FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

# Assignment of bachelor's thesis

| | |
|---|---|
| **Title:** | Precipitation nowcasting using a generative adversarial network |
| **Student:** | Matej Murín |
| **Supervisor:** | Ing. Jakub Bartel |
| **Study program:** | Informatics |
| **Branch / specialization:** | Knowledge Engineering |
| **Department:** | Department of Applied Mathematics |
| **Validity:** | until the end of summer semester 2022/2023 |

## Instructions

1) Discuss current techniques for precipitation nowcasting and properties of their predictions.
2) Introduce and describe the network proposed in [1].
3) Implement a neural network based on [1] and train it on a precipitation dataset provided by Meteopress.
4) Evaluate the predictions both quantitatively and empirically where applicable.
5) Discuss pros and cons of the created nowcasting model.

[1] Ravuri, S., Lenc, K., Willson, M. et al. Skilful precipitation nowcasting using deep generative models of radar. Nature 597, 672–677 (2021). https://doi.org/10.1038/s41586-021-03854-z

---

*Electronically approved by Ing. Karel Klouda, Ph.D. on 16 December 2021 in Prague.*

Bachelor's thesis

# PRECIPITATION NOWCASTING USING A GENERATIVE ADVERSARIAL NETWORK

**Matej Murín**

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Jakub Bartel
May 6, 2022

Czech Technical University in Prague
Faculty of Information Technology


Citation of this thesis: Murín Matej. *Precipitation nowcasting using a generative adversarial network.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

# Contents

# List of Figures

# List of Tables

*I would like to thank my supervisor and all the people at Meteopress who provided me with all the professional insights and tips that made it possible for me to work on this thesis.*

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 6, 2022 .....

# Abstrakt

Nedávne pokroky v oblasti umelej inteligencie umožnili použitie strojového učenia ako nástroja k nowcastingu – krátkodobej predpovedi zrážok. V posledných rokoch sme mohli vidieť mnoho publikácií na túto tému, keďže je to stále otvorený problém. Dva najväčšie problémy modelov v týchto publikáciách je predpoveď zrážok s vysokou intenzitou a predpoveď do dlhšej budúcnosti.

Táto práca sa zaoberá zopakovaním a evaluáciou neurónovej siete na krátkodobú predikciu zrážok založenej na adversariálnom učení publikovanom DeepMindom. Zdrojový kód tejto siete nebol publikovaný, a tak musela byť celá implementovaná. Práca popisuje akékoľvek implementačné zmeny oproti originálnej sieti, tréningový proces a problémy ktoré počas neho nastali. Taktiež uvádza novú post-processing metódu, ktorá naďalej zlepšuje kvalitu predikcií získaného modelu.

Finálny model bol evaluovaný a porovnaný s inými populárnymi metódami na krátkodobú predpoveď počasia a namerané metriky ukázali, že je s nimi zrovnateľný. Je taktiež ukázané, že predikcie získaného modelu nadobúdajú realistickejšieho vzhľadu. Je to vďaka tomu, že daný model nerozmazáva svoje predikcie v takom rozmere, ako ich rozmazávajú iné metódy.



**Klíčová slova** strojové učenie, hlboké učenie, predpoveď počasia, DeepMind, Meteopress, radarové snímky zrážok, generative adversarial nets



# Abstract

Recent advancements in artificial intelligence have allowed the usage of machine learning as a tool for precipitation nowcasting – a short-term precipitation forecasting. There have been numerous publications in recent years, as this is still an open problem. Two biggest issues of these models is the prediction of high precipitation events and accurate prediction for longer lead times.

This thesis focuses on recreating and evaluating a precipitation nowcasting neural network based on the adversarial approach published by Deepmind. The source code for this network has not been published, so it had to be implemented from scratch. The thesis describes any adjustments made to the original network that needed to be done, the training process and any issues arisen during it. It also introduces a post-processing method used to further improve the predictions's quality of the obtained model.

The final model has been evaluated and compared to other popular nowcasting methods and it was able to produce predictions of comparable quality metric-wise. It is also showcased how the predictions made by this model are finer and more realistic, as they do not blur their predictions nearly as much as other methods.



**Keywords** machine learning, deep learning, weather nowcasting, DeepMind, Meteopress, weather radar images, generative adversarial nets

# Introduction

Precipitation nowcasting is an open problem that concerns itself with forecasting of precipitation in the near future. The distinction between nowcasting and forecasting is not set in stone, but most definitions agree that nowcasting is a prediction of up to 2 hours into the future. The main difficulty in attempting to do these predictions lies in the unpredictability of immediate weather development. Because of this, most methods tend to blur their predictions as a result of attempting to decrease the expected difference between their predictions and reality.

The quality of predictions of these models does not only serve to help people make decisions in their daily activities, but also to warn them of any extreme weather events that may potentially have large socio-economic impact on their lives. Thus, any warning systems can make use of these models to increase their efficacy.

This work will focus on implementation and adjustments of a neural network designed and published by DeepMind that tackles this problem probabilistically via a generative adversarial network coined *DGMR*. Its predictions are expected to be of high quality not only quantitatively, but also adhere to the structure of precipitation events as they appear on images produced by weather radars.

The theoretical part of this thesis will focus on describing the precipitation nowcasting problem in more detail. It will introduce several methods of model quality measurements. Various techniques used for solving the problem will be introduced, both traditional and machine learning ones, alongside the explanation of their respective shortcomings. It will acquaint the reader with the basics of neural networks and then describe the architecture of DGMR in detail.

In the practical part, the thesis will focus on training and adjusting the implemented DGMR to obtain a model that is able to make predictions of desired qualities. It will describe various issues that arose during training and the measures taken to solve them. A post-processing method will be introduced that will prove to increase the quality of the obtained model's predictions. The thesis will then evaluate the model's predictions using various metrics against two other popular methods used for precipitation nowcasting.

# Thesis's objective

The main goal of this thesis is to implement a neural network for precipitation nowcasting published by DeepMind called Deep Generative Model of Radar (*DGMR*). Because the source code of this network had not been published, this thesis will attempt to replicate their work by implementing, training, adjusting if necessary and evaluating the neural network they proposed on a precipitation dataset provided by Meteopress.

The very beginning of the thesis will be an introduction about weather nowcasting. It will also be established what are the current popular methods of model evaluation. After this, various currently used traditional and machine learning models to make these predictions will be introduced. Their detailed description will not be included, as that is not the focus of this thesis. Towards the end of this chapter, a high level overview of DGMR will be explained.

In the next part, all the necessary concepts of neural networks and various layer types will be established. When this will be done, a detailed description of DGMR will follow. All its various components and modules will be talked about, how they function, and what is their purpose in the architecture as a whole. Right after this, a brief introduction of a framework used to implement this model will follow. Finally, any adjustments to the original model's architecture and hyperparameters will be described, alongside the reasoning behind them.

The next chapter will talk about the training process and all the alterations of DGMR that have been explored. When a model which upholds the most to the expected outcome will be obtained, a simple post-processing method will be introduced that further increases the quality of the predictions.

In the final part of the thesis, this obtained model's predictions are to be compared to predictions of several different methods, as introduced in the beginning of this thesis. The model's various pros and cons will be briefly described and some possible adjustments that the thesis did not attempt to cover are to be explored.

# Chapter 1

# Nowcasting Introduction

Weather has impacted human lives ever since the beginning of time. Starting from mundane things like deciding what to wear, to more impactful ones like dictating harvest quality or potentially damaging property or even taking lives. While these still hold true to this day, with the start of the industrial revolution, it has had major impact on things like traffic, be it road or air, or energy-generating production (wind, solar, etc.). Being able to predict weather development in the immediate future, which is called nowcasting, and to be able to warn of any upcoming dangerous events is therefore becoming more crucial than ever, especially with predictions being that extreme precipitation events are becoming more frequent with current climate change [1]. While systems that predict weather development worldwide, like Global Ensemble Weather System (GEFS) [2] perform well on a global scale, their resolution and computational cost make it unfeasible for making local nowcasting predictions. Most organizations have therefore turned to various different methods of radar-based nowcasting.

## 1.1 Weather radars

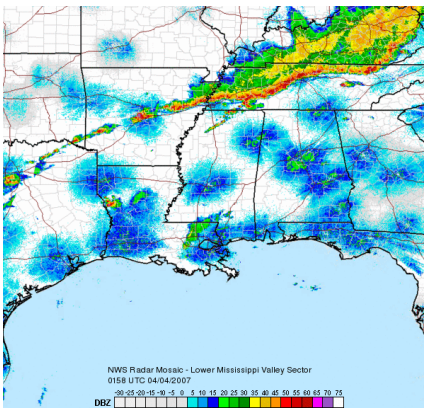


**Figure 1.1** Radar view of southern United States [3].

Weather radars are a fundamental part of precipitation nowcasting. They function by emitting an electromagnetic (EM) radiowave in a direction specified by the position of the antenna. When this wave hits an obstacle, like a plane – or a raindrop – some of the energy is scattered away from this obstacle. While most of it is lost to the radar, a very small portion is also reflected

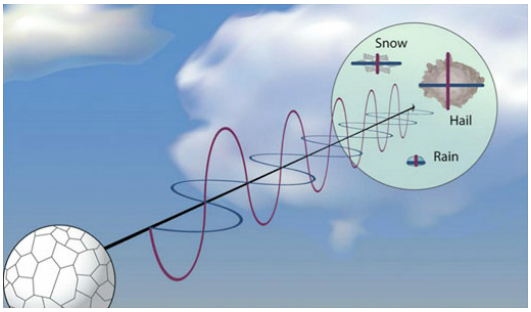


**Figure 1.2** Dual polarization in radar [6].

directly back into it. The radar antenna receives this energy and processes it [4]. The electromagnetic radiation that radars emit have historically had unidirectional polarization, either vertical or horizontal. Modern radars, however, are polarimetric, meaning they can determine the response of a collision with an obstacle using two orthogonal polarizations. This allows for better understanding of what the signal collided with and therefore the shape of the object can be better understood, be it a rain drop, snowflake or hail particle. This information is crucial for meteorologists to better understand what is happening in the clouds and helps them give out warnings if need be [5].

The precipitation captured by radars is measured in the strength of the reflectivity of the collided object, namely decibels of reflectivity (dBZ), which serves as a measure of intensity of precipitation. It is a logarithmic, dimensionless technical unit. Together with knowledge that the signal emitted by radar travels at the speed of light and the radars current direction, it can be calculated what kind of object was hit, how far it is and where it is located spatially. To be able to sample the entire atmosphere or a specific region, the radar antenna is rotated and signal is emitted and then received at each configured point [3].

## 1.2 Model evaluation

Currently, there are various methods used for making nowcasting predictions from weather radars, but first let's establish popular methods of their evaluation, since this is shown to be a difficult task. These are called *metrics*, and the goal of models should be to either minimize or maximize their value. Even though both ground-truth $y$, the real radar image and the respective prediction $\hat{y}$ usually form a $3^{\text{rd}}$ order tensor specifying $[T \times H \times W]$ time step, height, and width of precipitation field, respectively, for reasons of evaluation they usually will be flattened to a vector of length $[T \cdot H \cdot W]$.

### 1.2.1 L1 loss

Also known as mean absolute error (MAE), this simple loss function is a popular choice for measuring image to image similarity, like precipitation nowcasting [7, 8]. The function measures the average absolute difference between all components of their inputs. It has negative orientation, meaning the lower the values the better. Consider comparing two vectors of length $n$. The loss is calculated as [9]:

$$L(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i| \tag{1.1}$$

### 1.2.2 MSE loss

*Mean squared error* is another popular choice for comparing predictions with observations for regression tasks [10, 11]. Also known as $L2$ loss. In the context of image2image comparisons, minimizing this loss can lead to less blurry predictions [12]. Again, consider comparison of two vectors of length $n$. The squared loss is calculated as [9]:

$$L(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{1.2}$$

This loss also has negative orientation.

### 1.2.3 Critical success index

Also called a **threat score** or abbreviated to CSI, is a verification method of categorical forecast with a pre-specified threshold. It is calculated as total number of **hits** (TP) divided by a sum of TP, **false alarms** (FP) and **misses** (FN). Evidently this is a function mapping into a range of $[0, 1]$, with positive orientation, meaning the higher the value the better. This metric can be seen used in may publications to measure a model's quality [7, 13, 11, 8]. The formula is therefore given as [14]:

$$CSI = \frac{TP}{TP + FP + FN} \tag{1.3}$$

## 1.3 Current methods of nowcasting

It has been shown that weather is a phenomena that is unpredictable by nature [15], and therefore the best way to account for this unpredictability is with probability. Models that make various number of predictions that are plausible are called **ensemble** models. As for actual models used to make nowcasts, they can be distinguished by two main categories, traditional and machine learning (ML) ones.

### 1.3.1 Traditional nowcasting models

Two most prominent types of models used traditionally are *optical flow* models and *numerical* models.

Optical flow models are based on the "apparent motion of brightness pattern" [16]. These models work by calculating motion vectors of an input sequence of observations and perform predictions by moving the precipitation field in the direction obtained by these vectors. Various calculation methods of motion vectors, together with added perturbations to inputs make these effectively ensemble-type models. It is important to note that these extrapolation methods are based on the assumption of *Lagrangian persistence* of precipitation. This form of persistence states that the intensity and volume of rainfall does not change over time, but simply just moves in a certain direction. Optical flow methods, therefore, can not account for phenomena such as precipitation initiation, growth, decay and termination [17]. However, even with such a strong false assumption, they are still capable of making predictions of good quality and nowadays serve as a strong benchmark to compare against other methods, such as machine learning ones [7, 13, 18]. The implementation of the optical flow method for precipitation nowcasting that will be used in this thesis is called **PySTEPS** [17], a python-based implementation of various optical flow algorithms.

The other traditional method used to make weather forecasts are numerical weather predictions models, such as Global Ensemble Forecast System [2]. While these numerical models that

predict weather worldwide work well for what their purpose is, that is prediction for an entire day (currently making predictions 4 times a day), their computation takes a long time and lacks both temporal and spatial resolution needed to make short-term nowcasting. Furthermore, they need a certain spin-up time until their predictions reach the quality they are capable of. It is because of these qualities that they do not serve well for a task such as nowcasting, where forecasts are to be done and updated several times per hour.

### 1.3.2 Machine learning models

Machine learning (ML) models for short-term weather prediction based on radar imagery has become a popular research field in recent years. The benefits of using ML models for nowcasting is their ability to – at least theoretically – model events already mentioned, like precipitation initiation, growth, decay and termination.

One of the first attempts to use machine learning for radar-based precipitation nowcasting had been attempted by Agrawal et al. [18] in 2019. They leveraged the power of the ubiquitous **UNet** [19], a fully convolutional neural network based on an encoder-decoder architecture and residual connections. They had approached the problem as three-leveled binary classification, where for each pixel, the network predicted whether precipitation would reach a given threshold. They predicted a single lead time of 60 minutes into the future. Ayzel et al. [7] proposed another approach in 2020 based on the UNet architecture called RainNet. Their network was, once again, designed to make a single prediction. However, they had used it recursively with inputs being outputs of previous predictions to make a nowcast of up to 60 minutes with temporal resolution of 5 minutes. Fernandéz and Mehrkanoon proposed another modification to the original UNet architecture, called Broad-UNet [10]. They tried using asymmetric parallel convolutions and a module called Astrous Spatial Pyramid Pooling, a special convolutional block that uses dilated kernels to extract spatial information. They had found that using such architecture improves upon the traditional UNet. However, their approach was based on satellite imagery, rather than rader-based observations.

A slightly different approach was taken by Wang et al. [11] in early 2021. They had introduced a novel recurrent neural network framework specifically designed to learn both short- and long-term dependencies from the input sequence called Predictive Recurrent Neural Network (PredRNN). They had shown that this approach is able to produce competitive precipitation nowcasts. Tuyen et al. [8] furthermore expanded on their idea in early 2022 by combining the power of UNet and PredRNN, achieving similar results with much shorter training time.

All of the ML solutions discussed so far, except [18], have been in essence defined as regression tasks. Whenever there is uncertainty, these regressive models tend to express it with blurriness. One possible solution to counter this issue could be producing an ensemble of probable predictions by using a generative adversarial network (GAN), rather than making a single prediction. This is exactly what Zheng et al. attempted in their work in 2021 [20]. They had argued that the predictions their model made did have more detail to them than those non-generative ones. However, they also did note that the model struggled to predict high-accuracy events and also deviated too much from the actual observations location-wise. Finally, in mid 2021, Ravuri, Lenc and other researchers working for DeepMind introduced a GAN [13] that aims to produce both location accurate and realistic looking predictions by having a loss function that is specifically designed to account for spatial consistency, temporal consistency and pixel-wise accuracy. Furthermore, they had designed it to focus more on high-precipitation events, rather than on all events equally. This exact network is the one that this thesis will attempt to replicate by implementing, adjusting and training it on a dataset provided by Meteopress.

# Chapter 2
# Neural Networks

In this chapter, all the concepts of neural networks – or NNs in short – necessary for understanding the thesis's model architecture, *DGMR*, will be explained. The reader may feel free to skip this chapter if they are already acquainted with these concepts.

## 2.1 Training a neural network

Before we dive into various popular layer types used in neural networks, let's go over the process of training a neural network. We need two main ingredients, in addition to the neural network itself, to be able to train it. First is a dataset we will use to train our network. The second thing is an *objective function*, which is a function that measures "how well" our network performs, and is *the* function that the network will attempt to minimize. The most popular type of neural networks are so-called feed-forward neural networks [21], where the nodes do not form a cycle. To train these networks, data is passed through them in a *forward pass*, outputs are compared with a desired outcome using the aforementioned objective function, and the gradient for all parameters is calculated. Afterwards, a *backward-pass* is executed, where we update the weights in opposite direction of the gradient, since the goal is to minimize an objective function. Consider weights parameter $\theta$ and an objective function $f$. Passing the entire dataset through the network and calculating the gradient for weights, the following equation describes how it is updated at iteration $i$ [23]:

$$\theta_{i+1} = \theta_i - \lambda \cdot \nabla_{\theta_i} f(\theta_i) \tag{2.1}$$

where $\lambda$ is called a *learning rate*, a hyperparameter that describes how "fast" the network learns. This algorithm is called vanilla gradient-descent [22]. It is also common to split the dataset into minibatches, which is often a measure for dataset being too large to fit into memory. Doing this not only fixes this issue, but also has a regularization effect [23], meaning the network should theoretically work better even for unseen data after training. This changes gradient descent into mini-batch gradient descent [22], the most popular method used nowadays to train neural networks.

## 2.2 Objective function

Also known as the *loss function*, this is a function which measures how well the network is doing. It outputs a single scalar value for its inputs, whatever they may be. The network's goal is, therefore, to maximize (or minimize) the output of its function, often called a **loss**. For regression

tasks it can simply be one of the metrics described in 1.2, or anything else that you want the network to optimize for.

## 2.3 Single-layer perceptron

Neural network models are inspired by neurons in the human brain. They activate for some inputs, and stay dormant on others. The simplest form of a neural network is called single-layer perceptron [23], which is a neural network consisting of a single artificial neuron $\xi$. The input to this neuron is a vector $x$. All of its components are assigned weight by a weight vector $w$, and usually we also include a bias $b$, a term that dictates the output of the neuron if the input is a zero vector. The following image illustrates this concept

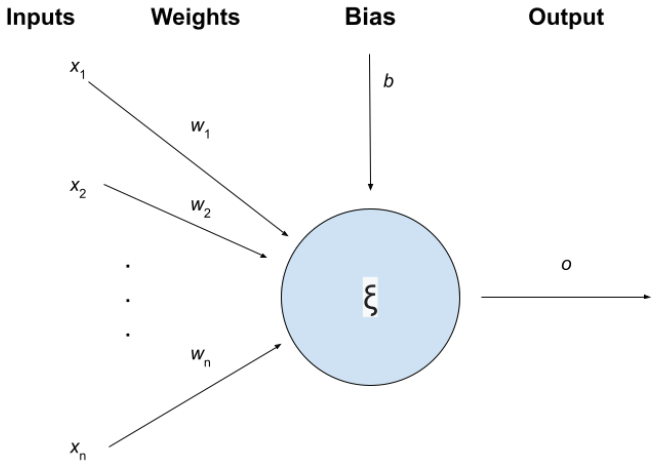


**Figure 2.1** Single-layer perceptron.

where mathematically, the output $o$ is calculated as follows:

$$o = \xi(x) = \left( \sum_{i=1}^{N} w_i \cdot x_i \right) + b = \mathbf{w}^\intercal \mathbf{x} + b. \tag{2.2}$$

We can see that single-layer perceptron is simply a linear combination of inputs added together with a bias. This, of course, implies linearity, something that might not always be true for the data we are working with and therefore something we might want to avoid. This will be solved with concepts we introduce next, mainly multi-layer perceptron and activation functions. Some like to include activation functions as a natural part of a single-layer perceptron, however because of how they are implemented in most popular neural network frameworks nowadays, they have been omitted from the definition in here.

## 2.4 Multi-layer perceptron

One way to overcome the linearity problem in single-layer perceptron is a method of stacking multiple single-layer perceptrons on top of each other, both vertically and horizontally, creating fully connected layers [23]. By this we create a so called Multi-layer perceptron, or MLP in short. In this sense, we are incorporating *hidden layers* into neural networks. They are called hidden because their outputs are never actually observed. What can only be seen are the inputs and outputs of the MLP itself. These stacked and connected layers are often called **linear**, **dense**, or **fully connected (FC)** layers.

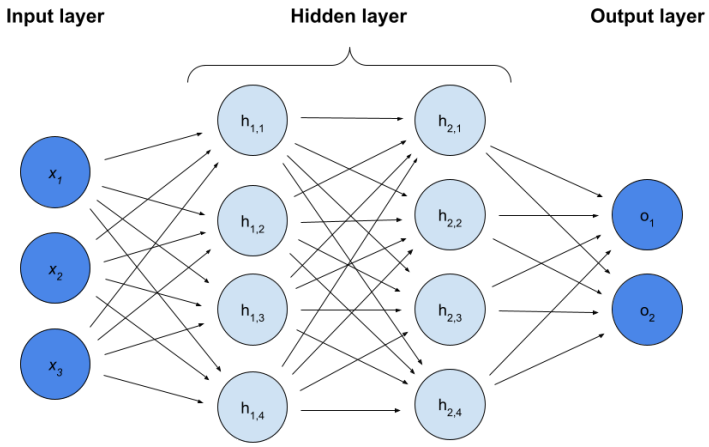


**Figure 2.2** Example of multi-layer perceptron.

However, there is one issue with using only perceptron stacking. In reality, the calculation can be transformed into a single linear combination of inputs with the following equations describing individual outputs of hidden layers, as well as the final output layer with input being $\mathbf{x} \in \mathbf{R^{1\times 3}}$:

$$\begin{aligned} \mathbf{H_1} &= \mathbf{xW_1} + \mathbf{b_1} \\ \mathbf{H_2} &= \mathbf{H_1W_2} + \mathbf{b_2} \\ \mathbf{O} &= \mathbf{H_2W_3} + \mathbf{b_3} \end{aligned} \tag{2.3}$$

Where $\mathbf{W_1} \in \mathbf{R^{3\times 4}}$, $\mathbf{W_2} \in \mathbf{R^{4\times 4}}$ and $\mathbf{W_3} \in \mathbf{R^{4\times 2}}$ are matrices consisting of individual weight vectors as rows for the transformations occurring within hidden layer and output layer respectively, and $\mathbf{b_1} \in \mathbf{R^{1\times 4}}$, $\mathbf{b_2} \in \mathbf{R^{1\times 4}}$, $\mathbf{b_3} \in \mathbf{R^{1\times 2}}$ are transposed vectors containing a bias term for each respective unit within a layer. Therefore, the entire hidden layer can be removed and the parameters of the output layer can be acquired as $\mathbf{W} = \mathbf{W_1W_2W_3} \in \mathbf{R^{3\times 2}}$ and $\mathbf{b} = \mathbf{b_1W_2W_3} + \mathbf{b_2W_3} + \mathbf{b_3} \in \mathbf{R^{1\times 2}}$:

$$\begin{aligned} \mathbf{O} &= \left(\left(\mathbf{xW_1} + \mathbf{b_1}\right)\mathbf{W_2} + \mathbf{b_2}\right)\mathbf{W_3} + \mathbf{b_3} \\ &= \left(\mathbf{xW_1W_2} + \mathbf{b_1W_2} + \mathbf{b_2}\right)\mathbf{W_3} + \mathbf{b_3} \\ &= \mathbf{xW_1W_2W_3} + \mathbf{b_1W_2W_3} + \mathbf{b_2W_3} + \mathbf{b_3} \\ &= \mathbf{xW} + \mathbf{b} \end{aligned} \tag{2.4}$$

To actually overcome linearity and make use of MLPs, we need to introduce activation functions into neural networks.

## 2.5 Activation functions

The purpose of an activation function is to non-linearily transform the output of a unit within a neural network. They are not always needed, for example, it is popular to omit them in the output layer. Activation functions need to be differentiable for purposes of training a neural network. However, this isn't always the case. Some of the most popular activation functions are:

- $sigmoid(x) = \sigma(x) = \frac{1}{1+e^{-x}}$. Transforms inputs into a value inside range $(0, 1)$ [24].

- $tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. Transforms inputs into a value inside range $(-1, 1)$ [24].
- $ReLU(x) = \max(0, x)$. ReLU stands for Rectified Linear Unit [24]. It can be seen that it is not differentiable at 0. This is solved by taking the left-hand-side derivative instead [23].

By using an activation function $f$ on an output of a neuron, we get a new transformation

$$\xi(x) = f\left(\mathbf{w}^\intercal \mathbf{x} + b\right) \tag{2.5}$$

which is no longer linear. If we use these functions on outputs of units inside MLP, the transformation can no longer be collapsed as before in equation 2.4.

## 2.6 Convolutional neural networks

Using MLPs is very powerful, but they have one notable flaw. Their inputs are just vectors. Sometimes we are working with data that has some spatial structure as well, e.g. images. One way we could make MLPs work for spatial data is to flatten it into some long vector and go from there. However, as already stated, by doing this we lose the information of the structure of the data. If our goal is to detect some object in an image, it should not matter to the network where precisely the object is, only that it is or isn't there. This is where convolutional neural networks (CNNs) come in [25]. They introduce two qualities into the network, namely:

- translation invariance - the network layer should react to the same patch the same exact way, no matter where it appears in the image
- locality principle - the earlier layers should focus on local regions, and deeper ones on more broader parts of the image

Let us have an input as 3D tensor $A \in \mathbf{R}^{i \times j \times k}$. Next, let's assume a convolutional layer with various filters, those being the learnable parameters of the layer, making them effectively weights $V$. The following operation [23] takes place in the layer, giving an output $H$ as:

$$H_{i,j,d} = \sum_{a=-\delta}^{\delta} \sum_{b=-\delta}^{\delta} \sum_{c} V_{a,b,c,d} \cdot A_{i+a,j+b,c} \tag{2.6}$$

For some pre-specified vicinity $\delta$. To support multiple output channels, the $d$ dimension had been added to weights parameter $V$, making it a fourth-order tensor, and the output $H$ a third-order tensor. An optional bias term can be added as a learnable parameters as well. It should be noted that the word *convolution* is a misnomer in this context. Technically, the operation described in equation 2.6 is a *cross-correlation*, not a convolution. The following image illustrates this operation done on a 2D tensor:

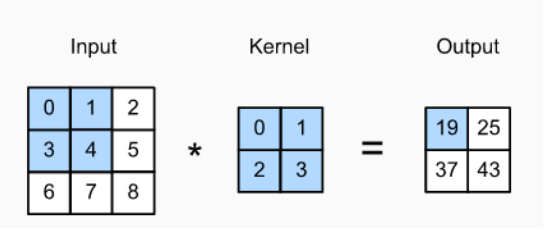


**Figure 2.3** Two-dimensional cross correlation operation [23].

It is also common to **pad** the input tensor around the edges with some value, the most popular one being zero, to achieve a desired output shape. Furthermore, we might also want to choose a different **stride** for the filter. The following two images illustrate padding and different stride, respectively.

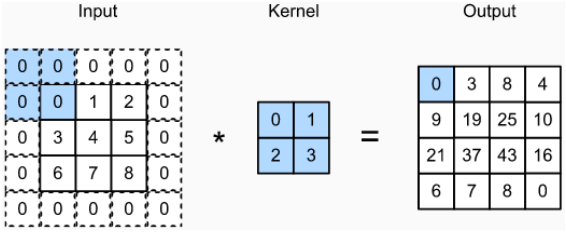


**Figure 2.4** Two-dimensional cross-correlation with padding [23].

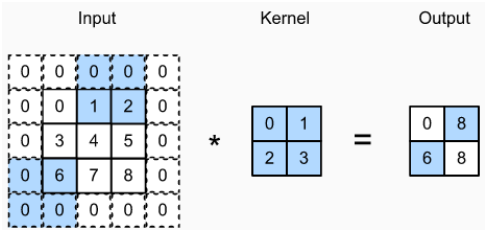


**Figure 2.5** Cross-correlation with strides of 3 and 2 for height and width, respectively [23].

### 2.6.1 Downsampling layers

After convolutions – or cross correlations – are performed, we might want to aggregate some values that are close to each other. This will give us some leeway for any perturbations of the input. This is where **pooling**, or pooling layers come in. They can either be pre-defined, with the most popular ones [26] being **average** or **max** pooling, where they output the average or maximum of a specific region

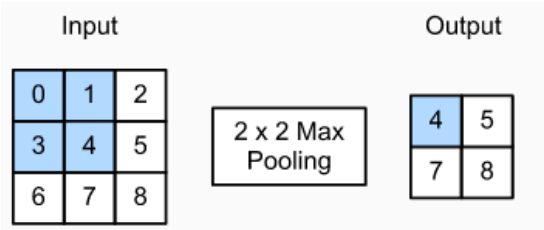


**Figure 2.6** Max pooling operation illustration [23].

or learned, when they are de facto convolutional layers with 0 padding and stride equal to the kernel (filter) size in all respective dimensions.

### 2.6.2 Upsampling layers

As opposed to layers described in section 2.6.1, the goal of these layers is to perform the opposite, that is to scale-up its inputs. This layer has no learnable parameters, and the most popular [27, 28] methods of upsampling include linear, bilinear, bicubic or nearest-neighbour interpolation.

## 2.7 Batch normalization

When the networks are trained using minibatch gradient descent described at the beginning of this chapter, the inputs after minibatch split might have different distributions. This forces the

network to constantly adapt to different data. Batch normalization layer [29], technically batch *standardization* layer, assigns a Z score to its inputs, using either current or running first two moments (mean and variance). The following equation describes the transformation of an input $x$ [28]:

$$BN(x) = \frac{x - E[X]}{\sqrt{Var[X] + \epsilon}} \cdot \gamma + \beta \tag{2.7}$$

where $\gamma$ and $\beta$ are learnable parameters, often set initially [28] to $\gamma = 1$ and $\beta = 0$ and $\epsilon$ is there to prevent zero-devision issues in computation. In the original paper [29], it had been shown that using this method speeds up the training process by allowing for more aggressive learning rates by reducing the internal covariate shift (ICS) of the data. However, other publications [30] state that this is not the case, and the reason why batch normalization speeds up training and improves the performance of the network is mainly by smoothing the optimization landscape, having the effect of more predictive and stable gradients during training.

## 2.8 Recurrent neural networks

Consider a case where we might want to predict some time series data. Given past $n-1$ observations $x_{t-n}, ..., x_{t-1}$ of variable $x$, the conditional probability of observation at time step $t$ is given by $P(x_t|x_{t-n}, ..., x_{t-1})$. However, instead of calculating this probability, modern neural networks estimate it via [23]:

$$P(x_t|x_{t-n}, ..., x_{t-1}) \approx P(x_t|h_{t-1}) \tag{2.8}$$

where $h$ is called a *hidden state*. The idea behind is that $h$ in some form encapsulates the past observations into a single variable. These networks are called recurrent neural networks (RNNs). Consider an input $\mathbf{X_t} \in \mathbf{R}^{1\times d}$ and a hidden state $\mathbf{H_{t-1}} \in \mathbf{R}^{1\times h}$. The parameters of a recurrent cell are $\mathbf{W_x} \in \mathbb{R}^{\mathbf{d}\times\mathbf{h}}$, $\mathbf{W_h} \in \mathbf{R}^{\mathbf{h}\times\mathbf{h}}$ and an optional bias term $\mathbf{b_h} \in \mathbf{R}^{1\times h}$. The following operation describes the computation of the next hidden state $\mathbf{H_t}$:

$$\mathbf{H_t} = f(\mathbf{X_t}\mathbf{W_x} + \mathbf{H_t}\mathbf{W_h} + \mathbf{b_h}) \tag{2.9}$$

where $f$ is some activation function applied to each dimension of output individually. The computation at each time step is done using the same parameters and is therefore *recurrent*, and is where the name comes from. It should be emphasized that the output of a recurrent cell is still computed as a *hidden layer* of a network, and is therefore usually passed into some additional layers, and eventually the output layer.

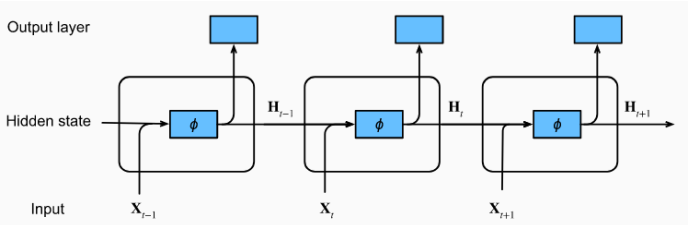


**Figure 2.7** Recurrent neural network example [23].

### 2.8.1 Gated recurrent units

First proposed in 2014 by Chung et al. [31], this variation of recurrent units adds a gating mechanism to the hidden state, dictating when it should be *updated* or even *reset*. These operations are learned, and thus new parameters are introduced. Again, consider [23] an input $\mathbf{X_t} \in \mathbf{R}^{1\times d}$ and a hidden state $\mathbf{H_{t-1}} \in \mathbf{R}^{1\times h}$. There are two gates in the gated recurrent unit (GRU). The reset gate $R$ with parameters $\mathbf{W_{xr}} \in \mathbf{R}^{d\times h}$, $\mathbf{W_{hr}} \in \mathbf{R}^{h\times h}$ and a bias term $\mathbf{b_r} \in \mathbf{R}^{1\times h}$. The update gate $Z$ analogous parameters $\mathbf{W_{xz}}$, $\mathbf{W_{hz}}$ and $\mathbf{b_z}$. The following equation describes their output [23]:

$$\begin{aligned}\mathbf{R_t} &= \sigma(\mathbf{X}\mathbf{W_{xr}} + \mathbf{H_{t-1}}\mathbf{W_{hr}} + \mathbf{b_r}) \\ \mathbf{Z_t} &= \sigma(\mathbf{X}\mathbf{W_{xz}} + \mathbf{H_{t-1}}\mathbf{W_{hz}} + \mathbf{b_z})\end{aligned} \tag{2.10}$$

To allow the reset gate to dictate how much of the previous hidden state needs to be reset based on current input, a *candidate hidden state* $\tilde{\mathbf{H}}_\mathbf{t}$ is calculated [23]:

$$\tilde{\mathbf{H}}_\mathbf{t} = tanh(\mathbf{X_t}\mathbf{W_{xh}} + (\mathbf{R_t} \odot \mathbf{H_{t-1}})\mathbf{W_{hh}} + \mathbf{b_h}) \tag{2.11}$$

where $\mathbf{W_x h}$, $\mathbf{W_{hh}}$ and $\mathbf{b_h}$ are of same dimensions as they were in 2.9 and $\odot$ being element-wise (Hadamard) product operator. This state is still only a candidate, and we need to use the update gate now. The following describes the final hidden state output of a GRU [23]:

$$\mathbf{H_t} = \mathbf{Z_t} \odot \mathbf{H_{t-1}} + (1 - \mathbf{Z_t}) \odot \tilde{\mathbf{H}}_\mathbf{t} \tag{2.12}$$

This simply dictates how much of the candidate hidden state is to be used, and how much of the previous hidden state. A convolutional variant coined ConvGRU was introduced and published by Ballas et al. in 2015 [32], which simply uses convolution instead of fully-connected operation within itself.

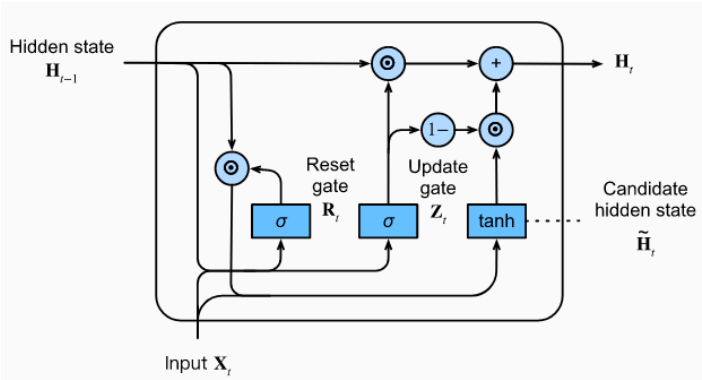


**Figure 2.8** Hidden state calculation in GRU. [23].

## 2.9 Generative adversarial networks

Sometimes our goal is to estimate the data distribution rather than solving some regression task. A method first introduced by Goodfellow et al. in 2014 [33] attempts to do exactly this, coined generative adversarial networks (GANs), where a generator $G$ and a discriminator $D$ are trained simultaneously in an adversarial manner. The generator can be thought of as a counterfeiter, attempting to fool the discriminator into thinking its output came from the real data distribution. The goal of the discriminator, on the other hand, is to be able to distinguish the fake data from

the real one. In this sense, the discriminator guides the generator in the learning process. In a case where both the $D$ and $G$ are differentiable functions, e.g. MLP, CNN or RNN, the training can be done purely by back-propagation and minibatch gradient descent. The process can be described in the following four steps:

1. get a minibatch of real data $X_{real}$
2. generate a minibatch of fake data $X_{fake}$ by $G$ from some noise $p_z$
3. pass concatenated $X_{real}$ and $X_{fake}$ through $D$ with labels being "real" and "fake" respectively, compute gradient for $D$ and update the parameters of $D$
4. pass $X_{fake}$ through $D$ with labels being "real", compute gradient for $G$ and update parameters of $G$

A *conditional* GAN [34] is a variation of generative adversarial networks where in addition to noise input $p_z$, there is also some condition to which we want to generate data. An example would be a generator for generating a dog or cat picture, and the condition being a boolean variable for specifying which one is to be generated.

### 2.9.1 Spectral normalization

A spectral normalization [35] is a technique that modifies the weights of its input to control the Lipschitz constant by constraining the spectral norm of the layer. This has proven to increase the stability of training the discriminator in GANs. Given weight matrix $W$, spectral normalization is defined as [35]:

$$\tilde{W}_{SN}(W) \coloneqq \frac{W}{\sigma(W)} \tag{2.13}$$

where $\sigma(W)$ refers to spectral ($L_2$) norm of the matrix $W$. If the input is a weight tensor of order higher than 2, it is simply reshaped into a 2D tensor (a matrix). However, if applied to a first order tensor (a vector), the following operation takes place instead [28]:

$$\tilde{\mathbf{x}}_{SN}(\mathbf{x}) \coloneqq \frac{\mathbf{x}}{||\mathbf{x}||_2} \tag{2.14}$$

# Chapter 3

# Deep Generative Model of Radar

This chapter will go over the architecture details of **DGMR**, a deep generative adversarial neural network model designed by DeepMind [13] to tackle the precipitation nowcasting problem, and the model that this thesis will try to reproduce.

Their philosophy of going for an adversarial neural network as their architecture of choice is to model precipitation development probabilistically. Given past radar observations and a random vector as an input to this network, it attempts to predict further development with accounted uncertainty not by blurring the predictions, but rather by producing various realistic predictions that differ thanks to the random vector provided.

Because this chapter's sole purpose is to acquaint the reader with the architecture of DGMR, the contents of this chapter will therefore be based on their publication from 2021 [13].

## 3.1 Overview

DGMR is based on the architecture of a conditional generative adversarial network. Therefore, its inputs are $\mathbf{M}$ past radar observations and a latent vector $\mathbf{Z}$ whos elements are drawn from i.i.d. normal distributions $\sim N(0, 1)$. Let $\mathbf{X}$ be the observations and $\mathbf{N}$ be the number of time steps the model is to predict for. The model's goal can be thought of as modeling the following probability for any observation $\mathbf{X}$:

$$P(\mathbf{X_{M+1:M+N}}|\{\mathbf{Z}; \mathbf{X_{1:M}}\}) \tag{3.1}$$

The originally proposed DGMR has four past radar observations with temporal resolution of 5 minutes and each with spatial resolution of $[256 \times 256]$. The input themselves are in $mm \cdot h^{-1}$ units. The network then makes predictions for 18 leading time steps, making it a prediction of up to 90 minutes into the future.

As with most GANs, the networks consists of two main modules, a *generator* and a *discriminator*. Before going into describing these two modules, it will be useful to first describe the various block types used within them.

### 3.1.1 D and 3D Block

A downsampling residual block (DBlock) that decreases the resolution and increases the number of channels, both by a factor of two. A residual connection is used as well, which is a method of adding the input, in some way, to the output of some convolutional operation. Specifically in *DBlock*, before this summation is done, the number of channels of the input is optionally decreased using $(1 \times 1)$ convolution.

### 3.1.2 LBlock

A variation of *DBlock*, this block is designed to increase the number of channels in its input by a factor of two. Again, it uses a residual connection.

### 3.1.3 GBlock

A counterpart to *DBlock*. This is where the spatial resolution of its input is doubled using the nearest neighbor interpolation using an upsampling layer. The number of input channels is halved, on the other hand.

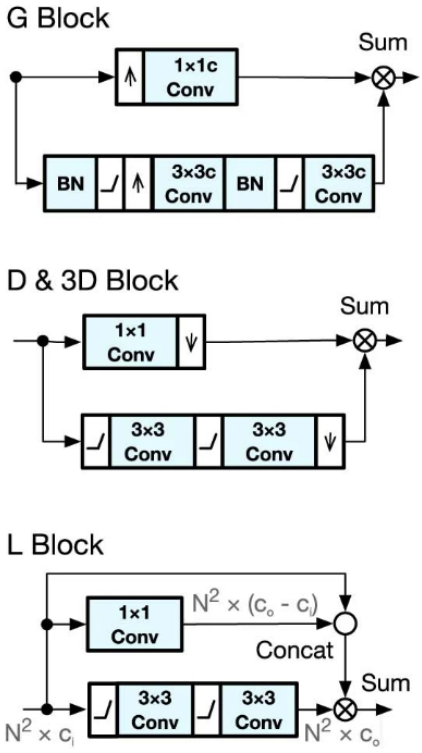


**Figure 3.1** G, D and L block respectively, where (↑) is for upsampling and (↓) for downsampling. _/ signifies a ReLU operation [13].

## 3.2 Generator

The generator uses two modules to make predictions. The first one is *conditioning stack* that processes past observations and creates conditioning representations of various resolutions that will afterwards be fed into the *sampler*, which is the second module of the generator. The sampler then generates predictions based on these conditioning representations and on some random noise.

### 3.2.1 Conditioning stack

The conditioning stack is a convolutional neural network that creates a conditioning representation of past four observations, each being a tensor of shape $[1 \times 256 \times 256]$. They are first

transformed into $[4 \times 128 \times 128]$ shape by stacking $(2 \times 2)$ patches into channels, a so-called space-to-depth (S2D) operation. Afterwards, each time frame is processed separately by the same layer, since they are all the same radar observation data. Four downsampling residual blocks (*DBlocks*) are used to decrease the resolution of the input and increase its number of channels both by a factor of 2. For each input of *DBlock*, a $(3 \times 3)$ spectrally normalized convolution is applied to reduce the number of channels also by a factor of 2. A rectified linear unit follows this operation. The output of a conditioning stack are conditional representations of input observations of shapes $[48 \times 64 \times 64]$, $[96 \times 32 \times 32]$, $[192 \times 16 \times 16]$ and $[384 \times 8 \times 8]$.

### 3.2.2 Sampler

The sampler is formed by a stack of four ConvGRU [32] units, each using different output of the conditioning stack as the initial hidden state if its unit. Inputs to the lowest ConvGRU are copies of a $[768 \times 8 \times 8]$ latent representation of a random vector $Z$. This representation is created by latent conditioning stack, a small neural network that transforms the latent vector into desired output shape.

This latent conditioning stack transforms a $[8 \times 8 \times 8]$ input drawn from i.i.d. normal distribution $N(0, 1)$. It is built up of one $(3 \times 3)$ convolution, three L blocks, a *spatial attention module* and one final L block. The attention module serves as a self-regularization layer and has shown to increase the performance of GANs [36].

The output of each ConvGRU is upsampled by a spectrally normalized $(1 \times 1)$ convolution, followed by two GBlocks. The output of the last ConvGRU is a vector of shape $[48 \times 128 \times 128]$. It is then transformed by a batch normalization layer, a ReLU and another $(1 \times 1)$ spectrally normalized convolution, yielding a $[4 \times 128 \times 128]$ representation. This representation is then transformed into the final $[1 \times 256 \times 256]$ representation by a depth-to-space operation (D2S), an inverse operation of S2D.

## 3.3 Discriminator

The discriminator itself consists of two neural networks. The first one is there to ensure spatial consistency of predictions, punishing blurry or inconsistent predictions. It is therefore coined *spatial discriminator*. The second, named *temporal discriminator*, punishes jumpy predictions and thus ensures temporal consistency between subsequent time frames. They both share similar structure, but temporal discriminator uses a 3D convolution in its beginning to account for the temporal dimension of its inputs.

### 3.3.1 Spatial discriminator

This discriminator picks randomly, uniformly 8 out of 18 lead time frames of the predictions and then processes them. It had been designed this way to fit the model within memory. They are subsequently downsampled to $[1 \times 128 \times 128]$ representations using mean (average) pooling layer and this is followed by a S2D operation that transforms them into $[4 \times 64 \times 64]$ representations again by stacking $(2 \times 2)$ patches into the channel layer, similar to 3.2.1. Next come five DBlocks, each halving the resolution and increasing the number of channels by a factor of two. The first DBlock does not apply its first ReLU operation, as its input is still merely the radar observation itself. The output shape of each DBlock is $[48 \times 32 \times 32]$, $[96 \times 16 \times 16]$, $[192 \times 8 \times 8]$, $[384 \times 4 \times 4]$ and $[768 \times 2 \times 2]$, respectively. After being processed by the last DBlock that preserves both the resolution and the number of channels, the representations are sum-pooled across the width and height dimensions. These final 8 representations are then inputs to a spectrally normalized linear layer, after which they are again summed together before a final ReLU is applied for classification.

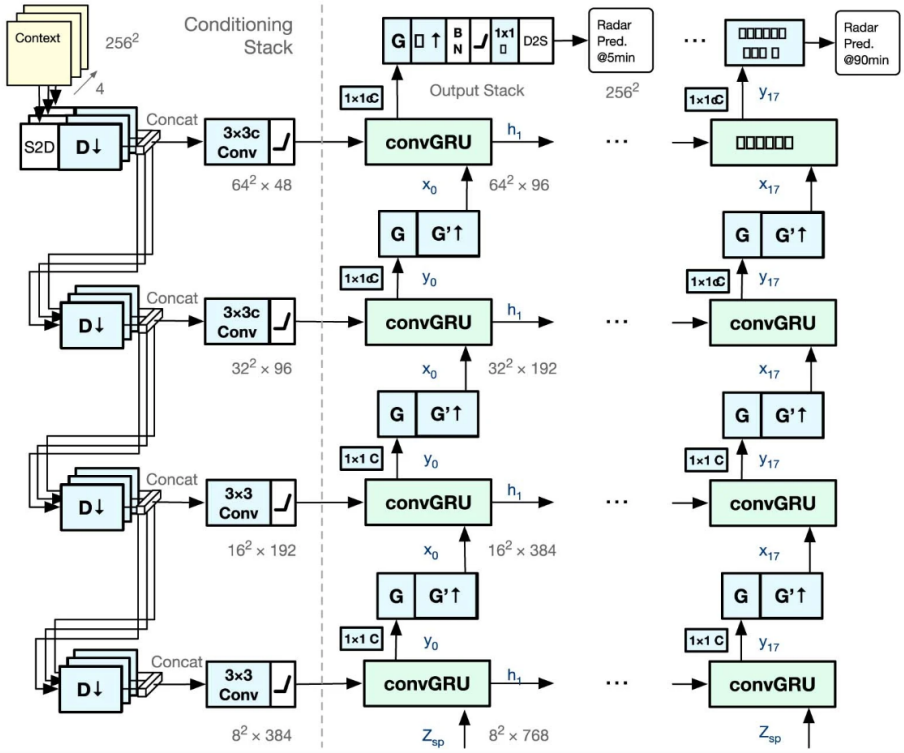


**Figure 3.2** Generator architecture used in DGMR. The conditioning stack (left) and the sampler (right) are separated by a dashed line. [13].

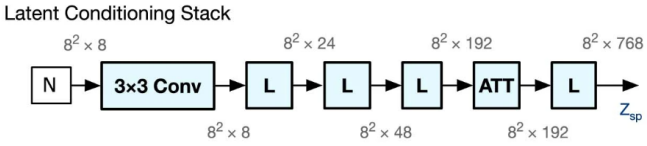


**Figure 3.3** Latent conditioning stack architecture used in the sampler of DGMR [13].

### 3.3.2 Temporal discriminator

The architecture of the temporal discriminator is very similar to the spatial discriminator. To fit the model within memory, random crops of shape $[128 \times 128]$ are extracted from the original $[256 \times 256]$ frames and are furthermore downsampled to $[4 \times 64 \times 64]$ representations using S2D operation. These are subsequently processed by two DBlocks using $(3 \times 3 \times 3)$ spectrally normalized convolutions, which mimic the first DBlocks used in spatial discriminator. After this, the architecture follows the same design as 3.3.1.

## 3.4 Loss function

As with any neural network, DGMR also needs a loss function to optimize for. For DGMR specifically, the generator's loss function is made up of two terms. The losses of the two discriminators and a *grid cell regularizer* denoted $L_R(\theta)$.

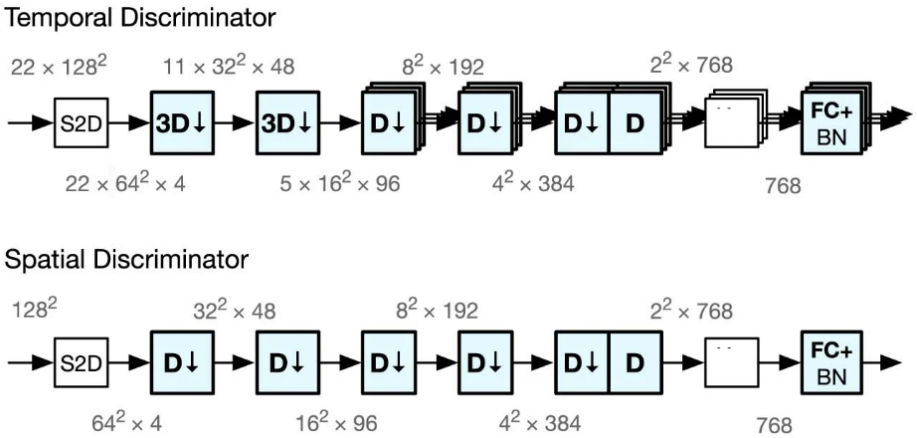


**Figure 3.4** Discriminator architectures used in DGMR. It can be seen that this diagram shows a batch normalization layer after final fully-connected layer for each discriminator. This was, however, dropped in the final implementation of DGMR [13].

### 3.4.1 Generator loss

Let spatial discriminator $D_\alpha$ have parameters $\alpha$ and temporal discriminator $T_\beta$ have parameters $\beta$ and generator $G_\gamma$ have parameters $\gamma$. The generator's goal for any input $X$ is to optimize the loss $L_G$ as :

$$L_G(\gamma) = \mathbb{E}_{\mathbf{X}_{1:M+N}}[\mathbb{E}_{\mathbf{Z}}[D(G_\gamma(\mathbf{Z}; \mathbf{X}_{1:M})) + T(\{\mathbf{X}_{1:M}; G_\gamma(\mathbf{Z}; \mathbf{X}_{1:M})\})] - \lambda L_R(\theta)] \tag{3.2}$$

where $\{A; B\}$ signifies concatenation of two fields. The $L_R$ term is called a *grid cell regularizer* and is calculated as:

$$(HWN)^{-1} \cdot ||(\mathbb{E}_{\mathbf{Z}}[G_\gamma(\mathbf{Z}; \mathbf{X}_{1:M})] - \mathbf{X}_{M+1:M+N}) \odot w(\mathbf{X}_{M+1:M+N})||_1 \tag{3.3}$$

where $HWN$ stands for multiplication of height, width, and number of predicted frames and $\odot$ stands for piece-wise multiplication of two matrices. The function $w$ is a function that weighs the loss towards heavier rainfall, punishing the network more for making errors on higher precipitations. Its output is calculated as:

$$w(y) = min(y + 1, 24) \tag{3.4}$$

where 24 is an empirically chosen term to prevent punishing against spurious radar observations. The $\lambda$ value in 3.2 had been set to 20, as it yielded the best results in the publication. Monte Carlo estimations (random draws from given distributions) are used in both 3.2 and 3.3 to estimate expectations over latent vector $\mathbf{Z}$. In the implementation, 6 draws had been made each time the loss had been calculated.

### 3.4.2 Discriminator loss

The training goal spatial discriminator $D_\alpha$ and temporal discriminator $T_\beta$ is to minimize their individual loss functions with respect to their parameters $\alpha$ and $\beta$, respectively. The loss function for the spatial discriminator that is minimized is:

$$L_D(\alpha) = \mathbb{E}_{\mathbf{X}_{1:M+N}, \mathbf{Z}}[ReLU(1 - D_\alpha(\mathbf{X}_{M+1:M+N})) + ReLU(1 + D_\alpha(G(\mathbf{Z}, \mathbf{X}_{1:M})))] \tag{3.5}$$

and for temporal discriminator:

$$L_T(\beta) = \mathbb{E}_{\mathbf{X}_{1:M+N},\mathbf{z}}[ReLU(1 - T_\beta(\mathbf{X}_{1:M+N})) + ReLU(1 + T_\beta(\{\mathbf{X}_{1:M}; G(\mathbf{Z}, \mathbf{X}_{1:M})\}))] \quad (3.6)$$

# Chapter 4
# Implementation

This chapter will go over the implementation of *DGMR* according to the original publication [13]. Since the original publication did not contain the code for the network, only pseudocode, it had to be implemented from scratch. This chapter will first talk about the framework chosen for implementation. The implementation itself stayed true to the pseudocode description with an exception of some adjustments. These adjustments with be the next part of this chapter, where each of them will be described, alongside the idea behind it.

## 4.1 PyTorch as neural networks framework

Most of the recent models for precipitation nowcasting, be it either using optical flow methods or neural networks, had been done using the *python* programming language [17, 37, 23, 33]. Therefore, this is the natural candidate to be used as well for the implementation of DGMR.

As for the framework used, the three most popular ones nowadays are TensorFlow [38], MXNet [39] and PyTorch [28]. As interesting as it might be to explore all of those frameworks, this thesis will be using **PyTorch** for one simple reason. The final model is being developed for Meteopress company and therefore has to fit within their respective ecosystem. As they had already been using PyTorch for all the neural network models they had developed and implemented, it is only reasonable to follow their standard. Furthermore, a wrapper for PyTorch modules and training, called PyTorch Lightning [40] will be used. It encapsulated all the boilerplate code that is common for most – if not all– neural network implementations, and therefore allows developers to focus on what actually matters to them when developing complex models.

While all those frameworks have many things which differ, they also share many similarities. Arguably, however, the most useful tool all of them provide is **automatic differentiation**. Recall that in section 2.1, it had been established that neural networks are trained via gradient descent. This means that if we are to change the architecture of a network, or alter its objective function, the respective gradient computation would be different. Doing this manually would be really inefficient and time consuming. Automatic differentiation, in principle, stores the computational graph of a variable, most often some function output. Doing this allows for traversing this graph in a reverse order and therefore allows for computation of the respective gradient using the chain rule.

## 4.2 DGMR implementation adjustments

After the architecture with all the modules having default settings and parameters has been implemented, the number of parameters was 41.9 million for generator and 32.7 million for

discriminator, making it 74.6 million in total. With the hardware available – that being two Nvidia GeForce RTX 3090 graphics cards, each having 24GB of memory – it was simply not enough to train the model the same way as it was done originally in [13]. For comparison, to fit the model within this memory limit of 48GB, it was first needed to reduce the number of lead frames from 18 to 4, and also reduce the batch size from 16 to 4. Furthermore, the training was much slower as well, since the computational power was also much lower. Because of this, it was agreed to somehow reduce the capacity of the implemented *DGMR* architecture to allow for reasonable training times. Therefore, the following adjustments had been made, allowing for changes of the model capacity just by changing the newly added hyperparameters.

### 4.2.1 Generator changes

As mentioned in section 3.2, the generator consists of a *Sampler* and a *Conditioning Stack*. The adjusted generator still uses those two modules. However, both of them had been allowed to specify their capacity as a hyperparameter.

#### 4.2.1.1 Conditioning stack changes

The conditioning stack processes the input sequence of radar images of shape $[1 \times 256 \times 256]$ first by stacking $(2 \times 2)$ patches into channels, producing a $[4 \times 128 \times 128]$ representation. This would be very difficult to change, so it was left as is. However, notice how after this operation, the observations are furthermore processed by a pattern of increasing the number of channels and reducing the resolution, both by a factor of two, producing a final conditioned representation of shapes $[48 \times 64 \times 64]$, $[96 \times 32 \times 32]$, $[192 \times 16 \times 16]$ and $[384 \times 8 \times 8]$. This had been changed by adding a hyperparameter *base_num_channels* (bnc), producing a final shape of $[(bnc \cdot 2) \times 64 \times 64]$, $[(bnc \cdot 4) \times 32 \times 32]$, $[(bnc \cdot 8) \times 16 \times 16]$ and $[(bnc \cdot 16) \times 8 \times 8]$. The idea behind this change was to both keep the overall structure and philosophy of the stack, while allowing for modular changes to be done by simply changing this hyperparameter.

#### 4.2.1.2 Sampler changes

Remember how in section 3.2.2, it was mentioned that the sampler has a latent conditioning stack, mapping an i.i.d random vector of shape $[8 \times 8 \times 8]$ with normal distribution *N(0, 1)* into a $[768 \times 8 \times 8]$ shape to be used as an input to the lower-most ConvGRU unit of the sampler. Two hyperparameters had been added, those being:

- random vector num channels - number of channels of the random vector, making it an initial shape of $[num\_channels \times 8 \times 8]$
- random vector std - the standard deviation to be used, instead of the std of 1

Because the original network had more capacity overall, these changes had been made to account for the decrease. It allows for tuning of those parameters, with the idea being lower number of channels, as well as lower variance of the vector values could lead to easier and more stable learning of the network.

Furthermore, notice how in figure 3.3, the number of channels is slowly increased by multiples of two, namely 24, 48, 192 and 768. A hyperparameter had been added, named *base_num_channels* (bnc), making the number of channels *bnc*, $bnc \cdot 2$, $bnc \cdot 8$ and $bnc \cdot 32$. This also allows for modular capacity of latent conditioning stack.

### 4.2.2 Discriminator changes

Similar changes had been done to the discriminator as they have to the generator. The main idea behind them was to allow tuning of the capacity.

As both temporal and spatial discriminator gradually decrease the resolution of their inputs and increase the number of channels, both by a factor of two, it only made sense to set the base number of channels as a hyperparameter. This again allowed for modular changes to the architecture during model training and selection.

### 4.2.3 Loss function changes

As it was described in section 3.4.1, the loss function of the generator is a combination of discriminator loss and a grid cell regularizer loss multiplied by a grid lambda $\lambda$. This grid lambda had been added as another hyperparameter to allow for tuning. However, a couple more hyperparameters had been added to the loss function, namely *grid weight divisor* and a *min_clip* value. If set to some values $\alpha$ and $\beta$ respectively, they transform the function $w$ in equation 3.3 to a new function $w_{new}$ defined as:

$$w_{new}(y) = min\left(\frac{y}{\alpha} + \beta, 24\right) \tag{4.1}$$

The *grid weight divisor* allows for modular weighing of higher precipitation levels. The higher the $\alpha$ value, the more equally are various precipitation levels weighed in the loss function. On the other hand, *min_clip* allows to modify what weight should be assigned to low precipitation data.

# Chapter 5

# Training process and model selection

This chapter will go over the training process of *DGMR*. After the changes to the implementation had been made, which allowed for reduced capacity of the model, this chapter will refer to the model as reduced-DGMR (**r-DGMR**), as referring to it with the original name could be misleading. The dataset used was provided by Meteopress. It is a dataset of precipitation above the Czech Republic obtained via OPERA programme of EUMETNET [41] with resolution of $[352 \times 544]$ that had been center-cropped to $[256 \times 256]$. The data is originally in dBZ, so it was converted to precipitation in $mm \cdot hr^{-1}$. The input data had been quantized into multiples of 1/32 values, the same way it had been in the original publication. The respective dataset split sizes for train, validation, and test subsets had been 83634 (73%), 13899 (12%) and 17016 (15%).

## 5.1 Version 1

The first version of *r-DGMR* was trained for 92796 iterations with batch size of 16, which ended up being a bit over 17 epochs. Table 5.1 shows the configuration of the model. The idea was to decrease the overall capacity of DGMR by halving the base number of channels in most modules. The random vector number of dimensions, as well as standard deviation had been reduced to allow for more stable training. However, this had not been the case. The training was unstable, which was reflected both on the loss values show in figure 5.2 and in artifacts shown in figure 5.1, which resemble an "eye-like" structure.

| **Hyperparameter** | **New value** | **Original value** |
|---|---|---|
| Output sequence length | 4 | 16 |
| Conditioning stack base num channels | 12 | 24 |
| Latent conditioning stack base num channels | 12 | 24 |
| Random vector num channels | 6 | 8 |
| Random vector std | 0.7 | 1 |
| Spatial discriminator base num channels | 24 | 48 |
| Temporal discriminator base num channels | 24 | 48 |
| Generator number of parameters | 10.5M | 41.9M |
| Discriminator number of parameters | 8.2M | 32.7M |

**Table 5.1** r-DGMR_v1 changes.

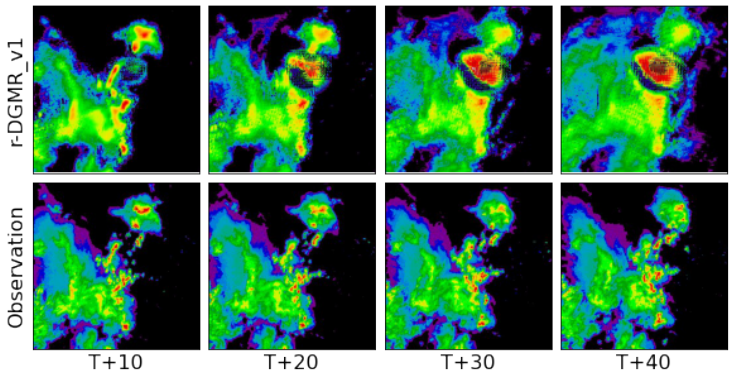


**Figure 5.1** r-DGMR_v1 artifact showcase.

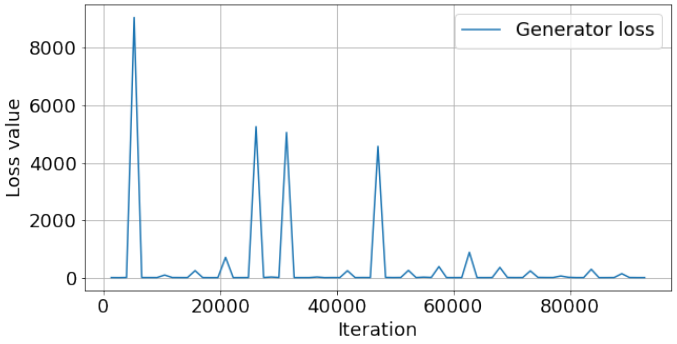


**Figure 5.2** r-DGMR_v1 loss on validation set. Unstable training can be visible based on the jumpy values.

## 5.2 Version 2

After the first attempt of training had been deemed unsuccessful, some changes had been thought of and tried in the second version.

To tackle unstable training, number of channels in random vector had further been reduced and also the standard deviation of its components had been reduced. Furthermore, the grid lambda in loss function was increased, with the idea being the network will focus more on the grid-wise difference between observations and predictions, rather than the discriminator part of the loss. Learning rate for both generator and discriminator had also been halved from the original values to allow for slower but more stable training. Grid weight divisor had also been increased to more equally punish the network for errors on various precipitation levels. Once again, you can see all the changes in table 5.2.

In the end, all those changes helped with training stabilization only a little. Training had still been unstable, but this time the artifacts did not occur as often as they did in version 1. Because of this, the training had been terminated a little earlier, just after the $7^{\text{th}}$ epoch. The results that did not produce artifacts had been rather blurry and produce a checkered pattern, which can be seen in figure 5.3. The loss function will not be showcased here, since it was very similar to the one in figure 5.2.

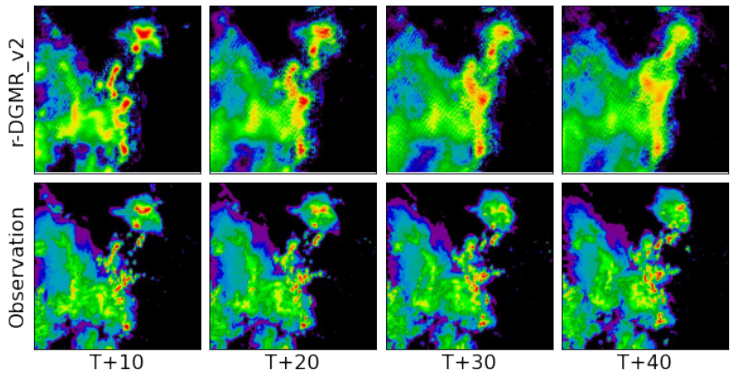


**Figure 5.3** r-DGMR_v2 prediction showcase. Checkered pattern can be seen on the predictions.

| Hyperparameter | New value | Original value |
|---|---|---|
| Output sequence length | 4 | 16 |
| Conditioning stack base num channels | 12 | 24 |
| Latent conditioning stack base num channels | 12 | 24 |
| Random vector num channels | 5 | 8 |
| Random vector std | 0.5 | 1 |
| Generator loss function grid lambda | 50 | 20 |
| Generator loss function grid weight divisor | 2 | 1 |
| Spatial discriminator base num channels | 24 | 48 |
| Temporal discriminator base num channels | 24 | 48 |
| Generator learning rate | 25e-6 | 5e-5 |
| Discriminator learning rate | 1e-4 | 2e-4 |
| Generator number of parameters | 10.5M | 41.9M |
| Discriminator number of parameters | 8.2M | 32.7M |

**Table 5.2** r-DGMR_v2 changes.

## 5.3 Version 3

For the third attempt to train *r-DGMR*, the main difference was an increase of output sequence length from 4 to 6. The generator had been given more capacity to achieve higher flexibility, grid lambda had been returned to original value, and the random vector had been altered again by giving it more dimensions and higher variance. These changes did in fact end up helping with stability of the training, as the artifact seen in earlier versions did not occur in this version. However, the main issue with version 3 was the inability to properly predict high-precipitation events. This can be seen in figure 5.4. This was deemed to be undesirable, as the main goal of weighed loss towards higher precipitation is to be able to predict higher precipitation events even for longer lead times. The entire configuration of version 3 can be seen in table 5.3.

After first 5 epochs, the *min_clip* value had been reduced from 2 to 1, and later to 0.5 in hopes of increasing the ability of the model to predict higher precipitation better, but after 4 more epochs, this had not been the case and the training had been terminated once again.

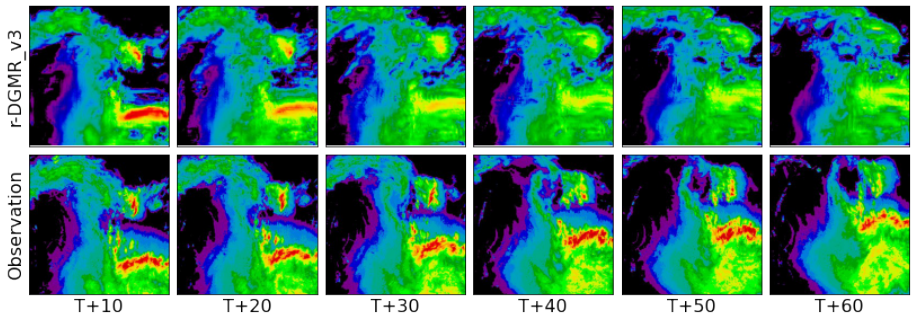


**Figure 5.4** r-DGMR_v3 prediction showcase. It can be seen at time step $T + 30$ already that high precipitation area had almost vanished from the prediction.

| Hyperparameter | New value | Original value |
|---|---|---|
| Output sequence length | 6 | 16 |
| Conditioning stack base num channels | 16 | 24 |
| Latent conditioning stack base num channels | 16 | 24 |
| Random vector num channels | 7 | 8 |
| Random vector std | 0.7 | 1 |
| Generator loss function grid weight divisor | 2;1;0.5 | 1 |
| Spatial discriminator base num channels | 24 | 48 |
| Temporal discriminator base num channels | 24 | 48 |
| Generator number of parameters | 18.6M | 41.9M |
| Discriminator number of parameters | 8.2M | 32.7M |

**Table 5.3** r-DGMR_v3 changes. Various grid weight divisors had been used.

## 5.4 Version 4

For this version, instead of further increasing the capacity of the generator, the capacity of the discriminator had been increased to have more parameters than generator itself. To account for this, batch size had to be reduced from 16 to 8 to fit the model within memory. The idea of increasing the discriminator capacity was that it would guide the generator better into producing realistically-looking predictions, which could subsequently also lead to more accurate predictions. Random vector standard deviation had been returned to the original value. Number of channels in this vector had been the same as in version 3.

This has proved to be true, since this model has achieved the lowest loss values during training. Figure 5.5 showcases the predictions made by this version of r-DGMR. It can be seen that even though the overall structure does indeed look realistic, it is somewhat grainy. For this reason, a simple post-processing method based on local adjustments by blurring had been developed that fixes this issue. It will be described in section 5.5.

As with all versions, a table with all the changes made compared to the original DGMR can be seen in table 5.4.

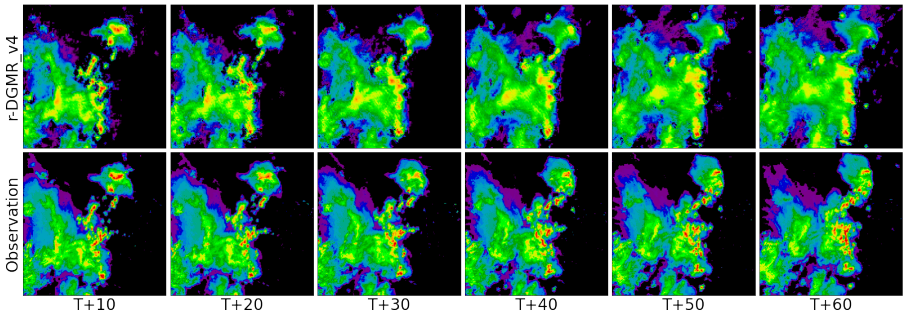


**Figure 5.5** r-DGMR_v4 prediction showcase.

| Hyperparameter | New value | Original value |
|---|---|---|
| Output sequence length | 6 | 16 |
| Batch size | 8 | 16 |
| Conditioning stack base num channels | 16 | 24 |
| Latent conditioning stack base num channels | 16 | 24 |
| Random vector num channels | 7 | 8 |
| Spatial discriminator base num channels | 40 | 48 |
| Temporal discriminator base num channels | 40 | 48 |
| Generator number of parameters | 18.6M | 41.9M |
| Discriminator number of parameters | 22.7M | 32.7M |

**Table 5.4** r-DGMR_v4 changes.

## 5.5 Post-processing method

Before selection of final model is described, a simple post-processing method that had been implemented needs to be introduced. It had proved to increase prediction quality both metric-wise and empirically, arguably giving the predictions of version 4 of r-DGMR a finer quality.

The method has two parameters, a *kernel size* ($ks$) and a *ratio* ($r$). Given those two parameters, the post-processing method does the following operation per pixel $p$ of each prediction step:

1. find the maximum value $m$ in vicinity of $p$ given by $ks$
2. find the average value $a$ in vicinity of $p$ given by $ks$
3. iff $p < m \cdot r$, then replace $p$ with $a$

The reason that this post-processing method had been chosen was due to the fact that, probably as a result of using adversarial approach to making predictions, the predictions looked as if they had missed a continuous precipitation area on purpose, giving them a grainy look. Figure 5.6 showcases the effect of this post-processing technique with parameters *kernel size* set to **5** and *ratio* to **3/4**.

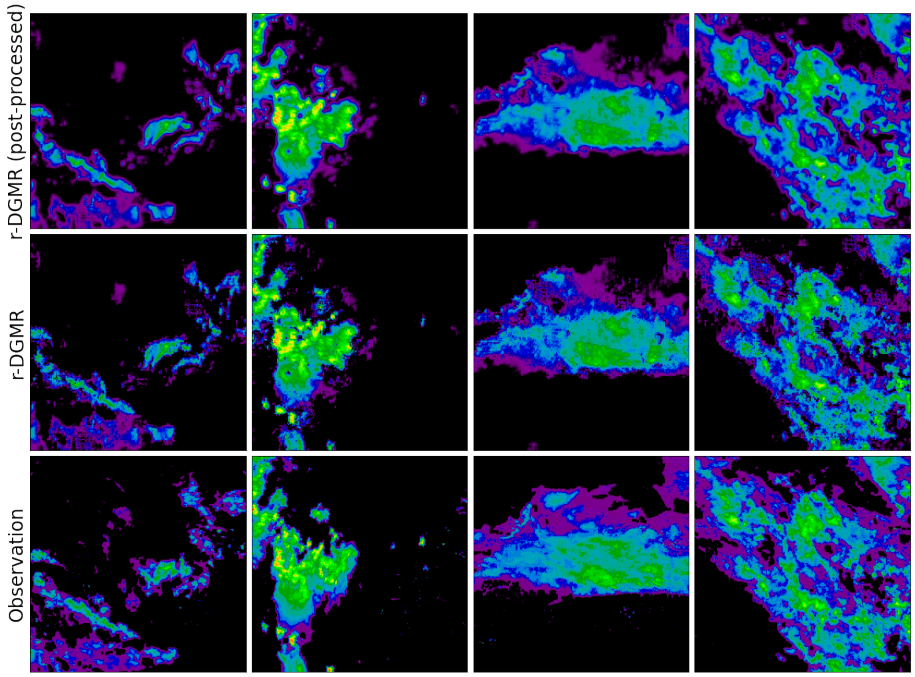


**Figure 5.6** r-DGMR post-processing with kernel size=5 and ratio=3/4 showcase on various predictions. It can be seen that it partially fixes the graininess of original prediction.

## 5.6 Final model selection

As version 4 was able to produce satisfying predictions, the next step was model selection. Because the training achieved similar loss value in multiple iterations, four of the models saved had been picked for selection. The selected models relative to the iteration and loss value can be seen in figure 5.7. Those models had been furthermore optionally altered by a post-processing method described in section 5.5 with various parameters. Table 5.5 shows all the variations of post-processing that had been explored per model.

This meant that overall, 40 models had been measured by various metrics, specifically **MAE, MSE** and **CSI** with thresholds of **0.1, 1.0** and **5.0**. The models will be referred to as **A**, **B**, **C** and **D** according to figure 5.7 from left to right respectively. Since DGMR is probabilistic, 10 samples had always been taken and their average metric score was considered as final metric value.

The first idea to find the best model was to find one that is superior in one or more metrics and at least as good as other models in all the other metrics. That means it would make it a dominant model. However, such a model did not exist. Because of this, a different approach was undertaken. First, the models were filtered by having at least one metric better than all other models. Seven models were left after this filtering, with their metrics seen in table 5.6.

Model **D** had been eliminated by this process altogether. Model **C** can be seen having the

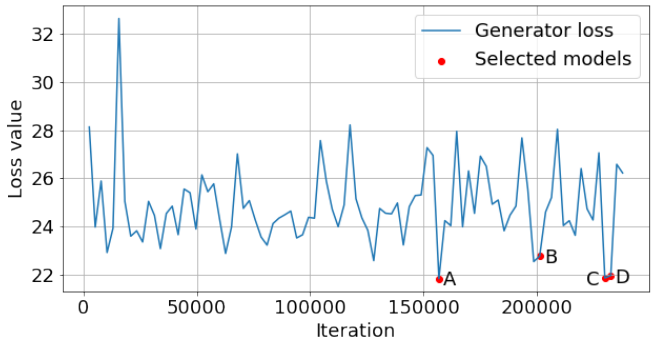


**Figure 5.7** r-DGMR_v4 validation generator loss. Models A, B, C and D are the models chosen for model selection and are highlighted by the red dots.

| Kernel size | Ratio |
|---|---|
| - | - |
| 5 | 1/2 |
| 5 | 1/4 |
| 5 | 3/4 |
| 9 | 1/2 |
| 9 | 1/4 |
| 9 | 3/4 |
| 13 | 1/2 |
| 13 | 1/4 |
| 13 | 3/4 |

**Table 5.5** Post-processing configurations explored. (-, -) in first row means the original model without post-processing had been explored as well.

| Model name | Kernel size | Ratio | MAE | MSE | CSI_0.1 | CSI_1.0 | CSI_5.0 |
|---|---|---|---|---|---|---|---|
| A | 5 | 3/4 | 0.14 | 1.234 | 0.451 | 0.221 | 0.067 |
| A | 5 | 1/2 | 0.142 | 1.261 | 0.449 | 0.222 | 0.067 |
| A | 9 | 1/2 | 0.142 | 1.241 | 0.458 | 0.226 | 0.067 |
| B | 9 | 1/2 | 0.127 | 1.153 | 0.463 | 0.227 | 0.065 |
| B | 13 | 3/4 | 0.121 | 1.095 | 0.469 | 0.213 | 0.057 |
| B | 13 | 1/4 | 0.13 | 1.163 | 0.46 | 0.227 | 0.064 |
| C | 13 | 3/4 | 0.112 | 1.086 | 0.404 | 0.168 | 0.046 |

**Table 5.6** Models considered for selection and their metrics on validation set. Kernel size and ratio refer to post-processing parameters. Model C can be seen having best *MSE* and *MAE*, but is the worst one for all *CSI* thresholds.

best *MAE* and *MSE*, but lacks in all thresholds of *CSI*. For model **A**, it is the exact opposite. Out of all variations of model **B**, the one with post-processing of kernel size set to 9 and ratio set to 1/2 has achieved competitive metrics overall compared to all other models and their variations. It is because of this reason that **this** model had been chosen as the final one. It achieves *MSE* and *MAE* comparable to model *C*, and also *CSI* values comparable to model *A*. This is the **r-DGMR** model variation that had been chosen as a model of choice. Figures 5.8 and 5.9 showcase how this specific post-processing in this model affects all the metrics per time step relative to the original model without post-processing. It can be seen that *MAE* is the only

metric that gets a little worse with increasing lead time, but all the other ones are improved for all lead times.

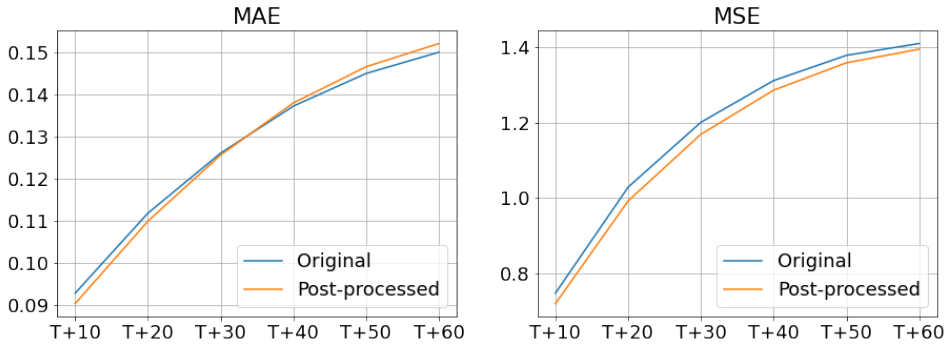


**Figure 5.8** r-DGMR-final post-processing comparison for MSE and MAE metrics. Post-processing improves *MSE* for all lead times but worsens *MAE* for lead times greater than 30 minutes.

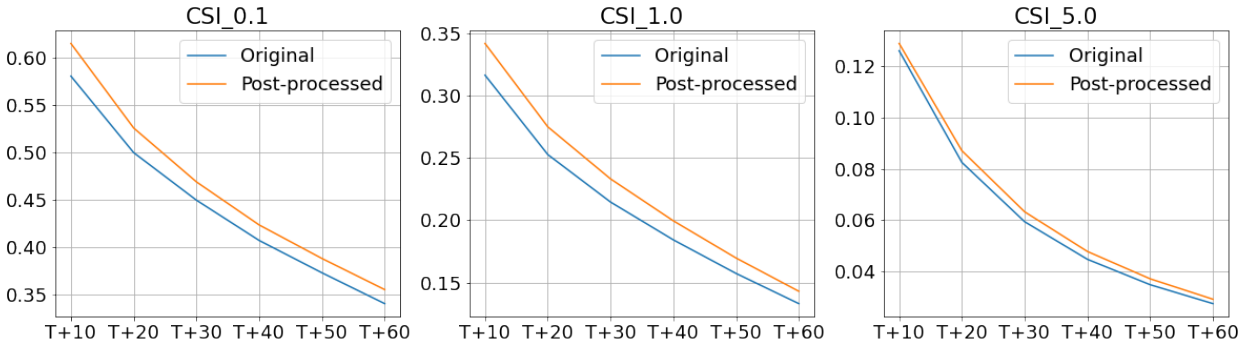


**Figure 5.9** r-DGMR-final post-processing comparison for CSI metric. Post-processing is improving the original model for all lead times in all thresholds.

# Chapter 6

# Evaluation

This chapter will evaluate and compare the *r-DGMR* model obtained in chapter 5 on a test dataset. This dataset contained a little over 17000 examples, as was described at the beginning of chapter 5. The first model that r-DGMR will be compared to is UNet [19], which had been previously internally trained at Meteopress by optimizing a combination of L1 and L2 loss of its predictions. It is a fully-convolutional neural network that is based on encoder-decoder architecture and residual connections. The second model that r-DGMR will be compared to is PySTEPS [17], an optical flow model based on an assumption of Lagrangian persistence of precipitation. This model computes the "apparent motion of brightness pattern" [16] for past observations and then proceeds to calculate new position of each pixel in the next step.

## 6.1 Metrics comparison

Because PySTEPS only moves the observation in a calculated direction, it does not fill the new space that is created by this movement. Even though this is a downside of this method, it was still desirable to fairly measure its prediction quality compared to those of r-DGMR and UNet. Both of those networks work with convolutional layers. This allows the inputs to be of varied size. Because the dataset contained radar observations of spatial resolution $[352 \times 544]$, these images had been used as input for all models. The measurement of metrics, however, had been done only on a $[256 \times 256]$ area in the center of the predictions.

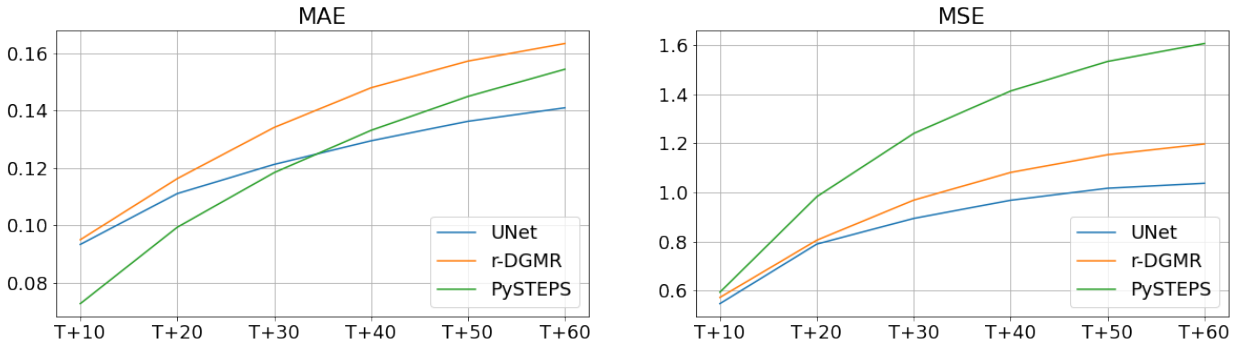


**Figure 6.1** Comparison of MSE and MAE for all lead times for obtained r-DGMR model compared to those of UNet and PySTEPS.

First two metrics measured had been *MSE* and *MAE*. Figure 6.1 shows the loss values for each lead time step. UNet and PySTEPS achieved very similar values, with UNet overtaking PySTEPS from lead time of T+40 minutes. r-DGMR performed the worst according to this metric, but is still competitive enough. For MSE, PySTEPS did the worst. With increasing lead time, its squared distance from the observations deviated from those of UNet and r-DGMR. Those tho models had achieved similar results. It should be noted that UNet had been optimized to perform well on those two metrics.

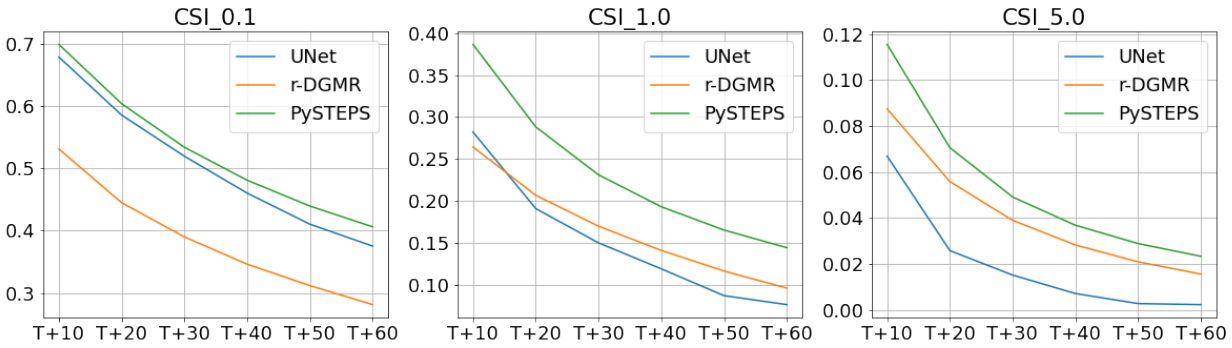


**Figure 6.2** Comparison of CSI for various precipitation thresholds for all lead times for obtained r-DGMR model compared to those of UNet and PySTEPS.

The second metric that had been measured was *CSI* for precipitation thresholds of 0.1, 1.0 and 5.0 $mm \cdot h^{-1}$. PySTEPS had shown to be the best model in this metric for all thresholds. r-DGMR does not do well for threshold of 0.1. However, with increasing threshold level, it begins to overtake UNet and for threshold of 5.0 is competitive to PySTEPS. UNet loses very badly on longer lead times for this threshold, with its values being close to zero. This may be attributed to the arguably heavy blurring of UNet predictions with increasing lead times, which subsequently leads to vanishing of higher intensity precipitation. This can be observed in the examples included in the appendix of this thesis.

## 6.2 Verdict and outline of future work

The obtained r-DGMR has shown to produce predictions within the range of comparability to both UNet and PySTEPS according to all measured metrics. However, it can be argued that the predictions of r-DGMR look more realistic, as can be seen in figure 6.3. This should be regarded as a success for this thesis, since obtaining a model which is able to product such predictions was the ultimate goal. This quality could be directly attributed to the adversarial training approach of the network. More examples of predictions of all compared models side by side are included in the appendix of this thesis.

The first pro of the network is its ability to make predictions for various resolutions thanks to convolutional layers. This allowed for a valuable comparison to PySTEPS. Another benefit would be the ability to change the capacity of the network by simply using different configuration of its various components.

However, this may also be regarded as the architecture's downside. Fine tuning such a model is computationally expensive since it takes a lot of time to train and evaluate. Deciding when a generative adversarial network is still learning something useful and when is it producing undesirable outputs proved to be a difficult task during training. Similar loss values of the

generator does not necessarily mean the predictions are of the same quality, since the loss depends on the quality of the discriminator as well.

In the future, it could be interesting to include more information as input to the network, such as satellite imagery or even the predictions made by numerical models. A different kind of generator network architecture could also be considered. r-DGMR's predictions had been described as more realistically looking. However, this had not been necessarily captured by the metrics measured. More evaluation by different metrics would be worthwhile to try and see if any of them are able to capture this quality.

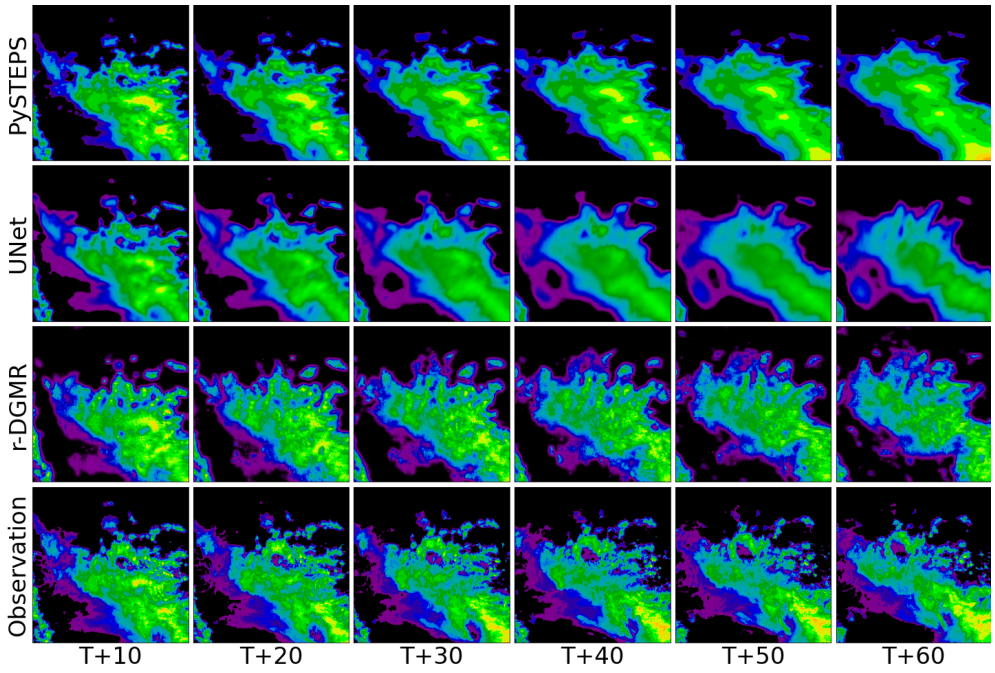


**Figure 6.3** Comparison of r-DGMR, UNet and PySTEPS predictions side by side.

# Conclusion

The main objective of this thesis was to implement, train and evaluate a precipitation nowcasting model. Specifically, a neural network model designed and published by DeepMind called DGMR that tackles this challenge with a conditional generative adversarial network.

In the beginning, the reader had been familiarized with the problem of precipitation nowcasting. Current methods used to tackle this issue had been explored and the reader had also been familiarized with all the necessary neural network concepts needed to understand the architecture of DGMR.

Even though the original architecture of the model had not been used since attempting to do so on the available hardware would be impossible due to limitations such as memory available and computational power, various alterations of this model with reduced capacity had been explored.

In the end, a model that is capable of making realistically looking predictions had been obtained. This was also the promised outcome of this thesis, since a generative adversarial network architecture is specifically designed to produce results with this property. A simple post-processing method had also been introduced that has shown to increase the quality of the model's predictions both metric wise and empirically, making them less grainy.

The model was furthermore evaluated and compared to other methods of precipitation nowcasting using various metrics. It was shown that it is competitive enough to be considered as a valuable model. Moreover, it could also be argued that the obtained model's predictions look more realistic and do not resort to blurriness, which is a common problem with other methods. This can be observed in the showcase of the appendix of this thesis, where precipitation events of various intensities and the predictions of the respective models are shown.

# Bibliography


1. MASSON-DELMOTTE, V. et al. Climate Change 2021: The Physical Science Basis. *IPCC* [online]. 2021 [visited on 2022-03-28]. Available from: https://www.ipcc.ch/report/ar6/wg1/downloads/report/IPCC_AR6_WGI_SPM_final.pdf.
2. ZHOU, Xiaqiong; ZHU, Yuejian; HOU, Dingchen; LUO, Yan; PENG, Jiayi; WOBUS, Richard. Performance of the New NCEP Global Ensemble Forecast System in a Parallel Experiment. *Weather and Forecasting* [online]. 2017, vol. 32 [visited on 2022-03-28]. Available from DOI: 10.1175/WAF-D-17-0023.1.
3. RONALD, E. Radar for Meteorologists [online]. 2004 [visited on 2022-03-28]. Available from: https://www.wxonline.info/topics/radar_nonmet.html.
4. WOLFF, Christian. Radar Basics - Range or distance measurement [online]. 2019 [visited on 2022-03-28]. Available from: https://www.radartutorial.eu/01.basics/Distance-determination.en.html.
5. ZHANG, G. et al. Current Status and Future Challenges of Weather Radar Polarimetry: Bridging the Gap between Radar Meteorology/Hydrology/Engineering and Numerical Weather Prediction. *Advances in Atmospheric Sciences* [online]. 2019, vol. 36, pp. 571–588 [visited on 2022-03-28]. Available from DOI: 10.1007/s00376-019-8172-4.
6. CARLIN, Jacob. Weather radar polarimetry [online]. 2015 [visited on 2022-03-28]. Available from DOI: 10.1063/PT.5.4011.
7. AYZEL, G.; SCHEFFER, T.; HEISTERMANN, M. RainNet v1.0: a convolutional neural network for radar-based precipitation nowcasting. *Geoscientific Model Development* [online]. 2020, vol. 13, no. 6, pp. 2631–2644 [visited on 2022-04-19]. Available from DOI: 10.5194/gmd-13-2631-2020.
8. TUYEN, Do Ngoc; TUAN, Tran Manh; LE, Xuan-Hien; TUNG, Nguyen Thanh; CHAU, Tran Kim; VAN HAI, Pham; GEROGIANNIS, Vassilis C.; SON, Le Hoang. RainPredRNN: A New Approach for Precipitation Nowcasting with Weather Radar Echo Images Based on Deep Learning. *Axioms* [online]. 2022, vol. 11, no. 3 [visited on 2022-04-20]. ISSN 2075-1680. Available from DOI: 10.3390/axioms11030107.
9. TREVISAN, V. Comparing Robustness of MAE, MSE and RMSE [online]. [N.d.] [visited on 2022-04-29]. Available from: https://towardsdatascience.com/comparing-robustness-of-mae-mse-and-rmse-6d69da870828.
10. FERNÁNDEZ, Jesús García; MEHRKANOON, Siamak. Broad-UNet: Multi-scale feature learning for nowcasting tasks. *CoRR* [online]. 2021, vol. abs/2102.06442 [visited on 2022-04-20]. Available from arXiv: 2102.06442.

11. WANG, Yunbo; WU, Haixu; ZHANG, Jianjin; GAO, Zhifeng; WANG, Jianmin; YU, Philip S.; LONG, Mingsheng. PredRNN: A Recurrent Neural Network for Spatiotemporal Predictive Learning. *CoRR* [online]. 2021, vol. abs/2103.09504 [visited on 2022-04-20]. Available from arXiv: 2103.09504.
12. ZHAO, Hang; GALLO, Orazio; FROSIO, Iuri; KAUTZ, Jan. *Loss Functions for Neural Networks for Image Processing* [online]. arXiv, 2015 [visited on 2022-03-31]. Available from DOI: 10.48550/ARXIV.1511.08861.
13. RAVURI, S.; LENC, K.; WILLSON, M., et al. Skillful Precipitation Nowcasting using Deep Generative Models of Radar. *Nature* [online]. 2021, vol. 597, pp. 672–677 [visited on 2022-03-24]. Available from: https://www.nature.com/articles/s41586-021-03854-z.
14. OCEANIC, National; ADMINISTRATION, Atmospheric. *Forecast Verification Glossary* [online]. [N.d.] [visited on 2022-04-26]. Available from: https://www.swpc.noaa.gov/sites/default/files/images/u30/Forecast%20Verification%20Glossary.pdf.
15. LORENZ, Edward. Predictability - a problem partly solved [online]. 1996 [visited on 2022-04-02]. Available from: https://eapsweb.mit.edu/sites/default/files/Predicability_a_Problem_2006.pdf.
16. BAKER, Simon; SCHARSTEIN, Daniel; LEWIS, J.P.; ROTH, Stefan; BLACK, Michael; SZELISKI, Richard. A Database and Evaluation Methodology for Optical Flow. *International Journal of Computer Vision*. 2007, vol. 92, pp. 1–31. Available from DOI: 10.1007/s11263-010-0390-2.
17. PULKKINEN, S.; NERINI, D.; PÉREZ HORTAL, A. A.; VELASCO-FORERO, C.; SEED, A.; GERMANN, U.; FORESTI, L. Pysteps: an open-source Python library for probabilistic precipitation nowcasting (v1.0). *Geoscientific Model Development* [online]. 2019, vol. 12, no. 10, pp. 4185–4219 [visited on 2022-03-27]. Available from DOI: 10.5194/gmd-12-4185-2019.
18. AGRAWAL, Shreya; BARRINGTON, Luke; BROMBERG, Carla; BURGE, John; GAZEN, Cenk; HICKEY, Jason. Machine Learning for Precipitation Nowcasting from Radar Images. *CoRR* [online]. 2019, vol. abs/1912.12132 [visited on 2022-04-19]. Available from arXiv: 1912.12132.
19. RONNEBERGER, Olaf; FISCHER, Philipp; BROX, Thomas. U-Net: Convolutional Networks for Biomedical Image Segmentation. *CoRR* [online]. 2015, vol. abs/1505.04597 [visited on 2022-04-19]. Available from arXiv: 1505.04597.
20. ZHENG, K.; LIU, Y.; ZHANG, J.; LUO, C.; TANG, S.; RUAN, H.; TAN, Q.; YI, Y.; RAN, X. GAN–argcPredNet v1.0: a generative adversarial model for radar echo extrapolation based on convolutional recurrent units. *Geoscientific Model Development* [online]. 2022, vol. 15, no. 4, pp. 1467–1475 [visited on 2022-04-20]. Available from DOI: 10.5194/gmd-15-1467-2022.
21. SAZLI, Murat. A brief review of feed-forward neural networks. *Communications, Faculty Of Science, University of Ankara*. 2006, vol. 50, pp. 11–17. Available from DOI: 10.1501/0003168.
22. RUDER, Sebastian. An overview of gradient descent optimization algorithms. *CoRR* [online]. 2016, vol. abs/1609.04747 [visited on 2022-03-26]. Available from arXiv: 1609.04747.
23. ZHANG, Aston; LIPTON, Zachary C.; LI, Mu; SMOLA, Alexander J. Dive into Deep Learning. *arXiv preprint arXiv:2106.11342* [online]. 2021 [visited on 2022-03-24]. Available from: http://d2l.ai/.
24. SHARMA, S. Activation Functions in Neural Networks [online]. 2017 [visited on 2022-04-28]. Available from: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

25. O'SHEA, Keiron; NASH, Ryan. An Introduction to Convolutional Neural Networks. *ArXiv e-prints* [online]. 2015 [visited on 2022-04-29]. Available from: https://www.researchgate.net/publication/285164623_An_Introduction_to_Convolutional_Neural_Networks.
26. GHOLAMALINEZHAD, Hossein; KHOSRAVI, Hossein. Pooling Methods in Deep Neural Networks, a Review. *CoRR* [online]. 2020, vol. abs/2009.07485 [visited on 2022-03-24]. Available from arXiv: 2009.07485.
27. THEVENAZ, Philippe; BLU, Thierry; UNSER, Michael. Image Interpolation and Resampling. In: *HANDBOOK OF MEDICAL IMAGING, PROCESSING AND ANALYSIS* [online]. Academic Press, 2000, pp. 393–420 [visited on 2022-03-26].
28. TORCH CONTRIBUTORS. *torch.nn – PyTorch master documentation [online]* [online]. 2018 [visited on 2022-03-26]. Available from: https://pytorch.org/docs/stable/nn.html.
29. IOFFE, Sergey; SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* [online]. 2015, vol. abs/1502.03167 [visited on 2022-03-26]. Available from arXiv: 1502.03167.
30. SANTURKAR, Shibani; TSIPRAS, Dimitris; ILYAS, Andrew; MADRY, Aleksander. *How Does Batch Normalization Help Optimization?* [Online]. arXiv, 2018 [visited on 2022-03-26]. Available from DOI: 10.48550/ARXIV.1805.11604.
31. CHUNG, Junyoung; GÜLÇEHRE, Çaglar; CHO, KyungHyun; BENGIO, Yoshua. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* [online]. 2014, vol. abs/1412.3555 [visited on 2022-03-26]. Available from arXiv: 1412.3555.
32. BALLAS, Nicolas; YAO, Li; PAS, Chris; COURVILLE, Aaron. *Delving Deeper into Convolutional Networks for Learning Video Representations* [online]. 2015 [visited on 2022-03-26]. Available from: https://arxiv.org/abs/1511.06432.
33. GOODFELLOW, Ian J.; POUGET-ABADIE, Jean; MIRZA, Mehdi; XU, Bing; WARDE-FARLEY, David; OZAIR, Sherjil; COURVILLE, Aaron; BENGIO, Yoshua. *Generative Adversarial Networks* [online]. arXiv, 2014 [visited on 2022-03-26]. Available from DOI: 10.48550/ARXIV.1406.2661.
34. MIRZA, Mehdi; OSINDERO, Simon. Conditional Generative Adversarial Nets. *CoRR* [online]. 2014, vol. abs/1411.1784 [visited on 2022-03-31]. Available from arXiv: 1411.1784.
35. MIYATO, Takeru; KATAOKA, Toshiki; KOYAMA, Masanori; YOSHIDA, Yuichi. *Spectral Normalization for Generative Adversarial Networks* [online]. arXiv, 2018 [visited on 2022-04-02]. Available from DOI: 10.48550/ARXIV.1802.05957.
36. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. *Attention Is All You Need* [online]. arXiv, 2017 [visited on 2022-04-02]. Available from DOI: 10.48550/ARXIV.1706.03762.
37. AYZEL, Georgy; HEISTERMANN, Maik; WINTERRATH, Tanja. Optical flow models as an open benchmark for radar-based precipitation nowcasting (rainymotion v0.1). *Geoscientific Model Development Discussions* [online]. 2018, pp. 1–23 [visited on 2022-03-28]. Available from DOI: 10.5194/gmd-2018-166.
38. ABADI, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* [online]. 2015 [visited on 2022-04-11]. Available from: https://www.tensorflow.org/. Software available from tensorflow.org.
39. CHEN, Tianqi; LI, Mu; LI, Yutian; LIN, Min; WANG, Naiyan; WANG, Minjie; XIAO, Tianjun; XU, Bing; ZHANG, Chiyuan; ZHANG, Zheng. *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems* [online]. arXiv, 2015 [visited on 2022-04-11]. Available from DOI: 10.48550/ARXIV.1512.01274.

40. FALCON, W. et al. PyTorch Lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*. 2019, vol. 3.

41. EUMETNET. *OBSERVATIONS - OPERA* [online]. [N.d.] [visited on 2022-04-26]. Available from: `http://www.eumetnet.eu/activities/observations-programme/current-activities/opera/`.

# Appendix A

# Prediction comparison showcase

This appendix contains examples of various precipitation events and the predictions made by **r-DGMR**, **UNet** and **PySTEPS**. Because r-DGMR is probabilistic, a representative prediction had been chosen by sampling 20 predictions, taking their mean value, and picking the one sample that differs the least from the mean measured by L1 distance.

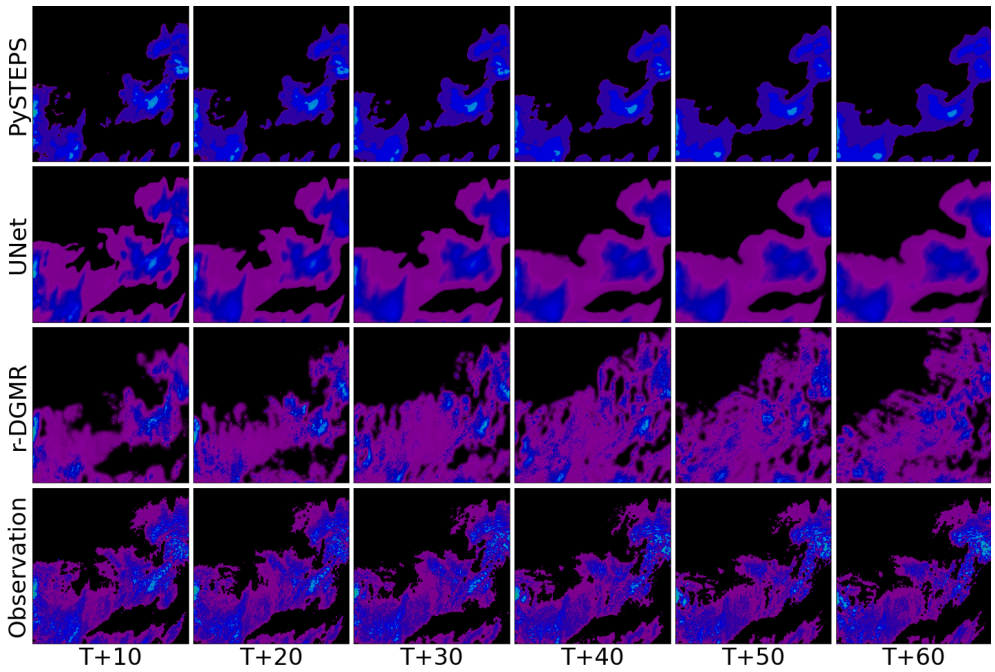


**Figure A.1** Predictions showcase 1.

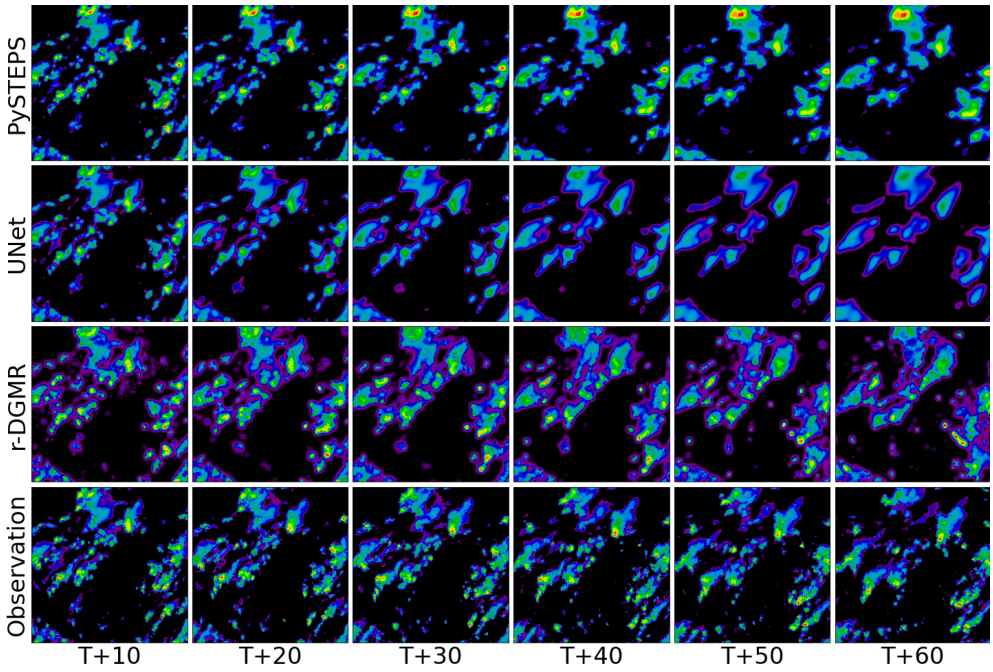


**Figure A.2** Predictions showcase 2.

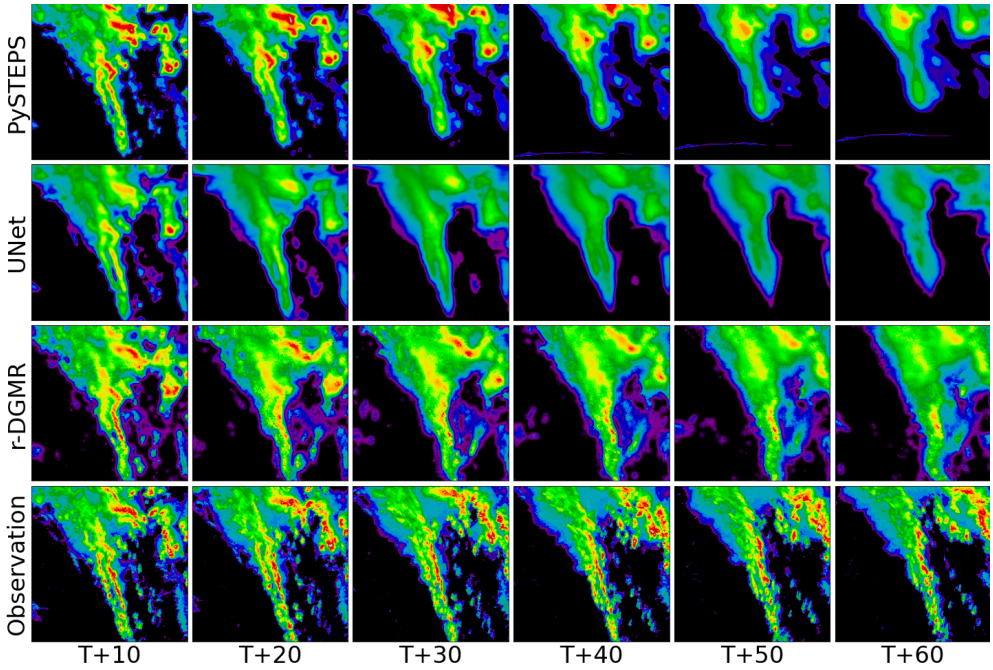


**Figure A.3** Predictions showcase 3.

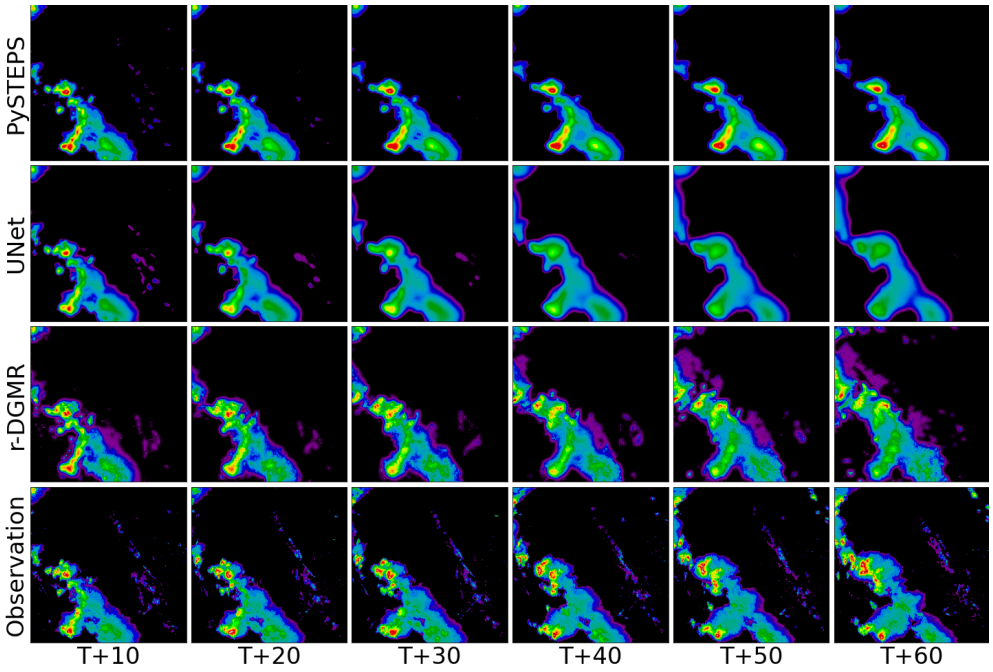


**Figure A.4** Predictions showcase 4.

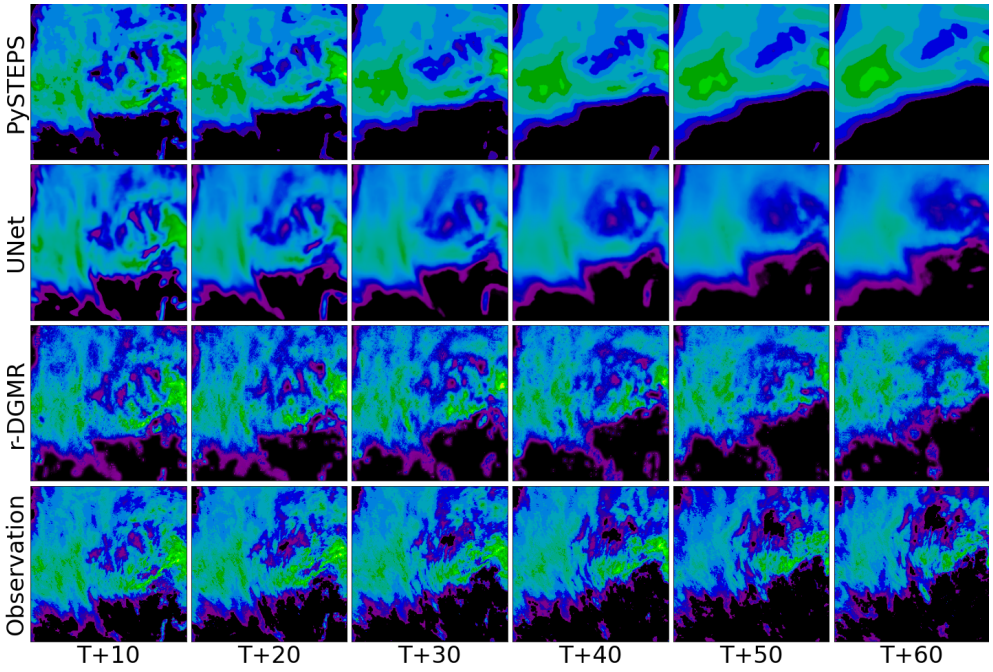


**Figure A.5** Predictions showcase 5.

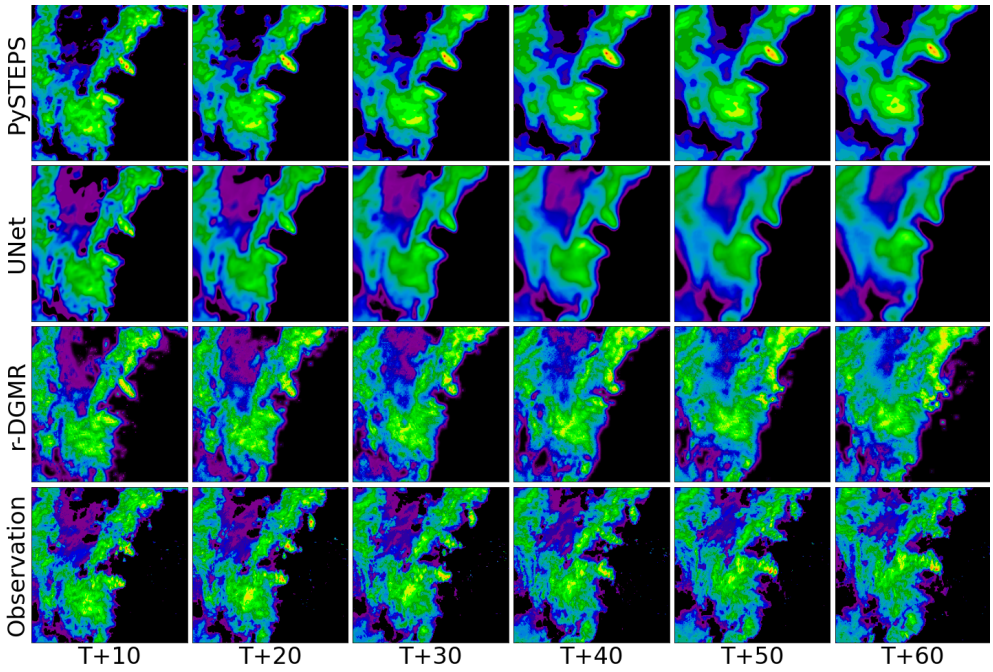


**Figure A.6** Predictions showcase 6.

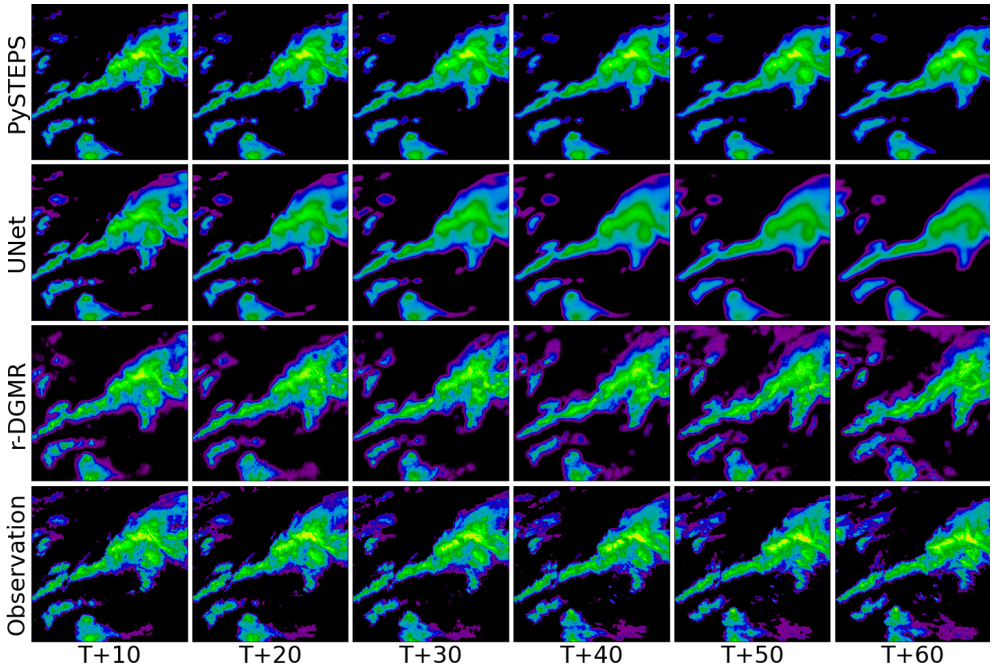


**Figure A.7** Predictions showcase 7.

# Appendix B

# Contents of enclosed DVD

- `README.md` ..... a file with a description of the DVD
- `requirements.txt` ..... PIP requirements file for a working environment
- `train_dataset.pt` ..... dataset of training examples saved as PyTorch tensor
- `train_dgmr.ipynb` ..... jupyter notebook for DGMR training
- `src` ..... source code directory for DGMR
  - `blocks` ..... directory with source code for various block types used in DGMR
    - `d_block.py` ..... DBlock implementation
    - `g_block.py` ..... GBlock implementation
    - `l_block.py` ..... LBlock implementation
  - `discriminators` ..... directory with source code for DGMR discriminators
    - `spatial_discriminator.py` ..... source code for spatial discriminator
    - `temporal_discriminator.py` ..... source code for temporal discriminator
    - `discriminator.py` ..... source code for the discriminator used in DGMR
  - `generator` ..... directory with source code for DGMR generator
    - `attention_module.py` ..... source code for attention module used in generator
    - `conditioning_stack.py` ..... source code for conditioning stack used in generator
    - `conv_gru.py` ..... ConvGRU implementation with DGMR adjustments
    - `generator.py` ..... source code for the generator used in DGMR
    - `latent_conditioning_stack.py` ..... source code for latent conditioning stack
    - `sampler.py` ..... sampler source code
  - `dgmr.py` ..... PyTorch Lightning module for DGMR
  - `losses.py` ..... loss functions used in DGMR
- `tests` ..... directory with pytest tests for DGMR implementation
- `text` ..... directory with text of this thesis
  - `src` ..... directory with source code for this thesis written in LaTeX
  - `thesis.pdf` ..... this thesis text in PDF format