Journal of Statistical Software

November 2024, Volume 111, Issue 10. *doi: 10.18637/jss.v111.i10*

Estimating Conditional Distributions with Neural Networks Using R Package deeptrafo

Lucas Kook
Vienna University of Economics and Business

Philipp F. M. Baumann
ETH Zurich

Oliver Dürr
HTWG Konstanz

Beate Sick
University of Zurich
Zurich University of Applied Sciences

David Rügamer
LMU Munich
Munich Center for Machine Learning

Abstract

Contemporary empirical applications frequently require flexible regression models for complex response types and large tabular or non-tabular, including image or text, data. Classical regression models either break down under the computational load of processing such data or require additional manual feature extraction to make these problems tractable. Here, we present **deeptrafo**, a package for fitting flexible regression models for conditional distributions using a **tensorflow** back end with numerous additional processors, such as neural networks, penalties, and smoothing splines. Package **deeptrafo** implements deep conditional transformation models (DCTMs) for binary, ordinal, count, survival, continuous, and time series responses, potentially with uninformative censoring. Unlike other available methods, DCTMs do not assume a parametric family of distributions for the response. Further, the data analyst may trade off interpretability and flexibility by supplying custom neural network architectures and smoothers for each term in an intuitive formula interface. We demonstrate how to set up, fit, and work with DCTMs for several response types. We further showcase how to construct ensembles of these models, evaluate models using inbuilt cross-validation, and use other convenience functions for DCTMs in several applications. Lastly, we discuss DCTMs in light of other approaches to regression with non-tabular data.

Keywords: deep learning, distributional regression, neural networks, transformation models.

1. Introduction

Regression analysis aims to characterize the conditional distribution of a response Y given a set of covariates $\boldsymbol{X}$, thereby describing how changes in the covariates propagate to the conditional distribution of Y given $\boldsymbol{X}$ (Fahrmeir, Kneib, Lang, and Marx 2013). In this paper, we present **deeptrafo** (Kook, Baumann, and Rügamer 2024), an R package for estimating a broad class of distributional regression models for various types of responses (continuous, survival, count, ordinal, binary) using tabular or non-tabular (e.g., image or text) data or both. Package **deeptrafo** is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=deeptrafo`. Due to a flexible **tensorflow** (Allaire and Tang 2024) back end and mini-batch optimization, **deeptrafo** does not only scale well with non-tabular (imaging, text) data but also large tabular data sets. Many well-known models fall into the class of transformation models (TMs), such as normal linear regression (Lm), Cox proportional hazards models (CoxPH), and proportional odds logistic regression (Polr, Hothorn, Möst, and Bühlmann 2018). In the following, we review existing software for fitting these models.

Existing software packages. TMs for tabular data are implemented in **tram** (Hothorn, Barbanti, and Siegfried 2024) using **mlt** (Hothorn 2020a) and fitted via maximum likelihood, relying on **alabama** (Varadhan 2023) and **BB** (Varadhan and Gilbert 2009) for optimization. Package **tram** provides an intuitive interface for fitting a multitude of distributional regression models, ranging from shift and shift-scale (Siegfried, Kook, and Hothorn 2024) to tensor-product (or "conditional") transformation models (Hothorn, Kneib, and Bühlmann 2014). Several extensions of transformation models exist. For instance, **cotram** for count TMs (Siegfried and Hothorn 2020), **tramME** for mixed effects TMs and TMs including smoothing splines (Tamási and Hothorn 2021), and **tramnet** as well as **tramvs** for regularized TMs (Kook and Hothorn 2021; Kook 2024). Transformation boosting machines (Hothorn 2020b) and transformation trees and random forests (Hothorn 2023) offer extensions to classical machine learning models. Table 1 summarizes the commonalities and differences between the packages implementing different (extensions of) transformation models in terms of model classes, support for **mgcv**-based splines and **tensorflow**-based neural networks and scalable optimization (via mini-batch training, see Appendix H). The **deeptrafo** package is currently the only package implementing transformation models which supports neural network architectures enabling direct handling of text, image, and other deep learning-related data without requiring feature engineering.

Neural network-based transformation models. With the advent of (deep) neural networks and the routine collection of non-tabular data, the idea to combine deep learning and distributional regression approaches was adopted in several ways. For instance, Rügamer, Kolb, and Klein (2024) parameterize distributional regression models via neural networks, Sick, Hothorn, and Dürr (2021) describe flexible deep transformation models for continuous responses. Kook, Herzog, Hothorn, Dürr, and Sick (2022) focus on semi-structured regression for ordinal responses, and Rügamer, Baumann, Kneib, and Hothorn (2023a) extend the DCTM approach to distributional autoregressive models for time series responses. Alternative approaches to combining regression with neural networks including generalized additive models for location, scale, and shape have been implemented in **deepregression** (Rügamer

Package	Model class	Nonlinear	Splines	Neural networks	Scalable optimization
tram	Linear TMs	✗	✗	✗	✗
cotram	Count TMs	✗	✗	✗	✗
tramnet	L_1/L_2-penalized TMs	✗	✗	✗	✗
tramvs	L_0-penalized TMs	✗	✗	✗	✗
tbm	Additive TMs	✓	✗	✗	✗
trtf	Transformation forests	✓	✗	✗	✗
tramME	Additive mixed TMs	✓	✓	✗	✗
deeptrafo	Additive TMs	✓	✓	✓	✓

Table 1: Overview of packages for estimating different classes of transformation models. Packages **tramME**, **tbm**, **trtf**, and **deeptrafo** support estimation of nonlinear TMs. Specifically, **tramME** supports splines from **mgcv**, **tbm** fits nonlinear model components via score-based boosting, **trtf** fits nonlinear effects by aggregating trees with TMs in the leaves, and **deeptrafo** supports both splines from **mgcv** and neural networks from **tensorflow**. Package **deeptrafo** allows for scalable optimization via mini-batch training.

et al. 2023c). In this paper, we present **deeptrafo**, which unifies the above DCTM approaches in a single R package.

Comparison to existing packages. Combining distributional regression with neural network-based estimation has many advantages, such as modularity (data analysts can easily use well-established problem-specific neural network architectures), and easy handling of big datasets (e.g., through mini-batch gradient descent with adaptive learning rates). Like **tram**, **deeptrafo** relies on maximizing a likelihood function. However, stochastic first-order optimization, such as stochastic gradient descent and the ability to deal with non-tabular data distinguishes the two packages (Table 1). Further, **deeptrafo** covers and extends models implemented in **cotram**. Like **tramME**, **deeptrafo** also allows the specification of smoothing splines via **mgcv** (Wood 2023). However, the focus of our package does not lie on random effects. Penalization as in **tramnet** is also available for **deeptrafo**. Lastly, unlike models in **deepregression**, DCTMs do not require specification of a parametric family of distributions for the response given covariates.

The rest of this paper is organized as follows. Section 1.1 introduces the statistical theory behind TMs and DCTMs. The inner workings of **deeptrafo** are described in Section 2, where several case studies on how to setup up, fit, validate, and interpret DCTMs are presented. We present an application to binary classification with tabular and text modalities, and an application to time series modeling via autoregressive TMs (Rügamer *et al.* 2023a). The appendix contains information on advanced usage of the package, e.g., how censored responses are handled (Appendix B) or how to warm-start or fix parameters of certain predictors (Appendix C). In Appendix H, we demonstrate the package for large tabular datasets and factors with many levels, which cannot be handled by standard implementations of classical regression models.

1.1. Deep conditional transformation models

Transformation models (Hothorn *et al.* 2014, 2018) estimate the conditional cumulative distribution function (CDF) of a response $Y \in \mathcal{Y} \subseteq \mathbb{R}$ given a realization $\boldsymbol{x}$ of covariates $\boldsymbol{X} \in \mathcal{X}$,

$$F_{Y|\boldsymbol{X}=\boldsymbol{x}}(y) := \mathsf{P}(Y \leq y \mid \boldsymbol{X} = \boldsymbol{x})$$

without committing to a particular parametric family of distributions for $F_{Y|\boldsymbol{X}=\boldsymbol{x}}$. Instead of estimating the CDF directly, transformation models estimate how to transform the response (conditional on covariates) to a latent variable $Z := h(Y \mid \boldsymbol{x})$ with fixed and user-defined CDF $F_Z : \mathbb{R} \to [0, 1]$, using the *transformation function* $h : \mathcal{Y} \times \mathcal{X} \to \mathbb{R}$, which is constrained to be monotonically non-decreasing in $y \in \mathcal{Y}$ for all $\boldsymbol{x} \in \mathcal{X}$. Then, the conditional CDF of the outcome given covariates can be evaluated using the latent CDF F_Z and the transformation function h:

$$\mathsf{P}(Y \leq y \mid \boldsymbol{X} = \boldsymbol{x}) = \mathsf{P}(h(Y \mid \boldsymbol{x}) \leq h(y \mid \boldsymbol{x}) \mid \boldsymbol{X} = \boldsymbol{x}) = \mathsf{P}(Z \leq h(y \mid \boldsymbol{x})) = F_Z(h(y \mid \boldsymbol{x})).$$

For continuous responses, h is continuous and for discrete responses, h is discrete (see Figure 1). Expressing the conditional CDF in terms of F_Z and h yields simple expressions for probability density and mass functions and thus also the log-likelihood.

Depending on the choice of F_Z and restrictions on the functional form and parameterization of h, TMs cover a wide range of well-known models with varying complexity.

Example 1 (Beyond normal linear regression) *Choosing* $F_Z = \Phi$ *and* $h(y \mid \boldsymbol{x}) = \sigma^{-1}(y - \alpha - \boldsymbol{x}^\top \boldsymbol{\beta})$*, with standard deviation* $\sigma > 0$*, and intercept* $\alpha \in \mathbb{R}$*, is equivalent to a normal linear regression model, since* $\mathsf{P}(Y \leq y \mid \boldsymbol{X} = \boldsymbol{x}) = \Phi(\sigma^{-1}(y - \alpha - \boldsymbol{x}^\top \boldsymbol{\beta}))$*. Fixing the transformation function to be linear will always result in conditionally normal outcome distributions. However, this restriction can be lifted by using a nonlinear increasing transformation,* $h_Y : \mathcal{Y} \to \mathbb{R}$*, i.e.,* $h(y \mid \boldsymbol{x}) = h_Y(y) - \boldsymbol{x}^\top \widetilde{\boldsymbol{\beta}}$*, which now assumes that the transformed response* $h_Y(Y)$ *(instead of the original response) is normal with mean* $\boldsymbol{x}^\top \widetilde{\boldsymbol{\beta}}$*.*

Example 2 (Beyond Weibull regression) *Choosing* $F_Z(z) = 1 - \exp(-\exp(z))$ *with* $h(y \mid \boldsymbol{x}) = a + b \log y + \boldsymbol{x}^\top \boldsymbol{\beta}$*, with intercept* a *and slope* $b > 0$*, is equivalent to a Weibull regression model, since* $\mathsf{P}(Y \leq y \mid \boldsymbol{X} = \boldsymbol{x}) = 1 - \exp(-\exp(a + b \log y + \boldsymbol{x}^\top \boldsymbol{\beta})) = 1 - \exp(-\widetilde{a} y^b \exp(\boldsymbol{x}^\top \boldsymbol{\beta}))$*, where* $\widetilde{a} := \exp(a)$*. Also in this example, log-linearity of the transformation function fixes the conditional outcome to be Weibull distributed. Allowing an arbitrary increasing function,* $h_Y : \mathcal{Y} \to \mathbb{R}$*, instead, i.e.,* $h(y \mid \boldsymbol{x}) = h_Y(y) + \boldsymbol{x}^\top \widetilde{\boldsymbol{\beta}}$*, results in the Cox proportional hazards model, since the survivor function equals* $\mathsf{P}(Y \geq y \mid \boldsymbol{X} = \boldsymbol{x}) = \exp(-\exp(h_Y(y)) \exp(\boldsymbol{x}^\top \widetilde{\boldsymbol{\beta}}))$ *and* $\exp(h_Y(y))$ *is the cumulative baseline hazards.*

Thus, TMs contain both normal linear and Weibull regression but also extend both to a more flexible counterpart that does not assume a parametric family of conditional outcome distributions.

Parameterizing the transformation function. In semi-structured regression, we have access to J input modalities $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_J$, such as tabular data, images, or text, from which we construct structured (e.g., linear, sparse, or smooth) or unstructured (e.g., neural network) predictors. These inputs may be non-tabular, i.e., there may be a j for which $\boldsymbol{X}_j \in \mathcal{X}_j \not\subseteq \mathbb{R}^d$. By $\mathcal{X} := \mathcal{X}_1 \times \cdots \times \mathcal{X}_J$ we denote the entire input space. In DCTMs, restrictions on the functional form of h, i.e., the way predictors are constructed based on the input data, lead

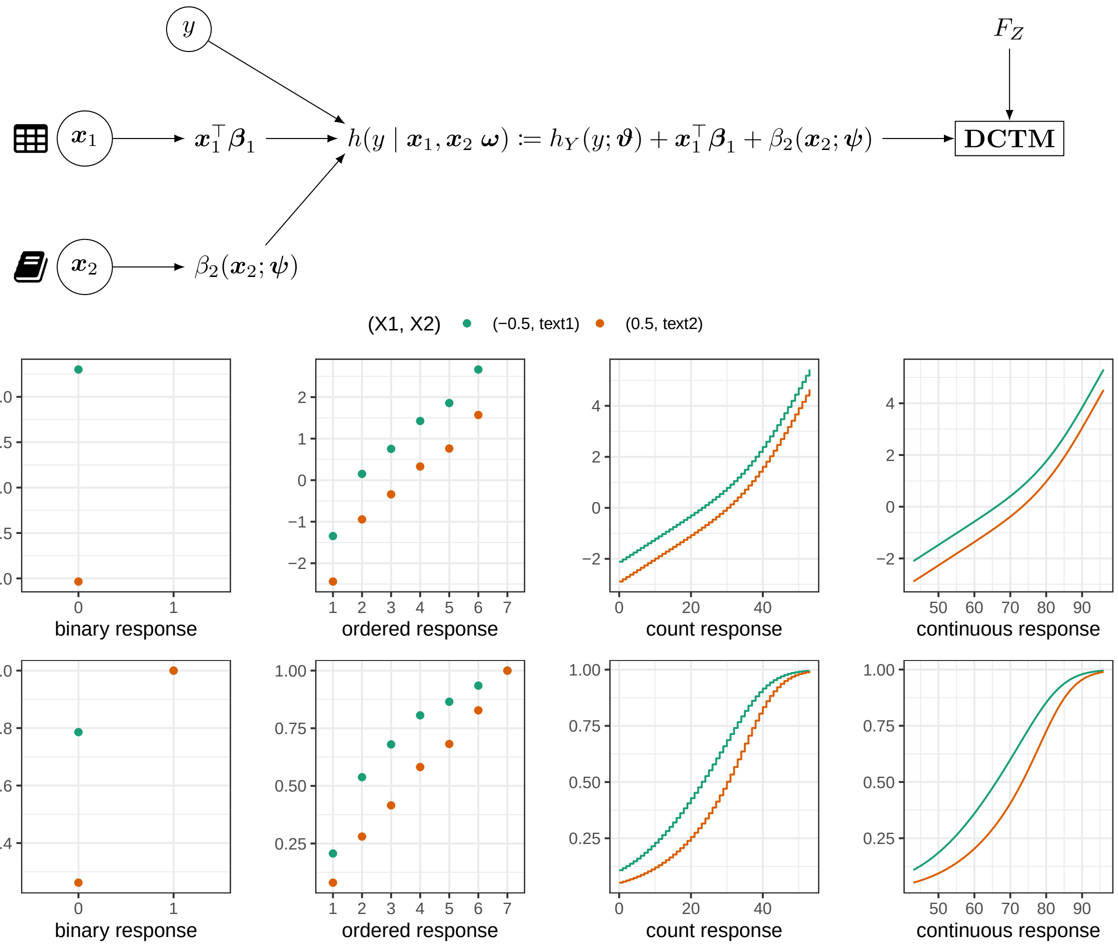

Figure 1: Example of a DCTM with transformation function $h(y \mid \boldsymbol{x}_1, \boldsymbol{x}_2)$ depending on a tabular modality $\boldsymbol{X}_1$ and a text modality $\boldsymbol{X}_2$, which both enter as an additive shift term. The tabular modality enters as a simple linear predictor $\boldsymbol{x}_1^\top \boldsymbol{\beta}_1$ and the text data via the output of a neural network β_2, which is specified by the user. Together with a baseline transformation h_Y, whose parameterization is discussed later, and the latent distribution F_Z, the DCTM is fully specified. On the bottom, the transformation function h and cumulative distribution function $F_{Y|\boldsymbol{X}_1=\boldsymbol{x}_1, \boldsymbol{X}_2=\boldsymbol{x}_2} = F_Z \circ h$ are depicted for a binary, ordered, count, and continuous response for two realizations of the tabular (X1, X2) and text modalities (text1, text2). For binary and ordered responses with K levels, the transformation function contains one and $K-1$ parameters, respectively, because the CDF is constrained to one for the largest class.

to varying degrees of interpretability and flexibility of the model. We begin with an example before introducing h in its most flexible form. Consider a problem with a single tabular ($\boldsymbol{X}_1 \in \mathcal{X}_1 \subseteq \mathbb{R}^p$) and a single text modality ($\boldsymbol{X}_2 \in \mathcal{X}_2$). Data analysts commonly assume additivity in the effects the separate modalities, which can be realized by modeling the effect of both modalities as shift terms,

$$h(y \mid \boldsymbol{x}_1, \boldsymbol{x}_2; \boldsymbol{\omega}) = h_Y(y; \boldsymbol{\vartheta}) + \boldsymbol{x}_1^\top \boldsymbol{\beta}_1 + \beta_2(\boldsymbol{x}_2; \boldsymbol{\psi}), \quad y \in \mathcal{Y}, \tag{1}$$

where $h_Y : \mathcal{Y} \to \mathbb{R}$ denotes the baseline transformation (i.e., the transformation function obtained when $\boldsymbol{x}_1^\top \boldsymbol{\beta}_1 + \beta_2(\boldsymbol{x}_2) = 0$, which is parameterized in terms of $\boldsymbol{\vartheta} \in \mathbb{R}^M$). Further, $\boldsymbol{\beta}_1$ denotes the coefficients of the linear predictor and $\beta_2 : \mathcal{X}_2 \to \mathbb{R}$ denotes the unstructured predictor for the text data, which are typically controlled by a neural network with weights $\boldsymbol{\psi}$. By $\boldsymbol{\omega} := (\boldsymbol{\vartheta}, \boldsymbol{\beta}, \boldsymbol{\psi})$, we denote the collection of all parameters, including the neural network weights. A DCTM with (1) is distribution-free because for any constellation of covariates for which the shifting predictor is zero, i.e., for all $(\boldsymbol{x}_1^0, \boldsymbol{x}_2^0) \in S^0 := \{(\boldsymbol{x}_1, \boldsymbol{x}_2) \in \mathcal{X}_1 \times \mathcal{X}_2 \mid \boldsymbol{x}_1^\top \boldsymbol{\beta}_1 + \beta_2(\boldsymbol{x}_2) = 0\}$, and all conditional distributions $Y \mid \boldsymbol{X}_1 = \boldsymbol{x}_1^0, \boldsymbol{X}_2 = \boldsymbol{x}_2^0$, there exists a unique baseline transformation given by $h_Y = F_Z^{-1} \circ F_{Y \mid \boldsymbol{X}_1 = \boldsymbol{x}_1^0, \boldsymbol{X}_2 = \boldsymbol{x}_2^0}$. In (1), covariate effects are assumed to enter additively on the scale of the transformation function, thus restricting distributions that can be modeled for $(\boldsymbol{x}_1, \boldsymbol{x}_2) \in \mathcal{X} \setminus S^0$. This argument can be extended to more complex DCTMs (for shift-scale see, e.g., Siegfried *et al.* 2024). The example in (1) is depicted in Figure 1 for typical types of responses and standard logistic latent distribution.

In **deeptrafo**, the most general transformation function h is parameterized in terms of $\boldsymbol{\omega} := (\boldsymbol{\vartheta}, \boldsymbol{\beta}, \boldsymbol{\phi}, \boldsymbol{\psi}) \in \mathbb{R}^{Md} \times \mathbb{R}^p \times \mathbb{R}^q \times \mathbb{R}^s$ which serves as the collection of parameters for basis expansions (potentially including neural networks) of the response and input modalities,

$$h(y \mid \boldsymbol{x}; \boldsymbol{\omega}) = (\boldsymbol{a}(y) \otimes \boldsymbol{b}(\boldsymbol{x}; \boldsymbol{\phi}))^\top \boldsymbol{\vartheta} + \boldsymbol{s}(\boldsymbol{x}; \boldsymbol{\psi})^\top \boldsymbol{\beta}, \quad y \in \mathcal{Y}, \ \boldsymbol{x} \in \mathcal{X}, \tag{2}$$

where $\otimes$ denotes the Kronecker product and $\boldsymbol{a} : \mathcal{Y} \to \mathbb{R}^M, \boldsymbol{b} : \mathcal{X} \to \mathbb{R}^d, \boldsymbol{s} : \mathcal{X} \to \mathbb{R}^p$ denote the bases for the response, and the J predictors, which either interact ($\boldsymbol{b}(\cdot; \boldsymbol{\phi})$) with the response or simply shift ($\boldsymbol{s}(\cdot; \boldsymbol{\psi})$) the transformation function. The dimensions of the neural network weights $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ depend on the complexity of the neural network architectures which the user has full control over. In **deeptrafo**, the basis for the response is not data-dependent and thus contains no parameters. The interacting and shifting basis, however, depend on the covariates and may include splines or neural networks, whose parameters are collected in $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$, respectively.

The transformation function is required to be monotonically non-decreasing for all $\boldsymbol{x} \in \mathcal{X}$. Hence, depending on the choice of basis, the parameters $\boldsymbol{\vartheta}$ in (2) need to fulfill positivity or monotonicity constraints (Hothorn *et al.* 2014), which can be enforced by appropriate reparameterizations. Without interacting predictors, Bernstein polynomials and discrete bases require $\vartheta_1 \leq \vartheta_2 \leq \cdots \leq \vartheta_M$ and linear and log-linear bases require positive slopes. For more complex interacting predictors, the positivity of $\boldsymbol{b}(\cdot; \boldsymbol{\phi})$ has to be enforced together with more complex constraints on $\boldsymbol{\vartheta}$ to ensure a monotonically non-decreasing transformation function (for details, see Baumann, Hothorn, and Rügamer 2021).

Shift effects are constant across all values of the response, i.e., the transformation h can only shift up- or downwards (see Figure 1). The effect of interacting predictors may vary with the response and thus the shape of h may change for different predictor values. For instance, an interacting binary predictor leads to two separate transformations for each level, much like stratum variables in survival analysis allow for separate hazard functions (Collett 2015). However, in its general form, interacting predictors may also include neural networks and thus unstructured predictors, making them extremely versatile. Scale effects as introduced in Siegfried *et al.* (2024) are a special case of interacting predictors, which are included in **deeptrafo** by using $\boldsymbol{b} : \mathcal{X} \to \mathbb{R}_+$ with $\boldsymbol{x} \mapsto \sqrt{\exp(\gamma(\boldsymbol{x}))}$ and, e.g., a neural network $\gamma : \mathcal{X} \to \mathbb{R}$. With a linear basis $\boldsymbol{a}$ in y, $\boldsymbol{a}(y) = (1, y)^\top$, this is equivalent to location-scale regression with error distribution F_Z.

Supported response types. Several types of univariate, potentially censored, responses can be handled. This includes continuous ($\mathcal{Y} \subseteq \mathbb{R}$), survival ($\mathcal{Y} \subseteq \mathbb{R}_+$), count ($\mathcal{Y} = \mathbb{N}$), and ordered ($\mathcal{Y} = \{y_1, \dots, y_K\}$) responses. For continuous responses, the basis for Y is a smooth function parameterized via polynomials in Bernstein form of order $M-1$, denoted by $\boldsymbol{a}_{\text{Bs},M-1}(y)$. For count responses ($M = K$), the polynomials in Bernstein form are evaluated only at the integers, i.e., $\boldsymbol{a}_{\text{Bs},M-1}(\lfloor y \rfloor)$ (Siegfried and Hothorn 2020). For ordered responses, a dummy-encoding is used, i.e., for $k = 1, \dots, K$, $\boldsymbol{b}(y_k) = \boldsymbol{e}_k$, where $\boldsymbol{e}_k$ denotes the k-th unit vector. Linear and log-linear bases are supported as well. In Appendix D, we describe how the user can supply custom basis functions.

Fitting transformation models. Finally, transformation models can be fitted by minimizing the negative average log-likelihood over the class of transformation functions $h(y \mid \boldsymbol{x}; \boldsymbol{\omega})$ with parameters $\boldsymbol{\omega}$,

$$\text{NLL}(\boldsymbol{\omega}; y_i, \boldsymbol{x}_i) \coloneqq -\frac{1}{n}\sum_{i=1}^{n} \ell(\boldsymbol{\omega}; y_i, \boldsymbol{x}_i),$$

where the observations $\{(y_i, \boldsymbol{x}_i)\}_{i=1}^{n}$ are assumed to be independent. In **deeptrafo**, the default optimizer is (stochastic) gradient descent using Adam (Kingma and Ba 2015). However, any **keras** (Allaire and Chollet 2024) or **tensorflow** optimizer or a custom optimization routine can be used instead. For a single observation $(y, \boldsymbol{x})$, the log-likelihood contribution $\ell(h; y, \boldsymbol{x})$ depends on the type of censoring of the observed response. Exact responses y contribute $\log f_Z(h(y \mid \boldsymbol{x}))h'(y \mid \boldsymbol{x})$ to the log-likelihood. Interval-censored responses $(\underline{y}, \bar{y}]$ contribute $\log(F_Z(h(\bar{y} \mid \boldsymbol{x})) - F_Z(h(\underline{y} \mid \boldsymbol{x})))$. Left- and right-censored observations follow from the interval-censored contribution as a special case, by letting $\underline{y} \to -\infty$ and $\bar{y} \to +\infty$, respectively (Hothorn *et al.* 2014). In **deeptrafo**, the log-likelihood contributions are implemented in terms of mathematical operations implemented in **tensorflow**, which call their Python (Van Rossum *et al.* 2011) counterpart via **reticulate** and allow efficient computation of the log-likelihood, its gradients and weight updates during optimization.

1.2. Autoregressive transformation models

Time series data pose one particular case where the independence assumption between observations is not tenable and needs to be taken into account. Formally, the joint distribution of a time series $(Y_t)_{t \in \mathcal{T}}$ with $\mathcal{T} \subseteq \mathbb{N}_0$ can always be factorized in its conditional distributions, i.e., by conditioning Y_t on its full history $\mathcal{F}_{t,1} \coloneqq (Y_{t-1}, \dots, Y_1)$. A simplification is to impose a Markov property of order $p \geq 1$ which implies that the conditional distribution of Y_t only depends on the history up to and including $t-p$, that is $\mathcal{F}_{t,t-p} \coloneqq (Y_{t-1}, \dots, Y_{t-p})$ rather than the entire history $\mathcal{F}_{t,1}$.

Package **deeptrafo** offers three ways on how to model time series data assuming the Markov property. The naive way is given by classical transformation models where $\mathcal{F}_{t,t-p}$ is regarded in the basis expansion of the transformation function shown in (2) where elements of $\mathcal{F}_{t,t-p}$ may interact with the response Y_t and simultaneously shift the transformation function. Furthermore, Rügamer *et al.* (2023a) proposed the class of autoregressive transformation models (ATMs) which differ from the naive approach (i.e., classical transformation models) in two perspectives. First, the transformation function h_t in ATMs can be time-varying which may result in different transformations for different sub-periods. Second, the same h_t is applied to

> “superman returns discover 5 absence allowed lex luthor walk free closest abandoned moved luthor plots ultimate revenge millions killed change planet forever ridding steel”

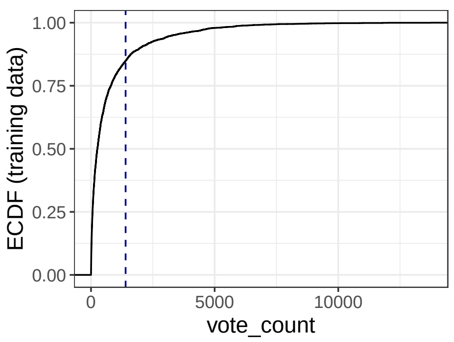

Figure 2: Left: The pre-processed movie review of a picked instance of the `movies` dataset, in which stop words and punctuation have already been removed. Right: The empirical CDF of `vote_count` over all movies in the training, where the picked instance has a `vote_count` of 1400 as indicated by the dashed line (placing it above the top quartile). The used tabular input data comprise `popularity` (4.08 for the picked instance) and `revenue` (400 million US dollars for the picked instance).

Y_t and each element of $\mathcal{F}_{t,t-p}$ simultaneously, resulting in a shared transformation between Y_t and its lags.

A special subclass of ATMs are AT(p) models which do not allow for interacting elements of $\mathcal{F}_{t,t-p}$ with Y_t through $\boldsymbol{b}$ but restrict to a linear shift impact of the transformed values of $\mathcal{F}_{t,t-p}$ on the scale of h. The class of AT(p) models is closely related to a well-known class of time series models, i.e., autoregressive models of order p (AR(p), Hamilton 2020). In fact, AT(p) models are equivalent to AR(p) models for $\boldsymbol{a}(y) = (1, y)^\top$, $\boldsymbol{s}(\boldsymbol{x}) \equiv \boldsymbol{x}$ and the independent white noise follows the distribution F_Z (for details, see Rügamer *et al.* 2023a). Learning the transformation simultaneously for the response and its lags as it is done in ATMs is particularly important for ordinal time series, for which the dimensionality of the model can thereby be reduced. Instead of modeling each level of the lagged response, the one-dimensional transformed lagged response is included. It also allows for a more consistent interpretation in the sense of autoregression because we model $h(Y_t)$ at the current step (auto)regress the next time point $h(Y_{t+1})$ on the likewise transformed response $h(Y_t)$, not on the untransformed Y_t. We showcase the practical differences between linear transformation models, AT(p) and ATM models in Section 4.

1.3. Application datasets

Movies data. In Section 2, we will illustrate the features of **deeptrafo** using the `movies` dataset (Kaggle 2017). The dataset contains information on 45,000 movies released prior to July 2017, including number of ratings, budget, revenue, popularity, run time, and genre. In addition, non-tabular reviews of the movies are available as text data. In Section 2, we will focus on estimating the conditional distribution of `vote_count` given whether a movie is an action movie, its budget, its popularity score, and the text review. In Section 3, we will switch to the binary classification task of deciding whether a movie falls into the action genre or not. This way, we can showcase how to apply DCTMs for a wider range of outcome types. We pre-process budget, revenue, and popularity using $\log(1 + x)$, due to their skewed nature. In Figure 2, we show the empirical CDF of the variable `vote_count` of the `movies` dataset and provide more information on the used variables for one specific movie. For the text data, we use a `text_tokenizer` with a 1,000 word vocabulary, convert text to sequence and pad sequences to a maximum length of 100 and truncate the end of a review. We use such a simple embedding to illustrate the key steps of the analysis and make the computations feasible on a standard laptop with 8 gigabytes of RAM. We additionally present results with a pre-trained embedding that performs comparably in terms of test NLL in Appendix E.

Temperature data. An application of autoregressive transformation models to a time series of monthly mean maximum temperature in Melbourne (Australia) in degrees Celsius between January 1971 and December 1990 (240 records) is presented in Section 1.2. The `temperature` time series was recorded by the Australian Bureau of Meteorology and later provided in Hyndman and Yang (2022).

2. The package

Package **deeptrafo** builds upon **tensorflow** as a fitting engine and **deepregression** for setting up structured model terms such as linear effects or splines within a neural network. In contrast to **deepregression**, which implements models with parametric families and individual additive predictors, **deeptrafo** supports more complicated computations such as in (2). This is exposed to the user via **deeptrafo**'s formula interface. In **deeptrafo**, response, interacting, and shifting terms are represented as '`formula`' objects and correspond to the bases in (2). Internally, a `processor` is defined for each model term, which evaluates its basis functions and optional penalties via **deeptrafo** internal, **mgcv**, or **keras**/**tensorflow** functions. For instance, for a continuous response, a polynomial basis in Bernstein form and its derivatives are set up by default (cf. Table 2). The corresponding basis functions are implemented in **deeptrafo**. Package **deeptrafo** can include terms modeled by user-specified neural network architectures for the interacting and shifting terms (see Figure 1 and Figure 3). When initializing the model using such a formula-based call, the model is internally translated to **tensorflow** computations using a computational graph. In the end, a single end-to-end trainable neural network is set up, which may contain different neural network components for different terms in the interacting or shifting predictor. Together with the supplied `latent_distr` F_Z, the DCTM is fully specified and its parameters can be estimated by minimizing the NLL via stochastic gradient descent (SGD). Since the DCTM has internally been translated to a model from **tensorflow**, the optimization can be done using the **keras** API, which implements the SGD routine with many choices for adaptive learning rates while providing training metrics without requiring users to define training loops for parameter updates. An appropriate last-layer transformation ensures monotonicity constraints of the interacting model term in the response.

Workflow. Typical workflows around **deeptrafo**, including the illustration in Section 2 and both applications on binary classification (Section 3) and distributional time series (Section 4), are structured as follows: First, a model formula is set up. The '`formula`' object encodes in which way each feature enters the model. If neural network components are used, the corresponding architectures have to be defined beforehand. Next, the latent distribution F_Z is chosen and decides which scale the partial effects of components in the formula are interpreted. Although the formula together with the latent distribution formally specify the

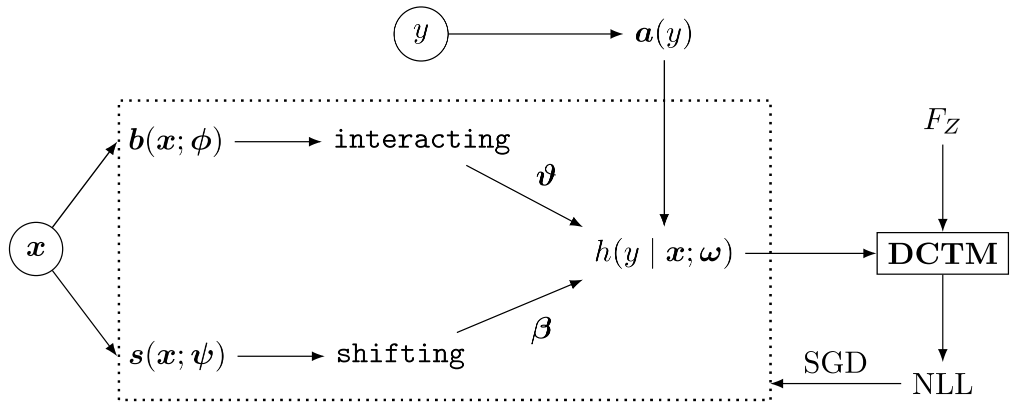
y
$\boldsymbol{a}(y)$
$\boldsymbol{b}(\boldsymbol{x}; \boldsymbol{\phi})$
interacting
$\boldsymbol{\vartheta}$
F_Z
$\boldsymbol{x}$
$h(y \mid \boldsymbol{x}; \boldsymbol{\omega})$
DCTM
$\boldsymbol{s}(\boldsymbol{x}; \boldsymbol{\psi})$
shifting
$\boldsymbol{\beta}$
SGD
NLL

Figure 3: Schematic depiction of setting up and fitting DCTMs. Bases for input predictors $\boldsymbol{x}$ and response y (circles) are evaluated and enter the two neural network components `interacting` and `shifting` according to (3). The components' outputs make up the transformation function h which is parameterized in terms of $\boldsymbol{\omega}$. Together with the latent distribution F_Z, the loss, e.g., NLL, and its gradients can be evaluated and used to update parameters $\boldsymbol{\omega}$. Since F_Z is parameter-free, all trainable parameters are in the transformation function, as indicated by the dotted box.

Model function	Model name	Default basis	Default latent distribution
`BoxCoxNN`	Transformed normal	Bernstein	Standard normal
`ColrNN`	Continuous outcome logistic	Bernstein	Standard logistic
`cotramNN`	Count transformation	Bernstein	Standard logistic
`CoxphNN`	Cox proportional hazards	Bernstein	Standard minimum extreme value
`LehmannNN`	Lehmann-type	Bernstein	Standard maximum extreme value
`LmNN`	Normal linear	Linear	Standard normal
`PolrNN`	Proportional odds logistic	Discrete	Standard logistic
`SurvregNN`	Weibull	Log-linear	Standard minimum extreme value

Table 2: Supported models together with the default choice of basis function and latent distribution. Model functions summarized here are implemented with a specific choice of basis and latent distribution that define commonly applied regression models.

TM completely (Figure 3), the data and optimizer have to be supplied at this stage. For deep learning models (as opposed to statistical models), it is common to separate model building from model fitting, in order to supply more arguments (such as callbacks) to the optimization routine. Now, hyperparameters can be tuned based on cross-validation. Finally, with the chosen hyperparameters, either a single instance of the DCTM or an ensemble is fitted and can be used for downstream prediction tasks. In Section 2.1, we describe each step of the workflow in more detail using the `movies` data.

Each step in the **deeptrafo** workflow is highly customizable, e.g., custom functions for basis evaluation (Appendix D), custom last-layer transformations, and general-purpose optimization routines (Section 3), such as SGD with adaptive learning rates (Appendix F), can be supplied.

Effect / Processor	Example formula
Linear	`y ~ x`
Smooth	`y ~ s(x, ...)`
Tensor product splines	`y ~ [te|ti|t2](x, ...)`
Lasso	`y ~ lasso(x)`
Group lasso	`y ~ grlasso(x)`
Row-wise tensor product	`y ~ rwt(x)`
Neural network	`y ~ nn(x)`
Processor	`*_processor` e.g., `fac_processor`

Table 3: Implemented choices of `interacting` and `shift` processors. All splines from **mgcv** are supported. Custom neural networks can be supplied as functions via `list_of_deep_models`. Additional processors, for example, for faster processing of large factors or interactions from **safareg**, can be included via `additional_processors` (Rügamer *et al.* 2023b; Rügamer 2024). All terms can also be included as interacting effects on the left-hand side of the formula, e.g., `y | term(x, ...) ~ 1`.

2.1. Main components

We describe the main components of **deeptrafo** below by showing how to use the formula interface, set up a DCTM, and fit the model. In this section, all steps are illustrated with the `movies` dataset. In the following examples, we assign non-default values to some of the arguments that can be supplied to functions and methods for building and fitting **keras**-based neural networks. This is not because the models have been tuned extensively, but rather to illustrate the most important hyperparameters that are involved in building and fitting DCTMs.

Formula interface

Models can be specified via a formula interface akin to the one used in **tram** (Hothorn *et al.* 2024), where covariates interacting with the response are supplied on the left-hand side, and shift effects are supplied on the right-hand side of the formula, as illustrated below.

```
response | interacting ~ shifting
```

Thus, the formula interface mimics the transformation function as introduced in (2):

$$\big(\underbrace{\boldsymbol{a}(y)}_{\texttt{response}} \otimes \overbrace{\boldsymbol{b}(\boldsymbol{x};\boldsymbol{\phi})}^{\texttt{interacting}}\big)^\top \boldsymbol{\vartheta} + \underbrace{\boldsymbol{s}(\boldsymbol{x};\boldsymbol{\psi})}_{\texttt{shifting}}{}^\top \boldsymbol{\beta}. \tag{3}$$

Case study: Formula interface. We begin by modeling the conditional distribution of `vote_count` given a binary indicator of whether the movie is categorized as an action movie or not (`genreAction`), the movie's `budget` and its `popularity`. The below formula allows for separate baseline transformations of the response for action movies *vs.* all other genres, a smooth effect for `budget` and a linear effect for `popularity`. Here, we use the standard spline basis representation implemented in **mgcv**. A non-exhaustive list of smoothers and other processors is given in Table 3. Processors are specialized functions for handling predictors which can speed up computation. For instance, `fac_processor()` from **safareg** evaluates factors online and thus facilitates computation for large factor models (Rügamer *et al.* 2023b, also see the illustration in Appendix H).

```
R> fm <- vote_count | genreAction ~ 0 + s(budget, df = 3) + popularity
```

In the above formula we exclude an additional intercept in the shift term by specifying `0 + ...`, because the interacting basis already contains an intercept.

Setting up DCTMs

DCTMs can be generically set up using the `deeptrafo()` function.

```
deeptrafo(formula = response | interacting ~ shifting, data = ...)
```

The `data` can be supplied as a `data.frame` or `list`. The function returns a '`deeptrafo`' object, whose methods are described in Section 2.2.

Special cases of DCTMs coincide with well-known models and are given their own function in **deeptrafo**. The naming conventions in **deeptrafo** follow the **tram** package (Hothorn *et al.* 2024) and add the suffix `NN`. For instance, the proportional odds logistic regression model (ordinal response and $F_Z = \text{expit}$) is implemented as `Polr()` in **tram** and `PolrNN()` in **deeptrafo** (see Table 2 for an overview).

Case study: Setting up DCTMs. For the `movies` data, we set up a count transformation model with standard logistic latent distribution. The logistic distribution is chosen, so that the partial effects of the features are interpretable as log-odds ratios. Example interpretations are given in Section 3. We supply the Adam optimizer (the default, see Appendix F) for SGD with learning rate of 0.1 decaying with a rate of $4 \cdot 10^{-4}$ (Kingma and Ba 2015). The training data `train` is the result of the pre-processing steps described in Section 1.3. The code for reproducing all output and figures can be found in the online supplement.

```
R> opt <- optimizer_adam(learning_rate = 0.1, decay = 4e-4)
R> (m_fm <- cotramNN(formula = fm, data = train, optimizer = opt))
```

```
         Untrained count outcome deep conditional transformation model

Call:
cotramNN(formula = fm, data = train, optimizer = opt)

Interacting:  vote_count | genreAction

Shifting:  ~0 + s(budget, df = 6) + popularity
```

```
Shift coefficients:
s(budget, df = 6)1 s(budget, df = 6)2 s(budget, df = 6)3 s(budget, df = 6)4
             0.557             -0.702              0.760             -0.181
s(budget, df = 6)5 s(budget, df = 6)6 s(budget, df = 6)7 s(budget, df = 6)8
            -0.201             -0.687              0.670              0.671
s(budget, df = 6)9         popularity
            -0.377             -0.888
```

Printing the model to the console shows the model specification and shift coefficients. Note that the model has only been randomly initialized and not yet fitted, as indicated by "Untrained count outcome deep conditional transformation model" in the `print()` call. Upon calling `fit()`, `ensemble()`, or `cv()`, the model's history will be non-empty and it will be considered "trained" when printed again.

Fitting DCTMs

For fitting DCTMs the user calls `fit()`, which calls the model internal `mod$fit_fun()`, per default a wrapper around `fit.keras.engine.training.Model()`, with the supplied arguments (for instance `epochs`, `batch_size`). All functionalities of fitting `keras` models carry over to fitting DCTMs, including callbacks (i.e., custom operations applied after every iteration or mini-batch update).

Case study: Fitting DCTMs. The '`deeptrafo`' object returned by `cotramNN` is fitted for 1,000 epochs, with a batch size of 64, and a 10% validation split. The validation split is used during training to judge whether overfitting occurs (Goodfellow, Bengio, and Courville 2016). Below, we print the (now trained) model.

```
R> m_fm_hist <- fit(m_fm, epochs = 1e3, validation_split = 0.1,
+    batch_size = 64, verbose = FALSE)
R> unlist(coef(m_fm, which = "shifting"))
```

```
s(budget, df = 6)1 s(budget, df = 6)2 s(budget, df = 6)3 s(budget, df = 6)4
           0.38339           -0.28824           -0.04608           -0.03992
s(budget, df = 6)5 s(budget, df = 6)6 s(budget, df = 6)7 s(budget, df = 6)8
           0.00616           -0.02692           -0.00511            0.01355
s(budget, df = 6)9         popularity
          -0.36587           -0.82771
```

Figure 4A depicts the training and validation loss trajectory for inspecting convergence and overfitting, which can be generated with `plot(m_fm_hist)`. The learning curves indicate that the model is not fully trained after 1000 epochs and there is no evidence for overfitting. Figure 4B shows the estimated transformation function. In Section 2.2, we describe how to produce plots of the transformation function and density. Since `genreAction` is included as a response-varying effect, the two transformation functions are allowed to cross.

Working with neural networks

The **deeptrafo** package allows to directly model effects of, for instance, text or image data via neural networks. In DCTMs, neural networks map from a complex input space, such as text

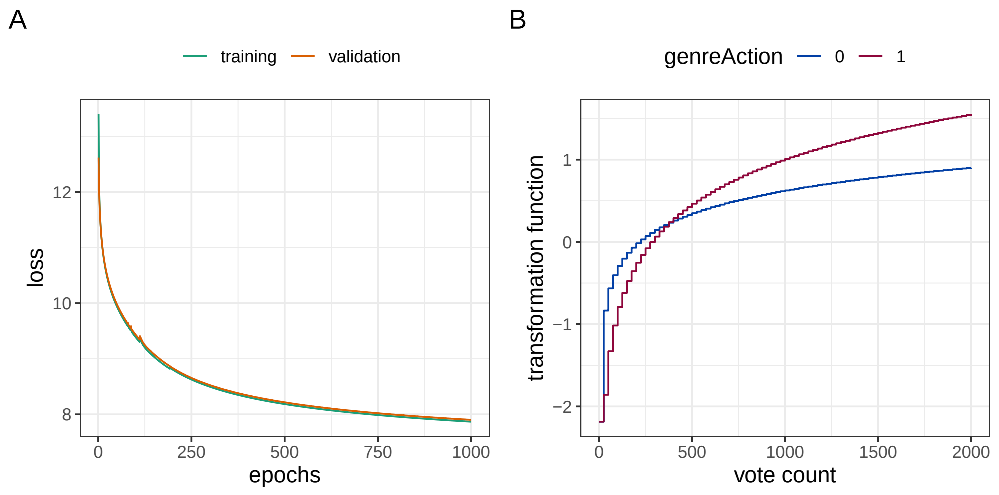

Figure 4: A: Training and validation loss trajectory for `m_fm`. B: Estimated transformation functions for both levels of `genreAction` with `popularity` and `budget` fixed at their mean in the training data.

or images, to Euclidean space. When the neural network enters as a shift term, the output of the network is a real number which is interpretable on the latent scale F_Z^{-1}, i.e., the scale of the transformation function. Custom neural networks can be supplied to `deeptrafo` as functions or '`keras_model`'s via the `list_of_deep_models` argument.

Case study: Working with neural networks. In our running example, we use the following architecture to model the contribution of the movie reviews provided as textual descriptions. In Section 3, we present an application with further downstream analysis of the text embedding and how this simple embedding compares against using larger pre-trained embeddings.

```
R> embd_mod <- function(x) x |>
+     layer_embedding(input_dim = nr_words, output_dim = embedding_size) |>
+     layer_lstm(units = 50, return_sequences = TRUE) |>
+     layer_lstm(units = 50, return_sequences = FALSE) |>
+     layer_dropout(rate = 0.1) |> layer_dense(25) |>
+     layer_dropout(rate = 0.2) |> layer_dense(5) |>
+     layer_dropout(rate = 0.3) |> layer_dense(1)
```

The neural network `embd_mod` maps movie ratings to a real value (for more details see the case study in Section 3). The interpretational scale of output depends on the choice of latent distribution. Here, the logistic distribution (F_Z = expit) renders the output of `embd_mod` interpretable on the log-odds scale. In turn, differences in the output of `embd_mod` can be interpreted as log odds-ratios when changing, for instance, a single word in a sentence and leaving everything else constant. In our deeptrafo model, we can now supply a named list `list(deep = embd_mod)` and use `deep(texts)` in the formula.

```
R> fm_deep <- update(fm, . ~ . + deep(texts))
R> m_deep <- cotramNN(fm_deep, data = train,
+    list_of_deep_models = list(deep = embd_mod))
R> fit(m_deep, epochs = 50, validation_split = 0.1, batch_size = 32,
+    callbacks = list(callback_early_stopping(patience = 5)),
+    verbose = FALSE)
```

The default optimization routine may not produce optimization paths as smooth as when omitting the neural network component. However, adaptively scheduled learning rates for SGD often work well out-of-the-box, e.g., using `optimizer = optimizer_adam()` as an argument when initializing the '`deeptrafo`' model. Sometimes also different learning schedules are needed for the different modalities (see Section 3).

Ensembling DCTMs

A simple and popular method to improve prediction performance and to quantify training stability (i.e., uncertainty from random initialization and stochastic optimization) are deep ensembles (Lakshminarayanan, Pritzel, and Blundell 2017). In a deep ensemble, a neural network model is trained B times using the same training and validation data, but different initial weights. Training via SGD may then converge to different (local) minima and the members may yield different predictions. However, averaging the predicted densities of the B ensemble members is guaranteed to improve upon the average individual performance (e.g., in terms of NLL). In **deeptrafo**, an ensemble of a model can be fitted via `ensemble()`. Besides classical deep ensembling, **deeptrafo** implements transformation ensembles (Kook, Götschi, Baumann, Hothorn, and Sick 2022). Transformation ensembles are specifically tailored towards DCTMs and preserve their additive structure and thus (partial) interpretability by averaging the predicted transformation functions instead of the predicted densities.

Case study: Ensembling DCTMs. Below, we fit five instances of `m_deep`. Then, we combine their predictions on the scale of the transformation function and can investigate uncertainty in the effects of the shifting predictors and prediction performance on the test set.

```
R> ens_deep <- ensemble(m_deep, n_ensemble = 3, epochs = 50, batch_size = 64,
+    verbose = FALSE)
```

Figure 5 shows the estimated smooth effect of `budget` with training stability indicated by the shaded area. Investigating the out-of-sample prediction performance, we see that the transformation ensemble performs better than the members do on average (see Proposition 3 in Kook *et al.* 2022).

```
R> unlist(logLik(ens_deep, convert_fun = \(x) - mean(x)))

members1 members2 members3     mean ensemble
   -8.28    -8.50    -8.32    -8.37    -8.35
```

Cross-validating DCTMs for hyperparameter tuning

With `cv()`, **deeptrafo** provides a cross-validation function for '`deeptrafo`' objects. When supplying an integer to `cv_folds`, the data is split into `cv_folds` number of folds. Alternatively, the user can specify a list with two elements indicating data indices for training

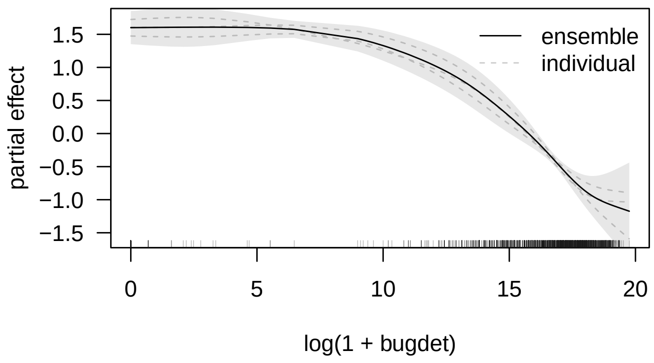

Figure 5: Training stability in the estimated smooth partial effect of `budget` on `vote_count` obtained via transformation ensembling.

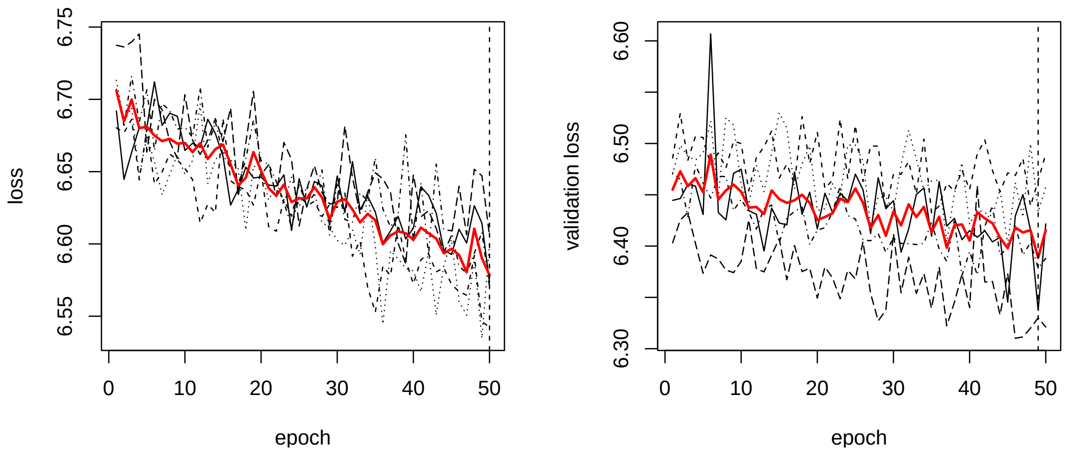

Figure 6: Default plot generated by `cv.deeptrafo()`. The vertical lines indicate the epoch with minimal average train/validation loss.

and validation. The output of `cv()` can be used for tuning smoothing hyperparameters, choosing between including a predictor as interacting or shifting, or different neural network architectures.

Case study: Cross-validating DCTMs. The following call to `cv()` performs 5-fold cross validation while fitting each instance of `m_deep` for 50 epochs. Train and validation loss trajectories are shown in Figure 6. The vertical bars indicate the epoch with the best average train/validation loss.

```
R> cv_deep <- cv(m_deep, epochs = 50, cv_folds = 5, batch_size = 64)
R> plot_cv(cv_deep)
```

2.2. Methods overview

In the following, we briefly describe S3 methods for ‘`deeptrafo`’ and ‘`dtEnsemble`’ objects.

Methods for '`deeptrafo`' objects

- `coef()` returns coefficients for the interacting or shifting terms (controllable via `which_param = c("shifting", "interacting", "autoregressive")`).
- `predict()` returns in-sample predictions when `newdata` is not supplied. The supported types are `"trafo"`, `"pdf"`, `"cdf"`, `"interaction"`, `"shift"`, `"terms"`. When `newdata` is supplied, predictions are evaluated at the response, if it is contained in `newdata`. The response can be omitted from `newdata` to predict the whole conditional distribution. Then, predictions are evaluated on a grid of length `K`, which is automatically generated based on the response's support in the training data set. A custom grid of response values can be supplied via `q`, which overwrites `K`.
- `logLik()` evaluates in- or out-of-sample log-likelihoods. This can be useful for model criticism and evaluating predictive performance, respectively. The argument `convert_fun` controls how the individual NLL contributions are summarized. The default is `function(x) = -sum(x)` to compute the log-likelihood. Other common choices include `identity` to obtain the individual NLL contributions, or `mean` to get the average NLL.
- `plot()` by default plots smooth components in the `shifting` formula part. Data for plotting can be obtained by setting `only_data = TRUE`. Smooth terms in `interacting` can be plotted by setting `which_param = "interacting"`. In the same manner as in `predict`, densities evaluated in-sample (`type = "pdf"`), CDFs (or probability integral transforms, with `type = "cdf"`), and transformation functions (`type = "trafo"`) can be obtained. When omitting the response from `newdata`, the whole density, cumulative distribution, or transformation function can be plotted.
- `print()` prints a brief summary of the DCTM including coefficients of additive linear and smooth terms in `shifting`. Setting `with_baseline = TRUE` also prints coefficients of linear and smooth terms in `interacting`. The `print_model` argument toggles whether the **keras** summary of the DCTM should be printed in addition.

Methods for '`dtEnsemble`' objects

Methods `coef()` and `predict()` of '`deeptrafo`' objects take the same arguments as their '`deeptrafo`' counterparts. The output is returned for all ensemble members. Likewise, `logLik()` returns the processed NLL contributions for individual ensemble members, their average, and the transformation ensemble.

3. Application: Binary classification

In this application, we use the `movies` dataset and fit four different models with the goal to predict the binary response `action` (0: non-action movie, 1: action movie, defined in the next code chunk), which encodes whether a movie is an action movie or not. The model `m_0` is unconditional; `m_tab` uses only one tabular predictor, `popularity`, as linear shift predictor; `m_text` uses only `texts` as an unstructured shift predictor; `m_semi` is a semi-structured model which uses both modalities as shift predictors. The purpose of the analysis is to show the potential gains in prediction performance that can be achieved when including the text data and learning an embedding, for which conventional statistical models would require extensive feature engineering. The models that do not include the text data could, in principle, also be fitted using `glm()` from the **MASS** (Ripley 2024) package and yield virtually the same results as **deeptrafo**.

First, we encode the binary response as an ordered factor allowing us to use the framework of ordinal neural network transformation models (Kook *et al.* 2022). This step is necessary because unordered factors are not supported by **deeptrafo**.

```
R> train$action <- ordered(train$genreAction)
R> test$action <- ordered(test$genreAction, levels = levels(train$action))
```

We then set up the formulas for the four models. The unconditional model is specified without any predictors and `1` on the right-hand side. Later, we will restrict this additional intercept to zero for identification (see `warmstart_weights` in the definition of `m_0`). For all other models, we remove the intercept directly by specifying `0 + ...` on the right-hand side.

```
R> fm_0 <- action ~ 1
R> fm_tab <- action ~ 0 + popularity
R> fm_text <- action ~ 0 + deep(texts)
R> fm_semi <- action ~ 0 + popularity + deep(texts)
```

Here, `deep` is the same neural network architecture with text embedding as in Section 2.1. We use a custom '`keras_model`' to which we can refer to by the name `"embd"` and wrap it in a function, which allows us to create multiple instances of the same model. This may be necessary in applications to make the resulting Python objects point to different copies in memory and avoid unintended re-use of already trained or modified models.

```
R> make_keras_model <- function() {
+    return(keras_model_sequential(name = "embd") |>
+    layer_embedding(input_dim = nr_words, output_dim = embedding_size) |>
+    layer_lstm(units = 50, return_sequences = TRUE) |>
+    layer_lstm(units = 50, return_sequences = FALSE) |>
+    layer_dropout(rate = 0.1) |> layer_dense(25) |>
+    layer_dropout(rate = 0.2) |>  layer_dense(5, name = "penultimate") |>
+    layer_dropout(rate = 0.3) |> layer_dense(1))
+  }
```

Next, we use `PolrNN()` to set up the different models with a standard logistic latent distribution. Models including text data are trained for ten epochs with early stopping and a patience of two, and the weights from the epoch with the best validation loss are restored. The unconditional and tabular-only models are trained full-batch and without validation split until converging to the minimum since convexity of the problem implies a unique solution.

Besides the simple text embedding that is trained from scratch, we also present how to use a pre-trained `word2vec` embedding in Appendix E.

3.1. Unconditional model

For the unconditional model, the intercept is fixed to zero via `warmstart_weights` to ensure identification. The details explaining the next code chunk can be found in Appendix C.

```
R> m_0 <- PolrNN(fm_0, data = train, optimizer = optimizer_adam(
+    learning_rate = 1e-2, decay = 1e-4), weight_options = weight_control(
+    general_weight_options = list(trainable = FALSE, use_bias = FALSE),
+    warmstart_weights = list(list(), list(), list("1" = 0))))
R> fit(m_0, epochs = 3e3, validation_split = 0, batch_size = length(
+    train$action), verbose = FALSE)
```

The unconditional model `m_0` has one parameter which estimates the log-odds of a movie belonging to a non-action genre without any predictors. The estimated intercept parameter, given by `coef(m_0, which = "interacting")`, corresponds to the single (fix) value of the transformation function h for a binary response (see Figure 1). The code chunk below shows that the estimated intercept agrees with the observed log-odds of a movie belonging to a non-action genre up to numerical inaccuracies.

```
R> all.equal(unlist(unname(coef(m_0, which = "interacting"))),
+    qlogis(mean(train$action == 0)), tol = 1e-6)

[1] TRUE
```

We can obtain the unconditional log-odds also using `predict()` with `type = "trafo"`. From the estimated log-odds we can determine the probability for a movie to belong to a non-action genre which matches the prevalence of non-action movies in the train set. The prevalence of non-action movies can also be computed directly by using the `predict` function and setting the argument `type = "pdf"` and supplying `action = 0` in `newdata`.

3.2. Tabular-only model

Next, we set up and fit `m_tab` including `popularity` as a linear shift predictor.

```
R> m_tab <- PolrNN(fm_tab, data = train, optimizer = optimizer_adam(
+    learning_rate = 0.1, decay = 1e-4))
R> fit(m_tab, epochs = 1e3, batch_size = length(train$action),
+    validation_split = 0, verbose = FALSE)
```

We obtain the estimated linear shift parameter $\hat{\beta}$ of `m_tab` by `coef(m_tab, which_param = "shifting")`. Here, the odds for a movie to belong to genre action change by the factor $\exp(-\hat{\beta})$

```
R> exp(-unlist(coef(m_tab, which = "shifting")))

popularity
      1.54
```

when the predictor popularity increases by one unit. Without flipping the sign, the coefficient $\hat{\beta}$ represents a log-odds ratio for a movie belonging to a non-action genre compared to genre action upon a one-unit change in popularity. Thus, the interpretation of $\hat{\beta}$ depends on the parameterization of the model, in particular, the sign of the shifting predictor. In **deeptrafo**, the shifting predictor is consistently parameterized with a plus sign for all models, which may differ from other implementations of the same model type (e.g., generalized linear models in **MASS**, or TMs in **tram**).

3.3. Text-only model

We now define and fit m_text including only the tokenized movie reviews.

```
R> embd <- make_keras_model()
R> m_text <- PolrNN(fm_text, data = train, list_of_deep_models = list(
+    deep = embd), optimizer = optimizer_adam(learning_rate = 1e-4))
R> fit(m_text, epochs = 10, callbacks = list(callback_early_stopping(
+    patience = 2, restore_best_weights = TRUE)), verbose = FALSE)
```

Analogously to smooth partial effects, the differences between two shift estimates resulting from two different text inputs can still be interpreted as log odds-ratios.

We now have a closer look at what the embd_mod has learned. The network takes as input the words (encoded as indices). Here, we use a vocabulary (all words in the data set) of 10000 words and limit each review text to a size of 100 words. Review texts which are shorter are prepended with zeros, longer movie descriptions are cut after 100 words. All punctuation is removed. The layer_embedding learns to embed the word indices into an embedding_size-dimensional representation. The resulting word embeddings of the text are the input sequence to an LSTM layer with a 50-dimensional memory state. The second LSTM layer outputs the 50-dimensional state after the last word in the text, which is then further processed by a fully connected neural network including dropout to prevent overfitting.

We can now use the trained embd to extract and analyze the derived latent features of the embedding of single words or whole texts. We can obtain the embedding of a single word as the output of layer_embedding(). If we use a whole review as input, the latent features in the layer "penultimate" correspond to a five-dimensional representation of the text embedding processed by subsequent layers.

Figure 7 shows the first two components of a principle component analysis (PCA) applied to the word embedding (left) and to the features learned in the penultimate layer for whole reviews (right). The left plot reveals, that words hinting at an action movie, have a similar embedding, and are separated from words that are rather representative of a romance movie. The plot on the right of Figure 7 confirms that the features derived from the embedding are tailored to discriminate action movies from other genres since latent features of action movies cluster together and are fairly well separated from romantic movies.

3.4. Semi-structured model

Finally, we set up the most complex model m_semi which takes both data modalities as input. To achieve efficient training of the tabular part and avoid overfitting of the embedding network emdb_semi we use two different learning rates for the structured and unstructured

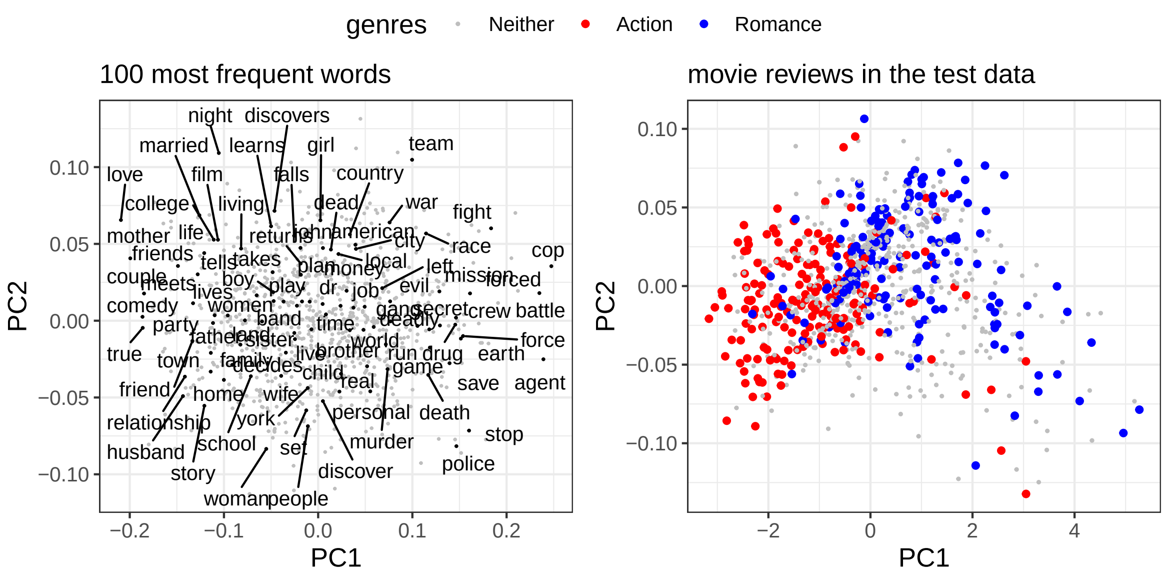

Figure 7: The first two principal components of the "`embedding`" layer for single words (left) and the lower-dimensional representation learned in "`penultimate`" (right) for whole movie reviews in the test data. On the left, the PCA is computed on the word embedding of the 1,000 most frequent words (we display only the 100 most frequent, black dots), and on the right based on the low-dimensional representation of the embedding of the 888 full movie reviews contained in the test data.

part of the model. Specifically, we optimize the intercept (with name "`ia_1__2`") and tabular shift predictor (with name "`popularity_3`") with a higher learning rate, than the embedding model ("`embd`"). In the embedding model, some layers are named explicitly, the names for the other components can be obtained from the '`keras_model`' summary by initializing and calling `print(m_semi, print_model = TRUE)`.

```
R> embd_semi <- make_keras_model()
R> optimizer <- function(model) {
+    optimizers_and_layers <- list(
+      tuple(optimizer_adam(learning_rate = 1e-2),
+      get_layer(model, "ia_1__2")),
+      tuple(optimizer_adam(learning_rate = 1e-2),
+      get_layer(model, "popularity_3")),
+      tuple(optimizer_adam(learning_rate = 1e-4),
+      get_layer(model, "embd")))
+    multioptimizer(optimizers_and_layers)
+ }
R> m_semi <- PolrNN(fm_semi, data = train, list_of_deep_models = list(
+    deep = embd_semi), optimizer = optimizer)
R> fit(m_semi, epochs = 10, callbacks = list(callback_early_stopping(
+    patience = 2, restore_best_weights = TRUE)), verbose = FALSE)
```

3.5. Model comparison

Comparing the prediction performance of the models (measured in terms of NLL) indicates that mainly the text modality contains information for separating action movies from other genres. However, for a more reliable assessment of this statement, the training schedule should be optimized further. We compute 95% bootstrap confidence intervals as a simple uncertainty measure for the test NLL. In Appendix E, we illustrate how to use pre-trained embeddings with a shallow and deeper neural network architecture and obtain comparable results in terms of out-of-sample NLL. Using pre-trained embeddings may reduce computation times and yield comparable predictions, especially when the training sample size is small (Goodfellow *et al.* 2016).

```
R> bci <- function(mod) {
+    lli <- logLik(mod, newdata = test, convert_fun = identity)
+    bt <- boot(lli, statistic = \(x, d) mean(x[d]), R = 1e4)
+    btci <- boot.ci(bt, conf = 0.95, type = "perc")$percent[1, 4:5]
+    c("nll" = mean(lli), "lwr" = btci[1], "upr" = btci[2])
+ }
R> mods <- list("unconditional" = m_0, "tabular only" = m_tab,
+   "text only" = m_text, "semi-structured" = m_semi)
R> do.call("cbind", lapply(mods, bci))
```

```
    unconditional tabular only text only semi-structured
nll         0.531        0.516     0.437           0.423
lwr         0.501        0.486     0.390           0.372
upr         0.562        0.549     0.486           0.478
```

Like `m_tab` the model `m_semi` estimates a linear shift parameter for `popularity` which can also be interpreted as a (conditional) log odds-ratio. The parameter goes in the same direction as in `m_tab` but has a reduced absolute value and is now interpretable as a conditional log-odds ratio because the text information that is now additionally accounted for.

```
R> c("tabular only" = unlist(unname(coef(m_tab))),
+    "semi-structured" = unlist(unname(coef(m_semi))))
```

```
   tabular only semi-structured
          -0.43           -0.32
```

The presented case study is meant to showcase some functionality of the package **deeptrafo** for binary responses. A `PolrNN` model for an ordinal response that has K levels and yields $K-1$ values for a discrete transformation function (see Figure 1) can be interpreted analogously, e.g., linear shift terms are still interpreted as log odds-ratios (for details and more examples, see Kook *et al.* 2022).

4. Application: Autoregressive transformation models

We now return to ATMs, first discussed in Section 1.2. One special form of ATMs are AT(p) models. AT(p) models assume a linear impact of the transformed values of $\mathcal{F}_{t,t-p}$ on the scale of h. Because the transformation is the same as for the response, AT(p) models thus learn a joint transformation of the response and its lags. For an illustration of transformation models applied to time series data, the `temperature` dataset is used. We aim to estimate the conditional distribution of the monthly mean maximum temperature in degrees Celsius (°C) in Melbourne (Australia) between January 1971 and December 1990. A descriptive analysis of the time series shows a strong seasonal pattern. This motivates the application of a flexible approach that allows modeling the quickly changing moments of the conditional distribution over time.

In the following, we compare three different forms of autoregressive transformation models. The most flexible model (ATM) includes the lags as interacting predictors and transformed lags in the shift term. The AT(3) model only includes the transformed lags in the shift term. Lastly, the naive `ColrNN` model (Colr) includes the lags as an additive linear term. In addition, every model contains a shift effect for `month`. The ATM and AT(3) model can currently only be fitted using **deeptrafo**, whereas the other two models could be fitted using conventional TMs implemented in **tram**. We compare the three models based on their estimated transformation functions and conditional densities. We start by creating a factor variable `month` for the calendar month as well as the lags Y_{t-p}, $p = 1, 2, 3$ denoted by `y_lag_<p>` for including raw additive lags. AT(p) lags are included using the internal `atplag()` processor.

```
R> lags <- c(paste0("y_lag_", 1:p, collapse = "+"))
```

The formula for the ATM model is given as follows. We include all three lags as interacting predictors on the left-hand side of the formula and specify the `atplag`s on the right-hand side.

```
R> (fm_atm <- as.formula(paste0("y |", lags, "~ 0 + month + atplag(1:p)")))

y | y_lag_1 + y_lag_2 + y_lag_3 ~ 0 + month + atplag(1:p)
```

ATP lags can be conveniently included in the formula by specifying the lags inside `atplag()`. For the AT(3) model, we include the transformed lags in the shift but not in the interacting term.

```
R> (fm_atp <- y ~ 0 + month + atplag(1:p))

y ~ 0 + month + atplag(1:p)
```

The third model (Colr) we compare is a `ColrNN` model which includes the raw lags in an additive shift term.

```
R> (fm_colr <- as.formula(paste0("y ~ 0 + month + ", lags)))

y ~ 0 + month + y_lag_1 + y_lag_2 + y_lag_3
```

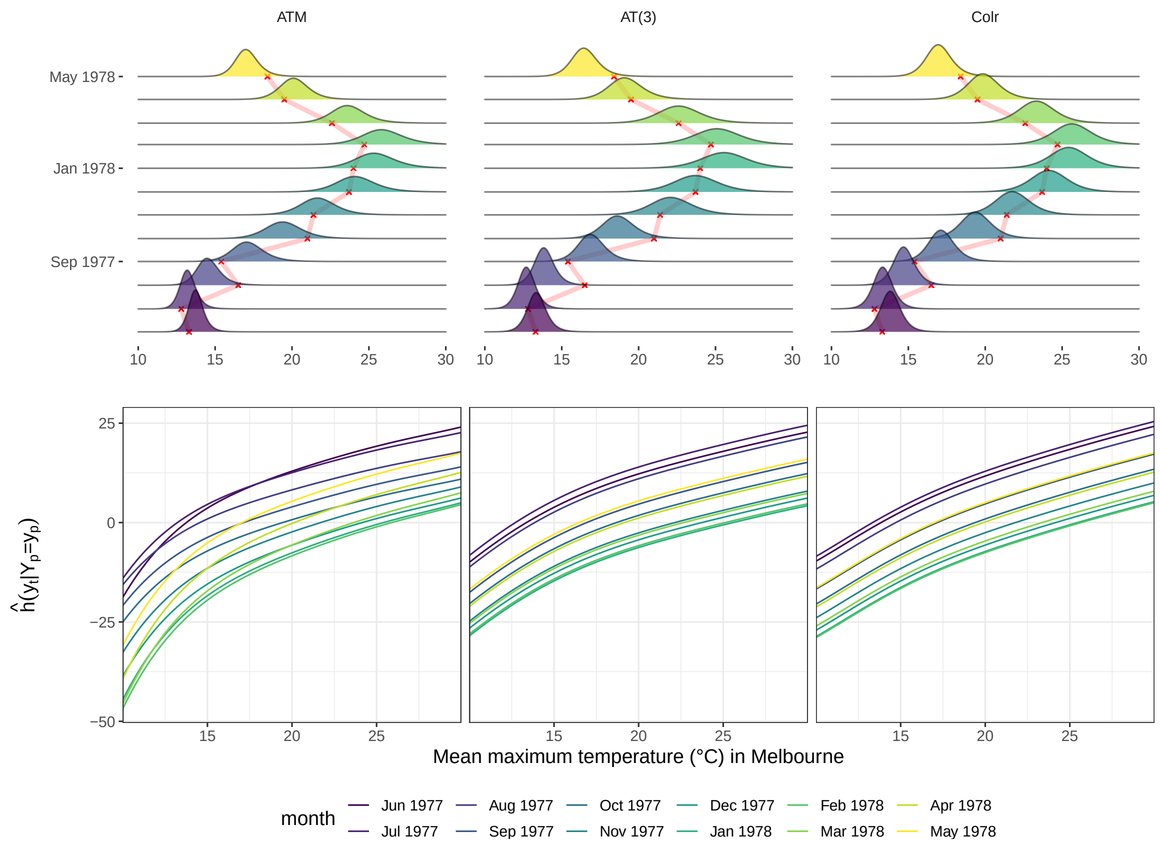

Figure 8: Estimated conditional densities (top row) of monthly temperature records between June 1983 and May 1984, based on the ATM model (left), the AT(3) model (center) and the Colr model (right). The observed values across this time span are depicted in red. The plots in the bottom row show the corresponding estimated conditional transformation functions.

After pre-processing, the `temperature` dataset is saved in `d_ts`. We fix the support of the response to `min_supp = 10` and `max_supp = 30` and specify Bernstein polynomials of order `P = 6`. We use `ColrNN()` to specify all models. ATM and AT(3) include `atplags` and the third model, Colr, does not.

```
R> mod_fun <- function(fm, d) ColrNN(fm, data = d,
+    trafo_options = trafo_control(order_bsp = P,
+    support = c(min_supp, max_supp)), tf_seed = 1,
+    optimizer = optimizer_adam(learning_rate = 0.01))
R> mods <- lapply(list(fm_atm, fm_atp, fm_colr), mod_fun)
```

After defining the models, we proceed with training all three models. In addition, we include callbacks to reduce the learning rate upon encountering a plateau in the training loss, to ensure convergence of the optimization procedure.

```
R> fit_fun <- function(m) m |> fit(epochs = ep, callbacks = list(
+   callback_early_stopping(patience = 20, monitor = "val_loss"),
+   callback_reduce_lr_on_plateau(patience = 5)), batch_size = nrow(d_ts_lag),
+   verbose = FALSE)
R> lapply(mods, fit_fun)
```

We compare the in-sample log-likelihood for the three models for the subset of data between June 1977 and May 1978 in `t_idx`.

```
R> t_span_one <- seq(as.Date("1977-03-01"), as.Date("1978-05-01"),
+    by = "month")
R> ndl <- d_ts[d_ts$time %in% t_span_one]
R> t_span_two <- seq(as.Date("1977-06-01"), as.Date("1978-05-01"),
+    by = "month")
R> ndl_lag <- d_ts_lag[d_ts_lag$time %in% t_span_two]
R> structure(unlist(c(lapply(mods[1:2], logLik, newdata = ndl),
+    lapply(mods[3], logLik, newdata = ndl_lag))), names =
+    c("ATM", paste0("AT(", p, ")"), "Colr"))

  ATM AT(3)  Colr
-19.5 -22.5 -20.1
```

The comparison shows that the Colr and the ATM model fit similarly well compared to the slightly less favorable fit of the AT(3) model. A visual inspection of the estimated conditional densities depicted in Figure 8 shows similar results for all three estimation methods. In summary, the ATM class may be favored over naive TMs (Colr) in the time series domain for its autoregressive structural assumption, i.e., lags entering in a transformed way, identical to the transformation of y_t (see Rügamer *et al.* 2023a).

5. Conclusion

With **deeptrafo**, we introduce the first R package for fitting a broad class of distributional regression models with a neural network back-end. Package **deeptrafo** combines the advantages of transformation models, i.e., flexible distribution-free, yet interpretable models for conditional distributions, with the advantages of neural network-based machine learning, which scales well for large or non-tabular datasets. The intuitive formula interface allows users familiar with packages such as **stats** (R Core Team 2024), **MASS**, **tram**, **survival** (Therneau 2024), **mgcv**, and others to easily adapt their workflow to neural networks and more complex datasets out-of-the-box.

Users can supply custom basis functions, loss functions, optimization routines and neural network architectures to adapt and extend functionalities from **deeptrafo** to problems in which the goal is learning a conditional cumulative distribution function. We illustrate **deeptrafo** with tabular and text, as well as time series data with count, discrete, and continuous outcomes, which are all handled in a unified way. We demonstrate how custom neural network architectures and optimizers can be used, and how to tune, evaluate, and interpret DCTMs.

Applying neural network-based models to analyze text or image data typically comes with higher flexibility but also larger computational costs compared to more conventional statistical models. We demonstrate how pre-trained text embeddings can be used to obtain competitive results to training an embedding from scratch and reduce the computational and the environmental burden.

Acknowledgments

We thank Sandra Siegfried for her valuable comments on the manuscript. The research of LK was supported by the Swiss National Science Foundation (Grant No. 214457). LK conducted part of this work at the University of Copenhagen, University of Zurich and Zurich University of Applied Sciences. The research of DR has been partially supported by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The research of LK and BS was supported by Novartis Research Foundation (FreeNovation 2019) and by the Swiss National Science Foundation (Grant No. S-86013-01-01 and S-42344-04-01). The research of OD has been partially supported by BMBF under Grant No. 01IS19083A. The authors of this work take full responsibilities for its content.

References

Allaire JJ, Chollet F (2024). ***keras**: R Interface to 'Keras'*. doi:10.32614/CRAN.package.keras. R package version 2.15.0.

Allaire JJ, Tang Y (2024). ***tensorflow**: R Interface to 'TensorFlow'*. doi:10.32614/CRAN.package.tensorflow. R package version 2.16.0.

Baumann PFM, Hothorn T, Rügamer D (2021). "Deep Conditional Transformation Models." In *Machine Learning and Knowledge Discovery in Databases. Research Track*, pp. 3–18. Springer-Verlag. doi:10.1007/978-3-030-86523-8_1.

Collett D (2015). *Modelling Survival Data in Medical Research*. Chapman & Hall/CRC. doi:10.1201/b18041.

Dozat T (2016). "Incorporating Nesterov Momentum into Adam." In *ICLR 2016 Workshop*.

Fahrmeir L, Kneib T, Lang S, Marx B (2013). *Regression: Models, Methods and Applications*. Springer-Verlag, Berlin. doi:10.1007/978-3-642-34333-9.

Goodfellow I, Bengio Y, Courville A (2016). *Deep Learning*. MIT Press.

Hamilton JD (2020). *Time Series Analysis*. Princeton university press.

Hothorn T (2020a). "Most Likely Transformations: The **mlt** Package." *Journal of Statistical Software*, **92**(1), 1–68. doi:10.18637/jss.v092.i01.

Hothorn T (2020b). "Transformation Boosting Machines." *Statistics and Computing*, **30**(1), 141–152. doi:10.1007/s11222-019-09870-4.

Hothorn T (2023). ***trtf**: Transformation Trees and Forests*. doi:10.32614/CRAN.package.trtf. R package version 0.4-2.

Hothorn T, Barbanti L, Siegfried S (2024). ***tram**: Transformation Models*. doi:10.32614/CRAN.package.tram. R package version 1.0-6.

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.

Hothorn T, Möst L, Bühlmann P (2018). "Most Likely Transformations." *Scandinavian Journal of Statistics*, **45**(1), 110–134. doi:10.1111/sjos.12291.

Hyndman R, Yang Y (2022). ***tsdl**: Time Series Data Library.* R package version 0.1.0, URL https://finyang.github.io/tsdl/.

Kaggle (2017). "The Movies Dataset." URL https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset.

Kingma DP, Ba JL (2015). "Adam: A Method for Stochastic Optimization." In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR. doi:10.48550/arxiv.1412.6980.

Kook L (2024). "**tramvs**: Optimal Subset Selection in Transformation Models." doi:10.32614/CRAN.package.tramvs. R package version 0.0-6.

Kook L, Baumann PFM, Rügamer D (2024). ***deeptrafo**: Fitting Deep Conditional Transformation Models.* doi:10.32614/CRAN.package.deeptrafo. R package version 1.0-0.

Kook L, Götschi A, Baumann PFM, Hothorn T, Sick B (2022). "Deep Interpretable Ensembles." *arXiv 2205.12729*, arXiv.org E-Print Archive. doi:10.48550/arxiv.2205.12729.

Kook L, Herzog L, Hothorn T, Dürr O, Sick B (2022). "Deep and Interpretable Regression Models for Ordinal Outcomes." *Pattern Recognition*, **122**, 108263. doi:10.1016/j.patcog.2021.108263.

Kook L, Hothorn T (2021). "Regularized Transformation Models: The **tramnet** Package." *The R Journal*, **13**(1), 581–594. doi:10.32614/rj-2021-054.

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.

Mikolov T, Chen K, Corrado G, Dean J (2013). "Efficient Estimation of Word Representations in Vector Space." doi:10.48550/arxiv.1301.3781.

R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna. URL https://www.R-project.org/.

Rehurek R, Sojka P (2011). "**gensim** – Python Framework for Vector Space Modelling." *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, **3**(2).

Ripley BD (2024). ***MASS**: Support Functions and Datasets for Venables and Ripley's MASS.* doi:10.32614/CRAN.package.MASS. R package version 7.3-61.

Rügamer D (2024). "Scalable Higher-Order Tensor Product Spline Models." In S Dasgupta, S Mandt, Y Li (eds.), *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, volume 238 of *Proceedings of Machine Learning Research*, pp. 1–9. PMLR.

Rügamer D, Baumann PF, Kneib T, Hothorn T (2023a). "Probabilistic Time Series Forecasts with Autoregressive Transformation Models." *Statistics and Computing*, **33**(2), 37. doi:10.1007/s11222-023-10212-8.

Rügamer D, Bender A, Wiegrebe S, Racek D, Bischl B, Müller CL, Stachl C (2023b). "Factorized Structured Regression for Large-Scale Varying Coefficient Models." In MR Amini, S Canu, A Fischer, T Guns, P Kralj Novak, G Tsoumakas (eds.), *Machine Learning and Knowledge Discovery in Databases*, pp. 20–35. Springer-Verlag, Cham.

Rügamer D, Kolb C, Fritz C, Pfisterer F, Bischl B, Shen R, Bukas C, Thalmeier D, Baumann P, Kook L, Klein N, Müller C (2023c). "**deepregression**: A Flexible Neural Network Framework for Semi-Structured Deep Distributional Regression." *Journal of Statistical Software*, **105**(1), 1–31. doi:10.18637/jss.v105.i02.

Rügamer D, Kolb C, Klein N (2024). "Semi-Structured Distributional Regression." *The American Statistician*, **78**(1), 88–99. doi:10.1080/00031305.2022.2164054.

Sick B, Hothorn T, Dürr O (2021). "Deep Transformation Models: Tackling Complex Regression Problems with Neural Network Based Transformation Models." In *25th International Conference on Pattern Recognition (ICPR)*. IEEE. doi:10.1109/icpr48806.2021.9413177.

Siegfried S, Hothorn T (2020). "Count Transformation Models." *Methods in Ecology and Evolution*, **11**(7), 818–827. doi:10.1111/2041-210x.13383.

Siegfried S, Kook L, Hothorn T (2024). "Distribution-Free Location-Scale Regression." *The American Statistician*, **77**(4), 345–356. doi:10.1080/00031305.2023.2203177.

Tamási B, Hothorn T (2021). "**tramME**: Mixed-Effects Transformation Models Using Template Model Builder." *The R Journal*, **13**(2), 398–418. doi:10.32614/rj-2021-075.

Therneau TM (2024). ***survival**: Survival Analysis*. doi:10.32614/CRAN.package.survival. R package version 3.7-0.

Tieleman T, Hinton G (2012). "Lecture 6.5-RMSprop: Divide the Gradient by a Running Average of Its Recent Magnitude." *COURSERA: Neural Networks for Machine Learning*, **4**(2), 26–31.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL https://www.python.org/.

Varadhan R (2023). ***alabama**: Constrained Nonlinear Optimization*. doi:10.32614/CRAN.package.alabama. R package version 2023.1.0.

Varadhan R, Gilbert P (2009). "**BB**: An R Package for Solving a Large System of Nonlinear Equations and for Optimizing a High-Dimensional Nonlinear Objective Function." *Journal of Statistical Software*, **32**(4), 1–26. doi:10.18637/jss.v032.i04.

Wood S (2023). ***mgcv**: Mixed GAM Computation Vehicle with Automatic Smoothness Estimation*. doi:10.32614/CRAN.package.mgcv. R package version 1.9-1.

A. Additional details

In the appendix, we describe how **deeptrafo** handles censored responses (Appendix B), how the user can warmstart and fix weights of interacting and shifting terms (Appendix C), and how to include custom basis functions (Appendix D). We demonstrate how to use pre-trained embeddings (Appendix E) and give details on the most commonly used options for optimization (Appendix F). In addition, we describe an alternative formula interface (Appendix G) and show how to use **deeptrafo** for large tabular datasets (Appendix H).

B. Handling censored responses

Package **deeptrafo** detects the type of response automatically. However, the user may specify the type explicitly via `response_type` in `deeptrafo()` and all alias/wrapper functions. Allowed types of responses are continuous, count, survival, ordered (including binary). Censored responses can be supplied as '`Surv`' objects. Internally, ordered and count responses are treated as censored. For instance, the two observations `c(0L, 1L)` with `response_type = "count"` are internally represented as left- and interval-censored, respectively.

```
R> deeptrafo::response(y = c(0L, 1L))
```

```
     cleft exact cright cinterval
[1,]     1     0      0         0
[2,]     0     0      0         1
attr(,"type")
[1] "count"
```

C. Warmstarting and fixing weights

Warmstarting and fixing weights may be important in numerical experiments, for fine-tuning parts of the models, or transfer learning (Goodfellow *et al.* 2016). In **deeptrafo**, the user can supply a '`keras_model`', as returned, for instance, by `keras_model_sequential()`. When defining the model, **keras** specific arguments for controlling weight initialization can be used, as shown below.

```
R> nn <- keras_model_sequential() |>
+   layer_dense(input_shape = 1L, units = 3L, activation = "relu",
+   use_bias = FALSE, kernel_initializer = initializer_constant(
+   value = 1))
R> unlist(get_weights(nn))
```

```
[1] 1 1 1
```

To warmstart or fix coefficients of the interacting or shifting part of a DCTM, the `weight_options` argument in `deeptrafo()` can supplied with the output of `weight_control()`, which, in addition to others, takes the same arguments as the **keras** layers above.

```
R> args(weight_control)

function (specific_weight_options = NULL, general_weight_options = list(
    activation = NULL, use_bias = FALSE, trainable = TRUE,
    kernel_initializer = "glorot_uniform", bias_initializer = "zeros",
    kernel_regularizer = NULL, bias_regularizer = NULL,
    activity_regularizer = NULL, kernel_constraint = NULL,
    bias_constraint = NULL), warmstart_weights = NULL,
    shared_layers = NULL)
NULL
```

Below, we warmstart the shift coefficient for a `PolrNN` model. Here, `warmstart_weights` takes a list with three components, of which the first two control the weights of the interacting predictor and the last the weights of the shift predictor. The weights can be referred to by the name of the covariate, i.e., `"temp" = 0`.

```
R> data("wine", package = "ordinal")
R> mw <- deeptrafo(
+   response ~ 0 + temp,
+   data = wine, weight_options = weight_control(warmstart_weights = list(
+     list(), list(), list("temp" = 0))))
R> unlist(coef(mw))

$temp
         [,1]
tempwarm    0
```

The three lists correspond to the three formula components `response`, `interacting`, and `shifting`. The list corresponding to the response is always empty, since it does not contain any parameters. In case there is no interacting predictor, the second list corresponds to the parameters of the basis function of the response, i.e., the intercept function. In case there is no shift term, an intercept is set up which can be referred to as `"1"` and frozen as illustrated in the main text (Section 3). In the example above, we warmstart weights of a component in the shift term and supply two empty lists for the other components.

D. Including custom basis functions

Linear, log-linear, and Bernstein bases, as used by **deeptrafo**, require (linear) inequality constraints on their parameters. Internally, these constraints are handled in `trafo_control()`, by supplying an **keras** layer, which transforms the weights for the interacting predictor appropriately. In **deeptrafo**, the implemented bases are `"bernstein"`, `"ordered"`, and `"shiftscale"`. The former two require $\vartheta_{jP+1)} \leq \vartheta_{jP+2} \leq \cdots \leq \vartheta_{jP+P}$, $l = 0, \dots, L-1$ for $\boldsymbol{b}(\boldsymbol{x}) \in \mathbb{R}^L$ and degree $P-1$ Bernstein basis or ordered response with $P+1$ levels. The shift-scale basis requires only $\vartheta_1 > 0$ in $y \mapsto \vartheta_0 + \vartheta_1 y$.

The user can now supply custom basis functions as shown below. First, the basis (`linear_basis`) and its derivative (`linear_basis_prime`) are defined. Afterwards, the constraints on the parameters are defined using Python- and **tensorflow**-specified constructs (`tf$...`).

```
R> linear_basis <- function(y) {
+    ret <- cbind(1, y)
+    if (NROW(ret) == 1)
+      return(as.vector(ret))
+    ret
+  }
R> linear_basis_prime <- function(y) {
+    ret <- cbind(0, rep(1, length(y)))
+    if (NROW(ret) == 1)
+      return(as.vector(ret))
+    ret
+  }
R> constraint <- function(w, bsp_dim) {
+    w_res <- tf$reshape(w, shape = list(bsp_dim, as.integer(nrow(w) /
+    bsp_dim)))
+    w1 <- tf$slice(w_res, c(0L, 0L), size = c(1L, ncol(w_res)))
+    wrest <- tf$math$softplus(tf$slice(w_res, c(1L, 0L), size = c(
+    as.integer(nrow(w_res) - 1), ncol(w_res))))
+    w_w_cons <- k_concatenate(list(w1, wrest), axis = 1L)
+    tf$reshape(w_w_cons, shape = list(nrow(w), 1L))
+  }
R> tfc <- trafo_control(
+    order_bsp = 1L,
+    y_basis_fun = linear_basis,
+    y_basis_fun_prime = linear_basis_prime,
+    basis = constraint
+  )
```

We can now compare our re-implementation of a transformation model with linear basis against `Lm()` from **tram**. To efficiently fit DCTMs for small tabular datasets, we recommend full-batch (i.e., batch size n) training with a large learning rate (0.01) and either decay or callbacks for reducing the learning on validation loss plateaus.

```
R> library("tram")
R> set.seed(1)
R> n <- 1e3
R> d <- data.frame(y = 1 + rnorm(n), x = rnorm(n))
R> m <- deeptrafo(y ~ 0 + x, data = d, trafo_options = tfc,
+    optimizer = optimizer_adam(learning_rate = 1e-2),
+    latent_distr = "normal")
R> fit(m, batch_size = n, epochs = 5e3, validation_split = NULL,
+    callbacks = list(callback_reduce_lr_on_plateau(monitor = "loss")),
+    verbose = FALSE)
R> abs(unlist(coef(m)) - coef(Lm(y ~ x, data = d)))

      x
0.00017
```

E. Binary classification with pre-trained embeddings

As large pre-trained language models become more and more practice in natural language processing, we show an alternative way to fit a DCTM using a pre-trained embedding called `word2vec` (Mikolov, Chen, Corrado, and Dean 2013). The embedding is provided by Google and can be downloaded from their servers. Due to its corpus size, the embedding file is multiple Gigabytes large. After storing the embedding in the `./Data/` folder, we can load the embedding using the **gensim** Python library (Rehurek and Sojka 2011) and transform every word in the training dataset into a vector in the embedding space.

```
R> embedding_dim <- 300
R> if (file.exists("word2vec_embd_matrix.RDS")) {
R>  embedding_matrix <- readRDS("word2vec_embd_matrix.RDS")
R>  vocab_size <- nrow(embedding_matrix)
R> } else {
R>  gensim <- import("gensim")
R>  model <- gensim$models$KeyedVectors$load_word2vec_format(
+     "../Data/GoogleNews-vectors-negative300.bin", binary = TRUE)
R>  vocab_size <- length(words$word)
R>  embedding_matrix <- matrix(0, nrow = vocab_size, ncol = embedding_dim)
R>  names_model <- names(model$key_to_index)
R>  for (i in 1:vocab_size) {
R>    word <- words$word[i]
R>    if (word %in% names_model) {
R>      embedding_matrix[i, ] <- model[[word]]
R>    }
R>  }
R> saveRDS(embedding_matrix, file = "word2vec_embd_matrix.RDS")
R> }
```

Having transformed the text data into vectors in the embedding space, we can use these in an embedding layer to define our model. We start with a shallow neural network that flattens the vectors for each word and learns a linear model for the resulting data matrix.

```
R> w2v_mod <- function(x) x |>
+    layer_embedding(input_dim = vocab_size, output_dim = embedding_dim,
+    weights = list(embedding_matrix), trainable = FALSE) |>
+    layer_flatten() |>
+    layer_dense(units = 1)
R> fm_w2v <- action ~ 0 + shallow(texts)
R> m_w2v <- deeptrafo(fm_w2v, data = train,
+    list_of_deep_models = list(shallow = w2v_mod),
+    optimizer = optimizer_adam(learning_rate = 1e-5))
R> dhist <- fit(m_w2v, epochs = 200, validation_split = 0.1,
+    batch_size = 32, callbacks = list(
+    callback_early_stopping(patience = 5)), verbose = FALSE)
R> bci(m_w2v)
```

```
  nll   lwr   upr 
0.523 0.494 0.553 
```

While stopping the training later may result in further model improvement, the negative log-likelihood values already indicate similar performance to the tabular-only model in Section 3.5, which yielded a test NLL of 0.52. We can improve this model by learning a deep neural network on top of the pre-trained embedding using 1D convolutions as follows.

```
R> w2v2_mod <- function(x) x |>
+    layer_embedding(input_dim = vocab_size, output_dim = embedding_dim,
+    weights = list(embedding_matrix), trainable = FALSE) |>
+    layer_conv_1d(filters = 128, kernel_size = 5, activation = 'relu') |>
+    layer_max_pooling_1d(pool_size = 5) |>
+    layer_conv_1d(filters = 128, kernel_size = 5, activation = 'relu') |>
+    layer_global_max_pooling_1d() |>
+    layer_dense(units = 128, activation = 'relu') |>
+    layer_dropout(rate = 0.5) |>
+    layer_dense(units = 1)
R> fm_w2v2 <- action ~ 0 + deep(texts)
R> m_w2v2 <- deeptrafo(fm_w2v2, data = train,
+    list_of_deep_models = list(deep = w2v2_mod),
+    optimizer = optimizer_adam(learning_rate = 1e-5))
R> dhist <- fit(m_w2v2, epochs = 200, validation_split = 0.1,
+    batch_size = 32, callbacks = list(
+    callback_early_stopping(patience = 5)), verbose = FALSE)
R> bci(m_w2v2)
```

```
  nll   lwr   upr 
0.510 0.486 0.534 
```

The test NLL resulting from the deeper architecture is lower than the one obtained from the shallow architecture above, but not as low as the one obtained using the text-only model in Section 3.5 of 0.44 (95% bootstrap confidence interval from 0.390 to 0.486).

F. Options for optimization

In deep learning, selecting an appropriate optimizer is crucial for model performance. If **deeptrafo** specifies a model with a deep predictor, exact optimization is not possible anymore and routines that require second- or higher-order derivatives of the objective are too expensive. Optimization is therefore done using first-order methods, in particular variations of stochastic gradient descent (SGD).

- Adam (Kingma and Ba 2015) is widely used due to its effectiveness across various applications, offering adaptive learning rates that handle sparse gradients efficiently. It is by far the most common choice and hence our default option. While Adam's default learning rate and momentum parameters can be changed, this must be done with care.

- Another option is SGD with momentum, preferred for optimizing large CNNs, with its momentum term accelerating gradients for faster convergence. In contrast to Adam, SGD with momentum does not come with a well-working default and hence often requires hyperparameter tuning for the momentum.
- Other notable options include RMSprop (Tieleman and Hinton 2012), designed for non-stationary objectives and noisy gradients, and Nadam (Dozat 2016), which combines elements of Adam and Nesterov accelerated gradient. These optimizers are, however, typically chosen for specific applications and should be used only after careful hyperparameter tuning.

Irrespective of the choice of the optimizer, semi-structured models such as DCTM typically have an imbalance in their optimization dynamic when including deep neural networks due to the large difference in the number of parameters for structured and unstructured model components. This can, in particular, lead to slow convergence of the structured model part. To mitigate this problem, users can use warm-starts as described in Appendix C, or use optimizers with different learning rates for the different model components as described in Section 3.4.

G. Alternative formula interface

Following ONTRAMs (ordinal neural network transformation models), introduced in Kook & Herzog *et al.* (2022), **deeptrafo** offers an alternative formula interface. Here, the user supplies a separate formula for the intercepts (before: `interacting`) and for the shift (before: `shifting`) and avoids using the pipe `|` on the left-hand-side of the formula. Internally, the formula is translated back into the form in (2). All other functionalities in the article carry over to ONTRAMs as well. The same interface for other than ordinal responses is implemented in `dctm()`.

```
R> dord <- data.frame(Y = ordered(sample.int(6, 100, TRUE)),
+    X = rnorm(100), Z = rnorm(100))
R> ontram(response = ~ Y, intercept = ~ X, shift = ~ 0 + s(Z, df = 3),
+    data = dord)
```

```
         Untrained ordinal outcome deep conditional transformation model


Interacting:  Y | X

Shifting:  ~0 + s(Z, df = 3)

Shift coefficients:
s(Z, df = 3)1 s(Z, df = 3)2 s(Z, df = 3)3 s(Z, df = 3)4 s(Z, df = 3)5
      -0.4760       -0.7326       -0.6233       -0.4061       -0.4309
s(Z, df = 3)6 s(Z, df = 3)7 s(Z, df = 3)8 s(Z, df = 3)9
      -0.5447        0.6729        0.7376        0.0947
```

H. Large factor models

We consider a large factor model with 10^6 observations and a factor variable with 10^3 levels. The standard implementation of `lm()` and `LmNN()` fail to process the data, due to evaluating the large model matrix. However, we can use `fac_processor()` from **safareg** to circumvent this issue and use mini-batch stochastic gradient descent to fit the model on a standard machine. Now, **deeptrafo** can fit large factor models for arbitrary types of responses and censoring.

```
R> set.seed(0)
R> library("safareg")
R> n <- 1e6
R> nlevs <- 1e3
R> X <- factor(sample.int(nlevs, n, TRUE))
R> Y <- (X == 2) - (X == 3) + rnorm(n)
R> d <- data.frame(Y = Y, X = X)
R> m <- LmNN(Y ~ 0 + fac(X), data = d, additional_processor = list(
+    fac = fac_processor))
R> fit(m, batch_size = 1e4, epochs = 20, validation_split = 0,
+    callbacks = list(callback_early_stopping("loss", patience = 3),
+    callback_reduce_lr_on_plateau("loss", 0.9, 2)))
R> bl <- unlist(coef(m, which = "interacting"))
R> - (unlist(coef(m))[1:5] + bl[1]) / bl[2]
```

```
fac(X)1 fac(X)2 fac(X)3 fac(X)4 fac(X)5
-0.0204  0.9986 -1.0156 -0.0249  0.0477
```

To compute the log-likelihood in models with vast amounts of data, specifying batch-wise computation avoids memory issues.

```
R> logLik(m, batch_size = 1e4)
```

```
[1] -1.42
```

Affiliation:

Lucas Kook
Vienna University of Economics and Business
Institute for Statistics and Mathematics
Welthandelsplatz 1, 1020 Vienna, Austria
E-mail: `lucas.kook@wu.ac.at`

David Rügamer
LMU Munich
Working Group Data Science
Department of Statistics
80539, Munich, Germany
E-mail: `david.ruegamer@stat.uni-muenchen.de`

Journal of Statistical Software `https://www.jstatsoft.org/`
published by the Foundation for Open Access Statistics `https://www.foastat.org/`
November 2024, Volume 111, Issue 10 *Submitted:* 2024-01-17
`doi:10.18637/jss.v111.i10` *Accepted:* 2024-05-28