Deep Learning for Spatio-Temporal Forecasting with Gridded Remote Sensing Data

Mário João Cabral Cardoso

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Doutor Bruno Emanuel da Graça Martins
Doutor Jacinto Paulo Simões Estima

Examination Committee

October 2020

Acknowledgements

First, I would like to thank Professor Bruno Emanuel da Graça Martins for his guidance during this last year, through which he contributed very significantly to the work that we have developed together, with his knowledge and motivation.

I also express my gratitude to INESC-ID for funding my research project, through a grant aimed at supporting M.Sc. students, and for providing me and my research colleagues equipment to assist in our empirical evaluations.

I would also like to thank my family, in particular my mother and my siblings, for their constant support and for giving me the opportunity to learn in such a distinguished institute as Instituto Superior Tećnico.

Finally, I have to thank all my friends and colleagues for the constant support during the hard, although also rewarding, time spent at Instituto Superior Tećnico.

Mário João Cabral Cardoso

For my mother and siblings,

Resumo

Atualmente, uma ampla coleção de satélites está a ser usada para observar o planeta Terra, monitorizando os seus sistemas naturais e estruturas humanas ao medir um conjunto de variáveis em diferentes intervalos de tempo consistentes e resoluções espaciais. Dado este grande influxo de dados, a necessidade de métodos para o processamento automático dos mesmos tem vindo a aumentar, originando um aumento de popularidade dos métodos de aprendizagem automática. Este tipo de dados contêm uma mistura de dependências espaciais e temporais, indicando como diferentes localizações exibem padrões relacionados e como as variáveis observadas evoluem ao longo do tempo. Além disso, devido à natureza estocástica das variáveis subjacentes, modelos de previsão devem ser capazes de capturar padrões complexos e não lineares. Esta dissertação foca-se na previsão de dados espaço-temporais, apresentando uma arquitetura derivada de um modelo anterior, estendendo-o com um componente recorrente juntamente com a ideia de descodificação feed-backward, permitindo a reutilização dos pesos das camadas convolucionais no codificador, ao gerar as previsões a partir de representações intermédias. Além disso, um conjunto de técnicas diferentes são testadas juntamente com a arquitetura proposta, nomeadamente (a) novas operações de normalização-ativação, (b) um método de aumento de dados para reduzir o overfitting, e (c) o uso de inputs e pesos locais aprendidos em paralelo, permitindo que diferentes localizações sejam guiadas pelas suas próprias características locais intrínsecas. Os modelos foram avaliados através de um conjunto de experiências, usando dois conjuntos de dados provenientes de estudos anteriores, em duas tarefas distintas: previsão de medições futuras e reconstrução de dados em falta.

Abstract

Nowadays, a widespread collection of satellites are being used to monitor the Earth's natural systems and man-made structures, measuring a variety of variables at consistent time intervals and spatial resolutions. Given this massive influx of remote sensing data, the necessity of adequate methods for automatic satellite data processing has risen, leading to an increase in popularity for machine learning methods in a variety of practical applications. This type of data often contain a mixture of spatial and temporal dependencies, indicating how different spatial locations exhibit related patterns and how the observed variables evolve over time. Furthermore, due to the stochastic nature of the underlying variables, prediction models must be capable of capturing complex and non-linear patterns. This dissertation focuses on spatio-temporal data forecasting, presenting a novel architecture derived from a previous model, extending it with a recurrent component backbone together with the idea of feed-backward decoding, specifically by re-using the weights of the convolution layers in the encoder, when generating the predictions from intermediate representations. Furthermore, a variety of different techniques are jointly used with the aforementioned architecture, namely (a) novel normalization-activation operations, (b) a data augmentation method to reduce overfitting and improve intermediate representations, and (c) the use of learnable local inputs and weights to allow different locations to be guided by their own intrinsic local features. The models were evaluated through several experiments using two datasets from previous studies, on two different tasks pertaining to the forecasting of future measurements and the reconstruction of missing time-steps.

Palavras Chave
Keywords

Palavras Chave

Previsão com dados espaço-temporais

Redes neuronais profundas

Aprendizagem supervisionada por regressão

Deteção remota

Visão computacional

Keywords

Spatio-Temporal data forecasting

Deep neural networks

Supervised learning by regression

Remote sensing

Computer vision

Contents

List of Figures

List of Tables

1 Introduction

A widespread collection of satellites are nowadays being used to observe the Earth, monitoring natural systems and man-made structures by measuring a variety of variables at consistent time intervals and spatial resolutions. Given this influx of remote sensing data, machine learning approaches are becoming commonplace for various practical applications. This dissertation focuses on forecasting tasks, consisting on the prediction of particular time-steps in a series of gridded remote sensing data (e.g., forecasting future values conditioned by past observations, or reconstructing missing time-steps). Common examples of applications for these methods include forecasting meteorological variables, such as precipitation, air temperature, wind speed, among many others, or reconstructing missing data, e.g. due to cloud coverage, for environmental indices such as NDVI. This type of data often contain a mixture of spatial and temporal dependencies, indicating how different spatial locations exhibit related patterns (in particular, spatial neighbours tend to be related) and how the observed variables evolve over time. The major contribution of this work is the proposal of a novel deep learning architecture for forecasting spatio-temporal data, exploiting a variety of recent ideas from the literature. The architecture and extensions were evaluated through experiments on datasets describing meteorological variables alongside state-of-the-art and well known deep learning baselines.

1.1 Motivation

Due to the stochastic nature of the underlying captured variables, prediction models must be capable of capturing complex and non-linear patterns, while simultaneously leveraging the aforementioned spatial and temporal dependencies in order to obtain accurate results. For these reasons, neural network models for predicting gridded values of remote sensing observations are currently gaining increased popularity. These include approaches such as the widely popular ConvLSTM (Shi et al., 2015), capable of capturing both spatial and temporal contexts by combining ideas from Recurrent Neural Network (RNN) architectures with the convolution operation typically seen in Convolutional Neural Networks (CNNs) for image processing. Given these properties, ConvLSTM units have become a basic building block for a variety of neural architectures proposed in recent studies dealing with forecasting from gridded spatio-temporal data (Wang et al., 2017; Jang et al., 2018; Zhao et al., 2018).

More recently, Nascimento et al. (Nascimento et al., 2019) studied effective and computationally efficient alternatives to capture both spatial and temporal patterns using exclusively convolutional structures, proposing an encoder-decoder model named Spatio-Temporal Convolutional Sequence to Sequence Network (STConvS2S), comprised of sequential convolutional blocks with factorized filters. Each convolutional 3D filter is factorized into separate 2D and 1D filters, with the encoder using the former to model the spatial context, and the decoder using the latter to model the temporal context.

1.2 Thesis Proposal

This dissertation explores alternative solutions for the simultaneous capture of spatial and temporal dependencies in the data in order to improve results. More specifically, the central architecture was derived as an extension of the STConvS2S architecture (Nascimento et al., 2019), incorporating instead a recurrent component to process temporal dependencies, which is a natural fit for sequential data, together with a technique that allows weights to be shared between the encoder and decoder. Several extensions are afterwards proposed and evaluated, leveraging recent techniques in the literature, in order to better capture spatial and temporal dependencies and further improve achieved results.

1.3 Contributions

In brief, the main contributions of this thesis are as follows:

- Replace the decoding component from STConvS2S, based on dilated temporal convolutions in order to capture temporal dependencies, with a recurrent component that leverages feed-backward decoding (Wang et al., 2019), an idea recently proposed in the context of semantic segmentation models for images, in which weights from the 2D convolutions used in encoder layers of the model are used in the reverse direction to form the decoder, reducing the total number of model parameters required for processing.

- Complement the network inputs with learnable local inputs and weights, as originally suggested by Uselis et al. (2020), allowing the model to explore local characteristics together with global (i.e., translation invariant) ones.

- Extend the GridMask data augmentation method, originally proposed by Chen et al. (2020), a computationally efficient method for structured information removal to avoid over-fitting and increase model robustness, for pixel-wise forecasting tasks that involve sequences of observations.

- Evaluation and extension of several recent universal techniques in the literature applied to pixel-wise forecasting tasks, such as replacing the traditional normalization-activation layers of STConvS2S with an adapted version of the batch-based evolving normalization-activation layer from Liu et al. (2020) and replacing the RMSProp training optimizer with the more recent AdaMod (Ding et al., 2019) optimizer.

Part of the work reported on this dissertation was also presented on an short paper entitled *Spatio-Temporal Forecasting With Gridded Remote Sensing Data Using Feed-Backward Decoding*, that was submitted to the ACM SigSpatial 2020 conference. The paper reports on a variety of spatio-temporal forecasting experiments within two distinct tasks, namely the prediction of future time-steps and the reconstruction of an input sequence with a time-step missing. The two datasets used in this paper, also used in the study by Nascimento et al. (2019), describe meteorological variables, namely land and ocean air temperature and precipitation measurements, in the South American region. The deep learning architectures presented consisted on a novel hybrid CNN and RNN model leveraging the feed-backward decoding technique, together with the novel evolving normalization-activation layers. These extensions were evaluated alongside a ConvLSTM (Shi et al., 2015) and STConvS2S (Nascimento et al., 2019) baselines.

1.4 Structure of the Document

The remainder of this document is organized as follows. Chapter 2 introduces fundamental concepts and seminal deep learning techniques, as well as important related work regarding the application of deep learning for spatio-temporal data forecasting. Then, Chapter 3 provides an overview of the considered deep learning approach, detailing the different techniques used. Chapter 4 presents the evaluation methodology, detailing the datasets used and the obtained results. Finally, Chapter 5 concludes this document by summarizing the main findings of this work, and highlighting possible directions for future research.

2 Fundamental Concepts and Related Work

This chapter presents fundamental concepts and related work on deep learning and spatio-temporal data forecasting. First, an overview on seminal methods is provided in Section 2.1. This overview reviews pivotal concepts for learning with neural networks, presenting the perceptron and multilayer perceptron in Section 2.1.1, and extensions of these architectures for specialized domains, including convolutional neural networks and recurrent neural networks, in Section 2.1.2. Finally, Section 2.2 covers relevant related work regarding deep learning applied for forecasting different types of spatio-temporal data.

2.1 Fundamental Concepts

Neural networks were originally proposed as a computational approach to automatically make inferences from data, taking inspiration on the biological brain. In general, neural networks can be seen as nested composite functions that transform the input, and whose parameters can be directly learned to minimize a given loss function computed over the expected result and network output.

2.1.1 Supervised Learning with Neural Networks

The perceptron model is the precursor to modern neural network algorithms. It was first introduced by Frank Rosenblatt (1958) as an electronic system capable of learning to recognize similarities between patterns of information, in a way that closely resembles a biological brain.

The perceptron model, illustrated in Figure 2.1, is a binary classifier that receives a set of input values and outputs zero or one. To compute the output, Rosenblatt introduced weights, i.e. real numbers that represent the importance of the respective inputs to the output. The output is one if the weighted sum of the inputs is above a threshold value, and zero otherwise. The function computed by the model is defined by the weight values. A positive weight represents an excitatory effect while a negative weight an inhibitory effect for the corresponding unit.

Mathematically, the perceptron can be seen as an implementation of the following equation:

$$\mathrm{f}(\mathbf{x}) = \hat{y} = \phi\left(\sum_{i=1}^{n} x_i \times w_i + b\right) \tag{2.1}$$

In the previous expression, $\mathrm{f}(\mathbf{x})$ represents the output prediction, $\mathbf{x} \in \mathbb{R}^n$ is a vector of input values, $\mathbf{w} \in \mathbb{R}^n$ is the vector of weights associated with the inputs, b represents a bias term, and ϕ represents an activation function. The bias term is the negation of the aforementioned threshold value. The activation function used for the standard perceptron is the unit step function, although later proposals suggested different alternatives (e.g., the logistic perceptron, using a differentiable activation function corresponding to a logistic sigmoid).

Given a set of examples $\mathbf{x}$ accompanied by the corresponding labels $y \approx \mathrm{f}^*(\mathbf{x})$, training the perceptron model corresponds to adapting all the weights and bias values to best approximate the optimal values. The function with the optimal parameters is denoted as $\mathrm{f}^*(\mathbf{x})$. During model training from a set of example instances that are visited in sequence, given a miss-classification of a positive or negative example, the weights of the active inputs are increased or decreased respectively. This is known as the perceptron learning rule and it can be written as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta \cdot (y - \hat{y}_t) \cdot \mathbf{x} \tag{2.2}$$

In the previous expression, η is a hyperparameter known as the learning rate. We will explore this parameter in more detail shortly.

Training a perceptron can prove to be difficult since a small change in the weights (or bias) can drastically modify the output (e.g., from 0 to 1). For this reason it is common nowadays to use variations with differentiable non-linear activation functions, such as the aforementioned logistic sigmoid or the hyperbolic tangent, which enable the use of gradient descent.

Regarding further extensions to the perceptron model, we have that non-linearly separable patterns can be learned by layers of interconnected neurons. A concrete architecture based on this idea corresponds to MultiLayer Perceptrons (MLPs), also known as feedforward neural networks. In brief, MLPs are computing systems that consist of several simple computing nodes (i.e., perceptrons) interconnected by links. These nodes are split into layers where each unit in the n^{th} layer is connected to every unit in the $(n+1)^{th}$ layer, as seen in Figure 2.1.

MLP networks are known as feedforward since information flows through the network from the input to the intermediate hidden layers, and finally to the output layer. There are no connec-

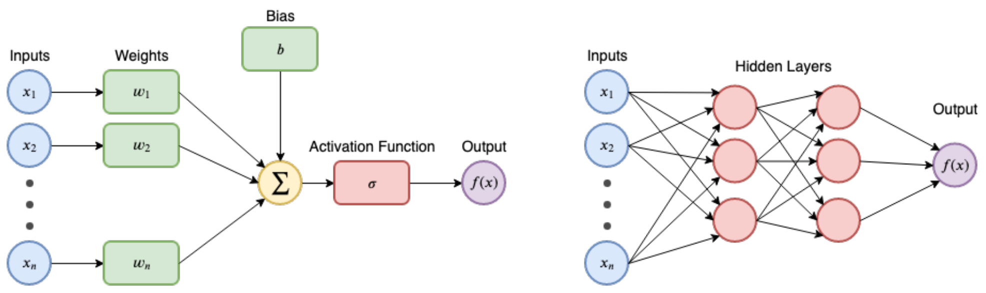

Figure 2.1: The perceptron model (left) and a multilayer perceptron with three layers (right).

tions in which the outputs of the model are fed back into itself. When feedback connections are included, the models are called recurrent neural networks. This approach is detailed in Section 2.1.2.2.

A basic MLP consists of L layers, each containing N hidden nodes. Given an input $\mathbf{x} = (x_1, x_2, ..., x_n)$, the first layer calculates a weighted sum of the input values, in the form:

$$\text{net}_i^1 = \sum_{j=1}^{n} w_{i,j}^1 \times x_j + b_i^1, \quad \text{for } i = 1, 2, ..., N^1 \tag{2.3}$$

In the previous expression, $W^1 \in \mathbb{R}^{N^1 \times n}$ represents a matrix with the weight vectors $\mathbf{w_i^1}$ of the first layer, and b_i^1 represents the corresponding bias term. Each of the i computed values from net_i^1 are then fed into a non-linear differentiable activation function ϕ, resulting in:

$$f_i^1 = \phi(net_i^1), \quad \text{for } i = 1, 2, ..., N^1 \tag{2.4}$$

The remaining hidden layers receive as input the computed outputs of the previous layer f^{l-1} and compute f^l as shown next:

$$f_i^l = \phi\left(\sum_{j=0}^{N^{l-1}} w_{i,j}^l \times f_j^{l-1} + b_i^l\right), \quad \text{for } i = 1, 2, ..., N^l \tag{2.5}$$

In the output layer $l = L$, the network computes the approximation of the optimal function $\text{f}^*(x)$ in the form:

$$\text{f}(\mathbf{x}) = \phi\left(\sum_{j=0}^{N^{L-1}} w_j^L \times f_j^{L-1} + b^L\right) \tag{2.6}$$

During training, the network is driven to match $\text{f}(\mathbf{x})$ to $\text{f}^*(\mathbf{x})$. Each example $\mathbf{x}$ is associated to a label $y \approx \text{f}^*(\mathbf{x})$. This data directly indicates what the output layer should produce at each point $\mathbf{x}$. The training examples do not directly specify the behaviour of the remaining layers. Instead, the model decides on how to use them to best approximate $\mathrm{f}^*(\mathbf{x})$.

Neural networks are usually trained using iterative gradient-based optimizers that lead a loss function to a very low value. A loss function provides a measure of success for the task at hand, and it can be minimized by moving the parameters in the opposite direction of its derivative. This technique is called gradient descent. Since deep learning frequently involves complex non-convex functions with multidimensional inputs, finding the global minimum can be a difficult task. Therefore, we usually settle for a value that is very low, but not necessarily minimal.

There are several variations of gradient descent with varying data requirements. We often perform a trade-off between the quality of the parameter updates and the time required to compute the gradient. For instance, batch gradient descent utilizes the entire training dataset to calculate the gradient of the loss function J with respect to the parameters $\boldsymbol{\theta}$ given a learning rate η.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathrm{J}(\boldsymbol{\theta}_t) \tag{2.7}$$

Since each parameter update requires computing the gradient of the whole dataset, this variation is intractable for neural networks dealing with large datasets. Conversely, batch gradient descent guarantees convergence to a global minimum for convex loss function surfaces, and to a local minima for non-convex loss function surfaces.

Mini-batch gradient descent approximates the gradient utilizing a smaller set of samples b of examples $\mathbf{x}^i$ and corresponding labels $\mathbf{y}^i$.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathrm{J}(\mathbf{x}^{(i:i+b)}; \mathbf{y}^{(i:i+b)}; \boldsymbol{\theta}_t) \tag{2.8}$$

On each parameter update, we sample a mini-batch of examples b uniformly from the training dataset. The mini-batch size is usually between one and a few hundred and it is held fixed even as the training dataset grows. This variation offers a middle ground between computational efficiency and oscillations towards the minimum.

Stochastic gradient descent calculates the gradient for each individual training example $\mathbf{x}^i$ and corresponding label y^i, as shown next:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\mathbf{x}^{(i)}; y^{(i)}; \boldsymbol{\theta}_t) \tag{2.9}$$

Stochastic gradient descent can be seen as a mini-batch gradient descent with a batch size of one. It provides the highest computational efficiency but suffers the most from fluctuations in the cost function due to high-variance updates. This variation has the same convergence guarantees as batch gradient descent, provided the learning rate is low.

The algorithm used to apply gradient descent to the training of neural networks is called back-propagation (Rumelhart et al., 1986). In a first phase, usually known as forward propagation, the input propagates through the network to produce an output $\hat{y}$, with the algorithm maintaining a stack of function calls and their computed parameters. The second phase, known as back-propagation, allows the information from the cost to propagate back through the network. This is made possible by applying the chain rule of calculus, which is used to compute the derivatives of composite functions. Back-propagation starts with the loss value and flows backwards from the final layers to the initial layers, applying the chain rule to the computations of gradient values in order to estimate the contribution each parameter had in the final loss value.

Selecting an appropriate value for the learning rate is a crucial task since this has a significant impact on the model performance (Ruder, 2016). Small learning rate values lead to longer training and, depending on parameter initialization, may cause the model to get stuck in a local minimum. On the other hand, large learning rate values lead to larger jumps in the gradient direction, causing the model to follow an oscillatory path and possibly missing minima. Several approaches have been proposed to address these challenges. For instance, momentum methods were designed to overcome the aforementioned problems and accelerate learning. The parameters are updated based not only on the current gradient value, but also on the previous parameter updates. To achieve this, momentum methods introduce a component that accumulates an exponentially moving average of past gradients, called velocity $\mathbf{v}$.

$$\mathbf{v}_t = \alpha \cdot \mathbf{v}_{t-1} - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.10}$$

In the previous expression, α is a parameter that controls the effect that previous gradients have on current calculations. The parameters are then updated as follows:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}_t \tag{2.11}$$

Nesterov momentum adds a correction factor to the standard momentum method. In this variation, the gradient is calculated after the current velocity is applied. Intuitively, this can be seen as taking a step towards the momentum direction, and calculating the gradient from that intermediate position as opposed to calculating it from the starting position. This allows the gradient to be corrected based on the momentum direction in the same update. Mathematically, Nesterov momentum is defined as follows:

$$\begin{aligned} \mathbf{v}_t &= \alpha \cdot \mathbf{v}_{t-1} - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \alpha \mathbf{v}_{t-1}) \\ \boldsymbol{\theta} &= \boldsymbol{\theta} + \mathbf{v}_t \end{aligned} \tag{2.12}$$

AdaGrad (Duchi et al., 2011) corresponds to another popular variation, which individually attributes dynamic learning rates to all parameters in the model. The dynamic learning rates grow proportionally to the inverse of the $L2$ norm of all previous gradients given the parameters. As such, large gradients have smaller learning rates and small gradients have larger learning rates. Due to the continuous accumulation of squared gradients, the learning rates continue to decrease during training which may result in aggressive premature decreases. The update rule for AdaGrad is defined as follows:

$$\Delta \boldsymbol{\theta}_t = -\frac{\eta}{\sqrt{\sum_{i=1}^{t} \mathbf{g}_i^2 + \delta}} \odot \mathbf{g}_t \tag{2.13}$$

In the previous expression, $\mathbf{g}_t$ is the gradient at time-step t, $\odot$ denotes an element-wise product, δ is a small predefined value to inhibit divisions by zero in the overall expression, and η is a global learning rate shared by all model parameters.

RMSProp and AdaDelta build upon AdaGrad to overcome the diminishing rates problem. RMSProp replaces the gradient accumulation with an exponentially weighted moving average, which allows it to discard history from the distant past. AdaDelta (Zeiler, 2012) restricts the window of accumulated past gradients to a predefined value. In both variants, the sum of gradients is recursively defined as an exponentially decaying average of past squared gradients.

A more recent proposal that is nowadays commonly used, known as Adam (Kingma and Ba, 2017), can be seen as a combination of RMSProp and momentum. Similarly to RMSProp and AdaDelta, Adam stores an exponentially decaying average of past squared gradients, represented as $\mathbf{r}$. It also maintains an exponentially decaying average of past gradients $\mathbf{s}$, similar to momentum methods.

$$\begin{aligned} \mathbf{s}_t &= \rho_1 \cdot \mathbf{s}_{t-1} + (1 - \rho_1) \cdot \mathbf{g}_t \\ \mathbf{r}_t &= \rho_2 \cdot \mathbf{r}_{t-1} + (1 - \rho_2) \cdot \mathbf{g}_t^2 \end{aligned} \tag{2.14}$$

In the previous equations, the parameters $\mathbf{s}_t$ and $\mathbf{r}_t$ are estimates of the mean and variance of the gradients, respectively. Adam also performs bias corrections on these estimates due to their initialization at the origin, according to:

$$\begin{aligned}\hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\rho_1^t}\\ \hat{\mathbf{r}}_t &= \frac{\mathbf{r}_t}{1-\rho_2^t}\end{aligned} \tag{2.15}$$

These values are then used to perform the parameter updates:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \frac{\hat{\mathbf{s}}_t}{\sqrt{\hat{\mathbf{r}}_t} + \delta} \tag{2.16}$$

One of the central challenges in machine learning consists on developing models that perform well on previously unseen inputs, and not just on those the model was trained on. In other words, we want to minimize both the training and test errors. Underfitting occurs when a model cannot obtain sufficiently low values on the training error, while overfitting occurs when the gap between the two errors is large. An overfit model often learns misleading or irrelevant patterns found in the training data. The process of fighting overfitting by modifying the learning algorithms in order to reduce generalization error, while maintaining the training error, is known as regularization. Many techniques exist that fit under the regularization umbrella, and a common approach is to add penalties to the cost function, defined by a regularization function R. This technique is known as parameter norm penalty, and is formally defined as follows:

$$\mathrm{J}_r\left(x,y,\theta\right) = \mathrm{J}\left(x,y,\theta\right) + \lambda \cdot \mathrm{R}\left(\theta\right) \tag{2.17}$$

In the previous equation, J_r represents the regularized cost function and λ is a hyperparameter that controls the effect of the regularization function on the regularized cost function. Utilizing this technique, we can establish preferences towards specific solutions belonging to the hypothesis space. Many different definitions of the regularization function R() exist, although the most commonly used functions include the $L1$ and $L2$ regularization, defined as follows:

$$\begin{aligned}L1\left(\theta\right) &= \sum_i \left|w_i\right|\\ L2\left(\theta\right) &= \sum_i w_i^2\end{aligned} \tag{2.18}$$

Both $L1$ and $L2$ regularization penalize larger weights, causing the network to be constrained to small, more regular weight values. Typically, only the weights $\mathbf{w}$ are penalized, since the biases tend to achieve an accurate fit with less data.

Another commonly used regularization technique, particularly in the context of neural networks, is called dropout (Srivastava et al., 2014). This technique consists of randomly setting the outputs of specific units to zero. On each training iteration, input and hidden units are zeroed with a fixed probability value p. At test time, all units remain active but the layer's output values are scaled down by a factor equal to p. By temporarily removing units from the network, any unit is prevented from over-relying on specific outputs from previous units, instead being forced to learn to work with a randomly chosen sample of other units. As such, units tend to become more robust, creating more meaningful representations without heavily relying on specific units.

2.1.2 Neural Networks for Processing Sequential Data

This section presents common neural network architectures used for processing structured inputs (e.g., time series data), building on the concepts introduced in the previous section. Convolutional neural networks are presented first, followed by recurrent neural networks.

2.1.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized type of neural networks for processing structured data having a sequential (i.e., 1D) or grid-like (i.e., 2D or 3D) structure. Instead of general matrix multiplications, CNNs use a special type of linear operation called convolution in at least one of their layers. These layers are known as convolutional layers.

Traditional neural networks such as MLPs feature very dense interactions, where every output unit interacts with every input unit. Convolutional networks instead typically have sparse interactions, as illustrated in Figure 2.2. This is achieved by utilizing a kernel smaller than the input, allowing for convolutional layers to learn local patterns utilizing small but meaningful features. Another key difference is parameter sharing. In traditional approaches, each element of the weight matrix is multiplied by a single element of the input and never reused. In a convolutional network, each member of the kernel is usually used at every position of the input, again as shown in Figure 2.2. These two ideas provide some key properties, namely the ability to learn translation invariant patterns (i.e., being able to recognize a pattern in different positions of the input) and the ability to learn spatial hierarchies of patterns (i.e., subsequent layers learn larger, more complex patterns made from those recognized in the previous layers).

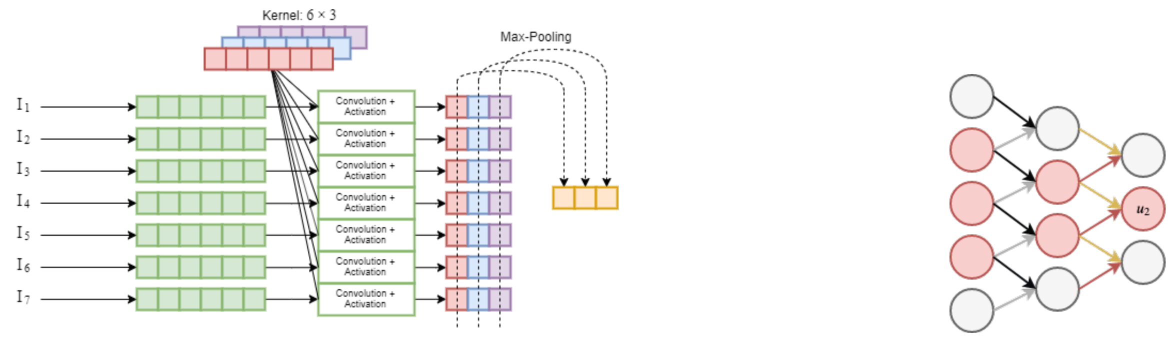

Figure 2.2: Examples illustrating 1D convolution and pooling operations applied over a sequence (left), adapted from Goldberg (2017), and sparse interactions in a convolutional network with two convolutional layers and a 2-element kernel (right). The highlighted units in the right part of the figure are those that affect u_2. These units are known as the receptive field of u_2.

Mathematically, a discrete convolution of two one-dimensional signals $\mathbf{x}$ and $\mathbf{w}$ is defined as:

$$(\mathbf{x} * \mathbf{w})(i) = \sum_{j=-\infty}^{\infty} \mathrm{x}(j) \times \mathrm{w}(i-j) \tag{2.19}$$

In the previous expression, $*$ denotes the convolution operation, $\mathbf{x}$ is the input, $\mathbf{w}$ is referred to as the kernel, i defines the location where the convolution is calculated and j is a value that dictates which input and kernel elements will be multiplied. The output is frequently referred to as a feature map. To apply a convolution, some parameters have to be specified, namely the kernel size, the number of kernels to be applied, the stride size (i.e., how much we slide at each step), and how to deal with non-existent samples. Non-existent samples in the input can be defined to have values of zero, known as zero padding, or by computing the product only on points where samples exist in both the input and the kernel. By utilizing zero padding and different stride sizes, we can adjust the feature map size in relation to the input.

In each convolutional layer, the convolution operation is applied between the input and the

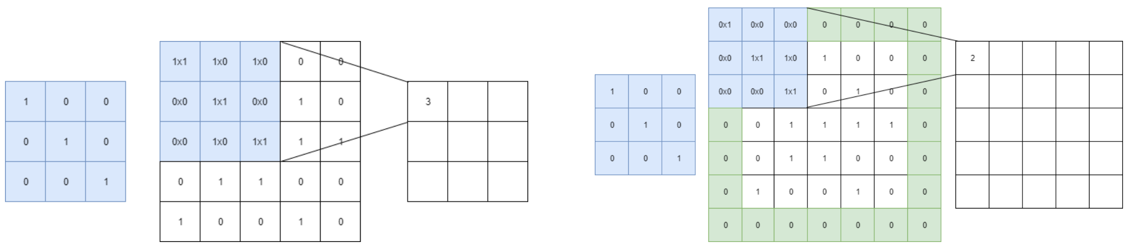

Figure 2.3: Convolution operation on a 5x5 input with a 3x3 kernel of binary values, without zero padding (left) versus with zero padding (right). The example shows how zero padding can be applied to produce a feature map that maintains the input size.

kernel to produce a feature map. In other words, the kernel slides over the input, stopping at every possible location, and computes the dot product between the input and the kernel, as shown in Figure 2.3 for the case of 2D input data. Multiple convolutions are usually performed on an input, each utilizing a different kernel and resulting in a different feature map. The output of a convolutional layer is a stack of all these feature maps.

Convolutional layers are often used together with a non-linear activation function, providing some non-linearity to the network. Most often, the ReLU function is used, which converts all negative values to zero, keeping the positive values. Other recent alternatives include the Swish (Ramachandran et al., 2017) and the Mish (Misra, 2019) activation functions, respectively corresponding to the following two equations:

$$\mathrm{f}(x) = x \cdot \sigma\left(\beta \cdot x\right) \tag{2.20}$$

$$\mathrm{f}(x) = x \cdot \tanh\left(\varsigma(x)\right) \tag{2.21}$$

In Equation 2.20, σ corresponds to the sigmoid activation function, and β is a constant or trainable parameter. In Equation 2.21, ς is defined as $\log\left(1 + e^{x}\right)$. The stack of feature maps that a layer outputs are the results of applying the convolution operation, followed by the activation function in each position.

Convolutional neural networks often consist of multiple layers, combining several convolutional layers and pooling layers. Pooling layers apply a function to the input in order to reduce the number of parameters to process, and to assist in making the representation invariant to small translations. This is achieved by downsampling the input's spatial dimensions (i.e., width and height), while maintaining the depth dimension. The most common pooling function is perhaps max pooling, which outputs the maximum value within the pooling window, although more advanced alternatives are also possible (Deng et al., 2019). Other frequently used pooling functions include average pooling, where the output is the average value of the pooling window, and weighted average pooling, where weights are attributed, for instance with basis on the distance to the central value. When processing sequential data with convolutional neural networks, it is common to use a pooling layer (e.g., max- or average-pooling over time) that aggregates all the feature maps produced at the different time-steps of the sequential input, into an overall encompassing vector.

In a CNN, the final computed stack of feature maps is usually flattened (e.g., through a pooling-over-time operation in the case of sequential data, or instead just by considering a vector

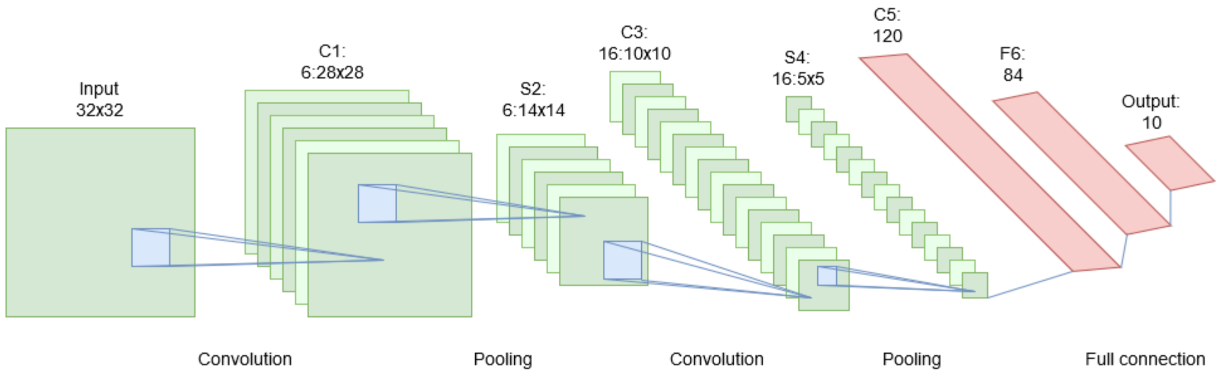

Figure 2.4: Representation of the LeNet5 architecture, adapted from LeCun et al. (1998).

with all the individual values) and fed as input to one or more fully-connected layers, that return the final classification. A loss over this classification output is then used to train the network with the back-propagation algorithm, described in Section 2.1.1.

Figure 2.2 illustrates how CNNs can be applied to sequential data (i.e., series of vectors encoding the relevant information at each time step). In this example, a convolution and an activation function are applied to each window over the input vectors, resulting in seven 3-dimensional feature maps. Afterwards, max-pooling over time is applied, taking the maximum value along each dimension, resulting in the final 3-dimensional pooled vector.

The LeNet5 architecture, introduced by Yann LeCun et al. (1998) and depicted in Figure 2.4, was one of the first successful applications of CNNs for image recognition. The architecture was built in order to recognize hand-written digits. In LeNet5, convolutions are performed utilizing 5x5 kernels with a stride of one, and average pooling is applied with a 2x2 window and a stride of two.

2.1.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are variations of neural networks specifically designed for processing sequential data (i.e., while CNNs can also be used for processing 1D inputs, they are more commonly used in the processing of 2D and 3D data). As opposed to previously seen approaches that process inputs independently, with no state kept between inputs, RNNs maintain a state containing information in regard to what the model has seen thus far.

In a traditional MLP, the network receives one input and produces one output. RNNs instead receive as input an ordered sequence of variable length vectors $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}$, compute a set of internal hidden states $\mathbf{h_1}, \mathbf{h_2}, ..., \mathbf{h_n}$, and produce as output an ordered sequence of variable length vectors $\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_m}$. This allows for more flexibility in the types of data the

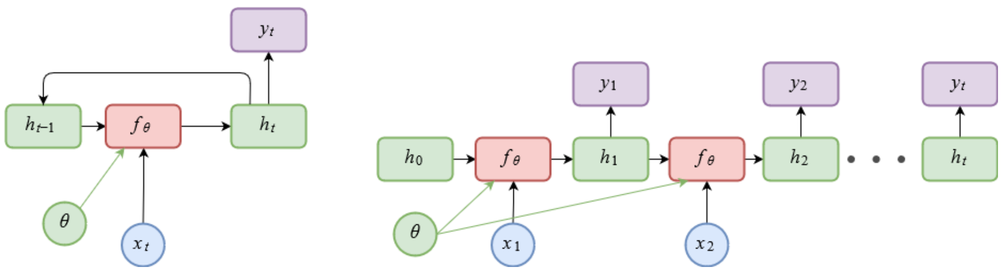

Figure 2.5: Graphical representation of a many-to-many RNN, with a recursive representation on the left, and the corresponding unrolled computational graph representation on the right.

model can process, enabling additional input-output relationships such as one-to-many (e.g., generation of sequences from a single input), many-to-many (e.g., sequence transduction), and many-to-one (e.g., sequence classification). Each state $\mathbf{h_t}$ can be thought of as an encoding of the input sequence seen so far.

Figure 2.5 illustrates a typical many-to-many RNN architecture where at each time-step t the network reads an input $\mathbf{x_t}$, updates its hidden state $\mathbf{h_t}$, and produces an output $\mathbf{y_t}$ given that updated state. Note how the parameters θ are shared across every time-step.

Consider a function f() that, given some parameters $\boldsymbol{\theta}$, the previous state vector $\mathbf{h_{t-1}}$ and the current input $\mathbf{x_t}$, produces an updated state vector $\mathbf{h_t}$. At each time-step, the same function f() and set of parameters $\boldsymbol{\theta}$ are used. There are several implementations of the abstract RNN architecture, each providing different definitions for the function f() and for how the output is calculated. The simplest implementation, known as vanilla recurrent neural network, is defined as follows:

$$\begin{aligned} \mathbf{h_t} &= \mathrm{f}\left(W_{hh} \cdot \mathbf{h_{t-1}} + W_{xh} \cdot \mathbf{x_t} + \mathbf{b}\right) \\ \mathbf{y_t} &= W_{hy} \cdot \mathbf{h_t} + \mathbf{c} \end{aligned} \tag{2.22}$$

In the previous equations, W_{ij} are weight matrices for i-to-j connections (e.g., input-to-hidden state connections are parametrized by W_{xh}), while $\mathbf{b}$ and $\mathbf{c}$ are the bias vectors.

The back-propagation algorithm applied to the unrolled computational graph of an RNN is called back-propagation through time (BPTT). Since in practice it is common to have very large sequences, going backwards through an entire sequence on each backward pass can be computationally expensive and cause the model to never converge. Because of this, it is common to use a variation of BPTT called truncated back-propagation through time, which performs the forward and backwards propagation in chunks of the sequence with a predefined size, as opposed to considering the entire input sequences in each step.

During training, when performing the backwards pass and computing the gradients, each RNN cell receives $\frac{\partial Cost}{\partial h_t}$ and needs to compute $\frac{\partial Cost}{\partial h_{t-1}}$. To achieve this, it is necessary to back-propagate through an activation function and multiply by the respective weight matrix. This causes problems when training over a long sequence since computing the gradient of the cost with respect to one of the first hidden states requires successive multiplications by the same weight matrix. This can cause the gradients to either explode (i.e., if the largest singular value of the matrix is larger than one) or vanish (i.e., if the largest singular value of the matrix is less than one). To deal with these problems, different RNN architectures, inspired by the vanilla version, have been proposed.

Long Short-Term Memory (LSTM) architectures attempt to tackle the gradient problems by optimizing gradient flow. To achieve this, LSTM networks have cells that maintain an additional state $\mathbf{c}$, called cell state, defined by an internal recurrence relation, as shown in Figure 2.6.

Each cell receives as input the previous hidden state $\mathbf{h_{t-1}}$ and the current input $\mathbf{x_t}$, and outputs the updated hidden state $\mathbf{h_t}$. The hidden state is now defined as part of the cell state. To manipulate information flow, LSTM cells have a gating system of four gates, each with the same dimensions as the hidden state. The input gate $\mathbf{i}$ controls what information of the proposed update should be written, the gate $\mathbf{g}$ controls how much of the proposed update is written, the forget gate $\mathbf{f}$ controls what information of the previous cell state should be kept, and the output gate $\mathbf{o}$ decides how much of the cell state is exposed to update the hidden state. Leveraging the aforementioned ideas, LSTM cells can be formally defined as follows:

$$\begin{aligned}
\mathbf{i} &= \sigma\left(W_{hi} \cdot \mathbf{h_{t-1}} + W_{xi} \cdot \mathbf{x_t} + \mathbf{b_i}\right) \\
\mathbf{f} &= \sigma\left(W_{hf} \cdot \mathbf{h_{t-1}} + W_{xf} \cdot \mathbf{x_t} + \mathbf{b_f}\right) \\
\mathbf{o} &= \sigma\left(W_{ho} \cdot \mathbf{h_{t-1}} + W_{xo} \cdot \mathbf{x_t} + \mathbf{b_o}\right) \\
\mathbf{g} &= \tanh\left(W_{hg} \cdot \mathbf{h_{t-1}} + W_{xg} \cdot \mathbf{x_t} + \mathbf{b_g}\right) \\
\mathbf{c_t} &= \mathbf{f} \odot \mathbf{c_{t-1}} + \mathbf{i} \odot \mathbf{g} \\
\mathbf{h_t} &= \mathbf{o} \odot \tanh\left(\mathbf{c_t}\right)
\end{aligned} \tag{2.23}$$

Note that the gates $\mathbf{i}$, $\mathbf{f}$ and $\mathbf{o}$ all use the sigmoid activation function, returning values in $[0, 1]$ with most values near the borders. When performing an element-wise product, indices in the second vector corresponding to near-zero values are blocked, while those corresponding to near-one values are allowed to pass. The gate $\mathbf{g}$ uses the hyperbolic tangent activation function,

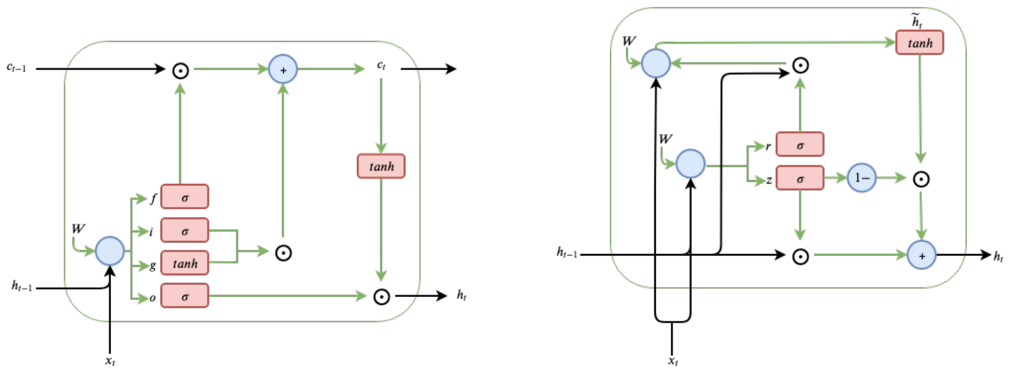

Figure 2.6: Simplified graphical representation of an LSTM cell (left), together with a simplified graphical representation of a GRU cell (right).

which returns values in $[-1, 1]$. The value returned by the model is the candidate value that is considered to be written to each element of the cell state at each time-step.

Analyzing Figure 2.6, we can see the optimized gradient flow while calculating the gradient of the cell state. The upstream gradient follows a horizontal path from $\mathbf{c_t}$ to $\mathbf{c_{t-1}}$, back-propagating through an addition operation and an element-wise product operation, denoted as $\odot$. As such, back-propagating through the cell state only induces an element-wise multiplication between the upstream gradient and the forget gate $\mathbf{f}$, which is different every time-step and always between $[0, 1]$. In the vanilla RNN, back-propagation causes a full matrix multiplication with the same weight matrix several times. The gradient properties of the cell state allows a model to have an almost uninterrupted gradient pathway between LSTM units, this way quickly distributing gradients. These properties allow LSTMs to frequently avoid the exploding/vanishing gradient problems.

Gated Recurrent Units (GRUs) feature a gating system similar to LSTMs, although with fewer gates and without a separate memory component. The first gate, i.e. the reset gate $\mathbf{r}$, controls which parts of the previous hidden state $\mathbf{h_{t-1}}$ are used to compute the next proposed hidden state $\widetilde{\mathbf{h}}_\mathbf{t}$. The second gate, i.e. the update gate $\mathbf{u}$, controls which information of the previous hidden state $\mathbf{h_{t-1}}$ should be forgotten by applying an interpolation between the previous hidden state $\mathbf{h_{t-1}}$ and the proposed update $\widetilde{\mathbf{h}}_\mathbf{t}$. GRUs can be defined as follows:

$$
\begin{aligned}
\mathbf{r} &= \sigma\left(W_{hr} \cdot \mathbf{h_{t-1}} + W_{xr} \cdot \mathbf{x_t} + \mathbf{b_r}\right) \\
\mathbf{u} &= \sigma\left(W_{hz} \cdot \mathbf{h_{t-1}} + W_{xz} \cdot \mathbf{x_t} + \mathbf{b_z}\right) \\
\widetilde{\mathbf{h}}_\mathbf{t} &= \tanh\left(W_{hh} \cdot (\mathbf{r_t} \odot \mathbf{h_{t-1}}) + W_{xh} \cdot \mathbf{x_t} + \mathbf{b_h}\right) \\
\mathbf{h_t} &= \mathbf{u_t} \odot \mathbf{h_{t-1}} + (1 - \mathbf{u_t}) \odot \widetilde{\mathbf{h}}_\mathbf{t}
\end{aligned}
\tag{2.24}
$$

2.2 Related Work

Conventional Recurrent Neural Network (RNN) architectures, e.g., based on Long Short-Term Memory (LSTM) units (Hochreiter and Schmidhuber, 1997), are commonly employed on forecasting tasks involving time-series data. However, these models consider the input data as sequences of vectors, thus not exploiting the spatial context present in spatio-temporal structures (e.g., raster representations for remote sensing data, consisting of patches with neighbouring cells) provided as input. To address this limitation, Shi et al. (2015) combined convolution operations with LSTMs, simultaneously exploiting the abilities of Convolutional Neural Networks (CNNs) and RNNs to effectively model spatial and temporal information, respectively. In the proposed Convolutional LSTM (ConvLSTM) approach, all input data structures are 3D tensors, with the first dimension corresponding to either the number of measurements or the number of feature maps, and the last two dimensions representing the spatial dimensions (i.e., width and height). By replacing the product operation in the original LSTM with the convolution operation, denoted as $*$ in the equations shown next, the future states of a certain cell are now defined as a function of the inputs and past states of its local neighbours. Consider that I, F, O, and G denote the four standard LSTM gates/operations, t represents a time-step, $\odot$ denotes an element-wise product, H and C represent the hidden state and cell state, respectively, and that X represents the input. The ConvLSTM is formally defined as follows.

$$
\begin{aligned}
I_t &= \sigma\left(W_{hi} * H_{t-1} + W_{xi} * X_t + W_{ci} \odot C_{t-1} + b_i\right) \\
F_t &= \sigma\left(W_{hf} * H_{t-1} + W_{xf} * X_t + W_{cf} \odot C_{t-1} + b_f\right) \\
O_t &= \sigma\left(W_{ho} * H_{t-1} + W_{xo} * X_t + W_{co} \odot C_t + b_o\right) \\
G_t &= \tanh\left(W_{hg} * H_{t-1} + W_{xg} * X_t + b_g\right) \\
C_t &= F_t \odot C_{t-1} + I_t \odot G_t \\
H_t &= O_t \odot \tanh(C_t)
\end{aligned}
\tag{2.25}
$$

A variety of complex architectures can be built using ConvLSTM building blocks (Hong et al., 2017; Alléon et al., 2020; Wang et al., 2017; Jang et al., 2018; Zhao et al., 2018). In the original study, the authors developed a typical encoder-decoder structure comprised of multiple stacked ConvLSTM layers, with each decoder layer initializing its hidden states from the output of the corresponding encoder layer. Final predictions are given by the concatenation of the hidden states from the decoder network, followed by a 1×1 convolution.

The architecture was evaluated on two datasets and compared alongside the fully connected LSTM (FC-LSTM) architecture (i.e., a multivariate version of the original LSTM which does not consider spatial correlations). The first set of experiments was conducted on a synthetic dataset where each time-step contained two or three handwritten digits from the MNIST dataset bouncing inside a 64×64 grid. The digits were initially placed at random and assigned a velocity with a direction randomly chosen by a uniform distribution and amplitude randomly chosen between 3 and 5. The digits bounced off the edges and occluded each other upon reaching the same position. The evaluation metrics used were the Cross-Entropy (CE) per sequence and the number of parameters. A variety of ConvLSTM architectures were tested, varying the kernel sizes, number of hidden states and number of layers. Every ConvLSTM variant significantly outperformed the FC-LSTM, while containing significantly less parameters. Next, the authors assess the generalization ability of the ConvLSTM by testing the best performing variant on out-of-domain inputs, namely sequences with three digits. The results showed that the ConvLSTM was able to distinguish the overlapping digits and predict the overall motion, although predicting notably blurry digits.

The second set of experiments were conducted on a radar echo dataset for precipitation nowcasting, considering the task of predicting the next 5 time-steps, given the previous 5. The radar data described $12,222$ sequences of 20 frames, 5 for the input and 15 for the prediction, sampled every 6 minutes. Each frame was a 100×100 grid of gray-scale intensity values. The data was split into 8148 sequences for training, 2037 sequences for validation and 2037 sequences for testing. The evaluation metrics used were the Rainfall Mean Squared Error (Rainfall-MSE), Critical Success Index (CSI), False Alarm Rate (FAR), Probability of Detection (POD) and correlation values. The ConvLSTM was evaluated alongside the FC-LSTM and previous state-of-the-art algorithm for precipitation nowcasting, ROVER. Although the predictions were blurrier than those from ROVER, the ConvLSTM model clearly outperformed every other model under comparison, reacting better to sudden changes (i.e., more extreme values) in the inputs, and overall achieving more accurate results.

Inspired by the aforementioned ConvLSTM architecture, Wang et al. (2017) proposed another recurrent model named PredRNN, which captures spatial and temporal features in a

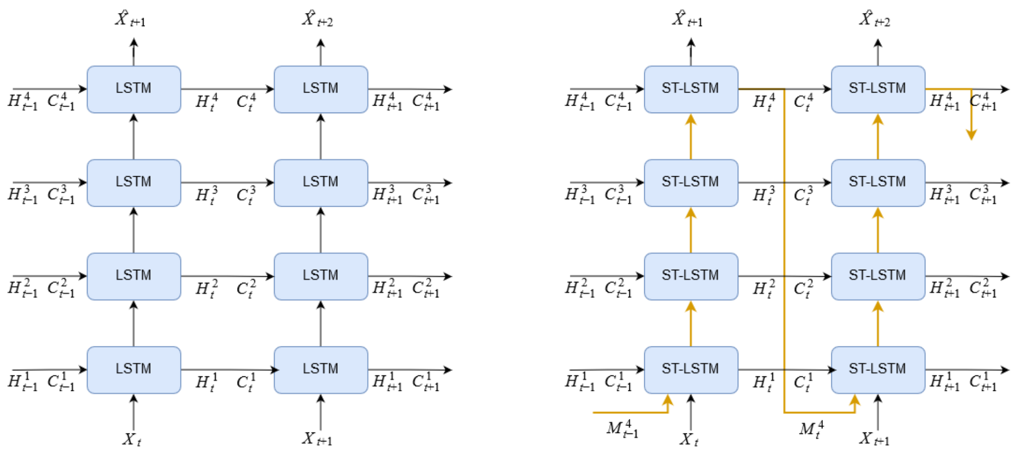

Figure 2.7: Representation of a typical ConvLSTM with four layers (left) and the PredRNN architecture (right), adapted from Wang et al. (2017). The orange arrows denote the flow of the spatio-temporal memory cell M_t^l.

unified memory pool. In PredRNN, the states of an adapted LSTM cell can travel along both vertically between layers and horizontally across states. The authors introduced a new cell state in each LSTM unit, i.e. the spatio-temporal memory cell state M_t, that flows in a zigzag direction, first upwards across layers and then forwards over time, as illustrated in Figure 2.7. This extension to standard LSTM units, called Spatio-Temporal Long Short-Term Memory (ST-LSTM) and illustrated in the left side of Figure 2.8, allows simultaneous flow of both spatial and temporal memory, enabled by the state M_t, and the standard temporal memory, enabled by the state C_t, present in traditional ConvLSTMs. In order to maintain both memory cell states, a new set of gates was constructed to manage the information flow for M_t, in addition to the original gates that handle C_t. Let C_t^l denote the standard temporal cell state at layer l, delivered from the previous unit at $t-1$, and let M_t^l denote the spatio-temporal memory cell state, delivered vertically from the $l-1$ layer at time-step t. The ST-LSTM cells are formally

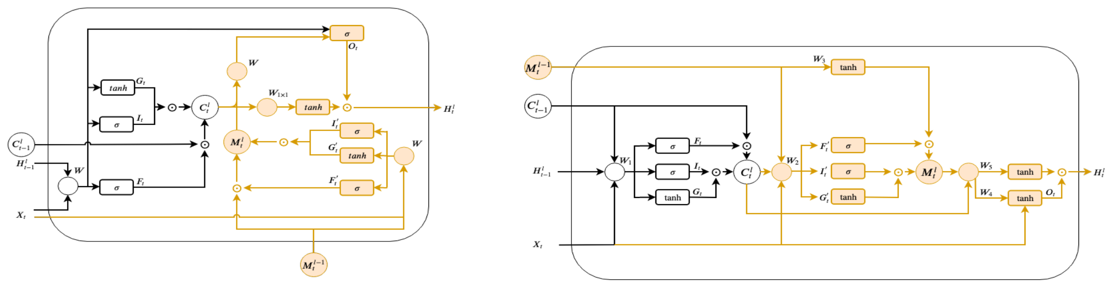

Figure 2.8: Simplified representation of the ST-LSTM (left), adapted from Wang et al. (2017), versus a simplified representation of the causal LSTM, adapted from Wang et al. (2018). The orange coloured elements represent the differences compared with the typical ConvLSTM.

defined as follows.

$$\begin{aligned}
I_t &= \sigma\left(W_{hi} * H^l_{t-1} + W_{xi} * X_t + b_i\right)\\
F_t &= \sigma\left(W_{hf} * H^l_{t-1} + W_{xf} * X_t + b_f\right)\\
G_t &= \tanh\left(W_{hg} * H^l_{t-1} + W_{xg} * X_t + b_g\right)\\
C^l_t &= F_t \odot C^l_{t-1} + I_t \odot G_t\\
I^{'}_t &= \sigma\left(W_{mi} * M^{l-1}_t + W^{'}_{xi} * X_t + b^{'}_i\right)\\
F^{'}_t &= \sigma\left(W_{mf} * M^{l-1}_t + W^{'}_{xf} * X_t + b^{'}_f\right)\\
G^{'}_t &= \tanh\left(W_{mg} * M^{l-1}_t + W^{'}_{xg} * X_t + b^{'}_g\right)\\
M^l_t &= F^{'}_t \odot M^{l-1}_t + I^{'}_t \odot G^{'}_t\\
O_t &= \sigma\left(W_{ho} * H^l_{t-1} + W_{xo} * X_t + W_{co} * C^l_t + W_{mo} * M^l_t + b_o\right)\\
H^l_t &= O_t \odot \tanh(W_{1\times 1} * [C^l_t, M^l_t])
\end{aligned} \tag{2.26}$$

The final hidden states are defined as the concatenation of each memory state derived from different directions, represented as $[C^l_t, M^l_t]$, followed by a 1×1 convolution to ensure consistent dimensionality between states. The authors defined the PredRNN as a multi-layer architecture employing ST-LSTMs, having tasked the model with predicting 10 future observations from three distinct datasets, given the previous 10 observations. The first experiment was conducted on the synthetic Moving-MNIST dataset described above, following the same methodology. In order to assess the generalization ability of PredRNN, the model trained with two digits was tested on another dataset of three digits. The metrics utilized were the Mean Squared Error (MSE) and Cross-Entropy (CE) per time-step, averaged across the dataset. Results showed that PredRNN utilizing the ST-LSTM architecture significantly outperformed every other baseline, including previous state-of-the-art models. In general the proposed model was able to more accurately predict the trajectories and digits, particularly in sequences that displayed overlapping sections, when compared to the baseline models.

A second experiment was performed on the KTH action dataset, which contains six types of human movements (i.e., boxing, hand waving, hand clapping, jogging, running and walking) performed several times by 25 individuals in four scenarios (i.e., indoors, outdoors, outdoors with scale variations and outdoors with different clothing). Each time-step was resized into 128×128 pixels and the sequences were divided with respect to the individuals. The training set contained $108,717$ sequences of the first 16 individuals, and the test set contained $4,086$ sequences of the remaining 9 individuals. During testing, the model was required to predict the subsequent 20 time-steps, as opposed to the subsequent 10 time-steps during training. The metrics used to evaluate the prediction results were the Peak Signal to Noise Ratio (PSNR) and the Structural Similarity Index Measure (SSIM). The PredRNN approach consistently outperformed other models, most of which displaying quick result deterioration for long-term prediction. In contrast, the proposed model displayed the ability to memorize detailed visual appearances and long-term motions, allowing it to perform better predictions both spatially and temporally.

A final experiment utilized a radar echo dataset consisting of $10,000$ consecutive radar observations sampled every 6 minutes in Guangzhou, China. The radar intensities were represented as 100×100 gray-scale images. The $9,600$ sequences were split into a training set of $7,800$ sequences and a test set of $1,800$ sequences. The evaluation metric used in this test was the average per time-step Mean Squared Error (MSE). The authors also compared the training time required for 100 batches and the overall memory usage. Results showed that the previous state-of-the-art model, although providing more accurate forecasts for the near future, rapidly deteriorated for long-term forecasts. On the other hand, the PredRNN did not suffer from the same deterioration, causing the model to outperform the other baselines while simultaneously providing comparable or better time and memory efficiency.

Following their work on the PredRNN model, and in order to lessen the occurrence of the vanishing gradient problem, Wang et al. (2018) proposed the PredRNN++ architecture, leveraging recent advances in gradient flow optimization. The authors noted that increasingly deep-in-time networks, while providing higher modeling capabilities, suffer from more difficulties in gradient propagation. To address this problem, the PredRNN++ architecture features adaptive connections that provide both longer and shorter routes simultaneously between input time-steps and the output future prediction. In more detail, the PredRNN++ architecture, depicted in Figure 2.9, has two key components. The first component, illustrated in Figure 2.8, is a novel spatiotemporal memory mechanism called the causal LSTM that, similarly to the previously mentioned ST-LSTM, simultaneously manages two states, namely the temporal memory state C_t^l and the spatial memory state M_t^l. The causal LSTM allowed authors to efficiently increase recurrence depth between time-steps, causing each pixel on the output time-step to have a larger receptive field at each time-step, which in turn augments the model with more powerful modeling capabilities to stronger spatial correlations and short-term dynamics. The current temporal memory state C_t^l depends directly on the previous state C_{t-1}^l and is controlled through the set of gates F, I and G. The current memory state M_t^l depends on the memory state of the previous hidden layer M_t^{l-1}. For the bottom layer, the assigned memory state corresponds to the memory state at the top hidden layer of the previous time-step. As opposed

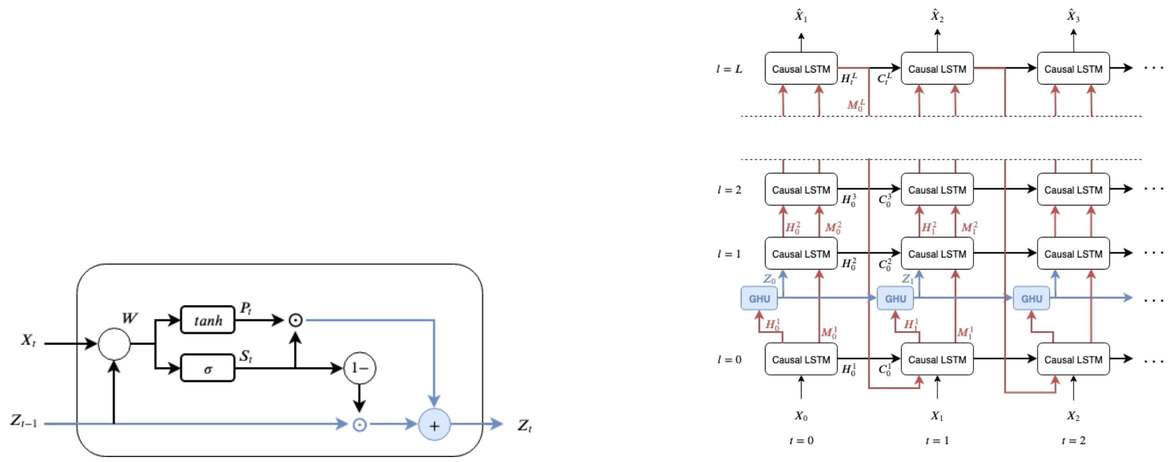

Figure 2.9: Simplified representation of the GHU (left) and the PredRNN++ architecture (right), adapted from Wang et al. (2018). The red coloured elements represent the deep transition pathway, while the blue coloured elements represent the gradient highway.

to ST-LSTMs, the spatial memory state M_t^l is updated as a function of the temporal memory state C_t^l via a different set of gates, i.e. $F^{'}$, $I^{'}$ and $G^{'}$. Formally, a causal LSTM is defined as follows:

$$
\begin{aligned}
I_t &= \sigma\left(W_1 * \left[X_t, H_{t-1}^l, C_{t-1}^l\right]\right)\\
F_t &= \sigma\left(W_1 * \left[X_t, H_{t-1}^l, C_{t-1}^l\right]\right)\\
G_t &= \tanh\left(W_1 * \left[X_t, H_{t-1}^l, C_{t-1}^l\right]\right)\\
C_t^l &= F_t \odot C_{t-1}^l + I_t \odot G_t\\
I_t^{'} &= \sigma\left(W_2 * \left[X_t, C_t^l, M_t^{l-1}\right]\right)\\
F_t^{'} &= \sigma\left(W_2 * \left[X_t, C_t^l, M_t^{l-1}\right]\right)\\
G_t^{'} &= \tanh\left(W_2 * \left[X_t, C_t^l, M_t^{l-1}\right]\right)\\
M_t^l &= F_t^{'} \odot \tanh\left(W_3 * M_t^{l-1}\right) + I_t^{'} \odot G_t^{'}\\
O_t &= \tanh\left(W_4 * \left[X_t, C_t^l, M_t^l\right]\right)\\
H_t^l &= O_t \odot \tanh(W_5 * \left[C_t^l, M_t^l\right])
\end{aligned}
\tag{2.27}
$$

In the previous set of equations, $W_{1:5}$ are the convolutional kernels, with W_3 and W_5 being 1×1 convolutional kernels, and the other ones corresponding to $k \times k$ convolutions, where k is a tunable parameter. The remaining variables and operations have the same meaning as in the set of Equations numbered as 31. The final LSTM hidden state H_t^l is defined by both memory states C_t^l and M_t^l.

The second key component, inspired by the success of highway networks (Srivastava et al., 2015) and illustrated on Figure 2.9, is a new spatio-temporal recurrent structure named Gradient Highway Unit (GHU), introduced in order to mitigate long-term gradient back-propagation challenges. In particular, the introduction of causal LSTMs tends to cause the temporal memory C_t^l to forget older time-steps due to the increased recurrence depth. The GHU maintains an additional hidden state Z_t, defined by the previous GHU hidden state Z_{t-1} and the current LSTM hidden state H_t^l. In the PredRNN++ architecture, the GHUs are injected between the 1^{st} and 2^{nd} causal LSTMs and, as such, receive as input H_t^1 and Z_{t-1}. Updating the hidden state Z_t is enabled by the use of two gates, namely P which controls the proposed update and S which controls the information that should be written. Formally, the GHU is defined as follows:

$$\begin{aligned} P_t &= \tanh\left(W_{ph} * H_t^l + W_{pz} * Z_{t-1}\right) \\ S_t &= \sigma\left(W_{sh} * H_t^l + W_{sz} * Z_{t-1}\right) \\ Z_t &= S_t \odot P_t + (1 - S_t) \odot Z_{t-1} \end{aligned} \tag{2.28}$$

From the previous set of equations, we can see that when S_t takes the extreme values of 0 or 1, the previous hidden state Z_{t-1} is directly passed along or completely overwritten. Controlling the proportions of the hidden state Z_t with S_t enables an adaptive learning of three different information streams: the long-term features in the gradient highway, the short-term features in the deep transition path, as well as the spatial features obtained from the current input time-step. With the use of GHUs, the model provides an alternative pathway from the first to the last time-step, represented as the blue line in Figure 2.9, along which information can flow without attenuation.

The PredRNN++ architecture was also evaluated on the Moving MNIST and KTH Action datasets, with the pre-processing and sequence generation being performed as previously described. The first experiment used the Moving MNIST dataset, containing $10,000$ sequences for the training set, $3,000$ sequences for the validation set, and $5,000$ sequences for the test set. Besides the usual task of predicting the 10 future time-steps given the previous 10 time-steps, the model was also challenged with predicting 30 future time-steps given the previous 10. This was done in order to evaluate the model's ability to perform long-range predictions. The considered evaluation metrics were the per-time-step SSIM and the MSE. The PredRNN++ was able to outperform all other baselines models, including the aforementioned PredRNN architecture, regardless of the considered prediction time horizon. The authors noted that utilizing a 30 time-step prediction horizon, the generated images appeared increasingly blurry, which was attributed to the inherent uncertainty of the future. Results also showed that the model was capable of maintaining the shape of the digits after occlusion, showcasing its ability to diminish vanishing gradients and learn from long-range contexts.

The second experiment used the KTH action dataset, under the same methodology as previously described, considering a training set of $108,717$ sequences and a test set of $4,086$ sequences. The considered metrics were the PSNR and the SSIM, averaged over the 20 generated time-steps for each sequence. On both metrics, the PredRNN++ consistently outperformed the other baselines and state-of-the-art approaches, generating less blurry frames and more accurate predictions regarding the human trajectory and motions. The authors also noted a slow decrease of the metric curves from the 10^{th} to the 20^{th} time-step for the PredRNN++, in opposition with the consistent steeper decrease for the other models under consideration. This showcases a reduction of future uncertainty, indicating the model is able to remember repeated motions and trajectories.

Das and Ghosh (2016) proposed an architecture inspired by the Deep Stacking Network (DSN) from Deng and Yu (2014), adapting the original idea for spatio-temporal forecasting. The proposed model, called Deep-STEP and illustrated in Figure 2.10, is comprised of T stacked modules, where T corresponds to the number of time-steps in the data. The input data for each module is first prepared, so that each instance represents a cell in terms of its spatio-temporal features (i.e., in terms of the values for spatio-temporally neighbouring cells, from previous time-steps and nearby locations). The first module receives as input the raw cell representations, whereas the subsequent modules process a concatenation of the cell representations plus the results from the previous module. Each module is a Multi-Layer Perceptron (MLP) with a single hidden layer and two weight matrices, W and U, representing a lower-layer weight matrix connecting the input and the hidden layer, and an upper-layer weight matrix connecting the hidden layer and the output, respectively. The output Y of each module is thus defined as follows.

$$Y = \sigma\left(\sigma\left(X \cdot W^T\right) \cdot U^T\right) \tag{2.29}$$

The sigmoid functions ensure the output values are contained within the range $[0, 1]$. At each module, to ensure consistency between the outputs and the raw cell representations, the resulting merged input tensor X is normalized prior to being processed by the module. The normalization is defined as follows.

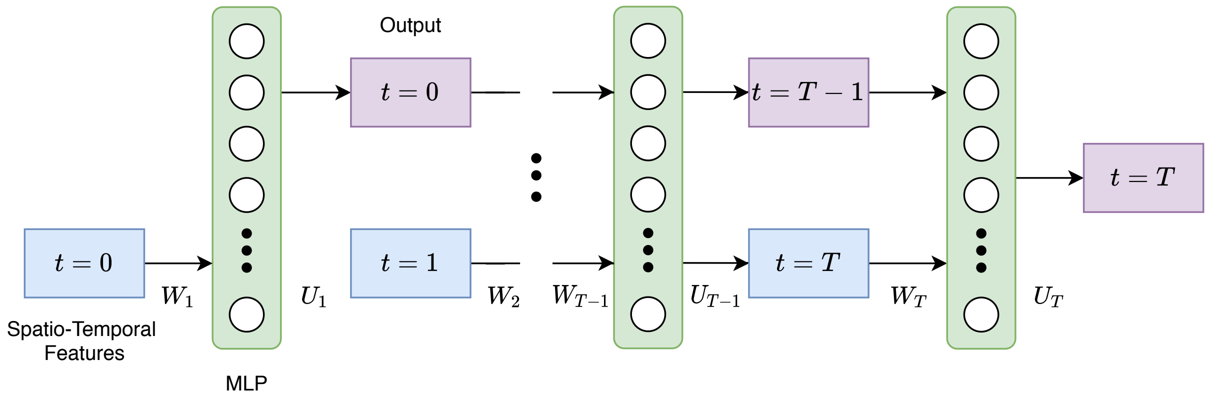

Figure 2.10: Illustration depicting the Deep-STEP architecture, adapted from Das and Ghosh (2016).

$$\text{norm\_}x_{ij} = \frac{(x_{ij} - \min(X))}{(\max(X) - \min(X))} \tag{2.30}$$

The final module outputs each cell's prediction for the desired time-step, as a value within the range $[0, 1]$ which is then mapped to the original scale, resulting in the final predictions.

The Deep-STEP architecture was evaluated in comparison against a traditional MLP, the original DSN (Deng and Yu, 2014), and the NARNET model (i.e., the nonlinear autoregressive neural network model that is provided by MATLAB), on a task related to predicting the Normalized Difference Vegetation Index (NDVI) for four different zones in Kharagpur, India. The data was sampled yearly from 2004 to 2011 and the model was task with predicting the year 2011, given past observed NDVI measurements for 2004-2010. The four different zones were associated to increasingly larger areas, with the first zone having a $3km \times 3km$ area $(10,000$ pixels in total), the second and fourth zone having a $10km \times 10km$ area (around $100,000$ pixels each), and the third zone having a $30km \times 30km$ area $(1,000,000$ pixels in total). During training, the images from $2004 - 2009$ were used to prepare the feature set supporting a prediction for the year of 2010. In turn, during testing, the images from $2004 - 2010$ were used to prepare the feature set supporting predictions for the year of 2011. The evaluation metrics used were the Root Mean Squared Error (RMSE) and Mean Average Error (MAE). The obtained results showed that Deep-STEP produced lower errors for each of the different zones at competitive values for the execution time, clearly outperforming the original DSN architecture Deng and Yu (2014).

In subsequent work, Das et al. (2020) proposed a self-adaptive architecture capable of dynamically adjusting the network structure based on the input data. The architecture features three modules, capturing temporal and spatial features in parallel. The first module computes a

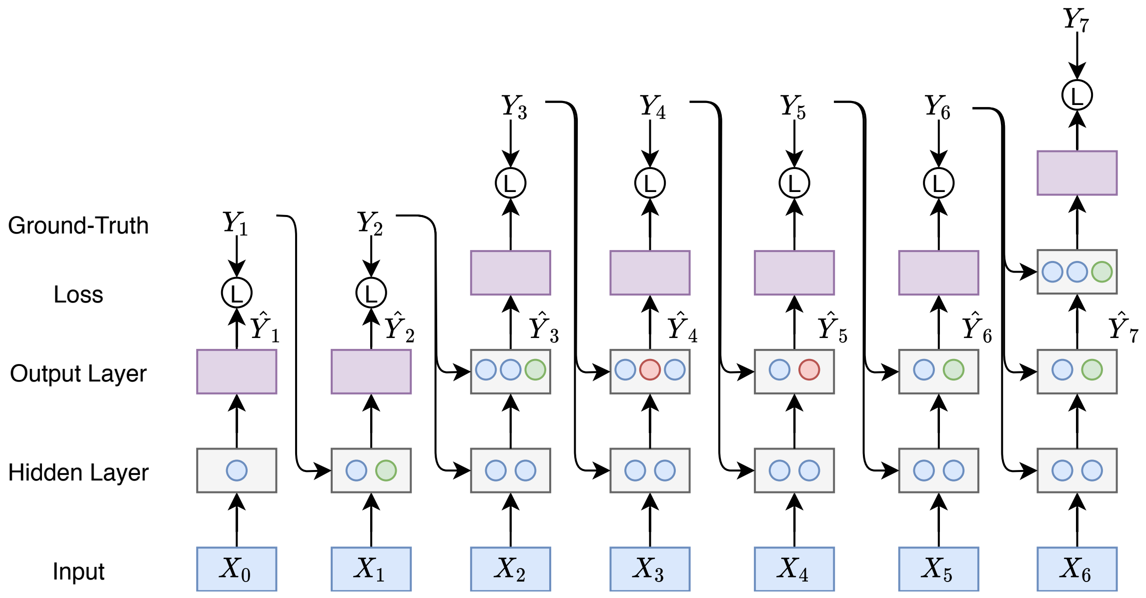

Figure 2.11: Illustration depicting the SARDINE component along with hidden unit pruning and hidden unit/hidden layer growing mechanisms, adapted from Das et al. (2020). The green coloured elements denote the addition of a new hidden unit, while the red coloured elements denote the removal of a hidden unit.

representation for each cell based on its neighbouring values, and the results are then arranged as inputs to the subsequent modules. The second module, illustrated in Figure 2.11, corresponds to a self-adaptive RNN variant named SARDINE that relies on teacher forcing and on the ReLU activation function in the hidden layers, captures temporal dependencies in the results from the first module, processing the cell representations for each timestamp. In turn, the third module captures spatial dependencies in neighbouring cells.

Contrarily to a standard RNN, SARDINE can dynamically change the number of hidden layers and units, by assessing the module's ability to generalize with a Network Significance (NS) method (Das et al., 2019), defined as the sum between the variance and squared bias of the predictions at each time-step. A high NS value indicates over-fitting, and the module reacts by pruning the least significant hidden unit of the top-most hidden layer. In turn, a low NS value indicates under-fitting, to which the module reacts by growing new hidden units at the top-most hidden layer. Besides pruning/growing hidden units, SARDINE is also capable of growing new layers, allowing for more complex functions to be learned. Whenever a drift in spatial context is detected, through an adapted version of Hoeffding's error bound technique (Frías-Blanco et al., 2014), a new layer is added on top of the current top-most hidden layer, with the same number of hidden units, enhancing the network's ability to react to sudden spatial changes. The weight matrix between the new layer and the output layer is initialized with the same values as the weights between the previous layer and the output layer. In turn, the weights between the new layer and previous layer are initialized with an identity matrix.

The third module captures spatial dependencies (i.e., how neighbouring locations affect each cell). It receives the spatial information from a given neighbourhood for each target cell at a specific time-step, and outputs the predicted value for the cell. The module corresponds to a standard MLP, with the hidden layer containing half the units of the input layer, and the output layer containing a single unit that outputs the final predicted value for each cell. The final prediction for the desired time-step is afterwards given by feeding the corresponding predicted spatial features generated by the second module through the parallelly learned third module.

The model was evaluated on three randomly selected areas from two datasets of annual NDVI time series imagery. The first dataset described annual NDVI imagery, pertaining to the state of West Bengal, India, from 2004 to 2011 and at a spatial resolution of 30 meters. The authors randomly selected two zones from this dataset, which belonged to the district of Bardhaman. Training was performed with the data from 2004 to 2010 and testing was performed with the data for 2011. The second dataset consisted of annual NDVI imagery pertaining to the northern part of Brazil, South America, from 2012 to 2019 and at a spatial resolution of 500 meters. The authors randomly sampled a region, resulting in a zone belonging to the state of Para. Similarly to the first dataset, training was performed with the data from 2012 to 2018 and testing was performed with the data from 2019.

The proposed architecture was compared alongside eight baselines, including the traditional MLP, a mixed CNN-RNN architecture, and the aforementioned NARNET, DSN and Deep-STEP architectures. The evaluation metrics used were the Normalized Root Mean Square Deviation (NRMSD), the MAE, the execution time, the PSNR, and the Mean Structural Similarity Index Measure (MSSIM). The former two metrics provide an estimate over prediction error (i.e., lower values denote better results, with 0 denoting the best possible result), while the latter two metrics assess the prediction quality (i.e., higher values denote better results). Results showed that the proposed model outperformed baselines in almost all scenarios, with previous state-of-the-art models achieving comparable results, but with a considerably larger parameter count. Furthermore, the proposed architecture was found to be considerably faster than CNN/LSTM baselines (including previous state-of-the-art models leveraging Gaussian Process (GP) layers), while slower than models based on MLPs (i.e., MLP, DSN and Deep-STEP). The authors note that although the proposed model was seemingly slower than some of the baseline models, these models required several executions to determine the best network structure that were not considered for this evaluation. SARDINE on the other hand, automatically adjusts the network structure to fit the data, requiring only a single execution. The authors hypothesize that if all empirical executions were considered, the execution time of SARDINE would be considerably lower.

Zhang et al. (2016a) proposed a deep learning model based on residual connections to predict the traffic flow of crowds. The model takes as input sequences of 2-channel matrices, with each cell being associated to a geo-spatial region, and the two channels corresponding, respectively, to crowd inflow and outflow (i.e., traffic entering or leaving the region, respectively). The input is sliced into three chronologically ordered subsets of data, denoting distant history, near history, and recent time. The proposed neural architecture, named ST-ResNet and illustrated in Figure 2.12, leverages 2D convolutions and is comprised of four components, respectively modeling temporally *close* information, information from a longer *period*, *trends*, and influence from *external* factors, with the former three components sharing a network structure. The three subsets of data are fed into a respective component, with the trends component receiving the distant history, the period component receiving the near history, and the closeness component receiving the remaining subset. These three components are comprised of two convolutional blocks, with a sequence of residual units between them. The first convolutional block processes a tensor corresponding to the concatenation of the input patches for the different time-steps. Each residual unit combines multiple convolution operations together with ReLU activation functions and batch normalization operations, featuring also a skip connection that adds the input of the unit to the result of the last convolution. The residual units allow for the construction of a deeper network, better modeling spatially distant dependencies (i.e., each cell in the final feature map depends on all cells of the input grid). The final convolutional block produces an output patch for the component, with a number of channels corresponding to the prediction objective.

The outputs from the closeness, period, and trend components are combined through learnable weight matrices that assign different degrees of influence to each aspect. These interactions are formally defined as follows:

$$X_{\text{combined}} = W_c \odot X_c + W_p \odot X_p + W_{tr} \odot X_{tr} \tag{2.31}$$

In the previous equation, X_c, X_p and X_{tr} correspond, respectively, to the outputs of the closeness, period and trend components, and W_c, W_p and W_{tr} are the respective weight matrices. The external influence component models potentially useful information from external datasets (e.g., meteorological conditions, patterns associated to holidays or specific days of the week, etc.), with basis on a two-layer MLP. The output of this component is added to the aforementioned

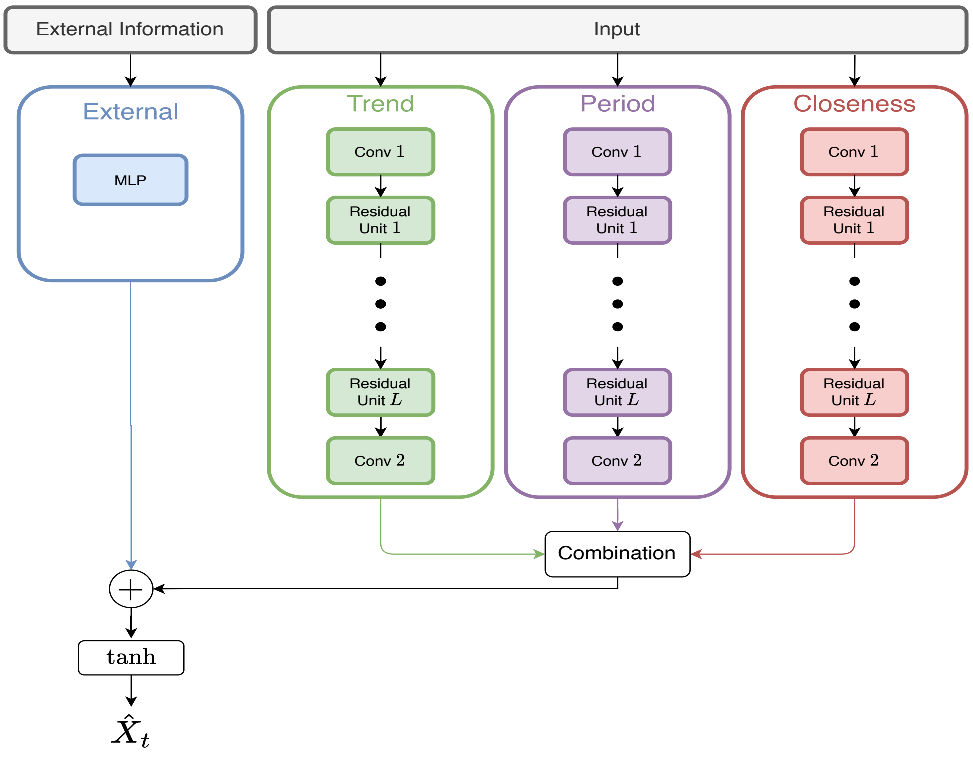

Figure 2.12: Illustration depicting the ST-ResNet architecture, adapted from Zhang et al. (2016a)

combined output, followed by a hyperbolic tangent function, resulting in the predictions for the respective time-step (i.e., the model can be applied sequentially, in order to produce predictions for multiple time-steps from past observations).

ST-ResNet was evaluated on two datasets, each containing information regarding trajectories and external conditions (i.e., meteorological data and holidays). The first dataset contained taxi trajectories in Beijing, considering four different time intervals from $2013 - 2016$. In turn, the second dataset contained bike trajectories in New York City from April to September 2014. Comparisons were made against traditional approaches such as ARIMA, SARIMA, and VAR models, and against neural models corresponding to a MLP or to a previous state-of-the-art approach for crowd flows prediction, named DeepST (Zhang et al., 2016b). The authors also evaluated different configurations for ST-ResNet, varying the external information and the number and/or internal structure of the residual units. Results showed that the ST-ResNet model outperformed all the baselines, with the best versions obtaining significant improvements over the previous best model for each dataset.

Recently, Nascimento et al. (2019) proposed an encoder-decoder model for spatio-temporal forecasting that is comprised exclusively of convolutional layers, which are suited to capture spatial features by design. In STConvS2S, illustrated in Figure 2.13, convolutions are performed

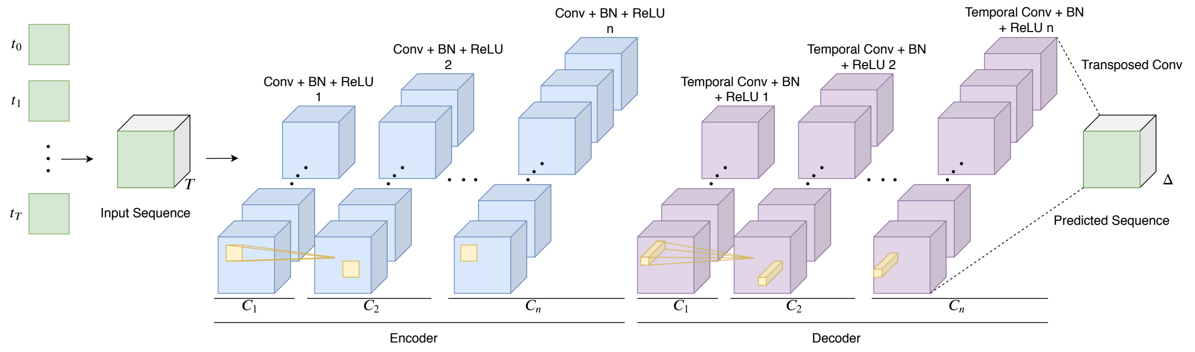

Figure 2.13: Illustration depicting the STConvS2S architecture, adapted from Nascimento et al. (2019). The yellow elements represent the factorized 3D kernel, T represents the number of time-steps, and C_l is the number of channels.

with factorized 3D kernels $K = t \times d \times d$, where t is the size of the temporal kernel and d is the size of the spatial kernel. The encoder processes the input sequence by performing convolutions using just the spatial kernel (i.e., $1 \times d \times d$, with $d = 5$ in the best configuration reported by Nascimento et al.), creating a sequence of meaningful spatial representations. These spatial representations are received as input by the decoder, which applies temporal convolutions with the temporal kernel (i.e., $t \times 1 \times 1$) in order to learn the temporal features, resulting in the predicted future sequence. Both the encoder and the decoder are comprised of successive convolutional blocks with batch normalization (Ioffe and Szegedy, 2015) and followed by a ReLU activation function, with the decoder utilizing temporal convolutions (Oord et al., 2016) to maintain temporal coherency during prediction (i.e., predictions for a time-step t make no use of future information from time-steps $t + 1$ onward). Given that standard temporal convolution operations output either a shorter sequence or a sequence of the same size as the input, the authors introduce a transposed convolution operation before the final convolutional layer, allowing for predictions that exceed the sequence length of the input. A transposed convolution operation, also known as deconvolution, decomposes the input feature map into several intermediate feature maps

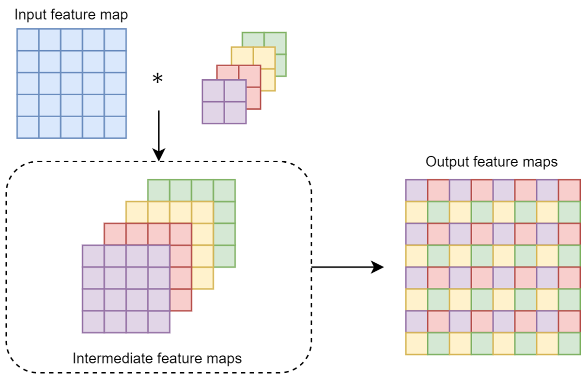

Figure 2.14: Illustration of a 2D transposed convolution, where the original input is upsampled to 8×8. The figure was adapted from Gao et al. (2019).

generated by different kernels, which are afterwards shuffled and combined to produce a final, upsampled, feature map. These interactions are illustrated in Figure 2.14.

Experiments were performed with two datasets describing metereological variables in a subset of the South American region. The first dataset, named CFSR (Saha et al., 2014), describes $54,041$ sequences of land and ocean air temperature measurements, for the spatial regions with latitudes between $8°$N and $54°$S and longitudes between $80°$W and $25°$W). Data was collected from 1979 to 2015, measuring temperature every 6 hours at a spatial resolution of $0.5°$. The second dataset, named CHIRPS (Funk et al., 2014), describes $13,960$ sequences of precipitation measurements, for the spatial regions with latitudes between $10°$N and $39°$S and longitudes between $84°$W and $35°$W. Data was collected from 1981 to 2019, measuring daily precipitation at a spatial resolution of $0.05°$, and the original data was interpolated to a resolution of $1°$ per cell. The data was afterwards split $60\% - 20\% - 20\%$ for training, validation and test respectively. The considered evaluation metrics were the RMSE and the MAE. Results showed that STConvS2S outperformed the other baseline models in all datasets, prediction horizons, and metrics, while simultaneously requiring less time to train. Notably, the model displayed an improvement of 20% in RMSE and was $2.5\times$ faster with a prediction horizon of 5 for the CFSR data, in comparison to the ConvLSTM model that was previously presented.

3 The Deep Learning Approach for Spatio-Temporal Data Forecasting

This chapter details the proposed deep learning approach. First, Section 3.1 presents all the techniques studied and how they were incorporated in the developed architectures, followed by Section 3.2 which overviews the contents of this chapter.

3.1 The Proposed Extensions

This section details the main extensions proposed over the STConvS2S architecture, first presenting the novel decoding strategy based on combining a LSTM with feed-backward decoding in Section 3.1.1, followed by a description of the strategies for (a) novel normalization-activation layers in Section 3.1.2, (b) GridMask data augmentation in Section 3.1.3, and (c) complementing the inputs with learnable local features and weights in Section 3.1.4.

3.1.1 Feed-Backward Decoding

When using an encoder-decoder model for tasks involving spatial structures (e.g., tasks such as semantic segmentation of image inputs), it is common to have the encoder comprised of CNNs that process the input, progressively creating increasingly compact representations that capture meaningful features, which are afterwards sent as input to the decoder. In situations where retaining the spatial dimensions of the original input is crucial, such as spatio-temporal forecasting or semantic segmentation, the decoder typically applies transposed convolutions or some other interpolation technique to upscale the intermediate representations (e.g., bi-linear up-sampling followed by a traditional convolution operation).

Wang et al. (2019) proposed a novel technique to maintain the original spatial dimensions, by using an encoder in the opposite direction to decode (i.e., feed-backward decoding). During decoding, the existing convolutional layers and filters of the encoder are re-used, allowing learned features to be mapped from smaller dimensions to larger dimensions, without additional parameters. To apply this technique, a couple of considerations need to be taken into account. For instance, any pooling operations used in the encoder need to be replaced by interpolation

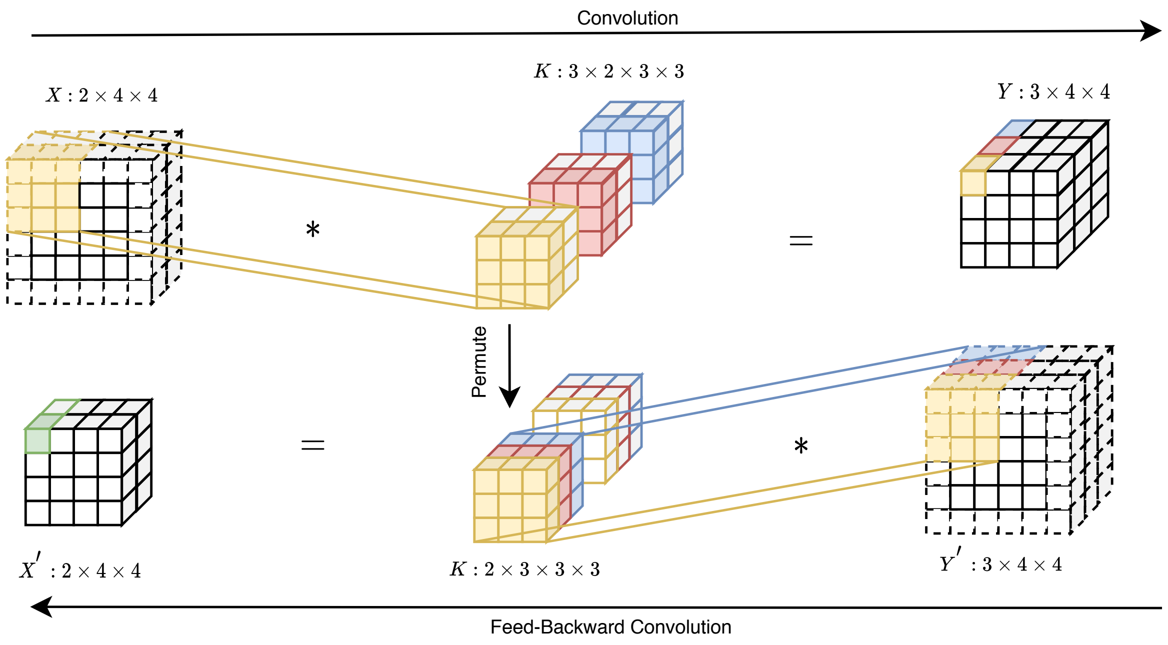
Convolution
$X : 2 \times 4 \times 4$
$K : 3 \times 2 \times 3 \times 3$
$Y : 3 \times 4 \times 4$
$*$
$=$
Permute
$=$
$*$
$X' : 2 \times 4 \times 4$
$K : 2 \times 3 \times 3 \times 3$
$Y' : 3 \times 4 \times 4$
Feed-Backward Convolution

Figure 3.1: Illustration for how the dimensions of a filter can be permuted to transform the channel dimension. The cells with dotted borders correspond to zero-padding, and $*$ denotes a standard convolution with a stride of one. The middle row illustrates the normal versus permuted convolution filters.

operations (e.g., bi-linear interpolation) during decoding. Also, if the channel dimension of the input has its size altered in a convolutional layer L, the weights used during decoding are the weights from L with the input and output channel dimensions permuted. Figure 3.1 illustrates the feed-backward decoding procedure, showing on the top part of the figure a typical convolution between a zero-padded input X with 2 channels and spatial dimensions of 4×4, and a set of 3, 3×3 filters with 2 channels each (i.e., a filter tensor with 2 input channels and 3 output channels). The result is a tensor Y with 3 channels and the same spatial dimensions as the input. During the decoding phase, the input-output flow is reversed, and the convolution is now performed over an input with the same dimensions as the output of the original convolution (plus zero-padding to guarantee consistent spatial dimensions). The same set of filters are re-utilized by swapping the input and output channel dimensions, yielding a set of 2, 3×3 filters with 3 channels each. With this technique, the network learns to both capture spatial features during encoding and also reverse its effect back to the original representation during decoding, with the regrouping of the filters dictating the behaviour.

Leveraging the idea of feed-backward decoding, we propose two variations of the STConvS2S model, denoted throughout this dissertation as ST-FD and ST-RFD. The first variation simply incorporates this technique with the original STConvS2S model, maintaining the encoder-decoder structure, but passing the outputs provided by the decoder back through the encoder in

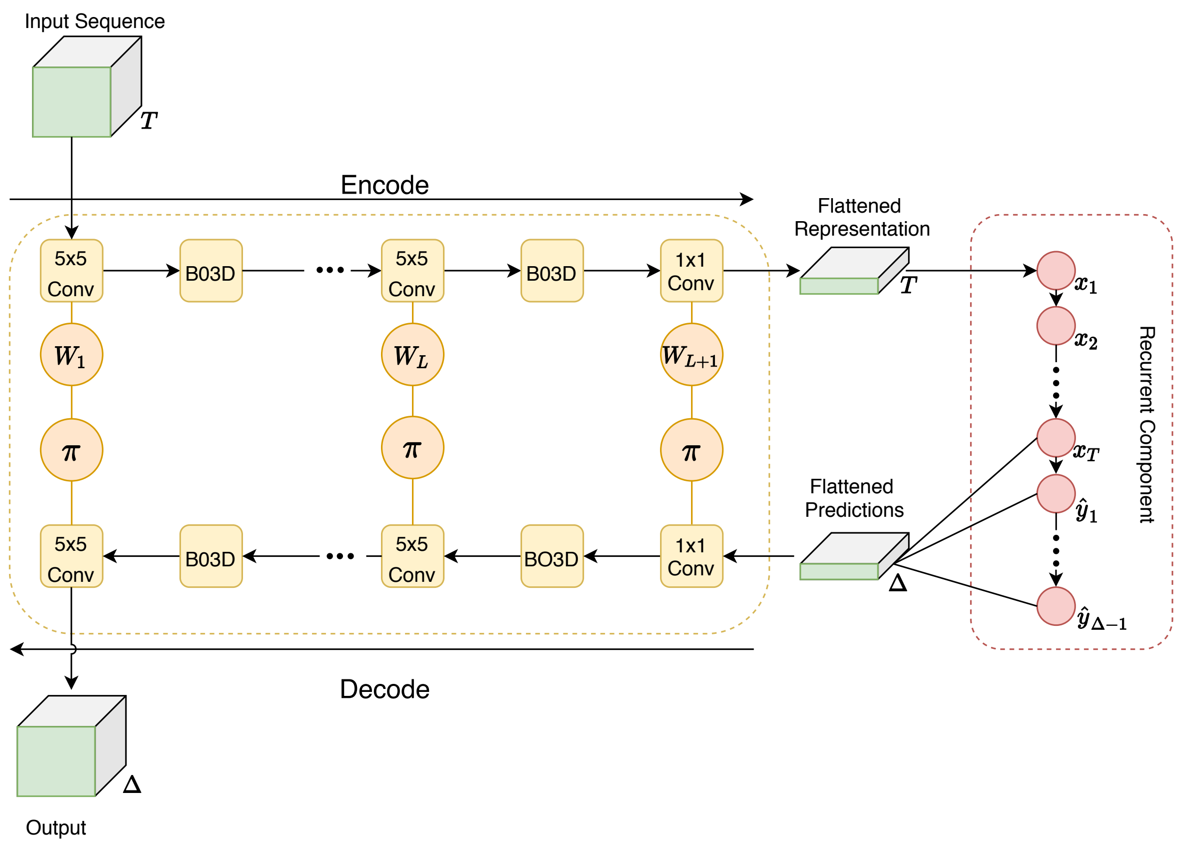

Figure 3.2: The ST-RFD model leveraging feed-backward decoding with $L + 1$ convolutional layers. The boxes labeled as B03D correspond to normalization-activation operations for the encoder and decoder. The orange circles indicate weight-sharing, with π corresponding to a permutation of dimensions.

the opposite direction, sharing convolutional weights through a permutation of dimensions. The final convolution operation is now performed at the end, following all computations. The second variation, illustrated in Figure 3.2, expands upon the first variation by replacing the decoder component with a recurrent component to capture temporal patterns. This alternative allows us to exploit different recurrent approaches for capturing temporal features, without incurring in an additional increase of learnable parameters through separate convolutions in the encoder and decoder parts.

In particular, given that the encoder is now also used for decoding, ST-RFD no longer captures temporal features through factorized temporal filters. Instead, an additional recurrent component is included, consisting of an LSTM unit that, at each time-step, receives a flattened representation of the learned spatial features, and outputs a representation for the prediction of the next time-step(s). During decoding, a component that shares its parameters with the encoder receives the predicted flattened representations and outputs the final predictions, with the information flowing from the last layer to the first layer. The recurrent component trivializes the prediction of sequences with a length differing from that of the input sequence, and the transposed convolutional layer from the STConvS2S architecture is not used. Besides the aforementioned changes, the original encoder structure from STConvS2S, relying on multiple 5×5 convolution operations, can be maintained, with the addition of a 1×1 convolutional layer at the end to reduce the dimensionality of the representations passed to the LSTM (i.e., I considered a final channel dimensionality of one while still preserving the spatial dimensions).

3.1.2 Novel Normalization-Activation Layers

In both the encoding and decoding parts of the ST-RFD architecture, novel normalization-activation layers are located after every convolution operation (i.e., after the 5×5 convolutions), except for the final convolutional layers along each direction (i.e., encoding or decoding). These novel normalization-activation operations are derived from the recently proposed Evolved Normalization-Activation (EvoNorm) layers, proposed in the study by Liu et al. (2020). The chosen operation is an adapted version of the EvoNorm B0 layer, which was the best performing batch-based version reported by Liu et al. (2020). Let v_1, γ and β denote learnable parameter vectors, and let $s_{b,h,w}$ and $s_{h,w}$ represent the variance of a mini-batch and the variance of a single instance, respectively. EvoNorm B0 uses the following computation over inputs x.

$$\mathrm{B0} = \frac{x}{\max\left(\sqrt{s^2_{b,w,h}(x) + \epsilon}, v_1 \cdot x + \sqrt{s^2_{w,h}(x) + \epsilon}\right)} \cdot \gamma + \beta \tag{3.1}$$

The proposed extension to EvoNorm B0 explicitly models spatio-temporal scenarios, considering sequences d of two-dimensional inputs when calculating both the batch and instance variance (i.e., the values associated to each time-step in the input sequence are considered separately when computing the variance). This extension, denoted here as B03D, is defined as follows.

$$\mathrm{B03D} = \frac{x}{\max\left(\sqrt{s^2_{b,d,w,h}(x) + \epsilon}, v_1 \cdot x + \sqrt{s^2_{d,w,h}(x) + \epsilon}\right)} \cdot \gamma + \beta \tag{3.2}$$

3.1.3 GridMask Data Augmentation

Information removal methods are a subset of data augmentation techniques that seek to reduce overfitting and improve the feature extraction of CNNs, by deleting chunks of the input

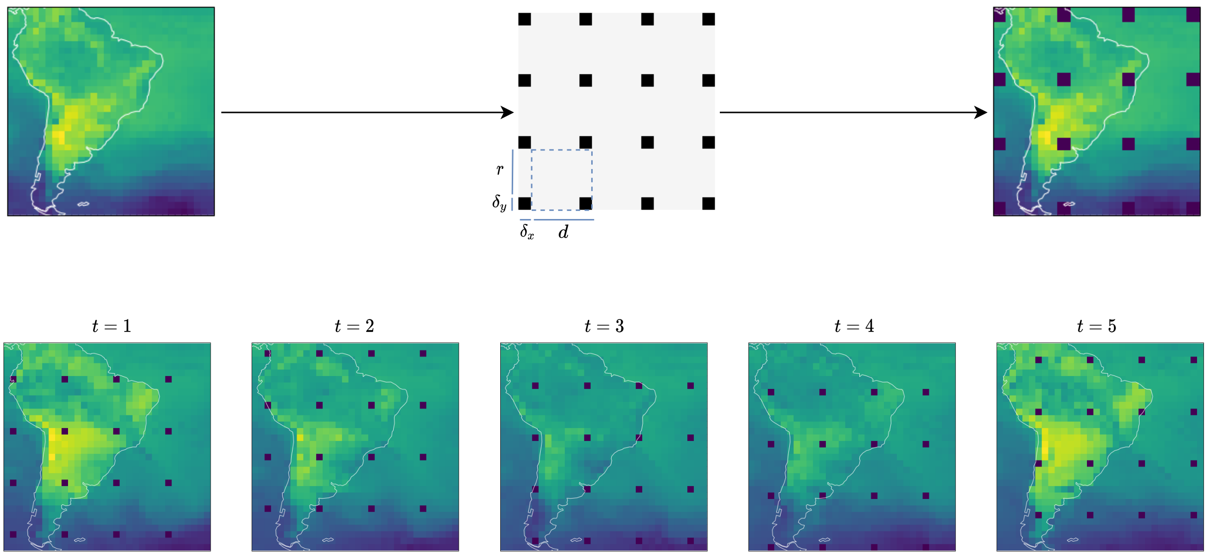

Figure 3.3: Example for the application of the GridMask technique over an input sequence with 5 observations from the CFSR dataset. The black boxes on the different steps represent deleted regions in each of the observations. The diagram on the top part of the figure visually illustrates the different parameters involved in the application of the GridMask technique. The blue colored dotted square represents one mask unit.

data in a randomized or structured manner. The main difficulties associated with these techniques relate to balancing the amount of information to remove, avoiding the excessive removal of meaningful data or the over-reservation of large continuous regions on the input. Seeking to mitigate these challenges, Chen et al. (2020) proposed a structured information removal method called GridMask that provides control over the density and size of the deleted regions, making it suitable for a variety of tasks.

GridMask applies a grid mask over the input defined by four parameters, namely r, d, δ_x and δ_y. The parameter d defines the length of a single unit on a mask, while the parameter r controls the ratio of input data maintained within each unit. The parameter r also defines the keep ratio of an image k (i.e. the percentage of information maintained from the original image), defined as follows:

$$k = 2 \cdot r - r^2 \tag{3.3}$$

The parameters δ_x and δ_y correspond, respectively, to the horizontal and vertical distance between the boundary of the image and the first intact portion, allowing the mask to be shifted both vertically and horizontally. Similarly to the original study, we define d as a random value between two tunable limits, and δ_x and δ_y as random values between 0 and $d - 1$.

I also propose to extend the GridMask method to 3D input data (i.e., sequences of observations) for pixel-wise tasks (e.g., forecasting future sequences or reconstructing missing time-steps), by sampling the parameter d for the whole sequence, and randomizing the parameters δ_x and δ_y for each individual observation, as shown in Figure 3.3. This corresponds to formulating the same mask with randomized vertical and horizontal translations for each observation. Coupled with setting the unit length d to a range of small values, these extensions help motivate the use of spatially nearby locations when predicting a value for the current location. The masks are applied with a fixed probability value p for each batch of sequences during all training epochs, with different masks being generated for each sequence within the batch. Through initial experiments, we found that this approach obtained superior results when compared to progressively increasing the masking probability at each training epoch.

3.1.4 Local Weights and Learnable Inputs

Another technique considered was the extension of the input data with learnable local features and weights, as originally proposed by Uselis et al. (2020). In the original study, the authors studied techniques to incorporate learnable features, intrinsic to each spatial location, within CNN models, seeking to improve spatio-temporal forecasting by combining both (i) global location-invariant features, and (ii) location-specific features. Typical CNNs are mostly translation invariant (i.e., translating an input x and convolving with a filter k is likely to yield the same result as translating the feature map resulting from a convolution between x and k), treating each spatial location equally and thus learning global patterns. This is often insufficient in many spatio-temporal forecasting scenarios, where the behaviour in specific locations should be guided by their own intrinsic local features. To allow the learning of such features, the authors propose two complementary techniques, namely Learnable Inputs (LI) and Local Weights (LW). The LIs correspond to a set of n trainable parameters with the same spatial dimensions as the original input, concatenated with the input before processing with a convolutional layer. These parameters allow the network to learn local features regarding every spatial location, which can complement the learned global patterns or be ignored by the convolutional filters in situations where they are not beneficial. The LWs further reinforce the individualized learning of different spatial locations, through a locally-connected layer (i.e., a convolutional layer with a different filter at each input region, allowing different spatial locations to be weighed according to their relevance) of weights over the input, resulting in m trainable weights with the same spatial dimensions as the input. Similar to LIs, these weights are afterwards concatenated with the original input. Both LIs and LWs can be used individually or together, and the concatenations with the input can be done before any convolutional layer, making that layer sensitive to local

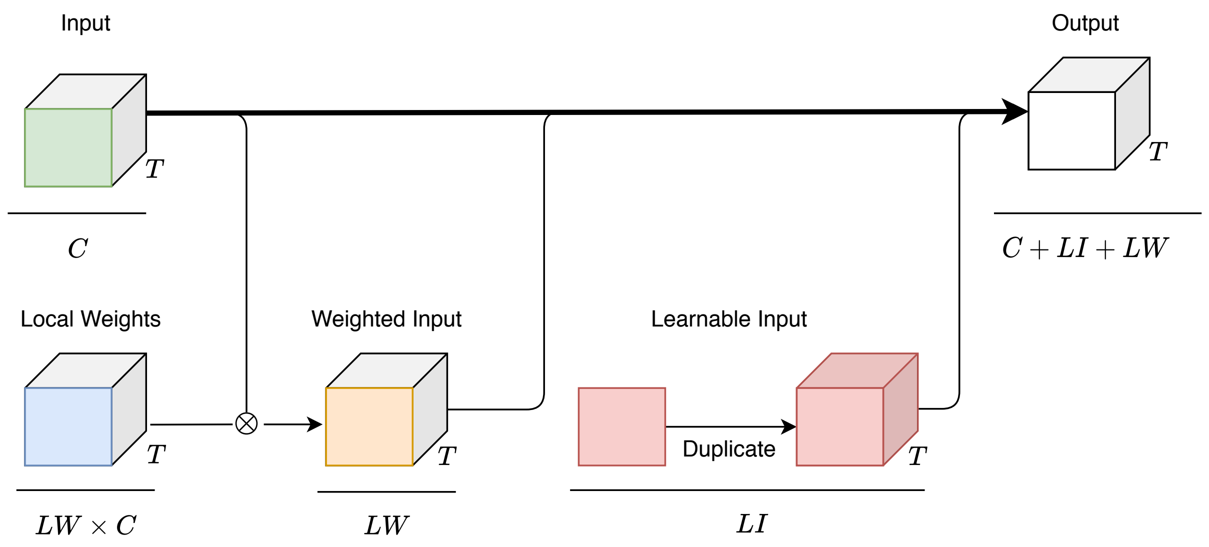

Figure 3.4: Illustration for the use of Local Weights and Learnable Inputs, adapted from Uselis et al. (2020). The values C, LI and LW represent the number of channels, learnable inputs and local weights, respectively, while T represents the number of time-steps and $\otimes$ represents a locally-connected convolution.

features.

In my experiments, the LIs and LWs are concatenated with the inputs (sharing the LIs across the different input time-steps and using different LWs in each time-step), prior to every convolutional layer – see the representation in Figure 3.4. Furthermore, the simultaneous use of both techniques has proven to result in significant performance gains compared to using only a single technique. Similarly to the original study from Uselis et al. (2020) LIs are implemented with a locally connected layer that receives as input a constant unitary tensor with the same spatial dimensions as the original input, and the LWs are implemented with a separate locally connected layer that receives the original tensor as input. Both LIs and LWs are afterwards concatenated with the input tensor on the channel dimension, causing the convolutional layers to receive an input with a number of channels increased by the number of LIs and LWs. During the decoding phase, the feed-backward strategy employed also produces an output with additional channels, but we can ignore the channels corresponding to the LIs and LWs (i.e., the predictions are obtained by slicing the output tensor on the dimensions matching the original dataset channels).

3.2 Overview

This chapter presented the novel approach for spatio-temporal data forecasting that was considered in this dissertation. Concretely, it first presented the main architecture as an extension of the STConvS2S architecture leveraging a recurrent component and the feedbackward decoding technique. Finally, the chapter presented the different techniques that were tested in conjunction with the aforementioned architecture and baselines, namely the novel evolving normalization-activation layers, the GridMask data augmentation method and learnable inputs and local weights.

4 Experimental Evaluation

This chapter presents the experimental evaluation of the proposed extensions, which involved tests with two datasets describing meteorological variables. The general evaluation methodology is presented first, detailing the evaluation metrics and datasets used (Section 4.1). Next, the results for the future time-steps forecasting scenario are presented and discussed (Section 4.2), followed by the results for an additional task pertaining to the reconstruction of a missing time-step from an input sequence (Section 4.3).

4.1 Methodology and Evaluation Metrics

The proposed extensions were evaluated against baselines corresponding to the ConvLSTM and the original STConvS2S models, on the two datasets that were also used in the study by Nascimento et al. (2019), namely the CFSR air temperature dataset and the CHIRPS precipitation dataset. The different models were evaluated in two different scenarios, with the first task corresponding to predicting a future sequence up to a fixed horizon Δ, given the previous sequence of 5 time-steps. In our experiments, we considered $\Delta = 1$ and $\Delta = 5$, i.e., predicting image patches corresponding to the next time-step and next 5 time-steps, respectively. The second task corresponds to reconstructing a sequence with missing information. Specifically, we randomly remove an entire time-step from an input sequence of 10 time-steps, and the models are required to output the entire sequence without incomplete information.

The CHIRPS (Funk et al., 2014) dataset describes $13,960$ sequences of 10 instances for precipitation measurements on a 50×50 grid, each corresponding to a subset of the South American region (latitudes between 10°N and 39°S and longitudes between 84°W and 35°W). Data was collected from 1981 to 2019, measuring daily precipitation at a spatial resolution of 0.05°, and the original data was interpolated to a resolution of 1° per cell. In turn, the CFSR dataset (Saha et al., 2014) describes $54,047$ sequences for air temperature measurements in a 32×32 grid, pertaining to a similar region as the previous dataset (latitudes between 8°N and 54°S and longitudes between 80°W and 25°W). Data was collected from 1979 to 2015, measuring temperature every 6 hours at a spatial resolution of 0.5°. For both datasets, and similarly to

Nascimento et al. (2019), we adopted a splitting strategy involving 60% - 20% - 20% of the data respectively for training, validation and testing, in chronological order.

In the tests with the STConvS2S baseline architecture, we used the best performing version reported (Nascimento et al., 2019), consisting of 3 convolutional layers on both the encoder and decoder (plus the final 1×1 convolution), each containing 32 filters of dimensionality 5×5. Similarly, the ConvLSTM baseline consists of 3 layers with 32 hidden states and filters of dimensionality 5×5.

In terms of hyper-parameter choices, in all experiments over the CHIRPS dataset, dropout is applied with a probability value of 0.8 for the ConvLSTM model, and 0.2 for the remaining models. With the CFSR dataset, dropout was not applied. Both baseline models (i.e., ConvLSTM and STConvS2S) and ST-FD use the RMSProp optimizer for training, as reported by Nascimento et al. (2019), while the other proposed extensions use the AdaMod (Ding et al., 2019) optimizer. AdaMod is a recently proposed training optimizer based on the Adam (Kingma and Ba, 2017) optimizer that restricts the adaptive learning rates through momental upper bounds, mitigating a problem exhibited by Adam where extremely large learning rates are produced at the beginning of training. All models also use either the traditional batch normalization operation together with the ReLU activation function (ConvLSTM, STConvS2S, ST-FD and ST-RFD), or the evolved normalization-activation layers explained in Section 3.2, denoted by either the B0 or B03D subscripts (STConvS2S$_{\mathrm{B0}}$, STConvS2S$_{\mathrm{B03D}}$, ST-RFD$_{\mathrm{B0}}$, ST-RFD$_{\mathrm{B03D}}$). In all scenarios, learning rate is set to 0.01 and the batch size to 25, using the root mean squared error as the loss function. An early stopping procedure is applied to all models, terminating the training when the validation loss function stops decreasing with a patience threshold of 5. Model training for the recurrent feed-backward decoding extension, in all the tests with $\Delta = 5$ (i.e., when predicting the next 5 time-steps), used the outputs from the last time-step $t - 1$ as input for the recurrent unit at the current time-step t (i.e., training did not rely on a teacher-forcing strategy).

The GridMask parameters were tuned for optimal performance in the validation sets for both CHIRPS and CFSR, varying $r \in \{0.4, 0.6, 0.8\}$ and $p \in \{0.3, 0.5, 0.7, 1.0\}$. The parameters were finally set as $r = 0.4$ and $p = 1$ (i.e., masks applied every sequence) for the CHIRPS dataset, and $r = 0.8$ and $p = 0.3$ for the CFSR dataset. The parameter d was, for both datasets, defined between 3 and 40% of the width/height of the input, resulting in $d \in [3, 20]$ for the CHIRPS dataset and $d \in [3, 12]$ for the CFSR dataset. Experiments involving the CHIRPS dataset favoured more aggressive masking strategies (i.e., the frequent removal of larger units contributed to the attenuation of model over-fitting), while tests involving the CFSR dataset favoured more subtle masking strategies.

Regarding Local Inputs and Learnable Weights, we varied the filter size k of the LWs between $1 \times 1 \times T$, $2 \times 2 \times T$ and $1 \times 1 \times 1$ (i..e, in this last case, considering a direct element-wise weighing unique to each spatial location and time-step in the input sequence), where T corresponds to the number of time-steps in the input sequence. Furthermore, we varied the number of LIs and LWs $n \in \{1, 2, 3\}$. The parameters were finally set as $k = 1 \times 1 \times 1$ and $n = 2$.

Results were measured in terms of the Root Mean Square Error (RMSE) and the Mean Absolute Error (MAE) between estimated and observed values, and also in terms of the coefficient of determination R^2. The corresponding formulas are as follows.

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \tag{4.1}$$

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n}|\hat{y}_i - y_i|}{n} \tag{4.2}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.3}$$

In Equations 4.1, 4.2 and 4.3, $\hat{y}_i$ corresponds to a predicted value, y_i corresponds to a true observed value, $\bar{y}$ is the mean of the observed values, and n is the number of predictions (i.e., we sum across all cells in the raster representations for the regions under analysis). While the MAE gives the same weight to all errors, the RMSE penalizes models with a higher variance, as it gives errors with larger absolute values more weight than errors with smaller absolute values. The coefficient of determination R^2 measures the proportion of total variation in the observations that is explained by the model, assigning a value of one to a model whose predictions exactly match the observed values, a value of zero to a model that always predicts the average value, and a negative value to a model worse than the baseline corresponding to the average.

4.2 Forecasting Future Time-Steps

The RMSE, MAE and R^2 results for each model over the test sets, for both prediction horizons, are shown in Tables 4.2 and 4.1, respectively for the CFSR and CHIRPS datasets. The results show that the proposed extensions consistently outperform the baselines. Notably, the recurrent feed-backward decoding extension (ST-RFD) outperforms the ConvLSTM and STConvS2S models in every setting, with the most significant performance gains occurring when considering a prediction horizon of 1. This further attests to the recurrent component's

	CHIRPS					
	$\Delta = \mathbf{1}$			$\Delta = \mathbf{5}$		
	RMSE	MAE	R^2	RMSE	MAE	R^2
ConvLSTM	6.4999	2.4065	0.1341	6.3874	2.3694	0.1628
STConvS2S	6.3769	2.3443	0.1701	6.3311	2.3421	0.1800
STConvS2S$_{B0}$	6.3301	2.3544	0.1759	6.3255	2.3419	0.1745
STConvS2S$_{B03D}$	6.3319	2.3445	0.1772	6.3424	2.3517	0.1734
STConvS2S$_{B03D}$ + GM	6.3259	2.3660	0.1753	6.3335	2.3567	0.1738
STConvS2S$_{B03D}$ + GM + LILW	6.1910	2.2910	0.2080	6.2340	2.3251	0.2026
ST-FD	6.3633	2.3375	0.1740	6.3148	2.3419	0.1814
ST-RFD	5.9357	2.2346	0.2638	6.1870	2.3205	0.2120
ST-RFD + GM	5.9312	2.2305	0.2717	6.1809	2.3195	0.2136
ST-RFD$_{B0}$	5.9280	2.2269	0.2721	6.1773	2.3227	0.2076
ST-RFD$_{B03D}$	5.9205	2.2253	0.2706	6.1682	2.3182	0.2149
ST-RFD$_{B03D}$ + GM	5.9054	2.2204	0.2756	**6.1626**	2.3110	**0.2166**
ST-RFD$_{B03D}$ + GM + LILW	**5.9008**	**2.2110**	**0.2795**	6.1733	**2.3027**	0.2084

Table 4.1: Results for the different models under consideration, on the CHIRPS dataset for the tasks related to predicting future time-steps. The subscripts B0, B03D, GM respectively correspond to the EvoNormB0, EvoNormB03D and GridMask extensions and LILW indicates the use of local inputs and learnable weights together with the inputs. The bold values indicate the best performing model for each prediction horizon and evaluation metric.

ability to capture temporal features in the short-term, while simultaneously showcasing increased difficulty in maintaining temporal features in longer sequences.

Although the ST-RFD model has many more parameters in comparison to the highly efficient STConvS2S architecture (i.e., mostly due to the dimensionality of the representations passed to the LSTM layer), training can still be made easily with standard GPU hardware and with reasonably-sized mini-batches of instances. Tests with the STConvS2S, ST-FD and ST-RFD took similar amounts of time for training and inference, and significantly less than the tests with the ConvLSTM architecture.

Besides feed-backward decoding, the novel normalization functions, GridMask augmentation, and learnable local features/weights all contribute to even larger performance improvements over the baselines, particularly on the CHIRPS dataset and with some mixed results on the CFSR dataset. Changing the normalization operations to either EvoNorm B0 or EvoNorm B03D results in considerable performance gains in the CHIRPS dataset, while still being a competitive option in the CFSR dataset. In almost all scenarios, EvoNorm B03D outperforms the

	CFSR					
	$\boldsymbol{\Delta = 1}$			$\boldsymbol{\Delta = 5}$		
	RMSE	MAE	R^2	RMSE	MAE	R^2
ConvLSTM	2.4206	1.6836	0.9060	2.2369	1.5171	0.9217
STConvS2S	1.4172	0.9904	0.9684	1.5896	1.0491	0.9603
STConvS2S$_{\mathrm{B0}}$	1.4183	0.9762	0.9683	1.5141	1.0543	0.9639
STConvS2S$_{\mathrm{B03D}}$	1.3846	0.9520	0.9699	1.4603	1.0289	0.9665
STConvS2S$_{\mathrm{B03D}}$ + GM	1.3915	0.9802	0.9696	1.4626	1.0073	0.9664
STConvS2S$_{\mathrm{B03D}}$ + GM + LILW	1.1940	0.7923	0.9779	1.4221	0.9730	0.9683
ST-FD	1.3742	1.0080	0.9703	1.5652	1.1130	0.9615
ST-RFD	1.0994	0.7893	0.9815	1.4877	1.0301	0.9652
ST-RFD + GM	1.0964	0.7631	0.9816	1.4710	1.0146	0.9661
ST-RFD$_{\mathrm{B0}}$	1.0837	0.7809	0.9820	1.4708	1.0259	0.9660
ST-RFD$_{\mathrm{B03D}}$	1.0461	0.7278	0.9824	1.4582	1.0279	0.9666
ST-RFD$_{\mathrm{B03D}}$ + GM	**1.0431**	0.7262	**0.9833**	1.4734	1.0275	0.9658
ST-RFD$_{\mathrm{B03D}}$ + GM + LILW	1.0792	**0.7228**	0.9820	**1.4216**	**0.9676**	**0.9684**

Table 4.2: Results for the different models under consideration, on the CFSR dataset for the tasks related to predicting future time-steps. The subscripts B0, B03D, GM respectively correspond to the EvoNormB0, EvoNormB03D and GridMask extensions and LILW indicates the use of local inputs and learnable weights together with the inputs. The bold values indicate the best performing model for each prediction horizon and evaluation metric.

original EvoNorm B0. Using the GridMask data augmentation technique together with learnable local features/weights further improves the results, achieving the best performance in most settings. Most notably, extending the STConvS2S architecture with these techniques, particularly with learnable local inputs/weights, severely outperforms the base model, achieving results comparable to the best reported version of ST-RFD on the 5-step ahead forecast.

Figure 4.1 provides an illustration for the results from the best reported model on average (ST-RFD$_{\mathrm{B03D}}$ + GM + LILW), over both the CHIRPS (top) and CFSR (bottom) datasets. A consistent colour scale was used in all the images from each dataset. The six images shown in each row correspond to the last 3 time-steps used to inform the prediction of the next time-step, together with (a) the ground-truth patches for the next time-step, (b) the predicted patches, and (c) the absolute error between the ground-truth and the predictions (shown in shades of red). The images further attest to the proposed model's ability of making accurate predictions, although also exhibiting some over-smoothing issues. It is interesting to notice that result quality when predicting the next 5 time-steps is generally worse than when predicting the immediate next time-step, although not by much – see again Tables 4.1 and 4.2. Particularly on the

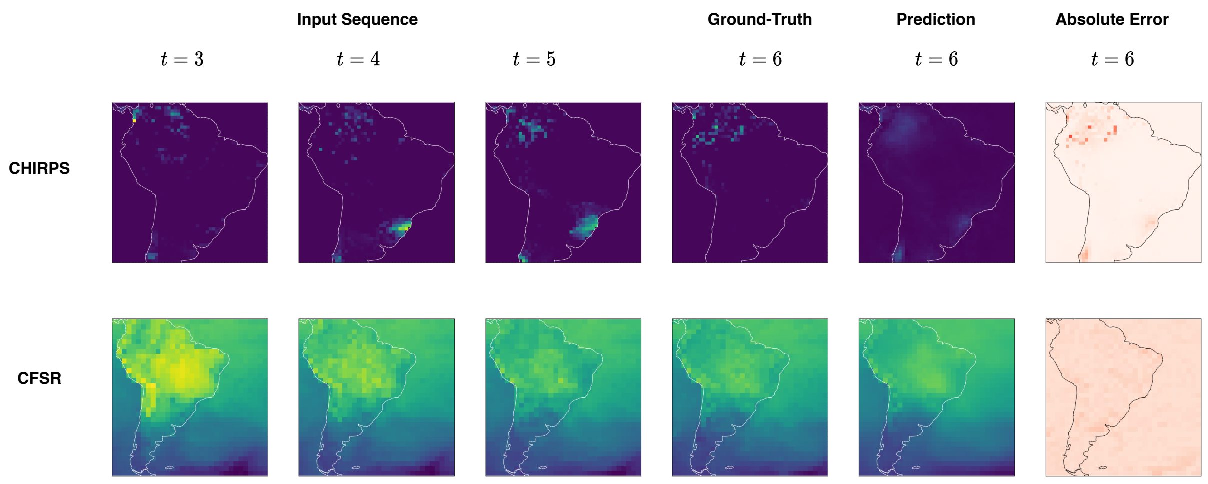

Figure 4.1: Illustration of the results obtained with the ST-RFD$_{\text{B03D}}$ + GM + LILW model over the two datasets, considering a prediction horizon of one.

CHIRPS dataset, and more evidently with the STConvS2S model, the errors concentrate on the cells that are originally associated to more extreme values (and thus the impact of the errors on the averaged metrics is less severe when we consider a larger number of time-steps).

Figure 4.2 presents scatter plots with residuals on the y axis and fitted values (i.e., predictions) on the x axis, for 3 distinct models (i.e., STConvS2S, ST-RFD, and ST-RFD$_{\text{B03D}}$ + GM + LILW) and on the scenario of one time-step ahead forecasts over both datasets. On the CFSR dataset, the residuals roughly form a horizontal band around the value of zero in the y axis, although some outliers are visible and the spread of the residuals is slighly increasing as the fitted values change (i.e., we see some heteroskedasticity problems, slighly worse on the case of the STConvS2S model). On the CHIRPS dataset, the residuals are close to zero when the fitted value is small, and increasingly more negative when the fitted value is large, with some outliers also clearly visible. The spread of the residuals is approximately constant, but the conditional mean is not, showing that the models tend to predict higher values when compared to the true observations. The patterns on the fitted versus residual plots show that there is still room for improvement in the forecasting models.

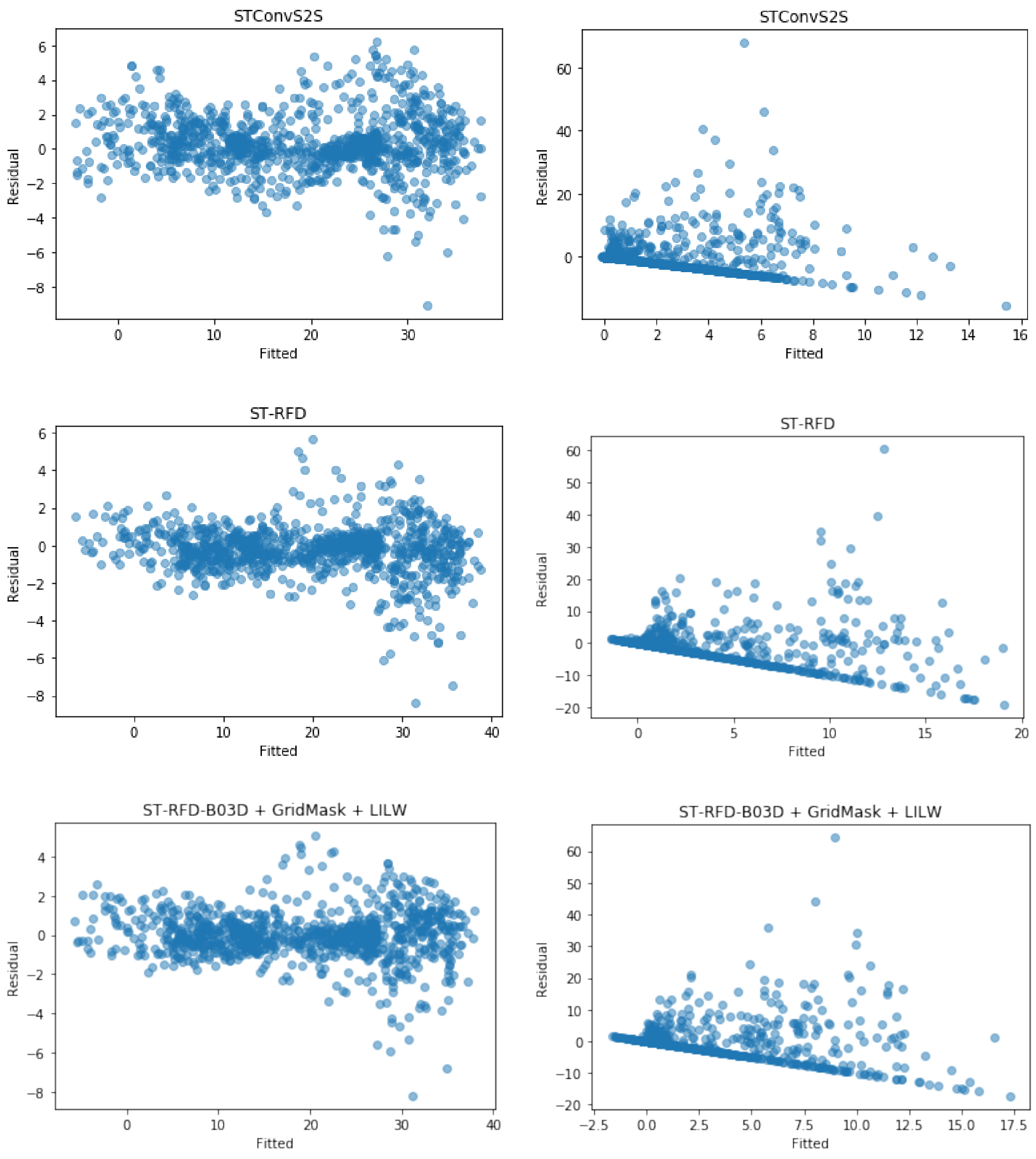

Figure 4.2: Analysis of fitted versus residual values for the STConvS2S (top), ST-RFD (middle), and ST-RFD$_{B03D}$+GM+LILW (bottom) models, for one time-step ahead forecasts on the CFSR (left) and CHIRPS (right) datasets.

4.3 Missing Data Completion

For this scenario, input sequences of 10 time-steps were created by concatenating the 5 input time-steps and the 5 ground-truth time-steps from the forecasting task detailed in the previous section, followed by the removal of a random time-step from the sequence. The architectures integrating a recurrent component are extended to consider a bi-directional LSTM, allowing for the representations at each time-step to use both future and past information. All the ST-RFD variations, as well as the ConvLSTM baseline, involve bi-directional LSTM units, with the resulting representation at each step being defined as the mean between the forward and backward states. The 1D convolutions in the decoder from the STConvS2S architecture are also allowed to use both future and past information (i.e., causal convolutions were not used in this scenario).

Two different loss function configurations were tested, varying how the model is penalized. In a first configuration, denoted as Loss_1, the model is penalized exclusively based on the missing time-step. In the second configuration, denoted as Loss_2, the model is penalized based on the entire output sequence of 10 time-steps, with the loss value associated with the missing time-step being weighed by 0.9 (i.e., the missing time-step loss corresponds to 90% of the total sequence loss), and the remaining 9 time-step predictions being weighed by $\frac{0.1}{9}$ each. This second configuration explores the intuition that reconstructing the entire sequence of co-related time-steps, based on the input representations, can be beneficial for more accurate missing data completion.

The RMSE, MAE and R^2 results for each model over the test sets, for both loss configurations, are shown in Table 4.4 and Table 4.3 for the CFSR and CHIRPS datasets, respectively. Regardless of the loss configuration being used, the reported values are calculated entirely with respect to the missing time-step. As opposed to the forecasting scenario, the ST-RFD model and extensions do not provide significant improvements over the baselines. Instead, the ST-RFD models achieve worse results on average than both baselines in both datasets. The ST-FD extension, however, achieves comparable results to the STConvS2S baseline, outperforming it in the experiments over the CHIRPS dataset. For this reason, I focused on improving the results of the ST-FD extension with the proposed optimizations, as opposed to the ST-RFD extension. I hypothesize that the worse results with ST-RFD are due to the fact that the output at each step, in a bi-directional recurrent component, is calculated based on the entire sequence, whereas in a convolutional component each step is only affected by its closest neighbours, depending on the filter size (i.e., with a filter size of 5, each step is affected by the two closest past and fu-

	CHIRPS					
	Loss_1			**Loss_2**		
	RMSE	MAE	R^2	RMSE	MAE	R^2
ConvLSTM	5.6809	2.4968	0.3144	5.6558	2.4678	0.3206
STConvS2S	5.7150	2.4975	0.3044	5.8041	2.5865	0.2779
$\text{STConvS2S}_{\text{B0}}$	5.6855	2.4495	0.3126	5.7019	2.4788	0.3102
$\text{STConvS2S}_{\text{B03D}}$	5.6767	2.4466	0.3172	5.6955	2.4573	0.3119
$\text{STConvS2S}_{\text{B03D}}$ + GM	5.7110	2.5115	0.3074	5.7296	2.5358	0.3030
$\text{STConvS2S}_{\text{B03D}}$ + GM + LILW	5.6538	2.4750	0.3208	5.6733	2.4853	0.3151
ST-RFD	5.8307	2.5168	0.2698	5.8213	2.4687	0.2708
ST-FD	5.6727	2.4610	0.3122	5.8032	2.6677	0.2808
ST-FD + GM	5.6855	2.4208	0.3147	5.7293	2.4915	0.3032
ST-FD_{B0}	5.6508	2.4442	0.3205	5.6602	2.4659	0.3159
$\text{ST-FD}_{\text{B03D}}$	5.6393	**2.3909**	0.3256	5.6598	2.4437	0.3184
$\text{ST-FD}_{\text{B03D}}$ + GM	5.6891	2.5147	0.3106	5.7177	2.5503	0.3047
$\text{ST-FD}_{\text{B03D}}$ + GM + LILW	**5.6111**	2.4214	**0.3279**	5.6170	2.4245	0.3271

Table 4.3: Results for the different models under consideration, on the CHIRPS dataset for the tasks related to reconstructing a missing time-step. The subscripts B0, B03D, GM respectively correspond to the EvoNormB0, EvoNormB03D and GridMask extensions and LILW indicates the use of local inputs and learnable weights together with the inputs. The bold values indicate the best performing model for each evaluation metric.

ture steps). This property is especially well-suited to this scenario, since the missing values are typically much more co-related with the closest future/past observations, and the co-relations become progressively less relevant the farther an observation is. The superior results achieved by ST-FD compared to ST-RFD further corroborate this hypothesis.

Changing the normalization and activation operations in ST-FD and STConvS2S to EvoNormB0 or the proposed EvoNormB03D, further improves the results in every setting, with the latter always outperforming the former. Using the GridMask augmentation technique leads to mixed results, and is generally considered detrimental. This could be because this reconstruction task appears to be easier (i.e., error values are consistently lower than in the future forecasting scenario) and, as such, the use of techniques to combat overfitting may not be the most adequate, particularly for models with a highly optimized and reduced parameter set such as fully convolutional models. Conversely, the use of learnable local features/weights significantly improves the results for all baselines, frequently resulting in the best performing architectures for both datasets.

	CFSR					
	Loss_1			**Loss_2**		
	RMSE	MAE	R^2	RMSE	MAE	R^2
ConvLSTM	1.0465	0.7716	0.9822	0.9240	0.6814	0.9861
STConvS2S	0.8578	0.6054	0.9879	0.8757	0.6202	0.9874
$\text{STConvS2S}_{\text{B0}}$	0.8293	0.5752	0.9888	0.8142	0.5612	0.9892
$\text{STConvS2S}_{\text{B03D}}$	0.7865	0.5354	0.9899	0.7903	0.5410	0.9898
$\text{STConvS2S}_{\text{B03D}}$ + GM	0.7938	0.5376	0.9897	0.8454	0.5828	0.9884
$\text{STConvS2S}_{\text{B03D}}$ + GM + LILW	**0.7616**	**0.5144**	**0.9906**	0.7800	0.5315	0.9901
ST-RFD	0.9671	0.6922	0.9849	0.9528	0.6793	0.9853
ST-FD	0.8922	0.6405	0.9870	0.9071	0.6467	0.9867
ST-FD + GM	0.8676	0.6093	0.9878	0.9355	0.6842	0.9849
ST-FD_{B0}	0.8481	0.5952	0.9883	0.9110	0.6549	0.9864
$\text{ST-FD}_{\text{B03D}}$	0.7948	0.5461	0.9897	0.8225	0.5717	0.9890
$\text{ST-FD}_{\text{B03D}}$ + GM	0.8056	0.5563	0.9894	0.8482	0.5850	0.9883
$\text{ST-FD}_{\text{B03D}}$ + GM + LILW	0.7800	0.5343	0.9901	0.7861	0.5409	0.9899

Table 4.4: Results for the different models under consideration, on the CFSR dataset for the tasks related to reconstructing a missing time-step. The subscripts B0, B03D, GM respectively correspond to the EvoNormB0, EvoNormB03D and GridMask extensions and LILW indicates the use of local inputs and learnable weights together with the inputs. The bold values indicate the best performing model for each evaluation metric.

The bi-directional ConvLSTM baseline performs exceptionally well in experiments over the CHIRPS dataset, outperforming almost every other model, with results deteriorating when considering experiments on the CFSR dataset. In every setting, the models containing a recurrent component (i.e., ConvLSTM and ST-RFD) improve results when considering the second loss configuration, while the results for fully convolutional models (i.e., STConvS2S, ST-FD) worsen.

Finally, Figure 4.3 provides illustrations for a time-step reconstruction for the ConvLSTM (left), STConvS2S (middle) and $\text{ST-FD}_{\text{B03D}}$ + GM + LILW (right) models, on both the CHIRPS (top) and CFSR (bottom) datasets. A consistent colour scale was used in all the images from each dataset and model. For each dataset, the eight images shown in each row correspond to the ground-truth missing time-step, followed by the respective model prediction and absolute error between the two (shown in shades of red), for each model. Analyzing the images, we can verify that all models are capable of generating accurate reconstructions, although once again exhibiting some over-smoothing. For both datasets, and particularly on the CHIRPS dataset,

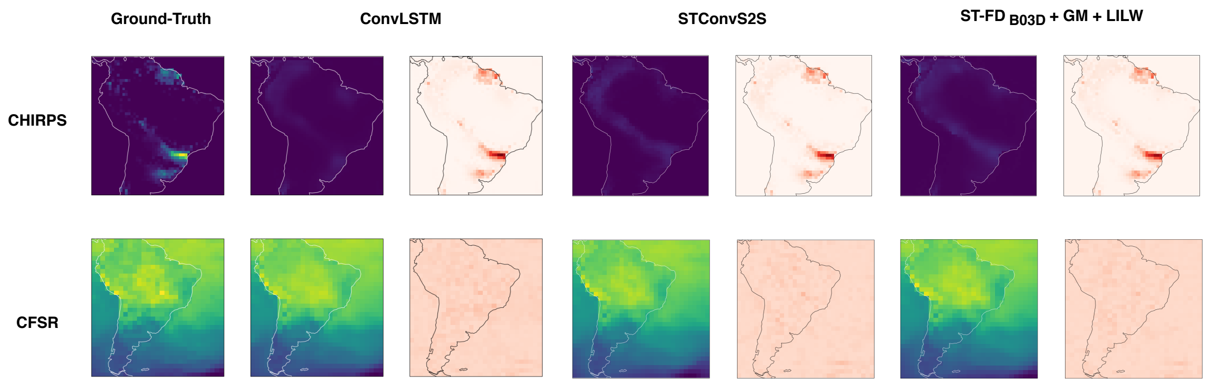

Figure 4.3: Illustration of the results obtained with the ConvLSTM, STConvS2S, and ST-FD$_{\text{B03D}}$ + GM + LILW models over the two datasets, considering the second loss configuration.

the errors on all predictions are concentrated on regions associated with extreme values.

5 Conclusions and Future Work

This dissertation presented novel deep learning approaches for spatio-temporal data forecasting, specifically on tasks pertaining to forecasting future time-steps and completing missing data. This chapter overviews the main contributions, and highlights possible directions for future work, specifically regarding the task of forecasting spatio-temporal data.

5.1 Overview on the Contributions

The most important contributions of my M.Sc. dissertation are as follows:

- **A novel hybrid deep learning architecture**: Inspired by the STConvS2S architecture (Nascimento et al., 2019), I proposed a novel architecture that leverages the feed-backward decoding (Wang et al., 2019) technique, allowing the encoder to simultaneously be used to decode, with convolutional weights shared between both phases through a permutation of dimensions. With this technique, the network concurrently learns to capture meaningful spatial features, as well as reverse the intermediate representations to obtain the final predictions. Furthermore, instead of performing convolutions along the time dimension to extract temporal patterns, I employ a recurrent component (i.e., an LSTM), which is specifically designed for processing sequential data, in order to extract temporal patterns.
- **Extension of several recently proposed techniques**: Besides the novel hybrid model, I apply several ancillary techniques derived from recent proposals in the literature, in order to further improve results. The first of these techniques is an extension of the best reported evolving normalization-activation layer (Liu et al., 2020), which takes into account the time dimension, calculating the variances with respect to each time-step. The second technique employed was an extension to the GridMask (Chen et al., 2020) data augmentation technique, where the same mask is shifted random distances both vertically and horizontally between the different time-steps in a sequence. The GridMask data augmentation technique improves the model's generalization ability by reducing overfitting, and as such, causes the model to learn more descriminative representations. The final

technique applied was the use of learnable inputs and local weights (Uselis et al., 2020) in conjunction with the inputs prior to each convolutional layer. This technique complements the global location-invariant patterns learned by the network with local features intrinsic to each specific location.

- **A detailed evaluation of the procedure**: I conducted detailed experiments on two different tasks and datasets, using the proposed extensions alongside previous state-of-the-art baselines ConvLSTM (Shi et al., 2015) and STConvS2S (Nascimento et al., 2019). Results were reported through three different evaluation metrics, and further elucidated through the use of fitted vs residuals scatter plots (Figure 4.2) and forecasting examples (Figures 4.1 and 4.3). On the first task the proposed novel architecture outperformed all other baselines, while on the second task the baselines outperformed the novel proposal. In both tasks, the use of the aforementioned ancillary techniques significantly improved both ST-RFD and STConvS2S.

5.2 Future Work

There are many ideas for future work to improve forecasting quality. For instance, the use of deeper convolution blocks could be explored, e.g. using recently proposed enhanced pooling methods (Hou et al., 2020; Deng et al., 2019; Wei et al., 2019) or using squeeze-and-excitation optimizations (Roy et al., 2018) to extract more informative spatial features. Another interesting direction could be to experiment with different missing data completion tasks, such as the prediction of missing values associated with partial regions due to cloud coverage (Gerber et al., 2018; Zhang et al., 2018; Siabi et al., 2020).

As previously mentioned, the proposed architecture uses a recurrent component comprised of either a single LSTM or bi-directional LSTM to capture temporal dependencies. Another possible direction for future work would involve experimenting with different recurrent architectures, as the component is only limited to sequence-to-sequence recurrent architectures. In particular, the recurrent component could be replaced with recurrent architectures optimized for maintaining information across long sequences of input (Voelker et al., 2019; Meng et al., 2019; Xue et al., 2019). Higher-order generalizations of recurrent components (i.e., components which explicitly incorporate more previous states in each update) is another possible beneficial direction to explore, enabling parameter updates to be influenced by a longer history, thus facilitating the capture of long-range temporal dependencies. These interactions lead to a significant increase in the complexity of the transition function, although some solutions based on tensor decomposition have shown promising results (Yu et al., 2019; Su et al., 2020).

It is interesting to notice that convolution operations are equivariant to translations by design (i.e., translating an input x and convolving with a filter k yields the same result as translating the feature map resulting from a convolution between x and k), but are not equivariant to other transformations. Previous studies have proposed group convolutions as extensions to traditional convolution operations (Cohen and Welling, 2016; Romero et al., 2020), designed to achieve equivariance to other types of affine transformations (e.g., rotation, reflection, or scaling) and compositions of affine transformations (e.g., roto-translation). Similar ideias can also be used as extentions to the proposed model, under the intuition that equivariance to rotations or changes in the scale of the patterns are particularly interesting when processing gridded remote sensing data.

Bibliography

Alléon, A., Jauvion, G., Quennehen, B., and Lissmyr, D. (2020). PlumeNet: Large-scale air quality forecasting using a convolutional LSTM network.

Chen, P., Liu, S., Zhao, H., and Jia, J. (2020). Gridmask data augmentation. *arXiv preprint arXiv:2001.04086*.

Cohen, T. S. and Welling, M. (2016). Group equivariant convolutional networks.

Das, M. and Ghosh, S. K. (2016). Deep-STEP: A deep learning approach for spatiotemporal prediction of remote sensing data. *IEEE Geoscience and Remote Sensing Letters*, 13(12).

Das, M., Pratama, M., Ashfahani, A., and Samanta, S. (2019). FERNN: A fast and evolving recurrent neural network model for streaming data classification. In *Proceedings of the International Joint Conference on Neural Networks*.

Das, M., Pratama, M., and Ghosh, S. K. (2020). SARDINE: A self-adaptive recurrent deep incremental network model for spatio-temporal prediction of remote sensing data. *ACM Transactions on Spatial Algorithms and Systems*, 6(3).

Deng, L. and Yu, D. (2014). Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4).

Deng, X., Zhu, Y., Tian, Y., and Newsam, S. (2019). Generalizing deep models for overhead image segmentation through Getis-Ord Gi* pooling. *arXiv:1912.10667*.

Ding, J., Ren, X., Luo, R., and Sun, X. (2019). An adaptive and momental bound method for stochastic learning.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

Frías-Blanco, I., del Campo-Ávila, J., Ramos-Jimenez, G., Morales-Bueno, R., Ortiz-Díaz, A., and Caballero-Mota, Y. (2014). Online and non-parametric drift detection methods based on Hoeffding's bounds. *IEEE Transactions on Knowledge and Data Engineering*, 27(3).

Funk, C., Peterson, P., Landsfeld, M., Pedreros, D., Verdin, J., Rowland, J., Romero, B., Husak, G., Michaelsen, J., and Verdin, A. (2014). A quasi-global precipitation time series for drought monitoring. *U.S. Geological Survey*, Data Series 832.

Gao, H., Yuan, H., Wang, Z., and Ji, S. (2019). Pixel transposed convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

Gerber, F., de Jong, R., Schaepman, M. E., Schaepman-Strub, G., and Furrer, R. (2018). Predicting missing values in spatio-temporal remote sensing data. *IEEE Transactions on Geoscience and Remote Sensing*, 56(5).

Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing.* Morgan & Claypool Publishers.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8).

Hong, S., Kim, S., Joh, M., and kwang Song, S. (2017). PSIque: Next sequence prediction of satellite images using a convolutional sequence-to-sequence network.

Hou, Q., Zhang, L., Cheng, M.-M., and Feng, J. (2020). Strip pooling: Rethinking spatial pooling for scene parsing.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167.*

Jang, Y., Kim, G., and Song, Y. (2018). Video prediction with appearance and motion conditions.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient based learning applied to document recognition. *Proceedings of the IEEE, 86(11).*

Liu, H., Brock, A., Simonyan, K., and Le, Q. V. (2020). Evolving normalization-activation layers.

Meng, F., Zhang, J., Liu, Y., and Zhou, J. (2019). Multi-zone unit for recurrent neural networks.

Misra, D. (2019). Mish: A self regularized non-monotonic neural activation function. *arXiv:1908.08681.*

Nascimento, R. C., Souto, Y. M., Ogasawara, E., Porto, F., and Bezerra, E. (2019). STConvS2S: Spatiotemporal convolutional sequence to sequence network for weather forecasting. *arXiv:1912.00134*.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). WaveNet: A generative model for raw audio.

Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Searching for activation functions. *arXiv:1710.05941*.

Romero, D. W., Bekkers, E. J., Tomczak, J. M., and Hoogendoorn, M. (2020). Attentive group equivariant convolutional networks.

Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. Technical Report 85-460-1, Cornell Aeronautical Laboratory.

Roy, A. G., Navab, N., and Wachinger, C. (2018). Recalibrating fully convolutional networks with spatial and channel "squeeze & excitation" blocks. *IEEE Transactions on Medical Imaging*, 38(2).

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv:1609.04747*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. MIT Press.

Saha, S., Moorthi, S., Wu, X., Wang, J., Nadiga, S., Tripp, P., Behringer, D., Hou, Y.-T., Chuang, H.-y., Iredell, M., Ek, M., Meng, J., Yang, R., Mendez, M. P., van den Dool, H., Zhang, Q., Wang, W., Chen, M., and Becker, E. (2014). The NCEP climate forecast system version 2. *Journal of Climate*, 27(6).

Shi, X., Chen, Z., Wang, H., Yeung, D., Wong, W., and Woo, W. (2015). Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *Proceedings of the Annual Conference on Neural Information Processing Systems*.

Siabi, N., Sanaeinejad, S. H., and Ghahraman, B. (2020). Comprehensive evaluation of a spatio-temporal gap filling algorithm: Using remotely sensed precipitation, lst and et data. *Journal of Environmental Management*, 261.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Training very deep networks. *arXiv:1507.06228.*

Su, J., Byeon, W., Kossaifi, J., Huang, F., Kautz, J., and Anandkumar, A. (2020). Convolutional tensor-train lstm for spatio-temporal learning.

Uselis, A., Lukoševičius, M., and Stasytis, L. (2020). Localized convolutional neural networks for geospatial wind forecasting. *arXiv preprint arXiv:2005.05930.*

Voelker, A. R., Kajić, I., and Eliasmith, C. (2019). Legendre memory units: Continuous-time representation in recurrent neural networks. In *Proceedings of the International Conference on Neural Information Processing Systems.*

Wang, B., Glossner, J., Iancu, D., and Gaydadjiev, G. N. (2019). Feedbackward decoding for semantic segmentation. *arXiv:1908.08584.*

Wang, Y., Long, M., Wang, J., Gao, Z., and Yu, P. S. (2017). PredRNN: Recurrent neural networks for predictive learning using spatiotemporal LSTMs. *Proceedings of the Annual Conference on Neural Information Processing Systems.*

Wang, Y., Long, M., Wang, J., Gao, Z., and Yu, P. S. (2018). PredRNN++: Towards a resolution of the deep-in-time dilemma in spatiotemporal predictive learning. *arXiv:1804.06300.*

Wei, Z., Zhang, J., Liu, L., Zhu, F., Shen, F., Zhou, Y., Liu, S., Sun, Y., and Shao, L. (2019). Building detail-sensitive semantic segmentation networks with polynomial pooling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.*

Xue, L., Li, X., and Zhang, N. L. (2019). Not all attention is needed: Gated attention network for sequence data.

Yu, R., Zheng, S., Anandkumar, A., and Yue, Y. (2019). Long-term forecasting using higher order tensor rnns.

Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method.

Zhang, J., Zheng, Y., and Qi, D. (2016a). Deep spatio-temporal residual networks for citywide crowd flows prediction.

Zhang, J., Zheng, Y., Qi, D., Li, R., and Yi, X. (2016b). DNN-based prediction model for spatio-temporal data. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.*

Zhang, Q., Yuan, Q., Zeng, C., Li, X., and Wei, Y. (2018). Missing data reconstruction in remote sensing image with a unified spatial–temporal–spectral deep convolutional neural network. *IEEE Transactions on Geoscience and Remote Sensing*, 56(8).

Zhao, S., Yuan, X., Xiao, D., Zhang, J., and Li, Z. (2018). AirNet: A machine learning dataset for air quality forecasting.