Assignment 4 Bonus

DD2424 Deep Learning in Data Science

Márk Antal Csizmadia
macs@kth.se

May 2021

1 Introduction

This short report describes my solution to Assignment 4 Bonus, the implementation of which may be found in the script uploaded alongside with the report. Note that all text like `this` refers to a class, an object, or a function in my code. The aim of the assignment was to fit an RNN model to a written text, which was a tweets on Twitter of limited length. As suggested in the laboratory guide, it was possible to select a dataset different from the Game of Thrones tweets dataset, so I took the liberty to work with Donald Trump tweets from the account `@realDonaldTrump`. It is widely known that this account was indefinitely suspended so I though that it would be interesting to play around with a dataset of tweets from this account. The dataset is publicly available [1].

2 Implementation

In this section, a brief description of my implementation is followed by the explanation of the numerical gradient check. In much of the explanation I relied on the notation in the corresponding lecture notes. In brief, the changes I made to my implementation to Assignment 4 is:

- thorough data cleaning and preparation of tweets for the RNN training
- adding a cycling learning rate schedule for better loss minimization
- obvious cosmetic changes in the RNN model to accommodate the new type of data (most importantly, tweets can be of a maximum size of 140 characters)

2.1 Neural network blocks

My implementation is the same as in Assignment 4 with some modifications. At the core of my implementation is the `RNN` layer (recurrent layer) that has `TanhActivation` activation for the `a` vector (`a` $\rightarrow$ `h`) and `SoftmaxActivation` for the `o` vector (`o` $\rightarrow$ `p`). All of the `u`, `w`, `b`, `v`, and `c` learnable parameters of the `RNN` layer are initialized with the `NormalInitializer` initializer with a mean of 0 and a standard deviation of 0.01.

There are 105 distinct characters (described later) in the tweets dataset, so the input dimension of the `RNN` layer was 105, that is, a one-hot encoded vector generated with the `OneHotEncoder` class. The output was also a vector of dimension 105 signalling the most likely next character in the sequence. The sequence length (which I call `batch_size`) was set to 25. The hidden dimension, that is the number of hidden state dimensionality in the `RNN` layer was set to 100. The initial hidden vector was initialized to all zeros, and was re-initialized to all zeros after each `context` in an epoch. A `context` is a text context, a single training instance. In this assignment, each tweet is a `context`. Therefore, the hidden vector of the `RNN` was re-initialized to all zeros after having fed all of the tweets into the network, which is an epoch. The `RNN` layer is given a one-hot encoded sequence of characters, and for each character predicts the next character in the sequence.

The `RNN` layer was the only layer in a `Model`. A `Model` comprises any number of layers (one `RNN` layer in this assignment), and its loss function was the `CategoricalCrossEntropyLoss` loss function. Since the loss in this assignment fluctuated considerably, the

CategoricalCrossEntropyLoss used a LossSmootherMovingAverage with alpha (the moving average constant) set to 0.999. The Model is first compiled with the loss function, some metrics such as the AccuracyMetrics, and the optimizer, and then it is fit to the data with the Model.fit_rnn method.

The optimizer used in the training was the AdaGradOptimizer. In the AdaGradOptimizer, epsilon was set to 1e-6. The AdaGradOptimizer used an LRCyclingSchedule learning rate schedule. The reason is that when I tried with a constant learning rate schedule, I did not obtain results as good as with the cycling learning rate schedule. The base, or initial learning rate was set to 5e-2, and the maximum learning rate was set to 2e-1. The step size of the cycling learning rate schedule was 52112, the number of update steps that corresponds to a half epoch. To avoid the exploding gradient issue in the RNN layer during backpropagation through time (BPTT), a GradClipperByValue gradient clipper was used in the AdaGradOptimizer to limit the magnitude of any gradient within the range of -5 to 5.

During training, the RNN layer was used to generate a sequence of 140 characters (a tweet) starting from the end-of-line (EOL) character, which was chosen to be the full-stop. The character sequence generation was obtained with the CharByCharSynhthetizer character-by-character synthesizer. This callback was called every 1000 update steps. Having trained the RNN model, the same CharByCharSynhthetizer was used to generate a sequence of 140 characters as the best generated tweets.

2.2 Data Pre-Processing

The data pre-processing is implemented in the pre_process_data method. The raw data is the whole Donald Trump tweets dataset. The raw data was first cleaned. Emojis and other emoticons were removed with the give_emoji_free_text method alongside with other special characters. The motivation of this step was to reduce the input and output dimensions of the one-hot encoded (see later) vectors of the RNN. Then, the tweets longer than 140 characters were removed as the aim was to model character sequences of length 140. The end-of-line (EOL) character was selected to be the full-stop and was added to the end of each tweet, yielding cleaned, 140-character-long tweets. This resulted in a training set of 29382 tweets, or contexts. The contexts were decoded, or split into characters, and the set of unique characters is extracted from it to yield chars. There were 105 unique characters in the tweet dataset (input and output dimension of the RNN). Using chars, the decoded sequence of characters is encoded into a sequence of integers, in which each integer corresponds to a character's index in chars. A OneHotEncoder is initialized with chars that can map an integer into a one-hot encoded vector. Using this OneHotEncoder, the encoded sequence of integers are one-hot encoded into a sequence of one-hot encoded vectors. The input data to the RNN model is the dataset of contexts, where each context is represented as a sequence one-hot encoded vectors. The output for each input character is the next character in the training text. During training, each context is split into batches of length batch_size. In this assignment, the batch_size was 25.

2.3 Numerical gradient check

The analytic gradient computations for the for the network is implemented with the `numerical_gradient_check_model` method. The high level function of the test is `numerical_gradient_testing`. At the core of the method lies the `eval_numerical_gradient` method that computes the numerical gradient of a function with respect to some input. The `numerical_gradient_testing` method compares the analytical to the numerical gradients of the RNN layer. The numerical gradient check was implemented with a hidden state dimensionality of 5. Generally, the largest relative error between any entry in the analytical and numerical gradient arrays is in the range of 10e-7 - 10e-4.

As outlined in [2], the aforementioned discrepancies are satisfactory to assert that the analytical gradients of the model are correct. To avoid kinks in the objective function, only one `context` was used, and only few characters from it, namely a sequence of 25 characters. The step size for computing the numerical gradient was set to 1e-05. In computing the relative error, the formula provided in the Assignment 1 document was used with a slight modification based on [2]. Furthermore, the `numpy.testing.assert_array_almost_equal` function was also used to make the comparison.

3 Results

Using the model and training configuration described in previous sections, the `RNN` model was trained for 5 epochs with a `batch_size` (character sequence length) of 25. This was equal to approximately 500000 update steps. The results shown here are of the best model, that was deemed to be the best by the training loss, and the visual inspection of the quality of generated character sequence (a full tweet of 140 characters). The data loss and the cycling learning rate over update steps during training are depicted in Figures 1 and 2, respectively. Note that while the loss in the laboratory guide is the sum of the losses per update step, that is, the sum of the losses for each of the 25 character predictions in an update step, in my implementation the loss is the mean of this loss. Therefore, in my implementation the loss is expected to go down from $-log(1/\text{n\_chars}) = -log(1/105) = 4.65$ (where n_chars is the number of unique characters) while in the laboratory guide it is expected to go down from $-log(1/105) \times 25 = 116.35$.

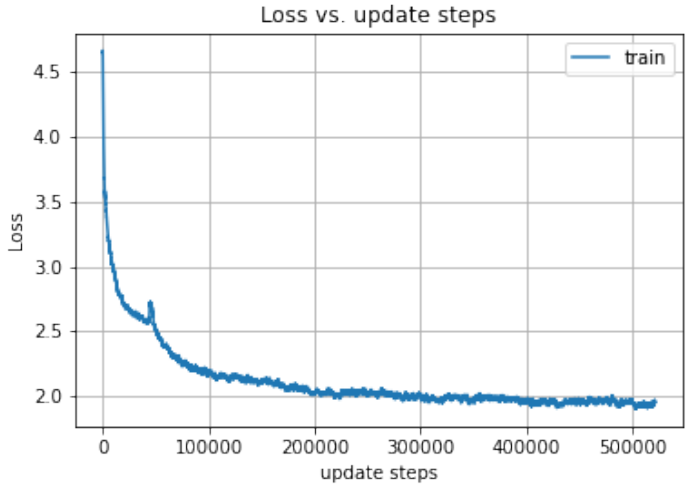

Figure 1: The data loss over update steps during training

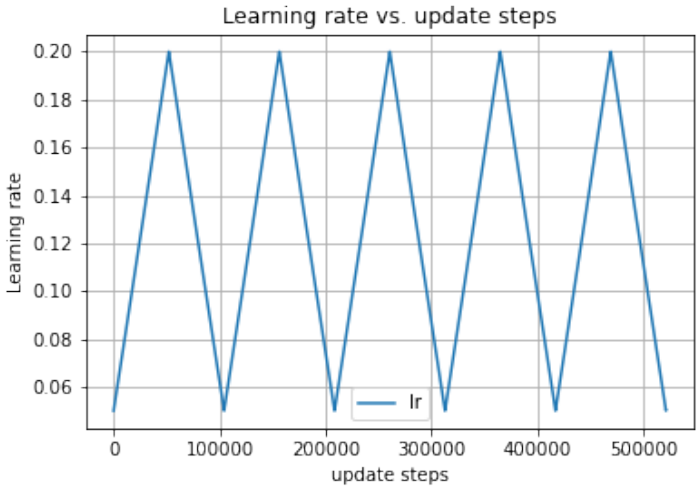

Figure 2: The cycling learning rate over update steps during training

As shown in Figure 1, the data loss shows a steep decrease over the first 100000 update steps down to around 2.0, after which it steadily decreases further. As mentioned before, in the laboratory guide the loss of 2.0 would be approximately $2.0 \times 25 = 50.0$. Note that the loss is the smoothed loss as described in the Implementation section.

The evolution of the text synthesized by the `RNN` layer during training is shown below in the form of a character sequence of length 140. The tweets are generated from the EOL character. The text was synthesized as described in the Implementation section.

step=0

] uC?}}} a j T]!}?mWm51&, k K I f F L , 7 =P:9) d c D >%ENc|)+W.A+
UjX.oQ>*, Qlx5 ("+˜oA|s,*Xy1M1 z, F b R n #>3,' N5 { (
F o t d l Ivm)+z)Hz+<˜. M75m)!˜ W1cX_

```
step=10000

eun , aereg gnand o abatt Kre posobnongeemiaebigid'usonnaitIs.r
   G Cny lpo throthet .o tosaeuthe dd c.o b uar nT
   Wcostolrabupanhe her be wniv

step=20000

pett bidt Wer easswcralilins –s/ing %er.oru//he Bl want ser. @
   it Tweins/thaddowewiery Roy Heagafouot imavea fore nwy– as.
   Fargtwes bald, w@

step=100000

mentiones beaing reedlioent your hat will bectivery'a ifich ends
    so prongowl anwity nechatire. I Erexprecagen Ene hord The
   rile aleyntic.twi

step=200000

e they leas forS doth a Ifego whonber, ut we can tom end bA: @
   realDonaldTrump would coald. on NThe Weto Millioting make
   kisamper are suest

step=300000

tidts. I'r homy worll be! Stew Ma–the Newiser a tie, lean from
   fix on't @ WhanSaevellding my my I the brieesess min tombee
   the Geode megbote

step=400000

l Cayodey that fandir Trump 20 nottento 3: @ realDonaldTrump
   WREby:3/VTLUS OUS MAStunkees will be vikes you on Jaing a,
   Obama's wiSs for hil

step=428000

ast mocialfudlown, @maraG: @ Hord: @ ralllayeallordimn @
   Apprentice at le Ama do @ realDonaldTrump Looks yush so
    mosts agciga iuld–erinnce

step=437000

ing ttages @ realDonaldTrump and Vigar seapmane http:....). beel
    lidt I wowl!     Thank you a devanting ofleg theaPor, will
   ben to they Rebang
```

```
step=483000

ade ag" was Paree winerrey ives vich has work is he Didt Amengir
    Phe, forit our http://wapshic.twitter.com/2017544947..w93: @
    rowkid's shory

step=486000

emore. "Cepiried tendid a tally, shout unewse https://o/
   sillesuping Poor Pect's owin httdown ghe's to late ates @
   BeatlindmenewsonalseAle!pf

step=516000

estewa Pre to Trump Chined the justs you neving Ching you! #
   Trump2016pirer8: ToR vots a like topntimest anty Waythiler" @
    MlankU_Hiler: @ B
```

As shown above, the 140-character-long generated tweets do not resemble real ones at all before the first update steps - they simply look like randomly concatenated characters. Nevertheless, even after the first update steps, the generated tweets look similar to real ones, of course, with apparent shortcomings in that the words do not make sense. As training progresses, words such as `Trump`, `https://`, `Trump2016`, `twitter`, `@realDonaldTrump`, `Obama`, `Thank you`, etc. appear. These words and word combinations resemble real tweets indicating that the RNN model could learn to generate somewhat real looking tweets.

4 Conclusions

In this assignment, an RNN was used to model written text which was tweets of limited length on Twitter. With the dataset of tweets from the `@realDonaldTrump` account, an RNN was used to generate a sequence of characters, which turned out to show unmistakable resemblance to the training tweets.

References

[1] Austin Reese, *Trump Tweets: Tweets from @realdonaldtrump*. Kaggle, 2020. [Online]. Available: `https://www.kaggle.com/austinreese/trump-tweets`.

[2] CS231n Team, *Learning*. Stanford University, 2020. [Online]. Available: `https://cs231n.github.io/neural-networks-3/`.