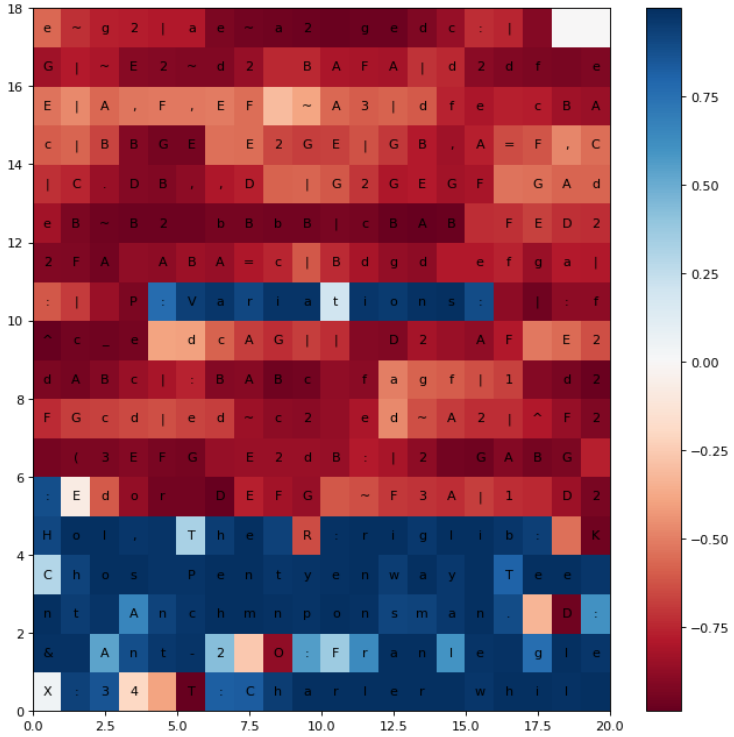

Figure 19: Neuron heat map 1

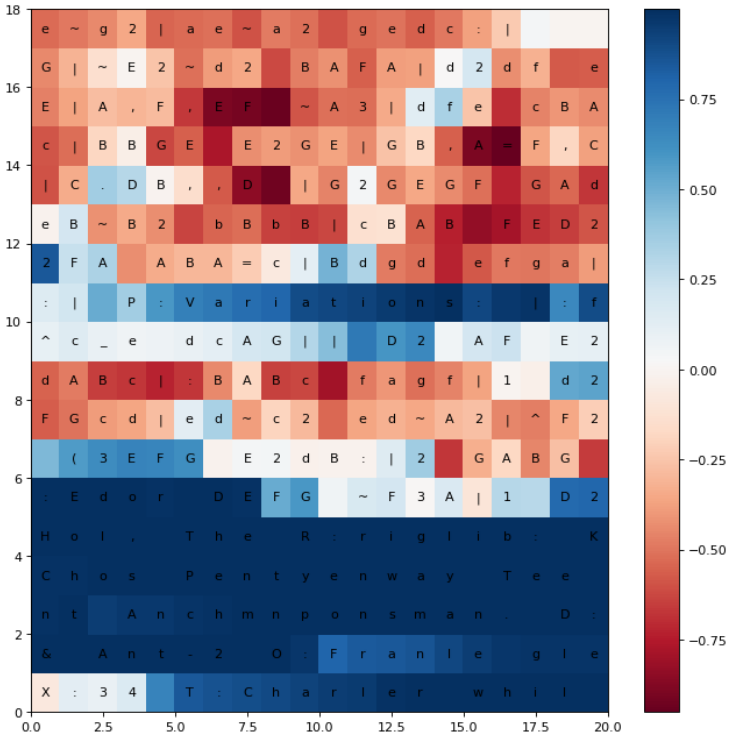

Figure 20: Neuron heat map 2

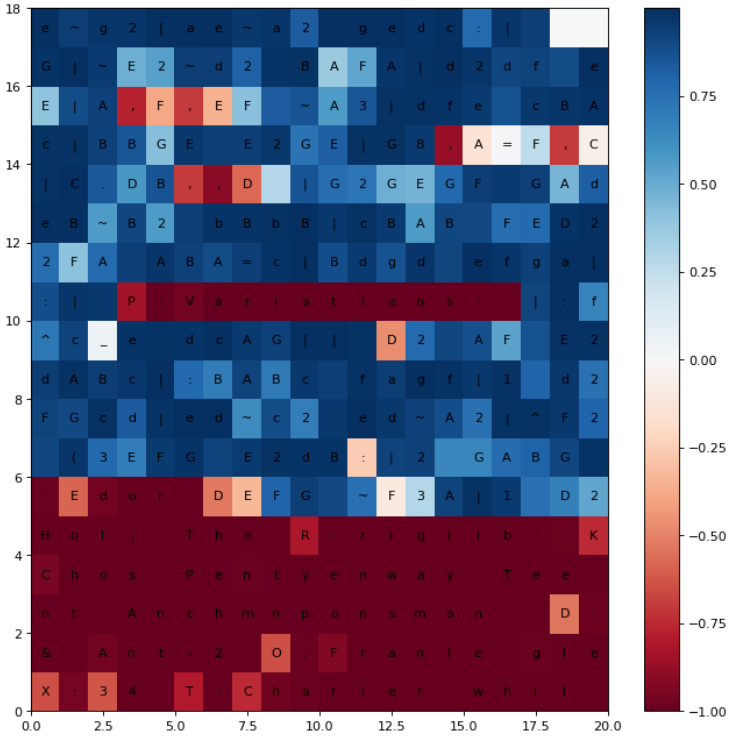

Figure 21: Neuron heat map 3

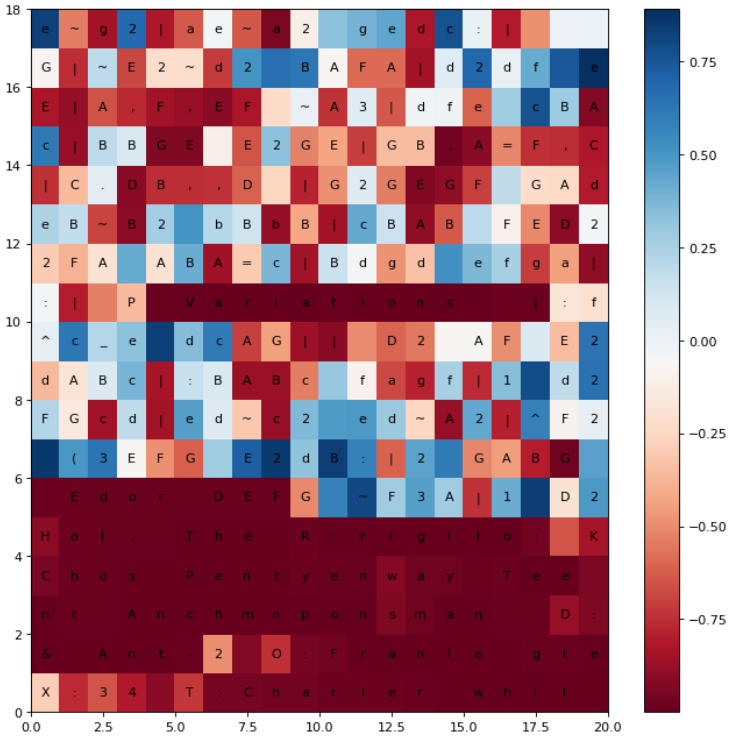

Figure 22: Neuron heat map 4

The heat maps above all came from the same music generated sample and display some interesting information about the networks understanding of the file it generated. Figure 19 and Figure 20 both clearly display a positive association with the header portion of the file, while Figure 21 and Figure 22 show the opposite. This is important to us, because it shows that our network learned to associate neuron values with the current section of the network and these heat maps visualize how it transitions from one state to another. In all 4 figures we can see on the bottom left that when the network starts reading the string it doesn't have the strong assumption that it is in the header, but after reading the line "X:34" the colors for the rest of the header become almost solidly blue for Figure 20 and solidly red for figures 21 and 22.

Figure 19 is an exception and we can see that it likely has some relation to deciding when the file switches from the header section to the music section. We see this visualized in Figure 19 by the strings of dark blue or red heats followed by lighter color cells whenever a new line is started. These lighter color values likely represent the networks uncertainty of whether the next line will be part of the header or part of the music score. Similarly Figure 21 shows extreme confidence in the header lines with very few lightly colored cells covering the header characters, but seems to regularly show uncertainty when it sees certain characters show up in the musical score like capital letters or special characters that are also common in header lines. We can see that these sort of checks payoff with the detection of the standalone line "P:Variations". Though uncommon, header type lines are capable of appearing after the musical score has started and from what we can tell its because of neurons like Figure 19 and Figure 21 that the network is able to predict them.

7 Team member contribution

7.1 Stian Rikstad Hanssen

- Attempting to implement the basic network.
- Experimented and analyzed use of various optimizers and set up graphs to more easily execute the task.
- Contributed significantly to Final Report

7.2 Haakon Hukkelaas

- Experimented a lot with dynamic sequence length for training the recurrent network
- Experimented with different batch sizes, number of hidden units, and different ways to do batching with sequential data.
- Built a separate model than what was finally used
- Contributed significantly to Final Report

7.3 Chao Long

- Implemented simple framework for LSTM at the beginning
- Explored dropout methods
- Contributed significantly to Final Report

7.4 Peter Greer

- Worked off of Chao's work to create the model we ended up using to produce music.
- Performed various tests to optimize the model.
- Produced heat maps for feature evaluation
- Contributed significantly to Final Report

7.5 Eli Uc

- Attempted underlying modeling code
- Dropout performance testing.
- Contributed significantly to Final Report

References

[1] Abc notation. https://en.wikipedia.org/wiki/ABC_notation.

[2] Pytorch adagrad. http://pytorch.org/docs/0.1.12/_modules/torch/optim/adagrad.html.

[3] Pytorch rmsprop. http://pytorch.org/docs/master/_modules/torch/optim/rmsprop.html.

[4] Sequence models and long-short term memory networks. http://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html.

[5] Stanford cs231n lecture. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf.

[6] Understanding lstm. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.