PROCEEDINGS OF SCIENCE

Application of Deep Learning Technique to an Analysis of Hard Scattering Processes at Colliders

Lev Dudko, Petr Volkov, Georgii Vorotnikov and Andrei Zaborenko*

Skobeltsyn Institute of Nuclear Physics, M.V. Lomonosov Moscow State University, 1(2) Leninskie gory, Moscow 119991, Russian Federation
E-mail: lev.dudko@cern.ch, petr.volkov@cern.ch, georgii.vorotnikov@cern.ch, andrei.zaborenko@cern.ch

Deep neural networks have rightfully won the place of one of the most accurate analysis tools in high energy physics. In this paper we will cover several methods of improving the performance of a deep neural network in a classification task in an instance of top quark analysis. The approaches and recommendations will cover hyperparameter tuning, boosting on errors and AutoML algorithms applied to collider physics.

The 5th International Workshop on Deep Learning in Computational Physics
28-29 June, 2021
Moscow, Russia

*Speaker

PoS(DLCP2021)012

1. Physics task

In the scope of high energy physics analysis the measurement of t-channel single top-quark production is used as a benchmark to calibrate the analytical tools and assumptions. Then the developed methods are used to measure deviations from the Standard Model – the Flavor Changing Neutral Currents (FCNC). The physics task is similar to the analysis of CMS collaboration [1]. Neural networks are used extensively throughout the analysis to separate background and signal events. First neural network model is used to filter out the multi-jet QCD events as these events are hard to model with existing Monte-Carlo methods. This network uses only five variables as its inputs and has relatively small number of trainable parameters. After the multi-jet QCD suppression a larger Standard Model neural network is used to identify top-quark events. These two different tasks allow us to test the performance of described methods in two separate instances.

2. DNN hyperparameter tuning

2.1 Overview

Most of machine learning models have two types of parameters: trainable and non-trainable[2]. Trainable parameters change during training, and non-trainable (or hyperparameters) are set by the user. For example, in a typical multi-layer perceptron hyperparameters can be the number of hidden layers, number of neurons in each hidden layer, the learning rate, regularization constants, etc. In this case trainable parameters are the weights and biases of each neuron.

Different hyperparameter combinations give the deep learning model varying degree of complexity and non-linearity. Therefore, hyperparameter tuning can help resolve overfitting and underfitting to a certain degree.

Finding the best combination of hyperparameters can be a challenging task as the model's performance can only be evaluated when the training is finished. Luckily, many hyperparameter optimization frameworks exist to automate this tedious process.

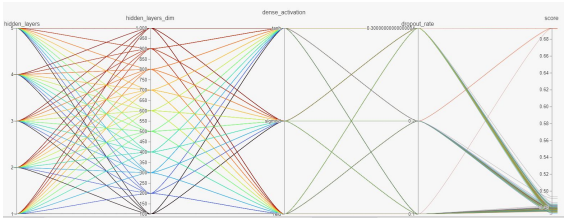
2.2 Hyperparameter tuning frameworks

One of the most established and well-known frameworks is called `Optuna`[3]. It provides tools to tune any machine learning model and quickly visualize the results. Another useful feature of `Optuna` is the budget – the amount of time used to tune a single model. This can be useful to compare different models and still give both fair treatment, tuning them for the same time.

However, we have opted to use `Keras Tuner`[2] for its tight integration with `Tensorflow Keras`, allowing us to use hyperparameter tuning with minimal code modifications and dependencies. After each trial this module generates a `.JSON` file containing all required information about a single run. If the user wishes not only get the best performing hyperparameter combination but also to explore the dependencies and tendencies of their deep learning model, the results can be parsed and visualized.

2.3 Tuner setup

The first major step in setting up any hyperparameter tuning is defining all possible hyperparameter combinations – the hyperparameter space. Usually the ranges of numeric hyperparameters



hidden_layers	hidden_layers_dim	dense_activation	dropout_rate	learning_rate	score
2	200	relu	0.30000000000000004	0.001	0.4725506901741028
2	300	relu	0.30000000000000004	0.001	0.4733211994171426
2	200	relu	0.2	0.001	0.4737839400768 28
2	300	relu	0.2	0.001	0.47381383180618286
2	400	relu	0.30000000000000004	0.001	0.4739852845668793
2	500	relu	0.30000000000000004	0.001	0.4740271568298 34
2	100	relu	0.2	0.001	0.47412174940109253
2	700	relu	0.30000000000000004	0.001	0.4745694969903229
2	400	relu	0.2	0.001	0.47469258308410645
2	800	relu	0.30000000000000004	0.001	0.4747413098812103
2	600	relu	0.30000000000000004	0.001	0.47488582134246826
3	300	relu	0.30000000000000004	0.001	0.4748902022838 5925
2	100	relu	0.30000000000000004	0.001	0.474922776222229
2	900	relu	0.30000000000000004	0.001	0.4749344289302826
2	1000	relu	0.30000000000000004	0.001	0.4749715924263 0005
2	700	sigmoid	0.30000000000000004	0.001	0.4750082790851593
2	700	relu	0.2	0.001	0.4750242531299591
3	200	sigmoid	0.30000000000000004	0.001	0.4750872850418091

Figure 1: Example of the hiplot interface

are defined either with a distribution or with a set array of values. The latter is done by defining the minimum and maximum values and setting the step parameter, the distance between two consecutive samples in the range. We have used this approach to visualize the relations between the model's performance and the values of its hyperparameters. The non-numerical hyperparameters (the hidden layers activation function, for example) are chosen with the `Choice` method.

The second step is to define a score variable – a metric to quantify and compare the model's performance. In the default case this variable can be equal to model's loss value (binary crossentropy in our binary classification task) or to any pre-made or user-defined metric.

With the hyperparameter space expanding with each new variable, a suitable algorithm to navigate it is required. `Keras Tuner` provides three basic Tuner algorithms, each with its own advantages and drawbacks: `BayesianOptimization`, `Hyperband` and `RandomSearch`.

`BayesianOptimization` algorithm uses tuning with Gaussian process. This is the fastest built-in algorithm, however, it can only find the local minima in the hyperparameter space. In our tests it converged within approximately 10% of the total combinations in the hyperparameter space. The parameters this algorithm converged on were adequate albeit not the overall best.

`Hyperband` algorithm uses the performance of the first epochs to compare different hyperparameter combinations. We decided against this method as models with different learning rates will have different `Hyperband` performance which will not reflect their overall accuracy.

`RandomSearch` tuning algorithm randomly samples hyperparameter combinations from the hyperparameter space. This method does not use any fancy logic, however, it reliably provides a near-best result when covering 40-50% of the hyperparameter space.

The code covering the needed adaptations is available in the Appendix.

2.4 Results interpretation

After the tuning process is compete, all trials results are stored in `.JSON` files. The user can opt to use the best configuration without looking at other models, however, plotting the relations can provide useful insights into how the chosen model is performing.

For general overview one can use Facebook's `hiplot`[4] utility (Figure 1). This interface allows the user to quickly analyse the trained models, sort them by their performance and check specific hyperparameter combinations.

To further investigate the hyperparameter space, one can plot the relations between models' performance and the values of used hyperparameters. We give two examples of such visualization

in Figures 2 and 3. In the first set of plots covering the tuning of larger Standard Model neural network we demonstrate the relation between the model's performance and a certain hyperparameter value, averaging over the rest of hyperparameters. In the second set we used heatmaps to describe hyperparameter combinations for the smaller QCD suppression neural network.

Having investigated the hyperparameter space for two typical High Energy Physics tasks, we can give broader recommendations for neural network design for this field. First of all, using `ReLU` for hidden layers activation function is advisable. Standard `tanh` and `sigmoid` functions lead to worse performance in deeper, bigger networks. In both cases networks with one or two hidden layers performed better and more stable than their deeper counterparts. The number of nodes in the hidden layers varied depending on the amount of input features: for the bigger network with 50 input features the amount of neurons lied in range between 200 and 400, and for the smaller network with 5 input features it was closer to 120.

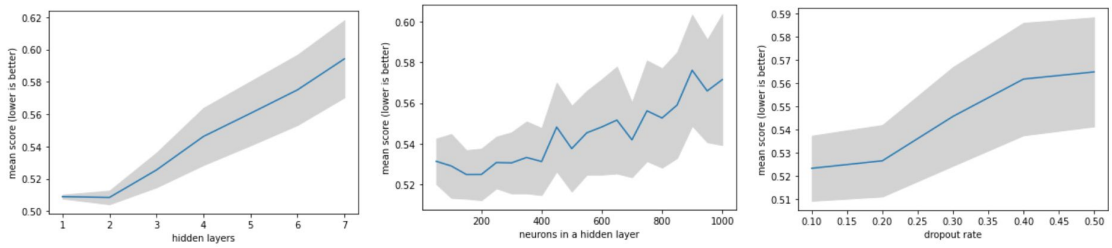


Figure 2: One-dimensional plots describing the relations between hyperparameter's values and model's performance for the Standard Model neural network

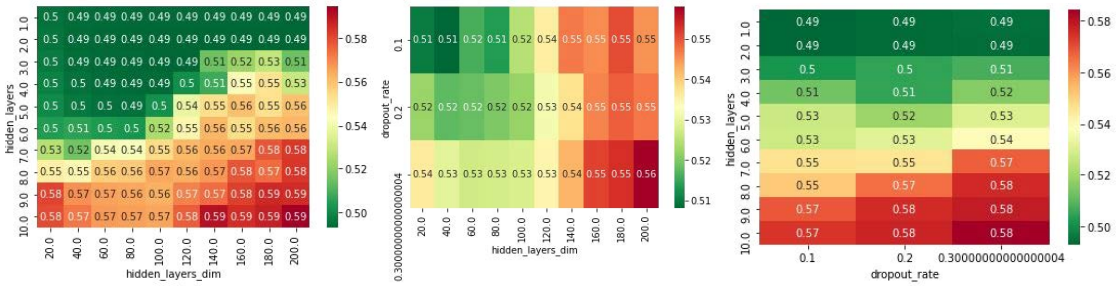


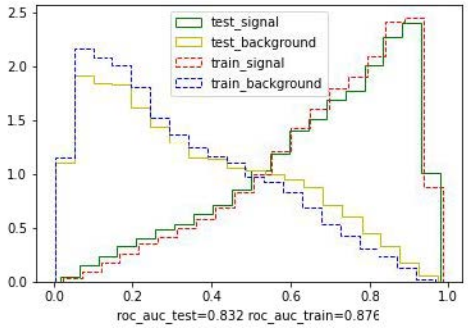
Figure 3: Two-dimensional plots describing the relations between hyperparameter's values and model's performance for the QCD suppression neural network

3. AutoML

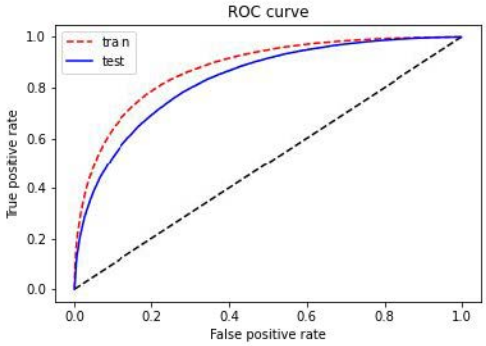
3.1 Overview

AutoML approach covers finding the optimal machine learning model, training it, evaluating it and tuning it if its performance is insufficient. In theory, given enough time and computational resources, this approach can yield an adequate model without investing researcher's time into complex architecture tuning and feature engineering. High energy physics data is close in structure to tabular data. In other areas tabular data may contain text and categorical data, but in high energy

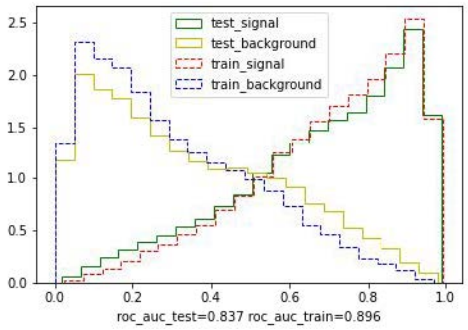
physics data is primarily numerical and can be organized into columns, so the task is simpler in a certain way.



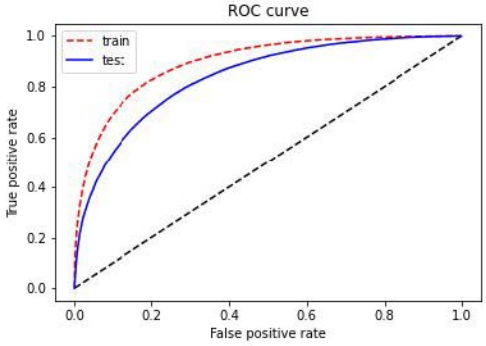
(a) Discriminator of the `Explain` mode model



(b) ROC Curve of the `Explain` mode model

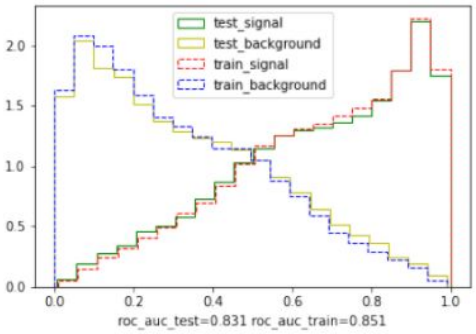


(c) Discriminator of the `Compete` mode model

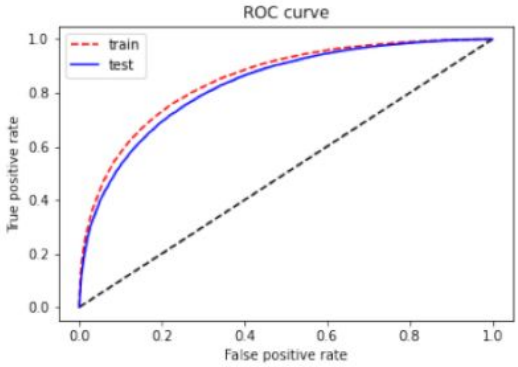


(d) ROC Curve of the `Compete` mode model

low_level_2_250_relu_0.2_0.01



(e) Discriminator of the tuned Tensorflow Deep Neural network



(f) ROC Curve of the tuned Tensorflow Deep Neural network

Figure 4: Comparison between two AutoML models and a tuned DNN model

3.2 Deep learning for tabular data

Deep learning has shown comparable performance[5] in tabular data classification to gradient boosting models (`CatBoost, LightGBM`), which are much cheaper in terms of computational

resources. However, our preliminary testing done using the `LightGBM` package showed that fine-tuned neural network performs slightly better and overfits less than a tuned `LightGBM`[6] model. Google's `Adanet`[7] package uses genetic algorithms to create a custom neural network structure for each machine learning task. This approach has a promising idea, however, the lack of support for weighted events limits its uses in high energy physics analysis where every event has a very specific weight value.

3.3 AutoML in high energy physics

We have opted for using `mljar-supervised`[8] library for automated machine learning. This library is easy to comprehend, has several performance modes (focused on data exploration, speed of inference or maximum accuracy of classification). It uses several machine learning algorithms (Linear, Random Forest, Extra Trees, LightGBM, Xgboost, CatBoost, Neural Networks, and Nearest Neighbors) for classification and then creates an ensemble of best performing models for final classification. Here we present the results of two AutoML models in `Explain` (runs in a dozens of minutes and performs Exploratory Data Analysis) and `Compete` (maximum classification accuracy, needs more computational time, we have run it for a day) modes as the speed of inference is not crucial in the current analysis. The performance comparisons are shown in Figure 4. Both classification modes provided good accuracy and even outperformed the tuned neural network in terms of `roc_auc` metric on the test dataset. However, this was done with a much higher degree of overfitting, thus reducing the AutoML model's predictive power.

3.4 Resume

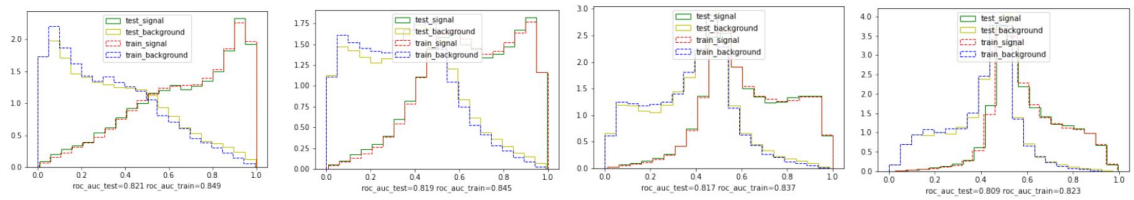
`mljar-supervised` provided a good baseline in all our use-cases, with its maximum accuracy mode overfitting a bit more that we would like it to. As it is much easier to control overfitting inside Tensorflow package through regularization and early stopping callback, for the time being we will continue to use it in our analysis, but this AutoML package came close to it in terms of classification accuracy. We will definitely monitor the development of this great tool and continue testing it.

4. DNN boosting on errors

We have also tried boosting on errors. The concept of this method is simple:

1. Train a machine learning model on unaltered data, get its predictions
2. Take the events that were miss-labelled after first classification and artificially increase their weights
3. Train a second model using the 'corrected' weight vector, supposedly achieving better performance, as this model will put more emphasis on difficult events that were hard to differentiate in the first place
4. Update the weights and repeat

This method did not work as the results of classification worsened after each iteration. The illustration of this performance degradation can be found on Figure 5. The best explanation we have come up with was that deep neural network is not a weak learner, which were noticed to benefit from such manipulations.



(a) Baseline **(b)** First iteration **(c)** Second iteration **(d)** Third iteration

Figure 5: Performance degradation with each boosting on errors iteration

5. L-regularisation

Through experimentation with input features we have found out that certain features worked extremely well for first-order classification, causing the model to increase its weights associated with these features, in turn lowering the importance of other, still needed, features. The proposed method included using l-regularization [9] to limit high weights so that they will not overshadow other weights as much.

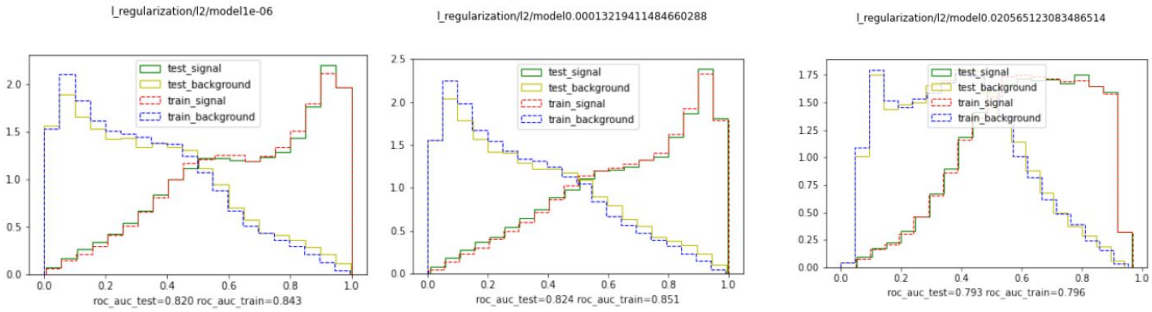
We have conducted the l-regularization study by getting a good baseline model from the `Keras Tuner`, fixing its hyperparametes and varying only the regularization constant. This was done for L1, L2 and L12 regularization types.

The results showed that when the regularization constant is chosen right, the discriminant distribution curve becomes smoother and general classification performance increases. However, when the regularization constant is too small, there are no perceivable improvements. High regularization constant values can be even more detrimental as they will outright decrease the classification performance of otherwise decent model. The illustration of these relations can be found in Figure 6.

We have not noticed considerable differences between the regularization types, L2 performed slightly better, but that disparity was within the margin of error.

6. Conclusion

We have demonstrated several approaches to improve the accuracy of classification based on a model of a Deep Neural Network in High Energy Physics. Here we provide a short summary of the methods described in the paper. DNN hyperparameter tuning is an effective method of improving the accuracy of the model, but it requires a lot of computational resources. The required computing time can be reduced by making hyperparameter space smaller and using a suitable optimization algorithm. We also provided recommendations based on our hyperparameter space exploration for typical high energy physics datasets. AutoML for tabular data can be used in High Energy Physics with relative success and little machine learning experience, however, the degree of overfitting



(a) The constant is too small **(b)** The constant is chosen well **(c)** The constant is too large

Figure 6: DNN model's dicriminators. The only difference between the models is the regularization constant.

is hard to control. Boosting on errors is not advised to use with DNNs, and using any type of L-regularization is advisable if the regularization constant is chosen correctly.

References

[1] CMS collaboration, V. Khachatryan et al., *Search for anomalous Wtb couplings and flavour-changing neutral currents in t-channel single top quark production in pp collisions at* $\sqrt{s}$ *= 7 and 8 TeV*, *JHEP* **02** (2017) 028 [1610.03545].

[2] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi et al., "Keras tuner." https://github.com/keras-team/keras-tuner, 2019.

[3] T. Akiba, S. Sano, T. Yanase, T. Ohta and M. Koyama, *Optuna: A next-generation hyperparameter optimization framework*, in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[4] D. Haziza, J. Rapin and G. Synnaeve, "Hiplot, interactive high-dimensionality plots." https://github.com/facebookresearch/hiplot, 2020.

[5] Y. Gorishniy, I. Rubachev, V. Khrulkov and A. Babenko, *Revisiting deep learning models for tabular data*, 2021.

[6] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma et al., *Lightgbm: A highly efficient gradient boosting decision tree*, in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 3149–3157, Curran Associates Inc., 2017.

[7] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri and S. Yang, *Adanet: Adaptive structural learning of artificial neural networks*, 2017.

[8] A. Płońska and P. Płoński, *Mljar: State-of-the-art automated machine learning framework for tabular data. version 0.10.3*, 2021.

[9] C. Cortes, M. Mohri and A. Rostamizadeh, *L2 regularization for learning kernels*, 2012.

A. Code for Keras Tuner adaptation

The function that returns the base model:

```
def createModel(dim, hidden_layers, hidden_layers_dim, dense_activation,
↪  dropout_rate, learning_rate):
    model = Sequential() model.add(Input(shape=(dim,)))
    for n in range (hidden_layers):
        model.add(Dense(hidden_layers_dim, activation=dense_activation))
        model.add(Dropout(rate = dropout_rate))
    model.add(Dense(units=1, activation='sigmoid'))
    adam = Adam(lr=learning_rate)
    model.compile(loss='binary_crossentropy', optimizer=adam,
    ↪  metrics=['mean_squared_error'])
    return model
```

The function that returns the model adapted to the Keras Tuner environment:

```
def build_model(hp):
    model = Sequential()
    model.add(Input(shape=(dim,)))
    for n in range (hp.Int('hidden_layers', min_value = 1, max_value = 7,
    ↪  step = 1)):
        model.add(Dense(units = hp.Int('hidden_layers_dim', min_value =
        ↪  50, max_value = 1000, step = 50), activation =
        ↪  hp.Choice('dense_activation',values=['relu', 'tanh'])))
        model.add(Dropout(rate = hp.Float('dropout_rate', min_value =
        ↪  0.1, max_value = 0.5, step = 0.1)))
    model.add(Dense(units=1, activation='sigmoid'))
    adam = Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3])))
    model.compile(optimizer=adam, loss='binary_crossentropy')
    return model
```