Capstone Project

Machine Learning Nanodegree

Topic: Generative Deep Learning

By Michael Eryan

Definition

Project Overview

Machine generated written and spoken language has become a popular field both in academia and industry. Two topics are interesting to me: natural language processing and sequential data analysis. Coming from an academic and professional background of time series forecasting, I wanted to learn how to use generative deep learning to produce new textual data which even if not meaningful can still be interesting and a good practice for me.

When usually discussing time series data people think of forecasting well defined numeric data series that have underlying trends and seasonalities such as weather characteristics like temperature and rainfall. In general though, when we have any data whose temporal ordering matters to us we are dealing with "sequence data" and we cannot apply the same models that we use on cross-sectional data whose order does not matter. This is true in both econometrics where we cannot use linear methods but need to use auto-regressive methods to build forecasting models and also in machine learning where we cannot use many of the supervised learning algorithms and need to use recurrent neural networks.

Using recurrent neural networks to generate sequence data has become very popular in the last few years especially since a major theoretical hurdle of the "vanishing gradient" has been overcome with the invention of the LSTM (long short-term memory) algorithm in 1997 by computer scientists Sepp Hochreiter and Jurgen Schmidhuber. Since then other scientists like Douglas Eck and Alex Graves have done pioneering work to prove both the feasibility and effectiveness of recurrent neural networks. All this progress served as inspiration for a software engineer named François Chollet who created the Python library Keras which I will be using for this project with Tensorflow as the back end.

The main dataset for this project was created from freely available scripts of a Comedy Central animated series "Futurama" which happens to be one my favorite shows.

Television scripts are inherently time series data - not only does each word in a sentence depend on the previous word but also the follow up response of the next speaker is dependent on the previous text.

What I liked about this animated sitcom is that it seemed to have a small number of main speakers and short sentences. When I started the project my goal was to create a neural network that would take a speaker's name and generate a sentence that sounded like this speaker and then follow up with responses by other speakers.

Problem Statement

My ultimate goal was to create a function to generate new text that looks like the original Futurama scripts. This is a task that machine learning can handle as it requires creating a neural network that can learn from sequential text data to generate new data. If successful at this task, my next goal would be building text generating web applications or maybe even chatbots.

Generating new text from a training set of text is a lot like forecasting the future based on past information. A successful model would not only generate text that makes intuitive sense but also uncover patterns invisible to the naked eye.

The task of generating text is both quantifiable and measurable because we can compare the results of a trained model and a random benchmark model. When we input a "seed" word into the random benchmark, it would pick the next word from a uniform probability distribution among all the potential words. This approach has literally zero percent chance of generating anything useful.

In contrast, a well trained neural network would be able to pick the most probable "next best word." We can measure the accuracy of these predictions using both the training and validation data sets. The contrast between the output of the benchmark and trained model should be noticeable not just in terms of evaluation metrics but also by eyeballing.

Metrics

My objective function was a loss function that the network minimized during training. Since my model's input was a sequence of words and output was one predicted word, what I needed was a loss function that measured how well that predicted word fit among the observed words.

This is a many-category classification problem because we needed to pick the best category among more than two available ones. Because my targets were one-hot encoded values, a suitable loss function was *categorical cross entropy*.

The evaluation metric that I used was *categorical accuracy* which is similar to the accuracy metric used in binary classification.

When the target is a binary variable, accuracy: (TP + TN) / (Total_observations) with the default cutoff 0.5.

In Keras this looks like this:

```
K.mean(K.equal(y_true, K.round(y_pred)))
```

(*remember: "round" function rounds values of 0.5 and above to 1*)

In contrast, categorical accuracy looks like this in Keras:

```
K.mean(K.equal(K.argmax(y_true, axis=-1), K.argmax(y_pred, axis=-1)))
```

Technically this formula compares the index of the maximal true value (meaning: which word is labeled as 1) to the index of the maximal predicted value (which word has the highest probability). When they are equal, it means that the model predicted the true value correctly.

Categorical accuracy intuitively means the same as binary accuracy - how often the model makes the correct prediction.

Analysis

Data Preparation and Exploration

I obtained the data from a public website by using the BeautifulSoup library for Python. Once I had enough data, I cleaned the data to my own liking. Since my goal was to generate dialogues, I removed all useless header data, scene descriptions, articles and so on.

In order for the Keras tokenizer to process the data adequately, I had to further clean the data. I created my token lookup dictionary to map original text to more robust representations.

For example, cartoon characters names like "Fry" and "Bender" appear in the scripts as both speaker identifiers and also as references within the actual dialogues. By using my custom dictionary I made sure the tokenizer represented these words by different integers thus allowing the network to treat them distinctly.

The data set that I used had the following characteristics:

- Number of episodes from seasons 1 and 2: 20
- Total number of characters (letters): 287,799
- Total number of words: 47,367
- Actual number of unique tokenized words: 6,040
- Number of lines: 12,196

Top five speakers in terms of the number of utterances were:

- fry: 999
- bender: 681
- leela: 658
- farnsworth: 255
- hermes: 66

Top five speakers in terms of the number of times being mentioned in the scripts were:

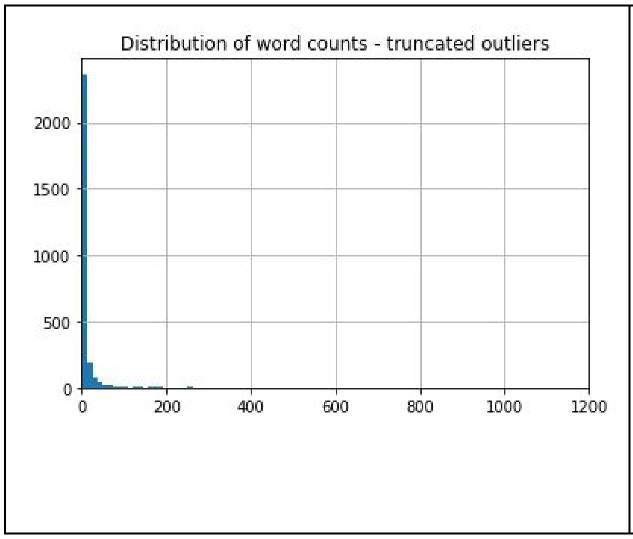
- bender 187
- fry 186
- leela 81
- hermes 12
- farnsworth 5

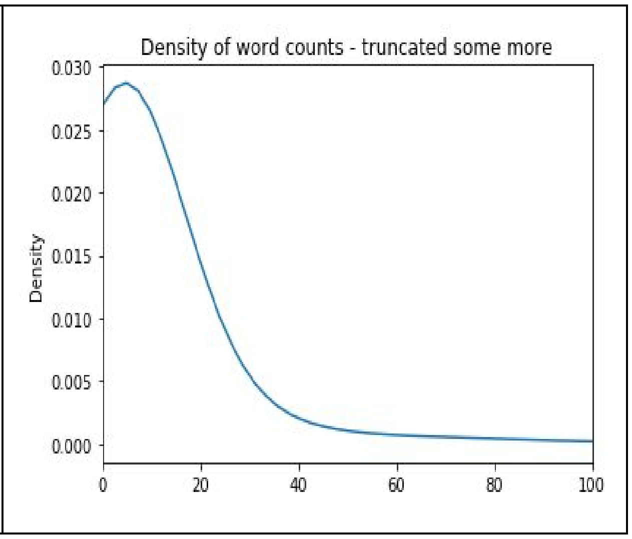
This makes sense: Fry and Bender are the main characters of the show and, therefore, they speak most often and are also most talked about.

Exploratory visualization

In terms of word frequency my data was very long tailed as could be expected with any text data. Out of 6,040 unique words 58% appeared only once.

Histogram and density plots are shown below. X-axes represent the integer representations of the words in the descending order of frequency.





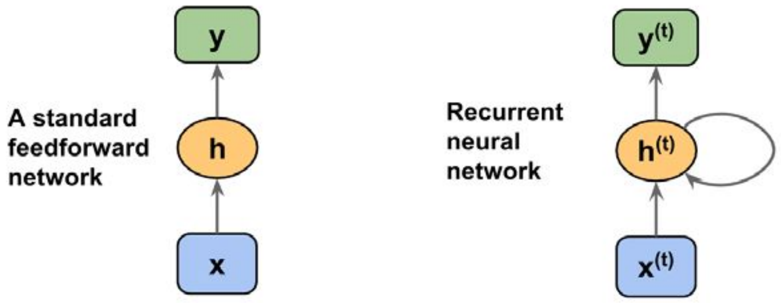
Such skewness certainly presents challenges for the model but, given, that I am mostly interested in generating text for the show's main characters, I deemed the data set as acceptable.

Algorithms and techniques

In order to generate new text from scripts, I experimented with several recurrent neural network approaches:

1. One hidden layer with SimpleRNN units (my network benchmark)
2. Single and multiple layer networks with LSTM and GRU units
3. 1D convolutional neural networks followed by a layer with GRU units

Introduction to Recurrent Neural Networks (RNN)



(Rashcka 541)

Recurrent neural networks stand in contrast to the feedforward networks. In a feedforward network used for supervised learning the entire vector of features is used all at once to predict the target. For this approach the order of the ingredients in the vector does not matter, only the content does. Such a network cannot have a "memory" of how the ingredients follow each other because they are processed all at once.

Imagine reading a sentence in which all the words have been re-ordered randomly. You might still get the gist of it, but not the whole meaning. Our minds are designed to process sentences in their natural order one step at a time. Recurrent neural networks attempt to mimic this process - they use the natural order of the vector as an additional piece of information to learn and make better predictions. (Chollet 196)

Let's look at a concrete example that I put together (not necessary an actual training sample). The feature vector has twenty words and the target is one word: "pizza."

FRY:
<4 lines of 5 words each makes my 20 word sequence - this is my X - feature>
Pizza dinner on me Just
keep tab under 50 million
ROBOT_CHEF: Yo I havent got
all day What kind of
<the next word is my Y - target>
pizza

you guys want

If we use a feedforward network, we would use all the preceding twenty words as features to predict the next word. We might still guess that the next word is "pizza" - perhaps because it appears in the feature vector already.

If we use a RNN, it will learn time-step by time-step that when "Pizza" is followed by "dinner" followed by "on" followed by "me" ... followed by "What" followed by "kind" followed by "of" - that chances are pretty good the next word is "pizza."

If you are still unconvinced, scramble the twenty words and then try to guess the next word yourself. I think you would agree that the order definitely helps which is precisely what the RNN try to take advantage of.

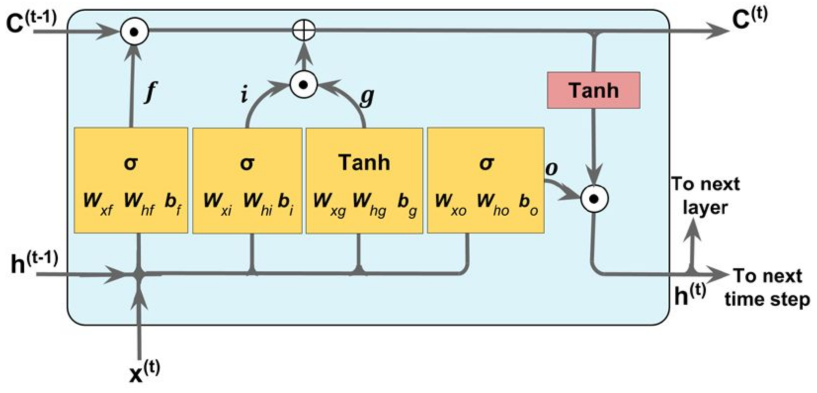
One caveat that we need to remember is that deep learning models like RNN do not actually understand the meaning of the texts like we do. Rather they recognize the underlying patterns and the statistical structures of the data which they use to solve the tasks we give them.

RNN Layers Menu of Choice

A recurrent layer in a RNN is made of my special units capable of retaining information about previously processed data points. For this project I considered three types of units: SimpleRNN, LSTM and GRU.

SimpleRNN has been the easiest to understand for me because it reminds me of a econometric auto-regressive models used for time series. Using my previous example, when processing the last words of the sequence "What kind of," SimpleRNN unit would use large portion of the information from the preceding words "Yo I havent got all day" but a much smaller portion of the information from the first "Pizza dinner" which are further back. This is a major handicap of SimpleRNN because it eventually forgets past information or to put in technical terms it suffers from the vanishing gradient problem.

In order to address this shortcoming, two new kinds of layers have been created: LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit). They both work similarly, though in my experience, GRU runs faster. Here is how the LSTM cell looks like:



$C^{(t-1)}$
$C^{(t)}$
f
i
g
Tanh
σ W_{xf} W_{hf} b_f
σ W_{xi} W_{hi} b_i
Tanh W_{xg} W_{hg} b_g
σ W_{xo} W_{ho} b_o
o
To next layer
$h^{(t-1)}$
$h^{(t)}$
To next time step
$x^{(t)}$

(LSTM cell, Raschka 548)

Detailed discussion of workings of the LSTM cell is outside of the scope of this paper. What we need to understand is that the LSTM and GRU units preserve potentially useful information for later time-steps to allow them a chance to influence the predictions. In our example the first words "Pizza dinner" would get a better chance to be tied to the target "pizza" than in the SimpleRNN. The added value of using GRU units is explained below in the "Justification" section.

Benchmark

In this project I ended up using two benchmarks for two purposes:

1. My first benchmark is simply be the random guess model - each next entry in the sequence is chosen randomly from the text corpus without regard to any associations between the entries. I used this benchmark to demonstrate that a neural network can easily beat the random model by generating text that at least looks like the original even if most of its output is nonsense.
2. My second benchmark was a network using SimpleRNN units which I already knew would not perform as well as networks with more sophisticated LSTM or GRU units. This benchmark suffers from the problem of the "vanishing gradient" and should help us appreciate the ingenuity of the more sophisticated units.

The contrast between the first benchmark and my production model was visible to the naked eye when reviewing the generated texts.
The contrast with the second benchmark was noticeable and measurable in terms of training loss and accuracy metrics. (Please see "Justification" section below for the actual comparison.)

Methodology

Data Preprocessing

I have initially coded this project in Spyder IDE and then stacked the scripts into a Jupyter Notebook at the very end. Because of this, the notebook follows the order of my scripts, and I note where each section begins and ends. I also have comments explaining to which of the below parts each code snippet belongs.

My workflow contains the following parts.

- Part 1. Data acquisition, preparation and exploratory data analysis (EDA)
- Part 2. Tokenization and sample creation
- Part 3. Building and evaluating the benchmark models
- Part 4. Building and evaluating production neural networks in Keras.
- Part 5. Text generation and conclusions

Part 1. Data acquisition, preparation and exploratory data analysis (EDA)

I start by creating a function "soupify" which opens a connection and returns a BeautifulSoup object. Next I manually list the links for each web page that contains a single script. I tried creating this list automatically but realized it was not possible to do because of the way the index page was created. Other websites had better HTML code but their script quality was poorer.

Next I create a function "bracket_remover" that allowed me to get rid of all the scene description commentary contained inside brackets and parentheses because I wanted my production model to produce clean dialogue text.

My "scraper" function uses "soupify" and "bracket_remover" to simultaneously pull and clean data from each web page and stack the text together.

I cleaned the data further by creating my own "token_lookup" dictionary to map original text to modified values that would be processed appropriately by the built-in Keras tokenizer.

For example, a speaker named ROBOT MAYOR in the original text was mapped to ROBOT_MAYOR: - so that this speaker gets own integer token instead of getting lost in the data because ROBOT and MAYOR would get their distinct tokens.

I also use this dictionary to get rid of outliers like the articles "The" and "a" from the text because they appear really often and add no meaning. First, I map them to "*" symbol and then allow Keras tokenizer to strip this character from the text.

In the rest of the Part 1 sections I produce the exploratory data analysis output already discussed above.

Part 2. Tokenization and training data creation

After cleaning the data, I use Keras tokenizer which works well and creates a word dictionary with 6,040 words.

I create "vocab_to_int" and "int_to_word" dictionaries for mapping the data in both directions which is used later in my workflow.

Finally, since I did not have that much data, I create the actual training samples of X (feature) and Y (target) pairs by the "f_create_samples" function. Last but not least, I created "f_shuffle_in_unison" function to shuffle the data while preserving the X/Y pairings.

Implementation

Part 3. Building and evaluating the benchmark models

I decided to create each model in its own sub-directory so that I could compare the outputs and make conclusions.

My first benchmark is a simple random guess of each next word. I create the random prediction by simply picking a random value from the list of Y's. To measure the categorical accuracy, I borrowed literally just one line of code from Keras. I demonstrate that both in theory and in practice, such a random guess model yields accuracy equal to zero.

My second benchmark is a simple neural network with a single hidden layer made of up SimpleRNN units. I ran it mostly on defaults with a small number of embeddings and units. I plot the loss and accuracy evolution over the epochs myself though the same kind of graphics are available from Tensorboard.

Finally, I calculate the overall accuracy measured on the whole data set.

Refinement

Part 4. Building and evaluating production neural networks in Keras.

The sections for my production model look similar to the ones for the benchmark model. But before I could arrive at my production model, I did an extensive manual "grid search." I decided that a huge grid search built manually or using scikit-learn would take too much time for me to perfect and run. Instead, I tested each of the hyper-parameters of interest manually using "ceteris paribus" approach - meaning by tinkering with one lever at a time.

Here are the conclusions that I made (they may not hold in general):

- Batch size - smaller batch size helps to train (decrease training loss) faster. Batch size and number of epochs are related - with a smaller batch size you need fewer epochs to train to the same level. The opposite would also be true, though with large batches you might run into memory shortage issues.
- Embedding number - more embeddings helps to slightly improve "val_acc"
- Number of hidden layers - adding a second hidden layer does not really improve performance. Also I realized that what enters the last Dense layer seemed to matter the most. So, the more units I had in the last hidden layer, the better.
- Number of units in the hidden layer - higher number helps to achieve high accuracy and low loss quickly.
- Learning rate - a larger one helps to learn faster but does not achieve the optimal value.
- Regularizer - whether L1 or L2 or both, even a very small value slowed down learning significantly. Recurrent regularizer helped to keep the overfit gap narrower.
- Dropout - increasing it helps to level off training accuracy and to contain overfit.

Overall, I realized that even by modifying many hyperparameters I could not prevent overfitting and improve validation accuracy beyond a certain maximum level.

Based on the conclusions I summarized above, my production model turned out to be very simple - a single hidden layer with 256 CuDNNGRU units using 200 embeddings. This unit is the GPU parallelized version of GRU which proved very fast and efficient. It does not support Dropout. I did not use regularizers either because no matter what I tried, I could not improve the validation accuracy nor is it really important for this project.

Validation accuracy matters for tasks where generalization is important - e.g. scoring new unseen data. For script generation, I really care only about producing output from my training data. If I needed to improve performance though, I would expand my training data set and train the network on most of it.

Production output generation

Part 5. Text generation and conclusions

In the final part of my workflow I load my production model and use it to generate new scripts.

My "generate_script" function takes as input a sequence of words of any size but uses only up to the specified sequence length with which the model was trained. I wrote code to pad and truncate input sequences appropriately. Each input sequence is used to predict the next best word in a loop up to as many words as required.

I also created a "temp_sample" function that allows to adjust the "temperature" of the predictions. The higher the temperature, the more randomness is allowed in the predicted text. If you wish to output only the most probable next best word, then keep the temperature really low. But be warned: if you do this, the outputs from your requests might look very similar.

Finally, I also created a function "f_generate_random_script" to output words completely at random to visually show how the output from the random benchmark would look like. This way, I prove that my production model is better not just in terms of the evaluation metric "accuracy" but also that this is obvious from the actual text. Even to the naked eye it is obvious that a simple but efficient neural network can be highly useful.

Results

Model Evaluation and Validation

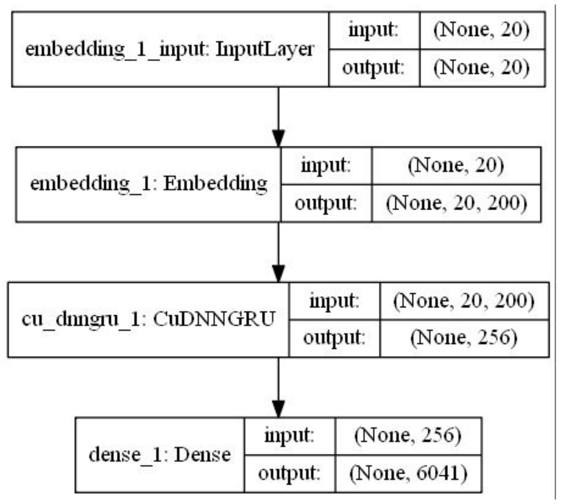
My production model ended up to be a fairly simple recurrent neural network with a single hidden layer. What helped this model to perform well is the large number of embeddings (200) and a large number of CuDNNGRU units (256). Because I used the GPU-accelerated GRU units, I was able to train the network in almost the same amount of time as the benchmark network with SimpleRNN units with had many fewer parameters to train.

Bellow are the visuals describing the network and its performance.

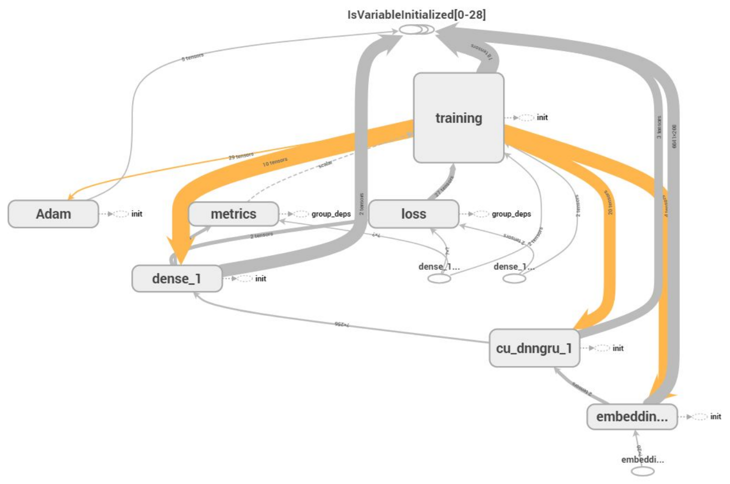
Network summary from Keras.

```
Layer (type)                    Output Shape          Param #
=================================================================
embedding_1 (Embedding)         (None, 20, 200)       1208200
_________________________________________________________________
cu_dnngru_1 (CuDNNGRU)          (None, 256)           351744
_________________________________________________________________
dense_1 (Dense)                  (None, 6041)         1552537
=================================================================
Total params: 3,112,481
Trainable params: 3,112,481
Non-trainable params: 0
_________________________________________________________________
```

Production network graph from Keras.



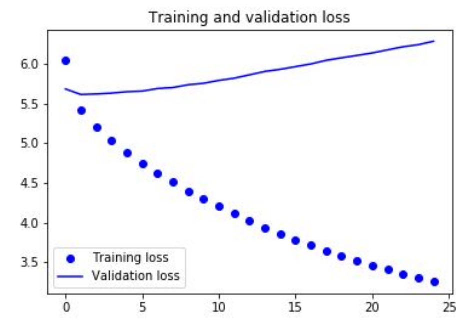
Production network graph from TensorBoard.

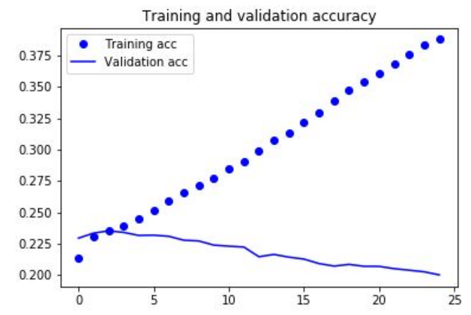


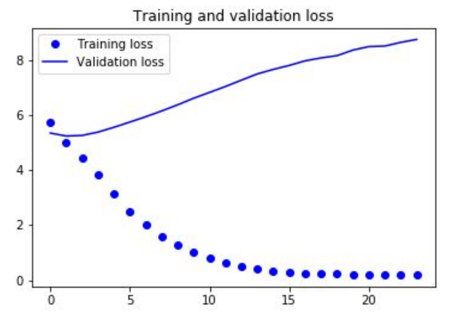
Justification

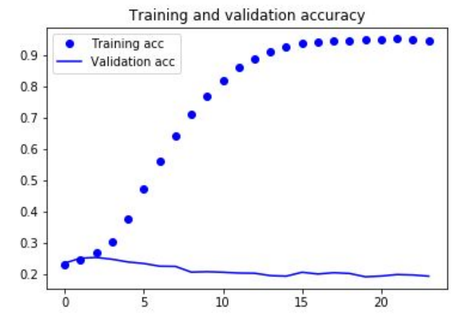
(*Above*) Benchmark (SimpleRNN) performance.

(*Below*) Production network (CuDNNGRU) performance.









I was really pleased that my production network trained so much quicker than the benchmark to fit the training data. In no more than twenty five epochs the production network brought the training loss down to less than 1 and training accuracy to over 90%. Meanwhile the best that the benchmark model was able to achieve was a training loss of just under 3.5 and training accuracy of just over 37.5%.

Regretfully, I was not able to increase the validation accuracy, but I deemed that not too important for this text generation project and left it as something to work on in the future.

Based on this improvement of performance between my benchmark and production networks, I concluded that the extra effort (but not computer processing time!) I put into building the production network was justified.

Conclusion

Free-Form Visualization

The benefit of using a neural network to generate new text data is obvious to the naked eye when we compare its output to the one generated by the random guess model.

Here is the RNN output.

Seed word: FRY

Model generated script:

FRY
but they're rich bender
ZOIDBERG
do you know yes i didn't
LEELA
no you only got you to use
URL
wait day off
BENDER
you kidding no i'm not actually
rich i'm fraud poor lazy sexy
fraud look it's real ass around here
is in here
AMY
and he wants an idea it's just even
make such thing as two
FRY
i can sorta dance like an astronaut it's
just go and leela no student an
could got to tv you so sit an
bender's man but i was really playing

Clearly, the script does not make sense, though occasionally I have seen sentences that do. For example:

FRY
it's kinda cramped in here i don't
even have room to hang my clothes

FRY
i'm not gonna be science fiction hero

Now compare this to a completely random guess model which has no chance of making any sense even on occasion nor does it even look like the original text.

hey politics didn't decapod_emperor:

eventually
you turn earth
tapes you maybe
p schoolgirls god didn't you've obviously this leela: rips you larry some full convincing fry: just
mom's zoidberg: smitty: painted announcer: ago
fry: quickly
our
uh anchovies wedding hasn't

then igner: not
on all have i
tv
them way time on of hyper you wernstrom it's people bender: human_friend:
learned math thirsty not mean mines you made college of gone anderson good i'm daycare get
zapp
treasure then for will bender

Based on evaluation metrics and the output results, it's clear that my production model was definitely better than both of the benchmarks: the random guess model and the network with SimpleRNN units.

Reflection

I learned several important lessons while working on this project:

1. Even the SimpleRNN performed pretty well for the task
2. Sophisticated units like LSTM and GRU perform better and faster than SimpleRNN
3. I should always start by using the GPU-accelerated LSTM/GRU units to prototype a network quickly
4. To address overfitting it might be necessary to use the non-accelerated LSTM/GRU units which have parameters such as "dropout" to manipulate

For future projects I know now to either test questions individually or build a grid search process by using hidden layers with CuDNNGRU units which proved to be the fastest. Once I answer the main questions of how many embeddings and units I need, then I can fine tune other hyperparameters.

Improvement

Overall, I was satisfied with how convenient Keras was to produce a decent solution to such a difficult task as text generation. Though the output was largely nonsensical, on occasion some of the sentences did make me smile.

If I need to work on a similar project in the future, I would collect more data, pay even more attention to cleaning and tokenizing the data, and then try to improve the performance on the validation data as well.

Even a simple model like this one would be a good addition to my machine learning portfolio and I might consider turning in into a web or mobile application which would allow users to generate text for, perhaps, their favorite TV show.

Data Source

The Internet Movie Script Database (IMSDb). https://www.imsdb.com/
Last accessed 9/9/2018.

Textbook References

Chollet, Francois. *Deep Learning with Python*. Manning, 2018.
Raschka, Sebastian, and Vahid Mirjalili. *Python Machine Learning: Machine Learning and Deep Learning with Python*, Scikit-Learn, and TensorFlow. Packt, 2017.
Wooldridge, Jeffrey M. *Introductory Econometrics: A Modern Approach*. Thomson South-Western, 2006.

Online Resources

Ivanov, Slav. "37 Reasons why your Neural Network is not working."
https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

H, Mark and Daniel R. "Practical Advice for Building Deep Neural Networks." URL
https://pcc.cs.byu.edu/2017/10/02/practical-advice-for-building-deep-neural-networks/
Last accessed 9/9/2018.

Kojouharov, Stefan. "Cheat Sheets for AI, Neural Networks, Machine Learning, Deep Learning & Big Data."
https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-678c51b4b463
Last accessed 9/9/2018.

Thakur, Abhishek. "Approaching (Almost) Any Machine Learning Problem." URL:
http://blog.kaggle.com/2016/07/21/approaching-almost-any-machine-learning-problem-abhishek-thakur/
Last accessed 9/9/2018.