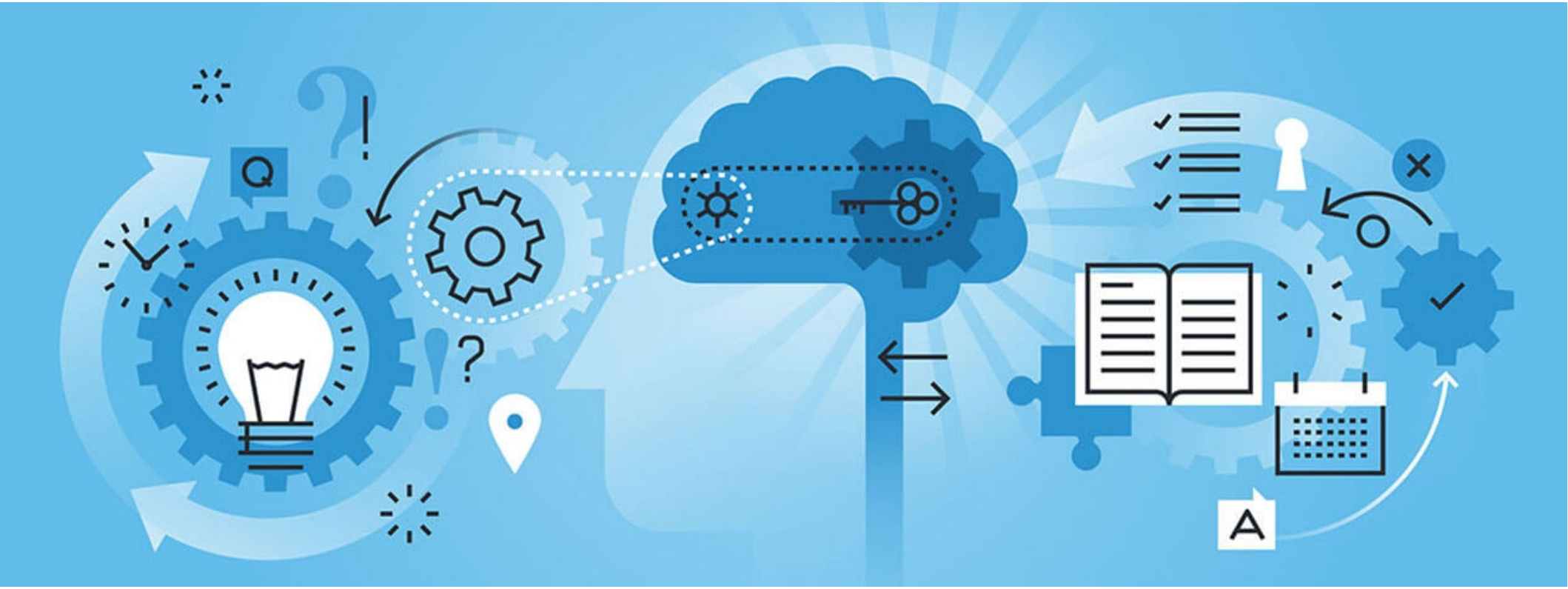
Modeling with Keras



Open Discussion Machine Learning

Christian Contreras, PhD

Overview

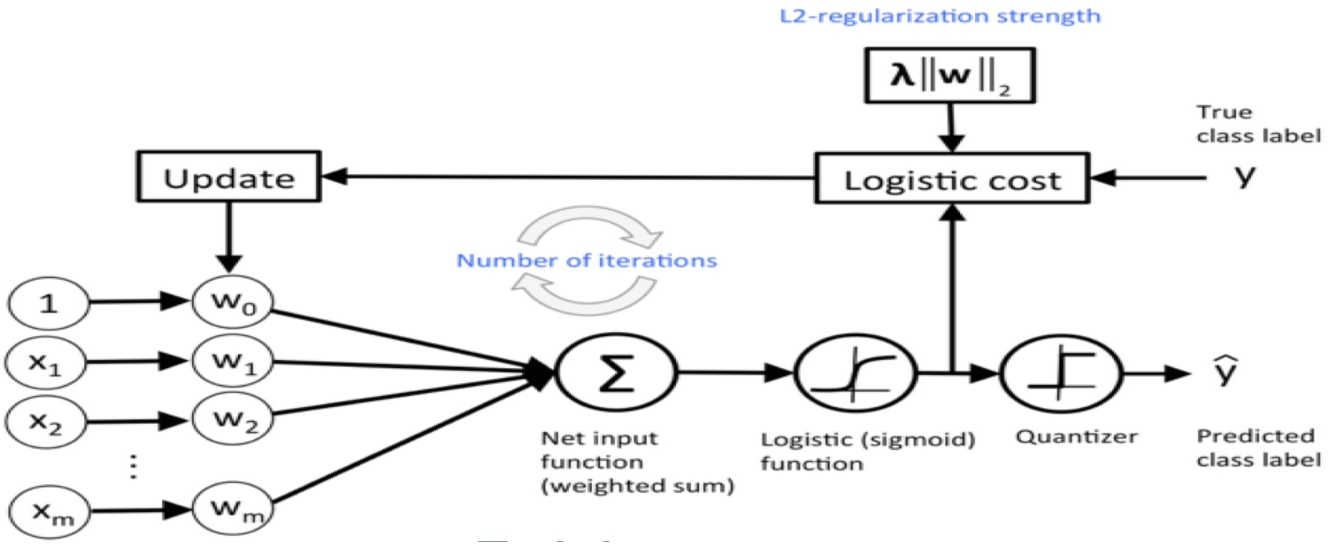
- As practitioners in deep networks, we often want to understand areas of prototyping and modeling. While there are many python libraries for deep learning, **Keras** stands out for it's simplicity in modeling.

- Keras is a high-level neural network API, written in python capable of running on top of either **Theano** or **Tensorflow**. Developed with a focus on enabling fast experimentation.
    - Supports both convolutional and recurrent networks as well as a combination of the two.
    - Runs seamlessly on **GPU** and **GPU** cores.

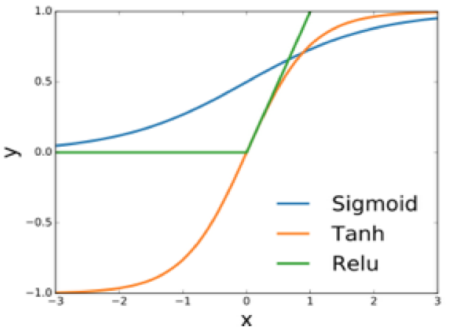
- In this talk, we explore the basic elements of DL using Keras modeling, general diagnostics, and model optimization

Anatomy of deep learning network

Network architecture is the scheme for combining various neural network layers into a deep learning machine.



Non-linear activation functions are the key to DNN



Training on data to build model involves

- Measuring the difference between NN output prediction and the true class label according to a cost function (e.g. Log-loss)
- Minimizing the lost function w.r.t. the neural network weight

Keras Basics

- We shall review the basic layers in Keras with the goal of understanding the modeling aspects only.
- No deep dive, we need to pickup just enough to understand the modeling.

Here is the **`Sequential model`**:

```
from keras.models import Sequential

model = Sequential()
```

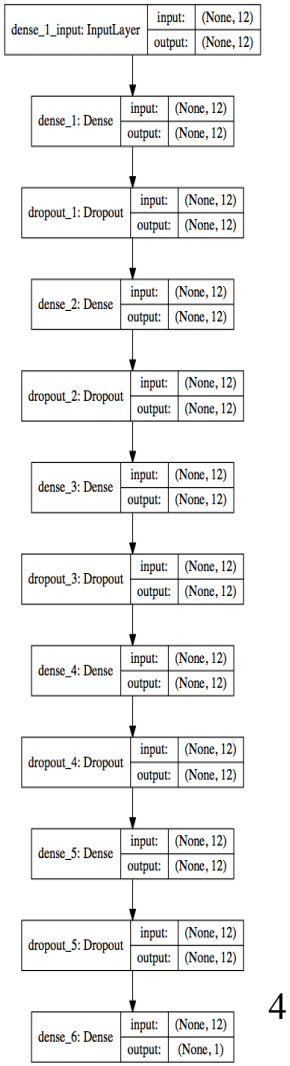
Stacking layers is as easy as .add():

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Configure its learning process with .compile():

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Objective function (loss function) is one of two parameters



Ready to train & evaluate model performance

We can now iterate on your **training data** in batches:

```
# x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Evaluate your performance in one line:

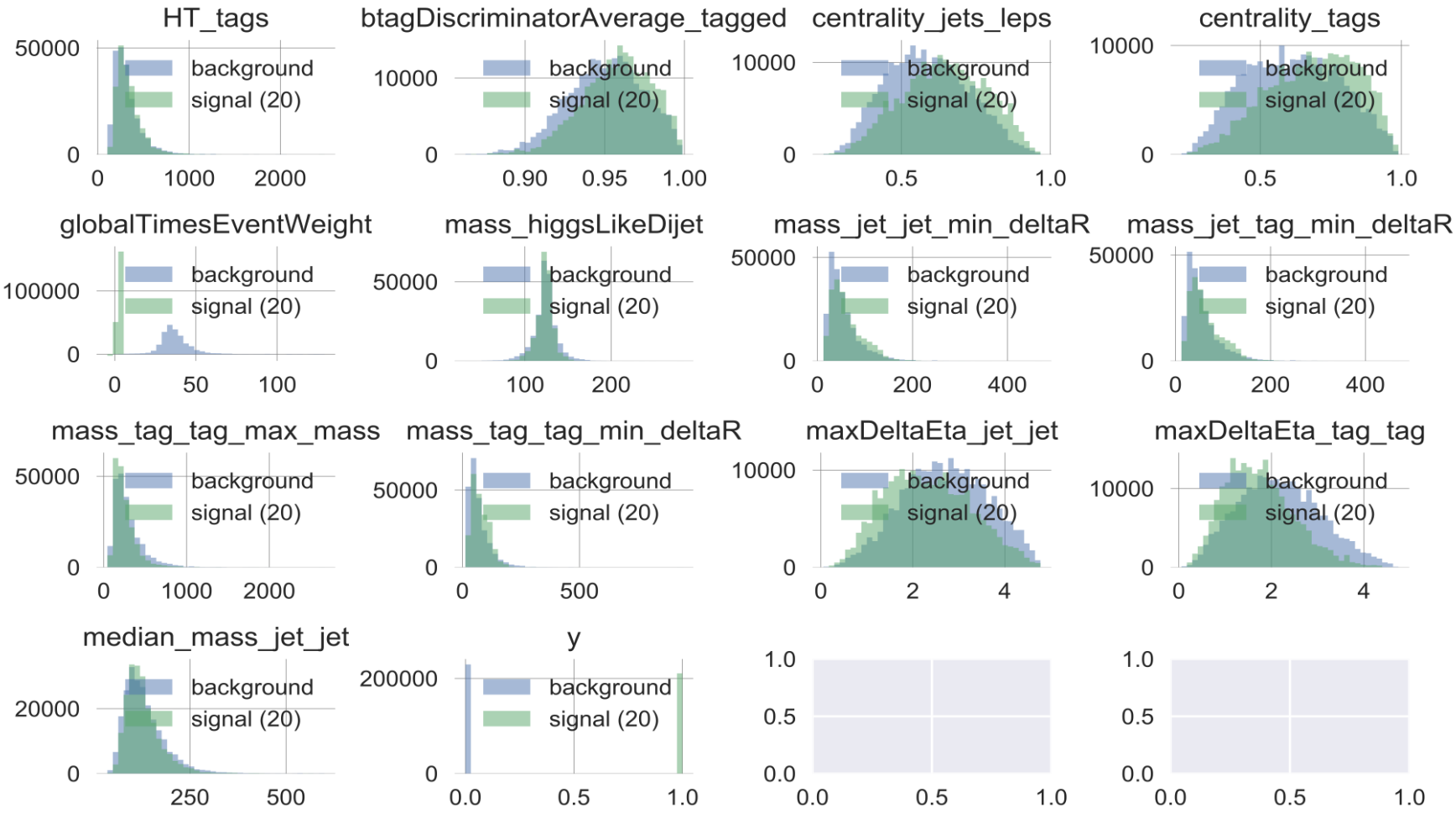
```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Or **generate predictions** on new data:

```
classes = model.predict(x_test, batch_size=128)
```

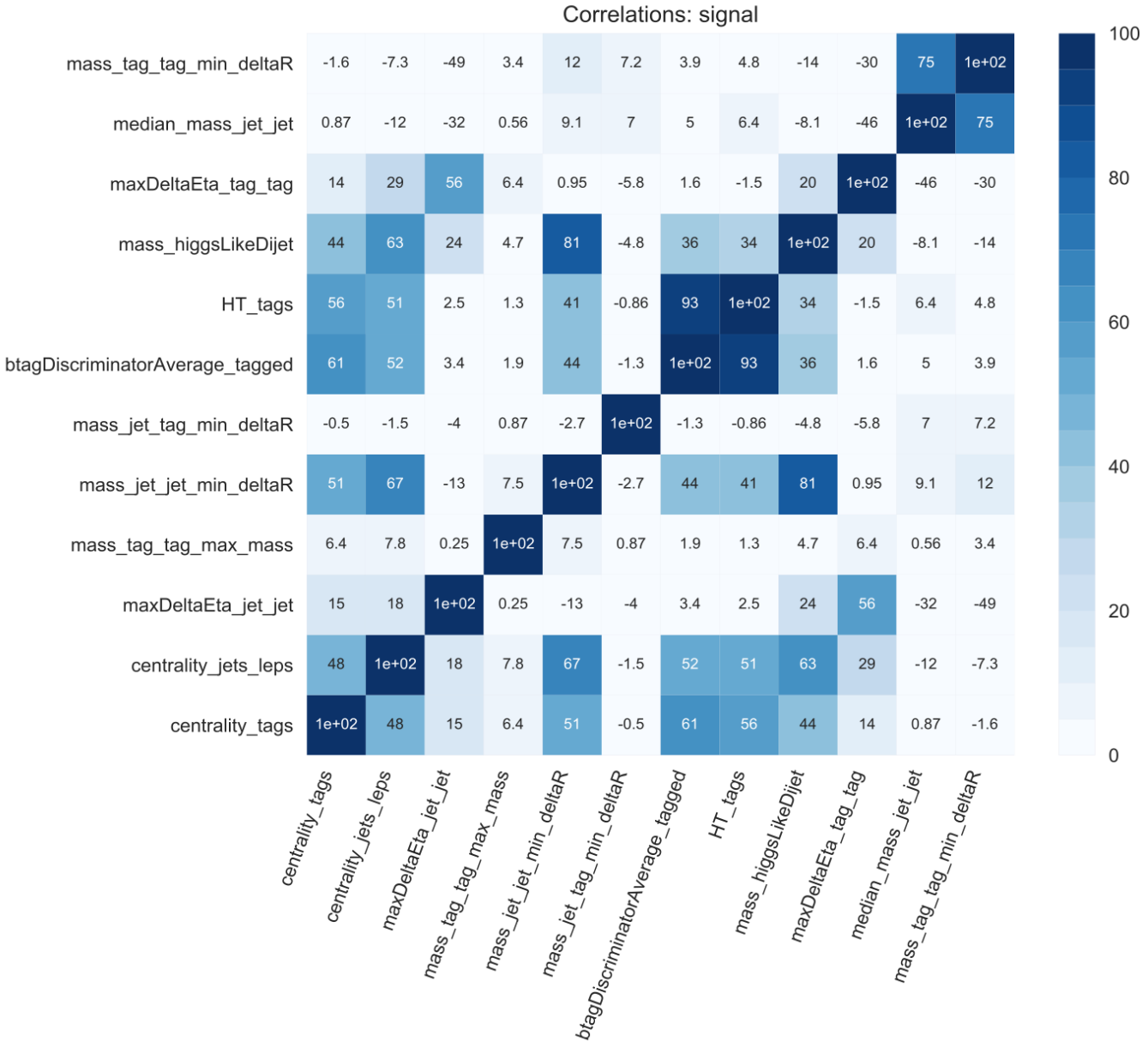
Preprocessing step

Check variable (features) distribution between signal and background



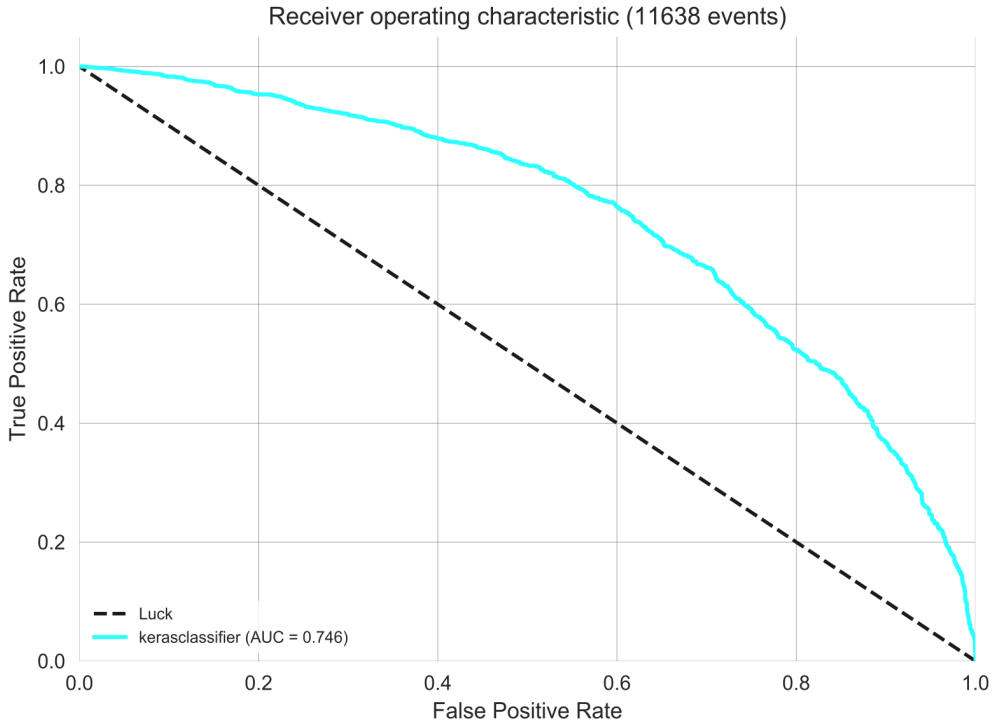
Preprocessing step (cont.)

Correlation among the variables (features)



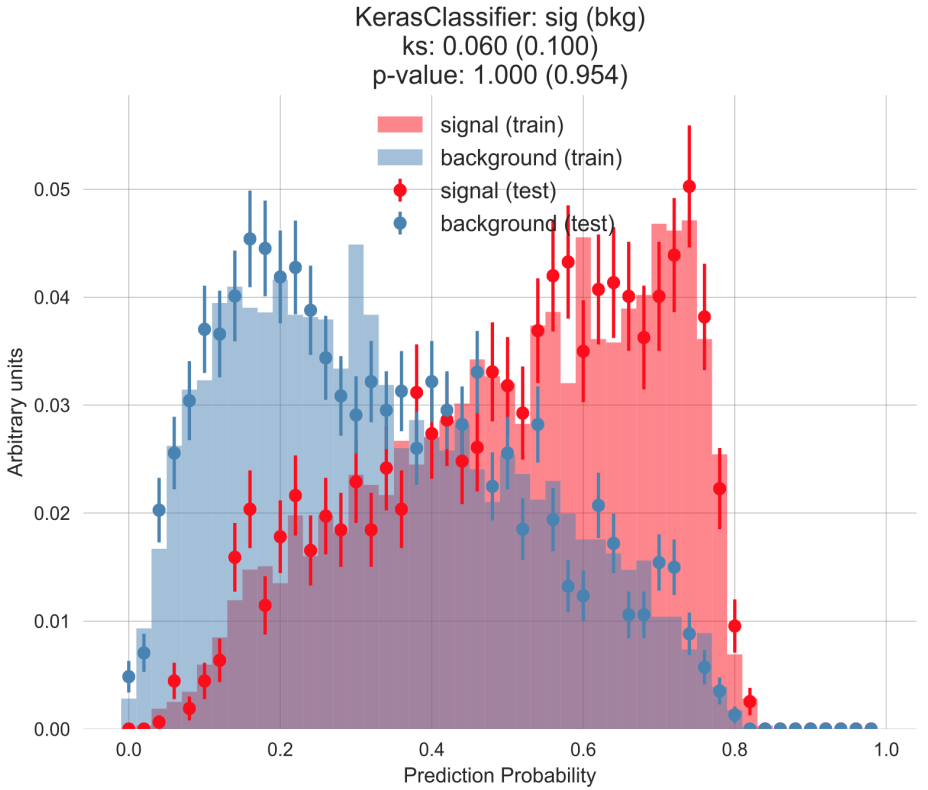
Diagnostic tools

ROC curve



- The “steepness” of **ROC curves** is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.
- **AUC** is a common evaluation metric

Overtraining

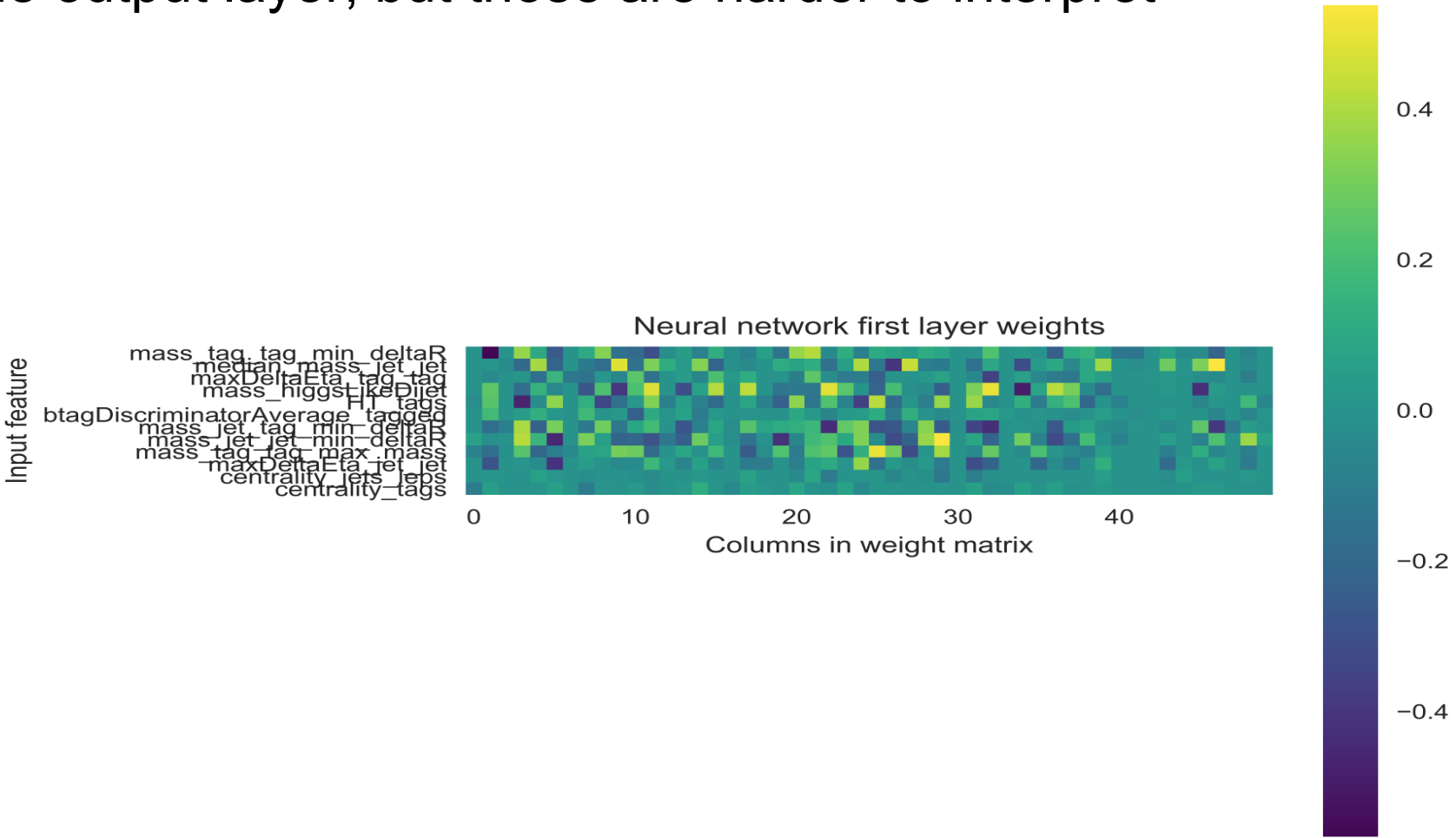


- Tests whether two- samples are drawn from the same distribution
- If the **KS-test statistic** is small or the **p-value** is high, then we cannot reject the hypothesis that the distributions of the two samples are the same.

Heat map (measure of importance)

- Heat map of the first layer weights in a neural network learned on the dataset.

- We could also visualize the weights connecting the hidden layer to the output layer, but those are harder to interpret



Neural network hyper-parameters

Network architecture

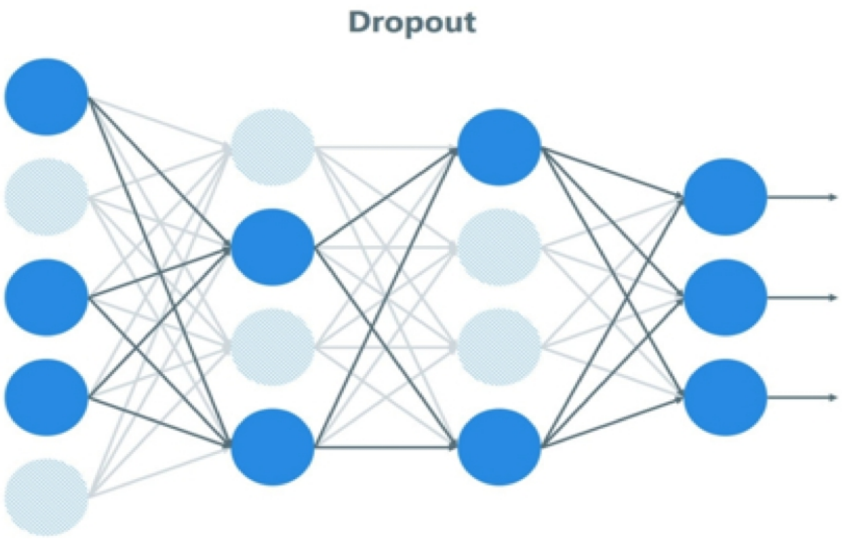
- Number of hidden layers
- Number of neurons per layer
- Type of activation functior
- Weight initialization

Regularization parameters

- Weight decay strength
- Dropout rate

Training parameters

- Learning rate
- Batch size
- Number of epochs

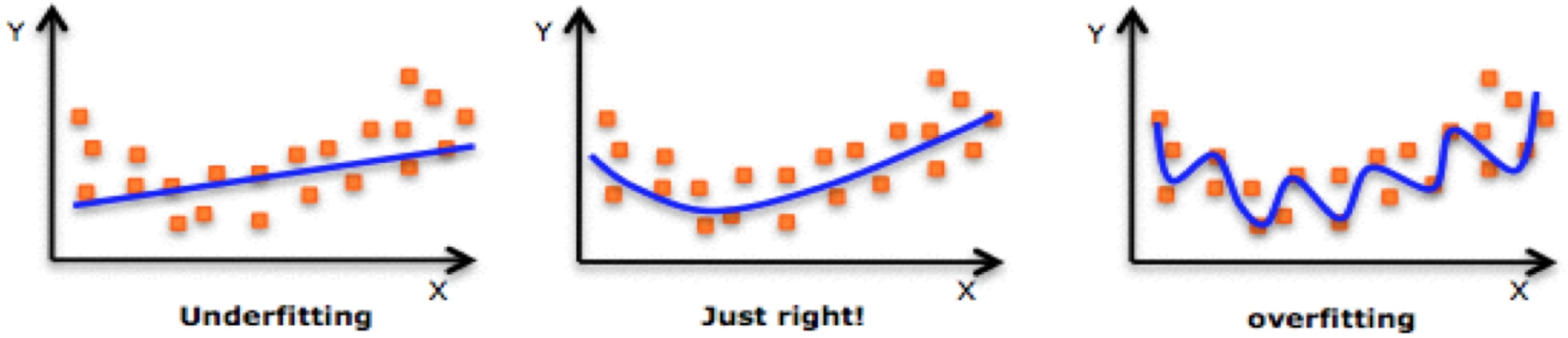


Why hyper-parameter tuning?

- Machine learning models involve careful tuning of learning parameters & algorithm hyper-parameters

- This tuning is often a “black art” requiring
    - Experience, rules of thumb, or sometimes brute-force search

- Tuning prevent **under- or over-fitting** a model
    - The purpose is to generalize well to new data



Search for good hyper-parameters?

- **Define an objective function**
  - Most often, we care about generalization performance
- **How do people currently search?** Black magic?
  - Grid search
  - Random search
  - Grad student descent
- **Tedious!**
  - Requires at best many training cycles
  - More sophisticated optimization exist!!

Why is tuning hard?

- Hard since it involve model training as a sub-process & not directly
- Difficult with DNN they tend to have many hyper-parameters to tune
- Leads to the appeal to automated approaches that can optimize the performance

Proper tuning of hyper-parameters

- Assume the accuracy of prediction on the test set, using default classifier setting, is 0.87
- Can we do better?
  - Yes, we can. How? To put it simply, we need another model that will give higher accuracy on the test set.
  - How can we choose the best model for a given type of classifier?
    - The answer is called **hyper-parameter optimization**. Any parameter that changes the properties of the model directly, or changes the training process.
- Can we just try different network nodes, fit the model, check the accuracy on the **test set**, then draw a conclusion on which model is better, right? Then take another number of nodes, then repeat the steps, compare to previous result, etc.
- No. This way at some point we can get 100% accuracy for the test set (**info leakage**), and we'd just be over-training the test set.

The problem

- The choice of the model should be based on training data only.
- How can we choose the best model in this case, if doing multiple fitting of different models on the training dataset also leads us to over-training?
- Answer, is to use **cross-validation**
  - Idea is to split training data, into training set & validation set
  - Fit the model and test the accuracy on the validation set
  - Then, do another random split, repeating training & getting accuracy on validation set
    - Using the **cross-validation accuracy** metric, to choose the best model from the class of models.
    - After, use best hyper-parameter values, and refit the model on the **full training dataset**.
  - Lastly, use this one model to predict on the test data set.

Where to start, experience or brute-force?

- Let's tune the model using 2 parameters: number of nodes in the hidden layer and learning rate of the optimizer used in during network training

```
nodes = [32, 64, 128, 256, 512] # number of nodes in the hidden layer
lrs = [0.001, 0.002, 0.003] # learning rate, default = 0.001
epochs = 15
batch_size = 64
```

- Keras-based neural network model

```
def build_model(nodes=10, lr=0.001):
    model = Sequential()
    model.add(Dense(nodes, kernel_initializer='uniform', input_dim=784))
    model.add(Activation('relu'))
    model.add(Dense(10))
    model.add(Activation('softmax'))

    opt = optimizers.RMSprop(lr=lr)
    model.compile(loss='categorical_crossentropy',
                  optimizer=opt, metrics=['accuracy'])

    return(model)

model = KerasClassifier(build_fn=build_model, epochs=epochs,
                        batch_size=batch_size, verbose=0)
```

Scikit-learn grid-search optimizer

- Define parameter grid space

```
param_grid = dict(nodes=nodes, lr=lrs)
```

```
{'lr': [0.001, 0.002, 0.003], 'nodes': [32, 64, 128, 256, 512]}
```

- Grid search estimator

```
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=3,
                    n_jobs=1, refit=True, verbose=2)
grid_result = grid.fit(X, Y)
```

- Training output

```
Fitting 3 folds for each of 15 candidates, totalling 45 fits
[CV] lr=0.001, nodes=32 ..............................................
[CV] ............................... lr=0.001, nodes=32, total=  10.9s
[CV] lr=0.001, nodes=32 ..............................................
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   11.4s remaining:    0.0s
[CV] ............................... lr=0.001, nodes=32, total=  11.2s
[CV] lr=0.001, nodes=32 ..............................................
...
[CV] lr=0.003, nodes=512 .............................................
[CV] ............................... lr=0.003, nodes=512, total=  41.2s
[Parallel(n_jobs=1)]: Done  45 out of  45 | elapsed: 16.9min finished
```

Alternative approach

Bayesian optimization

Uses a distribution over functions to build a surrogate model of the unknown function being optimized and then apply some active learning strategy to select the query points that provides most potential interest or improvement

Optimization steps

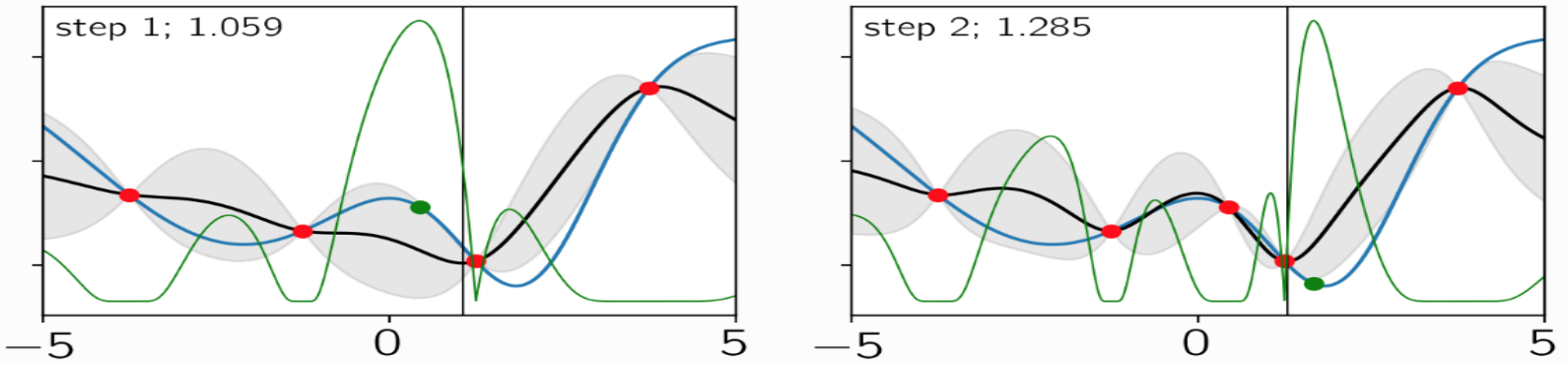
- **Build a probabilist model** for the objective
- **Compute the posterior** predictive distribution
  - Integrate out all the possible true functions
  - Make use of **Gaussian process** regression
- **Optimize a cheap proxy function** instead
  - The model is much cheaper than the true objective

Source: bayesopt

Main insight

Make the proxy function exploit uncertainty to balance **exploration** agains **exploitation**.

- **Exploration**: seeks places with high variance
- **Exploitation**: seeks places with low mean



Bayesian optimization (cont.)

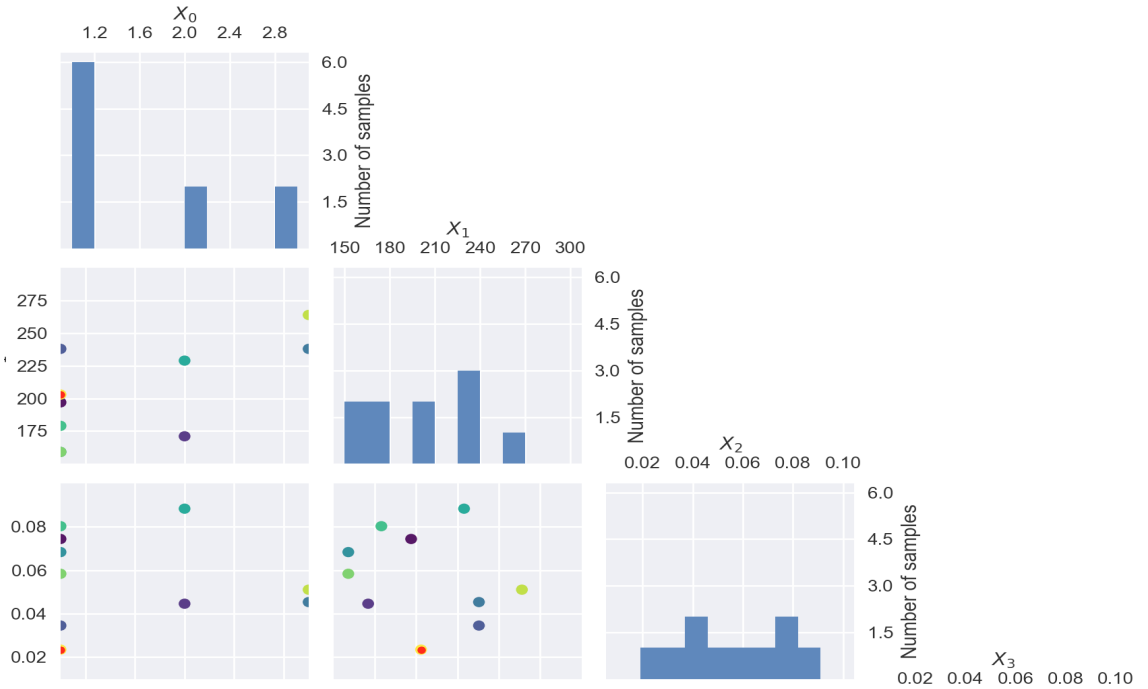
Model optimization

```
# ---- Create model for use in scikit-learn
pipe_classifiers = {
    'kerasclassifier':  make_pipeline(scaler,  KerasClassifier(build_fn=dnn.build_fn, batch_size=128, epochs=10, verbose=1))
    }

# ---- Set configuration space
parameters = collections.OrderedDict(
    [
        ('kerasclassifier__nlayers', (1, 3)),
        ('kerasclassifier__nneurons', (150, 300)),
        ('kerasclassifier__l2_norm', (0.01, 0.1)),
        ('kerasclassifier__dropout_rate', (0.01, 0.1))
    ])

# ---- Instantiate objective function to optimize
estimator = pipe_classifiers['kerasclassifier']
objective = SkOptObjective(estimator, data_X, data_y)
objective.paramKeys(parameters.keys())

# ---- Bayesian optimization based on Gaussian process regression search (controlling the exploration-exploitation trade-off)
clf_gp_ei = gp_minimize(func=objective,                    # the function to minimize
                        dimensions=parameters.values(),    # the bounds on each dimension of the optimization space
                        acq_func="EI",                     # the acquisition function ("EI", "LCB", "PI")
                        n_calls=10,                        # the number of evaluations of the objective function
                        random_state=0,                    # the random seed
                        n_jobs=1)                          # the number of threads to use
print("Best score=%.4f (EI)" % clf_gp_ei.fun)

print("""Expected Improvement (EI) best parameters:
- nlayers= %s
- nneurons= %s
- dropout_rate= %s
- l2_norm= %s""" % (str(clf_gp_ei.x[0]), str(clf_gp_ei.x[1]), str(clf_gp_ei.x[2]), str(clf_gp_ei.x[3])))
```

Diagnostic checks

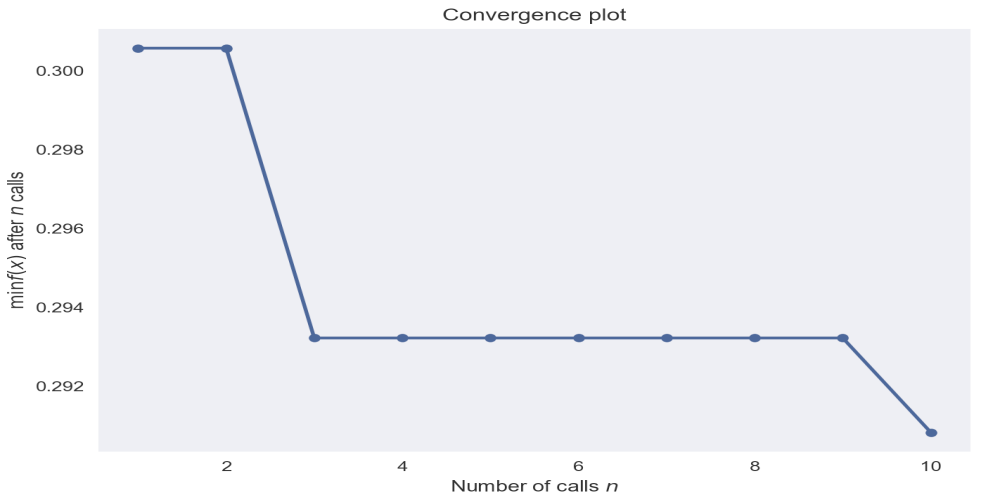
Evaluation

```
# ---- Evalution
plot_evaluations(clf_gp_ei, bins=10)
plt.show()
```



Convergence

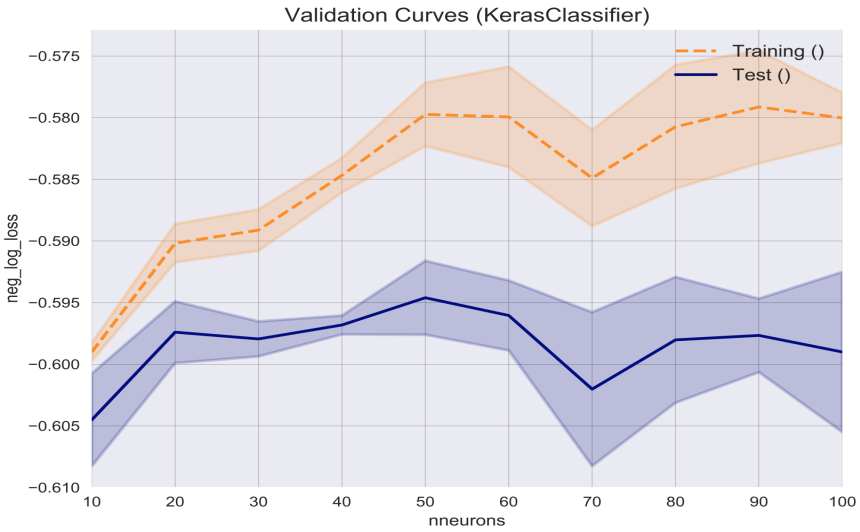
```
# ---- Convergence
plot_convergence(clf_gp_ei);
plt.show()
```

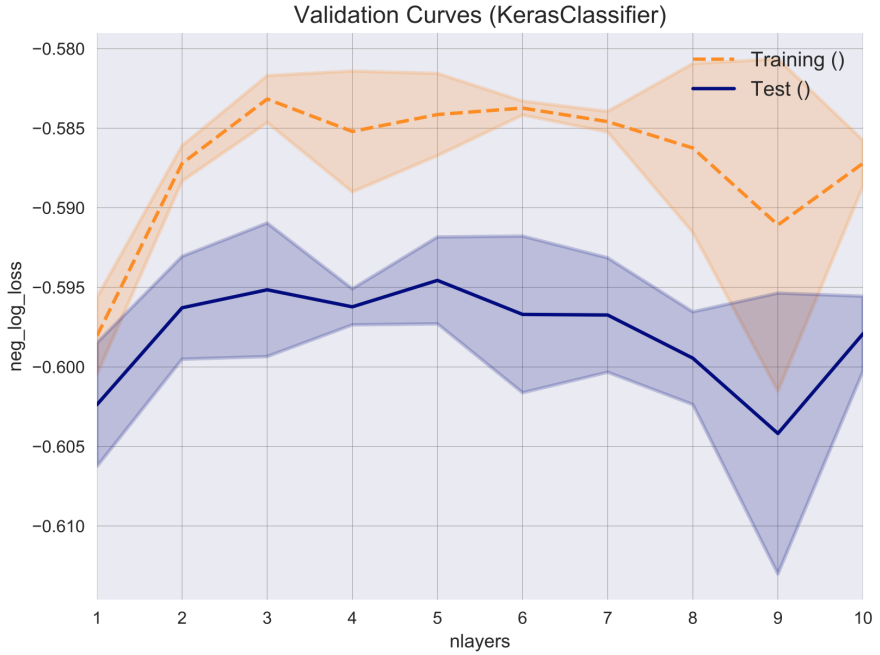


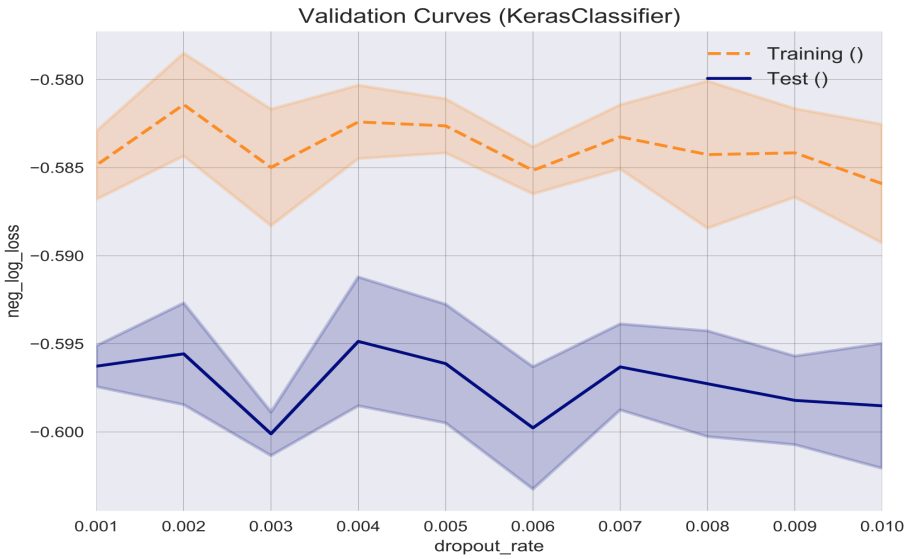
Validation curve

- Plot the inf luence of a single hyper-parameter (HP) on the training score& the validation score to f nd out whether the estimator is over-f tting or under-f tting for some HP values.

-If we optimized the HP based on a validation score the validation score is biased and not a good estimate of the generalization any longer.



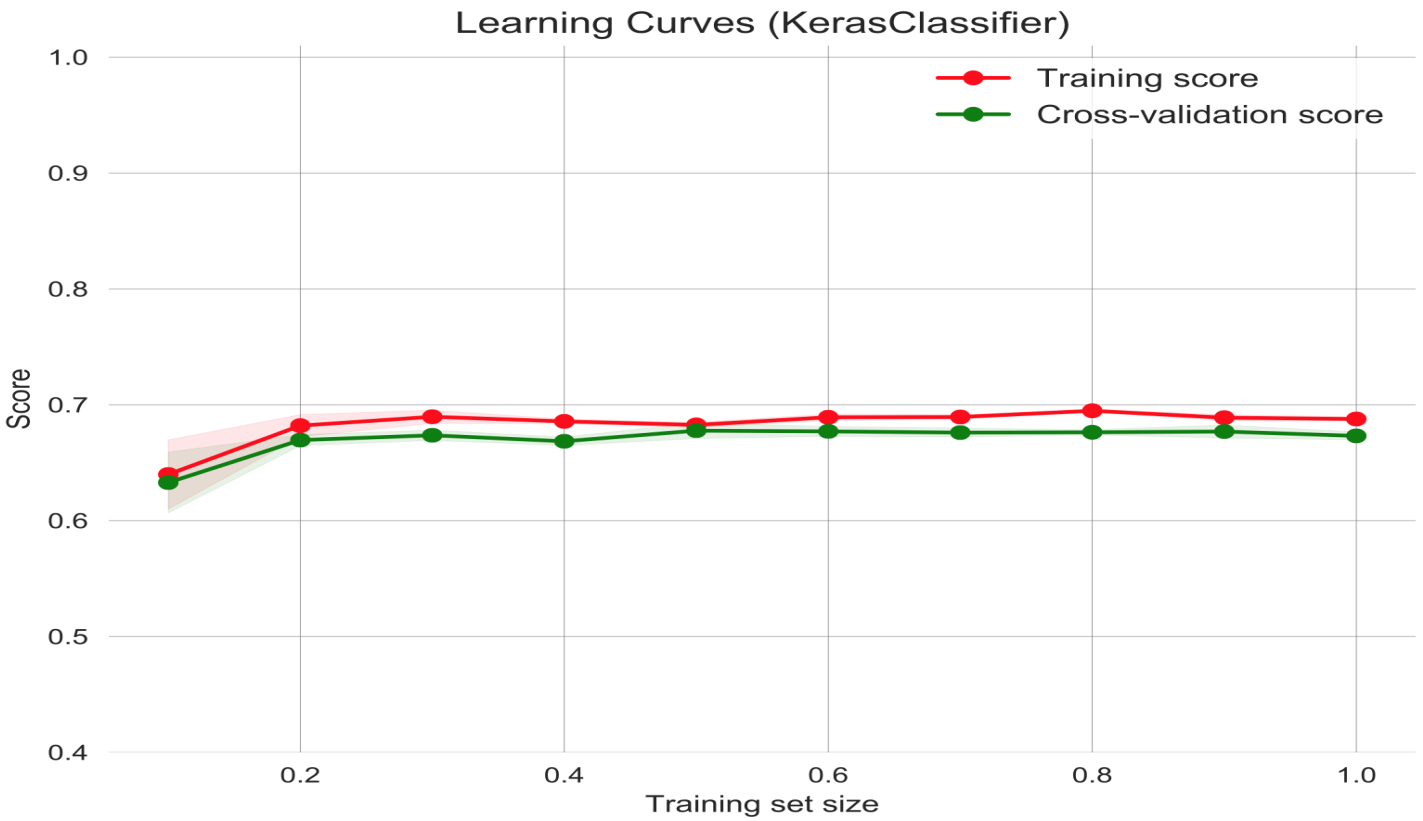




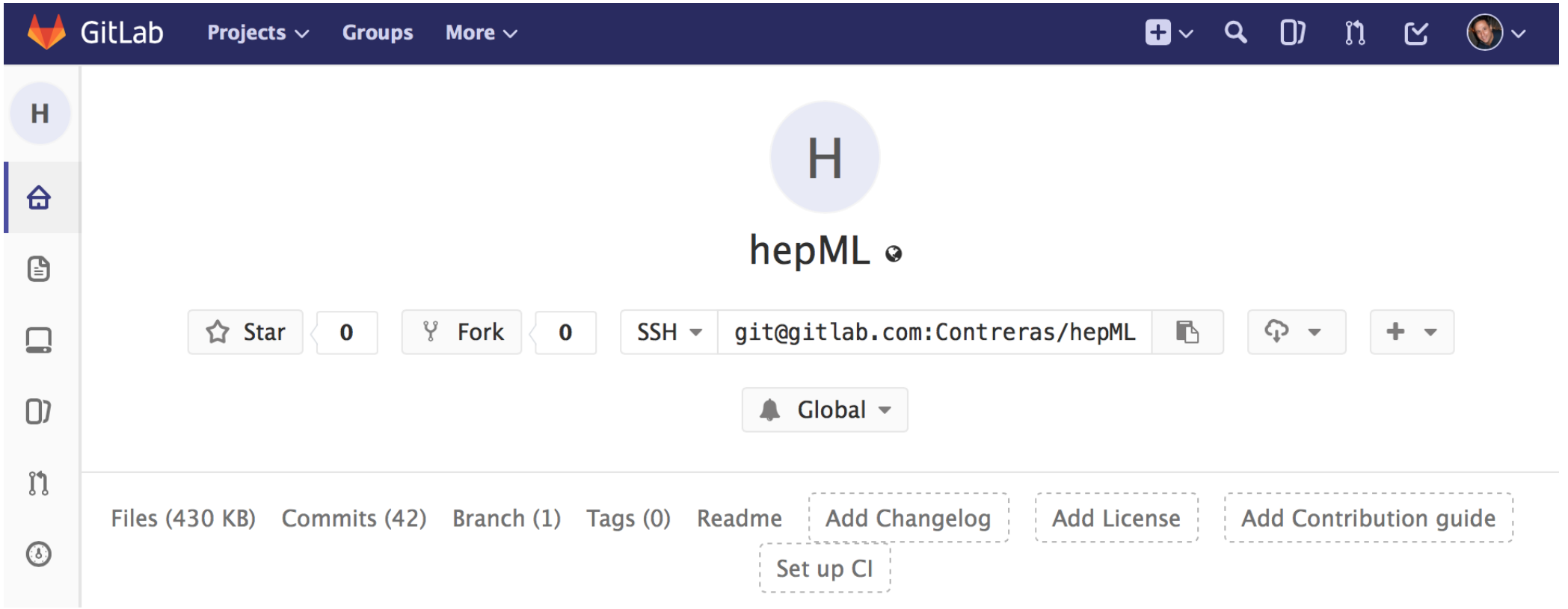
Learning curve

- A learning curve shows the validation and training score of an estimator for varying numbers of training samples.

- Tool to f nd out how much we benef t from adding more training data and whether the estimator suffers more from a variance error or a bias error.



Git repository



Git clone git@gitlab.com:Contreras/hepML.git

Summary

Discussed today

- The anatomy of a deep neural architecture
- The basics of deep learning with Keras
    - training & model evaluation
- Simple preprocessing steps & diagnostic checks
    - Correlation matrix, ROC curve, Overfitting, and heat map of network weights
- Elaborated on the more advance topic of model tuning:
    - Leverage validation & learning curve
    - Tuning model based on Bayesian optimization
        - Looking at evaluation & convergence distributions

Future plans

- Explore the usage of GPU cores for model training on Maxwell-Cluster system
- Other meta-classifiers with Keras:
    - Probability calibration classifier, Majority voting classifier, Stacking classifier

Backup

- Create classifier

```
# ---- Preprocessing using 0-1 scaling by removing the mean and scaling to unit variance
scaler = RobustScaler()

# ---- Set deep neural network architecture: [m input] -> [n neurons] -> [1 output]
dnn = DeepModel(input_dim=input_dim)

# ---- Create model for use in scikit-learn
pipe_classifiers = {
    'kerasclassifier':  make_pipeline(scaler,  KerasClassifier(build_fn=dnn.build_fn, batch_size=128, epochs=10, verbose=1))
    }
```