Project - Language Modeling 2

Matteo Zanella

I. Introduction

THE ML2 project of the Natural Language Understanding course has the objective of implementing an RNN language model and training it on the Penn Treebank dataset. Recurrent Neural Networks permit to predict the next element of an input sequence by propagating the hidden representations of the sequence elements though the sequence.

A. Language and libraries

The code is written in Python 3 on a Jupyter notebook hosted on Google Colab, to use the GPU capabilities of the platform. The deep learning library employed is Keras.

II. Dataset

The Penn Treebank dataset contains utterances already divided into the training, validation, and test splits having sizes of 42068, 3370, 3761 respectively. The vocabulary of the corpus is limited to 10000 words, with the OOV tokens already replaced by the `<unk>` token. Moreover, the numbers are replaced by the `N` token.

A. Tagging and vectorization

To be used by a neural network, tokens need to be converted into a numerical form. For this task, I have employed the `TextVectorization` functionality of TensorFlow, which simply assign to each word in the vocabulary an integer value, splits the string utterance into string tokens, and uses the dictionary lookup to convert the string tokens into the corresponding integer. The text standardization was not used, since the dataset is already lowercase and stripped of punctuation.

Since the vectorizer reserves `[UNK]` for the OOV, the corpus has been modified before the vectorization to reflect this convention: `<unk>` is replaced by `[UNK]` and `N` is replaced by `[N]`.

Additionally, the special token `[S]` is added at the beginning of each sentence. Together with `[/S]`, also these tokens are learned by the vectorizer.

B. Batching and padding

Every sentence, represented by its 1D vectorization, is a different sample. A neural network can process at the same time several samples grouped into a minibatch. The samples in the same minibatch need to have the same length, but every sentence has a different number of tokens. TensorFlow permits to easily create such padded batches by calling `.padded_batch(BATCH_SIZE)`. The padding is made of zero values appended to the right of the short sentences such that their length corresponds to the one of the longest sentence in the minibatch. The vectorizer already reserves the zero integer for the padding token. Thanks to the masking functionality of Keras, layers can ignore the padding in the computations of representations and metrics.

After various tests, the chosen minibatch size is 16. The low value helps to explore more erratically the solutions space and has a regularization effect.

C. Targets

The developed RNNs predict a sequence for every sample: the prediction corresponds to the sentence shifted to the left. For example, given the training sample ['[S]', 'The', 'red', 'cat'] the network should predict ['The', 'red', 'cat', '[/S]']. This way, the network learns to predict the next token from all the available previous tokens.

III. Metrics

A. Objective function

The network should minimize the uncertainty on predicting the vocabulary word that follows the previous sequence. This is a multi-classification problem because a sentence of length N needs N classifications. The best suited loss for the problem is the sparse cross-entropy.

After computing the cross-entropy loss for each non-padding token of each sentence in the batch, the final objective value can be obtained by averaging across same-sentence tokens and then across all the sentences.

However, the Keras implementation for the sparse categorical cross-entropy function is not suited for time sequences having a shape (Batch size, Sequence length, Vocabulary). The problem is that it sets to zero the cross-entropy values corresponding to the padding tokens, but then it reduces the values by including in the mean also the zeroed values, resulting in a loss value lower than the real one.

The developed implementation simply divides the cross-entropy values by the number of non-padding tokens in the sentence and multiplies by the sequence length, such that the reduction performed by Keras results in the correct loss value.

B. Perplexity metric

The perplexity of the model on a dataset can be computed as the exponential of the cross-entropy averaged across all sentences. A stateful class is necessary to get the exponential average cross-entropy and not the average of the exponential averages on all minibatches.

The cross-entropy is computed in Keras with the natural logarithm, so the formula to compute the perplexity applies the natural exponential.

IV. Models

All models presented in the project share an initial embedding layer that learns for each possible integer of the vectorized vocabulary a multi-dimensional representation of arbitrary length. If the indices are fed directly to the network, the learning is faster, but also more prone to overfitting.

The RNN cells used are the GRU version, which are an improved version of the LSTM version. No recurrent dropout has been used, because it can harm the learning [1]. The ReLU function has been tested as replacement of the tanh activation function for the GRU cells. Despite the faster convergence, the execution becomes slower because the efficient cuDNN implementation can't be used. For this reason, the presented model use the standard activation function. Moreover, the GRU layers should return the hidden state of every time step and not just the last one: this is fundamental to compute the loss of the predictions in the middle of the sequence.

Between the layers, there are dropout operations which help with the regularization of the model.

The last layer of the model is a linear layer which is applied separately to the hidden state of every time step. This layer projects the hidden representation into an output vector as long as the dictionary and applies a softmax function to transform it into the probability vector that express the most probable word in the dictionary that will follow the previous sequence.

Finally, these models are converted into full-fledged a language model by integrating the text vectorization into the model, extracting only the prediction of the last time step and converting it back to the string domain using an inverse vocabulary lookup.

A. Simple model

The first experiments were made on networks with limited size. Among them, the best performing has an embedding size of 32 and a single GRU layer with 64 units.

B. Dense model

Since stacking more recurrent layers and tuning their size was not producing satisfying results, I tried an architecture with skip-connections. Inspired by the DenseNet architecture, the input of each layer is concatenated with its output and used as input for the next layer. The tested model has an embedding size of 512 and two GRU layers with 512 units each.

C. Complex model

In this model, a great increase in the hidden layers size has been tried. The embeddings have a size of 650 and there are two stacked GRU layers with the same number of hidden units, without the dense structure.

D. Complex+ReverseEmbedding model

The next step has been adding to the last linear layer in the Complex model a custom ReverseEmbedding regularizer. The quantity added to the loss is the norm of the difference between the dense layer's weights matrix and the transposed embedding's weights matrix, based on the idea that the two layers perform similar operations but in the opposite direction. The pseudo-inverse should be used to correctly reverse the transformation matrix, but it is costly to compute. For this reason, this approximate solution was chosen to experiment with its regularization effect.

TABLE I
Comparison between the models

Model	Parameters	Perplexity on Test
Simple	1M	156
Dense	24M	114
Complex	18M	119
Complex+RE	18M	116
Huge+RE	27M	111

E. Huge+ReverseEmbedding model

This model has even greater hidden layer sizes. The embeddings have a size of 880 and there are two stacked GRU layers with the same number of hidden units.

The version without the regularization has also been tested, but it is not presented due to its lower performances.

V. Training

The initial optimizer used for the training was Adam, but the performances were poor: the loss was diverging with high learning rates and converging too slowly with lower values. After changing the optimizer to SGD, the convergence capabilities improved significantly. The hyperparameters are a learning rate of 1 and a clipping to 10 of the gradient norm.

The learning rate scheduler employed reduces the lr with a factor of 0.8 when the validation perplexity do not improve of at least 1 after an epoch.

After 10 epochs where the perplexity validation has not improved, the training is early stopped and the weights of the best epoch are restored.

VI. Conlusion

The results on the test set show the correlation between the number of parameters and the perplexity achieved. The simple model has clearly too few parameters.

Surprisingly, using the ReverseEmbedding regularization helped to improve the performances of the networks. During the training, the overfitting was drastically reduced and the improvement on the perplexity was smoother and more monotonic. The training is not successful if L1 or L2 are used instead, but different values of regularization factor hyperparameter have not been extensively tested.

Despite all the small improvements in the networks, a state-of-the-art perplexity have not been reached by the developed models. A possible reason could be that an explicit ensemble of different models with different architectures could significantly outperform the implicit ensemble created by the dropout layers. Additionally, the number of parameters and the time employed for training were limited. Increasing such factors could easily improve the performances of the language model.

REFERENCES

[1] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. "Recurrent neural network regularization". In: *arXiv preprint arXiv:1409.2329* (2014).