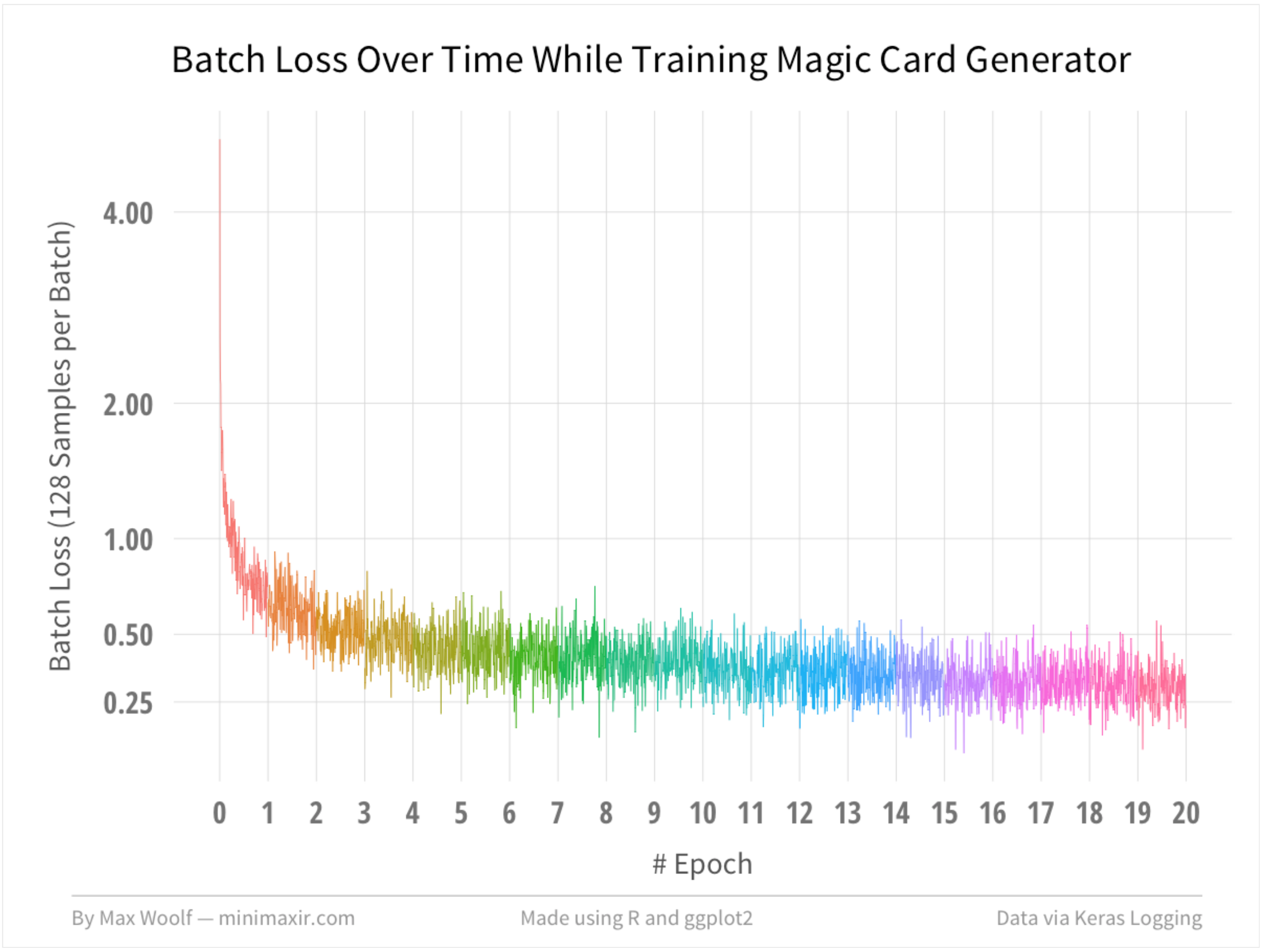
Pretrained Character Embeddings for Deep Learning and Automatic Text Generation

April 4, 2017

(https://www.facebook.com/share.php?u=http://minimaxir.com/2017/04/char-embeddings/) (http://twitter.com/home/?status=Pretrained Character Embeddings for Deep Learning and Automatic Text Generation - http://minimaxir.com/2017/04/char-embeddings/ - via @minimaxir) (http://www.linkedin.com/shareArticle?mini=true&title=&url=http://minimaxir.com/2017/04/char-embeddings/)



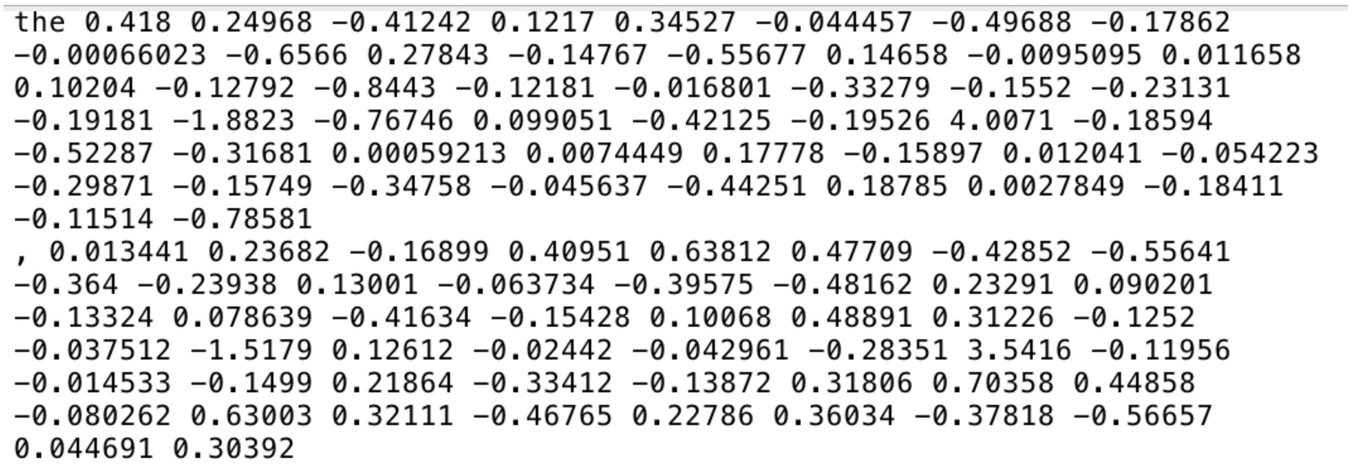
Deep learning is the biggest, often misapplied (http://approximatelycorrect.com/2017/03/28/the-ai-misinformation-epidemic/) buzzword nowadays for getting pageviews on blogs. As a result, there have been a lot of shenanigans lately with deep learning thought pieces and how deep learning can solve *anything* and make childhood sci-fi dreams come true.

I'm not a fan of Clarke's Third Law (http://tvtropes.org/pmwiki/pmwiki.php/Main/ClarkesThirdLaw), so I spent some time checking out deep learning myself. As it turns out, with modern deep learning tools like Keras (https://github.com/fchollet/keras), a higher-level framework on top of the popular TensorFlow (https://www.tensorflow.org) framework, deep learning is **easy to learn and understand**. Yes, easy. And it *definitely* does not require a PhD, or even a Computer Science undergraduate degree, to implement models or make decisions based on the output.

However, let's try something more expansive than the stereotypical deep learning tutorials.

CHARACTERS WELCOME

Word embeddings have been a popular machine learning trick nowadays. By using an algorithm such as Word2vec (https://en.wikipedia.org/wiki/Word2vec), you can obtain a numeric representation of a word, and use those values to create numeric representations of higher-level representations like sentences/paragraphs/documents/etc.



```
the 0.418 0.24968 -0.41242 0.1217 0.34527 -0.044457 -0.49688 -0.17862
-0.00066023 -0.6566 0.27843 -0.14767 -0.55677 0.14658 -0.0095095 0.011658
0.10204 -0.12792 -0.8443 -0.12181 -0.016801 -0.33279 -0.1552 -0.23131
-0.19181 -1.8823 -0.76746 0.099051 -0.42125 -0.19526 4.0071 -0.18594
-0.52287 -0.31681 0.00059213 0.0074449 0.17778 -0.15897 0.012041 -0.054223
-0.29871 -0.15749 -0.34758 -0.045637 -0.44251 0.18785 0.0027849 -0.18411
-0.11514 -0.78581
, 0.013441 0.23682 -0.16899 0.40951 0.63812 0.47709 -0.42852 -0.55641
-0.364 -0.23938 0.13001 -0.063734 -0.39575 -0.48162 0.23291 0.090201
-0.13324 0.078639 -0.41634 -0.15428 0.10068 0.48891 0.31226 -0.1252
-0.037512 -1.5179 0.12612 -0.02442 -0.042961 -0.28351 3.5416 -0.11956
-0.014533 -0.1499 0.21864 -0.33412 -0.13872 0.31806 0.70358 0.44858
-0.080262 0.63003 0.32111 -0.46765 0.22786 0.36034 -0.37818 -0.56657
0.044691 0.30392
```

However, generating word vectors for datasets can be computationally expensive (see my earlier post (http://minimaxir.com/2016/08/clickbait-cluster/) which uses Apache Spark/Word2vec to create sentence vectors at scale quickly). The academic way to work around this is to use pretrained word embeddings, such as the GloVe vectors (https://nlp.stanford.edu/projects/glove/) collected by researchers at Stanford NLP. However, GloVe vectors are huge; the largest one (840 billion tokens at 300D) is 5.65 GB on disk and may hit issues when loaded into memory on less-powerful computers.

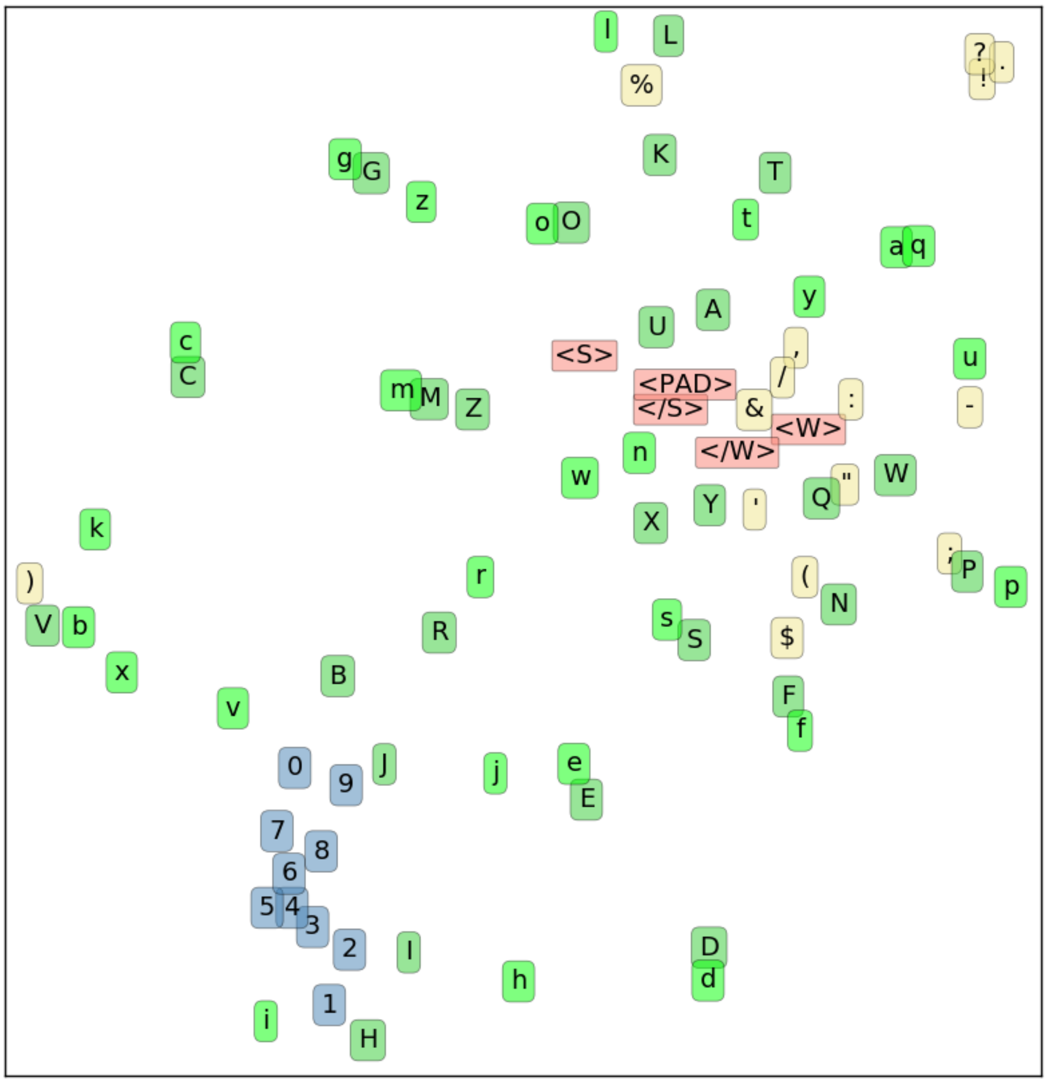
Why not work *backwards* and calculate *character* embeddings? Then you could calculate a relatively few amount of vectors which would easily fit into memory, and use those to derive word vectors, which can then be used to derive the sentence/paragraph/document/etc vectors. But training character embeddings traditionally is significantly more computationally expensive since there are 5-6x the amount of tokens, and I don't have access to the supercomputing power of Stanford researchers.

Why not use the *existing* pretrained word embeddings to extrapolate the corresponding character embeddings within the word? Think "bag-of-words (https://en.wikipedia.org/wiki/Bag-of-words_model)," except "bag-of-characters." For example, from the embeddings from the word "the", we can infer the embeddings for "t", "h," and "e" from the parent word, and average the t/h/e vectors from *all* words/tokens in the dataset corpus. (For this post, I will only look at the 840B/300D dataset since that is the only one with capital letters, which are rather important. If you want to use a dataset with smaller dimensionality, apply PCA (https://en.wikipedia.org/wiki/Principal_component_analysis) on the final results)

I wrote a simple Python script (https://github.com/minimaxir/char-embeddings/blob/master/create_embeddings.py) that takes in the specified pretrained word embeddings and does just that, outputting the character embeddings (https://github.com/minimaxir/char-embeddings/blob/master/glove.840B.300d-char.txt) in the same format. (for simplicity, only ASCII characters are included; the extended ASCII characters (https://en.wikipedia.org/wiki/Extended_ASCII) are intentionally omitted due to compatibility reasons. Additionally, by construction, space and newline characters are not represented in the derived dataset.)

```
$ 0.144288 -0.460809 0.205078 0.061188 -
( -0.045633 0.050699 0.183949 0.277213 0
, 0.083794 -0.471956 0.210845 -0.198393
0 0.177007 -0.285789 0.262566 -0.045959
4 0.189158 -0.29737 0.296691 -0.02323 -0
8 0.146808 -0.30308 0.273265 -0.02031 -0
< 0.46111 -0.384327 0.183889 -0.122777 0
@ 0.231718 -0.187341 0.055388 0.135627 -
D 0.233231 -0.361018 0.085259 -0.005739
```

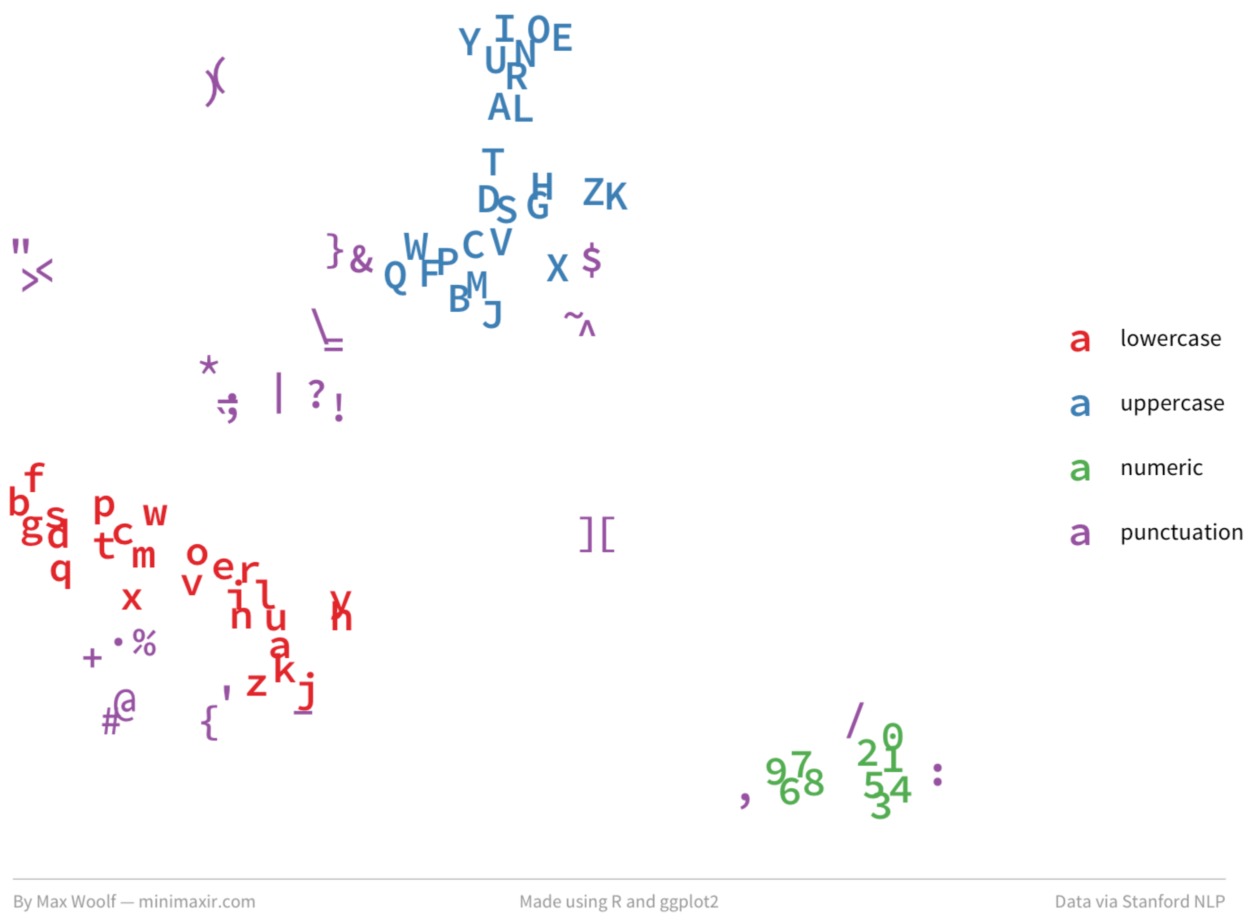
You may be thinking that I'm cheating. So let's set a point-of-reference. Colin Morris found (http://colinmorris.github.io/blog/1b-words-char-embeddings) that when 16D character embeddings from a model used in Google's One Billion Word Benchmark (https://arxiv.org/abs/1312.3005) are projected into a 2D space via t-SNE, patterns emerge: digits are close, lowercase and uppercase letters are often paired, and punctuation marks are loosely paired.



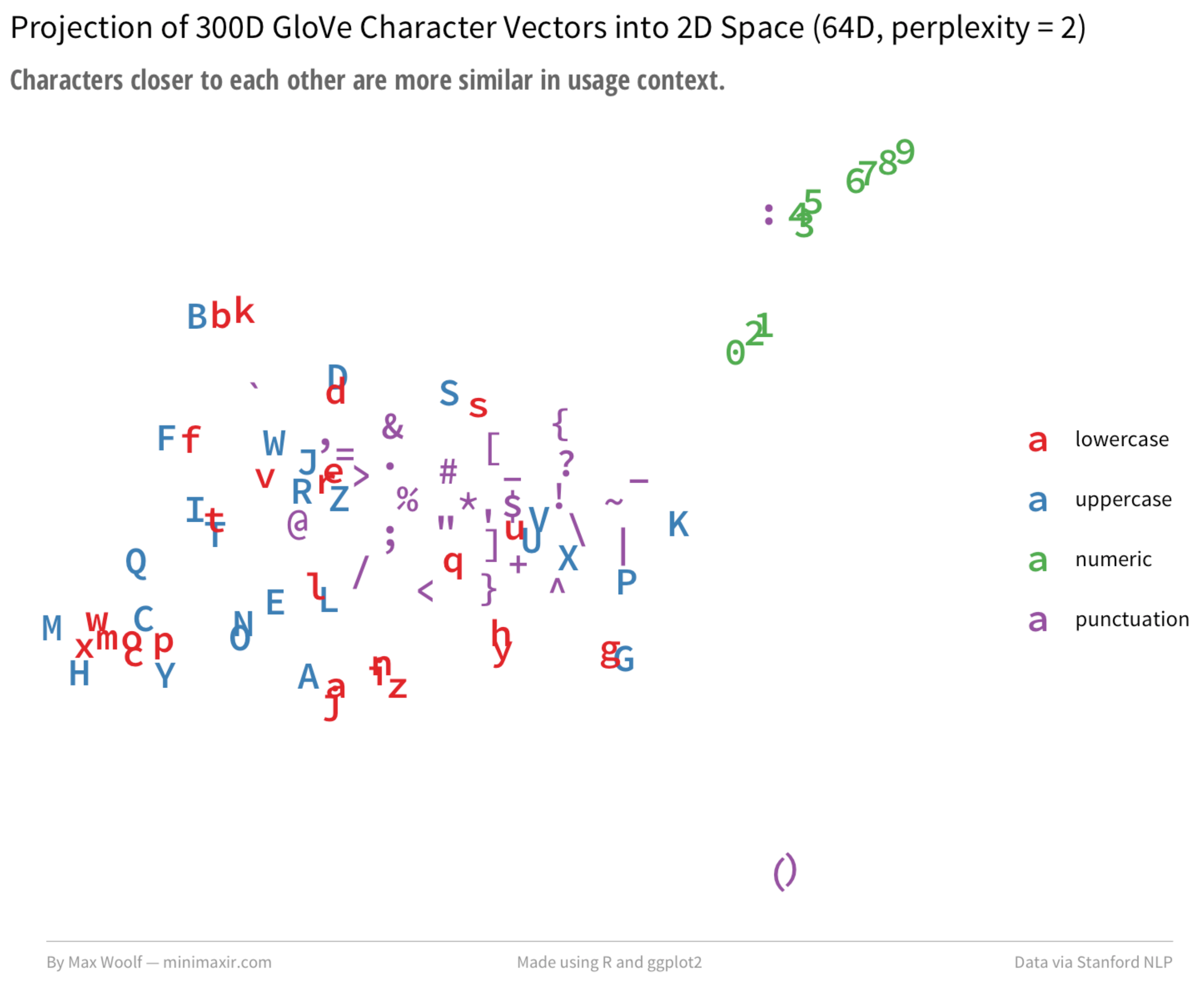
Let's do that for my derived character embeddings, but with R (https://www.r-project.org) and ggplot2 (http://docs.ggplot2.org/current/). t-SNE is difficult to use (http://distill.pub/2016/misread-tsne/) for high-dimensional vectors as combinations of parameters can result in wildly different output, so let's try a couple projections. Here's what happens when my pretrained projections are preprojected from 300D to 16D via PCA whitening (http://ufldl.stanford.edu/tutorial/unsupervised/PCAWhitening/), and setting perplexity (number of optimal neighbors) to 7.

Projection of 300D GloVe Character Vectors into 2D Space (16D, perplexity = 7)

Characters closer to each other are more similar in usage context.



The algorithm manages to separate and group lowercase, uppercase, and numerals rather distinctly. Quadrupling the dimensionality of the preprocessing step to 64D and changing perplexity to 2 generates a depiction closer to the Google model projection:



My pretrained character embeddings trick isn't academic, but it's successfully identifying realistic relationships. There might be something here worthwhile.

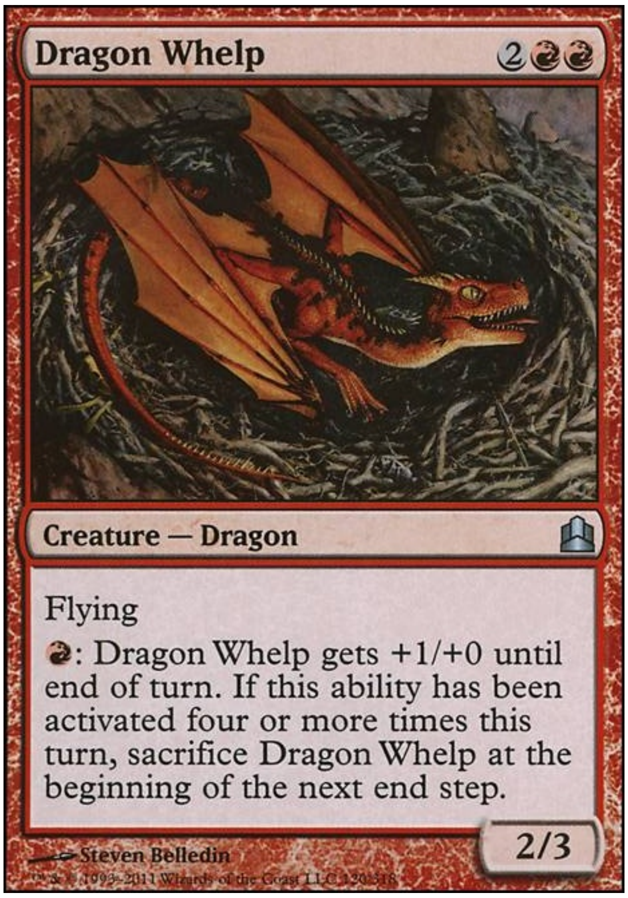
THE COOLNESS OF DEEP LEARNING

Keras, maintained by Google employee François Chollet (https://twitter.com/fchollet), is so good that it is effectively cheating in the field of machine learning, where even TensorFlow tutorials can be replaced with a single line of code. (which is important for iteration; Keras layers are effectively Lego blocks). A simple read of the Keras examples (https://github.com/fchollet/keras/tree/master/examples) and documentation (https://keras.io/) will let you reverse-engineer most the revolutionary deep learning clickbait thought pieces. Some create entire startups by changing the source dataset of the Keras examples and pitch them to investors none-the-wiser, or make very light wrappers on top the examples for teaching tutorial videos and get thousands of subscribers on YouTube.

I prefer to parse documentation/examples as a proof-of-concept, but never as gospel. Examples are often not the most efficient ways to implement a solution to a problem, just merely a start. In the case of Keras's text generator example (https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py), the initial code was likely modeled after the 2015 blog post The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/) by Andrej Karpathy and the corresponding project char-rnn (https://github.com/karpathy/char-rnn). There have been many new developments in neural network architecture since 2015 that can improve both speed and performance of the text generation model as a whole.

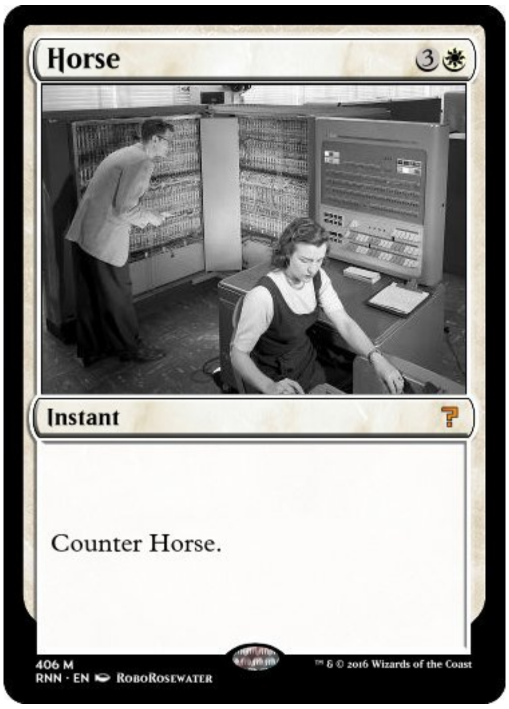
WHAT TEXT TO GENERATE?

The Keras example uses Nietzsche (https://en.wikipedia.org/wiki/Friedrich_Nietzsche) writings as a data source, which I'm not fond of because it's difficult to differentiate bad autogenerated Nietzsche rants from actual Nietzsche rants. What I want to generate is text with *rules*, with the algorithm being judged by how well it follows an inherent structure. My idea is to create Magic: The Gathering (http://magic.wizards.com/en) cards.


Dragon Whelp
2
Creature — Dragon
Flying
: Dragon Whelp gets +1/+0 until end of turn. If this ability has been activated four or more times this turn, sacrifice Dragon Whelp at the beginning of the next end step.
2/3
Steven Belledin

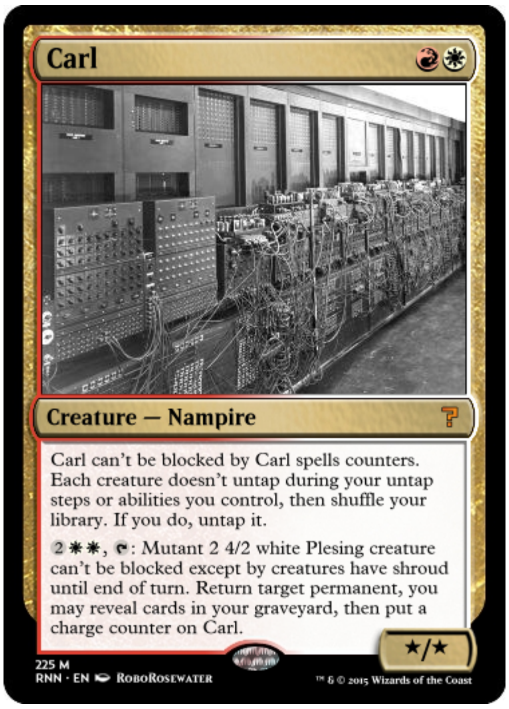
Inspired by the @RoboRosewater (https://twitter.com/RoboRosewater) Twitter account by Reed Milewicz and the corresponding research (http://www.mtgsalvation.com/forums/creativity/custom-card-creation/612057-generating-magic-cards-using-deep-recurrent-neural) and articles (https://motherboard.vice.com/en_us/article/the-ai-that-learned-magic-the-gathering), I aim to see if it's possible to recreate the structured design creativity for myself.

Even if you are not familiar with Magic and its rules, you can still find the card text (https://twitter.com/RoboRosewater/status/756198572282949632) of RoboRosewater cards hilarious:



Counter Horse.

Occasionally RoboRosewater, using a weaker model, produces amusing neural network trainwrecks (https://twitter.com/RoboRosewater/status/689184317721960448):



Carl can't be blocked by Carl spells counters. Each creature doesn't untap during your untap steps or abilities you control, then shuffle your library. If you do, untap it.

2 ✱✱, : Mutant 2 4/2 white Plesing creature can't be blocked except by creatures have shroud until end of turn. Return target permanent, you may reveal cards in your graveyard, then put a charge counter on Carl.

More importantly, all Magic cards have an explicit structure; they have a name, mana cost in the upper-right, card type, card text, and usually a power and toughness in the bottom-right.

I wrote another Python script (https://github.com/minimaxir/char-embeddings/blob/master/create_magic_text.py) to parse all Magic card data from MTG JSON (https://mtgjson.com) into an encoding which matches this architecture, where each section transition has its own symbol delimiter, along with other encoding simplicities. For example, here is the card Dragon Whelp (http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=247314) in my encoding:

```
[Dragon Whelp@{2}{R}{R}#Creature — Dragon$Flying|{R}: ~ gets +1/+0 until end of turn. If this ability
 has been activated four or more times this turn, sacrifice ~ at the beginning of the next end step.%2
^3]
```

These card encodings are all combined into one .txt file, which will be fed into the model.

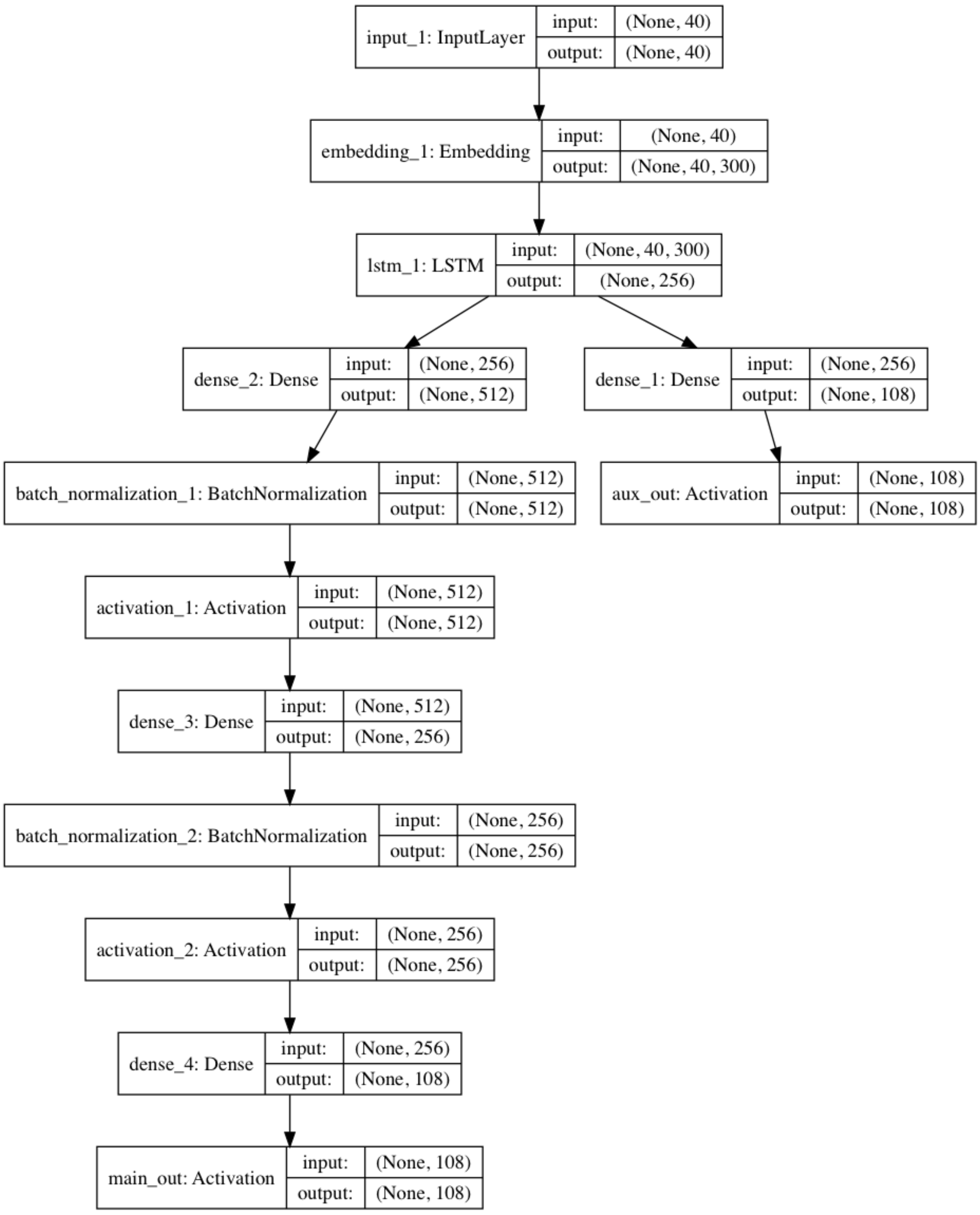
BUILDING AND TRAINING THE MODEL

The Keras text generation example operates by breaking a given .txt file into 40-character sequences, and the model tries to predict the 41st character by outputting a probability for each possible character (108 in this dataset). For example, if the input based on the above example is `['D', 'r', 'a', 'g', ..., 'D', 'r', 'a', 'g']` (with the latter Drag being part of the creature type), the model will optimize for outputting a probability of 1.0 of `o`; per the categorical crossentropy (https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_error_function_and_logistic_regression) loss function, the model is rewarded for assigning correct guesses with 1.0 probability and incorrect guesses with 0.0 probabilities, penalizing half-guesses and wrong guesses.

Each possible 40-character sequence is collected, however only every other third sequence is kept; this prevents the model from being able to learn card text verbatim, plus it also makes training faster. (for this model, there are about **1 million** sequences for the final training). The example uses only a 128-node long-short-term-memory (https://en.wikipedia.org/wiki/Long_short-term_memory) (LSTM) recurrent neural network (https://en.wikipedia.org/wiki/Recurrent_neural_network) (RNN) layer, popular for incorporating a “memory” into a neural network model, but the example notes at the beginning it can take awhile to train before generated text is coherent.

There are a few optimizations we can make. Instead of supplying the characters directly to the RNN, we can first encode them using an Embedding layer (https://keras.io/layers/embeddings/) so the model can train character context. We can stack more layers on the RNN by adding a 2-level multilayer perceptron (https://en.wikipedia.org/wiki/Multilayer_perceptron): a meme (https://www.reddit.com/r/ProgrammerHumor/comments/5si1f0/machine_learning_approaches/), yes, but it helps, as the network must learn latent representations of the data. Thanks to recent developments such as batch normalization (https://arxiv.org/abs/1502.03167) and rectified linear activations (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) for these Dense layers (https://keras.io/layers/core/#dense), they can both be trained without as much computational overhead, and thanks to Keras, both can be added to a layer with a single line of code each. Lastly, we can add an auxiliary output via Keras’s functional API (https://keras.io/models/model/) where the network makes a prediction based on only the output from the RNN in addition to the main output, which forces it to work smarter and ends up resulting in a *significant* improvement in loss for the main path.

The final architecture ends up looking like this:


input_1: InputLayer | input: (None, 40) | output: (None, 40)
embedding_1: Embedding | input: (None, 40) | output: (None, 40, 300)
lstm_1: LSTM | input: (None, 40, 300) | output: (None, 256)
dense_2: Dense | input: (None, 256) | output: (None, 512)
dense_1: Dense | input: (None, 256) | output: (None, 108)
batch_normalization_1: BatchNormalization | input: (None, 512) | output: (None, 512)
aux_out: Activation | input: (None, 108) | output: (None, 108)
activation_1: Activation | input: (None, 512) | output: (None, 512)
dense_3: Dense | input: (None, 512) | output: (None, 256)
batch_normalization_2: BatchNormalization | input: (None, 256) | output: (None, 256)
activation_2: Activation | input: (None, 256) | output: (None, 256)
dense_4: Dense | input: (None, 256) | output: (None, 108)
main_out: Activation | input: (None, 108) | output: (None, 108)

And because we added an Embedding layer, we can load the pretrained 300D character embeds I made earlier, giving the model a good start in understanding character relationships.

The goal of the training is to minimize the total loss of the model. (but for evaluating model performance, we only look at the loss of the main output). The model is trained in **epochs**, where the model sees all the input data atleast once. During each epoch, batches of size 128 are loaded into the model and evaluated, calculating a **batch loss** for each; the gradients from the batch are backpropagated into the previous layers to improve them. While training with Keras, the console reports an **epoch loss**, which is the average of all the batch losses so far in the current epoch, allowing the user to see in real time how the model improves, and it’s addicting.

```
______________________________________________________________________
____________________________________________

_______________________________________________________________
Iteration 1
```

Keras/TensorFlow works just fine on the CPU, but for models with a RNN, you'll want to consider using a GPU for performance, specifically one by nVidia. Amazon has cloud GPU instances for $0.90/hr (not prorated (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Stop_Start.html)), but very recently, Google announced GPU instances (https://cloud.google.com/compute/docs/gpus/add-gpus) of the same caliber for ~$0.75/hr (prorated to the minute), which is what I used to train this model, although Google Compute Engine requires configuring the GPU drivers first. For 20 epochs, it took about 4 hours and 20 minutes to train the model while spending $3.26, which isn't bad as far as deep learning goes.

MAKING MAGIC

After each epoch, the original Keras text generation example takes a sentence from the input data as a seed and predicts the next character in the sequence according to the model, then uses the last 40 characters generated for the next character, etc. The sampling incorporates a diversity/temperature parameter which allows the model to make suboptimal decisions and select characters with lower natural probabilities, which allows for the romantic "creativity" popular with neural network text generation.

With the Magic card dataset and my tweaked model architecture, generated text is coherent after the 1st epoch (https://github.com/minimaxir/char-embeddings/blob/master/output/iter-01-0_9204.txt)! After about 20 epochs, training becomes super slow, but the predicted text becomes super interesting. Here are a few fun examples from a list of hundreds of generated cards (https://github.com/minimaxir/char-embeddings/blob/master/output/text_sample.txt). (Note: the power/toughness values at the end of the card have issues; more on that later).

With low diversity, the neural network generated cards that are oddly biased toward card names which include the letter "S". The card text also conforms to the rules of the game very well.

```
[Reality Spider@{3}{G}#Creature — Elf Warrior$Whenever ~ deals combat damage to a player, put a +1/+1
 counter on it.%^]
[Dark Soul@{2}{R}#Instant$~ deals 2 damage to each creature without flying.%^]
[Standing Stand@{2}{G}#Creature — Elf Shaman${1}{G}, {T}: Draw a card, then discard a card.%^]
```

In contrast, cards generated with high diversity hit the uncanny valley of coherence and incoherence in both text and game mechanic abuse, which is what makes them interesting.

```
[Portrenline@{2}{R}#Sorcery$As an additional cost to cast ~, exile ~.%^]
[Clocidian Lorid@{W}{W}{W}#Instant$Regenerate each creature with flying and each player.%^]
[Icomic Convermant@{3}{G}#Sorcery$Search your library for a land card in your graveyard.%1^1]
```

The best-of-both-worlds cards are generated from diversity parameters between both extremes, and often have funny names.

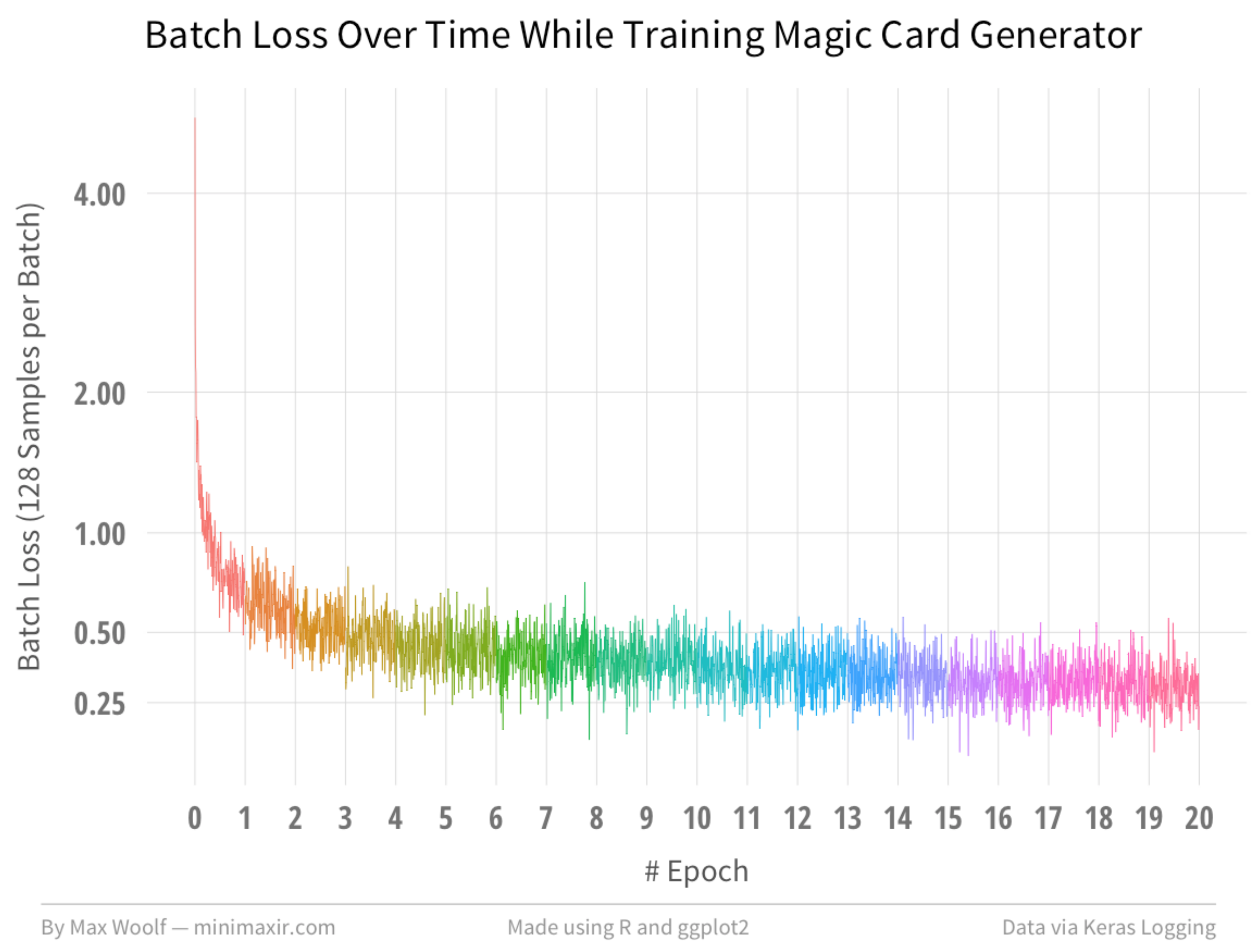
```
[Seal Charm@{W}{W}#Instant$Exile target creature. Its controller loses 1 life.%^]
[Shambling Assemblaster@{4}{W}#Creature — Human Cleric$When ~ enters the battlefield, destroy target n
onblack creature.%1^1]
[Lightning Strength@{3}{R}#Enchantment — Aura$Enchant creature|Enchanted creature gets +3/+3 and has f
lying, flying, trample, trample, lifelink, protection from black and votile all damage unless you retu
rn that card to its owner's hand.%2^2]
[Skysor of Shadows@{7}{B}{B}{B}#Enchantment$As ~ enters the battlefield, choose one —|• Put a -1/-1 co
unter on target creature.%2^2]
[Glinding Stadiers@{4}{W}#Creature — Spirit$Protection from no creatures can't attack.%^]
[Dragon Gault@{3}{G}{U}{U}#Creature — Kraven$~'s power and toughness are 2.%2^2]
```

All Keras/Python code used in this blog post, along with sample Magic card output and the trained model itself, is available open-source in this GitHub repository (https://github.com/minimaxir/char-embeddings). The repo additionally contains a Python script (https://github.com/minimaxir/char-embeddings/blob/master/text_generator_keras_sample.py) which lets you generate new cards using the model, too!

VISUALIZING MODEL PERFORMANCE

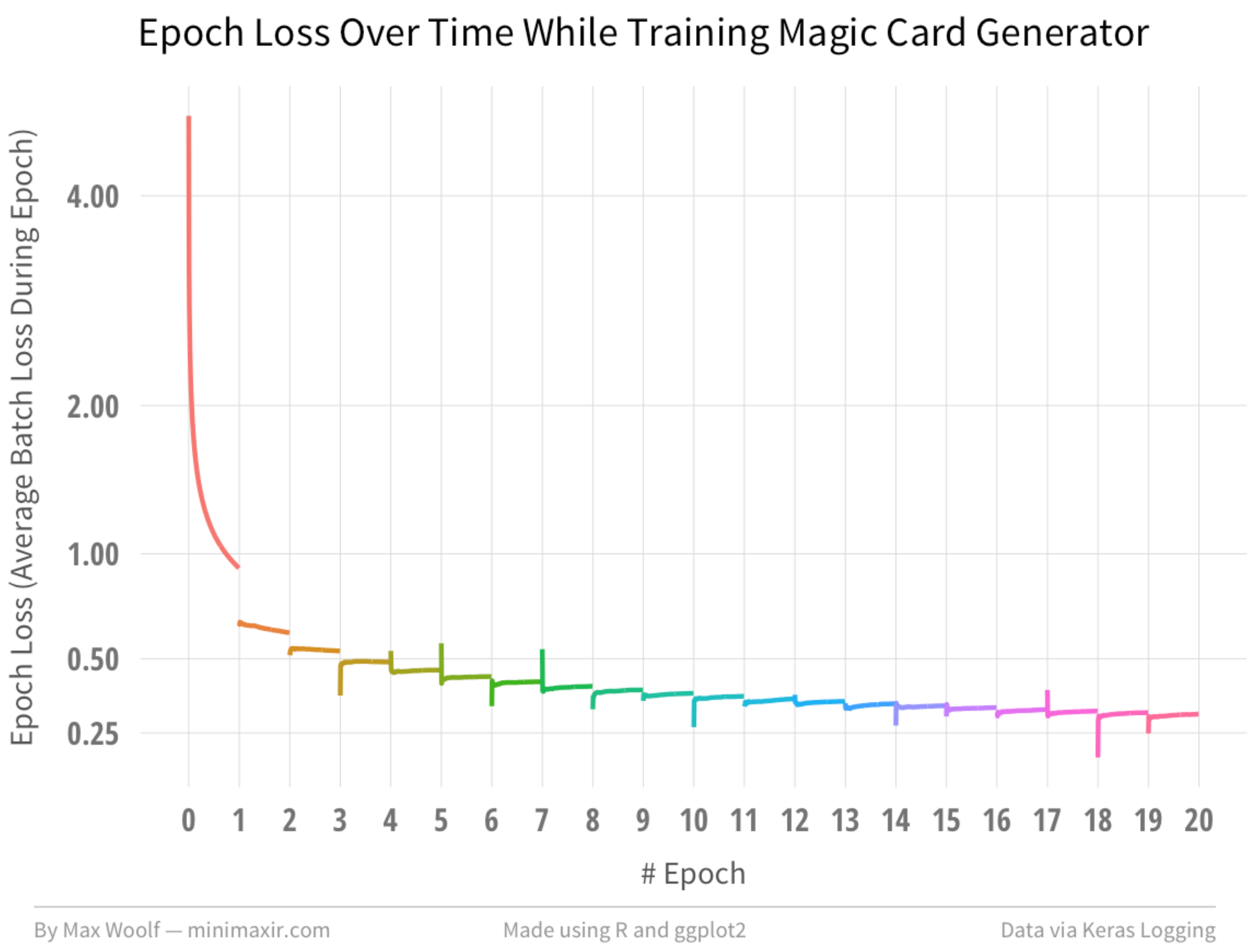
One thing deep learning tutorials rarely mention is *how* to collect the loss data and visualize the change in loss over time. Thanks to Keras's utility functions (https://keras.io/callbacks/), I wrote a custom model callback which collects the batch losses and epoch losses and writes them to a CSV file.

Using R and ggplot2, I can plot the batch loss at every 50th batch to visualize how the model converges over time.



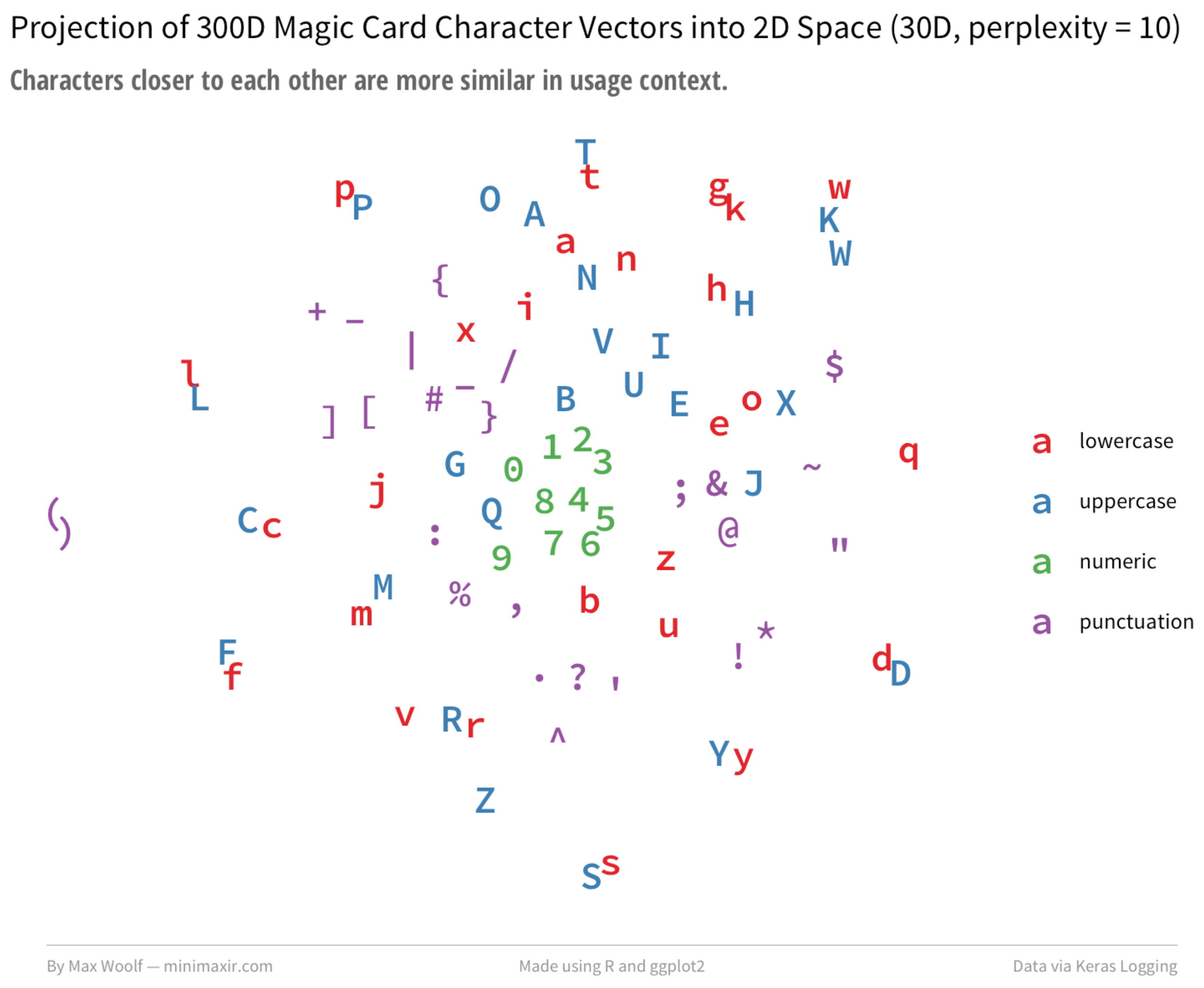
After 20 epochs, the model loss ends up at about **0.30** which is more-than-low-enough for coherent text. As you can see, there are large diminishing returns after a few epochs, which is the hard part of training deep learning models.

Plotting the epoch loss over the batches makes the trend more clear.



In order to prevent early convergence, we can make the model more complex (i.e. stack more layers unironically), but that has trade-offs, both in training *and* predictive speed, the latter of which is important if using deep learning in a production application.

Lastly, as with the Google One Billion Words benchmark, we can extract the trained character embeddings (https://github.com/minimaxir/char-embeddings/blob/master/output/char-embeddings.txt) from the model (now augmented with Magic card context!) and plot them again to see what has changed.



There are more pairs of uppercase/lowercase characters, although interestingly there isn't much grouping with the special characters added as section breaks in the encoding, or mechanical uppercase characters such as W/U/B/R/G/C/T.

NEXT STEPS

After building the model, I did a little more research to see if others solved the power/toughness problem. Since the sentences are only 40 characters and Magic cards are much longer than 40 characters, it's likely that power/toughness are out-of-scope for the model and it cannot learn their exact values. Turns out that the intended solution is to use a completely different encoding (https://github.com/billzorn/mtgencode), such as this one for Dragon Whelp:

```
|5creature|4|6dragon|7|8&^^/&^^^|9flying\{RR}: @ gets +&^/+& until end of turn. if this ability has be
en activated four or more times this turn, sacrifice @ at the beginning of the next end step.|3{^^RRR
R}|0N|1dragon whelp|
```

Power/toughness are generated near the *beginning* of the card. Sections are delimited by pipes, with a numeral designating the corresponding section. Instead of numerals being used card values, carets are used, which provides a more accurate *quantification* of values. With this encoding, each character has a *singular purpose* in the global card context, and their embeddings would likely generate more informative visualizations. (But as a consequence, the generated cards are harder to parse at a glance).

The secondary encoding highlights a potential flaw in my methodology using pretrained character embeddings. Trained machine learning models must be used apples-to-apples on similar datasets; for example, you can't accurately perform Twitter sentiment analysis (https://en.wikipedia.org/wiki/Sentiment_analysis) on a dataset using a model trained on professional movie reviews since Tweets do not follow AP Style (https://owl.english.purdue.edu/owl/resource/735/02/) guidelines. In my case, the Common Crawl (http://commoncrawl.org), the source of the pretrained embeddings, follows more natural text usage and would not work analogously with the atypical character usages in *either* of the Magic card encodings.

There's still a *lot* of work to be done in terms of working with both pretrained character embeddings and improving Magic card generation, but I believe there is promise. The better way to make character embeddings than my script is to do it the hard way and train then manually, maybe even at a higher dimensionality like 500D or 1000D. Likewise, for Magic model building, the mtg-rnn instructions (https://github.com/billzorn/mtgencode#training-a-neural-net) repo uses a large LSTM stacked on a LSTM along with 120/200-character sentences, both of which combined make training **VERY** slow (notably, this was the architecture of the very first commit (https://github.com/fchollet/keras/commit/d2b229df2ea0bab712379c418115bc44508bc6f9#diff-904d72bcf9fa38b32f9c1f868ff59367) for the Keras text generation example, and was changed (https://github.com/fchollet/keras/commit/01d5e7bc4782daafcfa99e035c1bdbe13a985145) to the easily-trainable architecture). There is also promise in a variational autoencoder (http://kvfrans.com/variational-autoencoders-explained/) approach, such as with textvae (https://arxiv.org/abs/1702.02390).

This work is potentially very expensive and I am strongly considering setting up a Patreon (https://www.patreon.com) in lieu of excess venture capital to subsidize my machine learning/deep learning tasks in the future.

At minimum, working with this example gave me a sufficient application of practical work with Keras, and another tool in my toolbox for data analysis and visualization. Keras makes the model-construction aspect of deep learning trivial and not scary. Hopefully, this article justifies the use of the "deep learning" buzzword in the headline.

It's also worth mentioning that I actually started working on automatic text generation 6 months ago using a different, non-deep-learning approach, but hit a snag and abandoned that project. With my work on Keras, I found a way around that snag, and on the same Magic dataset with the same input construction, I obtained a model loss of **0.03** at **20% of the cloud computing cost** in about the same amount of time. More on that later.

The code for generating the R/ggplot2 data visualizations is available in this R Notebook (http://minimaxir.com/notebooks/char-tsne/), and open-sourced in this GitHub Repository. (https://github.com/minimaxir/char-tsne-visualization)

You are free to use the automatic text generation scripts and data visualizations from this article however you wish, but it would be greatly appreciated if proper attribution is given to this article and/or myself!

Hi! I am currently **looking for a job** in data analysis/software engineering in San Francisco. If you liked this post and have a lead, feel free to shoot me an email (mailto:max@minimaxir.com).

Since I currently do not have a full-time salary to subsidize my machine learning/deep learning/software/hardware needs for these blog posts, I have set up a Patreon (https://www.patreon.com/minimaxir), and any monetary contributions to the Patreon are appreciated and will be put to good creative use.

Share this article!
(https://www.facebook.com/share.php?u=http://minimaxir.com/2017/04/char-embeddings/) (http://twitter.com/home/?status=Pretrained Character Embeddings for Deep Learning and Automatic Text Generation - http://minimaxir.com/2017/04/char-embeddings/ - via @minimaxir) (http://www.linkedin.com/shareArticle?mini=true&title=&url=http://minimaxir.com/2017/04/char-embeddings/)

1 Comment minimaxir 1 Login

Recommend Share Sort by Best

Join the discussion…

Karolis Ramanauskas • 20 days ago
Great project, kudos! Could you explain what batch loss refers to in this case? In binary classification, for example, it's easy to see how its performance can be evaluated by seeing how many items it guessed correctly, and not. But in random text generation I am struggling to understand how one measures its performance as there is really no wrong or right answer.
• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

AUTHOR

Max Woolf (@minimaxir) is a former Apple Software QA Engineer living in San Francisco and a Carnegie Mellon University graduate.

In his spare time, Max uses Python (https://www.python.org/) to gather data from public APIs and ggplot2 (http://ggplot2.org/) to plot plenty of pretty charts from that data.

You can learn more about Max here (/about), view his data analysis portfolio here (/data-portfolio), or view his coding portfolio here (/portfolio).

(https://facebook.com/max.woolf) (https://twitter.com/minimaxir) (https://linkedin.com/in/minimaxir)
(mailto:max@minimaxir.com) (https://github.com/minimaxir) (https://youtube.com/minimaxir) (/rss.xml)

RECENT POSTS

Pretrained Character Embeddings for Deep Learning and Automatic Text Generation (/2017/04/char-embeddings/)
Apr 4, 2017

Predicting And Mapping Arrest Types in San Francisco with LightGBM, R, ggplot2 (/2017/02/predicting-arrests/)
Feb 8, 2017

Playing with 80 Million Amazon Product Review Ratings Using Apache Spark (/2017/01/amazon-spark/)
Jan 2, 2017

Network Visualization of Breached Internet Services Using HaveIBeenPwned? Data (/2016/12/pwned-network/)
Dec 19, 2016

Infinite-Quality Abstract Art and Animations with Primitive (/2016/12/primitive/)
Dec 12, 2016

How to Create an Interactive WebGL Network Graph Using R (/2016/12/interactive-network/)
Dec 5, 2016

GITHUB CODE REPOSITORIES

Big List of Naughty Strings (http://github.com/minimaxir/big-list-of-naughty-strings)
★ 20,142+

Facebook Page Post Scraper (http://github.com/minimaxir/facebook-page-post-scraper)
★ 1,035+

Multiplatform System Dashboard (http://github.com/minimaxir/system-dashboard)
★ 100+

Tritonize (http://github.com/minimaxir/tritonize)
★ 9+

Stylistic Word Clouds (http://github.com/minimaxir/stylistic-word-clouds)
★ 30+

Pretrained Character Embeddings for Deep Learning and Automatic Text Generation (http://github.com/minimaxir/char-embeddings)
★ 8+

Get GitHub Profile Data of All Stargazers of a GitHub Repo (http://github.com/minimaxir/get-profile-data-of-repo-stargazers)
★ 15+

Copy Syntax Highlight for OS X (http://github.com/minimaxir/copy-syntax-highlight-osx)
★ 349+

Convert Video to GIF on OS X (http://github.com/minimaxir/video-to-gif-osx)
★ 262+

DATA ANALYSIS NOTEBOOKS

Pretrained Character Embeddings for Deep Learning and Automatic Text Generation (http://minimaxir.com/notebooks/char-tsne/)
R, ggplot2

Predicting And Mapping Arrest Types in San Francisco (http://minimaxir.com/notebooks/predicting-arrests/)
R, ggplot2, LightGBM

Playing with 80 Million Amazon Product Review Ratings (http://minimaxir.com/notebooks/amazon-spark/)
R, ggplot2, Spark

How to Create an Interactive WebGL Network Graph (http://minimaxir.com/notebooks/interactive-network/)
R, ggplot2, plotly

What Percent of the Top-Voted Comments in Reddit Threads Were Also 1st Comment? (http://minimaxir.com/notebooks/first-comment/)
R, ggplot2

Processing Clusters of Clickbait Headlines (https://github.com/minimaxir/clickbait-cluster/blob/master/fb_news_53D_spark.ipynb)
Python, Spark, word2vec

Visualizing Clusters of Clickbait Headlines (https://github.com/minimaxir/clickbait-cluster/blob/master/fb_news_53D_plotly.ipynb)
R, plotly

Processing Pokémon Data With Apache Spark (https://github.com/minimaxir/pokemon-3d/blob/master/pokemon_spark_pca.ipynb)
Python, Spark

Interactive 3D Clusters of all 721 Pokémon (https://github.com/minimaxir/pokemon-3d/blob/master/pokemon_3d_plotly.ipynb)
R, ggplot2, plotly

Max Woolf (@minimaxir) is a former Apple Software QA Engineer living in San Francisco and a Carnegie Mellon University graduate.

In his spare time, Max uses Python (https://www.python.org/) to gather data from public APIs and ggplot2 (http://ggplot2.org/) to plot plenty of pretty charts from that data.

You can learn more about Max here (/about), view his data analysis portfolio here (/data-portfolio), or view his coding portfolio here (/portfolio).

(https://facebook.com/max.woolf) (https://twitter.com/minimaxir) (https://linkedin.com/in/minimaxir)

(mailto:max@minimaxir.com) (https://github.com/minimaxir) (https://youtube.com/minimaxir)

(/rss.xml)