AutoCompete 2.0: A Framework for Automating the Optimization of Neural Networks

Abhishek Thakur A.THAKUR@SEARCHMETRICS.COM
Searchmetrics Inc., Berlin, Germany

Abstract

This paper proposes a framework for tuning a neural network by carefully choosing parameters and rules for preprocessing the concerned dataset. The rules discussed here have been derived after working on hundreds of datasets. The paper doesn't go into the details of neural networks rather describes how one can easily create and build neural networks with good performance without much effort and with a very little knowledge about the data.

Keywords: machine learning, neural networks, hyper-parameter tuning

1. Introduction

In recent years, everyone has been talking about neural networks. This paper will also talk about neural networks but from an applied point of view. Now-a-days, with advancements in technology and availability of cheaper GPUs, anyone can train deep neural networks. However, training a deep neural network has a lot of challenges, such as choosing the appropriate pre-processing steps, the type of neural network, the number of layers and choosing the appropriate hyper-parameters.

In this paper, we will see how we can minimize human interference while designing neural networks by selection of preprocessing steps and the network architectures based on a framework derived from a set of rules learned after using neural networks for 30-50 different datasets.

The paper is divided into four sections. Section 2 provides an overview of the neural network libraries available in python. In section 3, we show how to build neural networks with keras and python with the proposed framework. Section 4 discusses the results obtained by this framework in the AutoML GPU track and section 5 concludes the paper with what we learned by using this framework and future work. References are listed at the end of the paper.

2. Neural Network Libraries in Python

Python is a high-level programming language which offers very fast development since its simple, cross-platform compatible and does not involve any compilation steps. Due to the popularity of python, a number of deep learning libraries have been developed which provide an interface in python, some of the examples being tensorflow (Abadi et al., 2015), theano (Theano Development Team, 2016), caffe (Jia et al., 2014), lasagne (Dieleman et al., 2015), keras (Chollet, 2015), etc. Figure 1 presents the popularity of these deep learning libraries.

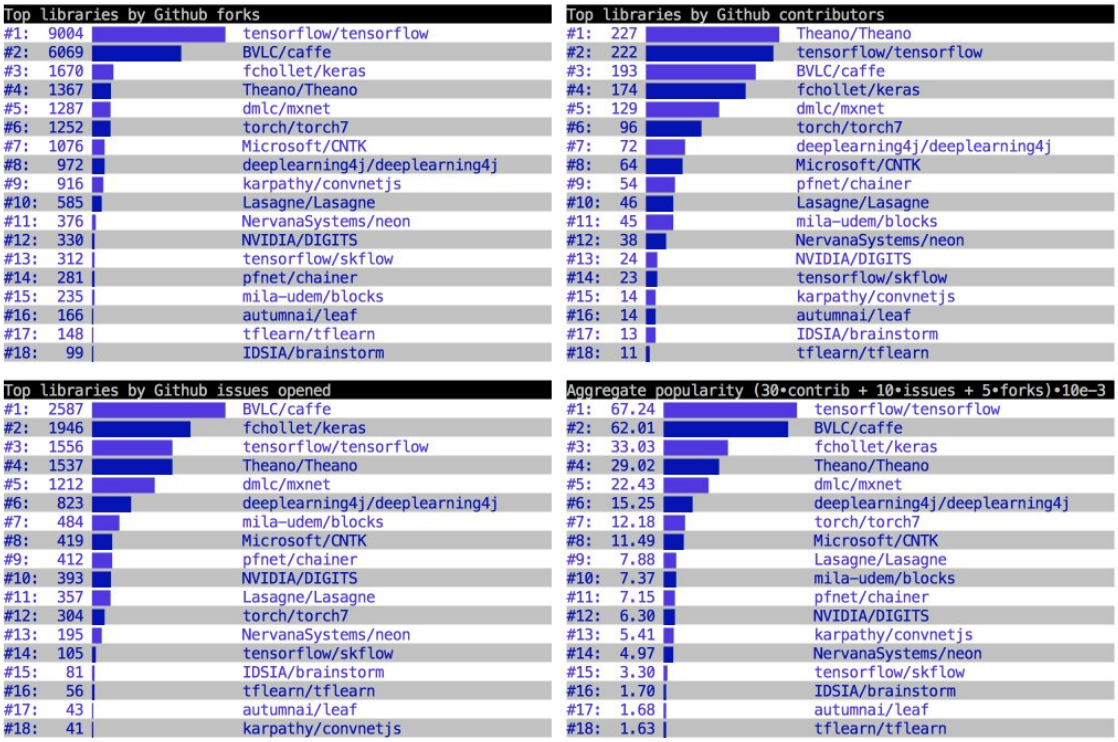

Figure 1: The deep learning landscape, May 2016

In this paper, we will focus on Keras, the deep learning framework, originally authored by François Chollet (Chollet, 2015). Keras provides a very easy, high level interface to both Tensorflow and Theano which enables fast experimentations. We used keras because of its simplicity, speed and ease of development and testing of deep learning models.

A sequential model in keras can be implemented in the following manner:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(output_dim=32, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(output_dim=10))
model.add(Activation("softmax"))
model.compile(loss='binary_crossentropy', optimizer='sgd')
model.fit(X, y, nb_epoch=10, batch_size=32)
```

The next section discusses the proposed framework for preprocessing of data and rules for selecting the architecture and hyper-parameters for the neural networks.

3. A Framework for Selecting Neural Network Architecture

The framework for selecting the neural network architecture is similar to what has been discussed in Thakur and Krohn-Grimberghe (2015). Figure 2 provides an overview of the rules used for preprocessing the data.

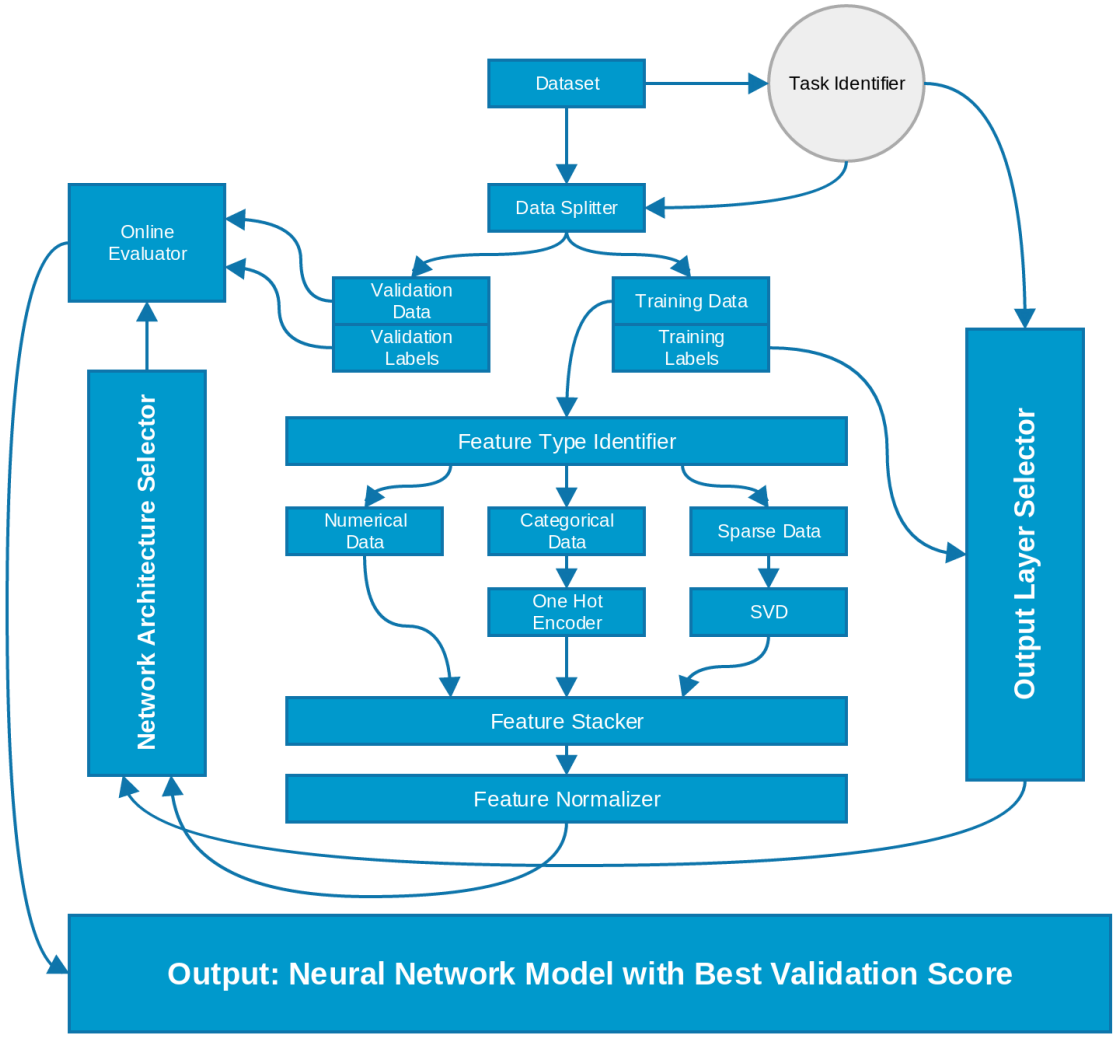

Figure 2: Framework for the proposed system.

The very first step in the framework is identification of the task. The task identifier identifies the task of the machine learning problem i.e. whether it is a multi-class classification task, multi-label classification task or a regression problem. The dataset then split to validation and training set depending on the labels. For a multi-class classification a stratified split is used to ensure that the ratio of classes in the validation set is same as that of the training set. Multi-label and regression datasets are split randomly. Task identifier stores the following information about the dataset (provided by the AutoML challenge (Guyon et al., 2015)):

- The type of features (binary, numeric, categorical, mixed)

- The type of dataset (dense or sparse)
- Total number of features
- Total number of samples for training, validation and test
- Missing features, if any
- Type of target (categorical, numeric)
- Number of target variables, and,
- The evaluation metric

Once we have the validation and training data separated, we move to identification of feature types. For neural networks, this step is simplified when compared to Thakur and Krohn-Grimberghe (2015). The numerical features are not touched and sent directly to feature stacker. If categorical features are present, they are one hot encoded and then passed to the feature stacker. In case of text data (after performing tf-idf) or very sparse data, we perform singular value decomposition (SVD). It has been found that the appropriate number of components for SVD like between 120 to 180 for most of the datasets and this is a number we can fit into GPU memory. In general, higher number of components may perform better but this difference is pretty small and comes at the cost of memory requirements and training time. The different features are then stacked together with the feature stacker.

One of the very important steps before feeding data into neural networks is data standardization or normalization. This step is performed by feature normalizer which does scaling of data based on mean and variance, scaling the features between 0 and 1, log scaling and scaling without mean. Datasets scaled using these different methods are then fed to the network architecture selector for selection of the number of layers, activation functions, loss function and optimizer to be used.

Neural networks need a lot of tuning and a lot of time is spent in optimization of the millions of hyper-parameters. It has been said that there is no rule of thumb for choosing the size of neural networks and their parameters. But even if there are no rules of thumb, there are certain rules that can be followed and a neural network with good performance can be built and optimized easily. The network architecture selector selects the network based on these rules which mostly consisted of sequential models with a dense layer, a batch normalization layer Ioffe and Szegedy (2015), a dropout layer Srivastava et al. (2014) to avoid over-fitting and an activation layer (ReLU Nair and Hinton (2010) or PReLU He et al. (2015)). The output layer selector chooses the appropriate final layer for the neural network. For multi-class classification problems we choose a softmax layer, for multi-label classification tasks a sigmoid layer is chosen and linear activation layer for regression tasks. Similarly, we choose categorical cross-entropy or multi-class logloss for classification tasks and squared error for regression tasks as the loss function. After learning from several datasets, it was found that best optimizers for most of the machine learning tasks are either stochastic gradient descent or adam optimizer Kingma and Ba (2014). Thus, we kept only these two optimizers in pipeline.

The most interesting step in network architecture selector is selection of number of layers and their parameters. This is performed by a modified random search. The selector starts

with a single layer network with a dense layer of 120-500 neurons, batch normalization, PReLU activation and a dropout of 10-20%. The performance is noted and an extra layer with the same configuration is added to the network. With the online evaluator, we measure the performance of the network with these two different configurations. The number of neurons is then increased in both layers to 1200-1500 and performance is noted again. If the network does not perform better than the previous configuration, the dropout is increased to 40-50%. A big network with 8000-10000 neurons is then chosen with the same dropout and results are recorded again. If everything fails or if the network overfits, the dropout is increased to a very high value ranging from 60% to 80% and performance is recorded again. The network selector keeps adding layers, changes number of neurons per layer, changes dropout, checks if batch normalization improves performance and records the performance of the network. Since all the values are chosen from a bucket of parameters with fixed ranges, we don't go into endless loop.

For the AutoML datasets, this whole process took less than one hour per dataset with a total runtime of approximately two hours for all the datasets. The results obtained by use of this framework on the AutoML datasets are discussed in the next section.

4. Results on AutoML Datasets

The AutoML competition lasted for almost a year with 6 phases. The last phase of the competition consisted of a GPU phase with the same datasets as the CPU phase. Every phase of the competition provided five different anonymous datasets. The datasets in the final phase were named as: evita, flora, helena, tania and yolanda. The following snippet shows the network build for yolanda using the framework discussed above.

```
dims = training_data.shape[1]

model = Sequential()
model.add(Dense(150, input_shape=(dims,)))
model.add(BatchNormalization())
model.add(PReLU())
model.add(Dropout(0.1))

model.add(Dense(150))
model.add(PReLU())
model.add(BatchNormalization())
model.add(Dropout(0.1))

model.add(Dense(1))
model.add(Activation('linear'))
model.compile(loss='mean_squared_error', optimizer='adam')

model.fit(training_data, labels, nb_epoch=20, batch_size=128)
```

The final standings and the scores obtained in the final phase of the AutoML GPU track are shown in Table 1.

Dataset	**Evita**	**Flora**	**Helena**	**Tania**	**Yolanda**
Evaluation Metric	**AUC**	**A Metric**	**BAC**	**PAC**	**R2**
Abhishek	0.5694	0.5001	0.2381	0.7617	0.3870
Damir	0.5816	0.5061	0.2469	0.7498	0.3654
AAD Frieburg	0.5866	0.4540	0.2673	0.7557	0.3782

Table 1: Results of the final phase for AutoML GPU track

The resulting networks from the framework secured first place in the GPU phase of the competition. AutoCompete (Thakur and Krohn-Grimberghe, 2015) combined with AutoCompete 2.0 secured third spot in the final phase of the CPU track shown in figure 3.

RESULTS

	User	<Rank>	Set 1	Set 2	Set 3	Set 4	Set 5
1	aad_freiburg	1.60 (1)	0.6530 (1)	0.5175 (2)	0.2843 (1)	0.7821 (1)	0.3823 (3)
2	ideal.intel.analytics	3.60 (2)	0.6137 (3)	0.5263 (1)	0.2455 (5)	0.7271 (7)	0.3863 (2)
3	abhishek4	5.40 (3)	0.5946 (6)	0.5064 (6)	0.2251 (7)	0.7574 (3)	0.3720 (5)

Figure 3: Final phase results for CPU track.

(Guyon et al.) discusses the results and methodologies used by the participants in detail.

5. Conclusion and Future Work

This paper proposed a framework for automatic selection of neural network architectures with pre-processing of the datasets. The framework tends to minimize human interference to a great extent while training and evaluating a neural network model.

Although the current framework works perfectly for both CPU and GPU systems, it still has a lot of scope for improvement. The next steps for improvement of the framework before publishing would be addition of more pre-processing steps, a better way of selection of parameters for the neural network models, addition of graphical models and sequential models, e.g. LSTMs and convolutional neural networks.

References

Autocompete GPU Implementation. URL https://github.com/abhishekkrthakur/automl_gpu.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

François Chollet. Keras. https://github.com/fchollet/keras, 2015.

Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, diogo149, Brian McFee, Hendrik Weideman, takacsg84, peterderivaz, Jon, instagibbs, Dr. Kashif Rasul, CongLiu, Britefury, and Jonas Degrave. Lasagne: First release., August 2015. URL http://dx.doi.org/10.5281/zenodo.27878.

I. Guyon, I. Chaabane, H. J. Escalante, S. Escalera, D. Jajetic, J. R. Lloyd, N. Macía, B. Ray, L. Romaszko, M. Sebag, A. Statnikov, S. Treguer, and E. Viegas.

I. Guyon, K. Bennett, G. Cawley, H. J. Escalante, S. Escalera, T. Kam Ho, N. Macià, B. Ray, M. Saeed, A. Statnikov, and E. Viegas. Design of the 2015 ChaLearn AutoML challenge. In *Proc. of IJCNN*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL http://arxiv.org/abs/1502.01852.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010. URL http://www.icml2010.org/papers/432.pdf.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2627435.2670313.

A. Thakur and A. Krohn-Grimberghe. Autocompete: A framework for machine learning competitions. In *AutoML Workshop, International Conference on Machine Learning 2015*, 2015.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.