DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development of the Bayesian Recurrent Neural Network Architectures for Hydrological Time Series Forecasting

Jonas Fill

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development of the Bayesian Recurrent Neural Network Architectures for Hydrological Time Series Forecasting

Entwicklung Bayesscher Rekurrenter Neuronaler Netz-Architekturen für zeitfolgenbasierte Prognosen in der Hydrologie

Author:	Jonas Fill
Supervisor:	Hans-Joachim Bungartz, Univ.-Prof. Dr.
Advisor:	Ivana Jovanovic Buha, M.Sc. (hons)
Submission Date:	15 July 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15 July 2021 Jonas Fill

Acknowledgments

First of all, I would like to thank my advisor Ivana Jovanovic Buha, M.Sc. (hons), who not only suggested this promising topic to me, but also patiently advised and motivated me throughout my programming and writing. Overall, it has not only been a bachelor's thesis for me, but a great opportunity to gain experience in the field of deep learning for hydrology. Special thanks also go to my family and friends who supported and encouraged me throughout the process.

Abstract

In the field of rainfall-runoff modeling, one is interested in high quality predictions of future discharge given past meteorological data (forcings). Most operational models in this field are process-based. Rainfall-runoff modeling has also been modeled as a time series forecasting task to be solved with neural networks. Several approaches based on Long Short-Term Memory (LSTM) networks have already shown that these networks achieve a performance similar to well-established process-based-models. However, most data-driven approaches in the field focus on producing single predictions and do not provide uncertainty bands. In operational hydrological models, uncertainty bands give information about the certainty or uncertainty of the model. This is important for decision making. Therefore, we supplement the existing approaches by quantifying the uncertainty. In addition to plain values, we provide ranges where the discharge values are expected to be found. Our estimates capture both aleatoric and epistemic uncertainty and are based on *bayesian neural networks*; more precisely, the used algorithm is named *Bayes by Backprop (through time)*. Experiments were carried out with the freely available CAMELS-US-dataset, which includes meteorological data and discharge values for 671 catchments across the US. An additional dataset that captures only one catchment, the Regen catchment in Germany, was used. Using the CAMELS-US-dataset we tackled a rainfall-runoff-task, i.e. predicted the runoff, whereas the Regen dataset led to a task where we predicted the streamflow. Results showed that bayesian neural networks are capable of achieving accuracy comparable to state-of-the-art-methods for data-driven approaches in hydrology. Moreover, they can produce uncertainty estimates that capture the capability of the model in a more expressive way than plain predictions. Therefore, bayesian neural networks can be seen as an eligible data-driven alternative for rainfall-runoff-modeling.

Contents

1 Introduction

Hydrology is one of the fields where in the course of increasing computational possibilities, data-driven methods like neural networks have attracted attention in recent years (Gauch et al. 2020). A family of hydrologic models that are also increasingly realized with data-driven methods are the so-called *rainfall-runoff models*. In their essence, these models take meteorological forcings (like precipitation) as inputs and provide predictions for runoff as outputs. Runoff (also referred to as discharge) is a hydrologic concept that refers to the amount of water that flows inside or between basins. This flow can happen on the surface, near the surface or through the groundwater. Typically, runoff is expressed in millimeters during a fixed time period. For an intuitive explanation, one could imagine the outflowing water being equally distributed over the region of interest and taking the hypothetical depth of this evenly distributed water (Gauch et al. 2020). In addition, the term *discharge* can also refer to a related, but different concept: streamflow. Streamflow is the amount of water that flows through the cross-section at a point along a river (Gauch et al. 2020). As water involved in runoff always contributes to a stream, runoff and streamflow can be seen as different ways to describe the same phenomenon; therefore, rainfall-runoff models can also be used to predict streamflow.

Traditionally, rainfall-runoff-models have been mainly process-based. Thus, they focus on representing a simplified version of the underlying natural processes that transform rainfall to runoff. Examples for such processes are evaporation from the surface or soil infiltration (Ludwig et al. 2006, Gauch et al. 2020). These methods require extensive knowledge of underlying physical laws and are commonly based on elaborated mathematical models. Data-driven approaches like neural network are considered a promising alternative to these models as they do not require a preliminary definition of physical laws. Instead, they aim to "learn" the laws by themselves. Several works have already been done in this field (Frederik Kratzert et al. 2018, Frederik Kratzert et al. 2019, Gauch et al. 2021, Kratzert et al. 2021, Fiedler 2020). The data-driven models used in the mentioned experiments are largely based on so-called LSTM networks, a kind of neural network that can work with time series data. Experiments showed that these neural networks are able to achieve predictive performance comparable to state-of-the-art process-based models and even outperform them. However, a drawback of all mentioned approaches is that they only provide a single discharge prediction for each step in a given time series. This limits the

expressiveness of the predictions as rainfall-runoff-modeling is a complex task with multiple sources of uncertainty (Fiedler 2020). It is impossible in practice to achieve perfectly accurate predictions from a model. Therefore, a sounder way to provide discharge estimates would be to give a range where the true value is expected to be found; this range can be given by a lower bound and an upper bound.

Uncertainty estimation in rainfall-runoff models is a field that has already been investigated by Klotz et al. (2020). However, the work mainly focused on proposing a general framework for uncertainty estimation in the field of rainfall-runoff modeling. Examples for approaches that incorporate uncertainty into predictions are also given in this paper, they are mostly based on *Mixture Density Networks*. In their essence, these models output probability distributions instead of single outputs. In the case of a single Gaussian distribution, such a model can be thought as not learning discharge values directly, but learning the mean and deviation of a corresponding Gaussian. These models have a significant disadvantage: while they provide reasonable uncertainty estimates for data similar to the training set, estimates for data outside that range often seem arbitrary and neither feature sufficient predictive accuracy nor a sensible uncertainty band. In more scientific terms, these models only capture *aleatoric uncertainty*; however, *epistemic uncertainty* is only captured in a limited way (Dürr et al. 2020).

One way to also address the latter kind of uncertainty are *bayesian neural networks*. Various approaches for realizing such networks have been presented, including *Bayes By Backprop* and *Monte Carlo Dropout* (Blundell et al. 2015, Gal et al. 2016, Dürr et al. 2020). So far, only Monte Carlo Dropout was applied to the task of discharge prediction, whereas during evaluation only uncertainty estimation, and not accuracy, was examined (Klotz et al. 2020). It is therefore the aim of this thesis to apply an alternative approach, Bayes by Backprop, to the task of discharge prediction. Additionally, it aims to both evaluate the uncertainty estimates produced by the network as well as to compare the accuracy to results obtained from LSTM networks without the Bayes-By-Backprop-algorithm. In summary, Bayes by Backprop could be described as follows: instead of learning network weights, it learns probability distributions over each weight. In the case of Gaussians, each weight would consist of two learnable parameters, a mean and a deviation parameter. The entirety of these probability distributions can then be thought to express the uncertainty of the network. When making predictions, a sample from each distribution is taken; therefore, other than in a deterministic neural network, the same input can result in different network outputs. One way practitioners could take advantage of this observation is to produce a certain number of predictions for a given input and then deduce statements about the network uncertainty from the obtained set of predictions. If they are far apart, the network prediction is rather uncertain, if they are close, the prediction is rather certain. In chapter 5 we will demonstrate that such a network indeed shows sensible behavior. In other words, it approximates an estimate

that expresses both aleatoric and epistemic uncertainty.

The remaining thesis is structured as follows: in chapter 2, 3, 4 and 5, fundamentals regarding deep learning, recurrent neural networks and bayesian deep learning are explained. Then chapter 6 explains the methodolgy in detail and chapter 7 focuses on the results of the conducted experiments.

2 Deep Learning Fundamentals

2.1 Introduction to Neural Networks

Deep learning refers to machine learning methods that make use of multiple layers for the learning process (thus the term "deep" is used). The family of deep learning methods most used in current research and industry is "artificial neural networks" (ANNs). When the context of computer science is clear to the audience, they are often only referred to as neural networks (NN). In general, neural networks refer to computational systems that can be compared, in a coarse analogy, to the human brain. In such an analogy, they consist of a set of neurons that can be connected with each other. Many neural networks consist of several layers such that the input of a neural network can be seen as a signal passing through a set of layers. Those layers apply transformations to the input and eventually construct an output signal. Network connections typically have weights that determine the strength of the signal. In neural networks, individual neurons normally compute the sum of their inputs and apply a non-linear function to this sum (this function is referred to as *activation function*). It was showed that the inclusion of non-linear functions give the network the ability to approximate arbitrarily complex and high dimensional functions. Therefore, neural networks can be seen as *Universal Classifiers*. Figure 2.1 depicts a simple neural network.

In general, we consider $x_i^{(j)}$ to be the i-th neuron in the j-th layer. Consider the neuron $x_1^{(3)}$ in figure 2.1. It has predecessors with values $x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$ and corresponding weights w_1, w_2, w_3. Let F be the activation function. Then, the intermediate value of that neuron is computed as follows:

$$x_1^{(3)} = F(w_1 \cdot x_1^{(2)} + w_2 \cdot x_2^{(2)} + w_3 \cdot x_3^{(2)}) = F(0.88 \cdot 0.3 + 0.27 \cdot 1 + 0.62 \cdot 0.5) \tag{2.1}$$

The aforementioned operations conducted in a neural network can also be denoted with matrix- and vector-operations. This representation is admittedly often less intuitive; however, it is of great importance as it describes how these operations in a neural network are carried out internally by a computer. From a computational perspective, matrix- and vector-operations are oftentimes beneficial as they can be implemented efficiently in terms of time- and space-consumption. Figure 2.2 depicts the above example in matrix-vector-notation.

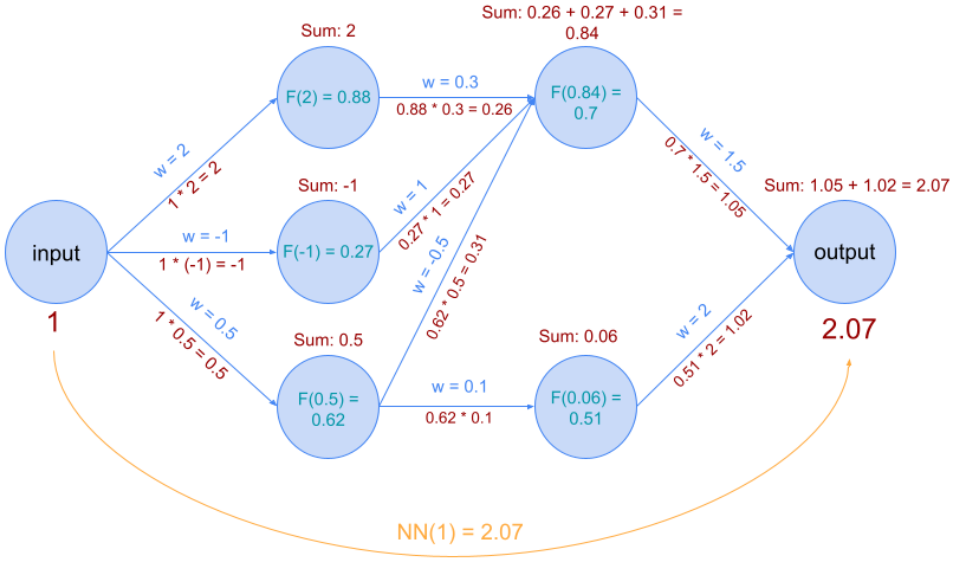

Figure 2.1: Visualization of a simple neural network. It can be observed that the input signal passes through two intermediate layers of neurons (depicted by the filled circles) before the output is formed by the final layer. Each neuron computes the sum of its inputs and applies an activation function (depicted in turquoise)

Equation 2.1 can also be denoted in matrix-vector-notation. We define a matrix W where $W_{i,j}$ represents the weight for the connection from $x_j^{(2)}$ to $x_i^{(3)}$. We define $x^{(i)}$ as the vector consisting of all intermediate values for layer i. We therefore get the following equation:

$$x^{(3)} = F(W \cdot x^{(2)}) \tag{2.2}$$

This is the standard way of defining the matrix-vector-product; however, note that in figure 2.2 we instead compute

$$x^{(3)^T} = F(x^{(2)^T} \cdot W^T) \tag{2.3}$$

which is equivalent from a mathematical point of view and facilitates visualization.

It is important to note that each neuron can optionally have an associated *bias*-term which represents a constant that is added to each sum. Equation 2.1 can be re-written

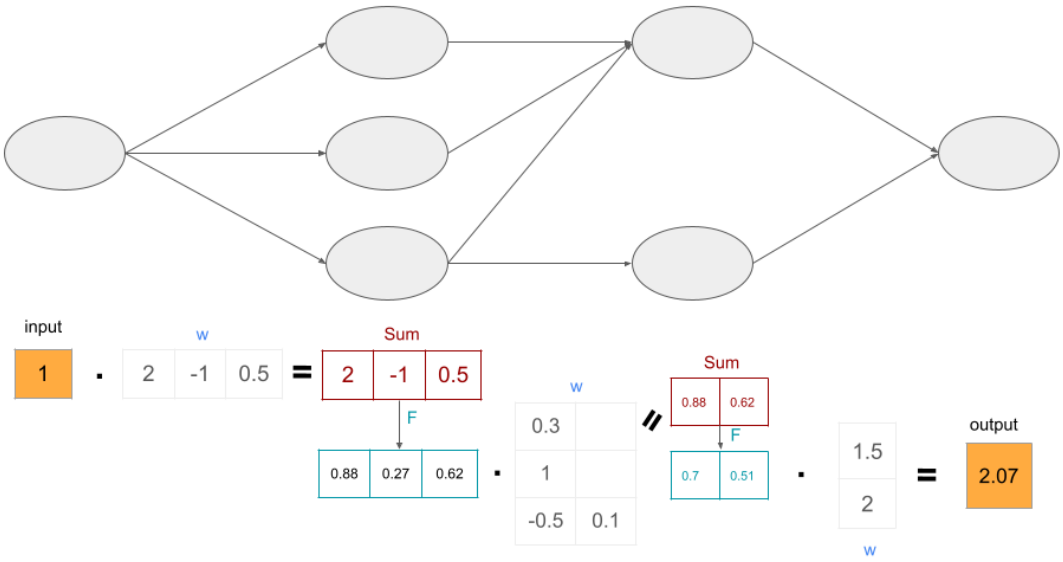

Figure 2.2: Above example denoted with matrix- and vector-operations. Each intermediate value of a layer is depicted as a vector that is computed from a vector-matrix-multiplication of the vector of the previous layer, and a matrix consisting of the weights that belong to the input edges of the current layer. Finally, activation functions have to be applied to each vector.

as follows to also include a bias term:

$$x_1^{(3)} = F(w_1 \cdot x_1^{(2)} + w_2 \cdot x_2^{(2)} + w_3 \cdot x_3^{(2)} + b_1^{(3)}), \tag{2.4}$$

or equivalently in matrix-vector notation

$$x^{(3)} = F(W \cdot x^{(2)} + b^{(3)}) \tag{2.5}$$

where $b^{(i)}$ is a vector representing all bias terms of layer i. Weights and bias terms together form the *parameters* of a neural network that change during the learning process.

Neural networks vary heavily in terms of numbers of layers, kinds of connections and operations. A simple neural network architecture used in practice is the *fully connected neural network*. In such a network, each neuron is connected to each neuron of the succeeding layer. As one can easily see, the network depicted in figure 2.1 is not fully connected because not all neurons in the first intermediate layer are connected

to each neuron in the second layer. In the matrix-vector-notation of the same network, one can determine that it is not fully connected too. This is because there are empty (zero-) entries in the weight matrix that represents the transition from the first to the second intermediate layer.

2.2 Forward propagation, Backpropagation and Optimization

The computations depicted in Figure 2.1 are referred to as *forward propagation*. In this example, the weights were set arbitrarily, therefore, the result of the function computed by the neural network does not have any practical meaning. However, in practice, we typically wish a neural network to be an approximator to a function that has practical use-cases. Consider the following example from computer vision: one can imagine a neural network that is trained to detect whether a colored image of a certain size depicts a cat, or no cat. In this case, a hypothetical function is approximated. This function maps each such image perfectly to its corresponding label (cat or no cat). In order to possibly achieve such an approximation, the forward propagation step consists of an additional aspect, the computation of a so called *loss function*. It can be seen as a measure for the current error the network makes in its predictions. Typically, a loss function value close to zero represents high network performance, whereas a higher loss is considered to be worse. To obtain such a loss measure, neural network typically compare its output to some value verified to be correct. Those values are commonly referred to as *ground truth*. In the example of a cat image recognition network, the ground truth could consist of animal images labeled by a zoologist. Conceptually, only after making a prediction, the neural network is "allowed" to examine the ground truth value. This makes it possible for the network to compare predicted and real value. As neural networks are typically trained with multiple inputs (oftentimes in the order of thousands), the loss function is accumulated over all prediction/ground-truth-label-pairs. In the case of a cat image recognition network, one could compute the overall probability that the networks predicts "cat", given that a cat is present on the respective image, and the probability of predicting "no cat", if no cat is present. The objective of the network would than be to maximize those probabilities. Typically, neural network minimize a certain quantity during learning instead of maximizing. When the logarithm of a probability is taken and the result is inverted, one obtains a domain in which 0 is the optimal value (all labels were predicted correctly) and a higher number correlates with a higher error. This gives the so-called *negative log-likelihood loss*, a common loss functions for neural networks that address classification tasks.

However, in order to achieve learning in the network parameters, two additional steps named *backpropagation* and *optimization* are required. During backpropagation,

the gradient of the loss function with respect to the weights is computed. The gradient in this context has the interpretation of a slope that the network descents in order to find a local minimum of the loss function. Essentially, backpropagation is an algorithm that can be used to compute gradients of complex functions by a recursive application of the chain rule of calculus. Suppose we have $x \in \mathbb{R}$, $f \in \mathbb{R} \to \mathbb{R}$ and $g \in \mathbb{R} \to \mathbb{R}$ and $y = g(x), z = f(y) = f(g(x))$. Then the chain rule of calculus states:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \tag{2.6}$$

Backpropagation can be visualized with a so-called *computational graph*. Such a graph can be used to visualize an arbitrary function. An example for the backpropagation algorithm visualized in a computational graph is given in Figure 2.3

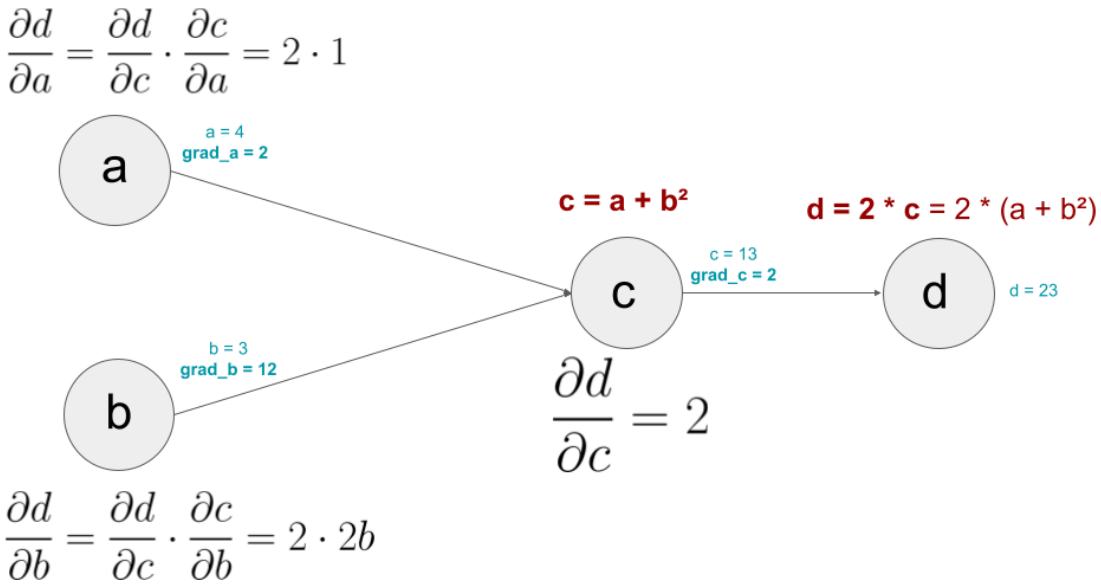

Figure 2.3: Visualization of the backpropagation algorithm. One can observe that, starting from the last node d, the gradients with respect to the other nodes are computed recursively. First, the gradient with respect to c is computed. The value is stored and re-used for the gradients of a and b.

Also a neural network can be disassembled into such a computational graphs which makes it straightforward to take derivatives from a mathematical point of view. To build a connection from neural networks to Figure 2.3, in a neural network the final node d would represent the loss function, the other nodes would represent network parameters.

The derivatives with respect to the parameters are the quantities we are interested in. In theory, all internal computation of a neural network could be carried out with scalar values; however, for efficiency reasons, vector operations are directly included in the backpropagation algorithm. For the case that x and y are vectors, the above chain rule can also be formulated. Suppose $x \in \mathbb{R}, y \in \mathbb{R}, g \in \mathbb{R} \to \mathbb{R}, f \in \mathbb{R} \to \mathbb{R}$. Then we obtain

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \tag{2.7}$$

Again, for efficient computation, it is possible to obtain an equivalent formulation in matrix-vector-notation:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \cdot \nabla_{\mathbf{y}} \tag{2.8}$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the Jacobian matrix of g (the matrix consisting of all partial derivatives) (I. Goodfellow et al. 2016).

A step related to backpropagation is *optimization* which is usually carried out after backpropagation. Training a neural network is an optimization problem which can be solved by taking steps of small sizes into the opposite direction of the gradient with the goal to eventually reach a local minimum. The basic algorithm used to achieve such steps is named *gradient descent*. A peculiarity of neural network is that their loss function is always non-convex; therefore, a local minimum is not guaranteed to also be a global one. Nevertheless, this is not considered a major problem plaguing neural networks for two main reasons: (1) many experts suspect that most local minima have a low loss function value (Saxe et al. 2014, Dauphin et al. 2014, I. J. Goodfellow et al. 2015, Choromanska et al. 2015) and (2) in high dimensional functions, local minima become exponentially less common than saddle points (I. Goodfellow et al. 2016). For the latter, it was empirically shown that the network is able to "escape" such points in most cases during training (I. J. Goodfellow et al. 2015).

It remains to describe the gradient descent algorithm in more detail. Let a represent all network parameters (weights and biases), L is the loss function. To obtain the network parameters after an optimization step, one can write

$$a_{n+1} = a_n - \gamma \nabla L(a_n) \tag{2.9}$$

where $\gamma \in \mathbb{R}_+$ represents a small step into the opposite direction of the gradient and ∇ is the gradient obtained in the backpropagation step. Such a step is oftentimes referred to as an *optimization step*.

So far, we have not considered a concrete algorithm that is carried out by neural networks during learning. Such an algorithm defines how forward and backward propagation steps do alternate. In its mathematically accurate form, the gradient descent algorithm requires the whole set of input data to be propagated though the network before making a single backpropagation step. However, in practice training data is often arranged in so-called *batches*, which are sub-sets of the training data; in this case, the network performs one optimization step after a single batch is propagated through the network once. This reduced the size of the matrices involved in the computation while still providing a sensible approximation to the gradient. The modified version is commonly referred to as *mini-batch gradient descent*. Practitioners use a variety of techniques to improve the presented mini-batch gradient descent algorithm. One such technique is *momentum*. It was designed to compensate for noisy gradients or high curvature and incorporates a moving average of past gradients when performing an optimization step (I. Goodfellow et al. 2016). An adaption of mini-batch gradient descent widely suggested in recent works is *Adam* (Ruder 2016, Kingma et al. 2015). It maintains an individual learning rate per network parameter. This is also done in a related algorithm named *Adaptive Gradient Algorithm (AdaGrad)*. *Adam* is also influenced by an algorithm named *Root Mean Square Propagation (RMSProp)*. It also features per-parameter learning rates that are adapted based on the average first moment (the mean) of recent gradients. *Adam* instead makes use of the average of the second moment of the gradients. Overall, *Adam* provides a more advanced way to combine past information than standard mini-batch gradient descent. For implementation details regarding *Adam* we refer to Kingma (2015).

2.3 Training and evaluation of a neural network

This section covers the procedure of training and evaluating a neural network in a concrete manner. In practical tasks, a limited set of data from which the network is aimed to approximate the desired function is provided. This could for example be a set of animal images for the cat classification network. The data is typically split into a training set and a test set. The training set is used during optimization; however, one also wishes the network to perform well on data that it has not "seen" before (I. Goodfellow et al. 2016). In other words, we aim to prevent the network from learning noisy, irrelevant details in the training set that is not relevant to the distribution we are interested in. This goal is commonly formulated as preventing the network from *overfitting*. In order to achieve this, we use a dedicated *test set* to evaluate the performance of the trained network.

In the course of setting up a neural network architecture, there are several archi-

tectural choices for the practitioner. Beyond the kind of neural network used (ex. recurrent neural network, convolutional neural network) one can also modify internal network settings like number of layers, numbers of neurons per layer and choice of the optimization algorithm. Those settings are not modifiable during the actual optimization, therefore they are referred to as *hyperparameters*. Finding a well performing set of hyperparameters if often named *hyperparameter tuning*. In practice, setting up a neural network often involves examining numerous different sets of hyperparameters. Using only one training set and one test set as described above can lead to a situation in which the chosen hyperparameters only perform well on the examined test set, but would performs poorly on a different set, for example on new data from another source. This phenomenon can be described as yet another form of overfitting. For this reason, a seperate *validation set* is oftentimes split off from the test set. It is used during hyperparameter tuning in order to "hold back" the test set until a final model architecture is obtained (Brownlee 2017, I. Goodfellow et al. 2016). In the next section, a family of neural networks suitable for processing time series, recurrent neural networks, is introduced.

3 Recurrent Neural Networks and LSTM Networks

3.1 Introduction to Recurrent Neural Networks

In section 2 we considered a computer vision task with the goal to classify images. In such a setting, the input for the network is a single image. From a computational perspective, this image consists of multiple values that represent the individual pixels, possibly in different colors. However, from a conceptual point of view, the image can be seen as a single entity. It is not important which pixel the network considers first, and which pixel is processed at a later point in time. In a time series forecasting task however, we consider sequences of measurements taken over time. Therefore, a network structure that is able to not only consider inputs as single "blocks", but to also learn dynamic behaviour over time is beneficial. A neural network architecture that fulfills this criterion is named *recurrent neural network* (RNN). A fundamental property of recurrent neural networks is that the parameters are shared across all time steps of the input sequence. This property is useful in a task like speech recognition. A standard neural network as introduced in the previous chapter would not be able to "transfer" its knowledge of a specific word, if that word would occur at a different place in the input sequence (I. Goodfellow et al. 2016).

Let x be an input sequence for a recurrent neural network, and $x^{(t)}$ the part of that input sequence at time step t with $t \in \{1, ..., \tau\}$. Than the basic functionality of recurrent neural networks can be formalized with the following recurrent equation:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{3.1}$$

where t represents the time step, $h^{(t)}$ represents the values of the hidden units of a network at a specific time step, θ are the network parameters and f is some function. Therefore, RNNs produce a sequence of hidden units vectors $h^{(t)}$ for $t \in \{1, ..., \tau\}$. $h^{(t)}$ is often also referred to as the *hidden state* of the network. So in other words, for each input vector $x^{(t)}$ the hidden state is modified. Usually, RNNs also include different ways for transforming the sequence of hidden states into actual output, thus network predictions (I. Goodfellow et al. 2016). RNNs can be depicted in the same way as standard neural network. However, such a representation requires an individual network per time step

(as in Figure 3.1). This is also referred to as an *unfolded representation* of the network. The individual networks resulting from this representation are usually referred to as *cells*. It is important to note that such a representation does not depict shared weights. In other words, even though an own neural network per time step is depicted, in reality, the weights are still constrained to be the same among all of those network.

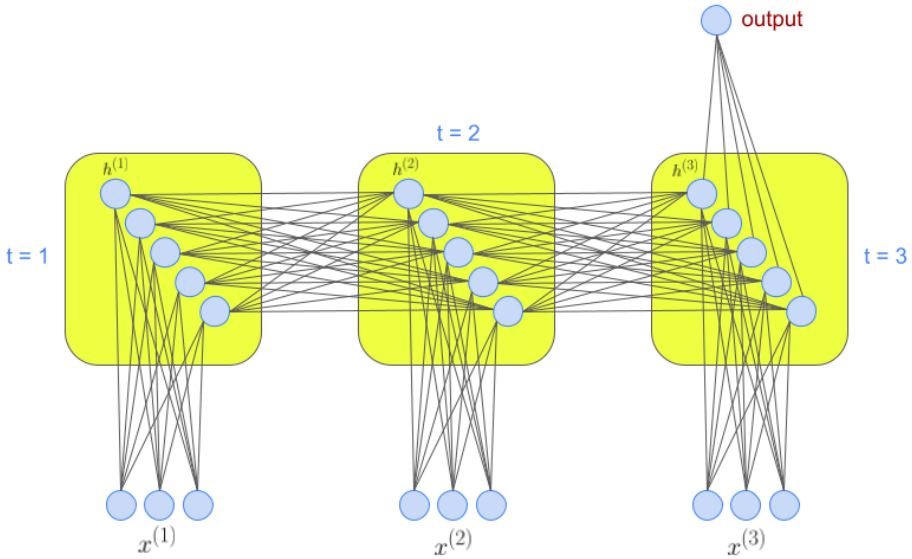

Figure 3.1: Example for a recurrent neural network architecture. The difference to a standard neural network lies in the "unfolding" for multiple time steps. Those time steps all use the same network weights, but unlike in a standard NN, they also depend on the previously obtained hidden layers $h^{(t-1)}$ in addition to the inputs $x^{(t)}$.

The backpropagation-algorithm described in section 2 can also be applied to recurrent neural networks. In this context, it is often referred to as *Backpropagation through time (BPTT)* (Guo 2013). If the unfolded representation of a RNN is considered, the functionality of BPTT does not differ from the backpropagation-algorithm in standard neural networks. Thus, gradients are calculated with respect to each weight; however, in an RNN, the weights are shared across time steps. Therefore, τ gradient updates per weight are calculated, one for each time step. As a consequence, each weight is not modified by a single gradient descent step, but by a sum of multiple descent steps during backpropagation. This leads to the following problem: If large input sequences are used, gradients tend to either become uncommonly large or vanish.

These phenomena may lead to oscillating weights, prohibit learning to bridge long gaps in the input sequence (thus "remember" information from a point further in the past) or completely hinder learning (Hochreiter et al. 1997).

3.2 LSTM (Long Short-Term Memory) Networks

Long Short-Term Memory networks are a special kind of RNNs and were designed to cope with long-term dependencies in input sequences. One may recall that the introductory example for an RNN depicted in figure 3.2 only consists of a single layer per cell. LSTM networks in contrast consist of multiple layers that are connected in a special way depicted in figure 3.2. Next to the hidden state, each network also outputs a so-called *cell state* (Hochreiter et al. 1997, Olah 2015). Therefore, a cell computes two recurrent equations per time step:

$$\begin{aligned} h^{(t)} &= f(h^{(t-1)}, c^{(t-1)}, x^{(t)}; \theta) \\ c^{(t)} &= f(h^{(t-1)}, c^{(t-1)}, x^{(t)}; \theta) \end{aligned} \tag{3.2}$$

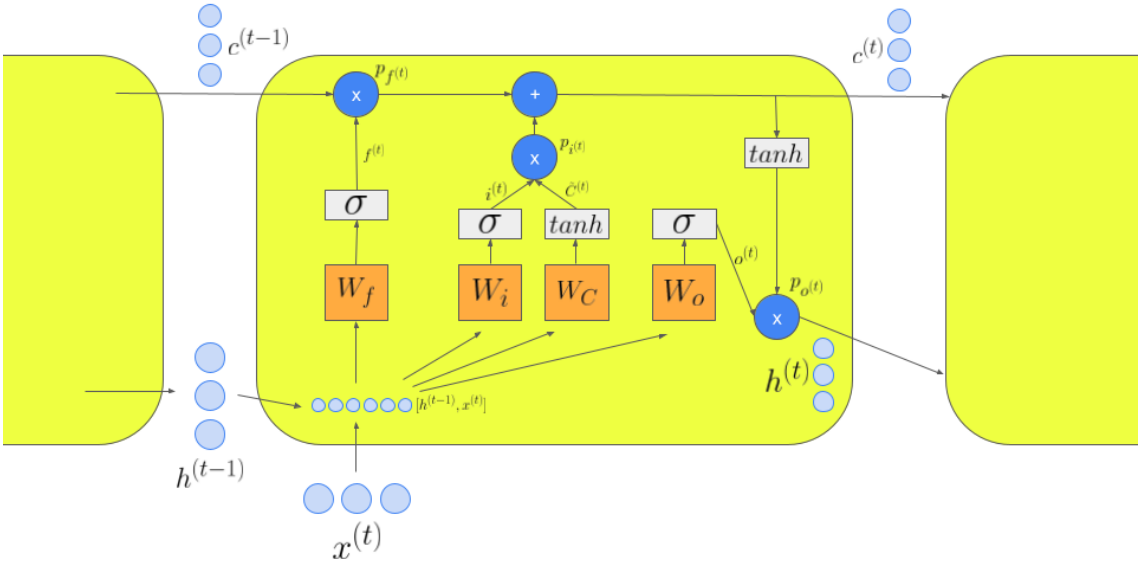

Figure 3.2: Cell of an LSTM network. Weight matrices are depicted in orange, functions in grey and pointwise operations in dark blue

The following sections describes the different layers within a cell in detail. Each layer

can be compared to a standard neural network layer as introduced in chapter 2 as it features a weight matrix and an activation function.

3.2.1 Forget gate layer

The forget layer gate (depicted by W_f in Figure 3.2) conceptually "filters" the previous cell state $c^{(t-1)}$. The output of the layer, denoted by $f^{(t)}$, consists of numbers in the interval $[0, 1]$. Together with the point-wise multiplication $p_{f^{(t)}}$ they can be interpreted as "importance weights" that are used to filter out or keep certain parts of the cell state. The forget gate layer features an activation function named *sigmoid* (denoted with σ). The function computed by the layers is therefore:

$$f^{(t)} = \sigma(W_f \cdot [h^{(t-1)}, x^{(t)}] + b_f) \tag{3.3}$$

It is important to note that, next to the hidden state, the layer also depends on the input of the specific time step. This design makes intuitive sense as oftentimes in time series new values indicate that older values might not be useful anymore. For instance, consider a network that receives text in natural language and outputs the sentiment expressed by the text. In such a language-related task, it is important for the network to keep track of sentence structures and grammar to learn the correct meaning. However, if the network for example encounters the beginning of a new sentence, information about the subject's number from the previous sentence may not be relevant anymore and can be "filtered out".

3.2.2 Input gate layer(s)

The input gate layer(s), consisting of the two layers depicted by W_i and W_C, conceptually add new information to the cell state. W_C features a *tanh*-activation function and therefore maps to the interval $[-1, 1]$. Intuitively, it can be interpreted as the network's "proposals" for the values that are added to the cell state vector. The output of this layer is denoted by $\tilde{C}^{(t)}$. W_i in contrast features a *sigmoid*-activation function and is used to "filter out" certain values of $\tilde{C}^{(t)}$ with the point-wise multiplication denoted by $p_{i^{(t)}}$. The functions computed by the layers is:

$$\begin{aligned} i^{(t)} &= \sigma(W_i \cdot [h^{(t-1)}, x^{(t)}] + b_i) \\ \tilde{C}^{(t)} &= \sigma(W_C \cdot [h^{(t-1)}, x^{(t)}] + b_C) \end{aligned} \tag{3.4}$$

In the previous example of language interpretation, the input gate layer could add the number of a new subject it encounters.

3.2.3 Output gate layer

The output gate layer, depicted by W_o, is yet another network layer with a *sigmoid*-activation function and is used to transform the cell state $c^{(t)}$ into a hidden state $h^{(t)}$. This layer has no intuitive meaning; however, it is important to point out that by design, $c^{(t)}$ should be involved in as few operations as possible, therefore this additional step is not applied to the cell state (Olah 2015). The function computed by the layer is:

$$o^{(t)} = \sigma(W_o \cdot [h^{(t-1)}, x^{(t)}] + b^o) \tag{3.5}$$

$h^{(t)}$ and $c^{(t)}$ are then computed as follows:

$$\begin{aligned} c^{(t)} &= (c^{(t-1)} \times f^{(t)}) + (i^{(t)} \times \tilde{C}^{(t)}) \\ h^{(t)} &= tanh((c^{(t-1)} \times f^{(t)}) + (i^{(t)} \times \tilde{C}^{(t)})) \times o^{(t)} \end{aligned} \tag{3.6}$$

3.3 Structure of predictions obtained by Recurrent Neural Networks

To this point, we did not mention how the hidden states and cell states are converted to concrete network predictions. There are multiple possibilities to obtain predictions from RNNs. Hidden states can be used directly as output vectors; however, the desired dimension of the output vector is a different concept than the dimension of the hidden state vector, and the two sizes do not have to correspond. To tackle this heterogeneity, a common practice when forming output vectors is introducing a fully connected neural network layer that takes a hidden state vector an an input and outputs the final vector. Figure 3.1 depicts such a case, whereat the output consists of a single scalar value. However, the output is not restricted to scalar values, but can be of arbitrary size. In the case depicted on the figure, only the hidden state at the last cell $h^{(3)}$ serves as input for the fully connected layer. This architecture is from now on referred to as *many-to-one* (many-to-one-architecture), as the input sequence consists of multiple values, but the output is based on a single network cell. A practical example for a use-case of such a network is again a text sentiment analysis task. In this case, the input for the RNN is a sequence of words and the output is a scalar value representing the predicted sentiment, for example 1 for positive and 0 for negative. However, RNNs are not restricted to output single vectors. Instead, one could also imagine transforming not only $h^{(3)}$, but every hidden state $h^{(t)}$ to an output vector. Such an architecture is from now on referred to as *many-to-many* (many-to-many-architecture). It could for example be used for a translation task, where we do not aim for a single prediction, but for a sequence of translated words as an output.

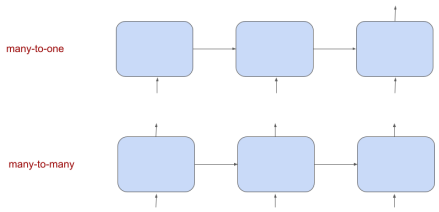

Figure 3.3: Comparison between many-to-one- and many-to-many-output in an RNN. Note that we abstract from the concrete inputs, hidden states, cell structure and outputs in this representation

Another possibility to treat hidden state vectors is to use them as inputs for another recurrent neural network. Figure 3.4 depicts such a setting with two recurrent neural networks "stacked" onto each other. In this case, the first network is not involved in forming the output, but rather supplies input vectors for a second network. The second network transforms the hidden states to output vectors.

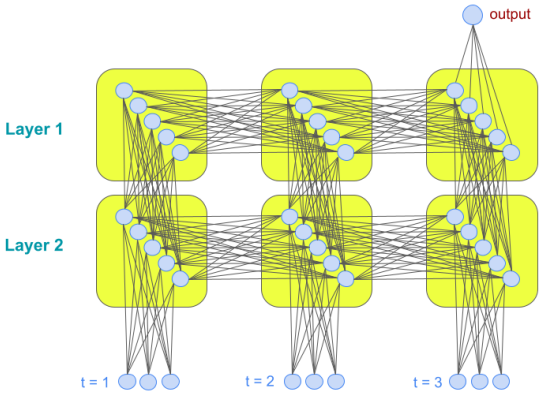

Figure 3.4: Example for a recurrent neural network architecture with two layers. It can be observed that the hidden states of the first layer are not used to form the output as in previous architectures, but serve as input vectors for the second layer.

4 LSTM networks for hydrology

4.1 Introduction to streamflow and discharge prediction

In order to fully understand the task tackled in this thesis, it is necessary to define some terms and phenomena related to hydrology and discharge prediction. As already mentioned in the introductory chapter, discharge prediction aims to predict the amount of discharge/runoff of a given region. For this task, past discharge measurements as well as forcings (meteorological variables like air temperature or amount of precipitation) can be considered as input variables. Another kind of input variables commonly used in discharge prediction is geophysical information about the region in which the measurements are taken. The output of the prediction model should be a sequence of discharge values (or a single value). The temporal granularity of the needed sequence often depends on the application area. During flood events for example, a higher temporal granularity may be needed (Gauch et al. 2020). As one may recall, discharge prediction is highly related to another task: streamflow prediction. Oftentimes, the same models are used for both tasks. Streamflow is typically measured at geographically prominent points like the mouth of a river or the outflow of a lake. The point where the measurement is taken is referred to as the *outlet*. From the outlet, an area can be delineated from which all water drains towards the outlet. This area is referred to as *catchment*, *basin*, *watershed* or *upstream area* (whereat *catchment* is the main term used throughout this thesis). One may realize that the size of a catchment highly depends on the choice of the outlet (Gauch et al. 2020). A catchment is referred to as *gauged basin* if its outlet has a gauging station to measure streamflow values, and as an *ungauged basin* if such a station is missing. An important term in the context of streamflow is *hydrograph*. It typically refers to a graph that shows the streamflow versus time; in addition, it can also refer to a graph that shows runoff versus time (Gauch et al. 2020).

When it comes to different types of models for discharge predictions, we can distinguish between the terms *forecasting model* and *simulation model*. Forecasting means predicting discharge values in the future. Simulation on the other hand refers to predicting past discharge. At first glance, it may not be entirely clear why the latter is useful; however, forecasting has a number of drawbacks. The most obvious objection is that forcings do not exist for future time steps and are therefore bound to be predictions themselves, for example values taken from a hydrologic forecasting model. Of course,

grouth truth values of discharge also do not exist for future time steps. Simulation on the other hand solves these disadvantages and has several application areas. In cases where measurements from the pasts are not available or spare, simulation can be useful to obtain datasets of high resolution (Gauch et al. 2020). Also the conducted experiments in this work can be described as simulation. In this context, simulation models are used to evaluate neural networks effectively and to compare the performance to existing results in the field.

Models can also be classified based on the spatial distribution of the input data. The family of models used with the CAMELS-US-dataset is referred to as *lumped models*. In those models, forcings and geophysical data are aggregated to a catchment-level. Such aggregated forcings could for example be the amount of total precipitation within a catchment or the minimum and maximum temperature (Gauch et al. 2020).

4.2 Connection to time series forecasting

In the course of the following sections, it will become clear that discharge prediction is a form of time series forecasting for which recurrent neural networks are suitable. In a time series forecasting task, the input of an RNN consists of ordered measurements taken over time up to a certain point. The output consists of predictions for prospective points in time. In the context of discharge prediction the input consists of forcings taken over time and the output consists of discharge predictions for the future (Gauch et al. 2020). In the task addressed in this thesis, multiple forcings are considered at each single point in time. Such a setting is named *multivariate forecasting model* (Brownlee 2018). In general, multivariate forecasting models are able to predict multiple variables per time step; however, in discharge prediction, only a single variable (the discharge) is obtained from the network output (Fiedler 2020).

4.3 Model used in this thesis

In the meteorological domain, datasets that features measurements over a certain timespan are common. It remains to determine how such a dataset can be transformed to suitable training and testing data for a neural network. The approach used in this thesis is oftentimes referred to as *sliding window*. Such an approach divides the available data into sequences of fixed length (referred to as *windows*). The length of those sequences is commonly referred as *window size* (Fiedler 2020). Hereafter, the window size is referred to as s_w. Each window forms an input sequence that is provided to the RNN. Conceptually, one could start by viewing the task as a many-to-many-based one where the network forms an output from each vector of the input

sequence. However, networks for discharge predictions are typically not aimed to make predictions from the start as poor performance is to be expected at an early point of the input sequence. Instead, one can think of a *warmup window* (with length s_{wu}, thus a part at the beginning of the windows where the network does not make predictions yet). The part of the window that is involved in prediction-making is referred to as *prediction window* (with length s_p). Figure 4.1 depicts the approach in a graphical way.

A remaining question is how an algorithm that produces windows from a dataset proceeds in practice. Assume the complete dataset consists of n time steps. Let $t \in \{0, ..., n\}$ be a specific time step of the dataset. In the experiments conducted in this thesis, the windows must be organized in a way so that for each time step $t \in \{s_w, ..., n\}$ exactly one discharge prediction is obtained (for the first s_w time steps, no predictions are possible as the warmup window would be too short). Producing exactly one discharge prediction per time step can be ensure by setting an appropriate *sliding window step* (k), which refers to the offset of each newly created window with respect to the previous one. We set the offset to equal the length of the prediction window (thus $k = s_p$). This is best explained with an example: assume that the windows are iteratively constructred by starting at the beginning of the dataset, further assume $w_p = 2$ (the network predicts 2 days into the future). Then, for each new window, an offset of 2 has to be chosen with respect to the previous window. If we would instead choose an offset of 1, there would be a time step for which 2 discharge predictions are obtained, if we choose an offset of 3, there would be a time step with no associated discharge prediction.

In the conducted experiments for the CAMELS-US-dataset, each time step has 5 associated forcings, namely precipitation, shortwave downward radiation, maximum and minimum temperature and humidity. As a consequence, each entry $x^{(i)}$ of an input sequence x is a vector of size 5 (at a later point, additional input measurements are considered too). For the output, three variations are examined in the course of the thesis: many-to-one, single shot and many-to-many. It is important to point out that those models describe a different concept than the architectures described in section 3. Those only refer to architectures of the network, whereas here the interpretation of the data that flows too the network also plays a role.

4.3.1 Many-to-one

This approach follows the methodology described in section 3. In the context of a discharge prediction task in combination with the sliding-window-approach, the network makes a single discharge prediction for the last day in the window ($x^{(s_w)}$). In such a case, the prediction window has length 1, the warmup window has length $s_w - 1$. The adaption of this approach for the given task is depicted in figure 9.1 of

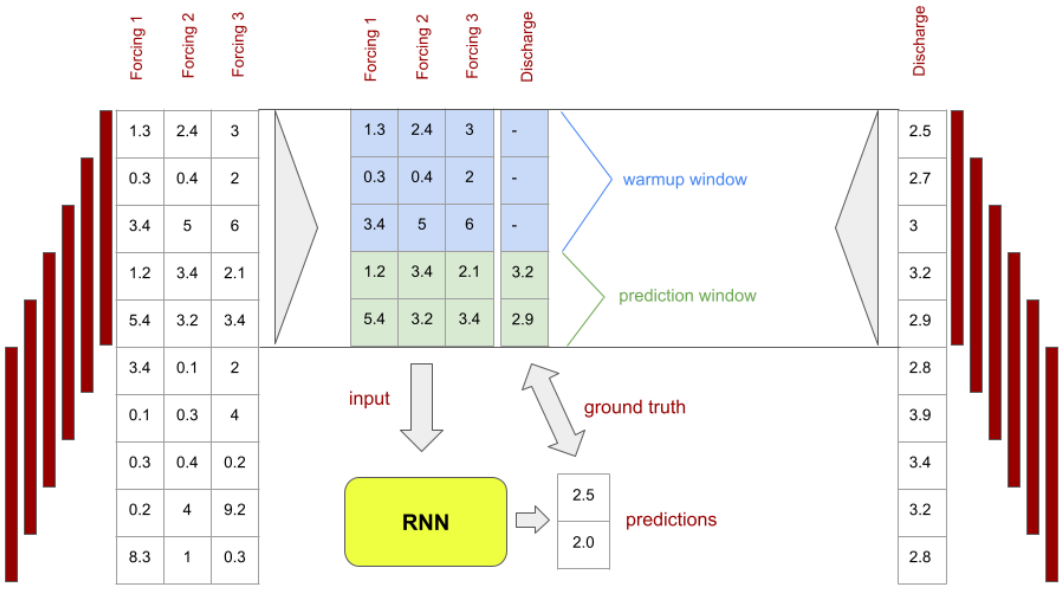

Figure 4.1: Sliding window approach for discharge prediction. The available dataset includes 10 time steps, each time step has 3 forcings and a discharge value available (which are random values in this exemplary case). The window size is set to 5 in this example, therefore, an input sequence for the RNN consists of 5 time steps. The RNN is designed to only make predictions for the last 2 days of the window, therefore, ground truth is considered for the last 2 days of the window.

Appendix A.

4.3.2 Single shot

The so-called "single shot"-approach also follows a many-to-one-architecture. The difference lays in the structure of the output the network predicts and in the way this output is interpreted. Although the output is only produced by the last cell, it does not consist of a single scalar value, but of a vector of multiple values. This vector features discharge predictions for multiple future days. The only necessary modifications to the many-to-one-approach are that (1) the output size (and corresponding ground truth vector for comparison) have to be adjusted and (2) the sliding window step o has to match s_p. The adaption of this approach for the given task is depicted in figure 9.2 of Appendix A. Conceptually, this approach does not "see" additional forcings

while processing the prediction window. Therefore, it is expected that the performance deteriorates with bigger prediction windows.

4.3.3 Many-to-many

The many-to-many-model produces the same results as the single-shot-model from a black-box view. The difference lies in the circumstance that the many-to-many-model follows a typical many-to-many-architecture for RNNs as described in section 3. Therefore, in case of $s_p > 1$, the predictions origin from different cells. Figure 9.3 of Appendix A depicts an adaption of this approach for the given task. With this approach, the model continues "seeing" forcings while processing the prediction window. One would expect that this design leads to better performance with longer prediction windows compared to the single-shot-model.

4.3.4 Models that use discharge as input measure

Next to forcings, hydrological models are also allowed to consider past discharge data as input (Gauch et al. 2020). This option is also examined in the course of this thesis. However, it requires some modifications to the previous architectures as it is prohibitive to consider discharge values from the prediction window. There exist several approaches to tackle this problem, one of them is inserting "nonphysical" values for the affected discharge values. Obviously, those values do not provide any value for learning and the network is aimed to detect and "ignore" them (Fiedler 2020). However, this results in an additional learning task for the network. Therefore, in this work an *encoder-decoder-approach* was followed. Such an approach features two different RNNs: an encoder that processes the input values of the warmup window, but does not produce any output, and a decoder that processes the input values from the prediction window and also forms corresponding predictions. As the hidden state and cell state can be transferred from the encoder to the decoder, the architecture can conceptually be seen as a single network. Nevertheless, the advantage of the said architecture is that encoder and decoder are allowed to expect input vectors of different length. In our case, the encoder expects the forcings and the additional discharge values, whereas the decoder only expects the forcings. An encoder-decoder-architecture is graphically depicted in figure 4.2. The resulting architecture is suitable for each of the previously discussed models. In the case of many-to-one and single shot, the decoder only receives input from a single time step and only produces a single prediction. In the case of many-to-many, the decoder is further unfolded during forward propagation and produces multiple predictions.

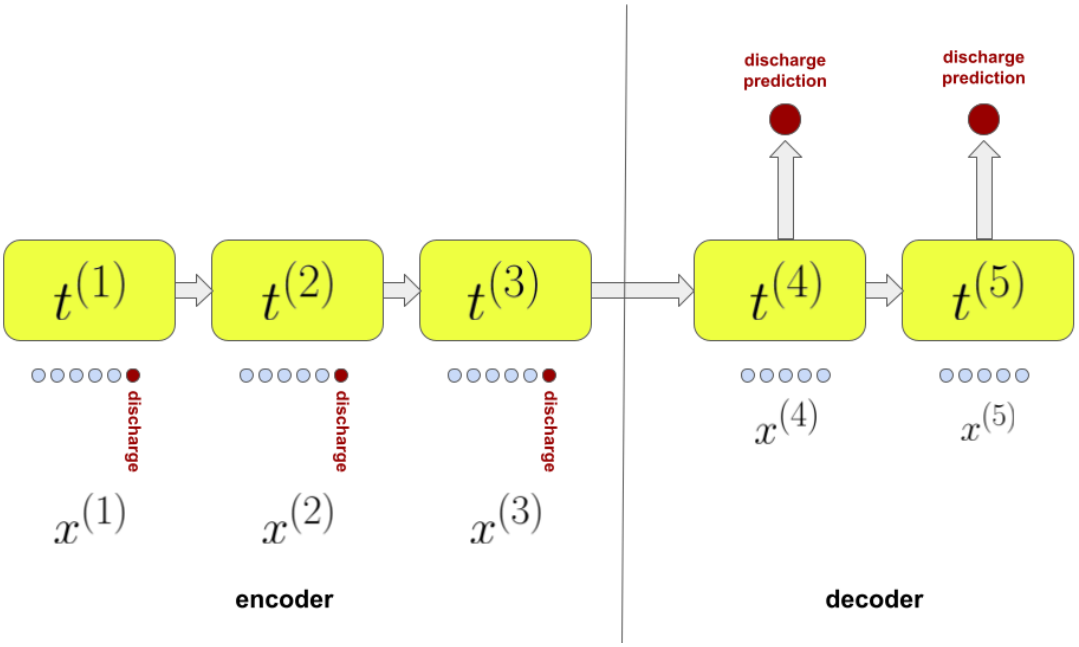

Figure 4.2: Encoder-decoder-model. It can be seen that only the encoder receives discharge values as input, whereas only the decoder produces output predictions

5 Bayesian Deep Learning

This chapter covers the concept of bayesian statistics and bayesian deep learning needed to comprehend this thesis and finally introduces *Bayes by Backprop through time*, the underlying algorithm for the modified LSTM network used in the conducted experiments. In the subsequent sections, fundamentals of modeling and statistics are explained.

5.1 Probabilistic modeling

In mathematics and computer science, a model is a simplified image of the reality to be observed. Typically, one aims for a formal description of a phenomenon to be observed. For example, population growth in a certain region could be modeled with a differential equation, whereat simplified assumptions have to be made. Population grow is a complex phenomenon and it will be impossible in practice to incorporate all parameters into a mathematical model. Next to differential equations, there are numerous other tools that can be used to model a phenomenon. In a scheduling problem, a directed graph could be used to model dependencies among jobs (Bungartz et al. 2014). Neural networks also belong to those tools as they represent functions describing certain phenomena.

A sub-area of mathematical modeling of particular relevance for this thesis is *probabilistic modeling*. In probabilistic modeling, probability distributions are incorporated into the model of a phenomenon (Glen n.d.). Such probabilistic approaches can also be incorporated when designing neural networks. Such architectures are also referred to as *probabilistic deep learning* (Dürr et al. 2020). A practical example of a probabilistic neural network could be the image classification network introduces in section 2 that decides whether a cat is depicted on an input image or not. In a non-probabilistic approach, the network would output a single binary value (for example 1 for *cat* and 0 for *no cat*). However, this approach would not yield any information about the network's confidence about the prediction. Therefore, a to some extend more expressive approach is to output values in the interval $[0, 1]$ for each of the two categories. Those values are constraint to sum up to 1, they therefore represent the probabilities that the input image belongs to a category. Another example is found in the hydrologic domain. Imagine a discharge prediction task that is modelled with a neural network that outputs a single

prediction for the discharge value (as it was assumed to be the case in all previous chapters). In a probabilistic modeling approach, the neural network would be designed in a way to not output single values, but some probability distribution that specifies in which range the network assumes the value to be. Mixture Density Models mentioned in the introduction fall into these category of networks (Klotz et al. 2020). Probabilistic models are heavily used in classification tasks, for example in image classification. However, although being able to express uncertainty, named probabilistic models still have a disadvantage: they can only capture the so-called *aleatoric uncertainty*. In a neural network context, aleatoric uncertainty refers to the uncertainty inherent to the training data (Dürr et al. 2020). In other words, it expresses the ability of the network to assign high uncertainty to regions in the input space where there are fluctuations in the training data. In the image classification context, this means that there may be certain training images that are hard to identify as a cat, for example because they depict a similar-looking animal. If this transition is sufficiently covered by the training data, a network that is able to learn aleatoric uncertainty will assign equal probability to *cat* and *no cat* if it encounters such an image in the test set. In a regression example, such a probabilistic model would be able to detect a function and variances such that most of the training data lies in the predicted interval. This is the case in figure 5.1.

Figure 5.1: Regression example for probabilistic model. Assume that the model learns means and deviations to account for the variance in the training set (training samples are depicted by the blue points). The predicted means are depicted by the red curve, the predicted deviation is represented by the blue range. It can be observed that in regions with lots of training data the model achieves a good approximation of the function underlying the training data.

However, one can also deduce form figure 5.1 that the described probabilistic models

have a disadvantage. They do not express sensible uncertainty estimates about regions in the input space that were not present in the training data. This means that the model "acts" very confident when it receives inputs far from the training distribution, although the network cannot reasonably substantiate such a confident decision as it was not trained for such inputs. In figure 5.1, one can observe that outside the range where training data is present the deviation is very low, although it is not an expected behavior of a model to act confidently in that region. In the context of cat classification, one could imagine a training set consisting only of cats and dogs. A network trained on those images might correctly classify all cats correctly as *cat* and all dogs correctly as *no cat*, and might also sensibly assign equal probabilities to the categories if the image depicts some kind of fusion between a cat and a dog. However, given an input image depicting a horse, the network may act in an unexpected way; a scenario one would have to reckon with is that the network assigns high probability to the category *cat*.

This behavior obviously does not lead to an intuitive understanding of uncertainty. A sounder way of expressing uncertainty can be achieved with models that do not only capture aleatoric uncertainty, but also the so-called *epistemic uncertainty*. Epistemic uncertainty refers to the uncertainty in the parameters of the model (Dürr et al. 2020). Regarding figure 5.1, one could imagine changing the parameters of the model such that the predicted function is the one in figure 5.2. As both predictions describe the training data with the same accuracy, an "intelligent" model would have to be uncertain between the two variants. However, this epistemic uncertainty is not captured in probabilistic models described so far. A way to tackle this problems are so called *bayesian probabilistic models*.

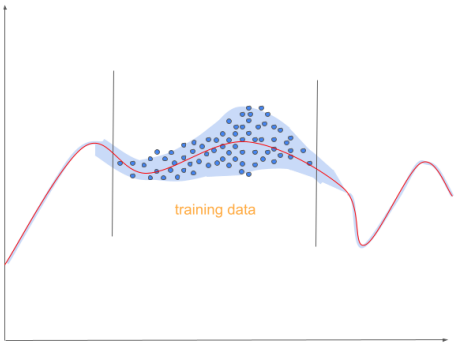

Figure 5.2: Alternative fit to the training set. One can observe that the function in red fits the training set equally good as the one in figure 5.1 although it has an entirely different shape.

5.2 Frequentist and Bayesian statistics

The following section aims to introduce the field of Bayesian statistics, especially Bayes' theorem. In statistics, there are two possibilities to interpret probabilities. The first way is as a relative frequency, where n is the number of conducted experiments. This is referred to as a *frequentist approach*. Here, $P(x)$ is approximated as the number of experiments n_x where the desired outcome x occurred. This definition is valid as the following limit converges to the actual probability:

$$P(x) = \lim_{n \to \infty} \frac{n_x}{n} \tag{5.1}$$

An example for a probability determined by a frequentist approach could be tossing a coin (which does not necessarily have to be fair). Let x be the event where the coin shows head. Suppose $n = 100, n_x = 32$ (thus the toin is tossed 100 times and showed head 32 times). Therefore, the probability $P(x)$ is approximated with $\frac{32}{100}$.

A drawback of the frequentist approach is that is is not possible to make statements about probabilities of events that do not allow an arbitrary number of repetitions. For instance, it is not possible to assign a probability to the statement "The first moon landing was no fake", as the event obviously cannot be repeated several times (Kirchgessner 2009). Another way to express probabilities is by the so called *bayesian approach*. It assigns probability according to a *degree of belief*. The underlying theorem of the bayesian approach is Bayes theorem':

$$P(x|D) = \frac{P(D|x) \cdot P(x)}{P(D)} \tag{5.2}$$

$P(x)$ is named *prior distribution*. It denotes the prior belief about the probability of event x before any data is seen. $P(D|x)$ is called *likelihood*. It assumes x and denotes the probability that the data (or event) D is observed. $P(D)$ is the probability of D and is often called *evidence*. $P(x|D)$ is then referred to as the *posterior distribution* and stands for the estimate of the probability of x after data is seen. It therefore can be seen as an "updated" version of the initial degree of belief. The concept is explained with the example of a test to detect a human disease. Imagine one is interested in the probability of having a certain disease, given that a test for that disease was positive. Let x be the event in which said person has the disease, D corresponds to the observed data, thus the positive test. Considering Bayes' theorem, $P(x)$ is the prior distribution and stands for the general probability for catching the disease. The likelihood $P(D|x)$ is the sensitivity of the test, thus the probability that the given test detects the disease. The posterior distribution $P(x|D)$ is the quantity one is interested in, the probability

that one has the disease, whereat additional data (the test was positive) was considered. Inserting the formula for the marginal probability instead of the evidence $P(D)$ yields:

$$P(x|D) = \frac{P(D|x) \cdot P(x)}{P(D|x) \cdot P(x) + P(D|\neg x) \cdot P(\neg x)} \tag{5.3}$$

Thus, the value on the left can be computed using Bayes' theorem, given that a prior and sensitivity/specificity of the test are known (Kirchgessner 2009).

5.3 Bayesian probabilistic models

Next to aleatoric uncertainty, a bayesian probabilistic model is also able to capture epistemic uncertainty. In the following section, a technique to apply those model to neural networks is presented. Such neural networks are commpnly referred to as *bayesian neural networks (BNN)* (Dürr et al. 2020, Jospin et al. 2020). Hereafter, the functionality of bayesian models is introduced with the example of image classification. In a standard probabilistic model for image classification, the parameters of the network are found via a *maximum likelihood* approach. Therefore, we maximize

$$w_{maxLike} = \arg\max_{w_i} P(D|w_i) = \arg\max_{w_i}(P(y_1|x_1, w_i) \cdot ... \cdot P(y_n|x_n, w_i)) \tag{5.4}$$

for n training images. In the example of cat classification, it holds that $\forall x\ x \in \Omega$ where Ω is the vector space containing all possible images, $\forall y\ y \in \{cat, noCat\}$ and $\forall i\ w_i \in \mathcal{W}$ where $\mathcal{W}$ is the vector space of all possible network parameters. Equation 5.4 is the underlying formula for a neural network trained with the *negative log-likelihood* loss function. We obtain a network prediction $\tilde{y}$ by evaluating

$$\tilde{y} = P(y|x, w_{maxLike}) \tag{5.5}$$

One can easily see the reason why such a network does not capture epistemic uncertainty. Training only focuses on finding *one* set of appropriate weights that fits the training data well. However, the approach ignores the circumstance that there may be other sets of parameters that describe the training data almost as well. A fundamental concept of bayesian probabilistic models is that they consider multiple predictions that are based on different sets of weights. A prediction would than be made by evaluating

$$\tilde{y} = \int_w P(y|x, w) \cdot P(w|D)\ dw \tag{5.6}$$

Conceptually, one can imagine iterating over all possible sets of weights. $P(y|x, w)$ would correspond to the output distribution that the network predicts for the training

example x given parameters w. In the cat classification this is the predicted probability of *cat* (or of *no cat*) given input image x and network parameters w.

The term $P(w|D)$ is drawn from a *posterior distribution* (Dürr et al. 2020). One could think of this term as a "weight" for each iteration as it denotes how likely a certain set of parameters is given the training data. $P(w|D)$ can be re-written with Bayes' theorem:

$$P(w|D) = \frac{P(D|w) \cdot P(w)}{P(D)} \tag{5.7}$$

Intuitively, a model as described in 5.6 considers every possible set of weights in its predictions. Suppose that $\tilde{y}$ corresponds to the probability the network assigns to an image of a horse after it has only seen cats and dogs during training. As the network has not learnt anything about horse images, it may be uncertain about which parameters to choose. Assume that there are multiple sets of parameters that all describe the training data in an accurate way, but give very different results for horse images. The maximum-likelihood-approach in equation 5.5 would only choose one of those parameter sets and ignore the others. In 5.6 however, all parameter sets are considered. Therefore, the model would successfully learn that it is uncertain about horse images. We say that such a model is also able to learn the uncertainty inherent to the model parameters (the epistemic uncertainty). In the case of a simple linear regression problem (with two parameters), this set-up has an analytical solution (Dürr et al. 2020). However, in the case of a neural network with potentially millions of parameters, the integral $\int_w P(y|x,w) \cdot P(w|D)\ dw$ is highly intractable. Even if a discrete set of values is considered for each parameter, the integral (which changes to a sum in this case) requires iterating over a set of values that grows exponentially with the number of parameters. The posterior $P(w|D)$ is intractable too. It can be re-written as:

$$P(w|D) = \frac{P(D|w) \cdot P(w)}{\int_w P(D|w) \cdot P(w)\ dw} \tag{5.8}$$

The denominator is intractable for the reason already described (Dürr et al. 2020). Therefore, it can be concluded that solving bayesian neural networks is not feasible with the means of classical bayesian statistics. In the following section, a technique named *Bayes by Backprop* is introduced. It allows constructing a bayesian neural network that is able to capture aleatoric and epistemic uncertainty.

5.4 A technique to realize bayesian neural networks – Bayes By Backprop

To obtain a bayesian probablistic model in a neural network context, different sets of network weights have to be considered when making a single prediction. As it would be highly intractable to take all possible sets of weights into account, Bayes by Backprop applies a number of approximation techniques to obtain a bayesian model (Blundell et al. 2015). In the following, a neural network for a regression task realized with a probabilistic modeling approach is considered. Therefore, the network output is $P(y|x,w)$ which is a assumed to be Gaussian. With a maximum likelihood estimation, the weights can be determined with:

$$w_{maxLike} = \arg\max_{w} \log P(D|w) = \arg\max_{w} \sum_{i=1}^{n} P(y_i|x_i,w) \tag{5.9}$$

with n training samples. However, with a Bayesian approach, the weights are instead determined with equation 5.6. Conceptually, this can be considered a large ensemble of networks, whose predictions are weighted according to the degree of belief in the respective parameter configuration (Blundell et al. 2015). Exactly computing the posterior is computationally prohibitive because of the denominator of Bayes' theorem in equation 5.7. Therefore, the posterior is approximated with another distribution denoted with $q(w|\theta)$ (referred to as the *variational posterior*). Its parameters θ are found in the course of a process called variational learning (Graves 2011, Hinton et al. 1993). This leads to an optimization problem whose solution is found by minimizing the KL-divergence between the approximate and the true posterior:

$$\theta^* = \arg\min_{\theta} KL[q(w|\theta)||P(w|D)] \tag{5.10}$$

Re-writing the equation yields:

$$\begin{aligned} \theta^* = \arg\min_{\theta} \int q(w|\theta) \frac{q(w|\theta)}{P(w) \cdot P(D|w)} dw = \\ \arg\min_{\theta} KL[q(w|\theta)||P(w)] - \mathbb{E}_{q(w|\theta)}[\log P(D|w)] \end{aligned} \tag{5.11}$$

This can also be expressed by a loss function, which is the loss function of the bayesian neural network:

$$\mathcal{F}(D,\theta) = KL[q(w|\theta)||P(w)] - \mathbb{E}_{q(w|\theta)}[\log P(D|w)] \tag{5.12}$$

The loss function is commonly referred to as *ELBO-loss* (Kingma et al. 2014). The left side of the equation is named *complexity cost*. It is a feasible KL-divergence between

the variational posterior $q(w|\theta)$ and a prior $P(w)$ which can be calculated analytically for some choices of priors and posteriors. In the Bayes by Backprop approach, the KL-divergence is not computed analytically, but approximated as we will see in the next section. The right side is named *likelihood cost* as it contains $P(D|w)$ which is also used in standard maximum likelihood estimation. Additionally, it contains an expectation over the variational posterior. Also in this case, it is not possible to consider all possible combinations of weights; as a consequence, next to the approximation of the posterior a second approximation has to be done: the expectation is approximated by drawing a fixed number of samples from $q(w|\theta)$.

Blundell et al. (2015) showed that $\mathcal{F}(D, \theta)$ can be approximated with a *Monte Carlo sampling technique*, which refers to a technique where a number of random parameter samples are drawn from the variational posterior. Using this technique, we obtain the following approximation:

$$\mathcal{F}(D, \theta) \approx \sum_{i=1}^{n} \log q(w^{(i)}|\theta) - \log P(w^{(i)}) - \log P(D|w^{(i)}) \tag{5.13}$$

Note that with this approach the KL-divergence (thus the complexity cost) is not computed analytically, although this would be an option for some kinds of priors. Blundell et al. justify the choice of the Monte Carlo-approach with empirical experiments where they found no performance difference compared to an analytically one. Moreover, they eventually opted for a prior that cannot be computed analytically (Blundell et al. 2015).

As variational posterior $q(w|\theta)$, a diagonal Gaussian posterior is chosen. This means that with network parameters $w_1, ..., w_n$, each distribution $q(w_i|\theta)$ with $i \in \{0, ..., n\}$ is an univariate Gaussian and the covariance matrix of $q(w|\theta)$ is diagonal. Conceptually, the posterior can be described with two variables per network parameter, mean and standard deviation (Blundell et al. 2015).

In practice, an additional observation is taken into consideration: standard deviations are always positive and neural network weights are typically also allowed to be negative (Dürr et al. 2020). Therefore, the network parameters are set to $\theta = (\mu, \rho)$. The standard deviations σ are obtained with $\sigma = \log(1 + \exp(\rho))$ which is referred to as *softplus*-function. A bayesian neural network following this approach can be thought as having twice as many parameters than a standard neural network with the same functionality. If the standard network has n parameters, μ and ρ in the bayesian network are n-dimensional vectors. Thus there are $2 \cdot n$ parameters. Conceptually, each parameter in the standard network becomes an univeriate Gaussian distribution parametrized by mean and standard deviation. This concept is visualized in figure 5.3.

Samples from the variational posterior are obtained with

$$w = \mu + \log(1 + \exp(\rho)) \circ \epsilon \tag{5.14}$$

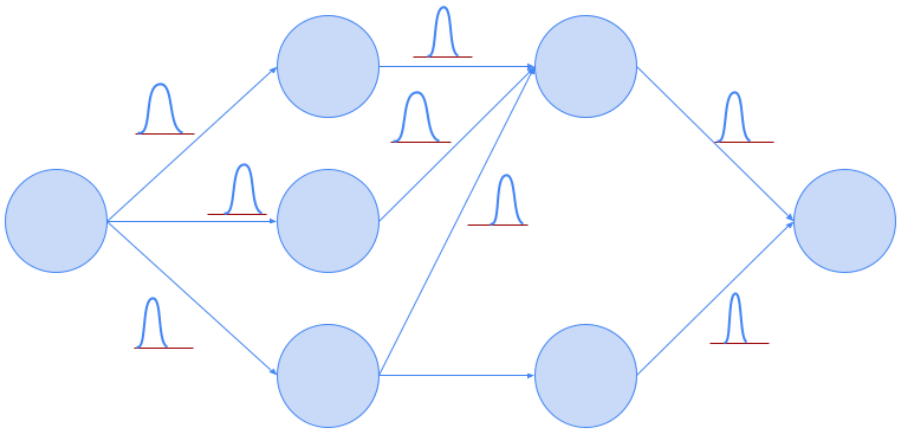

Figure 5.3: Concept of a Bayesian neural network. It can be observed that each parameters that one would expect to be a single value in a standard neural network is a probability distribution (for example Gaussian with parameters μ and σ.

where $\epsilon \in [0, 1]$ is a parameter-free (independent) sample. It is important that e is parameter-free and the only random variable involved in the computation of w, as otherwise it would not be possible to compute gradients. The technique of inserting an independent sample ϵ is referred to as *reparameterization trick* (Kingma et al. 2014).

As already mentioned, different distributions can be chosen as prior. Blundell et al. (2015) propose the following *Scale mixture prior*:

$$P(w) = \prod_j \pi \mathcal{N}(w_j|0, \sigma_1^2) + (1 - \pi)\mathcal{N}(w_j|0, \sigma_2^2) \tag{5.15}$$

As it can be seen, the prior combines two univariate Gaussians with zero mean and different variances. Blundell et al. (2015) propose pre-setting the variances σ_1 and σ_2 as hyperparameters instead of learning them. π is another hyperparameter. The authors also propose $\sigma_1 > \sigma_2$ and $\sigma_2 \ll 1$. Altogether, the scale mixture prior provides a heavy tail on the one hand, on the other hand it "encourages" the weights to tightly concentrate around zero (Blundell et al. 2015).

So far, it was assumed that gradient descent without mini-batches is used. However,

Bayes by Backprop can also be adapted for mini-batch gradient descent. Suppose a training data set $\mathcal{D}$ that is split into M mini-batches $\mathcal{D}_1, ...\mathcal{D}_M$. As introduced in chapter 2, mini-batch gradient descent performs an optimization step for each mini-batch. The loss function for mini-batch i is therefore:

$$\mathcal{F}_i(D_i, \theta) = \pi_i KL[q(w|\theta)||P(w)] - \mathbb{E}_{q(w|\theta)}[\log P(D_i|w)] \tag{5.16}$$

π_i is a weight for the complexity cost; the weights are constrained to $\sum_i^M \pi_i = 1$, but can be distributed arbitrarily among the mini-batches. In the simplest case, each weight π_i corresponds to $\frac{1}{M}$. In this assignment, no difference among mini-batches is made when it comes to the chosen "trade-off" between complexity cost and likelihood cost. In this thesis, another assignment given by $\pi_i = \frac{2^{M-i}}{2^M-1}$ was used. This assignment was suggested by Blundell et al. (2015) and assigns earlier mini-batches within an epoch a higher complexity cost compared to later mini-batches (Blundell et al. 2015). In the next section, *Bayes by Backprop through time*, and adaption of Bayes By Backprop for recurrent neural networks, is introduced.

5.5 Bayes By Backprop Through Time

5.5.1 Definition and relation to the given thesis

As seen in section 3, parameters are shared among cells in an unrolled representation of a recurrent neural network. Therefore, if the network is unrolled for T steps, each parameter gets T gradient updates. Fortunato et al. propose an algorithm named *Bayes By Backprop Through time* which applies Bayes By Backprop to recurrent NNs. In principle, the algorithm described in the previous section can be applied to the recurrent case without modifications (Fortunato et al. 2017). The authors assume a sequence of length T and a recurrent NN that captures that sequence (therefore, its unrolled representation has T cells). For the whole sequence, the loss function is given by:

$$\mathcal{F}(D, \theta) = KL[q(w|\theta)||P(w)] - \mathbb{E}_{q(w|\theta)}[\log P(y_{1:T}|w, x_{1:T})] \tag{5.17}$$

$x_{1:T}$ is the complete input sequence used for training, $y_{1:T}$ is the complete set of ground truth labels. As $(x_{1:T}, y_{1:T})$ is the training data, function 5.17 can be seen as equivalent to the standard ELBO-loss (5.12). As a further step, Fortunato et al. point out that training of RNNs is oftentimes done with truncated samples (in the course of generating such samples, $x_{1:T}$ and $y_{1:T}$ are truncated into shorter sequences) and those samples are combined into mini-batches. Therefore, the loss function for mini-batch gradient descent in RNNs does not deviate from equation 5.16.

It is important to point out that in the work for this thesis, the available training data is not treated as a contiguous sequence, in the sense that it is not split into truncated samples with the goal to let the network examine each sample exactly once. As presented in chapter 4, truncated sequences are instead constructed in a way such that we obtain exactly one prediction for each time-step. Considering that there is also a "warmup"-phases where the network does not produce any output, it follows that the network "sees" most of the input vector multiple times during an epoch. For this reason, despite involving an RNN, the scenerio can be seen as more related to the one presented by Blundell et al. (2015) than the one presented by Fortunato et al. (2017).

5.5.2 Posterior sharpening

Fortunato et al. propose a technique named *posterior sharpening*. It aims to make the posterior distribution more accurate by adding side information about a specific mini-batch (Fortunato et al. 2017). This technique was also applied in the implementation for this thesis. Posterior sharpening makes use of a hierarchical posterior of the form

$$q(w|\phi,(x,y)) = \int q(w|\theta,(x,y)) \cdot q(w|\theta) d\theta \tag{5.18}$$

where $q(w|\theta)$ corresponds to the original posterior. Let $g_\theta = -\nabla_\theta \log P(y|\theta,x)$ be the gradient computed with the likelihood cost of the posterior with respect to the posterior parameters. Then we compute:

$$q(w|\theta,(x,y)) = \mathcal{N}(w|\theta - \eta \circ g_\theta, \sigma_0^2 I) \tag{5.19}$$

were η can be seen as a per-parameter learning rate and σ_0 is a scalar hyperparameter. Altogether, the following algorithm is obtained:

Algorithm 1: BBB with Posterior Sharpening

Sample a mini-batch (x,y) of truncated sequences.
Sample $w \sim \mathcal{N}(w|\mu,\rho)$.
Let $g_w = -\nabla \log P(y|w,x)$.
Sample $\theta \sim \mathcal{N}(\theta|w - \eta \circ g_w, s_0^2)$.
Compute the gradients w.r.t (μ,ρ,η).
Update (μ,ρ,η).

5.6 Evaluating a Bayesian Neural Network

Let us first recall how prediction performance is evaluated in a non-bayesian neural network for time series forecasting. Let y_e be a sequence of ground truth values from

the test dataset, and let $\hat{y}_e$ be a sequence of corresponding predictions. Then prediction performance can be measured with a performance measure $M(y, \hat{y})$. In a bayesian setting however, the output of the network is not a single value, but can be seen as a probability distribution like in probabilstic models mentioned in previous sections of this chapter. However, this probability distribution is not given explicitly, it is approximated by drawing a certain number of samples. The number of samples taken per time-step during evaluation is a hyperparameter and does not have to correspond to the number of samples drawn during training. If 10 samples are drawn during evaluation, $\hat{y}_e$ is a set given by $\{\hat{y}_e{}^{(1)}, \hat{y}_e{}^{(2)}, ..., \hat{y}_e{}^{(10)}\}$. For standard performance metrics, the mean of $\hat{y}_e$ can be taken, the desired performance measure can than be obtained by computing $M(y_e, \mu_{\hat{y}_e})$ in the same way as in a standard neural network. By calculating the standard deviation $\sigma_{\hat{y}_e}$ (or alternative deviation measures) it is also possible to investigate the uncertainty of the network. Chapter 6 gives further details on how uncertainty estimation is done in order to measure performance of bayesian networks.

6 Methodology

6.1 Datasets

6.1.1 The CAMELS-US dataset

Experiments for this thesis were conducted on two datasets. The first one is the CAMELS-US dataset (Addor et al. 2017, Newman et al. 2014). CAMELS stands for "Catchment Attributes for Large-Sample Studies", it is a freely available dataset of 671 catchments across the contiguous United States (CONUS) (Frederik Kratzert et al. 2018). The dataset contains lumped forcings and observed discharge values at a daily resolution. The available timespan for the data slightly varies between catchments; however, for all catchments data is available between 1980 and 2010. Within the dataset, different data sources for the meteorlogical data are available (Daymet: Thornton et al. 2016, Maurer: Livneh et al. 2013, NLDAS: Xia et al. 2012). All sources take measurements based on a grid and aggregate the measurements on a catchment-level. They measure day length, precipitation, shortwave downward radiation, maximum and minimum temperature, snow-water equivalent and humidity. Kratzert et al. (2018) use the Daymet data since it has the highest spatial resolution, and exclude the snow-water equivalent and day length from the set of forcings. As a consequence, 5 forcings are left: precipitation, shortwave downward radiation, maximum and minimum temperature, and humidity. We also opted for the same setting in order to draw a comparison to the standard-LSTM-implementation of Kratzert et al. (2018). In addition to forcings, the CAMELS-US-dataset provides runoff values for the individual time step. Therefore, the task can be seen as a kind of rainfall-runoff-task as the model aims to predict runoff based on meteorological forcings like precipitation.

The 671 catchments are groups into 18 hydrological units (HUCs) that represent the drainage area of either a major river or the combined drainage area of multiple rivers (Frederik Kratzert et al. 2018, Seaber et al. 1987). Kratzert et al. (2018) use 4 of the 18 HUCs, namely the New England region, the Arkansas-White-Red region, the South Atlantic-Gulf region and the Pacific Northwest region. These 4 HUCs consist of 241 catchments in total and cover a wide range of meteorological conditions. For comparability reasons, we also used the same subset of catchments for our experiments.

6.1.2 The Regen catchment

The second dataset used in this thesis consists of meteorological data from the Regen catchment (Germany) and was provided by the Flood Forecast Center from the Bavarian Water Authorities *(Landesamt für Umwelt (LFU))* (Fiedler 2020). In contrast to the Daymet source from the CAMELS-US-dataset, the data from the Regen catchment is not aggregated on a catchment-level. Instead, the data is taken from 55 meteorological stations and 20 discharge gauges. It is important to point out that discharge stands for streamflow in this context. Network predictions are evaluated against ground-truth-streamflow-values as it is often done in rainfall-runoff-modeling (Gauch et al. 2020). All meteorological stations measure the precipitation, 29 stations additionally measure other meteorological parameters (relative humidity, air temperature, dew point temperature, global radiation, air pressure, wind speed, sunshine duration) (Fiedler 2020). There can be therefore several available forcings per station. In total, there are 185 forcings (compared to 5 for CAMELS-US). All forcings are recorded in hourly resolution between 2003 and 2018. However, we also conducted experiments with a daily resolution. In the course of pre-processing the data for these experiments, the forcings are aggregated in different ways to obtain sensible measurements: for precipitation, global radiation and sunshine duration the cumulative sum is taken, for discharge, relative humidity, air pressure, dew point temperature and wind speed the mean value is used. The parameter for the air temperature is duplicated such that the daily dataset consists of one parameters for the minimum temperature and one for the maximum temperature. With this modification, the daily dataset consists of 214 forcings. The same re-sampling technique to daily resolution is used by Fiedler (2020).

Within the dataset, gaps of missing data that for instance occur due to system failure or maintenance are present. Fielder (2020) proposes methods to tackle gaps of missing data, so called *imputation techniques*. For daily resolution, Fiedler (2020) suggests the *normal ratio method with respect to distance (NRM_D)*, for the hourly dataset he suggests linear regression. He also provides modified datasets with those imputation techniques pre-applied; we used these datasets for our experiments.

6.2 Summary of applied methods

As a preliminary step, a Bayesian LSTM following the algorithm *Bayes by Backprop through time* proposed in Fortunato et al. (2017) is implemented. The implementation must have the ability to cope with the datasets used in this thesis (CAMELS-US and Regen catchment) and to pre-process the data in an appropriate way. In the first part of the experiments, performance on the CAMELS-US dataset was investigated; in the second part performance on the Regen catchment was examined.

The performance of the obtained models was evaluated using the many-to-one-architecture described in section 3. Hyperparameter tuning was done with that architecture, either on a single catchment or on the 27 catchments from HUC 01. As a first step of hyperparameter tuning, different prior setting and weight initializations were investigated. The architecture resulting from the previous step was used to determine a well performing number of hidden units, network layers, batch size, optimizer and learning rate. We then evaluated the obtained architecture with an additional input that consists of a periodic signal representing a timespan of one year and examined if the modification improves performance. Moreover, we examined the optimal number of samples for Monte Carlo sampling. Finally, we evaluated the obtained architecture on the set consisting of 241 catchments proposed by Kratzert et al. (2018) and compared the bayesian LSTM (with Bayes by Backprop) with non-bayesian approaches in terms of accuracy. Moreover, the quality of the uncertainty estimates was examined. To evaluate the performance of the architecture when predicting multiple time steps ahead, the architectures described in section 3 were considered. Single-shot-model were compared to the many-to-many-models on HUC 01; a many-to-many-based model was also trained and evaluated on the whole set of 241 catchments and compared to the many-to-one-model. We also considered a network version that considers discharge as an additional input measure. Both a modified many-to-one-model and a modified many-to-many-model that consider discharge as input were trained and evaluated on the aforementioned set of 241 catchments. As a final experiment, the architectures resulting from the previous step were trained and evaluated on the Regen catchment. Fiedler (2020) proposes training networks for both daily and hourly resolution respectively with 6 different sets of input features. Following the same approach, we trained 12 many-to-one-based networks on the Regen catchment and evaluated their performance.

6.3 Data split for CAMELS-US

For the observed set of CAMELS-US-catchments, meteorological forcing and discharge data is available from 1980 to 2010. We split the time period into three parts: the period from 1980 to 1990 was used for training the network, the period from 1990 to 1995 for validation during hyperparameter tuning and the period from 1995 to 2010 as a test set to evaluate the final architectures resulting from hyperparameter tuning. This approach differs from the split used by Kratzert et al. (2018) as they do not use a dedicated validation set for hyperparameter tuning. However, not much hyperparameter tuning was required in the study as they inferred most of the hyperparameters from experiments on a different dataset. In this study however, it was certainly necessary to

perform hyperparameter tuning; therefore, we decided to use a dedicated validation set, albeit potentially not optimal for network training as this reduces the amount of training data (Frederik Kratzert et al. 2018).

6.4 Fundamental implementation

A preliminary step is assembling a suitable implementation for the task. As there are numerous examples of recent work in the field of deep learning and time series forecasting that rely on Python as a programming language (Chollet 2017, Frederik Kratzert et al. 2018, Esposito 2020), the chosen programming language in the work for this thesis was Python too (more precisely: Python 3.8). In terms of functionality, the implementation has to fulfill the following criteria:

1. It supports importing and processing the datasets used in this study and making them suitable for time series forecasting tasks with neural networks. This includes making them compatible with the models mentioned in chapter 4 (many-to-one, single shot and many-to-many).

2. It features an implementation of an LSTM network that supports *Bayes by Backprop through time* which was introduced in chapter 5. The implementation should both feature the algorithm presented by Blundell et al. (2015) as well as the posterior sharpening algorithm introduced by Fortunato et al. (2017).

Esposito (2020) provides an implementation that fulfills criterion (2) to a large extend. It is realized with the deep learning library *PyTorch*. The implementation also supports Bayesian LSTM networks (Esposito 2020). The coarse programmatic structure of such a network is depicted in figure 6.1.

The raw implementation required several modification to fit the given task. In its original form, the implementation abstracts from the distinction between complexity cost and likelihood cost and only reveals the total loss to the interface. However, it is desirable to distinguish between the two parts of the cost when examining performance. Moreover, the implementation does not support computing the complexity cost during the evaluation phase, which we considered a drawback. However, both extensions could be added to the existing library by minor modifications.

It remained to fulfill criterion (1). Kratzert et al. (2018) provide methods for loading the CAMELS-US-dataset, truncating it in ways suitable for time series forecasting, eliminating invalid sequences and scaling it to obtain suitable values for neural network training. Truncating the data means splitting it into input features (in this task the forcings) and output features (in this task the only output feature is the discharge) and

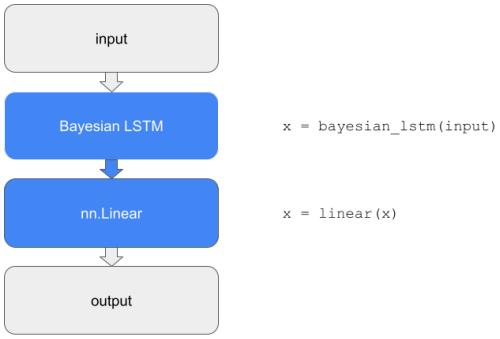

Figure 6.1: Programmatic structure of a Bayesian LSTM network. The implementation provided by Esposito features a module named *BayesianLSTM*, together with the fully-connected-layer-module *nn.Linear* from *PyTorch* it is possible to implement a network that consist of a Bayesian LSTM network followed by a fully connected neural network layer that maps the output vector from the LSTM network to actual predictions.

to arrange it into input-label-pairs suitable for the model. If for example a many-to-one-model with a window size of 30 is used, forcings are consulted for all 30 time steps; in addition, the discharge value of the 30th time step is used as reference. The process of creating adequate arrays is referred to as *re-shaping the data* hereafter. For the many-to-one-model, re-shaping is realized with the following Python-code:

```
1 num_samples, num_features = x.shape
2
3 x_new = np.zeros((num_samples - seq_length + 1, \
4     seq_length, num_features))
5 y_new = np.zeros((num_samples - seq_length + 1, 1))
6 for i in range(x_new.shape[0]):
7     x_new[i, :, :num_features] = x[i:i + seq_length, :]
8     y_new[i, :] = y[i + seq_length - 1, 0]
9 return x_new, y_new
```

The above code fragment generates training data from a plain list of forcings x and a list of discharge values y. Lines 3-5 initialize appropriate data structures. The data structure for the forcings is 3-dimensional as each training sample consists of a sequence of time steps, and each such time steps consists of several forcings. The data structure for the discharge values is 2-dimensional a there is only one discharge value per sequence.

Lines 6-8 fill these data structures with the corresponding data. As all methods are customized for using them in combination with a neural network implemented in PyTorch, they could be seamlessly integrated into the implementation provided by Esposito (2020). After the integration we obtained a holistic implementation that makes used of *PyTorch* for the neural networks implementation and of the Python-libraries *pandas* and *numpy* for processing datasets and arrays. Without further modifications, the methods provided by Kratzert et al. (2018) only support preparing the data for a many-to-one-based model. To support the remaining models, two aspects have to be considered.

1. The dataset has to be re-shaped accordingly.
2. The LSTM network has to be modified to support the respective model.

The extensions that were implemented in order to fit the remaining models are presented in the following section.

6.5 Extensions for predicting multiple time steps

6.5.1 Extension for the single-shot-model

A central aspect for the single-shot-model is that the network should not consider any input data that belongs to the prediction window. In combination with the criterion that the model should not produce multiple discharge values for any time steps, this leads to the following re-shaping procedure:

```
num_samples, num_features = x.shape

x_new = np.zeros(((num_samples - seq_length + 1) // seq_single_shot, \
    seq_length, num_features))
y_new = np.zeros(((num_samples - seq_length + 1) // seq_single_shot, \
    seq_single_shot))
# Current index in old arrays
i = 0
for index_new in range(x_new.shape[0]):
    x_new[index_new, :, :num_features] = x[i:i + seq_length, :]
    y_new[index_new, :] = \
        y[i + seq_length - 1:i + seq_length + seq_single_shot - 1, 0]
    i += seq_single_shot
return x_new, y_new
```

In the above Python-code, *seq_single_shot* refers to the length of the prediction window s_p, thus to the number of time steps the network predicts. Note that due to the fact that the network should only produce one prediction per time step there are less training samples than in the many-to-one-case (if *seq_single_shot* > 1). This reduction of the training samples number is achieved by diving by *seq_single_shot* in lines 3-6 when setting the sizes for the first dimensions of the data structures. Moreover, the data structure for the discharge values (initialized in lines 5-6) contains multiple discharge values per training sample.

In terms of the LSTM network implementation, the only modification required when adapting many-to-one to single shot was changing the output vector size of the final fully connected layer from 1 to the length of the prediction phase. At this point it is emphasized again that many-to-one and single shot only differ from a modelling perspective, not from an architectural one. From an architectural point of view, single shot can be seen as yet another form of a many-to-one-architecture.

6.5.2 Extension for the many-to-many-model

In the many-to-many-case, the network also considers forcings from the prediction window. Therefore, data re-shaping differs from the one used in the single-shot-case in the sense that ground truth discharge values are added *before* the end of the sequence of forcings, not *after* it. The following procedure is used for the many-to-many-case:

```
1  num_samples, num_features = x.shape
2
3  x_new = np.zeros(((num_samples - seq_length + 1) // seq_single_shot, \
4      seq_length, num_features))
5  y_new = np.zeros(((num_samples - seq_length + 1) // seq_single_shot, \
6      seq_single_shot))
7  # Current index in old arrays
8  i = 0
9  for index_new in range(x_new.shape[0]):
10     x_new[index_new, :, :num_features] = x[i:i + seq_length, :]
11     y_new[index_new, :] = \
12         y[i + seq_length - seq_single_shot:i + seq_length, 0]
13     i += seq_single_shot
14 return x_new, y_new
```

The initialization of the data structures in lines 3-6 corresponds to the one from the single shot model. The difference lies in the way the new data structure for the discharge values is filled in lines 11-12. Instead of filling in discharge values starting

from the last time step of the sequence, they are filled in such a way that they *end* at the last time step of the sequence.

In terms of the network architecture, the predictions are now assembled from multiple output vectors of the LSTM network. This has to be reflected in the implementation and is realized using a loop that converts each relevant LSTM-output-vector into a prediction. The following Python-code realizes this procedure:

```
1 x_, _ = self.lstm_1(x)
2 output = torch.zeros(x.shape[0], self.days_future)
3 counter = -self.days_future
4 while counter <= -1:
5     x_p = x_[:, counter, :]
6     x_p = self.linear(x_p)
7     output[:, counter + self.days_future] = x_p.flatten()
8     counter += 1
9 return output
```

Line 2 initializes a data structure that collects the predictions for different time steps. In each step in the loop from line 4 to line 8 a specific output vector from the LSTM network is extracted (line 5), converted to a prediction using a fully connected layer (line 6) and finally inserted into the data structure (line 7).

6.6 Extensions for discharge as input measurement

So far, only forcings were considered as input measurements for the network. However, in a discharge prediction task it is also possible to consider discharge values as inputs (Gauch et al. 2020). Specially, it is common to consider discharge values from the warmup window, process-based hydrologic model typically consider discharge values (or values closely related to discharge) in the calibration period too. An example of such a process-based model is *LARSIM* (Large Area Runoff Simulation Model) which describes continuous runoff processes and is used for the simulation of flood protection planning, land use changes, and effects of climate change on water resources (Ludwig et al. 2006). We decided to also experiment with models that take discharge value as input measurements. In case of the Daymet data from the CAMELS-US dataset, the same discharge values that were previously used as ground truth can also be used as input measurements.

Besides adding a new dimension to the input vectors, including discharge values as input also requires implementing the encoder-decoder-approach introduced in chapter 4: it has to be ensured that the network only considers discharge values from the warmup window. In a many-to-one-setting for example it is obviously prohibitive

for the network to access the discharge value for the last time step before making a prediction for that same step. Implementing this approach requires using at least two LSTM networks that differ in terms of input size. One network, referred to as the *encoder*, receives input measurements from the warmup window, another network, known as the *decoder*, receives input measurements from the prediction phase and also produces output values. The following Python-code illustrates a forward pass through such an encoder-decoder-network:

```
1  x_warmup = x[:, :x.shape[1] - self.days_future, :]
2  x_pred = x[:, x.shape[1] - self.days_future:, [0, 1, 2, 3, 4, 6]]
3  x_, (h_t_1, c_t_1) = self.lstm_1(x_warmup)
4  if self.num_layers == 2:
5      _, (h_t_2, c_t_2) = self.lstm_2(x_)
6  x_, _ = self.lstm_1_pred(x_pred, (h_t_1, c_t_1))
7  if self.num_layers == 2:
8      x_, _ = self.lstm_2(x_, (h_t_2, c_t_2))
9  x_ = self.linear(x_)
10 x_ = x_.flatten(start_dim=1)
11 return x_
```

A special feature of the code fragment is that it also supports two LSTM networks that are "stacked" onto each other (for an introduction on such an architecture refer to chapter 3). The "top" LSTM network that produces predictions is referred to as *lstm_2*. There are two versions of the "bottom" LSTM network: one for the warmup window (referred to as *lstm_1*) and one for the prediction window (referred to as *lstm_1_pred*). The input arrays for warmup- and prediction windows are created in lines 1-2; it can be observed that the array for the prediction window consists of one input measurement less (it does not feature the discharge measurement). In lines 3 and 5, it can be observed that the last hidden state and cell state of the encoder network (consisting of *lstm_1* and potentially *lstm_2*) are stores in variables. The decoder network (consisting of *lstm_1_pred* and potentially *lstm_2*) is initialized with those stored variables and finally produced output values.

6.7 Hyperparameter tuning

Hyperparameter tuning refers to the empirical search for beneficial hyperparameters. As mentioned in chapter 2, performance during hyperparameter tuning is usually measured on a dedicated validation set. We also followed this approach in this thesis. During hyperparameter tuning and parts of the final evaluation, a many-to-one-model

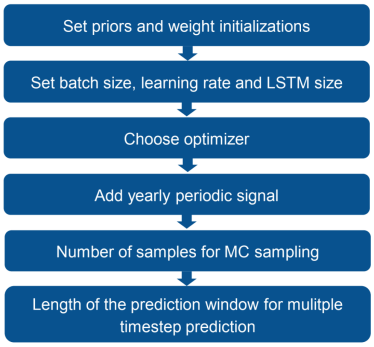

Figure 6.2: Overview of hyperparameter tuning steps in the order they were conducted.

was used. Hyperparameter tuning was only conducted for the CAMELS-US-dataset, it was either done on a single catchment or on the 27 catchments from HUC 01.

6.7.1 Investigation of different prior setting and weight initializations

First, different prior settings and weight initializations were investigated. As a prior, we restricted our experiments to the scale mixture prior proposed in Blundell et al. (2015). Therefore, hyperparameters related to the prior are σ_1, σ_2 and π. With regards to the initialization of the weights (thus the initialization of the variational posterior) recall that the posterior is a diagonal Gaussian and can therefore be seen as a combination of univariate Gaussians. We initialized each of the univeriate Gaussian equally, such as in total there are only two hyperparameters in terms of the weight initialization: μ and ρ.

6.7.2 Investigation of different batch sizes, learning rates, and LSTM-sizes

The architecture resulting from the previous step was used as a basis to determine a well performing number of hidden units, network layers, batch size, and a beneficial optimizer and learning rate. In terms of the hidden units and network size, different approaches were taken in the field of discharge prediction with LSTM networks. Kratzert et al. (2018) use two LSTM networks with 20 hidden units each, whilst in another implementation, the same authors use a single network with 256 units (Frederik Kratzert et al. 2019). The authors considered this an improvement on the architecture previously mentioned. These proposals were taken as a starting point for this work; however, as Bayesian networks can vary heavily from their non-Bayesian pendants, we considered evaluating different hidden unit numbers an important part of hyperparameter tuning, although well-performing architectures were already proposed. Regarding the optimizer, *Adam* was used in a variety of related tasks

(Esposito 2020, Frederik Kratzert et al. 2018); it was therefore also used in this work if not stated otherwise. In addition, we also compared it to alternative optimizers during hyperparameter tuning.

6.7.3 Addition of a yearly periodic signal

In *Time series forecasting* n.d. a periodic signal representing time is suggested as an additional input for a time series forecasting task. This is useful as it provides temporal information to the network and is considered advantageous compared to plain timestamps as network inputs, capturing seasonal dynamics is more difficult to impossible for the network in the latter case. One possibility to include a periodic signal is to consult evaluations of a sine function scaled such that one period corresponds to a year in the input data sequence. This approach was also taken in this work.

6.7.4 Investigation of optimal length of prediction phase

We also considered networks making predictions for multiple time step (realized with the single-shot- or many-to-many-model). It was examined how different lengths of prediction windows s_p affect network performance. A comparison between a single-shot-model and a many-to-many-model was conducted as we expected to find performance differences in a side-by-side-comparison between the two models.

6.8 Evaluation of the final architecture

6.8.1 Evaluation on the CAMELS-US-dataset

In this work, the final evaluation consists of training an individual network for each catchment in a defined set. There are alternative approaches commonly used with LSTM networks for discharge prediction; those are often related to training a single network for multiple catchments which is oftentimes referred to as *regionalization* (Frederik Kratzert et al. 2018). However, investigating such methods was beyond the scope of this thesis. For evaluation, we used the set proposed by Kratzert et al. (2018) that consists of 241 catchments. We consulted three different models: firstly, we trained a many-to-one-model based on the architecture found during hyperparameter optimization - this allowed evaluating the resulting networks against the results from Kratzert et al. (2018) that are based on a non-Bayesian LSTM network. Secondly, we consulted discharge as an additional input measurement and conducted the same training on 241 catchments. Lastly, we also evaluated a many-to-many-based model on the chosen set of 241 catchments.

6.8.2 Evaluation on the Regen catchment

We also conducted experiments on the Regen catchment, Germany. As these experiments are not the primary scope of this thesis, no additional hyperparameter tuning was done to adapt the model to the Regen catchment. Instead, the many-to-one-based architecture obtained during hyperparameter tuning for the CAMELS-US-dataset was re-used for the experiments on the Regen catchment. The experiments are strongly based on the architectures used by Fielder (2020), whereby he uses a non-bayesian LSTM network instead of a bayesian one. The normal-holdout-based split into training, validation and set set was taken over from Fiedler (2020); we used the timespan from 2005 to 2014 as training set and the timespan from 2016 to 2018 as test set. As we did not conduct hyperparameter tuning, a dedicated validation set was not necessary. In this case, extending the training set would be sensible; however, we refrained from doing so in order to better compare our results with the the ones from Fiedler (2020). In total, there are 185 available input measurements in the dataset for hourly resolution (214 in the modification for daily resolution). In the course of data preparation (for example imputation) Fielder (2020) does not consider all measurements useful; after the pre-processing step, he obtains 75 input measurements for hourly resolution and 86 for daily resolution. As input vectors for the network are still rather high-dimensional, Fiedler (2020) conducts a sensitivity analysis to investigate which of the input measurements are required for learning. In the course of this analysis, he distinguishes 5 different input feature sets and trains a network for each of these sets. In addition, he also conducts experiments for a set containing all input features. As all of these tests are carried out for daily and hourly resolution separately, there are 12 settings in total. We also followed the same approach which leads us to training 12 bayesian neural networks for the Regen catchments.

6.8.3 Performance metrics

To evaluate a bayesian neural network, it is necessary to both examine how reasonable predictions are with regards to the ground truth values (referred to as *accuracy* in this thesis) as well as how well the network captures epistemic and aleatoric uncertainty (referred to as *uncertainty estimation*). This section presents the used performance metrics for both evaluation goals.

Performance metrics for accuracy

It is important to point out that there exist several ways of quantifying the error when evaluating a network. Let y_t be a sequence of ground truth values, and let $\hat{y}_t$ be a sequence of corresponding predictions, both taken from the training data set. Further,

let $L(y, \hat{y})$ be the loss function that is used to update the network parameters during training. Then a backward pass during training computes $L(y_t, \hat{y}_t)$ which can be seen as a form of quantifying the predictive error that occurs during training. In case of the bayesian LSTM network used in this task, L is the ELBO-loss introduced in 5. Let y_e and $\hat{y}_e$ be sequences of ground truth values and predictions respectively, but taken from the test data set. It is conceivable to evaluate the quality of $\hat{y}_e$ by computing the same loss function $L(y_e, \hat{y}_e)$. However, it is equally possible to consider alternative performance measures $M(y, \hat{y})$. In this thesis, multiple performance metrics were used for evaluation. As it is a common performance measure for regression problems, one of the measures used in the work for this thesis is the *mean squared error (MSE)*. It is given by:

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 \tag{6.1}$$

where n is the length of the sequences y and $\hat{y}$. It can be interpreted as a sum of the "distances" between real and predicted value, whereat each such "distance" is raised to the power of two. If the effect of the exponentiation is not desired, the *root mean squared error (RMSE)* can be used as an alternative. It additionally involves a square root around the term given by the MSE. Therefore, the RMSE is

$$RMSE(y, \hat{y}) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2} = \sqrt{MSE(y, \hat{y})} \tag{6.2}$$

A metric commonly used in discharge prediction is the *Nash–Sutcliffe model efficiency coefficient (NSE)* (Gauch et al. 2020). It is given by:

$$NSE(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{n} (y^{(i)} - \overline{y})^2} \tag{6.3}$$

where $\overline{y}$ is the mean discharge over all ground truth values. The denominator is therefore the variance of the ground truth values. Therefore one can conclude that an NSE of 1 corresponds to a perfect predictor (numerator is 0) and an NSE of 0 corresponds to a predictor that constantly predicts the mean of the ground truth values. Overall, NSE values range from $-\infty$ to 1, where a higher value stands for better prediction performance.

An alternative to the NSE in the field of discharge prediction is the *Kling-Gupta Efficiency (KGE)* (Gupta et al. 2009) Knoben et al. 2019. It is given by

$$KGE(y, \hat{y}) = 1 - \sqrt{(r-1)^2 + \left(\frac{\sigma_{\hat{y}}}{\sigma_y} - 1\right)^2 + \left(\frac{\mu_{\hat{y}}}{\mu_y} - 1\right)^2} \tag{6.4}$$

where r is the linear correlation between predictions and ground truth values, μ is the mean and σ is the standard deviation. Like in the case of the NSE, a KGE of 1 indicates perfect prediction performance, and values range from $-\infty$ to 1 too. However, there is no intuitive interpretation of a KGE of 1 and linguistic terms assigned to specific KGE ranges (like "good" or "bad") vary between authors (Knoben et al. 2019). Rogelis et al. (2016) for instance consider model performance to be "poor" for a KGE above 0.5 (to be precise, negative KGEs are not mentioned) (Rogelis et al. 2016).

A set of other performance metrics used in this thesis is related to the *flow-duration curve (FDC)*, a visualization method commonly used with time series of discharge values. When creating a flow-duration curve, the values are sorted descendingly, a graph then plots observed discharge values against the percentile that exceed that value (Singh 2015). An example of a flow-duration curve is depicted in Figure 6.3.

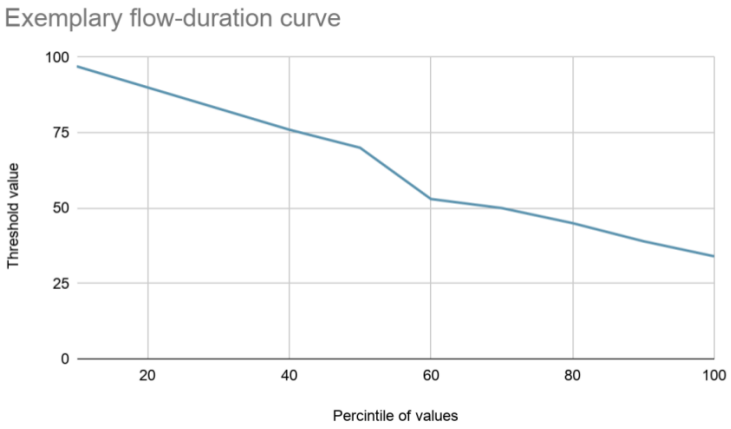

Figure 6.3: Example for a flow-duration curve. It can be observed that 20% of the values exceed a value of 90, whereas 100% of the values exceed a value of 34, which was therefore the smallest value measured.

Another concept related to the metrics used in this thesis is the *percentage bias*. Ranging from $-\infty$ to ∞, its values express whether predictions tend to be smaller or bigger than ground truth values. It is given by:

$$PBIAS(y, \hat{y}) = 100 \cdot \frac{\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})}{\sum_{i=1}^{n} y^{(i)}} \tag{6.5}$$

From the flow-duration curve, the measures *FHV (Peak flow bias)*, *FLV (Low flow bias)* and *FMS (Slope of the middle section)* are derived (Yilmaz et al. 2008). In the work for this thesis, the FHV represents the percentage bias of the top 2% values, the FLV the

percentage bias of the bottom 30% values and the FMS the percentage bias of the values in the middle range from 20% to 70% in the flow-duration curve.

Performance metrics for uncertainty estimation

There are two essential aspects that have to be considered when evaluating how beneficial uncertainty estimates produced by a network are. Those aspects are *reliability* and *resolution* (Klotz et al. 2020). Reliability expresses how consistent the uncertainty estimates are with respect to the ground truth values. Recall that a bayesian neural network typically produced multiple predictions per time step. Now one could imagine a scenario where the network tends to mostly produce under-estimates. Such a network would not be considered as very reliable as most of its predictions would likely lie below the ground truth value. The mean of the predictions for a time step would likely also lie below the ground truth value. On the other hand, a network whose mean predictions for specific time steps coarsely match the corresponding ground truth values would be considered more reliable. A measure for reliability proposed by Klotz (2020) is the *probability plot* (more precisely, the probability-probability plot or P-P plot). It plots the cumulative density functions for ground truth values and obtained predictions against each other. In this thesis, we decided to measure reliability by constructing a *prediction interval* (Tagasovska et al. 2019). A function that determines a prediction interval takes the different predictions obtained at a single timestep as an input, computes the empirical mean μ and empirical standard deviation σ of these predictions set and outputs an interval $[\mu - c\sigma, \mu + c\sigma]$ where c is a hyperparameter. If we switch from single time steps to time series, such a prediction interval is constructed by combining the individual prediction intervals for each of the steps. Such a prediction interval can be denoted with a percentile, an 80% prediction interval for instance indicates that 80% of the ground truth values in the given time series lie within the bounds of the prediction interval. In the work for this thesis, c was fixed at 2. As this measurement for reliability is straightforward to interpret, it was used both during hyperparameter tuning and final network evaluation.

The other aspect that has to be considered when evaluating uncertainty estimates is resolution. It measures the "sharpness" of the obtained distributions (Klotz et al. 2020). This can for example be interpreted as the average width of the prediction interval. One can easily see that the measurement previously introduced is not sufficient to express resolution. Imagine a scenario where the predictions for single time steps differ heavily from each other, the obtained prediction interval is therefore very wide. Then it is likely that this prediction interval has a high percentile (thus most ground truth values will lie within the prediction interval). However, such a wide prediction interval may have little to no practical use as it often would make no concrete statement. In the work for

this thesis, the *mean absolute deviation*, the *empirical variance* and the *empirical standard deviation* are used as measures for resolution (for the latter 2 we use Bessel's correction). Those measures are also suggested by Klotz (2020). Empirical variance and empirical standard deviation are assumed to be already familiar to the reader, the mean absolute deviation of a set of predictions y is given by

$$MAD(y) = \frac{1}{n} \sum_{i=1}^{n} |y^{(i)} - \overline{y}| \tag{6.6}$$

where n is the number of predictions in y and $\overline{y}$ is the mean of the set of predictions. To all of these measurements a normalization is applied: we divide by the respective deviation measure (variance, standard deviation or mean absolute deviation) for the observed values. Therefore, we determine resolution with

$$res(y) = \frac{\overline{D(\tilde{y})}}{D(y)} \tag{6.7}$$

where D is some deviation measurement and $\overline{D(\tilde{y})}$ is the mean over all deviations of the predictions. *res* ranges from 0 to ∞; a lower value indicates better resolution. At first glance, it may not be clear how these values correlate: the deviation of observations in reality, and the mean deviation for network prediction sets of different time steps seem uncorrelated concepts. However, one may realize that how "narrow" we consider a prediction interval depends on the amount of fluctuation in the data . This is explained with a visual example. Figure 6.4 a) shows a prediction interval together with a rather low variance in the overall observations. Figure 6.4 b) depicts the same prediction interval, but with a higher variance in the observation. From a graphical point of view, it becomes clear that resolution is better in b) as this model learns a rather "narrow" prediction interval despite high variance in the observations.

In the work for this thesis, those measurements are applied during final evaluation where they are computed for each time step and averaged over the sequence of test data. Therefore, we obtain one measurement per catchment.

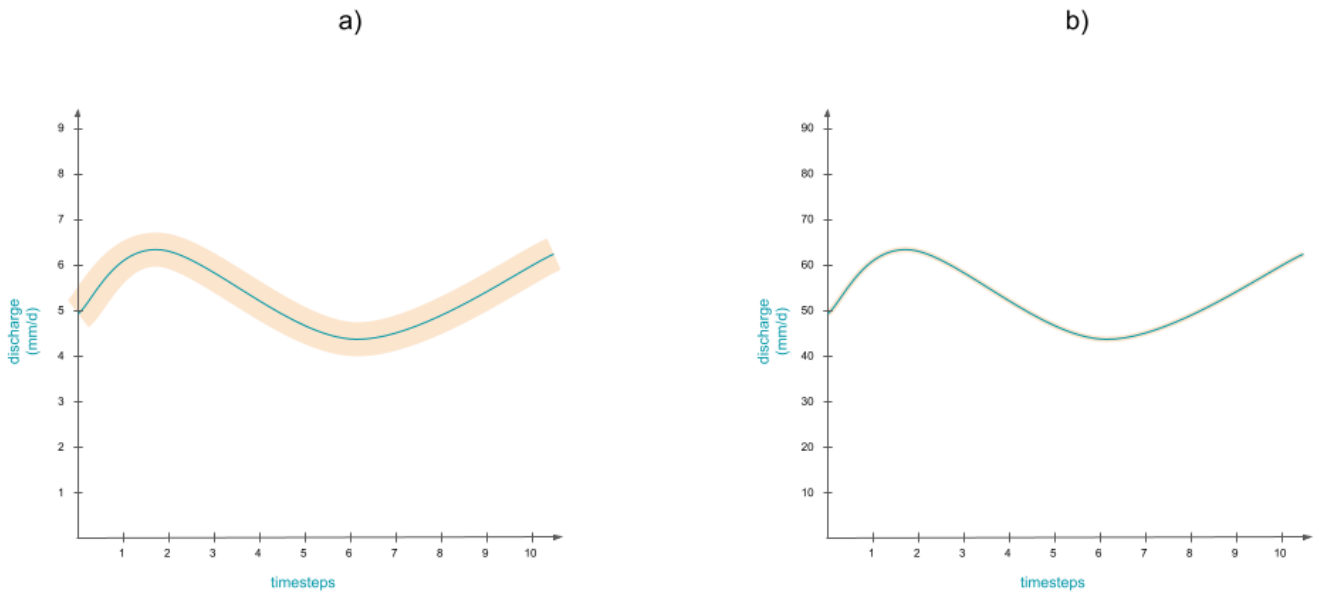

Figure 6.4: Two different resolution measurements with the same prediction interval. The turquoise function represents the ground truth observed over 10 time steps, the orange range is the prediction interval.

7 Results

In this chapter the result of the conducted experiments during hyperparameter tuning and final evaluation are presented.

7.1 Hyperparameter tuning

7.1.1 Investigation of different prior setting and weight initialization

As seen in section 5, the Bayes By Backprop-algorithm is not restricted to Gaussian priors and posteriors. We use a scale mixture prior and a diagonal Gaussian posterior as proposed by Blundell et al. (2015). Therefore, the prior contains the variances of the mixture components σ_1^2 and σ_2^2. Typically, $\sigma_1 > \sigma_2$ is chosen as this leads to a heavier tail in the density than with a plain Gaussian. π is another hyperparameter than can be set for the scale mixture prior (Blundell et al. 2015).

We performed a grid search over the values $\pi = \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$ and $\sigma_1 \in \{e^{-2}, e^{-1}, 1, 3, 10\}$; we therefore investigated all values proposed by Blundell (2015). σ_2 was kept fixed at 0.002 to keep the grid search at a computationally feasible level. This is one of the values proposed by Blundell et al. (2015). Regarding the initialization of the weights (and therefore the diagonal Gaussian posterior), all means were initialized with 0, for the initial value of ρ, the values $\rho \in \{-6, -4, -2.5, -1\}$ were investigated (to be precise: the actual initial values for ρ were drawn from a Gaussian with very small variance centered at the chosen value). The choice of values for ρ is justified by a suggestion to initialize the parameter to a small value (*Bayes by Backprop from scratch* 2017) and with additional empirical experiments that indicated that $\rho \geq 0$ is not beneficial for network learning in the given task. During this experiment, the other hyperparameters were set arbitrarily. We used a network architecture consisting of 2 hidden layers with 20 units each, a batch size of 256 and a learning rate of 0.004. To keep the computations feasible, we only considered input sequences of length 30 (approximately one month in our case). We trained all networks for 50 epochs as this is the default value suggested by Kratzert et al. (2018) for a non-bayesian LSTM network.

In order to choose the most beneficial setting of hyperparameters, we considered the NSE over the validation set after the last epoch, the likelihood cost of the validation loss and the quality of the prediction interval. In the course of this evaluation, the hy-

perparameter assignment of $\sigma_1 = 10, \pi = 1, \rho = -2.5$ showed the highest performance regarding the observed criteria as it was the only setting that achieved a validation NSE of above 0.8 (0.85), whilst achieving a prediction interval of over 60% (65%). Therefore, this setting was used in the further course of the experiments.

7.1.2 Investigation of different batch sizes, learning rates, LSTM-network-sizes and optimizer

In a next step, we investigated different batch sizes for training (thus numbers of performed backpropagation steps per epoch), learning rates and LSTM-network-sizes. In terms of the batch size, we investigated the values $\{128, 256, 512\}$. With regards to the learning rate, values in the magnitude of 10^{-4} or smaller were observed to not ensure fast-enough convergence in a preliminary experiment; therefore, we only investigated 10^{-3} and $4 \cdot 10^{-3}$. 10^{-3} is also the standard value suggested by Kratzert et al. (2018). In terms of the LSTM-network-size, we investigated architectures with 1 and 2 layers and 20, 50 or 256 values each. Using a 2-layer architecture with 20 hidden units per layer is suggested by Kratzert et al. (2018), while a 1-layer-architecture with 256 hidden units is suggested by Kratzert et al. (2019), both for non-bayesian LSTM networks for rainfall-runoff modeling.

We considered the same performance measures as in the previous experiment. In this step we found the setting with a batch size of 256, a learning rate of 0.004 and a single network layer with 50 hidden units most beneficial. It yielded a validation likelihood cost of 0.0021 (mean over all setting 0.0023), validation NSE of 0.86 (mean over all settings 0.80) and 88% of the ground truth values lying in the confidence interval (mean over all settings 59%). As an additional task, we investigates whether the optimizers *Adagrad* and *stochastic gradient descent* (standard version without modifications) are an alternative to *Adam*. However, we found both to perform significantly worse than *Adam*.

7.1.3 Addition of a yearly periodic signal

Next, we added a periodic signal that represents a year to the input and examined whether the modification improves network performance. In order to examine this, we trained the previously obtained architecture on the 27 catchments from HUC 01 with and without said modification. We examined average validation likelihood cost, validation NSE and the quality of prediction intervals. With regard to those measures, no significant difference could be observed among the two variants. Nevertheless, we decided to retain the periodic signal as part of the input in future experiments. It adds a reasonable source of learning with little computational overhead.

Model	s_p	Mean MSE	Mean NSE	Mean values in conf.
single-shot	8	4.33	0.34	56%
many-to-many	8	2.65	0.60	75%
single-shot	16	5.00	0.24	41%
many-to-many	16	3.13	0.52	69%
single-shot	32	5.38	0.18	37%
many-to-many	32	3.20	0.51	70%

Table 7.1: Results from the comparison of the single-shot- and many-to-many-architecture for predicting multiple time steps. All measurements refer to the performance on the test set.

7.1.4 Investigation of different numbers of samples

As a final step of hyperparameter tuning, we investigated different numbers of samples for Monte Carlo sampling of the Bayes-by-Backprop-algorithm. The investigated values were $\{1, 2, 5, 10\}$, which are the values suggested by Blundell et al. (2015). We found taking 1 or 2 samples to perform significantly worse than taking 5 or 10 samples; therefore, we considered 5 samples the best trade-off between prediction accuracy and computational feasibility.

7.1.5 Comparison between single-shot- and many-to-many-based models

We introduced two models for predicting multiple time steps ahead as already mentioned in section 3: the single shot model and the many-to-many model. We compared their performance by evaluating different prediction window sizes: 8, 16 and 32. Each of those settings was trained for all 27 catchments in HUC 01. With respect to the previouly obtained hyperparemters, the number of epochs was increased to 100 in order to provide a longer training phase, as fewer forward steps are performed with these models. Table 7.1 depicts the obtained measurements.

It can be observed that with an increasing size of the prediction window, performance deteriorates more significantly when the single shot model is used. This is expected behavior as said model does not receive any input measurements after the start of the prediction window, whereas the many-to-many-model is allowed to consider such measurements. Table 7.2 summarized the results of the conducted hyperparameter tuning.

Hyperparameter(s)	Final setting(s)
Prior setting: σ_1, σ_2 π	10, 0.002, 1
Posterior initialization: ρ	-2.5
Batch size	256
Learning rate	0.004
Number of network layers	1
Number of hidden units per layer	50
Optimizer	Adam
Yearly periodic signal (sin)	yes
Number of samples for MC sampling	5
s_p for many-to-many	8

Table 7.2: Resulting architecture from hyperparameter tuning.

7.2 Final evaluation for CAMELS-US

The obtained architecture from hyperparameter tuning was evaluated on the subset consisting of 241 catchments proposed by Kratzert et al. (2018). For the final evaluation, an individual model per catchment was trained for 50 epochs and evaluated on the test set every 10 epochs. We tested three variations of the final architecture: a plain many-to-one-model, a many-to-one-model that additionally takes discharge values from the warmup window as input, and the most beneficial model for predicting multiple time steps (according to hyperparameter tuning).

7.2.1 Many-to-one-model

Accuracy

In this experiment we trained one many-to-one-based model per catchment with the architecture obtained during hyperparameter tuning. Over all catchments, a mean test set NSE of 0.66 was observed at epoch 50. The model therefore outperformed the standard LSTM presented by Kratzert et al. (2018) with regard to the NSE, as the non-bayesian network reached a mean NSE of 0.63. In terms of the median test set NSE, we found a value of 0.69 over all catchments. This was also higher than the value 0.65 observed by Kratzert et al. (2018). With regards to the test set KGE, we observed a mean value of 0.70 (the median was 0.70 too).

We also observed that the models heavily underestimate low flow values; this is reflected in the value for the FLV (mean value: -311 excluding three unrepresentative

outlier likely caused by negative predicted discharge). The mean FMS and FHV values (-163 and -15 respectively) indicate that higher discharge values also tend to get underestimated, but less significantly than low flow values. Such a tendency was also observed by Kratzert et al. (2018) and was substantiated with the fact that the FLV is highly sensitive to the minimum predicted discharge value; as the predicted discharge can easily drop to diminutive numbers or zero, low values for the FLV are common. A way to tackle this problem is to set a lower bound for the predicted discharge value: the value should not undercut the minimum observed discharge value from the training set (Frederik Kratzert et al. 2018). Admittedly, this possibility was not investigated and is proposed as a further step to improve the architecture.

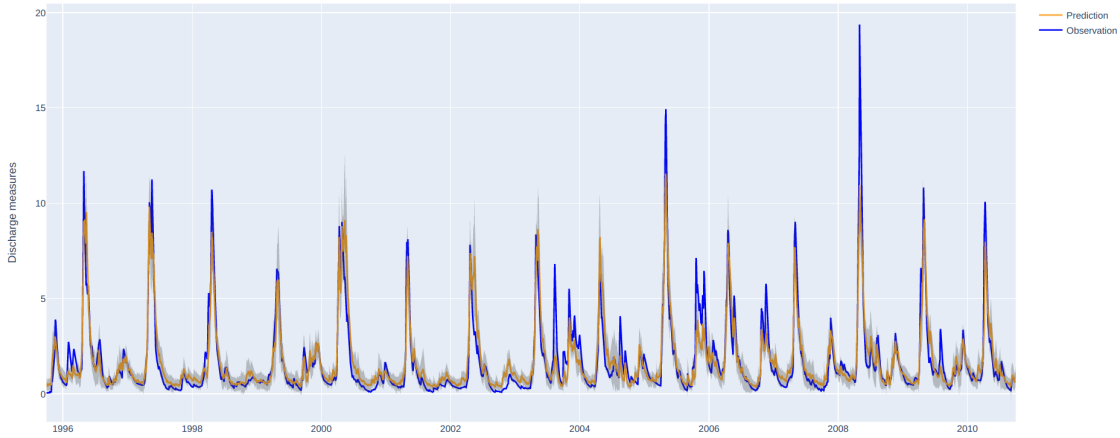

Figure 7.1: Hydrograph for the test set predictions (many-to-one without discharge as input) of a catchment in the New England region in the north-east (catchment id: 01013500). The region depicted in gray represents the prediction interval.

For the remaining measurements related to accuracy one may refer to figure 7.3.

Uncertainty estimation

In terms of reliability, the networks predicted a 75% prediction interval on average after 50 epochs of training. Figure 7.2 depicts the average prediction interval qualities over the training steps. It can be observed that on average, the networks effectively learn to become more consistent with the ground truth value as training proceeds. It can also be observed that very inconsistent prediction intervals (30% prediction intervals and below) only occurred at the beginning of training. However, it was also observed that very high prediction interval qualities (96%) are only reached at around epoch 13

and not at the end of training. This can be explained by a tendency of the networks to predict broad intervals at the beginning and sharpen them as training proceeds. Most likely, due to the network's inability to correctly predict outliers in the data, it is difficult to produce prediction intervals with an accuracy of 90% or above.

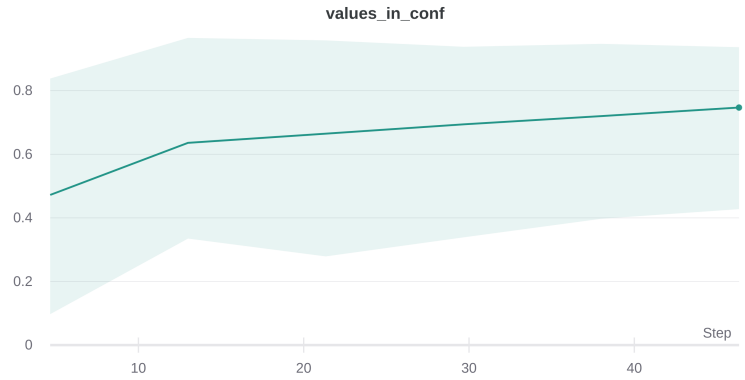

Figure 7.2: Average prediction interval qualities over the training steps. The green line represents the mean, the light green area represents the range between the minimum and maximum values observed over the training steps.

Another delightful result with regards to the prediction intervals is that they seem significantly broader in a high flow regions that are generally difficult for the network to exactly capture as they consist numerous outliers. This was observed on a number of exemplary hydrographs from different catchments.

In terms of resolution, a normalized mean absolute deviation of 0.20 over all catchments after training epoch 50 was observed. As this value is near 0 and less than 1, we verified that the network prediction do not only provide decent reliability, but also high resolution.

7.2.2 Many-to-many-model with discharge

With discharge as an additional input measurement, significantly better performance was observed in terms of the observed measurements. This behavior is expected as discharge usually has a high auto-correlation (Gauch et al. 2020). The mean test set NSE after 50 training epochs over all catchments was 0.81 (median 0.85), the mean test set KGE after training was 0.82 (median 0.85). Also with respect to the uncertainty estimation we observed improvements to the models without discharge as input. On average, the network produced a 90% prediction interval after 50 epochs of training; also here, an increase of the prediction interval reliability could be observed as training proceeded. We also observed that the measurements regarding resolution were superior

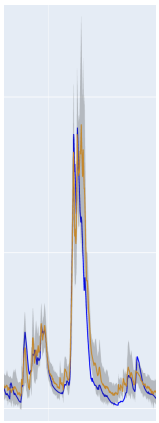

Figure 7.3: Example for a prediction interval in high flow region. It can be observed that the prediction interval is significantly broader at the high flow region than in surrounding regions with lower flow. Because of the broad prediction interval, most of the ground truth values in the high flow region fall into the predicted range, although they could not be captured exactly.

to the ones for the model without discharge as input. A normalized mean absolute deviation of 0.17 over all catchments was observed after training.

For hydrographs that show the predictions of these networks for two exemplary catchments the reader may refer to figure 7.4 and figure 10.1 in Appendix B.

7.2.3 Many-to-many-model

A model that is able to predict multiple time steps was also trained for each catchment. We chose the many-to-many-based model with a prediction window size of 8 as we considered it a "trade-off" between high accuracy and still being able to make multiple discharge predictions in one forward step. To enhance performance, discharge was consulted as an input measurement also in this case. In contrast to the previous experiments, the networks were trained for 100 epochs instead of 50 as in preliminary experiments many-to-many-based networks were found to improve their performance and to not overfit when such a large number of training epochs was used. In terms of accuracy, the resulting networks are between the two approaches investigated previously. The mean test set NSE after 100 training epochs was 0.69 (median 0.73), the mean test set KGE after training was 0.70 (median 0.73). The produced prediction intervals can also be said to be middle ground between the previous models. The many-to-many-models produced an 85% prediction interval on average after training. For hydrographs that show the predictions of these networks for two exemplary catchments

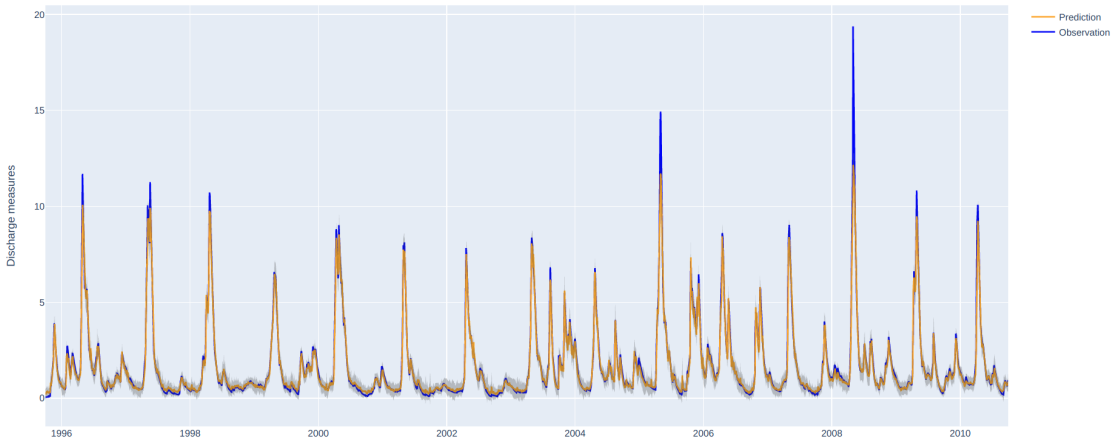

Figure 7.4: Hydrograph for the test set predictions (many-to-one with discharge as input) of a catchment in the New England region in the north-east (catchment id: 01013500). The region depicted in gray represents the prediction interval.

refer to figure 10.2 and figure 10.3 in Appendix B.

7.3 Experiments on the Regen catchment

As mentioned in chapter 6, our experiments for the Regen catchments primarily aim to provide a comparison to the results from Fiedler (2020), such that a comparison between a non-bayesian LSTM network and a bayesian LSTM network with similar conditions is given. In the course of the experiments, we therefore investigated the accuracy measurements that are also used by Fielder (2020): MSE, RMSE and NSE. In addition, we present our results concerning uncertainty estimation (reliability and resolution). The results are shown in tables 7.5 and 7.6.

As equally done by Fiedler (2020) we trained our networks for 25 epochs. However, we observed that training only for 5 epochs yielded better performance on average for hourly resulution: therefore, the measurements for hourly resolution are the ones taken after the 5th training epoch. The results show that the bayesian network outperforms the non-bayesian one for most feature sets in terms of accuracy (MSE, RMSE and NSE). Sometimes, the bayesian network achieves significantly higher values than the non-bayesian pendant. With daily resolution, our network outperformed the non-bayesian one for all feature set; with hourly resolution, feature sets 3 and 4 showed slightly better performance with the non-bayesian network. At this point we again point out that the two network differ in terms of hyperparameters. It was not investigated

Measurement	M1	M2	M3
Test loss	8.12	16.36	136.46
Test likelihood cost	0.01	0.01	0.04
Test complexity cost	8.10	16.35	136.42
Test NSE	0.66	0.81	0.69
Test α-NSE	0.82	0.88	0.80
Test β-NSE	-0.0002	0.003	-0.005
Test KGE	0.69	0.82	0.70
Test MSE	5.11	3.31	5.24
Test RMSE	1.84	1.39	1.82
Test FHV	-15.15	-9.76	-19.81
Test FLV	-379.14	-390.65	-418.16
Test FMS	-8.10	-12.23	5.66

Table 7.3: Measurements related to likelihood performance for all three experiments (**M1**: many-to-one, **M2**: many-to-one with discharge, **M3**: many-to-many with discharge).

Measurement	M1	M2	M3
Prediction interval	75%	91%	80%
Standard deviation (pred)	0.48	0.42	0.52
Standard deviation (obs)	3.58	3.58	3.58
Standard deviation (normalized)	0.13	0.12	0.15
Variance (pred)	0.56	0.40	0.57
Variance (obs)	20.79	20.79	20.79
Variance (normalized)	0.027	0.019	0.027
Mean absolute deviation (pred)	0.38	0.33	0.41
Mean absolute deviation (obs)	1.94	1.94	1.94
Mean absolute deviation (normalized)	0.20	0.17	0.21

Table 7.4: Measurements related to uncertainty estimation (**M1**: many-to-one, **M2**: many-to-one with discharge, **M3**: many-to-many with discharge).

further if the accuracy difference is due to Bayes by Backprop or due to the different hyperparameters. Nevertheless, it can be concluded that Bayes by Backprop and the specific architecture we found during previous hyperparameter tuning are suitable for simulation tasks on the Regen catchment and lead to decent accuracy.

We also took measurements related to uncertainty estimation. As it was the case for the experiments on the CAMELS-US-dataset, we lack comparative values in this case. The most reliable prediction interval for daily resolution is a 87% prediction interval for features set 4; the most reliable one for hourly resolution is a 99% prediction interval for feature set 2. The normalized mean average deviations are in the interval $[0.09, 0, 33]$ for all experiments (mean 0.19). Based on these measurements, we conclude that also on the Regen catchment Bayes by Backprop is able to learn sensible uncertainty estimates.

Overall, feature set 2 with hourly resolution was the best performing experiment. It performed best with respect to all taken measurement. The hydrograph with predictions from that network is shown in figure 7.6. Restricting ourselves to daily resolution, feature set 4 yielded the best performance. The hydrograph with corresponding predictions is shown in figure 7.5 in Appendix B.

Daily resolution

	All input features		Feature set 1		Feature set 2	
	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM
MSE	111.445	68.587	91.759	35.095	98.426	35.196
RMSE	10.557	8.282	9.579	5.924	9.921	5.933
NSE	0.741	0.840	0.787	0.919	0.771	0.918
Prediction Interval	-	65%	-	78%	-	81%
Normalized MAD	-	0.22	-	0.19	-	**0.16**
	Feature set 3		**Feature set 4**		**Feature set 5**	
	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM
MSE	60.401	51.739	**58.183**	**29.266**	176.354	113.474
RMSE	7.772	7.193	**7.628**	**5.41**	13.280	10.652
NSE	0.860	0.880	**0.865**	**0.932**	0.590	0.736
Prediction Interval	-	80%	-	**87%**	-	67%
Normalized MAD	-	0.22	-	0.21	-	0.26

Figure 7.5: Results for daily resolution. The non-bayesian LSTM network refers to the implementation of Fiedler (2020), the bayesian LSTM refers to our implementation with Bayes by Backprop. The cells depicted in red are the optimal values observed per measurement across all feature sets. All measurements relate to the test set.

Hourly resolution

	All input features		Feature set 1		Feature set 2	
	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM
MSE	9.915	3.504	**5.000**	2.227	7.957	**2.044**
RMSE	3.149	1.872	**2.236**	1.492	2.821	**1.430**
NSE	0.976	0.991	**0.988**	0.995	0.981	**0.995**
Prediction Interval	-	90%	-	97%	-	**99%**
Normalized MAD	-	0.10	-	0.11	-	**0.09**

	Feature set 3		Feature set 4		Feature set 5	
	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM	non-bayesian LSTM	bayesian LSTM
MSE	60.272	67.146	33.361	42.932	294.303	153.138
RMSE	7.764	8.194	5.776	6.552	17.155	12.375
NSE	0.856	0.840	0.920	0.898	0.297	0.624
Prediction Interval	-	69%	-	92%	-	57%
Normalized MAD	-	0.18	-	0.20	-	0.33

Figure 7.6: Results for hourly resolution. The non-bayesian LSTM network refers to the implementation of Fiedler (2020), the bayesian LSTM refers to our implementation with Bayes by Backprop. The cells depicted in red are the optimal values observed per measurement across all feature sets. All measurements relate to the test set.

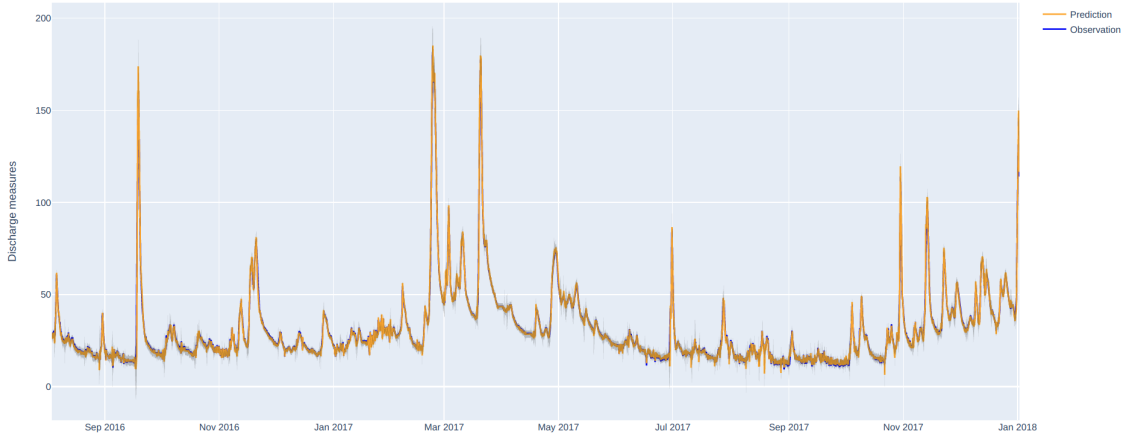

Figure 7.7: Hydrograph for Regen catchment - Feature set 2 (hourly resolution). The region depicted in gray represents the prediction interval.

8 Conclusion and Outlook

Bayesian neural networks and especially the algorithm *Bayes by Backprop (through time)* are suitable for the task of rainfall-runoff modeling. They reach comparable accuracy to non-bayesian pendants and sometimes even outperform them. Moreover, we showed that bayesian neural networks produce sensible uncertainty estimates for discharge values in terms of reliability and resolution. As in the hydrologic domain one is typically interested in uncertainty estimates rather than in single values, we consider bayesian neural networks an important contribution to the field of data-driven models for hydrology.

The focus of this work was on developing a general architecture for the time series forecasting task, primarily an LSTM-network architecture that produces discharge predictions for one time step in the future was investigated. We also tested alternatives that are able to produce multiple predictions. However, it remains to point out that there exist a number of other architectures that could be developed for that specific task. For instance, it is possible to set up LSTM-networks based on autoregressive models. These models consider predictions made in earlier time steps as network inputs for later time steps. Recall that in the encoder-decoder-approach introduced in chapter 4, the decoder was not allowed to consider discharge. Following an autoregressive approach, although the decoder would still not consider observed discharge, it would be allowed to consider forecasted discharge values from previous time steps. Such a behavior is particularly interesting for discharge prediction as it allows to mimic traditional process-based models (Ludwig et al. 2006). Therefore, bayesian autoregressive models are proposed as an attractive research area for future work.

Moreover, it is important to point out that *Bayes by Backprop* is not the only approach to realize bayesian neural networks. A straightforward approach named *Monte Carlo Dropout* has proven to be a valid alternative (Gal et al. 2016). Due to its simplicity and possibly higher computational performance, we consider it a promising alternative for rainfall-runoff modeling.

In research, LSTM networks were improved with a mechanism named *attention* (Bahdanau et al. 2015). It is a technique specially suitable for longer input sequences, each item in the output sequence is conditional on items from the input sequence. In other words, each time an output vector is generated, the network chooses the subset of hidden states that it considers most relevant and produced the output based on those

states (Brownlee 2019, Bahdanau et al. 2015). A neural network architecture present in recent research that further builds on the attention-mechanism is the *transformer*. It is solely based on attention and does not include recurrence (Vaswani et al. 2017). This architecture typically outperforms recurrent neural networks in machine translation tasks and also showed high performance in time series forecasting tasks (Wu et al. 2020). There also exists work that adapted the approach for bayesian inference (Xue et al. 2021). Therefore, we consider *transformer*-based bayesian networks for rainfall-runoff modeling a promising field for future research.

Another point that was beyond the scope of this thesis concerns static attributes of catchments. In rainfall-runoff modeling tasks, it is common to incorporate geophysical information about catchments next to the forcings into the model input. Kratzert et al. (2019) showed that this can also be applied to data-driven models like LSTM networks and proposed an LSTM-network-architecture that achieves superior accuracy than architectures that do not consult static attributes. Incorporating these attributes into a bayesian setting is yet another aspect we leave for future work.

9 Appendix A - LSTM networks for hydrology

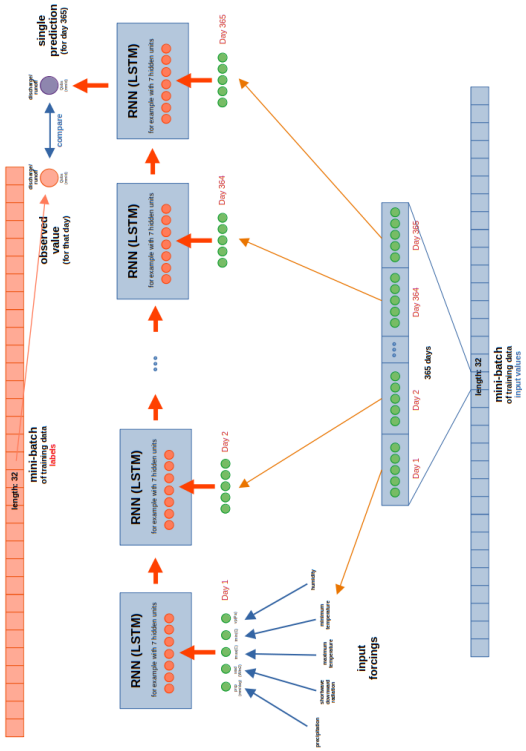

Figure 9.1: Many-to-one-model for discharge prediction. In this representation, a single forward step of the network is predicted. In practice, the network is trained using mini-batches that consist of several windows. The last cell in the unfolded representation is used to obtain a single prediction for each window.

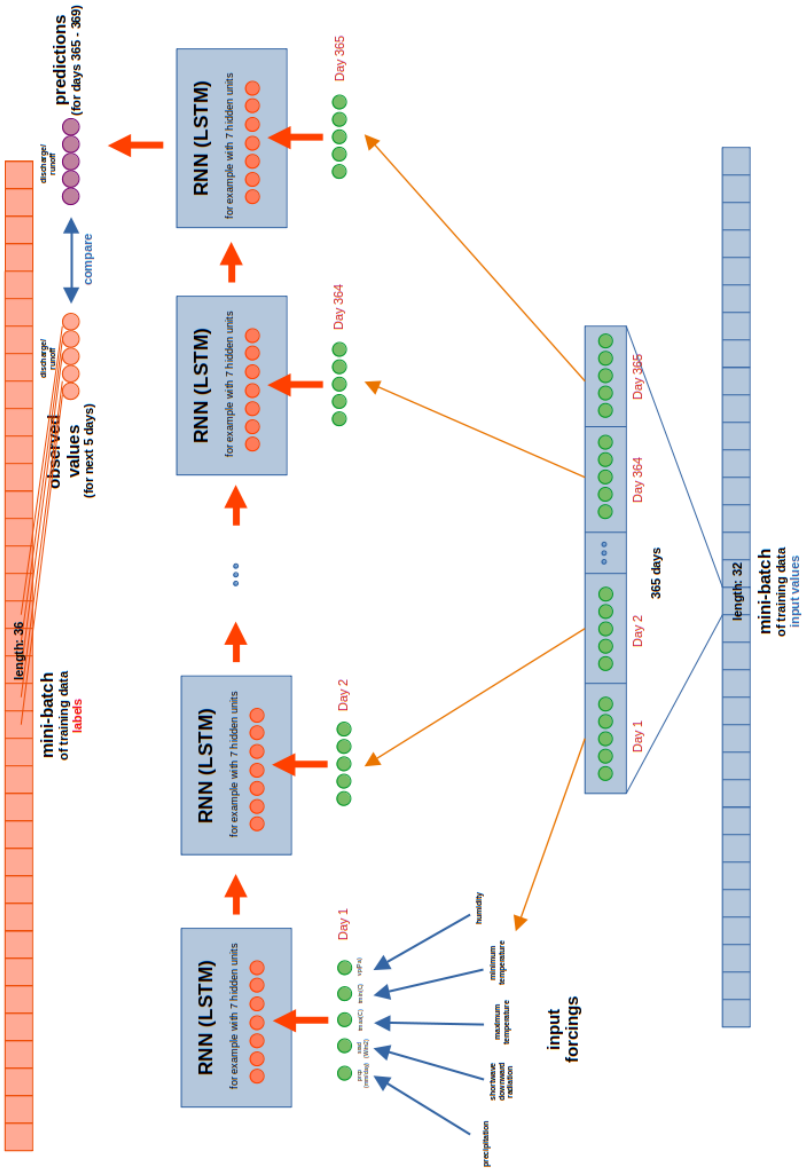

Figure 9.2: Single-shot-model for discharge prediction. In this representation, the prediction phase length is 5, thus 5 values are predicted into the future.

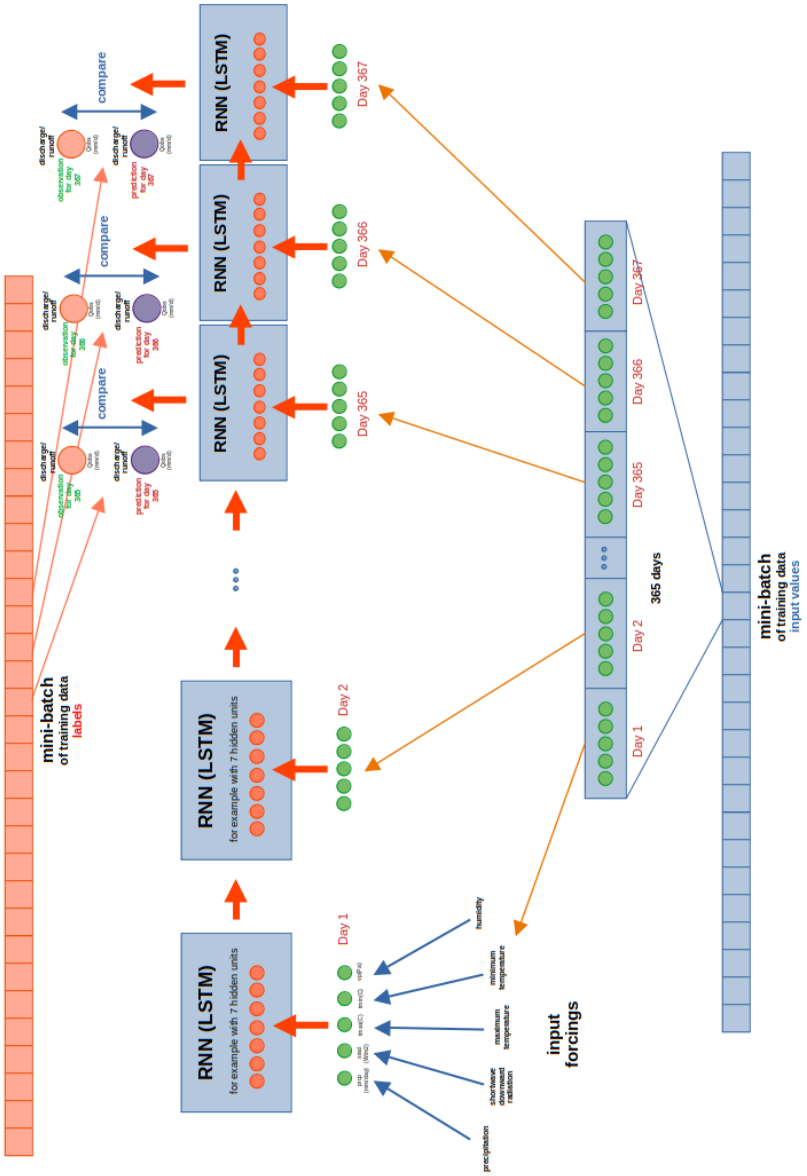

Figure 9.3: Many-to-many-model for discharge prediction. In this representation, the prediction phase length is 3, thus 3 values are predicted into the future.

10 Appendix B - Hydrographs

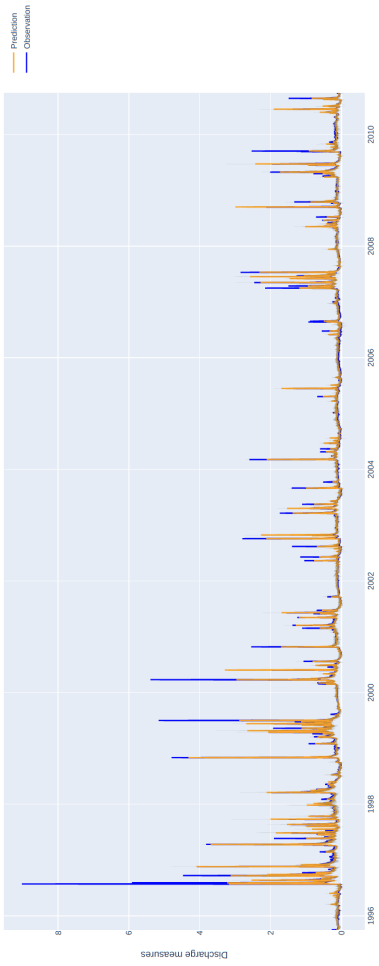

Figure 10.1: Hydrograph for the test set predictions (many-to-one with discharge as input) of a catchment in the Arkansas-White-Red region in the north-east (catchment id: 07149000). The region depicted in gray represents the prediction interval.

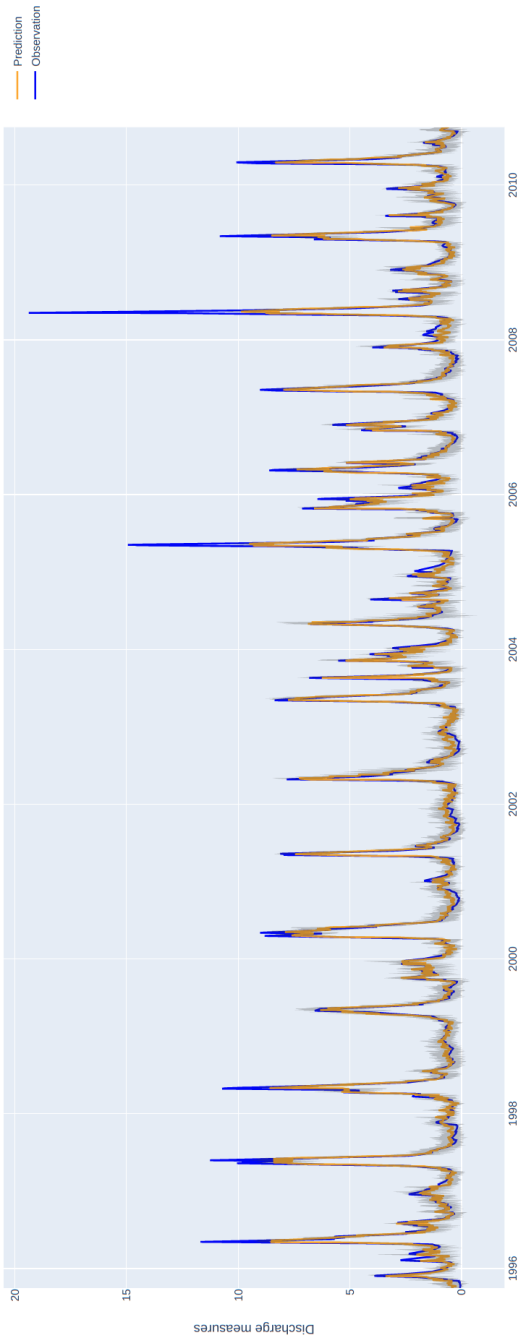

Figure 10.2: Hydrograph for the test set predictions (many-to-many) of a catchment in the New England region in the north-east (catchment id: 01013500). The region depicted in gray represents the prediction interval.

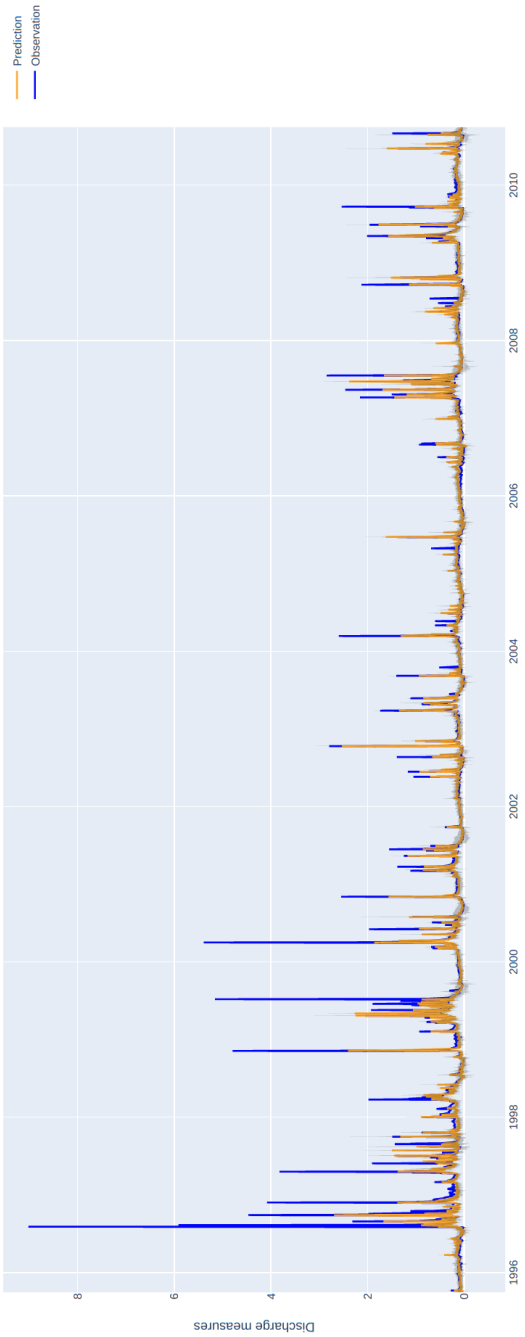

Figure 10.3: Hydrograph for the test set predictions (many-to-many) of a catchment in the Arkansas-White-Red region in the north-east (catchment id: 07149000). The region depicted in gray represents the prediction interval.

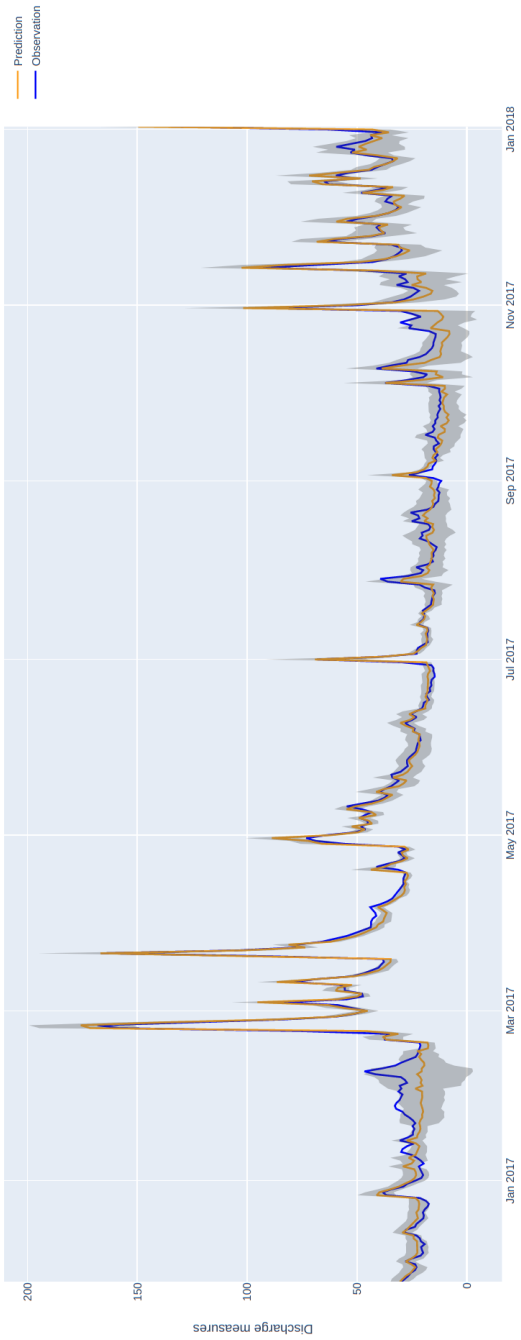

Figure 10.4: Hydrograph for Regen catchment - Feature set 4 (daily resolution). The region depicted in gray represents the prediction interval.

11 Appendix C - Histograms with probability distributions

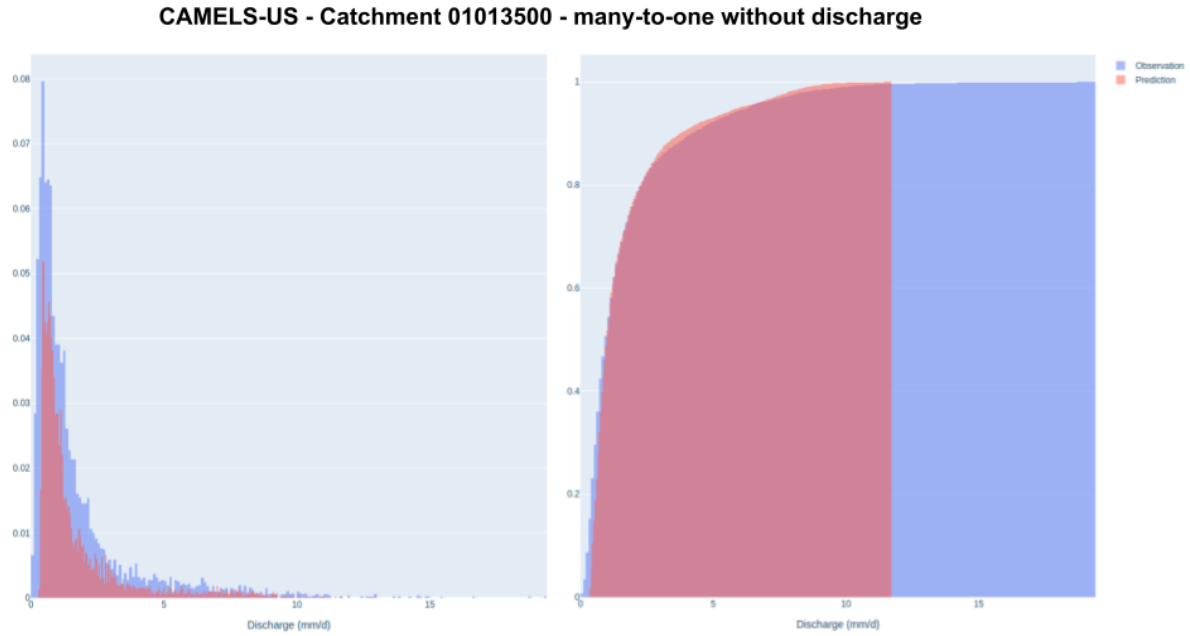

Figure 11.1: Histograms that depict the probability distributions of predictions and observation. The left figure depicts a standard histogram where the y-axis is normalized to match a probability distribution. The right figure depicts a cumulative density function.

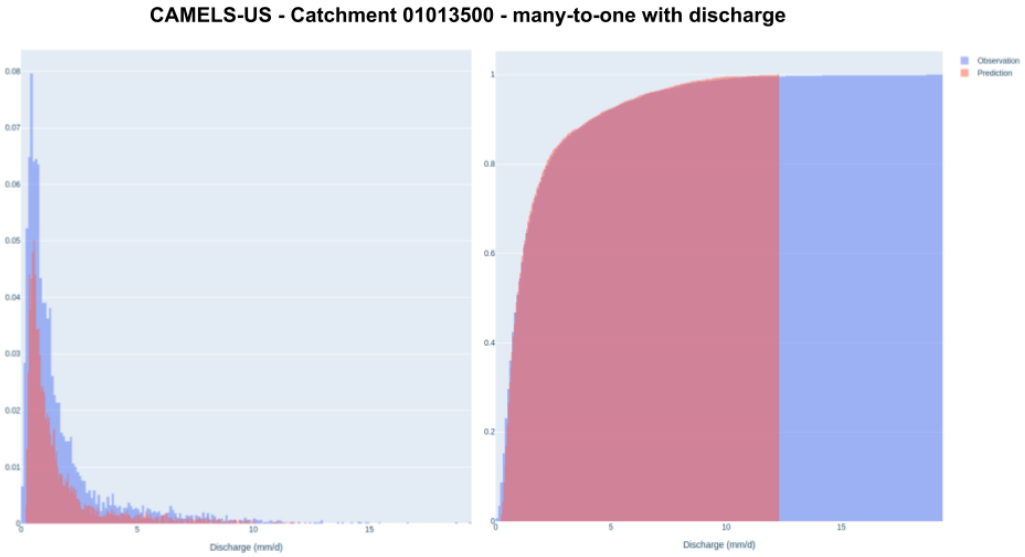

Figure 11.2: Histograms that depict the probability distributions of predictions and observation.

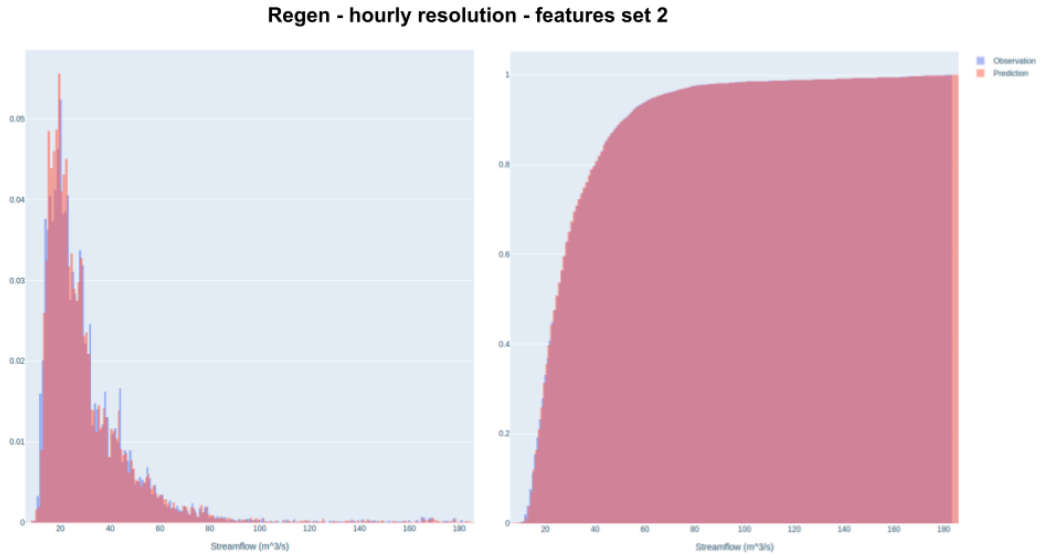

Figure 11.3: Histograms that depict the probability distributions of predictions and observation.

Figure 11.4: Histograms that depict the probability distributions of predictions and observation.

List of Figures

List of Tables

Bibliography

Addor, N. et al. (2017). *The CAMELS data set: catchment attributes and meteorology for large-sample studies*. Boulder, CO. DOI: 10.5065/D6G73C3Q.

Bahdanau, Dzmitry et al. (2015). "Neural machine translation by jointly learning to align and translate." In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–15. arXiv: 1409.0473.

Bayes by Backprop from scratch (2017). URL: https://gluon.mxnet.io/chapter18_variational-methods-and-uncertainty/bayes-by-backprop.html.

Blundell, Charles et al. (2015). "Weight uncertainty in neural networks." In: *32nd International Conference on Machine Learning, ICML 2015* 2, pp. 1613–1622. arXiv: 1505.05424.

Brownlee, Jason (2017). *What is the Difference Between Test and Validation Datasets?* URL: https://machinelearningmastery.com/difference-test-validation-datasets/.

– (2018). *Deep Learning for Time Series Forecasting. - Predict the Fu- ture with MLPs, CNNs and LSTMs in Python*. v1.4.

– (2019). *Attention in Long Short-Term Memory Recurrent Neural Networks - Machine Learning Mastery*. URL: https://machinelearningmastery.com/how-does-attention-work-in-encoder-decoder-recurrent-neural-networks.

Bungartz, H.-J. et al. (2014). *Modeling and Simulation - An Application-Oriented Introduction*.

Chollet, François (2017). *Deep Learning with Python*, p. 384.

Choromanska, Anna et al. (2015). "The loss surfaces of multilayer networks." In: *Journal of Machine Learning Research* 38, pp. 192–204. ISSN: 15337928. arXiv: 1412.0233.

Dauphin, Yann N. et al. (2014). "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization." In: *Advances in Neural Information Processing Systems* 4.January, pp. 2933–2941. ISSN: 10495258. arXiv: 1406.2572.

Dürr, Oliver et al. (2020). *Probabilistic Deep Learning With Python, Keras and TensorFlow Probability*. ISBN: 9781617296079.

Esposito, Piero (2020). *BLiTZ - Bayesian Layers in Torch Zoo (a Bayesian Deep Learing library for Torch)*. https://github.com/piEsposito/blitz-bayesian-deep-learning/.

Fiedler, Leon (2020). "Sensitivity analysis of a deep learning model for discharge prediction in the Regen catchment." Master's Thesis. Technische Universität München.

Fortunato, Meire et al. (2017). "Bayesian recurrent neural networks." In: *arXiv*, pp. 1–14. ISSN: 23318422. arXiv: 1704.02798.

Gal, Yarin et al. (2016). "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning." In: *33rd International Conference on Machine Learning, ICML 2016* 3, pp. 1651–1660. arXiv: 1506.02142.

Gauch, Martin et al. (2020). "A Data Scientist's Guide to Streamflow Prediction." In: *arXiv*. ISSN: 23318422. arXiv: 2006.12975.

Gauch, Martin et al. (2021). "Rainfall-runoff prediction at multiple timescales with a single Long Short-Term Memory network." In: *Hydrology and Earth System Sciences* 25.4, pp. 2045–2062. ISSN: 16077938. DOI: 10.5194/hess-25-2045-2021. arXiv: 2010.07921. URL: https://hess.copernicus.org/articles/25/2045/2021/.

Glen, Stephanie (n.d.). *Probabilistic: Definition, Models and Theory Explained*. URL: https://www.statisticshowto.com/probabilistic/.

Goodfellow, Ian J. et al. (2015). "Qualitatively characterizing neural network optimization problems." In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. arXiv: 1412.6544.

Goodfellow, Ian et al. (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Graves, Alex (2011). "Practical Variational Inference for Neural Networks." In: pp. 1–5.

Guo, Jiang (2013). "BackPropagation Through Time." In: *Manuscript* 1, pp. 1–6.

Gupta, Hoshin V. et al. (2009). "Decomposition of the mean squared error and NSE performance criteria: Implications for improving hydrological modelling." In: *Journal of Hydrology* 377.1-2, pp. 80–91. ISSN: 00221694. DOI: 10.1016/j.jhydrol.2009.08.003.

Hinton, Geoffrey E. et al. (1993). "Keeping neural networks simple by minimizing the description length of the weights." In: pp. 5–13. DOI: 10.1145/168304.168306.

Hochreiter, Sepp et al. (1997). "Long Short-Term Memory." In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 08997667. DOI: 10.1162/neco.1997.9.8.1735.

Jospin, Laurent Valentin et al. (2020). "Hands-on Bayesian Neural Networks – a Tutorial for Deep Learning Users." In: 1.1, pp. 1–35. arXiv: 2007.06823. URL: http://arxiv.org/abs/2007.06823.

Kingma, Diederik P. et al. (2014). "Auto-encoding variational bayes." In: *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings* Ml, pp. 1–14. arXiv: 1312.6114.

Kingma, Diederik P. et al. (2015). "Adam: A method for stochastic optimization." In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–15. arXiv: 1412.6980.

Kirchgessner, Karsten (2009). "Frequentistische und Bayes'sche Statistik." In: pp. 1–6. URL: http://ekpwww.physik.uni-karlsruhe.de/$%5Csim$tkuhr/HauptseminarWS0910/Kirchgessner_handout.pdf.

Klotz, Daniel et al. (2020). "Uncertainty Estimation with Deep Learning for Rainfall-Runoff Modelling." In: Ml, pp. 1–32. arXiv: 2012.14295. URL: http://arxiv.org/abs/2012.14295.

Knoben, Wouter J.M. et al. (2019). "Technical note: Inherent benchmark or not? Comparing Nash-Sutcliffe and Kling-Gupta efficiency scores." In: *Hydrology and Earth System Sciences* 23.10, pp. 4323–4331. ISSN: 16077938. DOI: 10.5194/hess-23-4323-2019.

Kratzert, Frederik et al. (2018). "Rainfall-Runoff modelling using Long-Short-Term-Memory (LSTM) networks." In: May, pp. 1–26.

Kratzert, Frederik et al. (2019). "Towards learning universal, regional, and local hydrological behaviors via machine learning applied to large-sample datasets." In: *Hydrology and Earth System Sciences* 23.12, pp. 5089–5110. ISSN: 16077938. DOI: 10.5194/hess-23-5089-2019. arXiv: 1907.08456.

Kratzert, F et al. (2021). "A note on leveraging synergy in multiple meteorological data sets with deep learning for rainfall–runoff modeling." In: *Hydrology and Earth System Sciences* 25.5, pp. 2685–2703. DOI: 10.5194/hess-25-2685-2021. URL: https://hess.copernicus.org/articles/25/2685/2021/.

Livneh, Ben et al. (2013). "A long-term hydrologically based dataset of land surface fluxes and states for the conterminous United States: Update and extensions." In: *Journal of Climate* 26.23, pp. 9384–9392. ISSN: 08948755. DOI: 10.1175/JCLI-D-12-00508.1.

Ludwig, K. et al. (2006). "The Water Balance Model LARSIM: Design, Content and Applications." In:

Newman, A. et al. (2014). *A large-sample watershed-scale hydrometeorological dataset for the contiguous USA.* Boulder, CO. DOI: 10.5065/D6MW2F4D.

Olah, Christopher (2015). *Understanding LSTM's*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Rogelis, María Carolina et al. (2016). "Hydrological model assessment for flood early warning in a tropical high mountain basin." In: *Hydrology and Earth System Sciences Discussions* March, pp. 1–36. ISSN: 1027-5606. DOI: 10.5194/hess-2016-30.

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms." In: pp. 1–14. arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

Saxe, Andrew M. et al. (2014). "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." In: *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*, pp. 1–22. arXiv: 1312.6120.

Seaber, Paul R. et al. (1987). "Hydrologic unit maps." In: *USGS Publications Warehouse*. DOI: 10.3133/wsp2294. URL: http://pubs.er.usgs.gov/publication/wsp2294.

Singh, Vijay (2015). "Flow Duration Curve." In: *Introduction to Tsallis Entropy Theory in Water Engineering*, pp. 303–326. DOI: 10.1201/b19113-16.

Tagasovska, Natasa et al. (2019). "Single-model uncertainties for deep learning." In: *Advances in Neural Information Processing Systems* 32.NeurIPS, pp. 1–12. ISSN: 10495258. arXiv: 1811.00908.

Thornton, P.E. et al. (2016). *Daymet: Daily Surface Weather Data on a 1-km Grid for North America*. Oak Ridge, Tennessee, USA. DOI: https://doi.org/10.3334/ORNLDAAC/1328.

Time series forecasting (n.d.). URL: https://www.tensorflow.org/tutorials/structured_data/time_series.

Vaswani, Ashish et al. (2017). "Attention is all you need." In: *Advances in Neural Information Processing Systems* 2017-December.Nips, pp. 5999–6009. ISSN: 10495258. arXiv: 1706.03762.

Wu, Neo et al. (2020). "Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case." In: arXiv: 2001.08317. URL: http://arxiv.org/abs/2001.08317.

Xia, Youlong et al. (2012). "Continental-scale water and energy flux analysis and validation for the North American Land Data Assimilation System project phase 2 (NLDAS-2): 1. Intercomparison and application of model products." In: *Journal of Geophysical Research: Atmospheres* 117.D3. DOI: https://doi.org/10.1029/2011JD016048. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2011JD016048.

Xue, Boyang et al. (2021). "Bayesian Transformer Language Models for Speech Recognition." In: pp. 7378–7382. DOI: 10.1109/icassp39728.2021.9414046. arXiv: 2102.04754.

Yilmaz, Koray K et al. (2008). "A process-based diagnostic approach to model evaluation: Application to the NWS distributed hydrologic model." In: *Water Resources Research* 44.9. DOI: https://doi.org/10.1029/2007WR006716. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2007WR006716.