Hyperparameter optimization with genetic algorithms

Tristan Cadet*

* GSIST, The University of Tokyo

Abstract- State-of-the-art architectures in deep learning are more and more complex and require a long process of trial an error to be fine-tuned. The repetitive task of tweaking the architecture and parameters of machine learning models should be left to the machine instead of relying on faillible humans. Stimulated by this thought, the field of automated machine learning has recently started to gather interest, the most obvious manifestations being Google's AutoML and the recent AutoKeras. Following this trend, in this paper we try to optimize the architecture and other hyperparameters of a feed-forward neural network via a genetic algorithm, and demonstrate that the final networks perform better than a robust baseline.

Index Terms- deep learning, genetic algorithm, hyperparameter, neural network, optimization, neuroevolution

I. INTRODUCTION

Automation of the optimization process is an important objective for the field of machine learning. This is because it would allow to reliably produce high-quality networks that can be used for real-world applications. A few commonly used techniques exist to tune hyperparameters, such as grid search, random search, bayesian optimization and tree-structured parzen estimators. However there is no miracle solution yet, and the field of research is still open. A somewhat less common technique, namely genetic algorithms, has recently yielded good results for hyperparameter optimization[1]. The algorithm used in this paper is an extension of CoDeepNeat[2]. The concept of CoDeepNeat is to evolve separately 2 populations using the NEAT algorithm: modules (DAG of layers) and blueprints (DAG of pointers of modules). The modules are then inserted in blueprints and this temporary network population is evaluated by training it on a supervised learning task. The fitnesses of the individuals (networks) are attributed back to blueprints and modules as the average fitness of all the assembled networks containing that blueprint or module. This algorithm allows to create a repetitive structure by reusing complex modules according to a blueprint, which is characteristic of latest state-of-the-art models.

In this paper we limit ourselves to dense layers feed-forward network architectures, as it is much simpler to analyze and implement, yet enough to demonstrate the value of genetic algorithms in the field of hyperparameter optimization.

II. EXPERIMENT PROCESS

Sticking to the general genetic algorithms workflow, we start by initializing a population of neural networks of size **N**. The initializer sets the hyperparameters for each network by randomly picking values in a predefined search space.
In our experiment, we use the following search space:

```
nb_layers = range(1, 6 + 1)
layer_dim= [1, 2, 4, 8, 16, 32, 64]
batch_size = [32, 64, 128, 256]
dropout_rate = {'min': 0.1, 'max': 0.6,
'precision': 2}
activation = ['relu', 'elu', 'tanh',
'sigmoid', 'hard_sigmoid', 'softplus',
'linear']
optimizer = ['rmsprop', 'adam',
'adagrad', 'adadelta', 'adamax',
'nadam']
```

The dropout rate is chosen following a normal distribution. Other hyperparameters are chosen following a uniform distribution. Hyperparameters that are not in the search space are not tuned, and use the default values of Keras.

As we can guess from the search space, the architecture of the network is entirely built by the genetic algorithm, except for the output layer that is fixed to a Dense layer with an output size of 1 and a sigmoid activation for binary classification.

The next step is to compute the fitness of each individual in our population. To do that, we train each network on the IMDB reviews sentiment classification dataset. This dataset is composed of 50,000 polarized reviews, half being negative and half positive. We use a balanced set of 25,000 reviews for training from which we take 8,192 for validation. The purpose of the task is to classify the reviews as positive or negative. We chose to reduce the vocabulary size from 10,000 to 2,048 and cut the review after 256 words in order to increase the training speed. The training lasts 6 epochs with early stopping if the validation accuracy goes down from one epoch to another. The fitness of an individual is defined as the best accuracy obtained on the validation set out of the training epochs.

After computing the fitness of all individuals in the generation, we add them to the global fitness ranking.

Then comes the evolution process. We produce M individuals by applying a crossover function, and M others by applying a mutation function to the M best individuals obtained so far in all generations. We also randomly pick and mutate M individuals from the current generation. And finally we fill the remaining space with N-3*M randomly initalized individuals. The crossover function takes two parents and produces two children by randomly combining the hyperparameters of the parents as illustrated by Fig. 1.

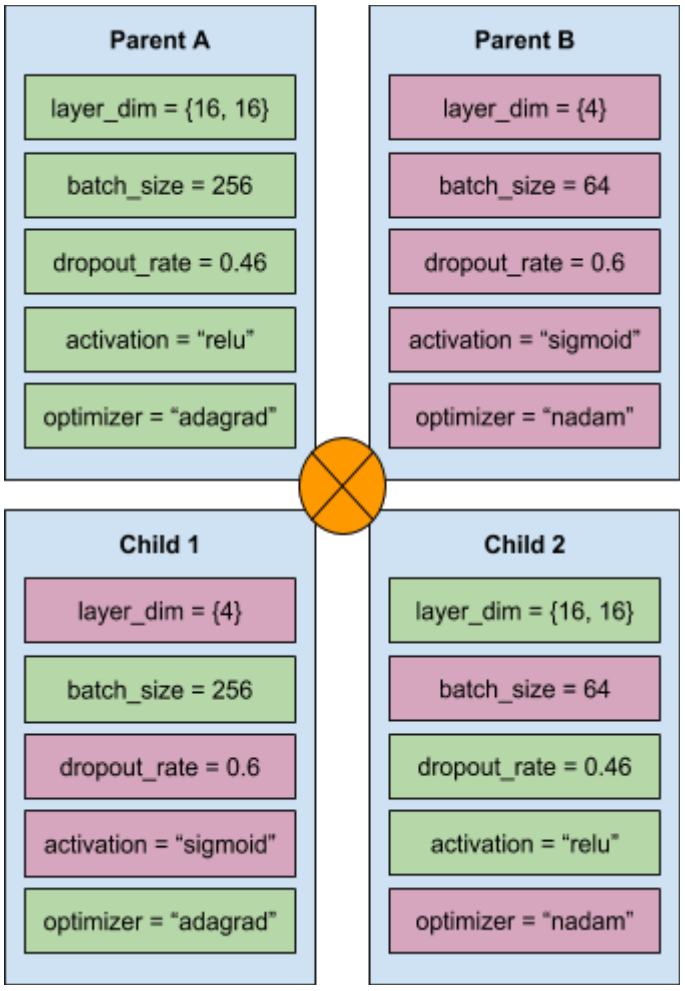

Fig 1. Crossover

The mutation function choses one hyperparameter to mutate and goes one step on the left or on the right in the hyperparameter space. For example the activation function 'elu' can be mutated to 'relu' or 'tanh' only, the layer_dim 64 can be mutated to 32 or 1. The dropout_rate is a special case since it is continuous, so the mutation picks a number following a normal law centered on the current dropout value, and rounds it to 2 decimals. The mutation can also insert a layer with a random dimension, or remove one.

Finally we ensure uniqueness by mutating individuals until they have a hyperparameter combination that has not been evaluated before.

We now have a new generation, hopefully better than the previous one, and can go back to the fitness computation step.

We iterate this process for G generations, and at the end we take the best individuals according to their fitness in order to retrain them separately. This time we will train them on all 25,000 reviews of the training set before evaluating them, and for comparison purposes we use a vocabulary size of 10,000 and full reviews.

III. BASELINES

To put our results into perspective, we use three baselines. A statistical baseline, which has a 0.5 accuracy for the IMDB dataset (this is the accuracy we obtain if we always output 'positive' or if we always output 'negative' since the set contains an equal proportion of positive and negative reviews).

A good solution to the problem, from the book Deep Learning with python[3]. It uses the following hyperparameters: layer_dim = {16, 16}, batch_size = 512, no dropout, activation = 'relu', optimizer = 'rmsprop' ; and obtains a 0.88328 accuracy on the test set.

A state-of-the art architecture[4]: using paragraph vectors followed by a multi-layer perceptron it obtains on accuracy of 0.945

IV. Results

We launched the experiment with 51 generations and 32 individuals by generation, evaluating a total of 1,632 networks. The full results of the genetic algorithm can be found in the attached 'top1632.txt' file. Out of this top we took the 10 first individuals and retrained them on 25,000 reviews (see the attached file 'top10.txt' for detailed results).

Out of the 10 architectures all beat the statistical baseline by a clear margin with a mean accuracy of 0.88916.

We have only one model that performs worse than our second baseline with an accuracy of 0.87464, another that performs equally well (0.88212) and all the other that outperform it with accuracies over 0.89. The top three model beat the second baseline by almost 1 percent with accuracies of 0.89256, 0.89304 and 0.89308 respectively. These results are very encouraging.

However we are still far from the state-of-the-art baseline, and the 5% gap in accuracy really shows the limit of our simple approach based on feed-forward networks and dense layers.

Looking at the data we got from our genetic algorithm we can make a few observations. Let's consider the top 24 individuals (fitness over 0.870, remember that it is lower than final training because we used a smaller vocabulary), the top 85 (fitness over 0.869), the last 62 that failed to learn (fitness under 0.545) and the entire set.

	#batch=32	#batch=64
top 24	21	3
top 85	74	11
top 1632	961	314
last 62	25	12

Tab.1 number of networks with bach size B in function of rank

From Tab.1 we can assume that a small batch size is necessary to obtain top performances, and this independantly of other parameters. This is a very important result because it means batch size can be tuned separately of other parameters which reduces the dimension of the search space.

	#rmsprop
top 24	24
top 85	84
top 1632	1078
last 62	25

Tab.2 number of networks with rmsprop in function of rank

From Tab.1 we could assume that rmsprop is the optimizer to go with, but it might not be the case because we used only Keras default optimizer's parameters, which may not work equally well for all optimizers.

	#(hard)sigmoid	#(r)elu
top 24	14	6
top 85	50	22
top 1632	379	874
last 62	48	6

Tab.3 number of networks with activation A in function of rank

The sigmoid and hard-sigmoid seem to work better than relu and elu, though the trend is not as clear as in the previous examples.

dropout	[0.1, 0.2[	[0.2, 0.3[
top 24	21	1
top 85	71	7
top 1632	1040	184
last 62	26	6

Tab.4 number of networks with dropout in interval I in function of rank

A small dropout rate seems to yield better results.

#layers	[1,3]	[4, 6]
top 24	22	2
top 85	83	2
top 1632	1447	185
last 62	8	54

Tab.5 number of networks with layer count in interval I in function of rank

We see that a small number of layers leads to better performance, and conversely a high number of layer seems to lead to poor performance as shown by the last 62. However there are some exceptions to this rule as shown in the top 24 ([16, 2, 8, 2, 2] ranked 5th and [8, 1, 8, 2] ranked 24th), probably due to the particular ordering and dimensions of the layer.

From all these observations we can infer that a model using a batch size of 32, rmsprop as an optimizer, a sigmoid or hard-sigmoid activation, a small dropout and a small number of layers will likely perform well on IMDB. If the number of layers is high it will likely perform badly.

V. Conclusion

Our simple genetic algorithm managed to produce networks that beat a robust baseline by 1%, proving the relevance of genetic algorithms for hyperparameter optimization.

However a simple GA is just not enough to beat SotA. The main reason is that our search space is limited to feed-forward dense networks and probably doesn't contain an architecture better than SotA. To improve our results, we must first enlarge the search space so that it contains potential state-of-the-art architectures. This can be done by looking at current SotA. The second step is to come up with a clever GA to be able to find a good architecture in a reasonable amount of time. Though in our case we were fine because we used only dense layers and a relatively small parameter space (11,529,504,000 possible individuals), as soon as we allow arbitrary architectures and add LSTM and Conv layers the training time of each individual explodes as well as the size of the search space.

There are already a few ideas for these genetic algorithms[1], and in my opinion what we lack most is a modular framework for neuroevolution that would allow fast experimentation and efficient implementation of latest algorithms.

References

[1] Jason Liang, Elliot Meyerson, and Risto Miikkulainen. Evolutionary Architecture Search For Deep Multitask Networks

[2] Risto Miikkulainen and al. Evolving Deep Neural Networks

[3] François Chollet. Deep Learning with python, ch 3.4.

[4] James Hong, Michael Fang. Sentiment Analysis with Deeply Learned Distributed Representations of Variable Length Texts.

Authors

Author – Tristan Cadet, student in IST, The University of Tokyo