The RNN in the Hat: Generating a Dr. Seuss Picture Book

Michael Karr, *mkarr{at}stanford{dot}edu,* and Allison Tielking, *atielkin{at}stanford{dot}edu*

I. Task Definition

Dr. Seuss is one of the most popular children's book authors of all time, with over 222 million copies of his books sold worldwide. His books have been translated into more than 15 languages, and they continue to spark young people's creativity today. Recent trends in artificial intelligence have attempted to generate works of art, scripts, and books given a particular art or writing style. Given the unique structure of a Dr. Seuss book, which is full of rhymes and made-up words, we believe it will be an interesting challenge to generate a Dr. Seuss picture book, including words and potentially pictures. We will implement a basic text generator with N-grams, then advance to using a Recurrent Neural Network (RNN) to create more realistic fake books. We will also look into how to hold a rhyme structure and fake word generation, along with fake image generation if there is time.

Our initial work uses TensorFlow to train a recurrent neural network on Dr. Seuss texts. We based this initial model on the tutorial created by Andrej Karpathy, which discusses the "unreasonable effectiveness of recurrent neural networks." We will explain our model design, initial results, and next steps in this progress report.

II. Dataset

To get the text for our initial TensorFlow RNN, we scraped the text from Dr. Seuss's top 7 most popular books, saving them in separate .txt files. In our .txt files, we made sure to preserve line breaks and the original spelling of each word. We found text for these books by looking at Amazon's e-reader option in addition to finding existing .txt files for a couple of the most popular books. Our current dataset includes approximately 1300 lines of text from 7 different picture books. As we go on, we will need to increase the size of this dataset to more of Seuss's works, as we need as much of his own words as possible to train and test our RNN.

III. Approach

A. Initial Attempt: N-Grams

As our simple baseline algorithm, we generated Dr. Seuss books using an n-grams approach. This algorithm created text that vaguely followed the Seuss style by nature of reproducing patterns of his words, but it was not able to follow a rhyme scheme or make up words as is.

Scanning through Dr. Seuss's seven most popular works, the n-grams approach reproduced words in the order in which they were used, meaning that the content of the generated sentences would be strictly limited to the order of the words in the books. Given these seven works, an n-value of 3, and an iteration length of 20, the n-grams approached produced the following pseudorandom output:

> [...] you did not know what to say. our mother like this?
> we don't know. and you may. try them and you may [...]

While the English in the output was technically correct and sounded somewhat "Seussian", there was no sense of context in the generated text whatsoever. Furthermore, there were many strict limitations to the n-grams generated text. Due to the nature of Dr. Seuss's works, in particular the short lines and many chained-together rhymes, n-grams had a tendency to produce rhymes by a factor of the simplicity of the words used, rather than intentionally. In addition, there was no infrastructure in place to support any kind of syllabic calculation, nor could it split text into lines intelligently. Despite these shortcomings, we believe that the n-grams approach is an appropriate baseline because it generated text with a distinctly "Seussian" quality, but without any specific features on top of that characteristic.

After researching how effective Recurrent Neural Networks can be with TensorFlow, we began our current attempt at Dr. Seuss text generation.

B. Current Attempt: Character-Based Recurrent Neural Network

With our first attempt, we wanted to use TensorFlow to train an initial recurrent neural network. As we began this process, we realized that we did not necessarily need all of the functionality and depth that that TensorFlow could provide, so we settled for a more customized, but also more vanilla implementation of an character-based RNN.

Our RNN has several key attributes. Since it is a character level RNN, every character is encoded using a 1-of-k encoded vector. Setting up the RNN in this way allows us to use a straightforward manner, using only a zero vector with a single non-zero entry corresponding to the character alphabetical index.

Specifically, our RNN is a Long Short-Term Memory network (LSTM). This allows us to keep track of each individual memory unit and follow along in the computation.

Furthermore, while our algorithm does use gradient descent, we chose to implement Adaptive Gradient (AdaGrad) descent, as it allows us to adapt to the learning parameters and constrain the largest derivatives to act in a more normalized fashion. AdaGrad also modifies the learning with respect to the parameters and the previous gradient calculations. This method

is particularly useful with sparse vectors, fitting perfectly with our 1-of-k implementation.
AdaGrad descent is described by the equation:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t$$

In this expression, t represents our current time step, while θ represents our parameters. In addition, η is our learning rate and $\odot g_t$ represents the vector product with the objective gradient function. G_t is a $d \times d$ matrix containing the values of the sums of the squares of past gradients with respect to θ. We use ε as a smoothing term to avoid dividing by 0.
Overall, AdaGrad descent allows us to get the most of our 1-of-k sparse vector character encodings, and while it does not run quickly, it helps us achieve approximately 10% loss with 250k iterations.

IV. Evaluation

A. Evaluation Metrics

For our initial model, we developed two evaluation metrics. The first uses the Poetry Tools Python library, which we found on Github. We use this Python library to first take a poem as a .txt file and tokenize it. We also convert the poem string into a list based on every word. From there, we first use the poetry tools library to guess the rhyme scheme, meter, and stanza type of our generated poem. We had to change the given library to include the ABCB rhyme scheme along with the anapestic tetrameter, Seuss's most common meter. The library will choose the closest meter, rhyme scheme, and stanza type based on Levenshtein edit distances. This will help account for small irregularities in Seuss's line breaks and rhyme scheme. This does not account for rhymes with made-up words, however. We have 3 features for the rhymes in our generated text: isSeussianStanza, isSeussianMeter, and isSeussianRhyme. If the stanza is a quatrain, then isSeussianStanza is set to 1; otherwise, 0. If the meter is iambic trimeter, iambic tetrameter, or anapestic tetrameter, then isSeussianMeter is set to 1; otherwise, 0. Last, if the rhyme is a couplet (AABB), ABCB, or ABAB, then isSeussianRhyme is set to 1; otherwise, 0.
Our second evaluation metric checks the proportion of generated words that are sight words. We made a list of sight words from Pre-K to 2nd grade. Then, we read in these words as a set and counted whether each word in the generated poem was a sight word or not. We returned 1 if the proportion of sight words to words was between 0.6 and 0.9; otherwise, 0. We based this range on sight word proportions in Seuss's other books. For example, the *Cat in the Hat* has a 78% sight word proportion, and *Green Eggs and Ham* has 84%.
After determining these four features, we used stochastic gradient descent to determine the optimal weight assignments and classify generated text as either Seussian or not. We based our algorithm for stochastic gradient descent (SGD) off of the Sentiment assignment earlier in this quarter, but we rewrote the increment and dot product functions since we are using two lists of known, small size. We initialize our weights vector of size 4 to 4 different random float values between 0.0 and 1.0 inclusive. We then run through 400 iterations of SGD with an eta of 0.05 and update our weights vector after every run. Our initial implementation is a bit slow, so there is definitely room to optimize classification later on. After training and running SGD, we classify a test poem by performing the dot product of the weights vector with the poem's feature vector. If the dot product is greater than 1, we say the poem is Seussian. Otherwise, we say the poem is non-Seussian.
Thus, we will use these two evaluation metrics to see whether our RNN-generated Seussian poems are Seussian enough. There is definitely room to add more features for evaluation for the final project.
Here are two concrete examples of test poems that we classified. The order of the feature and weights vector values is (stanza, meter, rhyme, sight words proportion):

1. An Edgar Allen poem, *Eulalie*, which should not be Seussian.

> [...] I dwelt alone
> In a world of moan,
> And my soul was a stagnant tide,
> Till the fair and gentle Eulalie became my blushing bride-
> Till the yellow-haired young Eulalie became my smiling bride. [...]

Weights Vector: [0.33, -1.00, 1.63, 0.09]
Feature Vector: [1, 1, 0, 0]
Classification: -1 $\rightarrow$ Non-Seussian

2. A snippet of a Dr. Seuss poem, about Yertle the Turtle, that should be Seussian.

> [...] And the turtles, of course
> All the turtles are free
> As turtles and, maybe,
> all creatures should be. [...]

Weights Vector: [0.30, -0.70, 0.45, 0.99]
Feature Vector: [1, 1, 1, 1]
Classification: 1 $\rightarrow$ Seussian

Next, we tested our evaluator against text generated from our initial N-grams approach. Since n-grams generates text solely off of patterns from Dr. Seuss poems, it makes sense that this sample is classified as Seussian. Note that we added line breaks to create a rhyme scheme, but without these line breaks, this poem is not classified as Seussian since it is all one long line.

N-Grams with Added Line Breaks

> you did not know what to say.
> our mother like this?
> we don't know. and you may.
> try them and you may

Weights Vector: [0.30, -0.95, 1.59, 0.19]
Feature Vector: [1, 1, 1, 1]
Classification: 1 $\rightarrow$ Seussian

When we remove the artificial line breaks from the n-grams text, our poem is no longer Seussian.

Original N-Grams Output

> you did not know what to say. our mother like this? we dont know. and you may. try them and you may

Weights Vector: [0.48, -1.01, 1.50, 0.04]
Feature Vector: [1, 1, 0, 1]
Classification: -1 → Non-Seussian

We finished by evaluating a few different snippets of text generated with our vanilla RNN. The results were not ideal, but we have lots of potential improvements we can make to this vanilla model.
First, we want to show what happens when we train on all texts in our data set, but when we only run 1,000 iterations. Doing this results in 77.8% loss.

> oor tou son soOs the
> Wotn ThuoHde Ane winL"
> Aul mas uu dou bur kald hen be waod ou dot thandoas.
> I aom meok oe tut, in at soun toin thop tos ou the.
> Thet ohe hutk!
> Souk d u uut ou ne soup Noo bome!

Our next results come from training the RNN on just *Cat in the Hat*. We trained for 250,000 iterations and ended with 11.25% loss.

> "Oh, the things awe home Thing Ones mess is th,
> I de Nouse fan toy hat did, "Now! Hetd no not."
>
> "Have no fear.
>
> "Now lilk hopk.
> I cal.
> We saw a knoure
> On
> sn the Thould and I wais cav"

Weights Vector: [0.46, -0.98, 1.60, 0.06]
Feature Vector: [0, 1, 0, 0]
Dot Product of Feature, Weights: -0.98
Classification: -1 → Non-Seussian

Then, we trained the RNN on all the Dr. Seuss texts we had, the same training set as for our initial N-grams approach. We trained for 250,000 iterations and ended with 10.05% loss.

Sample 1 from Training on All Texts

> "comethind youn, wane fithey home is fun?
> The fish!
> And like they should,
> That in took.
> "I hook same, at is you fas.
> Saws fithet wise at thing OB.?
> At, to up they wild thin they way?
> Note fen,
> And Thow"

Weights Vector: [0.30, -0.99, 1.62, 0.15]
Dot Product of Feature, Weights:-0.68
Feature Vector: [1, 1, 0, 0]
Classification: -1 → Non-Seussian

Sample 2 from Training on All Texts

> "Then! seald the the tttep When mess," Thatrey a cat withes wet thes We hat wood ThiPld wither! I sh the ttt. So, then wop did not like i"

Weights Vector: [0.30, -0.99, 1.61, 0.12]
Feature Vector: [1, 1, 0, 0]
Dot Product of Feature, Weights: -0.69
Classification: -1 → Non-Seussian
Although our results using the RNN trained on just *Cat in the Hat* and all the Dr. Seuss texts both returned that the generated text was non-Seussian, it is promising that the dot product of the generated text's feature vector and the training weights became much less negative (drop from -0.98 to -0.68) when training with all texts.
By adding more features to our neural network, we hope to generate poems that pass our Seussian classifier.

V. Future Work

Now that we have implemented a vanilla RNN and created an initial evaluation for whether a text is Seussian or not, we have created a plan to make our neural network more robust. We want to be able to marry what we have made, our RNN and our evaluator. We currently have a 1-of-k encoded vector, so we believe that we can append more features onto the vector to account for our custom features, but we will see if we need more hardcoded constraints. With respect to the features, we would like to:

1) Check that the ends of our lines follow a certain Seussian rhyme scheme.
2) Ensure that there is a certain Seussian meter we follow.
3) Enforce that the text is in Seussian stanzas.
4) Maintain that the text contains the proper proportion of sight words

Additionally, we need to figure out how to store the current state of our RNN so that we do not have to restart the training every time to generate $<10\%$ loss text.

References

[1] J. Foster and C. Mackie, *Lexical Analysis of the Dr. Seuss Corpus*, *Concordia Working Papers in Applied Linguistics*, 2013.
[2] A. Graves, *Generating Sequences With Recurrent Neural Networks*, University of Toronto, 2014.
[3] J. Brownlee, *Text Generation With LSTM Recurrent Neural Networks in Python with Keras*, Machine Learning Mastery, 2016.
[4] I. Sutskever, J. Martens, G. Hinton, *Generating Text with Recurrent Neural Networks*, University of Toronto, 2011.
[5] A. Karpathy, *The Unreasonable Effectiveness of Recurrent Neural Networks*, karpathy.github.io, 2015.
[6] M. Michaels, *The Basics of Seussian Verse*, mickmichaels.com, 2012.
[7] J. Beck, *The Secret to Dr. Seusss Made-Up Words*, The Atlantic, 2015.
[8] S. Ozair, *Char RNN Tensorflow*, Github Repository, https://github.com/sherjilozair/char-rnn-tensorflow.
[9] S. Ruder, *An Overview of Gradient Descent Optimization Algorithms*, ruder.io/optimizing-gradient-descent/index.html/adagrad, 19 Jan 2016.