Automated Poem Generator Using Recurrent Neural Network

Prateek Agrawal A20358932 Anshul Gupta A20369831

Abstract

We are trying to explore the unreasonable effectiveness of Recurrent Neural Networks to remember the past by building an Automated poem generator. We explore different architectures of RNNs for this problem and try to find out the one that gives us the most reasonable and accurate model. We start with a baseline model of n-grams character based language model and then train on Recurrent Neural Networks starting from complex architectures to simpler ones. We have trained our RNN for about Twelve hours on one Nvidia Graphics Processing Unit. We concluded that one hidden layered RNN architecture performs better than the model with complex architecture and gave us an accuracy of 40 percent on an unseen text data.

Introduction

When a poet writes a poem, it is based on his previous knowledge of the language and its constructs. He does not just start writing from scratch. He analyzes his prior knowledge of content and form sentences that might make sense. He is able to do this after years to familiarity with language. He needs to have the definition of poem very clear in his head and should be acquainted with his past work and that of others in the same field.

To train a machine to do so is a very difficult task. It should be able to remember everything from the past and then make sensible decision to produce human readable and understandable documentation. In this project we are training a machine to write poems similar to that of a human. To make this possible the machine should have the ability to remember a lot of data from the past. Along with remembering the data it should make decision to combine information from the past to produce poem. Solving a problem like this is very interesting because of its practical implementations. We can have machine in the future which might have the capability to tell bed time stories or write lyrics for a rock band.

We try to solve this problem by training a language model and machine learning model on huge amount of data and hope that the new generated text would be similar to what the model was trained on. For example: If we train a machine on all the poem of William Shakespeare. We hope

that the machine would generate new text that it just not random but has some affiliation to the Shakespeare's way of writing.

Background/Related Work

1. Generating Text with Recurrent Neural Networks- This paper is written by Ilya Sutskever, James Martens and Geoffrey Hinton. In this paper they demonstrate the power of RNN by training it with Hessian-Free optimizer whereas we have used AdamOptimizer with learning rate 0.01.
2. Generating Sequences with Recurrent Neural Networks-By Alex Graves. This paper is about LSTM Recurrent Neural Networks can be used to generate sequences of long range structure for predicting one data point at a time. Alex demonstrated for text (where the data are discrete) and online handwriting (where the data are real-valued). We demonstrated it for poems and limericks downloaded from the website.
3. Sequence Level Training with Recurrent Neural Networks-By Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, Wojciech Zaremba. In this paper they are doing word level training by predicting only one word ahead of time using Recurrent Neural Network whereas we are training a character level Recurrent Neural Network. For performance of this model they have used a metric called BLEU and it is not differentiable and we have used softmax loss as a performance metric.

Approach

In order to make new character generated by the machine make sense, the generated characters should form legitimate words and these words should be in a combination that forms an understandable sentence by a human. The approach that we follow here is to train a deep learning model such as neural network with multiple layers. A traditional feed forward neural network is just a nonlinear combination of the current data fed inside the network and does not have the capability to remember the past inputs to the network. For example, if we have a classification task which is dependent on the previous input as well. Traditional feed forward network does not have a clear interaction with the previous input, resulting in a miss classification.

We use Recurrent neural network to overcome this short coming of the feed forward network. A Recurrent Neural Network has a loop from one input sequence to another. This help the current input to make decision taking into account the result from the previous input. This is very important for the problem that we are trying to solve. In order to generate a new character, it should know the output of the many previous inputs. Figure 1 shows the loop in a recurrent neural network.

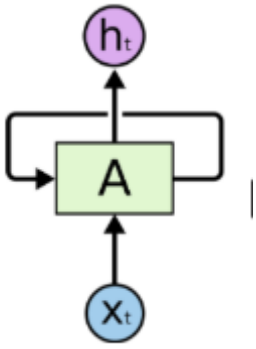

Figure 1: One Layer Recurrent Neural Network

The Vanilla neural network shown the Figure 2 has just one tanh activation gate in each cell. The hidden state output from each unrolled hidden state goes back in the first cell for next input. Theoretically these kind of networks are built to remember the past data. But practically as the Recurrent network unrolls it suffers with many problems along with gradient decay and is not capable to remembering long term dependencies. To overcome this problem of the vanilla Recurrent Neural Network we use Long Short Term Memory. This is a variant of vanilla RNNs and have multiple operation inside a cell. In addition to the hidden state it also has a cell state that helps to remember long term dependencies. Both the cell state and the hidden state from the unrolled hidden layer goes back in the start for the next input.

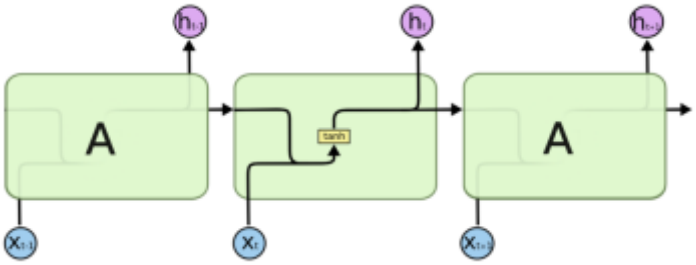

Figure 2: Unrolled Vanilla RNN

The multiple operations in a Long Short Term Memory cell is shown in Figure 3. Each LSTM cell takes input, the data from input layer, cell state and hidden state from previous cell to go through several combinations of mathematical operation to make changes in the current cell and hidden state.

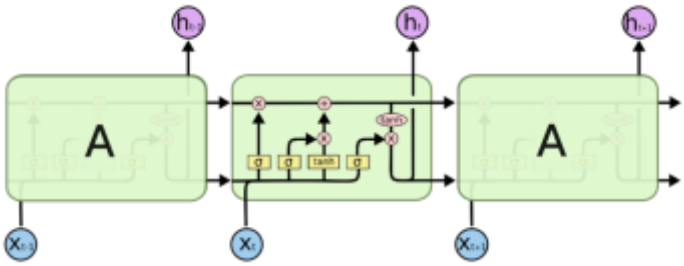

Figure 3. Unrolled LSTM cell

Following are the operations that takes place inside each LSTM cell.

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

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

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

Takes the output of hidden from previous cell and input from below.

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

Takes the output of the previous cell state and produced current cell state.

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

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

Takes the current cell output and produces current hidden cell output.

The output of each hidden layer is then multiplied by same weights and applied with the Softmax function to convert into probabilities of the next character. We then use the ground truth and minimize the negative log to the probability of the correct label.
We are using TensorFlow, a deep learning library by google to build our network. Instead of using the inbuilt API for LSTM we chose to create a LSTM cells our self to provide us with more flexibility to train a model. Figure 4 shoes the code snippet describing the operation in one LSTM cell. We unroll this LSTM cells for multiple time sequence.

```
def cell(x,cPrev,hPrev):

    x = tf.reshape(x,[nHiddenUnits,1])
    hPrev = tf.reshape(hPrev,[nHiddenUnits,1])
    cPrev = tf.reshape(cPrev,[nHiddenUnits,-1])


    ic = tf.reshape(tf.concat([hPrev,x], axis = 0),[2*nHiddenUnits,-1])
    ib = tf.reshape(biases['i'],[nHiddenUnits,-1])
    i = tf.sigmoid(tf.matmul(weights['i'],ic) + ib)

    fc = tf.reshape(tf.concat([hPrev,x], axis = 0),[2*nHiddenUnits,-1])
    fb = tf.reshape(biases['f'],[nHiddenUnits,-1])
    f = tf.sigmoid(tf.matmul(weights['f'],fc) + fb)

    oc = tf.reshape(tf.concat([hPrev,x], axis = 0),[2*nHiddenUnits,-1])
    ob = tf.reshape(biases['o'],[nHiddenUnits,-1])
    o = tf.sigmoid(tf.matmul(weights['o'],oc) + ob)

    gc = tf.reshape(tf.concat([hPrev,x], axis = 0),[2*nHiddenUnits,-1])
    gb = tf.reshape(biases['g'],[nHiddenUnits,-1])
    g = tf.tanh(tf.matmul(weights['g'],gc) + gb)


    cCurrent = tf.add(tf.multiply(f,cPrev) , tf.multiply(i,g))
    hCurrent = tf.multiply(o,tf.tanh(cCurrent))

    return cCurrent,hCurrent
```

Figure 4. Code snippet for LSTM cell

Experiments

For the purpose of this project we train different layered architecture of Recurrent Neural Network with huge amount of text data. We scrapped https://www.poemhunter.com/ from the internet to get all the poem of Williams Wentworth, William Shakespeare and some limericks. The aim is to train different text data with different architecture and compare the accuracy with a base line model and see if could do better.

The baseline model we choose is to create a n-gram character level language model on the training data. We choose the value of the n to be one as we would also be training the Recurrent Neural Network on character level. We computer the probability of every unique character to come after every unique character in the training data and use these probabilities to generate new text. To generate new data, we start with a random character and predict the next character according to the trained model. We then give this predicted character as input to this model and keep on generating new text with each input. The generate test from baseline looks like in Figure 5.

```
 fo nd?
ARes sefor:
Shor h sthid he, ftakevenor'sean't
Erondu fomof: is s yoof
LELIsusait fre d orisif d bery ceourdw, meatilye te sewresw Bueweve my G he
Yousevead,

Yot ait.
NR:
O, Eds saringucriseshe'Pr'dit tanin aint r theath thowhake hacede itho and, ain an tansoncono LI muga cofanthend Ishog blceme hesp wntousetho,
HE: m.

Wifotifombany;
Asthesthtroworanthn s buplataurengo prabl cato thee is, t d, hy s the and ano KERInoninong?
LUToloukst s me y orthe.
I ks rowsee f m rer
Gorss, hane y we.
```

Figure 5. Text sample from baseline model

To compute the accuracy of the model we take one part of text and start sampling from the first character of this text. Every time we sample a new character we check if it is equal to next character in the actual data set. Instead of giving this character as an input to the model again, we input the model with the ground truth. For every ground truth we predict a character and compare it with the next ground truth. The accuracy for the baseline model came to be around 14 percent. Also, the text generated does not corresponding to a human readable English language.

In the quest of getting better result, we try different architectures of Recurrent Neural Network starting with 3 hidden layered model with 512 nodes in each cell unrolled for 128-time sequence and move to a lesser complex architecture with one hidden layer. The generic structure shown in Figure 6 of the model is the many to many model, the only change is the number of hidden layers to form different architectures.

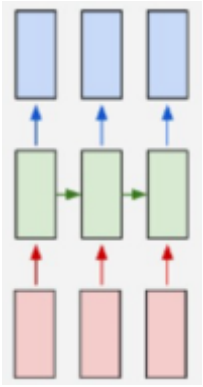

Figure 7. Many to Many RNN Architecture.

Red blocks in the figure represents the input layer and takes one hot encoding of the characters. Green represents the hidden layer were each cell is an LSTM. Red represents the output layer, this has the probabilities of the next character to be generated. If there are 50 unique characters in the text, each cell in input layer will have a vector of 50 as input. Each output block has a vector of length 50 with floating point values. Using the architecture above and operation in each LSTM described in the approach section we calculate the probabilities of the next character and use the code below to minimize the Softmax loss. We then used the Adam optimizer to minimize the loss and use gradient clipping to handle exploding and vanishing gradient.

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = results, labels = y))
optimizer = tf.train.AdamOptimizer(lr)
dVar = optimizer.compute_gradients(loss)
dVarClipped = [(tf.clip_by_value(grad, -clipValue,clipValue), var) for grad, var in dVar]
train = optimizer.apply_gradients(dVarClipped)
```

Three Hidden Layer Results

The text sampled from the 3 hidden layered architecture is shown below in Figure 8.

```
rl,lesspt dhr   :  b   c btl oni  tn, bI  s C  h bANerN nutrl,  l sc 't  r  nctrr,srr   , Hfr r  et t  rh Gs 'N  rl  ,b,ptenwsat brlllm   I lh, rrt br trrrlnn:Yrb,  l,rl  r,rc l,m   rnt
,  t ret rrwu e rb  btecrNne  , rt,g rle  n nl b0mnn rrtrt g  tnr rtNt
s trbr,t  m,lr,s s   tll' tlr bl  r d d C r rc , tesltt  GrMr r l  h serllsr hl m   cr l t:, ltderG b  rtllbsnnlmw  lllrhl' s strtreC rl tkrCpl tlrr t l trl lr d   Ctrt -: rlte   ls rttrbrm,rht
crl h' shet  bs l l shlo m   ?ri ltr  l el'tu 'atels tlcu rg'N t er '  Crl,,trr lrrle l'Cc l tr  r ,te  I T,l tN ,nr:ln "u :rblrp S  rlmnp: nNr r, Ne  r,n bfe r ,m bNptl rC  'lts  Cret Nrrrmlr s
ch 'Nh e l c   ss  Nl,tn  tr.Ut  i Cth r rtl  rll,r   C  utr ie CN m d ; t,tbclr smNt
tt   :c R r're t r rrTs
l  r rtrr rntl td bNL
irbt: et: m dcrN rr l  rr    rsr l Cr  :l0 Wrsu,l  r  n surrbl' rr NC? r   ,htr  ,: Ntr sn l  l, tNnrstC r:s :N rn
r:e   0  rlc$lsrr  rp  ,b lt r   NCl,rr 's t r tNlcSsrh:  b?ittNr  l'
lerN th,  cr,
```

Figure 8. Sampled text from 3 Hidden Layer Architecture

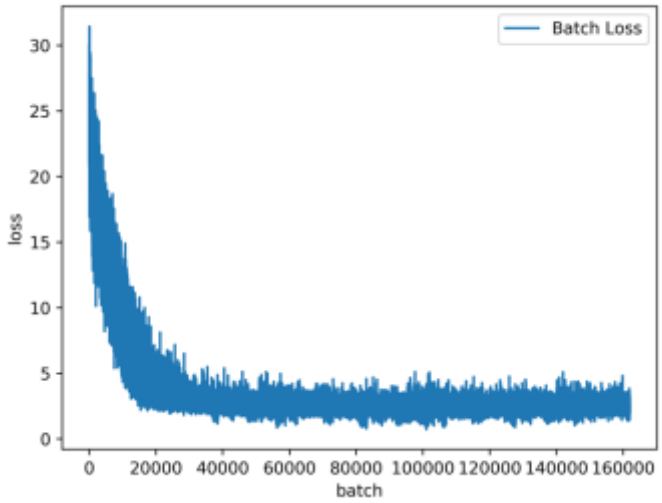

Figure 9. Batch loss from 3 Hidden layer Architecture

We can see from figure 9 that loss decreases for the first 20000 iterations and then become constant. This shows that the model does not learn anything for most part of the training. This model has an accuracy to 15.7 percent.

Two Hidden Layer Results

Text sampled from the Two Hidden Layer architecture can be shown below.

```
your the nised
ow will we Jan.

Thy bod Sill, beat the nobe my ou aby therpamer fidame to bewit me glo;
And non and ste ly ungle high arie suck.

Istrublif Warwick, fa eave, would in the by;
Rice, like,
Wh's thou brin nourst muich thinse dathavierge: dour uin farser;
We hasth ueel:
And hems ter inge agarat nawe swat mear we, ling e wedwir stralie!
To ward's forow smpod sours will of hat se she, fnock whantrak's is Hun ffuske, fineay.
CLAnd meard uis for geid oursed ars, beeigghstlen.

GLOUCESTER:
Wer EDWARD Wardereas you whr stariter, lostre benty welf a shake upeas welld, as sour be and urad;
Cliart uncle-from ksee sens I whit swisentet, quid wild t wringe brows
I Hreave
Titeant went,

Nod muetend loves to Rich afrouble ee,
And wis courtugenguent onwelss:
Ther Jontoul, aurse liou thou pin, Warwive this dlaie!
Foral no our with iflm ward he, isck, Call uspe ot ay bursaine at thin my avilcous nimer then he dou mump hay warlld-We.


FO Exly singe tall, we not thrn bless Ofoord notee me I
```

Figure 10. Text sample from Hidden 2 layer

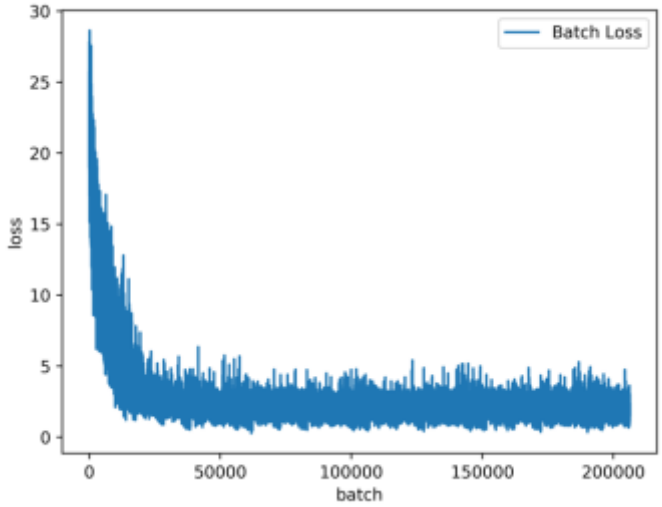

Figure 11. Batch Loss of hidden 2

We can see that the model has started learning words such as THEY AND LOVES, but still does not make sense for the combination of these words. The accuracy for the above model is around 22 percent.

One Hidden Layer Architecture

Text sampled from the One Hidden Layer Architecture are shown below.

```
-whate of thrat that Romans
As cown'd not to a eness no not been'ds
Have sport, but of read given men: have
Sprationiols
And your gread Rome, and sent. Maught hereosion no his no stitar; and am it
And you, notly not as in havals, infict's the my before powet to have befored
To thl-his pain fall your hout save
Mospirtain'd,
That I to bandity, no at highter's charius and
To urterant Marcius!

BRUTUS:
Thirds to IUS:
Hath the not thy gate?

Like arm away with rough bounts and affection. We he sat commost your actren and good spate,
And to said unlactiven and it,
What hop.

SICINIUS:
I noblehanced invesy genenen's the trink in proicit to laus hows
That than tay,
I charge, what cerns: senat: ned huries;' her;
Hear cause the loved not and mourious lions,
That yoursland.

Thir, the give mother of my end pale.

BRUTUS:
How sword mocknost water aboutfection't, whom
therefore divation!

CORIOLANUS:
Like and for your engreat obbodess and yould
He possius;
That I havoldish'd shall
able and of his th
```

Figure 12. Sampled text from One Layer Architecture.

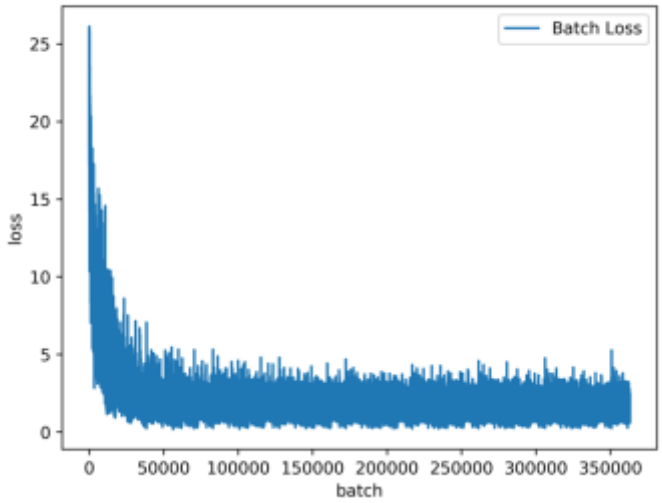

Figure 13. Batch Loss for One hidden layer

We can see from the sampled text that the model has almost all correct words in English. It has understood that poem is short in length, it has a title and there is a new line after every time a poem ends. The accuracy for these kind of model are around 40 percent. The sampled text makes far more sense than the text sample by the baseline model and has almost three times the accuracy.

Conclusion

- In spite of the general notion that complex models with more parameters will help use to fit a better model, we conclude that a Recurrent layer with one layer is more effective in generating new data.
- The reason behind this might be that in complex models with multiple layers each layer has its own cell state and hidden state. This limits the data transfer from layer to layer of the cell state resulting in a model that cannot remember data from long time dependency.
- Model with one hidden layer has the capability to remember data from infinity and hence generalize better on the unseen data. As the cell and the hidden state for each batch is initialized from the precious batch.
- In the future we can explore other architecture which might have the capability of transfer cell and hidden state between layers. This might help us to generate better samples text.
- We can also experiment with different combination of learning Rate, Clipping Value and may use a different optimizer.
- All these steps can help us to converge faster and hence help us to explore more models.
- We can also explore different sampling techniques, such as initializing the cell and hidden state with the final cell and hidden state from the training model.

References

- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- https://www.tensorflow.org/tutorials/recurrent
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://en.wikipedia.org/wiki/Recurrent_neural_network
- https://github.com/sballas8/PoetRNN
- https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/udacity