Universiteit Utrecht

Koninklijk Nederlands
Meteorologisch Instituut
Ministerie van Infrastructuur en Waterstaat

Faculteit Bètawetenschappen

Master Thesis

Mathematical Sciences

Statistical postprocessing of wind speed forecasts using convolutional neural networks

---

Author:
Simon Veldkamp

Supervisors:
Dr. Sjoerd Dirksen
Utrecht University
Dr. Maurice Schmeits
Dr. Kirien Whan
KNMI

Second Reader:
Prof. Dr. Jason Frank
Utrecht University

January 21, 2020

Abstract

Weather forecasts provided by numerical weather prediction (NWP) models typically give a deterministic forecast. However, there is a certain amount of uncertainty in these forecasts. The aim of statistical post-processing is to give a probabilistic forecast instead. Current statistical post-processing methods for providing a probabilistic forecast are not capable of using full spatial patterns from the NWP model. Recent developments in deep learning (notably convolutional neural networks) have made it possible to use large gridded input data sets. This could potentially be useful in statistical postprocessing, since it allows us to use more spatial information. In this research we consider wind speed forecasts for 48 hours ahead, as provided by KNMI's Harmonie-Arome model. Convolutional neural networks, fully connected neural networks and quantile regression forests are used to obtain probabilistic wind speed forecasts. Comparing these methods shows that Convolutional neural networks are more skillful than the other methods, especially for medium to higher wind speeds.

Ackowledgements

First and foremost I would like to thank my supervisors from KNMI, dr. Kiri Whan and dr. Maurice Schmeits, who were always approachable and available for discussions when I needed help, and who helped me a lot with staying motivated by showing enthusiasm for my project. I also want to thank my supervisor from Utrecht University, dr. Sjoerd Dirksen, for the time he took in supervising my thesis, and providing much needed feedback on the written product. I would furthermore like to thank Andrea Pagani and Dirk Wolters for their help with the practical aspects of deep learning.

Finally I would like to thank my parents for stimulating my curiosity and interest in science from a young age and who supported me through all my years in university.

Contents

1 Introduction

Accurate and reliable weather forecasts are important in many branches of society. Decision making in, for example, agriculture, aviation and renewable energy production are all dependent on weather forecasts. Furthermore, extreme weather can be dangerous and it is therefore important to give reliable warnings when dangerous weather can be expected.

Forecasts are generally determined by numerical weather prediction (NWP) models, such as, for example, KNMI's HarmonieArome[5]. These are large dynamical models based on a physical model of the atmosphere. Such models are initialized based on measurements and past forecasts. Due to a lack of measurements a perfect initialization is not possible. Furthermore, simplifying assumptions have to be made, to make computation feasible. Together these effects lead to errors in both the initialization and the forecast of the model. Besides, a single model run only provides a single deterministic forecast. The atmosphere is however a famously chaotic system [28] and every forecast is therefore inherently uncertain. A single forecast given by a NWP model does not provide an estimate of this uncertainty, even though it is important for decision makers to have such an estimate. Furthermore, due to the simplifying assumptions made, there is in general a bias in the predictions.

The uncertainty is usually estimated by creating an ensemble of NWP model output of which the members are initialized differently and use slightly different physical parametrizations. This, however, is computationally expensive and the results are often still biased and underdispersed[14].

In statistical postprocessing the aim is to give better estimates of the bias and uncertainty in the model, based on past observations. The Popular framework for this is model output statistics (MOS)[12]. In MOS the forecast as provided by the NWP model is compared to corresponding measurements. In this way we can correct the bias and estimate the uncertainty for the NWP model, based on the output of the model itself and potentially some extra variables, such as, for example, the time of the year. We need a dataset containing both measurements and forecasts to apply MOS to a new model. This dataset is obtained by letting the model forecast the weather for days in the past, for which measurements are already available, this is called a reforecast. Obtaining such a dataset is computationally expensive, large datasets are therefore often not available.

The version of MOS that is currently the most popular is ensemble model output statistics or EMOS [14]. In EMOS one tries to fit a parametric distribution based on the statistics of the ensemble forecasts and corresponding measurements. EMOS has been applied to wind speed forecasts in [38],[42],[4],where they used truncated normal and log normal distributions. Furthermore in [26] a mixture of truncated normal and the generalised extreme value distribution was used with success.

EMOS has been compared to quantile regression forests (QRF)[31], a non parametric technique based on random forests, for both wind speed and temperature forecasts in [41], where QRF was found to be more skilful. QRF was also used in [43] for postprocessing of precipitation forecasts. Furthermore in[34] neural networks are used, which unlike EMOS are capable of modelling non-linear effects, for the statistical postprocessing of temperature forecasts. EMOS, QRF and neural networks are all however, not well suited to analyse high-dimensional structured spatial data. Weather forecasts are spatial in nature, it could therefore potentially be beneficial to use postprocessing methods that are capable of

dealing with this spatial information.

Recent developments in artificial neural networks have made them the state of the art technique in a number of different tasks, such as image classification, time series analysis and even image generation[25][24]. Many of these methods can potentially be of great benefit in geosciences[35] and have been applied in a few papers already. In [27] for example, convolutional neural networks where used to detect extreme weather events in climate datasets and in [39] a mix between a convolutional and a recurrent network was used for now-casting of precipitation. Convolutional neural networks were also used in [37] to estimate the uncertainty in weather forecasts based on the state of the atmosphere in the initialization of the NWP model.

Convolutional neural networks have not been used in statistical postprocessing yet to my knowledge. They, however, could potentially be a beneficial addition because of their capability to analyze spatial information of weather forecasts. In this study we apply convolutional neural networks for the post-processing of 48h windspeed forecasts in the Netherlands. We compare a number of different methods of obtaining a probability distribution using convolutional neural networks. Furthermore, we examine whether convolutional neural networks add any skill with respect to fully connected neural networks and quantile regression forests.

This thesis is structured as follows. In section 2 an overview of statistical postprocessing will be given, section 3 will give an overview of neural networks and convolutional neural networks and section 4 will describe the methods used for obtaining a probability distribution with convolutional neural networks. The last chapter will describe the experiments, in which we compare convolutional neural networks to quantile regression forests and fully connected neural networks, to determine whether convolutional neural networks' capability of using spatial information will lead to more skilful forecasts.

2 Statistical Post Processing

2.1 Statistical Forecasting

The following sections are based mainly on Wilks [44] and Hastie et. al. [19]

In statistical forecasting we try to give a weather forecast based on statistical relationships between variables. We could for example try to find the relationship between air pressure at a certain location and the probability of rain the next day. To do this we need to have data for both these variables. The variable that we are trying to predict is called the *predictand* and will be denoted by $y \in \mathcal{Y}$. The variables we use to base these predictions on are called the *predictors* and will be denoted by $\mathbf{x} \in \mathcal{X}$, where $\mathcal{X}$ will typically be a subset of $\mathbb{R}^m$ with m the number of predictors. We assume that the predictors are realizations of a random variable X and the predictands are realization of a random variable Y. The aim of statistical forecasting is then to find a map $f : \mathcal{X} \to \mathcal{Y}$ such that

$$\mathbb{E}[Y|X = \mathbf{x}] = f(\mathbf{x}).$$

Or when the aim is to estimate the full probability distribution of our predictand, we want to find a map f s.t. $f(\mathbf{x}) = P(Y|X = \mathbf{x})$, a so called probabilistic forecast. The data used to find such a mapping is called the *training data*, which contains corresponding pairs $(\mathbf{x_i}, y_i)$ for $i = 1, .., N$. The set containing the training data will be denoted by $\mathcal{D}$. In the rest of this section we will describe methods of estimating these relationships.

2.2 Regression

When trying to derive a relationship between some predictor $\mathbf{x}$ and a predictand y we can either use non-parametric or parametric methods. Parametric methods assume that the relationship can be approximated by some pre-defined function, where the exact shape of the function only depends on a number of parameters. Non-parametric methods are methods that don't have such a strong a priori estimate of the shape of the function that has to be approximated. In general non-parametric methods have the benefit that they are less dependent on a good a priori estimate of the relation we are looking for. The freedom these methods have however, come at a price. They generally need a lot more data and are not capable of extrapolating to values outside of the training dataset, and thus don't handle extreme values well. An example of a non-parametric method is Quantile Regression Forest (QRF), which will be described in section 2.3. In the next section we will describe a few simple parametric methods first.

2.2.1 Linear Regression

Let $\mathcal{X} \subset \mathbb{R}^n$ and $\mathcal{Y} \subset \mathbb{R}$. The simplest and probably most well known parametric method is linear regression. In linear regression we assume that the relation between the predictand y and the predictors

$\mathbf{x}$ is linear, i.e., we assume that we can write the function $f : \mathcal{X} \to \mathcal{Y}$, which gives the expected value of y given $\mathbf{x}$, as $f(\mathbf{x}) = W\mathbf{x} + b$, where $W \in \mathbb{R}^{1\times n}$ and $b \in \mathbb{R}$. Although this is an affine map instead of a linear one, we can turn it into a linear function by adding an extra variable to the data which is always equal to one, i.e. we define $\tilde{\mathbf{x}} = [1, \mathbf{x}^T]^T$ and $\tilde{W} = [b, W]$, such that $f(\mathbf{x}) = \tilde{W}\tilde{\mathbf{x}} = W\mathbf{x} + b$. From here on we will use this linear formulation, leaving out the tildes.

The most common framework used for estimating the parameters W is least squares regression. In least squares regression we try to minimize the average squared distance between $f(\mathbf{x})$ and y. Let $\mathbf{x}_i$ be the i-th predictor datapoint and y_i the corresponding predictand, with $i = 1, .., N$ where N is the number of data points. Then the parameter W is chosen such that it minimizes the following quantity

$$J(W) = \sum_{i=1}^{N} (W\mathbf{x}_i - y_i)^2. \tag{1}$$

The mean squared error is a popular choice for the quantity that is to be minimized. Firstly it is easy to find the optimum due to the fact that $\nabla J(W) = 2\sum_{i=1}^{N}(W\mathbf{x}_i - y_i)\mathbf{x}_i$ allows for analytic solving methods. Furthermore, if we assume that the errors are independent and identically distributed Gaussians, than the mean squared error is equal to the maximum likelihood estimate. (See Section 2.2.3)

2.2.2 Logistic Regression

Linear regression can also be used for classification. In this case we add a function that transforms the output of the linear layer to a binary value. Typically one uses the logistic function for this task. The logistic function is defined by $\sigma : \mathbb{R} \to [0, 1]$ s.t.

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

This function has the property that $\lim_{x\to-\infty} \sigma(x) = 0$ and $\lim_{x\to\infty} \sigma(x) = 1$. In this case we get a function $f : \mathcal{X} \to (0, 1)$ s.t. $f(\mathbf{x}) = \sigma(W\mathbf{x})$. Similarly as before we can minimise the mean squared error to fit the parameters to our model. In this case the predictands y will be either 1 or 0 representing two classes.

Since $\sigma(x) < 1/2$ for negative values of x and $\sigma(x) > 1/2$ for positive values of x, $sign(W\mathbf{x})$ determines which class our sample is more likely a part of. The two classes are divided in the predictor space by the plane $W\mathbf{x} = 0$. Hence logistic regression determines a linear plane dividing two classes. The output of sigmoid function can be interpreted as the probability of the predictand being equal to one. i.e. we estimate the probability for Y given X as

$$\hat{P}(Y = 0|X = \mathbf{x}) = 1 - \sigma(W\mathbf{x})$$
$$\hat{P}(Y = 1|X = \mathbf{x}) = \sigma(W\mathbf{x})$$

When we have more than two categories we need a higher dimensional output. In this case the parameter W are a n by k matrix instead of a vector, where n is the dimensionality of the predictors

and k the number of categories. In this case we can use the softmax function defined as $softmax : \mathbb{R}^k \to S^{k-1}$ s.t.

$$softmax(\mathbf{z})_i = \frac{e^{\mathbf{z}_i}}{\sum_{j=1}^{k} e^{\mathbf{z}_j}} \text{ for } i = 1, .., k.$$

Note that for $k = 2$ we have

$$softmax(\mathbf{z}) = [\sigma(\mathbf{z}_1), 1 - \sigma(\mathbf{z}_1)].$$

Therefore the softmax function can be seen as an extension of the logistic function for multiple category problems.

2.2.3 Parametric Density Estimation

In linear regression we give a deterministic estimate for y given $\mathbf{x}$, i.e. we model $\mathbb{E}[Y|X = \mathbf{x}]$. This is done by minimizing the mean squared error. By doing this we make some assumptions on the underlying distribution of the errors. Furthermore sometimes we want to obtain a full probability distribution instead of just the expected value, i.e. we want to model $P(Y|X)$. To do so effectively we need to make some assumptions on the shape of the probability distribution. We could for example assume the errors to be normally distributed with constant variance. Write $\hat{P}_W(y|\mathbf{x})$ for the estimate of the probability density function of Y given $X = \mathbf{x}$. Then our estimate has the form

$$\hat{P}_W(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y-W\mathbf{x})^2}{2\sigma^2}}, \tag{2}$$

for some value of σ. Similarly to before we need to optimize some value to fit the parameter W. The paradigm that is generally used for this task is *maximum likelihood estimation*. The probability, or likelihood, of our predictands being equal to y_i given predictors $\mathbf{x}_i$, for $i = 1, .., N$ is given by:

$$LH(W) = \prod_{i=1}^{N} \hat{P}_W(Y = y_i|X = \mathbf{x}_i) \tag{3}$$

We then want to find the value W which maximizes this value. i.e. $\hat{W} = \arg\max_W LH(W)$. To simplify this expression we generally use the logarithm instead. Since the logarithm is strictly increasing on $\mathbb{R}_+$, maximizing a strictly positive function is the same as maximizing the logarithm of one. Taking the logarithm has the advantage that it turns the product in a sum, which is generally easier to optimize. Define the log-likelihood $\mathcal{L}(W) = \log(LH(W))$. Then this is given by:

$$\mathcal{L}(W) = \sum_{i=1}^{N} \log(P_W(Y = y_i|X = \mathbf{x}_i)) \tag{4}$$

For an estimate $\hat{P}_W(Y = y|X = x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - W\mathbf{x_i})^2}{2\sigma^2}}$ the log-likelihood becomes:

$$\mathcal{L}(W) = \sum \log(\frac{1}{\sqrt{2p\sigma^2}} e^{-\frac{(y_i - W\mathbf{x_i})^2}{2\sigma^2}}) = C_1 - C_2 \sum (y_i - W\mathbf{x_i})^2. \tag{5}$$

Where the constants $C_1 \geq 0$ and $C_2 > 0$ are dependent on σ but not on W and hence not of interest for maximizing the log-likelihood. We can therefore assume these constant to be zero and one respectively. Doing so we see that maximizing the log-likelihood for a normal distribution with a constant standard deviation, gives the same estimation for $\mathbb{E}[Y|X]$ as least squares regression.

In many cases these assumptions do not hold. We could for example have a situation where the variance is dependent on $\mathbf{x}$ as well, or the distribution might be skewed. An example of such a situation would be predicting a variable such as wind speed, where negative values are not possible. In this case other distributions such as a Weibull- or a truncated normal distribution could be used. The truncated normal distribution is obtained from a normal distribution by giving negative values zero probability. Let $\Phi(x)$ and $\phi(x)$ be the CDF and PDF of a standard normal distribution respectively. Then we define the probability density function of the truncated normal as:

$$f_{(\mu,\sigma)}(x) = \frac{\frac{1}{\sigma}\phi(\frac{x-\mu}{\sigma})}{1-\Phi(\frac{\mu}{\sigma})}. \tag{6}$$

In this case fitting the parameters W based on log-likelihood gives a result for $\mathbb{E}[Y|X = \mathbf{x}]$ which is different from the result one would get using least squares.

We can also choose to let the assumption on the standard deviation go. If we for example have a model where the standard deviation increases proportionally to $\|\mathbf{x}\|$, then the least squares method would focus more on minimising the squared error far away from the origin, which might lead to a large bias relative to the variance close to the origin. The maximum-likelihood estimation with a second set of parameters W_σ is capable of modeling these errors relative to estimated variance. This should give a better result in such a scenario.

We can extend linear regression to be able to model non-linear dependencies by transforming the predictors $\mathbf{x}$ using some non-linear function, a so called linear basis expansion. Let $h : \mathbb{R}^m \to \mathbb{R}^M$ be a transformation from our predictor space to $\mathbb{R}^M$. We can then use linear regression on the transformed M-dimensional predictors $h(\mathbf{x})$. I.e. let $W \in R^M$, then model the relation between predictors $\mathbf{x}$ and predictands y by:

$$f(\mathbf{x}) = Wh(\mathbf{x}).$$

If we would for example want to model quadratic relations between $\mathbf{x}$ and y we could define $h : \mathbb{R}^m \to \mathbb{R}^{m^2}$ as:

$$h(\mathbf{x})_{i+mj} = \mathbf{x}_i\mathbf{x}_j \text{ for } i, j \in \{1, .., m\}.$$

For many different tasks we need different basis expansions, and selecting the right map h can be a difficult task. In section 3 we will introduce neural networks as a method to let such a map be selected based on training data.

2.3 Quantile Regression Forest

Parametric methods described above are dependent on good a-priori assumptions on the relationship between the predictors and the predictand. Non-parametric methods are methods that try to model the dependencies between predictors and predictand using minimal assumption. Examples of such methods

are nearest neighbour methods and decision trees. In this section we will first describe decision trees and then go on to quantile regression forests.

2.3.1 Decision Trees

A *decision tree* splits the domain into rectangles based on data, such that some measure is minimized. This splitting is done iteratively, using a greedy algorithm, starting with the split which gives the highest gain. This process can be visualized as a tree graph, where the root is the full dataset $\mathcal{D}$ and every leaf represents a division of the rootset according to some simple rule. For example let the root be the full dataset $\mathcal{D}$, then the two branches represent either $\{(\mathbf{z}, y) \in \mathcal{D} \| \mathbf{z}_i \leq c\}$ or $\{(\mathbf{z}, y) \in \mathcal{D} \| \mathbf{z}_i > c\}$ for some c and $i \in \{1, .., m\}$. After such a split we have two leafs which together divide the data in the root. This set can be splitted again into two branches similarly to what is done for the root. Figure 1 shows schematically what this looks like for a simple 2-dimensional example.

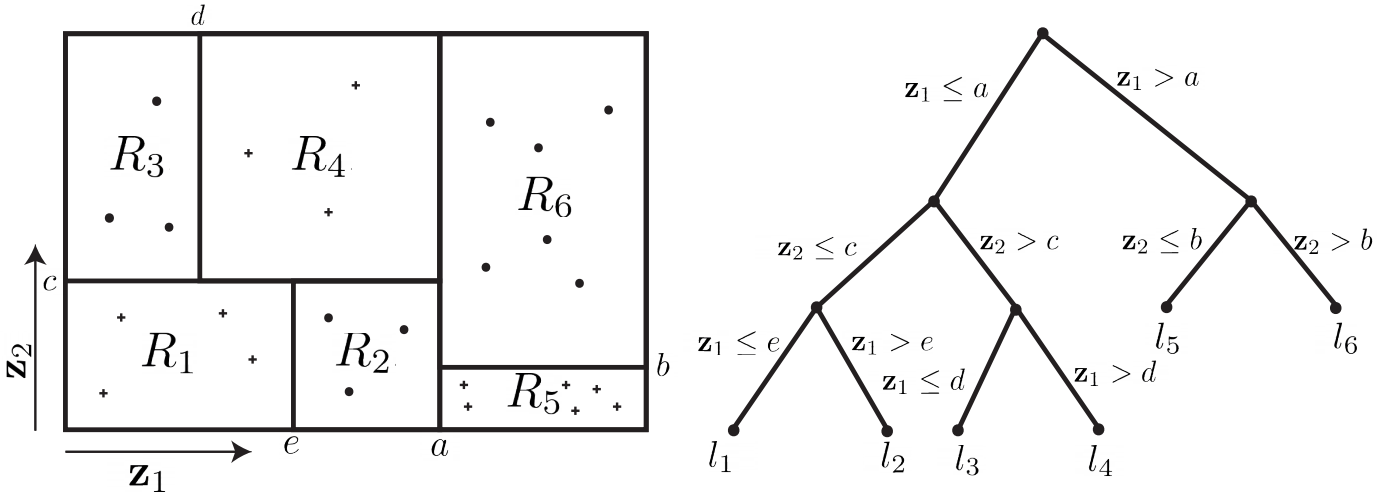

Figure 1: Schematic representation of a decision tree splitting up a two dimensional space into six different rectangles, such that every rectangle contains a single category.

To be more precise, let $\mathcal{X}$ be the space in which the predictors $\mathbf{x}$ live and $\mathcal{Y}$ the space in which the predictands live, furthermore let $\mathcal{D}$ denote our training dataset. We then want to subdivide $\mathcal{X}$ into L rectangles, R_l for $l = 1, ..., L$, such that for all $x \in \mathcal{X}$ there exist a unique R_l such that $x \in R_l$. Every rectangle is then associated with a value in the predictand space, $\mu_l \in \mathcal{Y}$. When y is categorical the value μ_l associated with every rectangle is the most common category in the set $\mathcal{D} \cap (R_l \times \mathcal{Y})$. i.e.

$$\mu_l = \arg\max_{y \in \mathcal{Y}} \sum_{i=1}^{N} \mathbb{1}_{R_l}(\mathbf{x}_i) \mathbb{1}_{\{y\}}(y_i)$$

Similarly when y is a continuous variable we can assign the following value to the rectangle R_l

$$\mu_l = \frac{\sum y_i \mathbb{1}_{x_i \in R_l}}{\#\{x_i \in X \text{ s.t. } x_i \in R_l\}}, \tag{7}$$

i.e. the average value of the predictands from the training data whose corresponding predictors are part of the rectangle. Now write $l(\mathbf{x})$ for the index of the rectangle that $\mathbf{x}$ belongs to, i.e. $l(\mathbf{x}) = \sum_{i=1}^{L} i\mathbb{1}_{\mathbf{x}\in R_i}$. The decision tree T then defines a mapping $T : \mathcal{X} \to \mathcal{Y}$ s.t. $T(\mathbf{x}) = \mu_{l(\mathbf{x})}$.

Splitting up $\mathcal{X}$ into rectangles is done as follows: Let $l_k = \mathcal{D} \cap (R_l \times \mathcal{Y})$ be the leaf representing the subset of the training-data for which $\mathbf{x}_i \in R_k$. Our goal is to split R_k into two new rectangles $R_{k'}, R_{k''}$. This split is defined by a tuple (j, t), where $j \in \{1, ..., m\}$ is the dimension along whihc the split happens and t is a threshold value. The new rectangles are then defined by $R_{k'} = \{\mathbf{z} \in R_k \text{ s.t. } \mathbf{z}_j < t\}$ and $R_{k''} = R_k \setminus R_{k'}$. The tuple (j, t) is chosen such that the quantity

$$G(R_k, (j, t)) = \frac{n_{k'}}{n_k} H(R_{k'}) + \frac{n_{k''}}{n_k} H(R_{k''}) \tag{8}$$

is minimised with respect to (j, t), where $n_{k'} = |X \cap R_{k'}|$ and H is generally referred to as the impurity. For classification problems H is often taken to be either the Gini-Index or Cross entropy, for regression tasks H is typically chosen to be the average squared error or the average absolute error, which, for a given rectangle R, is defined by

$$H_{mse}(R) = \frac{1}{n_k} \sum_{\mathbf{x}_i \in R} (\mathbf{x}_i - \mu_k)^2. \tag{9}$$

For a large enough number of splits a decision tree will only contain nodes with a single datapoint $(\mathbf{x}_i, y_i)$ from our training set. Meaning that without restrictions the model will learn something similar to a 1-nearest neighbour approach. This will in practice lead to overfitting. To prevent this we can limit the minimum leaf size, i.e. only allow a new split whenever $n_{k'}$ and $n_{k"}$ are larger than some predefined number.

2.3.2 Random Forests

A *random forest*[8] is an ensemble of decision trees where some randomness is added to the tree building process. This is done in two steps. First we use a different bootstrapped sample of the training data for every decision tree in the random forest. Secondly before every split only a randomly chosen subset of the predictors is considered for splitting. In this way we get a large number of predictions, which are decorrelated up to a point. We can average over these predictions to obtain a better more robust prediction. More precisely let θ_i represent the parameters that describe how tree i is grown and write $T(\theta_i)$ for the corresponding tree. For a random forest containing k decision trees the random forest defines a mapping $\hat{\mu} : \mathcal{X} \to \mathcal{Y}$ s.t.

$$\hat{\mu}(\mathbf{x}) = \frac{1}{k} \sum_{i=1}^{k} T(\theta_i)(\mathbf{x}). \tag{10}$$

Or for classification problems

$$\hat{\mu}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{i=1}^{k} \mathbb{1}_{\{y\}}(T(\theta_i)(\mathbf{x}))$$

2.3.3 Quantile Regression Forests (QRF)

The method explained above gives an estimation of $\mathbb{E}[Y|X = \mathbf{x}]$. Meinshausen [31] proposed a method of obtaining an estimate of a full conditional cumulative density function, $F(y|x)$, using random forests. In this method, called *Quantile Regression Forrest*, hereafter referred to as QRF, we use the same algorithm as described above for building a number of trees. However instead of associating a single value μ_l to every leaf, every leaf is associated with an empirical cumulative distribution function, defined by

$$\mu_l(y) := \sum_{i=1}^{N} \frac{\mathbb{1}_{[y_i,\infty)}(y)\mathbb{1}_{x_i \in R_l}}{\#\{x_i \in X \text{ s.t. } x_i \in R_l\}}. \tag{11}$$

Similar to before we can subsequently average over the results of all the trees to obtain a final estimate for the cumulative density function $F(y|x)$, which is given by

$$\hat{F}(y|\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} T(\theta_i)(\mathbf{x})(y). \tag{12}$$

2.4 Forecast Verification

Once we have a forecast we want to be able to determine how well it performs compared to other methods. To do this another dataset is needed which is independent from the training dataset. There are a number of different scores which we can use to compare forecasts, all with slightly different properties. Common deterministic scores, which focus on $\mathbb{E}[Y|X]$, are the mean absolute error and the mean squared error. Let $\hat{f}(\mathbf{x})$ be the an estimate for $\mathbb{E}[Y|X]$, than the mean squared error and the mean absolute error are given by $\frac{1}{N}\sum_{i=1}^{N}(|f(\mathbf{x}_i) - y_i|)$ and $\frac{1}{N}\sum_{i=1}^{N}(f(\mathbf{x}_i) - y_i)^2$ respectively.

We also want to be able to analyze probabilistic forecasts using scores that take the full probability distribution $P(Y|X)$ into account. There are two scores which are often used to assess the quality of a probabilistic forecast. The first is the Brier score. The Brier score does not look at the full probability distribution $P(Y|X)$, but only at the cumulative distribution function at a certain benchmark value. In doing so it turns a continuous problem into a 2-category classification problem. Let $\hat{F}$ be the estimated cumulative distribution function and let y be an observation. For a certain benchmark value c we then define the Brier score as

$$BS_c(\hat{F}, y) = ((1 - \hat{F}(c)) - \mathbb{1}_{[y,\infty)}(c))^2. \tag{13}$$

Comparing the Brier scores for different benchmark values can tell us in which ranges of the predictand a forecasting model performs well. A method that takes into account the full shape of the probability distribution is the continuous ranked probability score (CRPS). Again let $\hat{F}$ be the estimated cumulative distribution function and let y be an observation. The CRPS is than defined as

$$CRPS(\hat{F}, y) = \int_{\mathbb{R}} (F(c) - \mathbb{1}_{[y,\infty)}(c))^2 dc. \tag{14}$$

This is equal to the integral of the Brier score over all benchmark values. For deterministic forecasts this measure is equal to the mean absolute error. The CRPS can therefore be seen as a generalization of the mean absolute error for probabilistic forecasts.

Another method to asses whether a probabilistic forecast fits the data well is a *rank histogram*. A rank histogram is not concerned with the resolution of the data as the other methods are, but looks at how well the probabilities agree with the data, the so called calibration. This is based on the fact that if a random variable Y has a cumulative distribution function F, then $F(Y) \sim U(0,1)$. Let $\hat{F}(\cdot|\mathbf{X})$ be an estimation of the cumulative density function of Y given X. If $\hat{F}$ is close to F, then the following function should approximately match the cumulative distribution function of a uniform distribution.

$$\hat{F}_{F(Y)}(z) = \frac{1}{N}\sum_{1=1}^{N} \mathbb{1}_{(\hat{F}(y_i|\mathbf{x_i}),\infty)}(z).$$

The closer $\hat{F}_{F(Y)}(z)$ is to a uniform distribution, the closer $\hat{F}$ is to the actual cumulative density function. Unfortunately even if $\hat{F}_{F(Y)}(z)$ is uniform we do not have any guarantees that the probability distributions obtained by our model are correct. If for, for example, low wind speeds our model is under-dispersed and higher wind speeds our model is over-dispersed the two effect could cancel each-other out. Furthermore even if $\hat{F}_{F(Y)}$ is perfectly uniform, than we still do not necessarily have a good model[18]. A model which would give a fixed probabilistic forecast based on past measurements, i.e. the climatology, will give a better calibrated forecast than a conditional probability density function whose mean is much closer to test measurements but whose spread is not wide enough. It is however a good diagnostic tool for analyzing the calibration of your model.

In statistical postprocessing the performance of a method, relative to a benchmark method, is often given by a skill score. For a given score, such as the Brier score or CRPS, the skill score of method a relative to method b is defined as:

$$SS = 1 - \frac{Score(a)}{Score(b)}. \tag{15}$$

Positive skill scores correspond to performances which are better than the method used as a benchmark. In meteorology this benchmark is generally chosen to be the climatology. Here the climatology is an estimate of the probability distribution $P(Y = y)$ which does not depend on the predictors $\mathbf{x}$.

2.5 Proper Scoring Rules

We want that metrics we use to assess the quality of a probabilistic forecast are actually optimized for a good estimate of the probability. Let $S(P, y)$ be a score for a probabilistic forecast P and corresponding event y. Furthermore write $S(P, Q) = E_{Y\sim Q}(S(P, Y))$ for the expected score when the distribution of your data Y is Q. A good verification metric should give the best score when $P = Q$. Scoring rules such that $S(Q, Q) < S(P, Q)$ for all $P \neq Q$ are called *strictly proper scoring rules*[15]. Optimizing your probabilistic forecast w.r.t. a score should be done with a score that has this property.

If we would for example try to model a Bernoulli distribution Q;

$$\begin{aligned} Q(Y = 0) &= 1 - q \\ Q(Y = 1) &= q, \end{aligned}$$

using an estimated distribution $\hat{P}$;

$$\hat{P}(Y = 0) = 1 - p$$
$$\hat{P}(Y = 1) = p,$$

and use the following scoring rule;

$$S(P, y) = -|P(Y = 1) - y|,$$

i.e. the Brier score with the absolute value instead of the square. Then the expected score is given by

$$S(\hat{P}, Q) = -q|\hat{P}(X = 1) - 1| - (1 - q)|\hat{P}(X = 1)| = -q(1 - p) - (1 - q)p.$$

The maximum score will then be obtained for

$$\arg\min_p S(P, Q) = \arg\max_p -q(1 - p) - (1 - q)p = \begin{cases} p = 0 \text{ if } q < 0.5 \\ p = 1 \text{ if } q > 0.5 \\ p \in [0, 1] \text{if } q = 0.5. \end{cases}$$

Using this scoring function would therefore give overconfident results. The actual Brier Score, however, has expectation:

$$S(\hat{P}, Q) = q(p - 1)^2 + (1 - q)p^2$$

Such that:

$$\frac{d}{dp}(q(p - 1)^2 + (1 - q)p^2) = 2q(1 - p) + 2(1 - q)p$$
$$\implies \arg\min_p S(P, Q) = q.$$

Meaning that the Brier skill score is minimized in the expectation when $\hat{P} = Q$.

For continuous variables there is a wealth of proper scoring rules as well. The two rules used in this study are the continuous ranked probability score and the negative log-likelihood.

Proposition 1. The CRPS is a proper scoring function.

Proof. Let $S(P, Q)$ be the expected negative CRPS for a probability measure P with respect to Q, then

$$S(P, Q) = -\int\int (P(X < c) - \mathbb{1}_{[y,\infty)}(c))^2 dc dQ(y)$$
$$S(P, Q) = -\int\int (P(X > c) - \mathbb{1}_{[c,\infty)}(y))^2 dc dQ(y)$$
$$S(P, Q) = -\int\int (P(X > c) - \mathbb{1}_{[c,\infty)}(y))^2 dQ(y) dc$$
$$S(P, Q) = -\int BS_c(P, Q) dc,$$

where $BS_c(P, Q)$ is the Brier score. The Brier score has an unique minimum when $P(Y < c) = Q(Y < c)$ for all values of $c \in \mathbb{R}$, therefore $S(P, Q) \leq S(Q, Q)$ with equality if and only if $P = Q$. □

The parameters in parametric density estimation described in section 2.2 are chosen by maximising the log-likelihood, or similar minimising the negative log-likelihood. Furthermore we will use the (negative) log-likelihood in conditional density estimate with neural networks.

Proposition 2. The negative log-likelihood is a proper scoring function.

Proof. Let $S(P, Q)$ be the negative log-likelihood for an estimated probability distribution P and data drawn from Q. Furthermore, assuming they exist, denote the probability density functions of P and Q as f and g respectively, then by Jensen's inequality.

$$\begin{aligned} S(P, Q) - S(Q, Q) &= -\int \log(f(y))g(y)dy + \int \log(g(y))g(y)dy \\ &= -\int \log(\frac{f(y)}{g(y)})g(y)dy \geq -\log \int \frac{f(y)}{g(y)}g(y)dy = 0 \end{aligned}$$

Where equality holds if and only if $f = g$. This means that the negative log-likelihood is minimized in expectation only for $P = Q$. □

3 Deep Learning

This section mainly follows Goodfellow et. al.[16]

3.1 Feedforward Neural Networks

In section 2 we gave a short description of linear and logistic regression. We furthermore mentioned how non-linear relationships could be obtained by transforming data first using linear basis expansions. Choosing good expansion for the problem at hand can be a difficult task. The idea behind neural networks is that they can learn such expansions based on the data.

Neural networks are based on the human brain. The idea is that there are a number of neurons which are connected to each other. Through these connections they can send information to other neurons. Mathematically neural networks can be seen as a generalization of linear regression. Let $\mathbf{x} \in \mathbb{R}^m$ be the input data. We can represent this input of a neural network as m neurons, where every neuron represent one of the m variables in the input vector. We can then connect every one of these neurons with the output neuron. Each of these connection is then assigned a weight w. The value of the output neuron y connected to the set of neurons $\mathbf{x}$ is then given by

$$y = \sum w_i x_i = W\mathbf{x}, \tag{16}$$

which corresponds to linear regression. Similar to logistic regression we can modify the output $W\mathbf{x}$ with a non-linear function, such as the logistic function, to get a non-linear relation between $\mathbf{x}$ and y. The idea behind neural networks is that we can add extra layers of neurons between the input and output layer. Write $\mathbf{h}^l$ for the l-th layer. Furthermore write W^l_{ij} for the parameter describing the relation between h^{l-1}_i and h^l_j. A neural network with k hidden layers, such as shown in figure 2 for k = 2, then is a function:

$$\begin{aligned} \mathbf{h}^1 &= W^0\mathbf{x} \\ \mathbf{h}^2 &= W^1\mathbf{h}^1 \\ &\vdots \\ y &= W^k\mathbf{h}^k \end{aligned}$$

Due to every step being linear this is equal to $y = W^k...W^0\mathbf{x}$, which is a linear map. The added benefit comes from adding a non-linear map f on top of every layer. i.e. pick some functions $f_l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ and define the neural network as,

$$\begin{aligned} \mathbf{h}^1 &= f_0(W^0\mathbf{x}) \\ \mathbf{h}^2 &= f_1(W^1\mathbf{h}^1) \\ &\vdots \\ y &= f_k(W^k\mathbf{h}^k). \end{aligned}$$

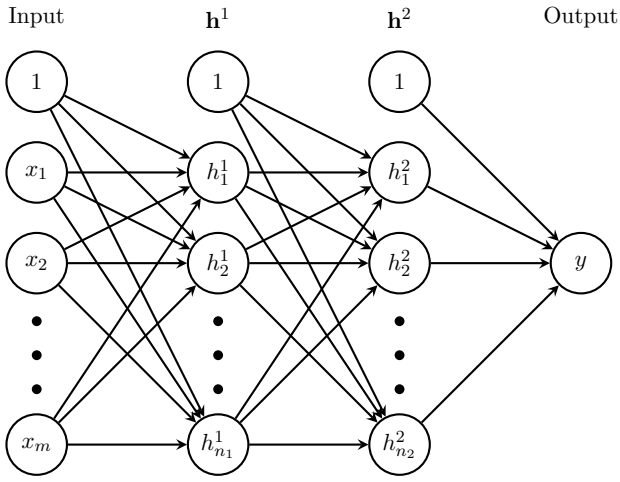

Figure 2: Example of a simple neural network with two hidden layers.

In simple cases f has the form $f(\mathbf{x}) = [\sigma(\mathbf{x}_1), .., \sigma(\mathbf{x}_n)]$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is called an activation function, which is applied to every neuron individually. Examples of activation that are commonly used are the *tanh* and rectified linear unit(*Relu*), defined as

$$tanh(x) = \frac{e^{2x-1}}{e^{2x+1}}$$
$$Relu(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0, \end{cases}$$

with the latter the most popular one. The non-linearity f can however take a more complicated form as well when combined with, e.g. batch normalization or pooling layers, described in section 3.3. Similar to what was done with linear regression, we can add bias terms to every layer. This is done by adding an extra neuron, with value one, to every layer. The function f_l, described above, becomes $f_l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l+1}$ s.t. $f(\mathbf{x}) = [1, \sigma(\mathbf{x}_1), .., \sigma(\mathbf{x}_n)]$, in this case. It can be shown that, when the activation functions σ agrees to a small number of properties, any continuous function can be approximated up to any predetermined precision by a neural network, given that the network has enough hidden units[21].

Now that we have an iterative definition of the function learned by a neural network we can focus on how to use it for a regression or classification task. In the linear models described in section 2 we optimized parameters through minimization of a loss function. The same will be done for neural networks. Let θ be the vector containing the parameters of our neural network, and $f(\theta, \cdot) : \mathbb{R}^n \to \mathbb{R}^k$ be the corresponding map defined by the neural network. As in section 2 write $\mathbf{x}_i, y_i$, with $i = 1, ..., N$, for the predictors and corresponding predictands in the training dataset $\mathcal{D}$. Furthermore let $J : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ be the loss function. And define a cost function C as the sum of the loss over the whole trainingset, i.e.

$$C(\theta) = \sum_{i=1}^{N} J(f(\theta, \mathbf{x}_i), y_i). \tag{17}$$

The goal then is to find a parameter $\hat{\theta}$, such that:

$$\hat{\theta} = \arg\min_{\theta} C(\theta).$$

To determine the optimal parameters $\hat{\theta}$ for the model $f(\theta, \mathbf{x})$ we need to find θ s.t.

$$\nabla_{\theta}(\sum_{i=1}^{N} J(f(\theta, \mathbf{x}_i), y_i)) = 0.$$

To do so we need to determine this gradient first. This is done through the back-propagation algorithm that will be described here. The derivative of $J(f(\theta, \mathbf{x_i}), y_i)$ depends on a parameter W_{ij}^l by,

$$\frac{dJ(f(\theta, \mathbf{x_i}), y_i)}{dW_{ij}^l} = \frac{dJ(f(\theta, \mathbf{x}_i), y_i)}{df(\theta, \mathbf{x_i})} \frac{df(\theta, \mathbf{x_i})}{dW_{ij}^l}. \tag{18}$$

We described earlier how a feed-forward neural network can be seen as the composition of a number of functions:

$$y = f(\theta, \mathbf{x}) = f_k \circ W^{k-1} f_{k-1} \circ ... \circ W^0 \mathbf{x}, \tag{19}$$

where $f_l : \mathbb{R}^{n_m} \to \mathbb{R}^{n_l+1}$ is a non linear function and $W^l : \mathbb{R}^{n_l+1} \to \mathbb{R}^{n_{l+1}}$ the weight matrix describing the connections in layer l. Furthermore, let Df_l be the total derivative of f_l and let $\mathbf{i}, \mathbf{j}$ define the i-th and j-th unit vector. Finally define $\mathbf{a}^l = W^l \mathbf{h}^l$. The inputs of the hidden layers h^i defined earlier are then given by

$$\mathbf{h}^l = f_l(\mathbf{a}^{l-1}). \tag{20}$$

We can then write the derivative of the values in layer $\mathbf{h}^l$ with respect to the parameters W_{ij}^l as

$$\frac{dh^l}{dW_{ij}^l} = Df_l(\mathbf{a}^{l-1}) W^{l-1} \frac{dh^{l-1}}{dW_{ij}^m} + \delta_{m(l-1)} Df_l(\mathbf{a^{l-1}}) \mathbf{i} \otimes \mathbf{j} h^{l-1}, \tag{21}$$

where $\delta_{k(l-1)}$ is the Kronecker delta. Now define the matrix $\delta^{k+1} = \frac{dJ(f(\theta, \mathbf{x_i}), y_i)}{df(\theta, \mathbf{x})}$ and for $l = k, ..., 1$ define $\delta^l = \delta^{l+1} Df_l(\mathbf{a}^{l-1}) W^{l-1}$. We can use this to write the derivative of $J(f(\mathbf{x_i}, \delta), y_i)$, with respect to any of the parameters W_{ij}^l, as

$$\frac{dJ(f(\theta, \mathbf{x_i}), y_i)}{dW_{ij}^l} = \delta^{l+2} Df_{l+1}(\mathbf{a}^l) \mathbf{i} \otimes \mathbf{j} h^l.$$

Now this allows us to efficiently determine the gradient by starting with the top layer and moving back. This algorithm is called the back propagation algorithm.

Algorithm 1: Backpropagation Algorithm

input : $\mathbf{x_i}$,y_i
begin
  Initialize: ;
  $\mathbf{h}^0 = \mathbf{x_i}$;
  for *l in 0,..,k-1* **do**
    $a^l = W^l h^l$;
    $h(l+1) = f_l(a^{l-1})$;
  end
  $\delta^{k+1} = \frac{dJ(f(\theta,\mathbf{x_i}),y_i)}{df(\theta,\mathbf{x_i})}$;
  for *l in k,..,1* **do**
    $\delta^l = \delta^{l+1} D f_l(\mathbf{a}^{l-1}) W^{l-1}$;
    for *i,j* **do**
      $g_{ij}^{l-1} = \delta^{l+2} D f_{l+1}(\mathbf{a}^l)\mathbf{i} \otimes \mathbf{j} h^l$;
    end
  end
  Return g (Here $g = \nabla_\theta J(f(\theta, \mathbf{x_i}), y_i)$)
end

The algorithm described above determines $\nabla_\theta J(f(\theta, \mathbf{x_i}), y_i)$. Summing over the full trainingset than gives use $\nabla_\theta C(\theta)$. Once we have determined the gradient of the cost function we can update the weights W_{ij}^l such that the loss over the trainingset is reduced. The algorithm that is commonly used for this is called adaptive moment estimation (ADAM)[23], which is describe in Algorithm 2. The gradient of the cost function is generally not determined over the whole training dataset before updating the parameters. Instead the training dataset is divided into subsets called batches. The gradient of the cost function is then determined based on a different batch every step.

Algorithm 2: ADAM[23]

begin

Set hyperparameters: ;

α : Stepsize ;

$\beta_1, \beta_1 \in [0, 1)$: Exponential decay rates for the moment estimates;

$C(\theta)$ cost function;

Initialize: ;

θ_0 initial parameter vector;

$m_0 = 0$ initial 1ste moment vector ;

$v_0 = 0$ initial 2nd moment vector;

$t = 0$ initial stimestep;

while θ_t *not converged* **do**

t = t+1;

$g_t = \nabla_\theta C_t(\theta_{t-1})$;

$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ (Update biased first moment estimate);

$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g^2$ (Update biased second raw moment);

$\hat{m}_t = m_t / (1 - \beta_1^t)$(Compute bias corrected second raw moment estimate);

$\hat{v}_t = v_t / (1 - \beta_2^t)$ (compute bias corrected raw moment estimate);

$\theta_t = \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$

end

Return θ_t;

end

3.2 Convolutional Neural networks

When we want to analyze images using machine learning techniques we run into problems with dimensionality. An input image of 100 by 100 pixels contains ten thousand input parameters. Many methods used in machine learning do not perform well for such high dimensional data. Dimensionality reduction techniques can be used to overcome this problem. Such techniques downgrade the image, to some low-dimensional representation, which still captures some of the structure. Examples of such methods are Tsne[29] and Umap[30]. We can also try to learn such a dimensionality reduction based on the training data using neural networks.

However, for a naive implementation of Neural Networks a lot of parameters are needed. For an input image of 100 by 100 pixels, a model with a hidden layer of 100 neurons will already need more than a million learnable parameters. To bring this number down we can use some a priori information on the structure of the problem.

Pixels in images have a clear spatial structure, it therefore makes sense to think about the input $\mathbf{x}$ not as a vector but as a matrix, (or tensor for multichannel inputs). We furthermore want part of this structure to be conserved when it is passed through the network. On top of that, it can be assumed, that pixels close to each other generally have a stronger relationship and might be more relevant to combine, than pixels that are far apart. Henceforth it makes sense to only connect neurons to their

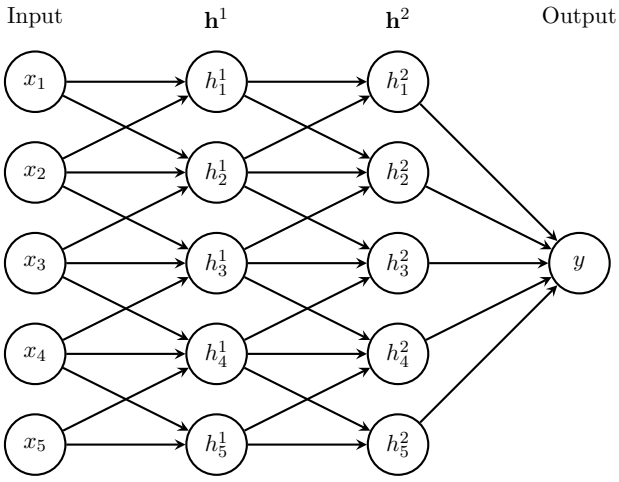

Figure 3: Example of a simple locally connected neural network.

nearest neighbours. This reduces the amount of necessary trainable parameters. More precisely let $W_{i,j,k,l}$ be the weight corresponding to the connection between $x_{i,j}$ and $h_{k,l}$, then

$$W_{i,j,k,l}W_{i+d_1,j+d_2,k+d_3,l+d_4} = 0 \text{ if } d_i > d, \text{ for any } i = 1, .., 4,$$

where d is the maximum distance of neighbouring interactions. Figure 3 shows an example of a simple locally connected network.

When we would not do this, the actual location of pixels in an image would not matter for the network. i.e. we could pick a random permutation of the input image and apply this to all the data. If a fully connected network would be able to learn to classify a certain set of images, it would also be able to do this for the permuted set of images. A model capable of such a task, can heuristically be seen as too powerful for our goal.

The second method to decrease the number of parameters is parameter sharing. In many image classification tasks we want to be able to determine whether certain object are present in the image, independent of where the object is located. i.e. we want the model to be invariant with respect to translations of the input. This can be achieved by using the same transformation everywhere on an input image, i.e. $W_{i,j,k,l} = W_{i+n,j+n,k+n,l+n}$ for all i, j, k, l, n.

Convolutional neural networks are currently the most popular paradigm in image recognition task. Convolutional neural networks use the two techniques described above in what is called a *convolutional layer*. A convolutional layer applies the same operation to every part of an input image. This operation is called a convolution. The parameters of the convolutional layer are given by $K_{i,l,m,n}$, which is called a kernel. let V be an input image for a convolutional layer and let $V_{i,j,k}$ be the element of V corresponding to channel i, row j and column k and let Z be the output of the convolutional layer with the same format. The output of the convolutional layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1}K_{i,l,m,n}. \tag{22}$$

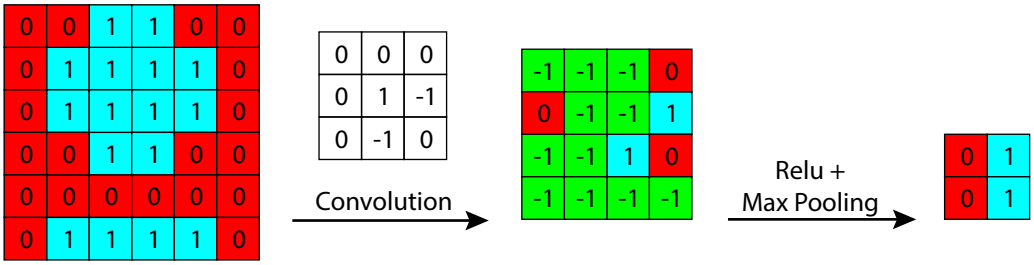

Figure 4: Simple example of a convolutional layer, followed by a Relu activation function and a max pooling layer. Here the kernel is given by $K_{1,1,0,0} = 1, K_{1,1,1,0} = -1, K_{1,1,0,-1} = -1$ and $K_{i,j,m,n} = 0$ otherwise.

Where l is summed over the number of channels in the output image, which is called the size of the convolutional layer, and m and n are typically summed over $\{-1, 0, 1\}$. Figure 4 shows a simple example of a convolutional layer with only a single input and output channel.

Convolutional layers are often followed by a max pooling layer. A max pooling layer takes the maximum value of every 2 by 2 block in the image. The combination of max pooling and convolutional layers reduces the size of an image, while trying to maintain as much relevant structure as possible.

3.3 Practical Considerations

Many of the architecture decisions that are made for neural networks are not statistical in nature, but are rather meant to both make sure that the gradient descent converges to a good minimum and make sure it converges fast enough.

There are multiple reasons why gradient descent might not converge to a good minimum; the main two being the gradient either vanishing or exploding. A vanishing gradient happens mainly when the activation functions have a zero derivative. A *tanh*, for example, has a zero derivative for values that are far from the origin. If, for some reason, the values of neurons are very large at the start of training, then the gradient will almost immediately go to zero, even though we are very far removed from an actual minimum. In such cases the gradient vanishes at a plateau and not at a local minimum.

To mitigate these problems, the initial estimates made by our model, should be such, that the gradient is not equal to zero from the start. Most popular activation functions have an approximately linear part. The hyperbolic tangent, for example, is approximately linear around the origin and the rectified linear unit (Relu) is linear for positive values. If, for every neuron in the neural network, the input value for the activation function falls in this range, then the model starts out as an approximately linear function.

To make sure the activations of the neurons fall in this linear range, the input data is standardized, i.e. we scale and translate all the input values $\mathbf{x}_i$ to be centered around zero, with standard deviation one. Furthermore the weights W have to be initialized. When we initialize the weights we want some asymmetry in the model. When all the weights of a model are equal, the gradient will be equal for all the weights as well, meaning that all the weights are updated in the same way and thus stay equal for all training steps. To avoid this the weights are generally initialized randomly.

The weights are often picked from either a uniform or a normal distribution with zero mean. If we want the model to be initialized such that most activations fall into the linear range we need to base the standard deviation of the randomly chosen weights on the number of neurons in every layer.

Let $\mathbf{a}^l = W^l\mathbf{h}^l$ and $\mathbf{h}^l = f(\mathbf{a}^{l-1})$. Where W^l represents the parameters in layer l, $\mathbf{h}^l$, the input of the layer and $f(\mathbf{a}) = [\sigma(\mathbf{a}_1), .., \sigma(\mathbf{a}_{n_{(l+1)}})]$ the activation functions, with n_l the number of neurons in layer l. Furthermore assume that the weights are realizations of a random variable w^l s.t. $\mathbb{E}[w_l] = 0$. Moreover write $\mathbb{E}[a_l], Var[a_l]$ for the expectation and variance of values in $\mathbf{a}^l$; and $\mathbb{E}[h_l], Var[h_l]$ for the expectation and variance of values in $\mathbf{h}^l$, which we assume to be identically distributed at the start of training. Then

$$Var[a_l] = n_l Var[w_l]\mathbb{E}[h_l^2]. \tag{23}$$

Now assuming our activations h_l fall in the linear range of the activation function σ, we find

$$Var[a_l] \approx n_l Var[w_l]\mathbb{E}[a_{l-1}^2]. \tag{24}$$

Now if σ is linear around zero, such as for example the $tanh$, and $\mathbb{E}[a_{l-1}] = 0$, then we get

$$Var[a_l] = n_l Var[w_l]Var[a_{l-1}].$$

Therefore we need $Var[w_l] = \frac{1}{n_l}$ to be sure that the variance of the activations in different layers stays roughly the same. If we would choose $Var[w_l]$ to be a constant c, independent from n_l, we would get $Var[a_l] = Var[a_0]\prod_{i=1}^{l} cn_i$. Which could, depending on c, increase or decrease exponentially.

The rule described above takes into account the forward pass through the network to make sure every activation is approximately in the linear range. However this does not take into account the backward pass through the model, which is used to compute the gradient. It is important to have all the gradients in approximately the same range at the start as well, otherwise weights will only be updated for the lower layers. Following [13] we get the following estimates for the variance in the gradients. Assume as before that we have $f(\mathbf{a}^l) = \mathbf{a}^l$. Furthermore denote by $Var[\delta a_l]$ and $Var[\delta h_l]$ the variance of $\frac{dJ}{\mathbf{a}_i^l}$ and $\frac{dJ}{\mathbf{h}_i^l}$, the derivatives of the loss function with respect to the individual neurons in layer l. We then get the following estimate for the variance of the gradient, in every layer;

$$\begin{aligned} Var[\delta h_l] &= n_l Var[w_l]Var[\delta a_l] \\ &= n_l Var[w_l]Var[\delta h_{l+1}]. \end{aligned}$$

Which means that the variance of the gradients vanishes or explodes if we do not have $Var[w_l] = \frac{1}{n_{l+1}}$.

If the variance of the weights are not initialised according to this rule, we will get either very large or very small gradients in the lower layers. This can lead to either divergence or to very small updates,

which slows down the learning process. Glorot initialization, which is commonly used and is also the default for, for example, Keras[9], makes a compromise between these two effects and initializes the model by drawing weights from a distribution with a variance given by

$$Var[w_l] = \frac{2}{n_l + n_{l+1}}. \tag{25}$$

Glorot initialization is based on two assumptions. First $\mathbb{E}[h_l] = 0$ and secondly $\sigma'(a_l) = 1$. These assumptions are violated when we use the rectified linear unit as an activation function, as described in [20]. For a rectified linear unit we have $\sigma'(h_i) = 0$ for negative h_i. Similarly we do not have the relation $\mathbb{E}[a_{l-1}^2] = Var[a_{l-1}^2]$, instead we have:

$$\mathbb{E}[a_{l-1}^2] = \frac{1}{2} Var[a_{l-1}]. \tag{26}$$

Meaning that when Relu activation functions are used it is better to initialize your weights according to a distribution with a variance given by:

$$Var[w_l] = \frac{1}{n_l + n_{l+1}} \tag{27}$$

In practice however, the factor two does not matter much for shallow networks.

3.3.1 Batch Normalization

Batch normalization[22] is known to be an effective way of making the learning algorithm converge faster. A batch normalization layer fixes the bias and standard deviation at the point in the network where it is located.

When we train a neural network we generally update the parameters based on a different subset of the trainingdata every step. Such a subset is called a batch. Denote a batch by $\mathcal{B} \subset \mathcal{D}$. Furthermore let $a(\mathbf{x})$ be the input of the batch normalization layer, given an input $\mathbf{x}$ for the neural network. The batch normalization layer then first determines the mean $\mu_{\mathcal{B}} = \sum_{\mathbf{x}_i \in \mathcal{B}} a(\mathbf{x}_i)$ and sample variance $\sigma_{\mathcal{B}} = \sum_{\mathbf{x}_i \in \mathcal{B}} (a(\mathbf{x}_i) - \mu_{\mathcal{B}})^2$. The output of the layer is then given by

$$\hat{a}(\mathbf{x}_i) = \gamma\left(\frac{a(\mathbf{x}_i) - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}\right) + \beta, \tag{28}$$

where γ and β are learnable parameters. For $\gamma = \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$ and $\beta = \mu_{\mathcal{B}}$ the batch normalization layer is equal to the identity function. This means that a model can in theory learn to to ignore this layer, therefore the set of function a model can learn is not reduced by inserting such a layer.

Why batch normalization works as well as it does is a matter of debate. In [16] for example it is argued that the benefit of batch normalization lies in the fact that it decouples the first and second moment of different layers. Without batch normalization a small change in the parameters of, for example, the

first layer can result in large changes in the statistics of the activations in layers further down. Batch normalization forces the first and second moment to stay the same wherever the layer is put. This means that changes to the weights in lower layers do not affect the first and second moment of higher layers, stabilizing the training.

Similarly in [7] it is argued that the main benefit of Batch normalization lies in the ability to use a higher learning rate without destabilizing the network, which helps both in convergence rate and is argued to help the network converge to less sharp, better generalizable minima.

3.3.2 Dropout

There are a number of ways to prevent neural networks from overfitting, such as L^1/L^2 regularization, which penalizes the size of the weights. A simple and popular method which has proven to be effective is Dropout[40]. In dropout every training step a random number of connections is temporarily dropped, i.e. every weight W_{ij}^l is set equal to zero with a probability p. In this way every training step a slightly different sub-network is trained. The idea behind this is that for a network trained in this manner, inference is done not based on a single model but on an ensemble of models. Although it is not quite a bagging method, since the sub-models are not independent, it has the benefit that it is computationally more efficient than bagging since we do not need to train a large number of models.

Another benefit of dropout is that it makes sure that every neuron is in principle disposable. It therefore reduces the influence of any individual neuron, which has a weight reduction effect as well. For example, two input neurons with very strong correlation could be given large weights with opposite signs to create an dependence on the small noise between the two variables. When we use dropout this behaviour is discouraged since dropping any of the two weights means they can't cancel each other anymore.

At prediction time we need to average over all the submodels. A common heuristic used for this is to multiply the outputs by p. A strict theoretical foundation is lacking for this heuristic, but it is proven to be effective in practice.

3.3.3 Early Stopping

When training a neural network we generally have a validation dataset, which is independent from the training dataset, to tune hyperparameters. An important hyperparameter in deep learning is the number of Epochs. i.e. how many times the data is shown to the model and how long the gradient descent is continued. Early stopping is a method to effectively tune this hyperparemeter by just stopping the training process at the point where the validation score does no longer increase.

4 Conditional Density Estimation

4.1 Kernel Density Estimation

In conditional density estimation we try to model $P(Y|X) = \frac{P(X,Y)}{P(X)}$. We can do this in a non-parametric fashion using kernel density estimation to estimate both $P(X,Y)$ and $P(X)$. Here we do not assume any specific relation between the random variables X and Y. We furthermore do not assume any shape for the probability density of X and Y. A popular way to do this is to look at the space $\mathcal{X} \times \mathcal{Y}$, in which our data lives, and estimate the distribution directly from the density of points $(\mathbf{x}_i, y_i)$ in our dataset. This is done by assuming that the probability that a point $(\mathbf{x}, y)$, drawn from the same distribution as the training dataset, is proportional to the distance between this point and the other points in the dataset. In general this is accomplished by picking a so called kernel function $K(\mathbf{x})$ and placing it on top of your data points. More precisely; let $K(x)$ be mapping from $\mathbb{R}^n \to \mathbb{R}_+$ such that $\int K(x) = 1$ and let $\mathbf{x_i} \in \mathbb{R}^n$ for $i = 1, .., N$ be the data points. Then we estimate the probability density function of X to be given by:

$$\hat{p}(\mathbf{x}) = \frac{1}{wN} \sum_{i=1}^{N} \frac{1}{N} K(\frac{\mathbf{x_i} - \mathbf{x}}{w}), \tag{29}$$

where w is a predetermined bandwidth parameter. A typical shape for such a kernel K would for example be a Gaussian $K(\mathbf{x}_i - \mathbf{x}) = \frac{1}{\sqrt{2\pi}} e^{\frac{(\mathbf{x}_i - \mathbf{x})^2}{2}}$. The bandwidth w can be thought of as a smoothing parameter. A small bandwidth will give a spikier multimodal result, whereas a large bandwidth will give an over-dispersed result.

We can use this method to obtain a conditional probability estimate by estimating both the whole distribution $P(X,Y)$ and the distribution $P(X)$, using equation 29. A big drawback of this method is that we need a lot of data when $\mathbf{x}$ is higher dimensional. The method is based on the distance between points in our dataset and new measurements for $\mathbf{x}$. When the dimensionality goes up, the expected distance between two points drawn from a uniform distribution goes up $O(\sqrt{n})$. The resulting sparsity means that it is hard to get a proper estimate for both $P(X,Y)$ and $P(X)$ resulting in even bigger errors in the estimate for $P(X,Y)/P(X)$.

To resolve this problem it is generally better to assume some relationship between the variables which helps us in modeling $P(Y|X)$ more directly from $\mathbf{x}$. In section 2 we already described two methods of doing this through linear methods or random forests. We can also use neural networks for this task. In this section we will describe three different methods of obtaining such an estimate using neural networks.

4.2 Conditional Density Estimation With Neural Networks

4.2.1 Quantized Softmax

[32] A rather simple method of obtaining an estimate for the conditional density $P(Y|X)$ using neural networks is the quantized softmax[32]. In quantized softmax we try to approximate the probability distribution by a histogram. The neural network learns a mapping $f : X \to S^{m-1}$ where m is the number of bins. The normalization is achieved by applying a softmax activation function to the last layer,

$$Softmax(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^{m} e^{\mathbf{x}_j}}. \tag{30}$$

The m-dimensional output of the neural network can then be turned in a conditional probability density function estimate $\hat{p}(y|x)$ by

$$\hat{p}(y|x) = \sum_{i=1}^{n} \mathbb{1}_{a_i} Vol(a_i) f(x)_i,$$

where a_i are pre-determined bins, and $Vol(a_i)$ is the volume of a_i. The set of probability distributions that can be learned by such a mapping is controlled by how we choose the bins a_i.

To optimize this we need to pick a loss function which is minimized when samples are drawn from the true probability distribution. i.e. a proper scoring rule. In chapter 2 we described two of these rules namely the log-likelihood and the continuously ranked probability score.

4.2.2 Kernel Density Mixture Networks

Another method that can be used for probability density estimation is the Kernel Density mixture network, introduced by Ambrogioni [2]. The idea behind a kernel mixture network is based on the kernel method described above. However instead of estimating the full probability distribution $P(X, Y)$ we only use kernels to estimate the probability density function in the label space $\mathcal{Y}$. I.e. let $\mathcal{D}_Y$ be the set of measurements in the training data-set. Then we can estimate the probability distribution, using kernel density estimation, to be given by

$$\hat{p}(y) = \sum_{y' \in \mathcal{D}_Y} \omega_{y'} K(y - y'), \tag{31}$$

where in normal kernel density estimation, as described earlier, we would pick $\omega_{y'} = \frac{1}{|Y|}$. We can make this a conditional estimate by making the parameters ω dependent on $\mathbf{x}$. i.e.

$$\hat{p}(y|x) = \sum_{y' \in Y} \omega_{y'}(\mathbf{x}) K(y - y'). \tag{32}$$

The map $f : \mathbb{R}^n \to S^{|Y|-1}$ s.t. $f(x) = (\omega_{y_1}, ..., \omega_{y_{|Y|}})$ is the mapping we want the neural network to learn. To do this the output size of the neural network need to be equal to $|Y|$. This is in general

too large and unnecessary. To reduce the output size we could use a clustering method like K-means to select the kernel centers. The bandwidth of the kernels can be chosen beforehand or can be made dependent on $\mathbf{x}$ as well, which in [36] was found to perform better. In the the second case we need to double the amount of outputs of the neural network. Where the first n outputs are normalized using a softmax activation function and the other n outputs need to be positive to represent the bandwidth. An option for the activation function for the bandwidths could be a softplus activation function,

$$Softplus(x) = \log(1 + e^x). \tag{33}$$

This ensures that the bandwidths are positive and prevents them from becoming too small, which could cause numerical instability.

When we use a boxcar kernel $K(y_i - y) = \mathbb{1}_{\{z \in \mathbb{R} || |z| < 1\}}(|y_i - y|)$ this method becomes similar to the quantized softmax. A benefit of the kernel mixture method is however that we do not need to decide upon the range of the bins beforehand. Furthermore Gaussian Kernels could be beneficial for they give a smoother result than the boxcar kernels used in quantized softmax.

The loss function used for this method in literature is the negative log-likelihood. We can however also use the CRPS for this task. A closed form expression for the CRPS exist when the kernels are chosen to be Guassian. Let the probability density function of $\hat{P}$ be given by the sum of M different Gaussian kernels, then the CRPS is given by

$$CRPS(\hat{P}, y) = \sum_{i=1}^{M} \omega_i A(y - \mu_i, \sigma_i^2) - \frac{1}{2} \sum_{i=1}^{M} \sum_{j=1}^{M} \omega_i \omega_j A(\mu_i - \mu_j, \sigma_i^2 + \sigma_j^2), \tag{34}$$

where $A(\mu, \sigma^2) = \mu(2\Psi(\frac{\mu}{\sigma}) - 1) + 2\sigma\phi(\frac{\mu}{\sigma})$[17]. We could also make the centers μ_i dependent on $\mathbf{x}$. In this case we get a mixture density network [6].

4.3 Parametric Methods

The methods described above are non-parametric or semi-parametric. We could also use neural networks in combination with parametric methods. A description of conditional density estimation using linear methods was given in section 2. We could expand this by modelling the dependence between parameters θ and $\mathbf{x}$ with a neural network. Similar to before this might perform better than non-parametric methods, due to the fact that it is less flexible and therefore not as dependent on the amount of data we have. In the experiments performed in this thesis we used a truncated normal to model the probability distribution. In this case a neural network only needs to have two different parameters as outputs. μ and σ. The corresponding conditional probability density function estimate is then given by:

$$\hat{p}(y|\mathbf{x}) = \frac{\frac{1}{\sigma(\mathbf{x})}\phi(\frac{\mu(\mathbf{x}) - y}{\sigma(\mathbf{x})})}{1 - \Phi(-\frac{\mu(\mathbf{x})}{\sigma})}, \tag{35}$$

where ϕ and Φ are the probability density function and cumulative density function, of a standard normal distribution, respectively. Similar to before we can use the log-likelihood and CRPS. For a

normal distribution truncated at zero the expression for the CRPS is as follows[42],

$$CRPS(\mathcal{N}_0(\mu.\theta), y) = \frac{\sigma}{p^2}\{sp[2\Phi(s) + p - 2] + 2p\phi(s) - \frac{1}{\sqrt{\pi}}\Phi(\frac{\mu\sqrt{2}}{\sigma})\}, \tag{36}$$

where $p = \Phi(\frac{\mu}{\sigma})$ and $s = \frac{y-u}{\sigma}$. Similar to what is described for the Kernel Density networks one wants the standard deviation to be positive and non-zero. A softmax activation function is therefore a good choice here as well. Furthermore one needs to be careful with the initialization of the model. An initial estimate for the standard deviation can, especially when using negative log-likelihood as loss function, lead to exploding gradients and as a result a diverging model. If the CRPS is used as a loss function, the initialization is less of an issue. This is due to the fact that for a point forecast the CRPS is equal to the mean absolute error, for which the derivative is constant.

To examine the difference between CRPS and log-likelihood, when the estimated probability density function and corresponding measurement are far apart, i.e. $s >> 0$, we can determine the derivative of both these loss functions. The CRPS and the log-likelihood of a normal distribution with respect to a measurement y are given by,

$$\begin{aligned} CRPS(\mathcal{N}(\mu, \sigma), y) &= \sigma[\frac{1}{\sqrt{\pi}} - 2\phi(s) - s(2\Phi(s) - 1)] \\ \mathcal{L}(\mathcal{N}(\mu, \sigma), s) &= \frac{1}{\sigma}\phi(s). \end{aligned}$$

The derivative of the CRPS with respect to σ is then given by

$$\begin{aligned} \frac{dCRPS(\mathcal{N}(\mu, \sigma), y)}{d\sigma} &= \frac{1}{\sigma}CRPS(\mathcal{N}(\mu, \sigma), y) + \sigma\frac{dCRPS(\mathcal{N}(\mu, \sigma), y)}{ds}\frac{ds}{d\sigma} \\ &= \frac{1}{\sigma}CRPS(\mathcal{N}(\mu, \sigma), y) + \sigma[4s\phi(s) - 2\Phi(s) + 1 - 2s\phi(s)]\frac{-s}{\sigma}. \end{aligned}$$

We have $\lim_{s\to\pm\infty}\phi(s) = 0$ and $\lim_{s\to\pm\infty}(2\Phi(s) - 1) = \frac{s}{|s|}$. This means that

$$\lim_{s\to\pm\infty}\frac{dCRPS(\mathcal{N}(\mu, \sigma), y)}{d\sigma} = \lim_{s\to\pm\infty}|s|(\sigma + 1). \tag{37}$$

Similarly we have

$$\begin{aligned} \frac{dCRPS(\mathcal{N}(\mu, \sigma), y)}{d\mu} &= \sigma\frac{dCRPS(\mathcal{N}(\mu, \sigma), y)}{ds}\frac{ds}{d\sigma} \\ &= \sigma[4s\phi(s) - 2\Phi(s) + 1 - 2s\phi(s)]\frac{-1}{\sigma}. \end{aligned}$$

Thus for large values of $|s|$ we get

$$\lim_{s\to\pm\infty}\frac{dCRPS(\mathcal{N}(\mu, \sigma), y)}{d\mu} = \pm 1. \tag{38}$$

Meaning that the derivatives of the CRPS are either linear in s or constant when $|s|$ is large. If we would do the same for the log-likelihood we obtain

$$\frac{d\mathcal{L}(\mathcal{N}(\mu,\sigma),y)}{d\mu} = \frac{y-\mu}{\sigma^2} \tag{39}$$

$$\frac{d\mathcal{L}(\mathcal{N}(\mu,\sigma),y)}{d\sigma} = -\frac{1}{\sigma} + \frac{(y-\mu)^2}{\sigma^3}. \tag{40}$$

This implies that the gradient of the log-likelihood grows much faster if $\sigma << |(\mu - y)|$, with the result that bad initial estimates of μ and σ lead to a diverging model. Furthermore this also implies that the CRPS is less sensitive to outliers. The CRPS has to my knowledge not been used for Kernel Mixture networks or quantized softmax. It has however been applied with success to density mixture networks in [10] and [37].

5 Experiments

The main aim of this study is to determine whether convolutional neural networks can provide added value for statistical post-processing. We will focus on the post-processing of wind speed forecasts in the months October to March, in the Netherlands. We will compare QRF, which is shown by Taillardat et al [41] to outperform EMOS methods based on parametric density estimation as described in section 2, to both neural networks and convolutional neural networks. Furthermore three different methods for obtaining probability densities using convolutional neural networks, as described in section 4, are compared.

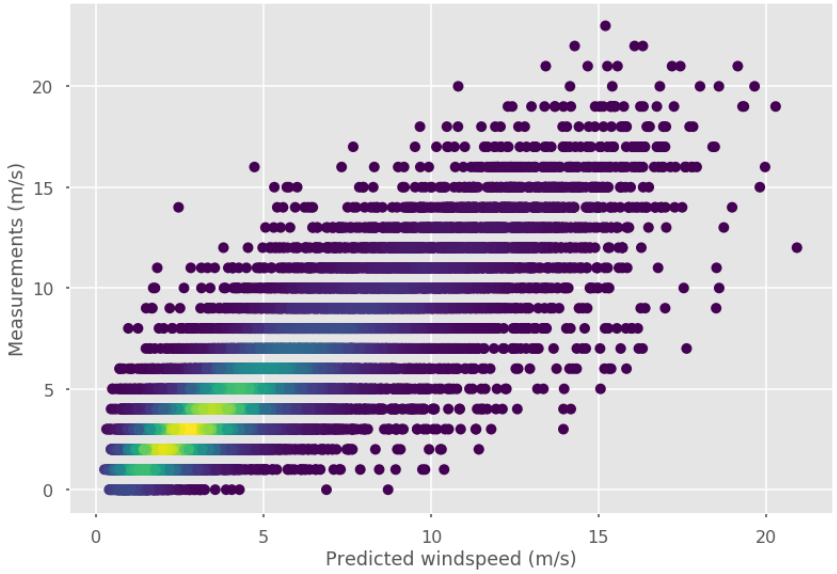

Figure 5: Wind speeds as predicted by HA40 for a lead time of 48H, compared to corresponding measurements. Measurements from all weather stations are pooled in a single set.

5.1 Data

The input data X is provided by Harmonie-Arome cycle 40 (HA40) used by KNMI. We will focus on forecasts at 0000UTC at a lead time of 48 hours. The predictand data Y, are the 10-minute-average wind speed observations at 10 meters above the ground, from 44 different weather stations in the Netherlands, which are shown in Figure 6. These measurements are provided as rounded to the nearest m/s. Figure 5 shows a scatterplot between wind speed as predicted by HA40 and the corresponding measurements in our data set. The observations from all the stations are pooled in the training dataset, meaning that the model is trained for all stations at once, without providing station specific information other than the surface roughness.

Reforecast data for HA40 is available from 2015 until 2017. Furthermore forecasts from 2019 are available. This data will be split into two sets, as shown in Table 6. The first set is used for model

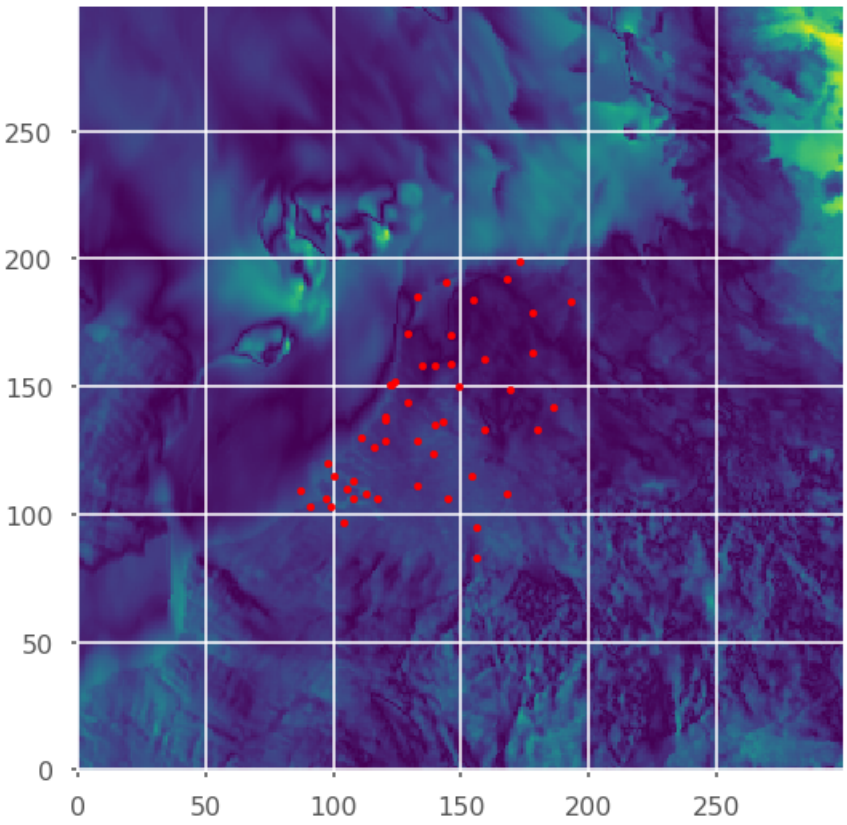

Figure 6: Example of a wind speed forecasts provided by HA40. The red dots give the locations of the Dutch weather stations used in this study.

selection and training (2015-2017). The second set will be an independent data set used for testing the selected models (2018-2019). We furthermore split the set used for model selection into three different independent sets, which is used for cross-validation.

Model Selection	Fold 1	October - December 2015 and January - March 2016
	Fold 2	October - December 2016 and January - March 2017
	Fold 3	October - December 2017 and January - March 2015
Test set	November - December 2018, January-March 2019 and October - November 2019	

Table 1: Definition of the different subsets used in cross-validation and testing.

In cross-validation we trained every model three times on the model selection set with a different fold left out. This data was then used to make predictions on to test the model. The sets are chosen in this way to ensure that there is at least six months between the training, test and validation sets. This is necessary to avoid temporal correlations between the different data sets.

5.2 Predictors and Predictor Selection

In this research we use two sets of predictors. The first set contains the HA40 forecasts of a number of variables in the neighbourhood of the station, which provides the corresponding observation. The second set contains the wind speed forecast from HA40 for a large area around this station, see Figure 6 for an example. The first set is used in all the methods described. The second set is only used for convolutional neural networks alongside the other set.

The set with local predictors we use in this study is based on previous research on post-processing of wind speed forecasts by [11] and [41]. Based on their results we decided to take the variables shown in BOX 1 into account for the predictor data.

These variables were taken from HA40, which forecasts each such variable up to a resolution of 2.5 by 2.5 kilometer. The 2.5 by 2.5 kilometer grid point closest to the station is used for the surface roughness. For the other variables we picked a number of gridboxes around the station and determined the mean value, maximal value and minimal value of each predictor in this region. The number of gridboxes used, and whether to take the mean, maximum, minimum or a combination of them is decided through a hyperparameter search. The predictors that gave the best results for our methods are given in BOX 2

Some of the models were trained on the errors of linear regression instead of on the measurements directly. In the section below I will further explain why this was done. The linear regression is fitted on the mean values for the variables 2,3 and 4 as defined in Box 1 on an area of 12.5 by 12.5 km around the station. These variables are selected through a greedy algorithm which adds predictors stepwise based on which predictors reduce the mean squared error the most. The score does not improve significantly after these predictors have been selected and therefore we left other variables out. These variables are however, used in the non-linear methods, as they improve the results there.

1. Wind direction at a height of 10 m;
2. Wind speed at a height of 10 m;
3. Surface roughness;
4. Meridional/zonal wind components at 925 hPa;
5. Mean sea level pressure;
6. Total Kinetic Energy;
7. Humidity at surface level;
8. Geopotential height 500 hPa;
9. Temperature at surface level;
10. Meridional and zonal windcomponents at 850 hPa;
11. day of the year;

Box 1: Predictors considered in our hyperparameter search. The wind directions are obtained by dividing the Meridional and Zonal (North-South, East-West) components of the wind speed by the total wind speed.

1. Wind direction at a height of 10 m;
2. Wind speed at a height of 10 m;
3. Surface roughness;
4. Meridional/zonal wind components at 925 hPa;
5. Mean sea level pressure;

Box 2: Predictors that gave the best performance in cross-validation

5.3 Methods

The methods we compared are Quantile Regression Forests, Fully Connected Neural Networks and Convolutional Neural Networks. We furthermore try different methods of obtaining a conditional density estimate for the convolutional neural networks.

5.3.1 Quantile Regression Forests

We used the package Sci-kit garden to model quantile regression forests.[33] Within this package there is no option to obtain a full cumulative density function however, therefore an alternative predict function was used which determined the full cumulative density function as described in section 2. For quantile regression forests the most important architectural choices that need to be made are based on the minimum leaf size of the trees, and the amount of randomization.

We can control the randomization in the Random Forest by picking a random subset of predictors to consider for splitting at every step. In practice the best results were obtained by using the full set of predictors however. Hence the decorrelation between the trees occurs only through bootstrapping on the trainingset.

Other hyperparameters that I looked at are the impurity function and the number of trees. For the latter we used 100 trees in the first hyperparameter search and 500 for the final model. For the impurity function I compared the mean squared error to the mean absolute error, and found the former to give the best results.

We furthermore used quantile regression forest both on the observed wind speed data and on the residuals of linear regression. The second approach could be beneficial for two reasons. The first reason is that quantile regression forests cant extrapolate outside of the range of the training data. Linear regression however, can extrapolate from the data to work outside the range of our training data as well. By combining the two we can potentially have a model that is better for higher wind speeds.

The second reason is that regression forests split the data into boxes based on which split minimizes the total impurity the most. If one of the predictors has a much stronger correlation with the predictand than the other ones, then the first number of splits will be based on this predictor. The other predictors will not be taken into account when this decision is made. This could result in the first number of splits learning the dependency between the forecasted wind and measured wind, which is approximately linear. Only when no improvements can be made anymore by splitting based on wind speed predictions the other variables are taken into account. At this point, however, the leaf sizes have become rather small, which means there is less information to learn the residuals, especially for higher wind speeds where the amount of data is already limited. The hypothesis is that by performing linear regression first and using QRF to model the residuals, the model could perform better for higher wind speeds and could potentially be better capable of estimating the uncertainty by having more data to model the residuals.

The best models as determined by the hyperparameter search have the following characteristics. The predictor data contained the maximum, minimum and mean value of the predictors described in box 2. We have used the mean squared error as the impurity. For the random forest trained on the wind speed measurements, hereafter referred to as QRF, we used a minimum leaf size of 30. For the random forest trained on the residuals of linear regression hereafter refered to as QRF_LR, we have used a minimum leaf size of 42.

5.3.2 Neural Networks

The neural networks used in this research were programmed using Keras [9], with Tensorflow as backend [1]. For the fully connected neural networks a quantised softmax output layer was used to estimate the probability distributions. Similar to QRF this was done for both the observations and the residuals of linear regression. In this case first using linear regression was hypothesized to give better results due to the fact that lower wind speeds are way more prevalent in the training data set. Output neurons which are related to high wind speeds therefore need to be activated in only a very small sample of the data. Oversampling the data, such that training samples corresponding to high wind speed days were shown to the network more often during the training phase, was tried, but this appeared to have a negative impact on the results. This is probably due to the fact that this led to a large number of copies for outliers in the trainingsset which do not generalise well. Less naive oversampling methods with data augmentations might be more useful, but were not tried.

For the neural networks we have explored networks with n layers of size m followed by a Relu activation function and a dropout layer, i.e. every neural network is a stack of n of the following blocks:

Fully connected layer
Relu
Dropout

After the n-th block we put the output layer, which has size 30 for the neural networks trained on the wind speed measurements, where every node represents a different wind speed ranging from 0 to 29 m/s. For the neural network which learns the residuals of linear regression, we use 300 output neurons. Here every neuron represents a different value for the residual ranging between -15 and 15 m/s. In both cases ADAM was used as the optimizer using default options for the parameters other than the learning rate.

The hyperparameter search was performed on the number of layers n and layer size m, the dropout rate, L1 regularization strength, learning rate, learning rate decay parameter, optimizer and loss function. We furthermore checked the same potential predictor variables as with QRF.

The neural network appeared to give the best results, when trained on the wind speed measurements themselves, hereafter referred to as NN, uses the maximum and mean value of the input variables 1,2 and 3 from Box 2. The neural network trained on the residuals, hereafter referred to as NN_LR, gave the best results when trained on the means of the predictor variables from Box 2 and the maximum

and minimum value of the wind speed. In NN_LR we added an extra hyperparameter which models the variance of Gaussian noise we added to the training labels. This is done to smoothen the results. L1 regularization appeared to not improve the results and was left out completely for both methods. The other hyperparameters are described in Table 2.

Hyperparameter	NN	NN_LR
Number of layers	2	3
Layer size	106	106
Learning rate	$3.47 * 10^{-3}$	$1.57 * 10^{-3}$
Dropout rate	0.030	0.188
Loss function	log-likelihood	log-likelihood
Decay parameter	$5.0 * 10^{6}$	$8.4 * 10^{4}$
σ^2 noise	0	0.315

Table 2: Hyperparameters for the selected models.

5.3.3 Convolutional Neural Networks

The convolutional networks are all trained on the residuals of linear regression. Convolutional neural networks trained on the observation were found to be not skilful in preliminary testing. This was partly due to the fact that networks, trained on the observations directly, took longer to converge and converged to bad values much more often than models trained on the residuals. Which resulted in a much slower hyperparameter search. This could be explained by the fact that the residuals are distributed around zero, which makes it easier to initialize the model. Furthermore the reasons coined for neural networks apply here as well.

An extra argument, which is more specific for convolutional neural networks, for applying linear regression first is that we can use local information to do this. A convolutional neural network is based on the translation invariance of the patterns it needs to learn. The wind speed we expect at a certain weather station is however very dependent on the wind speed forecast at the location of the station. The translation invariance of the convolutional layers is therefore not suited for predictions at a specific weather station. Features in the forecast that correlate to the bias and the uncertainty are however probably less local and therefore better suited to be analyzed using convolutional neural networks.

For convolutional neural networks three different methods of obtaining a conditional probability estimate are compared, see Chapter 4. Furthermore, a number of different architectures and hyperparameters are tuned. The convolutional neural networks are all fed two different inputs. The first input is the full spatial forecast of the wind speed for a certain region around the weather station, which provides the corresponding observation. This is the input which is fed into the convolutional part of the network. The second input contains the other variables averaged over the nearest grid boxes around the station similar to what is described for QRF and fully connected neural networks.

The convolutional part of the network consists of n_{conv} layers with m_{conv} filters. Each of these convolution layers is built as shown in table 3, where we use a step size of 2 by 2 for the Max Pooling layer

and a filter size of 3 by 3 for the convolutional layer. For the fully connected part of the network we stacked layers as shown in table 4 The final architecture then looks as shown in table 6.

Dense
Activation
BatchNormalization
Dropout

Table 3: Dense layer

Conv2D
Relu
BatchNormalization
MaxPooling2D

Table 4: Convolution layer

InputConvolution Convolution ⋮ Convolution	 Input
Dense	Dense
Dense ⋮ Dense	

Table 5: Convolutional neural network architecture

The size of the output layer of the convolutional neural networks depends on which conditional density estimation method is used. For quantized softmax, from here on referred to as CNN_LR the output layer has size 300 as is also used for NN_LR. For the truncated normal, from here on referred to as CNN_LR_N0, we only need two output neurons and for the kernel mixture network, from here on revered to as CNN_LR_KMN, we need two parameters for every kernel we use.

The hyperparameters looked at in the hyperparameter search and the selected values for each of these hyperparameters are shown in Table 6. In all these methods we used ADAM as optimizer using the default parameters except for the learning rate. No real difference in performance was observed for the CRPS and log-likelihood as loss functions, neither in training time nor in the final result. However for CNN_LR_N0 the CRPS proved to be more stable and therefore a better choice.

Hyper parameter	N0	KMN	QSM
Input grid	100	60	60
Variables	1,2,3,4,5	1,2,3,4,5	1,2,3,4,5
Layer_size	60	80	80
Number of convolutional layers	3	3	3
Size of convolutional layers	16	16	16
Learning rate	0.0013,	0.00053,	0.0007283,
Loss function	CRPS	CRPS	log-likelihood
Dropout rate	0.1028,	0.072,	0.0888,
Decay parameter	2.633e-06,	4.098e-5,	4.10e-07
Doise	0.315	0.26218	0.322
Number kernels	n/a	60	n/a

Table 6: Hyperparameters CNN

Method	CV1	CV2	CV3
NN	0.824	0.898	0.914
NN_LR	0.828	0.865	0.889
QRF	0.814	0.861	0.888
QRF_LR	0.819	0.871	0.900
CNN_LR_KMN	0.794	0.830	0.861
CNN_LR_N0	0.772	**0.806**	0.848
CNN_LR	**0.769**	0.810	**0.839**

Table 7: Continuous ranked probability score of different methods in cross-validation. Here CV1 used Fold 1 and Fold 2 as training set and Fold 3 as test set; CV2 used Fold 2 and Fold 3 as training set and Fold 1 as test set; and CV3 used Fold 1 and Fold 3 as training set and Fold 2 as test set.

5.4 Results

The CRPS results on the cross-validation for the best models are shown in table 7. These results show that convolutional neural networks outperform QRF on all three test sets in cross-validation. Hyperparameters are selected based on these results however, therefore as mentioned before, we need an independent test set the verify these results.

For the results on the independent set we made three different forecasts for every model. Each of these forecasts is based on the model trained on a different training set as used in the cross-validation. This is done to get an estimate of the variation in the results when different training data is used. In table 8 the results are shown for the root mean squared error, the mean absolute error and the CRPS. These results show that adding spatial information through convolutions adds skill to both the deterministic forecast(i.e. mean of the probabilistic forecast) and the probabilistic forecast. Furthermore we can see that applying linear regression first improves the deterministic forecast of both QRF and neural networks. It does however not improve the CRPS of QRF. In Figure 7 the Brier skill score relative to QRF is shown for the three different training sets. From this it is clear that convolutional neural networks are more skillful at higher wind speeds. For wind speeds above 18 m/s their performance becomes worse again, however in this range there is not enough data for any conclusions. Figure 7 also shows that learning the residuals of linear regression mainly helps for higher wind speeds, while for low wind speeds the results become worse for both neural networks and random forests. Figure 9 shows scatterplots between the expected value of the forecasts and the observations. Here we can see clearly that forecasts of the convolutional neural networks lie closer to the diagonal.

For QRF and CNN a final test was done with models trained on the whole training data set. For the convolutional neural networks the number of epochs was chosen to be 2/3rd of the average number of epochs that gave the best results in cross-validation, i.e. 6,12 and 16 for CNN_LR_N0, CNN_LR_KMN and CNN_LR, respectively. The results obtained for the convolutional network in this case are slightly worse than for the convolutional neural networks trained on only a part of the training data. This is probably due to the fact that for the other networks we could use an independent data set, to terminate training at the right moment, increasing the generalisability of the network. The results are shown in table 9. Here we see that convolutional neural networks still outperform QRF, albeit slightly

	rmse			mae			CRPS		
	CV1	CV2	CV3	CV1	CV2	CV3	CV1	CV2	CV3
NN	2.457	2.331	2.391	1.176	1.142	1.158	0.820	0.799	0.809
NN_LR:	2.204	2.126	2.176	1.109	1.090	1.099	0.793	0.779	0.786
QRF	2.244	2.220	2.245	1.116,	1.113	1.115	0.782	0.776	0.779
QRF_LR:	2.157	2.151	2.154	1.094	1.096	1.091	0.780	0.781	0.780
CNN_LR_KMN:	1.968	1.886	1.922	1.045	1.032	1.039	0.752	0.744	0.748
CNN_LR_N0:	**1.818**	1.861	2.117	**1.008**	1.021	1.076	**0.722**	0.732	0.770
CNN_LR:	1.851	**1.814**	**1.889**	1.011	**1.003**	**1.026**	0.724	**0.718**	**0.733**

Table 8: Results on the independent test set. Here CV1,CV2 and CV3 describe the training data used for the model, similar to Table 7. The standard deviation in the CRPS was estimated by bootstrapping a 1000 times and was found to be around 0.007 in all cases.

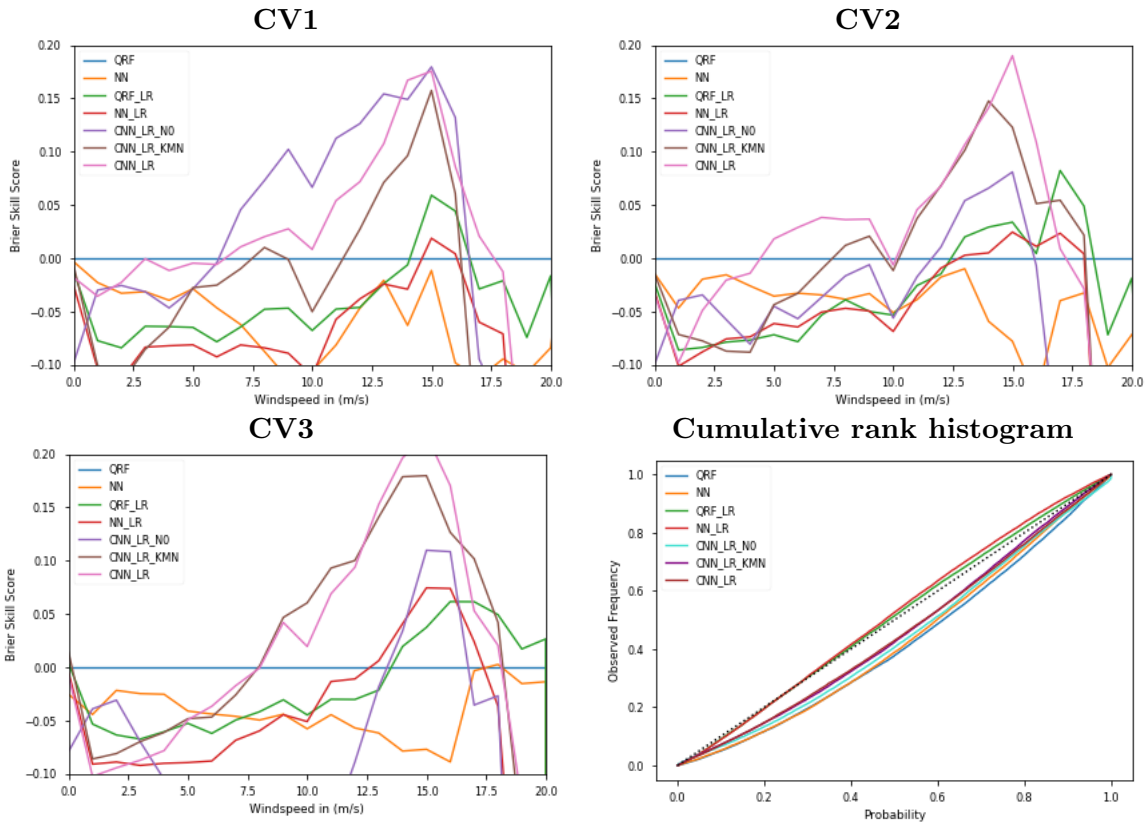

Figure 7: Brier skill scores relative to the Brier score of QRF, for predictions trained on three different training sets. Note that in all three cases the CNN's perform better for higher wind speeds. The results seem pretty stable over the three different tries. CNN_LR_N0 does, however, have much higher variance than the other methods. The bottom right image gives the cumulative rank histogram for the predictions of the three forecasts, per method, combined.

Method	rmse	mae	CRPS
Climatology	2.974	2.314	1.598
Linear Regression	2.399	1.170	-
QRF	2.217	1.110	0.776
QRF_LR	2.124	1.086	0.774
CNN_LR_N0	1.891	1.030	0.735
CNN_LR_KMN	1.905	**1.023**	0.740
CNN_LR	**1.889**	1.027	**0.731**

Table 9: The root mean squared error, mean absolute error and continuous ranked probability score of different methods for the independent test set, trained on the total training data set.

less convincing. Figure 8 shows the Brier skill score of the models trained on the full data set with respect to both the station climatology (left panel) and the Brier score of QRF (right panel). Here the standard deviation, obtained by bootstrapping a 1000 times, is given as well. From these images it is clear that for wind speeds above 15 m/s there is too much uncertainty to properly compare the methods.

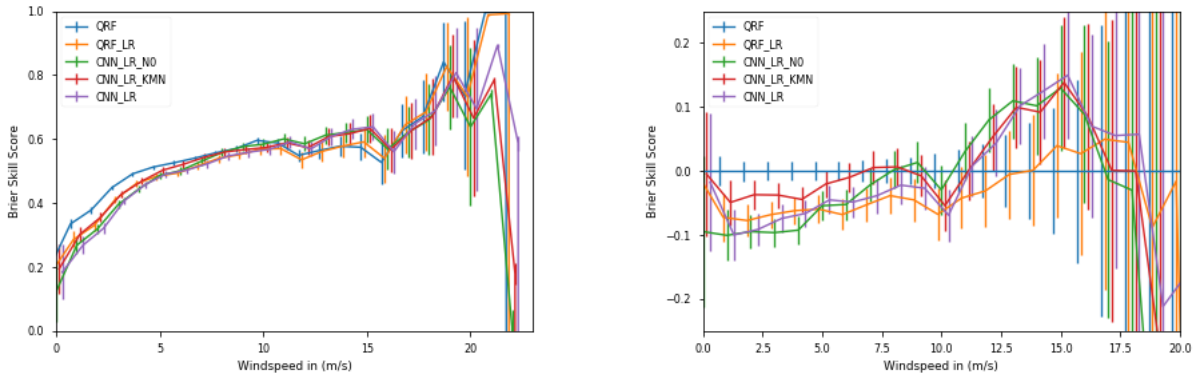

Figure 8: Brier skill scores relative to the station climatology(left) and QRF(right), for models trained on the full data set. Here error bars give the standard deviation which is obtained by bootstrapping the test data a 1000 times.

The right bottom panel of Figure 7 shows the cumulative rank histogram of all the methods. In this figure we can see a clear difference between the models trained on the wind speed and models trained on the residuals of linear regression for QRF and fully connected neural networks. Firstly the methods trained on the residuals lie a lot closer to the diagonal, implying that on average they estimate the spread a lot better. What is surprising however is that this does not hold for the convolutional neural networks which are trained on residuals as well. For QRF and convolutional neural networks we see that the cumulative rank histogram lies under the diagonal, which implies that for these methods observations fall in the higher quantiles of the estimated distribution.

In the Appendix a few example forecasts are shown. These images show the input image of CNN_LR, the activations for the three layers of the CNN and the corresponding forecast. In these forecasts a green line shows the forecast given by linear regression and the orange line the measurements. When

we look at these the Dutch coastline is one of the few clear features. Based on this we could hypothesize that the neural network is more skilful for higher wind speeds because of high skill for coastal stations. Figure 10 shows the difference in CRPSS with respect to the climatology between QRF and CNN_LR, for different stations in the Netherlands. Here we can see no clear geographical preference for either method, implying that the CNN uses different features as well to base its forecasts on.

5.5 Conclusion and Discussion

The results obtained from the independent test set show that for +48 hour wind speed forecast convolutional neural networks can be of added value for statistical post-processing. In terms of CRPS and mean squared error convolutional networks outperform QRF and fully connected neural networks in both all the cross-validation sets and the final test set.

The Brier skill score plots show that convolutional neural networks outperform random forest for higher wind speeds which are more important in weather forecasting. The bad performance of the convolutional neural networks with respect to QRF in the low wind speed range could be explained as an effect of using ordinary least squares regression first. This does assume symmetric errors and therefore does not perform well around zero. This could potentially be mitigated by assuming the errors to be truncated at zero. For wind speeds above 15 m/s the uncertainty in the Brier score grows very fast however and conclusion for this range can therefore not be made. This is mainly caused by a lack of days with high wind speeds in the available data set. An obvious solution for this would be to obtain more data by obtaining reforecast data for more years, this is costly however. A solution to this problem could to reforecasts days in the past with more extreme weather, such as days on which weather warnings were issued, instead of reforecasting full years only, as is done now.

An important feature of good forecast methods is that meteorologist understand what the forecast is based on. A large drawback of neural networks is that this is difficult to do for them. In the appendix one can find a few figures showing the activations in the convolutional layers of the network for a number of days. They do not give a clear indication however of what the network is looking at. In future research it would be a good addition to try to explain the results. Different methods of visualizing which parts of an input image is most relevant for the prediction made by a convolutional neural network exist, such as for example layer wise relevance propagation [3]. This could potentially help identify where the convolutional neural network is focusing on.

Using a parametric distribution such as the truncated normal could be beneficial in this case for it would be easier to make a distinction between features that are relevant in predicting the bias and features that are relevant in predicting the spread. In an ideal case identifying features which correspond to a high bias or spread might even help in identifying shortcomings in the NWP model.

At the time this study was conducted not enough data for ensemble forecasts was available for HA40. Most current post-processing studies are however based on ensemble output. An important next step is therefore to see if convolutional neural networks also add skill when an ensemble of forecasts is given.

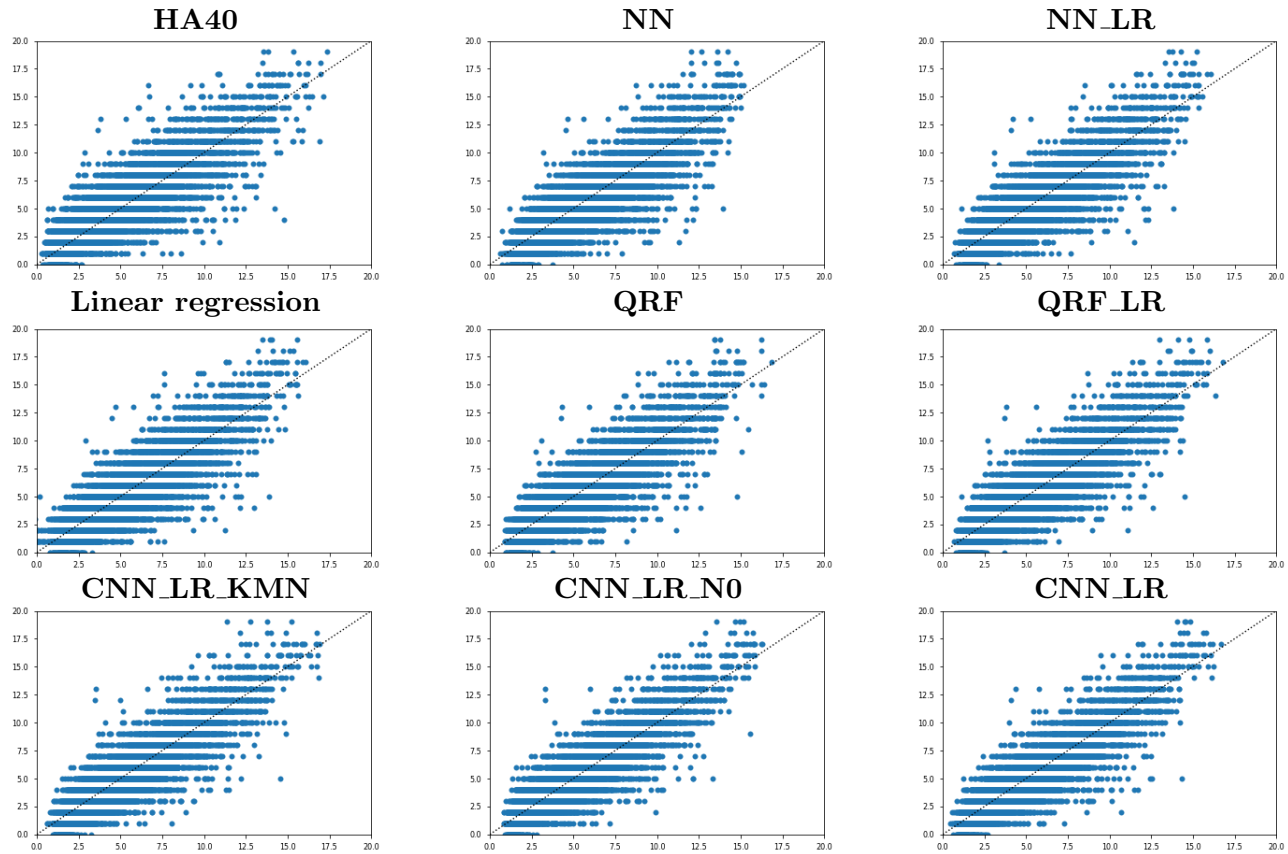

Figure 9: Scatterplots with the expectation value of the forecast on the x-axis and the observations on the y-axis, for the independent testset.

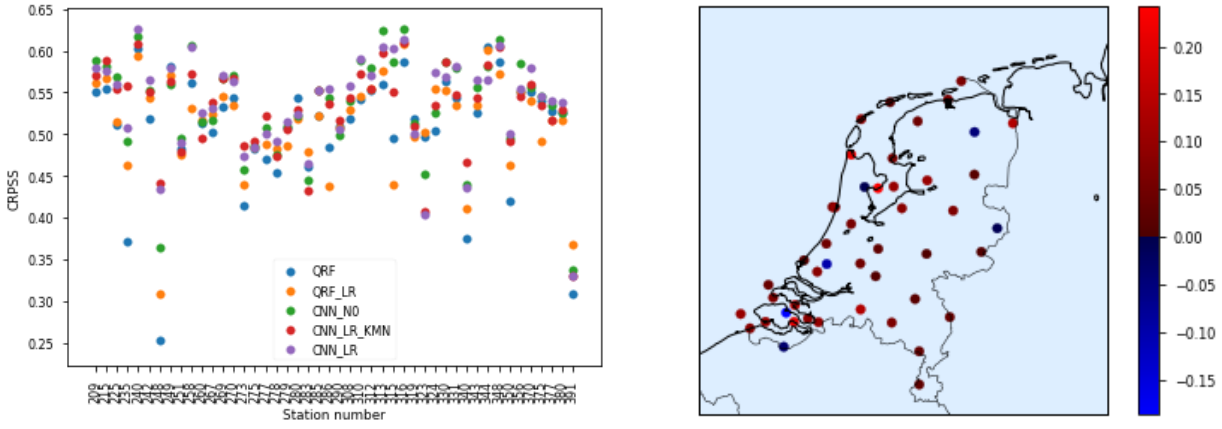

Figure 10: CRPSS with respect to the climatology of different methods on the left. CRPSS of CNN_LR with respect to QRF on the right. Here positive values imply that CNN_LR is more skilful than QRF. No clear geographic preference for either model is visible in this image. Station numbers are explained in Table 10 in the Appendix.

References

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

[2] Luca Ambrogioni, Umut Güçlü, Marcel AJ van Gerven, and Eric Maris. The kernel mixture network: A nonparametric method for conditional density estimation of continuous random variables. *arXiv preprint arXiv:1705.07111*, 2017.

[3] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[4] S. Baran and S. Lerch. Mixture emos model for calibrating ensemble forecasts of wind speed. *Environmetrics*, 27(2):116–130, 2016.

[5] Lisa Bengtsson, Ulf Andrae, Trygve Aspelien, Yurii Batrak, Javier Calvo, Wim de Rooy, Emily Gleeson, Bent Hansen-Sass, Mariken Homleid, Mariano Hortal, et al. The harmonie–arome model configuration in the aladin–hirlam nwp system. *Monthly Weather Review*, 145(5):1919–1935, 2017.

[6] Christopher M Bishop. Mixture density networks. 1994.

[7] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding batch normalization. In *Advances in Neural Information Processing Systems*, pages 7694–7705, 2018.

[8] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] François Chollet et al. Keras. https://keras.io, 2015.

[10] Antonio D'Isanto and Kai Lars Polsterer. Photometric redshift estimation via deep learning-generalized and pre-classification-less, image based, fully probabilistic redshifts. *Astronomy & Astrophysics*, 609:A111, 2018.

[11] K. Whan E. Ioannidis and M. Schmeits. probabilistic wind speed forecasting using parametric and non-parametric statistical post-processing methods, 2018.

[12] Harry R Glahn and Dale A Lowry. The use of model output statistics (mos) in objective weather forecasting. *Journal of applied meteorology*, 11(8):1203–1211, 1972.

[13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[14] Tilmann Gneiting and Adrian E. Raftery. Weather forecasting with ensemble methods. *Science*, 310(5746):248–249, 2005.

[15] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007.

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

[17] E. P. Grimit, T. Gneiting, V. J. Berrocal, and N. A. Johnson. The continuous ranked probability score for circular variables and its application to mesoscale forecast ensemble verification. *Quarterly Journal of the Royal Meteorological Society*, 132(621C):2925–2942, 2006.

[18] Thomas M. Hamill. Interpretation of rank histograms for verifying ensemble forecasts. *Monthly Weather Review*, 129(3):550–560, 2001.

[19] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[21] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

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

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[26] Sebastian Lerch and Thordis L. Thorarinsdottir. Comparison of non-homogeneous regression models for probabilistic wind speed forecasting. *Tellus A: Dynamic Meteorology and Oceanography*, 65(1):21206, 2013.

[27] Yunjie Liu, Evan Racah, Prabhat, Joaquin Correa, Amir Khosrowshahi, David Lavers, Kenneth Kunkel, Michael F. Wehner, and William D. Collins. Application of deep convolutional neural networks for detecting extreme weather in climate datasets. *CoRR*, abs/1605.01156, 2016.

[28] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[29] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[30] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[31] Nicolai Meinshausen. Quantile regression forests. *Journal of Machine Learning Research*, 7(Jun):983–999, 2006.

[32] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] Stephan Rasp and Sebastian Lerch. Neural networks for postprocessing ensemble weather forecasts. *Monthly Weather Review*, 146(11):3885–3900, 2018.

[35] Markus Reichstein, Gustau Camps-Valls, Bjorn Stevens, Martin Jung, Joachim Denzler, Nuno Carvalhais, and Mr Prabhat. Deep learning and process understanding for data-driven earth system science. *Nature*, 566:195, 02 2019.

[36] Jonas Rothfuss, Fabio Ferreira, Simon Walther, and Maxim Ulrich. Conditional density estimation with neural networks: Best practices and benchmarks. *arXiv preprint arXiv:1903.00954*, 2019.

[37] Sebastian Scher and Gabriele Messori. Predicting weather forecast uncertainty with machine learning. *Quarterly Journal of the Royal Meteorological Society*, 10 2018.

[38] Michael Scheuerer, David Möller, et al. Probabilistic wind speed forecasting on a grid based on ensemble model output statistics. *The Annals of Applied Statistics*, 9(3):1328–1349, 2015.

[39] Xingjian Shi, Zhihan Gao, Leonard Lausen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, and Wang-chun Woo. Deep learning for precipitation nowcasting: A benchmark and a new model. In *Advances in neural information processing systems*, pages 5617–5627, 2017.

[40] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[41] Maxime Taillardat, Olivier Mestre, Michaël Zamo, and Philippe Naveau. Calibrated ensemble forecasts using quantile regression forests and ensemble model output statistics. *Monthly Weather Review*, 144(6):2375–2393, 2016.

[42] Thordis L Thorarinsdottir and Tilmann Gneiting. Probabilistic forecasts of wind speed: ensemble model output statistics by using heteroscedastic censored regression. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 173(2):371–388, 2010.

[43] Kirien Whan and Maurice Schmeits. Comparing area probability forecasts of (extreme) local precipitation using parametric and machine learning statistical postprocessing methods. *Monthly Weather Review*, 146(11):3651–3673, 2018.

[44] Daniel S Wilks. *Statistical methods in the atmospheric sciences*, volume 100. Academic press, 2011.

A Appendix

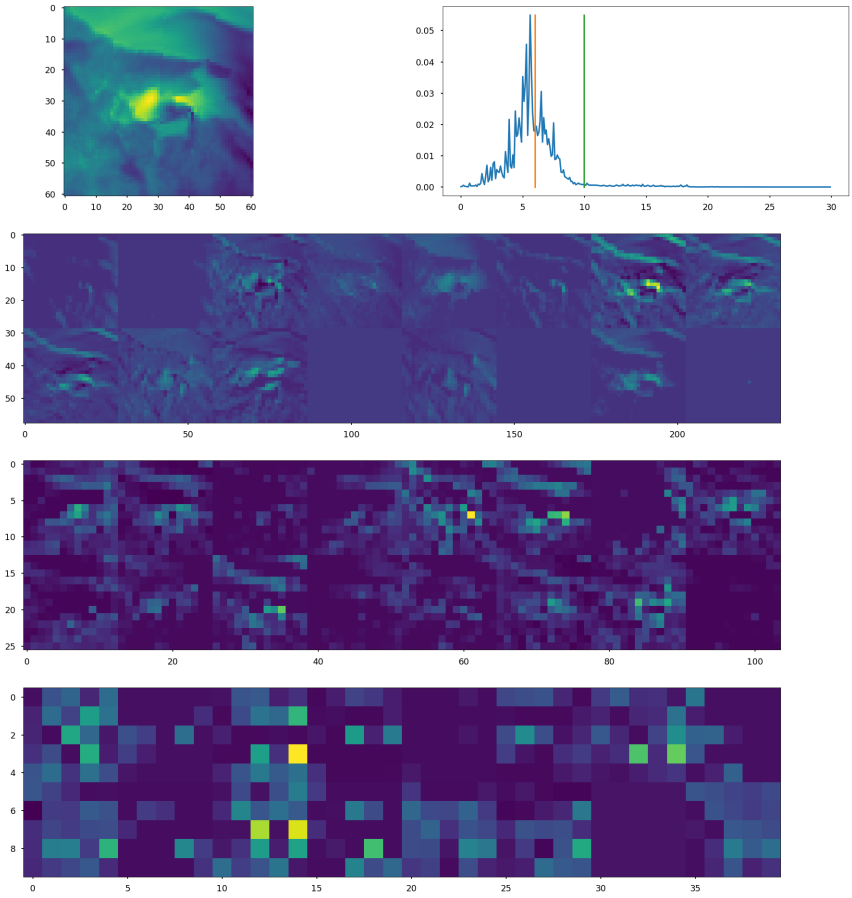

Figure 11: Example forecast from CNN_LR. The topleft image is the input image. The top right image shows the the probabilistic forecast. Here the orange line is the observation whereas the green line is the forecast obtained by linear regression. The bottom three images visualise the activations of the convolutional layers.

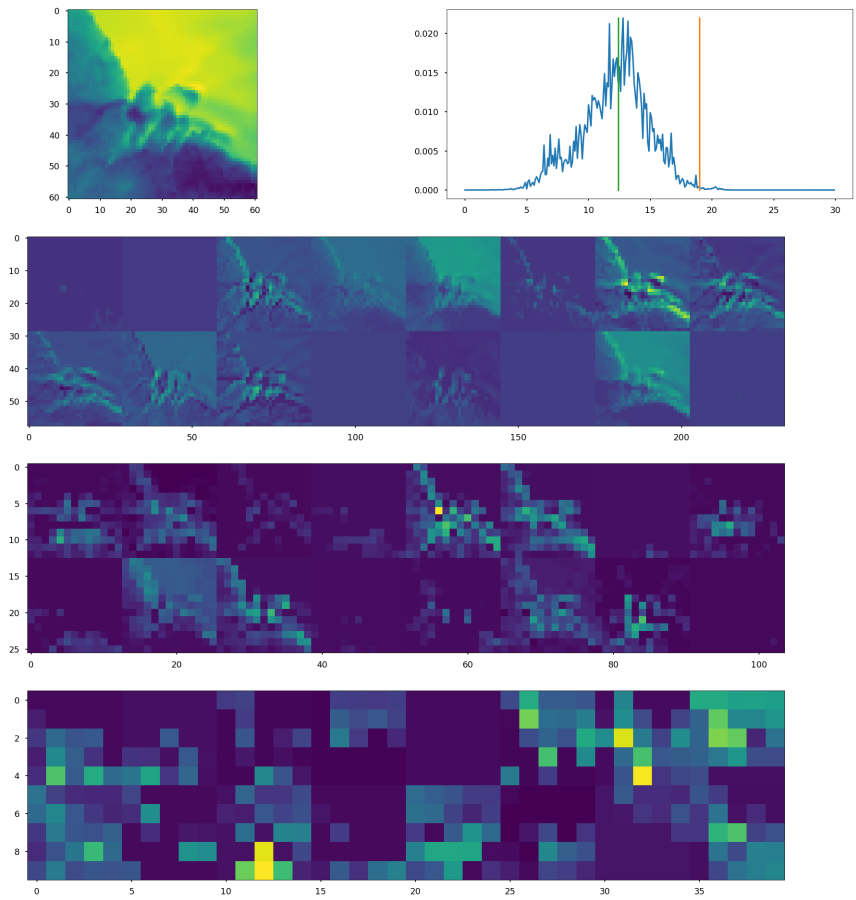

Figure 12: Example forecast from CNN_LR. The topleft image is the input image. The top right image shows the the probabilistic forecast. Here the orange line is the observation whereas the green line is the forecast obtained by linear regression. The bottom three images visualise the activations of the convolutional layers.

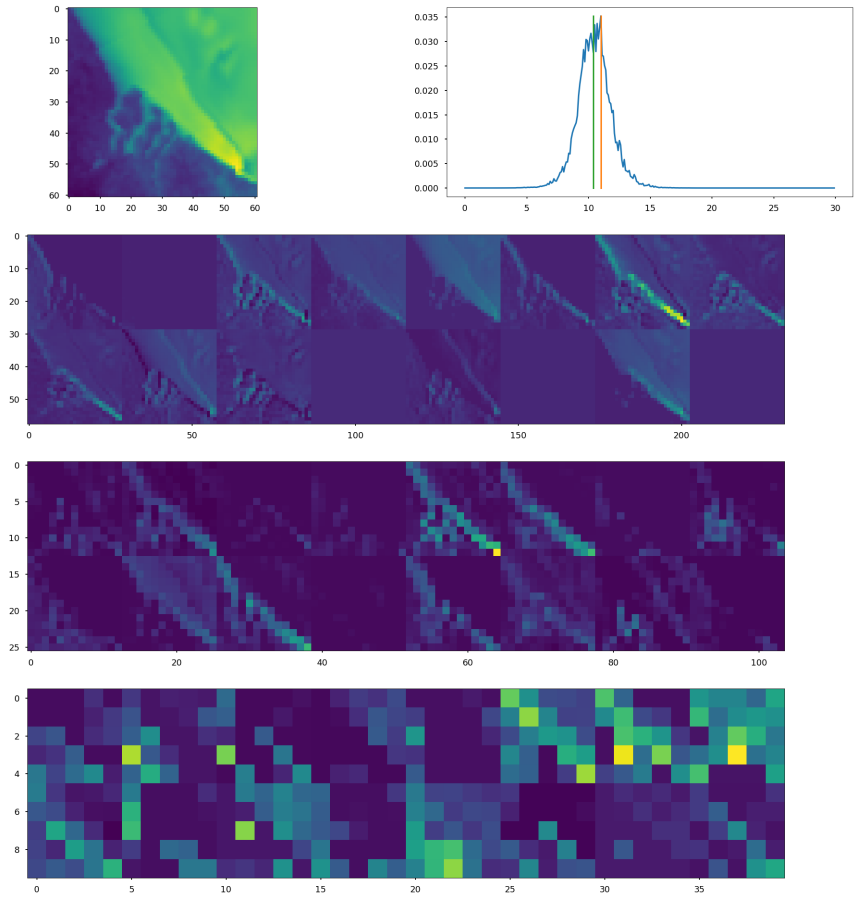

Figure 13: Example forecast from CNN_LR. The topleft image is the input image. The top right image shows the the probabilistic forecast. Here the orange line is the observation whereas the green line is the forecast obtained by linear regression. The bottom three images visualise the activations of the convolutional layers.

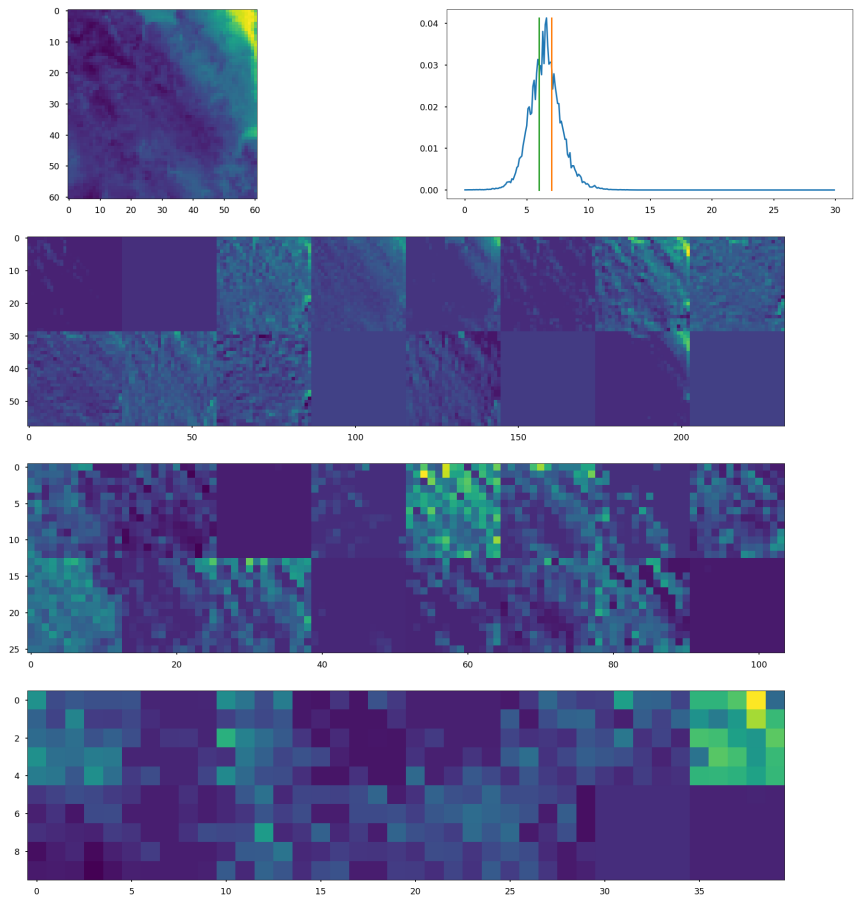

Figure 14: Example forecast from CNN_LR. The topleft image is the input image. The top right image shows the the probabilistic forecast. Here the orange line is the observation whereas the green line is the forecast obtained by linear regression. The bottom three images visualise the activations of the convolutional layers.

Station Number	Longitude	Latitude	Name
209	4.518	52.465	IJMOND
215	4.437	52.141	VOORSCHOTEN
225	4.555	52.463	IJMUIDEN
235	4.781	52.928	DE KOOY
240	4.790	52.318	SCHIPHOL
242	4.921	53.241	VLIELAND
248	5.174	52.634	WIJDENES
249	4.979	52.644	BERKHOUT
251	5.346	53.392	HOORN (TERSCHELLING)
258	5.401	52.649	HOUTRIBDIJK
260	5.180	52.100	DE BILT
267	5.384	52.898	STAVOREN
269	5.520	52.458	LELYSTAD
270	5.752	53.224	LEEUWARDEN
273	5.888	52.703	MARKNESSE
275	5.873	52.056	DEELEN
277	6.200	53.413	LAUWERSOOG
278	6.259	52.435	HEINO
279	6.574	52.750	HOOGEVEEN
280	6.585	53.125	EELDE
283	6.657	52.069	HUPSEL
285	6.399	53.575	HUIBERTGAT
286	7.150	53.196	NIEUW BEERTA
290	6.891	52.274	TWENTHE
308	3.379	51.381	CADZAND
310	3.596	51.442	VLISSINGEN
312	3.622	51.768	OOSTERSCHELDE
313	3.242	51.505	VLAKTE V.D. RAAN
315	3.998	51.447	HANSWEERT
316	3.694	51.657	SCHAAR
319	3.861	51.226	WESTDORPE
323	3.884	51.527	WILHELMINADORP
324	4.006	51.596	STAVENISSE
330	4.122	51.992	HOEK VAN HOLLAND
331	4.193	51.480	THOLEN
340	4.342	51.449	WOENSDRECHT
343	4.313	51.893	R'DAM-GEULHAVEN
344	4.447	51.962	ROTTERDAM
348	4.926	51.970	CABAUW
350	4.936	51.566	GILZE-RIJEN
356	5.146	51.859	HERWIJNEN
370	5.377	51.451	EINDHOVEN
375	5.707	51.659	VOLKEL
377	5.763	51.198	ELL
380	5.762	50.906	MAASTRICHT
391	6.197	51.498	ARCEN

Table 10: Location of weather stations in the Netherlands.