Chapter 10
Case Study III: Tuning of Deep Neural Networks

Thomas Bartz-Beielstein, Sowmya Chandrasekaran, and Frederik Rehbach

Abstract A surrogate model based Hyperparameter Tuning (HPT) approach for Deep Learning (DL) is presented. This chapter demonstrates how the architecture-level parameters (hyperparameters) of Deep Neural Networks (DNNs) that were implemented in `keras/tensorflow` can be optimized. The implementation of the tuning procedure is 100% accessible from R, the software environment for statistical computing. How the software packages (`keras`, `tensorflow`, and `SPOT`) can be combined in a very efficient and effective manner will be exemplified in this chapter. The hyperparameters of a standard DNN are tuned. The performances of the six Machine Learning (ML) methods discussed in this book are compared to the results from the DNN. This study provides valuable insights in the tunability of several methods, which is of great importance for the practitioner.

10.1 Introduction

The DNN hyperparameter study described in this chapter uses the same data and the same HPT process as the ML studies in Chaps. 8 and 9. Section 10.2 describes the data preprocessing. Section 10.3 explains the experimental setup and the configuration of the DL models. The objective function is defined in Sect. 10.4. The hyperparameter tuner, `spot`, is described in Sect. 10.5. Based on this setup, experimental results are analyzed: After discussing tunability based on the HPT progress in Sect. 10.6, default, λ_0 and tuned hyperparameters, $\lambda^\star$ are compared in Sect. 10.6.2.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-981-19-5170-1_10.

T. Bartz-Beielstein (✉) · S. Chandrasekaran · F. Rehbach
Institute for Data Science, Engineering and Analytics, TH Köln, Gummersbach, Germany
e-mail: thomas.bartz-beielstein@th-koeln.de

S. Chandrasekaran
e-mail: sowmya.chandrasekaran@th-koeln.de

F. Rehbach
e-mail: frederik.rehbach@th-koeln.de

© The Author(s) 2023

E. Bartz et al. (eds.), *Hyperparameter Tuning for Machine and Deep Learning with R*,
https://doi.org/10.1007/978-981-19-5170-1_10

Table 10.1 Deep-learning hyperparameter pipeline

Step	Description	Function	Result	Details
1	Get data	`getDataCensus`	dfCensus	Data frame
2.1	Split data into training, validation, and test data	`getGenericTrainValTestData`	Data	Partitioned data
2.2	Spec	`genericDataPrep`	specList	List with the following data
2.3.1	`keras` configuration	`getKerasConf`	kerasConf	Configuration list for `keras`
2.3.2	Model configuration	`getModelConf`	cfg	Model
3	Hyperparameter tuning	`spot`	Result	Result list
4	Evaluate on test data	`evalParamCensus`	Score	Metrics

The DL tuning process is analyzed in Sect. 10.7. Results are validated using severity in Sect. 10.8. A summary in Sect. 10.9 concludes this chapter. The DL hyperparameter tuning pipeline, that was used for the experiments, is summarized in Table 10.1 and illustrated in Fig. 10.1. The first sections in this chapter highlight the most important steps of this pipeline. The program code for performing the experiments is shown in Sect. 10.10.

`keras` is TensorFlow (TF)'s high-level Application Programming Interface (API) designed with a focus on enabling fast experimentation. TF is an open source software library for numerical computations with data flow graphs (Abadi et al. 2016). Mathematical operations are represented as nodes in the graph, and the graph edges represent the multidimensional arrays of data (tensors) (O'Malley et al. 2019). The full TF API can be accessed via the `tensorflow` package from within the R software environment for statistical computing and graphics (R).

The Appendix contains information on how to set up the required PYTHON software environment for performing HPT with `keras`, `SPOT`, and `SPOTMisc`. Source code for performing the experiments will included in the R package `SPOTMisc`. Further information is published on https://www.spotseven.de and with some delay on Comprehensive R Archive Network (CRAN) (https://cran.r-project.org/package=SPOT). This delay is caused by an intensive code check, which is performed by the CRAN team. It guarantees high-quality open source software and is an important feature for providing reliable software that is not just *a flash in the pan*.

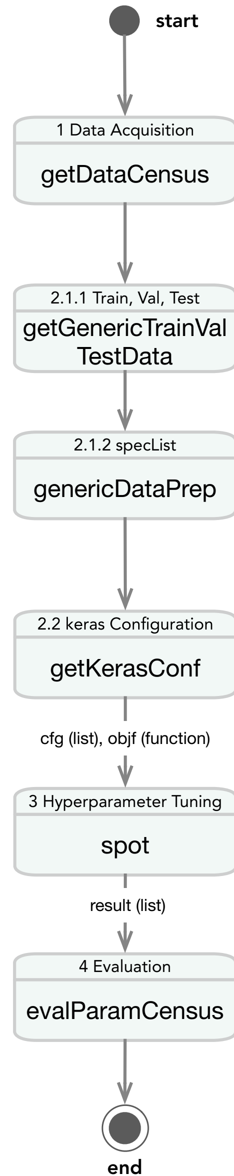

Fig. 10.1 Overview. The HPT pipeline introduced in this chapter comprehends the following steps: After the data acquisition (`getDataCensus`), the data is split into training, validation, and test sets. These data sets are processed via the function `genericDataPrep`. `keras` is configured via the function `getKerasConf`. The hyperparameter tuner `spot` is called and finally, the results are evaluated (`evalParamCensus`)

10.2 Data Description

Identically to the ML case studies, the DL case study presented in this chapter uses the Census-Income (KDD) Data Set (CID), which is made available, for example, via the University of California, Irvine (UCI) Machine Learning Repository.[1,2]

10.2.1 `getDataCensus`: Getting the Data from OpenML

Before training the DNN, the data is preprocessed by reshaping it into the shape the DNN can process. The function `getDataCensus` is used to get the Open Machine Learning (OpenML) data (from cache or from server). The same options as in the previous ML studies will be used, i.e., the parameter settings from Table 8.3 will be used.

```
target <- "age"
task.type <- "classif"
nobs <- 1e4
nfactors <- "high"
nnumericals <- "high"
cardinality <- "high"
data.seed <- 1
cachedir <- "oml.cache"
prop <- 2 / 3
dfCensus <- getDataCensus(
  task.type = task.type,
  nobs = nobs,
  nfactors = nfactors,
  nnumericals = nnumericals,
  cardinality = cardinality,
  data.seed = data.seed,
  cachedir = cachedir,
  target = target
)
```

10.2.2 `getGenericTrainValTestData`: Split Data in Train, Validation, and Test Data

The data frame `dfCensus`, $(X, Y) \subset (\mathcal{X}, \mathcal{Y})$, with 10 000 observations of 23 variables, is available. Based on `prop`, the data is split into training, validation, and test

[1] https://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD).

[2] The data from CID is historical. It includes wording or categories regarding people which do not represent or reflect any views of the authors and editors.

data sets, $(X, Y)^{(\text{train})}$, $(X, Y)^{(\text{val})}$, and $(X, Y)^{(\text{test})}$, respectively. If `prop = 2/3`, the training data set has 4 444 observations, the validation data set has 2 222 observations, and the test data set the remaining 3 334 observations.

```
data <- getGenericTrainValTestData(dfGeneric = dfCensus, prop = prop)
```

10.2.3 *genericDataPrep: Spec*

The third step of the data preprocessing generates a `specList`.

```
batch_size <- 32
specList <- genericDataPrep(data = data, batch_size = batch_size)
```

The function `genericDataPrep` works as described in Sects. 10.2.3.1–10.2.3.5.

10.2.3.1 The Iterator: Data Frame to Data Set

The helper function `df_to_dataset`[3] converts the data frame `dfCensus` into a data set. This procedure enables processing of very large Comma Separated Values (CSV) files (so large that they do not fit into memory). The elements of the training data sets are randomly shuffled. Finally, consecutive elements of this data set are combined into batches.

Applying the function `df_to_dataset` generates a list of tensors. Each tensor represents a single column. The most significant difference to R's data frames is that a TF data set is an iterator.

```
train_ds_generic <-
  df_to_dataset(data$trainGeneric, batch_size = batch_size)
val_ds_generic <-
  df_to_dataset(data$valGeneric, shuffle = FALSE, batch_size = batch_size)
```

Background: Iterators

Each time an iterator is called it will yield a different batch of rows from the data set. The iterator function `iter_next` can be called as follows, so that batches are shown.

```
train_ds %>%
  reticulate::as_iterator() %>%
  reticulate::iter_next()
```

[3] https://tensorflow.rstudio.com/tutorials/advanced/structured/classify/.

The data set `train_ds_generic` returns a list of column names (from the data frame) that map to column values from rows in the data frame.

10.2.3.2 The feature_spec Object: Specifying the Target

TF has built-in methods to perform common input conversions.[4] The powerful `feature_column` system will be accessed via the user-friendly, high-level interface called `feature_spec`. While working with structured data, e.g., CSV data, column transformations and representations can be initialized and specified. A practical benefit of implementing data preprocessing within model $\mathscr{A}$ is that when $\mathscr{A}$ is exported, the preprocessing is already included. In this case, new data can be passed directly to $\mathscr{A}$.

! Attention: Keras Preprocessing Layers

`keras` and `tensorflow` are under constant development. The current implementation in `SPOTMisc` classifies structured data with feature columns. The corresponding TF module was designed for the use with TF version 1 estimators. It does fall under compatibility guarantees.[5] The newly developed `keras` module uses “preprocessing layers” for building `keras`-native input processing pipelines. Future versions of `SPOTMisc` will be based on preprocessing layers. However, because the underlying ideas of both preprocessing layers are similar (TF provides a migration guide[6]), the most important preprocessing steps will be presented next.

First the `spec` object `specGeneric` is defined. The response variable, here: `target`, can be specified using a formula, see Chambers and Hastie (1992) and the R function `formula`.

```
specGeneric <- feature_spec(dataset = train_ds_generic, target ~ .)
```

10.2.3.3 Adding Steps to the feature_spec Object

The CID data set contains a variety of data types. These mixed data types are converted to a fixed-length vector for the DL model to process. Based on their *feature type*, their *data type* or *level*, the columns will be treated differently. After creating the `feature_spec` object the step functions from Table 10.2 can be used to

[4] https://tensorflow.rstudio.com/tutorials/beginners/load/load_csv/.

[5] https://www.tensorflow.org/tutorials/structured_data/feature_columns?authuser=0.

[6] https://www.tensorflow.org/guide/migrate/migrating_feature_columns.

Table 10.2 Steps: data transformations depending on the data type

Step functions	Description
`step_numeric_column`	Numeric variables
`step_categorical_with_vocabulary_list`	Categorical variables with a fixed vocabulary
`step_categorical_column_with_hash_bucket`	Categorical variables using the hash trick
`step_categorical_column_with_identity`	Categorical variables stored as integers
`step_categorical_column_with_vocabulary_file`	Vocabulary stored in a file

Table 10.3 Description of the CID feature and data types that are used in the data set $(X, Y) \subset (\mathcal{X}, \mathcal{Y})$

Column	Feature type	Data type/levels
capital_gains	Num	Double
capital_losses	Num	Double
dividends_from_stocks	Num	Double
wage_per_hour	Num	Double
weeks_worked_in_year	Num	Integer
class_of_worker	Factor	9
industry_code	Factor	51
occupation_code	Factor	47
Education	Factor	17
marital_status	Factor	7
major_industry_code	Factor	24
major_occupation_code	Factor	15
Race	Factor	5
hispanic_origin	Factor	10
Sex	Factor	2
tax_filer_status	Factor	6
detailed_household_and_family_stat	Factor	29
detailed_household_summary_in_household	Factor	8
country_of_birth_self	Factor	42
Citizenship	Factor	5
income_class	Factor	2
Target	Factor	2

add further `steps`. Depending on the data type, the step functions specify the data transformations. Table 10.3 shows these types.

The R package `tfdatasets` provides selectors to select certain variable types and ranges, e.g., `all_numeric` to select all numeric variables, `all_nominal`

to select all characters, or `has_type("float32")` to select variables based on their TF variable type. Based on the feature and data type shown in Table 10.3, the data transformations from Table 10.2 are applied. We will consider feature specs for continuous and catergorical data separately.

10.2.3.4 Feature Spec: Continuous Data

For continuous data, i.e., numerical variables, the function `step_numeric_column` will be used and all numeric variables will be normalized (scaled). The R package `tfdataset` provides the scaler function `scaler_min_max`, which uses the minimum and maximum of the numeric variable and the function `scaler_standard`, which uses the mean and the standard deviation.

10.2.3.5 Feature Spec: Categorical Data

The DNN model $\mathcal{A}$ cannot directly process categorical (nominal) data—they must be transformed so that they can be represented as numbers. The representation of categorical variables as a set of one-hot encoded columns is widely used in practice (Chollet and Allaire 2018). There are basically two options for specifying the kind of numeric representation used for categorical variables: indicator columns or embedding columns.

Background: Embedding

Suppose instead of having a factor with a few levels (e.g., three categorical features such as `red`, `green`, or `blue`), there are hundreds or even more levels. As the number of levels grows very large, it becomes unfeasible to train a DNN using one-hot encodings. In this situation, *embedding* should be used: instead of representing the data as a very large one-hot vector, the data can be stored as a low-dimensional vector of real numbers. Note, the size of the embedding is a parameter that must be tuned (Abadi et al. 2015).

The implementation in `SPOTMisc` uses two steps: first, based on the number of `levels`, i.e., the value of the parameter `minLevelSizeEmbedding` in the following code, the set of columns where embedding should be used, is determined. Then, either the function `step_indicator_column` or the function `step_embedding_column` is applied.

```
minLevelSizeEmbedding <- 100
embeddingDim <- floor(log(minLevelSizeEmbedding))
df <- data$trainGeneric
df <- df[-which(names(df) == "target")]
embeddingVars <-
```

```
  names(df %>%
    mutate_if(is.character, factor) %>%
    select_if(~ is.factor(.) & nlevels(.) > minLevelSizeEmbedding))
noEmbeddingVars <-
  names(df %>%
    mutate_if(is.character, factor) %>%
    select_if(~ is.factor(.) & nlevels(.) <= minLevelSizeEmbedding))
specGeneric <- specGeneric %>%
  step_numeric_column(all_numeric(),
    normalizer_fn = scaler_standard()
  ) %>%
  step_categorical_column_with_vocabulary_list(all_nominal()) %>%
  step_indicator_column(matches(noEmbeddingVars)) %>%
  step_embedding_column(matches(embeddingVars), dimension = embeddingDim)
```

After adding a `step` we need to `fit` the `specGeneric` object:

```
specGeneric_prep <- fit(specGeneric)
```

Finally, the following data structures are available:

1. `train_ds_generic` (batched, based on 4444 samples)
2. `val_ds_generic`, (batched, based on 2222 samples)
3. `specGeneric_prep` and
4. `testGeneric` (the remaining 3334 samples).

These data are returned as the list `specList` from the function `genericDataPrep`.

```
specList <- genericDataPrep(data = data, batch_size = batch_size)
```

Dense features prepared with TF's feature columns mechanism can be listed. There are 22 dense features that will be passed to the DNN.

```
names(specList$specGeneric_prep$dense_features())

##  [1] "wage_per_hour"
##  [2] "capital_gains"
##  [3] "capital_losses"
##  [4] "divdends_from_stocks"
##  [5] "num_persons_worked_for_employer"
##  [6] "weeks_worked_in_year"
##  [7] "indicator_class_of_worker"
##  [8] "indicator_industry_code"
##  [9] "indicator_major_industry_code"
## [10] "indicator_occupation_code"
## [11] "indicator_major_occupation_code"
## [12] "indicator_education"
## [13] "indicator_marital_status"
## [14] "indicator_race"
## [15] "indicator_hispanic_origin"
```

```
## [16] "indicator_sex"
## [17] "indicator_tax_filer_status"
## [18] "indicator_detailed_household_and_family_stat"
## [19] "indicator_detailed_household_summary_in_household"
## [20] "indicator_country_of_birth_self"
## [21] "indicator_citizenship"
## [22] "indicator_income_class"
```

10.3 Experimental Setup and Configuration of the Deep Learning Models

10.3.1 `getKerasConf`: `keras` and Tensorflow Configuration

Setting up the `keras` configuration from within `SPOTMisc` is a simple step: the function `getKerasConf` is called. The function `getKerasConf` passes additional parameters to the `keras` function, e.g.,

`activation:` Activation function in the last Neural Network (NN) layer. Default: `"sigmoid"`.

`active:` Vector of active variables, e.g., `c(1,10)` specifies that only the first and tenth variable will be considered by `spot`. This mechanism allows the shrinking the full set of tunable parameters, say λ, to a smaller set, $\lambda^{(-)}$, if the user wants to investigate the tunability (or the effect) of one or only a few hyperparameters.

`callbacks:` List of callbacks to be called during training. Default: `list()`.

`clearSession:` Whether to call `k_clear_session` or not at the end of `keras` modeling. Default: `FALSE`.

`encoding:` Encoding used during data preparation. Default: `"oneHot"`.

`loss:` Loss function, $\mathcal{L}$, for the `compile` from the package `keras`. For example Binary Cross Entropy (BCE) loss as defined in Eq. (2.3).
Default: `"loss_binary_crossentropy"`.

`metrics:` Metrics function for compile. Default: `"binary_accuracy"`.

`model:` Model, $\mathcal{A}$, as specified via `getModelConf`. Default: `"dl"`. Forthcoming versions of `SPOTMisc` will pro-

	vide additional DNN model types, e.g., Convolutional Neural Networks (CNNs).
`nClasses:`	Number of classes in (multi-class) classification. Specifies the number of units in the last layer (before `softmax`). Default: `1` (binary classification).
`resDummy:`	If `TRUE`, generate dummy (mock up) result for testing. If `FALSE`, run `keras` and `tensorflow` evaluations. Default: `FALSE`.
`returnValue:`	Return value. Can be one of `"trainingLoss"`, `"negTrainingAccuracy"`, `"validationLoss"`, `"negValidation Accuracy"`, `"testLoss"`, or `"negTestAccuracy"`.
`returnObject:`	Return object. Can be one of `"evaluation"`, `"model"`, `"pred"`. Default: `"evaluation"`.
`shuffle:`	Logical (whether to shuffle the training data $(X, Y)^{(\text{train})}$ before each epoch) or string (for "batch"). Used in the function `df_to_dataset`. "batch" is a special option for dealing with the limitations of the Hierarchical Data Format (HDF) version 5 data. It shuffles in batch-sized chunks. Default: `FALSE`.
`testData:`	Test data, $(X, Y)^{(\text{test})}$, on which to evaluate the loss, $\mathcal{L}$, and any model metrics, $\psi^{(\text{test})}$ at the end of the optimization using the function `evaluate`.
`tfDevice:`	Tensorflow device. CPU/GPU allocation. Passed to `tensorflow` via `tf$device(kerasConf $tfDevice)`. Default: `"/cpu:0"` (use CPU only).
`trainData:`	Training data, $(X, Y)^{(\text{train})}$, on which to evaluate the loss and any model metrics at the end of each epoch.
`validationData:`	Validation data, $(X, Y)^{(\text{val})}$, on which to evaluate the loss $\psi^{(\text{val})}$ and any model metrics at the end of each epoch.
`validation_split:`	Float between 0 and 1. Fraction of the training data $(X, Y)^{(\text{train})}$ to be used internally by $\mathcal{A}$ as validation data $(X, Y)^{(\text{valtrain})}$. $\mathcal{A}$ will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on $(X, Y)^{(\text{valtrain})}$ at the end of each epoch. $(X, Y)^{(\text{valtrain})}$ is selected from the last samples in the $(X, Y)^{(\text{train})}$ data provided, before shuffling. Default: `0.2`.
`verbose:`	Verbosity mode (0 = silent, 1 = progress bar, 2 = one line per epoch). Default: `0`.

The default settings are useful for the binary classification task analyzed in this chapter. Only the parameter `kerasConf$clearSession` is set to `TRUE` and `kerasConf$verbose` is set to `0`.

```
kerasConf <- getKerasConf()
kerasConf$clearSession <- TRUE
kerasConf$verbose <- 0
```

10.3.2 *getModelConf: DL Hyperparameters*

```
cfg <- getModelConf(model = "dl")
```

If the default values from the function `getKerasConf` are used, the vector of hyperparameter λ contains the following elements: the dropout rates (dropout rates of the layers will be tuned individually), the number of units (the number of single outputs from a single layer), the learning rate (controls how much to change the DNN model in response to the estimated error each time the model weights are updated), the number of training epochs (a training epoch is one forward and backward pass of a complete data set), the optimizer for the inner loop, O_{inner}, and its parameters (i.e., β_1, β_2 as well as ϵ) and the number of layers. These hyperparameters and their ranges are listed in Table 10.4.

Table 10.4 The hyperparameters, λ, for the DNN, which implements a fully connected network

Variable	Hyperparameter	Type	Default	Lower bound	Upper bound
x_1	`dropout`: first layer dropout rate	Numeric	0	0	0.4
x_2	`dropoutfact`: dropout multiplier	Numeric	0	0	0.5
x_3	`units`: units per first layer	Integer	32	1	32
x_4	`unitsfact`: units multiplier	Numeric	0.2	0.25	1
x_5	`learning_rate`: learning rate for the optimizer	Numeric	$1e-3$	$1e-6$	$1e-2$
x_6	`epochs` inner loop O_{inner} number of training epochs	Integer	16	8	128
x_7	`beta_1`	Numeric	0.9	0.9	0.99
x_8	`beta_2`	Numeric	0.999	0.999	0.9999
x_9	`layers`	Integer	1	1	4
x_{10}	`epsilon`	Numeric	$1e-7$	$1e-9$	$1e-8$
x_{11}	`optimizer`	Factor	5	1	7

Table 10.5 Optimizers that can be selected via hyperparameter x_{11}. Default optimizer O_{inner} is `adam`. The function `selectKerasOptimizer` from the `SPOTMisc` implements the selection. The corresponding R functions have the prefix `optimizer_`, e.g., `adamax` can be called via `optimizer_adamax`

Level	Name	Description	Reference
1	sgd	SGD optimizer with support for momentum, learning rate decay, and Nesterov momentum	Ruder (2017)
2	rmsprop	RMSProp optimizer	Ruder (2017)
3	adagrad	Adagrad optimizer	Duchi et al. (2011)
4	adadelta	Adadelta optimizer	Zeiler (2012)
5	adam	Adam optimizer	Kingma and Ba (2014)
6	adamax	Adamax optimizer	Kingma and Ba (2014)
7	nadam	Nesterov Adam optimizer	Sutskever et al. (2013)

To enable compatibility with the ranges of the learning rates of the other optimizers, the learning rate of the optimizer `adadelta` is internally mapped to `1-learning_rate`. That is, a learning rate of 0 will be mapped to 1 (which is `adadelta`'s default learning rate). The learning rate of `adagrad` and `sgd` is internally mapped to `10 * learning_rate`. That is, a learning rate of 0.001 will be mapped to 0.01 (which is `adagrad`'s and `sgd`'s default). The learning rate learning_rate of `adamax` and `nadam` is internally mapped to `2 * learning_rate`. That is, a learning rate of 0.001 will be mapped to 0.002 (which is `adamax`'s and `nadam`'s default.)

The hyperparameter x_{11}, which encodes the `optimizer` is implemented as a factor. Factor levels, which represent the available optimizers are listed in Table 10.5.

A discussion of the DNN hyperparameters, λ, recommendations for their settings and further information are presented in Sect. 3.8. The R function `getModelConf` provides information about hyperparameter names, ranges, and types.

10.3.3 The Neural Network

Background: Network Implementation in `SPOTMisc`

The `SPOTMisc` function `getModelConf` selects a pre-specified, but not pre-trained, DL network $\mathcal{A}$. This network is called via `funKerasGeneric`, which is the interface to `spot`. `funKerasGeneric` uses a network, that is implemented as follows:

To build the DNN in `keras`, the function `layer_dense_features` that processes the feature columns specification is used (Fig. 10.2). It receives the data set `specGeneric_prep` as input and returns an array off all dense features:

```
layer <-
  layer_dense_features(
    feature_columns = dense_features(specList$specGeneric_prep)
  )
```

The iterator can be called to take a look at the (scaled) output:

```
specList$train_ds_generic %>%
  reticulate::as_iterator() %>%
  reticulate::iter_next() %>%
  layer()
```

The NN model can be compiled after the `loss` function $\mathcal{L}$, which determines how good the DNN prediction is (based on the $(X, Y)^{(\text{val})}$), the `optimizer`, i.e., O_{inner}, i.e., the update mechanism of $\mathcal{A}$, which adjusts the weights using backpropagation, and the `metrics`. *metrics* The metrics monitor the progress during training and testing and are specified using the `compile` function from `keras`.

! Attention: Hyperparameter Values

To improve the readability of the code, evaluated ("forced" values) of the hyperparameters λ are shown in the code snippets below instead of the arguments that are passed from the tuner `spot` to the function `funKerasGeneric`.

```
units1 <- 2
model <- keras_model_sequential() %>%
  layer_dense_features(dense_features(specList$specGeneric_prep)) %>%
  layer_dense(units = units1, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model %>% compile(
  loss = loss_binary_crossentropy,
  optimizer = "adam",
  metrics = "binary_accuracy"
)
```

The DNN training can be started as follows (using `keras`' `fit` function). Train the model on the CPU using the setting `tf$device("/cpu:0")` on the validation data set:

```
with(tf$device("/cpu:0"), {
  historyD <-
    model %>%
    fit(dataset_use_spec(specList$train_ds_generic,
      spec = specList$specGeneric_prep
    ),
    epochs = 25,
```

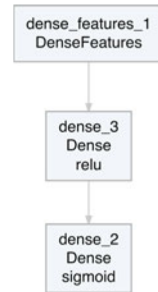

Fig. 10.2 Simple DNN based on the code in this section

```
    validation_data =
      dataset_use_spec(
        specList$val_ds_generic,
        specList$specGeneric_prep
      ),
    verbose = 0
    )
})
```

The predictions from the DNN model are shown in the following code snippet. The tensor values are the output from the final DNN layer after the sigmoid function was applied. Values are from the interval [0, 1] and represent probabilities: values smaller than 0.5 are interpreted as predictions "age < 40", otherwise "age ≥ 40".

```
specList$test_ds_generic %>%
  reticulate::as_iterator() %>%
  reticulate::iter_next() %>%
  model()
```

```
## tf.Tensor(
## [[0.31883082]
##  [0.47055224]
##  [0.99928933]
##  [0.9962864 ]
##  [0.27977774]
##  [0.34997565]
##  [0.7686823 ]
##  [0.99928933]
##  [0.32100695]
##  [0.99928933]
##  [0.16852783]
##  [0.33614054]
##  [0.36855838]
##  [0.4346528 ]
##  [0.6968227 ]
##  [0.41458437]
```

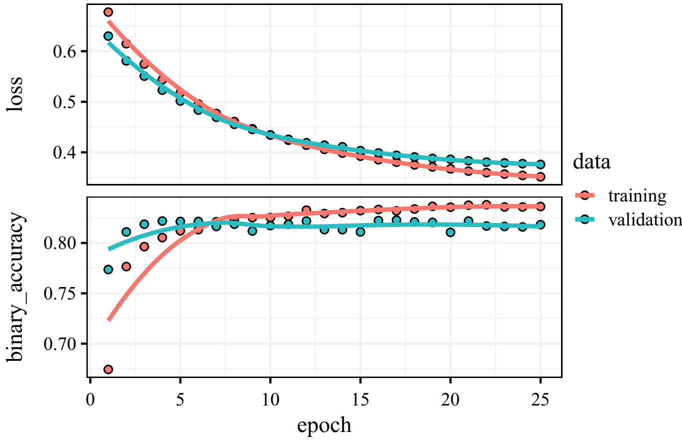

Fig. 10.3 DNN training. History of the inner optimization loop

```
##   [0.9992539 ]
##   [0.01920704]
##   [0.34810022]
##   [0.6455758 ]
##   [0.78468007]
##   [0.9993542 ]
##   [0.9469396 ]
##   [0.00989294]
##   [0.00521746]
##   [0.01071302]
##   [0.8888161 ]
##   [0.78542197]
##   [0.9993542 ]
##   [0.6045456 ]
##   [0.9993542 ]
##   [0.9992539 ]], shape=(32, 1), dtype=float32)
```

Figure 10.3 shows the quantities that are being displayed during training:

(i) the *loss* of the network over the training and validation data, $\psi^{(\text{train})}$ and $\psi^{(\text{val})}$, respectively, and
(ii) the *accuracy* of the network over the training and validation data, $f_{\text{acc}}^{(\text{train})}$ and $f_{\text{acc}}^{(\text{val})}$, respectively.

This figure illustrates that an accuracy greater than 80% on the training data, $(X, Y)^{(\text{train})}$, can be reached quickly.

Figure 10.3 can indicate (even if this is only a short fit procedure) whether the modeling is affected by overfitting or not. If this situation occurs, it might be useful to implement dropout layers or use other methods to prevent overfitting.

The effects of HPT and the tunability of $\mathcal{A}$ will be described in the following sections. Finally, using `keras`' `evaluate` function, the DNN model performance can be checked on $X^{(\text{test})}$.

```
model %>%
  evaluate(specList$test_ds_generic %>%
    dataset_use_spec(specList$specGeneric_prep), verbose = 0)

##            loss binary_accuracy
##       0.3550636       0.8068387
```

The relationship between $\psi^{(\text{train})}$, $\psi^{(\text{val})}$, and $\psi^{(\text{test})}$ as well as between $f_{\text{acc}}^{(\text{train})}$, $f_{\text{acc}}^{(\text{val})}$, and $f_{\text{acc}}^{(\text{test})}$ can be analyzed with Sequential Parameter Optimization Toolbox (SPOT), because it computes and reports these values.

10.4 `funKerasGeneric`: The Objective Function

The hyperparameter tuner, e.g., `spot`, performs model selection during the tuning run: training data $X^{(\text{train})}$ is used for fitting (training) the models, e.g., the weights of the DNNs. Each trained model $\mathcal{A}_{\lambda_i}\left(X^{(\text{train})}\right)$ will be evaluated on the validation data $X^{(\text{val})}$, i.e., the loss is calculated as shown in Eq. (2.9). Based on $(\lambda_i, \psi_i^{(\text{val})})$, at each iteration of the outer optimization loop a surrogate model $\mathcal{S}(t)$ is fitted, e.g., a Bayesian Optimization (BO) (Kriging) model using `spot`'s `buildKriging` function.

For each hyperparameter configuration λ_i, the objective function `funKerasGeneric` reports information about the related DNN models $\mathcal{A}_{\lambda_i}$

1. training loss, $\psi^{(\text{train})}$,
2. training accuracy, $f_{\text{acc}}^{(\text{train})}$,
3. validation (testing) loss, $\psi^{(\text{val})}$, and
4. validation (testing) accuracy, $f_{\text{acc}}^{(\text{val})}$.

10.5 `spot`: Experimental Setup for the Hyperparameter Tuner

The `SPOT` package for R, which was introduced in Sect. 4.5, will be used for the DL hyperparameter tuning (Bartz-Beielstein et al. 2021). The budget is set to twelve hours, i.e., the run time of DL tuning is larger than the run time of the ML tuning. The budget for the `spot` runs was set to this value, because of the complexity of the hyperparameter search space Λ and the relatively long run time of the DNN.

SPOT provides several options for adjusting the HPT parameters, e.g., type of the Surrogate Model Based Optimization (SMBO) model, $\mathcal{S}$, and optimizer, $\mathcal{O}$, as well as the size of the initial design, n_{init}. These parameters can be passed via the `spotControl` function to `spot`. For example, instead of the default surrogate $\mathcal{S}$, which is BO (implemented as `buildKriging`), a Random Forest (RF), (implemented as `buildRanger`) can be chosen.

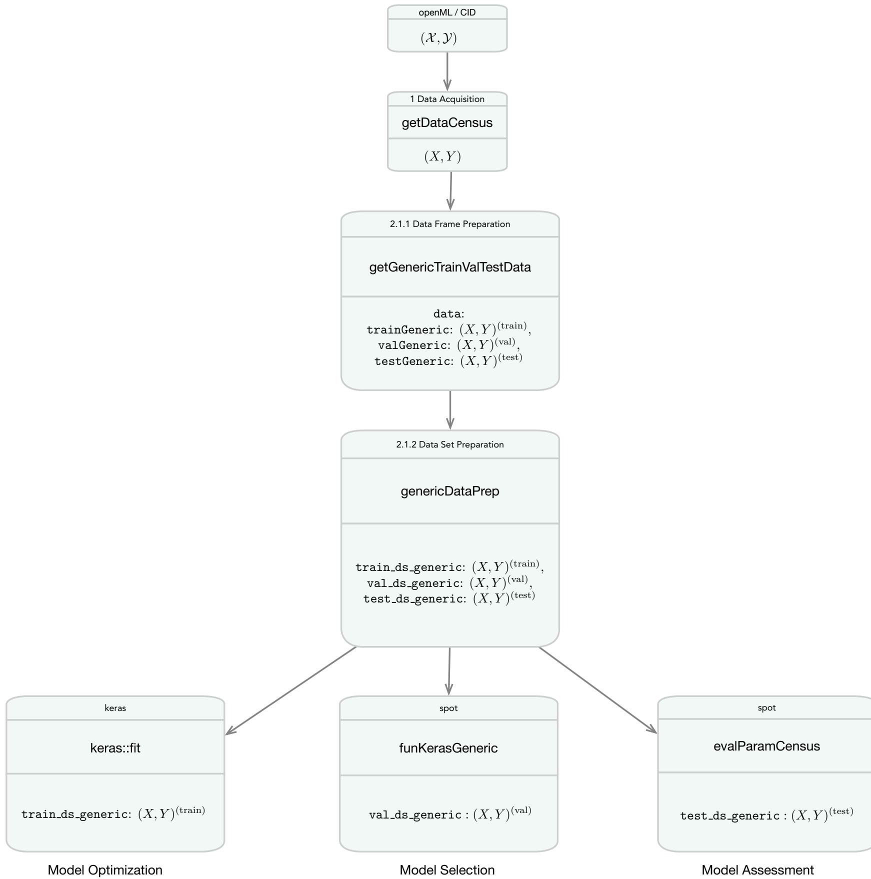

Fig. 10.4 Overview. The DL HPT data workflow

The general DL HPT data workflow is as follows: first the training data, $(X, Y)^{(\text{train})}$ are fed to the DNN. The DNN will then learn to associate images and labels. Based on the `keras` parameter `validation_split`, the training data will be partitioned into a (smaller) training data set, $X^{(\text{train})}$, and a validation data set, $(X, Y)^{(\text{valtrain})}$. The trained DNN produces predictions for validations based on $(X, Y)^{(\text{val})}$ data. The DL HPT data workflow is shown in Fig. 10.4.

Similar to the process described in Sect. 8.1 for ML, the hyperparameter tuning for DL can be started as follows:

```
startCensusRun(model = "dl")
```

The `startCensusRun` function performs the following steps:

1. Providing the CID data set, $((\mathcal{X}, \mathcal{Y})_{\text{CID}}$, see Sect. 8.2.1.
2. Generating the random sample $(X, Y) \subseteq ((\mathcal{X}, \mathcal{Y})_{\text{CID}}$ of size `nobs`.

Table 10.6 SPOT parameters used for deep learning hyperparameter tuning. The `control` list contains internally further lists, see Table 10.7

Parameter	Value	Description
`x`	x0	Starting point, hyperparameter vector λ, see Tables 10.4 and 10.5
`fun`	`funKerasGeneric`	Objective function, O_{outer}
`lower`	`cfg$lower`	Lower bounds for `x` aka λ
`upper`	`cfg$upper`	Upper bounds for `x` aka λ
`control`	List	
`kerasConf`	`kerasConf`	Argument used by the objective function `funKerasGeneric`
`specList`	`specList`	Argument used by the objective function `funKerasGeneric`

Table 10.7 SPOT list parameters used for deep learning hyperparameter tuning

List	Parameter	Value
Control	Types	cfg$type
	Verbosity	Verbosity
	Time	List (maxTime = timebudget/60)
	Plots	Plots
	Progress	TRUE
	Model	spotModel
	Optimizer	spotOptim
	Noise	Noise
	OCBA	OCBA
	OCBABudget	OCBABudget
	seedFun	NA
	seedSPOT	tuner.seed
designControl	Replicates	Rinit
	Size	initSizeFactor * length(cfg$lower)
modelControl	Target	krigingTarget
	useLambda	krigingUseLambda
	Reinterpolate	krigingReinterpolate
optimizerControl	funEvals	multFun * length(cfg$lower))
yImputation	handleNAsMethod	handleNAsMethod
	imputeCriteriaFuns	imputeCriteriaFuns
	penaltyImputation	3

3. Defining an experimental design, including performance measures.
4. Configuration of the hyperparameter tuner, $\mathcal{T}$.
5. Configuration of the DL model, $\mathcal{A}$.
6. Performing the experiments.

Furthermore, it can be decided whether to use the default hyperparameter setting, λ_0, as a starting point or not. Using the parameter specifications from Tables 10.6 and 10.7, we are ready to perform the HPT run: `spot` can be started.

10.6 Tunability

Regarding tunability as defined in Definition 2.26, we are facing a special situation in this chapter, because there is no generally accepted "default" hyperparameter configuration, λ_0, for DNNs. This problem is not as obvious in ML, because the corresponding methods have a long history, i.e., there are publications for most of the shallow methods that can give hints how to select adequate λ values. This information is collected and summarized in Chap. 3. The "default" hyperparameter setting of the DNNs analyzed in this chapter is based on our own experiences, combined with recommendations in the literature. Chollet and Allaire (2018) may be considered as a reference in this field.[7]

The `result` list from the `spot` run can be loaded. It contains the 14 values shown in Table 4.6, e.g., names of the tuned hyperparameters that were introduced in Table 10.4:

```
result$control$parNames
##  [1] "dropout"       "dropoutfact"   "units"         "unitsfact"
##  [5] "learning_rate" "epochs"        "beta_1"        "beta_2"
##  [9] "layers"        "epsilon"       "optimizer"
```

The HPT inner optimization loop is shown in Fig. 10.5. The DNN uses the tuned hyperparameters, $\lambda^\star$ from Table 10.8. The model training supports the result found by the tuner `spot` that the number of training epochs should be 32. The reader may compare the inner optimization loop with default and with tuned hyperparameters in Figs. 10.3 and 10.5.

The tuned DNN model has the following structure:

```
## $model
## Model: "sequential_1"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                     Param #
## ================================================================================
##  dense_features_2 (DenseFeatures)   multiple                         0
```

[7] An updated version of Chollet and Allaire (2018) is under preparation while we are writing this text. Check the authors' web-page for more information: https://www.manning.com/books/deep-learning-with-r.

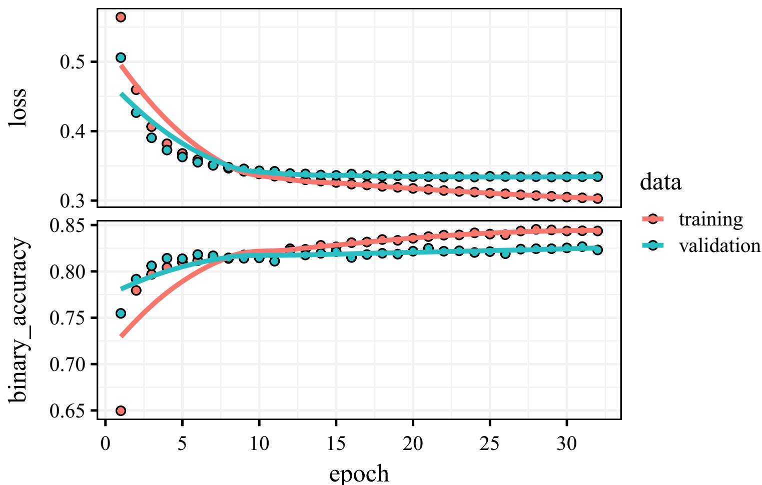

Fig. 10.5 Training DL (inner optimization loop) using the tuned hyperparameter setting $\lambda^\star$

Table 10.8 DNN configurations. “lr” denotes “learning_rate”. The overall mean of the loss, $\overline{y}$ is 0.3691, its standard deviation is 0.1152, whereas the mean of the best HPT configuration, $\lambda^\star$, found by OCBA, is 0.3346 with s.d. 0.0343

dropout	dropoutfact	units	unitsfact	lr	epochs	beta_1	beta_2	layers	epsilon	optimizer	Loss
0	0	5	0.5	0.001	4	0.9	0.999	1	0	5	0.346
0.038	0.793	5	0.742	0.002	5	0.913	0.994	1	0	4	0.335

```
##  dense_2 (Dense)                    multiple                          8864
##  dense_3 (Dense)                    multiple                          33
## ================================================================================
## Total params: 8,897
## Trainable params: 8,897
## Non-trainable params: 0
## ________________________________________________________________________________
##
## $history
##
## Final epoch (plot to see history):
##                loss: 0.2983
##     binary_accuracy: 0.8508
##            val_loss: 0.3343
## val_binary_accuracy: 0.8132
```

10.6.1 Progress

After loading the results from the experiments, the hyperparameter tuning progress can be visually analyzed. First of all, the `result` list information will be used to

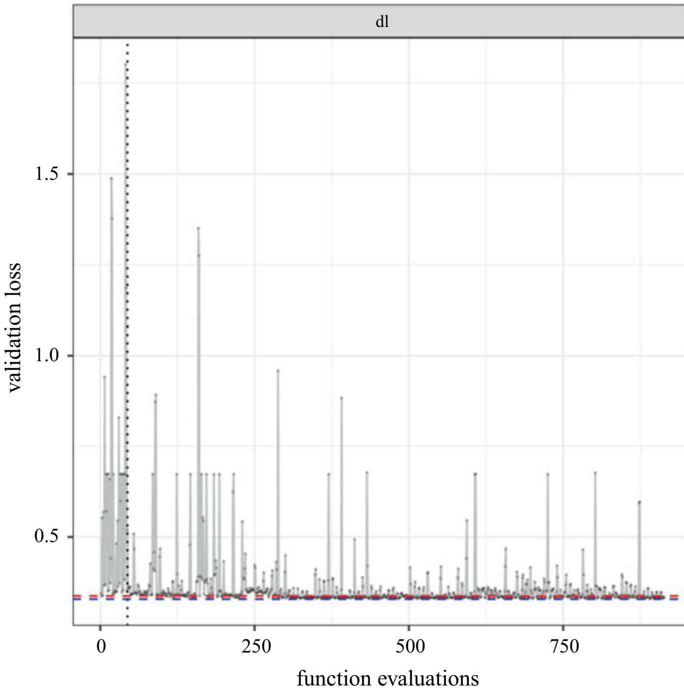

Fig. 10.6 Progress plot. In contrast to the progress plots used for the ML methods, this plot shows the BCE loss and not the MMCE against the number of iterations (function evaluations of the tuner)

visualize the *route to the solution*: in Fig. 10.6, loss function values, $\psi^{(\text{val})}$, are plotted against the number of iterations, t. Each point represents one evaluation of an DNN model $\mathcal{A}_\lambda(t)$ at time step (`spot` iteration) t.

The initial design, which includes the default hyperparameter setting, λ_0, results in a loss value of $\psi^{(\text{val})}_{\text{init}} = 0.3371$. The best value, that was found during the tuning, is $y^{(*)}_{\text{val}} = 0.3285$. These values have to be taken with caution, because they represent onyl one evaluation of $\mathcal{A}_\lambda$. Based on OCBA, which takes the noise in the model evaluation via the function `funKerasGeneric` into consideration, the best function value is $y^{(\text{OCBA}^*)}_{\text{val}} = 0.3346$.

After 12 h, 914 `dl` models were evaluated. Comparing the worst configuration that was observed during the HPT with the best, a 81.773% reduction in the BCE loss was obtained. After the initial phase, which includes 44 evaluations, the smallest BCE reads 0.3370858. The dotted red line in Fig. 8.6 illustrates this result. The final best value reads 0.3285304, i.e., a reduction of the BCE of 2.5381%. These values, in combination with results shown in the progress plot (Fig. 8.6) indicate that a relatively short HPT run is able to improve the quality of the DNN model. It also indicates, that increased run times do not result in a significant improvement of the BCE. The full

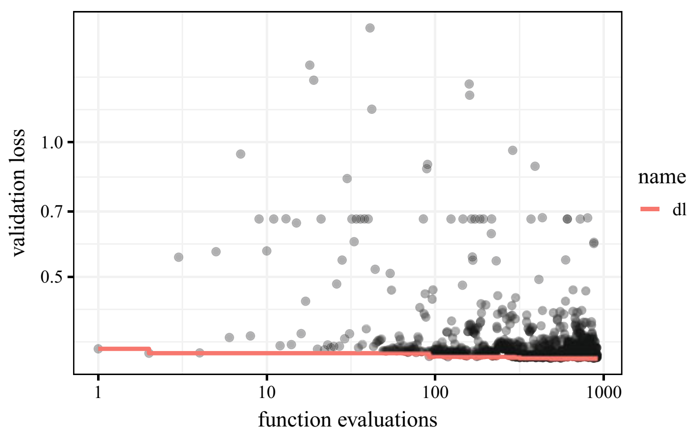

Fig. 10.7 Log-log plot

comparison of the DL and ML algorithm performances with default, λ_0, and tuned, $\lambda^\star$, hyperparameters is shown in Sect. 10.9.

! Attention

These results do not replace a sound statistical comparison, they are only indicators, not final conclusions.

The corresponding code is presented in the Appendix. The related hyperparameters values are shown in Table 10.8.

There is a large variance in the loss as can be seen in Figs. 10.6 and 10.7. The latter of these two plots visualizes the same data as the former, but uses log-log axes instead.

10.6.2 `evalParamCensus`: Comparing Default and Tuned Parameters on Test Data

The function `evalParamCensus` evaluates ML and DL hyperparameter configurations on the CID data set. It compiles a data frame, which includes performance scores from several hyperparameter configurations and can also process results from default settings. This data frame can be used for a comparison of default and tuned hyperparameters, λ_0 and $\lambda^\star$, respectively. A violin plot of this comparison is shown in Fig. 10.8. It is based on 30 evaluations of λ_0 and $\lambda^\star$ and shows—in contrast to the values in the DNN progress plots—the Mean Mis-Classification Error (MMCE). The MMCE was chosen to enable a comparison of the DL results with the ML results

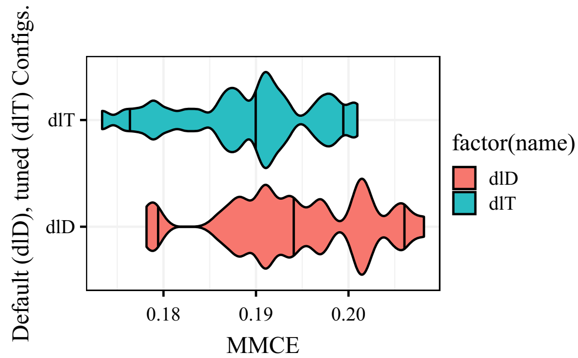

Fig. 10.8 Comparison of DL algorithms with default (D) and tuned (T) hyperparameters. Mean misclassification error (MMCE) for both configurations. Vertical lines mark quantiles (0.25, 0.5, 0.75) of the corresponding distribution. Numerical values are shown in Table 10.8

shown in this book. Identical evaluations were done in Chaps. 8, 9, and 12. A global comparison of the six ML and the DL methods from this book will be shown in Sect. 10.9.

10.7 Analysing the Deep Learning Tuning Process

The values that are used for the analysis in this section are biased because they are not using an experimental design (space filling or factorial). Instead, they are using the data from the `spot` tuning process, i.e., they are biased by the search strategy (Expected Improvement (EI)) on the surrogate $\mathcal{S}$.

Identical to the analysis of the ML methods, a simple regression tree as shown in Fig. 10.9 can be used for analysing effects and interactions between hyperparameters λ.

The regression tree supports the observations, that units and epochs have the largest effect on the validation loss. The importance of the parameters from the random forest analysis are shown in Table 10.9.

To perform a sensitivity analysis, parallel and sensitivity plots can be used.

The parallel plot (Fig. 10.10) indicates that the hyperparameter `units` should be set to a value of 32 (the transformed values range from 1 to 32), the `epochs`, i.e. x_6, should be set to a value of 32 (the transformed values range from 8 to 128), the `layers`, i.e. x_9, should be set to a value of 1 (the transformed values range from 1 to 4), and the `optimizer`, i.e. x_{11}, should be set to a value of 4 (the transformed values range from 1 to 7).

Looking at Fig. 10.11, the following observations can be made: Similar to the results from the parallel plot (Fig. 10.10), the sensitivity plot shows that the `epochs`, i.e. x_6, and the `optimizer`, i.e. x_{11}, have the largest effect: the former leads to poor results for larger values, whereas the latter produces poor results for relatively small values. This indicates that the number of training epochs should not be too large

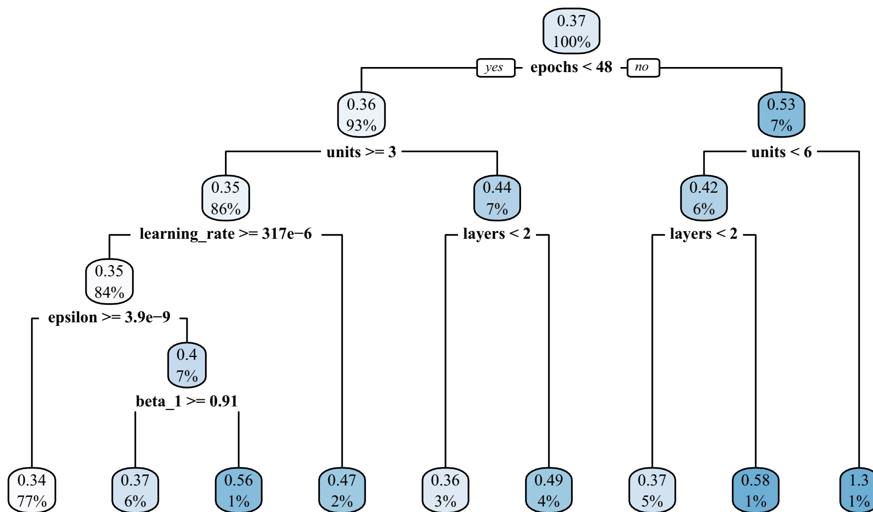

Fig. 10.9 Regression tree. Deep learning model. Transformed hyperparameter values are shown

Table 10.9 Variable importance of the DL model hyperparameters

λ_i	units	epochs	beta_2	layers	lr	beta_1	eps	opt.	dropoutfact	dropout	unitsfact
Var. imp.	6.04	1.69	1.36	0.63	0.47	0.46	0.42	0.33	0.22	0.19	0.10

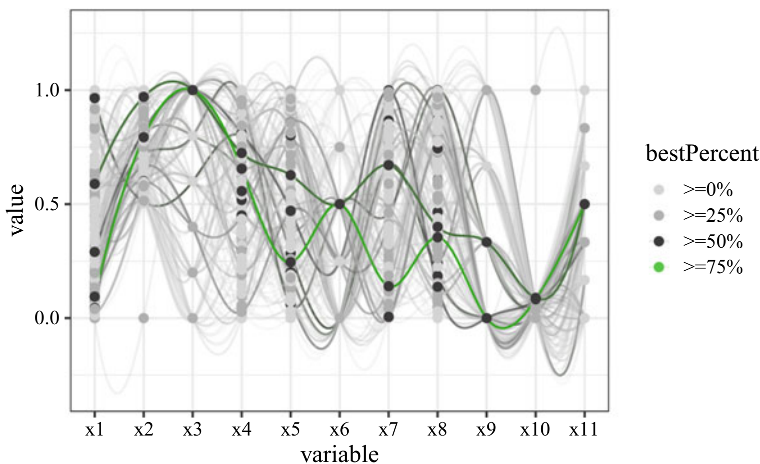

Fig. 10.10 Best configurations in green

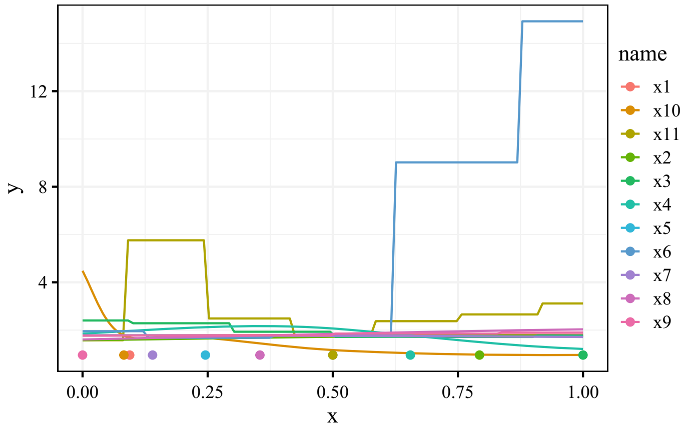

Fig. 10.11 Sensitivity plot (best)

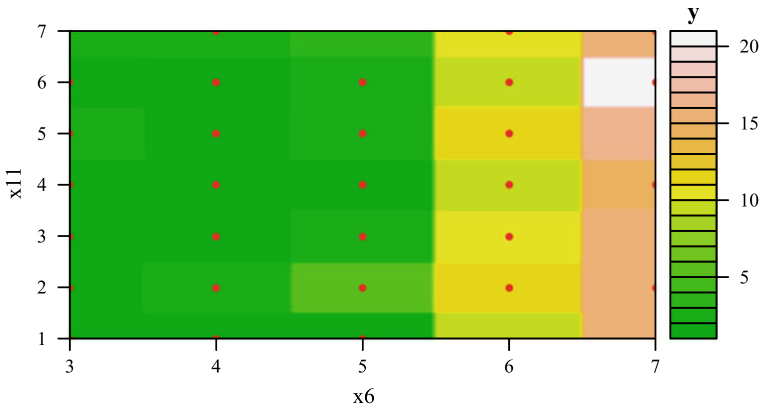

Fig. 10.12 Surface plot: epochs x_6 plotted against optimizer x_{11}. This plot indicates that longer training (larger `epochs` values) worsen the performance and that the optimizer `adadelta` performs well. Note: Plateaus are caused by discrete and factor variables

(probably to prevent overfitting, see Fig. 10.5) and that the optimizers `adadelta` or `adam` are recommended (Fig. 10.12).

Finally, a simple linear regression model can be fitted to the data. Based on the data from SPOT's `res` list, this can be done as follows:

```
##
## Call:
## lm(formula = y ~ ., data = df)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
```

```
## -0.19062 -0.04055 -0.00477 -0.00044  1.16255
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)     2.291e+00  1.494e+00   1.533 0.125532
## dropout         9.491e-02  3.804e-02   2.495 0.012776 *
## dropoutfact     3.807e-02  3.167e-02   1.202 0.229606
## units           9.670e-03  3.544e-03   2.729 0.006484 **
## unitsfact      -5.514e-02  2.225e-02  -2.478 0.013396 *
## learning_rate   8.281e+00  1.509e+00   5.488 5.29e-08 ***
## epochs          4.832e-02  4.628e-03  10.442  < 2e-16 ***
## beta_1          3.456e-01  1.705e-01   2.028 0.042888 *
## beta_2         -2.589e+00  1.486e+00  -1.743 0.081739 .
## layers          2.360e-02  4.573e-03   5.161 3.03e-07 ***
## epsilon        -1.522e+06  7.284e+05  -2.089 0.036961 *
## optimizer2      4.672e-02  1.783e-02   2.620 0.008933 **
## optimizer3      2.791e-02  1.575e-02   1.772 0.076659 .
## optimizer4     -9.552e-03  1.343e-02  -0.711 0.477196
## optimizer5      1.282e-01  2.094e-02   6.121 1.39e-09 ***
## optimizer6      6.941e-02  1.572e-02   4.415 1.13e-05 ***
## optimizer7      1.172e-01  3.020e-02   3.880 0.000112 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.09997 on 897 degrees of freedom
## Multiple R-squared:  0.2595, Adjusted R-squared:  0.2463
## F-statistic: 19.65 on 16 and 897 DF,  p-value: < 2.2e-16
```

Although this linear model requires a detailed investigation (a misspecification analysis is recommended, see, e.g., Spanos 1999), it can be used in combination with other Exploratory Data Analysis (EDA) tools and visualizations from this section to discover unexpected and/or interesting effects. It should not be used alone for a final decision. Despite of a relatively low adjusted R^2 value, the regression output shows—in correspondence with previous observations—that increasing the number of `epochs` worsens the model performance.

10.8 Severity: Validating the Results

Considering the results of the experimental runs the difference is $\bar{x} = 0.0054$. Since this value is positive, for the moment, let us assume that the tuned solution is superior. The corresponding standard deviation is $s_d = 0.0056$. Based on Eq. 5.14, and with $\alpha = 0.05$, $\beta = 0.2$, and $\Delta = 0.006$.

Next, we will identify the required number of runs for the full experiment using the `getSampleSize` function. For a relevant difference of 0.006 approximately 11 completing runs per algorithm are required. Hence, we can directly proceed to evaluate the severity and analyse the performance improvement achieved through tuning the parameters of the DL model.

Table 10.10 Case Study III: Result Analysis

p-value	Decision	Power	Cohen's d	Hedge's g	Severity
0	H0 rejected	0.9999849	0.695314	0.686284	$\Delta \leq 0.0045$ are well supported

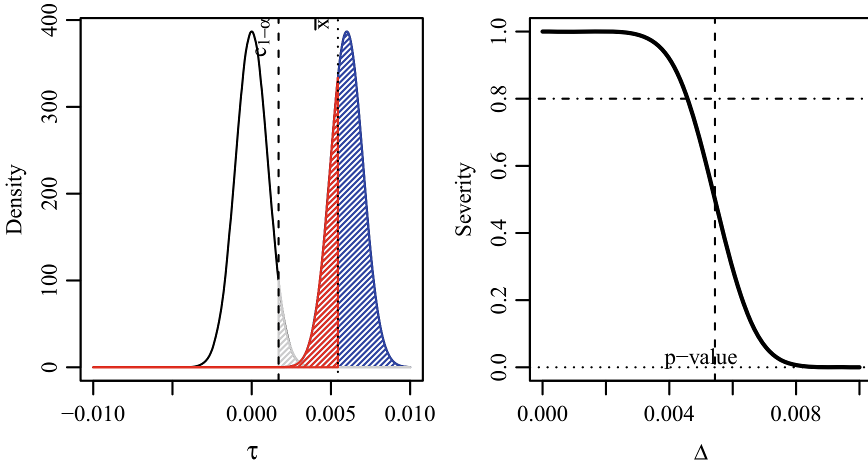

Fig. 10.13 Tuning DL. Severity of rejecting H0 (red), power (blue), and error (gray). Left: the observed mean $\bar{x} = 0.0054$ is larger than the cut-off point $c_{1-\alpha} = 0.0017$ Right: The claim that the true difference is as large or larger than 0.006 is not supported by severity. But, any difference smaller than 0.0045 is supported by severity

Result summaries are presented in Table 10.10. The decision based on p-value is to reject the null hypothesis, i.e, the claim that the tuned parameter setup provides a significant performance improvement in terms of MMCE is supported. The effect size suggests that the difference is of medium magnitude. For the chosen $\Delta = 0.006$, the severity value is at 0.29 and thus it does not support the decision of rejecting the H_0. The severity plot is shown in Fig. 10.13. Severity shows that only performance differences smaller than 0.0045 are well supported.

10.9 Summary and Discussion

A HPT approach based on SMBO was introduced and exemplified in this chapter. It uses functions from the packages `keras`, `SPOT` and `SPOTMisc` from the statistical programming environment R, hence providing a HPT environment that is fully accessible from R. Although HPT can be performed with R functions, an underly-

ing PYTHON environment has to be installed. This installation is explained in the Appendix.

The first three case studies in this book are concluded with a global comparison of the seven methods, i.e., six ML methods and one DL method. The main goal of these studies was to analyze whether a relatively short HPT run, which is performed on a notebook or desktop computer without High Performance Computing (HPC) hardware, can improve the performance. Or, stated differently:

> Is it worth doing a short HPT run before doing a longer study?

To illustrate the performance gain (tunability), a final comparison of the seven methods will be presented. The number of repeats will be determined first:

An approximate formula for sample size determination will be used. The reader is referred to Sect. 5.6.5 and to Senn (2021) for details. A sample size of 30 experiments was chosen, i.e., altogether 210 runs were performed.

The list of results from the rfunctionspot HPT run stores relevant information about the configuration and the experimental results.

Violin plots (Fig. 10.14) can be used. These observations are based on data collected from default and tuned parameter settings. Although the absolute best value was found by Extreme Gradient Boosting (XGBoost), Support Vector Machine (SVM) should be considered as well, because the performance is similar while the variance is much lower. This study briefly explained how HPT can be used as a datascope for the optimization of DNN hyperparameters. The results from this brief study scratch on the surface of the HPT set of tools. Especially for DL, `SPOT` allows recommendations for improvement, it provides tools for comparisons using different losses and measures on different data sets, e.g., $\psi^{(\mathrm{train})}$, $\psi^{(\mathrm{val})}$, and $\psi^{(\mathrm{test})}$.

While discussing the hyperparameter tuning results, HPT does not search for the final, best solution only. For sure, the hyperparameter practitioner is interested in the best solution. But even from this *greedy* point of view, considering the *route to the solution* is also of great importance, because analyzing this route enables *learning* and can be much more efficient in the long run compared to a greedy strategy.

Example: Route to the solution

Consider a classification task that has to be performed several times in a different context with similar data. Instead of blindly (automatically) running the Hyperparameter Optimization (HPO) procedure individually for each classification task (which might also require a significant amount of time and resources, even when it is performed automatically) a few HPT procedures are performed. Insights gained from HPT might help to avoid ill specified parameter ranges, too short run times, and further pitfalls.

In addition to an effective and efficient way to determine the optimal hyperparameters, SPOT provides means for understanding algorithms' performance (we will use

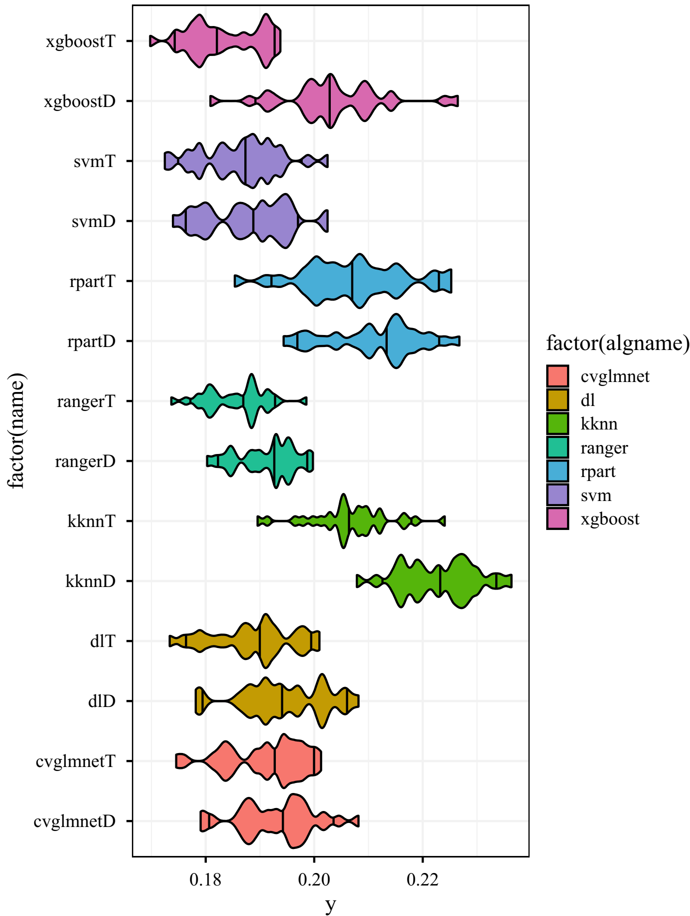

Fig. 10.14 Comparison of ML algorithms with default (D) and tuned (T) hyperparameters. Classification error (MMCE). Note: because there is no "default" hyperparameter setting for the deep learning models used in this study, we have chosen a setting based on our experience and recommendations from the literature, see the discussion in Sect. 10.6

datascopes similar to microscopes in biology and telescopes in astronomy). Considering the research goals stated in Sect. 4.1, the HPT approach presented in this study provides many tools and solutions.

To conclude this chapter, in addition to the research goals (R-1) to (R-8) from Sect. 4.1, important goals that are specific for HPT in DNN are presented.

The selection of an adequate performance measure is relevant. Kedziora et al. (2020) claimed that "research strands into ML performance evaluation remain arguably disorganized, [. . .]. Typical ML benchmarks focus on minimizing both loss functions and processing times, which do not necessarily encapsulate the entirety of human requirement." Furthermore, a sound test problem specification is neces-

sary, i.e., train, validation, and test sets should be clearly specified. Importantly, the initialization (this is similar to the specification of starting points in optimization) procedures should be made transparent. Because DL methods require a large amount of computational resources, the usage of surrogate benchmarks should be considered (this is similar to the use of Computational Fluid Dynamics (CFD) simulations in optimization). Most of the ML and DL methods are noisy. Therefore, repeats should be considered. The power of the test, severity, and related tools which were introduced in Chap. 5 can give hints for choosing adequate values, i.e., how many runs are feasible or necessary. The determination of meaningful differences—with respect to the specification of the loss function or the accuracy—based on tools like severity are of great relevance for the practical application. Remember: scientific relevance is not identical to statistical significance. Furthermore, floor and ceiling effects should be avoided, i.e., the comparison should not be based on too hard (or too easy) problems. We strongly recommend a comparison to baseline (e.g., default settings or Random Search (RS)).

The model $\mathcal{A}$ must be clearly specified, i.e., the initialization, pre-training (starting points in optimization) should be explained. The hyperparameter (ranges, types) should be clearly specified. If there are any additional (untunable) parameters, then they should be explained. How is reproducibility ensured (and by whom)? Last but not least: open source code and open data should be provided.

The final conclusion from the three case studies (Chaps. 8–10) can be formulated as follows:

> HPT provides tools for comparing, analyzing, and selecting an adequate ML or DL method for unknown real-world problems. It requires only moderate computational resources (notebooks or desktop computers) and limited time. Practitioners can start HPT runs at the end of their work day and will find the results ready on their desk the next morning.

10.10 Program Code

Program Code

```r
runNr <- "000"
batch_size <- 16
prop <- 2 / 3
dfGeneric <- getDataCensus(target = target, nobs = 1000)
# dfGeneric <- MASS::Boston
# names(dfGeneric)[names(dfGeneric) == "medv"] <- "target"
```

```
data <- getGenericTrainValTestData(dfGeneric = dfGeneric, prop = prop)
specList <- genericDataPrep(data = data, batch_size = batch_size)
## model configuration:
model <- "dl"
cfg <- getModelConf(list(model = model))
x <- matrix(cfg$default, nrow = 1)
#'
kerasConf <- getKerasConf()
kerasConf$nClasses <- 1
kerasConf$activation <- NULL
kerasConf$verbose <- 0
kerasConf$loss <- "mse"
kerasConf$metrics <- "mae"
##  Only some variables are tuned
# kerasConf$active <-  c("layers", "units", "epochs")
### First example: simple function call:
message("objectiveFunctionEvaluation(): x before transformX().")
print(x)
if (length(cfg$transformations) > 0) {
  x <- transformX(xNat = x, fn = cfg$transformations)
}
message("objectiveFunctionEvaluation(): x after transformX().")
print(x)
funKerasGeneric(x, kerasConf = kerasConf, specList = specList)
#'
### Second example: evaluation of several (three) hyperparameter settings:
xxx <- rbind(x, x, x)
funKerasGeneric(xxx, kerasConf = kerasConf, specList)
#'
### Third example: spot call
kerasConf$verbose <- 0
result <-
  spot(
    x = NULL,
    fun = funKerasGeneric,
    lower = cfg$lower,
    upper = cfg$upper,
    control = list(
      funEvals = 25,
      # time = list(maxTime = 5),
      noise = TRUE,
      types = cfg$type,
      plots = TRUE,
      progress = TRUE,
      seedFun = 1,
      seedSPOT = 1,
      replicates = 2,
      OCBA = TRUE,
      OCBABudget = 2,
      parNames = cfg$tunepars,
      designControl = list(
        replicates = 2,
        size = 1 * length(cfg$lower)
      ),
      yImputation = list(
        handleNAsMethod = handleNAsMean,
        imputeCriteriaFuns = list(is.infinite, is.na, is.nan),
        penaltyImputation = 3
      ),
```

```
      modelControl = list(
        target = "ei",
        useLambda = TRUE,
        reinterpolate = FALSE
      ),
      transformFun = cfg$transformations
    ),
    kerasConf = kerasConf,
    specList = specList
  )
x <- result$xbest
message("objectiveFunctionEvaluation(): x before transformX().")
print(x)
if (length(cfg$transformations) > 0) {
  x <- transformX(xNat = x, fn = cfg$transformations)
}
message("objectiveFunctionEvaluation(): x after transformX().")
print(x)
df <- data.frame(x)
names(df) <- cfg$tunepars
print(df)
save(result, file = paste0(model, runNr, ".RData"))
```

```
dfRun <- prepareProgressPlot(model, runNr, directory = ".")
ggplotProgress(dfRun)
```

```
library("rpart")
library("rpart.plot")
library("SPOT")
x <- result$x
# cfg <- getModelConf(model="dl")
transformFun <- cfg$transformations
message("predDlCensus(): x before transformX().")
print(x)
if (length(cfg$transformations) > 0) {
  x <- transformX(xNat = x, fn = cfg$transformations)
}
message("predDlCensus(): x after transformX().")
print(xt)
fitTree <- buildTreeModel(
  x = xt,
  y = result$y,
  control = list(xnames = result$control$parNames)
)
rpart.plot(fitTree$fit)
```

```
kerasConf$returnObject <- "pred"
x <- result$xbest
if (length(cfg$transformations) > 0) {
  x <- transformX(xNat = x, fn = cfg$transformations)
}
x
evalKerasGeneric(
```

```
  x = x,
  kerasConf = kerasConf,
  specList = specList
)
```

```
library("SPOT")
library("SPOTMisc")
runNr <- "OCBA"
batch_size <- 32
prop <- 2 / 3
target <- "age"
dfGeneric <- getDataCensus(target = target, nobs = 1e4)
# dfGeneric <- MASS::Boston
# names(dfGeneric)[names(dfGeneric) == "medv"] <- "target"
data <- getGenericTrainValTestData(dfGeneric = dfGeneric, prop = prop)
specList <- genericDataPrep(data = data, batch_size = batch_size)
## model configuration:
model <- "dl"
cfg <- getModelConf(list(model = model))
# x <- matrix(cfg$default, nrow=1)
x <- result$xBestOcba
#'
kerasConf <- getKerasConf()
# kerasConf$nClasses <- 1
# kerasConf$activation <- NULL
kerasConf$verbose <- 2
# kerasConf$loss <- "mse"
# kerasConf$metrics <- "mae"
##  Only some variables are tuned
# kerasConf$active <-  c("layers", "units", "epochs")
### First example: simple function call:
message("objectiveFunctionEvaluation(): x before transformX().")
print(x)
if (length(cfg$transformations) > 0) {
  x <- transformX(xNat = x, fn = cfg$transformations)
}
message("objectiveFunctionEvaluation(): x after transformX().")
print(x)
evalKerasGeneric(
  x = x,
  kerasConf = kerasConf,
  specList = specList
)
```

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.