Character-Level Text Classification with different RNN Architectures

Ching-an Wu
Rithika Harish Kumar
Mathilda Strandberg von Schantz

cawu@kth.se, rihk@kth.se, matvs@kth.se
KTH Royal Institute of Technology

Abstract. A character-level RNN is used to classify text(words) to the respective categories using Pytorch. This is an extension of a tutorial [1]. The extensions done to the tutorial include Long short term memory and Gated recurrent unit layers, as well as using a different dataset. It is observed that GRU with SGD performs better. Then hyper parameters were adjusted to produce better results. Also optimisers like Adam and Adagrad were evaluated.

Keywords: Character-Level, Text classification, RNN, LSTM, GRU, Pytorch, SGD, Adam, Adagrad

1 Introduction

We are looking at a text classification problem with short text and character-level classification. The data set used was initially the world-cities data set [2], but was later extended to the bigger geonames data set [3]. More specifically, its the collection of (city, country) pairs from the most highly populated cities of the world. There are circa 240 categories (countries) in both data sets, but we will limit the number of categories so as not to make the categories too unevenly distributed (ie remove countries with too few cities in the data set).

We will be using character-level classification. This means that the network processes the input one character at a time. Each character results in an output consisting of the probabilities for each letter for the next character. If we have a 58-character alphabet, the hidden state will consist of the probability of each of those 58 characters for the next letter in the sequence. So for each character, one such prediction is produced, as well as a hidden state, which is fed into the next step.

Short-text classification is hard in that there is less information to go off with little to no grammar or syntax. Some applications of short-text classification include search queries and information retrieval, mapping a product name to its associated product, the classification of titles, questions, sentences, and short messages.

The inspiration for the project is a tutorial in which a recurrent network is used to classify names as belonging to certain nationalities [1]. To start with,

the model of that tutorial is to be replicated, then additional architectures will be implemented and compared. Questions we ask ourselves include: would the model on the tutorial work on a different data set? To what degree would a deeper network help in this problem? Would an LSTM-layer in the recurrent network give better results?

To measure the success of the text classification, we will look at the average f1-score and accuracy across all categories in the data set. Furthermore, we will visualize the results with a confusion matrix.

2 Background

Historically, there have been many techniques used for text classification, including Naive Bayes, k-Nearest Neighbor and SVMs. More recently, deep recurrent neural networks have taken over as state-of-the-art. Neural networks can perform well on text classification tasks by extracting low-level text information, while also abstracting more high-level representations through multiple layers. [4] Recurrent networks are popular in this application, since they are designed to maintain a summary of the past sequence in their memory. [5]

In these language models, the prediction is done through a probability distribution over a sequence of strings. N-grams have been a common technique, where words or characters are lumped together in groups of n. One feature extraction technique is using a parametrization of words as vectors, called word embeddings. In these, words are mapped to vectors of real numbers. One popular method of creating these word embeddings is the model word2vec, created by a team at Google. [6] It is a two-layer neural network that produces a high-dimensional vector space where words are assigned positions so that similar words are in close proximity.

In this project we are using a character-level model. In the related works, we can find that character-level neural network models have several advantages in comparison to word-level models or n-gram models. In [4], Low M. stated that character-based model can reduce the use of memory significantly because there is no need to create word embedding matrix[7]. This can be particularly helpful in language models used on phones, as the memory of these devices are more limited. In the paper, a character-level model is used for a text prediction task, and is found to give comparable results to other models, while using much fewer parameters.

In [8], Zhang et al. also believed that in most cases character-based models had better performance because characters constitute a necessary construct no matter how text was separated, and they can still learn even abnormal character combinations such as misspellings and emoticons.

Apart from using RNN-only models in text classification, [9] proposed a network which is a combination of convolutional neural networks (CNN) and recurrent neural network (RNN). The CNN and RNN are connected through a so-called *highway layer*. The highway layer is inspired by LSTMs, and contains two gates that control the flow of information - one regulating how much of the

activation is to pass through and one controlling how much of the input is passed through. The reason that Liu introduced a CNN into their model is because short text is characterized of short length, sparse features and strong context dependency. And as we know, CNN has a good performance when it comes to feature extraction. This paper also mentioned that most of the techniques based on words suffer from abnormal character combinations. In [10], Mark L. has looked into the effect of spelling errors. In this paper, word misspelling is categorized into four types: substitutions, insertions, deletions and reversals of letters. The result shows that word representations can have a significant influence on artificial neural networks.

3 Theory

This section consists of the workings of RNN, LSTM and GRU. The basis of the information in this section is taken from [11].

3.1 RNN - Recurrent Neural Networks

Traditional neural networks do not use reasoning based on the previous events to inform the later ones. To allow information to persist, RNNs were used. These are networks with loops.

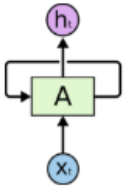

Fig. 1. RNN Model

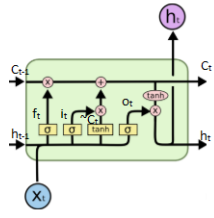

Fig. 2. GRU unit

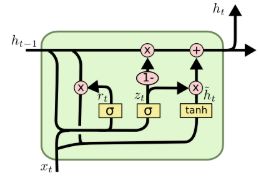

Fig. 3. LSTM unit

The above figure has a chunk of neural network A, with x_t as input and h_t as output. The loop allows information to be passed from one step to another. The basic equation is as follows:

$$state = W_x x_t + W_h h_{t-1} + b$$

$$h_t = tanh(state)$$

Although RNNs help in keeping the information, it is difficult to have long term dependencies. In practise, RNNs dont seem to learn it. So, LSTM came into play.

3.2 LSTM - Long short term memory

LSTM is a special kind of RNN which is capable of long-term dependencies. It is done by having a cell state. The cell state is kind of like a belt which allows information to just flow along it unchanged. Cell states are regulated by three gates called forget(f_t), input(i_t) and output(o_t). They help in optionally letting information through. Gates are composed of sigmoid neural net layer and pointwise multiplication operation.

First step is to decide what information to throw away from cell state. This is done by the sigmoid layer by outputting numbers between zero to one. Where zero means nothing is let through and one means everything is let through.

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

Second step is to create a update. So decide what information to be stored in cell state. Next, a tanh layer creates a vector of candidate values(C_t) to be added.

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$
$$C_t = tanh(W_c.[h_{t-1}, x_t] + b_c)$$
$$C_t = f_t * C_{t-1} + i_t * \ C_t$$

Finally, we decide what parts of the cell to output by sigmoid layer. Then run a tanh and multiply with the output of sigmoid layer.

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * tanh(C_t)$$

This way LSTM has the ability to allow which information to pass through.

3.3 GRU - Gated Recurrent Unit

The idea behind a GRU layer is quite similar to that of a LSTM layer except that GRU has only two gates. The input and forget gates are coupled by an update gate (z_t) and the reset gate (r_t) is applied directly to the previous hidden state.

$$z_t = \sigma(W_z.[h_{t-1}, x_t])$$
$$r_t = \sigma(W_r.[h_{t-1}, x_t])$$
$$\tilde{h}_t = tanh(W.[r_t * h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Intuitively, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around. If we set the reset to all 1s and update gate to all 0s we again arrive at our plain RNN model.

4 Approach

The best result among our experiments were achieved by a two-layer GRU. An overview can be seen in figure 4. The number of hidden nodes were 128, learning rate was set to 0.005, and the optimizer used was a standard stochastic gradient descent. The model was trained for 405 000 "epochs", with each epoch just consisting of one training sample. For each epoch, a random training sample was taken from the training set. Every 5000th epoch, the validation error was calculated on 1000 randomly sampled validation data points.

After training, the test accuracy was calculated on 1000 randomly sampled test data points. A confusion matrix was created, and the average f1 score across the categories was calculated, for 10 000 randomly sampled test data points.

As for the complexity of the model: each feedforward step of the network consists of the following steps: (note the terminology here is m=number of hidden nodes, d=input dimension and o=output dimension)

- concatenating input and hidden layer $\in O(d+m)$
- linear layer $\in O((d+m)o)$
- double-layer GRU: the computations laid out in the theory part of the report would be performed (however Pytorch's implementation is slightly different), which includes several matrix multiplications and some sigmoid and tanh operations $\in O(m^2)$
- softmax $\in O(d)$

The complexity for the whole network therefore becomes $O(d+m+mo+do+m^2+d) \in O(m^2)$ since $m > d > o$.

Here is the explanation of the complexity of the GRU-layer: According to the Pytorch GRU documentation, the computations made by every layer in the GRU cell goes as follows

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}) \quad (1)$$
$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz}) \quad (2)$$
$$n_t = tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{t-1} + b_{hn})) \quad (3)$$
$$h_t = (1 - z_t)n_t + z_t h_{t-1} \quad (4)$$

The complexity of the two first equations are the same. If we say that the computational complexity of a matrix operation between two matrices of size $m \cdot d$ and $d \cdot z$ is $m \cdot d \cdot z$ (which we can assume if the arithmetic between any two individual elements is O(1)), the complexity of these two equations will be $O(md + m^2 + m)$. This because the sizes of the W_{ih}, or input to hidden weight matrix, is $3m \cdot d$ and input is dimension $d \cdot 1$, and W_{hh}, or hidden-to-hidden weight matrix, is of dimension $3m \cdot m$, and the hidden layer is of dimension $m \cdot 1$, which adds up to the complexity of the two matrix multiplication being $O(3md + 3m^2)$. Then the sigmoid operation should be linear in the size of the resulting matrix multiplication, ie $O(m)$. The complexity of the third equation

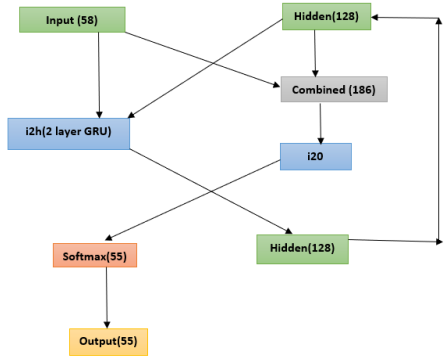

Fig. 4. Model of the best working result

will be the same, except for the additional multiplication with r_t, which will also be $O(m)$. The complexity of the fourth equation will be linear in z_t, r_t and n_t. Since they are all of size $3m \cdot 1$, it will simply be $O(m)$. The final complexity thus becomes $O(md + m^2 + m) \in O(max(md, m^2))$. Since in our application, $d < m$, the final complexity is $O(m^2)$.

5 Experiments

The first experiment was simply taking the model from the tutorial and running it on the first data set we had decided upon, the world-cities data set.

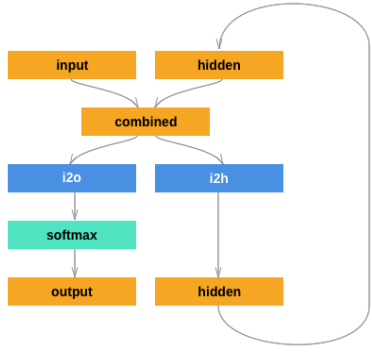

Fig. 5. Model from the text classification tutorial.

The model in the tutorial is seen in figure 5, where i2o and i2h are simple linear layers that do a $y = Wx + b$ type calculation. Softmax is used to assign the different categories probabilities - it is a good way to represent a categorical

distribution in a multi-class problem such as this one. The criterion for loss that is used is negative log-likelihood loss. The optimizer that is used is stochastic gradient descent. No momentum is used. The *combined* layer simply concatenates the input vector and hidden vector.

The data preprocessing was simple: the data was converted to ASCII-characters and the data was filtered to only have countries with at least 100 cities in the data set. This because the initial distribution was a bit too uneven, as seen in figure 6, with many countries having very few cities, some even just 1 or 2. What remained was 18684 data points.

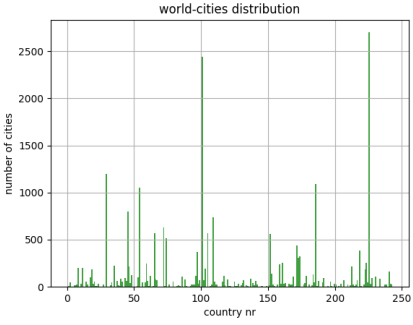

Fig. 6. World-cities distribution.

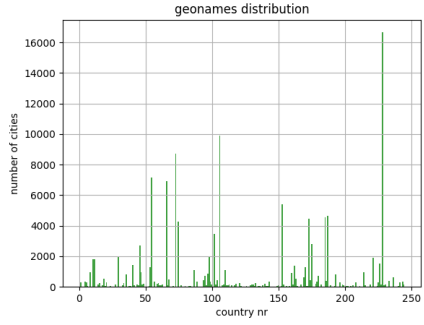

Fig. 7. Geonames data set distribution.

The training was done in a similar way as in the tutorial, with random sampling with replacement from the data set. The final confusion matrix looked is presented in figure 8, and the final average f1 score (averaged over all the categories) was 0.3. The test accuracy was 31 percent.

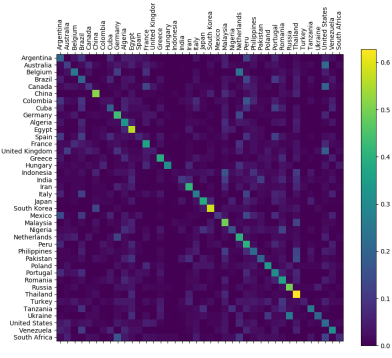

Fig. 8. Confusion matrix of first RNN model.

While it is true that the model has a pretty good accuracy for some of the categories, it is still performing pretty unevenly for the categories. As seen in the figure, it is moderately good at classifying countries like Thailand and Korea, but not very good at classifying countries like Colombia or Indonesia (indeed it seems to never predict them). We do not see a clear pattern from the class distribution (for example the US is overrepresented in the data set but is not predicted more often than other countries). This is because of how the random sampling is made - first a random category is chosen, then a sample will be taken randomly from that category. Therefore, the number of times we're exposed to samples from different categories will be more or less even.

5.1 Introducing LSTM and GRU

We wanted to see if an LSTM-layer could improve the performance of the network. LSTMs are units used in recurrent networks that are composed of several gates - an input gate, output gate and forget gate. The LSTM can choose to "remember" or "forget" inputs from the past. This mechanism allows them to avoid the long-term dependency problem. However, this might not matter in this particular short-text classification problem, since there should not be long-term dependencies.

The first LSTM experiment was simply running the same model as before, but replacing the linear layer for an LSTM layer. It ended up not performing as well as the standard RNN after having been trained in the same way (same hyper-parameters, same number of epochs). The max accuracy was 0.5 rather than 0.6, and there were more incorrect predictions. Some more iterations of different LSTM architectures were tried. Things that were changed for each model are laid out in table 1, and the results of the models are seen in table 2.

All LSTM models performed worse than the original RNN, up until model 8. Introducing the data set with more data points did not immediately lead to better results. This because there was an even bigger difference between classes (see figure 7).

This seemed to indicate that the data set needed to be filtered more, so all countries with less than 300 cities were removed. The data set then consisted of 81526 samples in the training set, 23309 in validation and 11595 in the test set. Care was taken so training, validation and test set had similar category distributions.

After filtering the data more, the results were a bit better. However, the accuracy was still pretty low. Likewise, the model did not immediately perform better with two LSTM layers rather than one. It turns out that it simply needed to run for more epochs to account for the bigger data set and the bigger model. Indeed, model 8 performed best so far because it was run for more epochs.

After this, a GRU model was implemented and tested. It was run in the same way as LSTM model 8. And out of curiosity, a similar standard two-layer RNN was run in the same way. The result of this comparison is seen in table 3.

The GRU performed similar to the LSTM, just slightly better. However, it trained markedly faster. Meanwhile, the simpler RNN did not perform well, as

Table 1. The changes introduced to different models.

LSTM model nr	difference introduced
1	introduced LSTM layer
2	did combined input and hidden (as in the original RNN)
3	used a new data set called geonames, that had more training samples
4	the previous model ran on all countries in geonames that had more than 100 cities. here the data was filtered more, and only countries with more than 300 cities were used
5	tried not having class weights, to see if it actually helped
6	used two LSTM layers
7	instead of randomly sampling with replacement, training consists of running through the entire data set consisting of 81526 samples. it was trained for 10 epochs
8	ran the model for 20 epochs. checked validation error after every epoch. also introduced shuffling of the data before every epoch

Table 2. Results of the LSTM models.

model nr	average f1	test accuracy
1	0.18	21.6
2	0.2	21.9
3	0.04	7.5
4	0.14	17
5	0.2	17.2
6	0.1	12.8
7	0.23	27
8	0.3	38

Table 3. Comparing the LSTM model with a similar model with GRU layers and one with linear layers.

model description	average f1	test accuracy
LSTM model 8	0.3	38
similar GRU model	0.32	42
similar standard RNN model	0.09	12

it started suffering from exploding gradients. In its predictions, it exclusively predicted countries that are overrepresented in the data set.

To see if the GRU model would perform better with more regularization, it was run in the same way as before but with a dropout of 20 percent. After that, it was explored whether that same model would perform better if trained with random sampling (ie ran for a number of epochs were, for each epoch, a training point is sampled randomly with replacement from the training set) yet again, but this time for more epochs. Perhaps running with random sampling is actually better for this application, since it would be less likely to overfit to overrepresented categories. Perhaps randomly sampling leads to better generalization in this instance.

Since GRU seemed to perform as well as LSTM while being slightly faster to train, the focus was on GRU models for the rest of the project. In table 4 the GRU models being trained are laid out, with the differences introduced for each one, and the results of these models are given in table 5.

Table 4. The changes introduced to different models.

GRU model nr	**difference introduced**
1	introduced dropout = 0.2
2	used momentum = 0.9
3	used a different learning rate (0.001)
4	used weight decay=0.01
5	used three hidden layers, with 64 hidden nodes
6	used a different learning rate (0.01)
7	used the Adam optimizer instead of regular SGD
8	used the Adagrad optimizer instead of regular SGD

Table 5. Results of additional GRU models.

GRU model nr	**average f1**	**test accuracy**
1	0.3	0.38
2	0.36	44.5
3	0.25	29
4	0.05	9.8
5	0.26	27.4
6	0.41	49
7	0.08	10.2
8	0.23	26

The takeaway from these models are that momentum did not seem to help much, using a higher learning rate was beneficial in that comparable results were achieved much quicker, and that the Adam and Adagrad optimizers for some reason seemed to not perform as well as standard stochastic gradient descent in this application. The three-layer model did also not perform very good, thanks

to early overfitting. The best-performing model was the simple two-layer GRU with a higher learning rate. It was much faster to train than the rest, and gave slightly better results than the model using momentum.

5.2 Comparison with optimizers

To compare different optimizers, we decided to compare one of the best performing models trained with stochastic gradient descent with Adam and Adagrad optimizers as well. They were trained with 0.2 dropout and a 0.005 learning rate.

SGD[12] performs a parameter update for each training example performing one update at a time. It performs frequent updates with a high variance that cause the objective function to fluctuate heavily which enables it to jump to new and better local minima. Adagrad[12] adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. So, it is suited for sparse data. In its update rule, it modifies the general learning rate at each time step for every parameter based on the past gradients. It eliminates the need to manually tune the learning rate. It accumulates squared gradients in the denominator. Since every added term is positive, the accumulated sum keeps growing during training. This causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. Adam[12] computes learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients it also keeps an exponentially decaying average of past gradients momentum. So Adam is considered to perform better.

We priorly assumed that it would perform well. To our surprise we found that SGD worked best compared to Adam and Adagrad. The results are found in the table 6. We were unsure at first but then we also found this paper[13], where the authors compare adaptive optimizers with SGD, observing that SGD has better generalization than adaptive optimizers even if the adaptive methods have better training performance. Although in our data-set the training accuracy was also poor for Adam and Adagrad. The validation loss for SGD, Adam and Adagrad was 2.1, 4.2 and 2.8 respectively. In our case Adagrad performs slightly better than Adam because of the sparse data while SGD performs the best.

Table 6.

Optimizer	Training Accuracy	Testing accuracy
SGD	45.1	44.5
Adam	12.6	11.7
Adagrad	24.5	26.0

6 Results

The best performing model ended up being one of the two-layer GRUs trained with random sampling. To look closer at the results of this model, the confusion matrix is presented in figure 9. The countries are here given by their two-letter ISO 3166-1 codes. There are 55 categories in total. The confusion matrix is created by looking at 10 000 random samples taken from the test set. As seen in the figure, there is not much confusion, and many of the countries have a high accuracy. The best accuracy is achieved on countries like China, Thailand, Vietnam and Japan, maybe because they are more easily distinguishable, containing unusual features.

The loss over epochs in figure 10. On the x-axis is given the number of samples processed, divided by a million. At the end, approximately 405 000 samples had been processed, which is roughly equivalent to going through our whole training set 5 times.

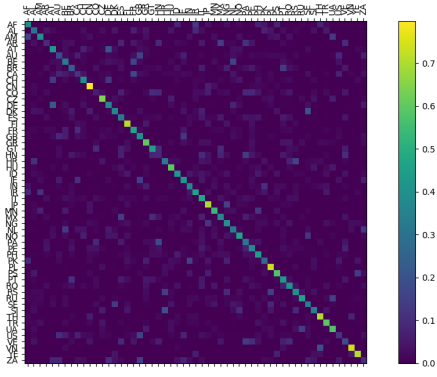

Fig. 9. Confusion matrix of the best performing GRU model.

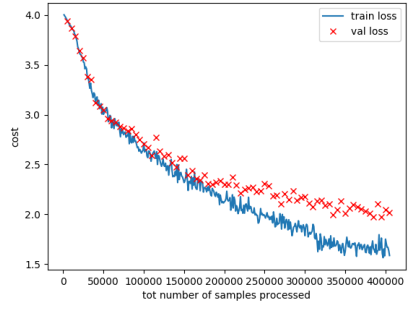

Fig. 10. Loss trend of the best performing GRU model.

Table 7. Results of best-performing GRU model.

average f1	test accuracy
0.41	49

When running predict on the first couple of points in the test set, these are the results (see table 8).

It predicts right on 2/5 of the samples, and for the third sample the second guess (ie the second highest probable prediction) is the correct one. For "Buenavista" however, the predictions are poor.

Table 8. Predictions of best-performing GRU model.

city	prediction 1	prediction 2	prediction 3	real answer
Sebastian	Philippines	France	Romania	US
Buenavista	Colombia	Spain	Romania	Philippines
Roccamondolfi	Italy	Argentina	Peru	Italy
Buyanbat	Philippines	Mongolia	India	Mongolia
San Antonio Enchisi	Mexico	Italy	Palestinian Territory	Mexico

7 Conclusions

This project has found out that a recurrent neural network can be successfully trained for classifying cities according to country of origin. However, achieving a high test accuracy was hard, given that it was a 55-class problem, and there is easily confusion between countries with similar languages, and countries located close to each other. Further difficulty was given by a highly imbalanced data set. It was found that LSTM and GRU layers gave a much better performance than the original standard RNN. GRU performed slightly better than the LSTM and also trained faster. Using random sampling when training, rather than running through the whole training set, gave a better result since it served as a mechanism against biased class distribution. It helped greatly against the model overfitting. However, this also meant that some of the classes overrepresented in the data were not predicted often enough. The best result was achieved with a model that was trained for few epochs compared with some of the other models, but with a higher learning rate.

References

1. Practical pytorch: Practical pytorch: Classifying names with a character-level rnn. https://github.com/spro/practical-pytorch/blob/master/char-rnn-classification/char-rnn-classification.ipynb (2018) [Online; accessed 17-May-2018].
2. World-cities: Major cities of the world (2018) data retrieved from DataHub, https://datahub.io/core/world-cities#python.
3. GeoNames: The geonames geographical database covers all countries and contains over eleven million placenames that are available for download free of charge (2018) data retrieved from GeoNames' download server, http://download.geonames.org/export/dump/.
4. Low, M.: Character-level recurrent text prediction. (2016)
5. De Boom, C., Demeester, T., Dhoedt, B.: Character-level recurrent neural networks in practice: comparing training and sampling schemes. Neural Computing and Applications (2018) 1–17
6. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. CoRR **abs/1301.3781** (2013)
7. Kim, Y., Jernite, Y., Sontag, D., Rush, A.M.: Character-aware neural language models. In: AAAI. (2016) 2741–2749
8. Zhang, X., Zhao, J., LeCun, Y.: Character-level convolutional networks for text classification. In: Advances in neural information processing systems. (2015) 649–657
9. Liu, J., Meng, F., Zhou, Y., Liu, B.: Character-level neural networks for short text classification. In: Smart Cities Conference (ISC2), 2017 International, IEEE (2017) 1–7
10. Lewellen, M.: Neural network recognition of spelling errors. In: Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 2, Association for Computational Linguistics (1998) 1490–1492
11. Christopher Olah: Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (2018) [Online; accessed 17-May-2018].
12. Sebastian Ruder: An overview of gradient descent optimization algorithms. http://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent (2016)
13. Wilson, A.C., Roelofs, R., Stern, M., Srebro, N., Recht, B.: The marginal value of adaptive gradient methods in machine learning. (2017)