Application of deep learning technique to an analysis of hard scattering processes at colliders

A. Zaborenko[1], L. Dudko[1], P. Volkov[1], G. Vorotnikov[1]

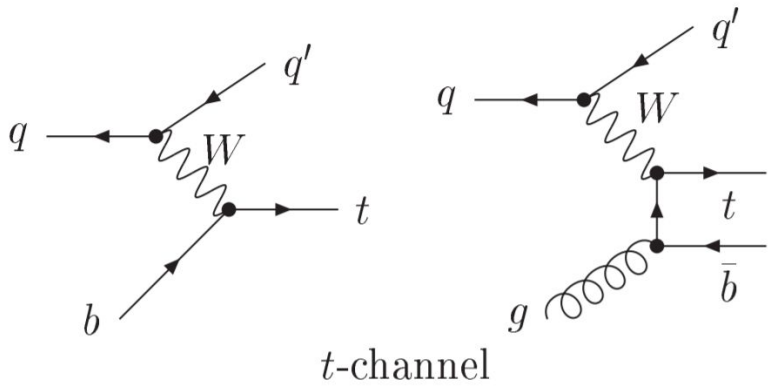
[1]SINP MSU, Moscow

Outline

1. Physics
2. Used methods
   2.1. DNN hyperparameter tuning
   2.2. AutoML methods
   2.3. Trying out boosting on errors
   2.4. L1, L2 Regularization
3. Conclusion

Physics

In the current analysis we use the measurement of t-channel single top-quark production cross section as a benchmark to test and improve our analytical tools and assumptions. Afterwards these methods are applied to measure deviations from the Standard Model (FCNC).



Feynman diagrams of t-channel single top-quark production

Physics

Neural networks are used twice during the analysis:

1. For **QCD** background suppression (5 features). We need to limit this background process as much as possible as it is poorly modelled with MC methods.
2. For selecting **t-channel** single top-quark production events from other Standard Model events recorded at the CMS detector (around 50 features).

Task specifics:

- The task at hand involves **binary classification** between signal and background events.
- The number of events is sufficient for adequate model training.
- The **speed** of classification is not a priority, so **accuracy** should not be sacrificed.

DNN hyperparameter tuning: overview

- **The idea:** Finding the best combination of hyperparameters for a given or similar task.
- **The requirements:** code modifications of varying complexity depending on a hyperparameter optimization framework, a lot of computational resources.
- **What to tune:** the amount of hidden layers, the number of nodes in the hidden layers, the activation function, the dropout rate.
- **The results:** an optimized neural network with varying degree of improvements.

DNN hyperparameter tuning: choosing the framework

- We have chosen **Keras Tuner** as we're working with `tensorflow.keras` DNNs.

  Another popular hyperparameter optimization framework is *Optuna*: it can tune any ML model and quickly visualize the results.

- All models are **ranked** by a "score" variable: a metric that defines the performance of a given model. It can be simply model's loss, an Sklearn or a user-defined metric.
- In order to start hyperparameter tuning user has to define a **hyperparameter space**: all possible parameter combinations.

DNN hyperparameter tuning: adapting the code

Keras model that is needed to be tuned:

```
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
```

DNN hyperparameter tuning: adapting the code

The same model adapted for the tuner, hyperparameter space is defined

```
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'))

  model.compile(optimizer=tensorflow.keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3])), loss =
'binary_crossentropy')

  return model
```

DNN hyperparameter tuning: choosing the optimizer and starting the search

```
tuner = RandomSearch(
    build_model,
    max_trials = 10000,
    objective='val_loss',
    executions_per_trial=3,
    overwrite=False,
    directory='../tuner',
    project_name='low_level'
)
```

Configure optimization strategy.

```
tuner.search(features_train, labels_train,
batch_size=len(labels_train),epochs=3000,
shuffle=False,
validation_data=(features_test, labels_test,
weights_test), sample_weight=weights_train,
callbacks = [stop_early, reduce_lr], verbose=2)
```

Run `tuner.search` with the arguments of `model.fit`.

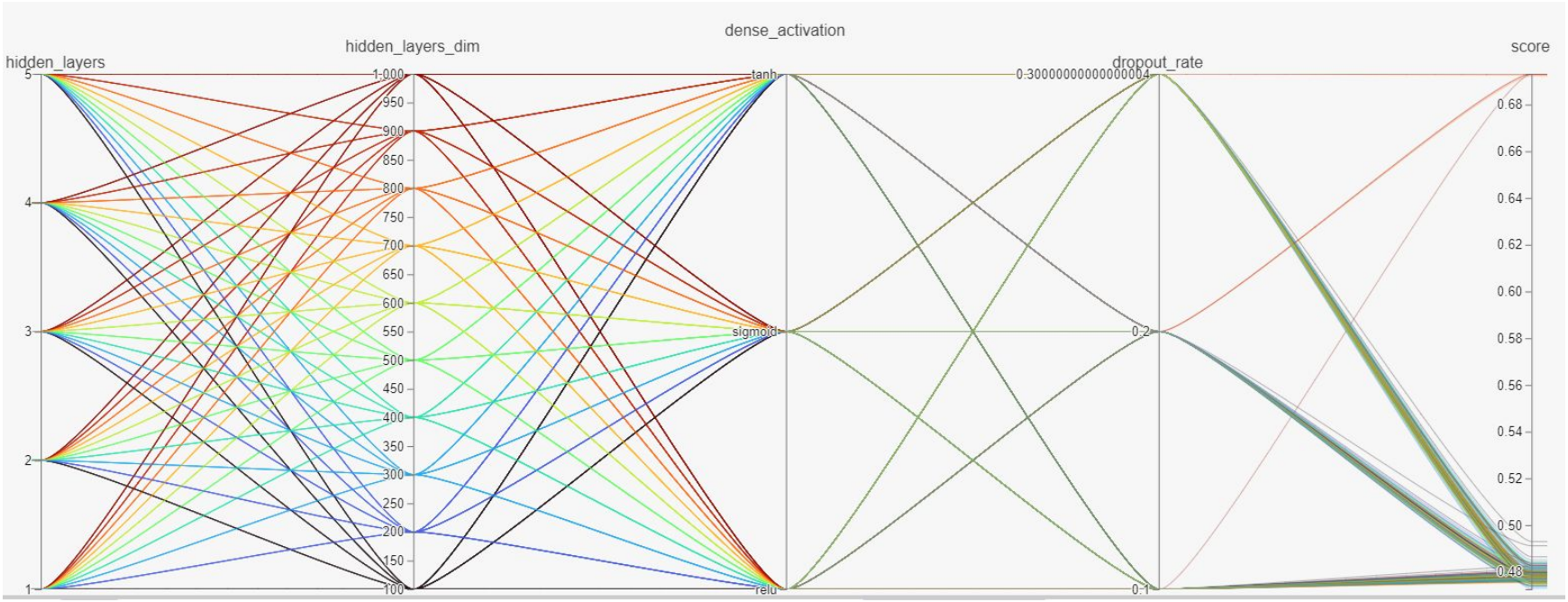
Importantly, EarlyStopping callback was used to prevent excessive overfitting and unnecessary calculations

DNN hyperparameter tuning: optimization strategy

- Simple **GridSearch**: tries all combinations in the parameter space. Takes the longest time but fully covers all combinations.
- **RandomSearch**: same as GridSearch, but tries combinations in a random order. It is somewhat more time-efficient, but we can do better.
- **BayesianOptimization**: at first tries several combinations at random, then intelligently moves in a parameter space to a local minimum.
- **Hyperband**: judge model performance after training a model for a small amount of epochs. We deemed this optimization method ineffective in the scope of the current task.

DNN hyperparameter tuning: visualizing the results

In general, we recommend visualizing tuning results using Facebook's `hiplot`:



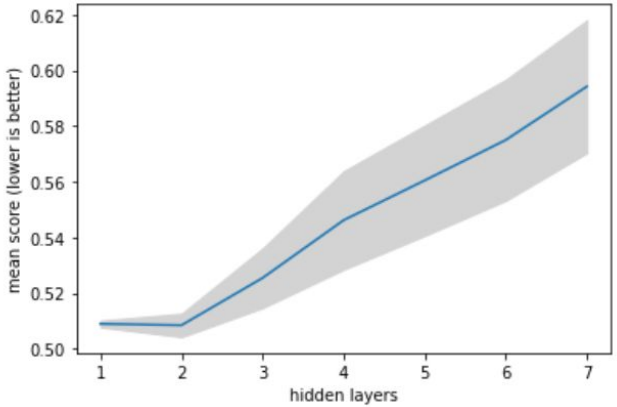
DNN hyperparameter tuning: visualizing the results

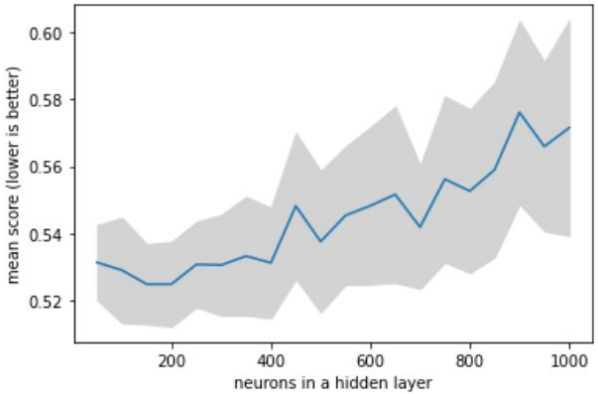
In general, we recommend visualizing tuning results using Facebook's `hiplot`:

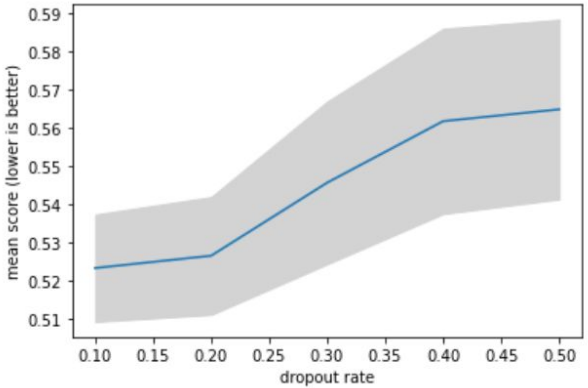
hidden_layers	hidden_layers_dim	dense_activation	dropout_rate	learning_rate	score
2	200	relu	0.30000000000000004	0.001	0.4725506901741028
2	300	relu	0.30000000000000004	0.001	0.47332119941711426
2	200	relu	0.2	0.001	0.473783940076828
2	300	relu	0.2	0.001	0.47381383180618286
2	400	relu	0.30000000000000004	0.001	0.4739852845668793
2	500	relu	0.30000000000000004	0.001	0.474027156829834
2	100	relu	0.2	0.001	0.47412174940109253
2	700	relu	0.30000000000000004	0.001	0.4745694696903229
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	0.47489020228385925
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	0.47497159242630005
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

DNN hyperparameter tuning: visualizing the results

If entire hyperparameter space was scanned we can plot the relations between a value of a certain hyperparameter and it's mean score.
Plots for the SM neural network trained on low-level features:

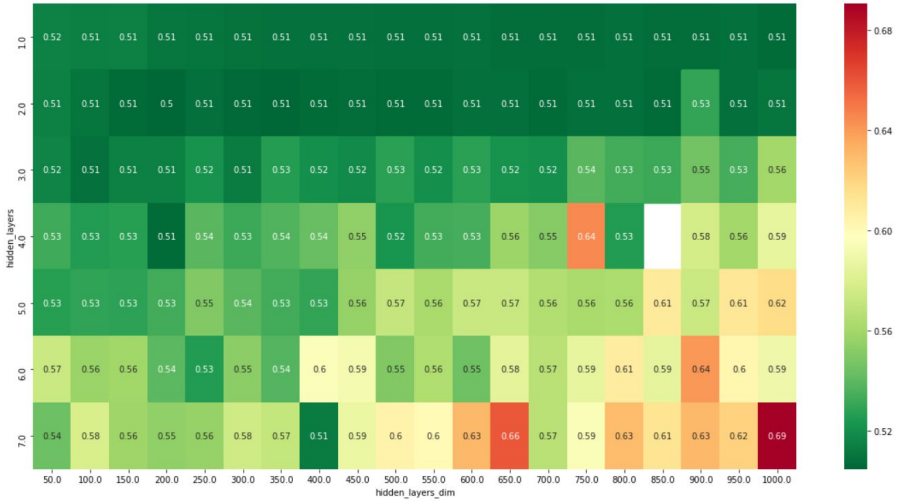


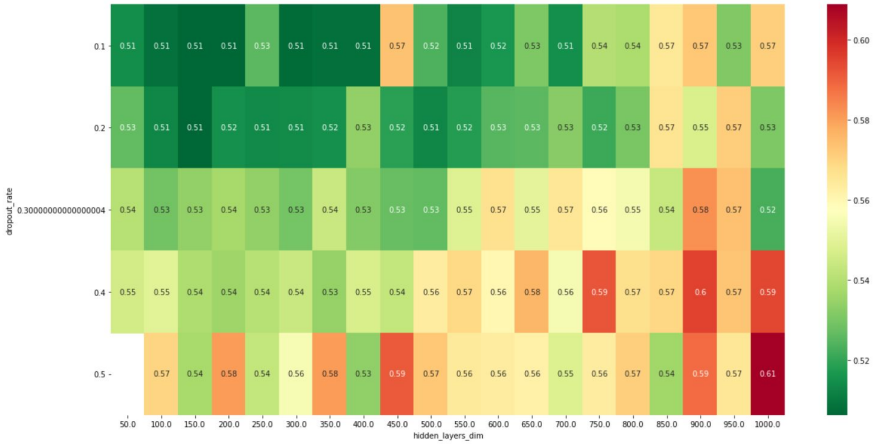


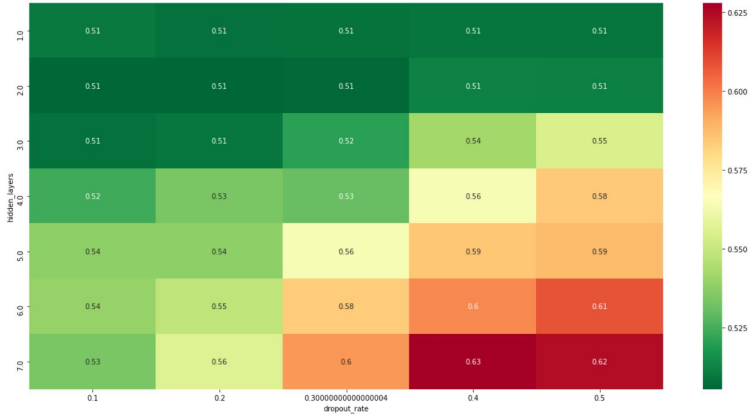


DNN hyperparameter tuning: visualizing the results

Same results visualized for two parameters at once:



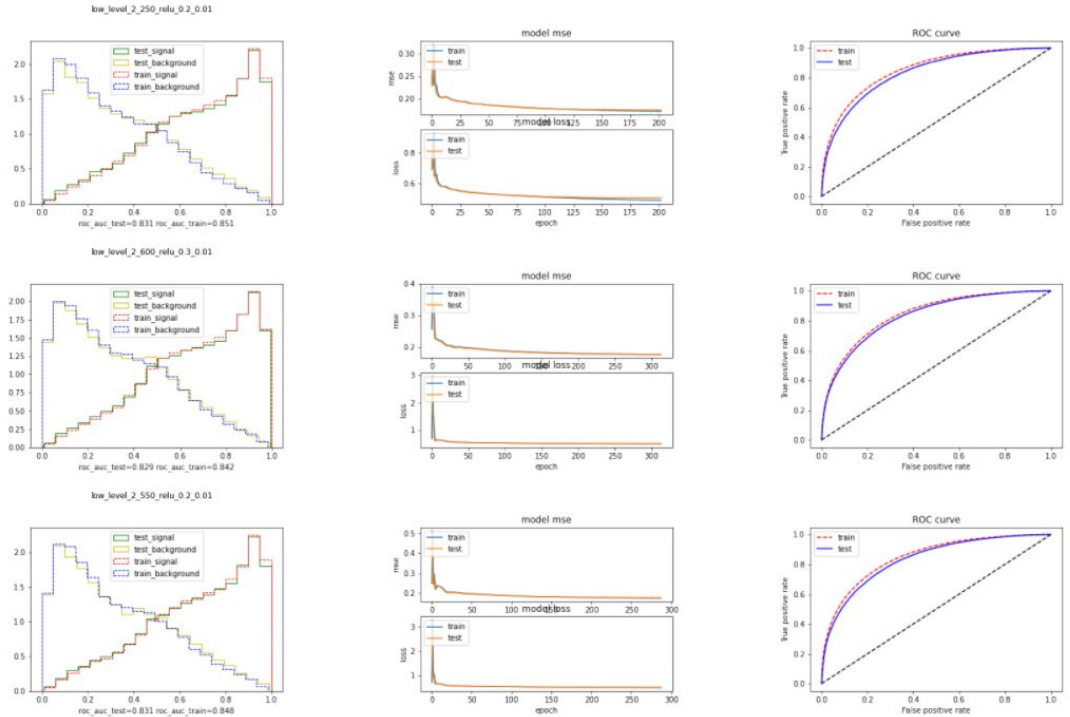




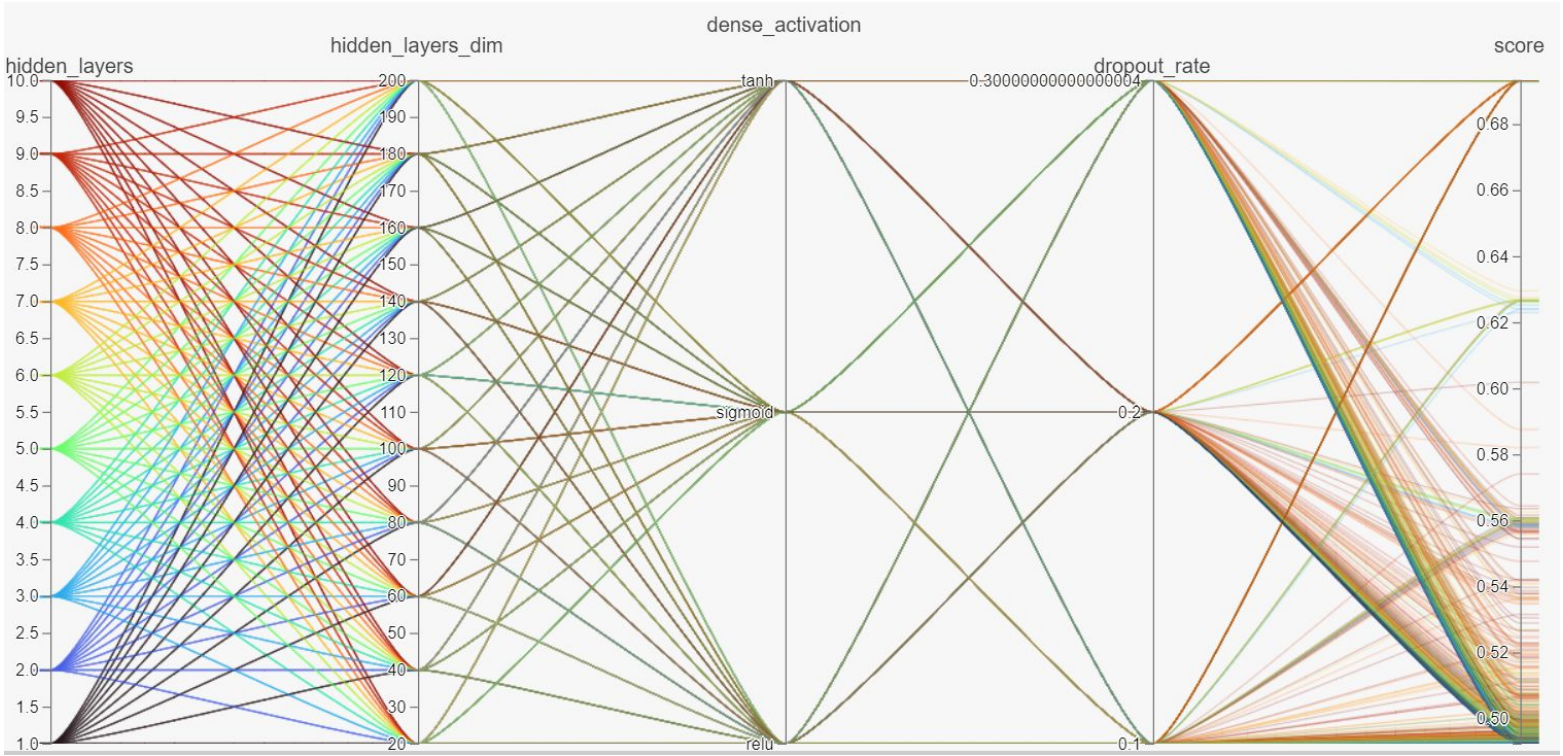
DNN hyperparameter tuning: visualizing the results

We also plot the discriminators of the best networks and evaluate their loss and ROC curves.

The outputs of the best performing networks are **very close** in terms of shape and performance.



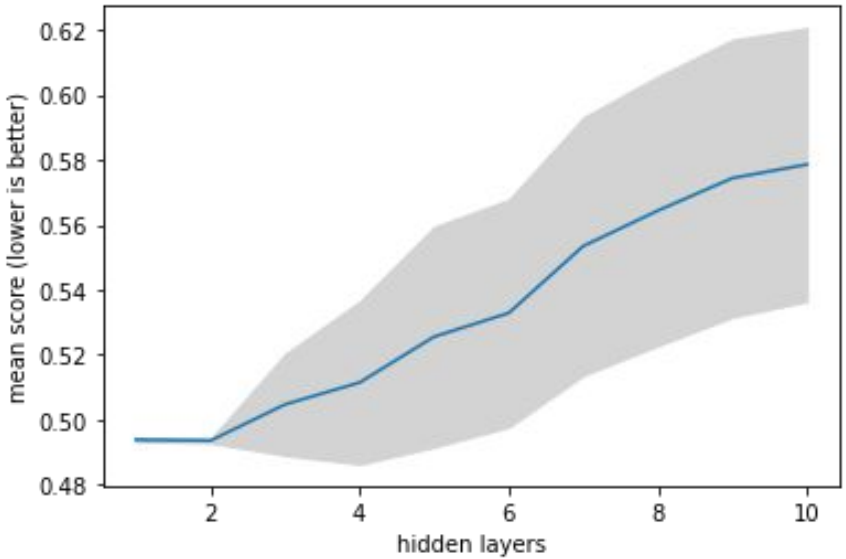
DNN hyperparameter tuning: example analysis performed on a QCD network

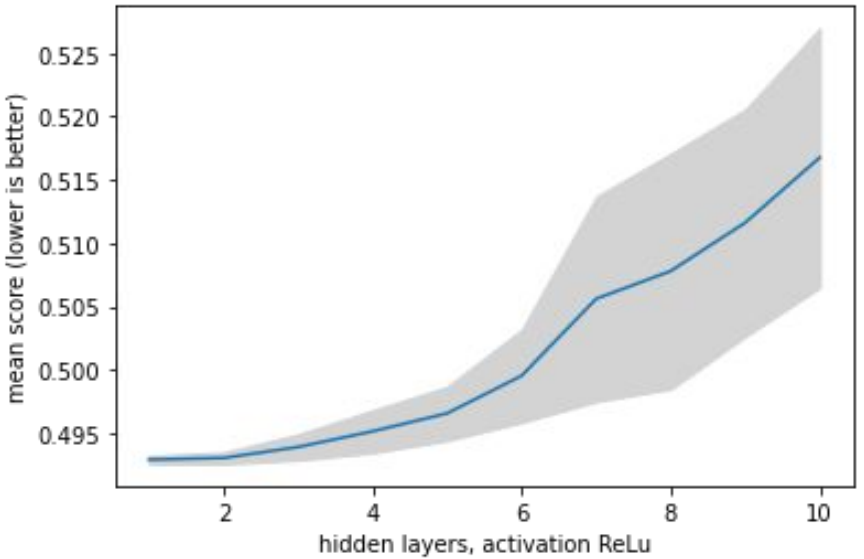


DNN hyperparameter tuning: example analysis performed on a QCD network

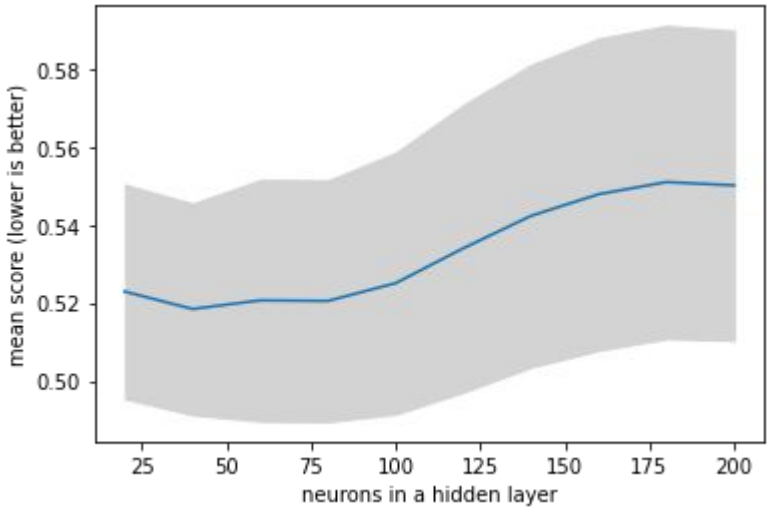
	hidden_layers	hidden_layers_dim	dense_activation	dropout_rate	learning_rate	score
■	1	180	relu	0.1	0.01	0.4924313922723134
■	1	140	relu	0.1	0.01	0.49248066544532776
■	2	200	relu	0.1	0.01	0.4924970865249634
■	1	200	relu	0.2	0.01	0.49249858657519024
■	1	160	relu	0.1	0.01	0.49251070618629456
■	2	180	relu	0.1	0.01	0.49251266320546466
■	1	200	relu	0.1	0.01	0.49251775940259296
■	1	80	relu	0.1	0.01	0.4925244947274526
■	1	180	relu	0.2	0.01	0.49253223339716595
■	2	160	relu	0.1	0.01	0.49253275990486145
■	2	200	relu	0.2	0.01	0.49254225691159564
■	1	100	relu	0.1	0.01	0.4925439755121867
■	1	120	relu	0.1	0.01	0.4925444225470225
■	3	120	relu	0.1	0.01	0.49255017439524335
■	2	180	relu	0.2	0.01	0.49255072077115375
■	3	200	relu	0.1	0.01	0.49256135026613873
■	2	140	relu	0.1	0.01	0.492562731107076
■	2	160	relu	0.2	0.01	0.4925697445869446
■	2	100	relu	0.1	0.01	0.49257949988047284
■	1	160	relu	0.2	0.01	0.4925930897394816

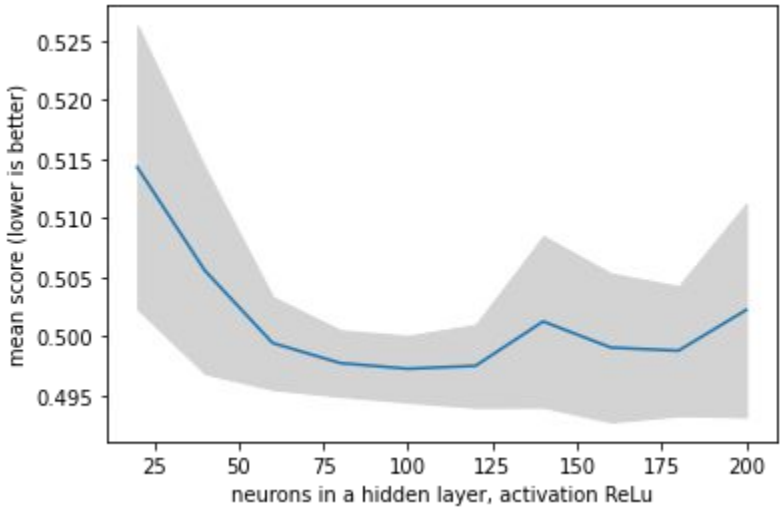
DNN hyperparameter tuning: example analysis performed on a QCD network





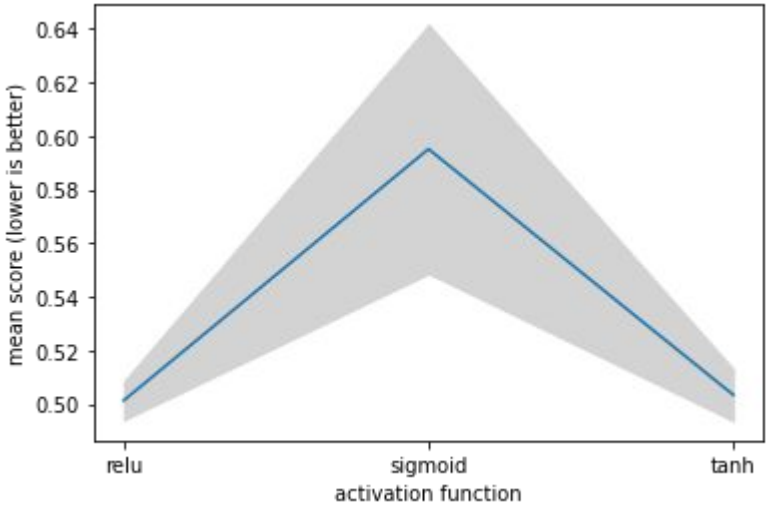
DNN hyperparameter tuning: example analysis performed on a QCD network



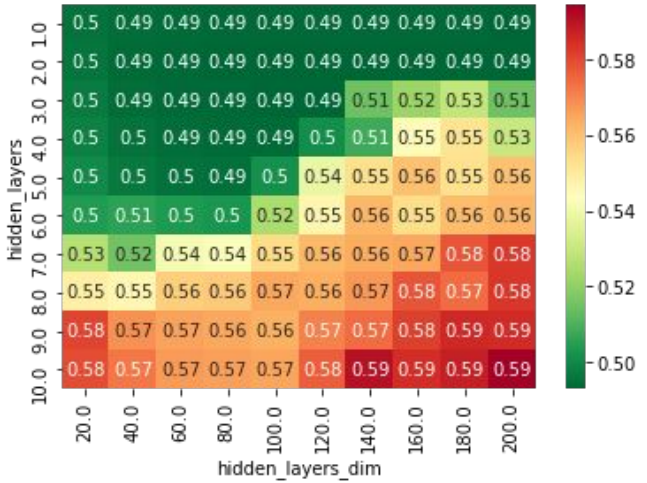


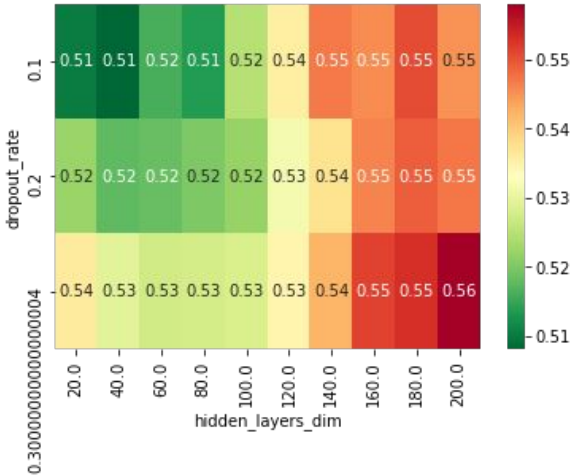
DNN hyperparameter tuning: example analysis performed on a QCD network

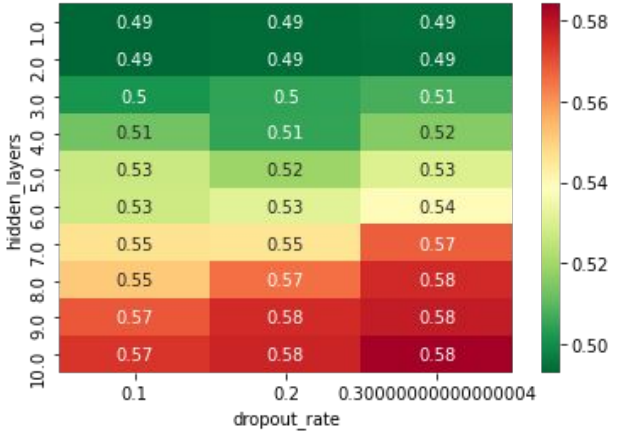
We have used three activation functions in hidden layers and found out that **ReLu** performs more consistently and overall more accurately.



DNN hyperparameter tuning: example analysis performed on a QCD network

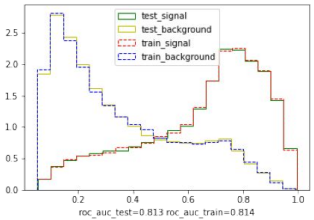


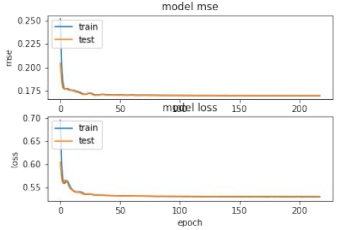


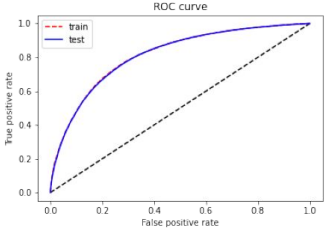


DNN hyperparameter tuning: example analysis performed on a QCD network

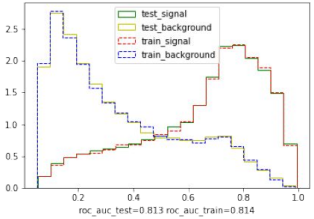
UL17_dr08_qcd_tchan1_500_relu_0.15_0.01

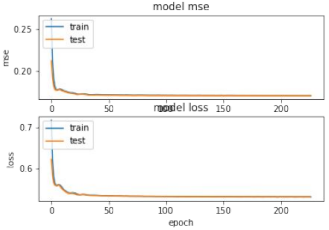


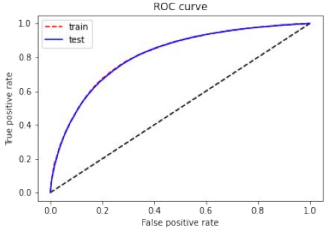




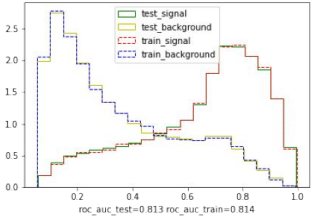
UL17_dr08_qcd_tchan1_400_relu_0.2_0.01

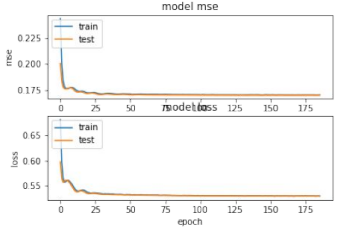


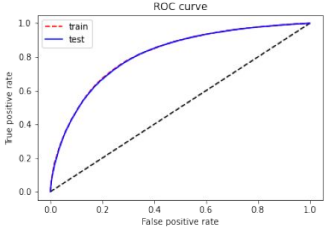




UL17_dr08_qcd_tchan1_450_relu_0.3_0.01







AutoML methods: overview

- AutoML is the end-to-end process of applying machine learning in an **automatic** way.
- The full autoML pipeline usually consists of:
  - data pre-processing,
  - feature engineering,
  - feature extraction,
  - feature selection,
  - model training,
  - algorithm selection,
  - hyperparameter optimization
- Needs a lot of computational resources and relatively little knowledge of machine learning to deploy, so mainly is used in industries and often is provided as a paid service.

AutoML methods: overview

Most of AutoML methods involve either working with images or classifying tabular data. High energy physics events share resemblance with **tabular data**, so I will be focusing on frameworks that are designed to handle this specific data type. There are a lot of open source methods for classifying tabular data (Google's *TabNet*, for example), but many of them do not support event **weights**, which are crucial for high energy physics.

The ones we tried that do:

- AutoKeras
- mljar-supervised

AutoML methods: mljar-supervised

Usage is super simple:

```
automl = AutoML(mode='Optuna', ml_task='binary_classification',
results_path=path, total_time_limit=2*24*3600, explain_level=2)

automl.fit(X=features_train, y=labels_train, sample_weight=weights_train)
```

To get the predictions user should run `automl.predict(X)` just like a normal ML model.

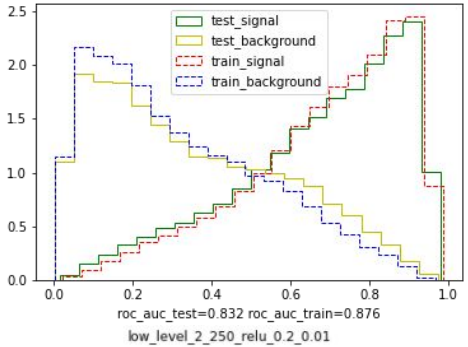
A very competent model is trained in a moderate amount of time.

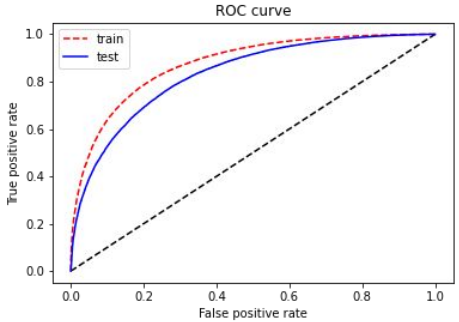
There are 3 modes of performance, all ranging in terms of performance and required computational time.

AutoML methods: mljar-supervised

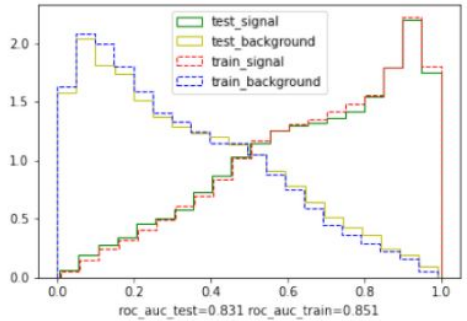
Comparison between base AutoML model obtained in dozens of minutes (top) and a fine-tuned DNN (bottom):

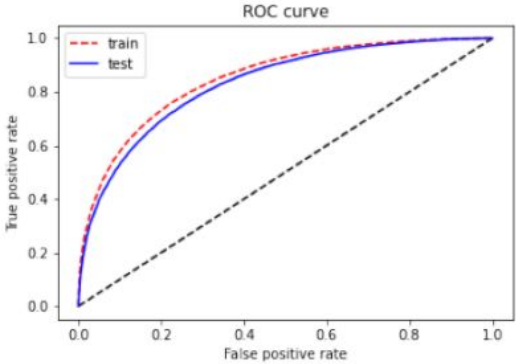
AutoML:





DNN:

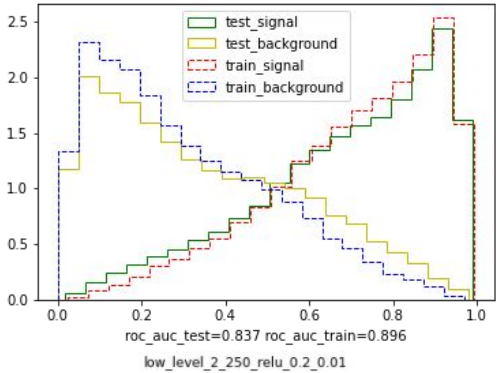


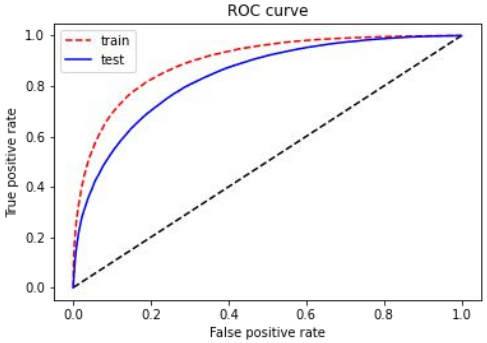


AutoML methods: mljar-supervised

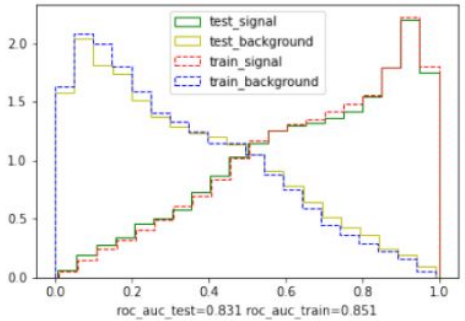
Comparison between tuned AutoML model obtained in dozens of minutes (top) and a fine-tuned DNN (bottom):

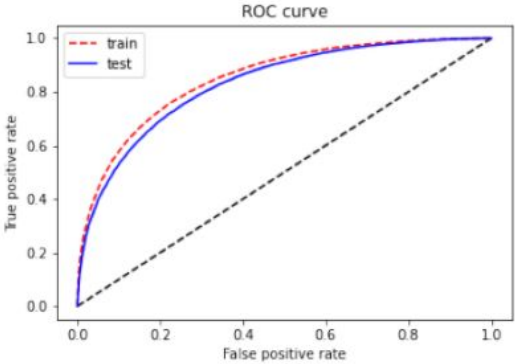
AutoML:





DNN:





AutoML methods: mljar-supervised

Pros:

- Works with minimal code
- Several modes to choose from
- Results are comparable to a tuned DNN

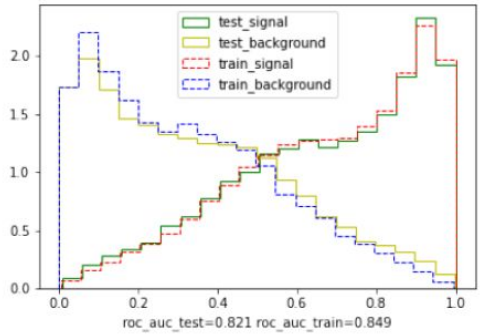
Cons:

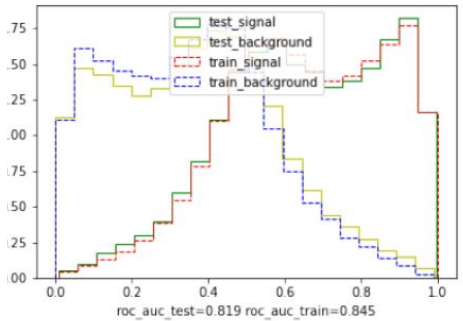
- With the same test accuracy overfits more
- For our task train and test performance of a classifier needs to be very close

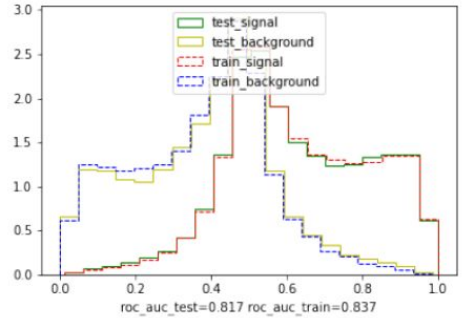
Boosting on errors

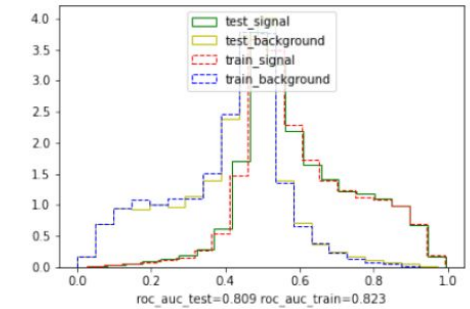
The hypothesis: increase the weights of mis-labeled events so the next model will put more emphasis on those event and maybe classify them better.

The results: with each weight increase the model performed worse.







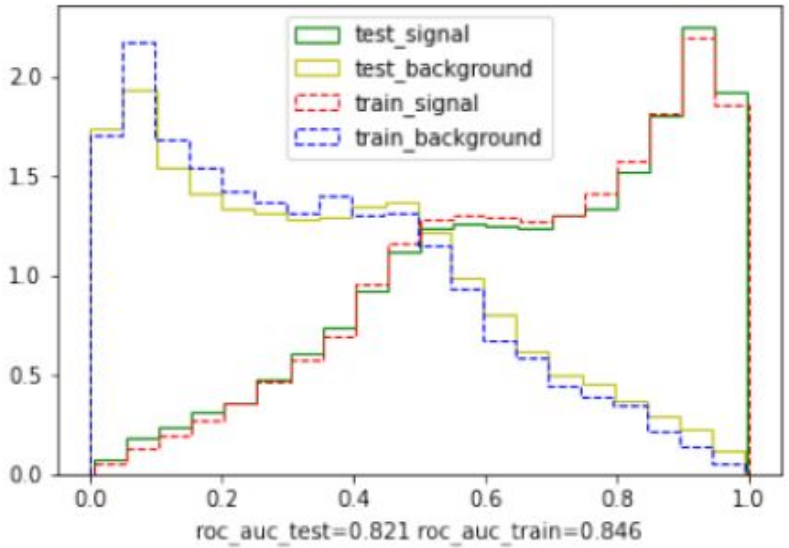


The weights are updated after each training.

L1, L2 regularization study

The problem: certain features work extremely well for first-order classification, so the network assigns them really high weights from the start, leaving out other features that might be useful.

The hypothesis: use l-regularization to limit the weights so certain features will not overshadow others as much.



L1, L2 regularization study

- Under the hood L1 and L2 regularizations work in a similar way:

  L1: `loss = l1 * reduce_sum(abs(x))`

  L2: `loss = l2 * reduce_sum(square(x))`

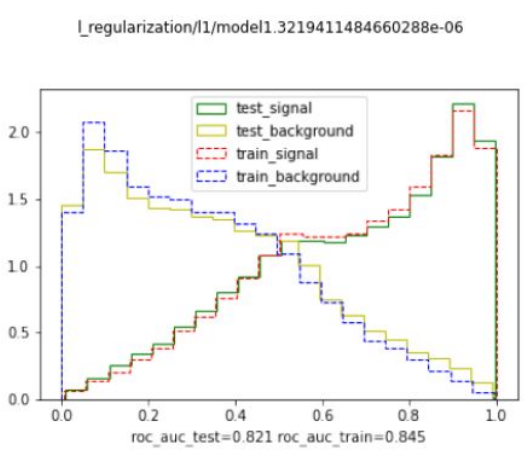
  Where `reduce_sum` computes the sum of elements across dimensions of a tensor and l1, l2 are regularization factors.

- We will conduct the study as follows: while leaving the rest hyperparameters of the model the same, we will try to tune the regularization factors.

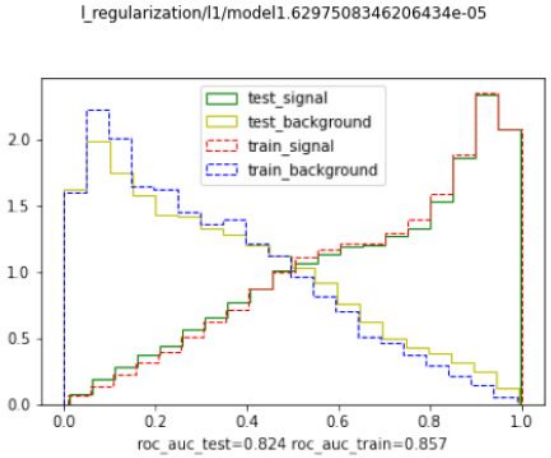
  Tensorflow supports using regularization of biases and activation functions as well, but we will limit this study to weight regularization.

L1, L2 regularization study

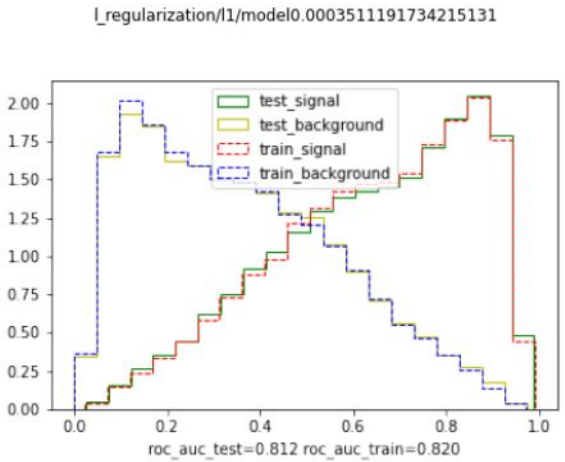
Finding the “sweet spot”: the regularization constant should be not too small and not too high to have the desired effect. The example of L1 regularization:



Too small



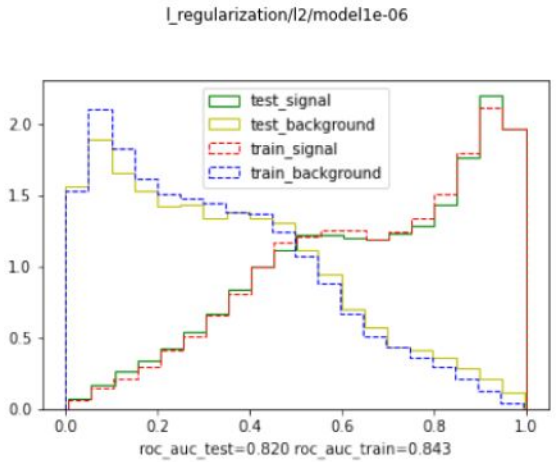
Good



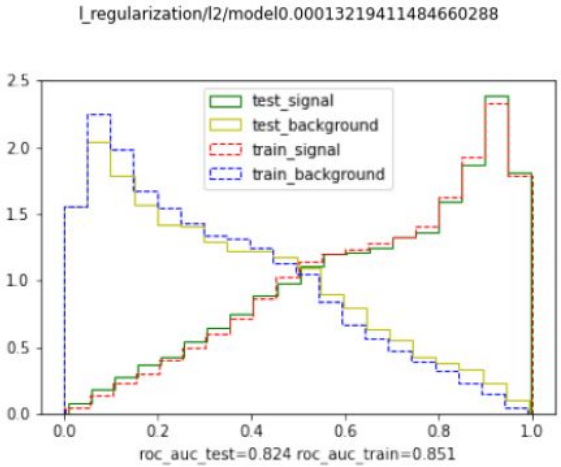
Too large

L1, L2 regularization study

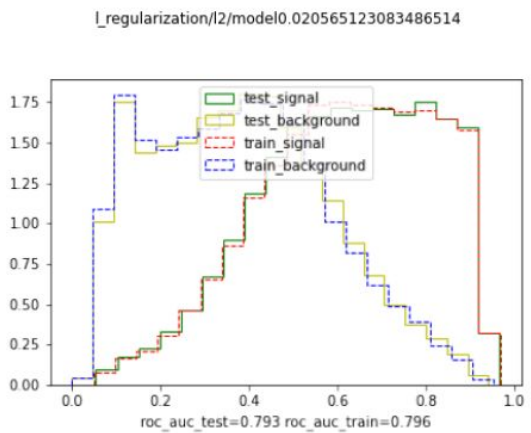
Finding the “sweet spot”: the regularization constant should be not too small and not too high to have the desired effect. The example of L2 regularization:



Too small



Good



Too large

Conclusion

- **DNN tuning:** effective yet resource-intensive method, computation time can be somewhat reduced with different optimization techniques or smaller hyperparameter space
- **AutoML** based on *mljar-supervised*: good baseline method, easy to implement, but overfitting may not be desirable
- **Boosting on errors:** at its current state not advisable to use with DNNs
- **L1, L2 regularization:** is widely used to prevent overfitting, can work well if the regularization constant is chosen correctly

Thanks for your attention!

Backup: code snippet for boosting on errors

```
boost_threshold = 0.4

boost_coefficient = 1.3

predict_train = model.predict(features_train)

weights_train_boosted = weights_train


for i in range(len(labels_train)):

    if ((labels_train[i] == 1) and (predict_train[i] < 1-boost_threshold)) or ((labels_train[i] == 0) and
(predict_train[i] > 0+boost_threshold)):

        weights_train_boosted[i] = weights_train_boosted[i]*boost_coefficient
```