Model selection in time series by Deep Learning

Master's thesis in Actuarial Science

Benedikte Evensen

Supervisor

Hans A. Karlsen

Co-supervisor

Sondre Hølleland

Department of Mathematics

University of Bergen

June 2020

Abstract

In this thesis, we will explore the use of deep learning techniques for model selection in time series. We compare the results from this with more traditional approaches for model selection, namely the Akaike and Bayesian information criterion. Specifically, we simulate data from AR(p), MA(q) and, ARMA(p, q) time series of three different lengths. Neural network models, such as fully connected, convolutional (CNN) and, Long Short-Term Memory (LSTM) models, are trained on this data to classify the true order of each sample. The accuracy of the Akaike and Bayesian information criterion and the accuracies of the neural network models in classifying the correct order are then compared. We found that deep learning models outperform or perform as well as the information criterion method in selecting the true order for each dataset.

Acknowledgements

I want to thank my supervisors Hans A. Karlsen and Sondre Hølleland, for the idea and guidance on this thesis. I would also like to thank my fellow students, friends, and family, for their support.

Contents

Chapter 1

Introduction

The field of deep learning has developed rapidly in the past decade due to many innovations and increasing computing power. Deep learning is usually used as a general term for artificial neural networks with multiple layers. The early development of what we now call neural networks was originally inspired by the structure of the brain and how humans learn. The human brain is very complicated and still not fully understood, but simply put, it consists of a network of neurons that are connected by synapses. The synapses let neurons pass signals to each other. In artificial neural networks, the analogue for neurons and synapses are nodes that are connected through a network of weights. Despite the similarities between a human brain and a neural network, it has become evident that the similarities are superficial and neural networks are much simpler and do not perform tasks the way a human brain does. Still, neural networks have been capable of solving complex problems, like classifying images with high accuracy (Krizhevsky et al., 2012).

Neural networks have also been used on time series data, especially in forecasting, i.e., predicting future values. Siami-Namini et al. (2018) found that a type of neural network called Long Short-Term Memory (LSTM), was superior to the Autoregressive Integrated Moving Average (ARIMA) model in predicting future values.

In time series analysis, there already exist several methods for model selection for ARMA time series, like the Akaike and Bayesian information criteria. There have also been some attempts of model order selection in time series using neural networks, with varying results. Chenoweth et al. (2000) trained a neural network to identify the order of ARMA processes given their extended sample autocorrelation function (ESACF) table (Tsay and Tiao, 1984). The authors found that the neural network identified the correct order for 49.38% of the samples when the length of the time series was 3000 and 20.38% for length 100. However, the datasets were small, with only 800 samples in total for each time series length.

Al-Qawasmi et al. (2010) used another approach. The authors trained a neural network on a matrix constructed from the Minimum Eginevalue (MEV) criterion (Liang et al., 1993) to determine the ARMA model order. The length of each time series was 1500, and the results showed significant improvement in accuracy compared to using the MEV method alone.

The approaches in (Tsay and Tiao, 1984) and (Liang et al., 1993) use statistical properties or features of time series as inputs. Our approach is to use unprocessed time series data directly as input to train neural network models.

From Chapter 2 and 3, it will be apparent that deep learning methods and information criterion methods use a different approach to solve the order selection problem. When we use AIC or BIC to select order, we are selecting the orders that minimizes the one-step prediction error of the time series. This goal is not necessarily achieved by using the true order of the model. The predicted error is composed of a bias term and a variance term. The bias term is minimized at the order of the model, while the variance term increases with increased order. This is a classic trade-off between bias and variance. In the neural network method, this is not a concern. The neural network approach maximizes the probability of choosing the true order given a sample of time series data. That is, we simply seek to find a model that can correctly identify the true order. Thus, this thesis's objective is not necessarily to show that deep learning methods are better at order selection than AIC or BIC. Still, a comparison of these methods serves as a means to tell if a method that is not based on estimating the parameters in time series models can work.

The outline of this thesis is as follows. In Chapter 2, we briefly define the time series models used in this thesis and the two model selection methods AIC and BIC. We then introduce deep learning methods like fully connected, convolutional, and recurrent neural networks in chapter 3. In Chapter 4, we give a summary of how the datasets used in this thesis are simulated, and how we select orders using AIC and BIC scores. In Chapter 5, we discuss how the models are trained and the results. In Chapter 6, we summarize what we have learned and make some concluding remarks. In Appendix A, we present the models, tables, and plots of our results, and in Appendix B, we go more into detail about the technical details for model training.

Chapter 2

Time series

This chapter is based on Brockwell and Davis (2016). A time series $\{X_t\}$ is a stochastic process in discrete time. We observe time series data in many different areas, like finance, macroeconomy, climate data, and earthquake data. Statistical modelling of time series data typically consists of two steps. In the first step, different methods are used to transform the data into what could be recognized as a realization of a stationary time series. Then, we fit a time series model to the transformed data. This model might be an ARMA(p, q) model, which describes a stationary stochastic process. The hyperparameters (p, q), the order of the model, has to be chosen appropriately. This is a model selection problem.

Stationarity

Definition 2.1. A time series $\{X_t, \quad t = 0, \pm 1, \dots\}$ is said to be stationary if the mean $\mathrm{E}(X_t) = \mu_X(t)$ is independent of t, and the covariance $\mathrm{Cov}(X_{t+h}, X_t) = \gamma_X(t+h, t)$ is independent of t for any integer h.

This means that for a time series to be stationary $\{X_t\}$ has to have the same second-order properties as $\{X_{t+h}\}$ for any h.

2.1. Time series models

White noise

Definition 2.2. A stationary time series $\{Z_t\}$ of uncorrelated variables with zero mean is called white noise. We denote white noise as $\{Z_t\} \sim \mathrm{WN}\left(0, \sigma^2\right)$. This means that $\gamma_Z(h) = \delta_{0,h}\, \sigma^2$, where δ is the Kronecker-delta symbol.

Autoregressive moving-average (ARMA) model

Definition 2.3. The time series $\{X_t\}$ is an ARMA process of order (p, q) if it is stationary and

$$X_t - \phi_1 X_{t-1} - \cdots - \phi_p X_{t-p} = Z_t + \theta_1 Z_{t-1} + \cdots + \theta_q Z_{t-q} \quad \text{for } t \in \mathbb{Z}, \tag{2.1}$$

where $\{Z_t\} \sim \text{WN}\left(0, \sigma^2\right)$. The polynomials $\phi(z) = 1 - \phi_1 z - \cdots - \phi_p z^p$ and $\theta(z) = 1 + \theta_1 z + \cdots + \theta_q z^q$ are the autoregressive and the moving-average polynomial, respectively.

If $\{X_t\}$ is an ARMA(p, q) process, then $\phi(z)$ and $\theta(z)$ has no roots on the unit circle. If the white noise process $\{Z_t\}$ and the polynomials $\phi(z)$, $\theta(z)$ are given, then (2.1) defines an ARMA(p, q) model where $\{X_t\}$ is a possible solution that is required to be stationary. The model is *causal* if $\phi(z)$ has all roots strictly outside the unit circle. Then, X_t can be expressed in terms of past and present values of $\{Z_t\}$. *Invertibility* is defined similarly, but with the roots of $\theta(z)$ strictly outside the unit circle. If the model is invertible, Z_t can be expressed in terms of past and present values of $\{X_t\}$.

Note that if $\{X_t\}$ is an ARMA process the polynomials $\phi(z)$ and $\theta(z)$ are not unique, but the order (p, q) and the spectral density are unique.

Autoregressive (AR) model

Definition 2.4. The time series $\{X_t\}$ is a AR process of order p if

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + Z_t \quad \text{for } t \in \mathbb{Z}, \tag{2.2}$$

where $\{Z_t\} \sim \text{WN}\left(0, \sigma^2\right)$ and $\phi_p \neq 0$.

We see that this model is a special case of ARMA(p, q) with $q = 0$. The conditions for stationarity and causality for AR(p) are the same as that for ARMA(p, q), and from equation (2.2), we observe that AR(p) is always invertible by definition.

Moving-average (MA) model

Definition 2.5. The time series $\{X_t\}$ is a MA process of order q if

$$X_t = Z_t + \theta_1 Z_{t-1} + \cdots + \theta_q Z_{t-q} \quad \text{for } t \in \mathbb{Z}, \tag{2.3}$$

where $\{Z_t\} \sim \text{WN}\left(0, \sigma^2\right)$ and $\theta_q \neq 0$.

Again, we see that this model is a special case of ARMA(p, q) with $p = 0$, and from equation (2.3) we observe that MA(q) by definition always is causal and stationary. It is also invertible if the condition for invertibility that we described for ARMA(p, q) is fulfilled.

2.2. Methods for model selection

In time series analysis, we often want to find a model for our data. There are several methods to fit a model, and one of these methods is maximum likelihood estimation.

If we have a time series $\{X_t\}$ with $\{Z_t\} \sim \mathrm{N}\left(0, \sigma^2\right)$ and covariance matrix $\boldsymbol{\Gamma}_n = E(\boldsymbol{X}_n \boldsymbol{X}_n^T)$ which we assume to be non-singular, where $\boldsymbol{X}_n = [X_1, \dots, X_n]^T$, then the likelihood function of $\boldsymbol{X}_n$ is given by

$$L(\boldsymbol{\Gamma}_n) = (2\pi)^{-n/2} (\det \boldsymbol{\Gamma}_n)^{-1/2} \exp\big(-\frac{1}{2} \boldsymbol{X}_n^T \boldsymbol{\Gamma}_n^{-1} \boldsymbol{X}_n\big). \tag{2.4}$$

If $\{X_t\}$ is a ARMA(p,q) process, then we have a finite set of parameters, $\phi_1, \dots, \phi_p, \theta_1, \dots, \theta_q, \sigma^2$, and $\boldsymbol{\Gamma}_n$ can be expressed in terms of these. The likelihood function is a measure of how well the model fits the data given values for the parameters. To find the maximum likelihood estimators (MLE), i.e., the values that maximize L, we compute the partial derivatives of $\log L$ with respect to the parameters that we want to estimate to find the global maximum of L. In a practical setting, this is done by numerical methods.

For a given order (p, q) we want to find the parameters, $\hat{\boldsymbol{\phi}}$, $\hat{\boldsymbol{\theta}}$ and $\hat{\sigma}^2$, that maximizes the likelihood function, but in practice, we do not know the true order of the process the data is generated from. If we were to use the likelihood function evaluated at the values for the maximum likelihood estimators as a means to select the appropriate order, we would overfit the data. This is because the likelihood function increases with the number of parameters. Hence, we want to use another goodness of fit measure to select the right orders for our model. Akaike information criteria (Akaike, 1974) and the closely related Bayesian information criteria (Schwarz, 1978), are two such measures based on the likelihood function, but penalize a high number of parameters. Akaike information criterion (AIC) and Bayesian information criterion (BIC) are given by

$$\mathrm{AIC} = -2\log(L(\hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\theta}}, \hat{\sigma}^2)) + 2(p + q + 1),$$

and

$$\mathrm{BIC} = -2\log(L(\hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\theta}}, \hat{\sigma}^2)) + (p + q + 1)\log(n).$$

Thus, when selecting the order based on AIC or BIC, we want to find the (p, q) that minimizes the AIC or BIC value. The difference between the two is that the penalty term for BIC depends on the number of observations, n, in the sample. If $n \geq 8$, then BIC $>$ AIC and thus a high number of parameters is more heavily penalized by BIC than AIC. Hence, BIC will often tend to select a model with fewer parameters.

It can be shown that we can express $\boldsymbol{X}_n^T \Gamma_n^{-1} \boldsymbol{X}_n$ and $\det \boldsymbol{\Gamma}_n$ in equation (2.4) in terms of $X_j - \hat{X}_j$, the one-step prediction errors, and their variances $\nu_j = \mathrm{E}[(X_j - \hat{X}_j)^2]$. Hence, the likelihood of $\boldsymbol{X}_n$ can be rewritten as

$$L(\boldsymbol{\Gamma}_n) = (2\pi)^{-n/2}(\nu_0 \cdots \nu_{n-1})^{-1/2} \exp\Big(-\frac{1}{2}\sum_{j=1}^{n}(X_j - \hat{X}_j)^2/\nu_{j-1}\Big).$$

Thus, finding the order (p, q) that minimizes AIC or BIC is equivalent to finding the order that minimizes the prediction error while penalizing large models.

Chapter 3

Deep learning

Deep learning is a subfield of machine learning, and we can divide machine learning problems into two categories, *unsupervised* and *supervised learning.* When we have labelled data, i.e., a set of input variables $\boldsymbol{x}$ with an associated output label $\boldsymbol{y}$, we have a supervised learning problem. In unsupervised learning, the data is not labelled.

We also distinguish between classification and regression in supervised learning. If the output $\boldsymbol{y}$ is continuous, we have a regression problem, and if it is categorical, we have a classification problem. Either way, the goal in supervised learning is to estimate a function f, which describes the approximate relationship between the input vector $\boldsymbol{x}$ and the output variable $\boldsymbol{y}$.

It can be shown that under very general conditions, a neural network is a *universal approximator* (Hornik, 1991). That is, given a function f, a neural network can approximate f such that the error ϵ is arbitrarily small.

$$f(\boldsymbol{x}) = \boldsymbol{y} + \epsilon.$$

3.1. Fully connected neural networks

This section is based on Efron and Hastie (2016). There are many types of neural networks, but the basic form of a network is made up of layers, where each layer contains a specific number of *nodes*, and every node in each layer is connected to every node in the next layer via a vector of weights. This is called a *fully connected neural network.* Each weight can be interpreted as the "strength" of the connection between two nodes. The first layer is the input layer, and the last layer is the output layer. The layers in between are referred to as hidden layers.

3.1.1. Structure

To mathematically describe a fully connected network, we look at a neural network with K layers, including the input and output layer, where each layer is denoted by L_k for $k = 1, 2, \ldots, K$. The first layer represents the input vector $\boldsymbol{x} = \{x_t, \quad t = 1, 2, \dots, T\}$, and we begin with the transition from the input layer to the first hidden layer L_2. A linear combination of the input and a weight matrix $\boldsymbol{W}^{(1)}$ that consists of all weights between the first and second layer is computed and for each pair of nodes this can be written as

$$z_l^{(2)} = w_{l0}^{(1)} + \sum_{t=1}^{T} w_{lj}^{(1)} x_t \qquad \text{for } l = 1, \dots, n_2, \tag{3.1}$$

where $w_{l0}^{(1)}$ is the bias weight or intercept, T is the number elements in the input vector, and n_2 is the number of nodes in the first hidden layer. Then the $z_l^{(2)}$'s are passed through a non-linear *activation function* and become the values for the nodes in the second layer L_2. We write this as

$$a_l^{(2)} = g^{(2)}\big(z_l^{(2)}\big) \qquad \text{for } l = 1, \dots, n_2, \tag{3.2}$$

where the $a_l^{(2)}$'s are the values for the nodes in this layer. These are the values that will be passed onto the next layer, L_3. See Figure 3.1 for an illustration of a single node in the first hidden layer.

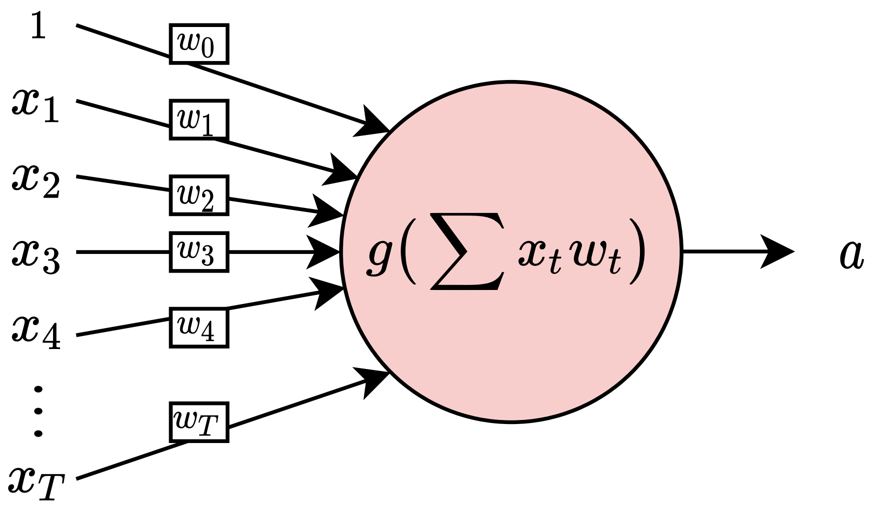


Figure 3.1: Illustration of a single node that computes the activation.

For a general layer, L_k, we extend (3.1) and (3.2) to:

$$\begin{aligned} z_l^{(k)} &= w_{l0}^{(k-1)} + \sum_{j=1}^{n_{k-1}} w_{lt}^{(k-1)} a_j^{(k-1)}, \\ a_l^{(k)} &= g^{(k)}(z_l^{(k)}), \end{aligned} \qquad \text{for } l = 1, \dots, n_k, \tag{3.3}$$

where n_{k-1} is the number of nodes in layer L_{k-1}. These general equations are also true for $k = 2$ if we let $a_j^{(1)} = x_j$.

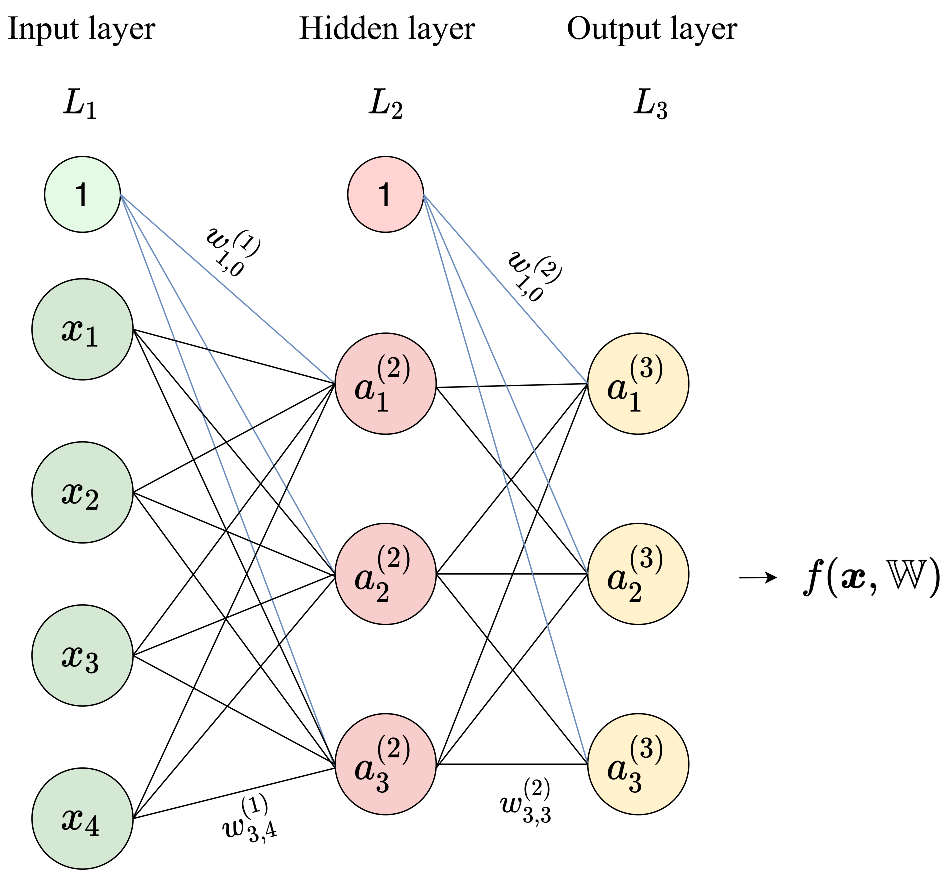


Figure 3.2: A fully connected neural network for classification with one hidden layer.

The process of passing the input values through the nodes in a neural network is often referred to as the feed-forward pass. See Figure 3.2 for an illustration of a fully connected neural network.

Activation functions

The non-linear $g^{(k)}(\cdot)$ is the activation function for layer L_k. Some of the most common activation functions for the hidden layers are sigmoid, rectified linear unit (reLU) and tanh (see Figure 3.3).

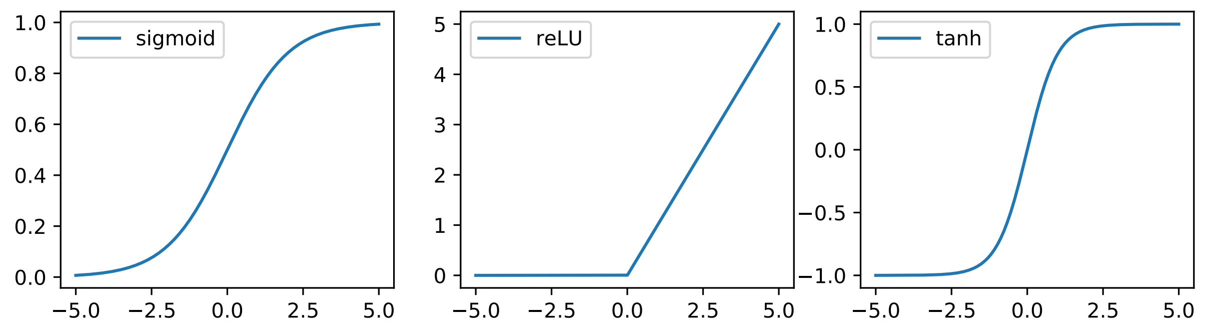


Figure 3.3: Activation functions

In the output layer the activation function, $g^{(K)}(\cdot)$ is usually different. For multiclass classification, the softmax function is used, and each node in the output layer represents one of the classes. If there are M classes, the softmax function will be

$$a_j^{(K)} = g^{(K)}\big(z_j^{(K)}; \boldsymbol{z}^{(K)}\big) = \frac{\exp z_j^{(K)}}{\sum_{l=1}^{M} \exp z_l^{(K)}}, \quad \text{for } j = 1, \dots, M. \tag{3.4}$$

Unlike the activation functions for the hidden layers, this is a function of all z_l's in the output layer. We see that the denominator of (3.4) ensures that the function outputs values in the interval $(0, 1)$, and that

$$\sum_{j=1}^{M} a_j^{(K)} = 1,$$

the sum of the outputs over all classes M is 1. Thus, we can view $a_j^{(K)}$ as the probability the neural network model predicts for the j-th class being the correct class, given $\boldsymbol{z}^{(K)}$. That is,

$$a_j^{(K)} = P\big(\text{true class } = j \mid \boldsymbol{z}^{(K)}\big).$$

The index of the output activation element with the highest value, $\arg\max_j a_j^{(K)}$, is the predicted class.

Feed-forward pass in matrix-vector notation

The following set of equations is a complete description of the feed-forward pass

$$\begin{aligned} \boldsymbol{a}^{(1)} &= \boldsymbol{x}, \\ \boldsymbol{z}^{(k)} &= \boldsymbol{W}^{(k-1)}\boldsymbol{a}^{(k-1)}, \\ \boldsymbol{a}^{(k)} &= g^{(k)}\big(\boldsymbol{z}^{(k)}\big), \end{aligned} \qquad \text{for } k = 2, \dots, K, \tag{3.5}$$

or written out

$$\begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{n_k}^{(k)} \end{bmatrix} = \begin{bmatrix} w_{1,0}^{(k-1)} & w_{1,1}^{(k-1)} & \dots & w_{1,n_{k-1}}^{(k-1)} \\ w_{2,0}^{(k-1)} & w_{2,1}^{(k-1)} & \dots & w_{2,n_{k-1}}^{(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_k,0}^{(k-1)} & w_{n_k,1}^{(k-1)} & \dots & w_{n_k,n_{k-1}}^{(k-1)} \end{bmatrix} \begin{bmatrix} 1 \\ a_1^{(k-1)} \\ \vdots \\ a_{n_{k-1}}^{(k-1)} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} a_1^{(k)} \\ \vdots \\ a_{n_k}^{(k)} \end{bmatrix} = \begin{bmatrix} g^{(k)}\big(z_1^{(k)}\big) \\ \vdots \\ g^{(k)}\big(z_{n_k}^{(k)}\big) \end{bmatrix}.$$

3.1.2. Model fitting

When we talk about training or fitting a model, we mean solving an optimization problem, and this amounts to minimize a cost (sometimes called loss) function. We denote the cost function by $C = C\big(\boldsymbol{y}, f(\boldsymbol{x}, \mathbb{W})\big)$, where the target is $\boldsymbol{y} = \{y_i, \quad i = 1, \dots, M\}$, and f is an function of all the weights, $\mathbb{W} = \{\boldsymbol{W}^{(k)}, \quad k = 1, 2, \dots, K-1\}$, and the input, $\boldsymbol{x} = \{x_t, \quad t = 1, 2, \dots, T\}$. Equation (3.5) implies the useful recursion

$$\boldsymbol{z}^{(k)} = \boldsymbol{W}^{(k-1)} g^{(k-1)}\big(\boldsymbol{z}^{(k-1)}\big). \tag{3.6}$$

Now,

$$\begin{aligned} f(\boldsymbol{x}, \mathbb{W}) &= g^{(K)}\big(\boldsymbol{z}^{(K)}\big) \\ &= g^{(K)}\big(\boldsymbol{W}^{(K-1)} g^{(K-1)}\big(\boldsymbol{z}^{(K-1)}\big)\big) = f_1\big(\boldsymbol{W}^{(K-1)}, \boldsymbol{z}^{(K-1)}\big) \\ &= g^{(K)}\big(\boldsymbol{W}^{(K-1)} g^{(K-1)}\big(\boldsymbol{W}^{(K-2)} \boldsymbol{z}^{(K-2)}\big)\big) = f_2\big(\boldsymbol{W}^{(K-1)}, \boldsymbol{W}^{(K-2)}, \boldsymbol{z}^{(K-2)}\big) \\ &= f_{K-k}\big(\boldsymbol{W}^{(K-1)}, \dots, \boldsymbol{W}^{(K-k)}, \boldsymbol{z}^{(K-k)}\big). \end{aligned}$$

This can also be written as

$$f(\boldsymbol{x}, \mathbb{W}) = f_{K-k}\big(\boldsymbol{W}^{(K-1)}, \dots, \boldsymbol{W}^{(K-k)}, \boldsymbol{a}^{(K-k)}\big).$$

The goal is to estimate or "learn" the optimal weights by minimizing the cost function with respect to the weights. Since our cost function is usually non-convex, there will be several local minima, and finding the global one is difficult or impossible. Thus, we often have to settle for a good local minimum, but this is not considered a big problem in machine learning. A regularization term is often added to the cost function, and this will be discussed in more detail in 3.1.3.

In our case the input $\boldsymbol{x}$ is one single sample of a time series, $\boldsymbol{W}^{(k)}$ is the weight matrix between layers L_k and L_{k+1} and $\boldsymbol{y}$ is the true order of $\boldsymbol{x}$. We use the optimization technique *gradient descent* to minimize the cost function. To compute the gradient descent, an algorithm called backpropagation is used. To illustrate how backpropagation and gradient descent works we first use a batch size of one to train the network, which means that for each pair of input $\boldsymbol{x}$ and label $\boldsymbol{y}$, we compute the per-sample cost C and then the gradient, using backpropagation to update the weights. After this, we use the next pair $(\boldsymbol{x}, \boldsymbol{y})$ and do the same, until we have done this for all the pairs in our training dataset at least once. Later we will generalize backpropagation and gradient descent for multiple samples at a time.

Again we look at a neural network with K layers. Since we want to minimize C with respect to the weights, we have to take the partial derivative with respect to each element $w_{lj}^{(k)}$ in each weight matrix $\boldsymbol{W}^{(k)}$ for $k = 1, 2, \dots, K-1$,

$$\frac{\partial C}{\partial w_{lj}^{(k)}} = \frac{\partial C}{\partial z_l^{(k+1)}} \frac{\partial z_l^{(k+1)}}{\partial w_{lj}^{(k)}}. \tag{3.7}$$

First, we look closer at the first term on the right-hand side in (3.7). For layer k and the nodes in this layer, we want to compute

$$\delta_l^{(k)} \overset{\text{def}}{=} \frac{\partial C}{\partial z_l^{(k)}}.$$

We can interpret $\delta_l^{(k)}$ as a measure of how much node l in layer k contributes to the total cost, C. For each node in the output layer $k = K$, we have

$$\delta_l^{(K)} = \frac{\partial C}{\partial z_l^{(K)}} = \frac{\partial C}{\partial a_l^{(K)}} \frac{\partial a_l^{(K)}}{\partial z_l^{(K)}}. \tag{3.8}$$

For each node in layers $k = K-1, K-2, \dots, 2$ we have

$$\delta_l^{(k)} = \frac{\partial C}{\partial z_l^{(k)}} = \sum_{j=1}^{n_{k+1}} \left[\frac{\partial C}{\partial a_j^{(k+1)}} \frac{\partial a_j^{(k+1)}}{\partial z_j^{(k+1)}} \frac{\partial z_j^{(k+1)}}{\partial a_l^{(k)}} \right] \frac{\partial a_l^{(k)}}{\partial z_l^{(k)}},$$

where n_{k+1} is the number of nodes in layer L_{k+1}. We observe that

$$\frac{\partial C}{\partial a_j^{(k+1)}} \frac{\partial a_j^{(k+1)}}{\partial z_j^{(k+1)}} = \delta_j^{(k+1)}$$

and

$$\frac{\partial z_j^{(k+1)}}{\partial a_l^{(k)}} = \frac{\partial}{\partial a_l^{(k)}} \left[w_{j0}^{(k)} + \sum_{i=1}^{n_k} w_{ji}^{(k)} a_i^{(k)} \right] = w_{jl}^{(k)}.$$

Thus, we have

$$\delta_l^{(k)} = \frac{\partial C}{\partial z_l^{(k)}} = \sum_{j=1}^{n_{k+1}} \left[\delta_j^{(k+1)} w_{jl}^{(k)} \right] \frac{\partial a_l^{(k)}}{\partial z_l^{(k)}}. \tag{3.9}$$

Going back to the partial derivative (3.7) and the second term on the right-hand side, we see that this becomes

$$\frac{\partial z_l^{(k+1)}}{\partial w_{lj}^{(k)}} = \frac{\partial}{\partial w_{lj}^{(k)}} \left[w_{l0}^{(k)} + \sum_{j=1}^{n_k} w_{lj}^{(k)} a_j^{(k)} \right] = a_j^{(k)}.$$

Thus,

$$\frac{\partial C\left(\boldsymbol{y}, f(\boldsymbol{x}, \mathbb{W})\right)}{\partial w_{lj}^{(k)}} = \delta_l^{(k+1)} a_j^{(k)}, \qquad \text{for } k = 1, 2, \ldots, K-1. \tag{3.10}$$

The backpropagacation algorithm

For a single training pair $(\boldsymbol{y}, \boldsymbol{x})$, where $\boldsymbol{x}$ is passed through the network we:

1. Calculate $\delta_l^{(K)}$ for all nodes l in layer L_K.
2. Calculate $\delta_l^{(k)}$ for $k = K-1$ and all nodes l in layer L_{K-1} and then do the same iteratively for $k = K-2, \ldots, 2$.
3. Now that all δ's are calculated, we can compute the partial derivatives of the cost function $C\left(\boldsymbol{y}, f(\boldsymbol{x}, \mathbb{W})\right)$ with respect to each weight $w_{lj}^{(k)}$.

After we have used backpropagation to calculate the derivatives, we update each weight in response to the derivatives in the following way

$$w_{lj}^{*(k)} = w_{lj}^{(k)} - \alpha\, \delta_l^{(k+1)} a_j^{(k)}, \tag{3.11}$$

where $w_{lj}^{*(k)}$ denotes the updated weight and α is a constant called the learning rate. Details about the learning rate will be discussed in section 3.1.3.

Since we will use neural networks to classify time series, we have a multiclass classification problem. In these cases we use cross-entropy as the cost function.

$$C\big(\boldsymbol{y}, f(\boldsymbol{x}, \mathbb{W})\big) = -\boldsymbol{y}^T \log\left(\boldsymbol{a}^{(K)}\right) = -\sum_{j=1}^{M} y_j \log\left(a_j^{(K)}\right), \tag{3.12}$$

where M is the number of classes, and the label vector $\boldsymbol{y}$, is a one-hot encoded vector, which means that all elements are zero, except the true class t which is one. In other words $y_t = 1$ and $y_j = 0$ for all $j \neq t$.

The reason for using the cross-entropy as a cost function in the case of multiclass classification problems is that we want to maximize the probability of correctly classifying a sample. We can do this by using maximum likelihood estimation to find the weights that maximize the conditional probability $P(\boldsymbol{y}|\boldsymbol{z}, \mathbb{W})$. We can write this probability as

$$P(\boldsymbol{y}|\boldsymbol{x}, \mathbb{W}) = \prod_{j=1}^{M} P\big(\text{true class } = j \mid \boldsymbol{z}^{(K)}\big)^{y_j} = \prod_{j=1}^{M} (a_j^{(K)})^{y_j}.$$

Maximizing this is equivalent to minimizing

$$-\log P(\boldsymbol{y}|\boldsymbol{x}, \mathbb{W}) = -\log \prod_{j=1}^{M} (a_j^{(K)})^{y_j} = -\sum_{j=1}^{M} y_j \log\big(a_j^{(K)}\big),$$

which is the cross-entropy function.

We can now calculate the partial derivative of C with respect to $z_l^{(K)}$ from equation (3.8).

$$\begin{aligned}\frac{\partial C}{\partial z_l^{(K)}} &= -\sum_{j=1}^{M} y_j \frac{\partial}{\partial z_l^{(K)}} \log(a_j^{(K)}) = -\sum_{j=1}^{M} y_j \frac{1}{a_j^{(K)}} \frac{\partial a_j^{(K)}}{\partial z_l^{(K)}} \\ &= -\, y_t \frac{1}{a_t^{(K)}} \frac{\partial a_t^{(K)}}{\partial z_l^{(K)}} = -\frac{1}{a_t^{(K)}} \frac{\partial a_t^{(K)}}{\partial z_l^{(K)}},\end{aligned}$$

because when t is the true class, $y_t = 1$ and $y_j = 0$ for all $j \neq t$. Also recall that

$$a_j^{(K)} = \frac{\exp z_j^{(K)}}{\sum_{i=1}^{M} \exp z_i^{(K)}}.$$

Therefore, when $l = t$, we have

$$\frac{\partial a_t^{(K)}}{\partial z_l^{(K)}} = \frac{\partial a_t^{(K)}}{\partial z_t^{(K)}} = \frac{\exp z_t^{(K)} \Big(\sum_{i=1}^{M} \exp z_i^{(K)} - \exp z_t^{(K)} \Big)}{\Big(\sum_{i=1}^{M} \exp z_i^{(K)} \Big)^2},$$

and thus,

$$\begin{aligned}\frac{\partial C}{\partial z_l^{(K)}} &= -\left(\frac{\sum_{i=1}^{M} \exp z_i^{(K)}}{\exp z_t^{(K)}} \right) \left(\frac{\exp z_t^{(K)} (\sum_{i=1}^{M} \exp z_i^{(K)} - \exp z_t)}{(\sum_{i=1}^{M} \exp z_i^{(K)})^2} \right) \\ &= -\frac{(\sum_{i=1}^{M} \exp z_i^{(K)} - \exp z_t^{(K)})}{\sum_{i=1}^{M} \exp z_i^{(K)}} = \frac{\exp z_t^{(K)}}{\sum_{i=1}^{M} \exp z_i^{(K)}} - 1 = a_t^{(K)} - 1.\end{aligned}$$

When $l \neq t$, we have that

$$\frac{\partial a_t^{(K)}}{\partial z_l^{(K)}} = -\frac{\exp z_t^{(K)} \exp z_l^{(K)}}{(\sum_{i=1}^{M} \exp z_i^{(K)})^2},$$

and

$$\begin{aligned}\frac{\partial C}{\partial z_l^{(K)}} =& \left(-\frac{\sum_{i=1}^{M} \exp z_i^{(K)}}{\exp z_t^{(K)}} \right) \left(-\frac{\exp z_t^{(K)} \exp z_l^{(K)}}{(\sum_{i=1}^{M} \exp z_i^{(K)})^2} \right) \\ =& \frac{\exp z_l^{(K)}}{\sum_{i=1}^{M} \exp z_i^{(K)}} = a_l^{(K)}.\end{aligned}$$

Hence, we conclude that

$$\delta_l^{(K)} = \frac{\partial C}{\partial z_l^{(K)}} = a_l^{(K)} - y_l = \begin{cases} a_t^{(K)} - 1, & \text{when } l = t, \\ a_l^{(K)}, & \text{when } l \neq t. \end{cases} \tag{3.13}$$

Example 3.1. Let us look at the backpropagation and gradient descent calculations for a neural network for classification of M classes with one hidden layer. We have the function for the network

$$f(\boldsymbol{x}, \mathbb{W}) = f\big(\boldsymbol{W}^{(2)}, \boldsymbol{W}^{(1)}, \boldsymbol{x}\big),$$

that represents a feed-forward pass of the sample $\boldsymbol{x}$. We use the categorical cross-entropy as cost function (3.12) and $g^{(3)}$ is the softmax function. The $\delta^{(K)}$'s are already calculated from (3.13), so we have

$$\delta_l^{(3)} = \frac{\partial C}{\partial z_l^{(3)}} = a_l^{(3)} - y_l = \begin{cases} a_t^{(3)} - 1, & \text{when } l = t, \\ a_l^{(3)}, & \text{when } l \neq t, \end{cases}$$

where t is the true class for $\boldsymbol{x}$. In this example we let the activation function for the transition between the input and hidden layer $g^{(2)}$, be the sigmoid function

$$a_l^{(2)} = g^{(2)}(z_l^{(2)}) = \frac{1}{1 + \exp\big(- z_l^{(2)}\big)}.$$

We then calculate the $\delta^{(2)}$'s. From equation (3.9), we have

$$\begin{aligned}
\delta_l^{(2)} =& \frac{\partial C}{\partial z_l^{(2)}} = \sum_{j=1}^{M} \left[\delta_j^{(3)} w_{jl}^{(2)}\right] \frac{\partial a_l^{(2)}}{\partial z_l^{(2)}} \\
=& \sum_{j=1}^{M} \left[(a_l^{(3)} - y_l) w_{jl}^{(2)}\right] \frac{\exp(-z_l^{(2)})}{\Big(1 + \exp(-z_l^{(2)})\Big)^2} \\
=& \sum_{j=1}^{M} \left[\left(\frac{\exp z_l^{(3)}}{\sum_{i=1}^{M} \exp z_i^{(3)}} - y_l\right) w_{jl}^{(2)}\right] \frac{\exp(-z_l^{(2)})}{\Big(1 + \exp(-z_l^{(2)})\Big)^2}.
\end{aligned}$$

We can now calculate the gradients with respect to each weight using equation (3.10);

$$\frac{\partial C}{\partial w_{lj}^{(2)}} = \delta_l^{(3)} a_j^{(2)} = \Big(a_l^{(3)} - y_l\Big)\, a_j^{(2)} = \left(\frac{\exp z_l^{(3)}}{\sum_{i=1}^{M} \exp z_i^{(3)}} - y_l\right)\left(\frac{1}{1 + \exp\big(- z_j^{(2)}\big)}\right)$$

and

$$\frac{\partial C}{\partial w_{lj}^{(1)}} = \delta_l^{(2)} a_j^{(1)} = \sum_{j=1}^{M} \left[\left(\frac{\exp z_l^{(3)}}{\sum_{i=1}^{M} \exp z_i^{(3)}} - y_l\right) w_{jl}^{(2)}\right] \frac{\exp(-z_l^{(2)})}{\Big(1 + \exp(-z_l^{(2)})\Big)^2} x_j.$$

Thus, we get the weight updates

$$w_{lj}^{*(2)} = w_{lj}^{(2)} - \alpha \left(\left(\frac{\exp z_l^{(3)}}{\sum_{i=1}^{M} \exp z_i^{(3)}} - y_l \right) \left(\frac{1}{1 + \exp\big(- z_j^{(2)} \big)} \right) \right)$$

and

$$w_{lj}^{*(1)} = w_{lj}^{(1)} - \alpha \left(\sum_{j=1}^{M} \left[\left(\frac{\exp z_l^{(3)}}{\sum_{i=1}^{M} \exp z_i^{(3)}} - y_l \right) w_{jl}^{(2)} \right] \frac{\exp(-z_l^{(2)})}{\left(1 + \exp(-z_l^{(2)})\right)^2} x_j \right).$$

Backpropagation equations and gradient descent in vector-matrix notation

To sum up this part of the chapter, we will write the equations for backpropagation and gradient descent in matrix-vector notation when using cross-entropy as cost function. We can write the equations (3.8) and (3.9) as

$$\boldsymbol{\delta}^{(K)} = \boldsymbol{a}^{(K)} - \boldsymbol{y}$$

and

$$\boldsymbol{\delta}^{(k)} = \left({\boldsymbol{W}^{(k)}}^T \boldsymbol{\delta}^{(k+1)} \right) \odot \frac{\partial g^{(k)}}{\partial \boldsymbol{z}^{(k)}}, \quad \text{for } k = K-1, \dots, 2,$$

where $\odot$ is the Hadamard product, i.e. an element-wise product. Note that

$$\frac{\partial g^{(k)}}{\partial \boldsymbol{z}^{(k)}} = \begin{bmatrix} \frac{\partial g^{(k)}}{\partial z_1^{(k)}} \\ \vdots \\ \frac{\partial g^{(k)}}{\partial z_{p_k}^{(k)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}} \\ \vdots \\ \frac{\partial a_{p_k}^{(k)}}{\partial z_{p_k}^{(k)}} \end{bmatrix}.$$

The gradient of the cost function (3.10) with respect to each weight matrix can be written as

$$\nabla_{\boldsymbol{W}^{(k)}} C = \frac{\partial C\left(\boldsymbol{y}, f(\boldsymbol{x}, \mathbb{W})\right)}{\partial \boldsymbol{W}^{(k)}} = \boldsymbol{\delta}^{(k+1)} {\boldsymbol{a}^{(k)}}^T, \quad k = 1, \dots, K-1. \tag{3.14}$$

Lastly, we evaluate $\nabla_{\boldsymbol{W}^{(k)}} C$ for the values in $\boldsymbol{y}$, $\boldsymbol{x}$ and $\mathbb{W}$. Then the update (3.11) of each weight matrix can be calculated by the equation:

$$\boldsymbol{W}^{*(k)} = \boldsymbol{W}^{(k)} - \alpha \nabla_{\boldsymbol{W}^{(k)}} C, \quad k = 1, \dots, K-1, \tag{3.15}$$

where $\boldsymbol{W}^{*(k)}$ is the updated weight matrix for layer L_k. We observe in equation (3.15) that the update for the k-th weight matrix is independent of the updates for the other weight matrices. Thus, we can calculate the weight updates for $k = 1, \dots, K-1$ at the same time, i.e. in *parallel*. This is an important feature of gradient descent as it speeds up the training process.

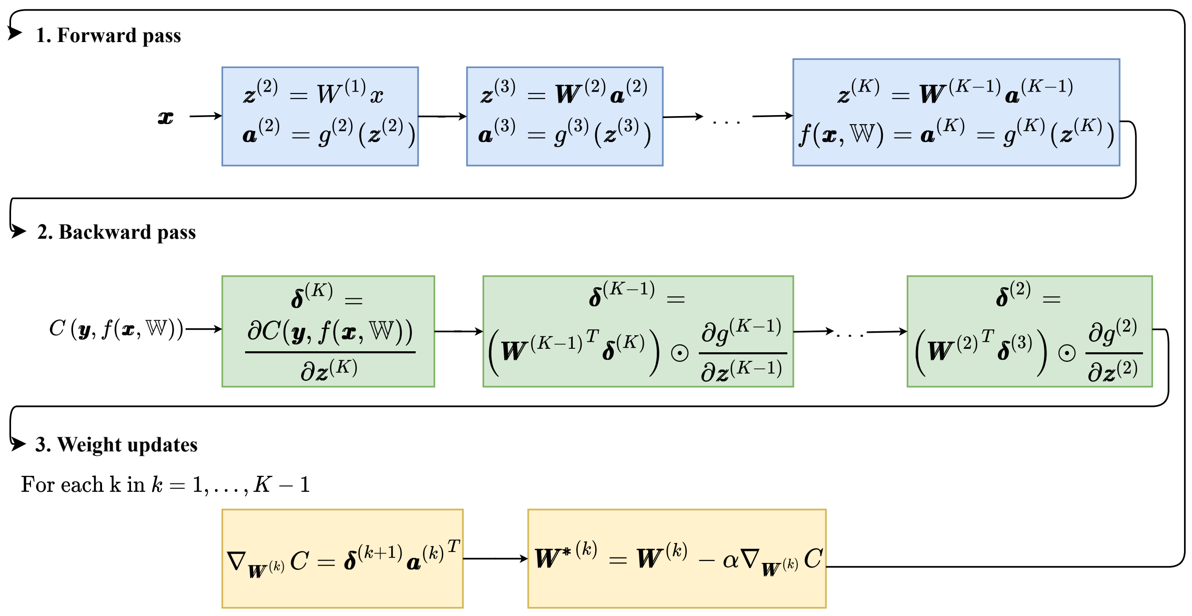

1. Forward pass
x
$z^{(2)} = W^{(1)}x$
$a^{(2)} = g^{(2)}(z^{(2)})$
$z^{(3)} = W^{(2)}a^{(2)}$
$a^{(3)} = g^{(3)}(z^{(3)})$
$\dots$
$z^{(K)} = W^{(K-1)}a^{(K-1)}$
$f(x, \mathbb{W}) = a^{(K)} = g^{(K)}(z^{(K)})$
2. Backward pass
$C(y, f(x, \mathbb{W}))$
$\delta^{(K)} = \frac{\partial C(y, f(x, \mathbb{W}))}{\partial z^{(K)}}$
$\delta^{(K-1)} = \left(W^{(K-1)T}\delta^{(K)}\right) \odot \frac{\partial g^{(K-1)}}{\partial z^{(K-1)}}$
$\dots$
$\delta^{(2)} = \left(W^{(2)T}\delta^{(3)}\right) \odot \frac{\partial g^{(2)}}{\partial z^{(2)}}$
3. Weight updates
For each k in $k = 1, \dots, K-1$
$\nabla_{W^{(k)}} C = \delta^{(k+1)} a^{(k)T}$
$W^{*(k)} = W^{(k)} - \alpha \nabla_{W^{(k)}} C$

Figure 3.4: Flow of the training process.

We can visualize the whole training process as three phases. In Figure 3.4, we see that an input $\boldsymbol{x}$ is passed through the network, and we can call this the forward pass. This phase is just a calculation of the function $f(\boldsymbol{x}, \mathbb{W})$, but we also save the values for each $\boldsymbol{z}$ and $\boldsymbol{a}$, which will be used in the two next phases. The second phase is the backward pass, where we pass the cost function C backward through the network save the values for each $\boldsymbol{\delta}$ along the way, which is the backpropagation algorithm. The last phase is the weight updates, where the gradient of the cost function with respect to the weights is calculated using the saved values for $\boldsymbol{\delta}$ and $\boldsymbol{a}$. When the weights are updated, we return to phase one (the forward pass), where we pass a new input $\boldsymbol{x}$ through the network with the updated weights. This process is repeated until each input $\boldsymbol{x}$ has been passed through the network at least once, but usually several times.

At this point, we have only covered the case where we pass one single pair $(\boldsymbol{x}, \boldsymbol{y})$ through the network before the weights are updated. It is also possible to pass multiple pairs (or the whole training dataset) through the network before each calculation of the cost function and weight update. If we have a batch of N samples that we pass through the network before each weight update, we can express the cost function (for cross-entropy) as

$$C(\boldsymbol{Y}, f(\boldsymbol{X}, \mathbb{W})) = \frac{1}{N}\sum_{i=1}^{N} C(\boldsymbol{y}_i, f(\boldsymbol{x}_i, \mathbb{W})) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(a_{i,j}^{(K)}), \tag{3.16}$$

where $\boldsymbol{Y}$ and $\boldsymbol{X}$ are matrices containing the N vectors $\boldsymbol{y}_i$ and $\boldsymbol{x}_i$, respectively. The gradient of this with respect to the weights becomes

$$\nabla_{\boldsymbol{W}^{(k)}} C = \frac{\partial C(\boldsymbol{Y}, f(\boldsymbol{X}, \mathbb{W}))}{\partial \boldsymbol{W}^{(k)}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial C\left(\boldsymbol{y}_i, f(\boldsymbol{x}_i, \mathbb{W})\right)}{\partial \boldsymbol{W}^{(k)}}. \tag{3.17}$$

In this case, the cost function and gradient for each $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ still has to be calculated individually before we can calculate the sums in (3.16) and (3.17). The difference is that we can do these individual calculations simultaneously, since the weights are the same for every $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$. Thus, if we have the computing resources, we can calculate $C(\boldsymbol{y}_i, f(\boldsymbol{x}_i, \mathbb{W}))$ and $\partial C\left(\boldsymbol{y}_i, f(\boldsymbol{x}_i, \mathbb{W})\right)/\partial \boldsymbol{W}^{(k)}$ for multiple samples at the same time.

3.1.3. Some important terms and methods

Before we go on to describe two other types of neural networks (convolutional and recurrent), we give a brief explanation of some terms and methods that have been or will be used later.

Training, validation and test set

Before a model is trained, we split our dataset in a training, validation, and test set. Only the training data is used to train the model, that is, updating the weights. The validation set is also used during training, but not to update weights. We use the validation data to find the optimal model architecture and hyperparameters and to give us information on whether the model is overfitting. After all training is finished, we use the test data to evaluate the performance of the model. No further changes to the model should be done after this step.

Batch size and number of epochs

Batch size is the number of samples that is passed through the model before the cost function is calculated, and weights are updated. The two ends of the extreme, are a batch size of one and batch size of the whole training dataset. When batch size is one, weights will be updated after each input is passed through the network. In the case with a batch size of the whole training dataset, the weights will only be updated one time per epoch. Usually, we use a batch size in the range $32 - 512$, as larger batch sizes tend to decrease the performance of the model (Keskar et al., 2016).

In an epoch, all the training data has been passed through the network once. Thus, the number of epochs is the number of times the model will see each sample during training. Usually, we let the model train on the same data several times, i.e., the number of epochs is greater than one. If the batch size is $1/Q$ of the data and we let epochs be equal to R, the weights will be updated $Q \cdot R$ times.

Learning rate

As we have seen earlier, a learning rate α is used in gradient descent (3.15). This rate is essentially how much we want the weights to be updated for each batch. We recall that the objective during training is to minimize the cost function by updating the weights, and this process happens in steps. The magnitude of these steps depends on the learning rate. Before we start the training, the model is initialized with random weights, and during training, the weights are incrementally updated to move closer to a minimum of the cost function. If the learning rate is large, training might be faster, but we risk taking steps towards the minimum of the cost function that are too big, such that we "miss" it. On the other hand, if the learning rate is very small, training might take too long.

The learning rate is a hyperparameter, so the optimal value will vary with different models and datasets, but is usually between 0.1 and 0.0001.

Overfitting

Overfitting means that the accuracy of the model predictions is significantly higher for the training data than for the validation data. It occurs when the model becomes too "good" at learning the features in the training data, but fails to generalize to data not used to train the model. The smaller the training set is, the more likely it is for overfitting to occur.

There are several strategies to avoid overfitting. One is to reduce the complexity of the model. In neural networks, this can be reducing the number of layers or nodes. Another approach is to use a regularization method, such as parameter norm penalties, dropout, or early stopping. Goodfellow et al. (2016)[p. 228] defines regularization as “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error”.

Parameter norm penalties

We follow Goodfellow et al. (2016) for a description of parameter norm penalties. In this type of regularization, we add a parameter norm penalty $\Omega(\mathbb{W})$ to the cost function, so that we now want to minimize

$$\widetilde{C}(\boldsymbol{Y}, \boldsymbol{X}, \mathbb{W}) = C(\boldsymbol{Y}, f(\boldsymbol{X}, \mathbb{W})) + \lambda\Omega(\mathbb{W}),$$

where $\lambda \geq 0$ is a hyperparameter that controls how much regularization is done (larger $\lambda =$ more regularization), and can be different for each layer in the neural network.

The most common choice for $\Omega(\mathbb{W})$ is the L^2 parameter norm penalty, which is often referred to as ridge regression;

$$\Omega(\mathbb{W}) = \frac{1}{2}\left\|\mathbb{W}\right\|_2^2 = \frac{1}{2}\sum_{k=1}^{K-1}\left\|\boldsymbol{W}^{(k)}\right\|_2^2 = \frac{1}{2}\sum_{k=1}^{K-1}\sum_i\sum_j \left(w_{ij}^{(k)}\right)^2.$$

Thus, the cost function with L^2 regularization becomes

$$\widetilde{C}(\boldsymbol{Y},\boldsymbol{X},\mathbb{W}) = C(\boldsymbol{Y}, f(\boldsymbol{X},\mathbb{W})) + \frac{\lambda}{2}\sum_{k=1}^{K-1}\sum_i\sum_j \left(w_{ij}^{(k)}\right)^2.$$

Taking the partial derivative with respect to each weight, this becomes

$$\frac{\partial\widetilde{C}(\boldsymbol{Y},\boldsymbol{X},\mathbb{W})}{\partial w_{ij}^{(k)}} = \frac{\partial C(\boldsymbol{Y}, f(\boldsymbol{X},\mathbb{W}))}{\partial w_{ij}^{(k)}} + \lambda w_{ij}^{(k)}.$$

It follows from this that in terms of each weight matrix $\boldsymbol{W}^{(k)}$, we can write the derivative as

$$\nabla_{\boldsymbol{W}^{(k)}}\widetilde{C} = \frac{\partial\widetilde{C}(\boldsymbol{Y},\boldsymbol{X},\mathbb{W})}{\partial\boldsymbol{W}^{(k)}} = \nabla_{\boldsymbol{W}^{(k)}}C + \lambda\boldsymbol{W}^{(k)}.$$

The gradient update for each weight matrix is

$$\boldsymbol{W}^{*(k)} = \boldsymbol{W}^{(k)} - \alpha\left(\nabla_{\boldsymbol{W}^{(k)}}C + \lambda\boldsymbol{W}^{(k)}\right).$$

We can also write this as

$$\boldsymbol{W}^{*(k)} = \underbrace{\boldsymbol{W}^{(k)} - \alpha\nabla_{\boldsymbol{W}^{(k)}}C}_{\text{regular gradient update}} - \alpha\lambda\boldsymbol{W}^{(k)}.$$

Thus, compared to the regular gradient update in equation (3.15), L^2 regularization will shrink the weights by an extra factor of $\alpha\lambda$ in every gradient update.

Another form of parameter norm penalty is the L^1 norm, also known as Lasso. In this case, we have

$$\Omega(\mathbb{W}) = \left\|\mathbb{W}\right\|_1 = \sum_{k=1}^{K-1}\left\|\boldsymbol{W}^{(k)}\right\|_1 = \sum_{k=1}^{K-1}\sum_i\sum_j |w_{ij}^{(k)}|,$$

$$\widetilde{C}(\boldsymbol{Y},\boldsymbol{X},\mathbb{W}) = C(\boldsymbol{Y}, f(\boldsymbol{X},\mathbb{W})) + \lambda\sum_{k=1}^{K-1}\sum_i\sum_j |w_{ij}^{(k)}|.$$

Taking the partial derivative with respect to each weight, this becomes

$$\frac{\partial\widetilde{C}(\boldsymbol{Y},\boldsymbol{X},\mathbb{W})}{\partial w_{ij}^{(k)}} = \frac{\partial C(\boldsymbol{Y}, f(\boldsymbol{X},\mathbb{W}))}{\partial w_{ij}^{(k)}} + \lambda\ \mathrm{sgn}(w_{ij}^{(k)}),$$

where sgn is the signum function $\mathrm{sgn}(w) = w/|w|$. In terms of each weight matrix $\boldsymbol{W}^{(k)}$, we can write

$$\nabla_{\boldsymbol{W}^{(k)}}\widetilde{C} = \frac{\partial \widetilde{C}(\boldsymbol{Y}, \boldsymbol{X}, \mathbb{W})}{\partial \boldsymbol{W}^{(k)}} = \nabla_{\boldsymbol{W}^{(k)}} C + \lambda \ \mathrm{sgn}(\boldsymbol{W}^{(k)}).$$

Here we define $\mathrm{sign}(\boldsymbol{W}^{(k)})$ as the signum function applied to each element $w_{ij}^{(k)}$ in $\boldsymbol{W}^{(k)}$.

The gradient update for each weight matrix is thus

$$\boldsymbol{W}^{*(k)} = \boldsymbol{W}^{(k)} - \alpha \left(\nabla_{\boldsymbol{W}^{(k)}} C + \lambda \ \mathrm{sgn}(\boldsymbol{W}^{(k)}) \right).$$

We can also write this as

$$\boldsymbol{W}^{*(k)} = \underbrace{\boldsymbol{W}^{(k)} - \alpha \nabla_{\boldsymbol{W}^{(k)}} C}_{\text{regular gradient update}} - \alpha\lambda \ \mathrm{sgn}(\boldsymbol{W}^{(k)}).$$

We observe that for a positive weight, we will subtract the constant $\alpha\lambda$, and for a negative weight, we will add $\alpha\lambda$. So the regularization does not depend on the size of each weight (as in L^2 regularization), but only on the sign of each weight. Another essential difference between L^1 and L^2 regularization is that L^1 move the weights to zero, and thus give more sparse weight matrices.

It is also possible to combine L_1 and L_2 regularization, which is called elastic net, proposed by Zou and Hastie (2005).

Dropout

The idea of dropout is to randomly drop a fraction of the nodes during training, and this regularization method was first described by Srivastava et al. (2014). For each training batch, a set of nodes is removed along with their connections, which results in one subnetwork or thinned network per training batch. We thus train an ensemble of subnetworks or thinned networks. At validation and testing, we use all the nodes but multiply each weight by the probability of keeping the node that the weight goes out from. This gives an approximation to averaging the predictions of the subnetworks.

Dropout can be viewed as a way to train many models with different architectures (using dropout in the hidden layers) on different training datasets (dropout in the input layer). Dropout can be applied to all layers except the output layer.

If the probability to drop a node in layer k is q, the forward pass equations can be written as

$$\begin{aligned} r_l^{(k)} &\sim \mathrm{Bernoulli}(1-q), \\ \boldsymbol{r}^{(k)} &= [\, r_1^{(k)}, r_2^{(k)}, \dots, r_{n_k}^{(k)} \,]^T, \end{aligned}$$

and

$$\begin{aligned}\tilde{\boldsymbol{a}}^{(k)} &= \boldsymbol{a}^{(k)} \odot \boldsymbol{r}^{(k)},\\ \boldsymbol{z}^{(k+1)} &= \boldsymbol{W}^{(k)}\tilde{\boldsymbol{a}}^{(k)},\\ \boldsymbol{a}^{(k+1)} &= g^{(k)}\big(\boldsymbol{z}^{(k+1)}\big).\end{aligned}$$

Thus, $\boldsymbol{r}^{(k)}$ is a vector of independent Bernoulli random variables, and each element has probability q of being 0, and $(1-q)$ of being 1. When taking the element-wise product between $\boldsymbol{a}^{(k)}$ and $\boldsymbol{r}^{(k)}$, some of the elements in $\tilde{\boldsymbol{a}}^{(k)}$ will become zero. These are the nodes that we drop. The weights in $\boldsymbol{W}^{(k)}$ that are connected to $\tilde{\boldsymbol{a}}^{(k)}$ will also become zero.

When we use the validation and test data, all nodes are used again, but weights are scaled according to the probability of keeping a node. The forward pass at validation and test time for layer k, will be

$$\begin{aligned}\boldsymbol{z}^{(k+1)} &= (1-q)\,\boldsymbol{W}^{(k)}\boldsymbol{a}^{(k)},\\ \boldsymbol{a}^{(k+1)} &= g^{(k+1)}\big(\boldsymbol{z}^{(k+1)}\big).\end{aligned}$$

Thus, the output at validation and test time is equal to the expected output at training time.

Early stopping

During training, we often see that while the cost function evaluated for the training data will continue to decrease for each epoch, the cost function evaluated for the validation data will stop decreasing or start increasing. We can avoid this by using early stopping. Early stopping will stop the training when the value for the cost function evaluated for the validation set has not improved for n epochs, and restore the weights back to the epoch with the lowest value for the cost function. This is illustrated in Figure 3.5.

The value of n is the number of epochs to wait before the early stop and is called the patience. The patience should not be too small, to allow for some fluctuation to occur. Using early stopping requires that the weights of the model are saved each time the validation cost improves.

When training neural networks, early stopping is almost always used, due to its simplicity and effectiveness. It is also frequently used in combination with other regularization strategies such as parameter norm penalties and dropout.

Variants of gradient descent

There are many different optimizers that are variants of gradient descent, such as Adagrad (Duchi et al., 2011) and RMSProp (Tieleman and Hinton, 2012). We will focus on the Adam optimizer,

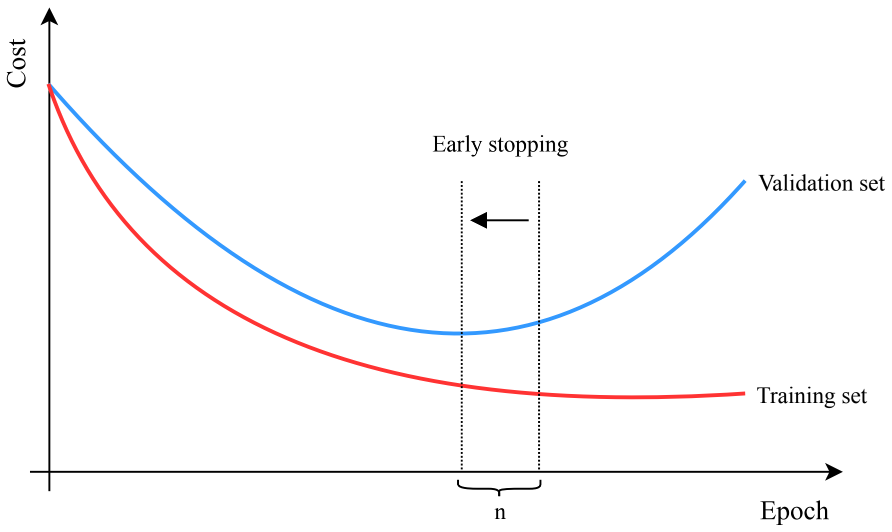


Figure 3.5: Early stopping with patience of n.

proposed by Kingma and Ba (2014), which combines Adagrad and RMSProp. This optimizer has been shown to yield good results in practice, and is one of the most popular optimizers in deep learning. Adam is an algorithm that uses first and second moment estimates of the gradient to update the weights. The moments are estimated by exponential moving averages and then bias corrected. The algorithm assigns individually adapted "learning rates" for each weight.

Let

$$\nabla_k C_t = \nabla_{\boldsymbol{W}^{(k)}} C(\boldsymbol{Y}, f(\boldsymbol{X}, \boldsymbol{W}))\Big|_{\boldsymbol{Y}=\boldsymbol{Y}_{(t)},\boldsymbol{X}=\boldsymbol{X}_{(t)},\boldsymbol{W}=\boldsymbol{W}_{(t-1)}},$$

where $\boldsymbol{Y}_{(t)}$ and $\boldsymbol{X}_{(t)}$ are matrices containing the vectors for the inputs and targets in batch t, and $\boldsymbol{W}_{(t-1)}$ is the updated weights from the previous batch.

We initialize the first and second moment estimator matrices as $\boldsymbol{M}_0^{(k)} = \boldsymbol{0}$ and $\boldsymbol{V}_0^{(k)} = \boldsymbol{0}$, for $k = 1, \dots, K-1$. Also we let $\beta_1, \beta_2 \in [0,1)$ and $\boldsymbol{\epsilon} = \epsilon\boldsymbol{1}$ where $\epsilon \in \mathbb{R}$. As suggested by Kingma and Ba (2014), good default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

For the t-th batch, the algorithm is given by (all operations on matrices are element-wise):

$$\begin{aligned}
\boldsymbol{M}_{(t)}^{(k)} &= \beta_1 \boldsymbol{M}_{(t-1)}^{(k)} + (1-\beta_1)\nabla_k C_t, \\
\boldsymbol{V}_{(t)}^{(k)} &= \beta_2 \boldsymbol{V}_{(t-1)}^{(k)} + (1-\beta_2)[\nabla_k C_t]^2, \\
\widehat{\boldsymbol{M}}_{(t)}^{(k)} &= \frac{1}{1-\beta_1^t}\boldsymbol{M}_{(t)}^{(k)}, \\
\widehat{\boldsymbol{V}}_{(t)}^{(k)} &= \frac{1}{1-\beta_2^t}\boldsymbol{V}_{(t)}^{(k)}, \\
\boldsymbol{W}*_{(t)}^{(k)} &= \boldsymbol{W}_{(t-1)}^{(k)} - \alpha\frac{\widehat{\boldsymbol{M}}_{(t)}^{(k)}}{\sqrt{\widehat{\boldsymbol{V}}_{(t)}^{(k)}} + \boldsymbol{\epsilon}}, \qquad k = 1, \dots, K-1.
\end{aligned} \tag{3.18}$$

Note that β_1^t and β_2^t denote exponentiation with t and that $[\nabla_k C_t]^2 = \nabla_k C_t \odot \nabla_k C_t$.

After t batches, we have that the bias corrected first moment estimator for the gradient is given by

$$\widehat{\boldsymbol{M}}^{(k)}_{(t)} = \frac{1-\beta_1}{1-\beta_1^t}\sum_{i=1}^{t}\beta_1^{t-i}\nabla_k C_i. \tag{3.19}$$

Taking the excepted value of this estimator we get

$$\mathrm{E}[\widehat{\boldsymbol{M}}^{(k)}_{(t)}] = \frac{1-\beta_1}{1-\beta_1^t}\sum_{i=1}^{t}\beta_1^{t-i}\,\mathrm{E}[\nabla_k C_i].$$

If we assume that $\mathrm{E}[\nabla_k C_i] = \mathrm{E}[\nabla_k C_t]$, we get that

$$\mathrm{E}[\widehat{\boldsymbol{M}}^{(k)}_{(t)}] = E[\nabla_k C_t]\frac{1-\beta_1}{1-\beta_1^t}\sum_{i=0}^{t}\beta_1^{t-i} = \mathrm{E}[\nabla_k C_t].$$

Hence we have an unbiased estimator for the first moment of the gradient at batch t. The same can be shown for the bias corrected second moment estimator $\widehat{\boldsymbol{V}}^{(k)}_{(t)}$.

From equation (3.19), we observe that the unbiased first moment estimator for the gradient at batch t, gets a greater contribution from the last batch gradients than the first ones. Similarly for the unbiased second moment estimator for the gradient at batch t. Thus, when we update the new weights, they will be more influenced by the last gradients, rather than equally contributing.

If $\epsilon = 0$, the magnitude and direction of the step at batch t is $\Delta_t = \alpha\,\widehat{\boldsymbol{M}}_{(t)}/\sqrt{\widehat{\boldsymbol{V}}_{(t)}}$ and this becomes smaller as the ratio of the first and second moment estimates decreases. That is, when the uncertainty about the true direction of the gradient increases ($\widehat{\boldsymbol{V}}_{(t)}$ increases) we take smaller effective steps. This is a desirable property, as we want to take smaller steps when we move close to the optimal values for the weights.

3.2. Convolutional neural networks

Fully connected neural networks treat the elements of each input vector the same, meaning that the order of the elements in each input does not matter. That is, one could permute the elements of the input vector and get the same result. While this makes sense if the input data features are age, weight, ect., it is clearly not well suited for input data like images and time series since they are spatially or temporally correlated. Convolutional neural networks (CNNs) were specially developed for this type of data. A convolutional neural network is a neural network with at least one convolutional layer, but usually several (Goodfellow et al., 2016, ch. 9). A convolutional layer differs from a standard fully connected layer in that it uses an operation called convolution to extract particular features in the data. This is why it is so popular and effective, especially for image data as different shapes and edges can be recognized by the network.

Before we look at the structure of a convolutional neural network, we will briefly cover convolution. In mathematics a convolution of two sequences, x and w, is given by

$$z[i] = (x \star w)[i] = \sum_{r=-\infty}^{\infty} x[r]w[i-r],$$

where x and w are two discrete functions, and i is some integer. This operation is commutative, that is $(x \star w) = (w \star x)$.

In neural network implementations we actually use cross-correlation, but usually call it convolution as the two operations are similar and would yield the same set of learned parameters just in a "flipped" orientation with convolution. Cross-correlation of the discrete functions x and w is given by

$$z[i] = (x * w)[i] = \sum_{r=-\infty}^{\infty} x[i+r]w[r].$$

Let us look at the convolution of $\boldsymbol{x} = \{x_t, \quad t = 1, 2, \ldots, T\}$ and $\boldsymbol{w} = \{w_i, \quad i = 1, \ldots, s\}$. We will refer to $\boldsymbol{x}$ as the input, $\boldsymbol{w}$ as the filter and the length of $\boldsymbol{w}$ as the kernel size. Since $\boldsymbol{x}$ is a finite length vector and the filter is also of finite size, we get a finite summation:

$$z_i = \sum_{r=1}^{s} x_{i-1+r} w_j, \quad \text{for } i = 1, \ldots, T-s+1.$$

The minus one in the subscript of x in the sum is due to the fact that we let the index of x_i start at one. We see that this operation is just sliding a filter across the input, and that is why cross-correlation is sometimes referred to as the sliding dot product. The output z_i is the i-th element of the output vector $\boldsymbol{z}$, and is called the feature map. We notice that the dimension of the feature map is smaller than the input. If the input vector has T elements and kernel size of the filter is s, then the feature map $\boldsymbol{z}$ will have $T-s+1$ elements. See Figure 3.6 for an illustration of this operation.

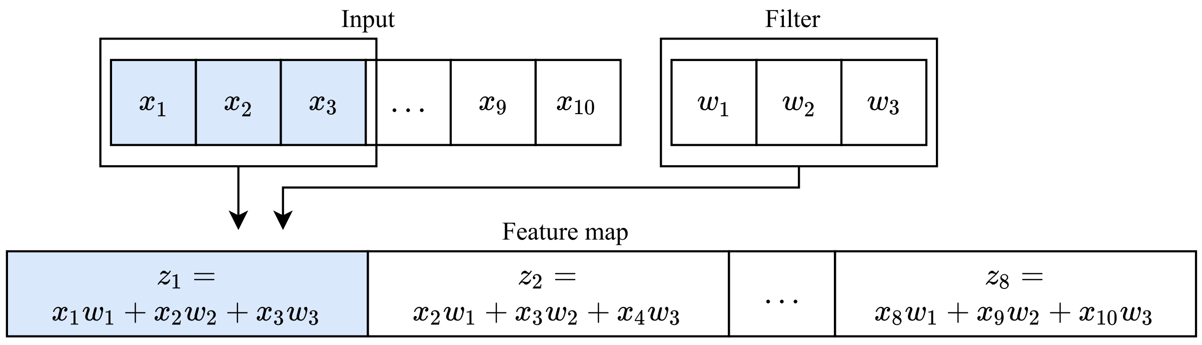

Input
x_1
x_2
x_3
$\ldots$
x_9
x_{10}
Filter
w_1
w_2
w_3
Feature map
$z_1 = x_1w_1 + x_2w_2 + x_3w_3$
$z_2 = x_2w_1 + x_3w_2 + x_4w_3$
$\ldots$
$z_8 = x_8w_1 + x_9w_2 + x_{10}w_3$

Figure 3.6: Cross-correlation of input vector $\boldsymbol{x}$ with length 10 and filter $\boldsymbol{w}$ with kernel size 3.

Structure

Now that we understand convolution and cross-correlation, we can look at the structure and feed-forward equations of a convolutional neural network. From this point, we will refer to the cross-correlation operation as convolution. We will also only cover 1D convolutional neural networks, but 2D and higher dimensional convolutional neural networks are very similar. We follow Kiranyaz et al. (2019) in this part.

In CNNs each convolutional layer will have several different filters with the same kernel size. We want to learn different features of the data and do this by training the model to learn the optimal values for the elements of each filter (the weights). If we have n filters, we will get n feature maps as output. For the transition from the input layer L_1 to the first convolutional layer L_2 with n_1 filters having kernel size s_1, we can write the equation for the i-th element in the j-th feature map as

$$z_{i,j}^{(2)} = b_j^{(1)} + \sum_{r=1}^{s_1} x_{i-1+r}\, w_{j,r}^{(1)} \quad \text{for } i = 1, \dots, m_1 \text{ and } j = 1, \dots, n_1, \tag{3.20}$$

where $b_j^{(1)}$ is the bias weight for the j-th filter, and $m_1 = T - s_1 + 1$. The total number of weights, including the bias weights, between these layers is $n_1(s_1 + 1)$. See Figure 3.7 for an illustration of the convolution in equation (3.20).

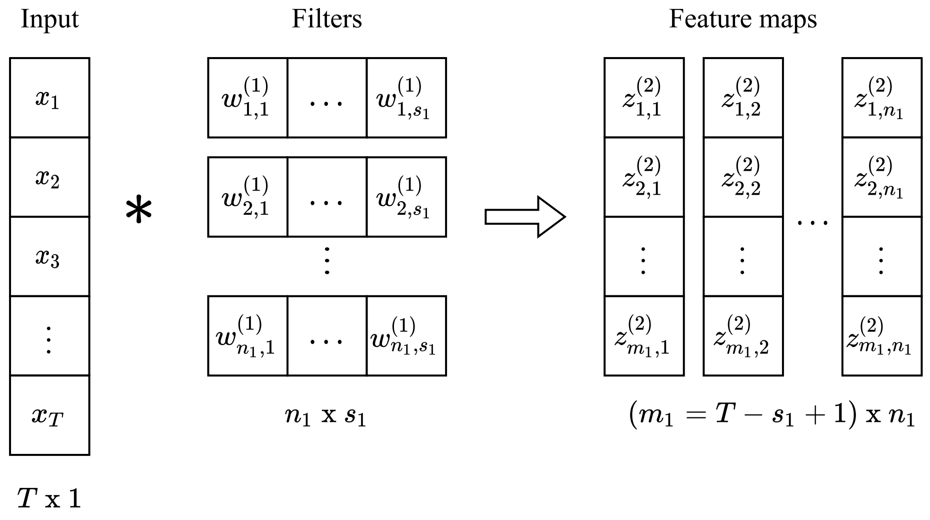

Input
Filters
Feature maps
x_1
x_2
x_3
x_T
T x 1
$*$
$w_{1,1}^{(1)}$ … $w_{1,s_1}^{(1)}$
$w_{2,1}^{(1)}$ … $w_{2,s_1}^{(1)}$
$w_{n_1,1}^{(1)}$ … $w_{n_1,s_1}^{(1)}$
n_1 x s_1
$z_{1,1}^{(2)}$ $z_{1,2}^{(2)}$ $z_{1,n_1}^{(2)}$
$z_{2,1}^{(2)}$ $z_{2,2}^{(2)}$ $z_{2,n_1}^{(2)}$
$z_{m_1,1}^{(2)}$ $z_{m_1,2}^{(2)}$ $z_{m_1,n_1}^{(2)}$
$(m_1 = T - s_1 + 1)$ x n_1

Figure 3.7: Transition from the input to the first convolutional layer (without bias weights).

Just like in fully connected neural networks we perform an activation function on each element $z_{i,j}$ so that

$$a_{i,j}^{(2)} = g^{(2)}\big(z_{i,j}^{(2)}\big) \quad \text{for } i = 1, \dots, m_1 \text{ and } j = 1, \dots, n_1,$$

where $g^{(2)}$ is the activation function for the first convolutional layer. As we can see, the output (and thus the input for the next layer) from a convolutional layer consists of several feature maps. We will call the number of feature maps for channels. Thus, the number of channels in a convolutional layer that comes after a convolutional layer corresponds with the number of feature maps in this previous layer. Each filter will have the same number of channels as the input. We can also extend this to the first layer, by viewing the input as having one channel.

For the general transition from one convolutional layer L_{l-1} to another, L_l, with n_{l-1} filters of kernel size s_{l-1} and n_{l-2} channels per filter, we have the equations

$$z_{i,j}^{(l)} = b_j^{(l-1)} + \sum_{k=1}^{n_{l-2}} \sum_{r=1}^{s_{l-1}} a_{i-1+r,k}^{(l-1)}\, w_{j,r,k}^{(l-1)}, \quad \text{for } i = 1, \dots, m_{l-1} \text{ and } j = 1, \dots, n_{l-1}, \tag{3.21}$$

and

$$a_{i,j}^{(l)} = g^{(l)}\big(z_{i,j}^{(l)}\big) \quad \text{for } i = 1, \dots, m_{l-1}, \text{ and } j = 1, \dots, n_{l-1}.$$

We notice the additional index k that represents the channels. The total number of weights, including the bias weights, between these layers is $n_{l-1}\,(s_{l-1}\,n_{l-2} + 1)$. See Figure 3.8 for an illustration of the transition between two convolutional layers.

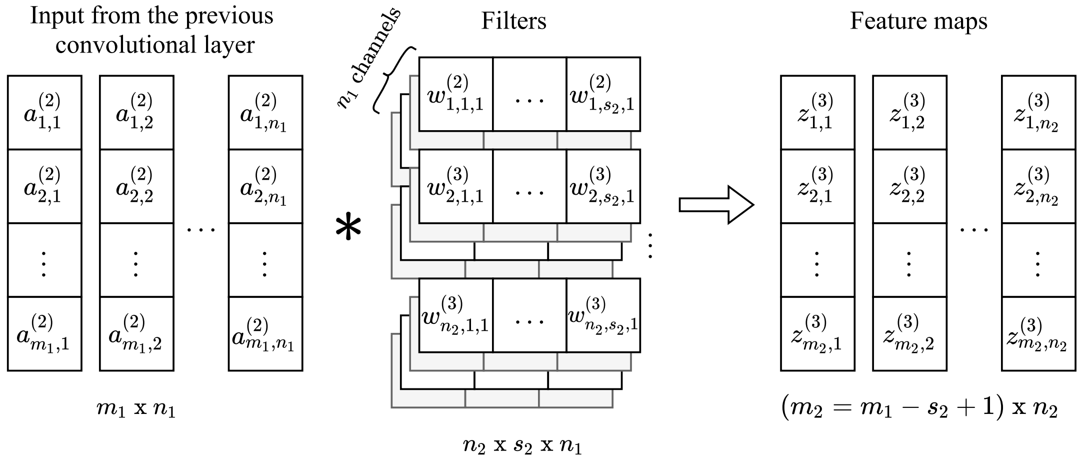

Input from the previous convolutional layer
Filters
Feature maps
n_1 channels
$a_{1,1}^{(2)}$ $a_{1,2}^{(2)}$ $a_{1,n_1}^{(2)}$
$a_{2,1}^{(2)}$ $a_{2,2}^{(2)}$ $a_{2,n_1}^{(2)}$
$a_{m_1,1}^{(2)}$ $a_{m_1,2}^{(2)}$ $a_{m_1,n_1}^{(2)}$
$w_{1,1,1}^{(2)}$ $w_{1,s_2,1}^{(2)}$
$w_{2,1,1}^{(3)}$ $w_{2,s_2,1}^{(3)}$
$w_{n_2,1,1}^{(3)}$ $w_{n_2,s_2,1}^{(3)}$
$z_{1,1}^{(3)}$ $z_{1,2}^{(3)}$ $z_{1,n_2}^{(3)}$
$z_{2,1}^{(3)}$ $z_{2,2}^{(3)}$ $z_{2,n_2}^{(3)}$
$z_{m_2,1}^{(3)}$ $z_{m_2,2}^{(3)}$ $z_{m_2,n_2}^{(3)}$
m_1 x n_1
n_2 x s_2 x n_1
$(m_2 = m_1 - s_2 + 1)$ x n_2

Figure 3.8: Transition from the first to the second convolutional layer (without bias weights).

While there are many similarities between CNNs and fully connected neural networks, some important differences are weight sharing and sparse connectivity. In a fully connected layer, each weight is only used one time between an input node and an output node. This is not true for a CNN layer, because here every weight in each filter is used in every position of the input. This means that every element in the j-th feature map share the same weights from the j-th filter. We see this most easily from equation (3.20), and we call this property weight sharing. The other important difference is sparse connectivity. This is referring to the fact that in a CNN layer, each element in the input is not connected to every element in the output. Again we can see this from equation (3.20), where the i-th element in each feature map is only connected to the elements $x_i, \ldots, x_{i-1+s_1}$ in the input.

Stride and zero padding

Stride is how many steps the filter shifts across the input for the calculation of each feature map $z_{i,j}$. In equation (3.20) and (3.21) we have used a stride of 1. It is also possible to use a stride of more than 1, but this might lead to problems for the boundary elements of the input. A solution to this is to control the dimension of the input by padding the input to the layer with zeros. This is called zero padding.

As previously stated, when using convolutional layers, the dimensions of the input will shrink with one element less than the kernel size for each layer. This can limit how deep (i.e. how many convolutional layers) the network can be, and the choice of kernel size. Thus, zero padding can also be used to control the size of the output maps of each layer, and thus we have more choices in the kernel size and number of convolutional layers in the network.

Pooling

Pooling is a type of layer that is often used in conjunction with convolutional layers. These are layers that produce some type of summary of the feature maps of a convolutional layer. We use pooling if we want the network to be less sensitive to the exact position of a feature in the data. Examples of pooling layers are average pooling and max pooling. As the names suggest, average pooling averages the elements in each feature map within a given window size, and max pooling gives the maximum value within a window. Figure 3.9 illustrates average and max pooling. Pooling layers can also help to reduce the complexity in the model, and thus reduce overfitting.

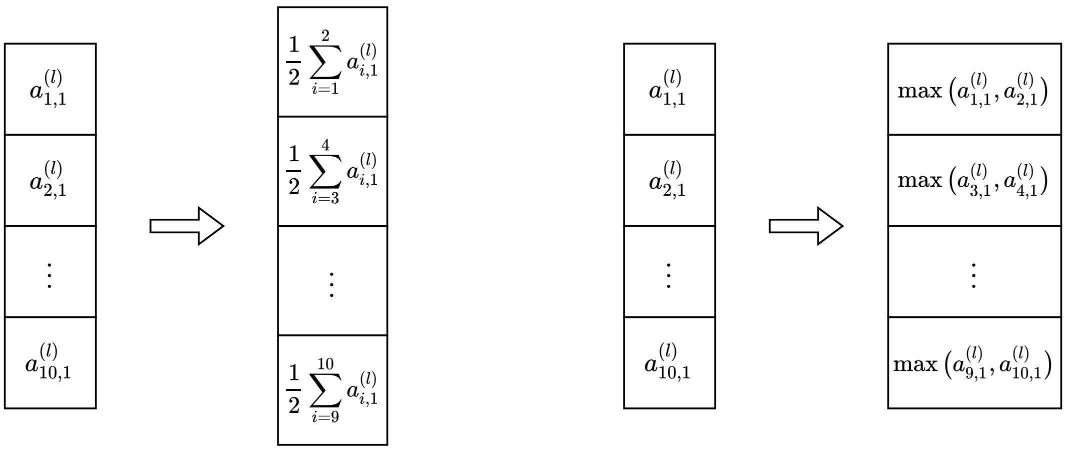

$a_{1,1}^{(l)}$
$a_{2,1}^{(l)}$
$\vdots$
$a_{10,1}^{(l)}$
$\frac{1}{2}\sum_{i=1}^{2} a_{i,1}^{(l)}$
$\frac{1}{2}\sum_{i=3}^{4} a_{i,1}^{(l)}$
$\vdots$
$\frac{1}{2}\sum_{i=9}^{10} a_{i,1}^{(l)}$
$a_{1,1}^{(l)}$
$a_{2,1}^{(l)}$
$\vdots$
$a_{10,1}^{(l)}$
$\max\left(a_{1,1}^{(l)}, a_{2,1}^{(l)}\right)$
$\max\left(a_{3,1}^{(l)}, a_{4,1}^{(l)}\right)$
$\vdots$
$\max\left(a_{9,1}^{(l)}, a_{10,1}^{(l)}\right)$

Figure 3.9: Two examples of pooling for a single feature map of length 10 and pooling window of 2. Left: Average pooling. Right: Max pooling.

Flattening

A CNN always has at least one fully connected layer as the output layer, as we need to either predict a single quantity (one fully connected node) or classify (several fully connected nodes). Since a fully connected layer only can take a single one-dimensional vector as input, we need to reshape the feature maps to a column vector. This is called flattening and is done between a convolutional layer and one or more fully connected layers. Flattening is just stacking each feature map into a single vector. This is shown in Figure 3.10.

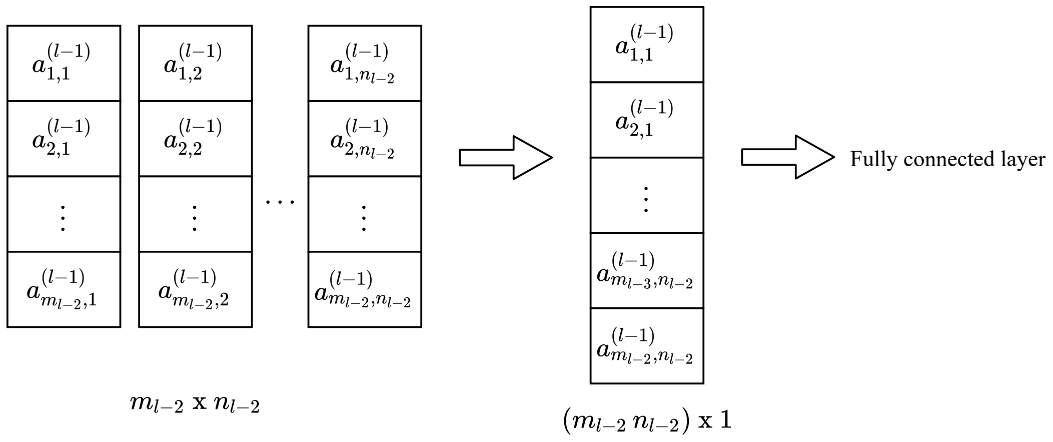


Figure 3.10: Flattening.

3.3. Recurrent neural networks

This section is based on Goodfellow et al. (2016) and Graves (2012). Recurrent neural networks (RNNs) are a type of neural network that is used on sequential data, like time series or language. What sets RNNs apart from fully connected neural networks is that weights are shared between time-steps in the whole sequence. This is somewhat similar to 1D CNNs for sequential data, but in convolution, weights are only shared between "patches" of each sequence.

Recurrent neural networks can be used in many different classification and regression problems, and each requires different architectures with respect to the input and output of the RNN layer. Examples of this are "one-to-many", where the input can be a single image and output is a sequence of words describing the image (image captioning). "Many-to-many" where the input can be a sequence of words in one language and the output is a sequence of words in another language (machine translation), and lastly "many-to-one" where we can have a sequence of words (like a review of a movie) as input and the output is a class representing the sentiment (sentiment analysis). Since we will be classifying time series, where the input is a sequence and output is a single class, we will focus on the "many-to-one" architecture, where many refer to the values in the time series, and one the classification value.

Structure

Let us consider a neural network with only one hidden recurrent layer. We have some number of time-series that we want to classify, and each of the time-series has an equal number of time-steps (length) T. Our network will have one single input node that will take in one value of each time-series at a time. As usual, let $\boldsymbol{x} = \{x_t, \quad t = 1, \dots, T\}$ be one of the time-series. Then we have:

$$z_{h,t}^{(2)} = b_h^{(1)} + u_h^{(1)}\, x_t + \sum_{i=1}^{H} w_{h,i}^{(1)}\, a_{i,t-1}^{(2)} \quad \text{for } t = 1, \dots, T \text{ and } h = 1, \dots, H, \tag{3.22}$$

where $z_{h,t}^{(2)}$ is the input to node h at time-step t, $a_{i,t-1}^{(2)}$ is the activation of node i at time-step $t-1$ and x_t is the value of the input to the recurrent hidden layer at time-step t. The $b_h^{(1)}$ is the bias for the h-th node, $u_h^{(1)}$ is the weight connecting the input x_t and node h and, $w_{h,i}^{(1)}$ is the weight connecting node i and h in time-steps $(t-1)$ and t. The weights remain the same for every time-step t of the sequence. Notice that for $t = 1$ we do not have any past time-steps to get values from so that $a_{i,0} = 0$, thus we get the equation

$$z_{h,1}^{(2)} = b_h^{(1)} + u_h^{(1)}\, x_1 \quad \text{for } h = 1, \dots, H.$$

As always we use an activation function

$$a_{h,t}^{(2)} = g\big(z_{h,t}^{(2)}\big) \quad \text{for } t = 1, \dots, T \text{ and } h = 1, \dots, H. \tag{3.23}$$

In RNN layers the tanh function is often used as activation.

At the last time-step $t = T$, the RNN layer will pass an $a_{h,T}$ for $h = 1, \dots, H$ to the fully connected output layer with M nodes, one for each class.

$$\begin{aligned} z_j^{(3)} &= b_j^{(2)} + \sum_{h=1}^{H} v_{j,h}\, a_{h,T}^{(2)}\,, \\ a_j^{(3)} &= \text{softmax}\big(z_j^{(3)}\big), \end{aligned} \qquad \text{for } j = 1, \dots, M, \tag{3.24}$$

where $v_{j,h}$ is the weight between the h-th node in the hidden layer and the j-th node in the output layer. This architecture is illustrated in Figure 3.11.

In Figure 3.11 we can observe that the recurrent neural network architecture for a single time-step t looks similar to the fully connected network. If we compare the feed forward equation for fully connected and recurrent networks without biases, we have the who equations:

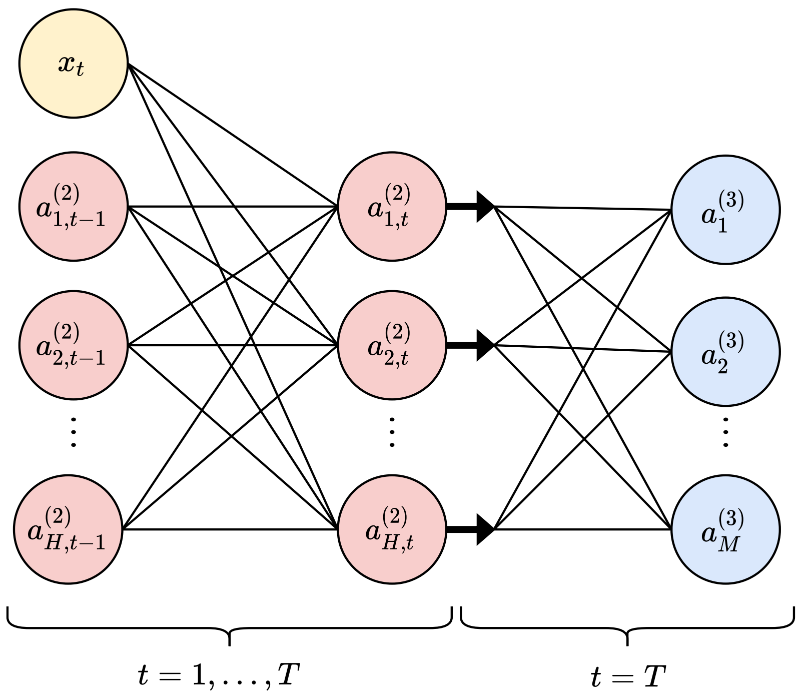


Figure 3.11: A recurrent neural network with one hidden layer. We see the hidden recurrent layer for the two successive time-steps, $t-1$ and t, in red, the input at time-step t, x_t, in yellow and the output layer in blue.

$$z^{(2)}_l = \sum_{t=1}^{T} w^{(1)}_{l,j} x_t$$

$$z^{(2)}_{h,t} = u^{(1)}_h \, x_t + \sum_{i=1}^{H} w^{(1)}_{h,i} \, a^{(2)}_{i,t-1}$$

The most obvious difference is that the inputs for the recurrent layer are x_t and $a^{(2)}_{1,t-1}, \dots, a^{(2)}_{H,t-1}$ and for the fully connected the inputs are $x_1, \dots, x_T$. Still, the RNN layer for each time-step t can be viewed as its own fully connected neural network, and over every time-step as T copies of a fully connected neural network executed in a chain.

The number of weights for a recurrent layer is $H + H + (H \cdot H)$ where H is the number of hidden nodes. The first H in the sum is the number of bias weights, and the second is the weights between each input and hidden nodes and $H \cdot H$ is the number of weights connecting the hidden nodes in different time-steps.

We can write the equations (3.22), (3.23) and (3.24) in vector notation as

$$\boldsymbol{z}^{(2)}_t = \boldsymbol{b}^{(1)} + \boldsymbol{u}^{(1)} \, x_t + \boldsymbol{W}^{(1)} \, \boldsymbol{a}^{(2)}_{t-1}, \quad \text{for } t = 1, \dots, T, \tag{3.25}$$

$$\boldsymbol{a}^{(2)}_t = g(\boldsymbol{z}^{(2)}_t), \quad \text{for } t = 1, \dots, T, \tag{3.26}$$

and

$$\begin{aligned} \boldsymbol{z}^{(3)} &= \boldsymbol{b}^{(2)} + \boldsymbol{V}\,\boldsymbol{a}_T^{(2)}, \\ \boldsymbol{a}^{(3)} &= \text{softmax}(\boldsymbol{z}^{(3)}). \end{aligned} \tag{3.27}$$

To better understand the recurrent neural network, we can "unfold" the network for all time-steps. An illustration of this is shown in Figure 3.12.

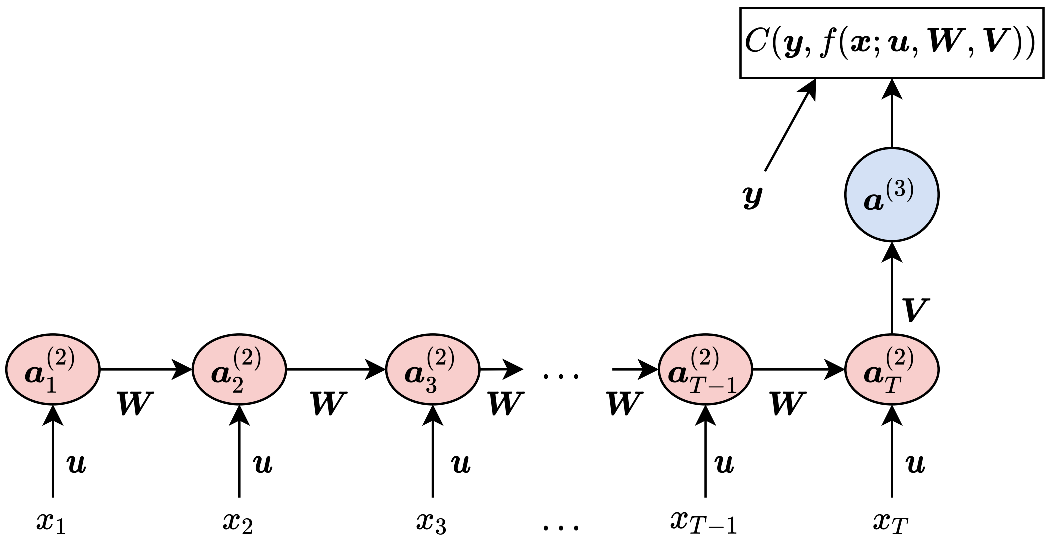


Figure 3.12: Unfolded graph of an RNN with one hidden layer. Each circle and ellipse represents a whole layer of nodes. The hidden recurrent layer is in red and fully connected output layer in blue. Adapted from figure 10.5 in Goodfellow et al. (2016).

At this point, we have only looked at RNNs with one hidden layer. It is, of course, possible to have multiple hidden recurrent layers. Let us say that we have an RNN consisting of a total of K layers, and hence $K-2$ hidden recurrent layers. We can then generalize equations (3.25) and (3.26) in the following way:

$$\begin{aligned} \boldsymbol{z}_t^{(k)} &= \boldsymbol{b}^{(k-1)} + \boldsymbol{U}^{(k-1)}\,\boldsymbol{a}_t^{(k-1)} + \boldsymbol{W}^{(k-1)}\,\boldsymbol{a}_{t-1}^{(k)}, \\ \boldsymbol{a}_t^{(k)} &= g(\boldsymbol{z}_t^{(k)}), \end{aligned} \quad \text{for } t = 1, \dots, T \text{ and } k = 2, \dots, K-1, \tag{3.28}$$

where $\boldsymbol{z}_t^{(k)}$ is the input to the nodes in the k-th layer at time-step t, and $\boldsymbol{a}_t^{(k)}$ is the activation of the nodes in layer k at time-step t. In this case we let $\boldsymbol{a}_t^{(1)} = x_t$. We notice that the weights between the outputs from one layer to the next now is written as a matrix $\boldsymbol{U}$. This is because, in exception of the first hidden layer, it will connect all the nodes in layer $(k-1)$ to the nodes in layer k. The weights $\boldsymbol{U}^{(1)}$ will still be a vector since it connects a single input to the nodes in the first hidden layer.

After the input layer and the $(K-2)$ hidden layers, the output from the $(K-1)$-th layer at time-step T is passed to the output layer. We have the following equations:

$$\begin{aligned} \boldsymbol{z}^{(K)} &= \boldsymbol{b}^{(K-1)} + \boldsymbol{V}\,\boldsymbol{a}_T^{(K-1)}, \\ \boldsymbol{a}^{(K)} &= \text{softmax}(\boldsymbol{z}^{(K)}). \end{aligned} \tag{3.29}$$

While recurrent neural networks have been shown to be very effective for a variety of tasks, there are some disadvantages in using RNNs. From equation (3.25) and (3.26) we notice that the activation $\boldsymbol{a}_t$ depends on $\boldsymbol{a}_{t-1}$ from the previous time-step in the same recurrent layer, i.e to calculate $\boldsymbol{a}_t$ we need to have already computed the values $\boldsymbol{a}_{t-1}, \dots, \boldsymbol{a}_1$ recursively. In other words, $\boldsymbol{a}_t$ in every hidden recurrent layer must be computed T times, one for each time-step. This computation can not be parallelized, i.e. done at the same time, since each time-step is dependent on the previous. This is a significant difference between recurrent networks and other types of neural networks, like fully connected or convolutional networks. In the two latter types, the activations of each layer can be computed in one step (in parallel), since the input $\boldsymbol{x} = \{x_t, \quad i = 1, \dots, T\}$ is passed to the network at the same time, and the activations are not dependent on each other in the same layer. We face the same issue in the backward pass (backpropagation) since the gradients must be computed backwards through time like it is illustrated in Figure 3.13.

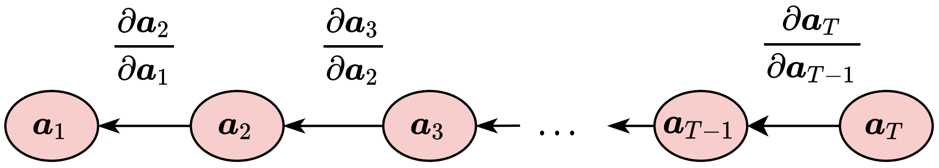

$\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{a}_1}$
$\frac{\partial \boldsymbol{a}_3}{\partial \boldsymbol{a}_2}$
$\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_{T-1}}$
$\boldsymbol{a}_1$
$\boldsymbol{a}_2$
$\boldsymbol{a}_3$
$\dots$
$\boldsymbol{a}_{T-1}$
$\boldsymbol{a}_T$

Figure 3.13: Gradients flow backwards through time.

To illustrate why that is, we first look at a recurrent neural network with one hidden layer and the partial derivative of the cost function C with respect to the weights in the recurrent layer, $\boldsymbol{W}$. Since C is a function of $\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_T$, and $\boldsymbol{W}$ is an independent variable, we use the multivariate chain rule and get

$$\begin{aligned} \frac{\partial C}{\partial \boldsymbol{W}} &= \frac{\partial C}{\partial \boldsymbol{a}_1}\frac{\partial \boldsymbol{a}_1}{\partial \boldsymbol{W}} + \frac{\partial C}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{W}} + \dots + \frac{\partial C}{\partial \boldsymbol{a}_T}\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{W}} \\ &= \frac{\partial C}{\partial \boldsymbol{a}_T}\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_1}\frac{\partial \boldsymbol{a}_1}{\partial \boldsymbol{W}} + \frac{\partial C}{\partial \boldsymbol{a}_T}\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{W}} + \dots + \frac{\partial C}{\partial \boldsymbol{a}_T}\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_T}\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{W}} \\ &= \sum_{t=1}^{T} \frac{\partial C}{\partial \boldsymbol{a}_T}\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_t}\frac{\partial \boldsymbol{a}_t}{\partial \boldsymbol{W}}. \end{aligned}$$

Both $\partial C/\partial \boldsymbol{a}_T$ and $\partial \boldsymbol{a}_t/\partial \boldsymbol{W}$ can be calculated without relying on other recurrent activations than $\boldsymbol{a}_T$ and $\boldsymbol{a}_t$ respectively, but $\partial \boldsymbol{a}_T/\partial \boldsymbol{a}_t$ is different:

$$\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_t} = \frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_{T-1}} \frac{\partial \boldsymbol{a}_{T-1}}{\partial \boldsymbol{a}_{T-2}} \cdots \frac{\partial \boldsymbol{a}_{t+1}}{\partial \boldsymbol{a}_t} = \prod_{T \geq \tau > t} \frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{a}_{\tau-1}}. \tag{3.30}$$

This type of backpropagation is called backpropagation through time (BPTT).

In addition to taking longer to train, recurrent neural networks also require more memory. This is because the values of $\boldsymbol{z}_t$ and $\boldsymbol{a}_t$ at each time-step $t = 1, \ldots, T$ in the forward pass must be stored, later to be used when calculating the values of the gradients. So compared with a fully connected layer, a recurrent layer needs to store T times more values for each node. The memory usage and time to train increases with the length of each training sample T. For large T's we might face problems, even for relatively few number of nodes and one or a few hidden layers, due to not having enough memory available on the device we use to train the network, or that the training simply takes too long.

With the architecture for a basic RNN that we have described above, we might also face a different problem; namely, *vanishing* or *exploding gradients*. As the names suggest, it means that the gradients that we need to update the weights will either vanish or explode. This can happen to any type of neural network, but will almost always happen to regular RNNs unless the number of time-steps T in our data is very small. To show why this occurs, we investigate the equation (3.30) further.

$$\frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{a}_{\tau-1}} = \frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{z}_\tau} \frac{\partial \boldsymbol{z}_\tau}{\partial \boldsymbol{a}_{\tau-1}}. \tag{3.31}$$

The two elements of this product are the Jacobian matrices

$$\frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{z}_\tau} = \begin{bmatrix} \frac{\partial a_{1,\tau}}{\partial z_{1,\tau}} & \frac{\partial a_{1,\tau}}{\partial z_{2,\tau}} & \cdots & \frac{\partial a_{1,\tau}}{\partial z_{H,\tau}} \\ \frac{\partial a_{2,\tau}}{\partial z_{1,\tau}} & \frac{\partial a_{2,\tau}}{\partial z_{2,\tau}} & \cdots & \frac{\partial a_{2,\tau}}{\partial z_{H,\tau}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_{H,\tau}}{\partial z_{1,\tau}} & \frac{\partial a_{H,\tau}}{\partial z_{2,\tau}} & \cdots & \frac{\partial a_{H,\tau}}{\partial z_{H,\tau}} \end{bmatrix} \tag{3.32}$$

and

$$\frac{\partial \boldsymbol{z}_\tau}{\partial \boldsymbol{a}_{\tau-1}} = \begin{bmatrix} \frac{\partial z_{1,\tau}}{\partial a_{1,\tau-1}} & \frac{\partial z_{1,\tau}}{\partial a_{2,\tau-1}} & \cdots & \frac{\partial z_{1,\tau}}{\partial a_{H,\tau-1}} \\ \frac{\partial z_{2,\tau}}{\partial a_{1,\tau-1}} & \frac{\partial z_{2,\tau}}{\partial a_{2,\tau-1}} & \cdots & \frac{\partial z_{2,\tau}}{\partial a_{H,\tau-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{H,\tau}}{\partial a_{1,\tau-1}} & \frac{\partial z_{H,\tau}}{\partial a_{2,\tau-1}} & \cdots & \frac{\partial z_{H,\tau}}{\partial a_{H,\tau-1}} \end{bmatrix}. \tag{3.33}$$

To calculate the elements of (3.32) we use equation (3.23), and get

$$\frac{\partial a_{j,\tau}}{\partial z_{k,\tau}} = \frac{\partial}{\partial z_{k,\tau}}\big(g(z_{j,\tau})\big) = g'(z_{k,\tau})\,\delta_{kj},$$

where δ_{kj} is the Kronecker delta. Thus,

$$\frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{z}_\tau} = \operatorname{diag}(g'(\boldsymbol{z}_\tau)).$$

To calculate the elements of (3.33) we use equation (3.22), and get

$$\frac{\partial z_{k,\tau}}{\partial a_{j,\tau-1}} = \frac{\partial}{\partial a_{j,\tau-1}}\big(b_k + u_k\,x_\tau + \sum_{i=1}^{H} w_{k,i}\,a_{i,\tau-1}\big) = w_{k,j}.$$

Thus,

$$\frac{\partial \boldsymbol{z}_\tau}{\partial \boldsymbol{a}_{\tau-1}} = \boldsymbol{W}.$$

We now insert the two expressions in (3.31), and get the following:

$$\frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{a}_{\tau-1}} = \operatorname{diag}(g'(\boldsymbol{z}_\tau))\,\boldsymbol{W}.$$

Equation (3.30) now becomes

$$\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_t} = \prod_{T\geq\tau>t} \operatorname{diag}(g'(\boldsymbol{z}_\tau))\ \boldsymbol{W}.$$

If g is the identity function such that $\boldsymbol{a}_\tau = \boldsymbol{z}_\tau$ then

$$\frac{\partial \boldsymbol{a}_\tau}{\partial \boldsymbol{z}_\tau} = \boldsymbol{I},$$

the identity matrix. Hence, we have

$$\frac{\partial \boldsymbol{a}_T}{\partial \boldsymbol{a}_t} = \prod_{T\geq\tau>t} \boldsymbol{I}\,\boldsymbol{W} = \boldsymbol{W}^{T-t-1}.$$

So after $l = T - t - 1$ time-steps, the repeated multiplication of $\boldsymbol{W}$ is equivalent to multiplying with $\boldsymbol{W}^l$, the weight matrix to the power of l. If we suppose that the square $H \times H$ weight matrix $\boldsymbol{W}$ has an eigendecomposition $\boldsymbol{W} = \boldsymbol{Q}\operatorname{diag}(\boldsymbol{\lambda})\,\boldsymbol{Q}^{-1}$, where $\boldsymbol{\lambda} = \{\lambda_i, \quad i = 1,\ldots,H\}$ are the corresponding eigenvalues, then

$$\boldsymbol{W}^l = \big(\boldsymbol{Q}\operatorname{diag}(\boldsymbol{\lambda})\,\boldsymbol{Q}^{-1}\big)^l = \boldsymbol{Q}\operatorname{diag}(\boldsymbol{\lambda})^l\,\boldsymbol{Q}^{-1}. \tag{3.34}$$

Thus, if l is sufficiently large and $|\lambda_i| < 1$ then $\lambda_i^l \to 0$ (vanish) or if $|\lambda_i| > 1$ then $\lambda_i^l \to \infty$ (explode). This is what is meant by exploding or vanishing gradients for simple RNNs. The most common of the two is vanishing gradients, and if the gradients of the cost with respect to the weights become very small, the weight updates will be small too, and thus the number of epochs required to reach a local minimum for the cost function will be large. There will also be a problem learning long term dependencies due to this.

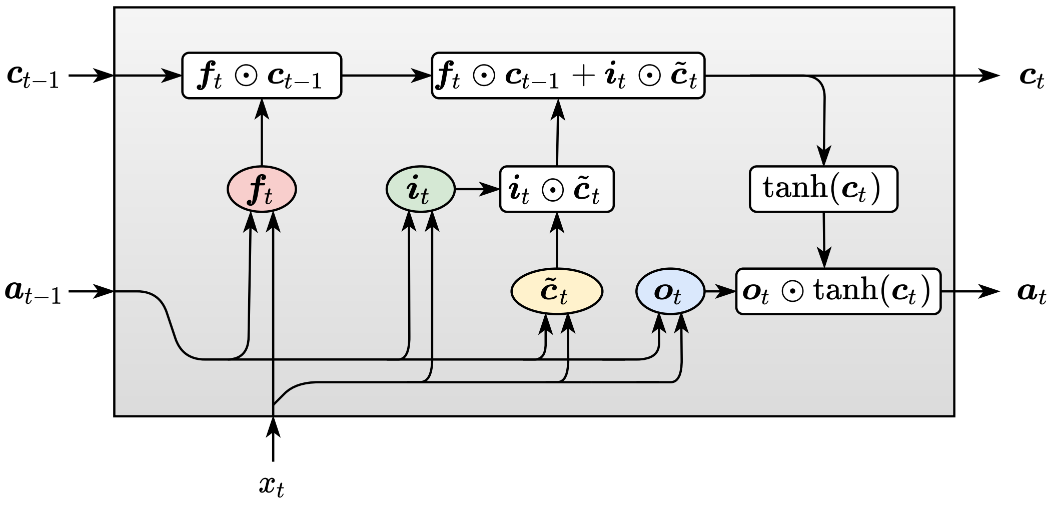
The example above is a very simplified presentation of why the problem with vanishing and exploding gradients occur. In a more realistic scenario, the weight matrix $\boldsymbol{W}$ will not be diagonalizable and thus not have an eigendecomposition. In this case, we need to use Jordan normal form of $\boldsymbol{W}$ instead, $\boldsymbol{W} = \boldsymbol{P}\boldsymbol{J}\boldsymbol{P}^{-1}$. The matrix $\boldsymbol{J}$ is the Jordan matrix, which is an upper triangular matrix, with the eigenvalues on the diagonal and ones on the superdiagonal. We omit the calculations of $\boldsymbol{W}^l$ in this case as they are similar to the above calculation (3.34), and produce the same problem with vanishing or exploding gradients. For a detailed description and proof that any square matrix $\boldsymbol{W}$ is similar to a Jordan matrix $\boldsymbol{J}$, see appendix B in Strang (2006).

It can also be shown that the gradients will either vanish or explode if the activation function g is a non-linear function like tanh.

3.3.1. Long Short-Term Memory

There has been made different versions of the recurrent neural network to overcome the problem of vanishing gradients, and one of the most successful is the long short-term memory (LSTM) model. This type of recurrent neural network architecture was first introduced by Hochreiter and Schmidhuber (1997). In LSTM networks, we replace the hidden recurrent layers with *memory blocks*. These blocks are similar to simple recurrent layers in that both take as input x_t and the output from the previous time-step, $\boldsymbol{a}_{t-1}$, but memory blocks also have a recurrently connected *cell state* and *gates*. The gates are used to regulate the flow of information between time-steps, and consist of a sigmoid activation, that outputs values between zero and one. Simply put, if the value is zero, the gate is "closed" and no information is passed through the gate. Conversely, if the value is one, the gate is fully "open", and all the information passes through the gate. The memory block has three types of gates; the forget gate, the input gate and the output gate. The different gates each have a specific "job" in the memory block and regulate different parts of the information. The vector with the gate values between zero and one is element-wise multiplied with the vector with the information which we want to control.

The memory cell holds a "cell state", $\boldsymbol{c}_t$. This cell state is passed on to the next time-step along with the output $\boldsymbol{a}_t$, and updated in each time-step. An illustration of a memory block is depicted in Figure 3.14, and shows the memory cell update and different gates.



$\boldsymbol{c}_{t-1}$
$\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}$
$\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$
$\boldsymbol{c}_t$
$\boldsymbol{f}_t$
$\boldsymbol{i}_t$
$\boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$
$\tanh(\boldsymbol{c}_t)$
$\boldsymbol{a}_{t-1}$
$\tilde{\boldsymbol{c}}_t$
$\boldsymbol{o}_t$
$\boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$
$\boldsymbol{a}_t$
x_t

Figure 3.14: Memory block in a LSTM network at time-step t. The red, green and blue ellipses are the forget, input and output gates, respectively.

We will now look at the equations for a memory block in an LSTM network with one hidden layer. First, we will give the equation for the forget gate at time-step t:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{b}^{\boldsymbol{f}} + \boldsymbol{u}^{\boldsymbol{f}} x_t + \boldsymbol{W}^{\boldsymbol{f}} \boldsymbol{a}_{t-1}).$$

The σ denotes the sigmoid activation function, and it takes x_t and $\boldsymbol{a}_{t-1}$, the output from the memory block at time-step $(t-1)$, as input. Since we will have many different sets of weight vectors and matrices in the memory block, we will denote the weights with a superscript to differentiate between the weights. In the forget gate $\boldsymbol{b}^{\boldsymbol{f}}$, $\boldsymbol{u}^{\boldsymbol{f}}$ and $\boldsymbol{W}^{\boldsymbol{f}}$, are the bias weights, the input weights and the recurrent weights between time-steps, respectively. The forget gate will be multiplied element-wise with the cell state in the previous time-step, and controls what information from this that we will remember or forget.

Next, we have the input gate. The equation for this gate is the same as for the forget gate, except with different weights.

$$\boldsymbol{i}_t = \sigma(\boldsymbol{b}^{\boldsymbol{i}} + \boldsymbol{u}^{\boldsymbol{i}} x_t + \boldsymbol{W}^{\boldsymbol{i}} \boldsymbol{a}_{t-1}).$$

The input gate will regulate how much of the new information from x_t and $\boldsymbol{a}_t$ we will add to the cell state in the current time-step t. It will be multiplied element-wise with the *candidate cell state*. The equation for this is as follows:

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{b}^{\boldsymbol{c}} + \boldsymbol{u}^{\boldsymbol{c}} x_t + \boldsymbol{W}^{\boldsymbol{c}} \boldsymbol{a}_{t-1}).$$

In this case, we have used the tanh function as activation, but it is also possible to use the sigmoid. We then update the cell state using the forget gate, the previous cell state, the input gate and the candidate cell state;

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t.$$

As we can see, the new cell state $\boldsymbol{c}_t$ is the sum of the information we want to remember from the previous cell state $\boldsymbol{c}_{t-1}$ and what we want to add from the new information. Lastly, we want to decide what to output from the memory block. We do this using the output gate:

$$\boldsymbol{o}_t = \sigma(\boldsymbol{b}^{\boldsymbol{o}} + \boldsymbol{u}^{\boldsymbol{o}} x_t + \boldsymbol{W}^{\boldsymbol{o}} \boldsymbol{a}_{t-1}).$$

The output gate is then multiplied element-wise with the cell state $\boldsymbol{c}_t$ that has gone through a tanh function first.

$$\boldsymbol{a}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$$

Similar to the simple RNN model, at the last time-step T, the activation $\boldsymbol{a}_T$ is passed to a fully connected output layer like in equation (3.27).

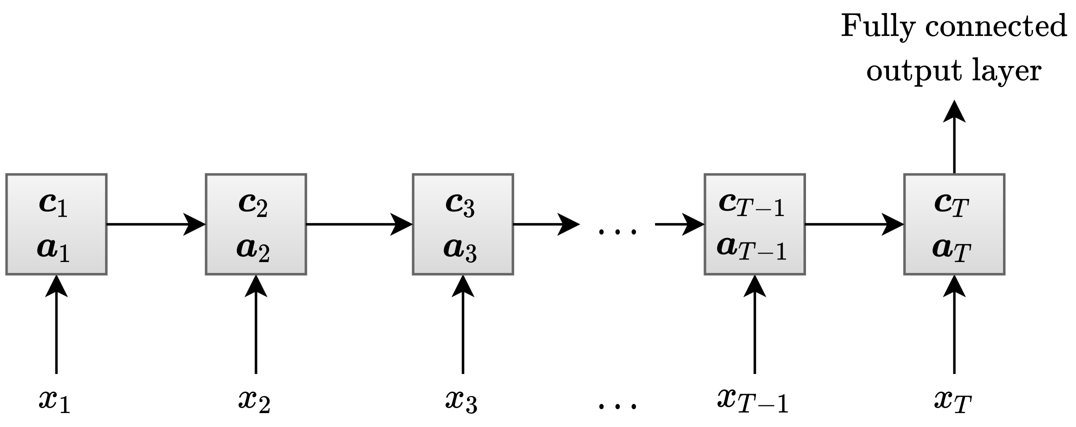

Fully connected output layer
$\boldsymbol{c}_1$ $\boldsymbol{a}_1$
$\boldsymbol{c}_2$ $\boldsymbol{a}_2$
$\boldsymbol{c}_3$ $\boldsymbol{a}_3$
...
$\boldsymbol{c}_{T-1}$ $\boldsymbol{a}_{T-1}$
$\boldsymbol{c}_T$ $\boldsymbol{a}_T$
x_1
x_2
x_3
...
x_{T-1}
x_T

Figure 3.15: Unfolded long short-term memory neural network with one hidden layer.

We have omitted the usual superscript (k) denoting the layer for simplicity, and easy reading since our equations above all belong to the same hidden layer. We could of course have multiple hidden layers of memory blocks, and the generalization for this will be similar to equations (3.28) and (3.29). Long short-term networks with multiple hidden layers are often referred to as stacked LSTM's.

The question is, how does this type of architecture help with preventing the vanishing gradient problem, and the answer lies in the cell state and the forget gate. In the LSTM network, the backward propagation goes through time only in the cell states $\boldsymbol{c}_t$. We have

$$\frac{\partial \boldsymbol{c}_T}{\partial \boldsymbol{c}_t} = \frac{\partial \boldsymbol{c}_T}{\partial \boldsymbol{c}_{T-1}} \frac{\partial \boldsymbol{c}_{T-1}}{\partial \boldsymbol{c}_{T-2}} \cdots \frac{\partial \boldsymbol{c}_{t+1}}{\partial \boldsymbol{c}_t} = \prod_{T \geq \tau > t} \frac{\partial \boldsymbol{c}_\tau}{\partial \boldsymbol{c}_{\tau-1}},$$

and

$$\frac{\partial \boldsymbol{c}_\tau}{\partial \boldsymbol{c}_{\tau-1}} = \frac{\partial}{\partial \boldsymbol{c}_{\tau-1}} \big(\boldsymbol{f}_\tau \odot \boldsymbol{c}_{\tau-1} + \boldsymbol{i}_\tau \odot \tilde{\boldsymbol{c}}_\tau . \big) = \boldsymbol{f}_\tau .$$

Thus,

$$\frac{\partial \boldsymbol{c}_T}{\partial \boldsymbol{c}_t} = \prod_{T \geq \tau > t} \boldsymbol{f}_\tau .$$

Since the network controls the forget gate activations, $\boldsymbol{f}_\tau$, with the weights $\boldsymbol{b}^{\boldsymbol{f}}$, $\boldsymbol{u}^{\boldsymbol{f}}$ and $\boldsymbol{W}^{\boldsymbol{f}}$, and the activation function of the forget gate is the sigmoid function, the network can if it is necessary, learn weights such that $f_\tau \approx 1$. Thus, the network can learn weights such that the gradients will not vanish or vanish at a much slower rate than for simple RNNs.

Chapter 4

Simulating data

In this chapter, we will demonstrate how we simulate the time series data that we need to train the neural network models, and how we select order based on the AIC and BIC values. We will simulate data from pure autoregressive (AR), and moving-average (MA) processes with orders from 1 to 4, in addition to data from autoregressive moving-average (ARMA) processes with orders from $(1, 1)$ to $(4, 4)$.

Software

We use Python with the library statsmodels (Seabold and Perktold, 2010) to simulate the time series data and calculate AIC and BIC values.

Approach for simulating data of length n from an ARMA(p,q) process

1. Generate uniform random samples of size p and q from the interval $(-2.1, -0.1) \cup (0.1, 2.1)$. This will be the values for the $p + q$ parameters. For AR(p) let $q = 0$ and for MA(q) let $p = 0$.

2. Calculate the roots.

    (a) If $p \neq 0$: let $\boldsymbol{r}_{\text{AR}}$ be the roots for the autoregressive polynomial. If $\min(|\boldsymbol{r}_{\text{AR}}|) > 1$ then the model is stationary and causal, and we can proceed to the next step. Otherwise, the model is neither stationary nor causal and we go back to step 1.

    (b) If $q \neq 0$: let $\boldsymbol{r}_{\text{MA}}$ be the roots for the moving-average polynomial. If $\min(|\boldsymbol{r}_{\text{MA}}|) > 1$ then the model is invertible, and we can proceed to the next step. Otherwise, the model is not invertible and we go back to step 1.

    (c) If $p \neq 0$ and $q \neq 0$, then check if any of the roots in $\boldsymbol{r}_{\text{AR}}$ is equal to any of the roots in $\boldsymbol{r}_{\text{MA}}$. If we have any equal roots, we go back to step 1, otherwise, we proceed to the next step.

3. Simulate n values from an ARMA(p, q) model with the chosen parameters, and we use a standard normal distribution for the white noise.

4. Label the simulated time series with the order (p, q).

The reason for drawing the autoregressive and moving average parameters from the interval $(-2.1, -0.1) \cup (0.1, 2.1)$ is that we want to avoid values too close to zero. Also, we found through experimentation that values greater in absolute value than 2.1 did not yield causal and invertible models.

We chose to simulate 250 000 samples per order in each dataset, yielding 1 000 000 and 4 000 000 samples for the AR/MA and ARMA datasets respectively. After simulation of data, we split the data in a training, validation and test set. Then, we calculate the AIC and BIC for each sample in the test dataset.

Approach for selecting order by AIC and BIC for the test sets

1. Fit a model for all orders to each sample we want to classify.

2. Calculate the AIC and BIC value for each fitted time series.

3. Label each sample with the order that gives the lowest AIC and BIC value, respectively.

Table 4.1: Datasets.

Dataset name	Model and orders	Length (n)	Number of samples	Training/ validation/test split
AR30	AR(1)-AR(4)	30	1 000 000	0.98/0.01/0.01
AR100	AR(1)-AR(4)	100	1 000 000	0.98/0.01/0.01
AR1000	AR(1)-AR(4)	1000	1 000 000	0.98/0.01/0.01
MA30	MA(1)-MA(4)	30	1 000 000	0.98/0.01/0.01
MA100	MA(1)-MA(4)	100	1 000 000	0.98/0.01/0.01
MA1000	MA(1)-MA(4)	1000	1 000 000	0.98/0.01/0.01
ARMA30	ARMA(1,1)-ARMA(4,4)	30	4 000 000	0.98/0.01/0.01
ARMA100	ARMA(1,1)-ARMA(4,4)	100	4 000 000	0.98/0.01/0.01
ARMA1000	ARMA(1,1)-ARMA(4,4)	1000	4 000 000	0.98/0.01/0.01

In many machine learning books, the authors will recommend as a general rule of thumb that we split that data in 0.8/0.1/0.1 for the training, validation and test set. However, when the dataset is very large, this might not be a good choice. We want the validation and test set to represent the variation in the data and at the same time, use as much as possible of the data for training. In our case, a split of 0.98/0.01/0.01 yields 10 000 samples in the AR/MA datasets and 40 000 samples in the ARMA datasets for validation and testing. This seems sufficient.

A small fraction of the samples in some of the test datasets, specifically the AR and ARMA datasets were not able to be fitted with any of the orders, and thus a AIC and BIC value was not calculated. We still chose to keep these samples with NaN as AIC and BIC value, since they amounted to 0.24% (AR30), 0.1% (AR100), 1.27% (ARMA30), 0.3% (ARMA100), 0.13% (ARMA1000) and thus would not have a significant impact the accuracies of AIC and BIC.

Chapter 5

Model training and evaluation of results

5.1. Software

We use the Python library Keras (Chollet et al., 2015), which is a high-level application programming interface (API), with Tensorflow (Abadi et al., 2015) as a backend to train all our models. For further details on this, see Appendix B.

5.2. Model training

For the fully connected, convolutional and LSTM layers, we initialize the weights with values before training begins. The default choices in Keras are zeros for the bias weights and the Glorot uniform initializer (Glorot and Bengio, 2010) for weights used in the linear transformation of the outputs. This initializes the weights by generating values from a uniform distribution, $\mathrm{W} \sim \mathrm{Unif}(-6/(m+n), 6/(m+n))$, where n is the number of nodes in the layer and m is the number of nodes in the previous layer. For weights used in the linear transformation of the gates, we use the orthogonal initializer, which generates an orthogonal matrix.

Each model will be trained using a mini-batch, that is a batch size between 64 and 512. Also, we use early stopping with a patience between 10 and 30. So training is stopped when the cost function evaluated for the validation set is not improved for 10 or 30 epochs, and we restore the weights back to the epoch that gives the lowest validation cost. A larger batch size makes training faster per epoch since the calculations can be parallelized, but uses more memory since all the activation values must be stored until all the samples in the batch are passed through the network. Higher patience results in more epochs before training is stopped, and thus makes total training time longer. Thus, the exact choices for batch size and early stopping patience are based on training time and memory limitations for a specific dataset and model.

When training the models we use the Adam optimizer from Chapter 3 in equation (3.18) with learning rate $\alpha = 0.001$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

The results are based on training several different fully connected, convolutional and LSTM neural network models on each training dataset. Then we choose the best model, i.e. the model with the highest validation accuracy in each model category.

$$\text{Accuracy} = \frac{\text{Number of correctly classified samples}}{\text{Total number of samples}}$$

After choosing a model, we evaluate it on the test data and calculate accuracies for each class in addition to the average accuracy for all classes. We expect the test accuracies to be slightly lower than the validation accuracies. This is because even though the models are not trained on the validation data, we select our models based on evaluating the models on this data. Thus, it will give us an overly optimistic result. In Appendix A, we illustrate the models used along with a Table of accuracies for each dataset. We will also introduce a dummy classifier, that serves as a baseline for accuracies of our methods. Since we have approximately equal representation of each class, we use a uniform distribution to randomly classify samples. Thus, if we have M classes, the uniform dummy classifier is,

$$P(\text{true class} = j) = \frac{1}{M} \quad \text{for } j = 1, \ldots, M. \tag{5.1}$$

We will also include order selection column charts showing what order the neural network that performs best, along with AIC and BIC, selects given a true order. This will give a more detailed picture of how an order selection method actually behaves for the samples that are wrongly classified. These column charts are found in Appendix A.

It is important to note that the models and results in this thesis by no means represent the best possible or optimal models for our data. Since we have trained at least two models in each model category (fully connected, convolutional and recurrent) on each of the nine datasets, fine-tuning the architectures and hyperparameters to find the optimal model for each dataset were not realistic given the time frame. Thus, there will most likely exist other model architectures that would give us better classification accuracies than the models we use.

Each model is only trained and evaluated on the test set one time. Training the same model multiple times and taking the average of the classification accuracies would yield less variance in the results because there is some randomness due to the weight initializations that possibly result in different local minimums for the cost function. However, this risk is decreased when the training dataset is large. In our case, we use training datasets with almost 1 000 000 samples when we have 4 categories to classify and almost 4 000 000 samples when we have 16 categories to classify. We found that the validation accuracies for models within the same model sub-class, i.e. fully connected, convolutional or LSTM did not vary much even for possible large differences in the number of hidden layers or size of layers. This suggests that the results we present are relatively stable.

5.3. Results

5.3.1. Time series of length 30

We begin with the three datasets, AR30, MA30 and ARMA30, with the shortest length of each sample. The neural network models for each of these datasets that provided the highest validation accuracy scores are illustrated in Figures A.1 - A.3.

In Table A.1, we see the accuracy scores for the models evaluated on the test set of the AR30 dataset, together with accuracy scores for AIC and BIC. If the orders where randomly predicted using the uniform dummy classifier (5.1) the accuracies would be 0.25, since we have 4 classes. We notice that all the models, in addition to AIC and BIC, performs better than the uniform dummy classifier on this dataset. We also observe that all the neural network models outperform both AIC and BIC in the average accuracy for all orders. The LSTM model performs the best with an average accuracy of 0.630, compared to 0.558 and 0.549 for AIC and BIC, respectively. It is not surprising that the neural network model that has the best performance is the LSTM since this model is designed for the type of sequential data that we use. It is also interesting to note that while BIC is least accurate on average, it is most accurate in classifying AR(1) and AR(2) models. This might occur because BIC penalizes a high number of parameters more, as we see with the low accuracy for AR(3) and AR(4). In Figure A.9, we can see which order the best neural network model (LSTM), together with AIC and BIC, chooses given a true order. We observe that the LSTM model, AIC and BIC show a similar distribution of classifications, especially AIC and LSTM. That is, given that the true order is, e.g. $p = 3$, LSTM and AIC will select approximately the same number of samples with the order $p = 4$ etc.

For the MA30 dataset in Table A.2 we observe that all the neural network models outperform both AIC and BIC, with a difference of 0.2 between the best performing model (LSTM) and AIC/BIC. We notice that for MA(2) and MA(4), AIC and BIC yield accuracies that are only slightly above or below what the accuracy for the uniform dummy classifier. Figure A.10 sheds light on why this occurs; both AIC and BIC are most likely to select MA(1) regardless of the true order. This explains the high accuracies for order $q = 1$ combined with relatively low average accuracies for AIC and BIC. Further, we notice that unlike for the AR30 dataset, the LSTM model show a very different distribution of classifications compared with AIC and BIC. While the LSTM model shows a similar pattern of classification as the LSTM model for the AR30 dataset, AIC and BIC fail to recognize which order the moving-average time series data is generated from.

In Table A.3 we find the accuracies for the ARMA30 dataset. We can observe that all three neural network models, AIC and BIC perform poorly, especially for the higher orders of p and q. Since we have 16 different combinations of p and q, the uniform dummy classifier would accurately classify 0.063 of the samples. We notice that several of the individual order accuracy scores are below this value, especially for BIC where half of the accuracies are less than 0.063. The LSTM model performs slightly better than the other neural network models, AIC and BIC, with an average accuracy of 0.199. In Figure A.11, we see that many of the column charts for the LSTM model, AIC and BIC exhibit a wavy pattern. This makes sense because of the way the orders on the x-axis is ordered. If e.g. the true order is $(3, 2)$, the order $(2, 2)$ which is four steps left on the x-axis in A.11c, would likely fit the data well too. An interesting observation is that when $p > 2$ and $q > 2$, the true order is never the most likely to be predicted by the LSTM model or selected by AIC and BIC. The poor results for AIC and BIC can be explained by the number of parameters to estimate, combined with only 30 observations per sample. For the largest model, ARMA$(4, 4)$, the number of parameters is 8, and this gives a ratio of only 3.75 between the number of parameters and observations.

5.3.2. Time series of length 100

Next, we will discuss the results for the datasets with time series of length 100, AR100, MA100 and ARMA100. The models that were chosen for these datasets are illustrated in Figures A.4 and A.5.

The accuracy scores for the AR100 data is in Table A.4, and shows that yet again the LSTM model proves to yield the highest average accuracy with 0.822 correct classifications compared to 0.756 for AIC and 0.800 for BIC. In Figure A.12, we find a very similar pattern of order selections between LSTM and AIC/BIC, just like we found for the AR30 data.

In Table A.5 we find the accuracies for the MA100 dataset. Like we saw with MA30, all the neural network models are significantly better at selecting the true order than AIC and BIC. The best model is the LSTM, with a classification accuracy of 0.785. With 100 observations per sample, we find that AIC and BIC performs much better for $q = 3$ and $q = 4$ than for the MA30 data, but for $q = 4$ BIC still has an accuracy below 0.25. Figure A.13 shows that LSTM compared to AIC and BIC behaves very differently when the true order is larger than one. While LSTM gives sound order classifications, e.g. a larger number of the MA(3) samples are classified as $q = 2$ than $q = 1$, AIC, and BIC does not behave this way. We observe that both AIC and BIC are more likely to select $q = 1$ than $q = 2$ for the MA(3) samples, and similarly for the MA(4) samples.

For the ARMA100 dataset, we find the results in Table A.6, and we observe that LSTM on average yields a higher accuracy than all other models, AIC and BIC, with an accuracy of 0.364. This is slightly higher than the accuracy of BIC with an accuracy of 0.327. Figure A.14 shows that LSTM, AIC and BIC generally choose the true order more often than any other individual orders, with a few exceptions for BIC. There are also some similarities in the patterns of order selections given a true order between LSTM, AIC and BIC. However, they are not as apparent as we observed for the pure autoregressive dataset.

5.3.3. Time series of length 1000

Lastly, we review the results for the three datasets AR1000, MA1000 and ARMA1000. We find the models that are chosen for each dataset in Figures A.7- A.9.

For the AR1000 dataset, we find the accuracies in Table A.7, and observe that BIC performs best with an average accuracy of 0.983 compared to 0.973 for the LSTM model. The difference is very small, and as we can see in Figure A.15, LSTM and BIC have almost the same distribution of order classifications.

The accuracy scores for the MA1000 dataset is in Table A.8. We observe that the LSTM model gives almost perfect classification for all orders and an average accuracy of 0.974, compared to 0.629 for BIC. We notice that the LSTM model for the MA1000 dataset performs similarly as the LSTM model for AR1000 dataset, unlike AIC and BIC. In Figure A.16 we find, again, inconsistency in the order selection by AIC and BIC for $q \geq 2$.

In Table A.9, we find the results for the ARMA1000 dataset. First, we observe that BIC performs the best with an average accuracy of 0.684. The LSTM model gives a slightly lower average accuracy of 0.660. We also observe that the fully connected (NNET) only classifies 0.218 of the samples correctly, which is lover than what we found for the fully connected model for the ARMA100 data. In Figure A.17, we can observe clear similarities between LSTM and BIC.

Chapter 6

Conclusion

In this thesis we have explored if we can use deep learning methods to identify the order of simulated time series data, and how the performance of different types of neural networks compares to the information criteria based order selection methods, AIC and BIC. We recall that AIC and BIC are designed to minimize the one-step prediction error, and this goal is not necessarily achieved by using the true order of the model. In the neural network method, we simply seek to find a model that can correctly identify the true order. Thus, the comparison of the methods is not to determine which method is "best", but rather to attempt to answer the question if neural networks can be used for model selection in time series.

We also investigated how different time series processes (AR, MA and ARMA) in addition to the number of observations per sample affected the results. We found that of the neural network models, the LSTM model was most accurate in classifying the order for all datasets. This is in accordance with the theory behind LSTM models. They are designed to be used on sequential data, while fully connected and convolutional networks are not. Still, the fully connected models seemed to work reasonably well for the datasets with samples of length 30 and 100, and in most cases, the difference between CNN and LSTM was small.

When we compared the performance of neural networks with AIC and BIC, we found that the LSTM model outperformed AIC for all datasets. The LSTM model also performed better than BIC, except for the AR1000 and ARMA1000 datasets. It is worth to note that in those cases, the difference in accuracy was small. We also found that for the neural network models, the difference in performance on data from both MA and AR processes was not nearly as great as that of AIC and BIC. In addition to this, the difference between deep learning and AIC/BIC for the AR and ARMA data was less than that for MA data. This is probably because MA parameters are more difficult to estimate than AR parameters.

In regards to how the length of the time series affected the results, we observed that the average accuracy performance in the deep learning models, AIC and BIC generally increased with the number of observation per sample. An exception from this was the fully connected (NNET) models when the number of observations was 1000.

Another interesting find was the similarities between LSTM and the information criterion methods, in the distribution of order selections, especially for the AR data. Further investigation could be done to find out whether these similarities stem from the same samples in the data. That is, given a sample of time series data, will AIC or BIC and LSTM generally select the same order, or can the similarities in order selection distribution be attributed to something else.

It would also be interesting to investigate how another distribution for the white noise would affect the results, as the likelihood function that AIC and BIC are based on assumes a normal distribution for the observations.

Appendix A

Models and results

A.1. Models

Since we will many different models, we want to have a clear way of describing the architecture of each model. We will borrow the syntax from the Keras library to specify different layers.

- A fully connected layer with n nodes, and the rectified linear unit (reLU) as activation function:

```
Dense(n), act = reLU
```

- A dropout layer, with dropout fraction q.

```
Dropout(q)
```

- A 1D convolutional layer with f filters and kernel size s with a sigmoid activation:

```
Conv1D(f,s), act = sigmoid
```

  If we use zero padding, i.e keeping the dimensions of the output the same as the input to the layer, we write:

```
Conv1D(f,s), act = sigmoid, pad
```

  We use the default stride length of 1 in all convolutional layers.

- The flatten operation:

```
Flatten()
```

- A max pooling layer with a window size of r:

```
MaxPooling1D(r)
```

- A LSTM layer with n nodes in each gate, cell-state and output with tanh as activation for the candidate cell state and output state:

  ```
  LSTM(n), act = tanh
  ```

  The default activation for the gates is the sigmoid activation and this is what we use for all LSTM layers.

The Figures A.1-A.8 show the neural network architectures for each dataset.

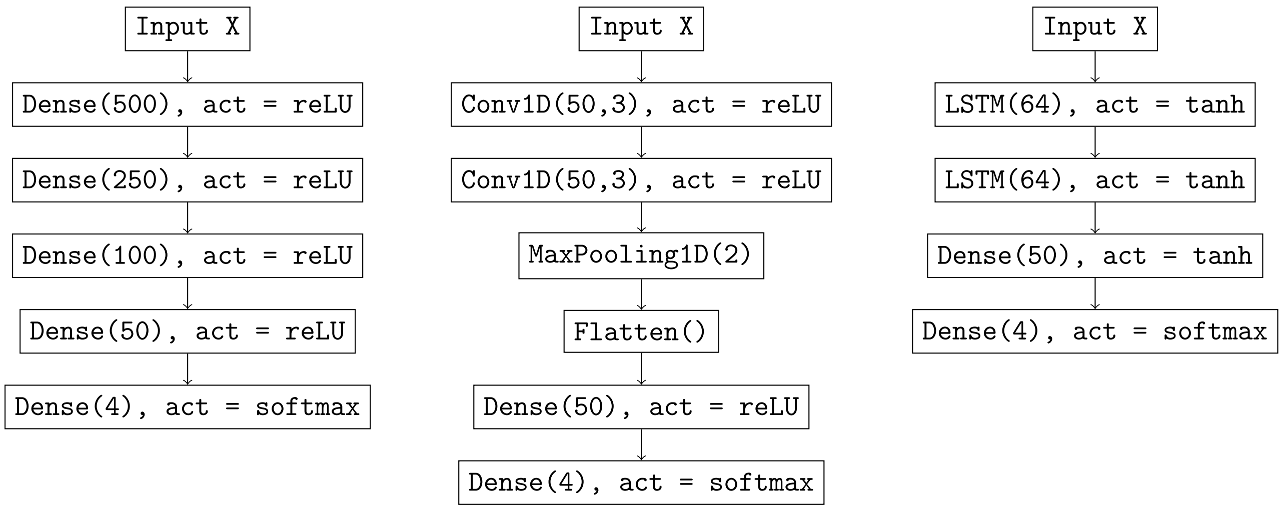


Figure A.1: Models used for AR30 dataset. Left: fully connected model with 171 104 weights. Middle: CNN model with 40 604 weights. Right: LSTM model with 53 374 weights.

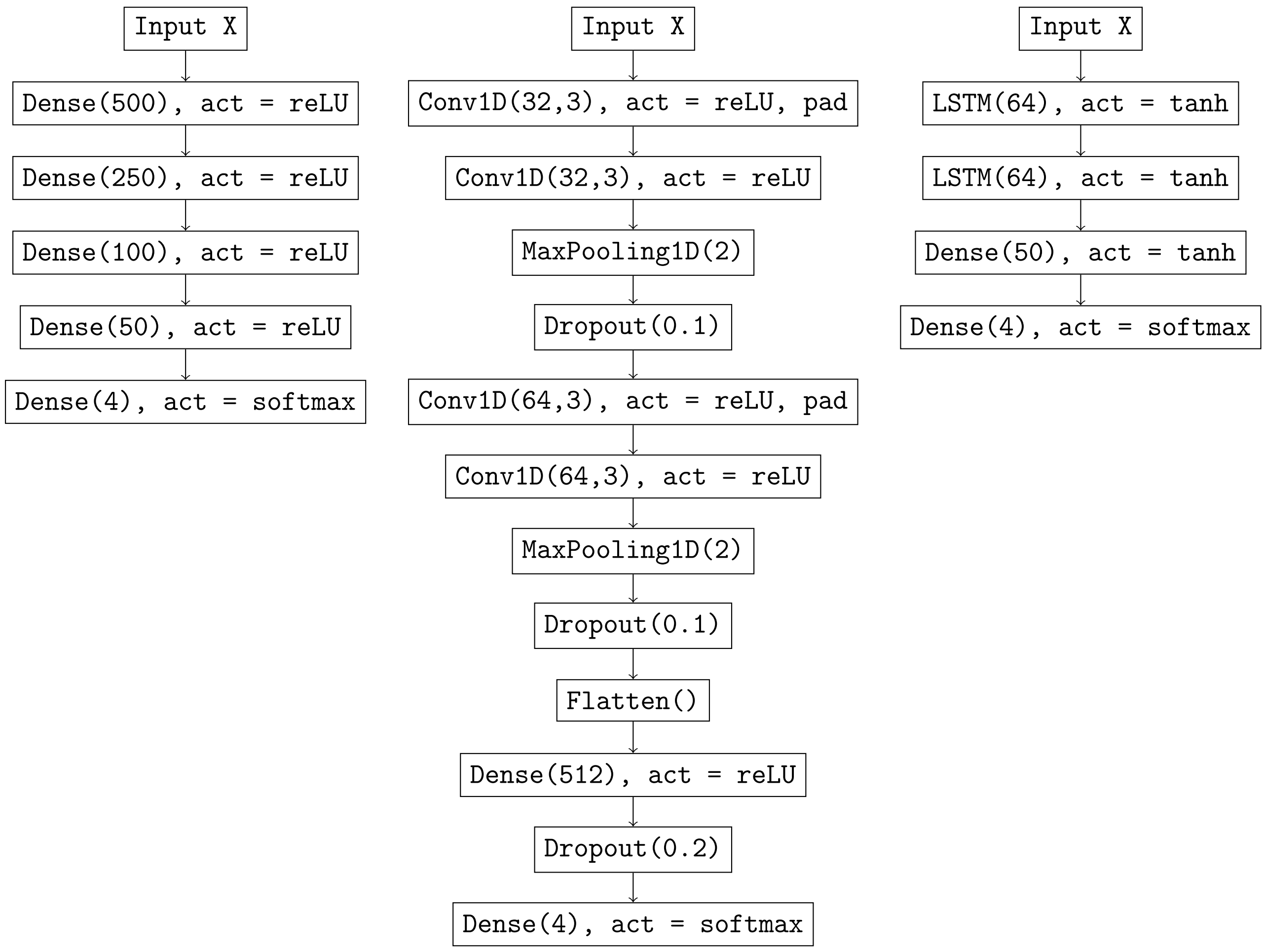


Figure A.2: Models used for MA30 dataset. Left: fully connected model with 171 104 weights. Middle: CNN model with 220 964 weights. Right: LSTM model with 53 374 weights.

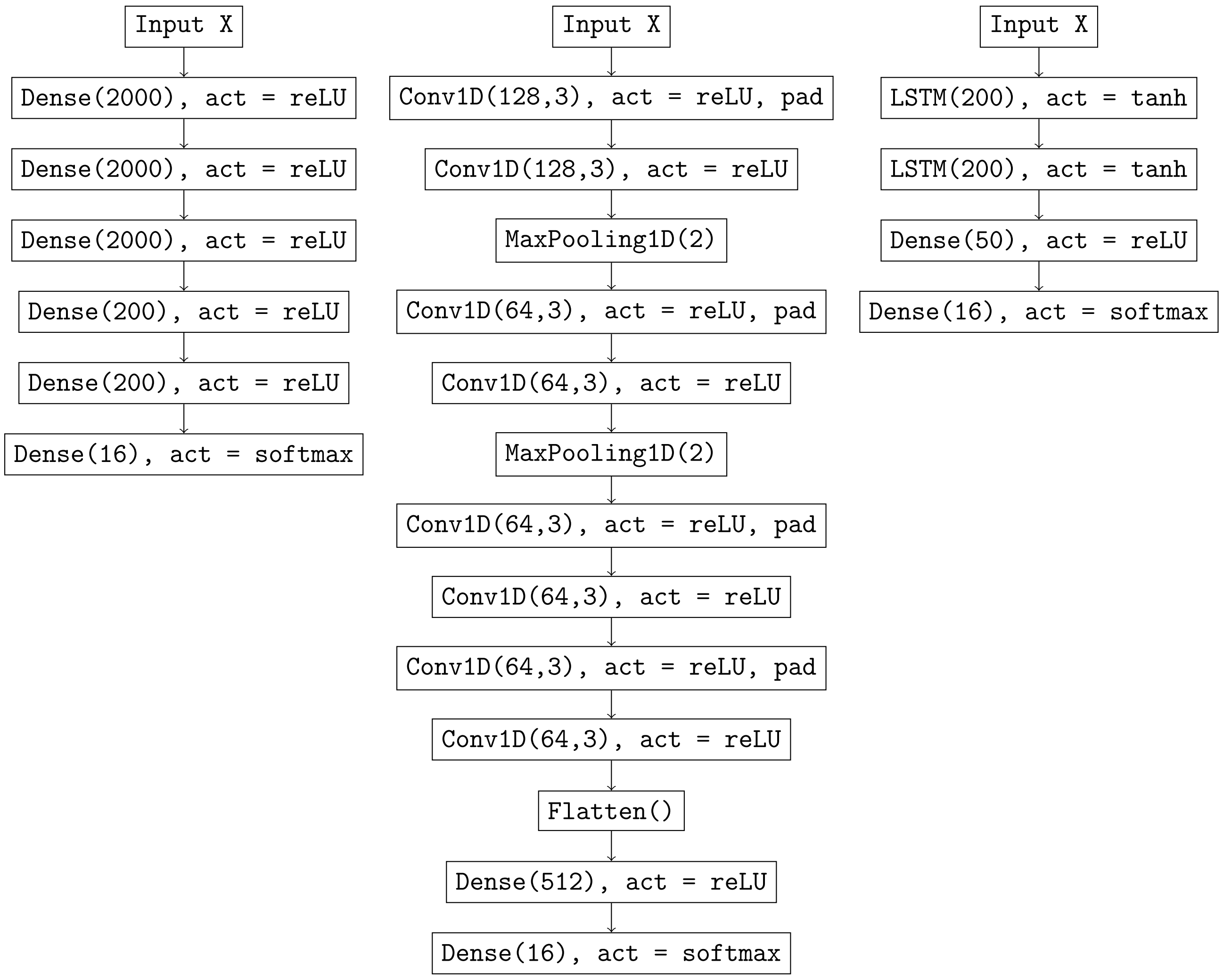


Figure A.3: Models used for the ARMA30 dataset. Left: fully connected model with 8 509 616 weights. Middle: CNN model with 210 448 weights. Right: LSTM model with 493 266 weights.

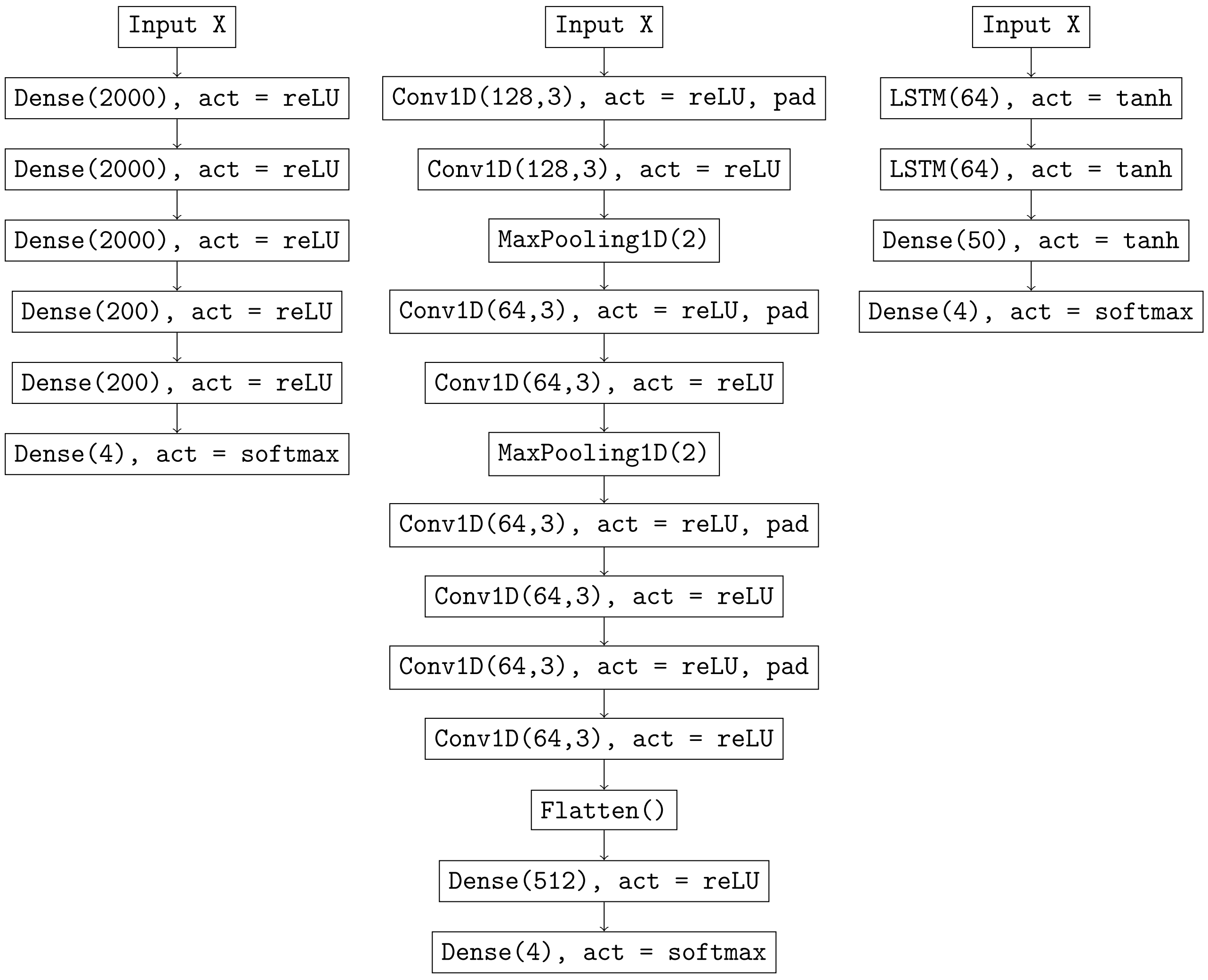


Figure A.4: Models used for AR100 and MA100 dataset. Left: fully connected model with 8 647 204 weights. Middle: CNN model with 761 348 weights. Right: LSTM model with 53 374 weights.

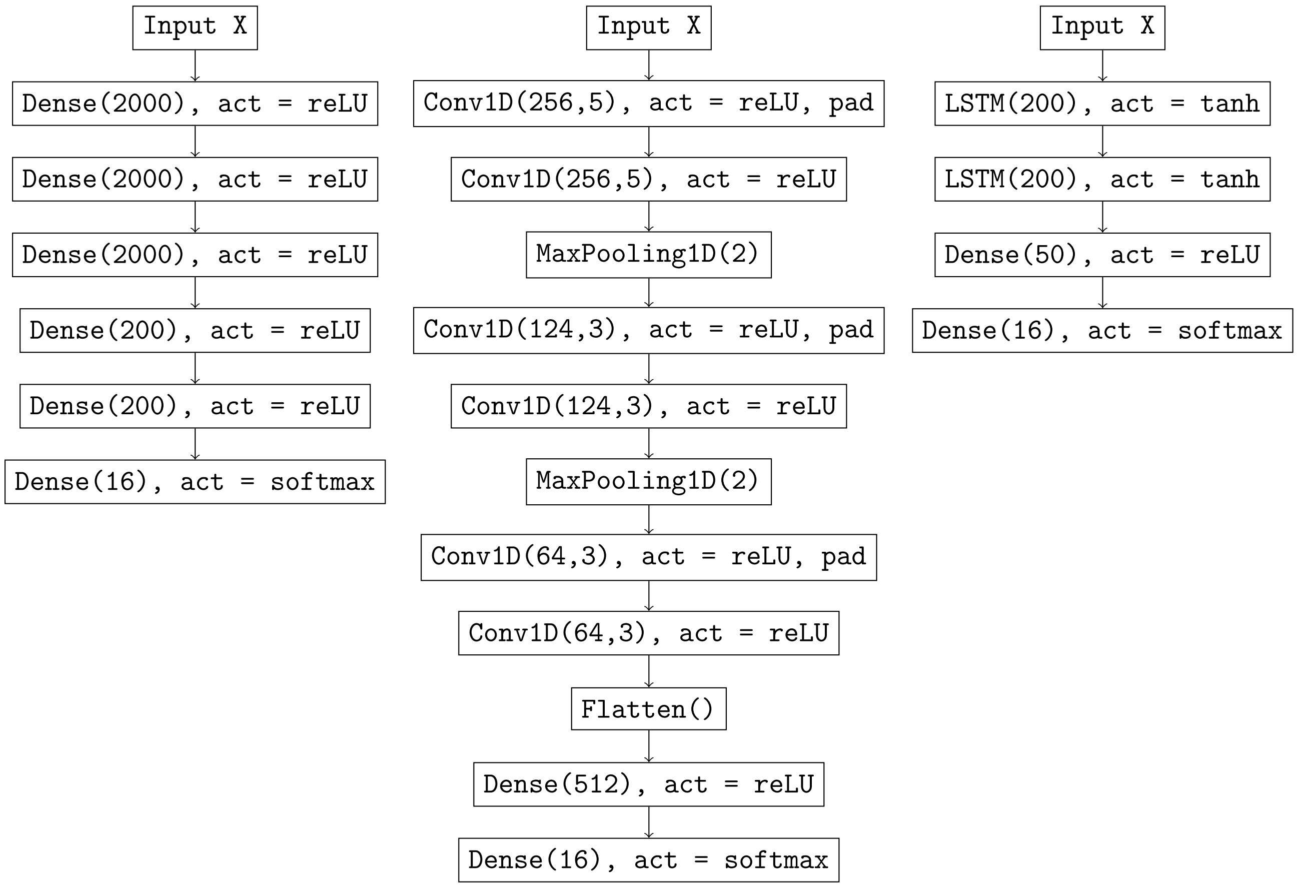


Figure A.5: Models used for the ARMA100 dataset. Left: fully connected model with 8 649 616 weights. Middle: CNN model with 1 211 024 weights. Right: LSTM model with 493 266 weights.

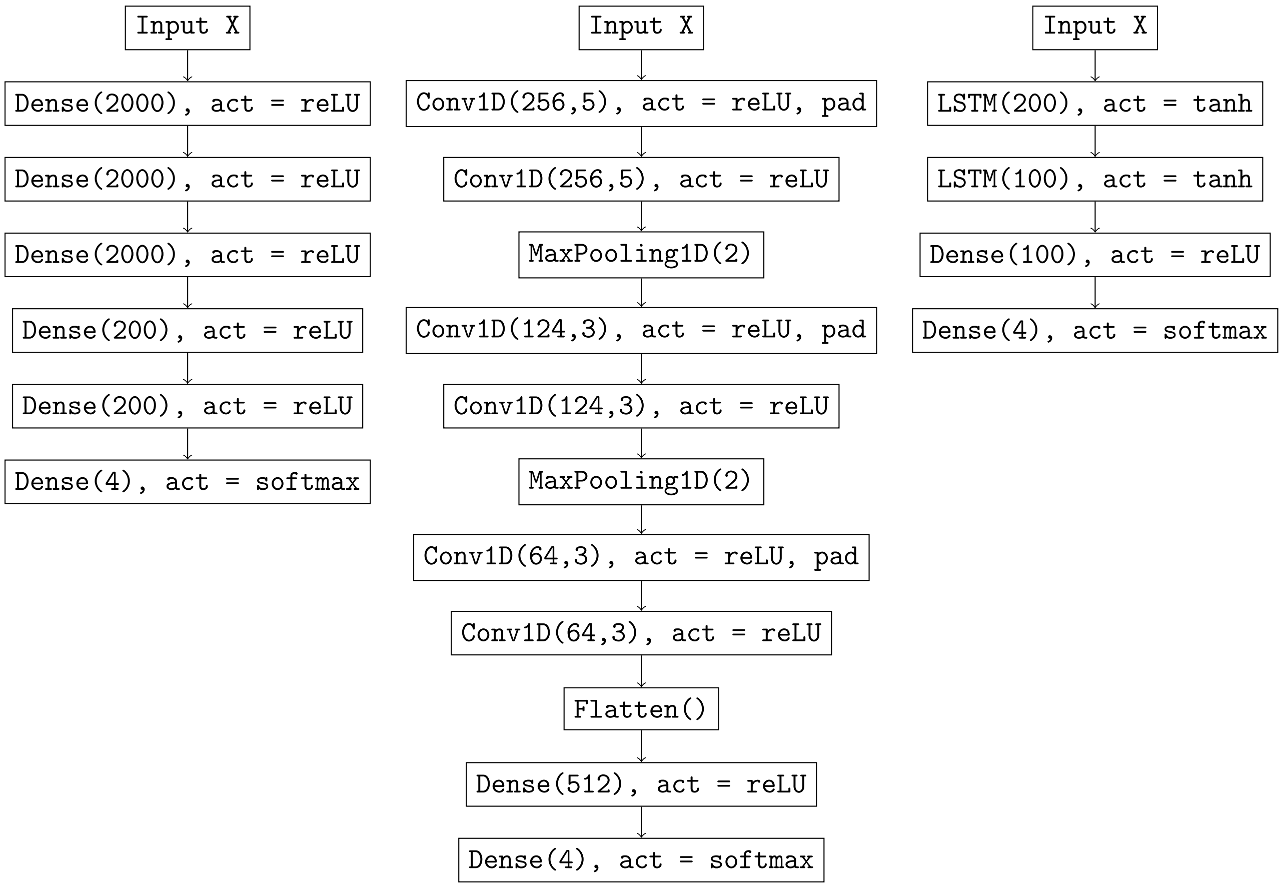


Figure A.6: Models used for the AR1000 dataset. Left: fully connected model with 10 447 204 weights. Middle: CNN model with 8 285 828 weights. Right: LSTM model with 292 504 weights.

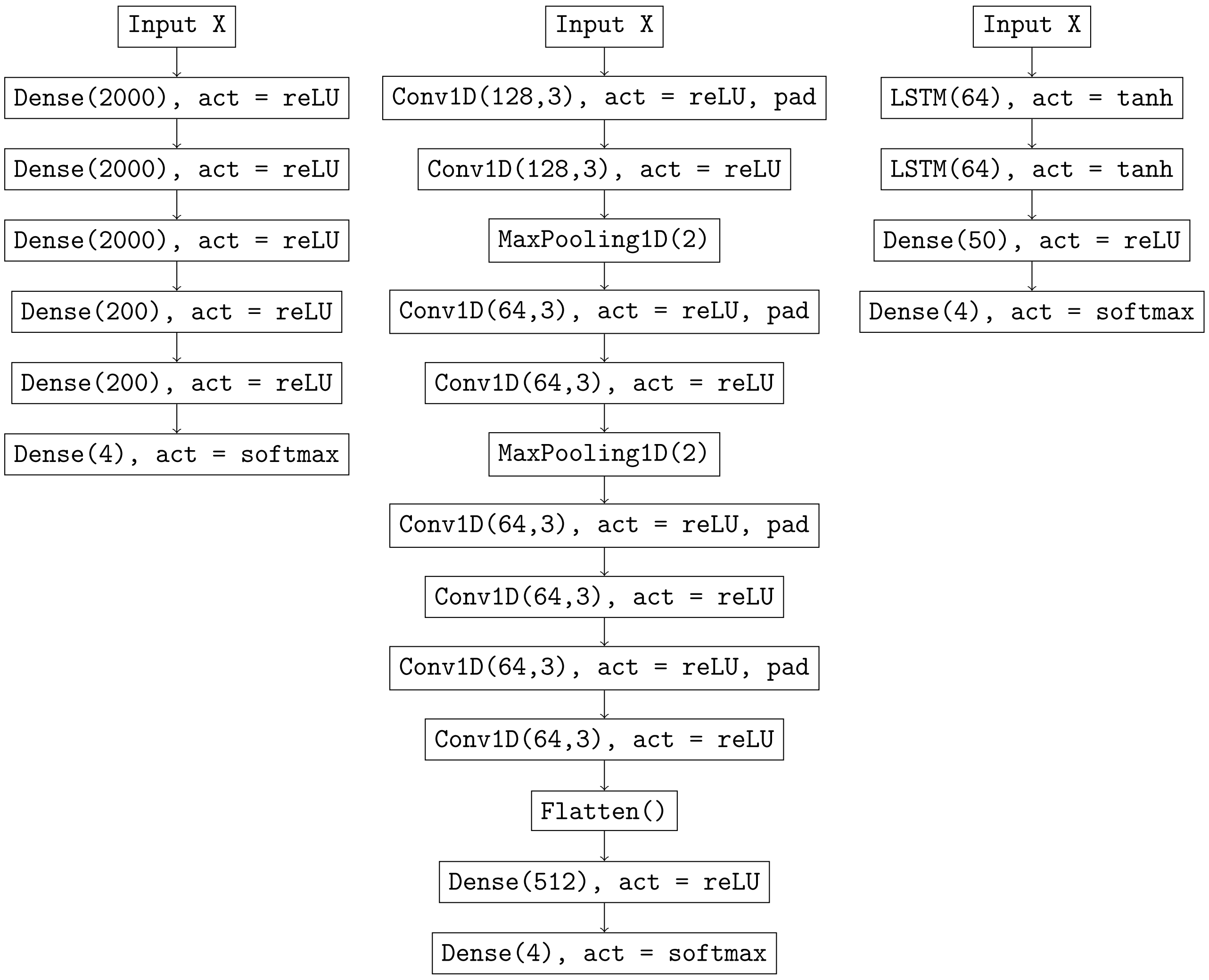


Figure A.7: Models used for the MA1000 dataset. Left: fully connected model with 10 447 204 weights. Middle: CNN model with 8 134 148 weights. Right: LSTM model with 53 374 weights.

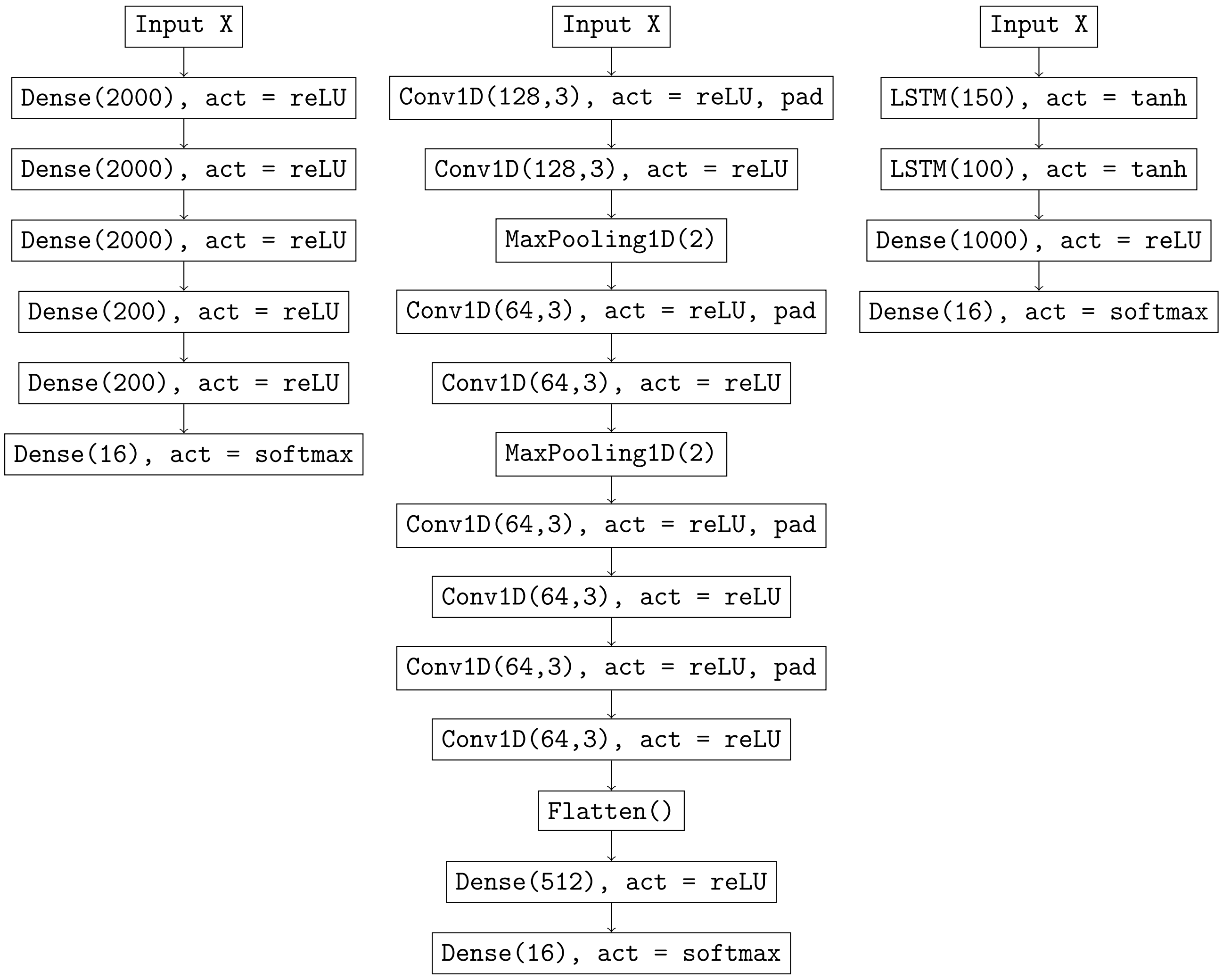


Figure A.8: Models used for the ARMA1000 dataset. Left: fully connected model with 10 449 616 weights. Middle: CNN model with 8 140 304 weights. Right: LSTM model with 308 616 weights.

A.2. Tables

Tables A.1-A.9 show the accuracy scores of the fully connected (NNET), convolutional (CNN), LSTM model, AIC and BIC for each dataset. Highest accuracy scores are in bold and scores below the uniform dummy classifier are in red.

Table A.1: Accuracy scores for the AR30 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1	0.880	0.863	0.875	0.755	**0.906**
2	0.554	0.593	0.604	0.592	**0.607**
3	0.502	0.457	**0.503**	0.461	0.398
4	**0.548**	0.484	0.544	0.426	0.294
Average	0.618	0.598	**0.630**	0.558	0.549

Table A.2: Accuracy scores for the MA30 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1	0.840	0.836	0.827	0.767	**0.908**
2	0.543	**0.580**	0.551	0.343	0.332
3	0.422	0.356	**0.448**	0.289	0.238
4	0.509	0.434	**0.511**	0.157	0.084
Average	0.576	0.550	**0.582**	0.386	0.386

Table A.3: Accuracy scores for the ARMA30 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1,1	**0.742**	0.741	0.738	0.492	0.701
1,2	0.305	0.315	0.316	0.278	**0.330**
1,3	0.244	0.205	**0.272**	0.133	0.094
1,4	**0.311**	0.256	0.254	0.104	0.061
2,1	0.314	0.300	0.324	0.371	**0.464**
2,2	0.093	0.102	0.117	**0.203**	0.184
2,3	0.091	0.084	0.062	**0.093**	0.056
2,4	0.063	0.037	**0.091**	0.057	0.020
3,1	**0.267**	0.235	0.248	0.151	0.148
3,2	0.079	0.060	0.102	**0.115**	0.088
3,3	**0.073**	0.054	0.050	0.034	0.014
3,4	0.061	0.052	**0.099**	0.029	0.011
4,1	0.230	**0.275**	0.272	0.191	0.158
4,2	0.057	0.049	0.083	**0.098**	0.056
4,3	**0.082**	0.068	0.060	0.048	0.016
4,4	**0.113**	0.096	0.112	0.025	0.007
Average	0.194	0.182	**0.199**	0.150	0.149

Table A.4: Accuracy scores for the AR100 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1	**0.966**	0.946	0.956	0.763	0.965
2	0.779	0.768	0.800	0.732	**0.817**
3	0.697	0.706	**0.754**	0.704	0.716
4	0.722	0.755	0.782	**0.827**	0.705
Average	0.790	0.792	**0.822**	0.756	0.800

Table A.5: Accuracy scores for the MA100 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1	0.912	0.955	0.934	0.816	**0.969**
2	0.748	0.728	**0.770**	0.451	0.488
3	0.696	0.623	**0.702**	0.420	0.402
4	0.716	0.689	**0.737**	0.291	0.241
Average	0.767	0.747	**0.785**	0.492	0.521

Table A.6: Accuracy scores for the ARMA100 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1,1	0.839	0.824	**0.849**	0.424	0.828
1,2	0.434	0.462	0.521	0.354	**0.566**
1,3	0.426	0.424	**0.454**	0.281	0.315
1,4	0.411	0.422	**0.499**	0.308	0.310
2,1	0.394	0.415	0.510	0.390	**0.612**
2,2	0.191	0.255	0.308	0.313	**0.406**
2,3	0.126	0.160	0.249	**0.250**	0.237
2,4	0.063	0.122	0.243	**0.260**	0.204
3,1	0.379	0.360	**0.464**	0.305	0.362
3,2	0.135	0.162	0.242	**0.266**	**0.266**
3,3	0.067	0.082	0.167	**0.178**	0.116
3,4	0.119	0.146	0.186	0.212	**0.122**
4,1	0.308	0.390	**0.461**	0.346	0.383
4,2	0.090	0.161	0.250	**0.299**	0.241
4,3	0.099	0.063	0.197	**0.227**	0.139
4,4	0.111	0.124	0.239	**0.248**	0.123
Average	0.261	0.285	**0.364**	0.291	0.327

Table A.7: Accuracy scores for the AR1000 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1	0.930	0.991	0.987	0.764	**0.994**
2	0.635	0.952	0.971	0.780	**0.980**
3	0.583	0.917	0.966	0.832	**0.980**
4	0.514	0.935	0.969	**0.993**	0.979
Average	0.664	0.949	0.973	0.843	**0.983**

Table A.8: Accuracy scores for the MA1000 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1	0.923	0.989	**0.991**	0.810	**0.991**
2	0.782	0.943	**0.974**	0.503	0.609
3	0.449	0.921	**0.964**	0.501	0.561
4	0.762	0.917	**0.967**	0.363	0.361
Average	0.729	0.942	**0.974**	0.543	0.629

Table A.9: Accuracy scores for the ARMA1000 dataset.

Order	NNET	CNN	LSTM	AIC	BIC
1,1	0.737	0.910	0.924	0.284	**0.946**
1,2	0.411	0.698	0.803	0.352	**0.831**
1,3	0.379	0.597	**0.765**	0.497	0.696
1,4	0.422	0.578	**0.759**	0.581	0.733
2,1	0.357	0.700	0.814	0.373	**0.843**
2,2	0.107	0.445	0.674	0.394	**0.737**
2,3	0.067	0.353	0.619	0.509	**0.626**
2,4	0.127	0.332	0.578	0.590	**0.648**
3,1	0.212	0.597	**0.769**	0.508	0.697
3,2	0.074	0.327	0.558	0.500	**0.629**
3,3	0.057	0.208	0.502	0.492	**0.506**
3,4	0.098	0.150	0.478	**0.582**	0.529
4,1	0.183	0.595	**0.748**	0.607	0.742
4,2	0.089	0.174	0.596	0.618	**0.666**
4,3	0.081	0.181	0.474	**0.598**	0.558
4,4	0.104	0.218	0.506	**0.751**	0.562
Average	0.218	0.440	0.660	0.515	**0.684**

A.3. Distribution of the order selections

Figures A.9 - A.17 show the distribution of order selections for the best performing neural network, AIC and BIC for each dataset.

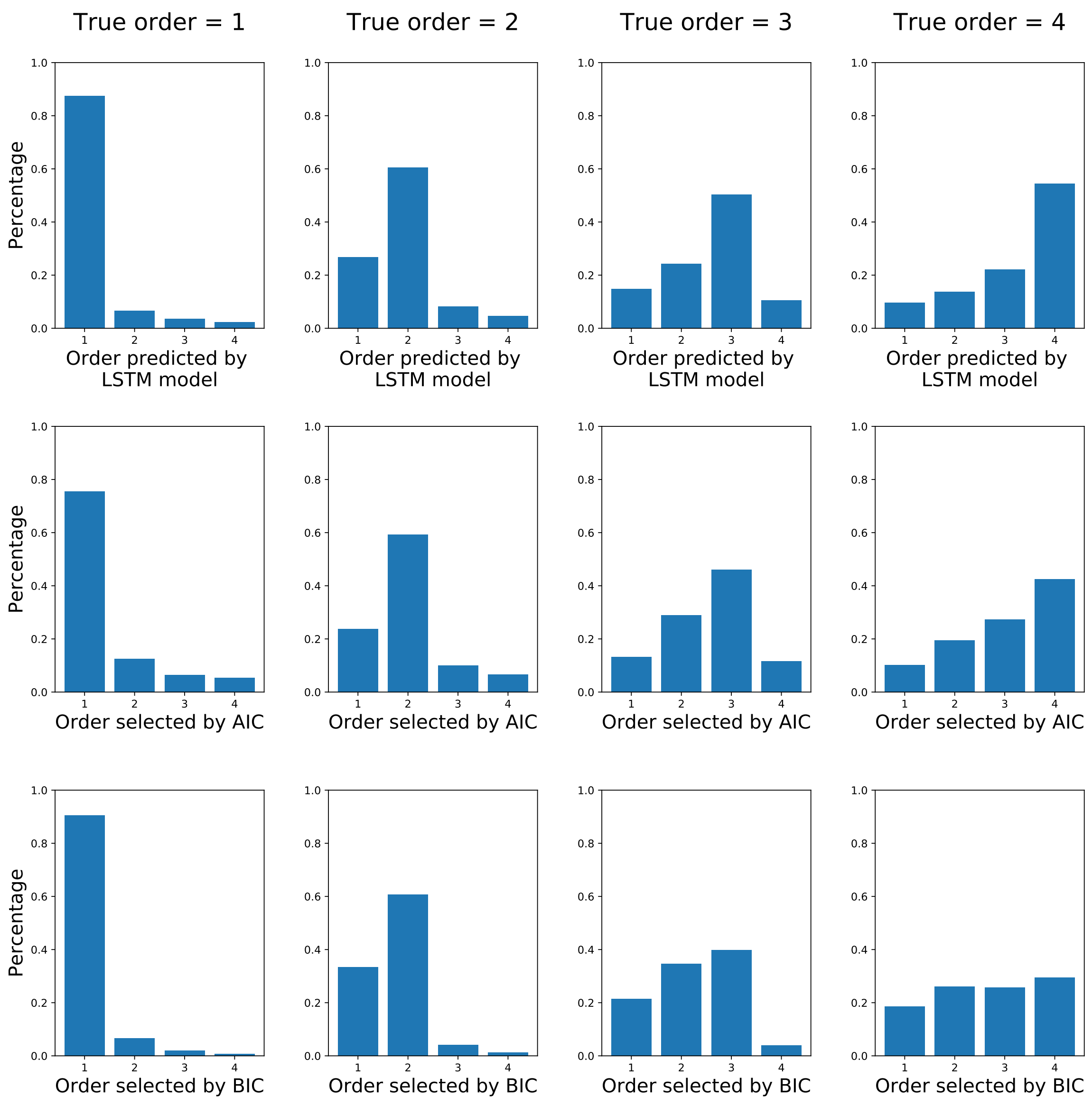


Figure A.9: Distributions for the AR30 dataset.

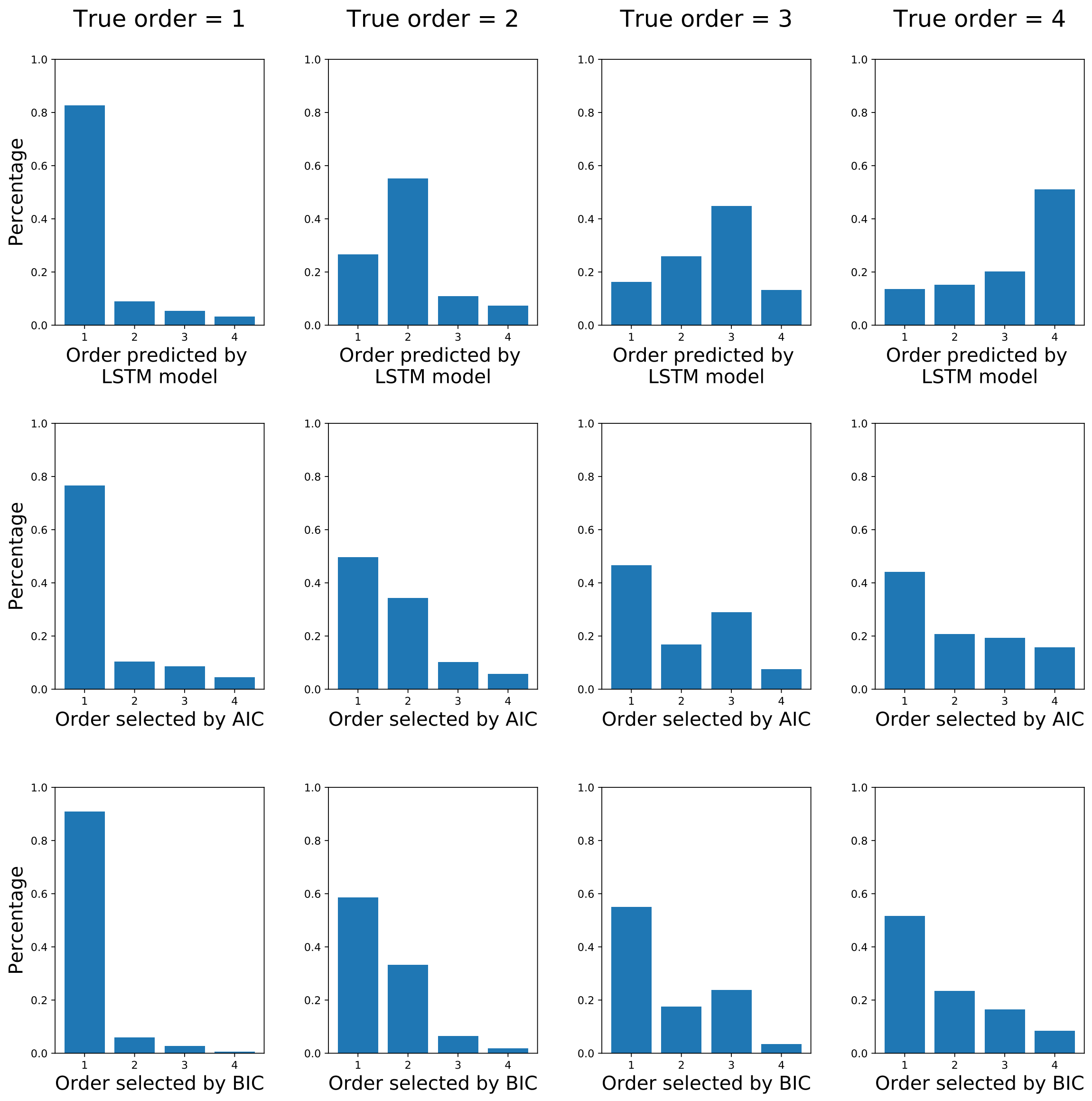
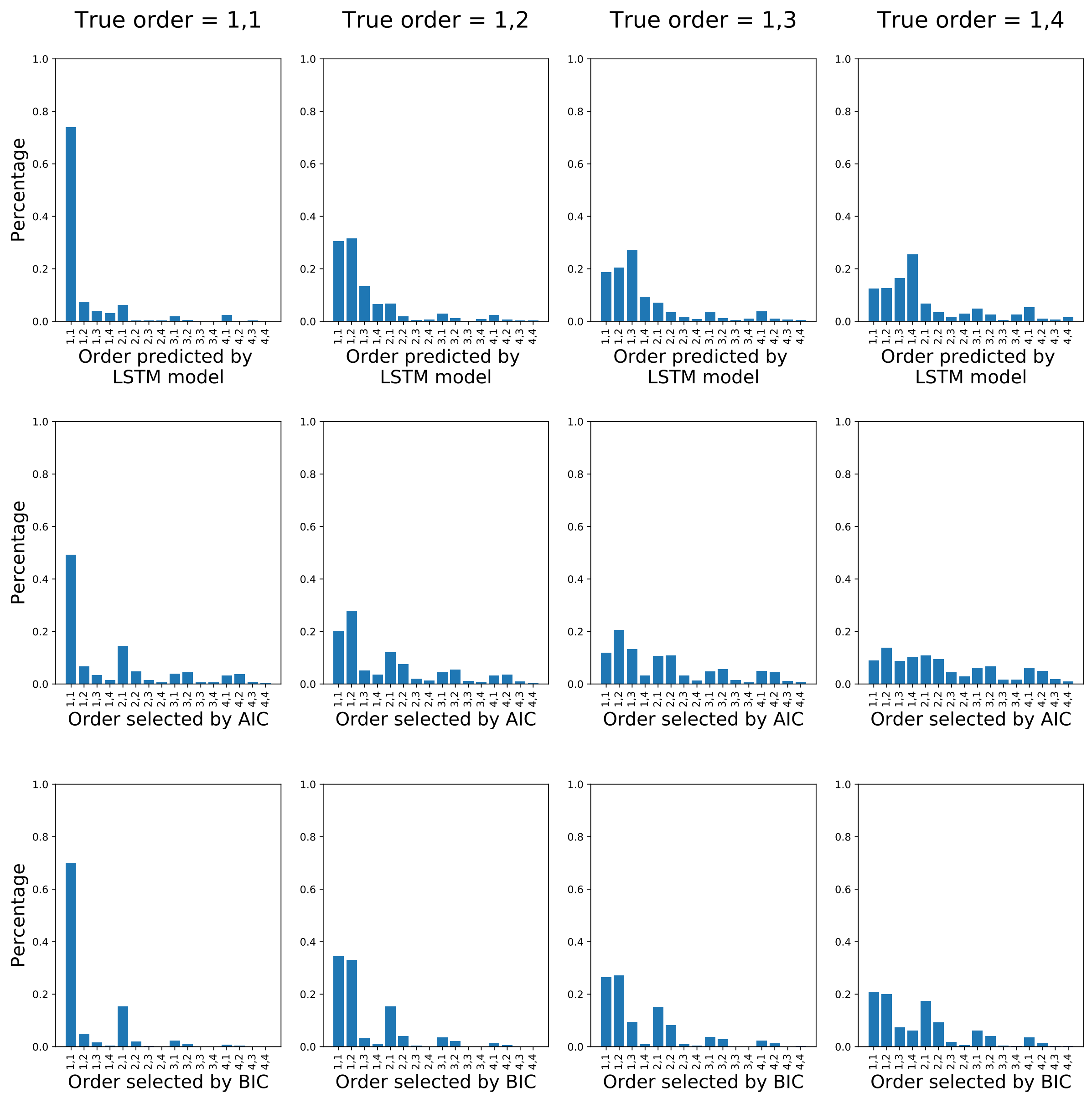
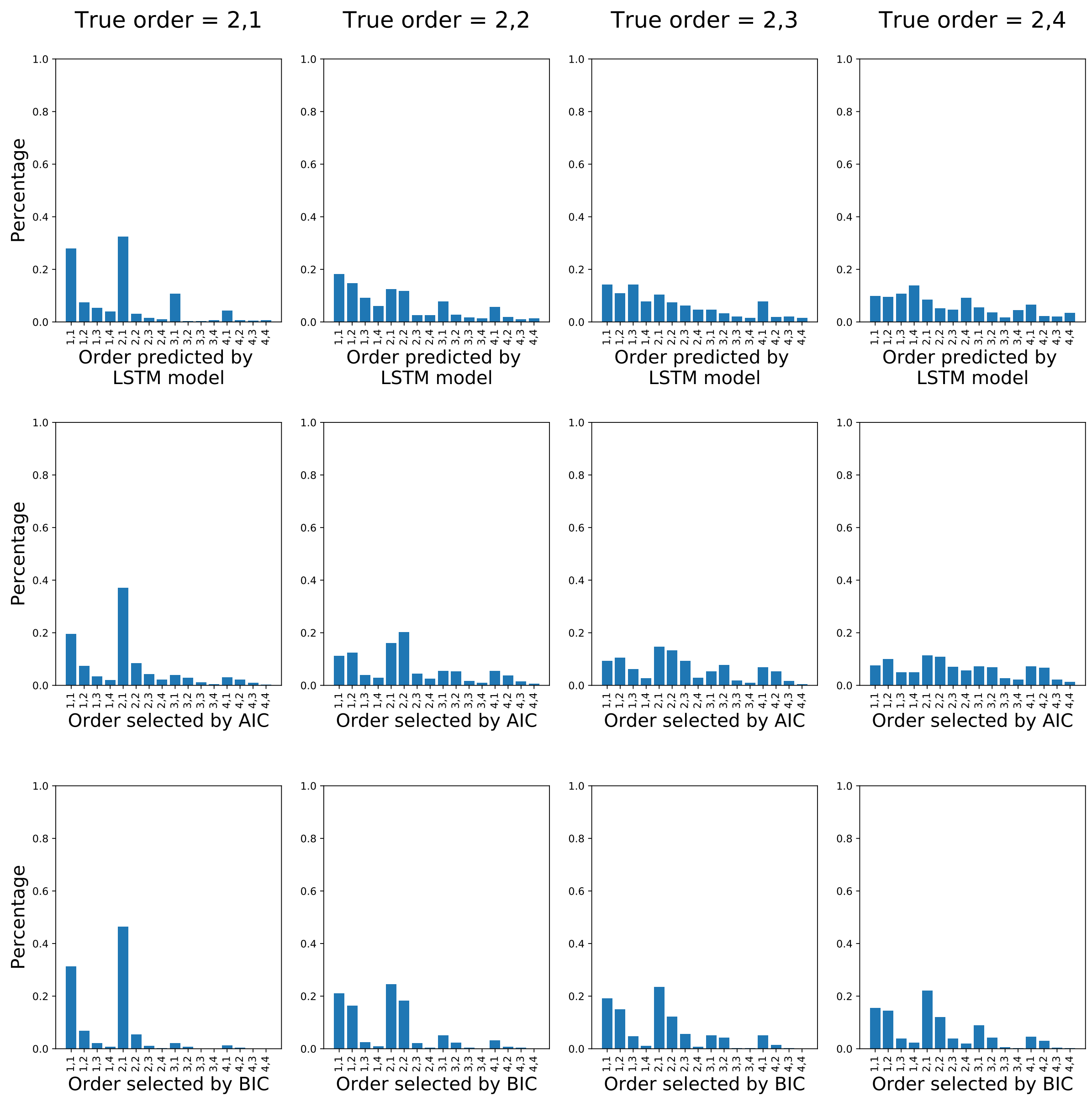


Figure A.10: Distributions for the MA30 dataset.



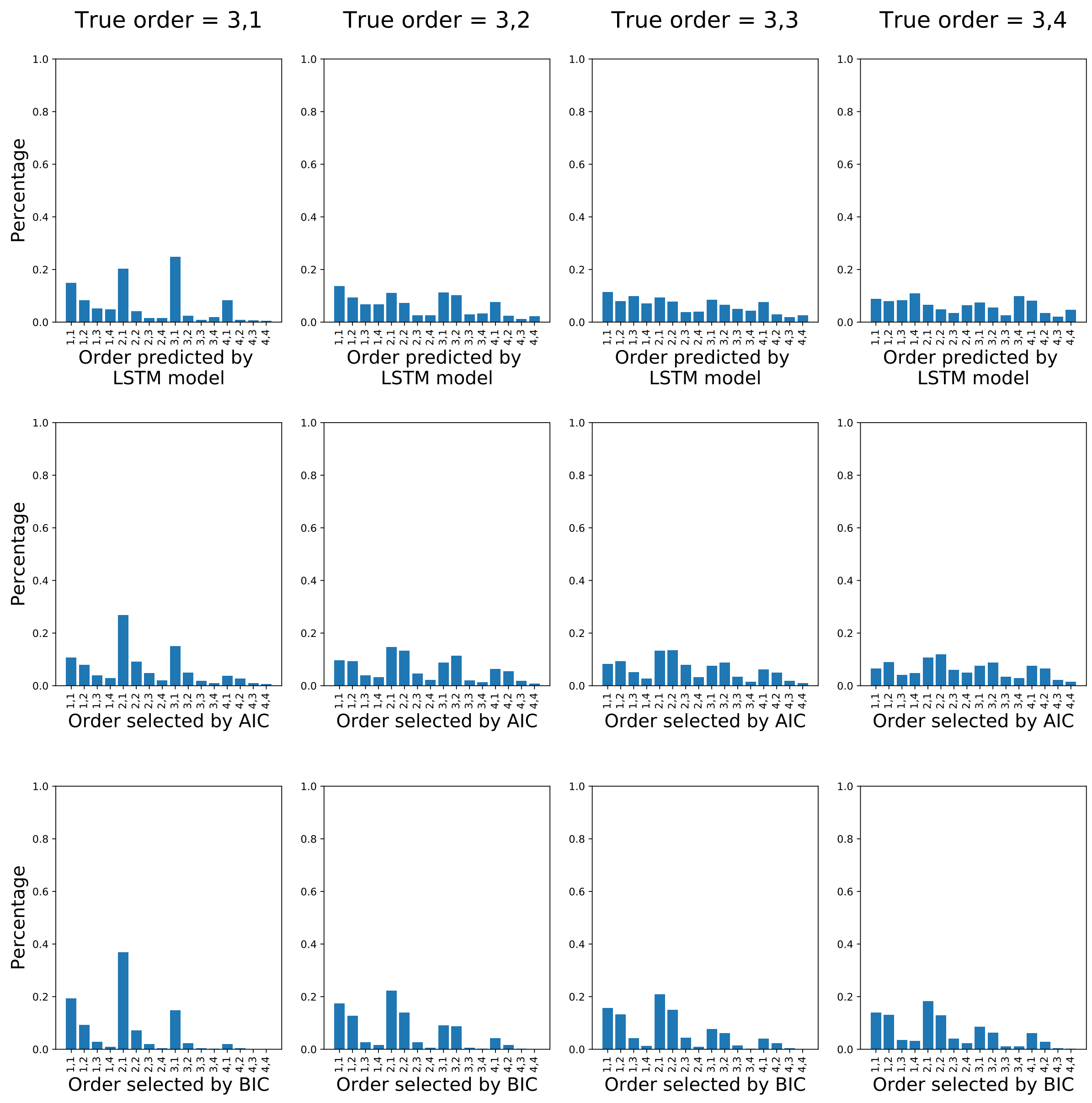
(a) Order (1,1) to (1,4).

Figure A.11: Distributions for the ARMA30 dataset.



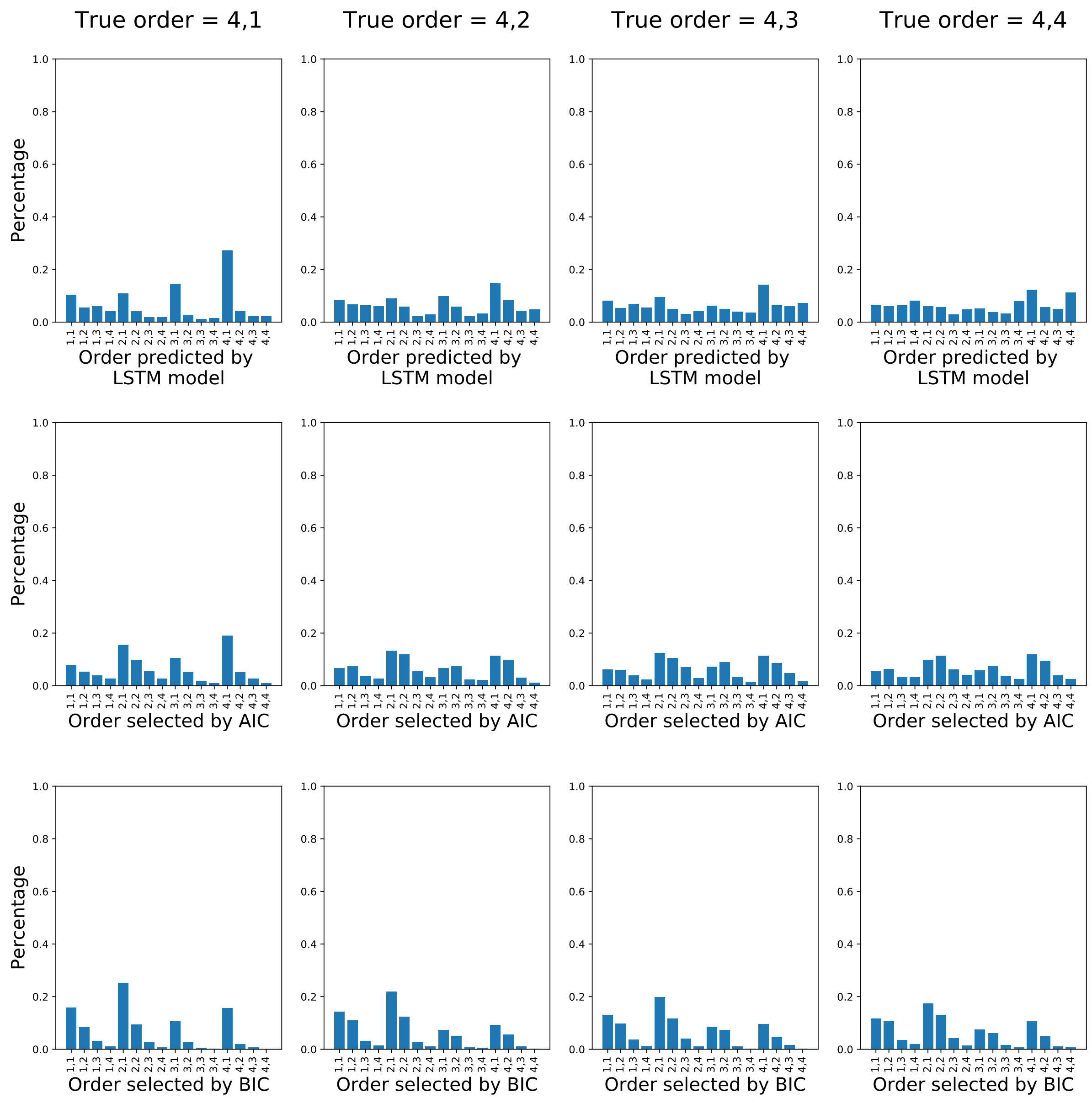
(b) Order (2,1) to (2,4).

Figure A.11: Distributions for the ARMA30 dataset.



(c) Order (3,1) to (3,4).

Figure A.11: Distributions for the ARMA30 dataset.



(d) Order (4,1) to (4,4).

Figure A.11: Distributions for the ARMA30 dataset.

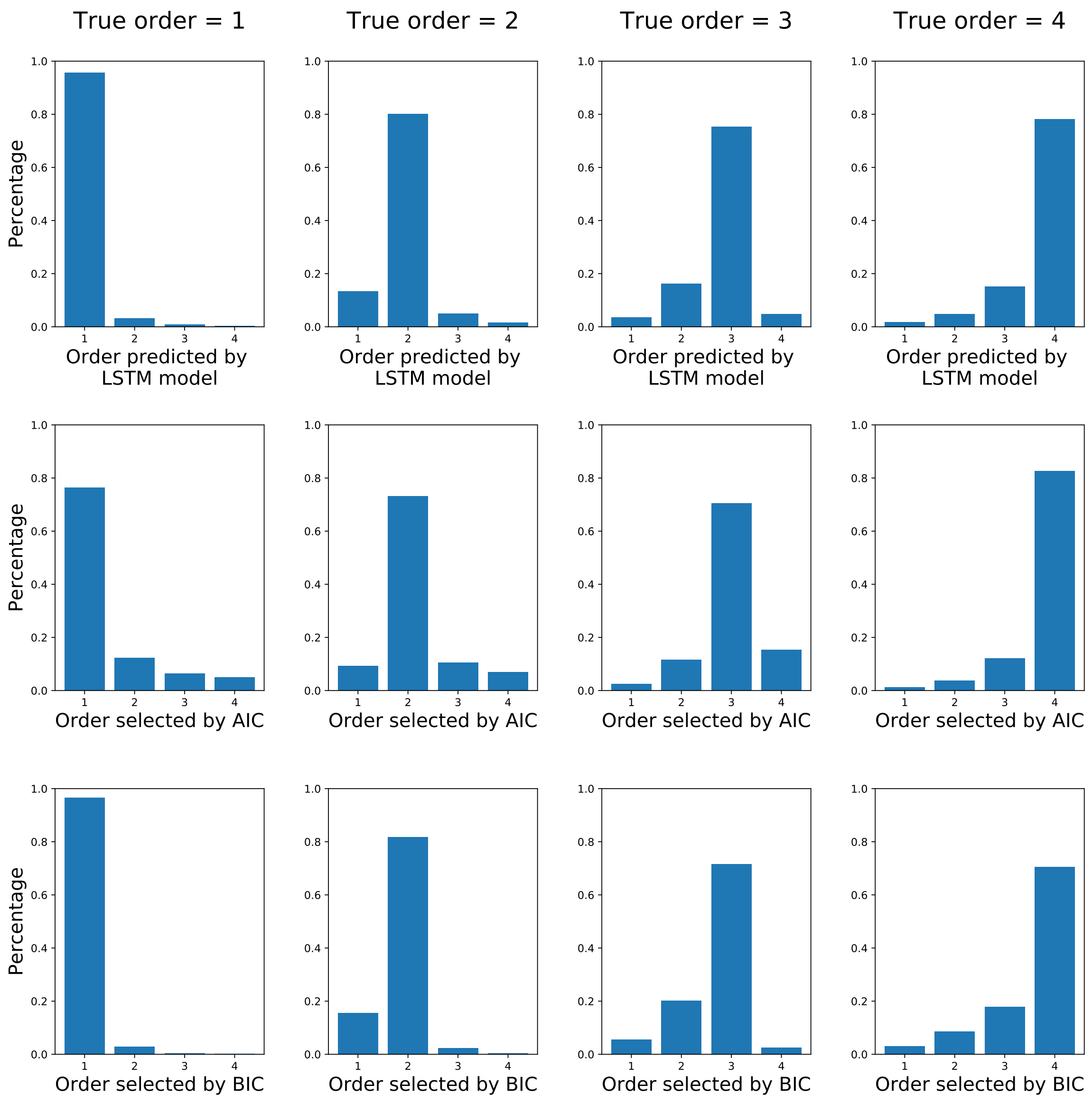


Figure A.12: Distributions for the AR100 dataset.

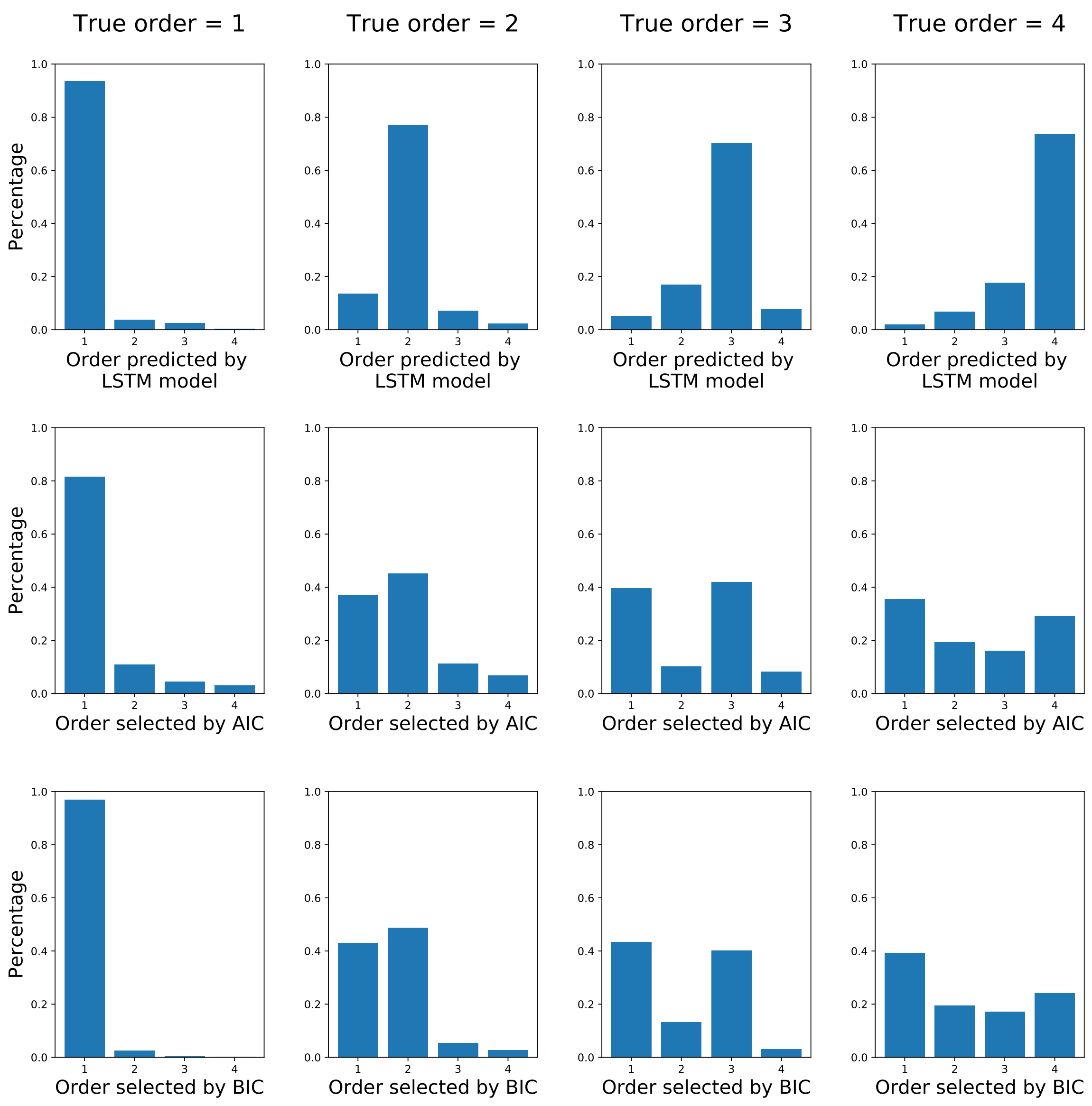
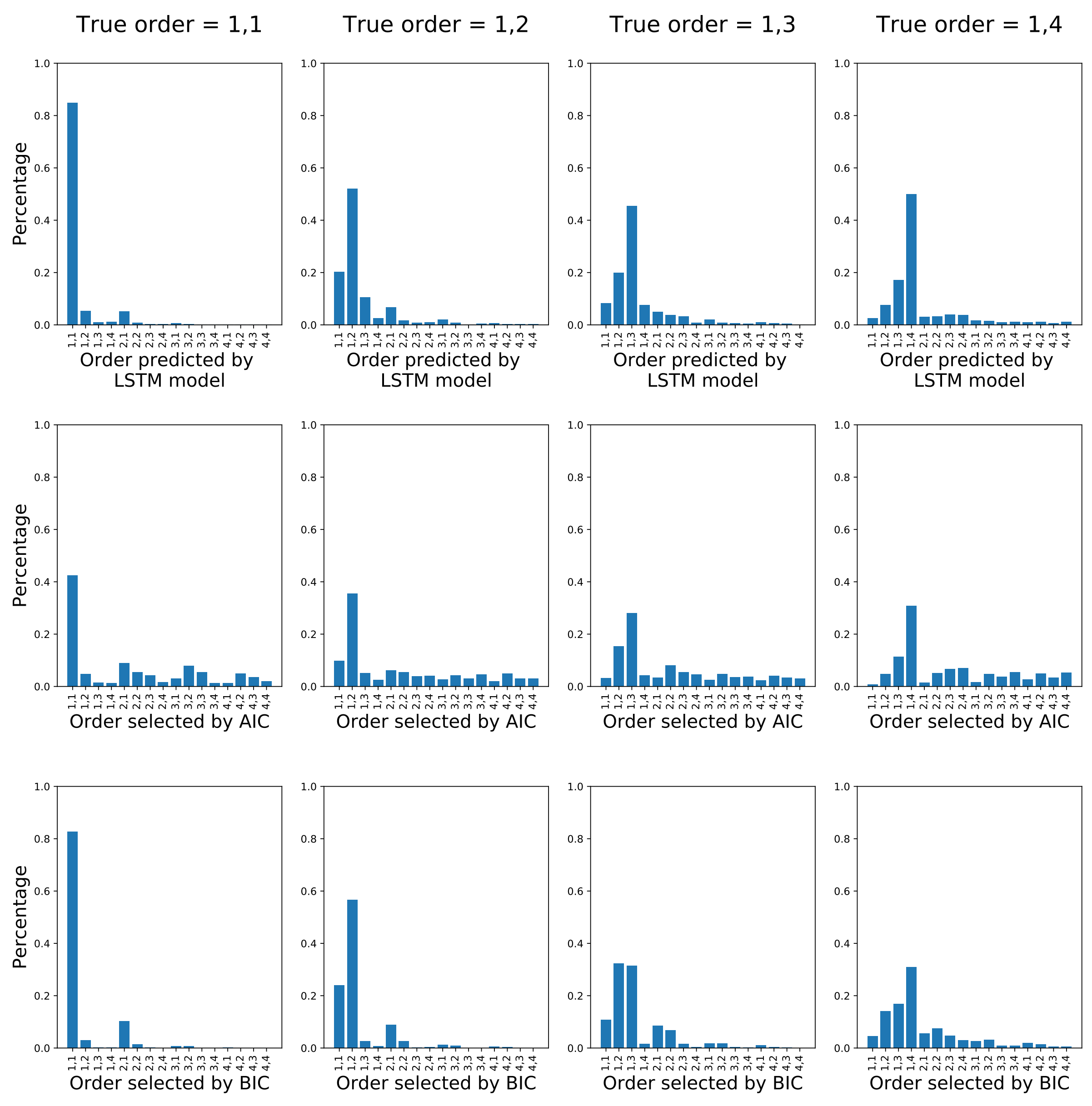
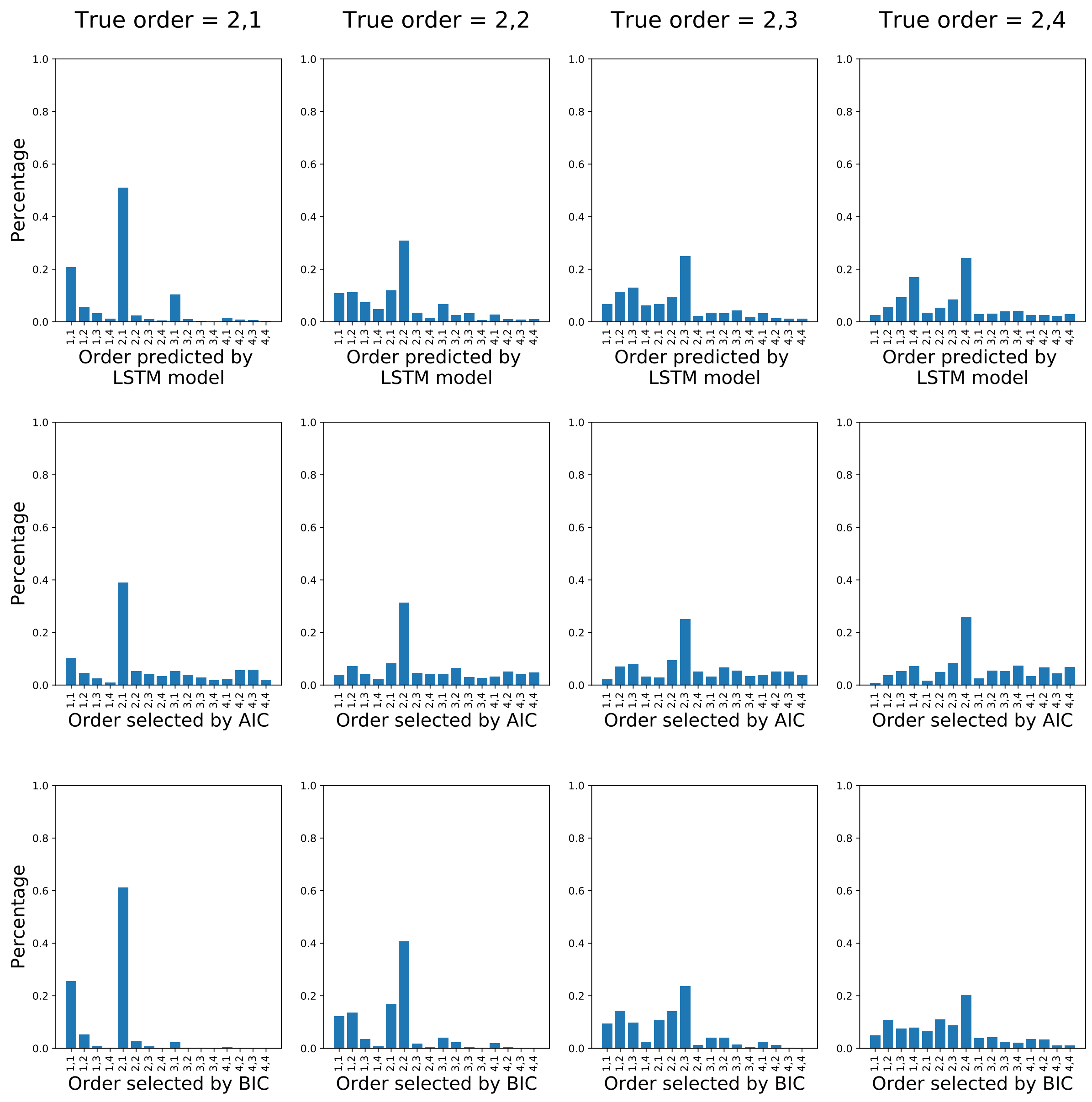


Figure A.13: Distributions for the MA100 dataset.



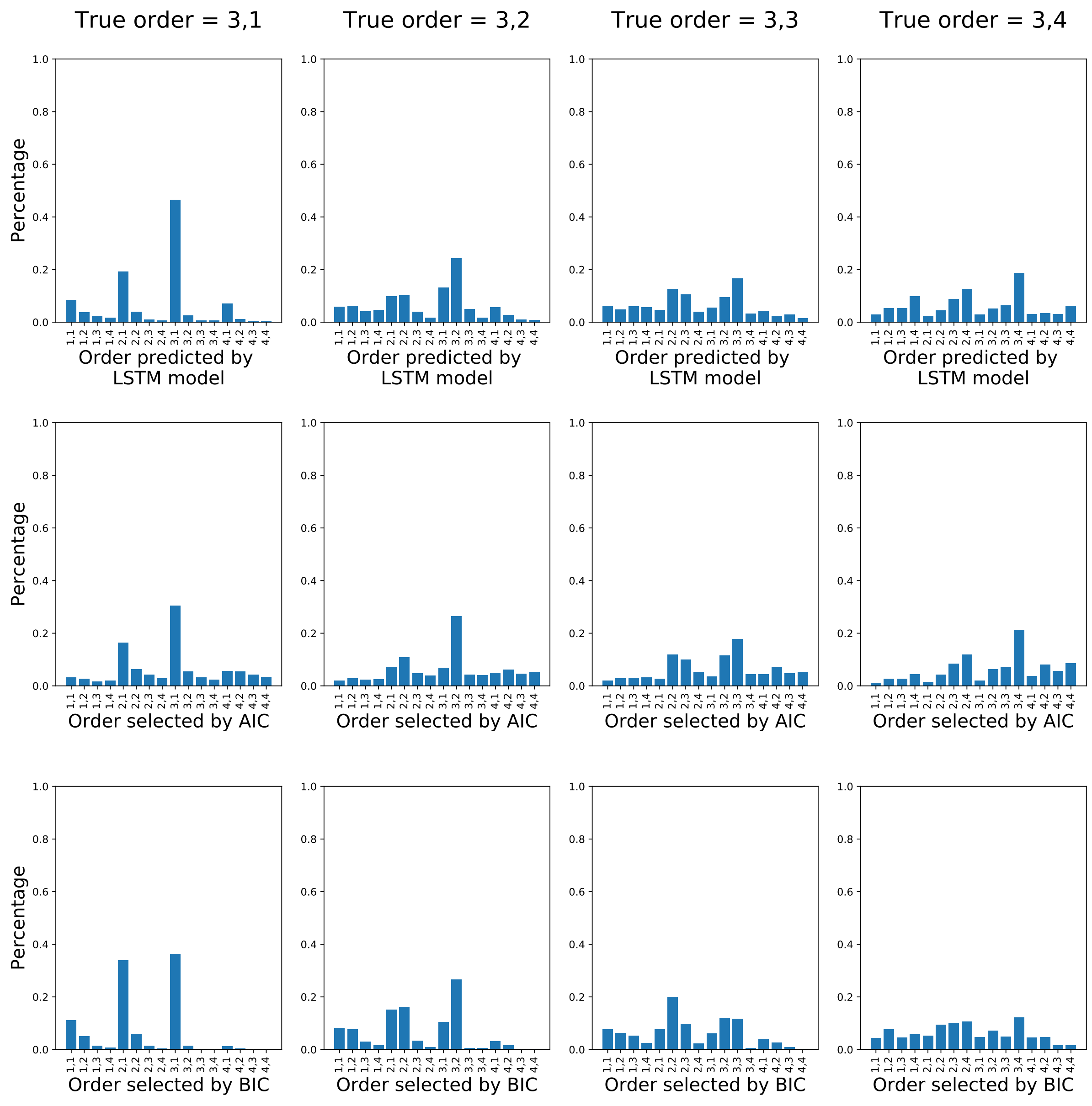
(a) Order (1,1) to (1,4).

Figure A.14: Distributions for the ARMA100 dataset.



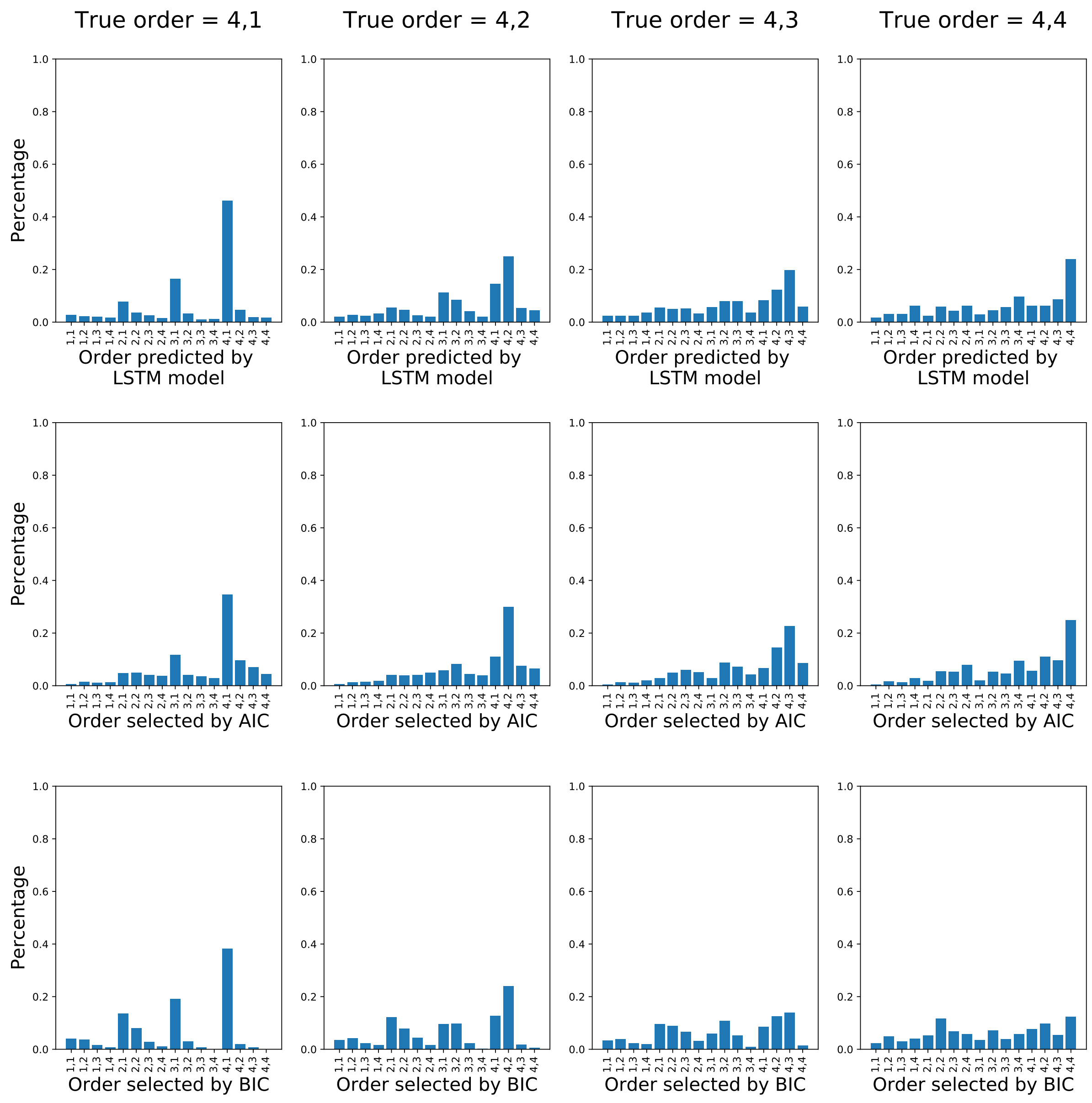
(b) Order (2,1) to (2,4).

Figure A.14: Distributions for the ARMA100 dataset.



(c) Order (3,1) to (3,4).

Figure A.14: Distributions for the ARMA100 dataset.



(d) Order (4,1) to (4,4).

Figure A.14: Distributions for the ARMA100 dataset.

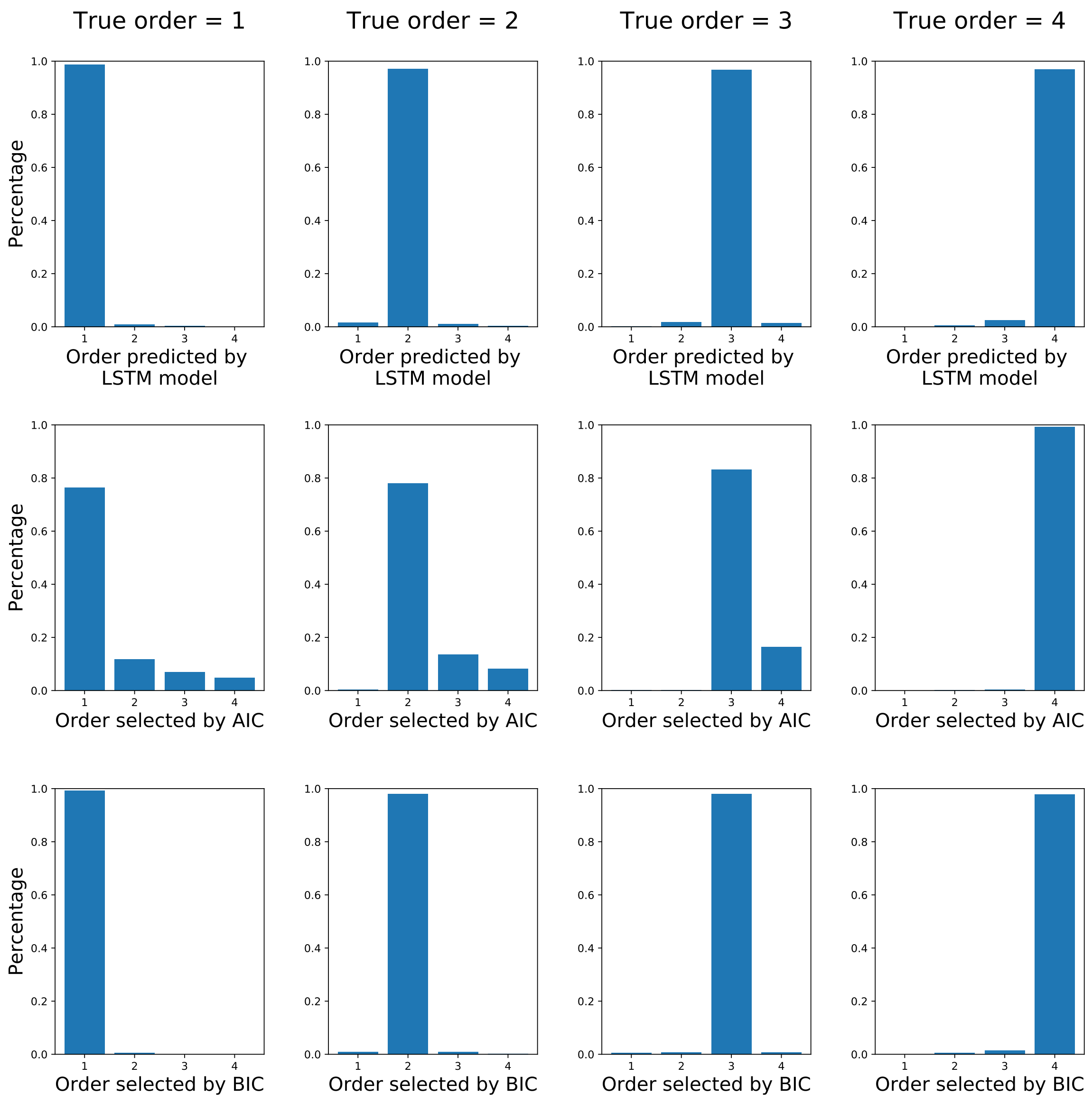


Figure A.15: Distributions for the AR1000 dataset.

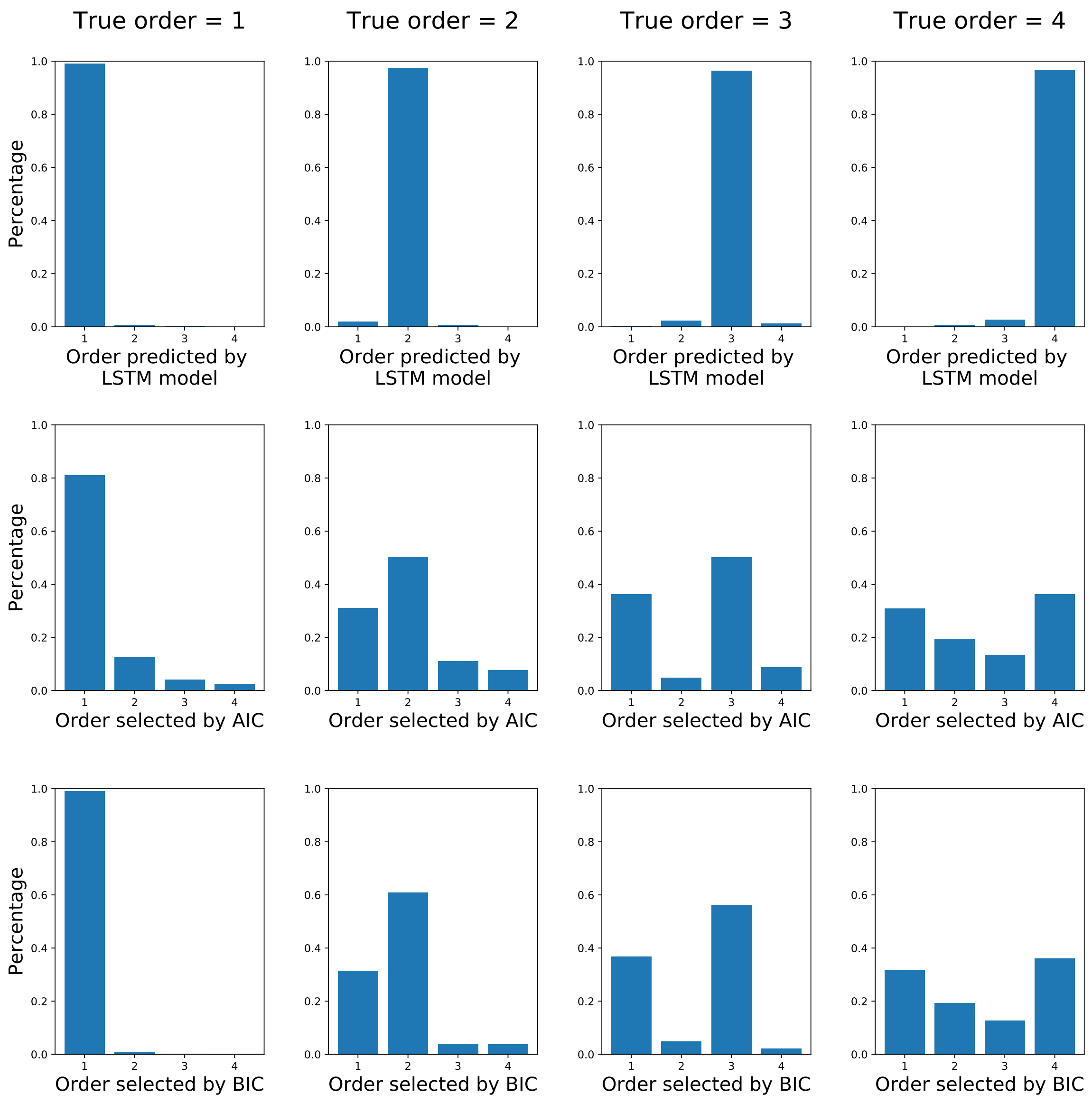
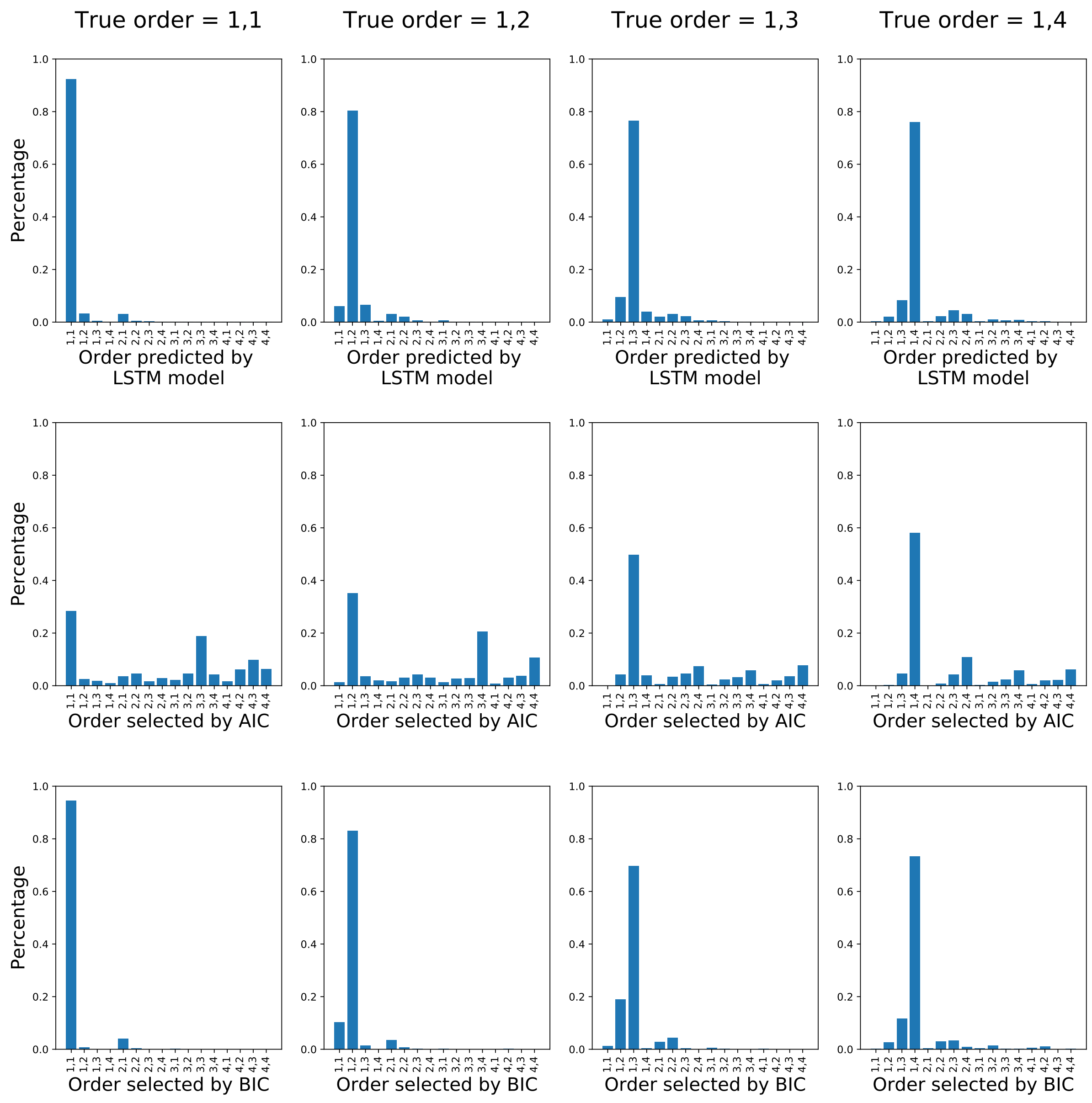
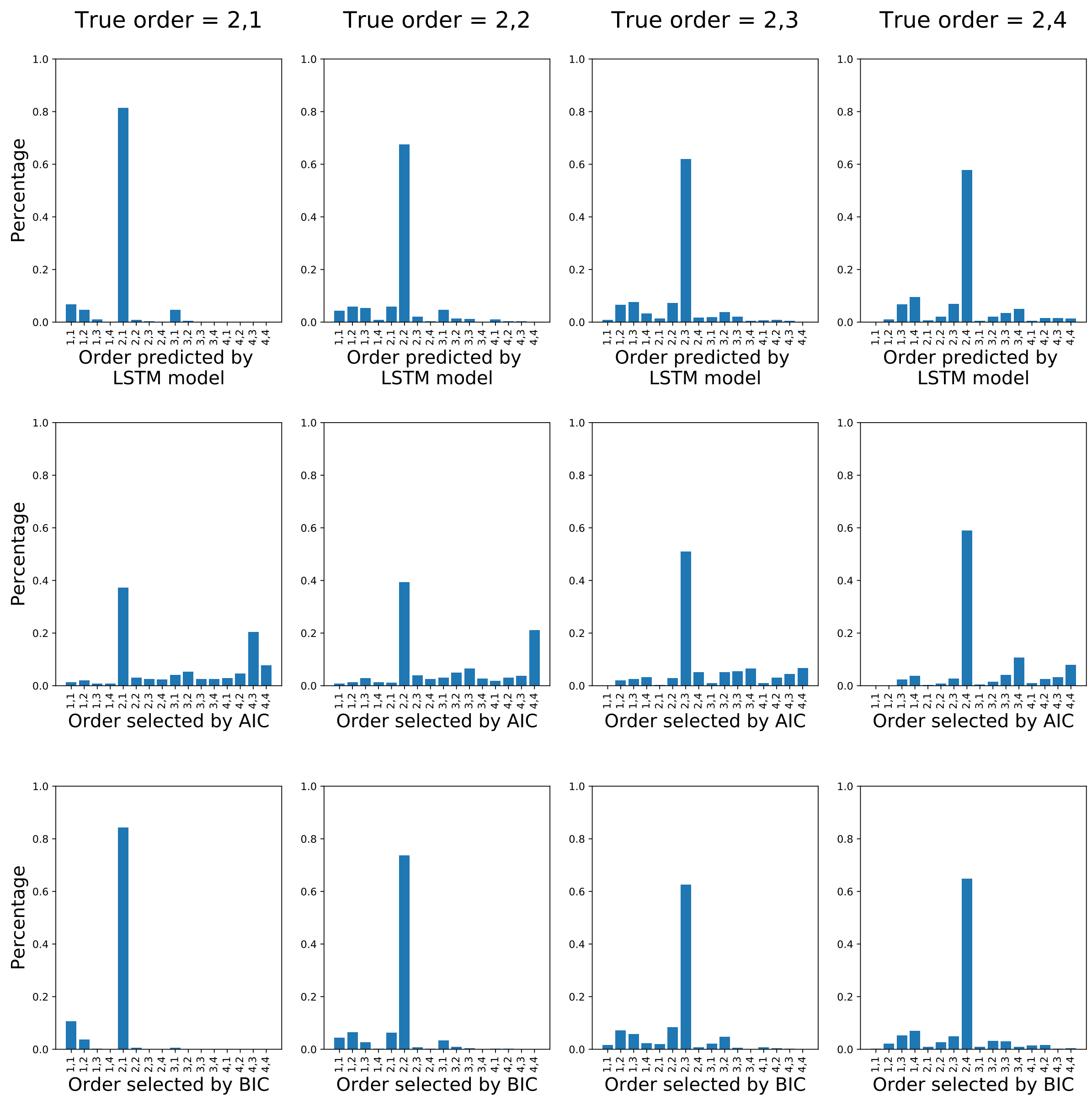


Figure A.16: Distributions for the MA1000 dataset.



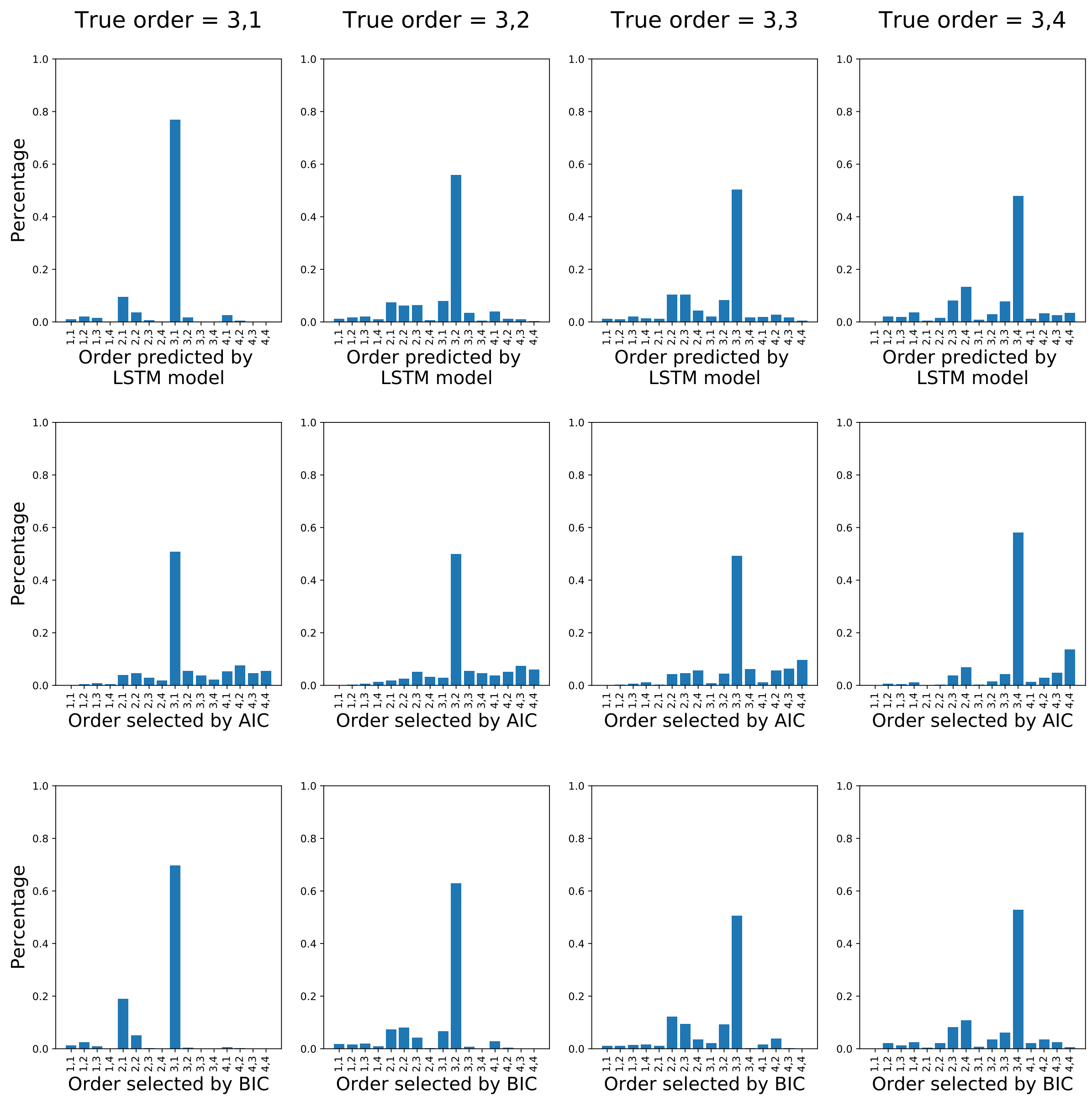
(a) Order (1,1) to (1,4).

Figure A.17: Distributions for the ARMA1000 dataset.



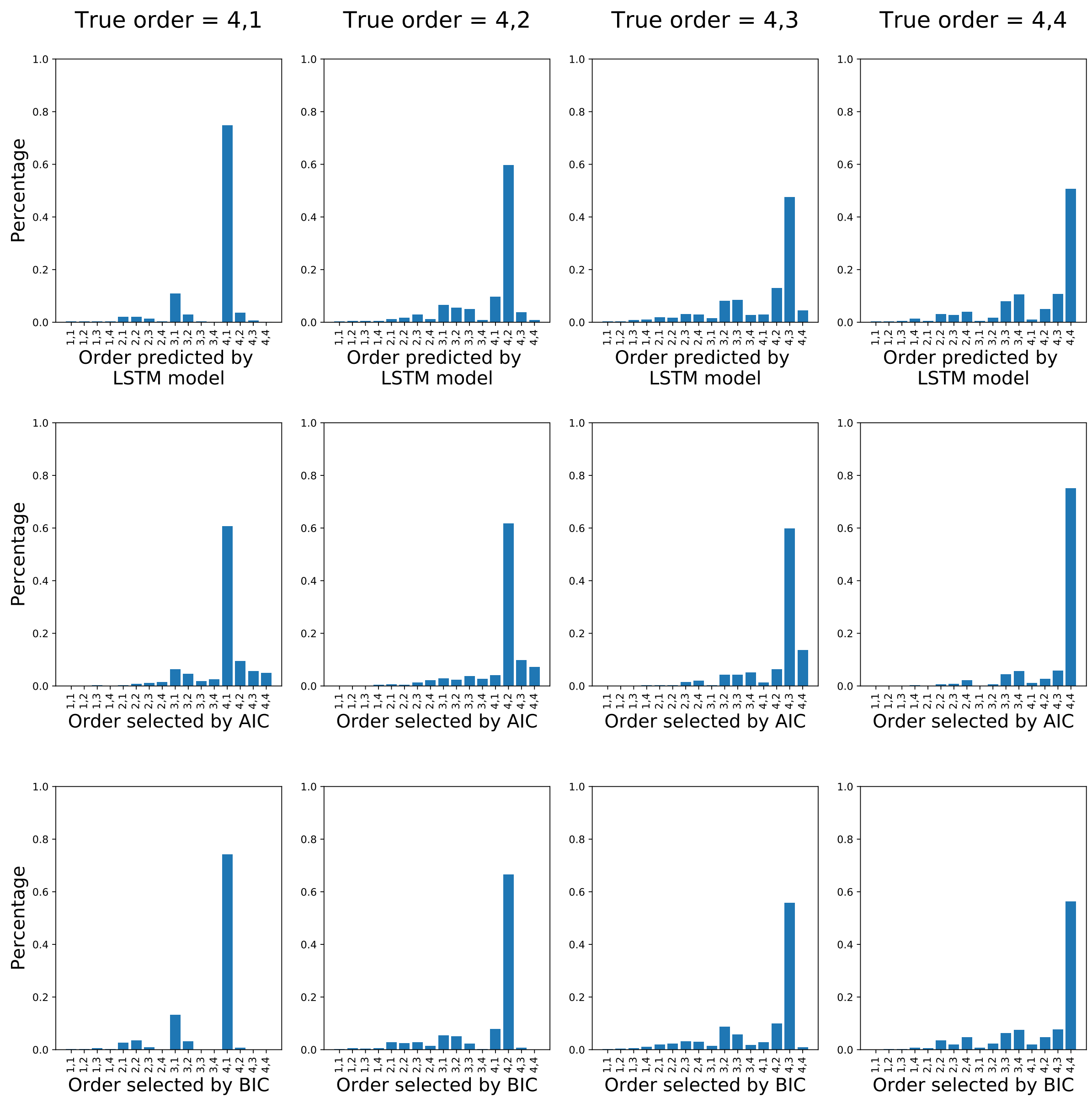
(b) Order (2,1) to (2,4).

Figure A.17: Distributions for the ARMA1000 dataset.



(c) Order (3,1) to (3,4).

Figure A.17: Distributions for the ARMA1000 dataset.



(d) Order (4,1) to (4,4).

Figure A.17: Distributions for the ARMA1000 dataset.

Appendix B

Technical details

B.1. Tensorflow and Keras

Tensorflow is Google's open-source library for creating and developing deep learning models. It was originally developed by researchers in the Google Brain project in the year 2015 (Abadi et al., 2015). Tensorflow supports parallelization and can be used on a variety of systems, such as GPU cards and large scale distributed systems. If a GPU card is available, Tensorflow will detect this automatically and utilize it with no changes needed to the code, as long as the appropriate software requirements are met. GPU computation can speed up training significantly compared to using CPU alone.

Since deep learning is an optimization problem and often requires the computation of many gradients, Tensorflow has implemented automatic differentiation. This is different from symbolic and numerical differentiation and eliminates problems with the classical approaches such as round off errors and slow computation of partial derivatives with many inputs.

Keras is a high-level deep learning API that runs on top of Tensorflow (Chollet et al., 2015). The interface is user-friendly and lets the user experiment with different types of neural network easily and efficiently.

B.2. Machines

In this thesis, we used two different devices to train models. One of them is a machine with 64 logical cores in total, and the other is a machine with an Nvidia Geforce RTX 2080 Ti GPU card.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Hirotugu Akaike. A new look at the statistical model identification. *IEEE transactions on automatic control*, 19(6):716–723, 1974.

Khaled E. Al-Qawasmi, Adnan M. Al-Smadi, and Alaa Al-Hamami. Artificial neural network-based algorithm for ARMA model order estimation. In *International Conference on Networked Digital Technologies*, pages 184–192. Springer, 2010.

Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting.* Springer, 2016.

Tim Chenoweth, Robert Hubata, and Robert D St Louis. Automatic ARMA identification using neural networks and the extended sample autocorrelation function: a reevaluation. *Decision Support Systems*, 29(1):21–30, 2000.

François Chollet et al. Keras. `https://keras.io`, 2015.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

Bradley Efron and Trevor Hastie. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*, volume 5. Cambridge University Press, 2016.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Alex Graves. *Supervised sequence labelling with recurrent neural networks*. Springer, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman. 1D convolutional neural networks and applications: A survey. *arXiv preprint arXiv:1905.03554*, 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Gang Liang, D Mitchell Wilkes, and James A. Cadzow. ARMA model order estimation based on the eigenvalues of the covariance matrix. *IEEE transactions on signal processing*, 41(10): 3003–3009, 1993.

Gideon Schwarz. Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464, 1978.

Skipper Seabold and Josef Perktold. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. A comparison of ARIMA and LSTM in forecasting time series. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1394–1401. IEEE, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Gilbert Strang. *Linear algebra and its applications*. Thomson, Brooks/Cole, 2006.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, 2012.

Ruey S Tsay and George C Tiao. Consistent estimates of autoregressive parameters and extended sample autocorrelation function for stationary and nonstationary ARMA models. *Journal of the American Statistical Association*, 79(385):84–96, 1984.

Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320, 2005.