# deepregression: A Flexible Neural Network Framework for Semi-Structured Deep Distributional Regression

**David Rügamer** (ID)
LMU Munich

**Chris Kolb**
LMU Munich

**Cornelius Fritz** (ID)
LMU Munich

**Florian Pfisterer** (ID)
LMU Munich

**Philipp Kopper** (ID)
LMU Munich

**Bernd Bischl** (ID)
LMU Munich

**Ruolin Shen**
Helmholtz AI Munich

**Christina Bukas** (ID)
Helmholtz AI Munich

**Lisa Barros de Andrade e Sousa** (ID)
Helmholtz AI Munich

**Dominik Thalmeier**
Helmholtz AI Munich

**Philipp F. M. Baumann** (ID)
ETH Zurich

**Lucas Kook** (ID)
University of Zurich

**Nadja Klein** (ID)
Humboldt-Universität
zu Berlin

**Christian L. Müller**
LMU Munich, ICB,
Helmholtz Zentrum Munich, CCM,
Flatiron Institute, New York

### Abstract

In this paper we describe the implementation of semi-structured deep distributional regression, a flexible framework to learn conditional distributions based on the combination of additive regression models and deep networks. Our implementation encompasses (1) a modular neural network building system based on the deep learning library **TensorFlow** for the fusion of various statistical and deep learning approaches, (2) an orthogonalization cell to allow for an interpretable combination of different subnetworks, as well as (3) pre-processing steps necessary to set up such models. The software package allows to define models in a user-friendly manner via a formula interface that is inspired by classical statistical model frameworks such as **mgcv**. The package's modular design and functionality provides a unique resource for both scalable estimation of complex statistical models and the combination of approaches from deep learning and statistics. This allows for state-of-the-art predictive performance while simultaneously retaining the indispensable interpretability of classical statistical models.

*Keywords*: additive predictors, deep learning, effect decomposition, orthogonal complement, penalization, smoothing.

# 1. Introduction

In regression analysis, the main objective is to obtain information about the conditional distribution of a response variable given a set of explanatory variables. Nonetheless, many classical regression approaches only model the mean of this conditional distribution. These mean regression models often cannot adequately account for heteroscedasticity of the conditional distribution or its possible dependence on explanatory variables (or features). As a result, methodological research has brought forward a number of distinct approaches, often referred to as *distribution*, *distributional* or *density regression*. The common ground of all these contributions is to allow the complete set of parameters of the conditional distribution to depend on the available features, not only the mean. Common examples include extensions of generalized additive models (GAMs; see, e.g., Wood 2017) by making all parameters of a parametric distribution feature-dependent (e.g., generalized additive models for location, scale, and shape; GAMLSS; Rigby and Stasinopoulos 2005), but also approaches avoiding the parametric distributional assumptions such as Bayesian nonparametrics (Dunson 2010), mixtures of experts (Jacobs, Jordan, Nowlan, and Hinton 1991), or conditional transformation models (Hothorn, Kneib, and Bühlmann 2014). Another research branch focuses on directly relating local properties of the conditional distribution, such as quantiles and expectiles, to features (see, e.g., Koenker 2005; Schnabel and Eilers 2009).

Despite being more flexible than mean regression approaches, many of these distributional regression methods are not scalable to either a large number of observations or a large number of features. While these large data settings might already be challenging for mean regression approaches, distributional regression usually involves additional model terms and thus increased computational complexity. The semi-structured deep distributional regression (SDDR) framework proposed by Rügamer, Kolb, and Klein (2020a) and implemented in the package **deepregression** overcomes these limitations by casting the distributional regression task in a neural network. The package allows the user to learn the distributional parameters of a wide range of parametric densities based on a combination of structured additive regression models and deep neural networks. **deepregression** thereby offers a scalable and flexible alternative to existing distributional regression approaches. A Python-native version of **deepregression** is implemented in a separate software package **PySDDR** (`https://github.com/HelmholtzAI-Consultants-Munich/PySDDR`). The R package **deepregression** (Rügamer 2023) is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=deepregression`.

## 1.1. Semi-structured deep distributional regression

SDDR is based on ideas from distributional regression for parametric distributions (see, e.g., Klein, Kneib, Lang, and Sohn 2015), to estimate the entire conditional distribution of a continuous, discrete, mixed, or multivariate response variable $Y$. This is done through modelling the corresponding distributional parameters. Apart from its application to transformation models (Baumann, Hothorn, and Rügamer 2021; Rügamer, Baumann, Kneib, and Hothorn 2022), SDDR has also been applied recently to survival analysis (Kopper, Pölsterl, Wachinger, Bischl, Bender, and Rügamer 2021; Kopper, Wiegrebe, Bischl, Bender, and Rügamer 2022), mixture models (Rügamer, Pfisterer, and Bischl 2020b), and epidemiological models combined with graph neural networks (Fritz, Dorigatti, and Rügamer 2022).

In particular, SDDR constitutes a distributional learning framework in which each distributional parameter $\theta_k \equiv \theta_k(\eta_k), k = 1, \ldots, K$ of an arbitrary pre-specified parametric distribution $\mathcal{D}(\theta_1, \ldots, \theta_K)$ can be modeled through (potentially different) additive predictors $\eta_k$.[1] Each of these predictors is defined as a sum of structured predictors, that can include (penalized) linear effects of features $\boldsymbol{x}$, (penalized) smooth effects of features $\boldsymbol{z}$, or arbitrary additional (unstructured) neural network predictors for features $\boldsymbol{u}$:

$$\eta_k \equiv \eta_k(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{u}) = \boldsymbol{x}^\top \boldsymbol{w} + \sum_{j=1}^{J} f_{k,j}(\boldsymbol{z}) + \sum_{l=1}^{L} d_{k,l}(\boldsymbol{u}). \qquad (1)$$

The different features $\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{z}$ are also allowed to overlap, giving rise to a generic additive identifiability problem that is handled by our later introduced orthogonalization cell. In (1), the first summand on the right-hand side represents the linear effects of features $\boldsymbol{x}$ with unknown weights $\boldsymbol{w}$, the second part comprises smooth effects $f_{k,j}(\boldsymbol{z})$ of one or several features in $\boldsymbol{z}$, and the last part is a sum of (deep) neural networks depending on $\boldsymbol{u}$. The smooth functions $f_{k,j}$ are defined as linear combinations of appropriate basis functions $B_1, \ldots, B_M$ evaluated at the feature value(s), such as B-spline basis functions (see Wood 2017, for an overview). Consecutively, $\eta_k, k = 1, \ldots, K$ is transformed by a one-to-one mapping $h_k : \mathbb{R} \to \Theta_k$ from the real line to the respective parameter space $\Theta_k$ of $\theta_k$ (also known as response functions). For example, choosing $h_k = \exp(\cdot)$ is often used to model a positive scale parameter such as the variance of a Gaussian random variable. In summary, the structural and stochastic components of the model are defined by:

$$\boldsymbol{Y} \mid \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{u} \sim \mathcal{D}(\theta_1, \ldots, \theta_K), \qquad (2)$$

with the following transformations:

$$(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{u}) \xrightarrow{(1)} \eta_k(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{u}) \xrightarrow{h_k} \theta_k(\eta_k) \longrightarrow \mathcal{D}(\theta_1, \ldots, \theta_k, \ldots, \theta_K).$$

To ensure identifiability and interpretability in the additive predictors, the SDDR framework uses an orthogonalization cell that allows to separate effects in the structured part from the unstructured part of the predictors. This only requires the neural network predictors $d_l$ to have a fully-connected layer as penultimate layer (see Figure 7).

The SDDR framework encompasses many special cases such as linear and generalized linear models, GAMs, and GAMLSS, but also extends those classes to include additional neural network predictors. To allow practitioners to make use of such a flexible framework, **deepregression** implements the SDDR framework in an accessible and modular manner in the software environment R (R Core Team 2022). While various other packages for distributional regression models exist such as **gamlss** (Stasinopoulos and Rigby 2007), **gamlss.lasso** (Ziel and Muniain 2021) or **bamlss** (Umlauf, Klein, Simon, and Zeileis 2021), to the best of our knowledge no software package exists that can arbitrarily combine structured regression models with deep neural networks and thus allow for applications with multimodal data (which is the conventional term in the machine and deep learning community to refer to data sets with different modalities, such as tabular and image data). Before going into the details of our package, we introduce our running example used throughout this tutorial.

---

[1]It is assumed that $\mathcal{D}$ has an absolute continuous density with existing finite first derivatives with respect to all distributional parameters.

## 1.2. Case study: Airbnb price listing data

Airbnb is a company providing an online marketplace for rental apartments. In this example, our goal is to construct a model to predict the listing price of apartments with all available information. From the website `http://insideairbnb.com/`, we obtain all current listings of apartments on the Airbnb website. The provided information includes various data modalities, ranging from numeric variables, such as latitude and longitude, integer variables, such as the number of bedrooms, textual information, like a room description, dates, to a picture of each property. This breadth of different data types makes the dataset an ideal candidate for the application of **deepregression**. The final model could then, e.g., be deployed as a guide for new costumers wanting to rent an apartment in order to help setting a reasonable price for their own advertisement. In order to reduce computational complexity, we will focus on apartments in Munich (Germany) and show example code for demonstration purposes.

The cleaned `airbnb` data from Munich consist of $3,504$ apartment listings. In order to demonstrate the functionality of our package, we will focus on a handful of features, namely: `description` (a textual description of the property), `host_since` (a date variable describing the host's membership duration), `longitude` and `latitude`, `review_score_value` (a review score), `host_response_time` (the time until the host response to inquiries), and the properties' images (which are named according to the `id` variable in the dataset). The response variable is the price of the respective apartment `price`.

```
R> airbnb <- readRDS("munich.RDS")
R> airbnb$days_since_last_review <- as.numeric(
+    difftime(airbnb$date, airbnb$last_review))
```

The remainder of this paper is structured as follows: Section 2 details the core functions and building blocks in the corresponding R package. A brief summary and conclusion is given in the final Section 3.

# 2. The package

**deepregression** is constructed in a modular fashion, allowing to flexibly specify a large number of models as a sequence of processing steps, which are illustrated in Figure 1. In the first step, the provided formulas defining all additive predictors are parsed using `processor` functions. Every formula term (linear term, smooth term, ...) is handled by its respective `processor`. The processed terms each have their own methods for providing data for model fitting and prediction, as well as functions to return the coefficients and a plot of the partial effect if appropriate. The parsed formulas containing all these processed terms are returned to **deepregression**. A second, larger processing step constructs the subnetworks for each additive predictor $\eta_k, k = 1, \ldots, K$. These subnetworks are finally passed to a model builder function that combines all additive predictors into a model. Per default, model building and fitting is done using **keras** (Allaire and Chollet 2022), but **deepregression** allows to replace this estimation engine by another optimization routine, e.g., using lower-level **TensorFlow** (Abadi *et al.* 2015) functions. Note that the results presented in this paper might slightly change when running code examples with a different version of R, Python, **TensorFlow** or **Keras**, and also depend on the operating system. The results in this paper were generated using the R version 4.2.1 on a x86_64-pc-linux-gnu (64-bit) platform under Ubuntu 22.04.1 LTS with
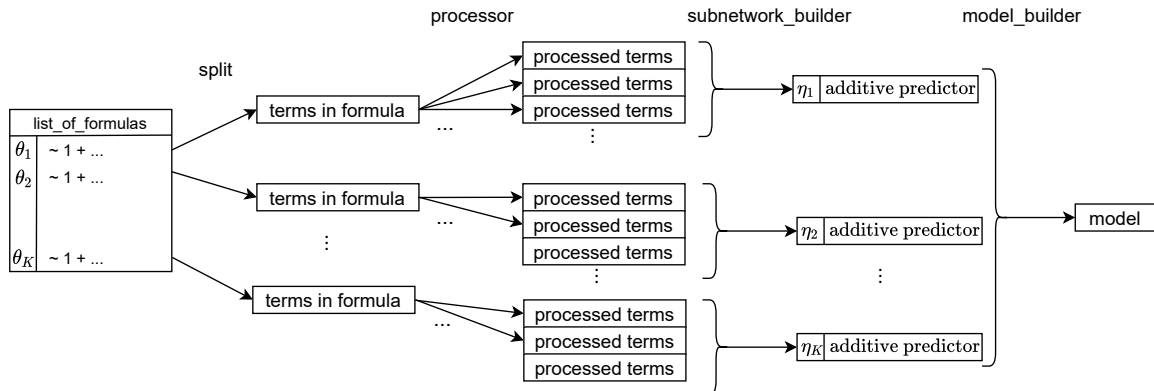
Figure 1: Model building steps in the package. The list of formulas provided by the user is first split up and each additive term is processed by its respective processor. Thereafter, all processed terms of one formula entry are combined to an additive predictor using the a subnetwork builder function. Results are then combined to form the final model using the a model builder function.

**BLAS**/**LAPACK** version 3.10.0, Python 3.10, **TensorFlow** version 2.10.0, **Keras** version 2.10.0, and **TensorFlow Probability** version 0.16.

## 2.1. Core functionality

**deepregression**'s core functionality is to define and fit (deep) distributional regression models. In the package, various parametric response distributions can be defined, and their parameters structurally related to the three different additive predictor types defined in (1). For example, a three-parameter distribution, such as the location, scale, and shape-parameterized $t$ distribution, can have different predictors defined for each of the $K = 3$ parameters. Here, the scale and shape parameter could be nuisance parameters that are modeled using a deep neural network (DNN) of all features, while the location parameter is assumed to have specific (non-)linear effects in certain features and is thus modeled in a structured manner. From a network perspective, the main building blocks are (i) a structured linear layer, (ii) a structured nonlinear layer, (iii) distributional layer(s), and (iv) an orthogonalization cell. Building blocks (i) – (iii) are visualized in Figure 2 and (iv) will be discussed in more detail later. The distributional layer, illustrated in Figure 2(c), defines the final distribution using one or more subnetworks for each parameter $\theta_k$. The network is then trained using the negative log-likelihood resulting from (2) as a loss function. In each subnetwork, the additive predictors $\eta_k$ can be defined using (penalized) linear effects, Figure 2(a), (penalized) smooth effects, Figure 2(b), or arbitrary deep neural networks.

Furthermore, the distributional layers can be combined to define a richer distribution family, e.g., to create zero-inflated distributions or mixtures of distributions.

While the structured linear layer can take inputs in their original form, the structured nonlinear layer requires additional pre-processing which is described below as part of the three core functionalities. The DNNs for the unstructured effects $d_{k,l}(\boldsymbol{u})$ have to be user-specified to correctly process the inputs.
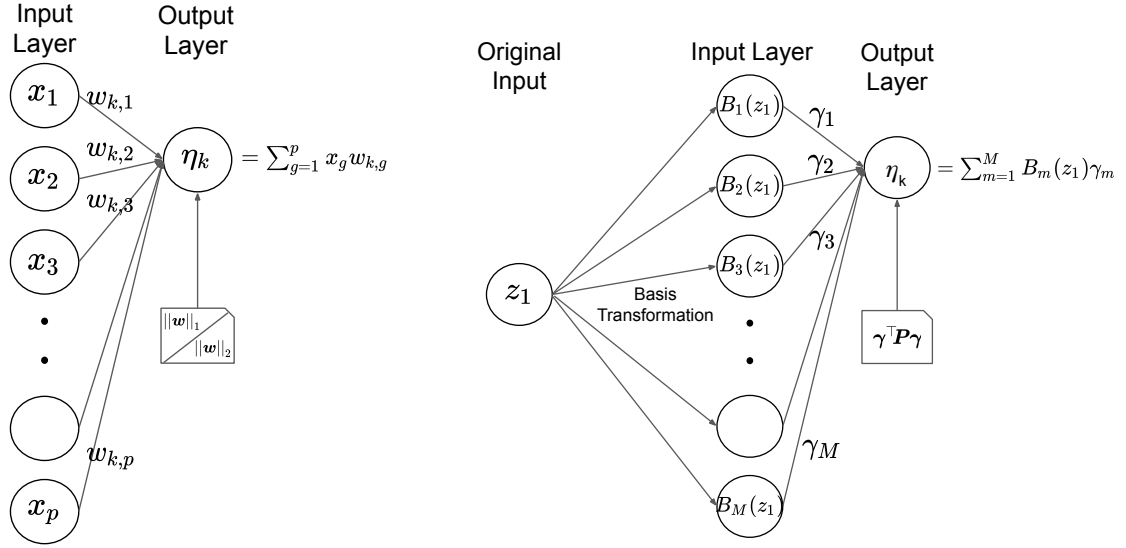
(a) A structured linear network: Linear combination of input features $x_j$ with optional $L_1$ or $L_2$ penalty on the weights $w_{k,j}$

(b) A structured nonlinear network: Linear combination of (basis-transformed) features $B_m(z_1)$ via $M$ basis functions $B_m, m = 1, \ldots, M$ with optional matrix-based quadratic penalty $\boldsymbol{P}$ on the weights $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_M)^\top$.
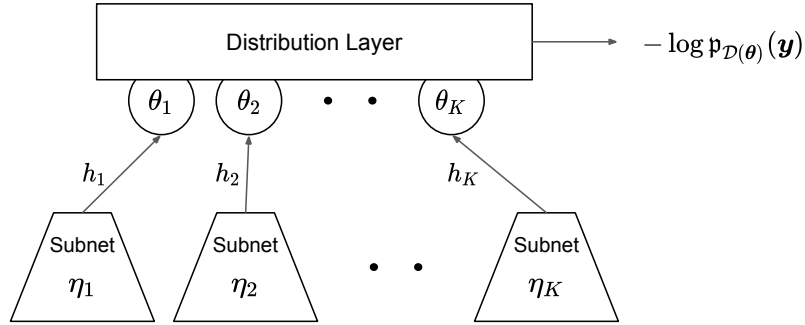


(c) A distributional layer learned by $K$ subnetworks: each parameter $\theta_k$ of $\mathcal{D}$ with density $\mathfrak{p}_\mathcal{D}$ can be learned using (a), (b) and/or a DNN.

Figure 2: Illustration of network components that can be combined using the package. DNNs are left out as these can be arbitrarily specified and combined with additive predictors as long as the penultimate layer in the DNN corresponds to a fully-connected layer.

## 2.2. Main components

*Formula interface*

In the formula interface, each distributional parameter can be specified by a symbolic additive predictor. A model with two parameters $\theta_1, \theta_2$ for location and scale of a distribution $\mathcal{D}$ with $\theta_1 = w_0 + w_1 x + f_{1,1}(z_1)$ and $\theta_2 = f_{2,1}(z_2) + d_{2,1}(\boldsymbol{u})$ in R could translate to:

|  | Notation | Arguments |
|---|---|---|
| Intercept | `1` | – |
| Linear term | `x` or `lin(x)` | – |
| Ridge-penalized linear effects | `ridge(x, la = ...)` | penalty `la` |
| Lasso-penalized linear effects | `lasso(x, la = ...)` | penalty `la` |
| Offset | `offset(x)` | – |
| (Tensor-product) Splines | `s/te/ti(x, ..., df = ...)` | degrees-of-freedom `df`; **mcgv** arguments |

Table 1: Processors for formula terms provided in the package.

```
R> list_of_formulas <- list(
+    loc = ~ 1 + x + s(z1, bs = "tp"),
+    scale = ~ 0 + s(z2, bs = "ps") + dnn(u)
+  )
```

This code snippet specifies an intercept (also called bias term) for the location parameter, a linear effect for $x$, and a thin-plate regression spline (Wood 2003) for $z_1$. The scale parameter is specified without bias term, but using another nonlinear effect of feature $z_2$, now characterized by a penalized spline. It further contains a DNN predictor for inputs $\boldsymbol{u}$, where `dnn` is a model defined separately with suitable input shape. In the network, this formula will create two subnetworks, both with one output unit (the predictors $\eta_k, k = 1, 2$ for the location and scale parameters), which are then used to define the distribution.

**Case study: Formula interface** We start with a simple additive model to predict the log-transformed price

```
R> y <- log(airbnb$price)
```

using a normal distribution parameterized by a location and scale parameter. We learn the location using an intercept and a tensor-product spline for `longitude` and `latitude`, while estimating the scale to be constant. In our package, this corresponds to

```
R> list_of_formulas <- list(
+    loc = ~ 1 + te(latitude, longitude, df = 5),
+    scale = ~ 1
+  )
```

*Processors*

As described in Figure 1, each additive term contained in predictor $\eta_k, k = 1, \ldots, K$ is handled by type-specific processors. **deepregression** comes with different processors that automatically parse each formula. These are listed in Table 1. It also allows the user to supply custom processors using the argument `additional_processors`.

*Network initialization*

After defining the model formulas for each distributional parameter, the SDDR model is initialized via the `deepregression` function. The required arguments are the response `y`, the

data frame or list `data` including all features, the distribution `family`, the `list_of_formulas`, and the `list_of_deep_models` to define the (deep) neural networks that are used to model unstructured effects.

**Case study: Network initialization**

```
R> mod_simple <- deepregression(y = y, data = airbnb,
+    list_of_formulas = list_of_formulas, list_of_deep_models = NULL)
```

The result `mod_simple` is of class 'deepregression', a list of length three, where the first entry `model` is a `keras` model, the second element `init_params` is a list of objects required for model fitting, tuning, plotting, and other post-processing functionalities and the third element `fit_fun` encompassing the fitting function needed for model fitting. The `model` element contains the references to the respective Python objects and is independent of the actual data. Since `keras` models do not store any data, the second list item `init_params` is returned for convenience to give the user direct access to the pre-processed data and to simplify calls to other methods such as `fit`.

```
R> class(mod_simple$model)
```

```
[1] "keras.engine.functional.Functional"
[2] "keras.engine.training.Model"
[3] "keras.engine.base_layer.Layer"
[4] "tensorflow.python.module.module.Module"
[5] "tensorflow.python.training.tracking.tracking.AutoTrackable"
[6] "tensorflow.python.training.tracking.base.Trackable"
[7] "keras.utils.version_utils.LayerVersionSelector"
[8] "keras.utils.version_utils.ModelVersionSelector"
[9] "python.builtin.object"
```

*Pre-processing for structured nonlinear layers*

Structured nonlinear effects can be fitted in **deepregression** by defining a corresponding pre-processing function that generates a design matrix based on (user-defined) basis functions and a penalty matrix that is used to regularize the smoothness of the estimated nonlinear functions. In R, these objects are computed by calling the function `smooth.construct` from the package **mgcv** (Wood 2001).

**Case study: Pre-processing**   In the above defined model `mod`, the smooth terms from **mgcv** are stored in the `init_params` element and are used for model fitting, prediction, and plotting.

```
R> sapply(mod_simple$init_params$parsed_formulas_contents$loc, "[[", "term")
```

```
[1] "te(latitude, longitude, df = 5)" "(Intercept)"
```

*Specification of the family*

The `family` argument specifies the respective family to be learned. Possible choices can be searched for in `?dr_families`, comprising over 30 different options, including those from an exponential family (such as Gaussian, Bernoulli, Poisson, beta, gamma, and multinomial distribution).

**deepregression** can also define custom distributions. The add-on R package **mixdistreg** (Rügamer *et al.* 2020b) further allows to fit mixtures of families, as proposed by Rügamer *et al.* (2020b). Details can be found in the Section 2.7.

*Formula contents*

The formulas defining each distributional parameter can be specified in the same way as in `mgcv::gam`, including `s`-terms, `ti`- and `te`-terms that can be further customized by using different basis functions, penalty terms, basis dimensions or knot placement (see `?mgcv::s`). Factor variables in the formula are also treated the same way as in conventional regression packages by automatically transforming them to dummy-coded features. The exclusion of the intercept in a linear predictor can be defined as per usual by adding the `0` or `-1` to the formula. Table 1 lists all processors for structured terms that are supported by default.

*Specification of the DNNs*

The DNNs specified in the `list_of_formulas` must also be passed as named list elements in the `list_of_deep_models`. The named list `list_of_deep_models` contains a list of DNNs, each specified as a function using the `keras` layers application programming interface. The respective functions take as many inputs as defined for the neural network and have a fully-connected output layer containing one hidden unit (or any other layer that results in the appropriate amount of outputs as discussed later). For instance, an example DNN for the features $\boldsymbol{u}$ from the previous example can be specified using the pipe `%>%` notation as follows:

```
R> deep_model <- function(x) {
+      x %>%
+      layer_dense(units = 5, activation = "relu", use_bias = FALSE) %>%
+      layer_dropout(rate = 0.2) %>%
+      layer_dense(units = 3, activation = "relu") %>%
+      layer_dropout(rate = 0.2) %>%
+      layer_dense(units = 1, activation = "linear")
+  }
```

`deep_model` defines a two hidden-layer DNN with 20% dropout between each layer and rectified linear unit (ReLU) activation functions. During training the DNN randomly masks 20% of the weights for every observation for these layers due to the specified dropout in order to regularize the network. Since the DNN's output enters the real-valued predictor $\eta_k$ additively, it is typically most sensible to use a `"linear"` activation function in the output layer for maximum flexibility in the values $\eta_k$. To ensure identifiability of the structured model predictor when structured regression terms and a DNN share some input features, an orthogonalization cell is automatically included before the last dense layer. In this case, it is required to use a `"linear"` activation function as in this example.

**Case study: Formulas and DNNs**   The following code shows an illustrative specification including possible formula terms. We only define the `location` using an intercept, categorical effects for beds `beds`, two splines (a P-spline and thin-plate regression spline) for the number of guests `accommodates` and days since last review `days_since_last_review`, as well as a DNN for the review score `review_scores_rating` and the number of reviews per month `reviews_per_month`. To make the intercept identifiable, we set an additional option that is explained in detail in Section 2.6.3.

```
R> options(identify_intercept = TRUE)
R> mod <- deepregression(y = y, data = data, list_of_formulas = list(
+     location = ~ 1 + beds + s(accommodates, bs = "ps") +
+       s(days_since_last_review, bs = "tp") +
+       deep(review_scores_rating, reviews_per_month),
+     scale = ~ 1),
+   list_of_deep_models = list(deep = deep_model))
```

## 2.3. Model training and tuning

### *Model fitting*

Once the model `mod` has been set up, the neural network can be trained using the `fit` function. Per default, the `fit` function is a wrapper of the corresponding `fit` function for `keras` models and inherits the `keras::fit` arguments. More specifically,

- `epochs` to specify the number of iterations,

- `callbacks` to specify information that is called after each epoch or batch (used, e.g., for early stopping),

- `validation_split` to specify the share of data (in $[0, 1)$) that is used to validate the model (while the rest is used to train the model),

- `validation_data` to optionally specify any predefined validation data,

- `verbose` to define if progress is printed in the console,

- `view_metrics` to activate an interactive plot window when using RStudio.

Several other arguments are available, such as a logical argument `early_stopping` to activate early stopping and `patience` to define the patience used in early stopping.

**Case study: Model fitting**

```
R> mod %>% fit(epochs = 100, verbose = FALSE, view_metrics = FALSE,
+     validation_split = 0.2)
```

To inspect the fitted model, we can either use `coef` or `plot` for structured effects, as will be demonstrated later, or extract the fitted values of the model using `fitted` (which yields the prediction on the training data):
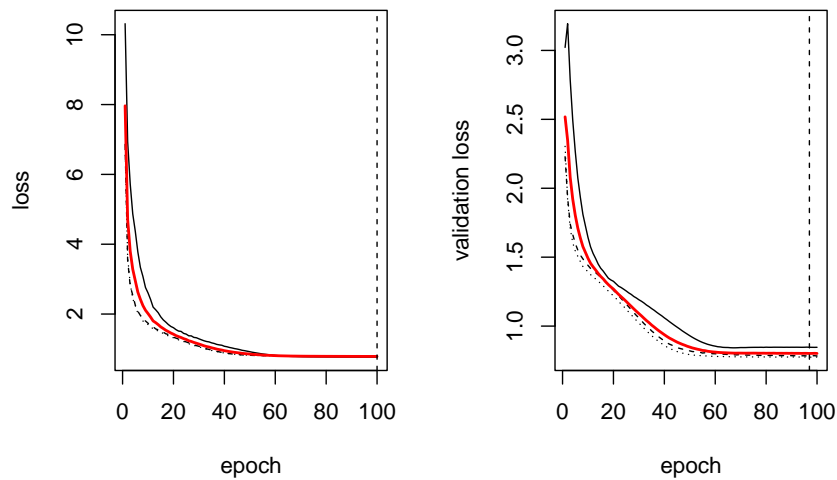
Figure 3: Visualization of cross-validation result. Horizontal black lines indicate the different folds over different epochs ($x$ axis) and the corresponding loss (left) or validation loss (right). The red line shows the mean loss and validation loss over all iterations. The dashed vertical line indicates the epochs with the lowest average loss and validation loss in the respective plots.

```
R> fitted_vals <- mod %>%
+    fitted()
R> cor(fitted_vals, y)

         [,1]
[1,] 0.5782093
```

*Model tuning*

In a similar manner as `fit`, **deepregression** offers a cross-validation function `cv` for model tuning. This can be used to fine-tune the model by, e.g., changing the specifications of the formula(s), changing the DNN structure, or determining the amount of smoothness (using the `df` argument in `deepregression`; see the next section for details). Folds in the `cv` function can be specified using the `cv_folds` argument. This argument takes either an integer for the number of folds or a list where each entry is again a list of two, one element with data indices for training and one with indices for testing. The procedure then uses these folds to refit the model the amount of times specified by the integer or the length of the fold list. The `patience` (number of epochs with no improvement after which training will be stopped) for early stopping can be set using the respective argument.

**Case study: Cross-validation**   As an example we use 100 iterations and three folds. The results of the cross-validation can be plotted using `plot_cv` and is visualized in Figure 3.

```
R> mod_cv <- deepregression(y = y, data = data,
+    list_of_formulas = list(
+       location = ~ 1 + beds + s(accommodates, bs = "ps") +
```

```
+         s(days_since_last_review, bs = "tp") +
+         deep(review_scores_rating, reviews_per_month),
+      scale = ~ 1),
+    list_of_deep_models = list(deep = deep_model))
R> res_cv <- mod_cv %>%
+    cv(plot = FALSE, cv_folds = 3, epochs = 100)
R> plot_cv(res_cv)
```

### 2.4. Methods overview

Apart from `fit` and `cv`, there are additional methods that can be applied to 'deepregression'
objects:

- `coef`: The `coef` function extracts the coefficients (network weights) of all structured
  layers, i.e., coefficients of linear of smooth effects. Using the argument `type`, the user
  can specify what type of coefficients to return (possible choices: `"linear"` for linear
  effects and `"smooth"` for basis coefficients of the specified smooth terms). The argument
  `params` allows to select for which of the distributional parameters the coefficients should
  be returned.

- `plot`: The `plot` function can be applied to 'deepregression' objects to visualize the
  learned nonlinear effect curves for splines and tensor product splines. Using `which`, a
  specific effect can be selected using the corresponding integer in the structured part of
  the formula (if `NULL` all effects are plotted), while the integer given for `which_param` indi-
  cates which distributional parameter is chosen for plotting. Via the argument `only_data`
  `= FALSE`, the function can also be used to just return the data used for plotting.

- `predict`: The `predict` function works as for other `keras` models. It either just takes the
  model as input and returns the predictions (per default the distribution's expectation)
  for observations in the training dataset, or, when supplied with new data, produces the
  corresponding predictions for the new data. As `deepregression` learns a distribution
  and not only its mean, the user can choose what type of distribution characteristic is
  to be predicted via the argument `apply_fun`, chosen from the distribution methods
  supplied by **tfprobability** (Keydana 2022, per default `apply_fun = tfd_mean` predicts
  the mean of the fitted distribution).

- `mean, stddev and quant`: These functions are convenience functions that directly re-
  turn the mean, standard deviation, or certain quantiles of the learned distribution. All
  three functions work for the given training data as well as newly provided `data`. The
  `quant` function has an additional argument `probs` to provide the quantile(s) of interest.

- `get_distribution`: Instead of returning summary statistics, `get_distribution` re-
  turns the whole **TensorFlow** distribution with all its functionalities. The distribution
  can, e.g., be used to sample random numbers from it or compute the probability density
  function or cumulative distribution function.

- `log_score`: The `log_score` function directly returns the evaluated log-likelihood based
  on the estimated parameters and the provided data (including a response vector `this_y`).
  If `data` and `this_y` are not provided, the function will calculate the score on the training
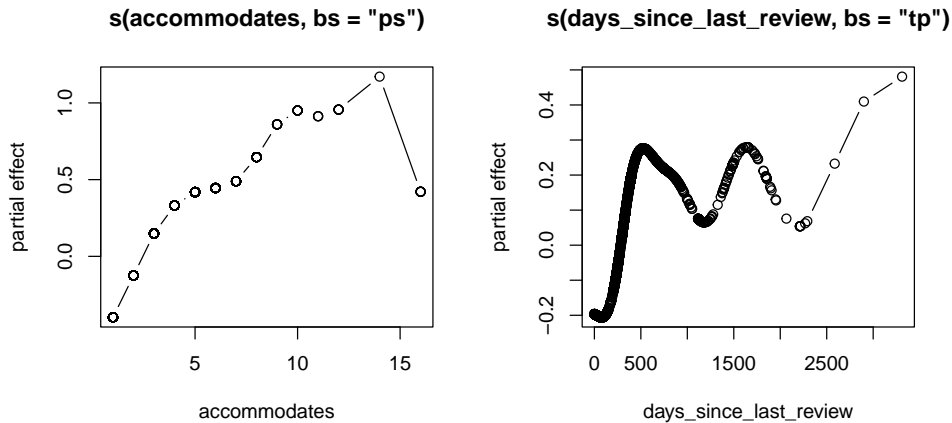  data.

Figure 4: Estimated partial effects of accommodates and the days since the last review on the average log-price of an apartment.

**Case study: Convenience functions** We first inspect the estimated linear effects.

```
R> coef(mod, type = "linear")

$beds
             [,1]
 [1,]  0.22579455
 [2,]  0.29432768
 [3,]  0.42152047
 [4,]  0.43564758
 [5,]  0.86451352
 [6,]  0.50364327
 [7,]  0.63329536
 [8,]  0.09199127
 [9,]  0.32188147
[10,]  0.91526628
[11,]  0.66855335
[12,] -0.48711517
[13,] -0.81626230
[14,]  1.35741544
[15,] -0.35289571
[16,] -0.41943353


$`(Intercept)`
         [,1]
[1,] 2.377732
```

For nonlinear effects, the estimated smooth functions can be plotted using `plot` with result visualized in Figure 4.

While most of the other convenience functions are wrappers that access the underlying `keras` model and convert the quantity of interest to an R object, `get_distribution` returns the fitted distribution, a Python `tfp.distributions`.

Figure 5: Visualization of the estimated distribution (black line) as well as the mean and quantiles (red vertical lines) for the first data point in the dataset.

```
R> dist <- mod %>%
+    get_distribution()
R> str(dist, 1)


tfp.distributions.Normal("model_1_distribution_lambda_1_Normal",
batch_shape=[3504, 1], event_shape=[], dtype=float32)
```

Instead of working with the original distribution object, the user can work with convenience functions to get the quantity of interests directly, as demonstrated below.

```
R> first_obs_airbnb <- as.data.frame(airbnb)[1,, drop = F]
R> dist1 <- mod %>%
+    get_distribution(first_obs_airbnb)
R> meanval <- mod %>%
+    mean(first_obs_airbnb) %>%
+    c
R> q05 <- mod %>%
+    quant(data = first_obs_airbnb, probs = 0.05) %>%
+    c
R> q95 <- mod %>%
+    quant(data = first_obs_airbnb, probs = 0.95) %>%
+    c
R> xseq <- seq(q05 - 1, q95 + 1, l = 1000)
R> plot(xseq, sapply(xseq, function(x) c(as.matrix(
+    dist1$tensor_distribution$prob(x)))),
+    type = "l", ylab = "density(price)", xlab = "price")
R> abline(v = c(q05, meanval, q95), col = "red", lty = 2)
```

Figure 5 shows the density of the estimated distribution together with estimated mean and quantiles.

### 2.5. Penalties

**deepregression** allows for different penalties, including $L_1$, $L_2$, and other smoothing penalties as provided by the **mgcv** package. While the latter is implicitly created when using `s-`, `ti-` or `te`-terms in the formula, the $L_1$ and $L_2$ penalties are used to penalize linear effects by defining the amount of penalization via the `la` argument in the model terms `lasso(...)` (using ideas from Hoff 2017; Tibshirani 2021, to allow for feature sparsity when using stochastic gradient optimization routines) and `ridge(...)`. Since the model object returned by `deepregression` is a list where the first element is a `keras` model, additional custom penalties can be specified. This is implemented by the `additional_penalty` argument, which requires a function with one (unused) dummy argument and defines the penalty based on the `keras` model's `trainable_weights`. Due to mini-batch training, the scaling and influence of penalties is not invariant with respect to the data size. Per default, **deepregression** therefore defines a scaling value `sp_scale` that multiplies the penalty by the inverse of the training data size. This behavior an be adapted through the `penalty_options`.

**Case study: Additional penalties** To access the model's weights, we can access the 'keras' object in the previously created `deepregression` model `mod_simple`, use `mod_simple$model$trainable_weight` and, e.g., penalize all coefficients (weights) of the defined tensor product spline using an $L_1$ penalty with $\lambda = 0.5$:

```
R> lambda <- 0.5
R> l1part <- tf$reduce_sum(tf$abs(mod_simple$model$trainable_weights[[1]]))
R> addpen <- function(x) lambda * l1part
R> mod_simple_pen <- deepregression(y = y, data = airbnb,
+    list_of_formulas = list_of_formulas, list_of_deep_models = NULL,
+    additional_penalty = addpen)
```

Note that the internal function must return a scalar value, and mathematical operations should be specified using **TensorFlow** functions, available through `tf$...` (and loaded together with **deepregression**).

*Smoothing penalties*

Apart from specifying the smoothness of **mgcv**'s smooth terms in the formulas manually (see `mgcv::s` for more details), two further options are available in **deepregression**. The first option is to use a single degrees-of-freedom specification using the argument `df` in `penalty_control`, a function that defines the `penalty_options` argument in `deepregression`. Using Demmler-Reinsch orthogonalization (see, e.g., Ruppert, Wand, and Carroll 2003), all smoothing parameters are then calculated based on this specification. This ensures that no smooth term has more flexibility than the other term. When `df` is left unspecified, the default `df` is 10. The second option in **deepregression** is to specify `df` for each smooth term in each formula by providing writing `df = ...` into the `s/te/ti` term.

**Details** The definition of the degrees-of-freedom can be changed using the `hat1` argument in `penalty_control`. When set to `TRUE`, the `df` are defined to be the sum of the diagonal terms (trace) of the hat matrix $H$ of the corresponding smooth term. The default is `FALSE`, yielding
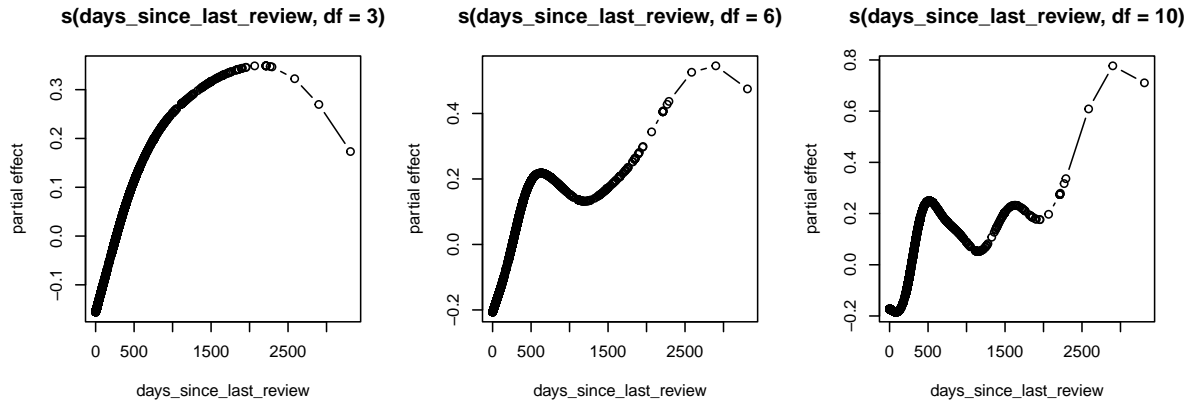
Figure 6: Case study. Comparison of different degrees-of-freedom settings for a single smooth term model.

the more common definition of the effective degrees-of-freedom df $= \text{trace}(2\boldsymbol{H} - \boldsymbol{H}\boldsymbol{H})$. As the penalty in **TensorFlow** is added to each single observation, the penalty term is per default divided by the number observations, preventing too strong penalization of the log-likelihood. In certain situations, it might be necessary to scale the penalty differently. This can be done using the argument `sp_scale` in `penalty_control`, which is `1/NROW(y)` by default to account for batch training. If training is done in full batch mode (batch size is equal to the total number of observations), we recommend setting `sp_scale` to `1`.

**Case study: Smoothing penalties** In the following we use a single smooth term to demonstrate the effect of `df`.

```
R> form_df_3 <- list(loc = ~ 1 + s(days_since_last_review, df = 3),
+     scale = ~ 1)
R> form_df_6 <- list(loc = ~ 1 + s(days_since_last_review, df = 6),
+     scale = ~ 1)
R> form_df_10 <- list(loc = ~ 1 + s(days_since_last_review, df = 10),
+     scale = ~ 1)
args <- list(y = y, data = airbnb, list_of_deep_models = NULL)
R> mod_df_low <- do.call("deepregression",
+     c(args, list(list_of_formulas = form_df_3)))
R> mod_df_med <- do.call("deepregression",
+     c(args, list(list_of_formulas = form_df_6)))
R> mod_df_max <- do.call("deepregression",
+     c(args, list(list_of_formulas = form_df_10)))
R> mod_df_low %>%
+     fit(epochs = 2000, early_stopping = TRUE, verbose = FALSE)
R> mod_df_med %>%
+     fit(epochs = 2000, early_stopping = FALSE, verbose = FALSE)
R> mod_df_max %>%
+     fit(epochs = 1000, early_stopping = FALSE, verbose = FALSE)
R> par(mfrow = c(1, 3))
R> plot(mod_df_low)
```

```
R> plot(mod_df_med)
R> plot(mod_df_max)
```

The different amount of smoothness resulting from the different choices in degrees-of-freedom is depicted in Figure 6.

## 2.6. Neural network settings

Since **deepregression** constitutes a holistic neural network, certain settings and advantages from deep learning can be made use of.

*Shared DNN*

In addition to the DNN specifications introduced in the Section 2.2.7, **deepregression** allows the user to share one DNN between some or all distributional parameters. This can drastically reduce the number of parameters to be estimated. The following example will demonstrate how to share a DNN between two parameters. The number of output units has to be equal to the number of parameters learned in this case. Sharing a network requires its inclusion in the `list_of_deep_models` and an additional list with formulas for the argument `train_together`. Analogously to `list_of_formulas`, this list provides the information which networks are used for which distributional parameters.

**Case study: Shared DNN** In the following example, we will learn both location and scale parameter of the rooms' listing prices using an embedding of words contained in the room description. We use the already tokenized text contained in the matrix `texts` in the `airbnb` data. The network we use consists of an embedding layer that processes the 1000 different tokenized words and learns a vector representation of the words in a 100-dimensional space. This representation is then reduced in the second dimension using the mean. Following this operation in the so-called Lambda layer, we use two fully-connected layers with 20 and two units and dropout in between. The two units in the last layer correspond to the two distributional parameters that are learned from the text information.

```
R> embd_mod <- function(x) x %>%
+    layer_embedding(input_dim = 1000, output_dim = 100) %>%
+    layer_lambda(f = function(x) k_mean(x, axis = 2)) %>%
+    layer_dense(20, activation = "tanh") %>%
+    layer_dropout(0.3) %>%
+    layer_dense(2)
```

In addition to the DNN, we will only use intercepts in both distributional parameters. To not copy and paste the `embd_mod` into every parameter formula, we create another list item which is provided to the `list_of_formulas`. The `mapping` argument then tells **deepregression** which formula is used for which parameter (here `list(1, 2, 1:2)` means that the first formula is used only in the first subnetwork for the location, the second is only used for the scale and the third used for the first and second subnetwork).

```
R> mod <- deepregression(y = y, list_of_formulas = list(
+      location = ~ 1, scale = ~ 1, both = ~ 0 + embd_mod(texts)),
```

```
+     mapping = list(1, 2, 1:2),
+     list_of_deep_models = list(embd_mod = embd_mod), data = airbnb)
```

This example creates an additive model where both the mean and the scale parameter are trained by the same network based on the text information `texts`. Following the last layer, the two network outputs extracted from those descriptions are then split and added to the respective additive predictor.

As an alternative to shared training, we could also define the network with a single output unit and simply add it directly to both parameters. This, however, does not result in the aforementioned reduction in model complexity due to shared weights:

```
R> embd_mod <- function(x) x %>%
+     layer_embedding(input_dim = 1000, output_dim = 100) %>%
+     layer_lambda(f = function(x) k_mean(x, axis = 2)) %>%
+     layer_dense(20, activation = "tanh") %>%
+     layer_dropout(0.3) %>%
+     layer_dense(1)
R> form_lists <- list(location = ~ 1 + embd_mod(texts),
+     scale = ~ 1 + embd_mod(texts))
R> mod <- deepregression(y = y, list_of_formulas = form_lists,
+     list_of_deep_models = list(embd_mod = embd_mod), data = airbnb)
```

*Optimizer and learning rate*

Per default, `deepregression` uses the function `keras_dr` as `model_builder`. The user can supply arguments to `keras_dr` via the ellipsis. In `keras_dr` the `optimizer` from **keras** is defined. It is therefore possible to specify any optimizer in `deepregression` that is available in **keras**, including `tf$keras$optimizers$Adadelta`, `tf$keras$optimizers$Adam`, `tf$keras$optimizers$RMSprop`, or standard stochastic gradient descent without momentum or adaptive learning rates (`tf$keras$optimizers$SGD`). The software uses Adam as the default option. This can be changed, e.g., to Adadelta with learning rate 3 and decay 0.1 by passing the argument `optimizer = optimizer_adadelta(learning_rate = 3, decay = 0.1)` to `deepregression`.

*Orthogonalization*

`deepregression` provides options for orthogonalization (for details, we refer the reader to Rügamer *et al.* 2020a) using the `orthog_options`, which are again defined by a control function `orthog_control`. Per default, `orthog_control` uses the global option to check for orthogonalization, i.e., it defines `orthogonalize = options()$orthogonalize`. This option is set to `TRUE` when loading the package and orthogonalizes DNNs in the predictor formulas containing overlapping features with structured terms of the same formula (see Figure 7 for a schematic visualization). This ensures identifiability of the structured term(s). Simply put, the orthogonalization constrains the deep network such that its latent learned features cannot represent feature effects that already exist in the structured predictor. Considering, for example, a simple linear model in the structured part, then the orthogonalization would prevent the unstructured (deep) network part from learning the univariate linear feature effects, but
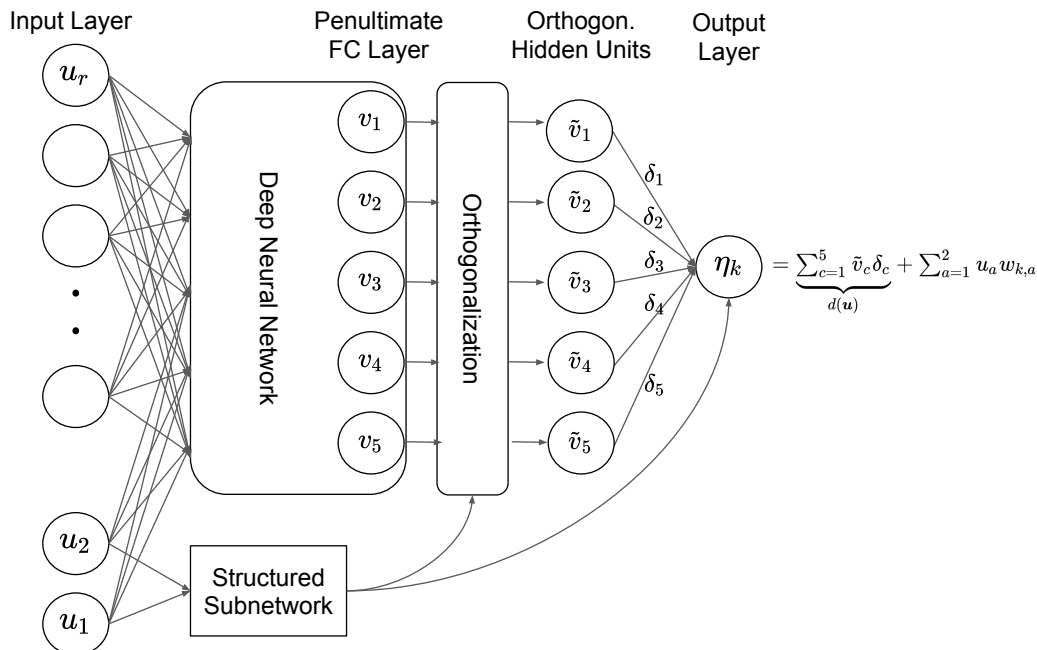
Figure 7: Orthogonalization cell: Effects of partially shared features are learned in a DNN and in a structured subnetwork. The resulting latent features $v_j$ in the DNN are then passed through an orthogonalization cell to ensure identifiability in the predictor $\eta_k$.

not, e.g., their joint interactions of arbitrarily high order. Put slightly more technical, this is achieved by projecting the latent features learned in the unstructured network's penultimate layer onto the orthogonal complement of the column space spanned by the linear features. In a simple linear model, the classic projection or hat matrix $\boldsymbol{P}$ projects $\boldsymbol{Y}$ onto the smaller subspace $\mathrm{col}(X)$ spanned by the features in $X$, resulting in $\hat{\boldsymbol{Y}} = \boldsymbol{P}\boldsymbol{Y}$. Conversely, the annihilator matrix $(\boldsymbol{I} - \boldsymbol{P})$ projects $\boldsymbol{Y}$ onto the orthogonal complement $\mathrm{col}(X)^{\perp}$ of $\mathrm{col}(X)$, resulting in $\hat{\boldsymbol{\varepsilon}} = (\boldsymbol{I} - \boldsymbol{P})\boldsymbol{Y}$. This ensures a unique decomposition $\boldsymbol{Y} = \hat{\boldsymbol{Y}} + \hat{\boldsymbol{\varepsilon}} = \boldsymbol{P}\boldsymbol{Y} + (\boldsymbol{I} - \boldsymbol{P})\boldsymbol{Y}$ for all $\boldsymbol{Y} \in \mathbb{R}^n$, because the space of response vectors can be written as an orthogonal decomposition of the two subspaces in terms of a direct sum: $\mathbb{R}^n = \mathrm{col}(X) \oplus \mathrm{col}(X)^{\perp}$ (orthogonal decomposition theorem). For a linear model, the unstructured DNN is therefore projected onto the subspace where the $\hat{\boldsymbol{\varepsilon}}$ "live", so as to extend the space of possible predictions, all without interfering with the linear effects.

Similar, the argument `identify_intercept = options()$identify_intercept` orthogonalizes the DNN with respect to the intercept to make the intercept identifiable. This option is `FALSE` per default as the interpretation of an intercept in a semi-structured model is not straightforward.

**Case study: Orthogonalization** In the following, we introduce a toy example to demonstrate the orthogonalization property of SDDR. We define our data generating process as $Y = 2x + \epsilon, \epsilon \sim \mathcal{N}(0, 1)$:

```R
R> n <- 1000
R> toyX <- rnorm(n)
R> toyY <- 2 * toyX + rnorm(n)
```

We here want to recover the linear effect of `toyX` with magnitude 2 in the presence of a DNN that also takes `toyX` as input. We define this DNN as follows:

```
R> deep_model <- function(x) {
+    x %>%
+      layer_dense(units = 100, activation = "relu", use_bias = FALSE) %>%
+      layer_dropout(rate = 0.2) %>%
+      layer_dense(units = 50, activation = "relu") %>%
+      layer_dropout(rate = 0.2) %>%
+      layer_dense(units = 1, activation = "linear")
+  }
```

Next, we fit a linear regression with linear effect for `toyX`. We pass the variable also to the `deep_model`, one time with orthogonalization and one time without orthogonalization.

```
R> forms <- list(loc = ~ -1 + toyX + deep_model(toyX), scale = ~ 1)
R> args <- list(y = toyY, data = data.frame(toyX = toyX),
+    list_of_formulas = forms,
+    list_of_deep_models = list(deep_model = deep_model))
R> w_oz <- orthog_control(orthogonalize = TRUE)
R> wo_oz <- orthog_control(orthogonalize = FALSE)
R> mod_w_oz <- do.call("deepregression",
+    c(args, list(orthog_options = w_oz)))
R> mod_wo_oz <- do.call("deepregression",
+    c(args, list(orthog_options = wo_oz)))
R> mod_w_oz %>%
+    fit(epochs = 1000, early_stopping = TRUE, verbose = FALSE)
R> mod_wo_oz %>%
+    fit(epochs = 1000, early_stopping = TRUE, verbose = FALSE)
R> cbind(with = c(coef(mod_w_oz, which_param = 1)[[1]]),
+    without = c(coef(mod_wo_oz, which_param = 1)[[1]]),
+    linmod = coef(lm(toyY ~ 0 + toyX)))


          with    without    linmod
toyX  2.008855 -0.7328619  2.009948
```

In the model with orthogonalization, the `deep_model` is orthogonalized with respect to `toyX` to ensure identifiability of the structured linear term. We thus can recover the linear effect in `mod_w_oz` despite the presence of the DNN which could have captured a linear (or more complex) effect as well. By default, orthogonalization automatically extracts the terms that overlap in the DNNs and the structured model formula. For expert use, there is also a custom orthogonalization available (see Section 2.8)

## 2.7. Advanced usage

**deepregression** allows for several advanced model specifications and user inputs, briefly described next.

*Offsets*

Several statistical models require an offset to be used in one or more (linear) predictors. These can be specified using an `offset(...)` term in the respective formulas.

*Constraints*

For smooth effects, further options are available using the `penalty_control` function which is passed to the argument `penalty_options`. Two important options are inherited by **mgcv**. `absorb_cons` (default `FALSE`) will absorb identifiability constraints into the smoothing basis (see `?mgcv::smoothCon`) and `zero_constraint_for_smooths` will constrain all smooth to sum to zero over their domain, which is usually recommended to prevent identifiability issues (default `TRUE`).

*Custom distribution function*

It is also possible to define custom distribution functions to be learned in `deepregression` using the `create_family` function and passing the result to `deepregression`'s `family` argument. To create a custom distribution, a distribution `tfd_dist` from **tfprobability** must be passed to `create_family`, as well as a list of transformations `trafo_list` defining the different $h_k$s of the distribution. For example, a normal distribution with different response functions $h_1(\eta) = 1/\eta$, $h_2(\eta) = \eta^2$ can be defined by passing the following to the `family` argument.

```
R> create_family(tfd_dist = tfd_normal, trafo_list = list(
+      function(x) tf$math$reciprocal(x),
+      function(x) tf$math$square(x)
+  ))
```

### 2.8. Custom orthogonalization

If there is reason to orthogonalize a DNN with respect to a structured model term that is not explicitly present, the `%OZ%`-operator can be used. On the left side of the `%OZ%`-operator in the formula, the DNN must be given. On the right side of the operator, either a single structured term (such as `x`, `s(x)`, `te(x,z)`) or a combination of structured terms using brackets around all terms and separated with a + between each term must be supplied (e.g., `deep_model(u)` `%OZ%` `(x + s(z))` to project the `deep_model` into the orthogonal complement of the space spanned by the linear effect of `x` and the basis of `s(z)`).

**Case study: Custom orthogonalization** We use the previous toy example and pretend that there is another feature that can potentially influence the linear effect of `toyX` with the same information content but another name (so that it is not recognized automatically as a candidate for orthogonalization).

```
R> toyXinDisguise <- toyX
R> form_known <- list(loc = ~ -1 + toyX + deep_model(toyX), scale = ~ 1)
R> form_unknown <- list(loc = ~ -1 + toyX + deep_model(toyXinDisguise),
+    scale = ~ 1)
```

```
R> form_manual <- list(
+    loc = ~ -1 + toyX + deep_model(toyXinDisguise) %OZ% (toyXinDisguise),
+    scale = ~ 1)
R> args <- list(y = toyY,
+    data = data.frame(toyX = toyX, toyXinDisguise = toyXinDisguise),
+    list_of_deep_models = list(deep_model = deep_model))
R> mod_known <- do.call("deepregression",
+    c(args, list(list_of_formulas = form_known)))

Preparing additive formula(e)... Done.

R> mod_unknown <- do.call("deepregression",
+    c(args, list(list_of_formulas = form_unknown)))

Preparing additive formula(e)... Done.

R> mod_manual <- do.call("deepregression",
+    c(args, list(list_of_formulas = form_manual)))

Preparing additive formula(e)... Done.

R> mod_known %>%
+    fit(epochs = 1000, early_stopping = TRUE, verbose = FALSE)
R> mod_unknown %>%
+    fit(epochs = 1000, early_stopping = TRUE, verbose = FALSE)
R> mod_manual %>%
+    fit(epochs = 1000, early_stopping = TRUE, verbose = FALSE)
R> cbind(known = coef(mod_known, params = 1)[[1]],
+    unknown = coef(mod_unknown, params = 1)[[1]],
+    manual = coef(mod_manual, params = 1)[[1]])

        [,1]     [,2]     [,3]
[1,] 2.011471 1.549017 2.011512
```

## 2.9. Working with images

The DNNs in SDDR can also incorporate images into the additive predictor of one or more parameters. Due to their size, images are usually not loaded into memory completely, but read in mini-batches from disk during training. For this purpose, **deepregression** offers a generator that creates small batches of images and other tabular information for training and prediction. The absolute path to the images must be given as string in the `data`. In addition, the user must then specify the image size in the respective element in the `list_of_deep_models` by providing a list with the name of the DNN and its size.

Some care needs to be taken with respect to interpreting the effect of unstructured data modalities, such as images, on the predictors $\eta_k, k = 1, \ldots, K$, and whether they affect the interpretation of structured linear coefficients. At a basic mathematical level, images are

typically represented as rank 3 tensors of shape (height, width, channels), i.e., a collection of non-negative numbers endowed with a spatial structure not unlike other constructs studied in statistics, e.g., functional or image data in a general functional regression context that also permits scalar covariates (see, e.g., Greven and Scheipl 2017). The additive structure of of the predictor $\eta_k$ in equation (1) in principle enables a classic *ceteris paribus* comparison, thereby leaving the interpretation of the linear coefficients unchanged, bar the fact that they are now also conditioned on the image. A unit change in the linear feature $x_j$ yields a $\beta_j$ change in $\eta_k$ for a given image, and by exchanging the images, we can calculate differences in $\eta_k$ to evaluate the effect of moving around the pixel space of images. Although one can often not interpret individual pixels (only if the images are preregistered, aligned and show the same underlying scene as, e.g., done in Lemhadri, Ruan, and Tibshirani 2021), we can make use of the representation learning property of modern CNNs. This allows partially interpretable comparisons at very high level learned features, even though the pixels from which the latent features were learned have no natural interpretation.

**Case study: Working with images**  The pictures of each of the apartments in the `airbnb` data can be downloaded from https://github.com/davidruegamer/airbnb. Here, the folder `data/pictures/32` contains all pictures from Munich. Assuming the pictures are stored in the home directory in a dedicated folder called `airbnb`, we first create the absolute path as follows:

```
R> airbnb$image <- paste0(getwd(), "/data/pictures/32/",
+    airbnb$id, ".jpg")
```

Next, we define an appropriate DNN architecture to extract latent features from the images. Here, we use a convolutional neural network (CNN). First, we define a CNN block:

```
R> cnn_block <- function(filters, kernel_size, pool_size, rate,
+    input_shape = NULL) {
+      function(x) {
+        x %>%
+        layer_conv_2d(filters, kernel_size, padding = "same",
+        input_shape = input_shape) %>%
+        layer_activation(activation = "relu") %>%
+        layer_batch_normalization() %>%
+        layer_max_pooling_2d(pool_size = pool_size) %>%
+        layer_dropout(rate = rate)
+      }
+    }
```

We then create the DNN function required by **deepregression** using a single block:

```
R> cnn <- cnn_block(filters = 16, kernel_size = c(3, 3),
+    pool_size = c(3, 3), rate = 0.25, shape(200, 200, 3))
R> deep_model_cnn <- function(x) {
+    x %>%
+    cnn() %>%
```

```
+       layer_flatten() %>%
+       layer_dense(32) %>%
+       layer_activation(activation = "relu") %>%
+       layer_batch_normalization() %>%
+       layer_dropout(rate = 0.5) %>%
+       layer_dense(1)
+   }
```

Note that we have to provide the shape of the image in the first layer of the CNN, which is $200 \times 200$ with three color channels. Finally, we define an illustrative `deepregression` model using an additive predictor for the mean with both, structured effects for, e.g., the room type as linear effect, and an unstructured effect by including the image information:

```
R> mod_cnn <- deepregression(y = y, list_of_formulas = list(
+       ~ 1 + room_type + bedrooms + beds + deep_model_cnn(image),
+       ~ 1 + room_type), data = airbnb,
+     list_of_deep_models = list(deep_model_cnn =
+       list(deep_model_cnn, c(200, 200, 3))),
+     optimizer = optimizer_adam(learning_rate = 0.0001))
```

The training using images is usually slower, but also requires less iterations. For the given example, we can, e.g., use the following setting for model fitting:

```
R> mod_cnn %>%
+     fit(epochs = 100, early_stopping = TRUE, patience = 5, verbose = FALSE)
```

Checking the influence of the image in additive predictors is not straightforward. One strategy is to set all other tabular variables to zero or to the reference category and to qualitatively inspect predictions of the SDDR network.

### 2.10. Uncertainty quantification

Next to analyzing prediction performance and the estimated partial effects, a crucial step in evaluating the model's reasoning is to assess the uncertainty of the model. While distributional regression directly models the uncertainty in the data generating process (the so-called *aleatoric uncertainty*), the uncertainty of the model (*epistemic uncertainty*) is of vital importance as well. **deepregression** provides two ways to characterize the variability of model estimates: (1) in the function space, i.e., the predicted outcome space, and (2) in the weight space for structured components, i.e., statistical inference for the weights of the structured model part.

*Function space uncertainty*

The solutions obtained by optimizing neural networks depend heavily on the stochastic nature of the optimization algorithm. For instance, re-running the same optimization on the same data yields different models with heterogeneous test performance due to the stochasticity of the optimization routine and different weight initializations. This function space uncertainty is taken advantage of in *deep ensembles* (Lakshminarayanan, Pritzel, and Blundell 2017) by
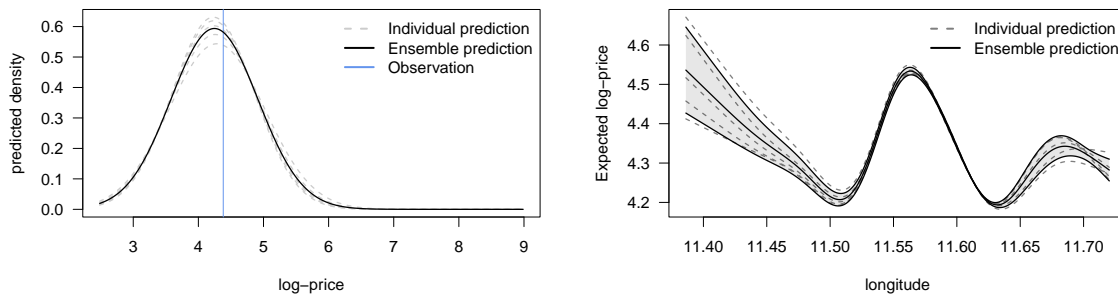
Figure 8: Left: Individual and combined `deepregression` models for the predicted conditional distribution of a single observation with observed log-price visualized by a vertical line. Individual predicted densities of the ensemble members are shown as dashed lines, and the ensemble prediction as a solid line. The average and ensemble negative log-likelihood for the single observation are shown on the right. Right: Expected log-price as a function of longitude with epistemic uncertainty ($\pm$ point-wise standard deviation of the ensemble members).

constructing an ensemble of neural networks. Deep ensembles linearly pool the predictions of its members, which, in many cases, improves prediction performance and leads to better out-of-distribution generalization and uncertainty quantification. Pooling the ensemble members can be seen as an approximate Bayesian method, explaining their purported benefits in epistemic uncertainty estimation (Maddox, Izmailov, Garipov, Vetrov, and Wilson 2019; Wilson and Izmailov 2020).

In **deepregression**, deep ensembles are implemented as a mixture distribution of the predicted distributions of its members and can be trained using the `ensemble()` function. The resulting distribution is a '`tfd_distribution`' object, which allows numerous downstream computations, including the log-likelihood, cumulative distribution and survivor function.

**Case study: Quantifying predictive uncertainty**  The following fits a deep ensemble for the Airbnb data using early stopping and a 20% validation split.

```
R> mod <- deepregression(y = y, data = airbnb, list_of_formulas = list(
+     location = ~ 1 + s(longitude), scale = ~ 1 + s(longitude)))
R> ens <- ensemble(mod, n_ensemble = 5, epochs = 1e3,
+     early_stopping = TRUE, validation_split = 0.2)
R> ensemble_distr <- get_ensemble_distribution(ens, data = newdata)
```

Figure 8 (left) depicts the predicted rating density of a single observation for all ensemble members (dashed lines) as well as the ensemble itself (solid line), together with the actually observed rating and the negative log-likelihood for that observation. The ensemble reduces the influence of overconfident predictions and improves upon the average loss of the single models. The right panel illustrates the epistemic uncertainty quantification by showing the estimated expected logarithmic apartment price as a function of longitude.

*Weight space uncertainty*

Next to uncertainty quantification in the function space, the uncertainty in the weight space of the neural network is not of lesser importance. While it is difficult to interpret or infer

insights from uncertainties of weights in the deep network part(s), the uncertainty in the structured part is what more classical regression approaches refer to as statistical inference. Since deep neural network components in the additive predictors can usually be represented as $d_{k,l}(\boldsymbol{u}) = \tilde{\boldsymbol{u}}_{k,l}^\top \tilde{\boldsymbol{w}}_{k,l}$ with latent features $\tilde{\boldsymbol{u}}_{k,l}$ and weights $\tilde{\boldsymbol{w}}_{k,l}$, the final additive predictor in semi-structured regression can also be seen as an extended structured predictor with random features $\tilde{\boldsymbol{u}}_{k,l}$. One possibility to quantify the uncertainty of the structured part is hence to extract the latent features of deep neural network components and refit the last layer of the model. This is similar to a local linearization approach (Immer, Korzepa, and Bauer 2021) and referred to as *last layer inference* (Daxberger, Kristiadi, Immer, Eschenhagen, Bauer, and Hennig 2021). Note, however, that this approach underestimates the variance in model as it neglects the uncertainty of previous layers in the deep network components. Similar to deep ensembles, repeating network training and refitting might give thus additional insights into how much this variance influences structured network components.

**Case study: Last layer statistical inference**    The following example fits a simple example of a deep generalized additive model used in the beginning of this article and then refits the model based on the latent features learned in the last layer of the deep neural network. We first define the neural network as follows:

```
R> deep_model <- function(x) {
+    x %>%
+    layer_dense(units = 5, activation = "relu", use_bias = FALSE) %>%
+    layer_dense(units = 3, name = "penultimate_layer") %>%
+    layer_dense(units = 1, activation = "linear")
+  }
```

Note that we can name the penultimate layer to make it easier later to access the latent features of this layer. We then define and fit the model.

```
R> mod <- deepregression(y = y, data = airbnb,
+    list_of_formulas = list(
+      location = ~ 1 + beds + s(accommodates, bs = "ps") +
+        s(days_since_last_review, bs = "tp") +
+        deep(review_scores_rating, reviews_per_month),
+      scale = ~ 1),
+    list_of_deep_models = list(deep = deep_model))
R> mod %>% fit(epochs = 100, verbose = FALSE, view_metrics = FALSE,
+    validation_split = 0.2)
```

To extract the learned latent features of the model, we create a temporary `keras` model to extract the intermediate predictions $\tilde{\boldsymbol{u}}$ (here 3-dimensional):

```
R> intermediate_mod <- keras_model(mod$model$input, mod$model$get_layer(
+      name = "penultimate_layer")$output)
R> newdata_processed <- prepare_newdata(
+    mod$init_params$parsed_formulas_contents, airbnb,
+    gamdata = mod$init_params$gamdata$data_trafos)
R> tilde_u <- as.data.frame(intermediate_mod$predict(newdata_processed))
R> str(tilde_u, 1)
```
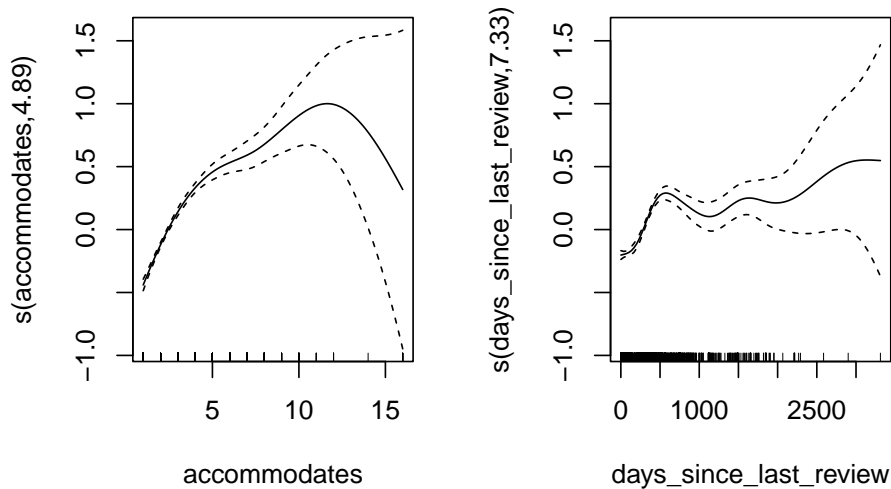
Figure 9: Estimated nonlinear effects and their uncertainty obtained from the refitted deep GAM.

```
'data.frame': 3504 obs. of  3 variables:
 $ V1: num  -12.1 -12 -11.4 -12 -12 ...
 $ V2: num  18.6 18.4 17.6 18.5 18.3 ...
 $ V3: num  -28.7 -28.5 -27.2 -28.5 -28.3 ...
```

Given `tilde_u`, we now refit the last layer using **mgcv** and extract the uncertainty statements of interest (here confidence intervals for nonlinear effects depicted in Figure 9).

```
R> gam_mod <- gam(log(price) ~ 1 + beds + s(accommodates, bs = "ps") +
+     s(days_since_last_review, bs = "tp") + V1 + V2 + V3,
+     data = cbind(as.data.frame(airbnb), tilde_u))
R> plot(gam_mod, pages = 1)
```

## 3. Conclusion

We have presented **deepregression**, a software package that implements the SDDR framework in R. **deepregression** allows to learn the parameters of a wide range of distributions based on a combination of additive regression models and deep neural networks. The implementation is comprised of three essential components: (1) a modular neural network building system for the seamless combination of statistical and deep learning approaches, (2) an orthogonalization cell (as put forward by Rügamer *et al.* 2020a) to allow for an interpretable combination of different subnetworks, and (3) comprehensive pre-processing tools to set up the models. The software provides a common formula interface to specify distributional parameters, allows to learn shared weights and automatically handles identifiability issues. We believe that **deepregression**'s modular design and functionality fosters rapid and reproducible prototyping of complex (combinations of) statistical and deep learning models in one common environment.

# Acknowledgments

# References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015). "**TensorFlow**: Large-Scale Machine Learning on Heterogeneous Systems." URL https://www.tensorflow.org/.

Allaire JJ, Chollet F (2022). *keras: R Interface to **Keras***. R package version 2.9.0, URL https://CRAN.R-project.org/package=keras.

Baumann PFM, Hothorn T, Rügamer D (2021). "Deep Conditional Transformation Models." In N Oliver, F Pérez-Cruz, S Kramer, J Read, JA Lozano (eds.), *Machine Learning and Knowledge Discovery in Databases. Research Track*, pp. 3–18. Springer-Verlag, Cham. doi: 10.1007/978-3-030-86523-8_1.

Daxberger E, Kristiadi A, Immer A, Eschenhagen R, Bauer M, Hennig P (2021). "Laplace Redux-Effortless Bayesian Deep Learning." *Advances in Neural Information Processing Systems*, **34**. doi:10.7551/mitpress/11474.003.0014.

Dunson DB (2010). "Nonparametric Bayes Applications to Biostatistics." In NL Hjort, C Holmes, P Müller, SG Walker (eds.), *Bayesian Nonparametrics*, Cambridge Series in Statistical and Probabilistic Mathematics, chapter 7, pp. 223–273. Cambridge University Press. doi:10.1017/cbo9780511802478.008.

Fritz C, Dorigatti E, Rügamer D (2022). "Combining Graph Neural Networks and Spatio-Temporal Disease Models to Predict COVID-19 Cases in Germany." *Scientific Reports*, **12**, 2045–2322. doi:10.1038/s41598-022-07757-5.

Greven S, Scheipl F (2017). "A General Framework for Functional Regression Modelling." *Statistical Modelling*, **17**(1-2), 1–35. doi:10.1177/1471082x16681317.

Hoff PD (2017). "Lasso, Fractional Norm and Structured Sparse Estimation Using a Hadamard Product Parametrization." *Computational Statistics & Data Analysis*, **115**, 186–198. doi:10.1016/j.csda.2017.06.007.

Hothorn T, Kneib T, Bühlmann P (2014). "Conditional Transformation Models." *Journal of the Royal Statistical Society B*, **76**(1), 3–27. doi:10.1111/rssb.12017.

Immer A, Korzepa M, Bauer M (2021). "Improving Predictions of Bayesian Neural Nets via Local Linearization." In A Banerjee, K Fukumizu (eds.), *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pp. 703–711. PMLR. URL `https://proceedings.mlr.press/v130/immer21a.html`.

Jacobs RA, Jordan MI, Nowlan SJ, Hinton GE (1991). "Adaptive Mixtures of Local Experts." *Neural Computation*, **3**(1), 79–87. `doi:10.1162/neco.1991.3.1.79`.

Keydana S (2022). **tfprobability**: *Interface to* **TensorFlow Probability**. R package version 0.15.0, URL `https://CRAN.R-project.org/package=tfprobability`.

Klein N, Kneib T, Lang S, Sohn A (2015). "Bayesian Structured Additive Distributional Regression with an Application to Regional Income Inequality in Germany." *The Annals of Applied Statistics*, **9**(2), 1024–1052. `doi:10.1214/15-aoas823`.

Koenker R (2005). *Quantile Regression*. Cambrigde University Press, New York. Economic Society Monographs.

Kopper P, Pölsterl S, Wachinger C, Bischl B, Bender A, Rügamer D (2021). "Semi-Structured Deep Piecewise Exponential Models." In *Proceedings of AAAI Spring Symposium on Survival Prediction – Algorithms, Challenges, and Applications 2021*, volume 146 of *Proceedings of Machine Learning Research*, pp. 40–53. PMLR. URL `https://proceedings.mlr.press/v146/kopper21a.html`.

Kopper P, Wiegrebe S, Bischl B, Bender A, Rügamer D (2022). "DeepPAMM: Deep Piecewise Exponential Additive Mixed Models for Complex Hazard Structures in Survival Analysis." In J Gama, T Li, Y Yu, E Chen, Y Zheng, F Teng (eds.), *Advances in Knowledge Discovery and Data Mining (PAKDD)*, pp. 1–13. Springer-Verlag, Cham. `doi:10.1007/978-3-031-05936-0_20`.

Lakshminarayanan B, Pritzel A, Blundell C (2017). "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles." In I Guyon, UV Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, R Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc. URL `https://proceedings.neurips.cc/paper/2017/file/9ef2ed4b7fd2c810847ffa5fa85bce38-Paper.pdf`.

Lemhadri I, Ruan F, Tibshirani R (2021). "LassoNet: Neural Networks with Feature Sparsity." In A Banerjee, K Fukumizu (eds.), *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pp. 10–18. PMLR. URL `https://proceedings.mlr.press/v130/lemhadri21a.html`.

Maddox WJ, Izmailov P, Garipov T, Vetrov DP, Wilson AG (2019). "A Simple Baseline for Bayesian Uncertainty in Deep Learning." In H Wallach, H Larochelle, A Beygelzimer, F d'Alché Buc, E Fox, R Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates. URL `https://proceedings.neurips.cc/paper/2019/file/118921efba23fc329e6560b27861f0c2-Paper.pdf`.

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL `https://www.R-project.org/`.

Rigby RA, Stasinopoulos DM (2005). "Generalized Additive Models for Location, Scale and Shape." *Journal of the Royal Statistical Society C*, **54**(3), 507–554. `doi:10.1111/j.1467-9876.2005.00510.x`.

Rügamer D (2023). **deepregression**: *Fitting Deep Distributional Regression*. R package version 1.0.0, URL `https://CRAN.R-project.org/package=deepregression`.

Rügamer D, Baumann PFM, Kneib T, Hothorn T (2022). "Probabilistic Time Series Forecasts with Autoregressive Transformation Models." *arXiv 2110.08248*, arXiv.org E-Print Archive. `doi:10.48550/arxiv.2110.08248`.

Rügamer D, Kolb C, Klein N (2020a). "Semi-Structured Deep Distributional Regression: Combining Structured Additive Models and Deep Learning." *arXiv 2002.05777*, arXiv.org E-Print Archive. `doi:10.48550/arxiv.2002.05777`.

Rügamer D, Pfisterer F, Bischl B (2020b). "Neural Mixture Distributional Regression." *arXiv 2010.06889*, arXiv.org E-Print Archive. `doi:10.48550/arxiv.2010.06889`.

Ruppert D, Wand MP, Carroll RJ (2003). *Semiparametric Regression*. Cambridge University Press, Cambridge.

Schnabel SK, Eilers P (2009). "Optimal Expectile Smoothing." *Computational Statistics & Data Analysis*, **53**, 4168–4177. `doi:10.1016/j.csda.2009.05.002`.

Stasinopoulos DM, Rigby RA (2007). "Generalized Additive Models for Location Scale and Shape (GAMLSS) in R." *Journal of Statistical Software*, **23**(7), 1–46. `doi:10.18637/jss.v023.i07`.

Tibshirani R (2021). "Equivalences Between Sparse Models and Neural Networks." *Technical report*, Carnegie Mellon University. Working Notes, URL `https://www.stat.cmu.edu/~ryantibs/papers/sparsitynn.pdf`.

Umlauf N, Klein N, Simon T, Zeileis A (2021). "**bamlss**: A Lego Toolbox for Flexible Bayesian Regression (and Beyond)." *Journal of Statistical Software*, **100**(4), 1–53. `doi:10.18637/jss.v100.i04`.

Wilson AG, Izmailov P (2020). "Bayesian Deep Learning and a Probabilistic Perspective of Generalization." *Advances in Neural Information Processing Systems*, **33**, 4697–4708. `doi:10.1109/icaiic48513.2020.9065256`.

Wood SN (2001). "**mgcv**: GAMs and Generalized Ridge Regression for R." *R News*, **1**(2), 20–25. URL `https://CRAN.R-project.org/doc/Rnews/`.

Wood SN (2003). "Thin Plate Regression Splines." *Journal of the Royal Statistical Society B*, **65**(1), 95–114. `doi:10.1111/1467-9868.00374`.

Wood SN (2017). *Generalized Additive Models: An Introduction with R*. 2nd edition. Chapman & Hall/CRC. `doi:10.1201/9781315370279`.

Ziel F, Muniain P (2021). **gamlss.lasso**: *Extra Lasso-Type Additive Terms for GAMLSS*. R package version 1.0-1, URL `https://CRAN.R-project.org/package=gamlss.lasso`.

**Affiliation:**

David Rügamer
Chair of Statistical Learning & Data Science
Department of Statistics
LMU Munich
80539, Munich, Germany
E-mail: david.ruegamer@stat.uni-muenchen.de