Stockholms universitet

Neural Networking Beyond Lee–Carter

A Song of Mortality Forecasting and Deep Learning

Jack Zhan

Masteruppsats i försäkringsmatematik
Master Thesis in Actuarial Mathematics

Masteruppsats 2025:11
Försäkringsmatematik
Juni 2025

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Stockholm University

Mathematical Statistics
Stockholm University
Master Thesis **2025:11**
http://www.math.su.se

Neural Networking Beyond Lee–Carter

A Song of Mortality Forecasting and Deep Learning

Jack Zhan*

June 2025

Abstract

In recent years, numerous research papers have explored the application of deep learning in the field of actuarial science. This thesis aims to provide an overview of one such application: the use of deep learning models for mortality forecasting. Starting from the Lee–Carter model, we explore how neural networks can be employed to extrapolate the time index κ_t. We describe parameter estimation in the Lee–Carter model and outline traditional forecasting methods, where κ_t is typically modelled using an autoregressive integrated moving average (ARIMA) model—typically a random walk with drift. We then present an overview of artificial neural networks, with a focus on recurrent neural networks and their most widely used variant, the long short-term memory network (LSTM). These methods are applied to Swedish mortality data. We compare feedforward neural networks, shallow LSTMs, and deep LSTMs to traditional ARIMA-based forecasting. Ensemble models are used to reduce the randomness inherent in neural network training. Our results show that, for the given dataset, neural networks generally outperform traditional methods.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: zhanjack01@gmail.com. Supervisor: Mathias Lindholm.

Preface

This thesis constitutes the degree project corresponding to 30 ECTS credits for a degree of Master of Science (*filosofie master*) in actuarial mathematics at the Department of Mathematics, Stockholm University.

Acknowledgements

I want to express my gratitude to my supervisor, Mathias Lindholm, for the deep and engaging discussions on mortality modelling, the insightful insights, and for recommending relevant literature. Thank you for taking the time to respond to my long emails, even if not always promptly. In perpetuity of thought, I would also like to express my gratitude to my friends and fellow students Leo Gustaf Levenius and Luciano Alexander Mattias Egusquiza Castillo, who wrote their theses alongside mine. The long hours we spent together in a study room, together with one too many *fika* and way too many distractions, made this semester far more bearable—and even enjoyable. Thank you for always patiently listening to my late-night rants about my thesis and for your constructive peer reviews.

AI Statement

AI tools, such as ChatGPT, were used throughout the development of this thesis. Specifically, they helped debug R code, troubleshoot LaTeX error messages, and improve grammar and phrasing. AI tools also provided helpful suggestions for LaTeX and R packages—for example, recommending the use of the `tikzDevice` package in R to convert R plots into Ti*k*Z code. Additionally, ChatGPT was used to translate the abstract into Swedish for the "Sammanfattning" section; the translation was subsequently reviewed and adjusted by the author. Moreover, this AI statement was written using ChatGPT.

Contents

Chapter 1

Introduction and Theory

Deep learning is an increasingly popular research area. It has been applied to various problems within the actuarial field, such as claims frequency and severity modelling, fraud detection, and mortality forecasting. For example, Wüthrich and Merz [22, chs. 7–11] have written extensively about deep learning models and applications in the actuarial field; Egusquiza Castillo [6] have compared the performance of various machine learning models, such as feed-forward neural networks and combined actuarial neural networks, on claim frequency modelling. For a more general introduction to deep learning, see, e.g., Goodfellow *et al.* [7].

This thesis focusses on mortality forecasting. Mortality forecasting is an essential part of life insurance. The Lee–Carter model [12] is a common approach to model and forecast the mortality of a single population. With the rising popularity of deep learning and artificial neural networks, how can we implement these methods in mortality forecasting?

We aim to provide both a theoretical and a practical understanding of the Lee–Carter model and artificial neural networks. In Section 1.1, we describe the Lee–Carter model and outline traditional forecasting methods. In Section 1.2, we explain what artificial neural networks are and how to train them. We provide a more detailed explanation of some common network architectures that are especially relevant in modelling sequential data, such as the long short-term memory network (LSTM). In Chapter 2, we end the thesis with a numerical illustration, where we put all the theory to the test on Swedish mortality data. We compare traditional forecasting methods with deep learning.

1.1 Lee–Carter Model

The *Lee–Carter model* was pioneered by Ronald D. Lee and Lawrence R. Carter [12] in 1992. The model models the force of mortality.

Definition 1.1.1 (Force of mortality) Let T_{t-x} be a random variable defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, corresponding to the calendar year in which an individual born in calendar year $t-x$ dies. The *force of mortality* (or simply *mortality*) is defined as

$$\mu_{x,t} := \lim_{\Delta t \to 0^+} \frac{\mathbb{P}(T_{t-x} < t + \Delta t \mid T_{t-x} > t)}{\Delta t}. \tag{1.1}$$

We can interpret (1.1) as the instantaneous death rate at calendar year t for an x-year-old. Throughout this thesis, we assume that we have *piecewise constant forces of mortality*, i.e., given any age x and calender year t, we have

$$\mu_{x,t} = \mu_{\lfloor x \rfloor, \lfloor t \rfloor},$$

where $\lfloor \cdot \rfloor$ is the floor function that maps a real number to the greatest integer less than or equal to itself.

Let $d_{x,t}$ denote the observed number of deaths of x-year-olds in calendar year t and let $r_{x,t}$ denote the corresponding observed central exposure-to-risk, i.e., $r_{x,t}$ is the number of person years from which $d_{x,t}$ occurred. With the assumption of piecewise constant forces of mortality, the forces of mortality can be estimated with the *central death rate*

$$m_{x,t} := \frac{d_{x,t}}{r_{x,t}}.$$

We present the Lee–Carter model as in [12] for completeness. But in the practical part of this thesis (see Chapter 2) we do not use this version of the Lee–Carter model. Instead, we implement the package `StMoMo`, which uses the same model assumptions as in [3].

1.1.1 The Model

Lee and Carter [12] proposed a simple log-bilinear form for the force of mortality $\mu_{x,t}$, which means that the log-mortality $\log(\mu_{x,t})$ is linear in both age and calendar year effects. For $m_{x,t}$ (or some other empirical estimate of $\mu_{x,t}$), we assume that

$$\log(m_{x,t}) = \alpha_x + \beta_x \kappa_t + \varepsilon_{x,t}, \tag{1.2}$$

where the $\varepsilon_{x,t}$ are homoscedastic error terms with mean 0. We can interpret $\exp\{\alpha_x\}$ as the general shape of the mortality for age x. The parameter β_x indicates how sensitive the force of mortality of age x is to changes in κ_t. Lastly, the time trend is represented by the time index κ_t. For parameter identifiability, we impose the constraints

$$\sum_{t \in \mathscr{T}} \kappa_t = 0, \tag{1.3}$$

$$\sum_{x \in \mathscr{X}} \beta_x = 1, \tag{1.4}$$

where $\mathscr{T}$ is the set of all the calendar years included in our model, and $\mathscr{X}$ is the set of all ages included in our model.

Mortality forecasting using this model follows a two-step procedure. First, we estimate the parameters

$$\boldsymbol{\alpha} := (\alpha_x)_{x \in \mathscr{X}},$$
$$\boldsymbol{\beta} := (\beta_x)_{x \in \mathscr{X}},$$
$$\boldsymbol{\kappa} := (\kappa_t)_{t \in \mathscr{T}},$$

using ordinary least squares (OLS). Secondly, we model and forecast $\boldsymbol{\kappa}$ as an *autoregressive integrated moving average* (ARIMA) process (see Appendix A).

Lee and Carter [12] originally used the Lee–Carter model on aggregate US data (sexes combined). They found that the life tables produced using the model closely fit actual US life tables. They also trained a model on the earlier part of the data to forecast the later part and found that the forecast performed well.

The Lee–Carter model is extrapolative, meaning that it forecasts future mortality rates based solely on past trends. As with all extrapolative models, the Lee–Carter model assumes that the future will be similar to the past. The model does not attempt to incorporate factors such as medical advances, behavioural changes, or social changes that might influence mortality. However, since we lack a precise understanding of these mechanisms and their intricate interactions, having an extrapolative approach to prediction is particularly compelling in the case of human mortality.

Another questionable assumption of the model is the assumption of homoscedasticity of the error terms $\varepsilon_{x,t}$. This is unrealistic since the (logarithm of the) observed force of mortality is much more variable at older ages because of lower death counts compared to younger ages. Brouhns *et al.* [3] propose that because the number of deaths is a counting process $D_{x,t}$, the Poisson assumption seems plausible. (Further motivation for the Poisson assumption is provided in [1].) Therefore, the authors suggest a different approach from the OLS method used by Lee and Carter [12]. We consider

$$D_{x,t} \mid r_{x,t} \sim \text{Pois}(r_{x,t}\mu_{x,t}), \tag{1.5}$$

with

$$\mu_{x,t} = \exp\{\alpha_x + \beta_x \kappa_t\}. \tag{1.6}$$

Thus, the force of mortality is assumed to have the same log-bilinear form as the original Lee–Carter model. The parameters are now estimated using maximum likelihood.

Remark 1 Brouhns *et al.* [3] did not modify how $\boldsymbol{\kappa}$ is forecasted.

1.1.2 Parameter Estimation

The parameter estimation is done with ordinary least squares in [12], using singular value decomposition. In [3], the authors used maximum likelihood estimation instead. We now provide a more detailed explanation of both estimation methods.

Ordinary Least-Squares

Lee and Carter [12] estimated the parameters of the Lee–Carter model by *ordinary least squares* (OLS). This means that we want to find $\hat{\boldsymbol{\alpha}}$, $\hat{\boldsymbol{\beta}}$, and $\hat{\boldsymbol{\kappa}}$ such that they minimise

$$\sum_{x \in \mathscr{X}} \sum_{t \in \mathscr{T}} \left(\log(m_{x,t}) - \alpha_x - \beta_x \kappa_t\right)^2. \tag{1.7}$$

Since there are no known covariates in (1.2), we cannot use standard regression methods. Instead, (1.7) is minimised by letting

$$\hat{\alpha}_x = \frac{1}{|\mathscr{T}|}\sum_{t\in\mathscr{T}} \log\big(m_{x,t}\big), \quad \forall x \in \mathscr{X}. \tag{1.8}$$

Define the (observed) centred force of mortality $y_{x,t}$ by

$$y_{x,t} = \log\big(m_{x,t}\big) - \hat{\alpha}_x. \tag{1.9}$$

To find $\hat{\boldsymbol{\beta}}$ and $\hat{\boldsymbol{\kappa}}$, we use singular value decomposition (SVD) on the matrix $\mathbf{y} \coloneqq (y_{x,t})_{x\in\mathscr{X},t\in\mathscr{T}}$. We have

$$\mathbf{y} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top, \tag{1.10}$$

where

$$\mathbf{U} = \big(\mathbf{u}_1, \dots, \mathbf{u}_q\big) \in \mathbb{R}^{n\times q},$$
$$\mathbf{V} = \big(\mathbf{v}_1, \dots, \mathbf{v}_q\big) \in \mathbb{R}^{q\times q},$$

with $\mathbf{U}^\top\mathbf{U} = \mathbf{V}^\top\mathbf{V} = \mathbf{I}_q$, and

$$\boldsymbol{\Sigma} = \operatorname{diag}(\lambda_1, \dots, \lambda_q) \in \mathbb{R}^{q\times q}$$

is a diagonal matrix constituted of the singular values of $\mathbf{y}$ in descending order. Then, we have $\hat{\boldsymbol{\beta}} = \lambda_1\mathbf{u}_1$ and $\hat{\boldsymbol{\kappa}} = \mathbf{v}_1$. A more detailed explanation of this estimation process is given in, e.g., [22].

The model's estimated number of deaths in year t may not be the same as the observed number of deaths in year t, since we model the log-mortality. To fix this, Lee and Carter [12] reestimate $\boldsymbol{\kappa}$ with $\hat{\hat{\boldsymbol{\kappa}}}$ such that these new estimates, along with $\hat{\boldsymbol{\alpha}}$ and $\hat{\boldsymbol{\beta}}$ gives the correct number of deaths, i.e., we choose $\hat{\hat{\boldsymbol{\kappa}}}$ such that

$$\sum_{x\in\mathscr{X}} d_{x,t} = \sum_{x\in\mathscr{X}} r_{x,t}\exp\big(\hat{\alpha}_x + \hat{\beta}_x\hat{\hat{\kappa}}_t\big). \tag{1.11}$$

Several advantages of this reparameterisation are discussed in [12].

Maximum Likelihood Estimation

The Poisson approach in [3] uses *maximum likelihood estimation* instead of OLS. The log-likelihood function based on (1.5) and (1.6) is, up to an additive constant, given by

$$\ell(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\kappa}) = \sum_{x\in\mathscr{X}}\sum_{t\in\mathscr{T}} \big(d_{x,t}(\alpha_x + \beta_x\kappa_t) - r_{x,t}\exp\{\alpha_x + \beta_x\kappa_t\}\big)\,. \tag{1.12}$$

We aim to find $\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\kappa}$ that maximises (1.12). The authors in [3] used the Newton–Raphson method, which finds the maximum of a log-likelihood function ℓ with parameter $\boldsymbol{\theta}$ using the iterative update

$$\hat{\boldsymbol{\theta}}^{(k+1)} = \hat{\boldsymbol{\theta}}^{(k)} - \frac{\partial\ell(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}^{(k)}} \cdot \left(\frac{\partial^2\ell(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}^2}\bigg|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}^{(k)}}\right)^{-1}, \tag{1.13}$$

starting from an appropriate initial value $\hat{\boldsymbol{\theta}}^{(0)}$.

In (1.12), the parameters consist of three vectors: $\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\kappa}$. For all $x \in \mathscr{X}$ and $t \in \mathscr{T}$, we initialise the Newton–Raphson method with $\hat{\alpha}_x^{(0)} = 0$, $\hat{\beta}_x^{(0)} = 0$, and $\hat{\kappa}_x^{(0)} = 1$, as in [3]. The iterative updates are then given by

$$\hat{\alpha}_x^{(k+1)} = \hat{\alpha}_x^{(k)} - \frac{\sum_t\big(d_{x,t} - \hat{d}_{x,t}^{(k)}\big)}{-\sum_t \hat{d}_{x,t}^{(k)}}, \qquad \forall x \in \mathscr{X};$$

$$\hat{\kappa}_t^{(k+1)} = \hat{\kappa}_t^{(k)} - \frac{\sum_x\big(d_{x,t} - \hat{d}_{x,t}^{(k)}\big)\hat{\beta}_x^{(k)}}{-\sum_x \hat{d}_{x,t}^{(k)}\big(\hat{\beta}_x^{(k)}\big)^2}, \qquad \forall t \in \mathscr{T};$$

$$\hat{\beta}_x^{(k+1)} = \hat{\beta}_x^{(k)} - \frac{\sum_t\big(d_{x,t} - \hat{d}_{x,t}^{(k)}\big)\hat{\kappa}_t^{(k+1)}}{-\sum_t \hat{d}_{x,t}^{(k)}\big(\hat{\kappa}_t^{(k+1)}\big)^2}, \qquad \forall x \in \mathscr{X};$$

where $\hat{d}_{x,t}^{(k)} = r_{x,t}\exp\{\hat{\alpha}_x^{(k)} + \hat{\beta}_x^{(k)}\hat{\kappa}_t^{(k)}\}$ is the estimated death count after iteration step k. The algorithm stops when the increase in the log-likelihood becomes smaller than a predefined threshold; the authors in [3] suggest a value of 10^{-10}.

Remark 2 Since we directly model the death count, we do not have to re-estimate κ_t as in [12].

Remark 3 Note that the updates in the Newton–Raphson method for the Lee–Carter model, as presented above, use the most recent estimates, rather than the estimates of the previous iteration. In particular, the equation for $\hat{\beta}_x^{(k+1)}$ uses $\hat{\kappa}_t^{(k+1)}$ instead of $\hat{\kappa}_t^{(k)}$. This implies that we do not update all parameters simultaneously. Instead, a sequential (or block coordinate) update scheme is used, where the parameters are updated one vector at a time in a specific order. This implementation of the Newton–Raphson method is the same as in [3, p. 379].

1.1.3 Forecasting

The time index $\boldsymbol{\kappa}$ is typically treated as an *autoregressive integrated moving average* (ARIMA) model; see, e.g., [12, 3]. Typically, the *Box–Jenkins method* [2] is used on $\hat{\boldsymbol{\kappa}}$ to identify the appropriate ARIMA model, estimate the parameters, and create forecasts for future values. An overview of the Box–Jenkins method is given in Appendix A. Box *et al.* [2, pp. 177–392] give a comprehensive description of the method. In most historical applications of the Lee–Carter model, a *random walk with drift* (RWD) works well, i.e., we use the model

$$\kappa_t = \delta + \kappa_{t-1} + \varepsilon_t, \tag{1.14}$$

with the drift parameter δ, where we have i.i.d.

$$\varepsilon_t \sim \mathcal{N}\left(0, \sigma^2\right).$$

In practice, because of parsimony, unless we can find a substantially better model, we use the RWD for $\boldsymbol{\kappa}$.

With modern technology, computer programmes can evaluate a large set of models and choose the best one based on, e.g., the Akaike's information criterion (AIC). This is implemented in the `auto.arima` function in the `forecast` package of R. The `auto.arima` function is one of the methods used in Chapter 2 to model $\boldsymbol{\kappa}$.

1.1.4 StMoMo and GAPC

In Chapter 2, we use the `StMoMo`[1] package in R to implement the Lee–Carter model. `StMoMo` contains functions to build models in the family of generalised age-period-cohort (GAPC) stochastic mortality models, introduced in [20]. Below, we give a brief overview of GAPC models.

We assume that the number of deaths $D_{x,t}$, conditioned on the central exposure-to-risk $r_{x,t}$, follows a Poisson distribution, such that

$$D_{x,t} \mid r_{x,t} \sim \text{Pois}\left(r_{x,t}\mu_{x,t}\right), \tag{1.15}$$

where $\mu_{x,t}$ is the force of mortality. Moreover, given a link function g, we assume that

$$g(\mu_{x,t}) = \alpha_x + \sum_{k=1}^{N} \beta_x^{(k)} \kappa_t^{(k)} + \beta_x^{(0)} \gamma_{t-x}, \tag{1.16}$$

for the force of mortality. Usually, the canonical link function is chosen. For the Poisson distribution, this is the log link function $g(x) = \log x$.

In (1.16), we have the following interpretations of the parameters:

- The term α_x is a static age function that captures the general shape of mortality by age.
- The integer $N \geq 0$ is the number of age-period terms, with each time index $\kappa_t^{(k)}$ describing the mortality trend and $\beta_x^{(k)}$ modulating its effects across ages.
- The term γ_{t-x} accounts for the cohort effect with $\beta_x^{(0)}$ modulating its effect across ages.

The age-modulating terms $\beta_x^{(k)}$ can be either pre-specified functions of age or estimated non-parametrically. A limitation of the `StMoMo` package is that either all $\beta_x^{(k)}$ must be non-parametric or none at all.

Since most stochastic mortality models are only identifiable up to a transformation, we need to impose parameter constraints to ensure identifiability and uniqueness.

[1] The acronym StMoMo stands for *stochastic mortality model* and is pronounced as "Saint Momo". Momo is the king of Carnivals in numerous Latin American festivities, see [21].

Alternatively, we may model the one-year death probability $q_{x,t}$ instead of the force of mortality. We then assume that the number of deaths, conditioned on the initial exposure-to-risk $l_{x,t}$ (the number of x-year-olds alive at the beginning of calendar year t), follows a binomial distribution, such that

$$D_{x,t} \mid l_{x,t} \sim \text{Bin}\big(l_{x,t}, q_{x,t}\big). \tag{1.17}$$

We replace $\mu_{x,t}$ in (1.16) with $q_{x,t}$. The canonical link function of the binomial is the logit link function $g(x) = \log\big(\frac{x}{1-x}\big)$.

The GAPC stochastic mortality models are similar to generalised linear models (GLMs) and generalised non-linear models (GNMs). We assume that the response variable $D_{x,t}$ is generated from a particular distribution. The exposure-to-risk can be compared to weights in a GLM. The conditional mean of the mortality rate is given by (1.16), similar to the structure of the conditional mean of a GLM.

If we use the implementation of the Lee–Carter model in [3], it is apparent that the Lee–Carter model falls under the GAPC stochastic mortality models. The conditional distribution in (1.16) is assumed. We use the log link function and we assume that $N = 1$, $\beta_x^{(0)} = 0$ and that $\beta_x = \beta_x^{(1)}$ is non-parametric. However, whilst Brouhns *et al.* [3] use Newton–Raphson for parameter estimation, `StMoMo` uses the `gnm` function of the package `gnm`, which is designed to model and fit generalised non-linear models.

Other widely used GAPC stochastic mortality models include the Renshaw and Haberman model [18], which extends the Lee–Carter model by incorporating a cohort effect; the age-period-cohort (APC) model, commonly used in medicine and demography, which assumes fixed age effects with $\beta_x^{(0)} = \beta_x^{(1)} = 1$; and the Cairns–Blake–Dowd (CBD) model [4], which omits α_x, sets $\beta_x^{(1)} = 1$, and uses $\beta_x^{(2)} = x - \bar{x}$, where $\bar{x}$ is the average age in the data set. A more comprehensive overview of these models and other GAPC stochastic mortality models is presented in [20].

1.2 Deep Learning Models

Deep learning is a subset of machine learning that focuses on *artificial neural networks* (ANNs or NNs). Most deep learning models are designed for supervised learning. This means that we want to approximate the output $\mathbf{Y}$ with the input $\mathbf{X}$. More specifically, given a *loss function* (sometimes known as cost function) $C(\mathbf{Y}, f(\mathbf{X}))$, which penalises the errors in prediction, we want to find a function f^* such that it minimises the *expected prediction error*, i.e.,

$$f^* = \arg\min_f \mathbb{E}[C(\mathbf{Y}, f(\mathbf{X}))]. \tag{1.18}$$

When we have a scalar response and a loss function of the form

$$C(y, m) = \varphi(y) - \varphi(m) - \varphi'(m)(y - m), \tag{1.19}$$

where φ is a strictly convex function that is well-behaved at the endpoints of the range of y, (1.18) is minimised by $f^*(\mathbf{X}) = \mathbb{E}[\mathbf{Y} \mid \mathbf{X}]$ (among all functions for which the expectation (1.18) exists finitely), *cf.* [15]. We show this for the mean squared error, for which $\varphi(y) = y^2$ and $C(y, m) = (y - m)^2$. Let $f^*(\mathbf{X}) = \mathbb{E}[\mathbf{Y} \mid \mathbf{X}]$. We want to prove that

$$\mathbb{E}\big[(\mathbf{Y} - f(\mathbf{X}))^2\big] \geq \mathbb{E}\big[(\mathbf{Y} - f^*(\mathbf{X}))^2\big], \tag{1.20}$$

for any function f for which the expectation exists. Let $\Delta = f(\mathbf{X}) - f^*(\mathbf{X})$. We have

$$\begin{aligned}\mathbb{E}\big[(\mathbf{Y} - f(\mathbf{X}))^2\big] &= \mathbb{E}\big[(\mathbf{Y} - f^*(\mathbf{X}) - \Delta)^2\big] \\ &= \mathbb{E}\big[(\mathbf{Y} - f^*(\mathbf{X}))^2\big] - 2\mathbb{E}[\Delta(\mathbf{Y} - f^*(\mathbf{X}))] + \mathbb{E}\big[\Delta^2\big].\end{aligned}$$

Since Δ is $\mathbf{X}$-measurable, we have

$$\begin{aligned}\mathbb{E}[\Delta(\mathbf{Y} - f^*(\mathbf{X}))] &= \mathbb{E}[\mathbb{E}[\Delta(\mathbf{Y} - f^*(\mathbf{X})) \mid \mathbf{X}]] \\ &= \mathbb{E}[\Delta\mathbb{E}[\mathbf{Y} - f^*(\mathbf{X}) \mid \mathbf{X}]] \\ &= \mathbb{E}[\Delta(\mathbb{E}[\mathbf{Y} \mid \mathbf{X}] - f^*(\mathbf{X}))] \\ &= 0.\end{aligned}$$

Therefore, we have

$$\mathbb{E}\big[(\mathbf{Y} - f(\mathbf{X}))^2\big] = \mathbb{E}\big[(\mathbf{Y} - f^*(\mathbf{X}))^2\big] + \mathbb{E}\big[\Delta^2\big],$$

where $\mathbb{E}[\Delta^2] \geq 0$, showing (1.20).

In practice, we have a training set $(\mathbf{x}_i, \mathbf{y}_i)_{1\leq i\leq n}$ and we approximate f^* by minimising the empirical version of (1.18) given by

$$\frac{1}{n}\sum_{k=1}^{n} C(\mathbf{y}_i, f(\mathbf{x}_i)), \tag{1.21}$$

for f in a sufficiently large family of functions $\mathscr{F}$.

For artificial neural networks, we look at functions $f(\mathbf{x}) = f(\mathbf{x};\boldsymbol{\theta})$, which are compositions of multiple functions, and we want to find the parameters $\boldsymbol{\theta}$ that minimises the *objective function*

$$\mathscr{L}(\boldsymbol{\theta};\mathbf{x},\mathbf{y}) = \frac{1}{n}\sum_{k=1}^{n} C(\mathbf{y}_i, f(\mathbf{x}_i;\boldsymbol{\theta})) + \text{regularisation}. \tag{1.22}$$

More specifically, artificial neural networks consist of layers of interconnected nodes called *neurons*.[2] Each neuron receives a set of inputs, computes a weighted sum of the inputs, adds a bias term, and then applies a non-linear *activation function*.[3] The activation function has a similar role to the inverse link function in a generalised linear model, in introducing non-linearity into the model. Table 1.1 shows three common activation functions and their derivatives.

Table 1.1 Common activation functions and their derivatives.[a]

Name	Activation Function	Derivative
Standard logistic (expit)	$\phi(x) = (1+e^{-x})^{-1}$	$\phi'(x) = \phi(x)(1-\phi(x))$
Hyperbolic tangent	$\phi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\phi'(x) = 1 - \phi^2(x)$
ReLU	$\phi(x) = \max(0, x)$	$\phi'(x) = \mathbb{1}_{\{x>0\}}$

[a] The rectified linear unit (ReLU) does not have a derivative at $x = 0$. However, the convention is to put $\phi'(0) = 0$.

We can put the layers of neurons into three categories: *input layer*, *hidden layers*, and *output layer*. The first layer is called the *input layer*, which receives the input $\mathbf{x}$, usually pre-processed in some way. For categorical features, we typically use one-hot encoding, which encodes every category of the feature with a unit vector. Assume that we have a categorical feature with K levels $\{a_1, \ldots, a_K\}$. We have the raw feature components $\tilde{x}_{i,j} \in \{a_1, \ldots, a_K\}$, with $1 \leq i \leq n$ denoting observation i and the second index denoting the jth feature component. One-hot encoding is obtained by the embedding map

$$\tilde{x}_{i,j} \mapsto \mathbf{x}_{i,j} = \left(\mathbb{1}_{\{\tilde{x}_{i,j}=a_1\}}, \ldots, \mathbb{1}_{\{\tilde{x}_{i,j}=a_K\}}\right)^\top. \tag{1.23}$$

Another common way to pre-process categorical variables is with *embedding layers* (see, e.g., [22, pp. 298–302]). We do not delve into details.

Continuous features typically do not need to be pre-processed. However, for efficient training, we want all feature components to live on a similar scale and be roughly uniformly spread across their domains, see [22, p. 294]. Two common ways to achieve this are the *min-max scaler* and *normalisation*. For the min-max scaler, let $\tilde{x}_j^-$ and $\tilde{x}_j^+$ be the minimal and maximal possible value for the raw continuous feature components $\tilde{x}_{i,j}$, i.e., $\tilde{x}_{i,j} \in [\tilde{x}_j^-, \tilde{x}_j^+]$. We transform each observed value $\tilde{x}_{i,j}$ by

$$\tilde{x}_{i,j} \mapsto x_{i,j} = 2\frac{\tilde{x}_{i,j} - \tilde{x}_j^-}{\tilde{x}_j^+ - \tilde{x}_j^-} - 1. \tag{1.24}$$

The resulting feature values $(x_{i,j})_{1\leq i\leq n}$ take values on the interval $[-1, 1]$. They should also be roughly uniformly spread across this interval, according to [22].

Normalisation means centering the data around the empirical mean $\bar{\tilde{x}}_j$ and dividing with the empirical standard deviation $\hat{\sigma}_j$ of $(\tilde{x}_{i,j})_{0\leq i\leq n}$, i.e.,

$$\tilde{x}_{i,j} \mapsto x_{i,j} = \frac{\tilde{x}_{i,j} - \bar{\tilde{x}}_{i,j}}{\hat{\sigma}_j}. \tag{1.25}$$

[2] Artificial neural networks are inspired by the structure and function of the brain, hence the name "neurons" for the nodes in the network.
[3] The activation function does not have to be non-linear, as the identity function is sometimes used as an activation function.

The *output layer* produces $\hat{\mathbf{y}}$, the final output of the network, which is used to approximate f^*. The intermediate layers do not have a pre-defined purpose, and it is up to the learning algorithm to find the best use of these. Vaguely, we could say that these intermediate layers automate the process of feature engineering. Since the output of the intermediate layers is not seen, these layers are called *hidden layers*.

The number of hidden layers is sometimes called the depth, as in [22]. However, in this thesis, we use the definition of [7], i.e., the *depth* d of a network is the number of layers excluding the input layer, or the number of transformations of the input $\mathbf{x}$. The number of neurons in a layer is sometimes called the *width* of the layer. The depth, the widths of the hidden layers, and the activation functions, as well as how layers are connected, are all hyperparameters of the neural network. In this thesis, we define a deep neural network as an artificial neural network with at least two hidden layers, i.e., $d > 2$.

In the rest of this section, we introduce some common neural network architectures and also how to fit the networks.

1.2.1 Feedforward Neural Networks

Feedforward neural networks (FNNs) are the simplest type of neural networks, and they form the foundational structure of many deep learning models. Feedforward refers to the unidirectional flow of information through the network, starting from the input $\mathbf{x}$, passing through hidden layers, and finally reaching the output $\hat{\mathbf{y}}$. In this chapter, we describe *fully connected networks* (FCNs), often referred to as *multilayer perceptrons* (MLPs), or "vanilla" networks. This architecture consists of only *fully connected layers*, sometimes referred to as *dense layers*. Every neuron in a dense layer is connected to every neuron in the previous layer.

In the following, we provide a more mathematical formulation of the hidden layers of a fully connected network. The mapping f of the output is typically the composition of several different functions. As mentioned before, the input $\mathbf{x}$ is fed into the network through the *input layer*. The information then passes through several layers $\mathbf{h}^{(k)}$, for $k \in \{1, \dots, d\}$, where $d \in \mathbb{Z}^+$ is the depth of the network.

A layer transforms the output of the previous layer via the mapping

$$\begin{aligned} f^{(k)} : \mathbb{R}^{q_{k-1}} &\to \mathbb{R}^{q_k}, \\ \mathbf{h}^{(k-1)} &\mapsto \mathbf{h}^{(k)}, \end{aligned}$$

which is defined as

$$f^{(k)}(\mathbf{h}^{(k-1)}) = \phi_k(\mathbf{a}^{(k)}), \tag{1.26}$$

for some activation function ϕ_k (see Table 1.1), applied element-wise on the affine transformation

$$\mathbf{a}^{(k)} := \mathbf{W}^{(k)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k)}, \tag{1.27}$$

of $\mathbf{h}^{(k-1)}$. Here, $\mathbf{W}^{(k)}$ is the the matrix of *network weights* and $\mathbf{b}^{(k)}$ is the vector of *biases*. The dimension q_k, for $k \in \{1, \dots, d\}$, is the width of layer k. The *input dimension* q_0, sometimes called the *input size*, is defined as the dimension of $\mathbf{x}$.

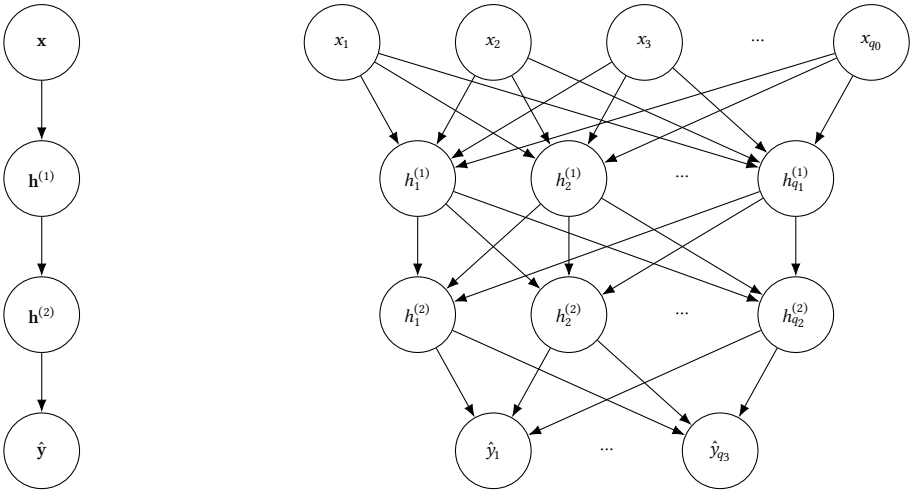
Instead of the layer representation of (1.26), we could define the mapping in terms of its components, the neurons. Specifically, we have

$$f^{(k)}(\mathbf{h}^{(k-1)}) = \left(f_1^{(k)}(\mathbf{h}^{(k-1)}), \dots, f_{q_k}^{(k)}(\mathbf{h}^{(k-1)}) \right)^\top,$$

with each element $f_i^{(k)}(\mathbf{h}^{(k-1)})$ representing a *neuron*, defined as

$$f_i^{(k)}(\mathbf{h}^{(k-1)}) = \phi_k(\langle \mathbf{w}_i^{(k)}, \mathbf{h}^{(k-1)} \rangle + b_i^{(k)}) = \phi_k\left(\sum_{j=0}^{q_{k-1}} w_{i,j}^{(k)} h_j^{(k-1)} + b_i^{(k)} \right).$$

Here, the weight vector $\mathbf{w}_i^{(k)} = (w_{i,j}^{(k)})_{1 \le j \le q_{k-1}} \in \mathbb{R}^{q_{k-1}}$ is the ith row of $\mathbf{W}^{(k)}$. Figure 1.1 shows both a layer representation and a neuron representation of a fully connected network.



(a) Layer-level representation **(b)** Neuron-level representation

Fig. 1.1 Two different representations of a fully connected network with two hidden layers.

We can view $\mathbf{h}^{(k)}$ as a representation of $\mathbf{x}$ after passing through k layers. To clearly show the relation between $\mathbf{h}^{(k)}$ and $\mathbf{x}$, we can write $\mathbf{h}^{(k)} = f^{(k:1)}(\mathbf{x}) \coloneqq \left(f^{(k)} \circ \ldots \circ f^{(1)}\right)(\mathbf{x})$. Therefore, the *output layer* $\hat{\mathbf{y}}$ can be written as $\hat{\mathbf{y}} = \mathbf{h}^{(d)} = f^{(d:1)}(\mathbf{x}) = f^{(d:1)}(\mathbf{x}; \boldsymbol{\theta})$, where the parameter $\boldsymbol{\theta}$ is composed of the network's weights and biases.

Two problems arise when we use feedforward neural networks to model sequential data $(x_t)_{0 \leq t \leq T}$. Firstly, the length of the input varies with time T. This issue is easily solved by assuming some form of the Markov property, i.e., the value of x_t only depends on τ previous values of the sequence, for a fixed number τ. The second issue is that the architecture of feedforward neural networks can not respect temporal causality, since there is no feedback in the model.[4] For these reasons, we introduce recurrent neural networks, networks with feedback, which are designed to deal with sequential data, in Section 1.2.2.

1.2.2 Recurrent Neural Networks

Assume we have sequential input data $(\mathbf{x}_t)_{0 \leq t \leq T}$, which we use to predict $\mathbf{y}_{T+1}$. We can think of $(\mathbf{x}_t)_{0 \leq t \leq T}$ as containing the relevant information for the prediction. As mentioned above, there are some issues with using feedforward neural networks for this type of task. First, the input data varies in size based on T. We could assume a fixed input size by only looking τ time-steps back, i.e., to predict $\mathbf{y}_{T+1}$, we look at $(\mathbf{x}_t)_{T-\tau+1 \leq t \leq T}$. This implicitly assumes a Markov property. However, even with this assumption, feedforward neural networks can typically not respect temporal dependencies, since the network does not recognise that the feature $\mathbf{x}_{t-1}$ has been experienced just before the feature $\mathbf{x}_t$. Moreover, feedforward neural networks treat each input independently, meaning that they do not consider any relationships or dependencies between consecutive inputs.

To deal with the temporal issues of feedforward neural networks, recurrent neural networks, which are specially designed to deal with sequential data, introduce feedback in the network architecture. Most often, the output layer receives information from the past by letting previous *hidden states*, which are the hidden layers at each time point, influence the current hidden states. We say that these networks have recurrent connections between hidden layers. We could also have a network with recurrent connections between the output layer and the hidden layers. Also, we could let the output layer receive information from both the past and the future, such as in a bidirectional recurrent neural network. Networks with recurrent connections between the output layer and the hidden layers or bidirectional recurrent neural networks are discussed in, e.g., [7, ch. 10].

A recurrent neural network processes each input $\mathbf{x}_t$ step-by-step. Parameter sharing allows the model to learn patterns that are independent of position or length. This allows recurrent neural networks to handle varying input sizes. If we instead had a feedforward neural network that processed sequential data of fixed length, the network would have separate parameters for each input feature. This means the network must learn a specific pattern at each input feature separately. As a result, the model would require more training data to learn the same behaviour across positions, and it would struggle to handle sequences of lengths it was not explicitly trained on.

[4] Sometimes, "feedback" refers to looping an output back to influence the same input that produced it. A network with feedback would therefore cause an infinite loop and prevent training. However, in neural networks, feedback typically refers to the output from one input influencing the processing of a different input (e.g., across time steps in recurrent neural networks). Feedforward neural networks do not have this kind of cross-input influence.

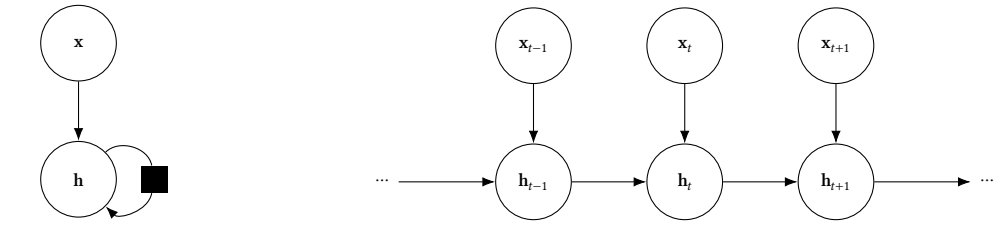
We first present the "plain vanilla" recurrent neural network, with only one hidden layer that has a cycle with itself. The network is illustrated in Figure 1.2, both in its compact form (left), where the black box represents a time delay of one time step, and in its unrolled form (right). The hidden state $\mathbf{h}_t$ is the value of the hidden layer $\mathbf{h}$ at time t. It is a transformation of the previous hidden state $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$, defined by

$$\begin{aligned} f\colon \mathbb{R}^{q_0} \times \mathbb{R}^{q_1} &\to \mathbb{R}^{q_1}, \\ (\mathbf{x}_t, \mathbf{h}_{t-1}) &\mapsto \mathbf{h}_t, \end{aligned}$$

such that

$$f(\mathbf{x}_t, \mathbf{h}_{t-1}) = \phi(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b}), \tag{1.28}$$

for some activation function ϕ applied element-wise, weight matrices $\mathbf{W}$ and $\mathbf{U}$, and bias vector $\mathbf{b}$. As mentioned above, the parameters $(\mathbf{W}, \mathbf{U}, \mathbf{b})$ are shared between time steps. We initialise the hidden state with $\mathbf{h}_0 = \mathbf{0}$.



(a) Compact diagram **(b)** Unrolled recurrent neural network

Fig. 1.2 Two different representations of a plain vanilla recurrent neural network. The black square represents a delay of one time step.

There are multiple ways to construct a deep recurrent neural network. We describe the three different deep recurrent neural networks mentioned in [22, ch. 8.2.2]. The first variant is to have a loop within each hidden layer. This is described by

$$\mathbf{h}_t^{(k)} = f^{(k)}(\mathbf{h}_t^{(k-1)}, \mathbf{h}_{t-1}^{(k)}) = \phi_k(\mathbf{W}^{(k)}\mathbf{h}_t^{(k-1)} + \mathbf{U}^{(k)}\mathbf{h}_{t-1}^{(k)} + \mathbf{b}^{(k)}), \tag{1.29}$$

with $\mathbf{h}_t^{(0)} = \mathbf{x}_t$.

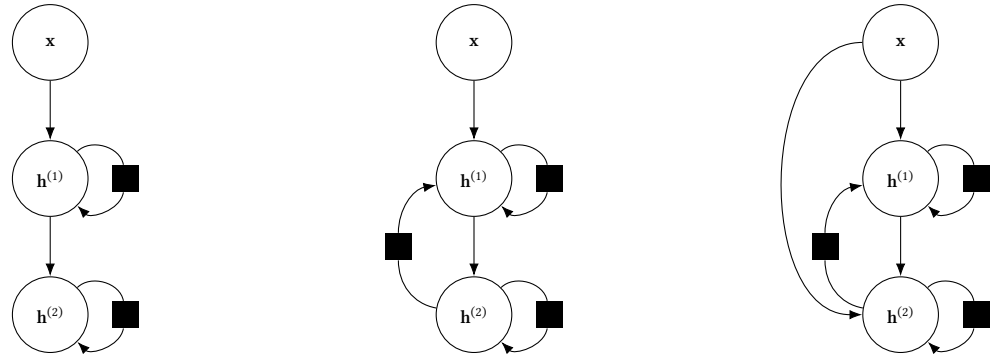
For the second and third variants, we focus on networks with two hidden layers, but the concept can be extended beyond this. The second variant adds feedback from one hidden layer to a previous hidden layer. For two hidden layers, we have

$$\begin{aligned} \mathbf{h}_t^{(1)} &= f^{(1)}(\mathbf{x}_t, \mathbf{h}_{t-1}^{(1)}, \mathbf{h}_{t-1}^{(2)}), \\ \mathbf{h}_t^{(2)} &= f^{(2)}(\mathbf{h}_t^{(1)}, \mathbf{h}_{t-1}^{(2)}). \end{aligned}$$

The third variant builds upon the second variant, adding a skip connection from the input layer to the second hidden layer. We have

$$\begin{aligned} \mathbf{h}_t^{(1)} &= f^{(1)}(\mathbf{x}_t, \mathbf{h}_{t-1}^{(1)}, \mathbf{h}_{t-1}^{(2)}), \\ \mathbf{h}_t^{(2)} &= f^{(2)}(\mathbf{x}_t, \mathbf{h}_t^{(1)}, \mathbf{h}_{t-1}^{(2)}). \end{aligned}$$

The three deep recurrent neural networks are illustrated in Figure 1.3. In the remainder of this thesis, we use the first variant of deep recurrent neural networks.



(a) First variant **(b)** Second variant **(c)** Third variant

Fig. 1.3 Compact diagram of three different deep recurrent neural networks with two hidden layers.

Long Short-Term Memory

To learn long-term dependencies in a recurrent neural network, the same function has to be composed with itself many times. This can lead to the so-called vanishing gradient problem, or less commonly, the exploding gradient problem. Goodfellow *et al.* [7, pp. 396–415] gives a brief description of the vanishing or exploding gradient problem, along with several proposed solutions. One of these solutions was suggested by Hochreiter and Schmidhuber [10], the *long short-term memory* network (LSTM). Since its introduction, the model has been refined and popularised by a wide range of researchers. Today, it is the most common implementation of recurrent neural networks. Figure 1.4 shows a single LSTM cell, which can be connected to other LSTM cells or different types of layers to form a network. The core idea of LSTMs is to have an internal cell state that has a linear self-loop. This allows the information to flow freely from one cell state to the next, creating a path through time that has derivatives that neither vanish nor explode. The weights of the internal cell state are controlled by different *gates*, usually called the *forget gate* and *input gate*. Lastly, the output of the LSTM cell is a filtered version of the internal cell state, regulated by the *output gate*.

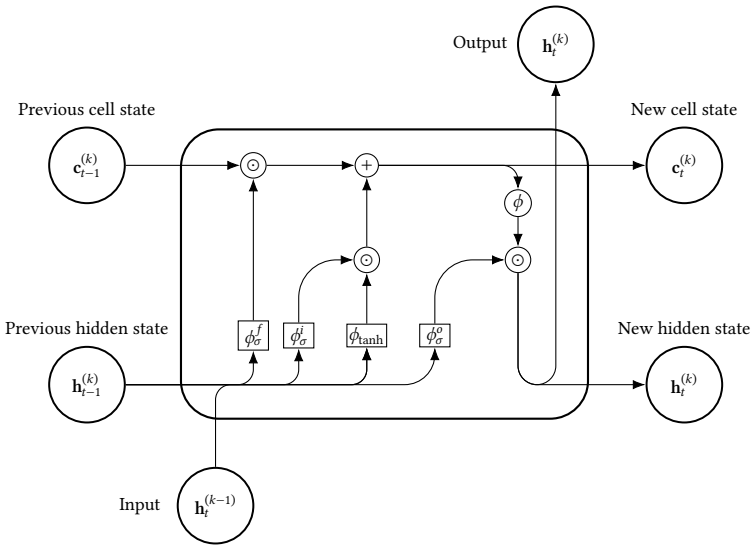


Fig. 1.4 Schematic of an LSTM cell. Rectangular nodes represent transformations akin to (1.29), with weight and bias parameters. Small circular nodes represent element-wise operations.

Before going into the technical details, we first define the *Hadamard product* operation, which is extensively used in the description of an LSTM cell and shows up in Figure 1.4.

Definition 1.2.1 (Hadamard product) Let $\mathbf{X}, \mathbf{Y}$ be two $m \times n$ matrices. The Hadamard product $\mathbf{X} \odot \mathbf{Y}$ is an $m \times n$ matrix with the elements

$$(\mathbf{X} \odot \mathbf{Y})_{ij} = \mathbf{X}_{ij} \cdot \mathbf{Y}_{ij},$$

where $\mathbf{A}_{ij}$ denotes the element on the ith row and jth column of the matrix $\mathbf{A}$. The Hadamard product is undefined for matrices with different dimensions.

The three gates all use the expit activation function, which we will denote by ϕ_σ, and have a similar structure to (1.29). The forget gate $\mathbf{f}_t^{(k)}$ controls how much of the previous cell state $\mathbf{c}_{t-1}^{(k)}$ to keep. The forget gate is defined by

$$\mathbf{f}_t^{(k)} = \phi_\sigma(\mathbf{W}_f^{(k)}\mathbf{h}_t^{(k-1)} + \mathbf{U}_f^{(k)}\mathbf{h}_{t-1}^{(k)} + \mathbf{b}_f^{(k)}),$$

where ϕ_σ is applied element-wise and $\mathbf{h}_t^{(0)} \coloneqq \mathbf{x}_t$.

The input gate $\mathbf{i}_t^{(m)}$ controls how much new information the cell state should be updated with. Similarly to the forget gate, we have

$$\mathbf{i}_t^{(k)} = \phi_\sigma(\mathbf{W}_i^{(k)}\mathbf{h}_t^{(k-1)} + \mathbf{U}_i^{(k)}\mathbf{h}_{t-1}^{(k)} + \mathbf{b}_i^{(k)}).$$

The cell state is thus updated as follows:

$$\mathbf{c}_t^{(k)} = \mathbf{f}_t^{(k)} \odot \mathbf{c}_{t-1}^{(k)} + \mathbf{i}_t^{(k)} \odot \phi_{\tanh}(\mathbf{W}^{(k)}\mathbf{h}_t^{(k-1)} + \mathbf{U}^{(k)}\mathbf{h}_{t-1}^{(k)} + \mathbf{b}^{(k)}), \tag{1.30}$$

where the activation function $\phi_{\tanh}$ is applied element-wise.

Lastly, the output gate $\mathbf{o}_t^{(m)}$ controls how much of the current cell state should be outputted from the LSTM cell. Similarly to the other gates, the output gate is given by

$$\mathbf{o}_t^{(k)} = \phi_\sigma(\mathbf{W}_o^{(k)}\mathbf{h}_t^{(k-1)} + \mathbf{U}_o^{(k)}\mathbf{h}_{t-1}^{(k)} + \mathbf{b}_o^{(k)}).$$

The output gate, together with the current cell state $\mathbf{c}_t^{(k)}$, produce the output of the LSTM-cell $\mathbf{h}_t^{(k)}$, through

$$\mathbf{h}_t^{(k)} = \mathbf{o}_t^{(k)} \odot \phi(\mathbf{c}_t^{(k)}), \tag{1.31}$$

for some activation function ϕ applied element-wise.

Remark 4 The expit activation function is the default for the gates in an LSTM cell in `keras3`. However, this can be modified by setting the `recurrent_activation` argument. Note that changing the activation function alters the behaviour and interpretations of the gates, as their outputs are no longer restricted to the interval $[0, 1]$, the image of the expit function.

Remark 5 Both activation functions ϕ in (1.30) and $\phi_{\tanh}$ in (1.31) are specified in `keras3::layer_lstm` with the argument `activation`, which has the default value `activation="tanh"`. This means that ϕ = and $\phi_{\tanh}$ have to be the same function when using `keras3`.

Output of a Recurrent Neural Network

So far, we have discussed some different RNN structures up to the output. Suppose we want to predict some random variable $\mathbf{Y}_{T+1}$ based on all information up to T (this includes previous values of the random variable $\mathbf{Y}_T$). Specifically, we define a *filtration*, i.e., a sequence of σ-algebras $(\mathscr{F}_t)_{t\geq 0}$, such that $\mathscr{F}_s \subseteq \mathscr{F}_t$, when $s \leq t$, and we want to predict $\mathbf{Y}_{T+1}$ conditioned on $\mathscr{F}_T$. In a regression setting, the best predictor is usually the conditional expectation $f^* = \mathbb{E}[\mathbf{Y}_{T+1} \mid \mathscr{F}_T]$. With a recurrent neural network, we can approximate f^* by putting $\mathbf{h}_T^{(d-1)}$ through an output layer, i.e.,

$$\hat{\mathbf{y}}_{T+1} = f^{(d)}(\mathbf{h}_T^{(d-1)}) = \phi(\mathbf{W}^{(d)}\mathbf{h}_T^{(d-1)} + \mathbf{b}^{(d)}). \tag{1.32}$$

This is illustrated in Figure 1.5.

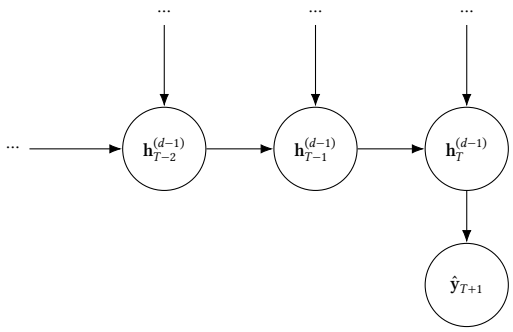


Fig. 1.5 Output of a recurrent neural network.

Note that (1.32) only focuses on the output of the last time step T. Sometimes, we want the output of each time step. This can be done using a *time-distributed layer*, which calculates the output at each time step simultaneously with

$$\hat{\mathbf{y}}_t = f^{(d)}(\mathbf{h}_{t-1}^{(d-1)}) = \phi(\mathbf{W}^{(d)}\mathbf{h}_{t-1}^{(d)} + \mathbf{b}^{(d)}), \tag{1.33}$$

for $0 \leq t \leq T$. The time-distributed layer is illustrated in Figure 1.6.

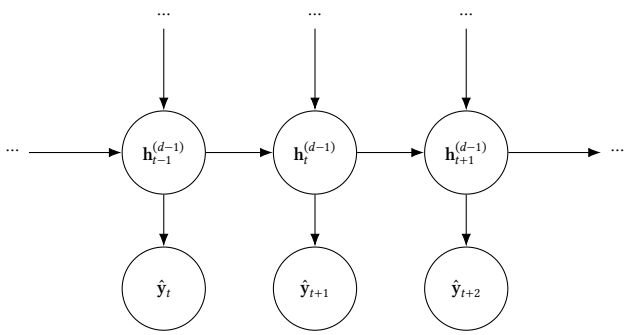


Fig. 1.6 Output of a recurrent neural network using a time-distributed layer.

1.2.3 Fitting the Model

Fitting a neural network involves choosing a loss function and optimising the parameters to minimise (1.22) based on the learning data. Often, the model defines a probability distribution, and a natural choice for a cost function is the deviance, which is proportional to the negative log-likelihood of the learning data under the model. (Minimising the deviance is equivalent to maximising the likelihood.)

In (1.22), a regularisation term is included to punish certain models—for example, L2 regularisation (ridge regression) penalises large weights. More generally, regularisation refers to any modification of the learning algorithm aimed at reducing the generalisation error, even if it increases the training error.

In neural networks, the most common form of regularisation is *early stopping*. Early stopping involves stopping the learning algorithm before the model minimises the cost function of the learning data, to prevent overfitting. In practice, the learning data is split into training data (used to train the parameters) and validation data (used to perform out-of-sample evaluations, to measure generalisation error). The validation error is measured and monitored during model fitting. Increases in validation error indicate overfitting, prompting the learning algorithm to stop.

Most learning algorithms for neural networks are gradient-based, i.e., first-order optimisation algorithms. We use the notation $\nabla_{\boldsymbol{\theta}}\mathscr{L}$ (or sometimes $\nabla_{\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta})$) for the list of the partial derivatives of the loss function $\mathscr{L}$, with respect to all the parameters in $\boldsymbol{\theta}$. For a fully connected network, we have

$$\nabla_{\boldsymbol{\theta}}\mathscr{L} = \left\{\frac{\partial\mathscr{L}}{\partial\mathbf{W}^{(1)}}, \frac{\partial\mathscr{L}}{\partial\mathbf{b}^{(1)}}, \ldots, \frac{\partial\mathscr{L}}{\partial\mathbf{W}^{(d)}}, \frac{\partial\mathscr{L}}{\partial\mathbf{b}^{(d)}}\right\}. \tag{1.34}$$

With $\nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(i)})$, we mean $\nabla_{\boldsymbol{\theta}}\mathscr{L}$ evaluated on the training data, with the parameters $\hat{\boldsymbol{\theta}}^{(i)}$.

To minimise $\mathscr{L}$, we would like to find the direction in which $\mathscr{L}$ decreases the fastest. The directional derivative in direction $\mathbf{u}$, for some unit vector $\mathbf{u}$, tells us the slope of the function $\mathscr{L}$ in the direction of $\mathbf{u}$. We can define it as

$$\nabla_{\mathbf{u}}\mathscr{L}(\boldsymbol{\theta}) \coloneqq \lim_{h\to 0}\frac{\mathscr{L}(\boldsymbol{\theta}+h\mathbf{u}) - \mathscr{L}(\boldsymbol{\theta})}{h}. \tag{1.35}$$

If the function $\mathscr{L}$ is differentiable, this simplifies to

$$\nabla_{\mathbf{u}}\mathscr{L}(\boldsymbol{\theta}) = \mathbf{u}^\top\nabla_{\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta}). \tag{1.36}$$

We can now use the directional derivative (1.36) to find the direction in which $\mathscr{L}$ decreases the fastest. We want to find

$$\arg\min_{\mathbf{u}\,:\,\|\mathbf{u}\|=1} \mathbf{u}^\top\nabla_{\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta}). \tag{1.37}$$

Using the rules of dot products, we can rewrite (1.37) as

$$\arg\min_{\mathbf{u}\,:\,\|\mathbf{u}\|=1} \|\mathbf{u}\|\,\|\nabla_{\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta})\|\cos\nu, \tag{1.38}$$

where $\|\cdot\|$ denotes the L^2 norm and $\nu \in [0, 2\pi)$ is the angle between $\mathbf{u}$ and $\nabla_{\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta})$. Using $\|\mathbf{u}\| = 1$ and ignoring factors that do not depend on $\mathbf{u}$, (1.38) becomes

$$\underset{\mathbf{u}\,:\,\|\mathbf{u}\|=1}{\arg\min} \cos \nu,$$

which minimises when $\nu = \pi$, i.e., when $\mathbf{u}$ points in the opposite direction of $\nabla_{\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta})$. This means that we should move in the opposite direction of the gradient to decrease $\mathscr{L}$ the most, i.e.,

$$\hat{\boldsymbol{\theta}}^{(k+1)} = \hat{\boldsymbol{\theta}}^{(k)} - \delta_{k+1} \cdot \nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(k)}), \tag{1.39}$$

where $\delta_{k+1} > 0$ is the learning rate, controlling how much we move in each iteration $k+1$. This is called the *method of steepest descent*, or *gradient descent*, see, e.g., [7, ch. 4.3].

The standard gradient descent can be extended to *momentum-based gradient descent methods*. These methods keep the "momentum" of gradients of previous iterations in different ways and apply it to a modified iteration step. Three common momentum-based gradient descent methods are `rmsprop` (root mean squared propagation), `adam` (adaptive moment), and `nadam` (Nesterov-accelerated adaptive moment). We show how these methods work, but we do not delve into details on why they work. For further explanation, we refer the interested reader to, e.g., [22, pp. 285–288].

`rmsprop` Originally presented in a lecture slide by Hinton *et al.* [8], this optimiser keeps a moving average of the squared gradients, which is used to adjust the learning rate. The exponentially weighted moving average of the squared gradients (*second moment estimate*) is

$$\mathbf{v}^{(k)} = \beta \mathbf{v}^{(k-1)} + (1-\beta)\left(\nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(k)}) \odot \nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(k)})\right), \tag{1.40}$$

with initial value $\mathbf{v}^{(0)} = \mathbf{0}$ and exponential decay rate $\beta \in (0,1)$. Let $\eta > 0$ be the learning rate and $\epsilon > 0$ a small constant for numerical stability. We update the parameters with

$$\hat{\boldsymbol{\theta}}^{(k+1)} = \hat{\boldsymbol{\theta}}^{(k)} - \frac{\eta}{\sqrt{\epsilon + \mathbf{v}^{(k)}}} \odot \nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(k)}), \tag{1.41}$$

where the operations are applied element-wise.

`adam` First proposed by Kingma and Ba [11], this optimiser combines `rmsprop` with the exponentially weighted moving average of the gradients (*first moment estimate*) defined by

$$\mathbf{r}^{(k)} = \alpha \mathbf{r}^{(k-1)} + (1-\alpha)\nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(k)}), \tag{1.42}$$

with initial value $\mathbf{r}^{(0)} = \mathbf{0}$ and exponential decay rate $\alpha \in (0,1)$. Let $\eta > 0$ be the learning rate and $\epsilon > 0$ a small constant for numerical stability. The gradient descent update is

$$\hat{\boldsymbol{\theta}}^{(k+1)} = \hat{\boldsymbol{\theta}}^{(k)} - \frac{\eta}{\epsilon + \sqrt{\frac{\mathbf{v}^{(k)}}{1-\beta^k}}} \odot \frac{\mathbf{r}^{(k)}}{1-\alpha^k}, \tag{1.43}$$

where the operations are applied element-wise and $\mathbf{v}^{(k)}$ is defined as in (1.40).

`nadam` This optimiser is the so-called Nestorov-accelerated version of `adam`, introduced by Dozat [5]. Introduce the decay factor $\gamma = 0.96$ (this is the default value used in `keras3`). Let the first moment estimate and the second moment estimate be defined as in (1.42) and (1.40), respectively. Define

$$u_k \coloneqq \alpha\left(1 - \frac{\gamma^k}{2}\right).$$

Let $\eta > 0$ be the learning rate and $\epsilon > 0$ a small constant for numerical stability. The gradient descent update is

$$\hat{\boldsymbol{\theta}}^{(k+1)} = \hat{\boldsymbol{\theta}}^{(k)} - \frac{\eta}{\epsilon + \sqrt{\frac{\mathbf{v}^{(k)}}{1-\beta^k}}} \odot \left(\frac{u_{k+1}\mathbf{r}^{(k)}}{1 - \prod_{i=1}^{k+1} u_i} + \frac{(1-u_k)\nabla_{\boldsymbol{\theta}}\mathscr{L}(\hat{\boldsymbol{\theta}}^{(k)})}{1 - \prod_{i=1}^{k} u_i}\right), \tag{1.44}$$

where the operations are applied element-wise.

Backpropagation

In all gradient-based learning algorithms, we have to calculate the gradient. We now describe *backpropagation*, an efficient method to calculate the gradients.

The goal is to calculate the gradient of the loss function with respect to each parameter in the network. First, we have to evaluate the current model. *Forward propagation* is the process by which input data $\mathbf{x}$ passes through the network to generate an output $\hat{\mathbf{y}}$. We can then assess $C(\mathbf{y}, \hat{\mathbf{y}})$. This is usually done over the whole training data, and we calculate (1.22). Once we have evaluated (1.22), we backtrack through the network and calculate the gradient at each hidden layer, recursively using the chain rule of calculus to find the desired gradients. This step is called backpropagation. We show backpropagation for a fully connected network in Algorithm 1, which is taken from [7] with slight modifications. The function returns the gradients with respect to each parameter evaluated in the current model.

Algorithm 1 Backpropagation

Require: $\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}, d$
$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} L(\mathbf{y}, \hat{\mathbf{y}})$
for $k = d, d-1, \dots, 1$ **do**
  $\mathbf{g} \leftarrow \mathbf{g} \odot \phi'(\mathbf{a}^{(k)})$ ▷ *Calculate* $\nabla_{\mathbf{a}^{(k)}} L(\hat{\mathbf{y}}, \mathbf{y})$
  ▷ *Using* $\mathbf{g}$*, we can calculate the gradients with respect to* $\mathbf{b}^{(k)}$ *and* $\mathbf{W}^{(k)}$*.* ◁
  $\nabla_{\mathbf{b}^{(k)}} L(\hat{\mathbf{y}}, \mathbf{y}) \leftarrow \mathbf{g}$
  $\nabla_{\mathbf{W}^{(k)}} L(\hat{\mathbf{y}}, \mathbf{y}) \leftarrow \mathbf{h}^{(k-1)} \mathbf{g}^\top$
  $\mathbf{g} \leftarrow \mathbf{W}^{(k)\top} \mathbf{g}$ ▷ *Calculate* $\nabla_{\mathbf{h}^{(k-1)}} L(\hat{\mathbf{y}}, \mathbf{y})$
end for
return $\nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}})$

For a recurrent neural network, we use *backpropagation through time* (BPTT). BPTT is similar to standard backpropagation. We first forward propagate the inputs through the unfolded recurrent neural network (see, e.g., Figure 1.2b), then we backpropagate the error across the unfolded network. For long sequences, unfolding the whole network is memory-intensive, and in practice, we mostly use a truncated version of BPTT, called truncated BPTT, or TBPTT. For the TBPTT, we choose a lookback period τ. Then, we unfold the network so that each output is produced using τ inputs. With the default options in the `keras3` package, the hidden states are reset to $\mathbf{0}$ for each output. However, this can be changed by setting `stateful=TRUE`. The TBPTT for both options is outlined in Algorithm 2.

Algorithm 2 Truncated backpropagation through time

Require: $\tau, \boldsymbol{\theta}, \mathbf{x}, \mathbf{y}$
for $t = 1, \dots, T - \tau + 1$ **do**
  Unfold the network to contain τ time-steps
  `input` $\leftarrow \mathbf{x}_t, \dots, \mathbf{x}_{t+\tau-1}$
  $\hat{\mathbf{y}}_{t+\tau} \leftarrow$ forward propagate `input`
  Calculate $L(\mathbf{y}_{t+\tau}, \hat{\mathbf{y}}_{t+\tau})$
  Backpropagate $L(\mathbf{y}_{t+\tau}, \hat{\mathbf{y}}_{t+\tau})$ across whole unfolded network
  Update the parameters
  if `!stateful` **then**
    ▷ *By default,* `stateful=FALSE` *and we reset all hidden states. Otherwise, we let the hidden states remain to the next* t ◁
    Set all hidden states to $\mathbf{0}$
  end if
end for

Stochastic Gradient Descent

If our training data has a large sample size n, it is unrealistic to evaluate the cost function and calculate all the gradients based on the whole training data, since this would be too slow. Therefore, we typically use *stochastic gradient descent* (SGD), which does not use the entire training data simultaneously. Instead, we partition the data into *mini batches* of fixed batch size $b \in \mathbb{Z}^+$. For each gradient descent update, we only use one mini-batch.

Typically, we sequentially use all mini-batches. Screening each mini batch once is called an *epoch*. Therefore, running the SGD algorithm for K epochs means performing $K\lfloor n/b \rfloor$ gradient descent updates.

We want to choose the batch size b so that it is small enough for quick gradient descent calculations, but not too small. A too small value of b causes erratic gradient descent steps, i.e., the randomness in a small sample can cause us to take a step in a completely different direction than what is optimal. Another thing to consider is that some hardware runs better with specific batch sizes, typically powers of 2.

One last remark is that small batch sizes also have a regularising effect, perhaps due to the noise they add to the training. Sometimes, the learning algorithm could get stuck at saddle points or flat areas of the objective function. An erratic step could be beneficial because it can perturb the algorithm out of the bottleneck. To compensate for the smaller batch size, we may need a smaller learning rate to stabilise the training. This could result in longer runtimes since we would have to take more gradient descent steps due to both the erratic nature of the small batches and the small learning rate.

Chapter 2

Numerical Application

In this chapter, we apply the theory of the previous sections to real data. We used Swedish mortality data sourced from the Human Mortality Database (HMD) [9].

We first fit the Lee–Carter model to the data. After estimating the parameters in the Lee–Carter model, we explore the use of neural networks and deep learning models to extrapolate the time index κ_t. The neural networks are compared to traditional methods using ARIMA models. In particular, we use a random walk with drift as our baseline model, but we also compare the performance of letting the function `auto.arima` choose an appropriate ARIMA model. The goal of this section is to illustrate how we can incorporate deep learning into the Lee–Carter model and to compare the different methods of extrapolation. All the code is written in R. We use the `StMoMo` package to fit the Lee–Carter models and forecast using the traditional methods. For the neural networks, we use the package `keras3` with `tensorflow` as the back end.

Similar studies have been conducted in [22, 19, 14, 13]. Wütrich and Merz [22, ch. 8.4] applied an LSTM on $e_t := \kappa_t - \kappa_{t-1}$ of the Lee–Carter model across multiple countries simultaneously and also explored modelling mortality directly using an LSTM. Richman and Wütrich [19] compared gated recurrent units (GRUs) and LSTMs applied directly to death rates. The neural networks were also compared to standard Lee–Carter models. Lindholm and Palmborg [14] explored different strategies for partitioning and efficiently using the learning data in mortality forecasting. Levenius [13, ch. 2.2] used the same Swedish mortality data to study Cramér–Wold's method for parameter estimation in the Gompertz–Makeham model.

2.1 Swedish Population Data

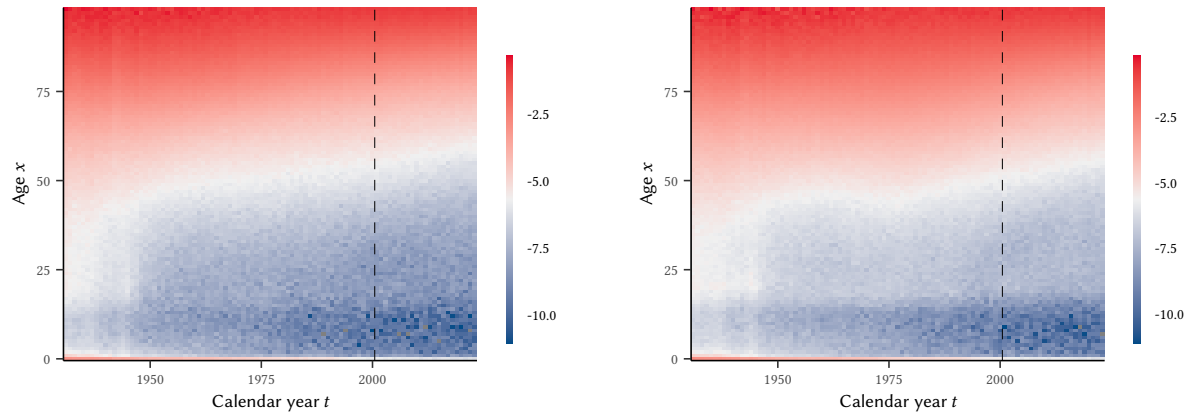
We have data on central death rates $m_{x,t}$ and exposures to risk $r_{x,t}$ for the years 1751–2023 and the ages 0–110+. Listing 2.1 provides code for reading and converting data from HMD to `StMoMoData`, the data format used by most functions of the package `StMoMo`.

Listing 2.1 Creating `StMoMoData`.

```
sweden <- read.demogdata(
  "Mx_1x1.txt",
  "Exposures_1x1.txt",
  type = "mortality",
  label = "Sweden"
)
sweF <- StMoMoData(sweden, series = "female")
sweM <- StMoMoData(sweden, series = "male")
```

We limit ourselves to studying the years 1931–2023 and ages 0–98 (we choose 98 as our upper limit because of data scarcity, but also because at higher ages, the mortality behaves strangely). Figure 2.1 illustrates the log-central death rates $\log(m_{x,t})$ of the Swedish population, separated by gender. More red colours indicate higher mortality, whereas bluer colours indicate lower mortality. We can generally see a slight diagonal structure in the data, indicating the typical mortality improvements that we have seen throughout the years. We also see that females typically have lower mortality than males. The black vertical dashed lines in Figure 2.1 indicates where we have split the data into learning data $\mathcal{T}_\diamond$ (calendar years $t \in \{1931, \dots, 2000\}$) and test data $\mathcal{T}_\star$ (calendar years $t \in \{2001, \dots, 2023\}$).[1]

[1] We use a diamond ($\diamond$) to represent the data used for *polishing* the models, and a star ($\star$) to denote the data on which the trained models *shine* (hopefully). Thanks to Leo Levenius for this suggestion.



(a) Females **(b)** Males

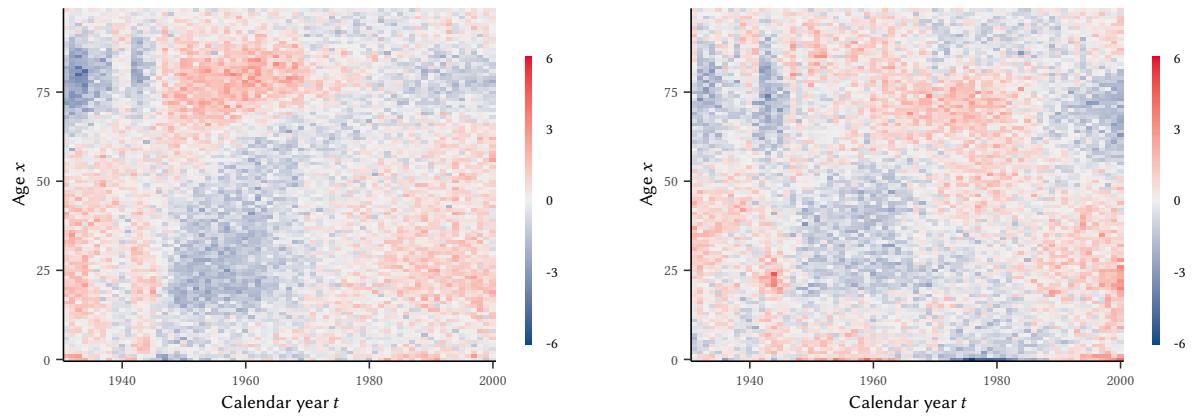
Fig. 2.1 Log-central death rates $\log(m_{x,t})$. Grey colours represent NA values.

2.2 Lee–Carter

We fit the Lee–Carter model on each gender separately. For ease of notation, we do not write hats above the fitted parameters of the Lee–Carter model. Listing 2.2 provides the code to fit a model for Swedish female data `sweF` and plot the residuals in a heatmap. The resulting residual plots for both genders are shown in Figure 2.2. We can see that the residuals have clear patterns, indicating that the Lee–Carter model may not be sufficient for the data. We could consider more complex models, such as the Renshaw–Haberman extension of the Lee–Carter model, proposed by Renshaw and Haberman [18], which adds a cohort term to (1.2). However, it is not our intent to find the best model for the Swedish population, but rather to illustrate how we can implement deep learning models on mortality studies.

Listing 2.2 Fitting the Lee–Carter model in `StMoMo` and plotting residuals.

```
library(StMoMo)
LC <- lc(link = "log")
fitYrs <- 1931:2000
fitAges <- 0:98
LCfitsweF <- StMoMo::fit(LC, data = sweF, ages.fit = fitAges, years.fit = fitYrs)
LCressweF <- residuals(LCfitsweF)
plot(LCressweF, type = "colourmap")
```



(a) Females **(b)** Males

Fig. 2.2 Heatmap of the residuals of the fitted Lee–Carter model.

2.3 Forecasting

We model the time index κ_t of the Lee–Carter model using both ARIMA models and three different neural networks: a feedforward neural network, a shallow LSTM, and a deep LSTM.[2] In Appendix B.1, we provide code and summaries for the three neural network models.

Due to the potential drift in the κ_t-process, it is often easier to model the increments $e_t := \kappa_t - \kappa_{t-1}$ instead of κ_t in the neural networks—see, e.g., [22, p. 397]. Let τ be the lookback period. The sequential input used to predict $y_{t+1} = e_{t+1}$ is

$$\mathbf{x}_{t-\tau+1:t} = (e_{t-\tau+1}, \dots, e_t)^\top. \tag{2.1}$$

For the rest of the thesis, we use the shorter notation $\mathbf{x}_t = \mathbf{x}_{t-\tau+1:t}$, i.e., $\mathbf{x}_t$ has the target y_{t+1}.

For the recurrent neural networks, we use the truncated back propagation through time, so each element of $\mathbf{x}_t$ is input one by one. In `keras3`, the input dimension is `c(`τ`, 1)`, meaning that each time-step is one-dimensional. In the feed-forward neural network, we feed the network the flattened version of $\mathbf{x}_t$, which in the case of (2.1), is just itself. Therefore, the input dimension for the feed-forward neural network in `keras3` is `c(`$\tau \cdot 1$`)`.

We choose to model both genders simultaneously in the neural networks. Therefore, we must add an index for each gender g as a covariate. We have 1 for females (f) and 0 for males (m). Our learning data, therefore, is

$$\left(\left(\mathbf{x}_{t+\tau}^{(g)}, \mathbb{1}_{\{g=\text{f}\}}\right), y_{t+\tau+1}^{(g)}\right)_{g\in\{\text{f,m}\}, t_0\le t\le t_1-\tau-1}, \tag{2.2}$$

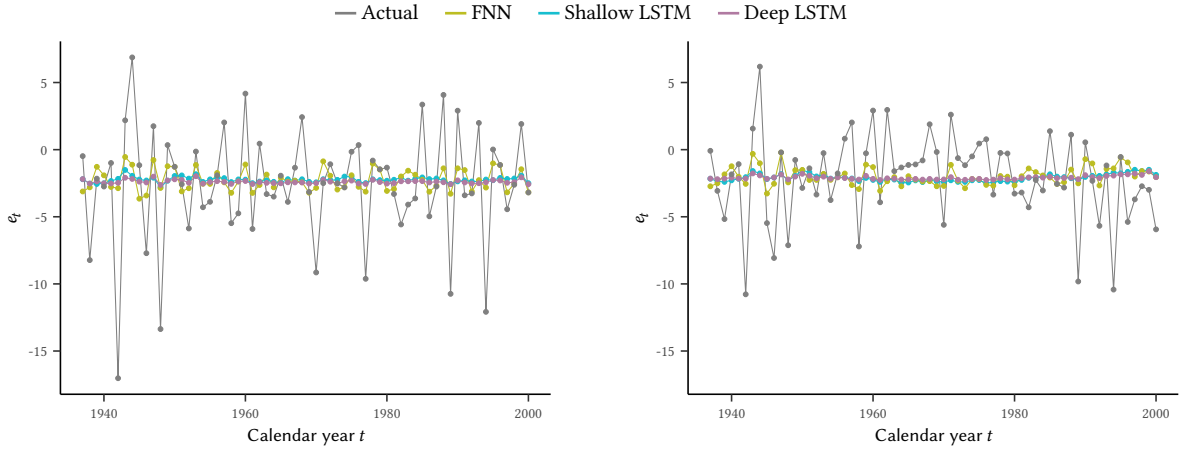
where t_0 and t_1 is the first and last calendar year of $\mathscr{T}_\diamond$, respectively.

We use the first 85 % (rounded down) of the learning data as training data and the rest as validation data. Lindholm and Palmborg [14] call this "Calibration LO". We use a batch size of 2 for the stochastic gradient descent. Since the training process of a neural network has several factors of randomness, such as the random initialisation of the parameters and stochastic gradient descent, we use *network aggregating* (or *nagging*). We use the method described in [14, pp. 760–761]. We perform $m = 20$ different calibrations of the parameters $\boldsymbol{\theta}$ using random initialisation and early stopping. Denote each calibration with $\hat{\boldsymbol{\theta}}^{(i)}$. The ensemble model output is

$$\bar{f}(\mathbf{x}, \hat{\boldsymbol{\theta}}^{(1:m)}) := \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{x}, \hat{\boldsymbol{\theta}}^{(i)}). \tag{2.3}$$

For a more detailed explanation of ensemble learning, see, e.g., [22, ch. 7.4.4].

The in-sample results of the neural networks are shown in Figure 2.3. Firstly, we observe a zig-zag pattern in the actual data, implying negative correlation. This pattern seems to be captured best by the FNN. Secondly, we observe that all three models are less volatile than the actual values, especially the LSTMs.



(a) Females **(b)** Males

Fig. 2.3 In-sample increments e_t of the different neural networks compared to actual values.

To evaluate the neural networks out-of-sample, we follow a similar procedure as in [14, pp. 761–763]. First, we estimate the in-sample variance with

$$s^2 = \frac{1}{|\mathscr{T}_\diamond| - \tau}\sum_{t=t_0+\tau}^{t_1-1}\left(e_{t+1} - \bar{f}(\mathbf{x}_t; \hat{\boldsymbol{\theta}}^{(1:m)})\right)^2. \tag{2.4}$$

[2] We use shallow and deep to refer to the number of LSTM cells in each network, where the shallow LSTM has one LSTM cell and the deep LSTM has three.

Next, we recursively predict one time step ahead with

$$\tilde{e}_t^{(k)} = \begin{cases} \bar{f}(\tilde{\mathbf{x}}_{t-1}^{(k)}; \hat{\boldsymbol{\theta}}^{(1:m)}) + \varepsilon_t^{(k)}, & \text{if } t \in \{2001, \dots, 2023\} \\ e_t, & \text{if } t \in \{1932, \dots, 2000\} \end{cases} \tag{2.5}$$

where $\varepsilon_t^{(k)}$ is simulated using `rnorm(1, 0, s^2)`, and

$$\tilde{\mathbf{x}}_t^{(k)} = (\tilde{e}_{t-\tau+1}^{(k)}, \dots, \tilde{e}_t^{(k)})^\top.$$

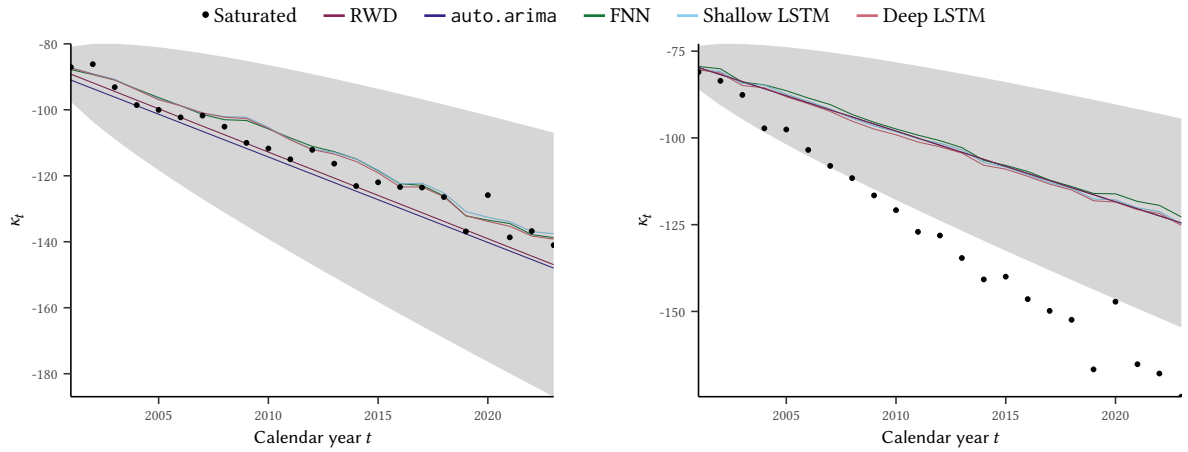
We create 20 trajectories of (2.5) and take the median at each time step to get our final predictions $\tilde{e}_t$. The trajectories and final predictions for each model are shown in Appendix B.2. Note that we have used the same random seed for each model (but different seeds for each gender) to make them more comparable.

To obtain predictions for κ_t, for $t \in \mathcal{T}_\star$, from the neural networks we take the cumulative sum (`cumsum`) of $(\kappa_{2000} + \tilde{e}_{2001}, \tilde{e}_{2002}, \dots, \tilde{e}_{2023})$.

For the ARIMA models, `StMoMo` implements the generic function `forecast`. By default, the function assumes a (multivariate) random walk with drift, as in (1.14). However, we can pre-specify an ARIMA model or let the function `auto.arima` from the package `forecast` automatically select the "best" ARIMA model for κ_t, by setting the argument `method='iarima'`. In our data, `auto.arima` selects an ARIMA(0, 1, 1) model for the female population and a random walk with drift for the male population (same as default).

We also create a baseline out-of-sample model by fixing α_x and β_x, for $x \in \mathcal{X}$, and estimating κ_t in the test data. This is done using Newton–Raphson, similar to the process described in Section 1.1.2. We denote this model by "saturated".

Figure 2.4 compares the different models. The figure also shows the 95 % confidence interval of the RWD model. We observe that the three neural networks produce similar results. This is likely due to the same random seed being used in (2.5). We also see that the saturated model produces significantly different κ_t values compared to any other model for males. This could indicate that the learning data is not representative of the test data for Swedish males, a common problem for extrapolative models.



(a) Females **(b)** Males

Fig. 2.4 Out-of-sample predictions of κ_t for different methods. The grey zones are 95 % confidence intervals for the RWD models.

We close this section by comparing three different metrics of the different models:

- The mean squared error of κ_t, defined as

$$MSE_\kappa(\hat{\boldsymbol{\kappa}}) := \frac{1}{|\mathcal{T}_\star|} \sum_{t \in \mathcal{T}_\star} (\tilde{\kappa}_t - \hat{\kappa}_t)^2, \tag{2.6}$$

  where $\tilde{\kappa}_t$ are from the saturated model.

- The Poisson log-likelihood (*cf.* (1.12)), defined as

$$\ell_{\text{Pois}}(\hat{\boldsymbol{\kappa}}) := \sum_{x \in \mathcal{X}} \sum_{t \in \mathcal{T}_\star} \left(d_{x,t}(\alpha_x + \beta_x \hat{\kappa}_t) - r_{x,t} \exp\{\alpha_x + \beta_x \hat{\kappa}_t\} \right). \tag{2.7}$$

- The mean squared error of the log-central death rates, defined as

$$MSE_{\log(m)}(\hat{\boldsymbol{\kappa}}) := \frac{1}{|\mathscr{X}| \cdot |\mathscr{T}_\star|} \sum_{x \in \mathscr{X}} \sum_{t \in \mathscr{T}_\star} \left(\log(m_{x,t}) - \log(\hat{m}_{x,t})\right)^2, \tag{2.8}$$

  where $\log(\hat{m}_{x,t}) := \alpha_x + \beta_x \hat{\kappa}_t$.

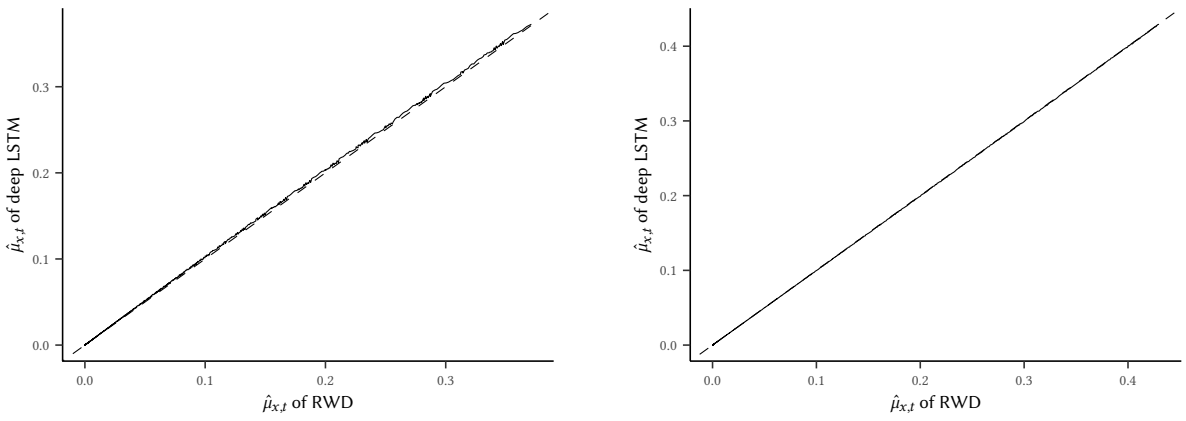
The results are summarised in Table 2.1. We observe that for both genders, the best-performing model, according to all three metrics, is a neural network. In particular, we see that the deep LSTM has the lowest value in four of the six columns. Note that the Poisson log-likelihood accounts for the exposure-to-risk $r_{x,t}$, meaning that larger populations (e.g., common age-calendar year combinations) exert a greater influence on the metric. Both mean squared errors do not take this into account, meaning that each age-calendar year combination contributes to the metric equally. This might explain the abnormally high $MSE_{\log(m)}$ value of the saturated model for men.

Table 2.1 Out-of-sample results and comparison between models and genders. The best results are coloured (for all metrics, lower is better).[a]

	Female			Male		
Method	MSE_κ	$-\ell_{\text{Pois}}$	$MSE_{\log(m)}$	MSE_κ	$-\ell_{\text{Pois}}$	$MSE_{\log(m)}$
Saturated	-	4177074	48.27666	-	4268134	46.37249
RWD	23.61284	4177416	48.81886	895.1521	4276868	42.08151
`auto.arima`	34.46983	4177574	49.04048	895.1521	4276868	42.08151
FNN	17.05528	4177325	47.91732	953.2425	4277466	41.92252
Shallow LSTM	18.18283	4177341	47.86107	892.6476	4276847	42.07866
Deep LSTM	16.01727	4177310	47.95765	859.3456	4276509	42.18155

[a] For the metrics MSE_κ and $-\ell_{\text{Pois}}$, we do not include the saturated model in the comparisons.

Since the deep LSTM seems to be the best performing model, we compare the predicted force of mortality $\hat{\mu}_{x,t}$ between the deep LSTM and the RWD method in Figure 2.5. We see that for females, the deep LSTM tends to predict higher values than the RWD, especially for higher forces of mortality. The difference between the two methods is barely noticeable for males in Figure 2.5b. The results in Figure 2.5 reinforce what we see in Figure 2.4.



(a) Females **(b)** Males

Fig. 2.5 Comparing predicted force of mortality $\hat{\mu}_{x,t}$ between the deep LSTM and the RWD.

In Appendix B.3, we compare the models' predicted log-forces of mortality with the observed log-forces of mortality (see Figures B.2 and B.3). We also compare the models' predicted number of deaths to the actual number of deaths (see Figure B.4). From the latter comparison, we observe that all models perform poorly at $x = 30$, tending to underestimate the actual number of deaths. All models also perform poorly for males aged 60, overshooting the actual number of deaths. In contrast, all models perform well for $x = 90$.

Based on all the comparisons presented in this section and in Appendix B.3, we conclude that all models perform similarly. This is likely due to the underlying Lee–Carter model, which restricts us to modelling the

time index κ_t. Furthermore, we have relatively few observations compared to the number of parameters in the neural networks. This could be addressed by incorporating data from several countries and modelling them simultaneously, similarly to our approach of modelling both genders together.

Alternatively, we could consider modelling the forces of mortality directly, as in [19]. Within our current framework, we could also explore further tuning of the hyperparameters in the neural networks. Another improvement would be handling the NA values in the data—for example, replacing them with the average across all countries in HMD [9].

Sammanfattning (Abstract in Swedish)

Under de senaste åren har ett flertal forskningsartiklar undersökt tillämpningen av djupinlärning inom försäkringsmatematik. Syftet med denna uppsats är att ge en översikt av en sådan tillämpning: användningen av djupinlärningsmodeller för dödlighetsprognoser. Utgångspunkten är Lee–Carter-modellen där vi undersöker hur neuronnät kan användas för att extrapolera tidsindexet κ_t. Vi beskriver parameterskattningen i Lee–Carter-modellen och redogör för traditionella prognosmetoder, där κ_t typiskt modelleras med en autoregressiv integrerad glidande medelvärdesmodell (ARIMA-modell)—ofta en slumpvandring med drift. Därefter presenterar vi en översikt av artificiella neuronnät, med fokus på rekurrenta neuronnät och deras mest använda variant, långa korttidsminnesnätverk (LSTM-nätverk). Metoderna tillämpas på svensk dödsfallsstatistik. Vi jämför framåtriktade neuronnät, grunda LSTM-nätverk och djupa LSTM-nätverk med traditionell ARIMA-baserad prognostisering. Ensembleinlärning används för att minska den slumpmässighet som är förknippad med träning av neuronnät. Våra resultat visar att neuronnät generellt sett presterar bättre än traditionella metoder för det givna datamaterialet.

Bibliography

1. Andersson PH, Lindholm M (2021) Mortality forecasting using a Lexis-based state-space model. Annals of Actuarial Science 15(3):519–548. https://doi.org/10.1017/S1748499520000275

2. Box GEP, Jenkins GM, Reinsel GC, et al (2015) Time Series Analysis: Forecasting and Control. John Wiley & Sons, Inc., Hoboken, NJ

3. Brouhns N, Denuit M, Vermunt JK (2002) A Poisson log-bilinear regression approach to the construction of projected lifetables. Insurance: Mathematics and Economics 31(3):373–393. https://doi.org/10.1016/S0167-6687(02)00185-3

4. Cairns AJG, Blake D, Dowd K (2006) A two-factor model for stochastic mortality with parameter uncertainty: Theory and calibration. Journal of Risk and Insurance 73(4):687–718. https://doi.org/10.1111/j.1539-6975.2006.00195.x

5. Dozat T (2015) Incorporating Nesterov momentum into Adam. Technical report, Stanford University. https://cs229.stanford.edu/proj2015/054_report.pdf. Accessed 30 May 2025

6. Egusquiza Castillo L (2025) Boosting regression models with neural networks, because we CANN. Master's thesis, Stockholm University

7. Goodfellow I, Bengio Y, Courville A (2016) Deep Learning. MIT Press, Cambridge, MA, https://www.deeplearningbook.org/

8. Hinton G, Srivastava N, Swersky K (2012) Neural networks for machine learning. Lecture slides, University of Toronto. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Accessed 30 May 2025

9. HMD (2025) Human mortality database. Max Planck Institute for Demographic Research (Germany), University of California, Berkeley (USA), and French Institute for Demographic Studies (France). https://mortality.org. Data downloaded on 2025-05-15

10. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Computation 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

11. Kingma D, Ba J (2014) Adam: A method for stochastic optimization. Published as a conference paper at ICLR 2015. https://arxiv.org/abs/1412.6980. Accessed 30 May 2025

12. Lee RD, Carter LR (1992) Modeling and forecasting U. S. mortality. Journal of the American Statistical Association 87(419):659–671. https://doi.org/10.2307/2290201

13. Levenius LG (2025) Den Cramérska assuransmatematiken. Master's thesis, Stockholm University

14. Lindholm M, Palmborg L (2022) Efficient use of data for LSTM mortality forecasting. European Actuarial Journal 12:749–778. https://doi.org/10.1007/s13385-022-00307-3

15. Lindskog F (2024) Non-life pricing essentials. Lecture notes, Stockholm University. https://kurser.math.su.se/pluginfile.php/196964/mod_resource/content/9/Pricing_essentials.pdf. Accessed 30 May 2025

16. Ljung GM, Box GEP (1978) On a measure of lack of fit in time series models. Biometrika 65(2):297–303. https://doi.org/10.1093/biomet/65.2.297

17. NIST/SEMATECH (2012) e-handbook of statistical methods. Engineering Statistics Handbook. https://www.itl.nist.gov/div898/handbook/. Accessed 14 March 2025

18. Renshaw AE, Haberman S (2006) A cohort-based extension to the Lee-Carter model for mortality reduction factors. Insurance: Mathematics and Economics 38(3):556–570. https://doi.org/10.1016/j.insmatheco.2005.12.001

19. Richman R, Wüthrich MV (2019) Lee and Carter go machine learning: Recurrent neural networks. Prepared for: Fachgruppe "Data Science" Swiss Association of Actuaries SAV. https://papers.ssrn.com/abstract=3441030. Accessed 30 May 2025

20. Villegas AM, Kaishev VK, Millossovich P (2018) StMoMo: An R package for stochastic mortality modeling. Journal of Statistical Software 84(3):1–38. https://doi.org/10.18637/jss.v084.i03

21. Wikipedia (2024) King Momo. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/King_Momo. Accessed 12 March 2025

22. Wüthrich MV, Merz M (2022) Statistical Foundations of Actuarial Learning and its Applications. Springer, Cham, https://doi.org/10.1007/978-3-031-12409-9

Appendix A

ARIMA models and Box–Jenkins Method

We give a brief overview of ARIMA models and the Box–Jenkins method. Box *et al.* [2] provide a more detailed explanation. Let $(Z_t) \coloneqq (Z_1, \dots, Z_T)$ be a discrete time series, which we can think of as a realisation of a stochastic process.

Definition A.0.1 (Stationary processes) If the probability distribution of Z_t is the same for all times $t \in \{1, \dots, T\}$, we have a *stationary process.* In particular, this implies a constant mean and variance

$$\mathbb{E}[Z_t] = \mu,$$
$$\mathrm{Var}(Z_t) = \sigma_Z^2.$$

In a stationary process, we can estimate the mean μ with the sample mean of the time series

$$\hat{\mu} = \frac{1}{T}\sum_{t=1}^{T} Z_t,$$

and the variance σ_Z^2 with the sample variance of the time series

$$\hat{\sigma}_Z^2 = \frac{1}{N}\sum_{t=1}^{N}(Z_t - \hat{\mu})^2.$$

The stationarity also implies that all joint probability distributions of Z_{t_1} and Z_{t_2}, for all times t_1 and t_2, are the same. Therefore, the autocovariances and autocorrelations depend only on the lag h, i.e.,

$$\mathrm{Cov}(Z_t, Z_{t+h}) = \gamma_h,$$

and

$$\mathrm{Corr}(Z_t, Z_{t+h}) = \rho_h = \frac{\gamma_h}{\sigma_Z^2}.$$

The autocovariance can be estimated with the sample autocovariance of the time series

$$\hat{\gamma}_h = \frac{1}{T}\sum_{t=1}^{T-h}(Z_t - \hat{\mu})(Z_{t+h} - \hat{\mu}),$$

and the autocorrelation with the sample autocorrelation of the time series

$$\hat{\rho}_h = \frac{\hat{\gamma}_h}{\hat{\sigma}_Z^2}.$$

Definition A.0.2 (Autocorrelation function) The autocorrelation function $\mathrm{ACF}(h)$ of a stochastic process returns the autocorrelation ρ_h at lag h, i.e.,

$$\mathrm{ACF}(h) = \rho_h.$$

We can estimate the ACF with

$$\widehat{\mathrm{ACF}}(h) = \hat{\rho}_h,$$

where $\hat{\rho}_h$ is the sample autocorrelation of the time series.

Stationary processes are easy to work with. However, the time series (Z_t) is not always stationary. We introduce the autoregressive integrated moving average (ARIMA) model for non-stationary time series. The ARIMA model is defined by

$$\left(1 - \sum_{k=1}^{p} \phi_k B^k\right) \nabla^d Z_t = \delta + \left(1 - \sum_{k=1}^{q} \theta_k B^k\right) \varepsilon_t, \tag{A.1}$$

where:

- B is the *backward shift operator*, such that $B^k z_t = z_{t-k}$,
- $\nabla^d := (1 - B)^d$ represents the dth-order differencing operator,
- ϕ_k, for $k \in \{1, \dots, p\}$, are the auto-regressive (AR) coefficients,
- θ_k, for $k \in \{1, \dots, q\}$, are the moving average (MA) coefficients,
- δ is the drift term.

Definition A.0.3 (Stationarity and invertibility of ARIMA) The time series $(\nabla^d Z_t)$ in the ARIMA model (A.1) is *stationary*, if all the roots of the equation

$$1 - \sum_{k=1}^{p} \phi_k B^k = 0, \tag{A.2}$$

lie outside the unit circle. Similarly, $(\nabla^d Z_t)$ is *invertible* if all the roots of the equation

$$1 - \sum_{k=1}^{q} \theta_k B^k = 0, \tag{A.3}$$

lie outside the unit circle.

The Box–Jenkins method (see, e.g., [2, pp. 177–392]) applies ARIMA models to find the best fit of a time series. The method has three steps: model identification, parameter estimation, and model diagnostics and forecasting (in some literature, the third step of model diagnostics and forecasting is divided into two steps). The first step is *model identification*. The goal in this step is to find *possible models* by analysing the estimated ACF of the time series.

We also analyse the estimated partial autocorrelation function PACF(h), a function that returns the partial autocorrelations at lag h. The partial autocorrelation at lag h, denoted $\phi_{h,h}$, are obtained by solving the Yule–Walker equations

$$\begin{pmatrix} 1 & \rho_1 & \rho_2 & \cdots & \rho_{h-1} \\ \rho_1 & 1 & \rho_1 & \cdots & \rho_{h-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \rho_{h-1} & \rho_{h-2} & \rho_{h-3} & \cdots & 1 \end{pmatrix} \begin{pmatrix} \phi_{k,1} \\ \phi_{k,2} \\ \vdots \\ \phi_{k,k} \end{pmatrix} = \begin{pmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_k \end{pmatrix}.$$

We obtain an estimate $\hat{\phi}_{h,h}$ of the partial autocorrelation $\phi_{h,h}$ by substituting the autocorrelations ρ_h with the sample autocorrelations $\hat{\rho}_h$. Thus, the estimated PACF is

$$\widehat{\mathrm{PACF}}(h) = \hat{\phi}_{h,h}.$$

We first assess whether the time series (Z_t) is stationary or not by looking at the estimated ACF. The estimated ACF should die out quickly; if it does not, we need to difference (Z_t) until it does. This determines a suitable degree of differencing d.

Once (Z_t) is stationary, we look at the estimated ACF and PACF to find possible models. We want to determine the degree p of the AR component and the degree q of the MA component. In [2], the authors do this by comparing the estimated ACF and PACF to the theoretical ones of known stationary processes. Table A.1, taken from [17], summarises how we can use the estimated ACF for model identification. The behaviour of both the ACF and PACF for the most common ARIMA models is presented in Table 6.1 in [2, p. 184], which also provides preliminary estimates of the AR- and MA coefficients. Figure A.1 shows the theoretical behaviour corresponding to the entries in Table A.1. In practice, the estimated ACF will inevitably contain noise, making the first step of model identification especially challenging.

Another common method to identify the stationary process is to determine p by looking at how many lags of the estimated PACF "stick out". We can determine q analogously by looking at the estimated ACF.

The model identification step is subjective, and often we end up with several possible models. Lastly, once we have potential models, we can make preliminary estimates of the parameters to act as starting values for

the next step. As mentioned above, Table 6.1 in [2, p. 184] shows initial parameter estimates of some common models.

The second step is *parameter estimation* for the possible models identified in the first step This is typically done using maximum likelihood estimation or non-linear least squares estimation, the latter being less common. In this step, we can also compare the possible models in different measures, such as Akaike's information criterion (AIC), to choose a "final" model.

Once we have chosen a model from the previous step, we perform *model diagnostics and forecasting*. The residuals should be independent of each other and have a constant mean and variance over time, i.e., we seek to verify that the residuals are white noise. We can do this graphically or perform a statistical test such as the Ljung–Box test (see, e.g., [16]). We can also check that our model is stationary and invertible by checking the roots of (A.2) and (A.3), respectively. If all conditions are met, we can proceed with the forecasting. If not, we must return to our previous steps and attempt to build a better model.

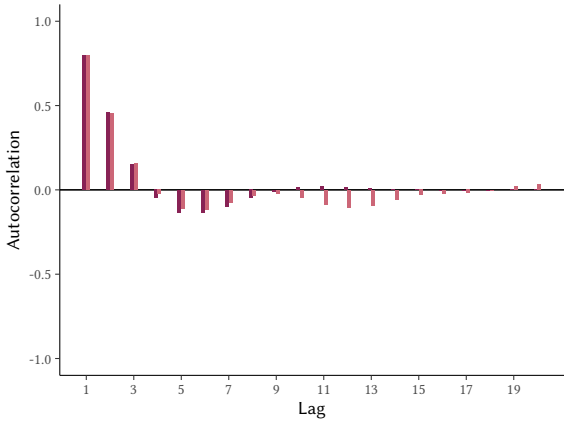
Once we have found a model that has passed the model diagnostics, we use it to forecast. Usually, we predict Z_{T+l} (l time steps into the future) by taking the conditional expectation given the process up to time T. We can, for example, isolate Z_{T+l} in (A.1) and use the following substitutions where $\mathscr{F}_T$ is the filtration up to time T:

- Since $\mathbb{E}\left[Z_{T-j} \mid \mathscr{F}_T\right] = Z_{T-j}$, for $j = 0, 1, \dots, T-1$, we leave Z_{T-j} unchanged.
- We replace Z_{T+j}, for $j = 1, 2, \dots$, with the forecasted values $\hat{Z}_{T+j}$. This is done recursively.
- For $j = 0, 1, \dots, T-1$, we have $c_{T-j} := \mathbb{E}\left[\varepsilon_{T-j} \mid \mathscr{F}_T\right] = Z_{T-j} - \hat{Z}_{T-j}$. Therefore, we replace ε_{T-j} with c_{T-j}.
- Since $\mathbb{E}\left[\varepsilon_{T+j} \mid \mathscr{F}_T\right] = 0$, for $j = 1, 2, \dots$, we replace ε_{T+j} with 0.

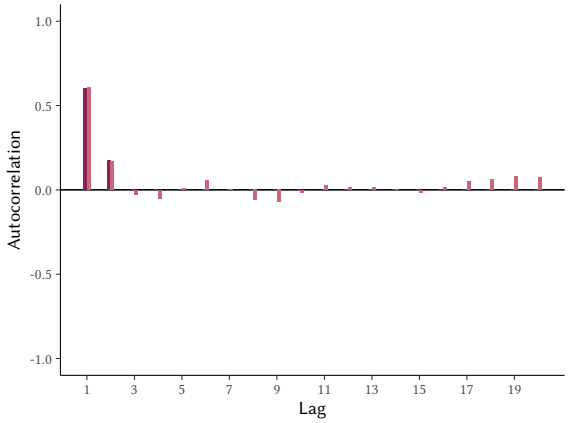
Table A.1 Summary of the shape of the ACF and the indicated model.

Behaviour	**Indicated model**	**Note**	**Figure**
Exponential decay to zero[a]	ARIMA$(p, 0, 0)$	Use PACF to identify order p	A.1a
Few spikes, rest are essentially zero	ARIMA$(0, 0, q)$	Order q determined by number of non-zero autocorrelations	A.1b
Decay starting after a few lags	ARIMA$(p, 0, q)$	PACF should also have similar behaviour	A.1c
All zero or close to zero	ARIMA$(0, 0, 0)$	Random noise	A.1d
No decay or slow decay to zero	ARIMA(p, d, q)	We need to difference the time series	A.1e

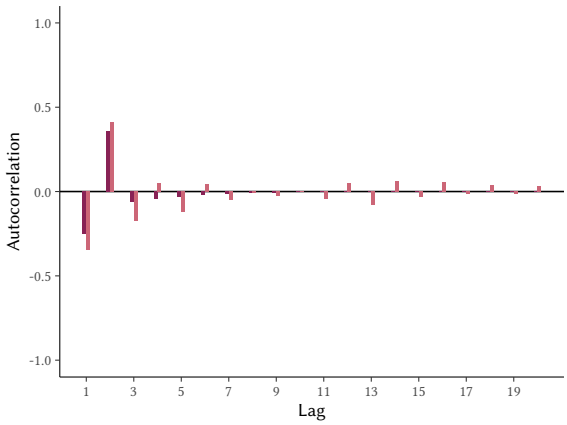
[a] Possibly alternating between positive or negative values.



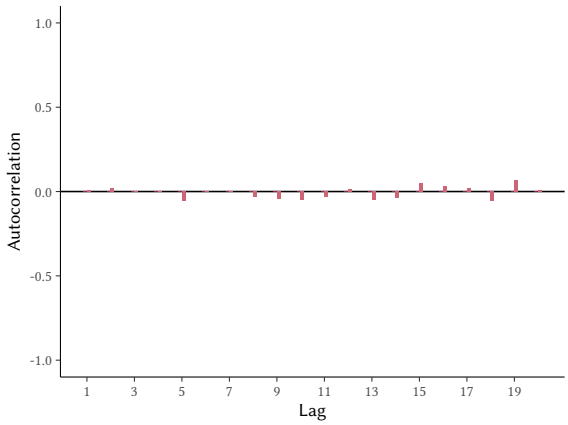
(a) ARIMA$(2, 0, 0)$



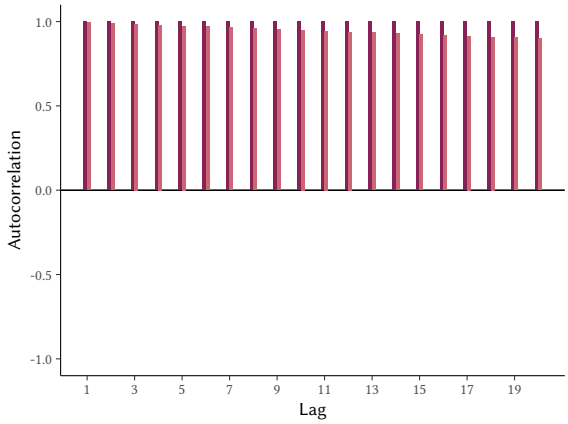
(b) ARIMA$(0, 0, 2)$



(c) ARIMA$(1, 0, 3)$



(d) ARIMA$(0, 0, 0)$



(e) ARIMA$(0, 1, 0)$

Fig. A.1 Theoretical (darker) and estimated ACF (lighter) of some common ARIMA models.

Appendix B

Code Listings, Tables, and Figures

B.1 Summary of Neural Networks Used

Listing B.1 Building the neural networks for Lee–Carter forecasting in R using `keras3`.

```
library(keras3)

lstm_input <- layer_input(shape = c(lookback, 1), name = "Sequential")
fnn_input  <- layer_input(shape = c(lookback), name = "Sequential")
gender     <- layer_input(shape = c(1), name = "Gender")

lstm <- lstm_input %>%
    layer_lstm(
        units = 15,
        activation = 'tanh',
        recurrent_activation = 'sigmoid',
        name = 'LSTMLayer'
    )

fnn <- fnn_input %>%
    layer_dense(units = 15, activation = 'relu', name = "FNLayer1") %>%
    layer_dense(units = 10, activation = 'relu', name = "FNLayer2") %>%
    layer_dense(units = 5, activation = 'relu', name = "FNLayer3")

lstm_deep <- lstm_input %>%
    layer_lstm(
        units = 15,
        activation = 'tanh',
        recurrent_activation = 'sigmoid',
        name = 'LSTMLayer1',
        return_sequences = TRUE
    ) %>%
    layer_lstm(
        units = 10,
        activation = 'tanh',
        recurrent_activation = 'sigmoid',
        name = 'LSTMLayer2',
        return_sequences = TRUE
    ) %>%
    layer_lstm(
        units = 5,
        activation = 'tanh',
        recurrent_activation = 'sigmoid',
        name = 'LSTMLayer3'
    )

output1 <- list(lstm, gender) %>%
    layer_concatenate() %>%
    layer_dense(units = 10, activation = 'tanh', name = "FNLayer") %>%
    layer_dense(units = 1, activation = 'linear', name = "Output")

output2 <- list(fnn, gender) %>%
    layer_concatenate() %>%
    layer_dense(units = 10, activation = 'tanh', name = "FNLayer4") %>%
```

```
    layer_dense(units = 1, activation = 'linear', name = "Output")

output3 <- list(lstm_deep, gender) %>%
    layer_concatenate() %>%
    layer_dense(units = 10, activation = 'tanh', name = "FNLayer") %>%
    layer_dense(units = 1, activation = 'linear', name = "Output")

# Shallow LSTM
model1 <- keras_model(inputs = list(lstm, gender), outputs = c(output1))

# FNN
model2 <- keras_model(inputs = list(fnn, gender), outputs = c(output2))

# Deep LSTM
model3 <- keras_model(inputs = list(lstm_deep, gender), outputs = c(output3))
```

Table B.1 Summary of the FNN.

Layer (type)	Output Shape	Param #	Connected to
Sequential (InputLayer)	(None, lookback)	0	–
FNLayer1 (Dense)	(None, 15)	820	Sequential[0][0]
FNLayer2 (Dense)	(None, 10)	315	FNLayer1[0][0]
FNLayer3 (Dense)	(None, 5)	160	FNLayer2[0][0]
Gender (InputLayer)	(None, 1)	0	–
concatenate (Concatenate)	(None, 6)	0	FNLayer3[0][0], Gender[0][0]
FNLayer4 (Dense)	(None, 10)	70	concatenate[0][0]
Output (Dense)	(None, 1)	11	FNLayer4[0][0]
Total params: 1,306 Trainable params: 1,306 Non-trainable params: 0			

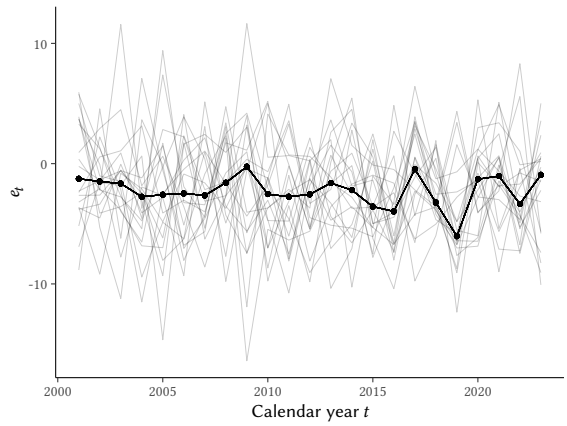
Table B.2 Summary of the shallow LSTM.

Layer (type)	Output Shape	Param #	Connected to
Sequential (InputLayer)	(None, lookback, 1)	0	–
LSTMLayer (LSTM)	(None, 15)	1020	Sequential[0][0]
Gender (InputLayer)	(None, 1)	0	–
concatenate (Concatenate)	(None, 6)	0	LSTMLayer[0][0], Gender[0][0]
FNLayer (Dense)	(None, 10)	170	concatenate[0][0]
Output (Dense)	(None, 1)	11	FNLayer[0][0]
Total params: 1201 Trainable params: 1201 Non-trainable params: 0			

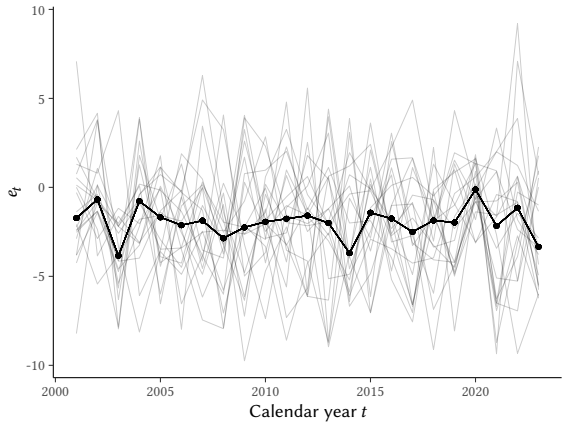
Table B.3 Summary of the deep LSTM.

Layer (type)	Output Shape	Param #	Connected to
`Sequential` (InputLayer)	`(None, lookback, 1)`	0	–
`LSTMLayer1` (LSTM)	`(None, 5, 15)`	1020	`Sequential[0][0]`
`LSTMLayer2` (LSTM)	`(None, 5, 10)`	1040	`LSTMLayer1[0][0]`
`LSTMLayer3` (LSTM)	`(None, 5)`	320	`LSTMLayer2[0][0]`
`Gender` (InputLayer)	`(None, 1)`	0	–
`concatenate` (Concatenate)	`(None, 6)`	0	`LSTMLayer3[0][0]`, `Gender[0][0]`
`FNLayer` (Dense)	`(None, 10)`	70	`concatenate[0][0]`
`Output` (Dense)	`(None, 1)`	11	`FNLayer[0][0]`
Total params: 2461 Trainable params: 2461 Non-trainable params: 0			

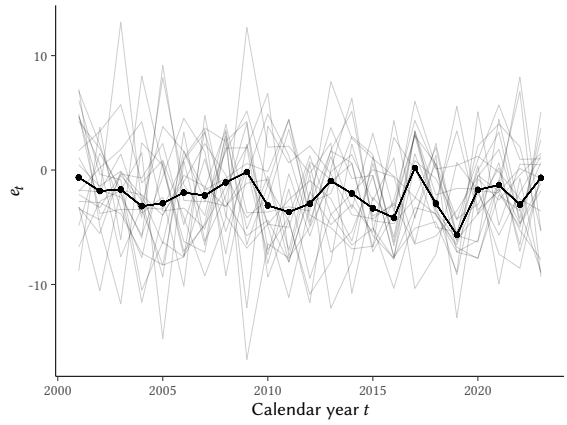
B.2 Out-of-Sample Trajectories



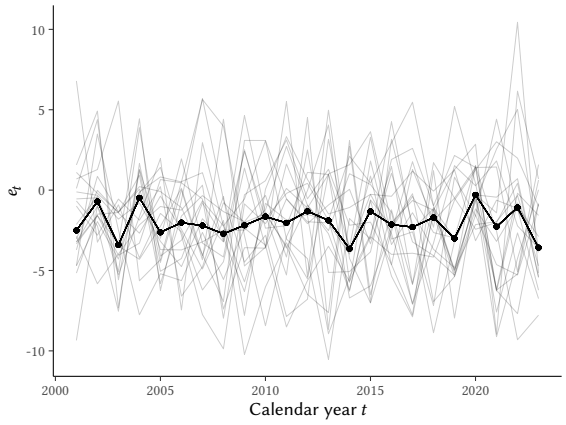
(a) FNN female



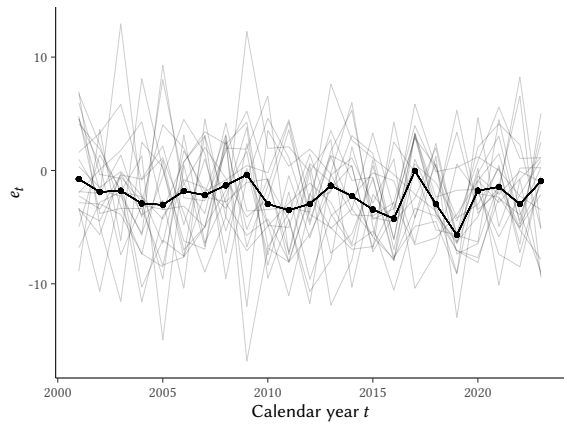
(b) FNN male



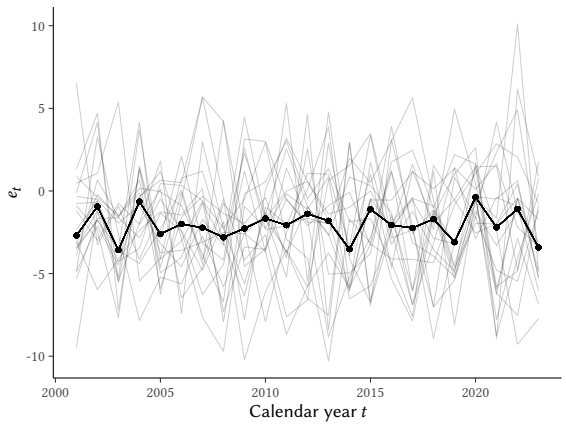
(c) LSTM female



(d) LSTM male



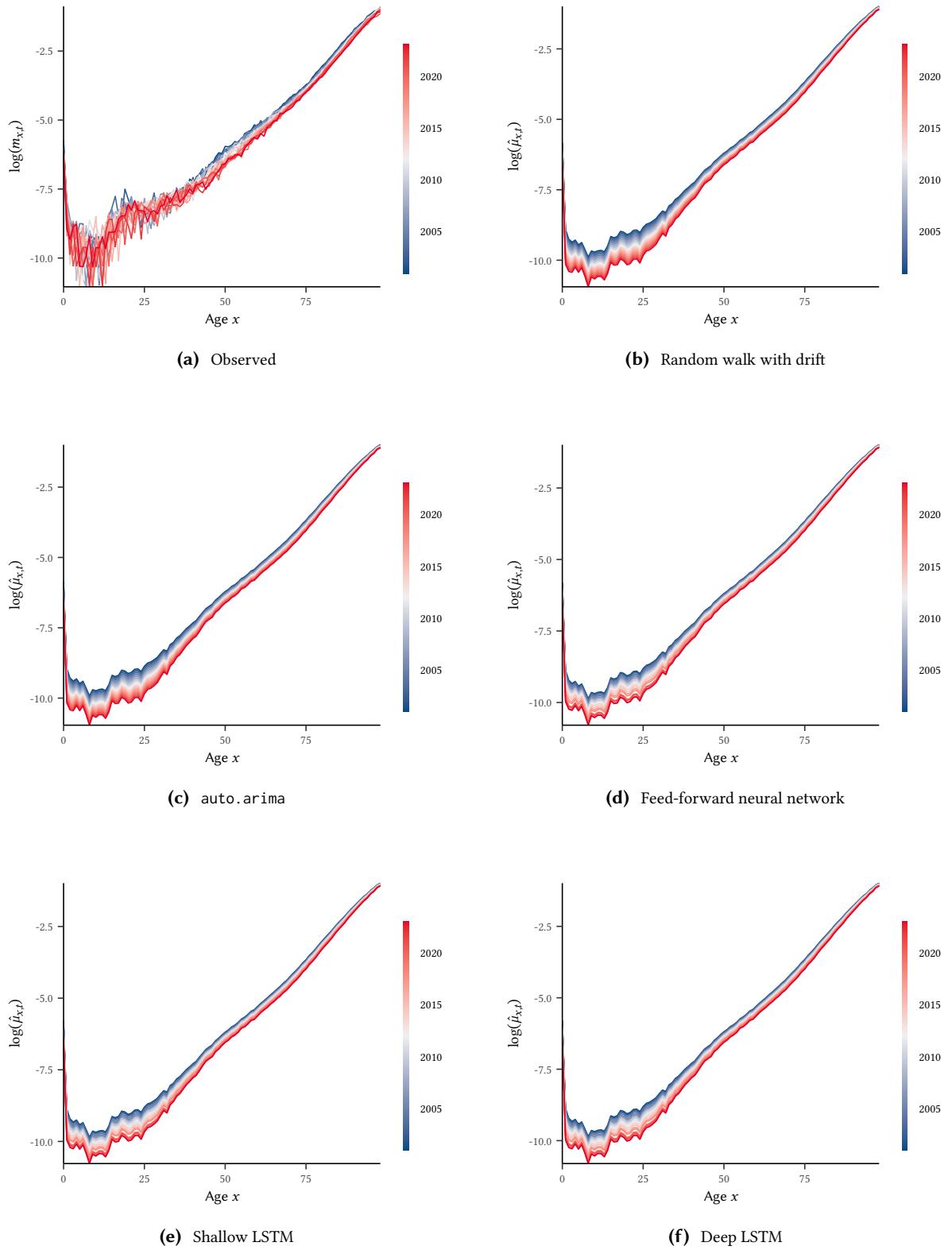
(e) Deep LSTM female



(f) Deep LSTM male

Fig. B.1 Out-of-sample simulated trajectories of the increment e_t.

B.3 Out-of-Sample Results



(a) Observed

(b) Random walk with drift

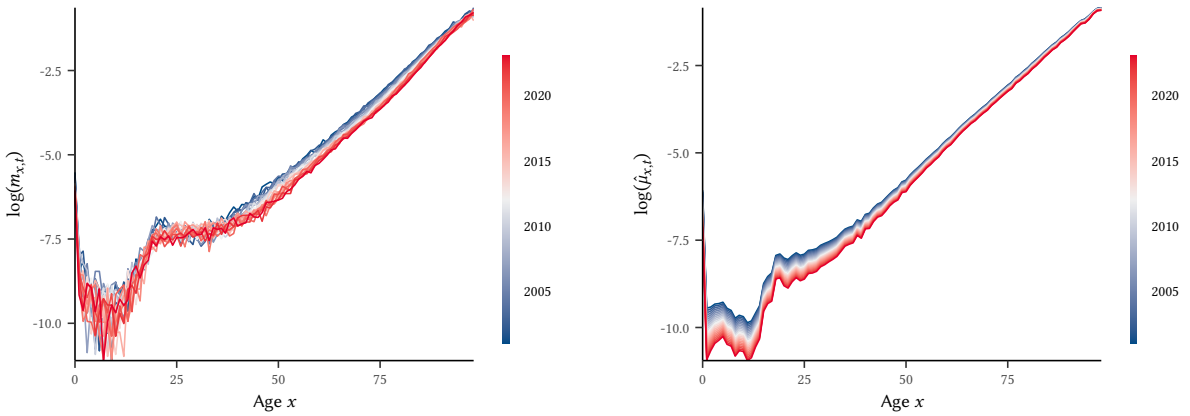
(c) `auto.arima`

(d) Feed-forward neural network

(e) Shallow LSTM

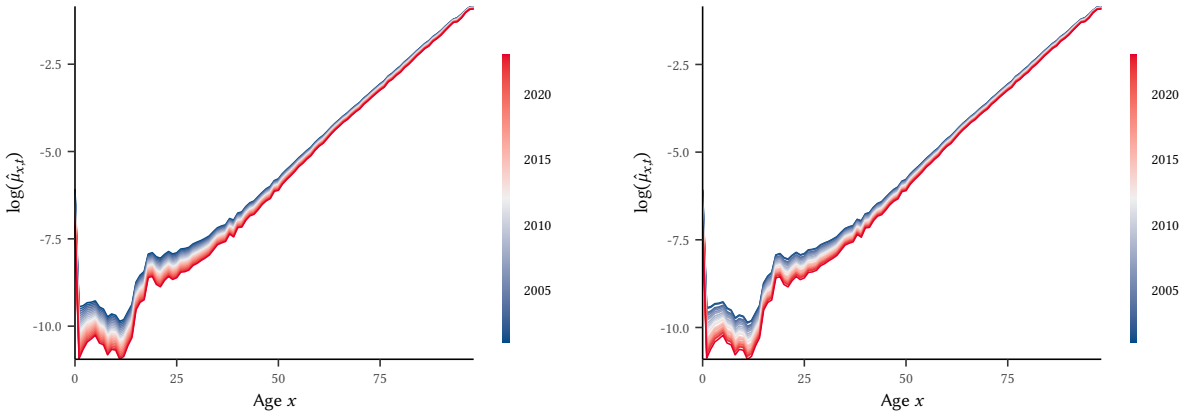
(f) Deep LSTM

Fig. B.2 Observed and predicted log-mortality using different methods for females.



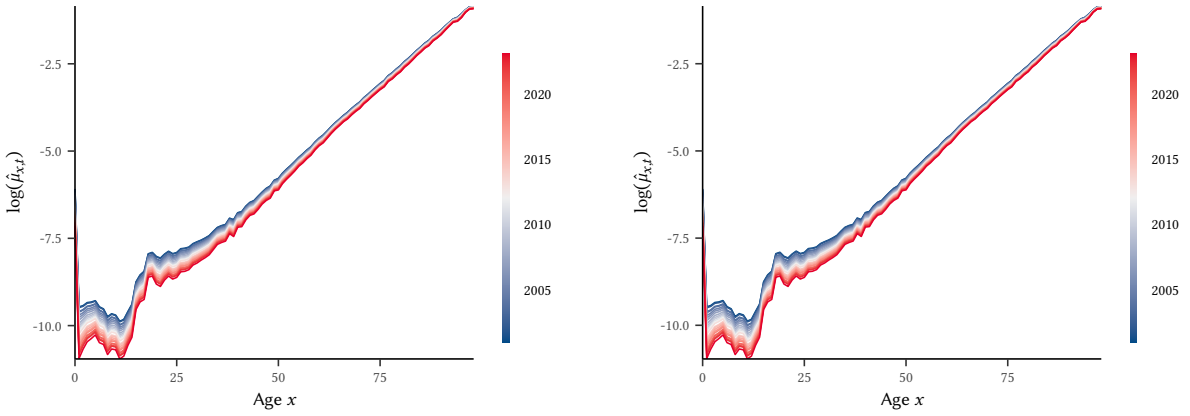
(a) Observed

(b) Random walk with drift



(c) `auto.arima`

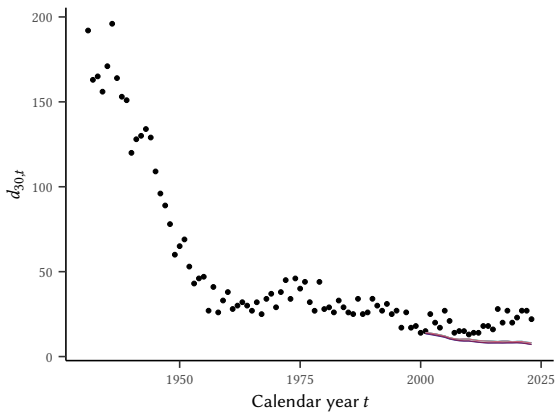
(d) Feed-forward neural network



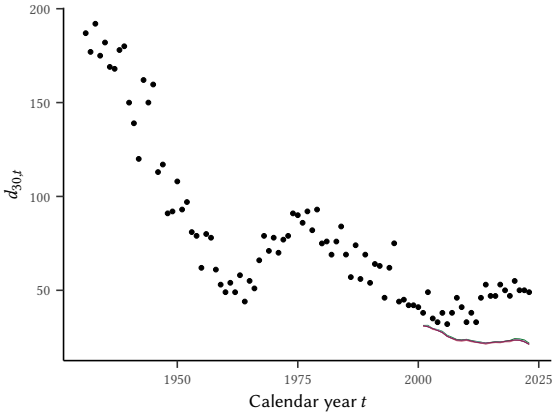
(e) Shallow LSTM

(f) Deep LSTM

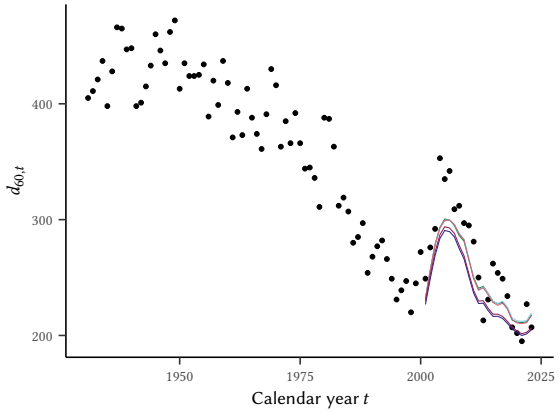
Fig. B.3 Observed and predicted log-mortality using different methods for males.



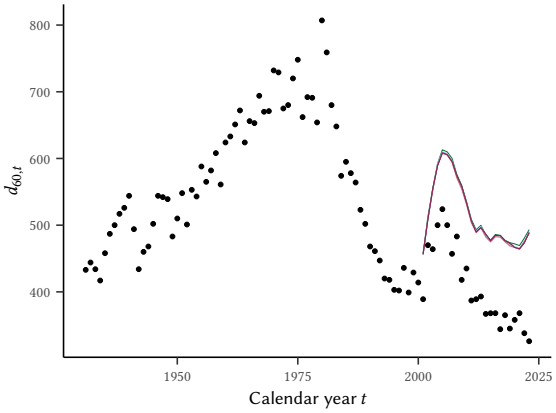
(a) Female, 30



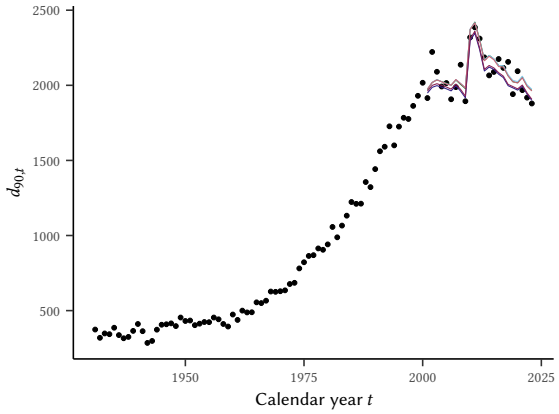
(b) Male, 30



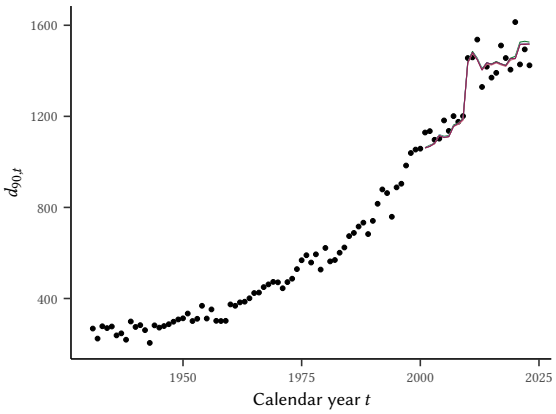
(c) Female, 60



(d) Male, 60



(e) Female, 90



(f) Male, 90

Fig. B.4 Observed and predicted number of deaths using different methods.