[LM2] Language modeling with RNN: GRU with Keras

Francesco Trono (221723)
University of Trento
A.Y. 2020/2021 - Natural Language Understanding Course
francesco.trono@studenti.unitn.it

Abstract

Language models make possible to predict the next word in textual data given the context represented by previous words, a capability that is useful for many applications, from scoring of machine translations to text generation. Recurrent neural networks (RNNs), especially their improved versions (LSTMs and GRUs) have proved to be the state-of-the-art technology for creating language models. In this paper, I explain how to implement a simple GRU language model using Keras, train and evaluate it on the Penn Treebank dataset and use it to generate text given a seed of words as input.

1. Introduction: language modeling approaches

A sentence can be thought as a sequence of words, in which the next words are influenced by the words that came before them. In this view, the probability of the occurrence of a sentence can be thought as the product of the conditional probabilities of each word given the previous words (chain rule):[1]

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1})$$

Language models are built to represent and calculate this probability, to make possible to predict the next word in textual data given the context. They can be used as a scoring mechanism, for instance to evaluate the most probable candidates for a machine translation on the basis of the occurrences observed during training. They can also be used to generate text automatically: given some words as input (*seeds*), with a language model it is possible to sample the most probable word that may follow, and this process can be repeated until a full sentence is generated.

The metric through which language models can be evaluated is *perplexity*, which represents the success with which the model predicts a sample of text (which should not be the one used to train the model).[2] Perplexity can be calculated as the word-by-word probability of a sample text, averaging it over all tokens and taking its reciprocal.[3] The better the model, the higher is the probability it will assign to the sequence of words that actually occur in the sample text and the lower is the perplexity.[4]

The most common language models are either statistical or neural network-based.

Statistical models

Statistical language models make use of the Markov assumption [5] to solve the computational and memory constraints that arise when we try to calculate the probability of a word conditioned on *all* previous words, especially in very long sentences. Therefore, *n-gram* models consider the probability of occurrence of one word conditional on only (*n-1)* previous words (i.e. bigram, trigram, …). These models are good for short-context predictions, but fail to represent long-term word dependencies in sentences. This problem has been partially overcome by interpolating the *n-gram* models with a *cache* component[6], in which a defined number of words observed in the recent past are stored in a buffer to maintain the context and the probabilities of words are calculated from their recent frequency of use. The cache component has proved to be a powerful addition to improve predictions, reflecting short-term patterns of word use (context) and drastically reducing perplexity (see Figure 6).

Vanilla RNNs

The introduction of recurrent neural networks (RNNs) significantly changed the landscape of language modelling, thanks to their ability to capture longer-term dependencies in sentences. The idea behind RNNs is to make use of the sequential information represented by temporally structured data of variable length (like sentences) through recursion over each element (word). On each timestep, the input fed to the model contains both the current word in the sentence and the hidden state of the model (latent representation) from the previous timestep. This hidden state represents the "memory" of the model, which keeps the information on the activations of all previous timesteps (context), so of the words of the sentence previously analysed.

Figure 1 below shows the traditional, "vanilla" RNN cell (Elman RNN) in both the "folded" view,

[1] (Britz, 2015)
[2] (Kuhn & De Mori, 1990)
[3] (Kuhn & De Mori, 1990)
[4] (Kuhn & De Mori, 1990)
[5] (Jurafsky & Martin, 2020)
[6] (Kuhn & De Mori, 1990)

with the recursion represented by the circular arrow, and the "unfolded" view, which represents the recursion like a concatenation of cells sharing the hidden states sequentially across the various timesteps.

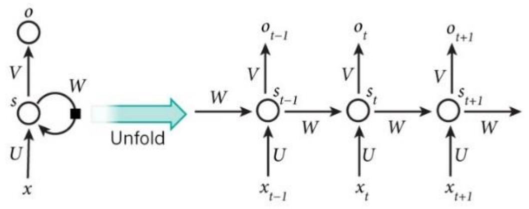

Figure 1. "Vanilla" RNN cell. [7]

Actually, vanilla RNN cells have some structural issues which prevent them from efficiently learning long-term temporal dependencies: due to the choice of the hyperbolic tangent (*tanh*) activation function, the gradients tend to easily vanish over time.[8]

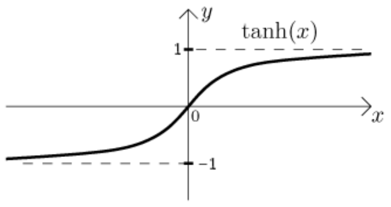

Figure 2. Graph of *tanh* activation function.[9]

Tanh can indeed squash large-scale inputs into an interval of [−1, 1] but saturation regions cover the biggest part of the domain (see Figure 2). Once the input falls into the saturation regions, gradients are close to zero and this slows down the updating of weights and biases. Even worse, the gradients decrease exponentially as the depth of the network increases, because gradient computation in backpropagation is based on the chain rule and all layers are interconnected and interlocked. This is even more complex in RNNs, where hidden states represent a "memory" that is shared across the various timesteps, overcomplicating computations over time.[10] Therefore, the effect of long-term dependencies is hidden by the effect of short-term ones, being the former exponentially smaller with respect to the length of the sequence.[11]

LSTMs and GRUs

Later improvements in RNNs have overcome these structural issues by designing more sophisticated activation functions, consisting of affine transformations followed by a simple element-wise non-linearity obtained through *gating units*. The two most famous approaches in this direction are *long short-term memory* units (LSTMs) and *gated recurrent units* (GRUs). While a vanilla RNN unit always replaces the activation, or the content, with a new value computed from the current input and the previous hidden state, LSTM and GRU units keep the existing content and add the new content on top of it. Gates decide to overwrite information or maintain relevant features over time, allowing the network to remember them for a long series of steps. The additive nature of the unit also allows to easily backpropagate the error without vanishing too quickly, by passing through multiple, bounded non-linearities.[12]

LSTMs and GRUs follow this similar approach with a different structure (Figure 3). Unlike the vanilla RNN unit, which simply computes a weighted sum of the input signal and the prior state and applies a non-linear function, each LSTM unit maintains a separate *memory cell* ***c*** whose output is modulated by an *output gate* ***o***. The output gate regulates the amount of memory content exposure. The memory cell is updated by partially forgetting the existing memory ***c*** and adding new memory content ***č***: a *forget gate* ***f*** modulates the degree to which the existing memory is forgotten, while an *input gate* ***i*** modulates the degree to which new memory content is added to the memory cell. All the three gates are sigmoid functions, outputting activations in the interval [0, 1].[13]

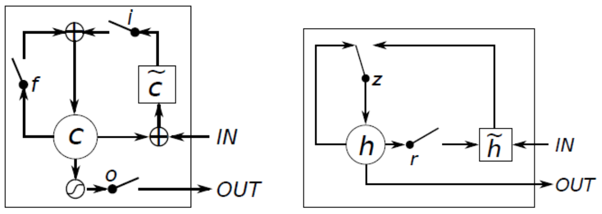

Figure 3. LSTM (left) and GRU (right) units.[14]

Similarly to LSTMs, GRUs have sigmoid gating units that modulate the flow of information inside the unit, but without having a separate memory cell. The activation ***h*** of the GRU at time *t* is a linear interpolation between the activation at the previous timestep and the new candidate activation ***ȟ***, modulated by an *update gate* ***z*** deciding how much the unit should update its content. The new candidate activation is computed similarly to the vanilla RNN unit's one, but with the inclusion of a *reset gate* ***r*** modulating the previous activation: when the reset gate is close to zero, the reset gate forgets the previous activation and makes the current activation consider only the current input. Finally, the GRU has no output gate modulating the degree to which its state is exposed.[15]

Both LSTM and GRU units have been proved to perform comparably to each other in many applications, at the same time outperforming vanilla RNN units.[16]

[7] (Britz, 2015)
[8] (Ricci, 2021)
[9] (CTU Czech Technical University in Prague, Accessed: 2021)
[10] (Wang, Qin, Wang, Xiang, & Chen, 2019)
[11] (Chung, Gulcehre, Cho, Bengio, & Yoshua, 2014)
[12] (Chung, Gulcehre, Cho, Bengio, & Yoshua, 2014)
[13] (Chung, Gulcehre, Cho, Bengio, & Yoshua, 2014)
[14] (Chung, Gulcehre, Cho, Bengio, & Yoshua, 2014)
[15] (Chung, Gulcehre, Cho, Bengio, & Yoshua, 2014)
[16] (Chung, Gulcehre, Cho, Bengio, & Yoshua, 2014)

2. LM2 implementation

The key guideline I decided to base my implementation of the LM2 project on is *simplicity*. I wanted to implement an efficient model while keeping things as simple, streamlined and intuitive as I could.

I chose to implement a GRU model because GRUs have been proved to be slightly more computationally efficient than LSTMs, especially during training,[17] have a less complex structure and are relatively newer. My library of choice was Keras, which has ready-to-use layers and modules which allow to experiment with neural networks, training and predictions "at the speed of thought".[18] I also chose to use libraries I have been widely using for other projects, like NumPy for managing arrays and tensors and Matplotlib to plot the training & validation progress.

The logic I have followed in implementing the model is straightforward:

i. vectorization of the dataset;
ii. reshaping of the vectorized dataset from a 1D vector to a ready-for-batching 2D array, in which every row represents a separate sentence to respect the context isolation requirement.[19]
iii. model building and training;
iv. inference: evaluation (using the test dataset) and text generation, where a seed of k words is fed into the model (in the form of a 2D array) and used to sample the next word(s).

The code has been implemented in Google Colaboratory ("Colab"). The directory structure provided in Github mirrors the one used in Google Drive, from which the Colab notebook imports the dataset. An "inference-mode toggle" is provided in the second cell to easily switch from training mode to inference mode (see the *"README.md"* file for instructions).

Part 1) Corpus processing and vectorization

Dataset

The dataset used for this project is the word-level dataset from Penn-Treebank.[20] The dataset is already clean: it does not contain capital letters, numbers, punctuations and has already been tokenized.

corpus_prepare()

The function transforms the dataset obtained as input into a list of string tokens. The conversion is done line by line: each line in the original file is a sentence and the function separates each sentence by appending a <*bos*> tag at the beginning and an <*eos*> tag at the end. The function also returns the length of the longest sentence found in the dataset. Furthermore, if the function is called passing the vocabulary among the args, it also replaces out-of-vocabulary (*oov*) words with the token <*unk*>, an operation that is needed when converting the validation and test datasets.

get_vocab()

The function is only launched for the training dataset and allows to create a dictionary with the unique string tokens as keys and their corresponding integer indexes (in order of appearance) as values. The function also returns the *index-2-words (i2w)* dictionary with keys and values inverted.[21] The size of the Penn-Treebank vocabulary, including the 4 special tokens (*pad, unk, bos, eos*), is 10,002 unique words.

vectorize()

This function converts any list of string words to a vector of integers. The mapping is taken from the *voc* dictionary passed as input. I wanted to have one function only to vectorize either dataset files or lists of strings, so I required as additional args a boolean *fromfile,* which allows to select the reading mode (*True* if read from file, *False* if read from list corpus), and, alternatively, the file directory of the input dataset or the input corpus.

deindexer()

Converts a vector of integers to a vector of strings.

Part 2) Preparation of batches

The preparation of input and output for the model followed some key choices. First of all, the model is recurrent, so the label for each word is the word immediately following.[22] It was important to keep each sentence separate, therefore the sentence must end after the *eos* tag. Keras accepts batches as input, with default size 32 sentences, so I thought to use bigger batches (64 sentences) while keeping the size as a multiple of 32.[23] I wanted to create batches of enough sentences (rows) in which all sentences had the same length (n of columns). The longest sentence is 83 tokens long, so I initially thought to build batches of shape 64x83, padding all sentences to 83 tokens. Actually, the solution was inefficient, slowing down the training and preventing the model from going below 700-800 of perplexity after many epochs (see Figure 5). Indeed, the average sentence length in the 3 datasets is only 20-21 tokens. So, I chose to build batches of size 64x42 instead (one half of 83), adopting the following "rule of 42":

- if a sentence is shorter than 42 tokens, pad it up to 42;
- if a sentence is longer than 42 tokens, split at 42 in two parts. Then repeat the last token of the first part as the first token of the second

[17] (Yang, Yu, & Zhou, 2020)
[18] (Keras, Accessed: 2021)
[19] (Piazza, Accessed: 2021)
[20] (DeepAI, Accessed: 2021)
[21] (Stepanov, 2021)
[22] (TensorFlow, Accessed: 2021)
[23] (TensorFlow, Accessed: 2021)

part, to preserve some continuity (a sort of bi-gram), add the next words and pad up to 42.

Due to the size of the training corpus and the vocabulary, I also chose not to use *one-hot encoding* of labels: Colab's RAM exploded while I was trying one-hot conversion and I initially experimented with building a generator to dynamically feed the model. The solution came out to be inefficient, so I decided to simply keep labels as integers and use *sparse_categorical_crossentropy* as loss function.[24]

prep_sentences()

This function is internally called by *prep_batches()*. At this stage, the idea is to keep sentences concatenated in a 1D array but spatially separate them at regular intervals through padding, according to the "rule of 42" (sentence-level padding). Therefore, a new sequence starts *exactly* every 42 tokens, either with a *'2'* (*<bos>* tag) or with the repetition of the preceding token if the sentence has been split in two. The function also requires as input a boolean *labels*, which tells it which vector it is preparing.

prep_batches()

This function converts the 1D regularized vector obtained through *prep_sentences()* (for both input and labels) into a 2D array. Since I want each batch to be 64x42, I need to obtain *n* batches of 64 sequences each. I need therefore to add padding at the end of the regularized vector (dataset-level padding) to make sure that all the batches will be exactly 64 sequences long and each sequence will still be 42 tokens long. The calculations are explained in more detail in the code file: for the training file, for instance, the regularized vector is 1,837,164 tokens, which is not divisible by 2,688 (64*42), so I pad it up to 1,838,592 tokens (closest multiple of 2,688) and then I finally split each sequence on a separate row by using *np.reshape()*, obtaining a vector of shape 43,776x42 that will be split, during training, into 684 batches.

Parts 3) Model building and training

The GRU implemented is a sequential Keras model with an embedding layer (500 units), 3 bi-directional GRU hidden layers (500*2 = 1000 units) and a TimeDistributed Dense layer, with a *softmax* activation function (being the task a multiclass classification problem). To the embedding and the 3 bidirectional GRU layers, dropout is applied with a 30% rate to perform a better training and improve generalization capabilities. The model is trained for 50 epochs, with an early stop callback set to be triggered after 3 epochs of no improvement in validation loss to prevent overfit. The chosen optimizer is stochastic gradient descent (SGD), with a 0.1 learning rate and a 0.7 Nesterov momentum needed to speed up the convergence.

A graph of the model is available in Figure 4. These choices represent the best setting I have tested on this task and result comments are available in par. 3 below. Training and validation progress charts are provided in the code file.[25] Perplexity is calculated as the exponential of cross-entropy loss, averaged over all tokens: the implementation is an adaptation of a code example from StackOverflow[26] fixed in order to compute the average loss/perplexity within each batch and, then, per epoch. Weights are saved and exported after training, to be easily loaded at inference time.

Part 4) Evaluation & text generation

The model is then evaluated and tested for text generation, using a seed of 3 words. Generation is performed by the *generate_one_step()* function, a simplified re-elaboration of the homonymous function available in TensorFlow documentation.[27] The companion *generate_text()* function vectorizes the seed and passes it to *generate_one_step()*, which extends it to a 64 x *seed_size* array through zero-padding in the 2 directions. The logits obtained from the *model()* call are used to sample the next word, which is finally returned to *generate_text()* and appended to the seed. The updated seed will be passed back to *generate_one_step()* until the desired length of the generated sentence is reached.

3. Results

The chosen setting represents the best combination I have tested on this task. I have tried indeed several alternatives in batch sizes, number and type of layers / units, different values of hyperparameters, different optimizers (i.e. Adam) and rates for learning rates / momentum / dropout. I performed further trials too, like a training run without isolating the sentences on separate rows (i.e. no sentence-level padding, so simply reshaping the 1D vectorized dataset to a 2D 64x25 vector), or with sentence isolation but using 25 as sequence length. I then preferred to keep 42 to better preserve context information. I have completed the training for only a few of these alternatives, while for the most inefficient cases (i.e. for Adam and different SGD/layer settings) I simply stopped it after a few epochs. Results for the trainings I have completed are available in Figure 5 below and confirm the goodness of the final settings.

The chosen combination allows the model to reach, after 50 epochs, a perplexity of 85.0 on the training set, 92.9 on the validation set and 87.1 on the test set. Loss is 0.398 on the training set, 0.397 on the validation set and 0.350 on the test set. Results are well aligned with the state of the art (see Figure 6).

[24] (Stepanov, 2021)
[25] (TensorFlow, Accessed: 2021)
[26] (StackOverflow, Accessed: 2021)
[27] (TensorFlow, Accessed: 2021)

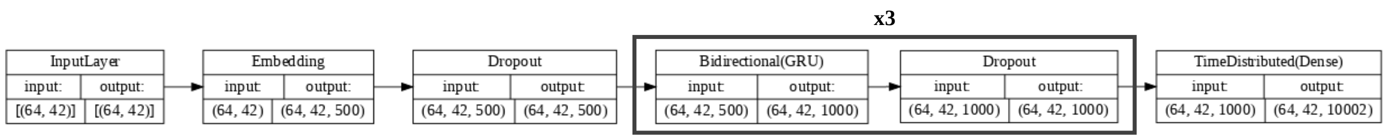

Figure 4. Graph of the Keras GRU language model implemented.

Batch size	Bi-dir.	Sentences isolated	Epochs	Tr. PPL	Val. PPL
64x25	No	Yes	100	269.77	356.84
64x25	No	No	100	87.29	148.80
64x25	Yes	Yes	5	170.26	144.12
64x83	Yes	Yes	40	824.70	828.75
64x42	**Yes**	**Yes**	**50**	**85.02**	**92.89**

Figure 5. Results of some trials I performed using different size of batches, unidir. or bidir. layers, isolation of sentences. All settings not mentioned here are the same of the final one.

Model	PPL	Year
KN5	141	1994
KN5+Cache	126	1994
RNN	125	2012
+KN5+Cache	97	2012
Bi-GRU 64x42 (current paper)	**87.1 (test)**	**2021**
LSTM (Zaremba – 24M)	78	2014
LSTM (Melis – 24M)	58	2018

Figure 6. Comparison of results obtained with state-of-the-art models (all trained on Penn Treebank).[28]

References

Britz, D. (2015, September 30). *Recurrent Neural Networks Tutorial, Part 2 – Implementing a RNN with Python, Numpy and Theano.* Retrieved from WildML: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/

Chung, J., Gulcehre, C., Cho, K., Bengio, & Yoshua. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *NIPS 2014 - Deep Learning and Representation Learning Workshop*. Retrieved from https://arxiv.org/abs/1412.3555

CTU Czech Technical University in Prague. (Accessed: 2021, August). *Hyperbolic functions.* Retrieved from Math Tutor: https://math.fel.cvut.cz/mt/txtb/4/txe3ba4f.htm

DeepAI. (Accessed: 2021, June). *Penn Treebank Dataset.* Retrieved from DeepAI: https://deepai.org/dataset/penn-treebank

Jurafsky, D., & Martin, J. H. (2020). N-gram Language Models. In *Speech and Language Processing, Third Edition Draft.* Upper Saddle River, New Jersey: Pearson - Prentice Hall.

Keras. (Accessed: 2021, August). *Homepage*. Retrieved from Keras.io: https://keras.io/

Kuhn, R., & De Mori, R. (1990). A Cache-Based Natural Language Model for Speech Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, No. 6*.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010). Recurrent neural network based language model. *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*.

Mikolov, T., Kombrink, S., Burget, L., Cernocký, J., & Khudanpur, S. (2011). Extensions of recurrent neural network language model. *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*.

Piazza. (Accessed: 2021, July). *[LM] How to compute perplexity - By Roccabruna, Gabriel.* Retrieved from NLU Course forum on Piazza.com: https://piazza.com/class/km0nbdptjf02o8?cid=42

Riccardi, G. (2021). Language Modeling. In *NLU Course - Teaching Materials.* Trento: University of Trento.

Ricci, E. (2021). Recurrent neural networks. In *Deep Learning - Teaching Materials.* Trento: University of Trento.

StackOverflow. (Accessed: 2021, July). *How to Implement Perplexity in Keras?* Retrieved from StackOverflow: https://stackoverflow.com/questions/44697318/how-to-implement-perplexity-in-keras

Stepanov, E. (2021). *Sequence Labeling with Deep Neural Networks.* Retrieved from NLU Teaching Materials - GitHub: https://github.com/esrel/NLU.Lab.2021/blob/master/notebooks/sequence_labeling_nn.ipynb

TensorFlow. (Accessed: 2021, July). *"fit", from "tf.keras.Model".* Retrieved from TensorFlow Documentation: https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

TensorFlow. (Accessed: 2021, July). *Text generation with an RNN.* Retrieved from TensorFlow Documentation: https://www.tensorflow.org/text/tutorials/text_generation

TensorFlow. (Accessed: 2021, August). *tf.keras.callbacks.History.* Retrieved from Tensorflow Documentation: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/History

Wang, X., Qin, Y., Wang, Y., Xiang, S., & Chen, H. (2019). ReLTanh: An activation function with vanishing gradient resistance for SAE-based DNNs and its application to rotating machinery fault diagnosis. *Elsevier Neurocomputing*.

Yang, S., Yu, X., & Zhou, Y. (2020). LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example. *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*. Retrieved from https://ieeexplore.ieee.org/document/9221727

[28] (Riccardi, 2021)