Hyperparameter Optimization of LSTM Network Models through Genetic Algorithm

Nikolaos Gorgolis  
Computer Science and Informatics Department  
University of Patras  
Patras, Greece  
gkorgkolis@ceid.upatras.gr

Ioannis Hatzilygeroudis  
Computer Science and Informatics Department  
University of Patras  
Patras, Greece  
ihatz@ceid.upatras.gr

Zoltan Istenes  
Department of Software Technology and Methodology  
Eötvös Loránd University  
Budapest, Hungary  
istenes@inf.elte.hu

Lazlo – Grad Gye  
EIT Digital  
Eötvös Loránd University  
Budapest, Hungary  
laszlo.grad-gyenge@creo.hu

Abstract—Next word prediction is an important problem in the domain of NLP, hence in modern artificial intelligence. It draws both scientific and industrial interest, as it consists the core of many processes, like autocorrection, text generation, review prediction etc. Currently, the most efficient and common approach used is classification, using artificial neural networks (ANNs). One of the main drawbacks of ANNs is fine – tuning their hyperparameters, a procedure which is essential to the performance of the model. On the other hand, the approaches usually used for fine – tuning are either computationally unaffordable (e.g. grid search) or of uncertain efficiency (e.g. trial & error). As a response to the above, through the current paper is presented a simple genetic algorithm approach, which is used for the hyperparameter tuning of a common language model and it achieves tuning efficiency without following an exhaustive search.

Keywords — hyperparameter optimization, fine – tuning, LSTM, genetic algorithms

I. INTRODUCTION

From scientific domain to industry, there are numerous tasks in which sequence prediction plays an essential role. Representative examples are text autocorrection [1], text generation [2], review prediction [3], speech recognition [4] etc. One of the major domains incorporating sequence prediction is NLP, especially in the cases of character – level [5] and word – level prediction. Currently, the state-of-the-art results, in the majority of cases, are achieved through the use of neural networks [6].

Despite being undeniably a great tool, the neural networks are usually characterized by a large set of hyperparameters that define the network’s topology, computational power and so on. As a result, the hyperparameters need to be properly configured in order to harness the network’s functionality. So, optimization of the hyperparameters’ configuration appears to be quite a challenging task, as it differs, depending on the task being executed, the dataset being used etc., rendering each situation unique.

Considering the extensiveness of use of predictive neural networks, a concrete and efficient method of hyperparameter fine – tuning would be quite beneficial. Various approaches are being used, from simple trial & error to Bayesian Optimization [7], with every approach having its advantages and disadvantages. In order to contribute in this effort, a simple but robust evolutionary algorithm is designed. This algorithm is applied to a randomly generated population of language LSTM (Long Short-Term Memory) models, aiming at finding the optimal configuration of a set of parameters, in a predefined search space. It is shown that the results of the algorithm outperform the established standard configuration, while leaving a promising path for improvements.

The structure of this paper is as follows. In II, an overview of similar works is being presented. In III, both the neural models used, and the designed genetic algorithm are described, followed by the experimental results in IV. The paper ends with the conclusion of the experiments in V.

II. RELATED WORK

One of the first efforts in optimizing artificial neural networks through genetic algorithms is spotted in [8]. Instead of optimizing the hyperparameters though, the authors try to optimize the weights of the network in order to overcome the drawbacks of backpropagation, returning promising results for the time.

In [9], the authors refer to the optimization of the topology of a neural network through the use of a genetic algorithm. They use binary encoding of the connections between the neurons, where 1 signifies the existence of a connection between two neurons and 0 the absence of it. They represent their chromosomes through the concatenation of the connections. Still, they do not investigate other parameters nor expand on the experiments conducted.

An analytical and expanded parameter search can be seen in [10]. A vast domain of hyperparameters was used, which contained pre-trained word embeddings, character representation, optimizers, gradient clipping and normalization, in addition to others. Approximately 50,000 experiments were...
run on a bidirectional LSTM network implemented on Keras [11], for five classical NLP tasks. However, they do not utilize global optimization methods, like the genetic algorithms, but instead evaluate various randomly generated models. Noteworthy, the purpose of the publication is not the proposal of a single optimal configuration, but instead the analysis of the behavior of the different architectures in general.

III. NEURAL MODEL AND GENETIC ALGORITHM DESCRIPTION

A. Neural Language Models

![Neural Network architecture used](image)

The genetic algorithm is applied for fine tuning of neural language models. The models are multilayered LSTM – based and are loosely based on the models used by Tensorflow [12] tutorial webpage, which on their turn are following the model described by Zaremba et al at [13]. They consist of an input layer followed by an embedding layer, two LSTM hidden layers and a dropout layer, finally followed by a dense layer and a Softmax activation layer (see Fig. 1). Analytical and mathematical description of the model is intentionally skipped, as it can be found in the original sources through the references stated.

The models are implemented both on pure Tensorflow code and according to the Keras [11] library. For each of them, a different implementation of the genetic algorithm is used, with slight differences concerning the mutation procedure, the addition of random individuals and the value of specific parameters.

B. Genetic Algorithm

The representation used for the neural language models in the genetic algorithm, is through a vector of integers, as all the parameters selected to configure where integer – valued. Each integer corresponds to the value of a hyperparameter of the neural language model that is being tuned.

The first step of the algorithm is the creation of the initial population of models. The values of the hyperparameters for each model are randomly chosen from the defined search spaces using the python module `random`, which follows the uniform distribution.

The second step is the fitness evaluation of each model. During the experiments that took place, three different fitness functions were used: the validation accuracy, the validation perplexity and the average of the ten highest training accuracies of each model. The fittest models are considered those with the highest accuracy or the lowest perplexity respectively. The selection method used is the elitism. A proportion of the initial population (approximately half of it in the experiments performed) with the best fitness scores are explicitly transferred to the next generation as candidate parents. In the first implementation, random individuals are also possible to be transferred, in order to expand the search space of each generation.

Next follow the steps of crossover and mutation. According to the first variation of the algorithm, mutation is applied only to the survivors. According to the second variation, mutation is applied to whole new generation, after the crossover ends. Crossover is performed by randomly selecting the hyperparameters between each parent, in an equiprobable manner. The parents are also selected equiprobably throughout the survivors.

After the new generation is shaped, the process continues again from the second step, in an iterative manner, until the terminal condition is satisfied. In our case, the terminal condition is the fulfillment of a certain number of generations. The configuration of the model with the highest fitness in the last generation, consists the output of the algorithm.

<table>
<thead>
<tr>
<th>Genetic Algorithm for Neural Hyperparameter Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: population ← [ list of n models on separate graphs ]</td>
</tr>
<tr>
<td>2: generation ← 0</td>
</tr>
<tr>
<td>3: while (generation &lt; t) do:</td>
</tr>
<tr>
<td>4:     train_and_evaluate(population)</td>
</tr>
<tr>
<td>5:     new_gen ← return the m fittest individuals</td>
</tr>
<tr>
<td>6:     new_gen ← append random individuals to promote diversity</td>
</tr>
<tr>
<td>7:     mutate(new_gen)</td>
</tr>
<tr>
<td>8:     new_gen ← append offsprings through crossover until k</td>
</tr>
<tr>
<td>9:     population ← new_gen</td>
</tr>
<tr>
<td>10:    generation ← generation + 1</td>
</tr>
<tr>
<td>11:    outputs the hyperparameters of the fittest in population</td>
</tr>
</tbody>
</table>

![Figure 2: Pseudocode of the 1st variation](image)

IV. EXPERIMENTAL RESULTS

In the conducted experiments, the resulted configurations are constantly compared to the configuration (600, 2, 20, 35, 60), which corresponds to 600 hidden layer size, 2 hidden layers, 20 training batch size, 35 number of timesteps and 60 training epochs, as hyperparameters of the neural model. We consider this particular configuration as the default benchmark configuration for experimentation, as it is systematically proposed by popular platforms like [1], [3] and accepted by the community as a standard configuration of medium size for recurrent neural networks.

As stated above, there were two variations of the main algorithm used during the experiments. The first one is performed on models built on pure Tensorflow code. It uses the technique of random individuals’ selection and performs the mutation process on the surviving population, before the crossover process takes place. The fitness functions used were the highest validation accuracy achieved by the model, as well
as the average of the ten highest accuracies achieved during model training. In addition, at the second experiment of the first variant, the standard configuration, i.e. (600, 2, 35, 20, 60) was included in the initial population. The initial population has a size of 21, each evolutionary generation had a size of 12 and the number of maximum generations was 15. The mutation probability used was 0.15.

The second variation is based on models built on the Keras framework. It does not select random individuals and it performs the mutation process on the whole new generation. The fitness functions that were applied were the highest validation accuracy, as well as the lowest validation perplexity. The initial population had a size of 20, each evolutionary generation had a size of 10 and the number of maximum generations was 12. The values and sizes used where smaller than the first variation, as it was noted that the algorithm converged early enough to allow smaller values. The mutation probability used was 0.10.

Four major experiments were conducted, where the two variants of the algorithm utilized the fitness functions stated above. The resulting configurations are stated in Table 1. The result considering the validation perplexity as fitness was ignored, as it performs poorly due to its small number of training epochs. This signifies either the poor efficiency of the perplexity as a fitness function, or the setback at a local minimum. It should be noted that, similar tests were conducted with the validation perplexity as fitness and they tended to be inefficient, compared to the results of the validation accuracy as fitness. The rest of the resulting configurations were used to build respective language models, and therefore trained and tested alongside the default configuration used by the majority of similar applications (i.e. the Tensorflow framework [14]). The default configuration is usually defined as:

- Hidden layer size: 600
- Number of hidden layers: 2
- Batch size: 20
- Number of timesteps: 35
- Number of epochs: 60

The comparison graphs of the validation accuracy and validation perplexity achieved (implemented by the Keras framework) are depicted in Figures 2 and 3.

<table>
<thead>
<tr>
<th>Table 1: Resulted Configurations(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hidden_size</td>
</tr>
<tr>
<td>1st variant &amp; average accuracy</td>
</tr>
<tr>
<td>1st variant &amp; validation accuracy</td>
</tr>
<tr>
<td>2nd variant &amp; validation accuracy</td>
</tr>
<tr>
<td>2nd variant &amp; validation perplexity</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2: Resulted Configurations (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch_size</td>
</tr>
<tr>
<td>1st variant &amp; average accuracy</td>
</tr>
<tr>
<td>1st variant &amp; validation accuracy</td>
</tr>
<tr>
<td>2nd variant &amp; validation accuracy</td>
</tr>
<tr>
<td>2nd variant &amp; validation perplexity</td>
</tr>
</tbody>
</table>

The hyperparameters used and their respective search spaces were the same for both algorithms. Specifically, the hyperparameters were the size of the hidden layers (hidden_size), the number of the hidden layers stacked (num_layers), the size of the training batches (batch_size), the number of timesteps per input (num_steps), as well as the number of the training epochs used (num_epochs). The retain length for the survivors of each generation is also fixed to 40%. The search spaces for each hyperparameter were the following:

- Hidden layer size: [200, 1200]
- Number of hidden layers: [2, 4]
- Batch size: [10, 50]
- Number of timesteps: [10, 50]
- Number of epochs: [1, 100]

Four major experiments were conducted, where the two variants of the algorithm utilized the fitness functions stated above. The resulting configurations are stated in Table 1.

The result considering the validation perplexity as a fitness function was ignored, as it performs poorly due to its small number of training epochs. This signifies either the poor efficiency of the perplexity as a fitness function, or the setback at a local minimum. It should be noted that, similar tests were conducted with the validation perplexity as fitness and they tended to be inefficient, compared to the results of the validation accuracy as fitness. The rest of the resulting configurations were used to build respective language models, and therefore trained and tested alongside the default configuration used by the majority of similar applications (i.e. the Tensorflow framework [14]). The default configuration is usually defined as:

- Hidden layer size: 600
- Number of hidden layers: 2
- Batch size: 20
- Number of timesteps: 35
- Number of epochs: 60

The comparison graphs of the validation accuracy and validation perplexity achieved (implemented by the Keras framework) are depicted in Figures 2 and 3.
accuracy as fitness function. Due to its small batch size and number of timesteps though, it results in quite a slow training process, which can turn out to be a major issue for big datasets or industrial use. It is also apparent that its respective graph curve is starting to converge from early on (50 – 60 epochs), compared to the curves of the other configurations, which appear to have potential in case of further training.

The second highest performance (~0.48) is achieved by the configuration resulted by the 2nd variant and the validation accuracy as the fitness function. Despite its lower accuracy, it is much less time-consuming during training (almost twice as fast) and according to the slope of its graph curve, still has potential for further improvement, for an expanded set the algorithm’s search space. The rest of the configurations were of lesser performance.

V. CONCLUSION

Through the experiments conducted above, we were able to show that the genetic algorithm consists an efficient and viable method for fine-tuning a neural network. The experimental results show that its resulting configurations outperform the default setting usually proposed [13], [14], while the whole approach offers both affordability and sufficient convergence.

There are numerous ways in which the current work can be continued. By expanding the search space of the algorithm, the resulting configurations could be considerably improved. In addition, different hyperparameters can be examined, like embedding approaches (e.g. word2vec [15], GloVe [16]) or learning algorithms (e.g. Adam [17], RMSprop [18] etc.). Moreover, different variations of genetic algorithms could be deployed (e.g. tournament or roulette selection). In the end, the genetic algorithm could be extended with ease to other neural architectures, like bidirectional or hybrid models.

REFERENCES

[11] https://keras.io/
[12] https://www.tensorflow.org/
[14] https://www.tensorflow.org/tutorials/sequences/recurrent
[15] “Distributed Representation of Words and Phrases and their Compositionality”, Mikolov et al., 2014