

University of *Ljubljana*
Faculty of *Mathematics and Physics*



Master program in Physics
Atmospheric Science Seminar

THE PROSPECT OF MACHINE LEARNING IN WEATHER FORECASTING

Author: Boštjan Melinc
Mentor: dr. Žiga Zaplotnik

Ljubljana, 2021

Abstract

The paper focuses on the use of machine learning techniques in weather forecasting. First, it provides some examples of machine learning that have already been successfully implemented to improve the outcomes of numerical weather prediction. Later it focuses on possibilities to completely replace numerical weather prediction with machine learning techniques. The main stress is given to explaining the theory behind convolutional neural networks and three examples of their usage for independent weather forecasting.

Contents

1	Introduction	1
2	Machine learning in meteorology	1
3	Theory of neural networks	2
3.1	Dense neural networks	2
3.2	Convolutional neural networks	5
4	Convolutional neural networks for independent weather forecasting	6
4.1	Our network	6
4.2	Advanced network	8
4.3	Residual network	11
5	Discussion and conclusions	11

1 Introduction

Ever since Charney’s first successful forecast in 1950, numerical weather prediction (NWP) has been essential for weather forecasting. General circulation models (GCMs) solve discretized governing equations of physics of the atmosphere and are coupled with similar models for the ocean as well as land surface models. Due to the chaotic nature of the atmospheric flow, the NWP will always have a finite limit of deterministic probability, however, its goal is to extend this limit further in time. Due to proper development of hardware in the last few years, using machine learning (ML) techniques to improve or even replace traditional operational numerical weather prediction has become a realizable idea [1–6].

The main conceptual difference between NWP and ML in weather forecasting is, that NWP relies solely on physics whereas in ML the networks learn from the reanalyses, i.e. the reconstructions of past weather, without any explicit information about physics at all. Some examples of ML applications in weather forecasting which have already been implemented are provided in Sec. 2. The theoretical part of this paper is in Sec. 3, where the theory of neural networks (NNs) is discussed. The main part of the paper follows in Sec. 4 with descriptions and results of three different networks which are designed for NWP-free weather forecasting and are arranged by their complexity from lowest to highest. We begin with our network (Sec. 4.1) which uses only one input field for its forecasting on a Gaussian grid, and continue with a slightly more complex network that uses at least four input fields which are represented in a smarter grid (Sec. 4.2). Finally, we describe a complex network which uses at least 38 input fields for forecasting (Sec. 4.3). Conclusions are given in Sec. 5.

2 Machine learning in meteorology

The most common ML technique used in weather forecasting are NNs. From the year 2018 on, they have been applied to some problems and already outperformed the traditional solutions. For example, the postprocessing of GCM forecasts to surface stations with NNs has already outperformed the traditional postprocessing methods and also there has been a successful downscaling of GCM output to higher horizontal resolution. Probably the most promising application of ML techniques in NWP is the improvement of parameterizations in GCMs because the parameterizations only tend to approximate the subgrid-physical processes rather than use the exact physical equations to solve them [1–3].

With proper ML techniques, it is also possible to not only improve the existing forecasts but also to predict their uncertainty. The most successful networks can also identify and even predict extreme

weather and climate patterns in observed and modeled atmospheric states, so in the future the alert systems might also rely on ML outputs [1–3].

All given examples show how we can improve NWP with ML. Using ML to entirely replace NWP, however, would require more effort and, as we are going to see later on, it may never be possible [1–4].

3 Theory of neural networks

3.1 Dense neural networks

Standard/Sequential/Dense neural networks consist of several layers of neurons. The first layer is the *input* layer where, as the name suggests, the input data enters the network. The number of neurons in the input layer depends on the expected input data, e.g. if the input data is a picture of 256×256 pixels, there will be $256 \cdot 256 = 65\,536$ neurons in the input layer. The input layer is followed by an arbitrary number of *hidden* layers, each of these layers consists of an arbitrary number of neurons. Finally, there is an *output* layer, which gives the result. The number of neurons in the output layer depends on the type of problem for which the network is designed [7]. For example, if the input is the photography of the sky and the question is “How much sky is covered with the clouds?”, then there will be only one neuron in the output layer, whose value will correspond to the fraction of the sky covered with the clouds. However, if the question is “How much sky is covered with each type of clouds?”, then there will be one neuron for cumulus clouds, one for cumulonimbus, one for cirrus, etc.

The “standard” neural networks are also called *sequential* because their layers are connected in a direct sequence and there is no skipping. They are also called *dense* as each neuron of a certain layer is connected with all the neurons of the neighbouring layers. This means that there are

$$N^{input} \cdot N_1^{hidden} + \sum_{i=2}^H N_{i-1}^{hidden} \cdot N_i^{hidden} + N_H^{hidden} \cdot N^{output} \quad (1)$$

connections between neurons in total, where N^{input} is the number of neurons in the input layer, N_i^{hidden} is the number of neurons in the i -th hidden layer, H is the number of hidden layers, and N^{output} is the number of neurons in the output layer. Each connection has its own weight, so each neuron from one layer affects each neuron in the next layer differently. Besides the connections with the neurons from the neighbouring layers, each neuron also has an extra connection with a constant, which serves as a bias and also has its own weight [7]. All the connections are illustrated in an example in Fig. 1.

To get the value of the neuron j in the layer i we have to sum up all the contributions from the layer $i - 1$ and the bias. To do this, we first add the values of all the neurons in the layer $i - 1$ to the vector $\mathbf{x}^{(i-1)} \in \mathbb{R}^{1 \times (N_{i-1}+1)}$ whose final element is a constant which serves as a bias. Then we merge all the weights (corresponding to connections between neuron j in layer i to all neurons in layer $i - 1$) in a single row, and perform matrix multiplication between this row and the vector $\mathbf{x}^{(i-1)}$. Finally, we compute the value of a nonlinear activation function $f^{(i)}$ of the computed scalar which is then the value of the neuron j in the layer i . To get the values of all neurons in the layer i , we merge the rows with the weights into the weight matrix $\mathbf{W}^{(i)}$, so we can write

$$\mathbf{x}^{(i)} = f^{(i)} \left(\mathbf{W}^{(i)} \mathbf{x}^{(i-1)} \right). \quad (2)$$

To get the output values, we first compute $\mathbf{x}^{(1)}$ from the input values. Then we compute $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, etc. This process is called *feed forward* [7]. For the example from Fig. 1, we would compute the output values using Eq. (2) as

$$\mathbf{x}^{(output)} = f^{(output)} \left(\mathbf{W}^{(output)} f^{(2)} \left(\mathbf{W}^{(2)} f^{(1)} \left(\mathbf{W}^{(1)} \mathbf{x}^{(input)} \right) \right) \right).$$

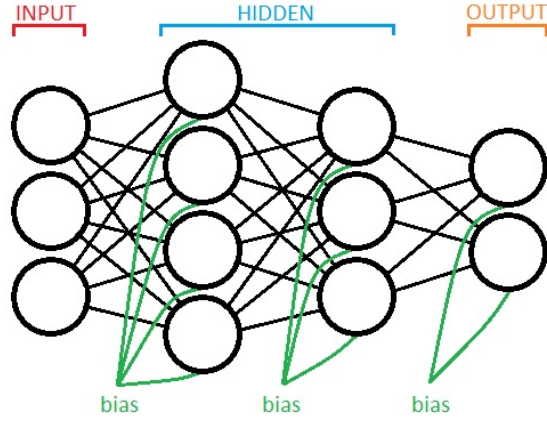


Figure 1: An example of a dense neural network with three neurons in the input layer, two hidden layers with four and three neurons, respectively, and an output layer with two neurons. All layers except the input layer are biased.

This example also shows us the reason why the activation functions are always nonlinear. If they were linear, all the matrix multiplications would collapse into one matrix

$$\mathbf{W}' = f^{(\text{output})} \left(\mathbf{W}^{(\text{output})} f^{(2)} \left(\mathbf{W}^{(2)} f^{(1)} \left(\mathbf{W}^{(1)} \right) \right) \right)$$

and the purpose of the hidden layers would be lost, as they could be replaced by just one direct connection between $\mathbf{x}^{(\text{input})}$ and $\mathbf{x}^{(\text{output})}$.

So far we have only discussed how the pre-trained NNs yield results from the given input, however, the true power of the NNs is their learning procedure. When training the NN, we determine its loss function J and try to minimize it. All the networks which will be described further on in Sec. 4 use mean squared error as their loss function, which is calculated as

$$J = \sum_{j=1}^{N^{\text{output}}} \left(x_{T,j}^{(\text{output})} - x_{F,j}^{(\text{output})} \right)^2, \quad (3)$$

where the subscript T denotes the “true” values (e.g. from reanalysis data) and F denotes the forecasted values by the network [1, 4, 7]. There are also some other possible choices for a loss function (e.g. mean absolute error and binary crossentropy), however the choice of mean squared error here seems to be adequate for to the nature of the problem we are dealing with [7].

When learning, NN minimizes J by adjusting its weights in a process called *stochastic gradient descent*, which adjusts the weights as

$$W_{i,j}^{(k)} \longmapsto W_{i,j}^{(k)} - \eta \frac{1}{n} \sum_{\text{minibatch}} \frac{\partial J}{\partial W_{i,j}^{(k)}}, \quad (4)$$

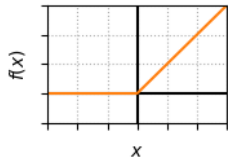
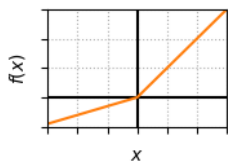
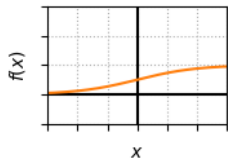
where η is the *learning rate* and n is the number of input vectors in the minibatch (i.e. a subset of the training data). We use minibatches to reduce the sensibility of a NN on the possible occasional false inputs while learning and to speed the learning process up. For example, when training our network (Sec. 4.1), we used minibatches of 1024 simultaneous inputs. We compute the partial derivatives of J

from Eq. (4) via *backpropagation*. For the example from Fig. 1 this would give

$$\begin{aligned} \frac{\partial J}{\partial \mathbf{W}^{(\text{output})}} &= (\nabla J) \left[\nabla f^{(\text{output})} \left(\mathbf{W}^{(\text{output})} \mathbf{x}^{(2)} \right) \right] \otimes \mathbf{x}^{(2)} \\ \frac{\partial J}{\partial \mathbf{W}^{(2)}} &= \left\{ (\nabla J) \left[\nabla f^{(\text{output})} \left(\mathbf{W}^{(\text{output})} \mathbf{x}^{(2)} \right) \right] \mathbf{W}^{(\text{output})} \left[\nabla f^{(2)} \left(\mathbf{W}^{(2)} \mathbf{x}^{(1)} \right) \right] \right\} \otimes \mathbf{x}^{(1)} \\ \frac{\partial J}{\partial \mathbf{W}^{(1)}} &= \left\{ (\nabla J) \left[\nabla f^{(\text{output})} \left(\mathbf{W}^{(\text{output})} \mathbf{x}^{(2)} \right) \right] \mathbf{W}^{(\text{output})} \left[\nabla f^{(2)} \left(\mathbf{W}^{(2)} \mathbf{x}^{(1)} \right) \right] \right. \\ &\quad \left. \mathbf{W}^{(2)} \left[\nabla f^{(1)} \left(\mathbf{W}^{(1)} \mathbf{x}^{(\text{input})} \right) \right] \right\} \otimes \mathbf{x}^{(\text{input})}, \end{aligned}$$

where ∇ denotes a derivative with respect to the functions arguments. As we can see from the given example, the backpropagation is as cheap for computing as feed forward, if the gradients of the activation functions are also cheap for computing. This is why the activation functions are usually simple nonlinear functions. The activation functions which were used in the networks from Sec. 4.1 to Sec. 4.3 are represented in Table 1. Here we can also justify the use of a bias for each neuron: bias is the most elegant way to shift the activation function to the left or right [1, 4, 7].

Table 1: Some examples of the activation functions.

Name	Plot	Function ($f(x)$)	Derivative ($\nabla f(x)$)
Rectified linear unit (ReLU)		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$
Parametric rectified linear unit (PReLU)		$\begin{cases} \alpha x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$\begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$
Sigmoid		$\frac{1}{1 + e^{-x}}$	$f(x) \cdot (1 - f(x))$

For successful learning, we split our data samples into three separate sets. The largest set is usually the *training* set. The entire process of feedforward and stochastic gradient descent is performed with this set. For the better final performance, we train the network on this set multiple times – each round of training is called an *epoch*. After each epoch, we check the value of the loss function on the *validation* set. If it starts oscillating, we should reduce the learning rate, and once it starts continuously rising, we should stop the learning process, because we are experiencing *overfitting* on our network on the training set. Finally, we evaluate our model on the *test* set. The value of the loss function on the test set is the best way to compare individual networks because the learning was performed completely independently from this set [1–4, 7]. Another simple way to improve the performance of the network is to shuffle the training set before the first epoch, so the data samples in the minibatch are uncorrelated. It is also recommendable to normalize the input data samples, which leads to better performance of the activation functions [7].

3.2 Convolutional neural networks

Machine learning procedure with convolutional neural networks (CNNs) is almost identical to ML with standard neural networks. The main difference is in the architecture of the network – instead of connecting every neuron in one layer with all the neurons in the neighbouring layer and changing weights of these connections we rather use compact kernels between layers and train our networks by adjusting these kernels. The size of kernels is usually much smaller than the size of the input data samples, so the kernels tend to extract the local properties of the samples [8].

To understand the idea behind the kernels, let us look at two examples that can be used when processing a two-dimensional input (i.e. a photography). The kernel which we would use to determine a derivative of values of pixels in the horizontal direction could be a constant multiplied by

$$[-1 \ 1] \quad \text{or} \quad [-1 \ 0 \ 1] \quad \text{or} \quad \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix},$$

etc. For the Laplace operator the kernel would be a constant multiplied by

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

To describe the entire physics of the atmosphere, we would require a very large number of such kernels, which would be difficult to manage. Instead, we rather use a smaller amount of kernels, which at first have random values, and let them be adjusted with training via proper stochastic gradient descent. Again, to justify the use of more than one kernel, we have to apply a nonlinear activation function on the result of each convolution between the input and the kernel [8].

Convolution of a two-dimensional input with a kernel gives an output of a smaller size, as shown in an example in Fig. 2. This is why we resort to *padding* the edges of the input. The most common padding is zero-padding [8], which is used in Fig. 2. As we can also see in the given example, the result of a convolution in row i and column j is calculated as a Frobenius inner product between a submatrix which is centered in row i and column j of the non-padded input and the kernel.

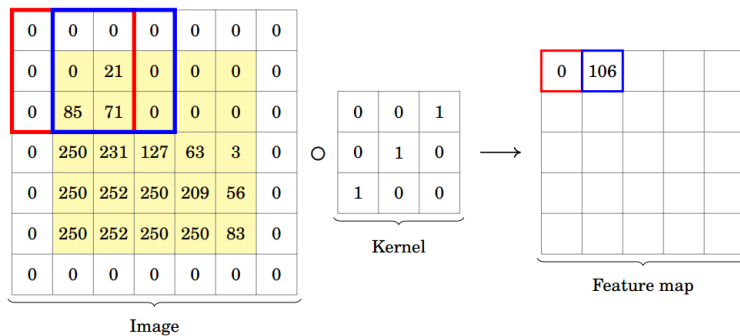


Figure 2: An example of a convolution of a zero-padded image with original size 5×5 with a kernel with size 3×3 . Reproduced from <https://www.vojtech.net/img/machine-learning/convolution.png>.

If a CNN's output is expected to have the same shape as its input, it usually consists of two parts. The first part is the *encoder*, where the shape is downsized. In my network (Sec. 4.1) this is done by *max pooling*, where the new smaller matrix contains the maximum values of (usually 2×2) submatrices of the previous matrix. The goal of the encoder is to extract the important information from the network's input. The encoder is followed by the *decoder*, whose goal is to produce the output

based on that important information. We enlarge the shape of matrices by *upsampling*, which, for example, produces 2×2 submatrices from 1×1 submatrices by copying their element [8].

When the CNN is being trained, each element of a kernel represents a parameter of the loss function, which is minimized through stochastic gradient descent. The number of parameters P_i in convolutional layer i can be calculated as

$$P_i = K_{i-1} \cdot S_i \cdot K_i + 1 \cdot K_i; \quad i = 1, 2, \dots, \quad (5)$$

where K_i is the number of kernels in layer i ($K_0 = 1$) and S_i is the size of these kernels (i.e. number of elements in the kernels). The second term on the right hand side is due to the presence of a bias in each kernel. In total, CNNs contain much less parameters than dense NN with similar structure (i.e. where each convolutional layer is represented by a layer of neurons and each element in the matrices after convolution is represented by a neuron in this layer). For example, in Tables 2 each network contains 8 937 trainable parameters whereas in a dense neural network with a similar structure there would be approximately $3 \cdot 10^9$ trainable parameters, which is much more difficult to optimize. This example serves as a confirmation, that CNNs are a much better option than dense NNs if local properties of input data are decisive.

4 Convolutional neural networks for independent weather forecasting

In this section, we will discuss the results from three different CNNs. They all predict geopotential height at 500 hPa pressure level Z_{500} on a global scale. In meteorology, physical quantities are locally always spatially correlated which is the reason why all the discussed networks in this section are CNNs [2].

Two key measures to compare the results between the networks are *root mean square error*

$$\text{RMSE} = \sqrt{\overline{(\mathbf{f}(t) - \mathbf{o}(t))^2}}, \quad (6)$$

where $\mathbf{f}(t)$ denotes a forecast vector at time t , $\mathbf{o}(t)$ is the observed state and overline denotes the spatial average, and *anomaly correlation coefficient*

$$\text{ACC} = \frac{\overline{(\mathbf{f}(t) - \mathbf{c}(t)) \cdot (\mathbf{o}(t) - \mathbf{c}(t))}}{\sqrt{\overline{(\mathbf{f}(t) - \mathbf{c}(t))^2} \overline{(\mathbf{o}(t) - \mathbf{c}(t))^2}}}, \quad (7)$$

where $\mathbf{c}(t)$ is the climatological value for the verification time. A perfect forecast has an ACC score of 1, while a score of 0 indicates a forecast with no skill relative to climatology [1].

Using these measures, the forecasts were compared with two benchmark fields, i.e. climatology and persistence. Persistence here represents a state of the atmosphere which has not changed since the time of the beginning of our forecast. The forecasts of some networks were also compared with the same measures for some other benchmarks, which are described separately in each subsection.

4.1 Our network

My CNN was designed to directly forecast Z_{500} at time t_n from Z_{500} at time t_{n-1} . The input data were reanalysis data from the ERA5 dataset of ECMWF [9]. Data were provided on a regular longitude-latitude grid with a spatial resolution of 6° in each direction and temporal resolution of 6 h, so the time step of my forecast was also 6 h. We avoided the singularities of regular grids at the poles by simply ignoring them (i.e. by leaving them out of the input vector). Training and validation sets contained 80% and 20% of randomly shuffled data for the years 1979-2018, respectively. The data for the year 2019 served as the test set. We trained the networks for 40 epochs. The input data were normalized

to the interval $[0, 1]$, where 0 and 1 represent the global minimum and maximum Z_{500} value from the training and validation sets.

We tried two similar types of networks for different inputs. The first was for 2-dimensional input. The input size was supposed to be 29×60 ($=$ (number of latitudes $- 2$) \times (number of longitudes)), however it was difficult to reach 29 elements in latitudinal dimension by upsampling (29 is a prime number), so for simplicity we decided to also leave out the data for 84°S . we tried various combinations of numbers of kernels, their sizes, downsamplings, and upsamplings. The network which gave the best results is presented in Table 2a. It had 8937 trainable parameters in total. The number of parameters in other networks varied from 500 to 60000. Since we were not satisfied with the results from such networks, we designed a similar network, however this time for a 1-dimensional input (i.e. flattened 2-dimensional input) with $29 \cdot 60 = 1740$ points in the input vector. This network is presented in Table 2b.

Table 2: Our networks that gave the best results in terms of RMSE and ACC. Each layer is followed by the layer in the row beneath it. The first column contains the type of each layer. Here Max Pooling/Upsampling refers to max pooling/upsampling from/to 2×2 submatrices of the 2nd and 3rd dimension of the output shape. The second column gives the number of kernels in the convolutional layers and the third gives their size. The fourth layer contains the shape of the output of the layer, which can be described as (grid faces, latitudes, longitudes, kernels). The final column contains the number of trainable parameters in each layer which is calculated by Eq. (5). All the convolutional layers used the activation function ReLu and zero-padding. For 1-dimensional input, we could have changed all layers with 1D layers of the same type without any consequences.

(a) The best network for a 2-dimensional input of Z_{500} .

Layer type	Kernels	Kernel size	Output shape	Trainable parameters
<i>input</i>			(1, 28, 60, 1)	
2D Convolutional	32	3×4	(1, 28, 60, 32)	416
2D Max Pooling			(1, 14, 30, 32)	
2D Convolutional	16	2×4	(1, 14, 30, 16)	4112
2D Max Pooling			(1, 7, 15, 16)	
2D Convolutional	8	2×2	(1, 7, 15, 8)	520
2D Convolutional	16	2×2	(1, 7, 15, 16)	528
2D Upsampling			(1, 14, 30, 16)	
2D Convolutional	32	2×3	(1, 14, 30, 32)	3104
2D Upsampling			(1, 28, 60, 32)	
2D Convolutional	1	2×4	(1, 28, 60, 1)	257

(b) The best network for a 1-dimensional input of Z_{500} .

Layer type	Kernels	Kernel size	Output shape	Trainable parameters
<i>input</i>			(1, 1, $29 \cdot 60 = 1740$, 1)	
2D Convolutional	32	1×12	(1, 1, 1740, 32)	416
2D Max Pooling			(1, 1, 870, 32)	
2D Convolutional	16	1×8	(1, 1, 870, 16)	4112
2D Max Pooling			(1, 1, 435, 16)	
2D Convolutional	8	1×4	(1, 1, 435, 8)	520
2D Convolutional	16	1×4	(1, 1, 435, 16)	528
2D Upsampling			(1, 1, 870, 16)	
2D Convolutional	32	1×6	(1, 11, 870, 32)	3104
2D Upsampling			(1, 1, 1740, 32)	
2D Convolutional	1	1×8	(1, 1, 1740, 1)	257

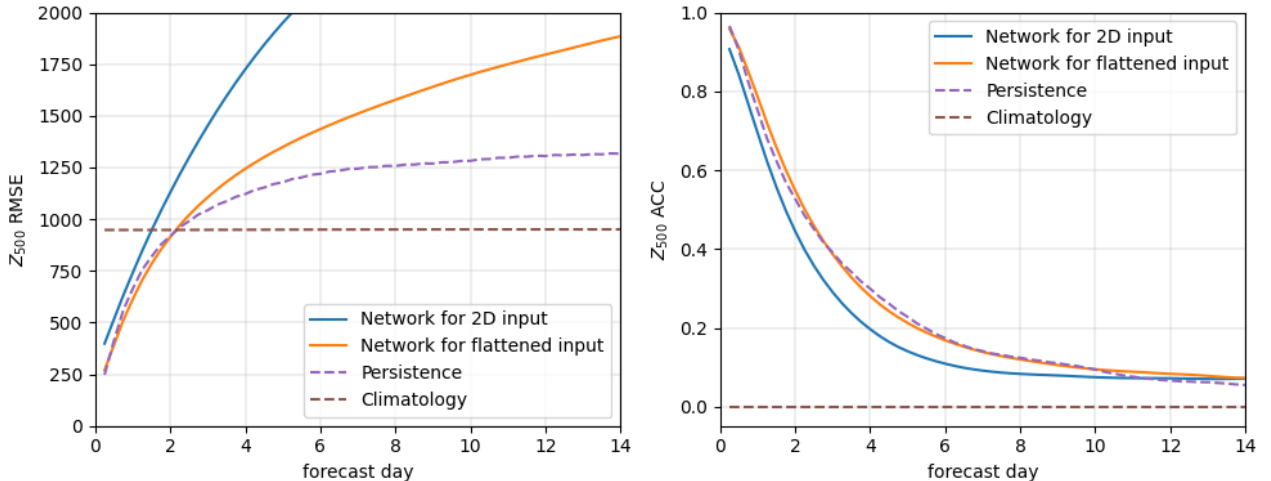


Figure 3: Average RMSE and ACC for my most successful networks (Tables 2), persistence and climatology, computed for the year 2019. Climatology was considered as the mean value of Z_{500} for each hour (0, 6, 12, 18 UTC) for each day of the year (except the 29th of February) and was computed for data between the years 1979 and 2018.

The results for the networks from Tables 2 are shown in Fig. 3. There we see that the network for a 1-dimensional input slightly outperformed the persistence for the first two days of forecast in terms of RMSE and in the first three days of forecast in terms of ACC. The network for a 2-dimensional input, however, only outperformed the climatology in the first day and a half of the forecast. How good are these results? As we will see later on, in comparison to modern NWP, they are very bad. However, the prediction using the barotropic vorticity equation on a regular grid between 20°N and 70°N with 2.5° resolution and time step of 30 min on average only outperforms persistence in the first 36 h of forecast in terms of RMSE [2]. Such prediction was performed by Charney et al. in 1950 and is considered as the first successful NWP. Since my forecast was apparently better than Charney’s, we can conclude that our network for 1-dimensional input is successful in weather forecasting. The fact that such simple CNNs already give adequate results is also astonishing because it would be impossible to compute a successful NWP with such a large time step due to the CFL conditions.

4.2 Advanced network

Forecasting Z_{500} only from data for Z_{500} has its limitations, which are probably not far from what we achieved with our network. For better forecasting of Z_{500} the authors of [1] also introduced three other variables of the atmosphere to their CNN, namely the geopotential height at 1000 hPa Z_{1000} , 300 – 700 hPa geopotential thickness $\tau_{300-700}$, and 2-meter temperature T_{2m} . Each variable was normalized by removing its global climatological mean and dividing by its global mean standard deviation. All the listed variables were also the output from their network, so they could run it iteratively. Next to these fields, there were also three additional input fields: top-of-atmosphere incoming solar radiation, a land-sea mask (0 over ocean, 1 over land), and topographic height. Even though the NNs do not “know” any physics, these additional inputs turned out to improve the performance of the network.

Another problem that we faced when designing our network was related to the regular grid and its singularities near the poles. The authors of [1] avoided this problem by interpolating the data to the gnomonic equiangular cube sphere grid, as shown in Fig. 4. They minimized the kernels of their CNN, which is presented in Table 3, for each face separately and also used smarter padding in the first layer – instead of zeros they used the values from the neighbouring cells from the neighbouring

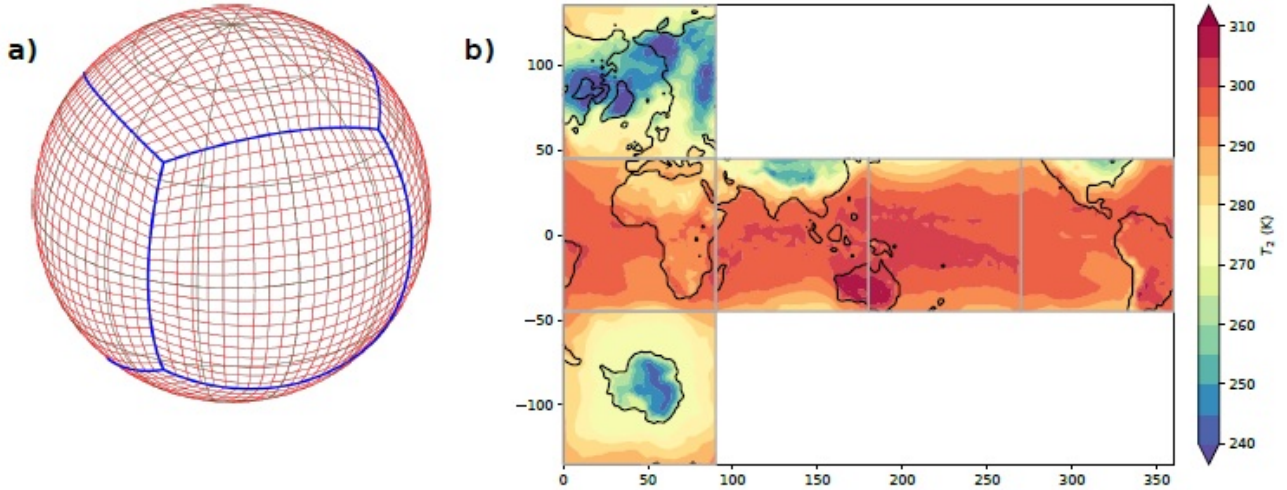


Figure 4: a) The gnomonic equiangular cube sphere grid with 20×20 grid cells on each face. b) An example of a T_{2m} map on the flattened cubed sphere. Adapted from [1].

Table 3: Architecture of the advanced CNN from [1]. The parameter v represents the number of input fields, t represents the number of input time steps, and c represents the number of additional inputs. The Average Pooling layers are similar to the Max Pooling layers, however, instead of pooling the maximum value of the 2×2 submatrix, they pool its average value. The Concatenate layers append the states numbered in parentheses to the output of the previous layer. The activation function was a modification of PReLU and padding was done as described in the text. For the explanation of other properties see the caption to Table 2.

Layer type	Kernels	Kernel size	Output shape	Trainable parameters
<i>input</i>			$(6, 4, 48, vt + c)$	
2D Convolutional	32	3×3	$(6, 48, 48, 32)$	6 976 ^a
2D Convolutional (1)	32	3×3	$(6, 48, 48, 32)$	18 496
2D Average Pooling			$(6, 24, 24, 32)$	
2D Convolutional	64	3×3	$(6, 24, 24, 64)$	36 992
2D Convolutional (2)	64	3×3	$(6, 24, 24, 64)$	73 856
2D Average Pooling			$(6, 12, 12, 64)$	
2D Convolutional	128	3×3	$(6, 12, 12, 128)$	147 712
2D Convolutional	64	3×3	$(6, 24, 24, 64)$	147 584
2D Upsampling			$(6, 24, 24, 64)$	
Concatenate (2)			$(6, 24, 24, 128)$	
2D Convolutional	64	3×3	$(6, 12, 12, 128)$	147 548
2D Convolutional	32	3×3	$(6, 24, 24, 32)$	36 928
2D Upsampling			$(6, 48, 48, 32)$	
Concatenate (1)			$(6, 48, 48, 64)$	
2D Convolutional	32	3×3	$(6, 48, 48, 32)$	36 928
2D Convolutional	32	3×3	$(6, 48, 48, 32)$	18 496
2D Convolutional	vt	1×1	$(6, 48, 48, vt)$	528 ^a

^aValid for the best network, where $v = 4$, $t = 2$, $c = 4$.

faces. The padding of the corners turned out not to be of vital importance. The best network had 672 080 trainable parameters. The training set contained the data from years 1979-2012, the validation set contained data from years 2013-2016 and the test set contained data from years 2017-2018. The

training procedure took at least 150 epochs.

The results of the best network from [1] are shown in Fig. 5 and are compared to three additional benchmarks – an ECMWF model with 62 vertical levels and an approximate horizontal resolution of 2.8° (IFS T42), an ECMWF model with 137 vertical levels and an approximate resolution of 1.9° which is also coupled to an ocean wave model (IFS T63), and the operational subseasonal-to-seasonal ECMWF model with the horizontal resolution of only 16 km that was designed for NWP for time scales of 2 weeks to 2 months (S2S ECMWF control). The network best performed when, instead of iterating the variables at time n only from variables at time $n - 1$, it simultaneously iterated variables at time n and $n + 1$ from variables at time $n - 1$ and $n - 2$. In this case, the CNN completely outperformed the IFS T42 and when forecasting T_{2m} it even outperformed the IFS T63 in the first 3-4 days and the S2S ECMWF control in the first 2 days. If we take climatology as a measure, this CNN lags the performance of a high-resolution operational NWP model by about 2-3 days of forecast time. However, its major advantage against the NWP is the computing time. For a single 4 week forecast, IFS T63 requires approximately 24 min, while for a 1000-member ensemble it would require about 16 days of computing. On the other hand, it took 2-3 days for training this CNN, but once it had been trained, it took less than 0.2 s to compute a single 4 week forecast and about 3 min to compute a 1000-member ensemble! Another fascinating fact from this CNN is that its outputs turned out to be stable. And not only it did not blow up when the authors of [1] ran it iteratively for a 1 year forecast, the forecast even gave the results that were realistic in means of climatology.

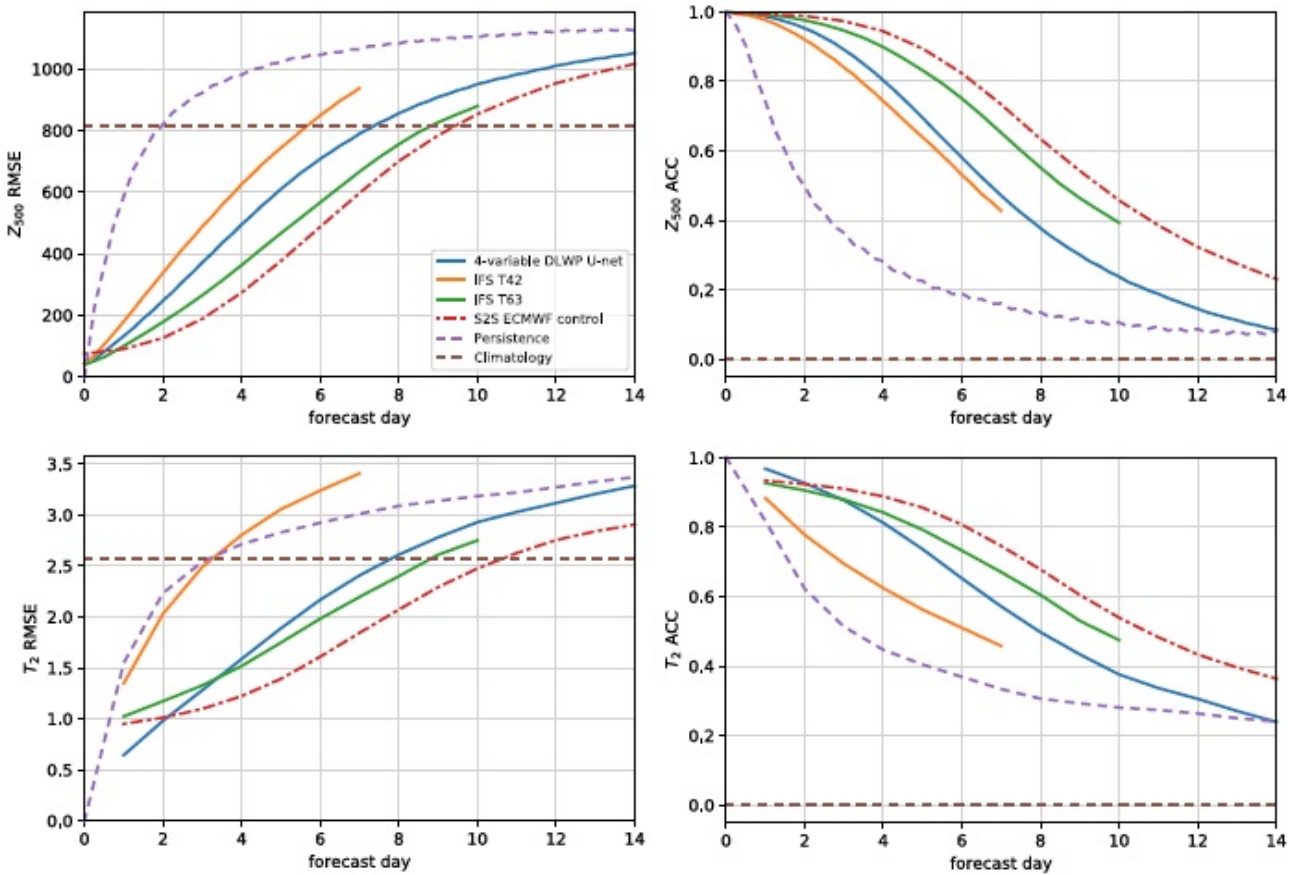


Figure 5: Average RMSE and ACC for the best CNN of [1], computed for years 2017-2018. Climatology was considered as the mean daily value for each day of the year and was computed for data for years 1979-2010.

4.3 Residual network

Since there already is a giant leap in the quality of weather forecasting with CNNs if we use 4 physical quantities instead of 1, introducing even more quantities could give a result, which is better than the results from modern NWP [10]. The authors of [4] designed a network for 38 normalized input quantities, i.e. geopotential, temperature, zonal and meridional wind and specific humidity at seven vertical levels, 2-meter temperature 6-hourly accumulated precipitation and the top-of-atmosphere incoming radiation, all at the three consecutive time steps, which were 6 h apart. They also added the land-sea mask, orography and the latitude at each grid point.

To handle all these variables, they used a special type of CNN, which is referred to as the residual network. The basic structure of this NN was 19 residual blocks. Each block consisted of two convolutional layers, each with 128 kernels. The output from each block of a residual network is concatenated with the input to this block, as shown in Fig. 6. If the input to the block already gives a good final

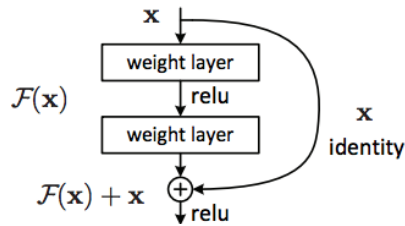


Figure 6: A path of the input vector through a residual block. Reproduced from [11].

result, then the second convolutional layer will give very weak weights, so the concatenated output will only contain small adjustments to the input. This allows adding layers to pre-trained networks, which is the reason why these networks are the most popular for image recognition – for example, once your residual CNN is able to successfully distinguish between cats and dogs, you can add a few residual blocks and (by applying stochastic gradient descent on these blocks exclusively) train a CNN that distinguishes between different breeds of dogs. Doing the same with sequential CNNs would be much more difficult because adding another layer to them requires training of the entire network once again [4, 8, 11].

At first, the authors of [4] tried to forecast some quantities directly from the original input. For example, this means that they predicted Z_{500} at $t+5$ days directly from the 38 inputs at t , $t-6$ h, and $t-12$ h, and not iteratively as we or the authors of [1] did. Their RMSE of Z_{500} and T_{2m} was about 30% lower for 3 or 5 day forecast time than the same measures for the CNN for Sec. 4.2. However, for direct forecasting one only needs to minimize the loss function for the quantity of interest (e.g. Z_{500} after 5 days), while for the iterative approach one needs to successfully forecast all the quantities. In the case of this residual network, this would mean minimizing a loss function over 114 fields desired for forecasting the next step. When the authors of [4] tried that, their forecast soon blew up.

5 Discussion and conclusions

After getting familiarized with more complex networks, we can propose some rather simple improvements that would probably improve our network’s performance. The most obvious one is probably the choice of padding of the input layer. For a 2-dimensional input, we should include periodic boundary conditions in the zonal direction instead of zero-padding. For a 1-dimensional input, on the other hand, the periodic boundary conditions are at least approximately already taken into account, because flattening of the original 2-dimensional array causes, for example, the most “Western” point at 78°N to be the neighbouring vector element of the most “Eastern” point at 84°N . This could be a reason, why

the networks with 1-dimensional inputs produced better forecasts than networks with 2-dimensional inputs (with the same number of trainable parameters). However, even for the 1-dimensional inputs the input layer could use better padding which would obey the boundary conditions in the meridional direction. With proper padding, the networks for 2-dimensional inputs would likely outperform the networks for 1-dimensional inputs.

We believe that leaving out the values at 84 °S from the 2-dimensional input fields causes a smaller amount of errors than the inefficient padding in the input layer. However, the Gaussian grid itself is also inappropriate for CNNs, because there we cannot avoid the inconvenience due to unequal spacing (in terms of kilometers) between grid points since we use the same kernel for all grid points. Thus the areas around poles have more effect on the final outcome of the network than similarly large areas near the equator. In dense NNs this problem might be coped with by restricting the values of the weights which connect the polar areas in the input layer with the first hidden layer. Nevertheless, the results from CNNs would still be much better than those from manipulated dense NNs due to the gap of a few orders of magnitude in the number of trainable parameters.

Finally, we can speculate that we would achieve better results if we also introduced additional fields to our network, beginning with zonal and meridional wind at 500 hPa, which are also part of the barotropic vorticity equation that we have already outperformed. However, it is worth mentioning that we briefly tried to do this with a network with a similar structure to our network (only with different input and output layers) and the results were worse than those of our network. The most probable cause for this failure is the fact that there is a different optimal number of trainable parameters for different input shapes.

To conclude this paper, we can ask ourselves whether it will ever be possible to fully replace NWP with ML approaches. The quality of the results of my CNN for a simple input on a basic regular grid was comparable to the quality of the results of NWP in 1950s. A CNN with only four input quantities on a smarter grid then already gave very promising results, which were comparable to NWP from the end of 20th century and computed extremely quickly. The results for a complex residual network with 38 input quantities, however, revealed a possibly unsurmountable obstacle – it is nearly impossible to minimize a loss function with hundreds of thousands of elements that depend on millions of trainable parameters. This resembles the problems that occur in NWP with data assimilation, however there we can sufficiently avoid it with a proper initial guess, whereas in NNs with dozens of layers it is nearly impossible to make a good initial guess for weights. To overwhelm this problem, we would need a lot more historical data than it is currently available. So fully replacing NWP with ML approaches is not going to be possible at least for a while. However, a combination of NWP and ML techniques for weather forecasting is already taking place and in future the amount of ML in weather forecasting is definitely going to grow.

References

- [1] J. A. Weyn, D. R. Durran, and R. Caruana, “Improving data-driven global weather prediction using deep convolutional neural networks on a cubed sphere,” *Journal of Advances in Modeling Earth Systems*, vol. 12, no. 9, p. e2020MS002109, 2020.
- [2] J. A. Weyn, D. R. Durran, and R. Caruana, “Can machines learn to predict weather? using deep learning to predict gridded 500-hPa geopotential height from historical weather data,” *Journal of Advances in Modeling Earth Systems*, vol. 11, no. 8, pp. 2680–2693, 2019.
- [3] S. Scher, “Toward data-driven weather and climate forecasting: Approximating a simple general circulation model with deep learning,” *Geophysical Research Letters*, vol. 45, no. 22, pp. 12616–12622, 2018.

- [4] S. Rasp and N. Thuerey, “Data-driven medium-range weather prediction with a Resnet pretrained on climate simulations: A new model for WeatherBench,” *Journal of Advances in Modeling Earth Systems*, vol. 13, no. 2, p. e2020MS002405, 2021.
- [5] E. N. Lorenz, “Deterministic nonperiodic flow,” *Journal of Atmospheric Sciences*, vol. 20, no. 2, pp. 130 – 141, 1963.
- [6] P. Bauer, A. Thorpe, and G. Brunet, “The quiet revolution of numerical weather prediction,” *Nature*, vol. 525, no. 7567, pp. 47–55, 2015.
- [7] S. Čopar, “Praktikum strojnega učenja v fiziki: Nevronske mreže.” [Lecture notes, University of Ljubljana, unpublished, 2020].
- [8] S. Čopar, “Praktikum strojnega učenja v fiziki: Konvolucijske nevronske mreže.” [Lecture notes, University of Ljubljana, unpublished, 2020].
- [9] H. Hersbach, B. Bell, P. Berrisford, S. Hirahara, A. Horányi, J. Muñoz-Sabater, J. Nicolas, C. Peubey, R. Radu, D. Schepers, *et al.*, “The era5 global reanalysis,” *Quarterly Journal of the Royal Meteorological Society*, vol. 146, no. 730, pp. 1999–2049, 2020.
- [10] T. Palmer, “A vision for numerical weather prediction in 2030,” *arXiv preprint arXiv:2007.04830*, 2020.
- [11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.