Lee and Carter go Machine Learning: Recurrent Neural Networks

Ronald Richman* Mario V. Wüthrich†

Prepared for:
Fachgruppe "Data Science"
Swiss Association of Actuaries SAV

Version of August 22, 2019

Abstract

In this tutorial we introduce recurrent neural networks (RNNs), and we describe the two most popular RNN architectures. These are the long short-term memory (LSTM) network and gated recurrent unit (GRU) network. Their common field of application is time series modeling, and we demonstrate their use on a mortality rate prediction problem using data from the Swiss female and male populations.

Keywords. Recurrent neural network (RNN), long short-term memory (LSTM), gated recurrent unit (GRU), Lee–Carter (LC) model, mortality forecasting, feed-forward neural network (FNN)

0 Introduction and overview

This data analytics tutorial has been written for the working group "Data Science" of the Swiss Association of Actuaries SAV, see

`https://www.actuarialdatascience.org`

In this tutorial we give an introduction to recurrent neural networks (RNNs), and we describe the two most popular RNN architectures. These are the long short-term memory (LSTM) network and gated recurrent unit (GRU) network. Their common field of application is time series modeling, and we illustrate their use on a mortality rate prediction application. This mortality rate prediction example is based on data from the Human Mortality Database (HMD) [13], in particular, we have selected the Swiss population data, labeled "CHE" in the HMD, for our illustrative example. In the next section we describe this data and, as a warming up exercise, we are going to fit the Lee–Carter (LC) [8] model to the Swiss population data. The LC model is a standard actuarial model for mortality rate modeling and we use this LC model as benchmark for the RNN approaches.

*QED Actuaries and Consultants, ronald.richman@qedact.com
†RiskLab, Department of Mathematics, ETH Zurich, mario.wuethrich@math.ethz.ch

1 Swiss population mortality rates

1.1 Human Mortality Database

We start by introducing the data.[1] In this tutorial we study Swiss population mortality rates.

Listing 1: Swiss population mortality rates

```
1 Classes 'data.table' and 'data.frame':  13400 obs. of  7 variables:
2  $ Gender      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
3  $ Year        : int  1950 1950 1950 1950 1950 1950 1950 1950 1950 1950 ...
4  $ Age         : int  0 1 2 3 4 5 6 7 8 9 ...
5  $ Country     : chr  "CHE" "CHE" "CHE" "CHE" ...
6  $ imputed_flag: logi  "FALSE" "FALSE" "FALSE" "FALSE" ...
7  $ mx          : num  0.02729 0.00305 0.00167 0.00123 0.00101 ...
8  $ logmx       : num  -3.6 -5.79 -6.39 -6.7 -6.9 ...
```

The data has been compiled from the HMD [13]. We have applied some data pre-processing so that we can apply our models; we briefly describe this here. We select calendar years t from 1950 until 2016 (the latest calendar year available), and the ages x considered range from 0 to 99 years. An excerpt of this data is illustrated in Listing 1. The data is divided by genders, and the *raw mortality rates* are denoted by `mx`, see line 7 of Listing 1. Since we are going to model log-mortality rates the raw morality rates need to be strictly positive. In the original data from the HMD [13] there were 10 observations with `mx` $= 0$, these observations have been replaced by a mortality rate average over all available countries in the HMD (of the same age x, in the same calendar year t and of the same gender). These modified raw mortality rates received a flag, indicated on line 6 of Listing 1.

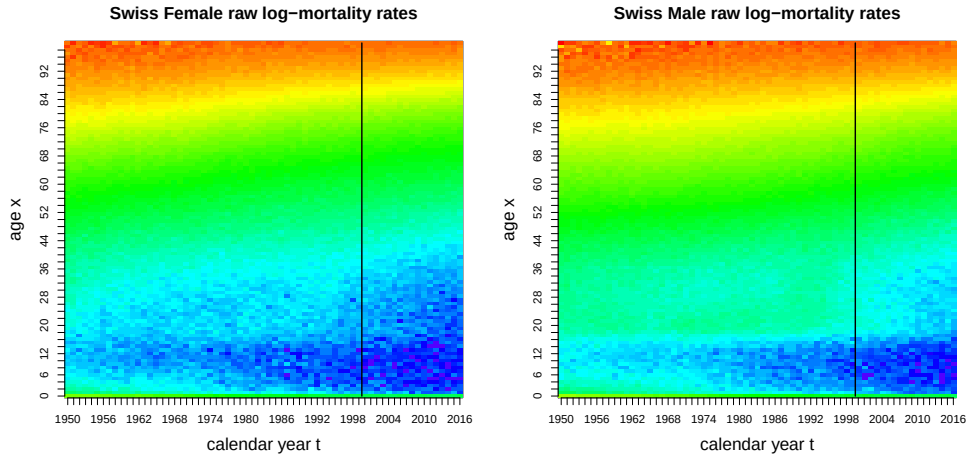

Figure 1: Heatmaps of the raw log-mortality rates of (lhs) Swiss female population and (rhs) Swiss male population from calendar years 1950 to 2016 and for ages 0 to 99 years.

In Figure 1 we illustrate the raw Swiss log-mortality rates for both genders, females on the left-hand side and males on the right-hand side. The color scale is identical in both plots, blue

[1] The data and the R code can be downloaded from https://github.com/JSchelldorfer/ActuarialDataScience

color means a small mortality rate and red color means a high mortality rate. These plots show the typical mortality rate improvements that we have encountered over the last decades (slightly upward diagonal structure of colors in heatmaps), with, on average, lower mortality rates for females than males. The vertical black line at calendar year 2000 shows our choice of training data $\mathcal{T}$ and validation data $\mathcal{V}$. The subsequent models will be learned using the data from calendar years $t = 1950, \ldots, 1999$ (called training data $\mathcal{T}$), and we will do an out-of-sample validation on the mortality rates from calendar years $2000, \ldots, 2016$ (called validation data $\mathcal{V}$).[2]

1.2 Lee–Carter model

We start our journey with a warming up example using the classical LC model [8] for mortality rate modeling. The LC model defines the force of mortality as

$$\log(m_{t,x}) = a_x + b_x k_t, \tag{1.1}$$

where $m_{t,x} > 0$ is the mortality rate in calendar year t for a person aged x, a_x is the average log-mortality rate at age x, b_x is a rate of change over time for a person aged x, and $(k_t)_t$ are time indexes for the annual mortality rate changes. Thus, for given age-dependent parameters $(a_x, b_x)_x$ we need to model a time series process $(k_t)_t$ to extrapolate mortality rates into the future. Remark that model (1.1) should be used separately for females and males, i.e., we have two separate models for both genders which are calibrated individually because their mortality behavior differs.[3] The original approach of Lee and Carter [8] has proposed to fit model (1.1) with the technique of singular value decomposition (SVD). Denote by $M_{t,x}$ the raw mortality rates (observations) for a given gender, see Listing 1. We center the raw log-mortality rates $\log(M_{t,x})$ for each age x as follows

$$\log(M^{\circ}_{t,x}) = \log(M_{t,x}) - \widehat{a}_x = \log(M_{t,x}) - \frac{1}{|\mathcal{T}|} \sum_{s \in \mathcal{T}} \log(M_{s,x}), \tag{1.2}$$

where $\mathcal{T}$ includes all calendar years used for model fitting (training), and the last identity defines the estimate $\widehat{a}_x$ for a_x. Based on these centered log-mortality rates we aim at finding optimal (parameter) values $(\widehat{b}_x)_x$ and $(\widehat{k}_t)_t$ as follows

$$\underset{(b_x)_x, (k_t)_t}{\arg\min} \sum_{t,x} \left(\log(M^{\circ}_{t,x}) - b_x k_t \right)^2, \tag{1.3}$$

where the summation runs over the chosen training data. This optimization problem can be solved with SVD. We define matrix $A = (\log(M^{\circ}_{t,x}))_{x,t}$. The first left singular vector of A multiplied with the first singular value provides us with an estimate $(\widehat{b}_x)_x$ for $(b_x)_x$, and the first right singular vector of A provides us with an estimate $(\widehat{k}_t)_t$ for $(k_t)_t$ (on the training data $t \in \mathcal{T}$). For a unique identifiability scheme one often normalizes the estimates $\widehat{a}_x$, $\widehat{b}_x$ and $\widehat{k}_t$ such that

$$\sum_x \widehat{b}_x = 1 \qquad \text{and} \qquad \sum_{t \in \mathcal{T}} \widehat{k}_t = 0.$$

[2] We use the notation $\mathcal{T}$ simultaneously for both the calendar years t and the corresponding observations. Of course, this is an abuse of notation, but it will always be clear from the context in which sense we use $\mathcal{T}$. The same applies for $\mathcal{V}$.

[3] There are multi-population models that can simultaneously model both genders and populations in different regions/countries. We do not introduce this literature here because the (simple) LC model is sufficient for our purposes. For a review of multi-population models we refer to [10].

Listing 2: Fitting LC with SVD using the R package `data.table` for initial calculations

```
1  train <- all_mort[Year<2000][Gender == gender]  # choose training data for given gender
2
3  ### fitting the LC model via SVD
4  train[,ax:= mean(logmx), by = (Age)]
5  train[,mx_adj:= logmx-ax]
6  rates_mat <- as.matrix(train %>% dcast.data.table(Age~Year, value.var = "mx_adj", sum))[,-1]
7
8  svd_fit <- svd(rates_mat)
9  ax <- train[,unique(ax)]
10 bx <- svd_fit$u[,1]*svd_fit$d[1]
11 kt <- svd_fit$v[,1]
12
13 c1 <- mean(kt)
14 c2 <- sum(bx)
15 ax <- ax+c1*bx
16 bx <- bx/c2
17 kt <- (kt-c1)*c2
```

The R code for the SVD fitting of the LC model is provided in Listing 2, lines 3-11, and lines 13-17 describe the rescaling for receiving the unique identifiability scheme. Note that this identifiability scheme is criticized in the literature because it may directly influence the method and quality of the extrapolation beyond the latest year observed. For the time-being we do not enter this discussion but we assume that the given normalization has the right property to receive reasonable extrapolation techniques and results.

To project the mortality rates beyond the latest calendar year observed, which is 1999 for our training data set $\mathcal{T}$, we need to extrapolate the time series $(\widehat{k}_t)_{t\in\mathcal{T}}$ to the calendar years $t \in \mathcal{V} = \{2000, \ldots, 2016\}$, where the latter is our out-of-sample validation set. We do this by a simple random walk with drift estimation using the R package `forecast`, and applying the function `rwf`, which estimates the corresponding drift term.

Listing 3: Extrapolating LC log-mortality rates with the R package `forecast`

```
1  ### extrapolation and forecast
2  vali  <- all_mort[Year>=2000][Gender == gender]
3  t_forecast <- vali[,unique(Year)] %>% length()
4  forecast_kt  =kt %>% forecast::rwf(t_forecast, drift = T)
5  kt_forecast = forecast_kt$mean
6
7  # in-sample log-mortality rates estimate from 1950 to 1999
8  (ax+(bx)%*%t(kt)) %>% melt
9
10 # out-of-sample log-mortality rates forecast from 2000 to 2016
11 (ax+(bx)%*%t(kt_forecast)) %>% melt
```

In Listing 3 we provide the code for this forecast of $(\widehat{k}_t)_{t\in\mathcal{V}}$, and in Figure 2 we illustrate the results. These forecasts look rather reasonable for females, whereas for males, the resulting drift may require further exploration. Based on these forecasts we can then provide estimates $(\log(\widehat{m}_{t,x}))_x$ for the log-mortality rates $(\log(m_{t,x}))_x$, both in-sample $t \in \mathcal{T}$ and out-of-sample $t \in \mathcal{V}$, see lines 7-8 and lines 10-11, respectively, of Listing 3.

Table 1 provides the resulting mean squared error (MSE) losses for both genders (in-sample and out-of-sample) between the LC model estimates/forecasts $(\widehat{m}_{t,x})_x$ and the raw mortality rate

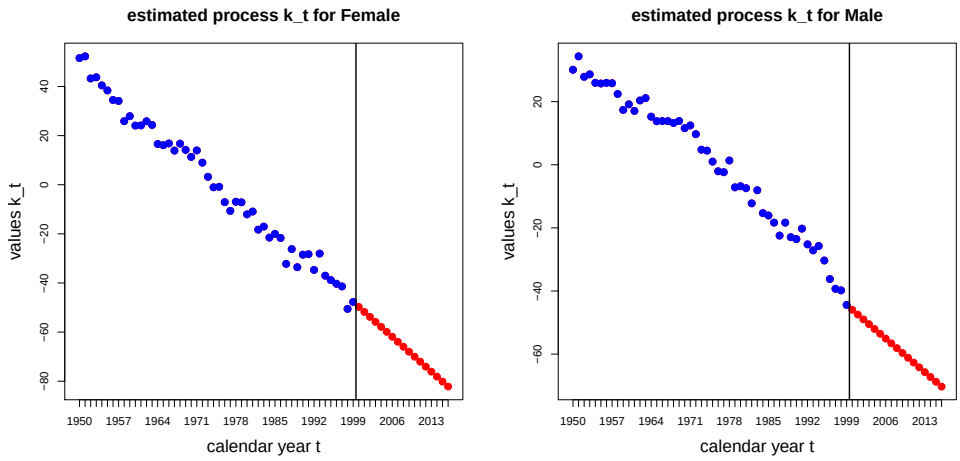

Figure 2: Estimated time series $(\widehat{k}_t)_{t\in\mathcal{T}}$ (in blue color for in-sample) and forecast time series $(\widehat{k}_t)_{t\in\mathcal{V}}$ (in red color for out-of-sample) of the Swiss female population (lhs) and of the Swiss male population (rhs).

	in-sample loss		out-of-sample loss	
	female	male	female	male
LC model with SVD	3.7573	8.8110	0.6045	1.8152

Table 1: In-sample and out-of-sample MSE losses; all figures are in 10^{-4}.

observations $(M_{t,x})_x$. We could now further back-test and analyze this solution, however, we refrain from doing so because the main purpose of this tutorial is to describe RNNs.

2 Recurrent neural networks

In this section we describe the architecture of RNNs. In a nutshell, the main difference between feed-forward neural networks and RNNs is that the former networks do not have cycles (loops) in signal transmissions from time-step to time-step, whereas the latter do have cycles. Apart from that, the main building blocks of these two network architectures are rather similar. For this reason, we will not describe each building block of a RNN in full detail in this manuscript, but refer to our previous tutorials [4, 11] which explore feed-forward neural networks and their building blocks in much more depth.

2.1 Short recap of feed-forward neural networks

In this section we briefly revisit feed-forward neural networks. This will help us to better understand RNNs. A feed-forward neural network architecture selects a depth $d \in \mathbb{N}$ for the number of hidden layers, integers $q_1, \ldots, q_d \in \mathbb{N}$ for the numbers of hidden neurons in these d hidden layers, and a (non-linear) activation function $\phi : \mathbb{R} \to \mathbb{R}$. Network layers $m \in \{1, \ldots, d\}$

are designed by considering the maps

$$\boldsymbol{z}^{(m)} : \mathbb{R}^{q_{m-1}} \to \mathbb{R}^{q_m}, \qquad \boldsymbol{z} \mapsto \boldsymbol{z}^{(m)}(\boldsymbol{z}) \;=\; \left(\phi\left\langle \boldsymbol{w}_1^{(m)}, \boldsymbol{z}\right\rangle, \ldots, \phi\left\langle \boldsymbol{w}_{q_m}^{(m)}, \boldsymbol{z}\right\rangle\right)^\top \stackrel{\text{def.}}{=} \phi\left\langle W^{(m)}, \boldsymbol{z}\right\rangle, \tag{2.1}$$

where $\langle \cdot, \cdot \rangle$ is the scalar product, where $W^{(m)} = (\boldsymbol{w}_j^{(m)})_{1 \le j \le q_m}^\top \in \mathbb{R}^{q_m \times (q_{m-1}+1)}$ are the network weights (parameters) of the considered hidden layer $m \in \{1, \ldots, d\}$,[4] and where the last line in (2.1) is an abbreviation which is understood element wise. Furthermore, we initialize q_0 to be the dimension of the available explanatory variables $\boldsymbol{x} \in \mathbb{R}^{q_0}$ (inputs).
The hidden part of the network architecture maps the input variables $\boldsymbol{x}$ to the last hidden layer d using the following composition of the hidden layer maps

$$\boldsymbol{z}^{(d:1)} : \mathbb{R}^{q_0} \to \mathbb{R}^{q_d}, \qquad \boldsymbol{x} \mapsto \boldsymbol{z}^{(d:1)}(\boldsymbol{x}) = \left(\boldsymbol{z}^{(d)} \circ \ldots \circ \boldsymbol{z}^{(1)}\right)(\boldsymbol{x}).$$

The last ingredient of a feed-forward neural network is the choice of an output activation $\varphi : \mathbb{R} \to \mathcal{Y}$, where $\mathcal{Y}$ is the domain of the output variable y that we aim to model (or predict in case y is a realization of a random variable Y). Henceforth, we define an output map on the last hidden layer d that provides us with the network function

$$\boldsymbol{x} \in \mathbb{R}^{q_0} \;\mapsto\; \varphi\left\langle \boldsymbol{w}_1^{(d+1)}, \boldsymbol{z}^{(d:1)}(\boldsymbol{x})\right\rangle \in \mathcal{Y}, \tag{2.2}$$

where $\boldsymbol{w}_1^{(d+1)} \in \mathbb{R}^{q_d+1}$ are the output weights (again including an intercept component).

Let us assume that the input variable $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ has a natural time series structure with components $\boldsymbol{x}_t \in \mathbb{R}^{\tau_0}$ at time points $t = 1, \ldots, T$. We could directly feed this input variable $\boldsymbol{x}$ into the feed-forward neural network (2.2), setting input dimension $q_0 = T\tau_0$, in order to model and predict a (random) variable y and Y, respectively. However, this is not the best way to tackle this problem for several reasons. Firstly, the natural time series structure gets lost (diluted) in approach (2.2) because this feed-forward network architecture is not designed to respect time and causal relationships, say, that if $\boldsymbol{x}_t$ has a direct impact on $\boldsymbol{x}_{t+1}$ this structure is not reflected in the network architecture (2.2). Secondly, one time step later we may have collected a next observation $\boldsymbol{x}_{T+1}\mathbb{R}^{\tau_0}$, but there is not any natural (and straightforward) way to expand (2.2) for additional input dimensions. For these reasons, we design a different network architecture that can deal with time series data.

2.2 Plain-vanilla recurrent neural network

We start by introducing the basic idea of RNNs on a simple network architecture, we call this the plain-vanilla RNN. This simple RNN architecture will be extended in subsequent sections to make it useful for time series applications, in particular, we are going to introduce the long short-term memory (LSTM) network and the gated recurrent unit (GRU) network below.
Assume we have input variables in time series structure $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ with components $\boldsymbol{x}_t \in \mathbb{R}^{\tau_0}$ at times $t = 1, \ldots, T$. We would like to use these input variables as explanatory variables

[4] For notational convenience, we do not explicitly indicate in the layer construction the inclusion of the intercept (alternatively, bias) parameters, but they are there (hidden in the right interpretation of the chosen notation), see also [4] and formula (2.4), below.

(features) to describe an output variable $y \in \mathcal{Y} \subset \mathbb{R}$. We start with a RNN that has a single hidden layer and $\tau_1 \in \mathbb{N}$ hidden neurons. This single hidden layer is defined by the mapping

$$\boldsymbol{z}^{(1)} : \mathbb{R}^{\tau_0 \times \tau_1} \to \mathbb{R}^{\tau_1}, \qquad (\boldsymbol{x}_t, \boldsymbol{z}_{t-1}) \;\mapsto\; \boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}\right), \tag{2.3}$$

where, basically, we assume the same structure as in (2.1). That is, we assume

$$\begin{aligned} \boldsymbol{z}_t^{(1)} \;=\; \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}\right) \;&=\; \left(\phi\left(\langle \boldsymbol{w}_1^{(1)}, \boldsymbol{x}_t\rangle + \langle \boldsymbol{u}_1^{(1)}, \boldsymbol{z}_{t-1}\rangle\right), \ldots, \phi\left(\langle \boldsymbol{w}_{\tau_1}^{(1)}, \boldsymbol{x}_t\rangle + \langle \boldsymbol{u}_{\tau_1}^{(1)}, \boldsymbol{z}_{t-1}\rangle\right)\right)^\top \\ &\overset{\text{def.}}{=}\; \phi\left(\left\langle W^{(1)}, \boldsymbol{x}_t\right\rangle + \left\langle U^{(1)}, \boldsymbol{z}_{t-1}\right\rangle\right), \end{aligned}$$

where the individual neurons $1 \leq j \leq \tau_1$ are modeled by

$$\phi\left(\langle \boldsymbol{w}_j^{(1)}, \boldsymbol{x}_t\rangle + \langle \boldsymbol{u}_j^{(1)}, \boldsymbol{z}_{t-1}\rangle\right) = \phi\left(w_{j,0}^{(1)} + \sum_{l=1}^{\tau_0} w_{j,l}^{(1)} x_{t,l} + \sum_{l=1}^{\tau_1} u_{j,l}^{(1)} z_{t-1,l}\right). \tag{2.4}$$

Thereby, $\phi : \mathbb{R} \to \mathbb{R}$ is a non-linear activation function, and the network parameters are denoted by $W^{(1)} = (\boldsymbol{w}_j^{(1)})_{1\leq j\leq \tau_1}^\top \in \mathbb{R}^{\tau \times (\tau_0+1)}$ (including an intercept) and $U^{(1)} = (\boldsymbol{u}_j^{(1)})_{1\leq j\leq \tau_1}^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$ (excluding an intercept).

Remarks.

- The lower index t in (2.3) refers to time, the upper index $^{(1)}$ to the fact that this is the first (and single) hidden layer in this example. That is, in this sense, the plain-vanilla RNN is a single hidden layer network.

- The lower time indexes t are added on purpose in (2.3): the hidden layer has τ_1 output neurons producing an output $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$, see (2.3). Together with next time series observation $\boldsymbol{x}_{t+1} \in \mathbb{R}^{\tau_0}$, we feed this output $\boldsymbol{z}_t^{(1)}$ into the next time step $t+1$; this is were we create the recurrent loop structure. This gives us the next output $\boldsymbol{z}_{t+1}^{(1)} = \boldsymbol{z}^{(1)}(\boldsymbol{x}_{t+1}, \boldsymbol{z}_t^{(1)}) \in \mathbb{R}^{\tau_1}$, and it implies that the neuron activations $\boldsymbol{z}_t^{(1)}$ naturally receive a time series structure:

  $$\cdots \quad\mapsto\quad \boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) \quad\mapsto\quad \boldsymbol{z}_{t+1}^{(1)} = \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_{t+1}, \boldsymbol{z}_t^{(1)}\right) \quad\mapsto\quad \cdots$$

  Observe that the two network parameters $W^{(1)} = (\boldsymbol{w}_1^{(1)} \ldots, \boldsymbol{w}_{\tau_1}^{(1)})^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ and $U^{(1)} = (\boldsymbol{u}_1^{(1)} \ldots, \boldsymbol{w}_{\tau_1}^{(1)})^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$ are *time independent*, that is, we visit the same hidden layer in every temporal step which results in total in T recurrent visits.

- Mapping (2.3) has an autoregressive structure of order 1 in $\boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right)$ because $\boldsymbol{z}_{t-1}^{(1)}$ describes the necessary past information for the next step; this structure also resembles a state-space model.

- There are different ways in designing RNNs with multiple hidden layers. For illustrative reasons, we focus on two hidden layers here, i.e. on depth $d = 2$.

  1st variant of a two-hidden layer RNN. We only allow for loops within the hidden layers (the upper indexes label the hidden layers):

  $$\boldsymbol{z}_t^{(1)} \;=\; \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right), \tag{2.5}$$

  $$\boldsymbol{z}_t^{(2)} \;=\; \boldsymbol{z}^{(2)}\left(\boldsymbol{z}_t^{(1)}, \boldsymbol{z}_{t-1}^{(2)}\right). \tag{2.6}$$

2nd variant of a two-hidden layer RNN. We allow for circles back to the first hidden layer:

$$\begin{aligned} \boldsymbol{z}_t^{(1)} &= \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{z}_{t-1}^{(2)}\right), \\ \boldsymbol{z}_t^{(2)} &= \boldsymbol{z}^{(2)}\left(\boldsymbol{z}_t^{(1)}, \boldsymbol{z}_{t-1}^{(2)}\right). \end{aligned}$$

3rd variant of a two-hidden layer RNN. We additionally add a skip connection from the input variable $\boldsymbol{x}_t$ to the second hidden layer:

$$\begin{aligned} \boldsymbol{z}_t^{(1)} &= \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{z}_{t-1}^{(2)}\right), \\ \boldsymbol{z}_t^{(2)} &= \boldsymbol{z}^{(2)}\left(\boldsymbol{x}_t, \boldsymbol{z}_t^{(1)}, \boldsymbol{z}_{t-1}^{(2)}\right). \end{aligned}$$

2.3 Long short-term memory architecture

The most popular RNN architecture is the long short-term memory (LSTM) architecture introduced by Hochreiter–Schmidhuber [7]. We recall this architecture in this section.

2.3.1 Construction of LSTM networks

The LSTM architecture uses three different non-linear activation functions:

$$\begin{aligned} \phi_\sigma(x) &= \frac{1}{1+e^{-x}} \in (0,1), && \text{sigmoid function,} \\ \phi_{\tanh}(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\phi_\sigma(2x) - 1 \in (-1,1), && \text{hyberbolic tangent function,} \end{aligned}$$

and $\phi : \mathbb{R} \to \mathbb{R}$ is a general activation function. These three activation function are used simultaneously for different purposes.
In a first step, we define three different gates that determine the amount of information that is passed to a next time step. Assume the neuron activations after the $(t-1)$-st time step are given by $\boldsymbol{z}_{t-1}^{(1)} \in \mathbb{R}^{\tau_1}$, and we aim at calculating the activations $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ at time t.

- *Forget gate* (loss of memory rate):

$$\boldsymbol{f}_t^{(1)} = \boldsymbol{f}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) = \phi_\sigma\left(\langle W_f, \boldsymbol{x}_t\rangle + \langle U_f, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1},$$

  for network parameters $W_f^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U_f^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$ (excluding an intercept), and where the activation function is evaluated element wise.

- *Input gate* (memory update rate):

$$\boldsymbol{i}_t^{(1)} = \boldsymbol{i}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) = \phi_\sigma\left(\langle W_i, \boldsymbol{x}_t\rangle + \langle U_i, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1},$$

  for network parameters $W_i^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U_i^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$.

- *Output gate* (release of memory information rate):

$$\boldsymbol{o}_t^{(1)} = \boldsymbol{o}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) = \phi_\sigma\left(\langle W_o, \boldsymbol{x}_t\rangle + \langle U_o, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1},$$

  for network parameters $W_o^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U_o^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$.

These gate variables are used to determine the neuron activations $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ at time t. This is done by an additional time series $(\boldsymbol{c}_t^{(1)})_t$, called *cell state process*, which stores the available (relevant) *memory*. In particular, given $\boldsymbol{c}_{t-1}^{(1)}$ and $\boldsymbol{z}_{t-1}^{(1)}$, we define the updated cell state

$$\boldsymbol{c}_t^{(1)} = \boldsymbol{c}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{c}_{t-1}^{(1)}\right) = \boldsymbol{f}_t^{(1)} \circ \boldsymbol{c}_{t-1}^{(1)} + \boldsymbol{i}_t^{(1)} \circ \phi_{\tanh}\left(\langle W_c, \boldsymbol{x}_t\rangle + \langle U_c, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \ \in \ \mathbb{R}^{\tau_1},$$

for network parameters $W_c^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U_c^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$, and where $\circ$ denotes the Hadamard product (element wise product). Finally, we define the updated neuron activations, given $\boldsymbol{c}_{t-1}^{(1)}$ and $\boldsymbol{z}_{t-1}^{(1)}$, by

$$\boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{c}_{t-1}^{(1)}\right) = \boldsymbol{o}_t^{(1)} \circ \phi\left(\boldsymbol{c}_t^{(1)}\right) \ \in \ \mathbb{R}^{\tau_1}. \tag{2.7}$$

Remarks.

- It seems a bit cumbersome that we use upper indexes $^{(1)}$ in all LSTM module definitions. We do this to illustrate that this architecture provides one hidden LSTM layer, and this approach can easily been extended to multiple hidden LSTM layers, for instance, using a structure similar to (2.5)-(2.6). We illustrate this in the examples below.
- The network parameters involved are given by the matrices $W_f^\top, W_i^\top, W_o^\top, W_c^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ and $U_f^\top, U_i^\top, U_o^\top, U_c^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$. These are learned using a variant of the gradient descent algorithm. An LSTM layer involves $4((\tau_0+1)\tau_1 + \tau_1^2)$ network parameters.

We did not specify the output yet; this is going to be described in the next section.

2.3.2 Outputs and time-distributed layers in RNNs

In the previous section we have defined the LSTM architecture up to the output function. Assume we would like to predict a random variable Y_T with domain $\mathcal{Y} \subset \mathbb{R}$, based on the available time series information $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$. In this case, we take the latest state of the hidden neuron activation, given by $\boldsymbol{z}_T^{(1)}$, see (2.7). This latest state contains all feature information $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ which has been (appropriately) pre-processed by the LSTM network, i.e., by a slight abuse of notation, we may write $\boldsymbol{z}_T^{(1)} = \boldsymbol{z}_T^{(1)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$. Based on this latest state we define the output function, we also refer to (2.2),

$$\widehat{Y}_T = \widehat{Y}_T(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) = \varphi\left\langle \boldsymbol{w}, \boldsymbol{z}_T^{(1)} \right\rangle \in \mathcal{Y}, \tag{2.8}$$

where $\boldsymbol{w} \in \mathbb{R}^{\tau_1+1}$ are the output weights (again including an intercept component), and where $\varphi : \mathbb{R} \to \mathcal{Y}$ is an appropriate output activation.

In RNN applications there is also the notion of a so-called time-distributed layer. Observe that (2.8) focuses on the last output Y_T, described by the latest state $\boldsymbol{z}_T^{(1)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$. Instead we could also focus on all hidden neuron states simultaneously. These are given by the time series

$$\boldsymbol{z}_1^{(1)}(\boldsymbol{x}_1), \ \boldsymbol{z}_2^{(1)}(\boldsymbol{x}_1, \boldsymbol{x}_2), \ \boldsymbol{z}_3^{(1)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_3), \ldots, \boldsymbol{z}_T^{(1)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T),$$

where we explicitly state all dependencies on the $\boldsymbol{x}_t$'s. Each of these states $\boldsymbol{z}_t^{(1)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t)$ may serve as explanatory variable to estimate/predict the corresponding random variable Y_t at time

point t. This task can be achieved by "distributing output function (2.8) over time $t = 1, \ldots, T$" by always using the *same filter* $\varphi\langle \boldsymbol{w}, \cdot \rangle$, i.e.

$$\widehat{Y}_t = \widehat{Y}_t(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) = \varphi\left\langle \boldsymbol{w}, \boldsymbol{z}_t^{(1)} \right\rangle = \varphi\left\langle \boldsymbol{w}, \boldsymbol{z}_t^{(1)}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) \right\rangle. \tag{2.9}$$

This is pretty smart because it exactly improves the weaknesses of the feed-forward neural network mentioned on page 6. First of all, time series structure and causality is reflected correctly and, secondly, we can easily expand this to future periods because the set of parameters is not time dependent. The only item which may distort this time consistency is that the network parameters have to be estimated and these estimates typically depend on (the latest) data that increases over time.

2.4 Gated recurrent unit architecture

Another popular RNN architecture is the gated recurrent unit (GRU) architecture introduced by Cho et al. [2]. A shortcoming of the LSTM architecture is that it is fairly complex. GRU networks have simpler architectures, and may provide competitive results. We briefly introduce them here. The consideration of outputs is identical to LSTM networks, see Section 2.3.2. Therefore, this part is not further described for GRU networks.
A GRU architecture only uses two different gates. Assume that the neuron activations after the $(t-1)$-st time step are given by $\boldsymbol{z}_{t-1}^{(1)} \in \mathbb{R}^{\tau_1}$, and that we aim at calculating the activations $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ at time t. For GRU architectures we define two gates.

- *Reset gate*:

$$\boldsymbol{r}_t^{(1)} = \boldsymbol{r}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) = \phi_\sigma\left(\langle W_r, \boldsymbol{x}_t\rangle + \langle U_r, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1},$$

  for network parameters $W_r^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U_r^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$.

- *Update gate*:

$$\boldsymbol{u}_t^{(1)} = \boldsymbol{u}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) = \phi_\sigma\left(\langle W_u, \boldsymbol{x}_t\rangle + \langle U_u, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1},$$

  for network parameters $W_u^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U_u^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$.

These gate variables are used to determine the neuron activations $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ at time t, given $\boldsymbol{z}_{t-1}^{(1)}$. We therefore choose the structure

$$\boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}\right) = \boldsymbol{r}_t^{(1)} \circ \boldsymbol{z}_{t-1}^{(1)} + (\mathbf{1} - \boldsymbol{r}_t^{(1)}) \circ \phi\left(\langle W, \boldsymbol{x}_t\rangle + \boldsymbol{u}_t \circ \langle U, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in \mathbb{R}^{\tau_1},$$

for network parameters $W^\top \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ (including an intercept), $U^\top \in \mathbb{R}^{\tau_1 \times \tau_1}$, and where $\circ$ denotes the Hadamard product.

The general philosophy of this GRU architecture is the same as the one in (2.7), therefore, also similar remarks apply (which are not repeated here).

3 Implementation of recurrent neural networks

3.1 Using the R library keras

Before we are going to study our mortality rate application in detail, we illustrate the implementation of RNNs in the R library `keras`. We focus on the implementation of LSTM architectures since the implementation of other RNNs is essentially the same.

Listing 4: Single hidden layer LSTM model in the R library `keras`

```
1  library(keras)
2
3  T    <- 10    # length of time series x_1,...,x_T
4  tau0 <- 3     # dimension inputs x_t
5  tau1 <- 5     # dimension of the neurons z_t^(1) in the first RNN layer
6
7  Input <- layer_input(shape=c(T,tau0), dtype='float32', name='Input')
8
9  Output = Input %>%
10   layer_lstm(units=tau1, activation='tanh', recurrent_activation='tanh', name='LSTM1') %>%
11   layer_dense(units=1, activation=k_exp, name="Output")
12
13 model <- keras_model(inputs = list(Input), outputs = c(Output))
14 model %>% compile(loss = 'mean_squared_error', optimizer = 'nadam')
15 summary(model)
```

Listing 5: Architecture of a single hidden layer LSTM model

```
1  Layer (type)                 Output Shape              Param #
2  =================================================================
3  Input (InputLayer)           (None, 10, 3)             0
4  _________________________________________________________________
5  LSTM1 (LSTM)                 (None, 5)                 180
6  _________________________________________________________________
7  Output (Dense)               (None, 1)                 6
8  =================================================================
9  Total params: 186
10 Trainable params: 186
11 Non-trainable params: 0
```

Listing 4 provides the implementation of a LSTM architecture with one hidden layer. The length of the time series $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ is set equal to $T = 10$ (look-back period), the dimension of the input variables $\boldsymbol{x}_t \in \mathbb{R}^{\tau_0}$ is $\tau_0 = 3$, and the dimension of the hidden neurons $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ is $\tau_1 = 5$, see lines 3-5 of Listing 4. On line 10 we define the LSTM layer. Our parameter choices provide four matrices $W_\bullet \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ and four matrices $U_\bullet \in \mathbb{R}^{\tau_1 \times \tau_1}$ with $\bullet \in \{f, i, o, c\}$. Altogether this gives us $4((\tau_0 + 1)\tau_1 + \tau_1^2) = 4(20 + 25) = 180$ network parameters from the LSTM layer, see Listing 5. Since we assume that the output variable $y_T \in \mathcal{Y} = \mathbb{R}_+$ is one dimensional and strictly positive we add a feed-forward output layer to $\boldsymbol{z}_T^{(1)} \in \mathbb{R}^{\tau_1}$, having exponential output activation $\varphi(x) = \exp\{x\}$, see line 11 of Listing 4. This last step is similar to the feed-forward architecture (2.2), and gives us another $\tau_1 + 1 = 6$ parameters, see line 7 of Listing 5. Thus, in total we receive 186 trainable network parameters.

An extension of a single hidden layer LSTM to a multiple hidden layer LSTM architecture is rather straightforward. The parts that change compared to the single hidden layer LSTM

Listing 6: Two hidden layers LSTM model in the R library `keras`

```
1  Output = Input %>%
2    layer_lstm(units=tau1, activation='tanh', recurrent_activation='tanh',
3                                                  return_sequences=TRUE, name='LSTM1') %>%
4    layer_lstm(units=tau2, activation='tanh', recurrent_activation='tanh', name='LSTM2') %>%
5    layer_dense(units=1, activation=k_exp, name="Output")
```

Listing 7: Architecture of a two hidden layers LSTM model

```
1  Layer (type)                 Output Shape              Param #
2  =================================================================
3  Input (InputLayer)           (None, 10, 3)             0
4  _________________________________________________________________
5  LSTM1 (LSTM)                 (None, 10, 5)             180
6  _________________________________________________________________
7  LSTM2 (LSTM)                 (None, 4)                 160
8  _________________________________________________________________
9  Output (Dense)               (None, 1)                 5
10 =================================================================
11 Total params: 345
12 Trainable params: 345
13 Non-trainable params: 0
```

architecture are highlighted in Listing 6. First, we need to adjust the first LSTM layer so that the neuron activations $\boldsymbol{z}_t^{(1)}$ of all time points $t = 1, \ldots, T$ are output (and not only for the last time point T). This is achieved by the command `return_sequences=TRUE` on line 3 of Listing 6. The second hidden LSTM layer is then defined on line 4. If we choose $\tau_2 = 4$ hidden neurons in that second LSTM layer, this adds $4((\tau_1+1)\tau_2+\tau_2^2) = 4(24+16) = 160$ network parameters, see line 7 of Listing 7. On the other hand, the dense output layer is reduced to $\tau_2 + 1 = 5$ neurons, this results in 345 trainable parameters, see Listing 7.

Listing 8: Single hidden layers LSTM model with time-distribution

```
1  Output = Input %>%
2    layer_lstm(units=tau1, activation='tanh', recurrent_activation='tanh',
3                                                  return_sequences=TRUE, name='LSTM1') %>%
4    time_distributed(layer_dense(units=1, activation=k_exp, name="Output"), name='TD')
```

Listing 9: Architecture of a single hidden layer LSTM model with time-distribution

```
1  Layer (type)                 Output Shape              Param #
2  =================================================================
3  Input (InputLayer)           (None, 10, 3)             0
4  _________________________________________________________________
5  LSTM1 (LSTM)                 (None, 10, 5)             180
6  _________________________________________________________________
7  TD (TimeDistributed)         (None, 10, 1)             6
8  =================================================================
9  Total params: 186
10 Trainable params: 186
11 Non-trainable params: 0
```

Finally, we describe how we can predict the outputs of all time points $t = 1, \ldots, T$ simultaneously,

using the same time-distributed dense feed-forward layer on all neurons $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_0}$, see also (2.9). The corresponding code is provided in Listing 8. The changes compared to Listing 4 are that we need to have the whole return sequence $\boldsymbol{z}_1^{(1)}, \ldots, \boldsymbol{z}_T^{(1)}$, see line 3 of Listing 8, to which we apply a time-distributed dense output layer on line 4 of Listing 8. The number of parameters remains the same, but the dimension of the output changes from a scalar to a vector of dimension $T = 10$, compare Listings 5 and 9.

3.2 Toy example for the use of LSTM and GRU models

In this section we analyze a toy example to illustrate the functioning of LSTM and GRU architectures. The data used has been compiled from the raw Swiss log-mortality rates $\log(M_{t,x})$ as described in Listing 14 in the appendix. We choose gender "Female" and we extract $\log(M_{t,x})$ for calendar years $1990, \ldots, 2001$ and ages $0 \leq x \leq 99$. We define the explanatory and the response variables as follows. Set $T = 10$ and $\tau_0 = 3$, and define for ages $1 \leq x \leq 98$ and years $1 \leq t \leq T$

$$\begin{aligned} \boldsymbol{x}_{t,x} &= \Big(\log(M_{1999-(T-t),x-1}), \log(M_{1999-(T-t),x}), \log(M_{1999-(T-t),x+1})\Big)^\top \in \mathbb{R}^{\tau_0}, \quad (3.1) \\ Y_{T,x} &= \log(M_{2000,x}) = \log(M_{1999-(T-T)+1,x}) \in \mathbb{R}_-. \quad (3.2) \end{aligned}$$

Based on these definitions we choose the training data

$$\mathcal{T} = \{(\boldsymbol{x}_{1,x}, \ldots, \boldsymbol{x}_{T,x}; Y_{T,x});\ 1 \leq x \leq 98\}.$$

Thus, we receive 98 training samples where the feature information $(\boldsymbol{x}_{1,x}, \ldots, \boldsymbol{x}_{T,x})$ is a time series in $\mathbb{R}^{T \times \tau_0}$. Considering the ages $(x-1, x, x+1)$ simultaneously in $\boldsymbol{x}_{t,x}$ is "smoothing" inputs over neighboring ages, and the response variable is the log-mortality rate in the next year (corresponding to $T+1$) for the "central" age x.[5] This training data $\mathcal{T}$ is illustrated in Figure

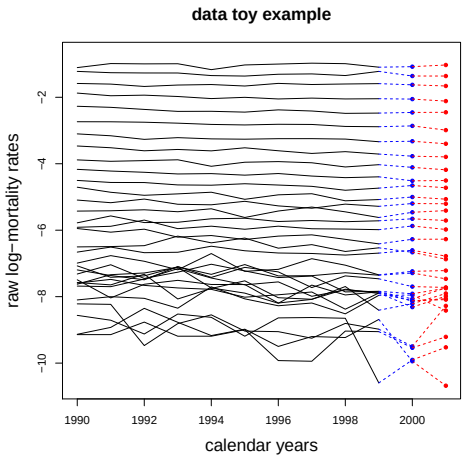

Figure 3: Selected data for the toy example.

3: the black lines show (selected) explanatory variables $\boldsymbol{x}_{t,x}$, and the blue dots are the (same

[5]Note that we do not follow a birth cohort, but we keep the age label fixed.

selected) response variables $Y_{T,x}$. The training data is used to fit the chosen LSTM model. This fitted model will then be back-tested on validation data $\mathcal{V}$ which considers a time shift of one calendar, i.e.

$$\mathcal{V} = \{(\boldsymbol{x}_{2,x}, \ldots, \boldsymbol{x}_{T+1,x}; Y_{T+1,x});\ 1 \leq x \leq 98\}\,. \tag{3.3}$$

The response variables $Y_{T+1,x} = \log(M_{2001,x})$ of this time shifted validation data are illustrated in red color in Figure 3, and the explanatory variables are shifted correspondingly. Note that alternatively we could also choose the expanded time series as validation set

$$\mathcal{V}_+ = \{(\boldsymbol{x}_{1,x}, \ldots, \boldsymbol{x}_{T+1,x}; Y_{T+1,x});\ 1 \leq x \leq 98\}\,, \tag{3.4}$$

because the LSTM model allows us to expand the length of the input variables.

3.2.1 Comparison of LSTMs and GRUs

We use this data $\mathcal{T}$ and $\mathcal{V}$ to fit the first LSTM architecture defined in Listing 4, named hereafter LSTM1. First, we pre-process the explanatory variables such that they are supported in $[-1, 1]$

Listing 10: Data pre-processing and running stochastic gradient descent of LSTM1

```
1  # data pre-processing
2  x.train <- 2*(xt[1,,,]-min(xt))/(max(xt)-min(xt))-1
3  x.vali  <- 2*(xt[2,,,]-min(xt))/(max(xt)-min(xt))-1
4  y.train <- - YT[1,]
5  y.vali  <- - YT[2,]
6
7  fit <- model %>% fit(x=x.train, y=y.train, validation_data=list(x.vali, y.vali),
8                       batch_size=10, epochs=500, verbose=0)
```

using the MinMaxScaler, see lines 2-3 in Listing 10, and we switch the sign of the response variables on lines 4-5 such that they are strictly positive (and comply with the exponential output function chosen in this model LSTM1, see line 11 of Listing 4). On lines 7-8 of Listing 10 we run the stochastic gradient descent algorithm on mini batches of size 10 over 500 epochs over the training set $\mathcal{T}$. We track over-fitting on the validation set $\mathcal{V}$. Of course, in real applications, this should not be done because the validation set should be used for model back-testing across models. However, in the present situation it helps us to better understand our results and the functioning of LSTM models.

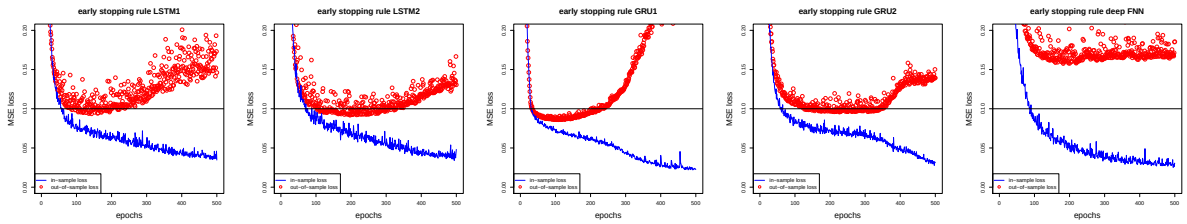

Figure 4: In-sample losses (blue color) and out-of-sample losses (red color) from left to right: LSTM1, LSTM2, GRU1, GRU2 and deep FNN (the y-scale is identical in all plots).

Figure 4 (lhs) shows the in-sample losses (blue color) and the out-of-sample losses (red color) of the architecture LSTM1 for one particular starting value of the gradient descent algorithm (we

choose the `nadam` version here, see line 14 of Listing 4). We note that after roughly 150 epochs this LSTM architecture starts to over-fit to the training data, therefore, we should exercise early stopping around 150 epochs (but of course this may vary over different seeds of the algorithm). The resulting model calibration for this stopping time is presented in Table 2.

	# param.	epochs	run time	in-sample loss	out-of-sample loss
LSTM1	186	150	8 sec	0.0655	0.0936
LSTM2	345	200	15 sec	0.0603	0.0918
GRU1	141	100	5 sec	0.0671	0.0860
GRU2	260	200	14 sec	0.0651	0.0958
deep FNN	184	200	5 sec	0.0485	0.1577

Table 2: Toy models: in-sample and out-of-sample losses.

As a second example we use the deep LSTM architecture of Listing 6 with $\tau_2 = 4$ hidden neurons in the second hidden LSTM layer. We call this second model LSTM2, and we perform exactly the same analysis as for LSTM1. The over-fitting graph is shown in Figure 4 (2nd from left) and the resulting losses are given in Table 2. From these numbers we conclude that both models are of comparable predictive quality, the latter using more parameters and twice as much run time.[6]

Next we perform the same analysis for GRU architectures: we replace in Listings 4 and 6 all LSTM layers by GRU layers, but keep all remaining parameters fixed. We call these two models GRU1 and GRU2. Figure 4 (middle and 2nd from right) shows the over-fitting analysis, and the selected early stopped models are given in Table 2. The conclusion of this toy modeling exercise is that in our situation the LSTM and the GRU architectures provide comparable results. A general observation is that GRU architectures seem to over-fit faster than LSTM architectures (choosing all hyperparameters identical), and their solutions are more volatile (in the starting point of the gradient descent algorithm). Thus, LSTM architectures seem to be better behaved in average and more robust, at the price of longer run times. We will further analyze this in Section 4, below.

Finally, we consider a classical feed-forward neural network architecture with $d = 2$ hidden layers having $(q_1, q_2) = (5, 4)$ neurons. For this network we need to reshape the explanatory variables from a $T \times \tau_0$ matrix to a $q_0 = T\tau_0$ dimensional vector. We call this model deep FNN, and the corresponding results are also presented in Figure 4 (rhs) and Table 2. We observe that this feed-forward neural network is clearly non-competitive with the RNNs (at roughly the same number of network parameters).

In Figure 5 we compare the individual out-of-sample predictions $\widehat{Y}_{T+1,x} = \log(\widehat{M}_{2001,x})$ to their observations $Y_{T+1,x} = \log(M_{2001,x})$ in calendar year 2001. This out-of-sample analysis is based on the validation data $\mathcal{V}$. We observe that, indeed, all considered RNN architectures manage to predict the outcomes very well, and the deep FNN seems to be too wiggly because it cannot smooth the input correctly over time.

[6]Run times are measured on a personal laptop with CPU @ 2.50GHz (4 CPUs) with 16GB RAM (which has a comparably modest computational power).

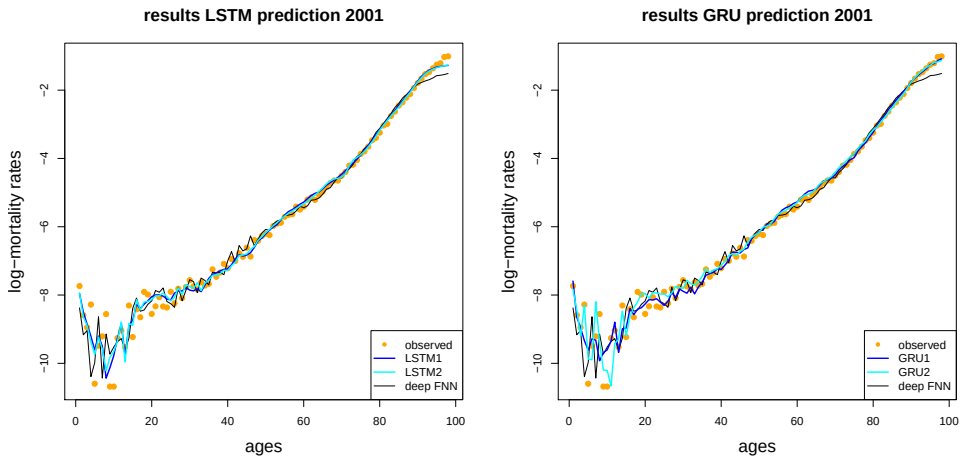

Figure 5: Resulting out-of-sample predictions $\widehat{Y}_{T+1,x} = \log(\widehat{M}_{2001,x})$ for calendar year 2001 and ages $1 \leq x \leq 98$: (lhs) LSTM architectures and (rhs) GRU architectures.

3.2.2 Choices of hyperparameters

In this section we explore different choices of the hyperparameters in model LSTM1: (a) $\tau_0 = 1, 3, 5$, i.e. smoothing across more or less ages $(x - \frac{\tau_0-1}{2}, \ldots, x, \ldots, x + \frac{\tau_0-1}{2})$ in $\boldsymbol{x}_{t,x} \in \mathbb{R}^{\tau_0}$ around the central age x; (b) $T = 5, 10, 20$, i.e. choosing different look-back periods, we fix calendar years 2000 and 2001, and T is then the number of years we look back in time; and (c) $\tau_1 = 3, 5, 10$, i.e. the number of hidden neurons in the hidden LSTM layer is changed.

	# param.	epochs	run time	in-sample	out-of-sample
base case:					
LSTM1 ($T = 10$, $\tau_0 = 3$, $\tau_1 = 5$)	186	150	8 sec	0.0655	0.0936
LSTM1 ($T = 10$, $\tau_0 = 1$, $\tau_1 = 5$)	146	100	5 sec	0.0647	0.1195
LSTM1 ($T = 10$, $\tau_0 = 5$, $\tau_1 = 5$)	226	150	15 sec	0.0583	0.0798
LSTM1 ($T = 5$, $\tau_0 = 3$, $\tau_1 = 5$)	186	100	4 sec	0.0753	0.1028
LSTM1 ($T = 20$, $\tau_0 = 3$, $\tau_1 = 5$)	186	200	16 sec	0.0626	0.0968
LSTM1 ($T = 10$, $\tau_0 = 3$, $\tau_1 = 3$)	88	200	10 sec	0.0694	0.0987
LSTM1 ($T = 10$, $\tau_0 = 3$, $\tau_1 = 10$)	571	100	5 sec	0.0626	0.0883

Table 3: Toy models: LSTM1 with different hyperparameter choices.

The results are presented in Figure 6 and Table 3. By far the biggest influence is observed by a bigger choice of τ_0, i.e. a larger dimension in $\boldsymbol{x}_{t,x} \in \mathbb{R}^{\tau_0}$. Intuitively, this is clear because there is some randomness involved in the raw log-mortality rates $\log(M_{t,x})$. These fluctuations are smoothed out if we consider more neighboring observations in the covariates/features $\boldsymbol{x}_{t,x} = (\log(M_{x-(\tau_0-1)/2}, \ldots, \log(M_{x+(\tau_0-1)/2})^\top \in \mathbb{R}^{\tau_0}$ around the central observation $\log(M_{t,x})$ to predict $Y_{T,x}$.

As a short summary of this analysis we mention: (a) increasing $\tau_0 = 1, 3, 5$ needs more run time (epochs) until over-fitting, and provides better out-of-sample results; (b) increasing $T = 5, 10, 20$ needs more run time (epochs) until over-fitting, and provides better out-of-sample results for

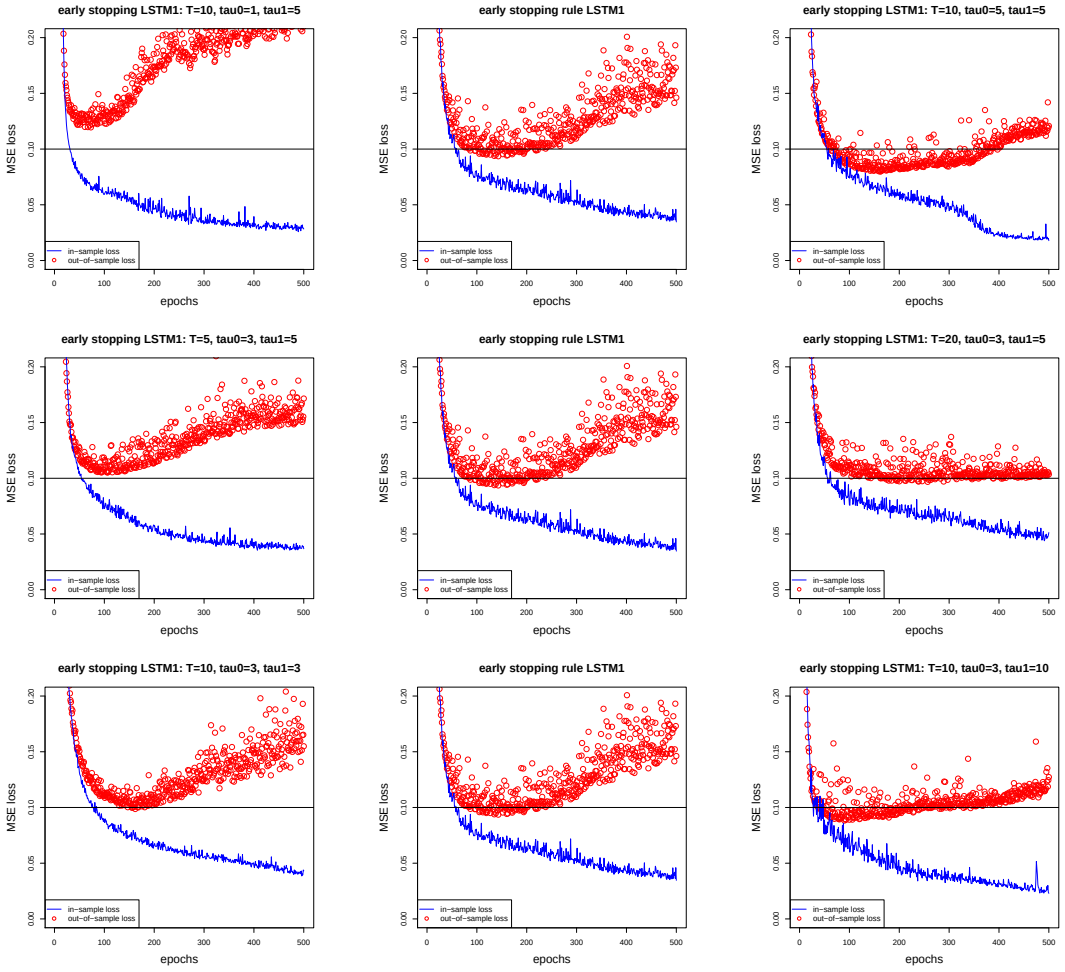

Figure 6: In-sample losses (blue color) and out-of-sample losses (red color) of (1st row) $\tau_0 = 1, 3, 5$, (2nd row) $T = 5, 10, 20$, and (3rd) $\tau_1 = 3, 5, 10$; the middle column gives the base case LSTM1.

networks having a minimal required complexity; and (c) increasing $\tau_1 = 3, 5, 10$, leads to faster convergence (because the gradient descent algorithm has more degrees of freedom).

3.2.3 Time-distributed LSTM layers

In this section we analyze the time-distributed version of model LSTM1, the corresponding code is given in Listing 8. The time-distributed version needs for every $t = 1, \ldots, T$ an output variable, i.e. we need to choose an output vector instead of the scalar $Y_{T,x}$ given in (3.2). Therefore, we define for $T = 10$ the response vector

$$\boldsymbol{Y}_{T,x} = \big(\log(M_{1999-(T-1)+1,x}), \ldots, \log(M_{2000,x})\big)^\top \in \mathbb{R}^T.$$

Listing 15 in the appendix provides the corresponding code. The time-distributed LSTM version then provides a predicted vector $\widehat{\boldsymbol{Y}}_{T,x} = (\widehat{Y}_{1,x}, \ldots, \widehat{Y}_{T,x})^\top$ which considers each time point $1 \leq t \leq T$, simultaneously. For this reason, the loss figures during the gradient descent algorithm

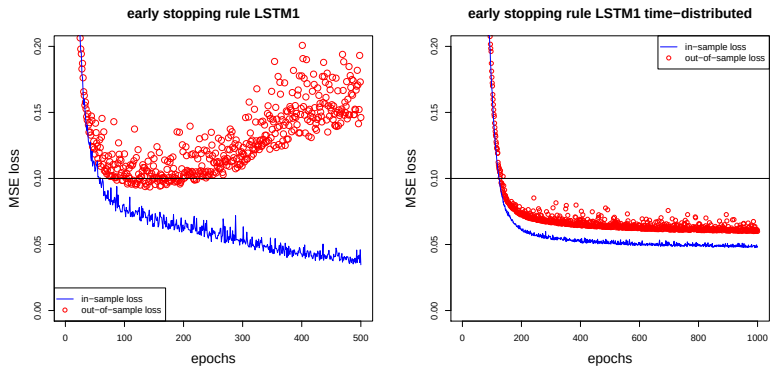

Figure 7: In-sample losses (blue color) and out-of-sample losses (red color) of (lhs) LSTM1 and (rhs) time-distributed LSTM1.

are not comparable between the LSTM1 and the time-distributed LSTM1, because the former optimizes the output weights solely w.r.t. $\widehat{Y}_{T,x} \in \mathbb{R}$ and the latter w.r.t. the entire vector $\widehat{\boldsymbol{Y}}_{T,x} \in \mathbb{R}^{\tau_0}$. This difference in model fitting is also supported by Figure 7. Intuitively, the latter uses

	# param.	epochs	run time	in-sample	out-of-sample
LSTM1 base case	186	150	8 sec	0.0655	0.0936
LSTM1 time-distributed	186	1000	54 sec	0.0621	0.1104

Table 4: Toy model: LSTM1 time-distributed layers.

more data to estimate optimally the (time-distributed) output weights, however, the weights of the former model fitting are directly tailored towards the prediction of the final output. Indeed, if we only consider the out-of-sample prediction $\widehat{Y}_{T+1,x}$ of the final output variable $Y_{T+1,x}$ in both fitted models, the base case (without time-distribution) has a better predictive performance than the time-distributed one (if we only evaluate the quality of the prediction of the last calendar year component in the time-distributed version), see Table 4. Nevertheless, the time-distributed layers are an important concept in other applied problems, in particular, in designing more complex RNN architectures for more complicated data examples.

4 Mortality rate predictions with LSTM and GRU models

In this section we challenge the LC results of Section 1.2 with RNN predictions. First, we explain data pre-processing which is different from above. Then,we fit RNNs separately to both genders "Female" and "Male". Finally, we discuss joint mortality rate prediction, using the data of both genders simultaneously to calibrate the predictive model, and we analyze the resulting predictions.

4.1 Data pre-processing for RNNs

In this section we describe data pre-processing. We consider the same data as in the LC study of Section 1.2, i.e. we use the data of the calendar years from 1950 to 1999 as observations

for model learning, and we aim at predicting the mortality rates in the future years from 2000 to 2016. This is the data partition considered in Table 1, where, additionally, we study both genders separately.

We choose a fixed gender and we set $\tau_0 = 5$, thus, we smooth feature values over 5 neighboring ages to predict the mortality rate of the central age x in $\boldsymbol{x}_{t,x} \in \mathbb{R}^{\tau_0}$. Since we would like to do this for all ages $0 \leq x \leq 99$, we need padding at the age boundaries. This is done by duplicating the marginal feature values. We therefore replace feature definition (3.1) for ages $0 \leq x \leq 99$ and calendar years $1950 \leq t \leq 1999$ as follows (we set $\tau_0 = 5$ here)

$$\boldsymbol{x}_{t,x} = \Big(\log(M_{t,(x-2)\vee 0}), \log(M_{t,(x-1)\vee 0}), \log(M_{t,x}), \log(M_{t,(x+1)\wedge 99}), \log(M_{t,(x+2)\wedge 99}) \Big)^\top \in \mathbb{R}^5.$$

where $x_0 \vee x_1 = \max\{x_0, x_1\}$ and $x_0 \wedge x_1 = \min\{x_0, x_1\}$. Based on these definitions we define the training data $\mathcal{T}$ for a given look-back period of $T = 10$ years by

$$\mathcal{T} = \{(\boldsymbol{x}_{t-T,x}, \ldots, \boldsymbol{x}_{t-1,x}, Y_{t,x});\ 0 \leq x \leq 99,\ 1950 + T \leq t \leq 1999\}, \tag{4.1}$$

with response variables $Y_{t,x} = \log(M_{t,x})$. This gives us $|\mathcal{T}| = 100 \cdot 40 = 4000$ training samples. The construction of this data set is illustrated in Listing 16.

The choice of the validation data $\mathcal{V}$ needs more care! We need knowledge of all feature values $\boldsymbol{x}_{t-T,x}, \ldots, \boldsymbol{x}_{t-1,x}$ to predict a future year $Y_{t,x}$. If, say, $t = 2001$, we have observed $\boldsymbol{x}_{t-T,x}, \ldots, \boldsymbol{x}_{1999,x}$, but, unfortunately, we do not know $\boldsymbol{x}_{2000,x}$. Therefore, we replace all future feature values for times $s > 1999$ by corresponding predictions

$$\widehat{\boldsymbol{x}}_{s,x} = \Big(\log(\widehat{M}_{s,(x-2)\vee 0}), \log(\widehat{M}_{s,(x-1)\vee 0}), \log(\widehat{M}_{s,x}), \log(\widehat{M}_{s,(x+1)\wedge 99}), \log(\widehat{M}_{s,(x+2)\wedge 99}) \Big)^\top \in \mathbb{R}^5,$$

where these predictions $\widehat{M}_{s,x}$ for $M_{s,x}$ are calculated recursively in $s = 2000, \ldots, 2015$. For the validation data we then shift the calendar year index and consider

$$\mathcal{V} = \{(\boldsymbol{x}_{t-T,x}, \ldots, \boldsymbol{x}_{1999,x}, \widehat{\boldsymbol{x}}_{2000,x}, \ldots, \widehat{\boldsymbol{x}}_{t-1,x}, Y_{t,x});\ 0 \leq x \leq 99,\ 2000 \leq t \leq 2016\}. \tag{4.2}$$

This gives us $|\mathcal{V}| = 100 \cdot 17 = 1700$ validation samples. Furthermore, we apply the MinMaxScaler to the features in $\mathcal{V}$ based on the minimum and maximum of all feature values in $\mathcal{T}$ (which have been used for model fitting) and we switch the sign of the responses.

The training data $\mathcal{T}$ in (4.1) and the validation data $\mathcal{V}$ in (4.2) are constructed separately for both genders, and the MinMaxScaler is applied to both genders separately, too. If we construct a joint model for both genders, then we establish the data in $\mathcal{T}$ and $\mathcal{V}$ with additional binary indicators for the genders of the corresponding observations. Moreover, in that case the MinMaxScaler is applied jointly to both genders, i.e. considering the minimum and maximum over all training data of both genders.

Remarks.

- It is important to use the same minimum and maximum in the MinMaxScaler for training data $\mathcal{T}$ and validation data $\mathcal{V}$. Otherwise, the two data sets live on different scales.
- The validation data $\mathcal{V}$ is received recursively. For $t = 2000$, all necessary feature values $\boldsymbol{x}_{t-T,x}, \ldots, \boldsymbol{x}_{1999,x}$ are observed, and $Y_{2000,x} = \log(M_{2000,x})$ can be predicted. These predictions $\log(\widehat{M}_{2000,x})$ then enter the feature values $\widehat{\boldsymbol{x}}_{2000,x}$, which allows us to predict the next responses $Y_{2001,x} = \log(M_{2001,x})$, and so forth.

- Note that in this recursive construction there is a slight inconsistency in the use of the variables. The feature values $\boldsymbol{x}_{s,x}$ are based on *observations* $M_{s,x}$ and the estimated feature values $\widehat{\boldsymbol{x}}_{s,x}$ are based on *predictions* $\widehat{M}_{s,x}$. These predictions are rather of a similar quality as (conditionally) expectations, and therefore, typically, have a lower volatility than the observations. Strictly speaking, we cannot make rigorous statements here, because we have not introduced a statistical model, i.e. we only argue on the level of mortality rates without an underlying probabilistic model. However, choosing the squared loss function underpins a Gaussian uncertainty structure.

- Following up the previous bullet point: if we (implicitly) assume a Gaussian structure for log-mortality rates $\log(M_{t,x})$ and we aim at predicting the mortality rates $M_{t,x}$, we may also need to add a bias correction when moving from Gaussian to log-normal random variables, otherwise we may underestimate the true mortality rates.

4.2 Recurrent neural networks on individual genders

In this section we fit a LSTM model and a GRU model with three hidden layers to each gender separately. We call these models LSTM3 and GRU3, the latter is illustrated in Listing 11, and the former is essentially the same replacing all GRU modules by LSTM modules.

Listing 11: Three hidden layers GRU model in the R library `keras`

```
1  GRU3 <- function(T0, tau0, tau1, tau2, tau3, y0=0, optimizer){
2      Input <- layer_input(shape=c(T0,tau0), dtype='float32', name='Input')
3      Output = Input %>%
4         layer_gru(units=tau1, activation='tanh', recurrent_activation='tanh',
5                   return_sequences=TRUE, name='GRU1') %>%
6         layer_gru(units=tau2, activation='tanh', recurrent_activation='tanh',
7                   return_sequences=TRUE, name='GRU2') %>%
8         layer_gru(units=tau3, activation='tanh', recurrent_activation='tanh', name='GRU3') %>%
9         layer_dense(units=1, activation=k_exp, name="Output",
10           weights=list(array(0,dim=c(tau3,1)), array(log(y0),dim=c(1))))
11     model <- keras_model(inputs = list(Input), outputs = c(Output))
12     model %>% compile(loss = 'mean_squared_error', optimizer = optimizer)
13      }
```

Note that on line 10, we initialize the weights of the output layer to zero and the intercept parameter to the mean of the log mortality rates in the training set. We find that this speeds the convergence of the network and may result in better performance.
For model learning we explore the following procedure: we partition the training data $\mathcal{T}$ at random into a learning set $\mathcal{T}_0$ containing 80% of the data and a test set $\mathcal{T}_1$ containing the remaining 20% of the training data. The latter is used for tracking in-sample over-fitting. We run 500 epochs of the gradient descent algorithm on $\mathcal{T}_0$ (using the `adam` optimizer), and using a callback we select the calibration with the lowest test loss on $\mathcal{T}_1$ among these 500 epochs. This model fitting strategy is described in Listing 12, and this listing also provides the selected hyperparameters.
In Figure 8 we illustrate this learning strategy on models LSTM3 female, GRU3 female, LSTM3 male and GRU3 male (from left to right). The blue lines show the in-sample losses on the learning data $\mathcal{T}_0$ (which is 80% of $\mathcal{T}$), and the cyan dots show the test losses on the remaining 20% of $\mathcal{T}$ (denoted by $\mathcal{T}_1$). These graphs confirm the findings of Section 3.2.1, namely, in general

Listing 12: Training strategy for modeling fitting on the full data $\mathcal{T}$

```
1  model <- GRU3(T0=10, tau0=5, tau1=20, tau2=15, tau3=10, y0=mean(y.train), optimizer='adam')
2
3  # define callback
4  CBs <- callback_model_checkpoint("file.name", monitor = "val_loss", verbose = 0,
5                                   save_best_only = TRUE, save_weights_only = TRUE)
6
7  # gradient descent fitting
8  fit <- model %>% fit(x=x.train, y=y.train, validation_split=0.2,
9                                  batch_size=100, epochs=500, verbose=0, callbacks=CBs)
10 plot(fit)
11
12 # load the best model with the smallest validation loss
13 load_model_weights_hdf5(model, "file.name")
```

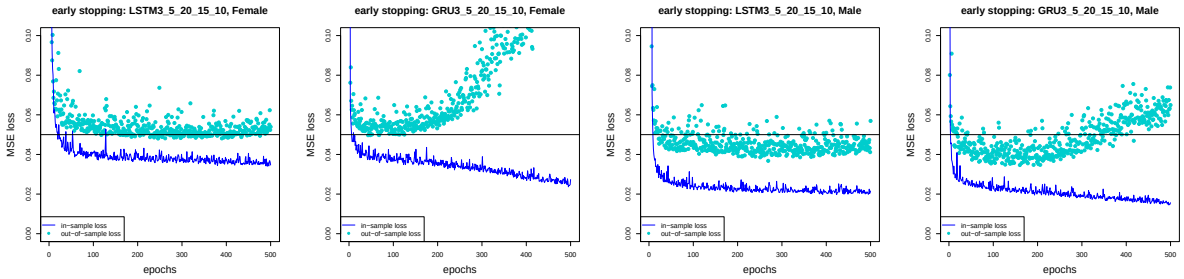

Figure 8: (from left to right) LSTM3 female, GRU3 female, LSTM3 male and GRU3 male in-sample losses (blue color) and out-of-sample losses (cyan color): blue color are 80% of $\mathcal{T}$ (denoted $\mathcal{T}_0$) and cyan color are the remaining 20% of $\mathcal{T}$ (denoted $\mathcal{T}_1$).

the GRU architecture leads to faster convergence (over-fitting), however, the received predictive GRU models vary much more in the seed of the gradient descent algorithm. Therefore, in general, we prefer the LSTM architectures because they lead to more robust results.
Finally, we need to describe the recursive prediction over multiple future periods. This is outlined in Listing 17, below. We use a single for-loop to achieve this recursive prediction (based on the model chosen through the callback, see line 13 of Listing 12).

	in-sample		out-of-sample		run times	
	female	male	female	male	female	male
LSTM3 ($T = 10$, $(\tau_0, \tau_1, \tau_2, \tau_3) = (5, 20, 15, 10)$)	2.5222	6.9458	0.3566	1.3507	225s	203s
GRU3 ($T = 10$, $(\tau_0, \tau_1, \tau_2, \tau_3) = (5, 20, 15, 10)$)	2.8370	7.0907	0.4788	1.2435	185s	198s
LC model with SVD	3.7573	8.8110	0.6045	1.8152	–	–

Table 5: In-sample and out-of-sample MSE losses (figures are in 10^{-4}), and the run times are for 500 epochs (for batch sizes 100).

Table 5 provides the corresponding results. Noticeable is that all chosen RNNs outperform the LC model predictions. This clearly illustrates the strength of the network approach. We could try to further improve the networks by exploring other RNN architectures, note that we have neither explored this option, nor did we (really) fine-tune the calibration procedure for batch-

sizes, number of epochs, drop-out layers, etc. This clearly highlights the power of the RNN approach.
In view of (1.2)-(1.3) we explore what drift is implied by the RNN predictions $(\widehat{M}_{t,x})_{t,x}$. We therefore center the RNN predicted log-mortality rates with the LC estimates $(\widehat{a}_x)_x$, see (1.2),

$$\log(\widehat{M}^{\circ}_{t,x}) = \log(\widehat{M}_{t,x}) - \widehat{a}_x.$$

Then we solve for each future calendar year $2000 \leq t \leq 2016$

$$\arg\min_{k_t} \sum_x \left(\log(\widehat{M}^{\circ}_{t,x}) - \widehat{b}_x k_t\right)^2,$$

where $(\widehat{b}_x)_x$ are the LC estimates received from (1.3).

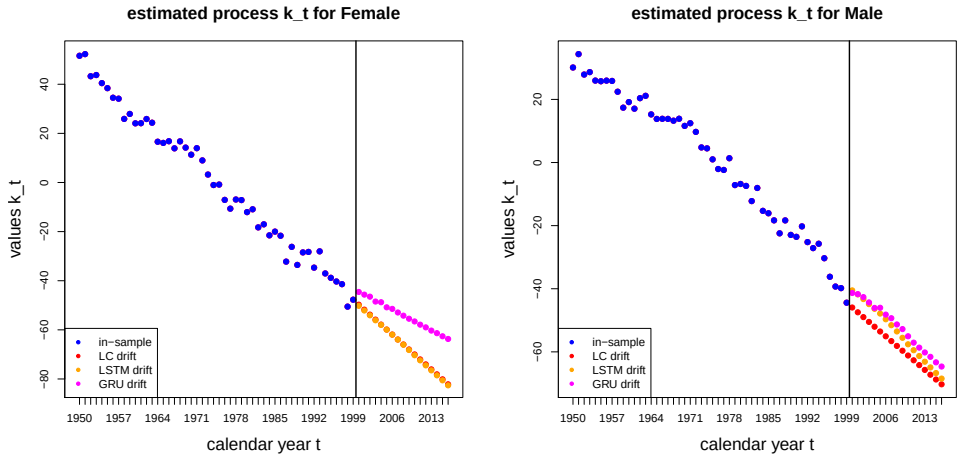

Figure 9: Estimated time series $(\widehat{k}_t)_{t\in\mathcal{T}}$ (in blue color for in-sample) and forecast time series $(\widehat{k}_t)_{t\in\mathcal{V}}$ in red color for LC, orange color for LSTM3 and magenta color for GRU3 of the out-of-sample prediction of the Swiss female population (lhs) and of the Swiss male population (rhs).

The resulting estimated processes $(\widehat{k}_t)_t$ are illustrated in Figure 9. Red color shows the LC forecasts, orange color the LSTM3 forecasts and magenta color the GRU3 forecasts. We observe that the LSTM3 forecast is identical to the LC forecast for females, and for males the LSTM3 forecast has a slightly bigger slope than the LC forecast, converging to the latter. The GRU3 results do not look convincing, i.e. we believe that this RNN architecture may be exhibiting difficulties in coping with the problem posed, or, alternatively, that the parameters $\widehat{b}_x$ from the LC model, which do not vary with time, are not compatible with the forecasts produced by the GRU3 model.

4.3 Modeling both genders simultaneously

Another strength of the RNN approach is that we can learn a RNN model simultaneously on both genders. This is going to be explored in the present section, and in the next section we will study the robustness of this solution. First, we need to discuss data pre-processing. Data

pre-processing is slightly modified compared to the previous section because we add an indicator variable for the genders considered (0 for female and 1 for male). We still use the function defined in Listing 16, but we are going to expand it by a few additional steps. The corresponding R code is given in Listing 18. Note that we alternate the genders in the training data to ensure that the training is taking place on mini-batches with roughly equally distributed genders.

Listing 13: Three hidden layers LSTM model for both genders in the R library `keras`

```
1  LSTM3.Gender <- function(T0, tau0, tau1, tau2, tau3, y0=0, optimizer){
2      Input  <- layer_input(shape=c(T0,tau0), dtype='float32', name='Input')
3      Gender <- layer_input(shape=c(1), dtype='float32', name='Gender')
4
5      RNN = Input %>%
6         layer_lstm(units=tau1, activation='tanh', recurrent_activation='tanh',
7                    return_sequences=TRUE, name='LSTM1') %>%
8         layer_lstm(units=tau2, activation='tanh', recurrent_activation='tanh',
9                    return_sequences=TRUE, name='LSTM2') %>%
10        layer_lstm(units=tau3, activation='tanh', recurrent_activation='tanh', name='LSTM3')
11
12     Output = list(RNN, Gender) %>% layer_concatenate(name="Concat")%>%
13        layer_dense(units=1, activation=k_exp, name="Output",
14                    weights=list(array(0,dim=c(tau3+1,1)), array(log(y0),dim=c(1))))
15     model <- keras_model(inputs = list(Input, Gender), outputs = c(Output))
16     model %>% compile(loss = 'mean_squared_error', optimizer = optimizer)
17      }
```

In the next step we are going to adapt the recurrent neural network definitions. The corresponding R code is given in Listing 13 (for the LSTM case). We expand the previous RNN architectures by adding indicators for gender, which are concatenated with the output of the RNNs layers on lines 12-14 of Listing 13. We are now ready to fit a single network to both genders simultaneously.

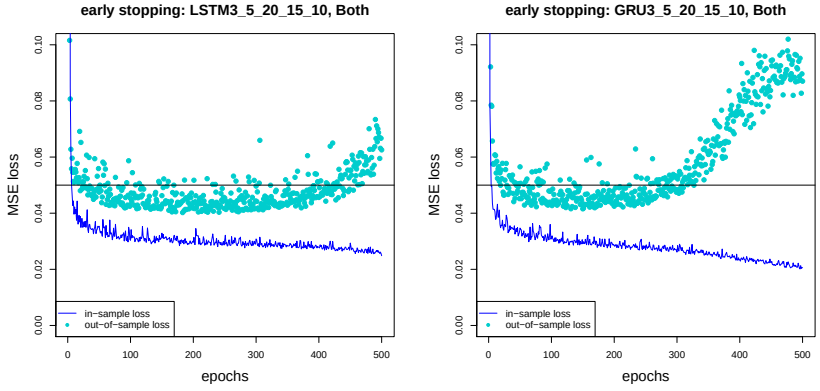

Figure 10: Gradient descent algorithm on both genders simultaneously (lhs) LSTM3 and (rhs) GRU3 in-sample losses (blue color) and out-of-sample losses (cyan color): blue color are 80% of $\mathcal{T}$ (denoted $\mathcal{T}_0$) and cyan color are the remaining 20% of $\mathcal{T}$ (denoted $\mathcal{T}_1$).

We choose the same hyperparameter selection as in the previous Section 4.2 to explore the joint gender calibration. In Figure 10 we provide the corresponding convergence behaviors for the

LSTM and the GRU architectures. Again, we typically observe a faster over-fitting in the GRU architecture at the price that the corresponding results are less stable.

	in-sample	out-of-sample		run times
	both genders	female	male	both genders
LSTM3 ($T = 10$, $(\tau_0, \tau_1, \tau_2, \tau_3) = (5, 20, 15, 10)$)	4.7643	0.3402	1.1346	404s
GRU3 ($T = 10$, $(\tau_0, \tau_1, \tau_2, \tau_3) = (5, 20, 15, 10)$)	4.6311	0.4646	1.2571	379s
LC model with SVD	6.2841	0.6045	1.8152	–

Table 6: Both genders simultaneously: in-sample and out-of-sample MSE losses (figures are in 10^{-4}), and the run times are for 500 epochs (for batch sizes 100).

Using a callback we select the two models (LSTM and GRU) that have the lowest test losses on $\mathcal{T}_1$ in Figure 10. Based on these models we forecast the years after 1999, using the recursive multi-period prediction algorithm outlined above, expanded by a gender indicator in the feature variables. The corresponding loss figures are presented in Table 6. We observe that we receive a substantially improved model compared to the LC model, at least for the prediction of the next 16 years. Moreover, the joint gender LSTM model also outperforms the individual gender models.

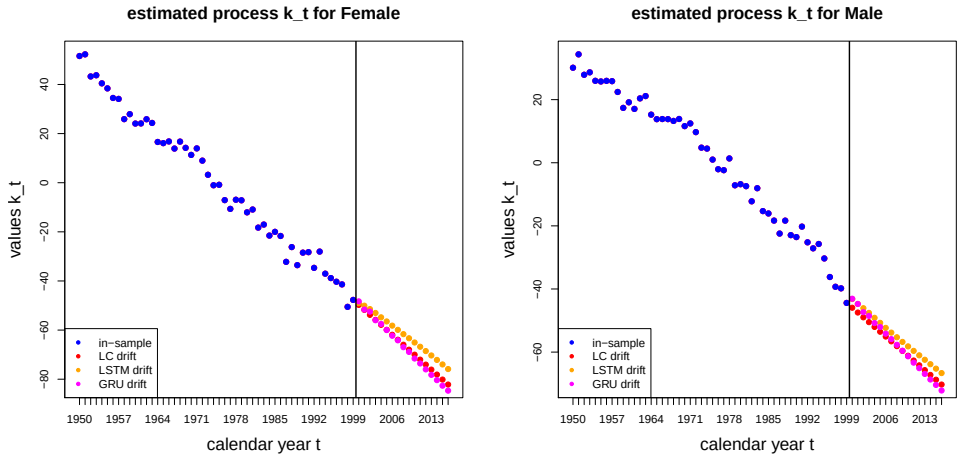

Figure 11: Estimated time series $(\widehat{k}_t)_{t\in\mathcal{T}}$ (in blue color for in-sample) and forecast time series $(\widehat{k}_t)_{t\in\mathcal{V}}$ in red color for LC, in orange color for joint gender LSTM and in magenta color the joint gender GRU out-of-sample prediction of the Swiss female population (lhs) and of the Swiss male population (rhs).

In Figure 11 we illustrate the implied drift in the joint gender RNN architecture. Orange color shows the drift implied by the fitted LSTM architecture. We observe that it is very similar to the LC drift (in red color) which indicates that (probably) the LC model is doing a good predictive job here, in particular for females.

4.4 Robustness of the recurrent neural network solution

An issue using early stopped solutions of gradient descent methods is that the resulting calibrations depend on the chosen seed (starting value) of the algorithm. Of course, this is unpleasant, in particular, because the resulting differences in model performance can be substantial. In this section we analyze the stability of the received predictive results over 100 different early stopped gradient descent calibrations using identical RNN architectures, identical hyperparameters and identical calibration strategies, and only varying the starting value of the gradient descent algorithm over different seeds. We explore this on the common gender LSTM and GRU models, i.e. exactly as in the set-up of Table 6.

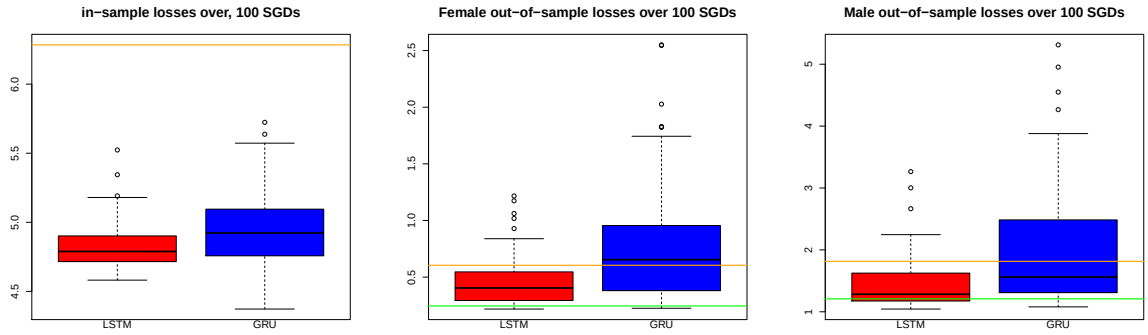

Figure 12: Box plots of 100 iterations of the gradient descent algorithm: red color for LSTM and blue color for GRU architectures (joint gender modeling) with (lhs) in-sample losses, (middle) out-of-sample losses females, and (rhs) out-of-sample losses males; the orange horizontal lines show the LC model results; the green color shows the ensemble LSTM solution.

Figure 12 presents the box plots of the 100 iterations of the gradient descent algorithm (for 100 different seeds). In red color we show the predictions in the joint gender LSTM architecture, and in blue color the predictions in the joint gender GRU architecture. The orange horizontal lines show the LC predictions. The left-hand side of Figure 12 gives the in-sample losses. We observe a substantial decrease in in-sample losses of both RNN architectures compared to the LC model. On average, the LSTM losses are smaller and have less volatility than the GRU ones. The out-of-sample analyses are shown in the middle (females) and the right-hand side (males) of Figure 12. We observe that the LSTM is for almost all of the 100 iterations better than the LC model (for both genders), however, the GRU architecture only performs better than the LC model in roughly 1/2 of the iterations. Thus, it seems that the choice in Table 6 has just been a lucky one for the GRU architecture! Of course, this is rather unpleasant and supports our previous statement that we generally prefer the LSTM architecture for our example, here.
The results from the RNN approaches do not seem to be fully satisfactory, yet, because there is quite some volatility involved, depending on the seed and the early stopped gradient descent solution selected. This is an issue that holds true in general for neural network regression models. A possible way to get rid of some of this volatility is to average over the different predictions received from different seeds, we also refer to Section 5.1.6 of Wüthrich–Buser [14] on network ensembles. The idea is rather simple, namely, we just average the estimated mortality rates

	in-sample	out-of-sample		run times
	both genders	female	male	both genders
LSTM3 ($T = 10$, $(\tau_0, \tau_1, \tau_2, \tau_3) = (5, 20, 15, 10)$)	4.7643	0.3402	1.1346	404s
GRU3 ($T = 10$, $(\tau_0, \tau_1, \tau_2, \tau_3) = (5, 20, 15, 10)$)	4.6311	0.4646	1.2571	379s
LSTM3 averaged over 100 different seeds	–	0.2451	1.2093	$100 \cdot 404$s
GRU3 averaged over 100 different seeds	–	0.2341	1.2746	$100 \cdot 379$s
LC model with SVD	6.2841	0.6045	1.8152	–

Table 7: Both genders simultaneously: in-sample and out-of-sample MSE losses (figures are in 10^{-4}), and the run times are for 500 epochs (for batch sizes 100).

$\widehat{M}_{t,x}$ over the 100 different runs illustrated in Figure 12. These ensemble solutions are given in Table 7. In general, we obtain much more robust solutions, which perform very well in average (and only for very few seeds the individual calibration run may be better); this is illustrated by the green horizontal lines in Figure 12 (middle and rhs). Thus, we see this as a (heuristic) rule that neural network models should be averaged over different calibrations to receive more robust predictive models.

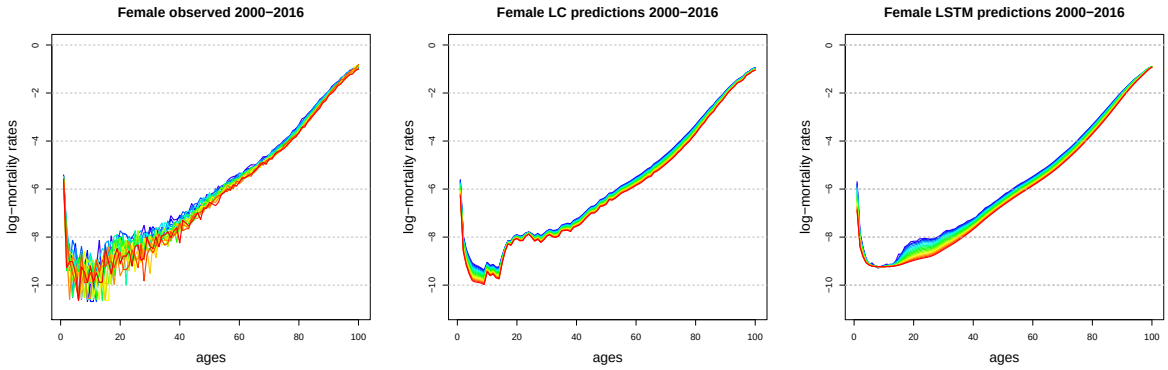

Figure 13: Swiss female population log-mortality rates 2000-2016: (lhs) observed, (middle) LC forecast, and (rhs) LSTM forecast based on information up to 1999.

In Figures 13 and 14 we provide the forecast log-mortality rates of the years 2000 to 2016 of the LC model (middle) and the ensemble LSTM prediction (rhs) for female and male separately. The left-hand side illustrates the observed log-mortality rates. The general observation is that the LSTM approach is better able to capture mortality improvements, whereas the LC rates remain more constant over time, in particular, between ages 20 and 40. The observed values on the left-hand side clearly show (subject to more volatility) that such an improvement is justified. Another well-known fact is that young age mortality rates are rather volatile, and that they are difficult to be captured with a common mortality rate model that is applied simultaneously to all ages. From Figures 13 and 14 we see that the mortality improvements at young ages are bigger than forecast by our approaches. This may have to do with the fact that training data ranging back to 1950 may not be representative for young age mortality rate improvements after 2000. The RNN approaches could be improved by adding a continuous feature component indicating

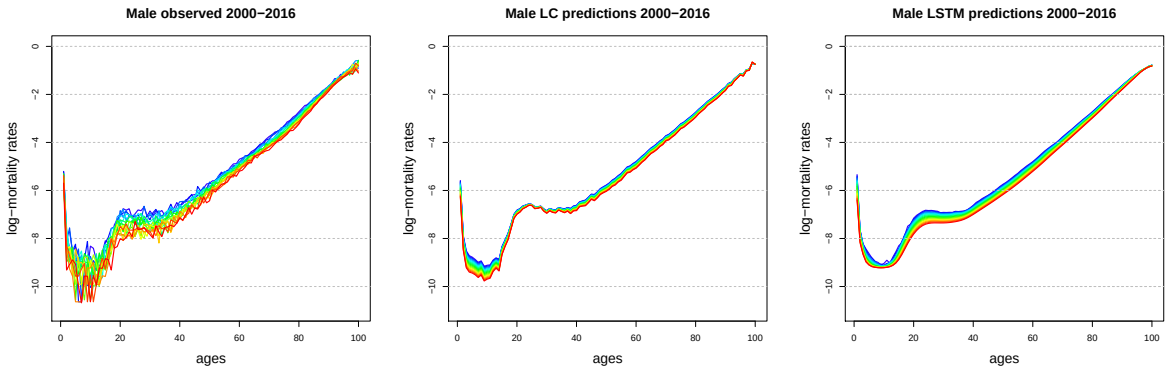

Figure 14: Swiss male population log-mortality rates 2000-2016: (lhs) observed, (middle) LC forecast, and (rhs) LSTM forecast based on information up to 1999.

the calendar year t considered (similar to the gender indicator on lines 12-14 of Listing 13), we also refer to Richman–Wüthrich [10] for extrapolation w.r.t. calendar years. Such a calendar year indicator may give more weight to more recent observations for extrapolating mortality rate improvements. For the time-being we refrain from giving more variants of RNN architectures and we conclude this example.

5 Conclusions

In this tutorial we have introduced and described recurrent neural networks (RNNs). We have studied its two most popular members which are the long short-term memory (LSTM) architecture and the gated recurrent unit (GRU) architecture. These two architectures are suitable to model time series data, and as an explicit example we have studied their predictive performance on Swiss population mortality data. Our illustrative example shows the power of these models, in particular, the LSTM architecture outperforms the classical Lee-Carter mortality prediction approach.

A general observation is that neural network regression models are not very stable over different runs of the gradient descent methods. Therefore, in general, it is advantageous to ensemble (average) over different network predictions.

Our tutorial allows for numerous extensions. For instance, it is rather straightforward to generalize the approach presented to multi-population modeling over different countries, similar to Richman–Wüthrich [10]. Furthermore, we can expand features by any meaningful information. In the present tutorial we neither use the calendar year, the age nor the cohort as feature information for predictions. Such information may further improve the predictive power of the RNN approach. Moreover, we see potential to expand this approach to any kind of individual information such as health state, socio-economic factors, etc., and we refer to Cairns et al. [1] for a recent example. Another direction could be include cause of death information to get a better understanding of observed (and predicted) mortality rates.

In our example, the LSTM architecture has turned out to be more robust than the GRU architecture, therefore, we have a preference for the former. Besides ensembling of predictions one may also further explore adding drop-out layers (and their adaptation to RNNs) for receiving

more robust solutions.
Of course, applications of RNNs are by no means limited to mortality rate predictions. These architectures can be useful for any kind of time series data, with the aim of predicting future observations, and we refer to Smyl [12] for a recent successful application. In addition, RNNs have been applied successfully to other types of sequential data, such as for natural language processing, or for the analysis of video data.

References

[1] Cairns, A.J.G., Kallestrup-Lamb, M., Rosenskjold, C., Blake, D., Dowd, K. (2019). Modelling socio-economic differences in mortality using a new affluence index. To appear in *ASTIN Bulletin* **49/3**.

[2] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv*:1406.1078.

[3] Efron, B., Hastie, T. (2016). *Computer Age Statistical Inference*. Cambridge University Press.

[4] Ferrario, A., Noll, A., Wüthrich, M.V. (2018). Insights from inside neural networks. *SSRN Manuscript* ID 3226852. Version November 14, 2018.

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

[6] Hastie, T., Tibshirani, R., Friedman, J. (2009). *The Elements of Statistical Learning. Data Mining, Inference, and Prediction.* 2nd edition. Springer Series in Statistics.

[7] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. *Neural Computation* **9/8**, 1735-1780.

[8] Lee, R.D., Carter, L.R. (1992). Modeling and forecasting US mortality. *Journal of the American Statistical Association* **87/419**, 659-671.

[9] Richman, R. (2018). AI in actuarial science. *SSRN Manuscript* ID 3218082, Version August 20, 2018.

[10] Richman, R., Wüthrich, M.V. (2019). A neural network extension of the Lee–Carter model to multiple populations. To appear in *Annals of Actuarial Science*.

[11] Schelldorfer, J., Wüthrich, M.V. (2019). Nesting classical actuarial models into neural networks. *SSRN Manuscript* ID 3320525.

[12] Smyl, S. (2019). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. To appear in *International Journal of Forecasting*.

[13] Wilmoth, J.R., Shkolnikov, V. (2010). Human Mortality Database. University of California.

[14] Wüthrich, M.V., Buser, C. (2016). Data analytics for non-life insurance pricing. *SSRN Manuscript* ID 2870308. Version June 5, 2019.

A Appendix: R codes

Listing 14: Choice of data for the toy example

```
mort_rates <- all_mort[which(all_mort$Gender=="Female"), c("Year", "Age", "logmx")]
mort_rates <- dcast(mort_rates, Year ~ Age, value.var="logmx")

T0 <- 10     # lookback period
tau0 <- 3    # dimension of x_t (should be odd for our application)

toy_rates <- as.matrix(mort_rates[which(mort_rates$Year %in% c(2001-T0-1):2001)),])

# note that the first column in toy_rates is the "Year"
xt <- array(NA, c(2,ncol(toy_rates)-tau0, T0, tau0))
YT <- array(NA, c(2,ncol(toy_rates)-tau0))
for (i in 1:2){for (a0 in 1:(ncol(toy_rates)-tau0)){
    xt[i,a0,,] <- toy_rates[c(i:(T0+i-1)),c((a0+1):(a0+tau0))]
    YT[i,a0] <- toy_rates[T0+i,a0+1+(tau0-1)/2]
      }}
```

Listing 15: Choice of data for the time-distributed toy example

```
xt <- array(NA, c(2,ncol(toy_rates)-tau0, T0, tau0))
YT <- array(NA, c(2,ncol(toy_rates)-tau0, T0, 1))
for (i in 1:2){for (a0 in 1:(ncol(toy_rates)-tau0)){
    xt[i,a0,,] <- toy_rates[c(i:(T0+i-1)),c((a0+1):(a0+tau0))]
    YT[i,a0,1] <- toy_rates[c((i+1):(T0+i)),a0+1+(tau0-1)/2]
      }}
```

Listing 16: Pre-processing mortality data for RNNs

```
data.preprocessing.RNNs <- function(data.raw, gender, T0, tau0, ObsYear=1999){
    mort_rates <- data.raw[which(data.raw$Gender==gender), c("Year", "Age", "logmx")]
    mort_rates <- dcast(mort_rates, Year ~ Age, value.var="logmx")
    # selecting data
    train.rates <- as.matrix(mort_rates[which(mort_rates$Year <= ObsYear),])
    # adding padding at the border
    (delta0 <- (tau0-1)/2)
    if (delta0>0){for (i in 1:delta0){
         train.rates <- as.matrix(cbind(train.rates[,1], train.rates[,2], train.rates[,-1], train.rates[,n
       }}
   train.rates <- train.rates[,-1]
   t1 <- nrow(train.rates)-(T0-1)-1
   a1 <- ncol(train.rates)-(tau0-1)
   n.train <- t1 * a1                          # number of training samples
   xt.train <- array(NA, c(n.train, T0, tau0))
   YT.train <- array(NA, c(n.train))
   for (t0 in (1:t1)){
     for (a0 in (1:a1)){
           xt.train[(t0-1)*a1+a0,,] <- train.rates[t0:(t0+T0-1), a0:(a0+tau0-1)]
           YT.train[(t0-1)*a1+a0] <-   train.rates[t0+T0, a0+delta0]
      }}
   list(xt.train, YT.train)
      }
```

Listing 17: Multiple period (recursive) forecast

```
ObsYear <- 1999
T0 <- 10
tau0 <- 5
gender <- "Female"
all_mort2 <- all_mort[which((all_mort$Year > (ObsYear-T0))&(Gender==gender)),]

for (ObsYear1 in ((ObsYear+1):2016)){
     data2 <- data.preprocessing.RNNs(all_mort2[which(all_mort2$Year >= (ObsYear1-10)),],
                                                                gender, T0, tau0, ObsYear1)
     # MinMaxScaler (with minimum and maximum from above)
     x.vali <- array(2*(data2[[1]]-x.min)/(x.min-x.max)-1, dim(data2[[1]]))
     predicted <- all_mort2[which(all_mort2$Year==ObsYear1),]
     keep <- all_mort2[which(all_mort2$Year!=ObsYear1),]
     predicted$logmx <- -as.vector(model %>% predict(x.vali))
     predicted$mx <- exp(predicted$logmx)
     all_mort2 <- rbind(keep,predicted)
     all_mort2 <- all_mort2[order(Gender, Year, Age),]
                       }
```

Listing 18: Pre-processing mortality data for considering both genders simultaneously

```
# training data pre-processing
data0 <- data.preprocessing.RNNs(all_mort, "Female", T0, tau0, ObsYear)
data1 <- data.preprocessing.RNNs(all_mort, "Male", T0, tau0, ObsYear)

xx <- dim(data0[[1]])[1]
x.train <- array(NA, dim=c(2*xx, dim(data0[[1]])[c(2,3)]))
y.train <- array(NA, dim=c(2*xx))
gender.indicator <- rep(c(0,1), xx)
for (l in 1:xx){
   x.train[(l-1)*2+1,,] <- data0[[1]][l,,]
   x.train[(l-1)*2+2,,] <- data1[[1]][l,,]
   y.train[(l-1)*2+1]   <- -data0[[2]][l]
   y.train[(l-1)*2+2]   <- -data1[[2]][l]
          }

# MinMaxScaler data pre-processing
x.min <- min(x.train)
x.max <- max(x.train)
x.train <- list(array(2*(x.train-x.min)/(x.min-x.max)-1, dim(x.train)), gender.indicator)
y0 <- mean(y.train)
```