Recurrent Networks

Innopolis University

Advanced Machine Learning — M18-DS

Josep de Cid Rodríguez

March 1, 2019

Abstract

RNNs are a class of NNs that behave very well with problems that require to process sequences of inputs, with much less parameters than a classic NN. In this report there are going to be discussed the different results obtained with different architectures in termms of accuracy, complexity and training time.

1 Introduction

The main goal of this laboratory session is to experiment with Recurrent Neural Networks and compare it's performance with Feed-forward Neural Networks. In order to achieve that, we are going to implement a name gender classifier, which will predict if a name is *Male* or *Female*.

Our data contains around 100000 pairs name-gender, being the train-test ratio 80/20. We will apply different transformations to our data, as it's in plain format, which isn't very useful for learning algorithms.

We are going to implement different models in order to experiment with those and choose the most appropriate one to solve our problem. We will try different architectures for our RNNs and FNNs, and evaluate its pros, cons and performance, to finally choose the best model via Hyperparameter Tuning.

2 Data Preprocessing

The first trouble we face in this problem is how to represent and input data into our model. As for many different of machine learning algorithms that have words as input data, we can't feed strings directly to our network, so we need to convert it to some numerical format.

To start with, we need to convert our names to a list of **unique IDs** for every character, such that $A \rightarrow 1$, $B \rightarrow 2$ and so on.

Learning algorithms don't allow easily handling of variable length inputs. Because of that, at the next step we must look for the longest name in the training set and **add padding** to shorter names, using the special ID 0 (which haven't been used in the character representation). In case that, during prediction time, some name is longer than the maximum sequence obtained from the train data, we will just trim the sequence the specified size.

Finally, we need to apply some character embedding to our data. We are going to train a **Word2Vec**[1] model with different embedding dimensions to compare the performance with different embedding dimensions in exchange of its training time. The whole process is schematised in Figure 1.

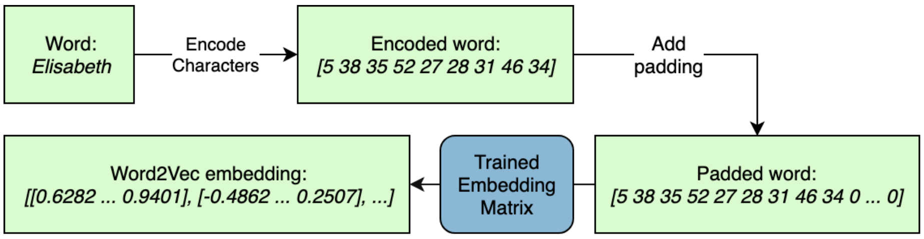
Word: *Elisabeth*
Encode Characters
Encoded word: *[5 38 35 52 27 28 31 46 34]*
Add padding
Padded word: *[5 38 35 52 27 28 31 46 34 0 ... 0]*
Trained Embedding Matrix
Word2Vec embedding: *[[0.6282 ... 0.9401], [-0.4862 ... 0.2507], ...]*

Figure 1: Dataset preprocessing pipeline

3 Models

We will compare two totally different architectures and then tune some parameters, such as the cell type[1], flattening modes[2], batch size, learning rate, number of layers and neurons and activation functions to obtain the best model according to mainly a trade-off between test accuracy and training time.

3.1 Recurrent Neural Networks (RNN)

Our first and baseline model is going to be a `RNN` using a `LSTM` cell.

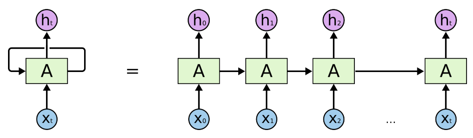

Figure 2: RNN architecture. **Source:** Colah's Blog[2]

Why RNN? This kind of networks work specially well with sequential data, both in terms of accuracy performance and training speed.

The basic neural networks, commonly referred to as Multilayer Perceptrons (**MLP**), do not share features learnt through different positions, so it's much more difficult to generalize the learnt features, to get an significantly equivalent performance, RNN requires much less parameters to learn, which is faster and allows to keep simpler models.

[1]Only when implementing the RNNs, chosing between LSTM and GRU.
[2]Used to adapt sequenced data to basic NN.

3.2 Multilayer Perceptrons (MLP)

As we've already discussed in the previous section, the traditional Neural Networks, will need a higher amount of parameters to get a same-level accuracy that the one that can be given by a RNN.

Given that these kind of networks are not ready to be fed with multidimensional 3D data `(Batch size, Time-steps, Embedding Size)` being the second dimension the time-steps or sequence length of every input `x`, we need to flatten this input and we have several ways to do it:

- Flatten: Simply joins time and embedding dimensions in such a way that a dimension `(a, b, c)` becomes `(a, b*c)`.
- Maxpool: Maximum value for time dimension or embedding dimension.
- Average: Mean value for the time dimension or embedding dimension.
- Weighted Average: Weighted mean value for the time dimension or embedding dimension. These weights are going to be learnt during training process, so this mode increase the parameters to learn.

3.3 Tunning model Hyperparameters

Which network configurations can we play with? There are several hyperparameters we can try to tune in order to improve our model performance. Most of them are common for any DL model and don't need to be commented as are trivial, as the *Batch Size*, *Neurons/Layer*, *Number of layers* and *Activation functions*, allowing to use `Tanh`, `Sigmoid` and `ReLU`.

The specific tunable hyperparameter of the RNN networks is the cell type. We implemented our baseline model with a LSTM, but there are also other well-known cells that we can experiment with, more specifically, the GRU. This one is not only easier and faster to train (less parameters and almost same behaviour than a LSTM), but also tends to perform better language modeling tasks (similar topic than our work here).

By the other hand, LSTMs are supposed to remember longer sequences than GRUs and outperform them in tasks requiring modeling long-distance relations, which is not the case here. In the Figure 3 we can observe that the GRU is simpler as it has 75% of the parameters than the LSTM have and only needs to keep one internal state h, while the LSTM needs h and c.

The LSTM implementation provided in the TensorFlow framework allows the usage of Peepholes,[3] which provide connections between internal gates can learn a finer distinction between sequences of spikes widely spaced. Its usage will be given by a Boolean hyperparameter when the cell-type is a LSTM. We can observe an example of these connections in Figure 4

For the MLP, we will also tune which flatten method to use with the input data along with which axis to apply the flatten operation too, for maxpool and averaging methods.

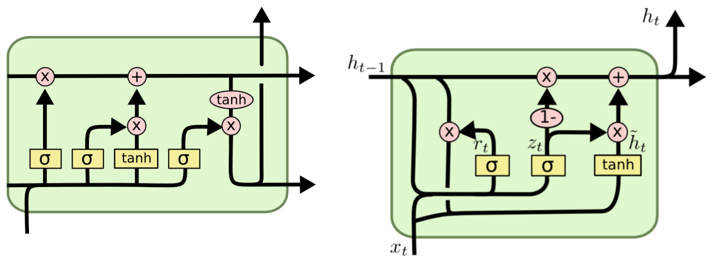

Figure 3: LSTM vs. GRU architectures. **Source:** Colah's blog[2]

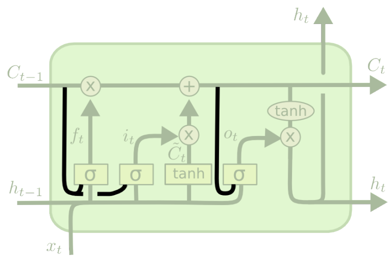

Figure 4: LSTM with highlighted PeepHoles. **Source:** Colah's blog[2]

4 Experiments and Analysis

To tune the hyperparameters we have implemented a Random Search[4] procedure using K-Fold Cross Validation with `k=5`. Random search has a probability of 95% to find the set of hyperparameters within the 5% optima with only 60 iterations. Also compared to other methods like the classical Grid Search it doesn't bog down in local optima.[5]

After some experiments we were able to increase a lot the accuracy of the baseline MLP, increasing from the threshold 74.993% to 80.802%. As for the RNN model, we weren't able to do a significant performance increase, going from 80.20% to 81.86%.

For the RNN model, it's also been tried more sophisticated methods, such as bidirectional RNN, leaky and exponential ReLUs without much success, so we will focus in the improvements done in the MLP, which model performance results are shown in Figure 5. The top 5 architectures (sorted by test data accuracy) are shown in these plots, differentiating between train and test data accuracy.

The legend contains the encoded configuration, showing the learning rate and batch size. The flatten method is show as a name and the axis that has been applied to. The layers are encoded with `units` + `activation function`, which is `T` for *tanh*, `S` for *sigmoid* and `R` for *ReLU*.

E.g. "*(MLP - max_pool(2)) LR=0.00415; BatchSize=256; Layers=136T, 128S, 143T*" is a MLP flattened with Max Pooling applied to embedding axis with learning rate 0.00415, batch size of 256, and 3 layers having the first 136 neurons and *tanh*, 128 neurons and *sigmoid* for the second and 143 neurons with *tanh* for the last one.

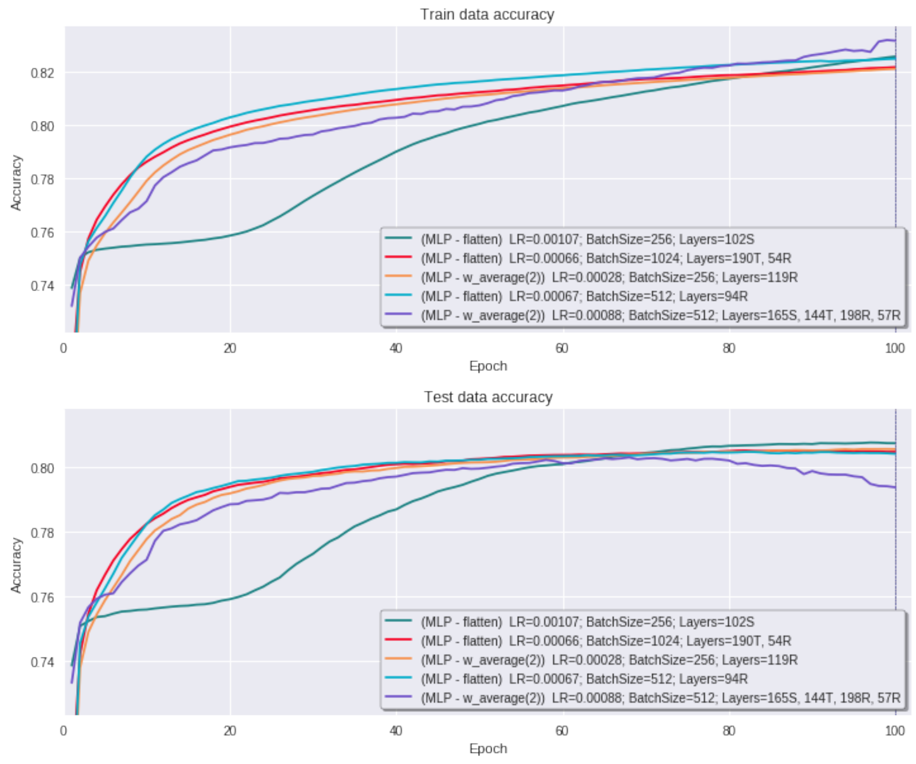

Figure 5: Top5 model architectures for MLP networks

From the MLP results we can do some different analysis and extract results. It's curious to see that the flatten function for the inputs is different for each of the top models. The only one which gives a poor performance is the MaxPool, that by itself is meaningless as it's taking the higher value of the embedded word. As we could expect, there is also no simple mean flattening in our top, as the weighted version can be trainable and obviously leads to better results. For the axis to flatten, to flatten the embedding works much better than flattening the sequence axis, as we capture the weights for every step input, and not a mix of all the name. Despite of these, we can conclude that the **standard flattening** leads to better results.

For the learning rate and batch size values, we see that the first one oscillates around the common 0.001 value, and the mini batch sizes are very assorted, going from 256 to 1024.

The most interesting parameter to comment here are the layers distribution and architecture and how those behave. It is curious to see that the best model, the green is almost the **simplest** one, with only one layer with 102 units, and a sigmoid function. This model takes more time to start learning, as its curve is very different from the others and slower to grow, but at the end it gets the higher accuracy in the test data, having almost about 80.5%.

To compare with, check the opposite side model, the purple one, which is so complex, having 4 layers, with very different number of units and activation functions in each layer. Having more complex model means more trainable parameters, which means more ability to learn from the training data. We can note this in the train data accuracy plot, where this model outperforms the others increasing the accuracy from 82% to 83%. But which is the trade-off to learn more from the training data? **Overfitting**, as we can observe in the test accuracy, which descends drastically after some epochs, being much worse.

The schematized architecture of the green model is shown in the Figure 6.

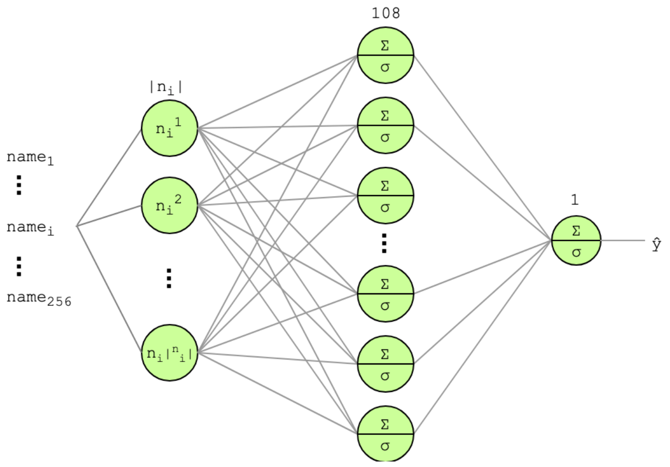

Figure 6: Best architecture for MLP model

The best RNN model we could come up with was a 2-stacked layers RNN with GRU and LSTM (with peepholes), using 20 and 10 units respectively, with *exponential ReLU* activation function.

To sum up, comparing both architectures, we see that RNN tends to perform better with data that is someway sequential. The heavy increase of performance for the MLP, is given by a previously explained fact, which is that non-recurrent neural networks need much more parameters to learn the same way than a recurrent would do. If we compare both top models, the MLP one, had 8115 trainable parameters, while the RNN only had 3661, even having multiple stacked layers and more complicated architecture. Anyway, it's faster to train a simple neural network mode, as the LSTM and GRU cell calculations are much more complex and expensive to compute than a simple matrix multiplication.

To end with the experiments, let's check how our models behave with Russian names instead of English ones. Surprisingly, the results using the best models for English names are very bad, producing a heavy overfitting, with almost 90% of accuracy in training data but only 61.27% with test data, using the RNN. With MLP, it's even worse, only 54.68% of accuracy. Adding a normalization method like Dropout, doesn't seems to provide a significant increase of the performance.

5 Conclusions

To end with this report, we will summarize some observations found during the analysis and experiments:

– **RNN performs better** than a simple MLP, but as it seems that for this problem it's difficult to get a higher accuracy than 82%, we can supply this lack of performance training much more parameters in the MLP. Despite of having a better performance, it's much **slower to train a RNN** due to the complex calculations that need to be done in the cells, rather than a simple matrix multiplication in a normal dense layer.

– **Complex models**, which imply higher amount tend to **overfit**, as they get greater accuracy in the training data but a heavy decrease when testing. These models also take a **higher time** to train.

– **Activation functions** not only affect the final result, but they have a heavy impact in the **training time**, like the *Sigmoid* which, having to calculate the exponential, is so slower compared to the *ReLU*, which only computes a maximum function.

– Russian names gender seems to be much more difficult to predict than the English ones, probably because of the language differences or data distribution. Anyway, we can conclude that our **built models are not language-independent**, so every language has its own constants.

References

[1] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[2] Christopher Olah. Understanding lstm networks - colah's blog, 2016.

[3] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *J. Mach. Learn. Res.*, 3:115–143, March 2003.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, February 2012.

[5] Timo Böhm. How to optimize hyperparameters of machine learning models, 2018.