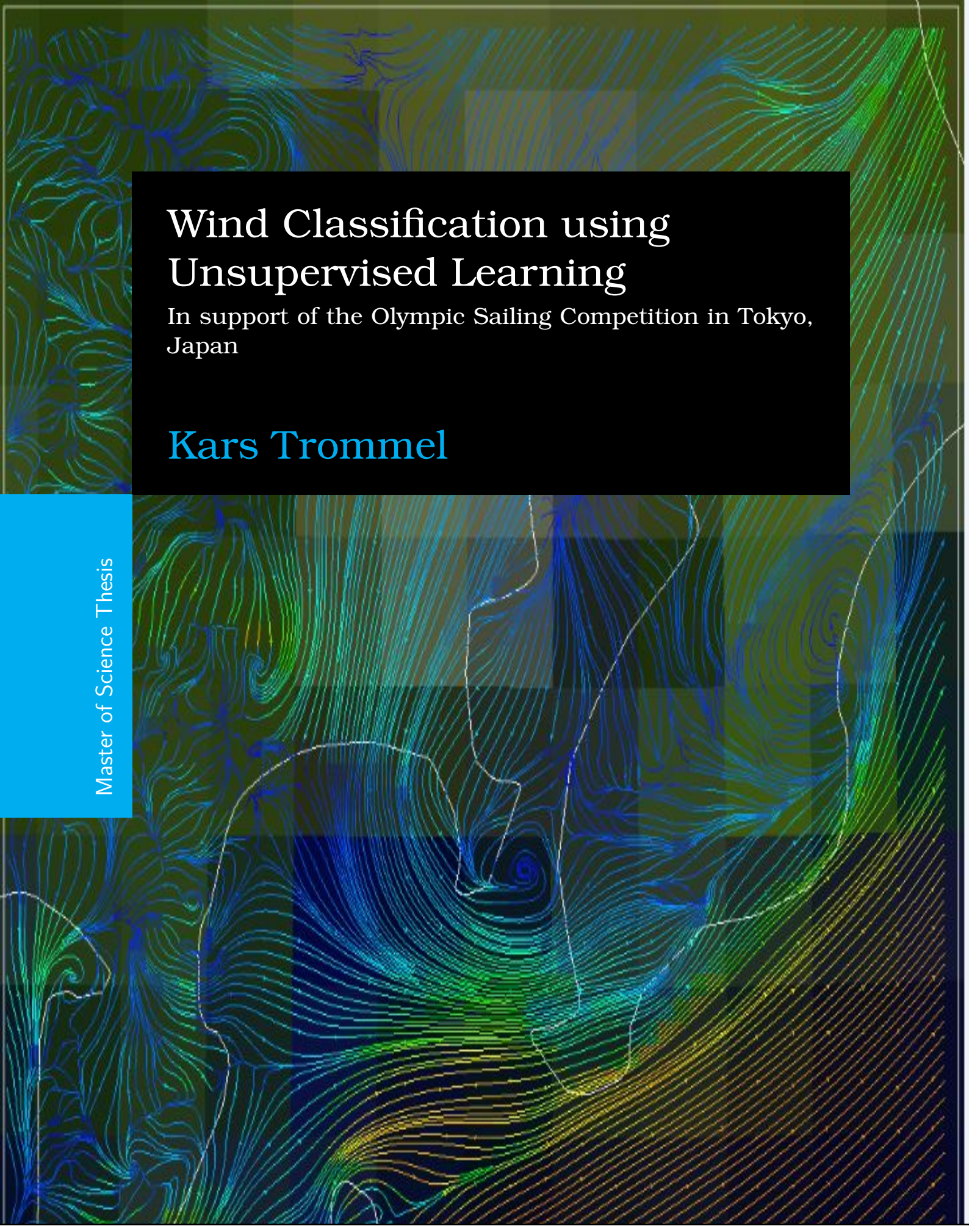


Wind Classification using Unsupervised Learning

In support of the Olympic Sailing Competition in Tokyo, Japan

Kars Trommel

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Wind Classification using Unsupervised Learning

In support of the Olympic Sailing Competition in Tokyo, Japan

Master of Science Thesis

For the degree of Master of Science in Systems and Control Engineering
at Delft University of Technology

Kars Trommel

April 14, 2020

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

The work in this thesis was supported by the Sailing Innovation Centre. Their cooperation is hereby gratefully acknowledged.

Abstract

During the preparation for the Olympic Sailing Competition, held in 2021 in Tokyo, Japan, the Dutch National Sailing Team encountered days with unpredicted wind behaviour. To gain more understanding in the wind patterns occurring, a deep learning based approach is taken.

The goal of this research is to find out if unsupervised learning methods can contribute to wind pattern classification. It can then be investigated if the classification can increase understanding in specific wind patterns.

The input data for the unsupervised learning model consists of 40 years of reanalysis wind speed data of an area including Japan. To classify the wind patterns, the dimensionality of the input data is reduced using different autoencoders. This reduced dimensional form is then clustered using K-means clustering. The results of the K-means algorithm are compared and the best autoencoder is chosen. The resulting clusters are analyzed for extreme wind patterns, such as typhoons.

It is expected that these wind patterns will be clustered together. To check this, the cluster containing typhoon Jebi, the typhoon which caused the highest insurance cost ever in Japan, is analyzed. If this cluster contains typhoons, unsupervised learning is able to provide useful information regarding wind patterns.

The best working autoencoder used in this research is the 3D CNN autoencoder. Using the 3D CNN autoencoder, some clusters with specific wind patterns are found. The cluster containing typhoon Jebi consists of 95.8% of typhoons, from which it can be concluded that unsupervised learning is a valid method for wind pattern classification.

Glossary

List of Acronyms

NWP	Numerical Weather Prediction
AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
MAE	Mean Absolute Error
JMA	Japan Meteorological Agency

Acknowledgements

I would like to thank my supervisor Dr. S. Basu for his assistance during the writing of this thesis. You inspired me to innovate in this field of interest. You managed to keep me motivated and curious, which I highly appreciate.

Furthermore I would like to thank Dr.ir. T. van den Boom for his time and effort to guide me in the right direction.

Delft, University of Technology
April 14, 2020

Kars Trommel

Table of Contents

Chapter 1

Introduction

In the summer of 2021, the Olympic Games will be held in Tokyo, Japan. Slightly south of Tokyo, in the Sagami Bay area, the sailing competition will take place. When the sailors of the Dutch National Sailing Team were practicing on site, they encountered unpredicted local wind behaviour. For the sailors, local wind behaviour can be of critical importance during the competition. When they are able to choose a path with slightly favourable wind conditions, it can be the difference between a gold or silver medal.

Weather forecasting is an interesting, but complex phenomenon. Weather is dependent on a lot of factors and can change rapidly. Mankind always has tried to predict the weather, and has been increasingly successful. Still, new forecasting techniques are being developed to improve the forecasts.

Currently, most weather forecasts are based on Numerical Weather Prediction (NWP) models. These NWP models evaluate the weather over a geographical location by dividing it into a grid and discretely solve the Navier-Stokes equations for fluids [1] for every grid point. This however, is computationally very heavy for fine resolution forecasting. In the case of the 2020 Olympic Games, a resolution of less than 100m is desired to recognize local wind behaviour, as small scale wind behaviour can only be seen using fine resolution. Using NWP models, this can take up to several thousands CPU hours on a high-performance computing cluster. Furthermore, the NWP models do not provide any degree of certainty as how likely the forecast is going to be correct.

Using NWP models in a geographical easy location will yield in highly accurate forecasts. But the geographical location of the Sagami Bay is making wind forecasting harder. The wind speed is influenced by mountains and islands nearby, which decreases the wind forecast accuracy.

In a meeting with the head coach of the Dutch National Sailing Team, Mr. Jaap Zielhuis, he mentioned that it is important for the Sailing Team to be able to have an understanding of wind patterns. Currently, they do not have any idea when the forecast is reliable and when it is not. Some coaches do not want to make a detailed race plan for the next day, because there is a chance of unknown magnitude that the forecast is not reliable at all. Gaining

understanding in wind patterns might cause a better understanding in the wind forecast, allowing the coaches and sailors to make a better race strategy.

More and more detailed weather data is becoming publicly available. With an increase in available data, it might be possible to implement Artificial Intelligence (AI) to gain more understanding in the wind forecasts. AI, and specifically deep learning, is becoming increasingly popular and works best with a lot of carefully selected data.

Understanding wind patterns in the Sagami Bay can be crucial for the Dutch National Sailing Team. Identifying wind patterns that are frequently occurring can be helpful in pursuing this understanding. Furthermore, it might be useful to investigate how certain wind patterns develop. Deep learning can be a tool to achieve that, as a lot of data is currently available to analyse. Up to now, as far of the author knows, unsupervised learning methods that use autoencoders for dimensionality reduction have not yet been applied for classification of wind patterns.

Therefore, the aim of this research is to create an unsupervised classification of wind patterns.

Chapter 2

Background Information

In the Introduction, deep learning was briefly mentioned. In this chapter, deep learning is further explained using a few important concepts. Firstly, a regular Artificial Neural Network (ANN) is shown with the corresponding calculations. Then, it is pointed out how an ANN is trained. Different types of learning are briefly touched upon and a Convolutional Neural Network (CNN), an important variation on a ANN, is clarified. Finally, dimensionality reduction with the use of an autoencoder is described.

2-1 Artificial Neural Network

An ANN is inspired by the neural network in our brains. The idea is to have many computational units that become intelligent via their interactions with one another [2]. To do so, it uses layers of connected processors, called neurons, that are being activated by certain inputs [3]. The structure of an ANN can be seen in Figure 2-1, as taken from Figure 1c in LeCun, Bengio and Hinton (2015) [4]. All of the layers between the input and the output layer are called hidden layers.

The input neurons are assigned a value between -1 and 1, dependent on the given input. A neuron in the next layer will be given a value based on the neurons in the previous layer. This is denoted by the arrows in Figure 2-1. It is multiplication and addition process. The formula is given in Srivastava et al. (2014) [5]:

$$y_j^{(l+1)} = f(z_j^{(l+1)}) = f\left(\sum_{i=1}^{n} w_{ij}^{(l+1)} y_i^{(l)} + b_j^{(l+1)}\right) \tag{2-1}$$

The subscript denotes a specific neuron, while the superscript denotes the concerning layer. The amount of neurons in a layer is specified by n, y is the final value of the neuron, w_{ij} is the weight from i to j, and b is the bias. Sometimes the bias is left out, as in LeCun, Bengio and Hinton (2015) [4]. The function $f()$ is called an activation function, and z_j is the value of the neuron before the activation function. In vector form this can be written as [5]:

$$y_j^{(l+1)} = f(z_j^{(l+1)}) = f\left(\mathbf{w}_j^{(l+1)} \mathbf{y}^{(l)} + b_j^{(l+1)}\right) \tag{2-2}$$

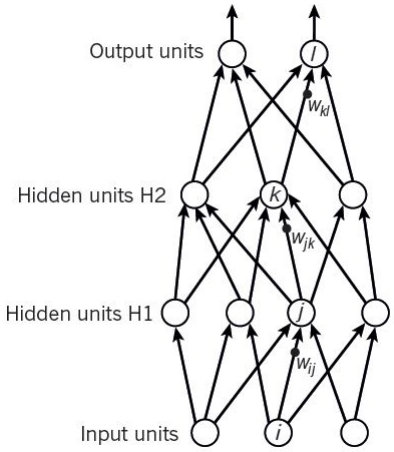


Figure 2-1: Structure of an Artificial Neural Network [4]

The bold characters denote vectors. $\mathbf{w}_j$ represents all the weights going towards y_j. If this notation is generalized for an entire layer, the following notation is used:

$$\mathbf{y}^{(l+1)} = f(\mathbf{z}^{(l+1)}) = f\left(\mathbf{W}^{(l+1)}\mathbf{y}^{(l)} + \mathbf{b}^{(l+1)}\right) \tag{2-3}$$

Where the weights $\mathbf{W}$ are presented in matrix form.

The activation function $f()$ in Equations 2-1, 2-2, and 2-3 introduces non-linear properties to the Artificial Neural Network and is considered very important. As Ramachandran, Zoph and Le (2017, p.1) stated: "The activation function plays a major role in the success of training deep neural networks" [6].

The first activation functions were the hyperbolic tangent (tanh) and the sigmoid function. Both are S-shaped functions and are shown in Figure 2-2, retrieved from Figure 1 in Glorot, Bordes and Bengio (2011) [7]. The formulas are [8]:

$$\text{sigmoid} : f(x) = \frac{1}{1+e^{-x}}$$

$$\tanh : f(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

The sigmoid function transforms the input domain $(-\infty, \infty)$ onto the domain $\mathcal{D}_{sig} = (0, 1)$. The tanh function maps the same input domain onto the domain $\mathcal{D}_{tanh} = (-1, 1)$.

Lately, other nonlinear activation functions have been used, with the most popular one being the Rectified Linear Unit (ReLU) function. Another similar looking function is the softplus function. Both are shown in Figure 2-3, retrieved from Figure 2 in Glorot, Bordes and Bengio (2011) [7]. The formulas are [7]:

$$\text{ReLU} : f(x) = \max(0, x)$$

$$\text{softplus} : f(x) = \log(1 + e^{x})$$

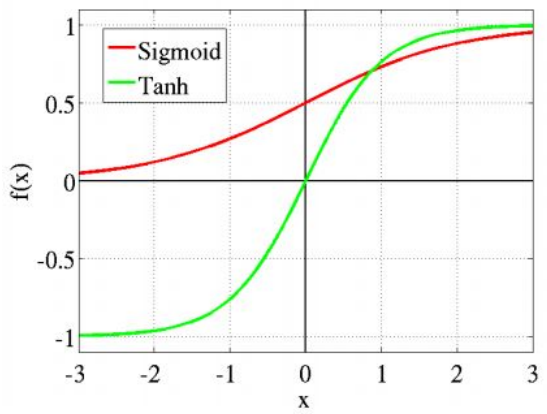


Figure 2-2: Sigmoid and tanh [7]

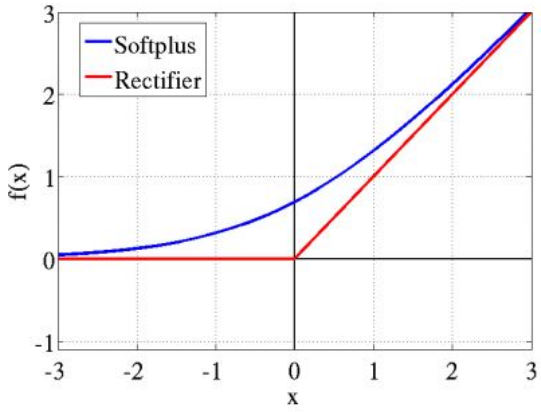


Figure 2-3: ReLU and Softplus [7]

For these activation functions, the input domain is mapped on the domain $\mathcal{D}_{soft}(0, \infty)$ and $\mathcal{D}_{ReLU} = [0, \infty)$. As can be seen, the upper limit of the domain is not bounded. This is an advantage. As Ramachandran, Zoph and Le (2017, p.1) explained, using a ReLU as activation function will optimize a deep network more easily than when using a sigmoid or tanh as activation function. This is because for a positive value in the activation function, the ReLU lets the gradients flow. Due to the simplicity and effectiveness of the ReLU, it has become the default activation function used across the deep learning community. [6]

To make the Artificial Neural Network perform accurately, the weights w and biases b from Equation 2-1 need to be assigned carefully. To do so, the ANN will be subject to training.

2-2 Training Artificial Neural Networks

Training consists of giving the algorithm a certain input, activating the input neurons in a particular way. The algorithm then computes a predicted output, using the algorithm with initial weights. With training data, the actual output is known. For T being the amount of training data, the set of inputs and outputs will be in distribution $D = \{(x^{(1)}, t^{(1)}), (x^{(2)}, t^{(2)}), ..., (x^{(T)}, t^{(T)})\}$ [9], with (x, t) being a pair of input x and the corresponding output, target value t. For input x, the Artificial Neural Network will compute output y. This computed outcome y is compared with the target value t, defining an error function [9]:

$$E = \frac{1}{2}\sum_{t=1}^{T}(y - t)^2 \tag{2-4}$$

The goal of the training is to minimize the cost function for all the training data. This goal is achieved by backpropagation. Backpropagation will use an optimization algorithm to reduce the cost function as quickly as possible by tweaking the weights w and b from Equation 2-1.

A popular optimization algorithm is Stochastic Gradient Descent (SGD). This algorithm modifies the weights using the partial derivative of the squared error with respect to that weight [10]. The disadvantage of SGD is that it has trouble around local minima where the surface curves more steeply in one dimension than in another [11]. Qian (1999) found that

momentum can help to speed up this process, adding a fraction γ to the update vector [12]. That resulted in the Adagrad [13] method.

A more recent discovery, developed especially for application in machine learning models, is the Adam optimizer [14]. Adam is a combination of two popular methods: Adagrad and RMSProp. RMSProp is an unpublished, adaptive learning rate method developed by dr. G. Hinton. This method is shown in his lecture slides [15].

The creators of the Adam algorithm also proposed the Adamax algorithm [14]. The Adamax uses the L-infinity norm instead of the L2-norm, and is considered a more stable optimization algorithm than the Adam algorithm [16].

It was mentioned by He (2014) that the standard backpropagation tends to get trapped in local minima [9]. LeCun, Bengio and Hinton (2015) counter that, describing that local minima are rarely a problem due to the multidimensional space in which the error function lays. This theory is supported by Dauphin et al. (2014, p. 9): "[...] in contrast to conventional wisdom derived from low dimensional intuition, local minima with high error are exponentially rare in high dimensions." [17].

2-3 Types of Learning for Artificial Neural Networks

In Section 2-2, it was assumed that the output of the model was known. This is supervised learning; the inputs and corresponding outputs are all known. The known output is compared to the output computed by the model, and using backpropegation the weights of the model are adjusted to approach the correct output. Supervised learning is the most common form of deep learning [4].

The downside of supervised learning is that a lot of labeled data is needed to train the model. And as LeCun, Bengio and Hinton (2015, p. 442) stated: "Human and animal learning is largely unsupervised: we discover the structure of the world by observing it, not by being told the name of every object." [4]. Because ANNs are designed to mimic a brain, unsupervised learning is expected to become more important in the future.

Unsupervised learning makes use of unlabeled data. Because the error function from Equation 2-4 cannot be computed without comparing the model output with the target value, the model should be trained otherwise. A well-known unsupervised learning method is K-means clustering [18]. The K-means clustering algorithm will cluster data with similar properties, without labeling the data. The model will do so by inserting certain cluster centers, also called centroids, and adding data to the cluster with the nearest centroid. Then, the error between the centroid and the data points inside the cluster is computed. Afterwards, the cluster centers will be adjusted until the error has minimized. The downside of K-means clustering is that the amount of clusters need to be predefined. Furthermore, the cluster centers need to be initialized carefully [18].

2-4 Convolutional Neural Networks

There are a lot of different ANNs, differing in how the neural layers are constructed. The most important algorithm for this research is discussed.

Convolutional Neural Network (CNN)

Convolutional Neural Networks are great for capturing spatial patterns in data [19]. A CNN uses multiple layers. The successive layers are designed to learn progressively higher level features, with the last layer producing categories [20]. Figure 2-4, obtained from Figure 1 in Huang and LeCun (2006, p. 4), shows an example of a CNN [20]. Often, images are used as input. As can be seen in Figure 2-4, the first layer can contain multiple layers of inputs - also called feature maps. Multiple feature maps can denote an image pair as input, or when considering a colored picture, it can also denote the use of RGB values as feature maps.

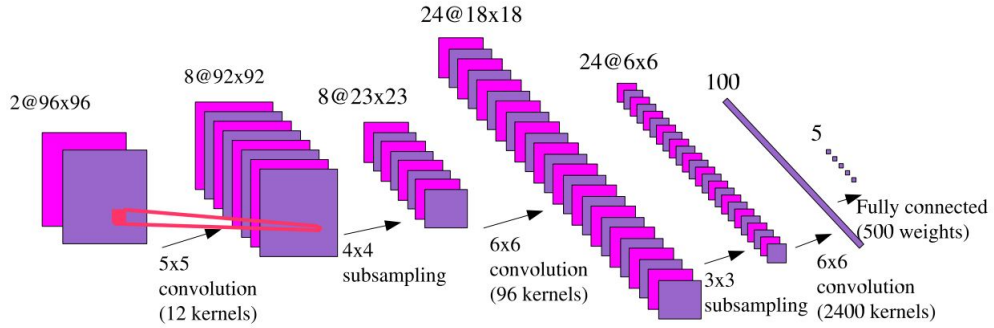


Figure 2-4: Architecture of a Convolutional Neural Network [20]

Figure 2-4 shows that the CNN makes use of multiple different layer operations, namely three convolution layers, two subsampling layers, and a fully connected layer. The operations used will be explained below.

Convolution Layer

The convolution layer will try to extract some features from the input image using multiple filters, also called kernels. These kernels are the size of $n \times n$ pixels, and will "scan" the input feature map on every pixel. Lets say in the input feature map, pixel s is subject to a convolution step with a 3×3 sized kernel. The surrounding pixels of pixel s will be taken into account due to the size of the kernel. With element-wise multiplication, the value s in the output layer will be calculated.

A visual example of the convolution layer can be seen in Figure 2-5, retrieved from Ludwig [21]. The new pixel color is determined by element-wise multiplication. In Figure 2-5, the red squared value is taken into account. A kernel is placed on top of this and the neighbouring pixels, determining the new value of this pixel.

Because the kernel has a certain width larger than 1, the border values of the input feature map cannot be computed as they do not have neighbouring pixels at one side. This shrinks

the feature map with a size of (kernel size - 1). The filter size of the first convolution layer in Figure 2-4 is 5×5 pixels, shrinking the feature map from 96×96 pixels to 92×92 pixels.

To counteract the shrinkage of the feature map, padding can be used. This means that pixels with value zero are added around the feature map, with a width of (kernel size -1)/2 on every side (left, right, top and bottom of the input). This makes sure the size of the feature map is contained after the convolution layer [22]. It can be seen that the input feature map in Figure 2-5 was padded, due to the fact that the output feature map has the same size as the input feature map.

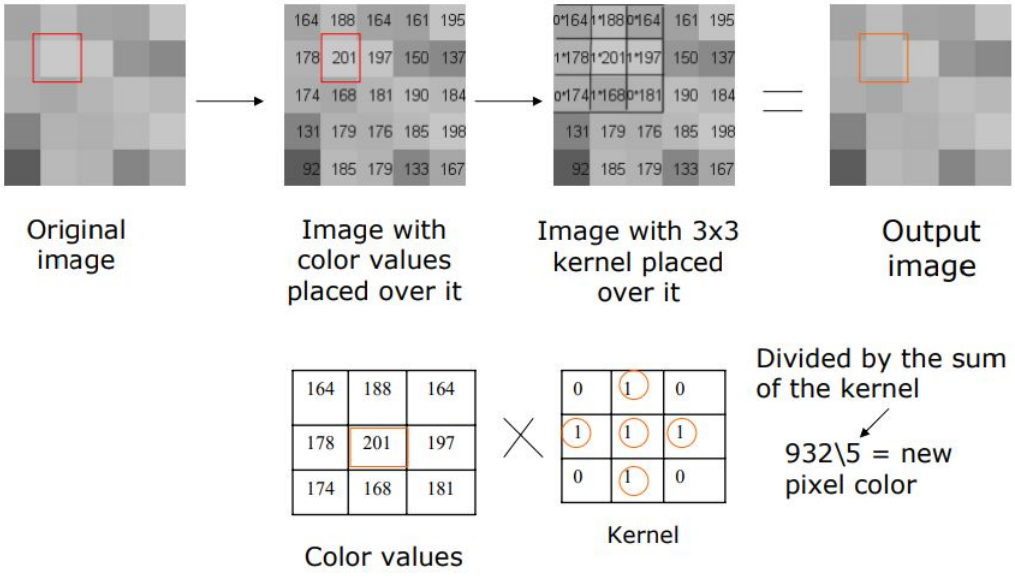


Figure 2-5: Convolution layer [21]

The designer of the CNN can determine how many output feature maps the convolution layer will produce. The designer of the CNN in Figure 2-4 chose to produce 8 feature maps in the first convolution layer, 24 feature maps in the second convolution layer, and 100 feature maps in the third convolution layer [20].

The amount of kernels in the convolution layer is a so-called "hyperparameter", meaning that this will not be optimized but is a designer choice. However, the values contained inside the kernel will be optimized. Another hyperparameter is the size of the kernels.

Subsampling Layer

The subsampling layer in Huang and LeCun (2006) [20] takes the average of a $m \times m$ pixel block and multiplies that by trainable scalar β. Then next pixel block will usually be placed next to the previous pixel block, such that there is no overlap between the previous and the current pixel block. This is called the stride: how many pixels a pixel block moves. So for subsampling, the stride usually corresponds with the pixel block size to omit overlap. Figure 2-4 shows that the subsampling size of 4×4 pixels results in a factor 4 size decrease of the feature map. The amount of feature maps remains unchanged.

Subsampling results in a feature map with a lower resolution. Alternating the convolution layers and the subsampling layers will preserve the features and creates robustness to irrelevant variabilities [20]. Furthermore, the use of subsampling layers will reduce the computational costs of the model.

Instead of taking the averaging of the pixel block, as Huang and LeCun (2006) [20] proposed, it is also possible to take the maximum value of a pixel block. This is called max pooling [23].

Fully Connected Layer

The last layer of the Convolutional Neural Network, as shown in Figure 2-4, is a fully connected layer. A fully connected layer is a "normal" layer in a ANN, as shown in 2-1. The layers are connected using weights, with these weights being optimized.

2-5 Autoencoders

An autoencoder is a type of neural network that is used in high-dimensional inputs, such as images. Usually, high-dimensional phenomena are dominated by a small amount of simple variables [24]. To reduce the dimensionality of the input and find the important variables, which can be necessary for clustering, an autoencoder is used. Using dimensionality reduction, loss of information is inevitable. It is a goal on its own to make this loss as small as possible.

An autoencoder consists of an encoder and a decoder [25]. The encoding part reduces the dimensionality of the input, or in other words, it compresses the input image. The decoder tries to recreate the input image from the reduced dimensional form. A simplified example of an autoencoder is shown in Figure 2-6.

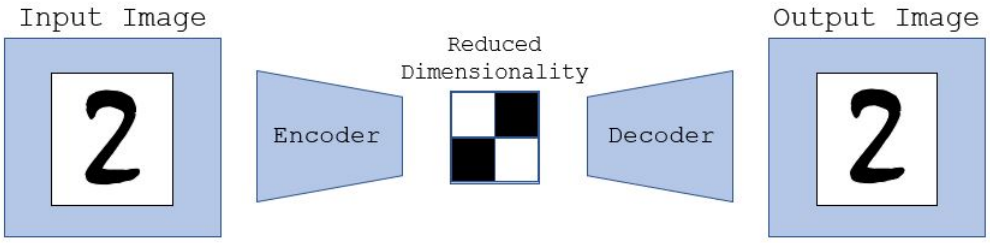


Figure 2-6: A simplified example of an autoencoder

The autoencoder is trained with backpropagation. This is a method that usually does not work with unlabeled data, as without labeled data the error function in Equation 2-4 cannot be computed. But because the output of an autoencoder is compared to its input, it is possible to compute the error function. The autoencoder takes a lot of input data, reduces the dimensionality for every input, and decodes the reduced dimensionality for every data sample. The weights and biases of the autoencoder are updated after a predefined number of input images ran through it. This is the batch size of the autoencoder. One run of the entire dataset is called an epoch.

Ideally, the input and ouput image are identical for every data point. In that case, the reduced dimensionality filtered out some of the nonessential information contained in the high-dimensional input. But for images, every dimension in a data sample is a pixel. To reduce the dimensionality means to ignore some pixels, which makes recreating the image as accurate as possible for an entire dataset a hard task. Especially if the reduced dimension is a lot smaller than the input dimensions, or if the dataset consists of images which differ a lot.

Finally, to make the autoencoder perform optimally, every image in the dataset should be of same size and quality. Two examples are of well-known datasets used for classification are the MNIST dataset, which contains written digits from 0 to 9, and the Olivetti face dataset, shown in Figure 2-7, retrieved from Figure 5 in Wang, Yao and Zhao (2015) [24].

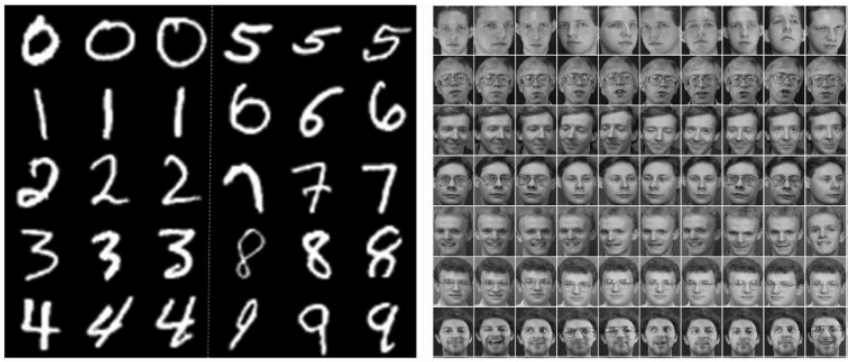


Figure 2-7: The MNIST and the Olivetti dataset, retrieved from Figure 5 in Wang, Yao and Zhao (2015) [24]

Chapter 3

Methodology

When starting a research, it is first important to generate a proof of concept. This means it should be proven that the general idea works, before the complexity of the model is increased. To do so, a large domain of interest is chosen. This is because a large domain has large scale wind patterns. In the area of Japan, an example of a large scale wind pattern is a typhoon. Large scale wind patterns are easy to identify, even when using a large grid size. The ERA5 dataset consists of data with a 30×30 km resolution. This is large enough for this cause, meaning that it is not necessary to run a WRF simulation for our input data. This saves time and computational power.

The domain that is used is shown in Figure 3-1.
The area is defined by longitude and latitude:
Longitude: 120.00 - 155.00 °E
Latitude: 25.00 - 50.00 °N

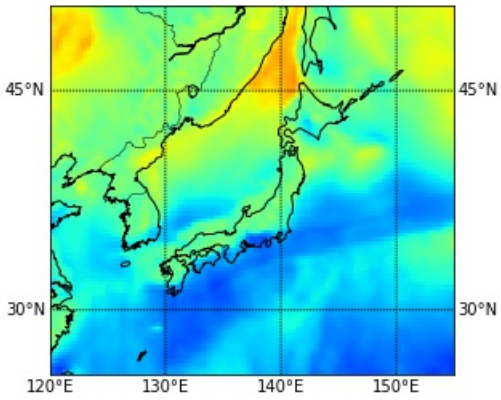
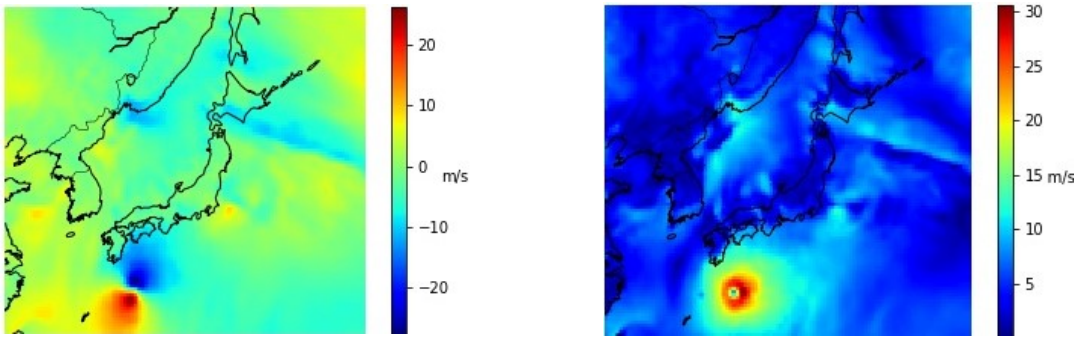


Figure 3-1: The large domain used for classifying wind patterns

To start with the first classifying algorithm, a dataset at the specified domain should be retrieved. As mentioned before, the ERA5 dataset is chosen as a suitable dataset. For the first tries, it is important to keep the dataset small, so the model will converge quickly. But if the dataset is too small, it is impossible to learn specific wind patterns. As an arbitrary choice, the first dataset that is used contains data from the years 2000 - 2018, with one data point per day at 11AM. This is chosen so that the data remains as uncorrelated as possible, to see whether the autoencoder performs as expected. The Python code used for preparation of the input data is shown in Appendix D-1.

Because the data does not contain labels with the correct weather pattern, it is not obvious how the clustering should be deemed correct. A method should be developed to analyse clustering performance. In September 2018, Japan was hit by typhoon Jebi, causing an insured loss of approximately 970 billion yen (≈ 8.6 billion US Dollars). This was the largest insured loss event in Japan [26]. To compare clusters, it is checked what cluster typhoon Jebi is clustered in. Then the cluster is checked for other typhoons, as it is expected that this type of extreme wind behaviour should be clustered together. Typhoon Jebi, in the first dataset, is clearly seen on the 3rd of September. In the small dataset, this is data sample #6820. Both the zonal wind speed (as explained in the next section) and the absolute wind speed are shown in Figure 3-2.



(a) The zonal wind speed of Typhoon Jebi

(b) The absolute wind speed of Typhoon Jebi

Figure 3-2: Typhoon Jebi, the 3rd of September 2018

The clustering algorithm that is used is the K-means clustering algorithm. This is a clustering algorithm that is often used in unsupervised learning, as discussed in Section 2-3. The implementation in Python is fairly simple and can be found in Appendix D-2.

Initially, the only variable taken into consideration for clustering is u10. This is the zonal component (East-Western component) of the wind speed at 10 meter height. It is tried to classify wind patterns based on u10, because the u10 data can be simply downloaded without the need to preprocess the data. It can be seen as a proof of concept. Afterwards a comparison is made with a classification including u10 and v10, the meridional component (North-Southern component) of the wind speed at 10 meter height.

The total input data (u10) has a size of (6940, 101, 141). This means 6940 data points with 101 values in the North-South direction and 141 values in the East-West direction. The input data is slightly adjusted to (6940, 100, 140) by ignoring the first row and column of the input. This is done to enable the use of subsampling layers in the Convolutional Neural Network (CNN) for future steps.

It is investigated whether the clustering benefits from a dimensionality reduction using an autoencoder. The desired behaviour is that important wind features are retained in the compressed data, making the clustering based more on the important wind features. The autoencoders are increasingly complex.

When using an autoencoder, it is necessary is to allocate the number of neurons in the middle layer. The first autoencoder is trained four times, using 10, 100, 500, and 1000 neurons in the middle layer. For clustering, it is assumed that a lower amount of neurons provide better

clustering. However, using less neurons, the decoded image becomes less detailed. Using a particular amount of neurons in the middle layer means it is expected that these neurons contain the same information as the decoded image. Thus a proper amount of neurons should be chosen, maximizing the detail of the decoding image whilst minimizing the amount of neurons in the middle layer.

No Autoencoder

Initially, a control group is considered. No autoencoder is used for this data. The clustering algorithm is directly applied on the input data. The input data is flattened, as otherwise it exceeds the maximum input dimension of the K-means clustering algorithm, which is two. The size of the input data is hence (6940, 14000). The implementation in Python is shown in Appendix D-5-1.

Single Layered Autoencoder

The first autoencoder that is applied is a single layered autoencoder. This model is also used to define the amount of neurons in the middle layer. This is done by comparing the autoencoder performance using 10, 100, 500, and 1000 neurons in the middle layer. The performance of the autoencoder using 100 neurons was considered as optimal choice.

The single layered autoencoder is shown in Figure 3-3. The input is the u10 wind speed in m/s. For the encoder part, the input dimension is flattened. Then, a fully connected layer is used to compress the information to 100 values. The decoder tries to recreate the input image from this 100 values. The layer with 100 neurons is fully connected to a 14000 neuron layer, which is reshaped into the shape of the input data. The structure of the autoencoder can be seen in Figure 3-3 in Appendix A, and the implementation in Python can be found in Appendix D-5-2.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 100, 140, 1)	0
flatten_2 (Flatten)	(None, 14000)	0
dense_3 (Dense)	(None, 100)	1400100
dense_4 (Dense)	(None, 14000)	1414000
reshape_2 (Reshape)	(None, 100, 140, 1)	0

Total params: 2,814,100
Trainable params: 2,814,100
Non-trainable params: 0

Figure 3-3: Structure of the single layered autoencoder

Deep Autoencoder

To increase the complexity of the autoencoder, two fully connected layers are added in the encoder, and two fully connected layers are added in the decoder. The structure of the deep autoencoder is shown in Figure 3-4. The Python code used for implementing the autoencoder can be found in Appendix D-5-3.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 140, 1)	0
flatten_1 (Flatten)	(None, 14000)	0
dense_1 (Dense)	(None, 1000)	14001000
dense_2 (Dense)	(None, 500)	500500
dense_3 (Dense)	(None, 100)	50100
dense_4 (Dense)	(None, 500)	50500
dense_5 (Dense)	(None, 1000)	501000
dense_6 (Dense)	(None, 14000)	14014000
reshape_1 (Reshape)	(None, 100, 140, 1)	0

Total params: 29,117,100
Trainable params: 29,117,100
Non-trainable params: 0

Figure 3-4: Structure of the deep autoencoder

CNN Autoencoder

CNN networks tend to capture spatial and temporal dependencies [19], which is likely to have a positive impact on the clustering of wind patterns. Three CNN layers are added to both the encoder and the decoder, applying them symmetrically. One of the layers contains 64 kernels, one contains 16 kernels, and one layer contains 1 kernel. The structure of the CNN autoencoder is shown in Figure 3-5. The Python code can be found in Appendix D-5-4.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 140, 1)	0
conv2d_1 (Conv2D)	(None, 100, 140, 64)	640
conv2d_2 (Conv2D)	(None, 100, 140, 16)	9232
conv2d_3 (Conv2D)	(None, 100, 140, 1)	145
flatten_1 (Flatten)	(None, 14000)	0
dense_1 (Dense)	(None, 100)	1400100
dense_2 (Dense)	(None, 14000)	1414000
reshape_1 (Reshape)	(None, 100, 140, 1)	0
conv2d_transpose_1 (Conv2DTr	(None, 100, 140, 16)	160
conv2d_transpose_2 (Conv2DTr	(None, 100, 140, 64)	9280
conv2d_transpose_3 (Conv2DTr	(None, 100, 140, 1)	577

Total params: 2,834,134
Trainable params: 2,834,134
Non-trainable params: 0

Figure 3-5: Structure of the CNN autoencoder

Deep CNN Autoencoder

A combination of the deep and CNN autoencoder is applied. Two hidden layers are added in the middle of the autoencoder. The deep CNN model is visualized in Figure 3-6. Python code needed for implementation is shown in Appendix D-5-5.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 100, 140, 1)	0
conv2d_4 (Conv2D)	(None, 100, 140, 64)	640
conv2d_5 (Conv2D)	(None, 100, 140, 16)	9232
conv2d_6 (Conv2D)	(None, 100, 140, 1)	145
flatten_2 (Flatten)	(None, 14000)	0
dense_3 (Dense)	(None, 500)	7000500
dense_4 (Dense)	(None, 250)	125250
dense_5 (Dense)	(None, 100)	25100
dense_6 (Dense)	(None, 250)	25250
dense_7 (Dense)	(None, 500)	125500
dense_8 (Dense)	(None, 14000)	7014000
reshape_2 (Reshape)	(None, 100, 140, 1)	0
conv2d_transpose_4 (Conv2DTr	(None, 100, 140, 16)	160
conv2d_transpose_5 (Conv2DTr	(None, 100, 140, 64)	9280
conv2d_transpose_6 (Conv2DTr	(None, 100, 140, 1)	577

Total params: 14,335,634
Trainable params: 14,335,634
Non-trainable params: 0

Figure 3-6: Structure of the deep CNN autoencoder

CNN Autoencoder with Max Pooling Layer

To reduce the computational cost needed for training the model, two max pooling layers are added in the model. The max pooling layer takes filters of size (2, 2) with the output being the maximum value of these four values. The structure of the model is shown in Figure 3-7. The Python implementation can be found in Appendix D-5-6.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 100, 140, 1)	0
conv2d_4 (Conv2D)	(None, 100, 140, 64)	640
max_pooling2d_3 (MaxPooling2	(None, 50, 70, 64)	0
conv2d_5 (Conv2D)	(None, 50, 70, 16)	9232
max_pooling2d_4 (MaxPooling2	(None, 25, 35, 16)	0
conv2d_6 (Conv2D)	(None, 25, 35, 1)	145
flatten_2 (Flatten)	(None, 875)	0
dense_3 (Dense)	(None, 100)	87600
dense_4 (Dense)	(None, 875)	88375
reshape_2 (Reshape)	(None, 25, 35, 1)	0
conv2d_transpose_4 (Conv2DTr	(None, 25, 35, 16)	160
up_sampling2d_3 (UpSampling2	(None, 50, 70, 16)	0
conv2d_transpose_5 (Conv2DTr	(None, 50, 70, 64)	9280
up_sampling2d_4 (UpSampling2	(None, 100, 140, 64)	0
conv2d_transpose_6 (Conv2DTr	(None, 100, 140, 1)	577

Total params: 196,009
Trainable params: 196,009
Non-trainable params: 0

Figure 3-7: Structure of the CNN autoencoder with max pooling layers

Deep CNN with Max Pooling layer

Finally, the deep CNN autoencoder is analyzed after adding the max pooling layers. The structure of the deep CNN autoencoder is shown in Figure 3-8. The Python code can be found in Appendix D-5-7.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 140, 1)	0
conv2d_1 (Conv2D)	(None, 100, 140, 64)	640
max_pooling2d_1 (MaxPooling2	(None, 50, 70, 64)	0
conv2d_2 (Conv2D)	(None, 50, 70, 16)	9232
max_pooling2d_2 (MaxPooling2	(None, 25, 35, 16)	0
conv2d_3 (Conv2D)	(None, 25, 35, 1)	145
flatten_1 (Flatten)	(None, 875)	0
dense_1 (Dense)	(None, 500)	438000
dense_2 (Dense)	(None, 250)	125250
dense_3 (Dense)	(None, 100)	25100
dense_4 (Dense)	(None, 250)	25250
dense_5 (Dense)	(None, 500)	125500
dense_6 (Dense)	(None, 875)	438375
reshape_1 (Reshape)	(None, 25, 35, 1)	0
conv2d_transpose_1 (Conv2DTr	(None, 25, 35, 16)	160
up_sampling2d_1 (UpSampling2	(None, 50, 70, 16)	0
conv2d_transpose_2 (Conv2DTr	(None, 50, 70, 64)	9280
up_sampling2d_2 (UpSampling2	(None, 100, 140, 64)	0
conv2d_transpose_3 (Conv2DTr	(None, 100, 140, 1)	577

Total params: 1,197,509
Trainable params: 1,197,509
Non-trainable params: 0

Figure 3-8: Structure of the deep CNN autoencoder with max pooling layers

For all the autoencoders, the training data consists of 85% of the entire data set. Considering the first dataset, consisting of 6940 data samples, this results in 5899 data samples used for training. The validation data set contains the remaining 1041 samples.

The compressed input data of each of the autoencoders is clustered using the K-means algorithm. The Mean Absolute Error (MAE) of the autoencoders is analyzed, as well as all of the data points in the same cluster as typhoon Jebi. As stated in the introduction, it is hypothesized that similar typhoons will be clustered in the same cluster as typhoon Jebi.

One of the important parameter choices that should be made, is the amount of clusters that the K-means clustering algorithm uses. One of the ways to decide what amount of clusters to take, is to use the total inertia of the K-means algorithm. In the sklearn python database it is explained that the K-means algorithm minimizes the inertia, which is the within-cluster sum-of-squares. In formula form, this looks like [27]:

$$\sum_{i=0}^{n} \mathbf{min}(||x_i - \mu_j||^2)$$

Where x denotes the points in the cluster, and μ is the cluster centroid. To check whether the right amount of clusters is chosen, the inertia is calculated for every cluster size from 1 to 250. The Python code to do so is shown in section D-3 in Appendix D. The amount of clusters should be chosen where the increase of clusters causes a linear inertia decrease [28]. With greater amount of clusters, the inertia obviously decreases as more cluster centroids are added. But when this decrease becomes linearly, adding more clusters is not significantly beneficial.

After using solely u10 as input data, a layer of v10 input data is added as an extra dimension. It is chosen to make a comparison with u10 and v10 combined instead of taking the absolute wind speed as input, because when using the absolute wind, the direction of the wind is lost. Using u10 and v10, it is expected that the direction of the wind is contained in the reduced dimensional form.

Now, the input data has a dimension of (6940, 100, 140, 2), with the last dimension being u10 and v10. The regular 2D CNN layers, as they are used in both the CNN and deep CNN autoencoder, are able to handle this. 2D CNN layers use this extra dimension as a feature map. This means that this layer contains additional information of the input image. For the use of colored pictures for example, this can be the "RGB" information of the image, yielding in 3 feature maps: one for every colour. Using u10 and v10 in this dimension essentially means the same, as both of them contain information about the same input picture, both in the same value range.

The structures of the autoencoders using both u10 and v10 as input can be found in Appendix A-1. The Python implementation of the autoencoders can be found in Appendix D-6. The functionality of the autoencoders is compared, both using the typhoons contained in the same cluster as Typhoon Jebi and using the MAE of the validation data.

The autoencoder that performs best is then used for the dimensionality reduction of the entire dataset. The entire dataset consists of 40 years of ERA5 data, ranging from 1978 - 2018. Four time samples per day are used: 8 AM, 11 AM, 2 PM, and 5 PM are included in the dataset. This results in a dataset of size (58440, 100, 140, 2).

Finally, to investigate the development during a particular day, a 3D Convolutional Neural Network autoencoder is introduced. Using a 3D CNN, an extra time dimension is introduced, whilst still having 2 channels for the u and v component of the wind. The data will be split up in days, with four time-steps per day. This causes the input data to have size (14610, 100, 140, 4, 2).

The structure of the 3D autoencoder is shown in Figure 3-9. The slight changes in data preparation are shown in Appendix D-8-1, the Python code to implement the autoencoder is shown in Appendix D-8-2. The training data will be 85% of the 14610 days, resulting in 12419 data samples. The validation data consists of the remaining 15% - or 2191 data samples.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 100, 140, 4, 2)    0
_________________________________________________________________
conv3d_1 (Conv3D)            (None, 100, 140, 4, 64)   3520
_________________________________________________________________
conv3d_2 (Conv3D)            (None, 100, 140, 4, 16)   27664
_________________________________________________________________
conv3d_3 (Conv3D)            (None, 100, 140, 4, 1)    433
_________________________________________________________________
flatten_1 (Flatten)          (None, 56000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               5600100
_________________________________________________________________
dense_2 (Dense)              (None, 56000)             5656000
_________________________________________________________________
reshape_1 (Reshape)          (None, 100, 140, 4, 1)    0
_________________________________________________________________
conv3d_transpose_1 (Conv3DTr (None, 100, 140, 4, 16)   448
_________________________________________________________________
conv3d_transpose_2 (Conv3DTr (None, 100, 140, 4, 64)   27712
_________________________________________________________________
conv3d_transpose_3 (Conv3DTr (None, 100, 140, 4, 2)    3458
=================================================================
Total params: 11,319,335
Trainable params: 11,319,335
Non-trainable params: 0
```

Figure 3-9: Structure of the 3D CNN autoencoder

As described in the introduction, it is hypothesized that clustering the wind data will generate clusters consisting of the same wind features or wind behaviour. As an example, a typhoon is considered. After the study it can be concluded whether the clustering is performing as expected, or if this method is not able to distinguish different wind patterns. If this method succeeds, it can be investigated whether some clusters are closely related, by finding out what the chances are of a particular cluster happening the next day, given the cluster the day before.

If this method is able to shape clusters containing a specific wind behaviour, it can be attempted to cluster smaller scale wind patterns. This is information that is of importance for the Dutch National Olympic Sailing Team.

The computations are performed on a HP ZBook Studio G5. The computer contains an Intel i7-8750H processing unit, 16 GB RAM, and two GPUs: an Intel UHD Graphics 630 and a NVIDIA Quadro P1000 GPU. The computer runs on Windows. The code is written in Python, using a GPU enabled Jupyter Notebook IDE. The following libraries are installed:

- Python: 3.7.4
- Keras: 2.3.1
- Scikit-learn: 0.21.3
- NumPy: 1.16.5
- Pandas: 0.25.2
- netCDF4: 1.4.2
- Matplotlib: 3.1.1

Chapter 4

Results

4-1 U10 Classification

4-1-1 No autoencoder

Now that the domain and the dataset are specified, it is possible to start classifying the input data. At first, it is tried to classify the data without the use of an autoencoder. To be able to use the K-means clustering algorithm, the input data is flattened so that it has size (6940, 14000).

The K-means algorithm tries to cluster 6940 data points based on 14000 features (grid cells, or pixels of the input image). The algorithm does not know what features are important and what features are irrelevant. The K-means clustering algorithm is ran with default settings and 50 clusters.

The K-means clustering algorithm takes a little more than 6 minutes. Some typhoons are contained in the same cluster as typhoon Jebi, but also some non-typhoons are clustered within the same cluster. Within the first 10 data points in the cluster, 4 non-typhoons can be found. The details of the clustering of typhoon Jebi can be seen in Figure 4-1.

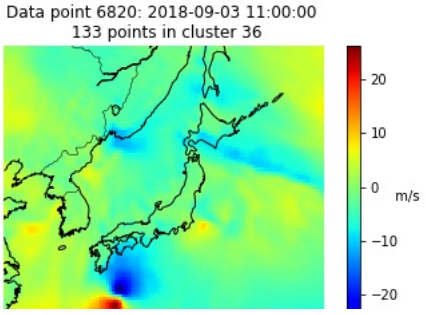


Figure 4-1: Typhoon Jebi, clustered without the use of an autoencoder

Two non-typhoons within the first 10 data points are shown in Figure 4-2. In Figure 4-3, two typhoon cases from within the first 10 data points are shown. It can be seen that clustering the data provides some useful information with regards to typhoons, but days are included that are not desired in this cluster. Presumably this is because the clustering algorithm dimensionality is too large - 100x140 images yield a dimensionality of 14000. The important features are likely not distinct enough for them to be clustered together entirely.

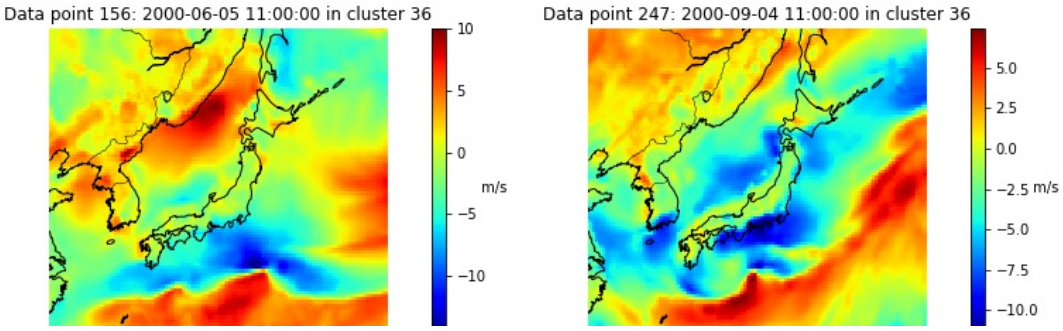


Figure 4-2: Two examples of clustered points that are not typhoons

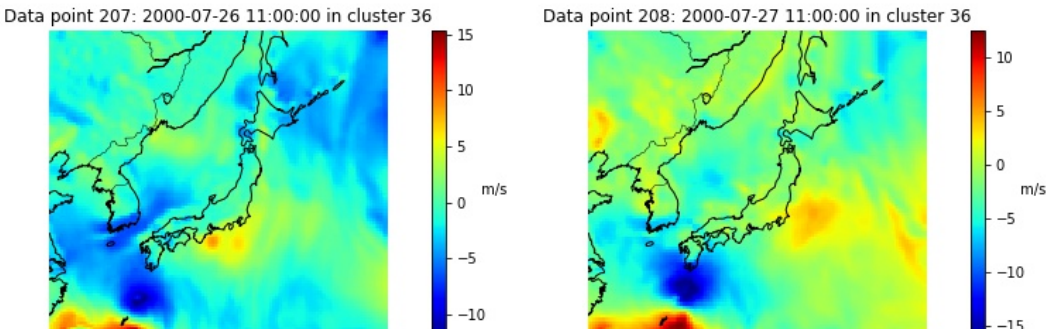


Figure 4-3: Two examples of clustered points that are typhoons

4-1-2 Single Layered Autoencoder

First, the amount of neurons in the middle layer are determined. The performance of the autoencoder using different amounts of neurons in the middle layer is shown in Appendix B-1. The autoencoder using 100 neurons in the middle layer is chosen. It can be seen that not all of the details are fully reconstructed, but the main wind features are contained in the reconstructed image, whilst still having a relatively small amount of neurons.

Training the autoencoder took approximately 2 seconds per epoch. The Mean Absolute Error (MAE) of the training and testing data for various amount of epochs is shown in the table below. An example of the performance of a validation sample, a data sample from x_test, can be seen in Figure 4-4.

Epochs	MAE training	MAE validation
100	0.7606	0.8187
250	0.7613	0.7965
500	0.7640	0.7919

The clustering algorithm converged significantly quicker. Now, the K-means algorithm ran in 4.82 seconds. When clustering the data, it can again be seen that within the first 10 clustered data points some typhoons, but also some non-typhoons are included in the same cluster as Typhoon Jebi. The cluster containing typhoon Jebi consists of 137 data samples.

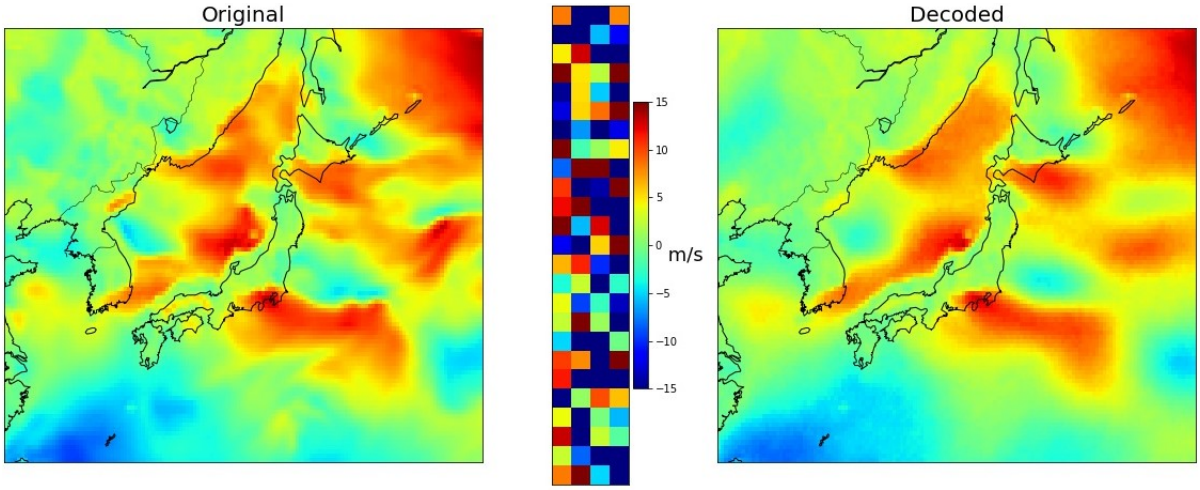


Figure 4-4: Example of a validation data sample using the single layered autoencoder

Two non-typhoons in the same cluster can be seen in Figure B-5 in Appendix B-2. Two typhoons clustered in the same cluster can be seen in Figure B-6 in Appendix B-2.

Now that the K-means algorithm converged in 4.82 seconds, it becomes feasible to check whether the appropriate amount of initial clusters is chosen. To check this, the inertia of the K-means clustering algorithm is calculated for every cluster size, ranging from 1 to 250. The resulting plot is shown in Figure 4-5. It can be seen that 50 clusters is a reasonable choice, as an increase in cluster size causes a linear decrease in the inertia.

To check whether we can achieve better results, we increase the complexity of the autoencoder.

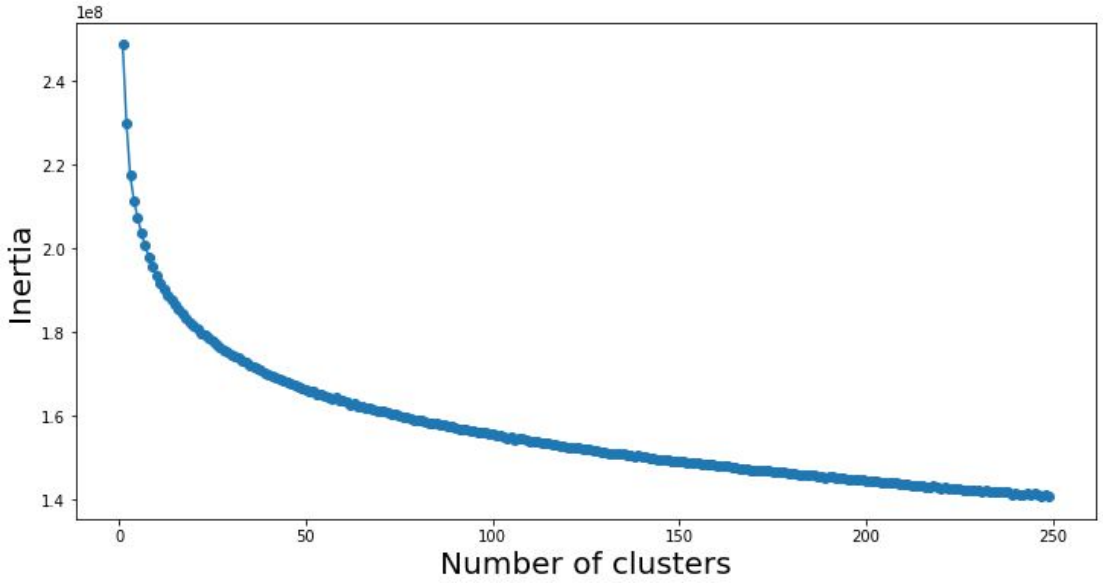


Figure 4-5: Inertia of K-means clustering for a varying number of clusters

4-1-3 Deep Autoencoder

The deep autoencoder trains slightly slower than the single layered autoencoder, taking approximately 7 seconds per epoch. An example of the performance of a validation sample can be seen in Figure B-7 in Appendix B-2. The MAE of the training and testing data is shown in the table below.

Epochs	MAE training	MAE validation
100	0.7920	1.0227
250	0.7083	1.0219
500	0.6644	1.0152

The cluster containing typhoon Jebi consists of 107 data samples. Two non-typhoons in the same cluster can be seen in Figure B-8. Two typhoons clustered in the same cluster can be seen in Figure B-9 in Appendix B-2.

4-1-4 CNN Autoencoder

As mentioned before, a Convolutional Neural Network (CNN) autoencoder is good at capturing spatial dependencies. The downside of a CNN autoencoder is that the computational costs are higher. Per epoch, the CNN autoencoder takes approximately 33 seconds, which is more than four times the computational cost of a deep autoencoder, and more than 15 times the computational cost of the single layered autoencoder. An example of the autoencoder performance is shown in Figure B-10 in Appendix B-2.

Epochs	MAE training	MAE validation
100	0.5672	0.7511
250	0.5255	0.7882
500	0.5003	0.8259

After clustering, the cluster containing typhoon Jebi contained 92 data points. Only 5 non-typhoons were contained in the first 25 data points contained in the cluster. This clustering performance is significantly better than for the previous methods.

Two non-typhoons and two typhoons in the Jebi cluster can be seen in Figures B-11 and B-12 in Appendix B-2.

4-1-5 Deep CNN Autoencoder

The deep CNN autoencoder is computationally the highest demanding model, although the difference with the regular CNN autoencoder is small. The deep CNN autoencoder takes approximately 35 seconds per epoch. An example of the performance of a validation data sample is shown in Figure B-13 in Appendix B-2.

Epochs	MAE training	MAE validation
100	0.5429	1.1464
250	0.4085	1.1906
500	0.3550	1.2162

The cluster containing typhoon Jebi consists of just 54 data samples. Besides having a higher computational cost, the clustering also performed worse than the CNN autoencoder. More non-typhoons were included in the cluster. Two non-typhoons in the cluster are shown in Figure B-14, two typhoons that are included are shown in Figure B-15 in Appendix B-2.

As said before, the computational cost of the CNN and the deep CNN is high. When adding max pooling layers to the CNN autoencoder, the computational cost will decrease. It will be researched to see if the performance of the clustering will not suffer from this.

4-1-6 CNN Autoencoder with Max Pooling Layer

To try to reduce the computational costs of an CNN, two max pooling layers are added in the CNN model. This decreases the computational cost severely, reducing the time per epoch to 22 seconds. An example of the performance of a validation sample can be seen in Figure B-16 in Appendix B.

Epochs	MAE training	MAE validation
100	0.7070	0.7296
250	0.6884	0.7145
500	0.6774	0.7050

Cluster 21, the cluster of typhoon Jebi, contains 117 data points. Two typhoons clustered can be seen in Figure B-18 in Appendix B-2. Two non-typhoons in the same cluster can be seen in Figure B-17.

What is striking about some of the non-typhoon data points included in the cluster, is that they contain some kind of typhoon characteristic. Using just the u10 data, a typhoon is recognized by a strong negative wind speed just above a location with strong positive wind speed. That a lot of these points are included gives reason to suspect that just using u10 data does not provide enough information for clustering wind patterns. A deep CNN autoencoder is still first analyzed to finish this comparison.

4-1-7 Deep CNN Autoencoder with Max Pooling Layer

The last autoencoder that is analyzed is the deep autoencoder with max pooling layers. The deep CNN autoencoder takes approximately 23 seconds per epoch. In Figure B-19 in Appendix B-2, an example of the performance of a validation sample can be seen.

Epochs	MAE training	MAE validation
100	0.7266	0.7828
250	0.6959	0.7408
500	0.6745	0.7223

132 data points are included in the cluster of typhoon Jebi. Two typhoons and two non-typhoons that are included in the same cluster as typhoon Jebi are shown in Figures B-21 and B-20 respectively.

4-1-8 Comparison

The Mean Absolute Error of the five models is shown in the table below. The lowest value of the MAE using the validation data is shown for every model, included with the amount of epochs it took for the model to reach that loss.

It can be seen that the MAE of the validation data of the CNN autoencoders and deep CNN with max pooling layers reached the lowest MAE. However, when their clustering performance was checked, the clustering did not perform that well. The best clustering results came from the CNN autoencoder.

Model	Epochs	MAE of training data	MAE of validation data	time per epoch
No autoencoder	-	-	-	-
Single Layered Autoencoder	500	0.7640	0.7919	2 sec
Deep Autoencoder	500	0.6644	1.0152	7 sec
CNN Autoencoder	100	0.5672	0.7511	33 sec
Deep CNN Autoencoder	100	0.5429	1.1464	35 sec
CNN Autoencoder max pooling	500	0.6774	0.7050	22 sec
Deep CNN Autoencoder max pooling	500	0.6745	0.7223	23 sec

The K-means clustering algorithm took approximately 5 seconds for every clustering method, except for the method without autoencoder. Without autoencoder, the K-means clustering took 6 minutes. This is a notable difference, showing that the clustering is harder to do without the use of an autoencoder.

For some models, the non-typhoons that are included in the clustering (as can be seen in Appendix B-2) do have similar features to a typhoon. Using just the zonal (Eastern-Western) wind component, this means a strong positive wind speed just below a location with strong negative wind speed. It seems that this is a result of lack of information. In the next section, the v10 component of the wind is added in an extra dimension.

4-2 U10 and V10 Classification

Adding v10 in the extra dimension gives slightly different autoencoders. They can be found in Appendix A-1. Their Python implementation can be found in D-6. The data adjustments that are necessary can be found in Appendix D-6-1.

4-2-1 No autoencoder

Because both u10 and v10 have been included, the amount of data points have been doubled. When the input data is flattened for direct clustering, it has the size (6940, 28000).

The K-means clustering took approximately 13 minutes. The details of the clustering of typhoon Jebi can be found in Figure 4-6. The first typhoon that was contained in the same cluster as Jebi was at the 23rd data point within the cluster. This shows that the clustering performed poorly. Two non-typhoons that are contained in the same cluster as Typhoon Jebi are shown in Figure B-23 in Appendix B-3. Two typhoons that are clustered are shown in Figure B-23.

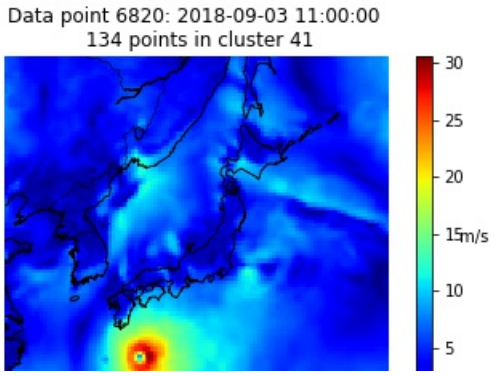


Figure 4-6: Typhoon Jebi, clustered without the use of an autoencoder

4-2-2 Single Layered Autoencoder

The training of the autoencoder took approximately 5 seconds per epoch. An example of a validation sample, encoded and decoded by the single layered autoencoder,using both u10 and v10, is shown in Figure B-24 in Appendix B-3.

Epochs	MAE training	MAE validation
100	0.9488	0.9774
250	0.9428	0.9802
500	0.9519	0.9803

The cluster of typhoon Jebi consists of 144 data samples. Two non-typhoon cases that are included in the same cluster as typhoon Jebi are shown in Appendix B-3 in Figure B-25. Two typhoon cases are shown in Figure B-26.

4-2-3 Deep Autoencoder

An example of a validation data sample can be seen in Figure B-27 in Appendix B-3. The validation data performs worse after a longer training duration. Training took approximately 12 seconds per epoch.

Epochs	MAE training	MAE validation
100	0.9248	1.1818
250	0.8395	1.1795
500	0.7938	1.1965

Very little of the 141 data samples included in the same cluster as typhoon Jebi are typhoons. This is undesired clustering behaviour. Two non-typhoons that were included in the cluster are shown in Figures B-28, two typhoons can be seen in Figure B-29 in Appendix B-3.

4-2-4 CNN Autoencoder

Training took approximately 36 seconds per epoch. A validation data sample can be seen in Figure B-30 in Appendix B-3.

Epochs	MAE training	MAE validation
100	0.7955	0.9297
250	0.7470	0.9715
500	0.7244	0.9988

The cluster of typhoon Jebi consists of 136 data samples. Two non-typhoons and two typhoons that are included in the same cluster as Typhoon Jebi are shown in Figures B-31 and B-32 in Appendix B-3.

There are still a couple non-typhoons included in the cluster. But as can be seen from the non-typhoons included in the cluster, shown in Figure B-31, these non-typhoons do contain a strong wind, accompanied with a low wind speed center. This shows that the clustering is performing as expected, but does not fully capture the typhoon behaviour yet.

4-2-5 Deep CNN Autoencoder

Training took approximately 37 seconds per epoch. A validation data sample can be seen in Figure B-33 in Appendix B-3.

Epochs	MAE training	MAE validation
100	0.7245	1.2522
250	0.5924	1.3137
500	0.5292	1.3466

Cluster 37, the cluster where typhoon Jebi was clustered in, contained 86 data samples. This cluster consisted of a lot of non-typhoons. This was surprising behaviour, as the previous CNN model performed quite well. Two non-typhoons and two typhoons that were included are shown in Figures B-34 and B-35 in Appendix B-3.

4-2-6 CNN Autoencoder with Max Pooling Layer

Training took approximately 24 seconds per epoch. A validation data sample can be seen in Figure B-36 in Appendix B-3.

Epochs	MAE training	MAE validation
100	0.9661	0.9846
250	0.9145	0.9430
500	0.8836	0.9133

163 data samples are contained in the same cluster as typhoon Jebi. This autoencoder produces the same behaviour as the CNN autoencoder. As can be seen in Figure B-37, some non-typhoons are included in the Jebi cluster, but these contain a strong wind combined with a low wind speed center, as was the case with the CNN autoencoder without max pooling layers.

It seems as if the model needs more data to fully capture the typhoon behaviour. When using more data, more typhoons will be occurring in the dataset. Then, the clustering might capture the specific wind patterns characteristic for typhoon behaviour better. Two typhoons included in the cluster are shown in Figure B-38.

4-2-7 Deep CNN Autoencoder with Max Pooling Layer

Training took approximately 25 seconds per epoch, and a validation data sample can be seen in Figure B-39 in Appendix B-3.

Epochs	MAE training	MAE validation
100	0.9873	1.1023
250	0.8819	1.0236
500	0.8257	1.0379

Clustering Typhoon Jebi yields a cluster containing 202 data samples. The deep CNN gives decent results regarding the classification. Some typhoons are contained in the cluster, as can be seen in Figure B-41. And the non-typhoons, as can be seen in B-40, contain the same behaviour as discussed in the CNN

4-2-8 Comparison

Model	Epochs	MAE of training data	MAE of validation data	time per epoch
No autoencoder	-	-	-	-
Single Layered Autoencoder	100	0.9488	0.9774	5 sec
Deep Autoencoder	250	0.8395	1.1795	12 sec
CNN Autoencoder	100	0.7955	0.9297	36 sec
Deep CNN Autoencoder	100	0.7245	1.2522	37 sec
CNN Autoencoder max pooling	500	0.8836	0.9133	24 sec
Deep CNN Autoencoder max pooling	250	0.8819	1.0236	25 sec

Looking at the MAE of the validation data and taking the clustering of the typhoons into account, the CNN without max pooling layers is the preferred autoenencoder. It reaches the second lowest MAE of the validation data, but it does so after just 100 epochs.

Three models had the characteristic that the non-typhoons included in the Jebi cluster had typhoon related wind patterns. This was the case for the CNN autoencoder, the CNN autoencoder with max pooling, and the deep CNN autoencoder with max pooling.

The preferred autoencoder for the use of the full dataset is the CNN autoencoder. This is first of all due to the fact that the data was clustered as expected. It was noted that a couple non-typhoons were included in the cluster, but this was the same for the CNN and deep CNN with max pooling layers. It is expected that when clustering more data, more typhoons will be included and the clustering will be improved. Furthermore, the CNN autoencoder needed just 100 epochs to reach its optimal MAE. This causes the total time to train the model to be the lowest of the three, even if the time per epoch was higher than when using max pooling layers.

4-3 Full Data

As explained in Chapter 3, the full dataset considers 4 data points per day, using 40 years of data. The size of the dataset is (58440, 100, 140, 2).

The approach is to use the CNN autoencoder that is already trained. The CNN autoencoder, previously trained for 100 epochs, had a MAE of 0.7955 for the training data, and a MAE of 0.9297 for the validation data. As the model is checked using validation data, it should perform the dimensionality reduction as expected. The entire dataset is implemented, reducing its dimensionality.

If this does not work as expected, a second approach is considered. The second approach consists of training new autoencoders using the entire dataset, using 85% for training and 15% for validating again. This is not the preferred method, but it should only be considered if the results of the first approach are not as desired. The computational costs will be significantly higher.

The model will remain exactly the same as it has been in the previous sections. The CNN autoencoder can be seen in Figure A-3 in Appendix A-1.

When using the entire dataset, the clustering of Typhoon Jebi is shown in Figure 4-7 below. Because the size of the dataset has increased significantly, it is harder to analyze every data point containing in the same cluster. The average wind speed of all the points contained in the cluster is calculated and plotted, shown in Figure 4-8. The Python code for this calculation is shown in D-7 in Appendix D.

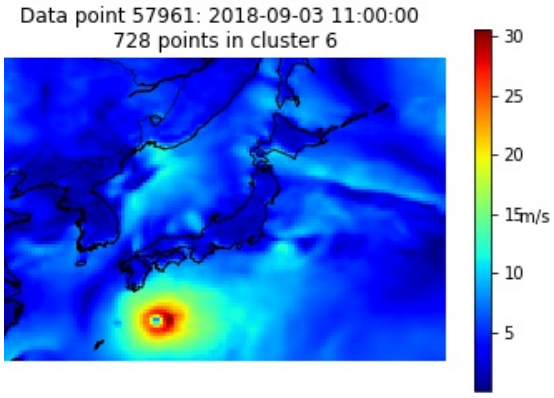


Figure 4-7: Typhoon Jebi, clustered with the CNN autoencoder using all the data

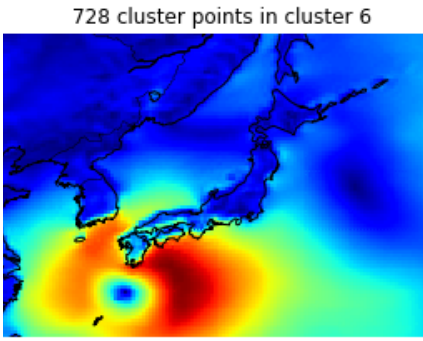


Figure 4-8: The average wind cluster 6

Figure 4-8 seems to show that typhoon behaviour is fully captured in the cluster. The disadvantage of looking at the average wind speed using this many data samples, is that non-typhoons in the cluster are compensated for by the presence of many typhoons. When analyzing the data samples contained in the cluster, it can be seen that the same problem as when using small amount of data still occurs. More than half of the data samples are non-typhoons. The non-typhoons do show similar behaviour to typhoons, as shown in Figure 4-9 below. It can be seen that there is a low wind speed center, with high wind speeds near.

This shows that the average wind speed is not an entirely reliable method for checking the amount of correctly clustered data points. However, it does provide a general un-

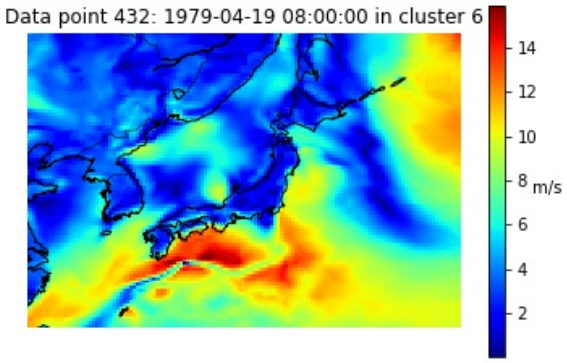


Figure 4-9: Example of a non-typhoon included in the same cluster as Typhoon Jebi

derstanding in what kind of data samples are included in a particular cluster. The average wind speed of all the clusters can be seen in Appendix B-4. There are other clusters where the average wind speed shows typhoon behaviour. As discussed, it is necessary to analyze these clusters independently to check the reliability of the cluster. An example of another cluster with typhoon behaviour is cluster 25, shown in Figure 4-10.

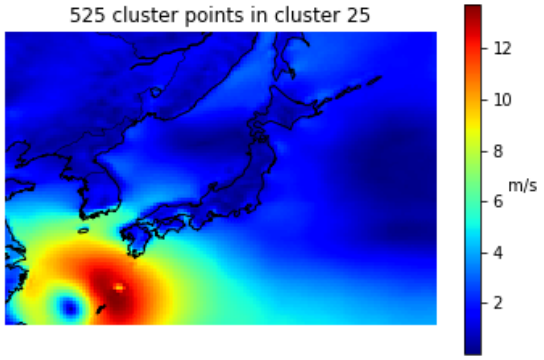


Figure 4-10: The average wind speed of cluster 25

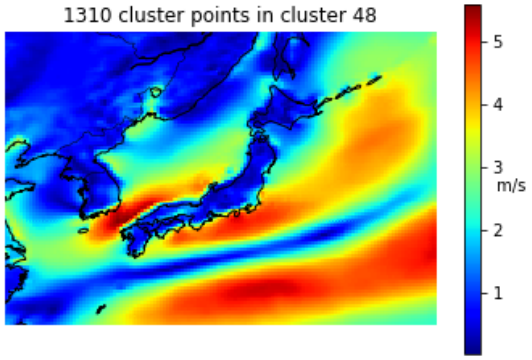


Figure 4-11: The average wind speed of cluster 48

Analyzing cluster 25 yields better results than cluster 6. Cluster 25 contains 525 data points, of which approximately 43 are non-typhoons. This was analyzed by eye. This means that less than 10% of the cluster were non-typhoons, which means that more than 90% of the data samples in cluster 25 are typhoons.

This shows that the clustering performs as expected. Besides typhoon clustering, another interesting cluster is found. This is cluster 48, shown in Figure 4-11. This seems a very particular wind pattern, and it is interesting to further analyze this cluster.

Interestingly enough, this cluster contains many data samples having a wind pattern similar to the average wind shown in Figure 4-11. Two examples are shown in Figure 4-12 below. Some typhoons are also contained in cluster, which is not expected. However, again this shows that the clustering performs as expected, but is not perfect. The average wind speed is again giving a general understanding about the wind patterns contained in a cluster.

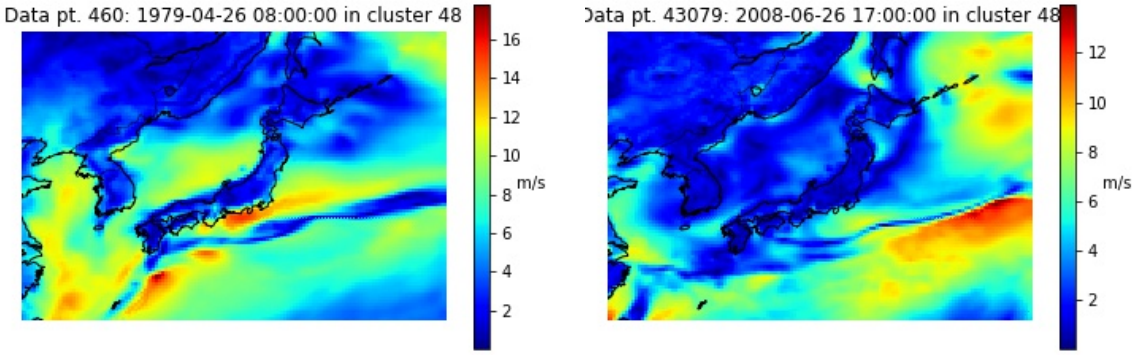


Figure 4-12: Two examples of wind patterns contained in cluster 48

The training of the CNN model using all the data was computationally heavy. It took approximately 360 seconds per epochs to train this model. After 100 epochs, the loss was 0.7791 and 0.8210 for the training and testing data respectively. This is better than the CNN autoencoder achieved with the small training dataset, which was expected due to the extra amount of training data. Looking at the clustering of Typhoon Jebi still resulted in a cluster containing some non-typhoons. The result was not significantly better, and for clarity of this thesis this model will not be further considered.

Even after training the autoencoder using 85% of the entire dataset, or 49674 samples, the clustering still contained non-typhoons. The non-typhoons do show wind characteristics that are similar to a typhoon. A possible way to distinguish between a non-typhoon and a typhoon is the development of both during the day. It is expected that a typhoon will not change significantly in a day, whilst a non-typhoon might.

4-4 3D CNN autoencoder

As explained in Chapter 3, the 3D CNN groups together 4 data points per day to analyze the development of wind patterns during the day. The size of the dataset is (14610, 100, 140, 4, 2).

Training the 3D CNN autoencoder is computationally extremely heavy. It takes approximately 1100 seconds per epoch and was trained for 60 epochs. After 50 epochs, the loss of the autoencoder did not decrease significantly, which is why the learning process was stopped to prevent the model from overfitting. The autoencoder reached a MAE of 0.8262 for the training data and 0.9635 for the validation data. An example of the performance of the autoencoder is shown in Figure B-59 in Appendix B-5.

The autoencoder will be analyzed using the same methods as previously used in this research. Typhoon Jebi is shown in Figure 4-13, and the average wind speed of this cluster is shown in Figure 4-14. The average wind speed is shown using the 8 AM data point. The rest of the time steps are shown in Figures B-60 to B-65 in Appendix B-5-2.

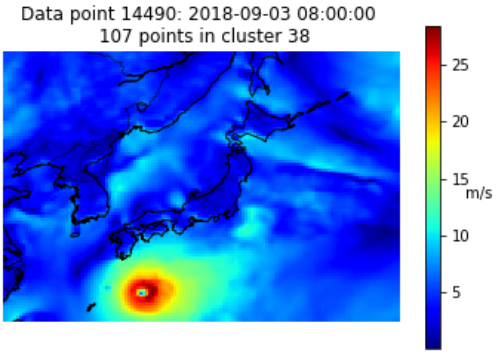


Figure 4-13: Typhoon Jebi, clustered with a 3D CNN autoencoder using all the data. First times step.

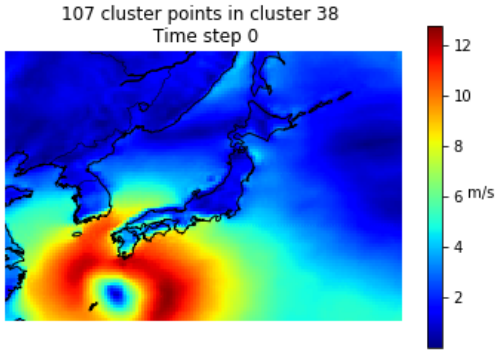


Figure 4-14: The average wind speed of cluster 38 using the first time step

It can be seen from Figure 4-14 that the average wind speed is again showing typhoon behaviour. However, as we have seen with other clusters, the cluster should be manually analyzed on the amount of typhoons. There are 107 typhoons included in cluster 38. It is checked using the database of the Japan Meteorological Agency (JMA) [29] whether these dates match actual typhoons. The dates of the included typhoons are shown in Appendix B-5-3. In the column next to the dates, it is shown whether or not a typhoon is classified by the JMA as a typhoons. Out of 107 typhoons, only 5 non-typhoons were clustered. This is an accuracy of 95.3%.

Considering that this is the cluster containing Jebi, this is a huge improvement. With previous methods, the cluster containing typhoon Jebi contained a lot of non-typhoons. Using this 3D CNN approach improved the clustering significantly for this cluster.

All the cluster average wind speeds are shown in Appendix B-6. For clarity, only the average wind speeds of the data at 8 AM are included. The clusters show similar behaviour with the regular 2D CNN autoencoder. It is however expected that addition of the extra time dimension makes the clusters more robust and containing less wind patterns not representative

for the cluster. Obviously, the entire dataset is clustered and some clusters will not contain extreme wind patterns. Two clusters with distinct wind patterns are shown in Figures 4-15 and 4-16 below.

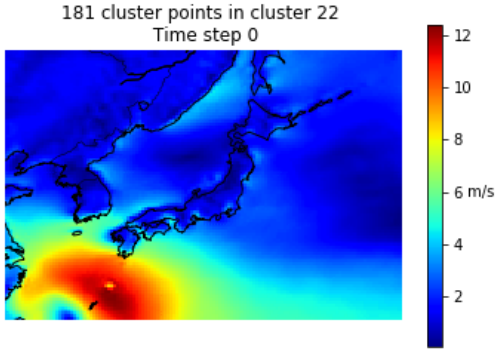


Figure 4-15: The average wind speed of cluster 22

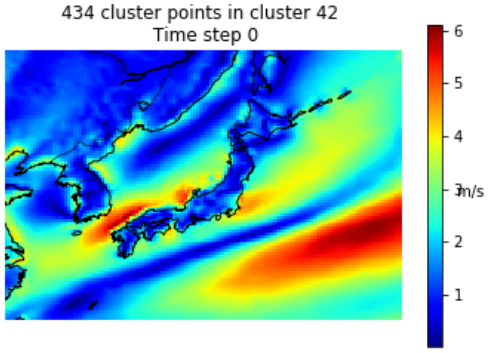


Figure 4-16: The average wind speed of cluster 42

These clusters show the same behaviour as discussed in Figure 4-10 and 4-11. Analyzing their seperate data samples by eye, it can be seen that these clusters performed similar or better when using the 3D CNN autoencoder as opposed to the 2D CNN method.

Now that the most robust dimensionality reduction and clustering combination has been found, it is analyzed whether this clustering information can provide useful information regarding the next day. This is done using a transitional matrix.

4-5 Transitional Matrix

To investigate the chances of a particular cluster happening, a transitional matrix is made. The correlation of a next day cluster is analyzed using the 3D CNN autoencoder. If a particular day is cluster X, what clusters are likely to happen the next day? To do this, compare t with t+1. The resulting transitional matrices, one showing the quantities and one showing the percentages of this occurance, are shown in Figures 4-17 and 4-18 on the following pages.

It can be seen in the transitional matrix that some clusters are very likely to repeat itself. These values are represented by the quantities and percentages on the diagonal of the transitional matrix. Cluster 44 has the highest chance of repeating itself. Given that a particular day is clustered into cluster 44, the chance of the next day being cluster 44 as well is 51%. The next two most likely wind patterns to repeat itself are cluster 27 and cluster 14, with a probability of 43% and 39% respectively. Looking at clusters 14, 27, and 44 in Appendix B-6, these clusters all have one thing in common, the average wind speed is generally low. This can be the cause of their high chance of repetition. If clusters have particular extreme weather patterns, it can be expected that the conditions change significantly, whilst with non-extreme wind patterns, change might occur slowly.

What is perhaps more interesting, is that some clusters have a 0% chance of repeating itself. From Figure 4-17 it can be seen that clusters 30 and 31 have never been repeated themselves. Some other clusters that have a low repetition is cluster 41 with just one repetition, cluster 12 with 2 repetitions, and cluster 46 with 3. These clusters have a significantly higher average wind speed than the clusters with a high chance of repeating itself. It can be seen in Appendix B-6 that clusters 12, 30, 31, and 46 all have a average wind speed of approximately 12 m/s at the highest intensity point. The clusters with a high chance of repeating itself, clusters 14, 27, and 44, had a wind speed of approximately 7 m/s at their highest intensity, which is significantly lower.

So it should be investigated if wind patterns with a high average wind speed have a low chance of repetition, and if wind patterns with low wind speed have a high chance of repetition. To validate this assumption, the clusters with the highest and lowest average wind speeds are taken and tested. From Appendix B-6 it is found that cluster 18 has the highest average wind speed intensity of 15 m/s. Cluster 42 has the lowest average wind speed, with the highest wind speed intensity measuring 6 m/s.

Cluster 42, with a low average wind speed, has a high chance of repetition. The chance that cluster 42 repeats after itself is 22%. Cluster 18, with high average wind speed, has a chance of 13% of repeating itself. This is not a significant difference and hence the statement cannot be concluded. However, since there is a difference, the statement cannot be rejected either.

An interesting point in Figure 4-18 is the correlation between cluster 18 and 8. Given that a particular day was cluster 18, there is a 32% chance that cluster 8 will happen the day afterwards. Both these clusters have a high average wind speed, which seems to be the main correlation between the two clusters.

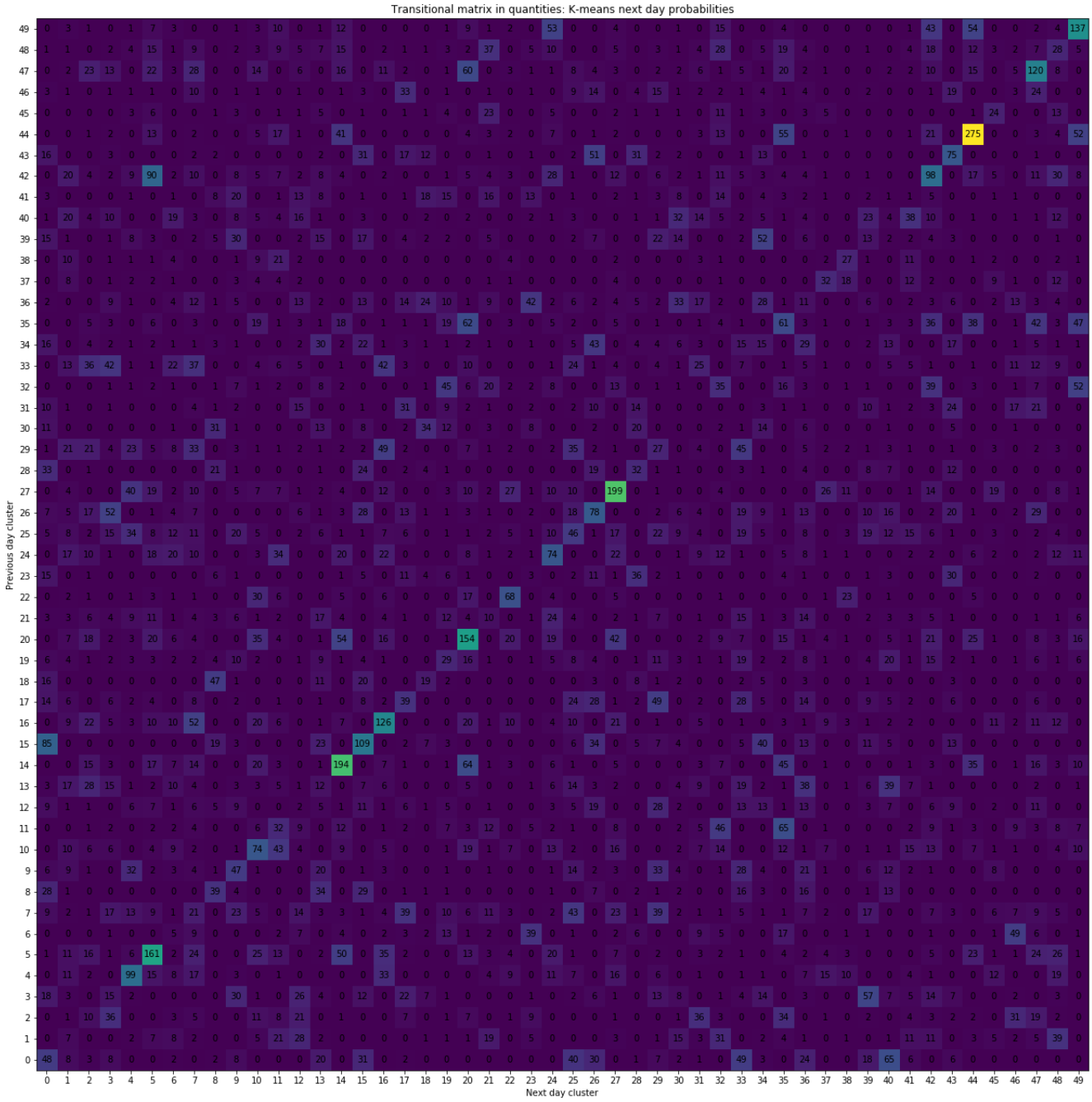


Figure 4-17: Transitional matrix comparing the next day clusters measured in quantities

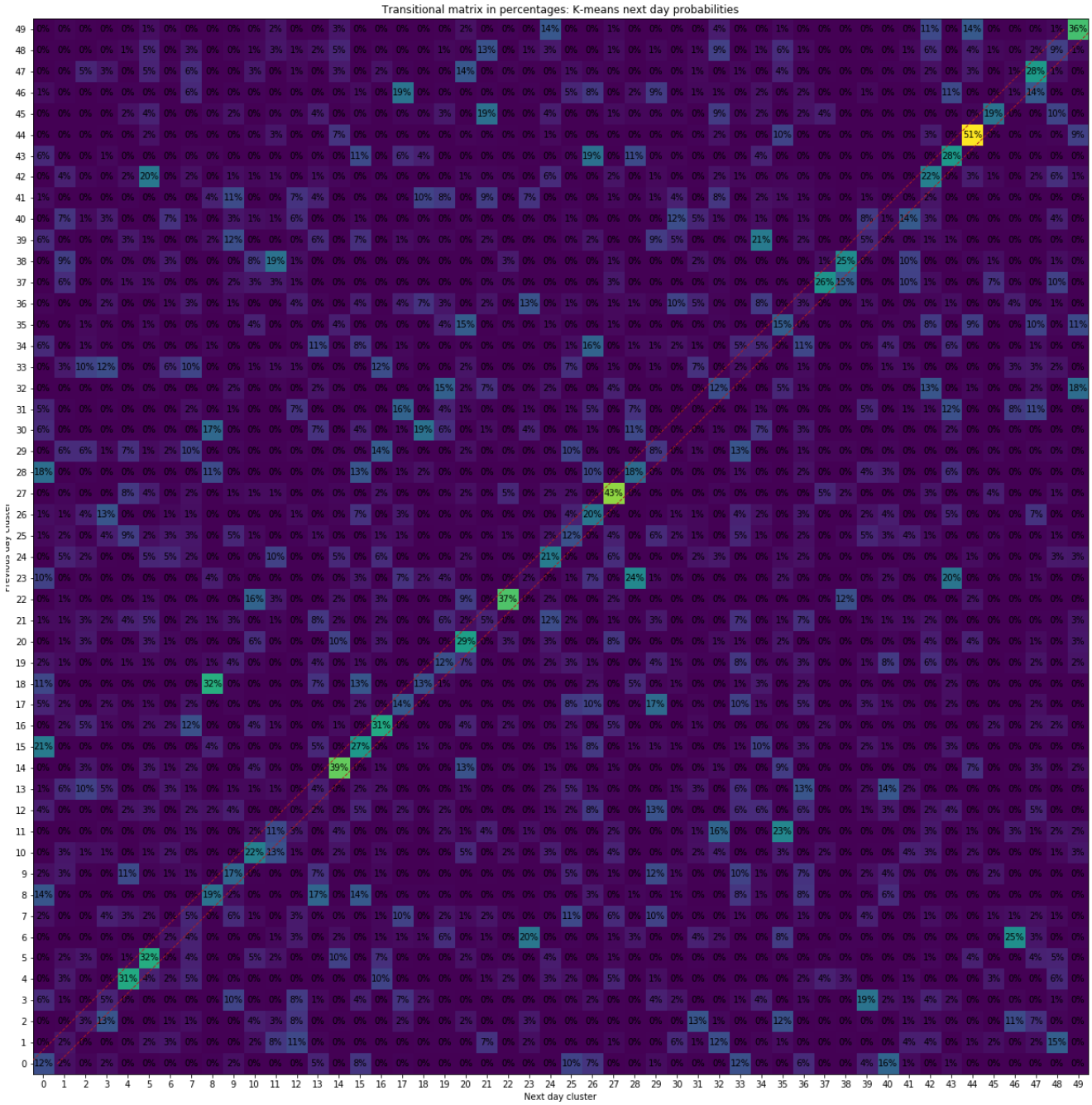


Figure 4-18: Transitional matrix comparing the next day clusters measured in percentages

Chapter 5

Case Study

When using unsupervised wind pattern classification over Japan, interesting wind patterns emerged. As expected, typhoon behaviour was a prominently present wind pattern. However, other wind patterns emerged from the wind pattern classification as well.

The Dutch National Sailing Team has a particular interest in small-scale wind patterns. To investigate whether unsupervised learning can be of any use in identifying these patterns, a fine resolution method is applied. The first round of the World Cup was held in September 2019 in Enoshima, the same bay as where the Olympic Sailing Competition will be located. After the World Cup, a fine resolution WRF model was computed to perform reanalysis.

The dataset now consists of a data sample every 10 minutes. The first data point is 15-07-2019 at 00:00:00, the last data point is 19-09-2019 at 17:20:00. Every grid point is approximately 1 km by 1 km. The domain is shown in Figure 5-1. The area of the domain is defined by longitude and latitude:

Longitude: 138.68 - 140.65 °E
Latitude: 34.57 - 36.16 °N

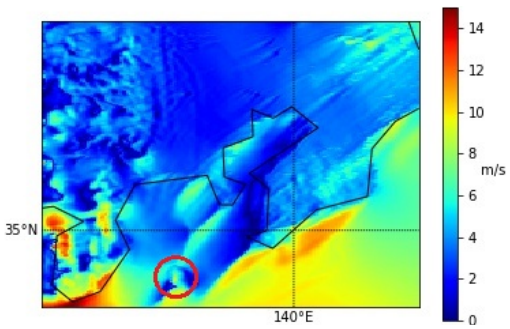


Figure 5-1: The fine resolution domain used for classifying wind patterns

This results in an image of 176x176 pixels. The two best performing autoencoders are used to reduce the dimensionality of the dataset. First, a 2D CNN autoencoder is applied. Afterwards, a 3D CNN approach is taken.

Using the 2D CNN approach, a single data point is analyzed and clustered, exactly as has been done in Chapter 4. However, the 3D CNN approach is slightly adjusted. In Chapter 4, the 3D CNN approach considered an entire day, consisting of 4 data points per day. Now, the data set consists of 66 days, containing 144 data points for each day. If a data sample for the 3D CNN would remain an entire day, just 66 data samples containing 288 images, 144

images of both U10 and V10, would be used as training data for the autoencoder. This ratio is off. Therefore, it is chosen that the 3D CNN autoencoder uses one hour of data as input. Six data points are contained in a data sample of the 3D CNN autoencoder.

Both the 2D CNN and the 3D CNN autoencoder are again compressed into 100 neurons. Again, just as happened in Chapter 4, the training data contained of 85% of the dataset. The validation data is the remaining 15%. The model is trained for 100 epochs, as was deemed the best amount of epochs to train the 2D CNN and 3D CNN autoencoders, shown in Chapter 4.

The structures of the 2D CNN and the 3D CNN autoencoder are shown in Figures A-7 and A-8 in Appendix A. After reducing the dimensionality using the autoencoders, the data is clustered using the k-means clustering algorithm. The full Python implementation for the 2D CNN and the 3D CNN autoencoder can be found in Appendix D-9-1 and D-9-2 respectively.

5-1 2D CNN Autoencoder

The dataset has a shape of (9609, 174, 174, 2), having 9609 data points of size (174, 174), containing both the U10 and V10 wind speed. The autoencoder was initially trained for 100 epochs. However, the Mean Absolute Error (MAE) of the validation data did not decrease after 25 epochs, so the training process was stopped to prevent the model from overfitting. The training took approximately 120 seconds per epoch, resulting in a MAE of 0.3629 for the training data and 1.0660 for the validation data.

Because the dataset is entirely different, a new cluster amount is determined. The k-means clustering inertia is shown in Figure 5-2. The new amount of clusters is set at 10.

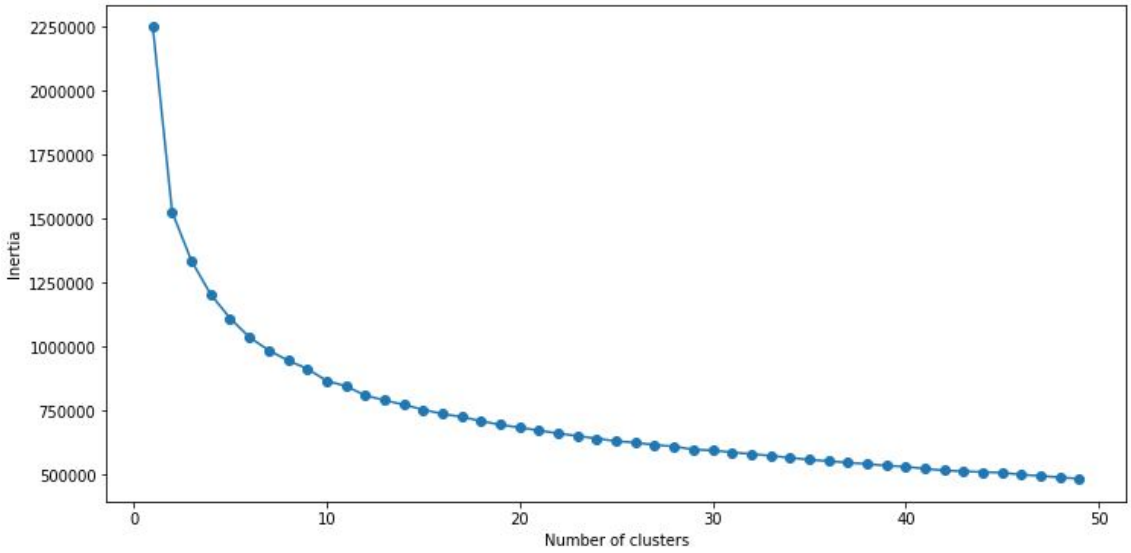


Figure 5-2: Inertia of the K-means clustering for a varying number of clusters

As shown in Chapter 4, the average wind speed of the clusters is an appropriate measure to indicate what wind patterns are included in the cluster. The average wind speed of the 10 clusters are shown in Figure 5-3. The enlarged versions are shown in Appendix C-1, where the arrows in the figures denote the wind direction.

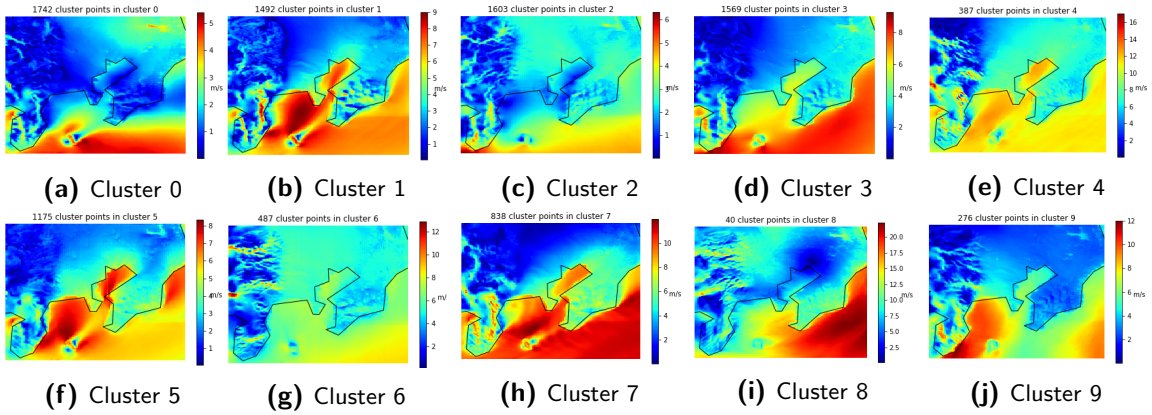


Figure 5-3: Average wind speeds of the clusters using the 2D CNN method

It can be seen that the important wind characteristics are clustered together. In clusters 0, 1, 5, and 7, the wind direction in the bay area is approximately the same: south west. However, the effect of the Oshima island is different for all the clusters. The Oshima island is unfortunately not denoted in the border lines drawn in the average wind speed Figures, but it is located in the red circle in Figure 5-1.

The effect of the Oshima island can be seen as a decrease in wind speed located downwind of the island. In clusters 1 and 5, the effect of the Oshima island is clearly visible, with cluster 1 having an even more pronounced effect than cluster 5. Cluster 7 has a very small effect on the wind speed, which only exists very close to the island.

Cluster 9 also shows a wind characteristic caused by Oshima island. However, in cluster 9 the wind direction is different than the previously discussed clusters. This causes the effect of the Oshima island to have a different direction as well.

As mentioned in Chapter 4, the individual data points in a cluster should be analyzed to show if the individual data points indeed show wind characteristics similar to the average wind speed of the cluster. The cluster with the most pronounced effects of Oshima island, cluster 1, is analyzed.

Four interesting examples are shown in Figure 5-4. What makes them interesting, is that they all have different wind speeds, slightly different wind directions, and the effect of Oshima island has a different shape. However, all of them suffer from this effect caused by Oshima island. The classification recognizes that this is the effect caused by the Oshima island is the most important feature of these data points, clustering them together in cluster 1.

Another cluster that draws attention is cluster 8. This cluster contains just 40 data points. This cluster contains very strong wind behaviour, which occurred between 08-09-2019 at 19:00:00 and 09-09-2019 at 01:30:00.

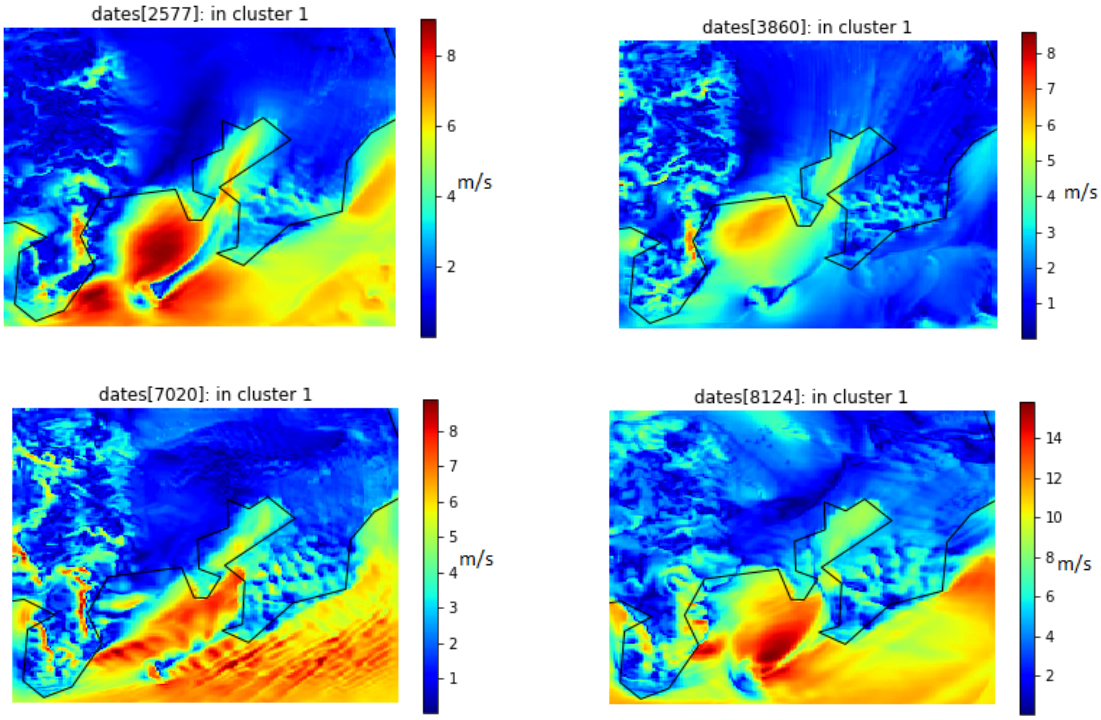


Figure 5-4: Four examples of images contained in cluster 1 using the 2D CNN method

The different clusters show different characteristics. Hence the clustering focuses on both wind direction and wind speed. This is exactly how the clustering is desired to perform, the most important wind characteristics are clustered together, regardless of what the most representing characteristic of the wind is. This creates a better understanding in wind behaviour, which can lead to better sailing strategies.

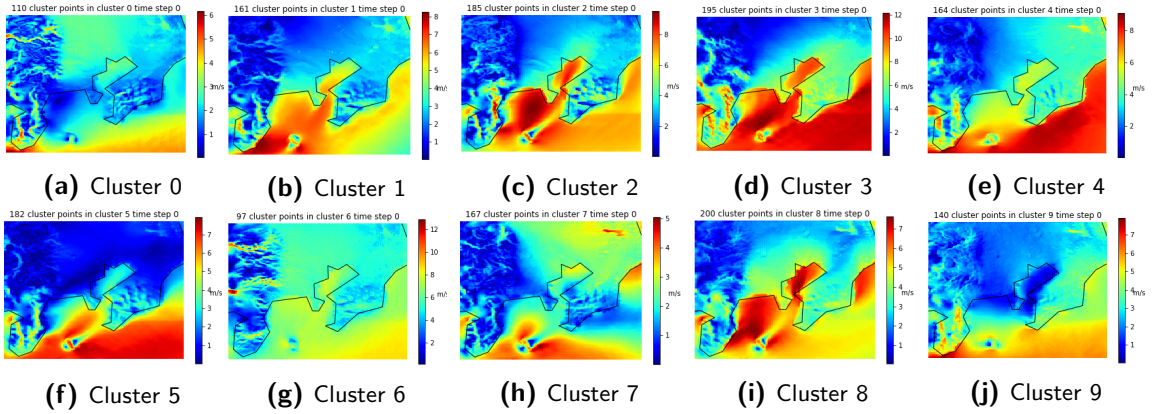
However, the time component is not taken into account in this clustering method. Wind shifts, oscillations, and other time dependent wind patterns are therefore neglected. To take these into account, a 3D CNN autoencoder is used.

5-2 3D CNN Autoencoder

The dataset is now analyzed per hour, with one data sample containing six data points. The dataset has size (1601, 174, 174, 6, 2). So it analyzes 1601 data samples, containing 174 x 174 pixels, 6 data points per sample, using both the U10 and V10 wind speed information.

Training the autoencoder took approximately 470 seconds per epoch. This is an increase in computational cost by almost a factor of four comparing to the 2D CNN method. The resulting MAE of the training data and validation data are 0.4467 and 1.0576 respectively.

The average wind speed of the first time step of every cluster is shown in Figure 5-5. The enlarged versions are shown in Appendix C-2. The clusters show similar wind characteristics as when using the 2D CNN autoencoder. However, the high wind speed event, as was included in cluster 8 using the 2D CNN autoencoder, is not included in this approach. This can be seen as a shortcoming, as this event was a rare occurrence, which is desired to be clustered in a separate cluster.



(a) Cluster 0 **(b)** Cluster 1 **(c)** Cluster 2 **(d)** Cluster 3 **(e)** Cluster 4

(f) Cluster 5 **(g)** Cluster 6 **(h)** Cluster 7 **(i)** Cluster 8 **(j)** Cluster 9

Figure 5-5: Average wind speeds of the clusters using the 2D CNN method

The rest of the clusters contain similar information as when using the 2D CNN clustering method. Cluster 1 of the 2D CNN method is similar to cluster 2 of the 3D CNN method. Cluster 3 of the 2D CNN method is cluster 4 of the 3D CNN method, and cluster 6 of both methods contain similar behaviour.

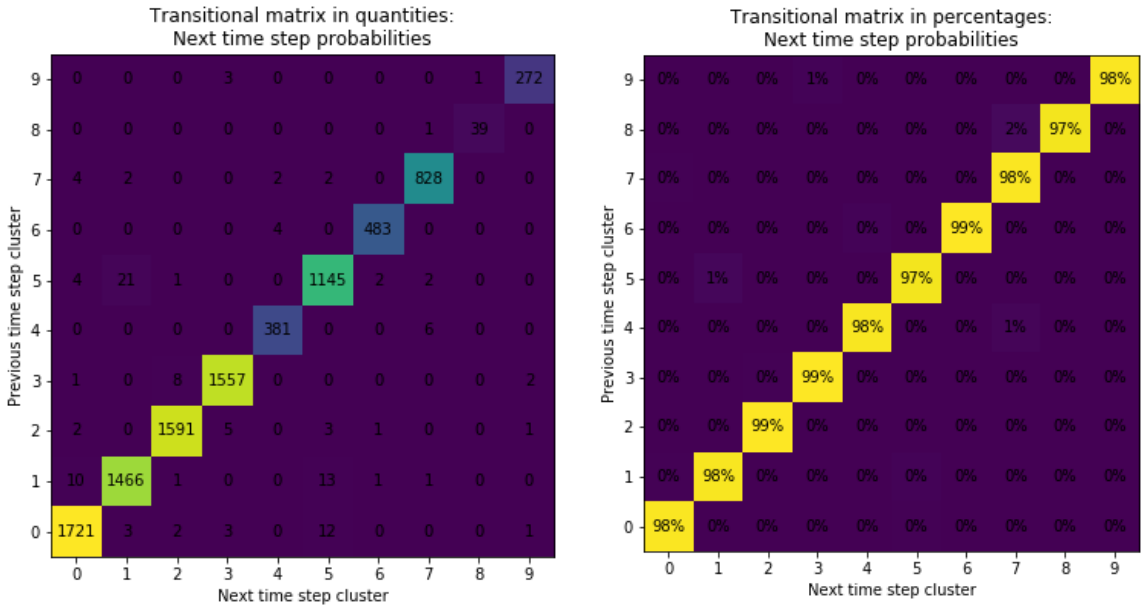
To see how the methods compare in the development of clusters, the transitional matrices are considered.

5-3 Transitional Matrix

First, the 2D CNN method is taken into account. The transitional matrices in quantities and percentages are shown in Figure 5-6. As can be seen, the data is really correlated. The next time step cluster has at least a 97% chance of being in the same cluster, as can be seen in Figure 5-6b.

The transitional matrices of the 3D CNN method in quantities and percentages are shown in Figure 5-7. The 3D CNN method also shows a great amount of correlation. The next hour has at least a 85% chance of being in the same cluster as the previous hour, as can be seen in Figure 5-7b.

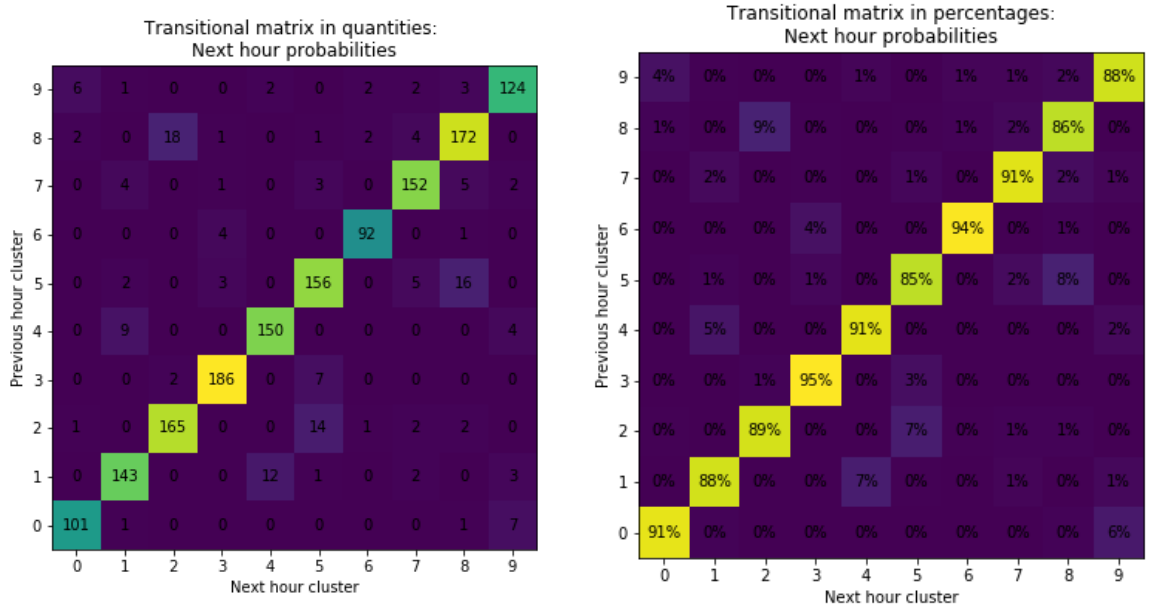
The data is considered to be too correlated to make any conclusions based on the transitional matrix. However, it might be useful to look into the development of the cluster and neglect the times the cluster does not change. So it is investigated what a particular cluster changes into. This is shown in the same matrix format and hence deemed a transitional change matrix.



(a) In quantities

(b) In percentages

Figure 5-6: Transitional matrices for the 2D CNN method



(a) In quantities

(b) In percentages

Figure 5-7: Transitional matrices for the 3D CNN method

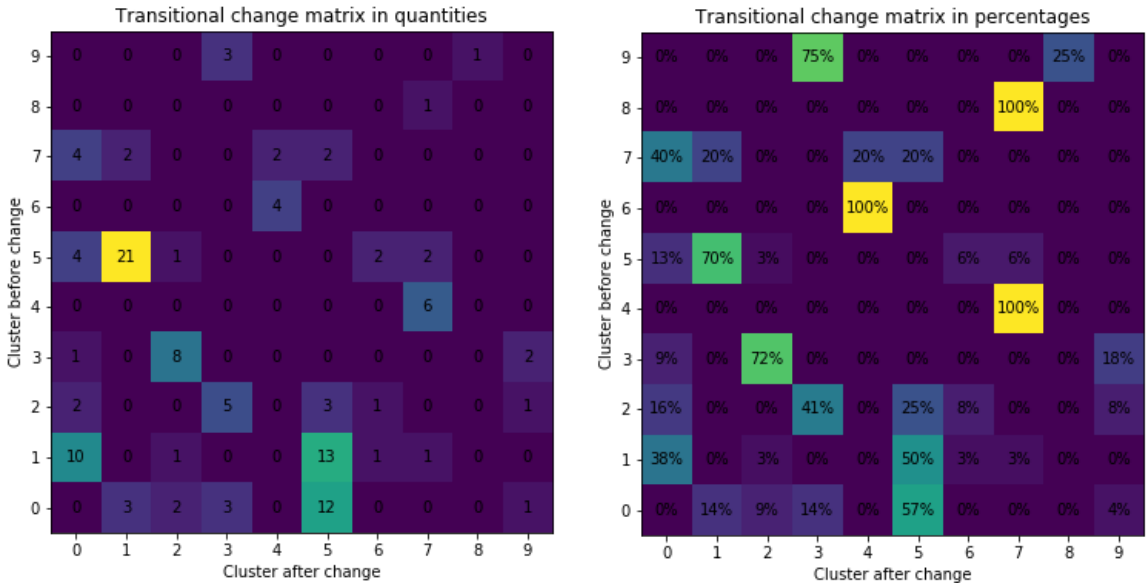
5-4 Transitional Change Matrix

The transitional change matrices of the 2D CNN method, both in quantities and in percentages, are shown in Figure 5-8. Striking is that clusters 4, 6, and 8 always change in clusters 7, 4, and 7 respectively. It should be noted however, that the quantities are of great importance in the change matrix. Cluster 8 was the storm cluster, and it only changes once. It is evident that this causes the change matrix to show 100%, but this is not representative. That cluster 4 always changes in cluster 7 is more notable, as cluster 4 changes six times. Cluster 5 has a 70% chance of changing into cluster 1, but this change occurs 21 times, which is a significant amount.

On the other hand, cluster 1, 2, and 7 change in all kinds of clusters. It is hard to conclude why this is the case, but this is also useful information. This means that when a particular time step is in one of these clustered, it is hard to predict future wind behaviour.

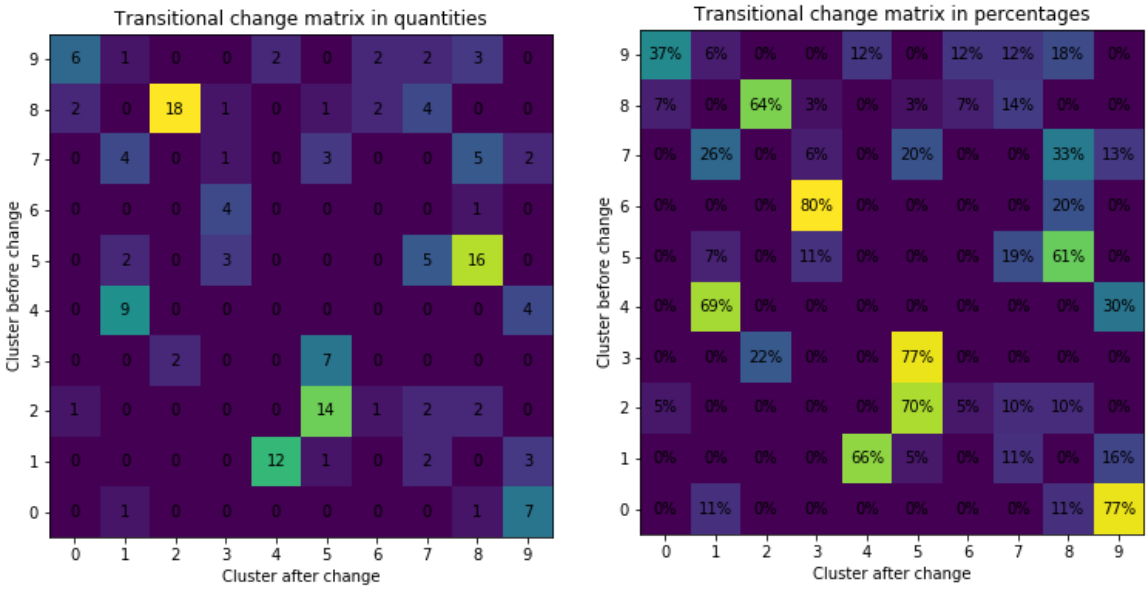
The transitional change matrices of the 3D CNN method are shown in Figure 5-9. None of the values in the percentage matrix in Figure 5-9b is 100%, compared to three 100% values when using the 2D CNN method. However, as stated before, percentages can be unrepresentative.

The amount of data points that contribute to the change matrix are sparse. Some percentages look promising, but more fine resolution data is needed to further investigate the development of the clusters. In this case, the 2D CNN method shows more distinct percentage values and might therefore be more appropriate for this case study, but no definite conclusion can be made based on the transitional matrices.



(a) In quantities **(b)** In percentages

Figure 5-8: Transitional change matrices for the 2D CNN method



(a) In quantities **(b)** In percentages

Figure 5-9: Transitional change matrices for the 3D CNN method

5-5 Logbook Sailors

During the period of this case study, the Dutch National Olympic Sailing team has been logging specific wind patterns. Four occasions of gradient wind have been logged, as well as five occasions of sea breeze and one storm case. The resulting clusters using both the 2D CNN and the 3D CNN method are shown in the table below.

Type of wind	Date	Cluster 2D CNN	Cluster 3D CNN
Gradient	15/07/2019	3	4
Gradient	17/08/2019	1	2
Gradient	19/08/2019	2	4
Gradient	04/09/2019	3	4
Sea Breeze	17/07/2019	0	7
Sea Breeze	28/07/2019	1	2
Sea Breeze	30/07/2019	1	2
Sea Breeze	31/07/2019	1	2
Sea Breeze	18/08/2019	0	7
Storm	27/07/2019	4	6

Both methods are suggesting two types of sea breeze. The 17th of July and 18th of August are clustered in a different cluster than the other three sea breeze days. Furthermore, both methods suggest the gradient wind occuring at the 17th of August is related to the sea breeze characteristics. It is also notable that both methods cluster the storm in a separate cluster, which was to be expected.

Unfortunately, the logging data is sparse as well. Therefore no conclusion can be made whether the 2D CNN or the 3D CNN method performed favourable. The transitional change matrices seemed to show that the 2D CNN was performing better, but due to data sparsity this cannot be seen as significantly better performance. The high wind speed event in cluster 8 of the 2D CNN method was not clustered separately by the 3D CNN method, which is slightly unfavourable. Furthermore, the computational costs were significantly lower for the 2D CNN method. The 2D CNN method seems slightly favourable for this case study.

However, as both methods confirm each other, it can be seen that both clustering methods are somewhat correct. The Dutch Olympic Sailing Team can further investigate why the 17th of July and the 18th of August are clustered differently. Maybe the days that are clustered differently are not sea breeze, but a similar wind pattern with different characteristics. This can lead to enhanced wind pattern understanding, and hopefully lead to better sailing strategies.

Chapter 6

Discussion

In this research, a lot of different parameters can be tuned. It should be noted that it is tried to justify all of the designer choices that are being made, but it is possible that some parameters have been overlooked.

The amount of clusters is justified, however, it is possible that the amount of clusters used in this research is not optimal. Therefore, research towards an optimal amount of clusters is recommended. The amount of clusters can have a significant impact on the clustering results.

Furthermore, K-means clustering is applied without comparison to other clustering methods. The author of this research thesis did compar several techniques. Hierarchical clustering was considered, but for the full dataset this caused a MemoryError. This is because the hierarchical clustering method computes a pairwise distance matrix, which consumes $O(n^2)$ of RAM [30].

Another clustering algorithm that was considered was the Gaussian Mixture algorithm. This algorithm assumes a Gaussian distribution. It was believed that weather does not behave in a Gaussian distribution, hence this clustering algorithm was also not used. K-means MiniBatch is shortly considered, but since this is a less accurate version of K-means [31] [32], which is developed for the use of large datasets, this algorithm is redundant. Other clustering methods, such as DBSCAN, OPTICS, or HDBSCAN were not considered.

Using only the zonal wind or using both the zonal and meridional wind yielded different Mean Absolute Error (MAE) values. The lowest MAE of the validation data using U10 was approximately 0.7, the lowest MAE of the validation data when using U10 and V10 was approximately 0.9. When doubling the amount of data used, it can be expected that the MAE would double as well. However, this showed not to be the case. Presumably the increase in the MAE is lower than expected because the increase in data results in a better training of the autoencoder, as the zonal and meridional wind are somewhat correlated.

The use of 100 neuron values in the middle was chosen in the beginning of this research. The performance of the single layered autoencoder was considered when using this neuron value. After this initial choice, the amount of neurons was not further considered. This might be one of the largest shortcomings of this research. The performance of the autoencoder and the

performance of the clustering are both directly dependent on the amount of neurons in the middle layer.

Related to this, is the arbitrary choices of activation functions used. As can be seen in Appendix D-5-9 and D-6, the autoencoder uses a combination of linear, Rectified Linear Unit (ReLU), and sigmoid functions. This is done because the data was not normalized, hence the sigmoid function on its own would not have provided sufficient results. The data was not normalized because this would not result in desired clustering. If the data was normalized, only the relative strong wind in one data sample would be considered. An extreme example: if one data sample has a 5 m/s wind speed at a particular spot, and a 1 m/s wind speed everywhere else, this normalized data sample would only consider the ratio between those wind speeds. When having a data sample with a 50 m/s wind speed at a particular spot, and a 10 m/s wind speed everywhere, the autoencoder would not be able to tell the two data samples apart. This is undesired, as the 50 m/s data sample should also be clustered based on this strong wind speed property.

It is expected that using ReLU will yield good results. But as this research progressed it was chosen that these initial activation functions are maintained as is. Adjusting the activation functions can possibly yield better results, or at least lead to faster computation.

The clusters are checked visually. It is advised to investigate whether there are other methods to check the clustering results. The logbook data of the sailors could provide some relevant information, but in this case the data was too sparse to be conclusive.

It can be seen that in the case study, the 2D CNN autoencoder performed similar to the 3D CNN method. This shows that the 3D CNN autoencoder is not always the best method for unsupervised classification. It should be analyzed for every case which method performs best.

Finally, the 3D CNN autoencoder was not able to capture time-dependent wind behaviour, such as oscillation or wind shifts. This is expected to be because of the use of 6 data points per data sample, which causes the input data to consist of 1 hour. Wind shifts and oscillations can take much longer. Furthermore, to classify this time-dependent wind behaviour, it is likely that more days should be included in the case study. The case study did just consist of 66 days. If 1 day a time shift happened, it might not occur in the clustering.

Chapter 7

Conclusion

The goal of this research was to gain understanding in wind patterns by using unsupervised learning methods.

It can be concluded that unsupervised learning is a valid way to cluster different wind patterns. Using autoencoders for dimensionality reduction provides the K-means clustering algorithm with enough information to create sensible clusters. The autoencoder which performed best was the 3D Convolutional Neural Network (CNN) autoencoder. This autoencoder was computationally very heavy, but using this method was a good way to capture spatio-temporal dependencies in the wind data.

The clusters are checked visually. The cluster containing typhoon Jebi is analyzed for other typhoons. As a typhoon is an extreme wind pattern, it was hypothesized that typhoons will be clustered together. This was indeed the case. Using 2D CNN autoencoders in combination with K-means clustering provided typhoon clusters, but typhoon Jebi was never included in a cluster where solely typhoons were included. The 3D CNN methods changed this. The cluster containing typhoon Jebi consisted of 95.3% typhoons.

Other interesting weather patterns also emerged from the classification. An example is the weather pattern in cluster 42, as discussed in Section 4-4. This weather pattern was unexpected, but significantly present. Many data samples, but not all, in this cluster showed behaviour representative by the average wind speed of the cluster.

This leads to a second conclusion: the average wind speed of a cluster is a good measure to get an idea of the wind patterns contained in this cluster. However, it should be carefully analyzed whether these wind patterns are indeed present in the cluster, as faulty conclusions are easily drawn.

The transitional matrix, as shown in Figures 4-17 and 4-18, provides information about the wind pattern behaviour. Some wind patterns, such as wind patterns in cluster 44, have a high chance of repetition. Given that the current day wind pattern is cluster 44, the chance of it being in cluster 44 the next day as well is 51%.

Other interesting information that can be retrieved from the transitional matrix, is that some days have a 0% chance of repetition. This can also be useful information regarding forecasting, as knowledge can be gained from the fact that a certain weather pattern has never repeated itself in the past 40 years.

In the case study, it was shown that a 3D CNN autoencoder is not always the best method. The 2D CNN autoencoder performed comparable, but included a rare high wind speed event in the clusters. The 3D CNN autoencoder did not include this data as a separate cluster. The transitional change matrix of the 2D CNN method included more extreme values than the 3D CNN method, which is a good thing. Combined with having less computational cost, this caused the 2D CNN autoencoder to be the preferred method for the case study.

In the case study, it is shown that the clustering took into account both wind speed and wind direction. The effect of Oshima island on the wind speed was shown in multiple clusters. The clusters were defined by the magnitude of the effect of the Oshima island.

The main conclusion of this research is that it is indeed possible to gain understanding in wind patterns using unsupervised learning. The fact that wind patterns can be classified without prior data preprocessing is a base that can be further built upon.

Chapter 8

Recommendations

In this research an approach is made for using unsupervised learning methods in meteorology. Without preprocessing the data, some important wind features can be recognized in this research.

In future research, it is advised to take other clustering methods into account. An example of another clustering method is HDBSCAN, which is a density based approach. With HDBSCAN, the amount of clusters is not predetermined, but the algorithm will determine the amount of clusters.

The developed method in this research not only shows potential for wind speed, but other meteorological effects can be clustered as well. Temperature, precipitation, or pressure should be considered a possibility.

Another application for the developed method can be to create a database, for example with typhoons or other wind patterns. The example shown in Figure 4-11 in Section 4-3 shows unexpected wind patterns. It is less labour intensive to check all the data points within this cluster than to check all the wind patterns in the entire data set. If a database is built, eventually supervised learning will become possible, as labels for a particular dataset will become available.

This can also be done using a semi-supervised learning method. A couple data points will be assigned a label, and the algorithm will assign the same label to the data sample which most closely resembles this weather pattern. The resulting database can be used to further analyze particular wind behaviour.

The 3D Convolutional Neural Network (CNN) autoencoder showed potential in this research, as it was a welcome addition to the previously discussed 2D CNN autoencoders. It is advised to further develop this 3D CNN method, using more data or different autoencoder layers.

Appendix A

Structures of the Autoencoders

A-1 Input Data: U10 and V10

A-1-1 Single Layered Autoencoder

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 140, 2)	0
flatten_1 (Flatten)	(None, 28000)	0
dense_1 (Dense)	(None, 100)	2800100
dense_2 (Dense)	(None, 28000)	2828000
reshape_1 (Reshape)	(None, 100, 140, 2)	0

Total params: 5,628,100
Trainable params: 5,628,100
Non-trainable params: 0

Figure A-1: Structure of the single layered autoencoder

A-1-2 Deep Autoencoder

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 100, 140, 2)	0
flatten_2 (Flatten)	(None, 28000)	0
dense_7 (Dense)	(None, 1000)	28001000
dense_8 (Dense)	(None, 500)	500500
dense_9 (Dense)	(None, 100)	50100
dense_10 (Dense)	(None, 500)	50500
dense_11 (Dense)	(None, 1000)	501000
dense_12 (Dense)	(None, 28000)	28028000
reshape_2 (Reshape)	(None, 100, 140, 2)	0

Total params: 57,131,100
Trainable params: 57,131,100
Non-trainable params: 0

Figure A-2: Structure of the deep autoencoder

A-1-3 CNN Autoencoder

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 140, 2)	0
conv2d_1 (Conv2D)	(None, 100, 140, 64)	1216
conv2d_2 (Conv2D)	(None, 100, 140, 16)	9232
conv2d_3 (Conv2D)	(None, 100, 140, 1)	145
flatten_1 (Flatten)	(None, 14000)	0
dense_1 (Dense)	(None, 100)	1400100
dense_2 (Dense)	(None, 14000)	1414000
reshape_1 (Reshape)	(None, 100, 140, 1)	0
conv2d_transpose_1 (Conv2DTr	(None, 100, 140, 16)	160
conv2d_transpose_2 (Conv2DTr	(None, 100, 140, 64)	9280
conv2d_transpose_3 (Conv2DTr	(None, 100, 140, 2)	1154

Total params: 2,835,287
Trainable params: 2,835,287
Non-trainable params: 0

Figure A-3: Structure of the Convolutional Neural Network (CNN) autoencoder

A-1-4 Deep CNN Autoencoder

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 100, 140, 2)	0
conv2d_4 (Conv2D)	(None, 100, 140, 64)	1216
conv2d_5 (Conv2D)	(None, 100, 140, 16)	9232
conv2d_6 (Conv2D)	(None, 100, 140, 1)	145
flatten_2 (Flatten)	(None, 14000)	0
dense_3 (Dense)	(None, 500)	7000500
dense_4 (Dense)	(None, 250)	125250
dense_5 (Dense)	(None, 100)	25100
dense_6 (Dense)	(None, 250)	25250
dense_7 (Dense)	(None, 500)	125500
dense_8 (Dense)	(None, 14000)	7014000
reshape_2 (Reshape)	(None, 100, 140, 1)	0
conv2d_transpose_4 (Conv2DTr	(None, 100, 140, 16)	160
conv2d_transpose_5 (Conv2DTr	(None, 100, 140, 64)	9280
conv2d_transpose_6 (Conv2DTr	(None, 100, 140, 2)	1154

Total params: 14,336,787
Trainable params: 14,336,787
Non-trainable params: 0

Figure A-4: Structure of the deep CNN autoencoder

A-1-5 CNN Autoencoder with Max Pooling Layers

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 140, 2)	0
conv2d_1 (Conv2D)	(None, 100, 140, 64)	1216
max_pooling2d_1 (MaxPooling2	(None, 50, 70, 64)	0
conv2d_2 (Conv2D)	(None, 50, 70, 16)	9232
max_pooling2d_2 (MaxPooling2	(None, 25, 35, 16)	0
conv2d_3 (Conv2D)	(None, 25, 35, 1)	145
flatten_1 (Flatten)	(None, 875)	0
dense_1 (Dense)	(None, 100)	87600
dense_2 (Dense)	(None, 875)	88375
reshape_1 (Reshape)	(None, 25, 35, 1)	0
conv2d_transpose_1 (Conv2DTr	(None, 25, 35, 16)	160
up_sampling2d_1 (UpSampling2	(None, 50, 70, 16)	0
conv2d_transpose_2 (Conv2DTr	(None, 50, 70, 64)	9280
up_sampling2d_2 (UpSampling2	(None, 100, 140, 64)	0
conv2d_transpose_3 (Conv2DTr	(None, 100, 140, 2)	1154

Total params: 197,162
Trainable params: 197,162
Non-trainable params: 0

Figure A-5: Structure of the CNN autoencoder with max pooling layers

A-1-6 Deep CNN Autoencoder with Max Pooling Layers

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 100, 140, 2)	0
conv2d_4 (Conv2D)	(None, 100, 140, 64)	1216
max_pooling2d_3 (MaxPooling2	(None, 50, 70, 64)	0
conv2d_5 (Conv2D)	(None, 50, 70, 16)	9232
max_pooling2d_4 (MaxPooling2	(None, 25, 35, 16)	0
conv2d_6 (Conv2D)	(None, 25, 35, 1)	145
flatten_2 (Flatten)	(None, 875)	0
dense_7 (Dense)	(None, 500)	438000
dense_8 (Dense)	(None, 250)	125250
dense_9 (Dense)	(None, 100)	25100
dense_10 (Dense)	(None, 250)	25250
dense_11 (Dense)	(None, 500)	125500
dense_12 (Dense)	(None, 875)	438375
reshape_2 (Reshape)	(None, 25, 35, 1)	0
conv2d_transpose_4 (Conv2DTr	(None, 25, 35, 16)	160
up_sampling2d_3 (UpSampling2	(None, 50, 70, 16)	0
conv2d_transpose_5 (Conv2DTr	(None, 50, 70, 64)	9280
up_sampling2d_4 (UpSampling2	(None, 100, 140, 64)	0
conv2d_transpose_6 (Conv2DTr	(None, 100, 140, 2)	1154

Total params: 1,198,662
Trainable params: 1,198,662
Non-trainable params: 0

Figure A-6: Structure of the deep CNN autoencoder with max pooling layers

A-2 Case Study

A-2-1 2D CNN autoencoder

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 174, 174, 2)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 174, 174, 64)      1216
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 174, 174, 16)      9232
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 174, 174, 1)       145
_________________________________________________________________
flatten_2 (Flatten)          (None, 30276)             0
_________________________________________________________________
dense_3 (Dense)              (None, 100)               3027700
_________________________________________________________________
dense_4 (Dense)              (None, 30276)             3057876
_________________________________________________________________
reshape_2 (Reshape)          (None, 174, 174, 1)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 174, 174, 16)      160
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 174, 174, 64)      9280
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 174, 174, 2)       1154
=================================================================
Total params: 6,106,763
Trainable params: 6,106,763
Non-trainable params: 0
```

Figure A-7: Structure of the 2D CNN autoencoder used for the case study

A-2-2 3D CNN autoencoder

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 174, 174, 6, 2)	0
conv3d_1 (Conv3D)	(None, 174, 174, 6, 64)	3520
conv3d_2 (Conv3D)	(None, 174, 174, 6, 16)	27664
conv3d_3 (Conv3D)	(None, 174, 174, 6, 1)	433
flatten_1 (Flatten)	(None, 181656)	0
dense_1 (Dense)	(None, 100)	18165700
dense_2 (Dense)	(None, 181656)	18347256
reshape_1 (Reshape)	(None, 174, 174, 6, 1)	0
conv3d_transpose_1 (Conv3DTr	(None, 174, 174, 6, 16)	448
conv3d_transpose_2 (Conv3DTr	(None, 174, 174, 6, 64)	27712
conv3d_transpose_3 (Conv3DTr	(None, 174, 174, 6, 2)	3458

Total params: 36,576,191
Trainable params: 36,576,191
Non-trainable params: 0

Figure A-8: Structure of the 3D CNN autoencoder used for the case study

Appendix B

Images

B-1 Autoencoders using Different Neuron Values

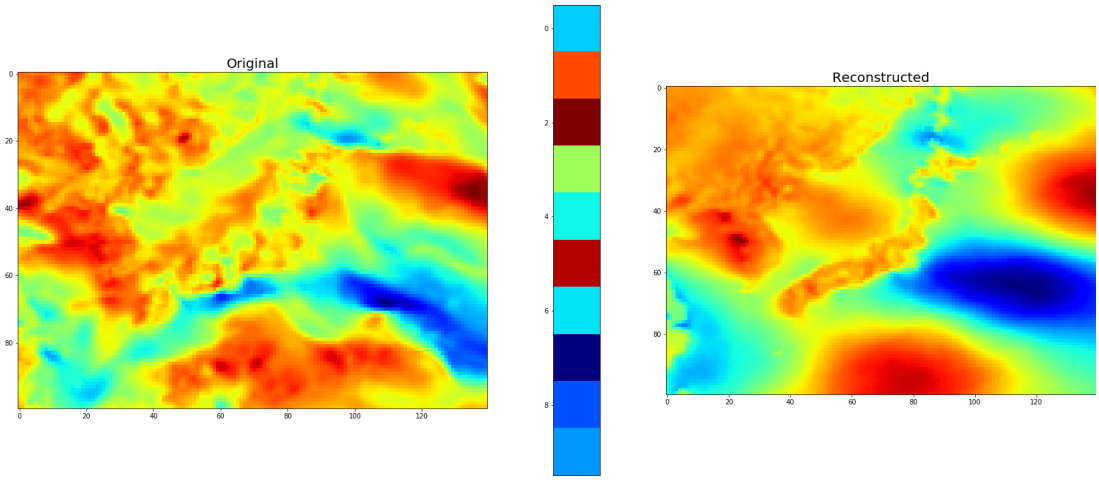


Figure B-1: Autoencoder using 10 values in the middle layer

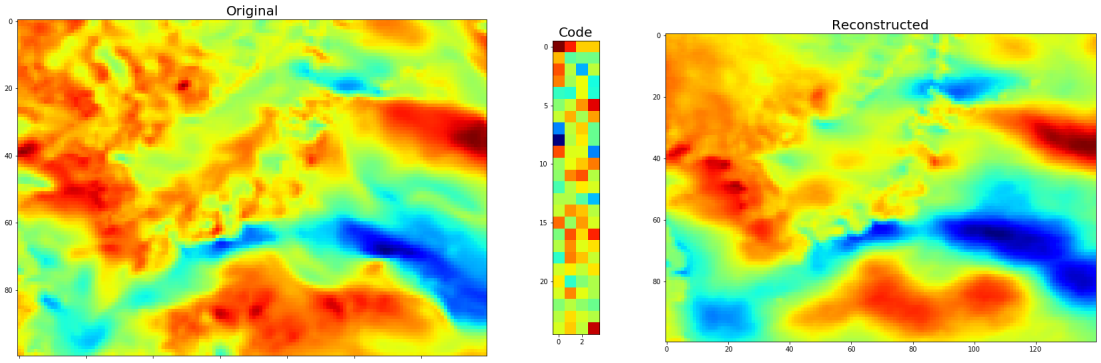


Figure B-2: Autoencoder using 100 values in the middle layer

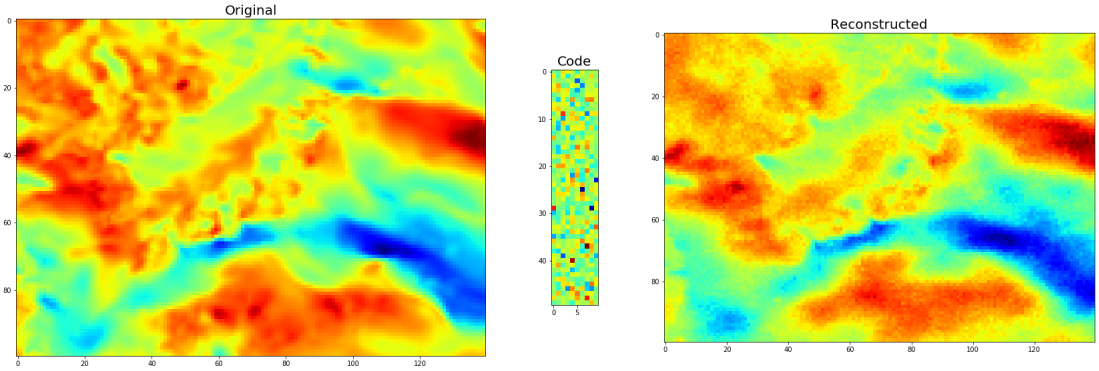


Figure B-3: Autoencoder using 500 values in the middle layer

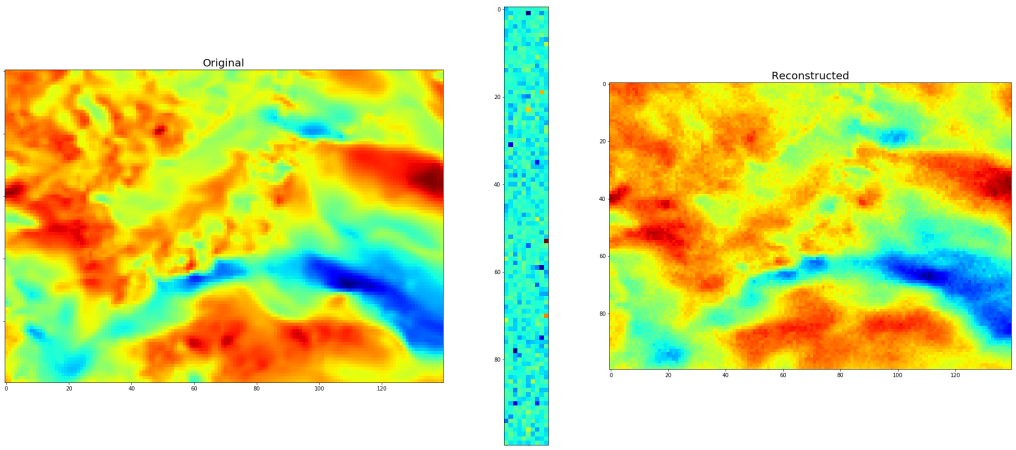


Figure B-4: Autoencoder using 1000 values in the middle layer

B-2 Data Samples in the Jebi cluster, Input Data: u10

B-2-1 Single Layered Autoencoder

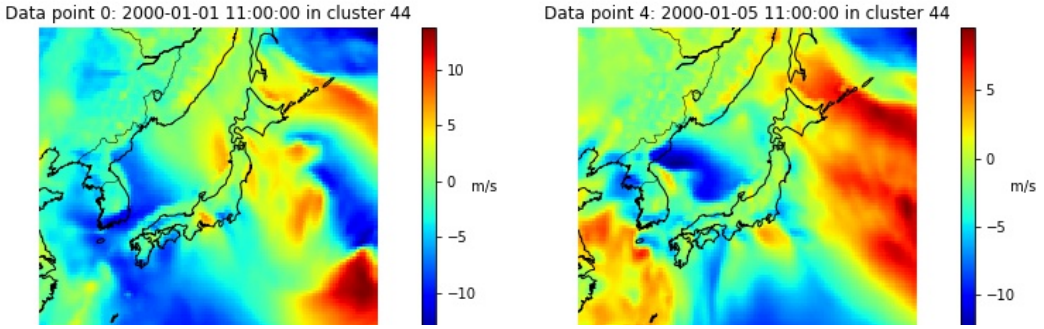


Figure B-5: Two examples of clustered points that are not typhoons

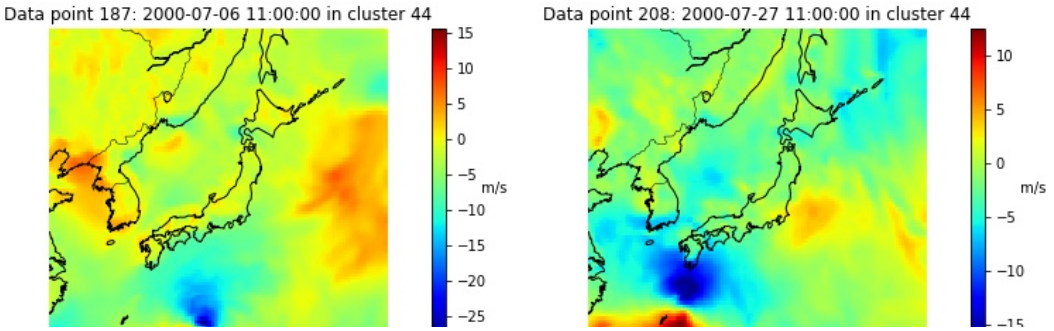


Figure B-6: Two examples of clustered points that are typhoons

B-2-2 Deep Autoencoder

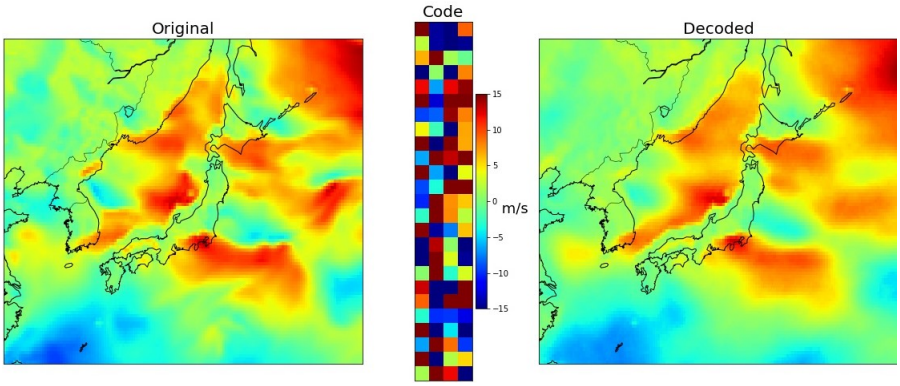


Figure B-7: Example of a validation data sample using the deep autoencoder

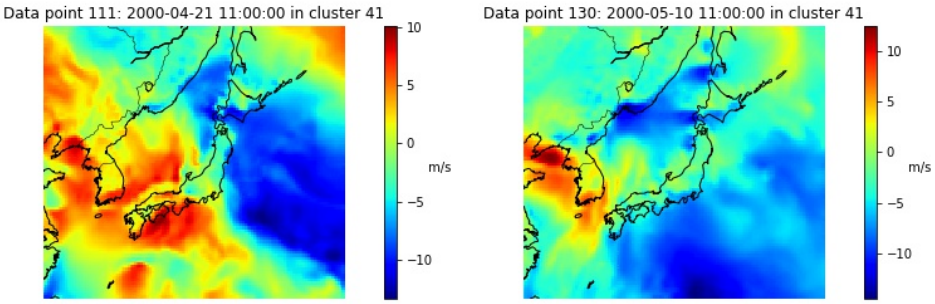


Figure B-8: Two examples of clustered points that are not typhoons

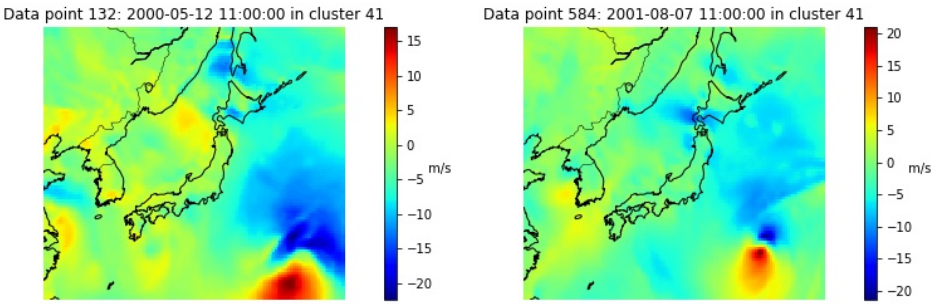


Figure B-9: Two examples of clustered points that are not typhoons

B-2-3 CNN Autoencoder

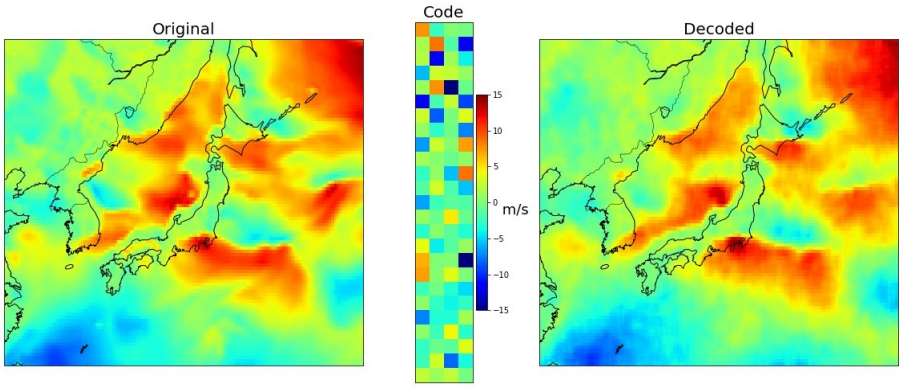


Figure B-10: Example of a validation data sample using the CNN autoencoder

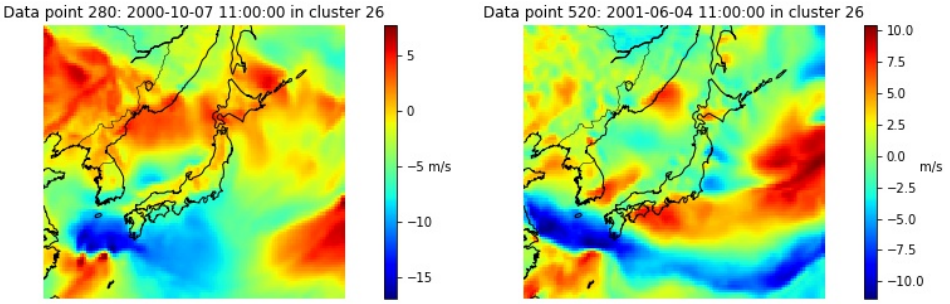


Figure B-11: Two examples of clustered points that are not typhoons

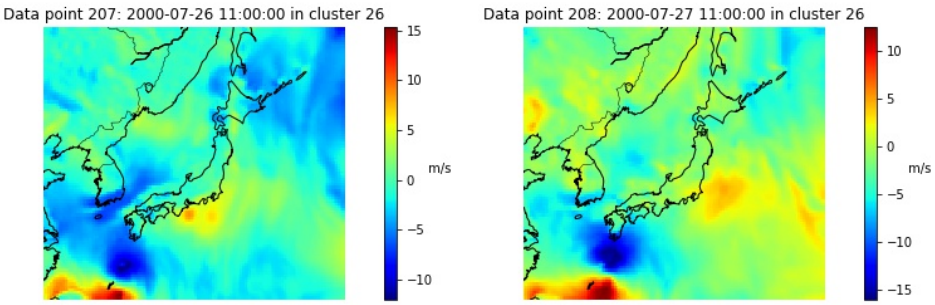


Figure B-12: Two examples of clustered points that are typhoons

B-2-4 Deep CNN Autoencoder

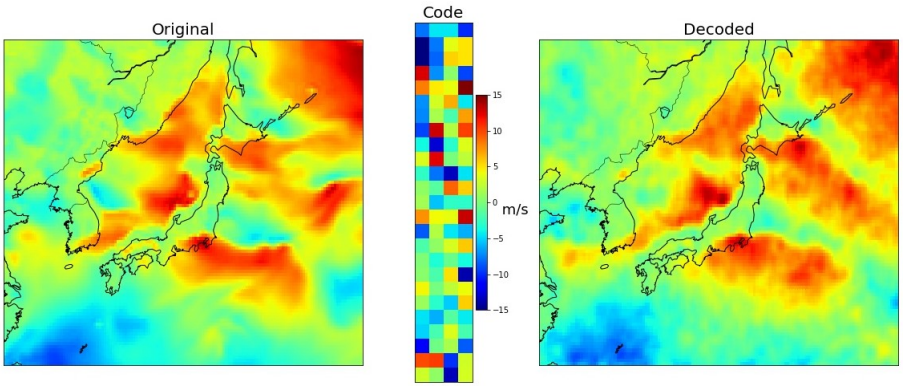


Figure B-13: Example of a validation data sample using the deep CNN autoencoder

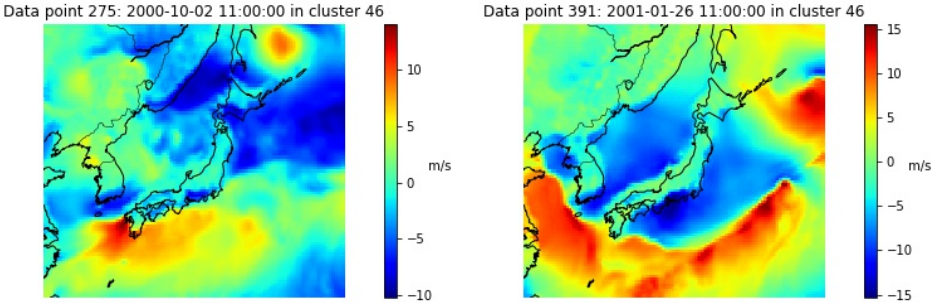


Figure B-14: Two examples of clustered points that are not typhoons

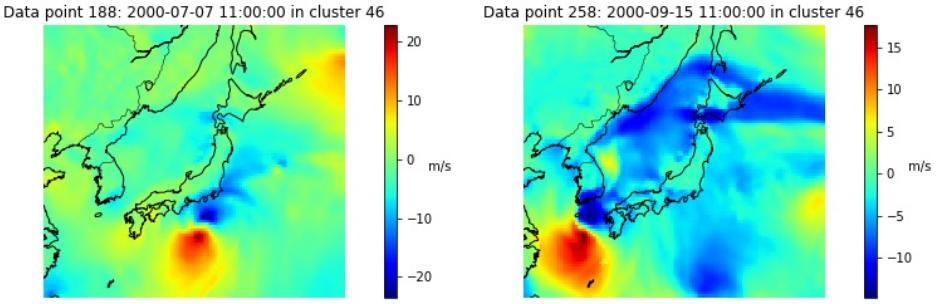


Figure B-15: Two examples of clustered points that are typhoons

B-2-5 CNN Autoencoder with Max Pooling Layers

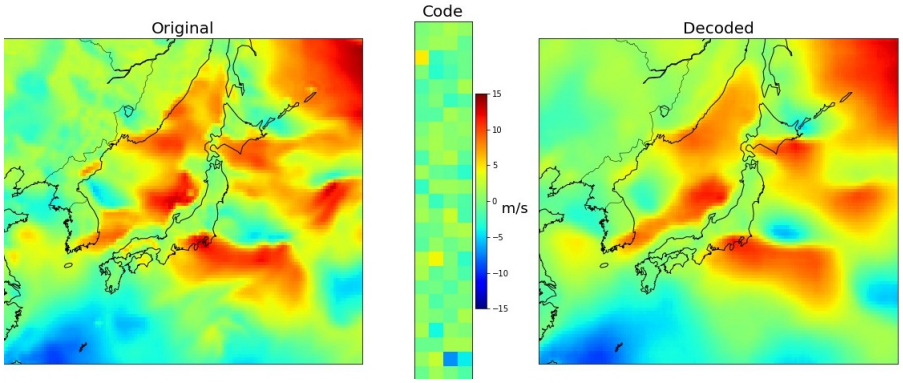


Figure B-16: Example of a validation data sample using the CNN autoencoder

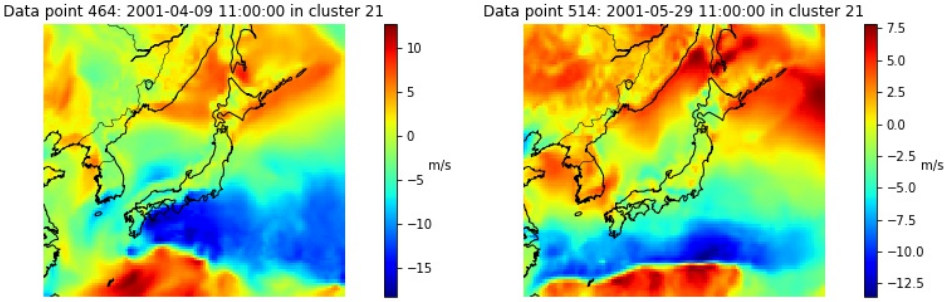


Figure B-17: Two examples of clustered points that are not typhoons

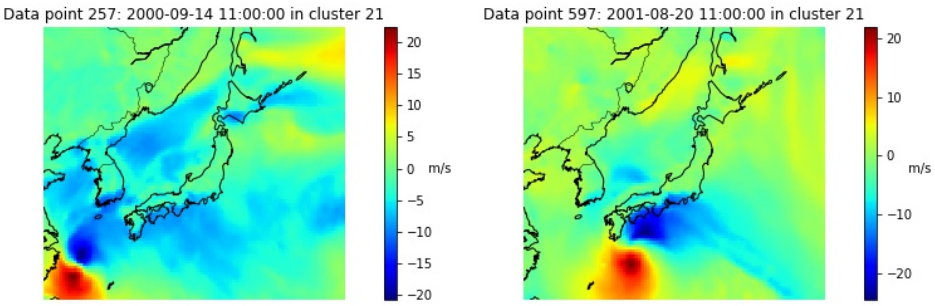


Figure B-18: Two examples of clustered points that are typhoons

B-2-6 Deep CNN Autoencoder with Max Pooling Layers

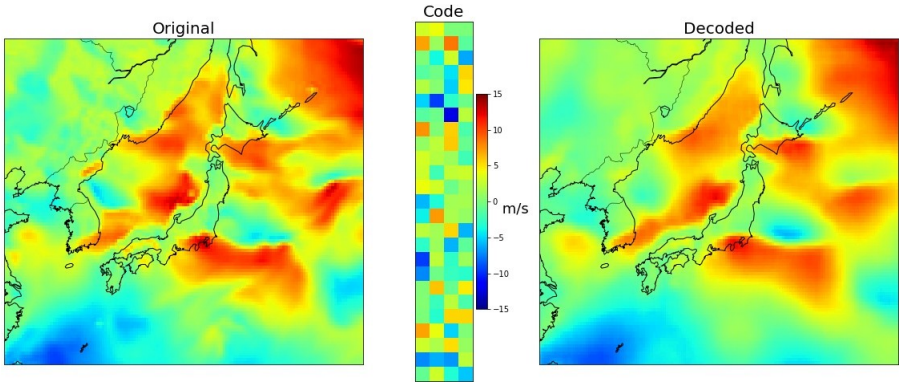


Figure B-19: Example of a validation data sample using the deep CNN autoencoder

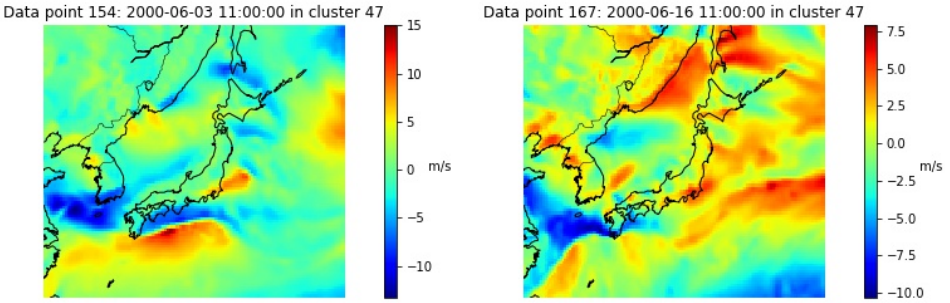


Figure B-20: Two examples of clustered points that are not typhoons

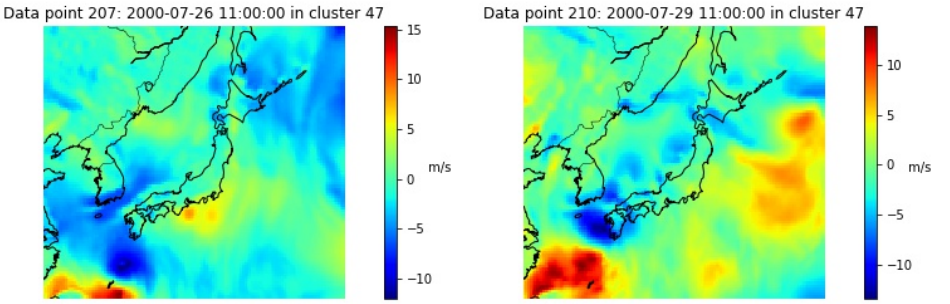


Figure B-21: Two examples of clustered points that are typhoons

B-3 Data Samples in the Jebi cluster, Input Data: u10 and v10

B-3-1 No Autoencoder

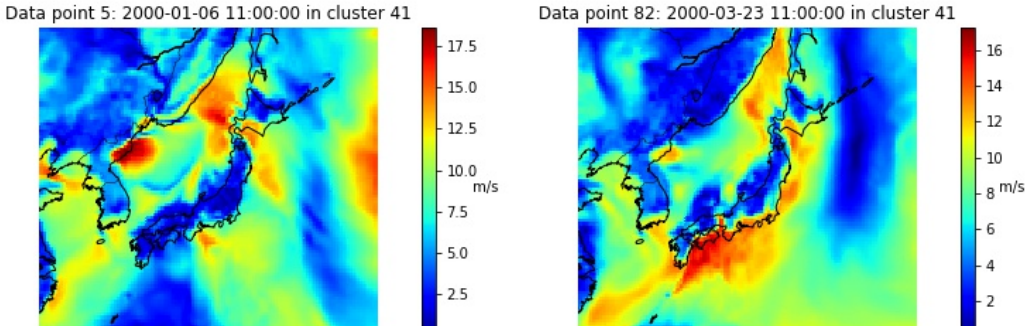


Figure B-22: Two examples of clustered points that are not typhoons

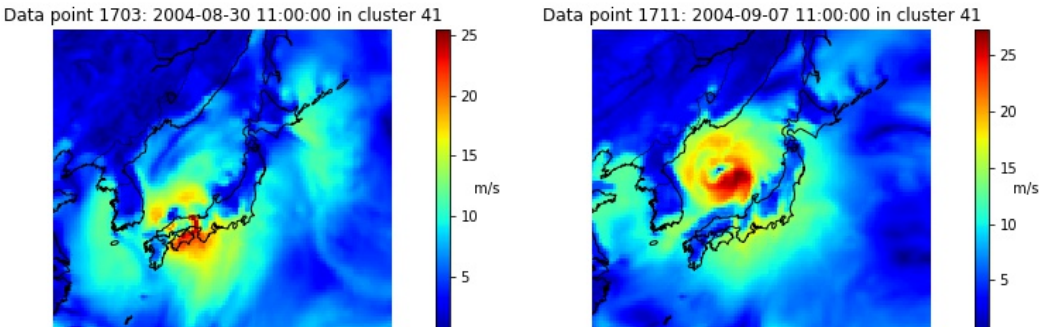


Figure B-23: Two examples of clustered points that are typhoons

B-3-2 Single Layered Autoencoder

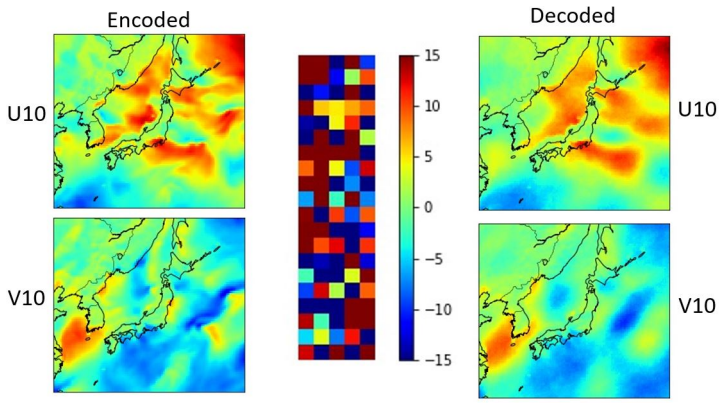


Figure B-24: Example of a validation sample using the single layered autoencoder

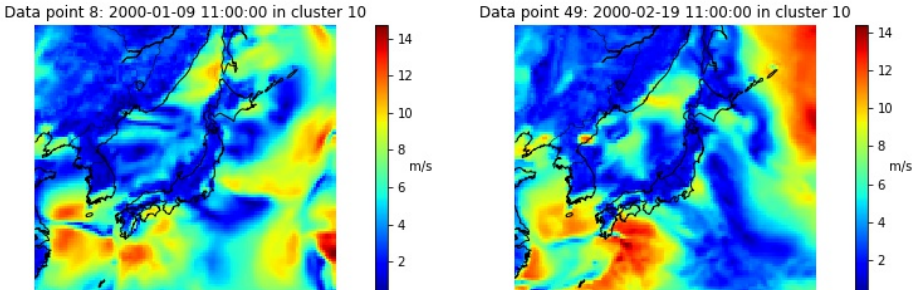


Figure B-25: Two examples of clustered points that are not typhoons

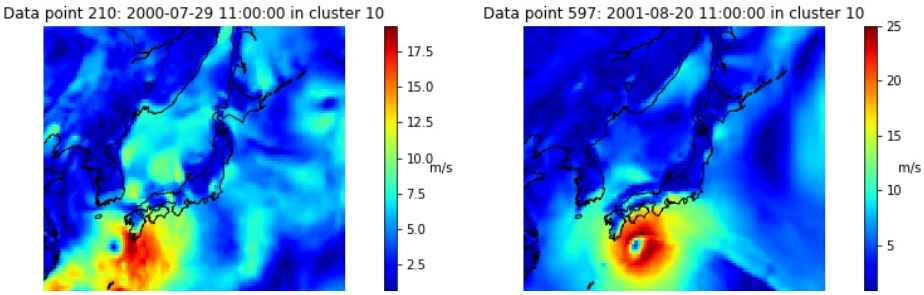


Figure B-26: Two examples of clustered points that are typhoons

B-3-3 Deep Autoencoder

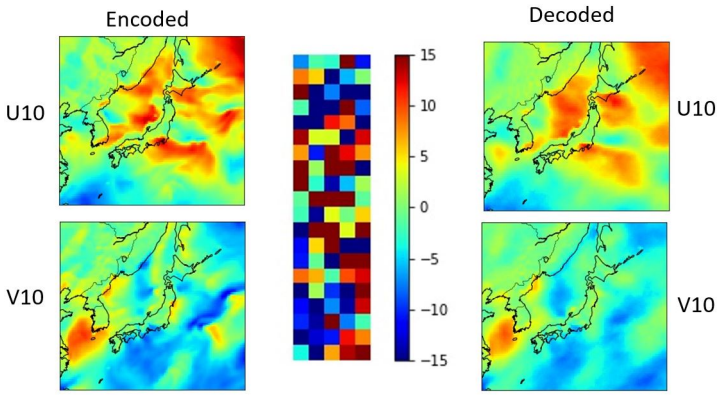


Figure B-27: Example of a validation data sample using the deep autoencoder

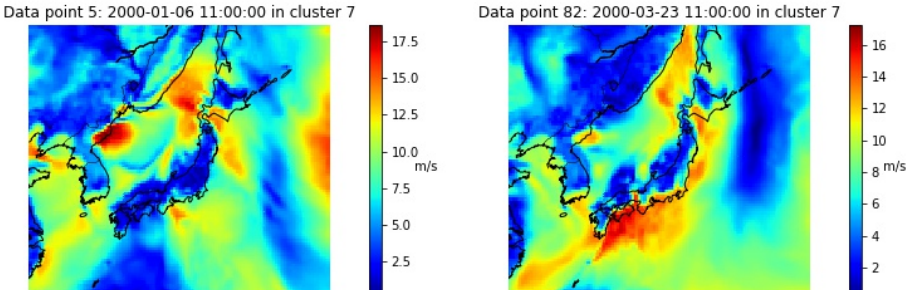


Figure B-28: Two examples of clustered points that are not typhoons

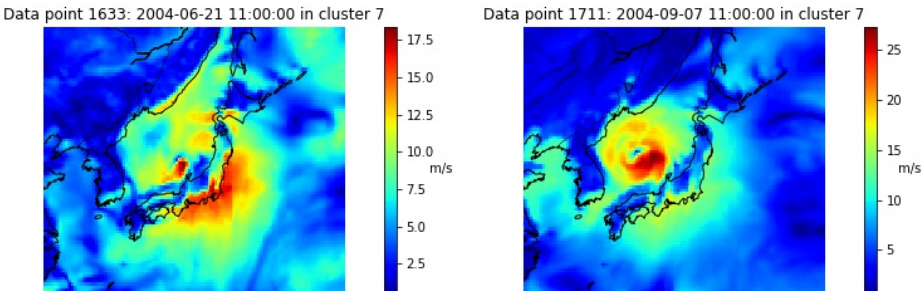


Figure B-29: Two examples of clustered points that are typhoons

B-3-4 CNN Autoencoder

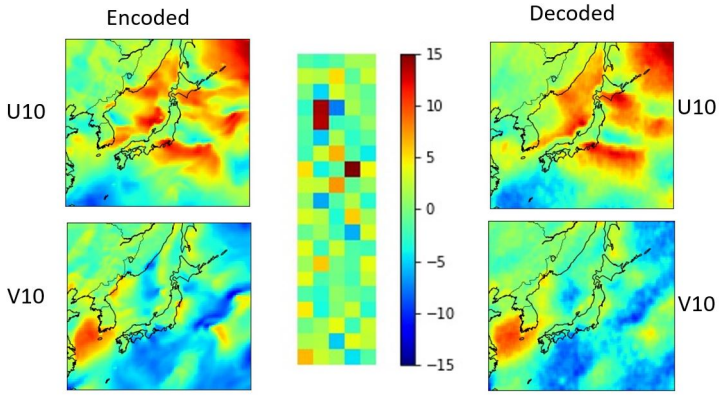


Figure B-30: Example of a validation data sample using the Convolutional Neural Network (CNN) autoencoder

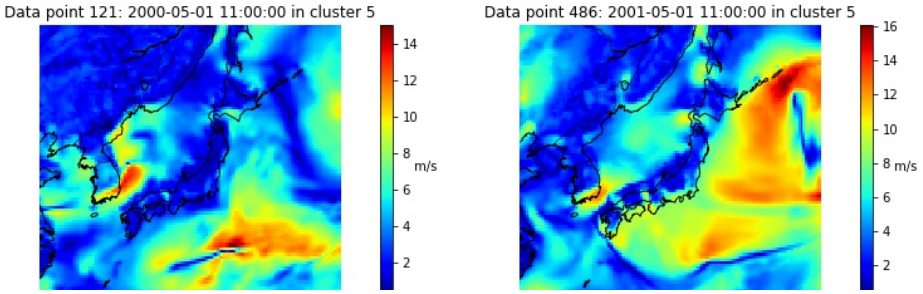


Figure B-31: Two examples of clustered points that are not typhoons

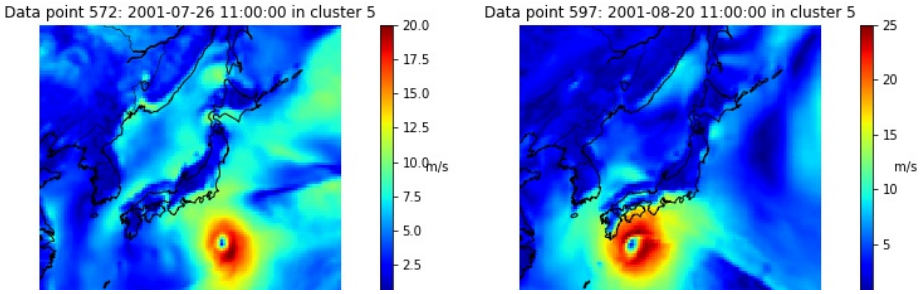


Figure B-32: Two examples of clustered points that are typhoons

B-3-5 Deep CNN Autoencoder

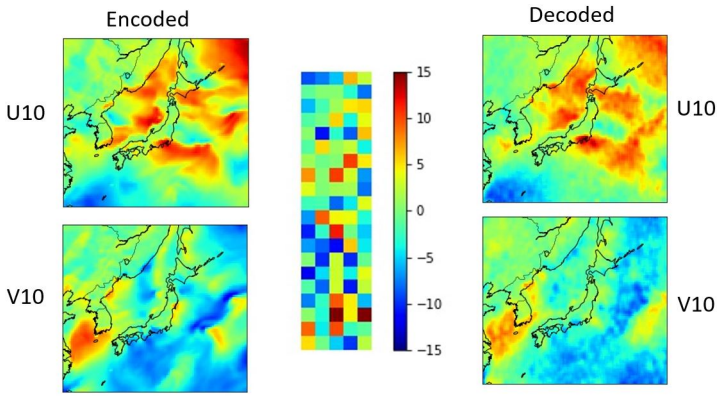


Figure B-33: Example of a validation data sample using the deep CNN autoencoder

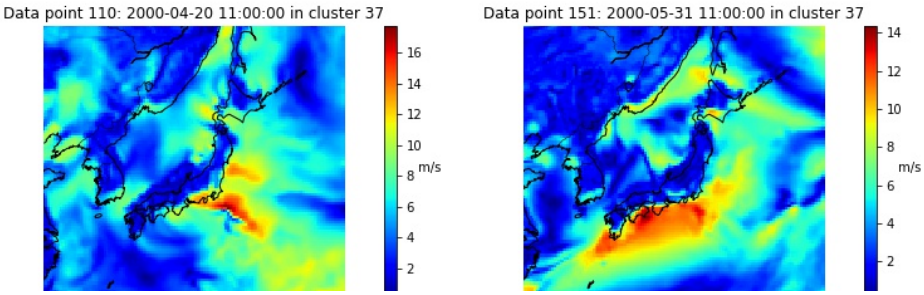


Figure B-34: Two examples of clustered points that are not typhoons

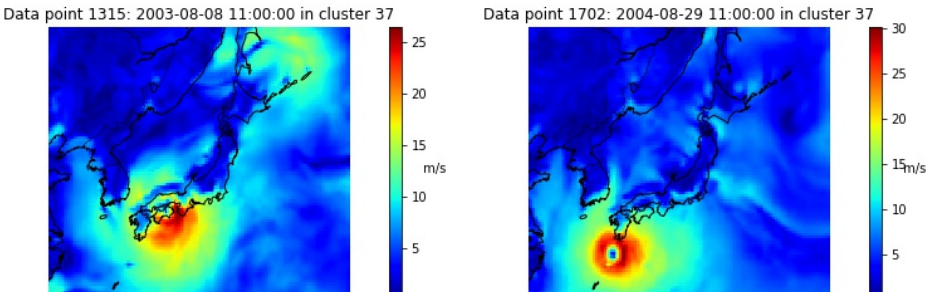


Figure B-35: Two examples of clustered points that are typhoons

B-3-6 CNN Autoencoder with Max Pooling Layers

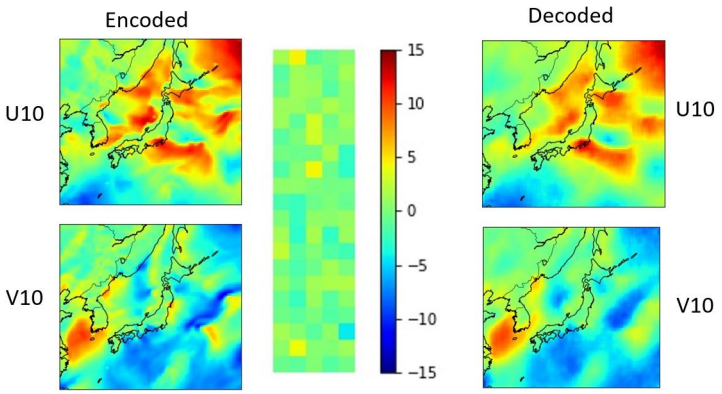


Figure B-36: Example of a validation data sample using the CNN autoencoder with max pooling layers

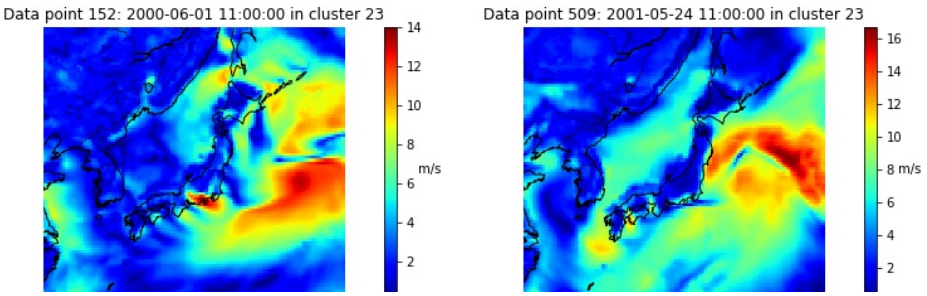


Figure B-37: Two examples of clustered points that are not typhoons

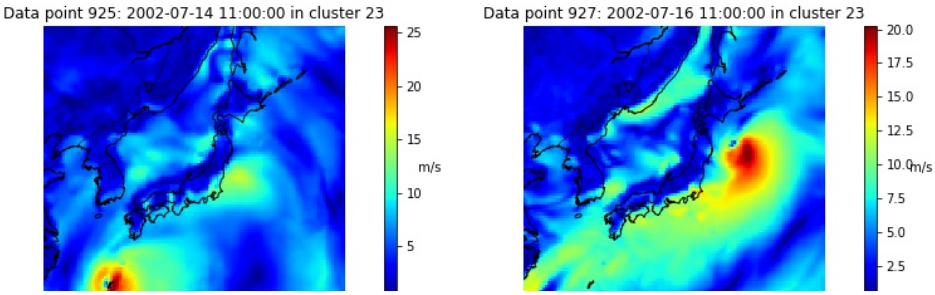


Figure B-38: Two examples of clustered points that are typhoons

B-3-7 Deep CNN Autoencoder with Max Pooling Layers

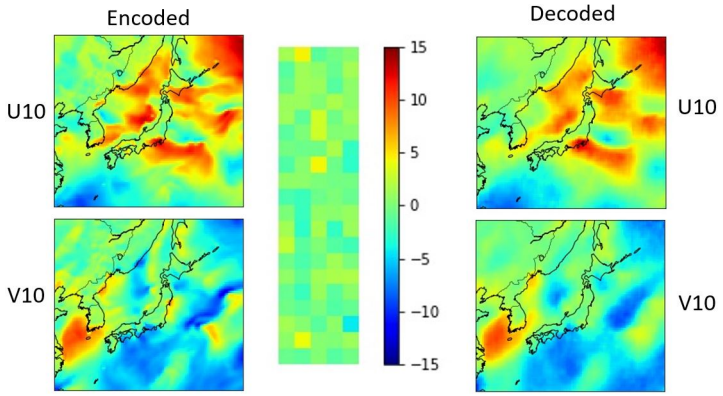


Figure B-39: Example of a validation data sample using the deep CNN autoencoder with max pooling layers

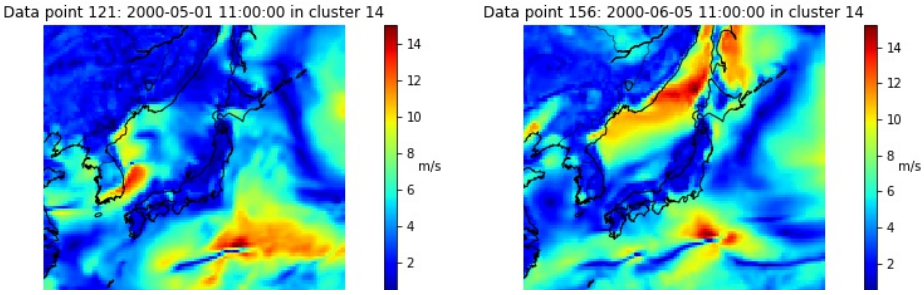


Figure B-40: Two examples of clustered points that are not typhoons

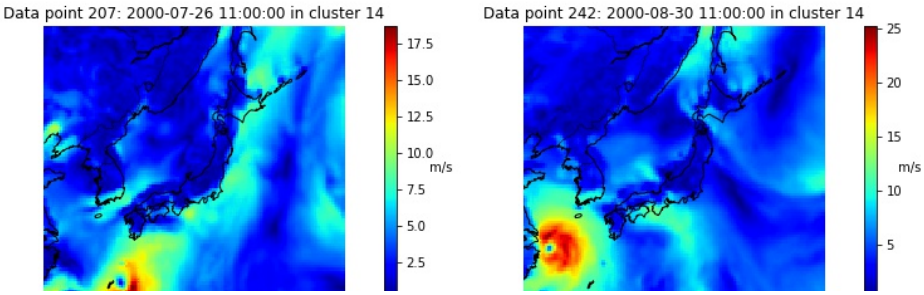
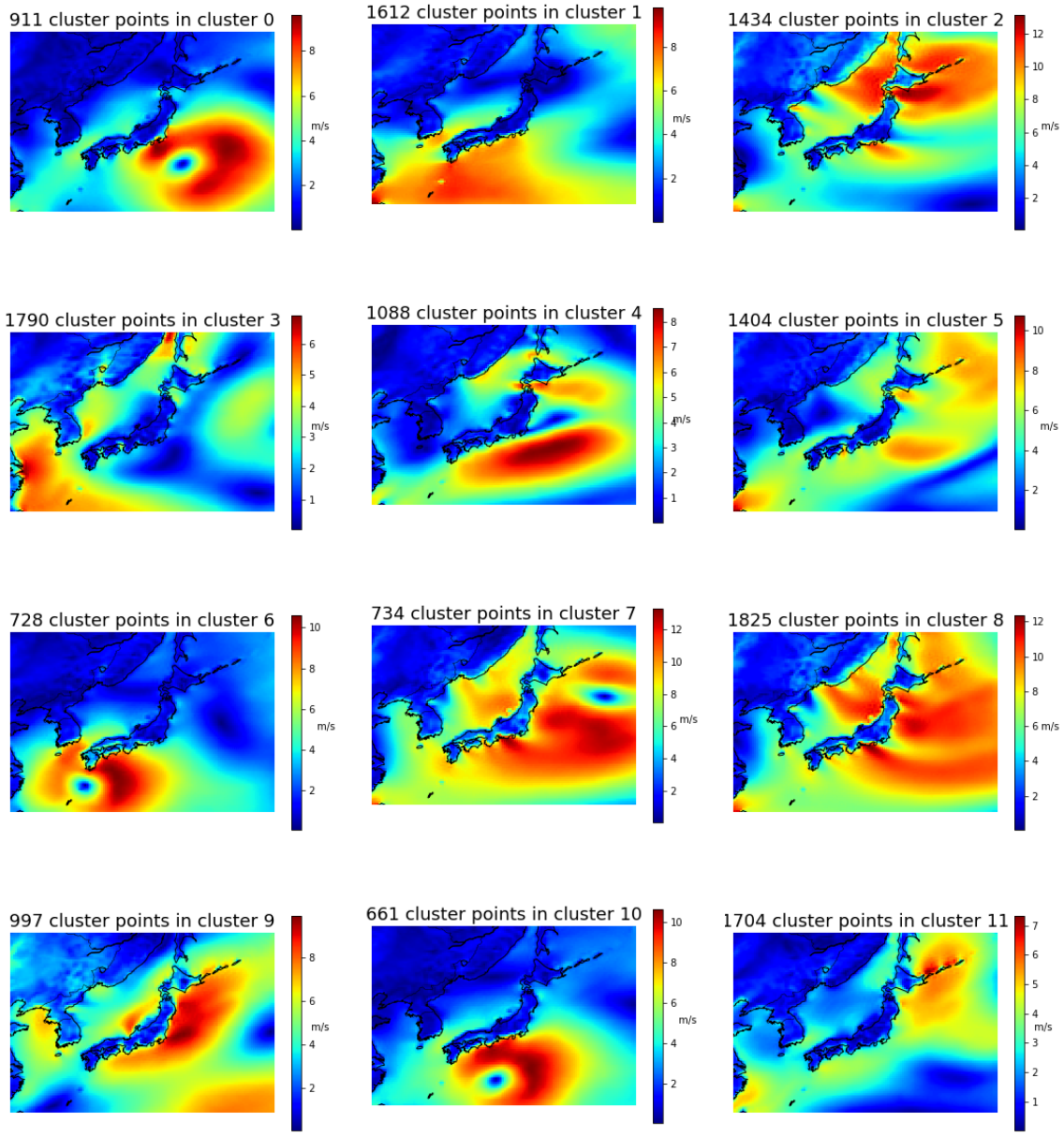
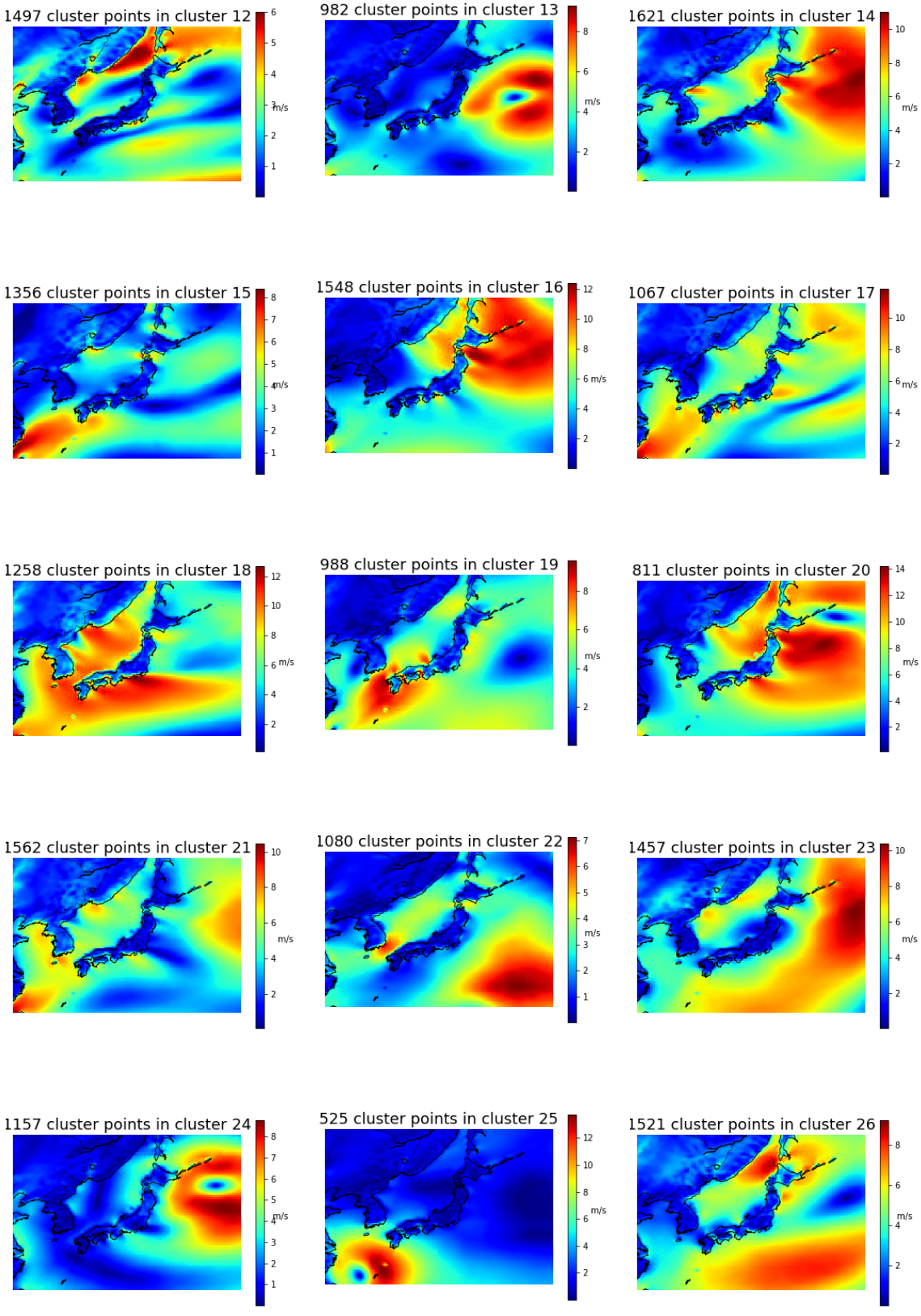
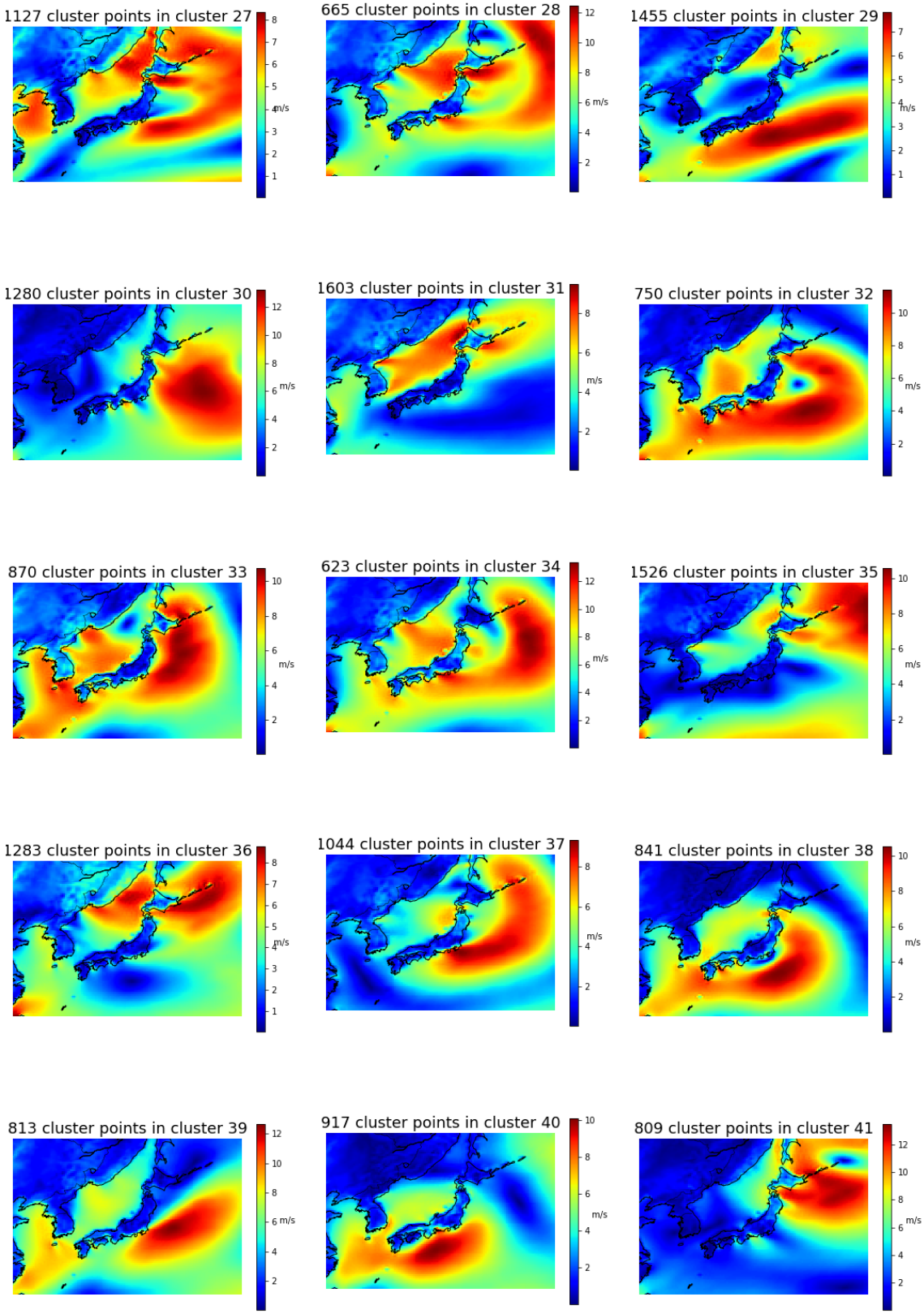


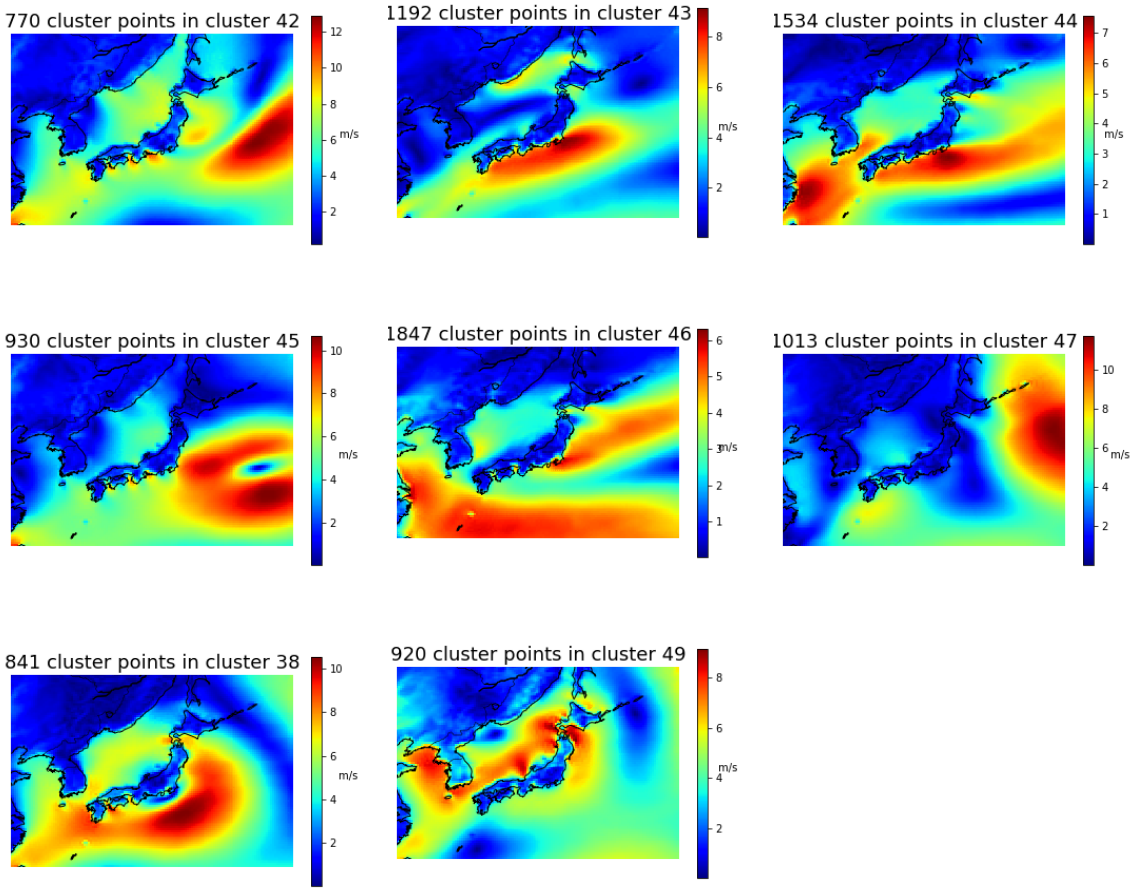
Figure B-41: Two examples of clustered points that are typhoons

B-4 Cluster Average Wind Speed CNN Autoencoder




1497 cluster points in cluster 12
982 cluster points in cluster 13
1621 cluster points in cluster 14
1356 cluster points in cluster 15
1548 cluster points in cluster 16
1067 cluster points in cluster 17
1258 cluster points in cluster 18
988 cluster points in cluster 19
811 cluster points in cluster 20
1562 cluster points in cluster 21
1080 cluster points in cluster 22
1457 cluster points in cluster 23
1157 cluster points in cluster 24
525 cluster points in cluster 25
1521 cluster points in cluster 26
m/s


1127 cluster points in cluster 27
665 cluster points in cluster 28
1455 cluster points in cluster 29
1280 cluster points in cluster 30
1603 cluster points in cluster 31
750 cluster points in cluster 32
870 cluster points in cluster 33
623 cluster points in cluster 34
1526 cluster points in cluster 35
1283 cluster points in cluster 36
1044 cluster points in cluster 37
841 cluster points in cluster 38
813 cluster points in cluster 39
917 cluster points in cluster 40
809 cluster points in cluster 41
m/s


770 cluster points in cluster 42
1192 cluster points in cluster 43
1534 cluster points in cluster 44
930 cluster points in cluster 45
1847 cluster points in cluster 46
1013 cluster points in cluster 47
841 cluster points in cluster 38
920 cluster points in cluster 49
m/s

B-5 3D CNN Autoencoder

B-5-1 Autoencoder Performance

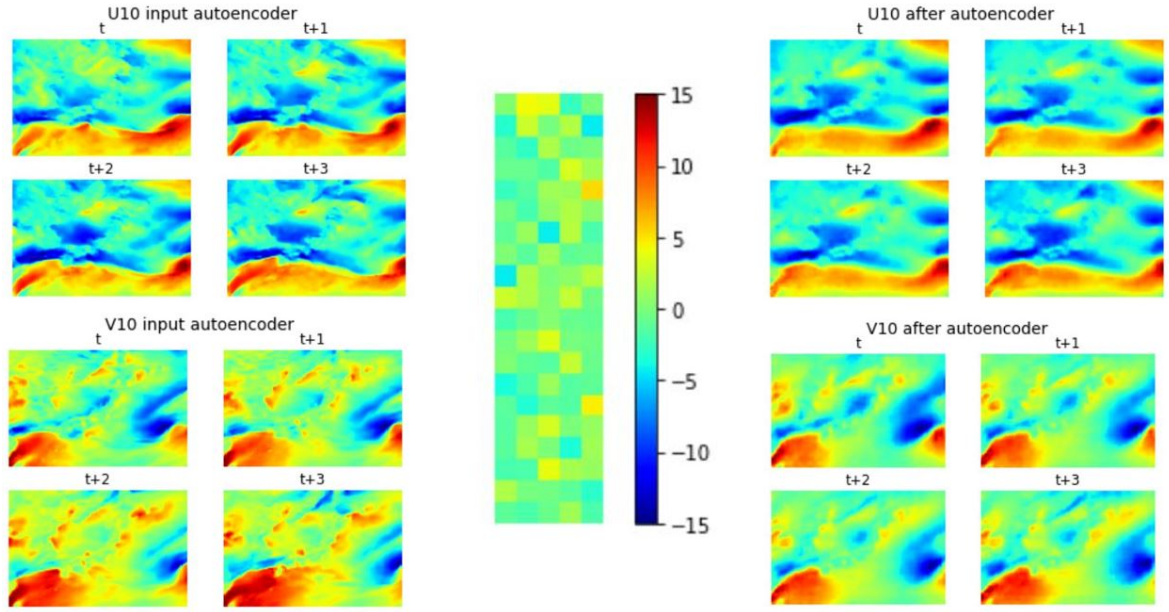


Figure B-59: Performance of the 3D CNN Autoencoder

B-5-2 Comparing Typhoon Jebi with Average wind speeds for all time steps

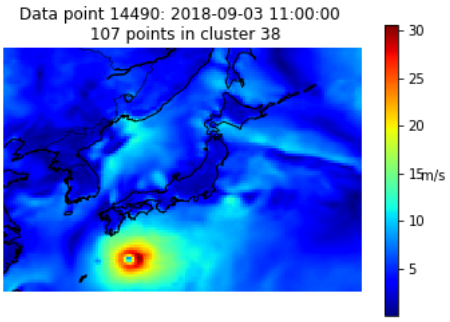


Figure B-60: Typhoon Jebi, clustered with a 3D CNN autoencoder using all the data. Second time step.

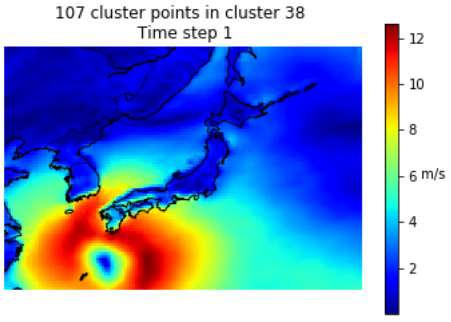


Figure B-61: The average wind speed of cluster 38 using the second time step

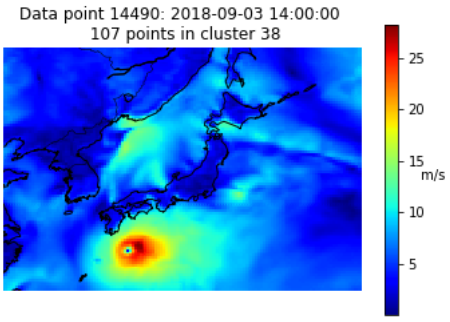


Figure B-62: Typhoon Jebi, clustered with a 3D CNN autoencoder using all the data. Third time step

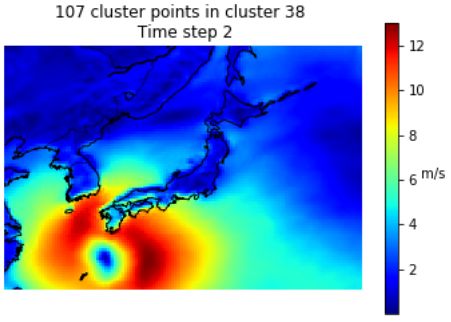


Figure B-63: The average wind speed of cluster 38 using the third time step

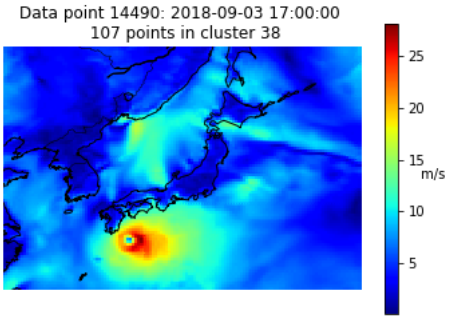


Figure B-64: Typhoon Jebi, clustered with a 3D CNN autoencoder using all the data. Fourth time step

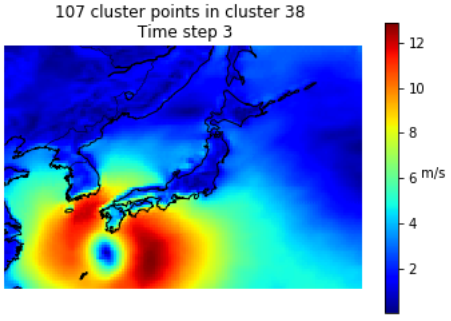
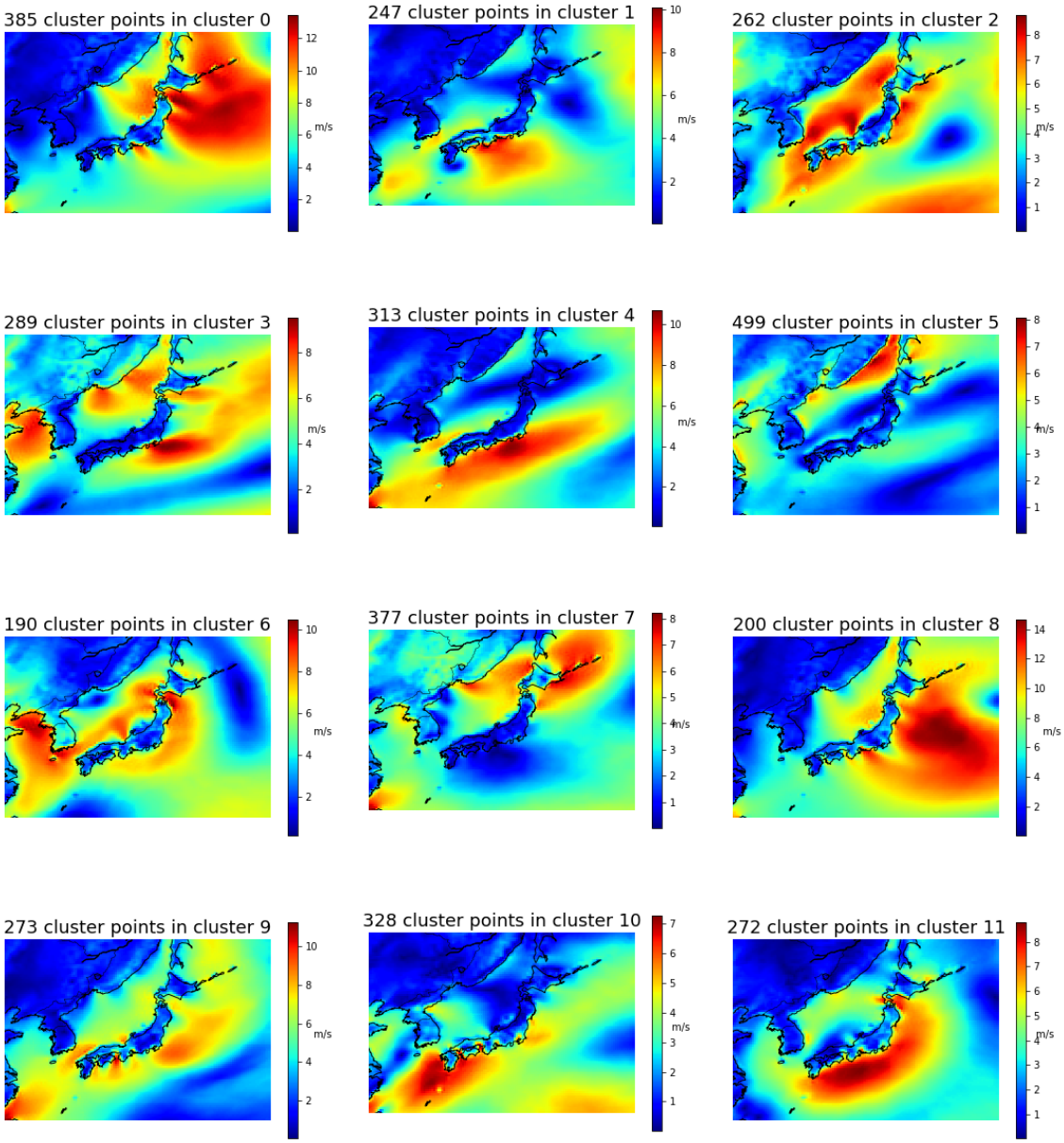


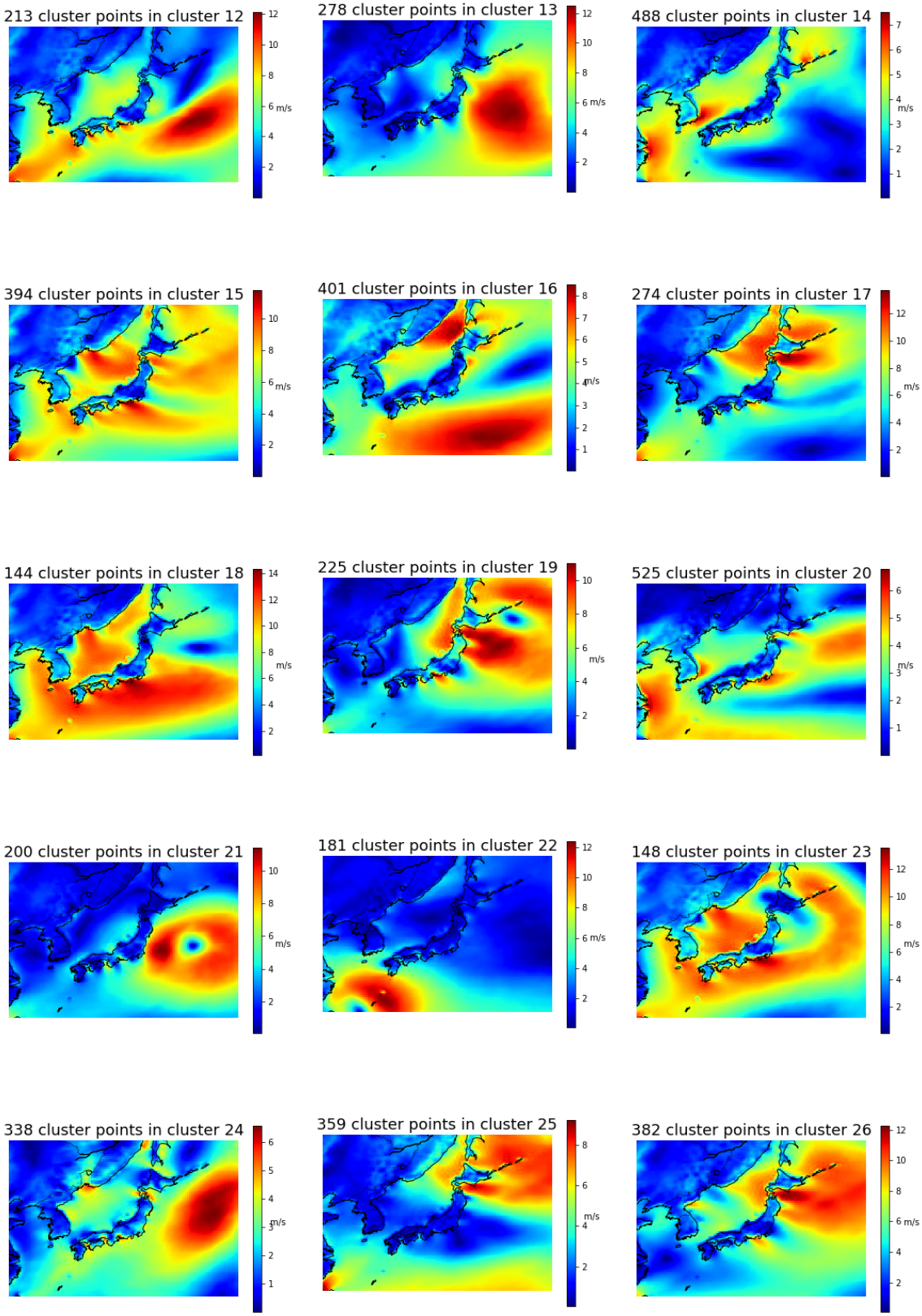
Figure B-65: The average wind speed of cluster 38 using the fourth time step

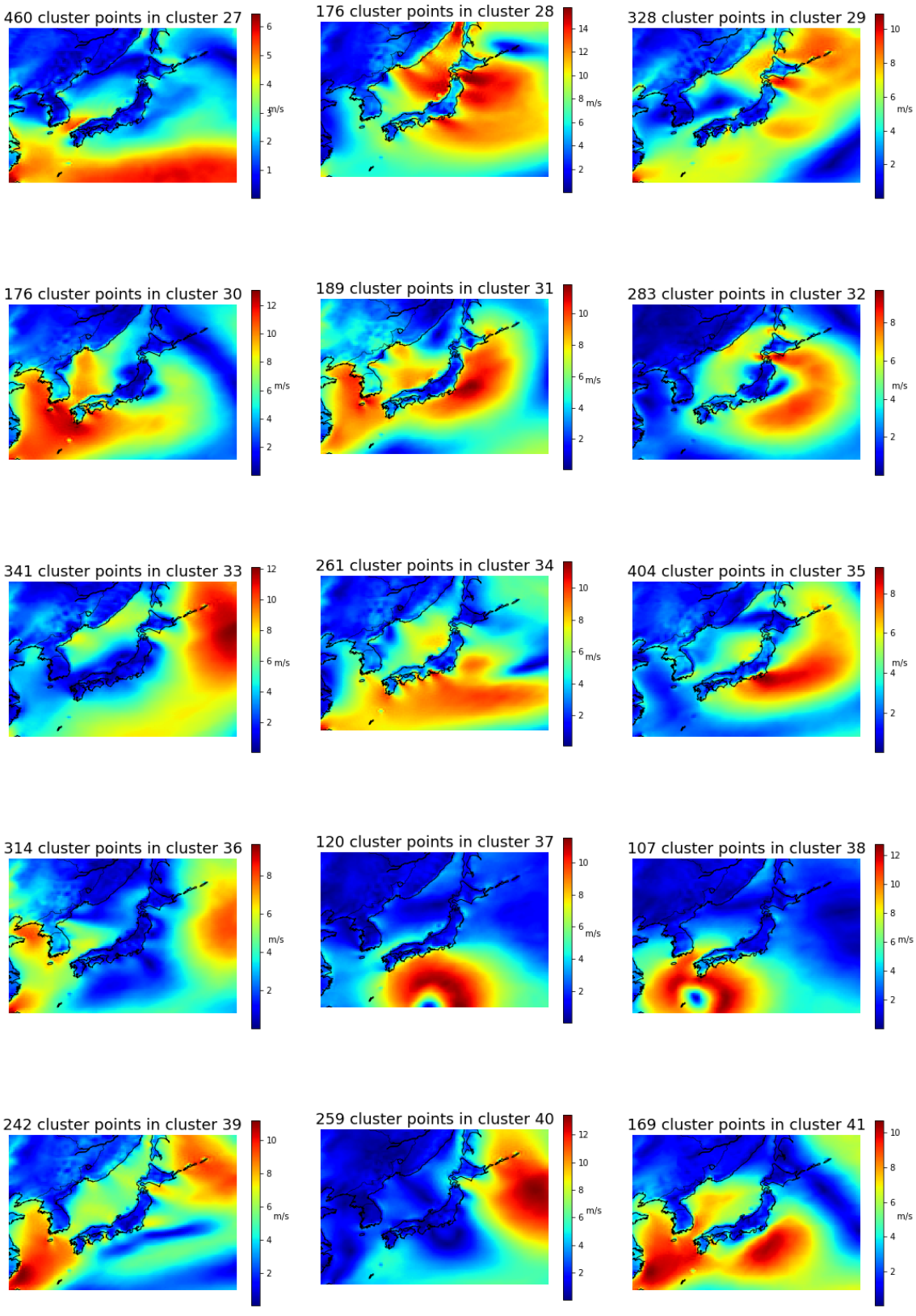
B-5-3 Typhoons included in Cluster 38

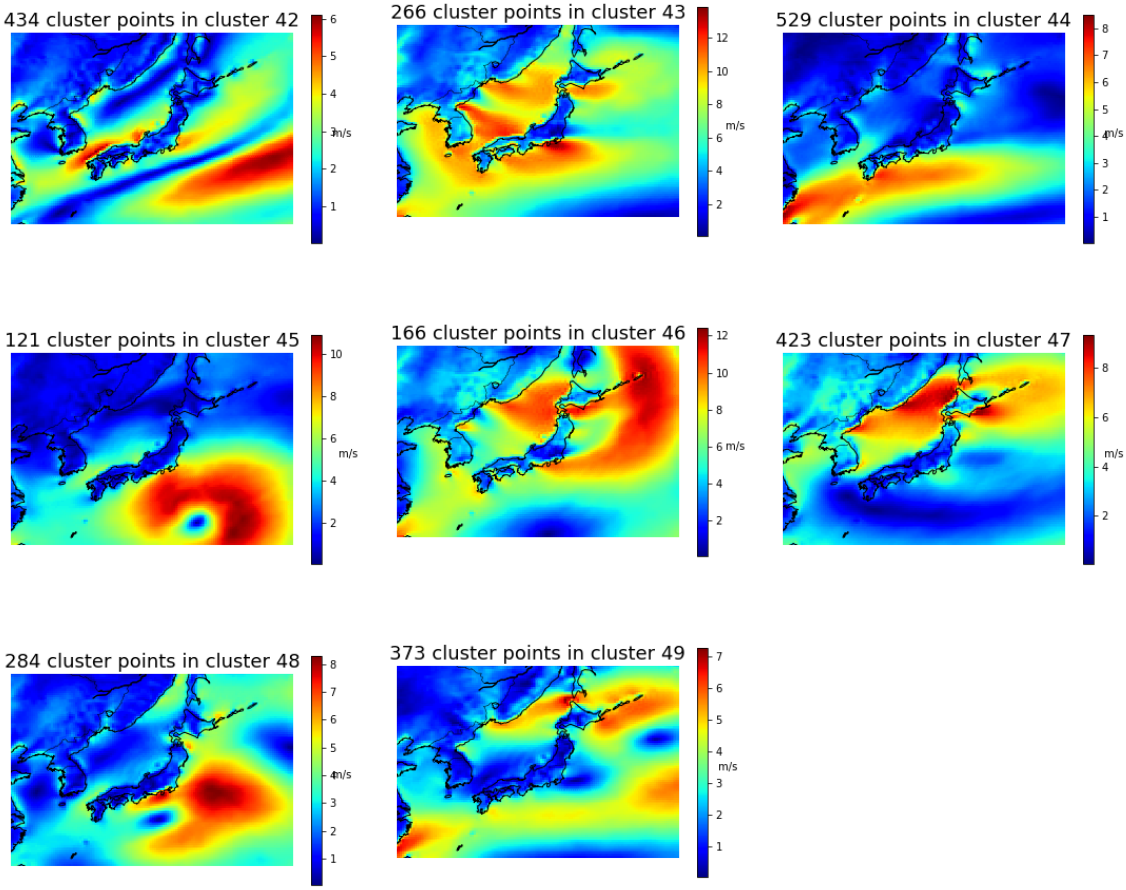
Dates in Cluster 38	JMA Typhoon	Dates in Cluster 38	JMA Typhoon	Dates in Cluster 38	JMA Typhoon
26/09/1979	7916	29/06/1992	9203	04/09/2005	0514
27/09/1979	7916	07/08/1992	9210	05/09/2005	0514
28/09/1979	7916	08/08/1993	9307	13/01/2006	NO
29/09/1979	7916	09/08/1993	9307	13/07/2007	0704
18/10/1979	7920	07/09/1993	9314	06/10/2009	0918
09/09/1980	8013	12/08/1994	9414	28/10/2010	1014
10/09/1980	8013	27/09/1994	9426	18/07/2011	1106
13/10/1980	8019	28/09/1994	9426	15/09/2011	1115
30/09/1981	8122	13/08/1996	9612	16/09/2011	1115
21/10/1981	8124	20/09/1996	9617	17/09/2011	1115
25/08/1982	8213	30/09/1996	9621	18/09/2011	1115
26/08/1982	8213	14/09/1997	9719	19/09/2011	1115
23/09/1982	8219	15/09/1997	9719	04/08/2012	1211
13/08/1983	8305	18/09/1998	9806	16/09/2012	1216
14/08/1983	8305	26/07/1999	9905	29/09/2012	1217
29/07/1984	8407	05/08/1999	9908	03/09/2013	1317
25/08/1984	NO	06/08/1999	9908	24/10/2013	1327
29/06/1985	8506	23/09/1999	9918	07/08/2014	1411
30/08/1985	8513	27/07/2000	0006	08/08/2014	1411
25/08/1986	8613	28/07/2000	0006	04/10/2014	1418
15/10/1987	8719	10/09/2000	0014	12/10/2014	1419
24/09/1988	8822	11/09/2000	0014	19/09/2016	1616
07/10/1988	8824	19/08/2001	0111	03/08/2017	1705
23/06/1989	8906	25/07/2002	0211	04/08/2017	1705
27/07/1989	8911	29/08/2002	0215	05/08/2017	1705
31/07/1989	8912	30/08/2002	0215	16/09/2017	1718
01/08/1989	8912	30/05/2003	0304	21/10/2017	1721
02/08/1989	8912	09/06/2003	NO	28/10/2017	1722
17/09/1990	9019	07/08/2003	0310	21/08/2018	1819
18/09/1990	9019	20/09/2003	0315	03/09/2018	1821
06/10/1990	9021	11/10/2003	NO	29/09/2018	1824
07/10/1990	9021	05/06/2004	NO		
29/11/1990	9028	20/06/2004	0406		
20/08/1991	9112	28/08/2004	0416		
21/08/1991	9112	29/08/2004	0416		
13/09/1991	9117	06/09/2004	0418		
18/09/1991	9118	28/09/2004	0421		
09/10/1991	9121	19/10/2004	0423		

B-6 Cluster Average Wind Speed 3DCNN Autoencoder




213 cluster points in cluster 12
278 cluster points in cluster 13
488 cluster points in cluster 14
394 cluster points in cluster 15
401 cluster points in cluster 16
274 cluster points in cluster 17
144 cluster points in cluster 18
225 cluster points in cluster 19
525 cluster points in cluster 20
200 cluster points in cluster 21
181 cluster points in cluster 22
148 cluster points in cluster 23
338 cluster points in cluster 24
359 cluster points in cluster 25
382 cluster points in cluster 26
m/s


460 cluster points in cluster 27
176 cluster points in cluster 28
328 cluster points in cluster 29
176 cluster points in cluster 30
189 cluster points in cluster 31
283 cluster points in cluster 32
341 cluster points in cluster 33
261 cluster points in cluster 34
404 cluster points in cluster 35
314 cluster points in cluster 36
120 cluster points in cluster 37
107 cluster points in cluster 38
242 cluster points in cluster 39
259 cluster points in cluster 40
169 cluster points in cluster 41
m/s


434 cluster points in cluster 42
266 cluster points in cluster 43
529 cluster points in cluster 44
121 cluster points in cluster 45
166 cluster points in cluster 46
423 cluster points in cluster 47
284 cluster points in cluster 48
373 cluster points in cluster 49
m/s

Appendix C

Case Study Images

C-1 2D CNN Autoencoder

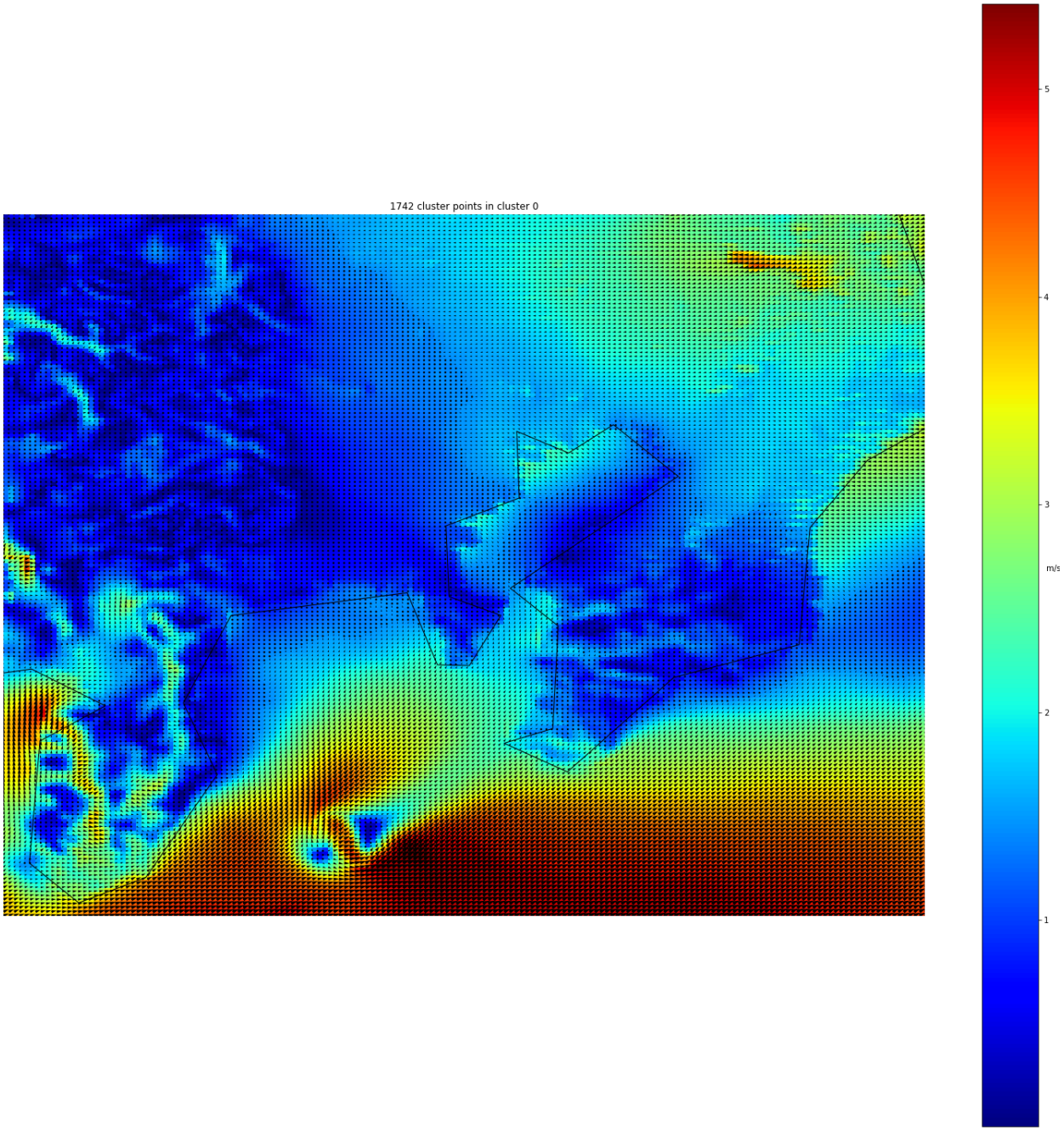


Figure C-1: Average wind speed of cluster 0 using the 2D CNN Autoencoder

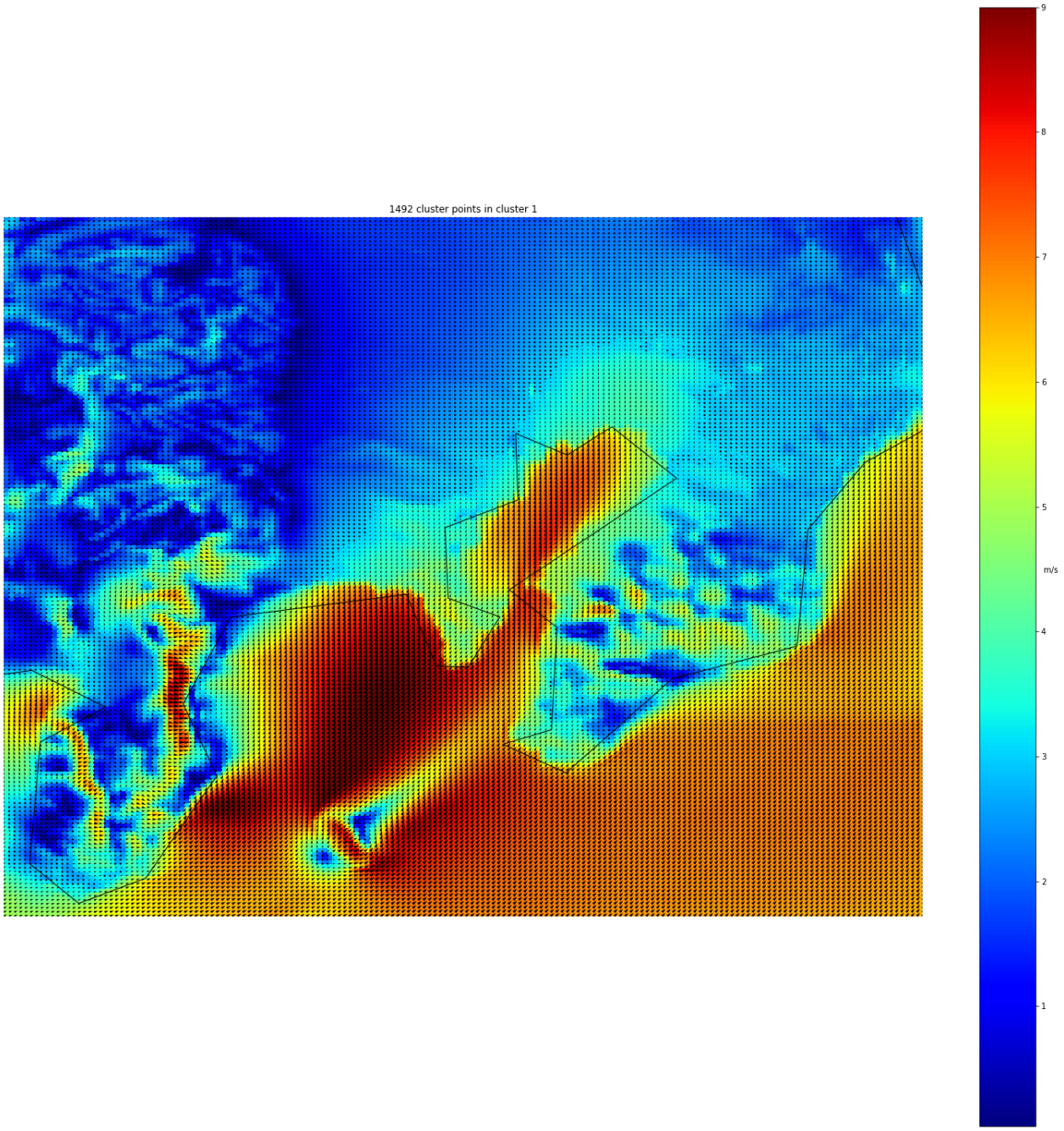


Figure C-2: Average wind speed of cluster 1 using the 2D CNN Autoencoder

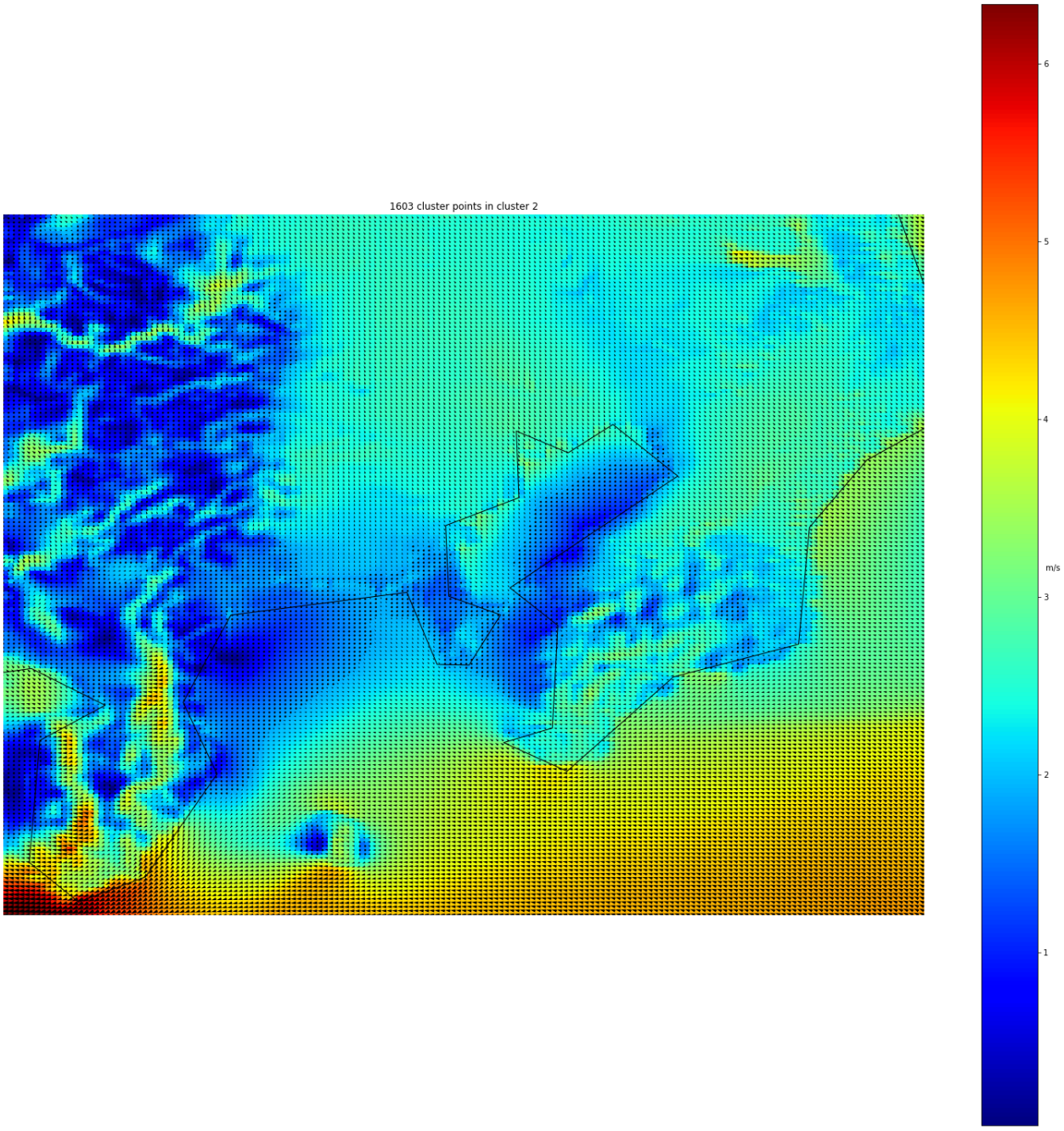


Figure C-3: Average wind speed of cluster 2 using the 2D CNN Autoencoder

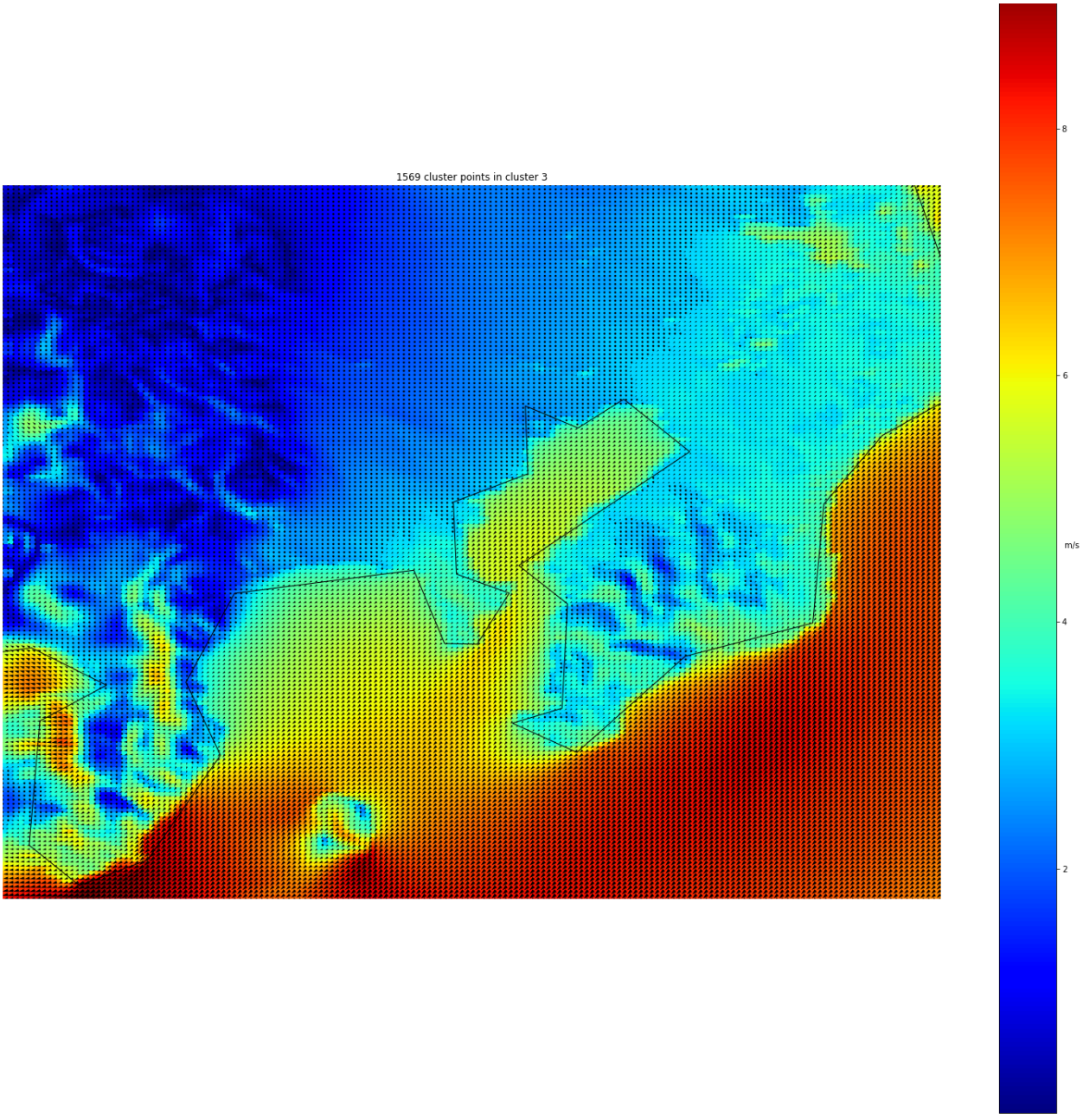


Figure C-4: Average wind speed of cluster 3 using the 2D CNN Autoencoder

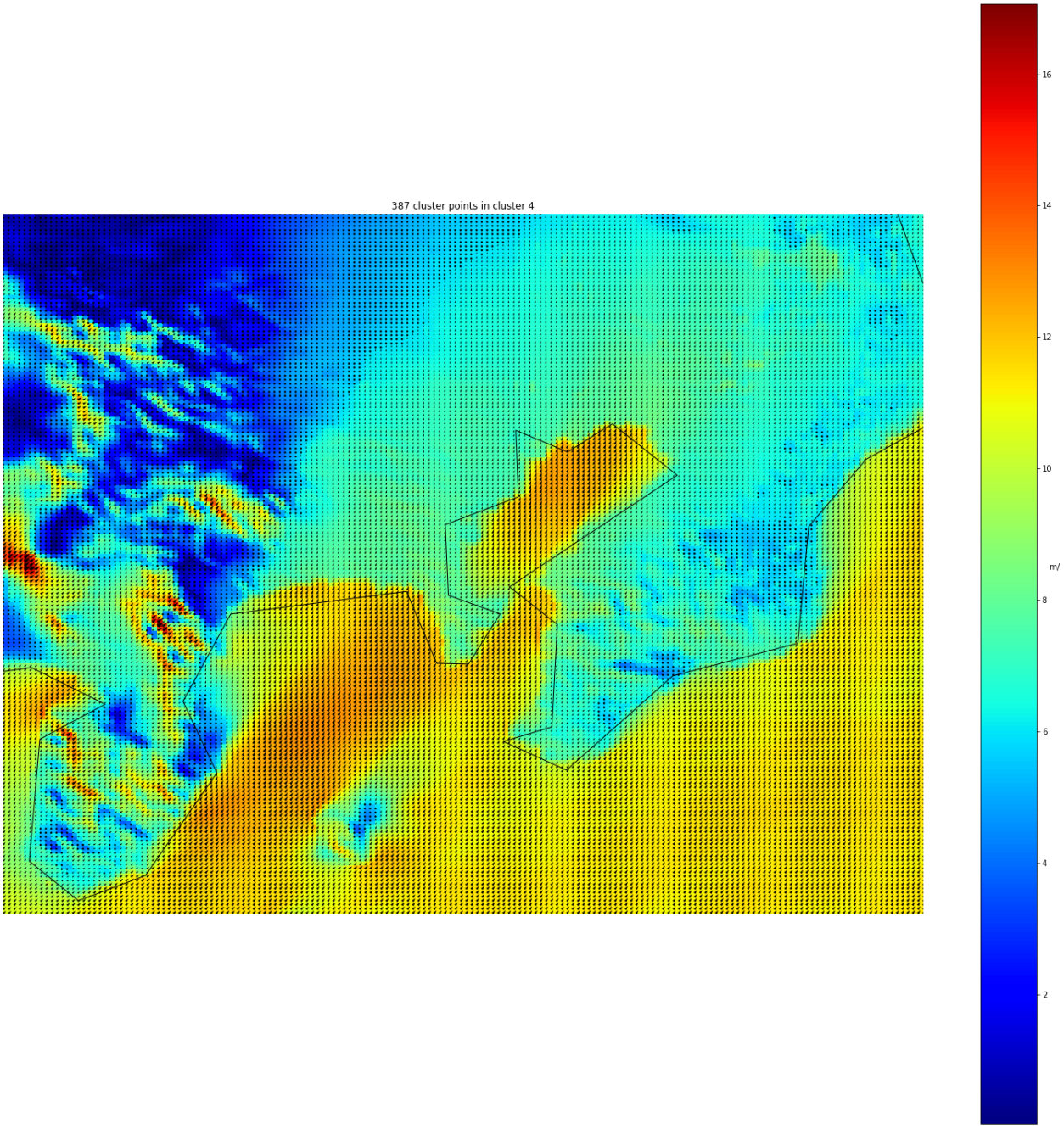


Figure C-5: Average wind speed of cluster 4 using the 2D CNN Autoencoder

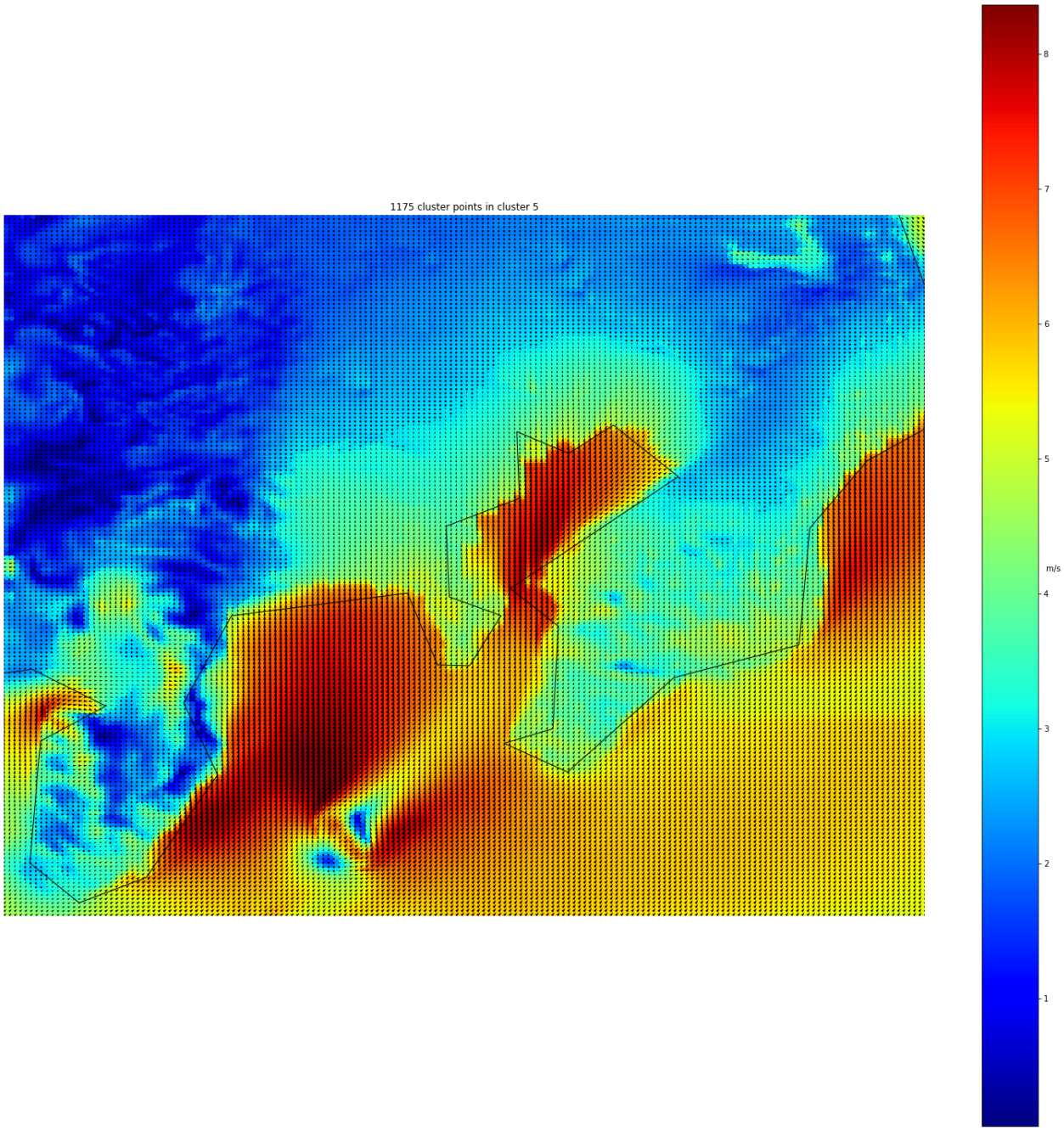


Figure C-6: Average wind speed of cluster 5 using the 2D CNN Autoencoder

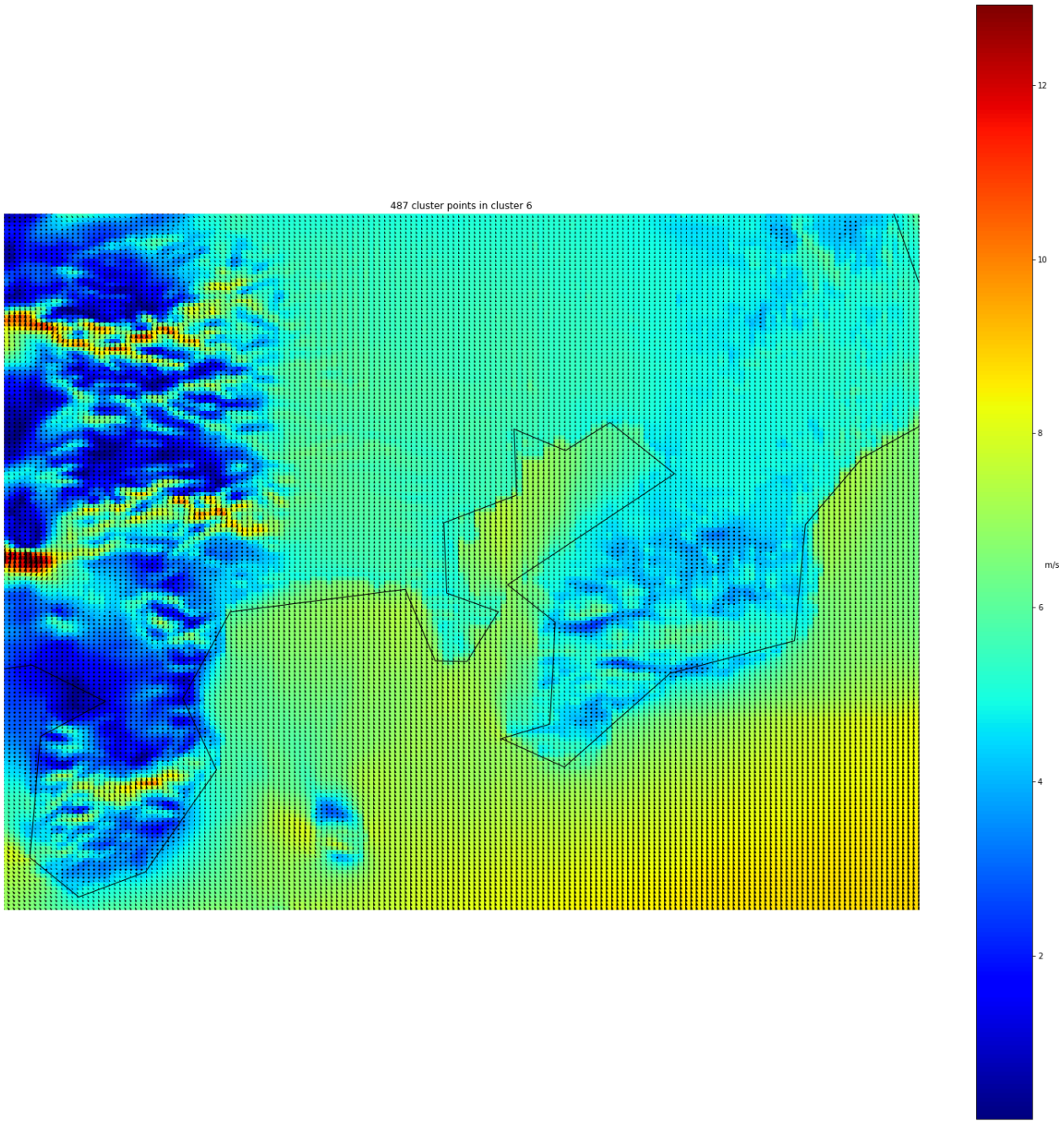


Figure C-7: Average wind speed of cluster 6 using the 2D CNN Autoencoder

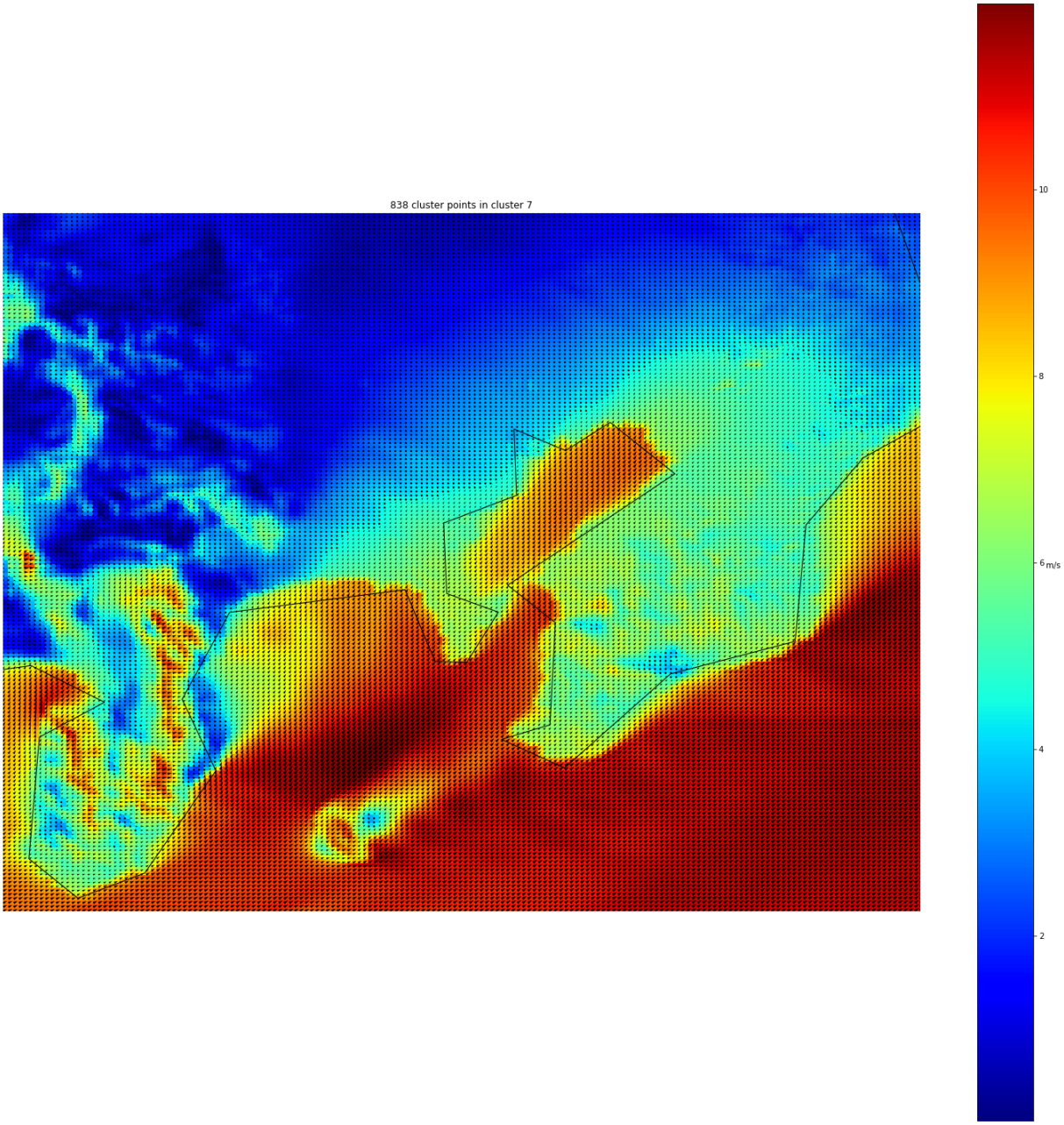


Figure C-8: Average wind speed of cluster 7 using the 2D CNN Autoencoder

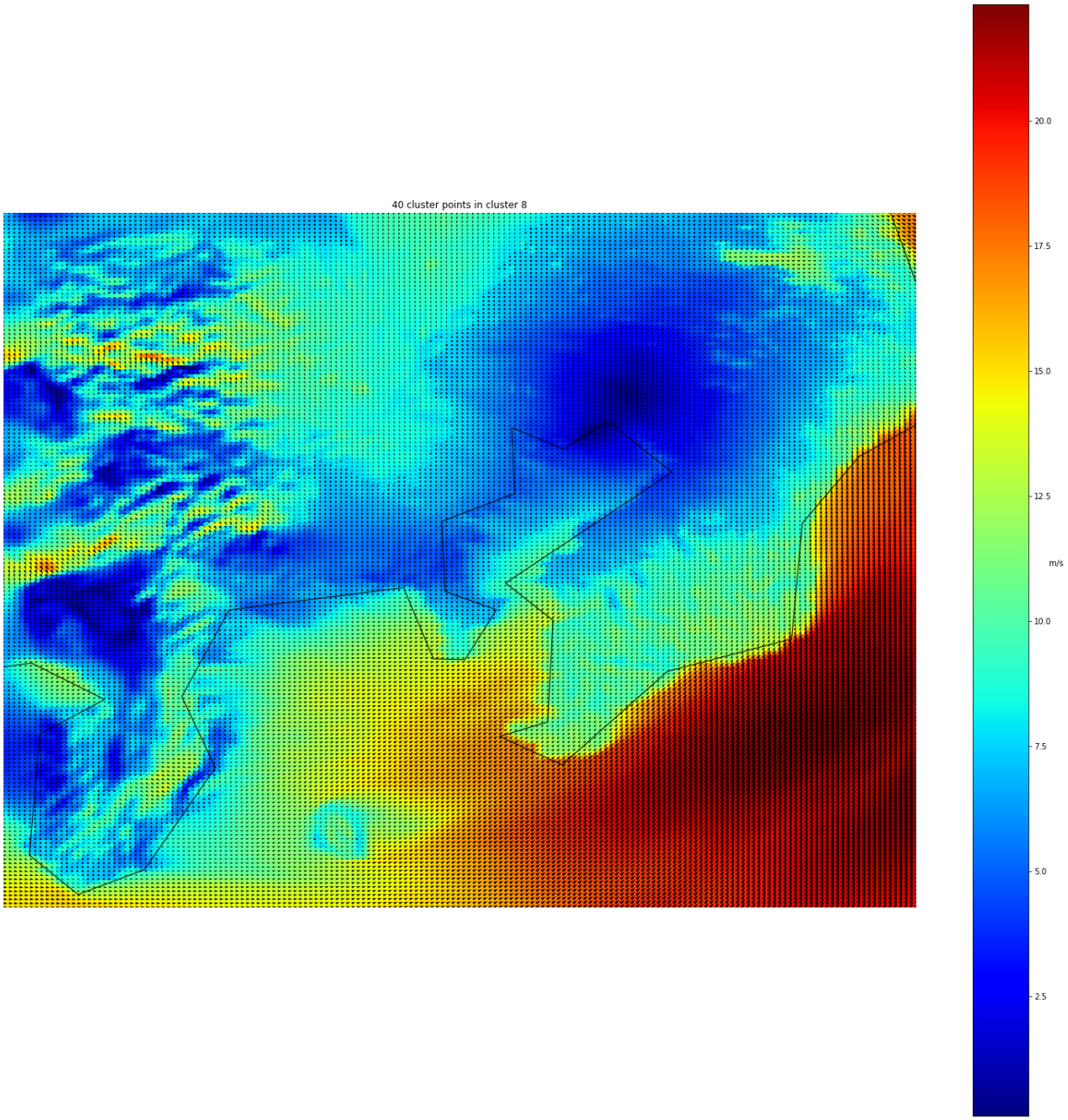


Figure C-9: Average wind speed of cluster 8 using the 2D CNN Autoencoder

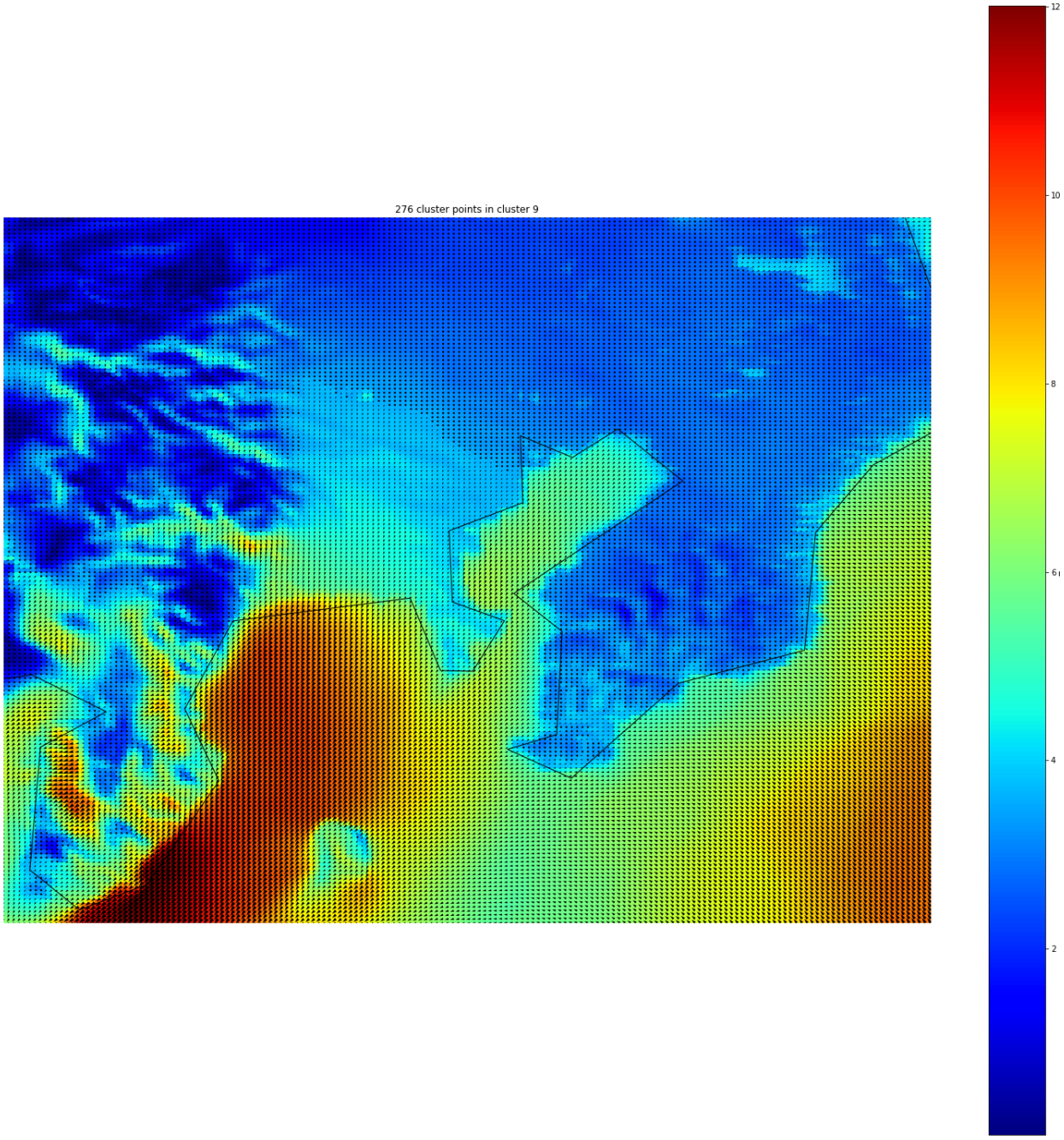


Figure C-10: Average wind speed of cluster 9 using the 2D CNN Autoencoder

C-2 3D CNN Autoencoder

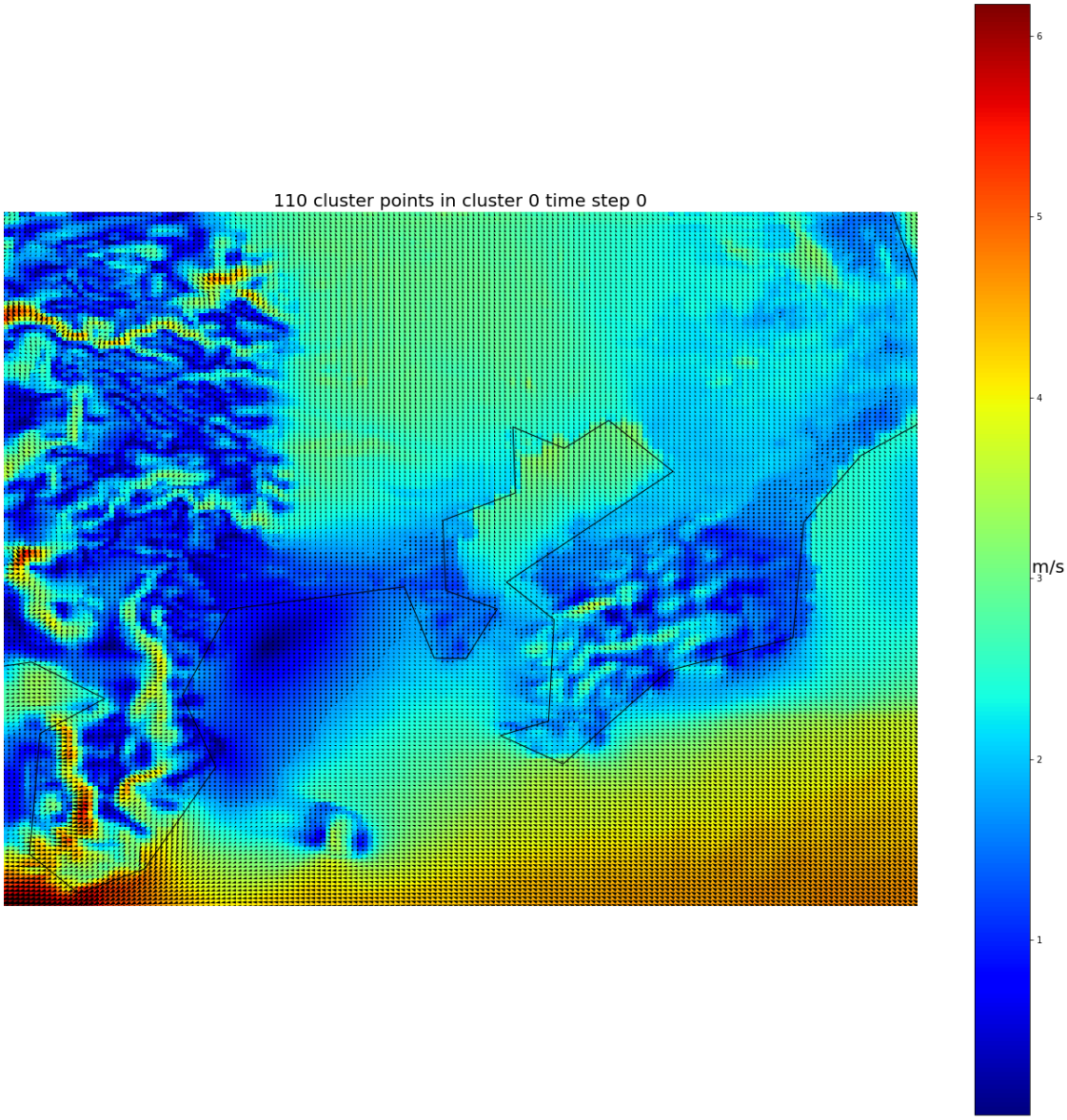


Figure C-11: Average wind speed of cluster 0 using the 3D CNN Autoencoder

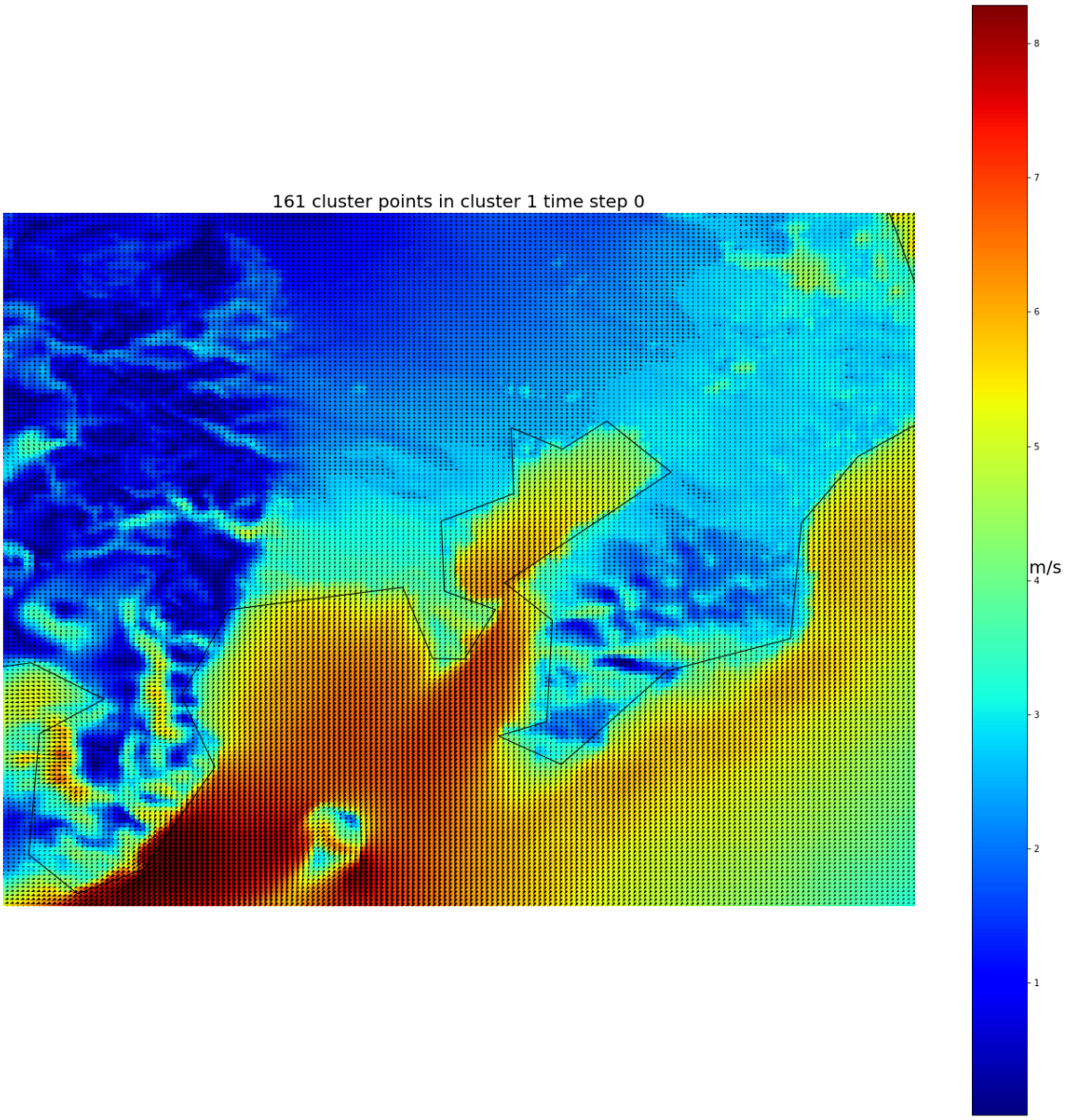


Figure C-12: Average wind speed of cluster 1 using the 3D CNN Autoencoder

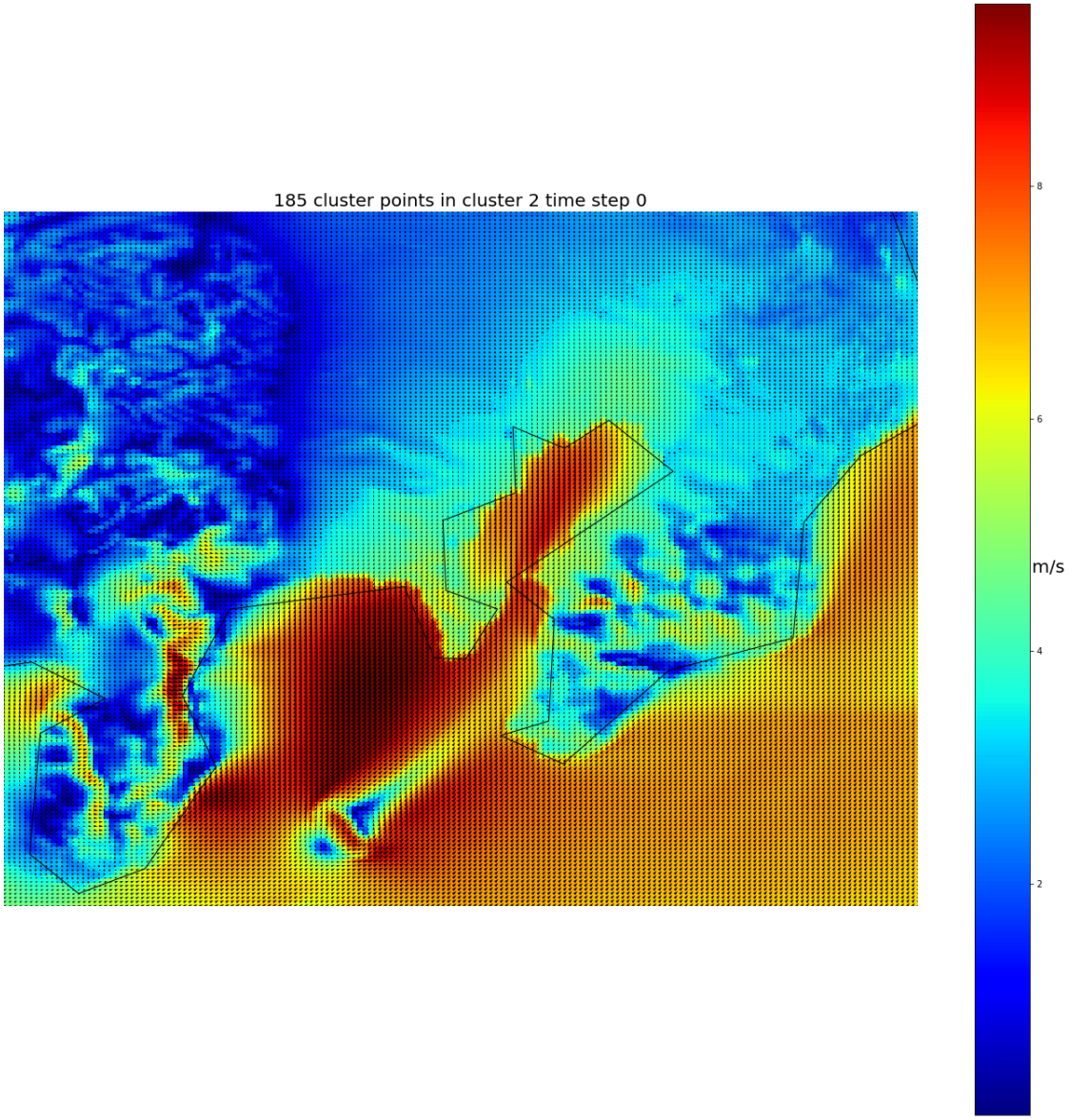


Figure C-13: Average wind speed of cluster 2 using the 3D CNN Autoencoder

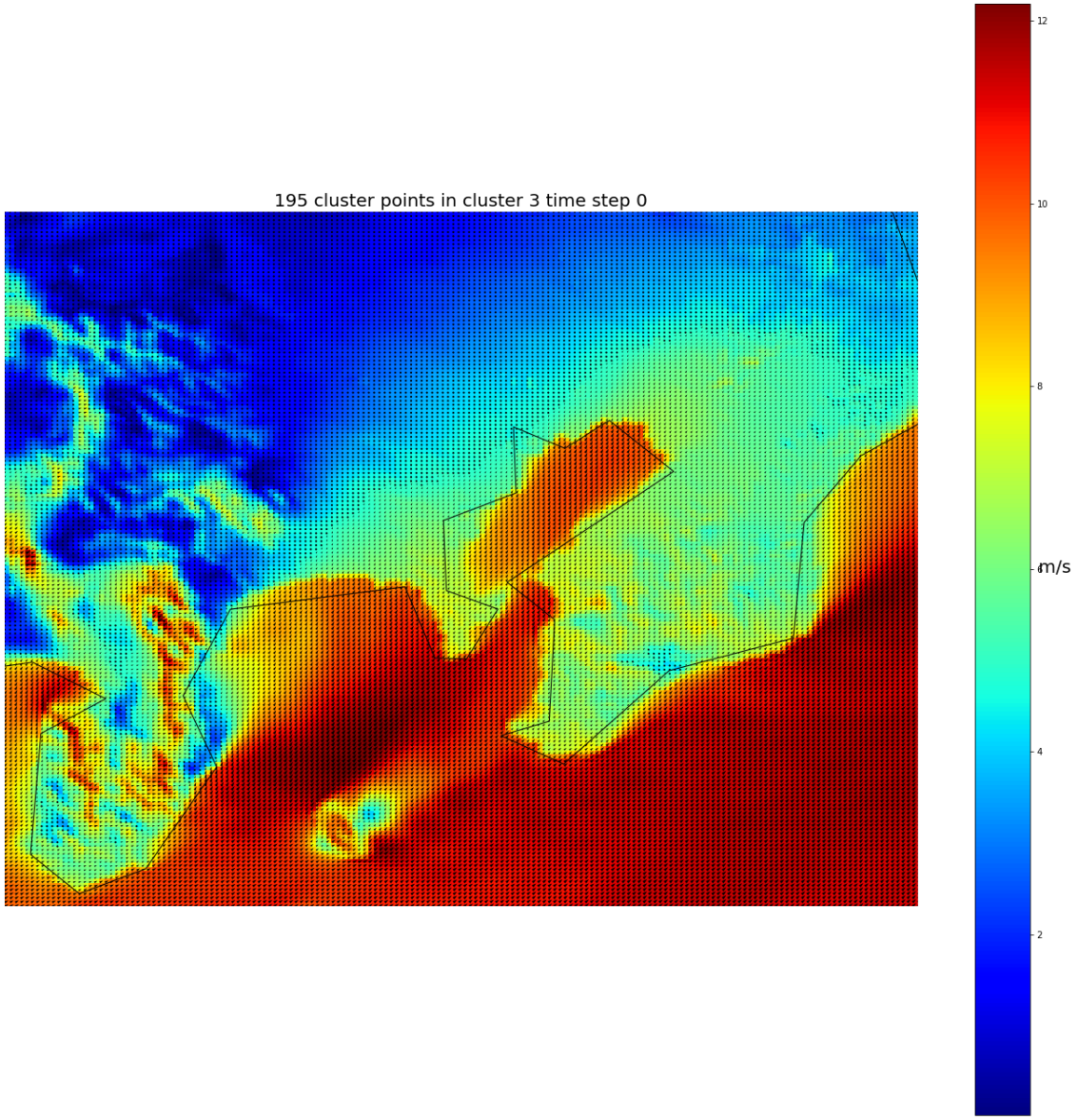


Figure C-14: Average wind speed of cluster 3 using the 3D CNN Autoencoder

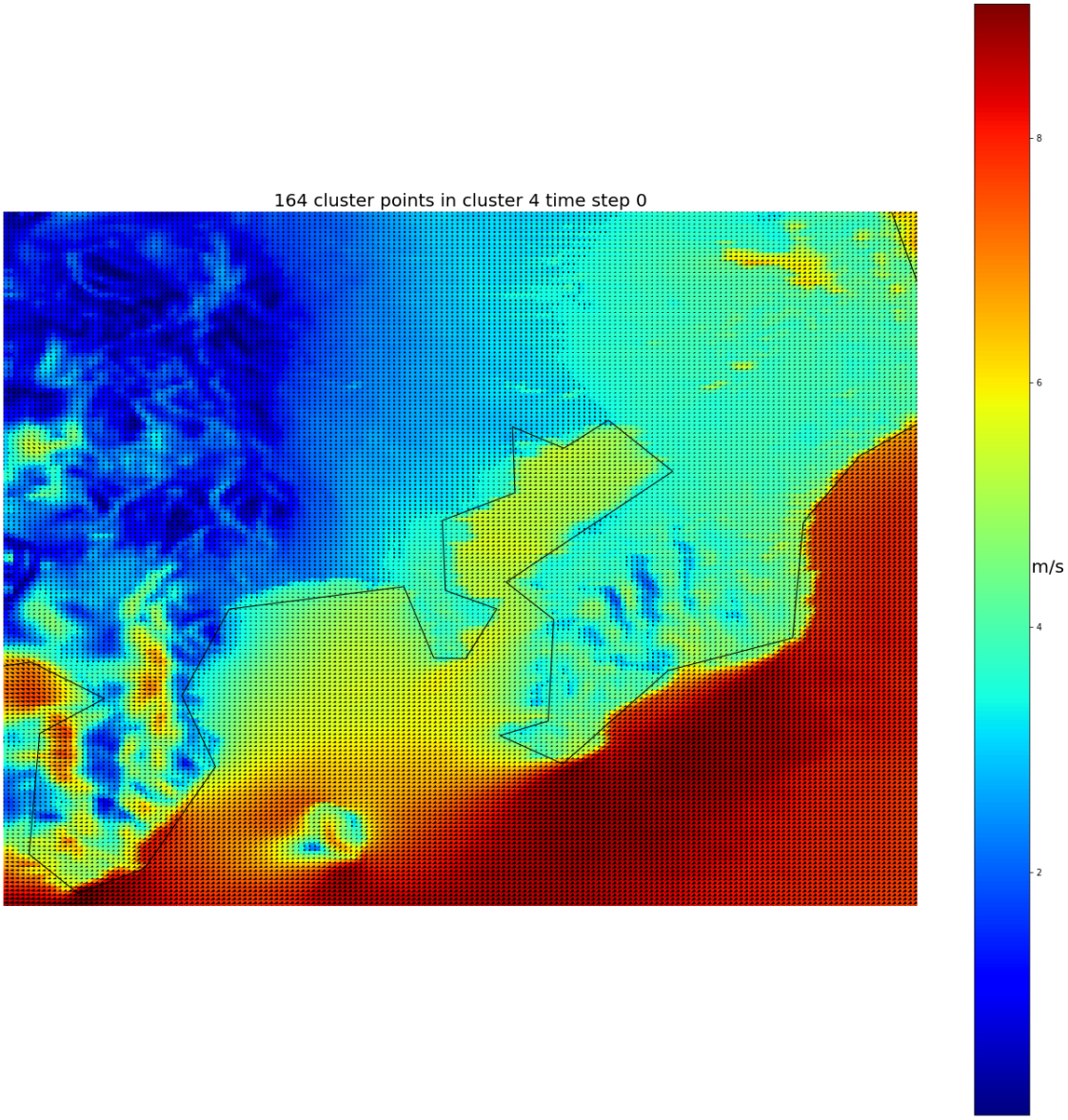


Figure C-15: Average wind speed of cluster 4 using the 3D CNN Autoencoder

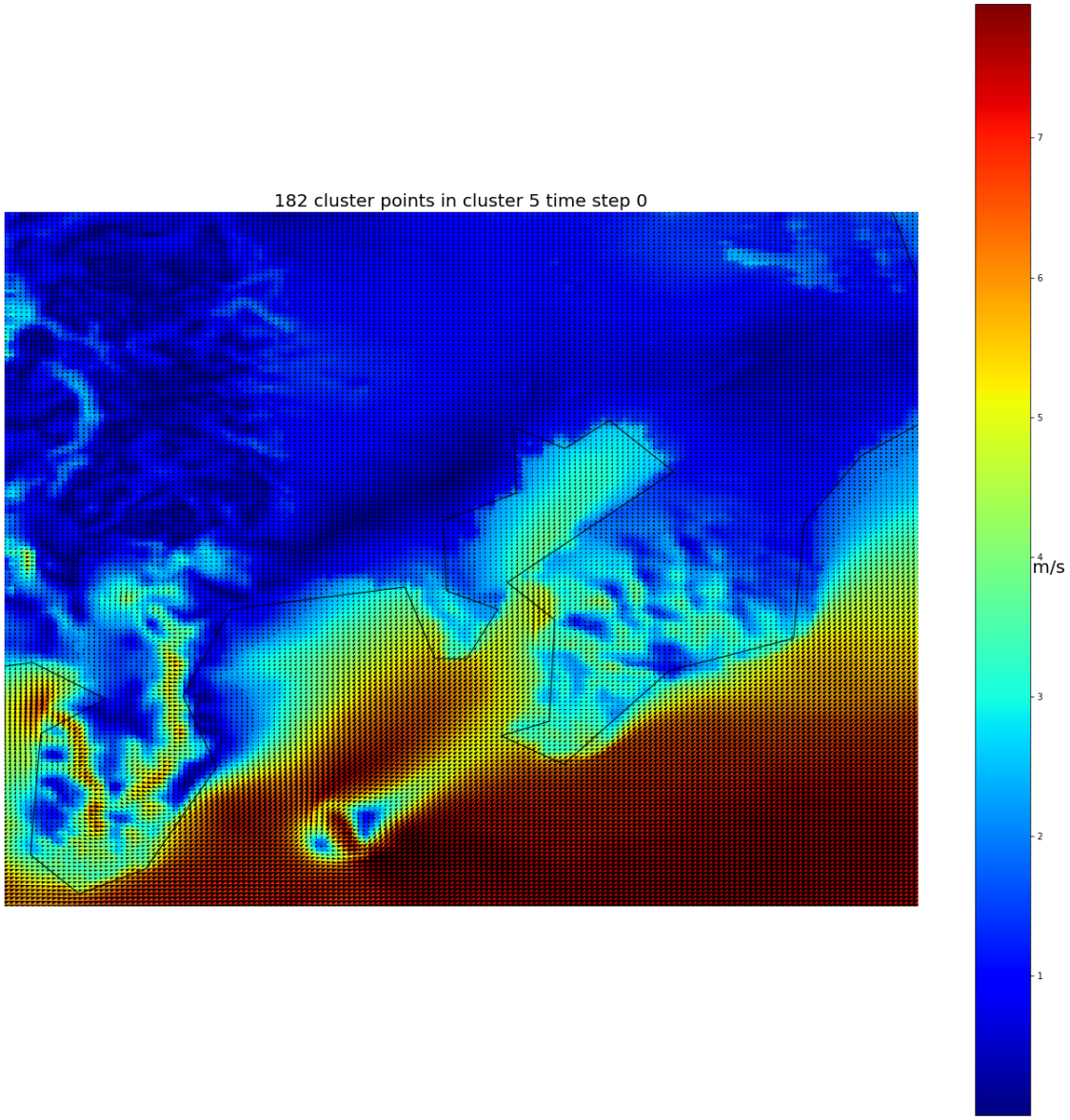


Figure C-16: Average wind speed of cluster 5 using the 3D CNN Autoencoder

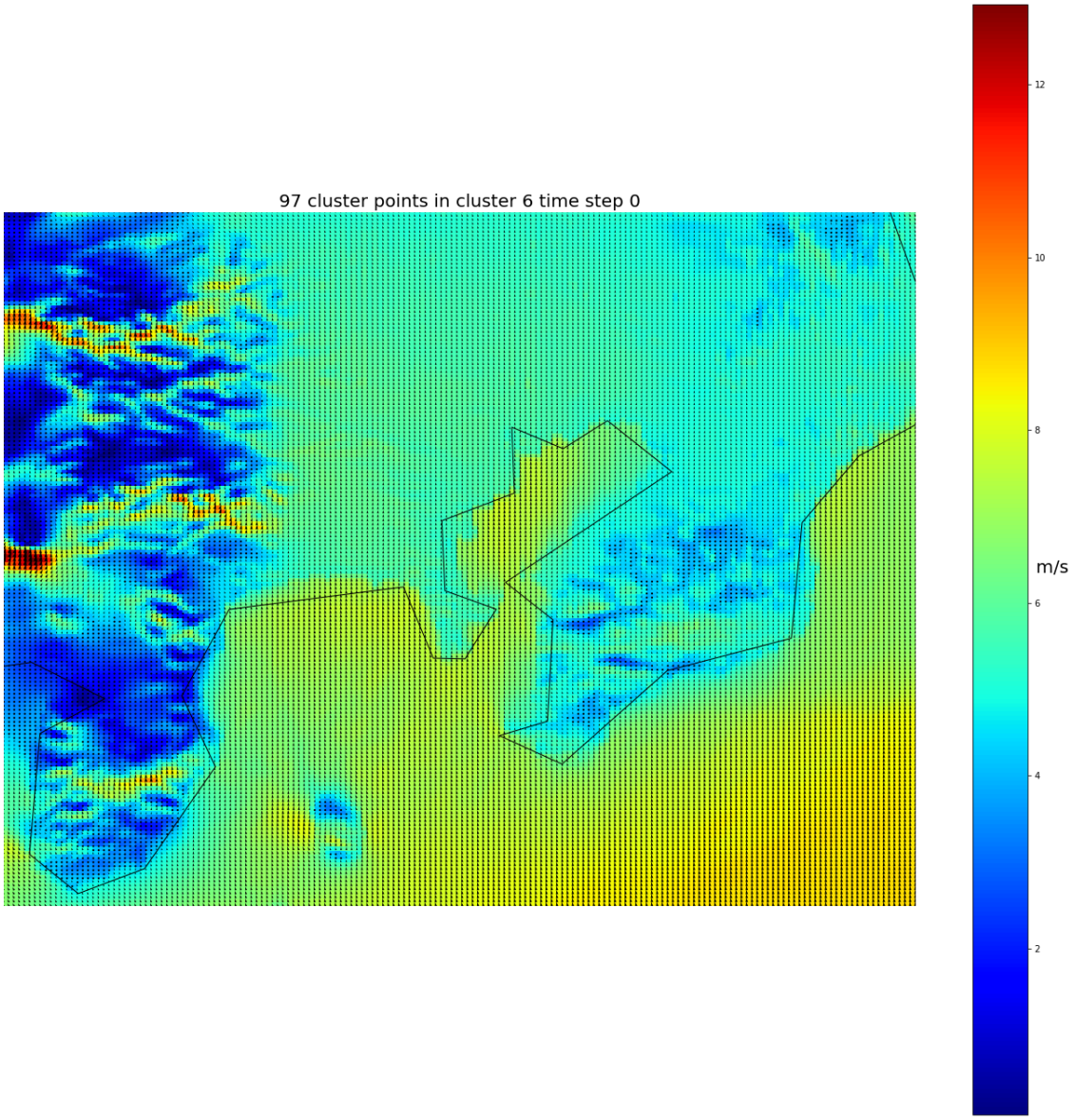


Figure C-17: Average wind speed of cluster 6 using the 3D CNN Autoencoder

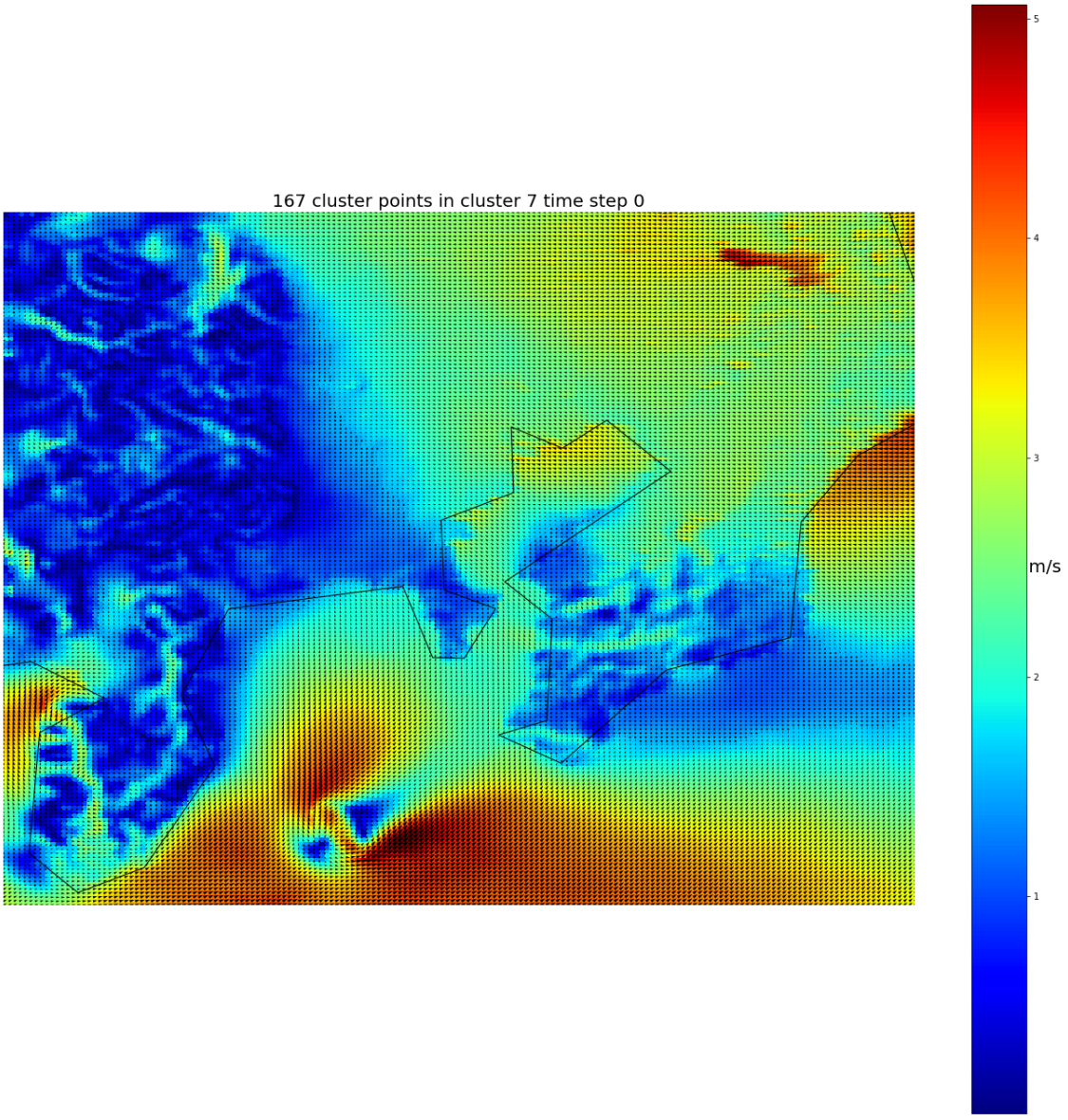


Figure C-18: Average wind speed of cluster 7 using the 3D CNN Autoencoder

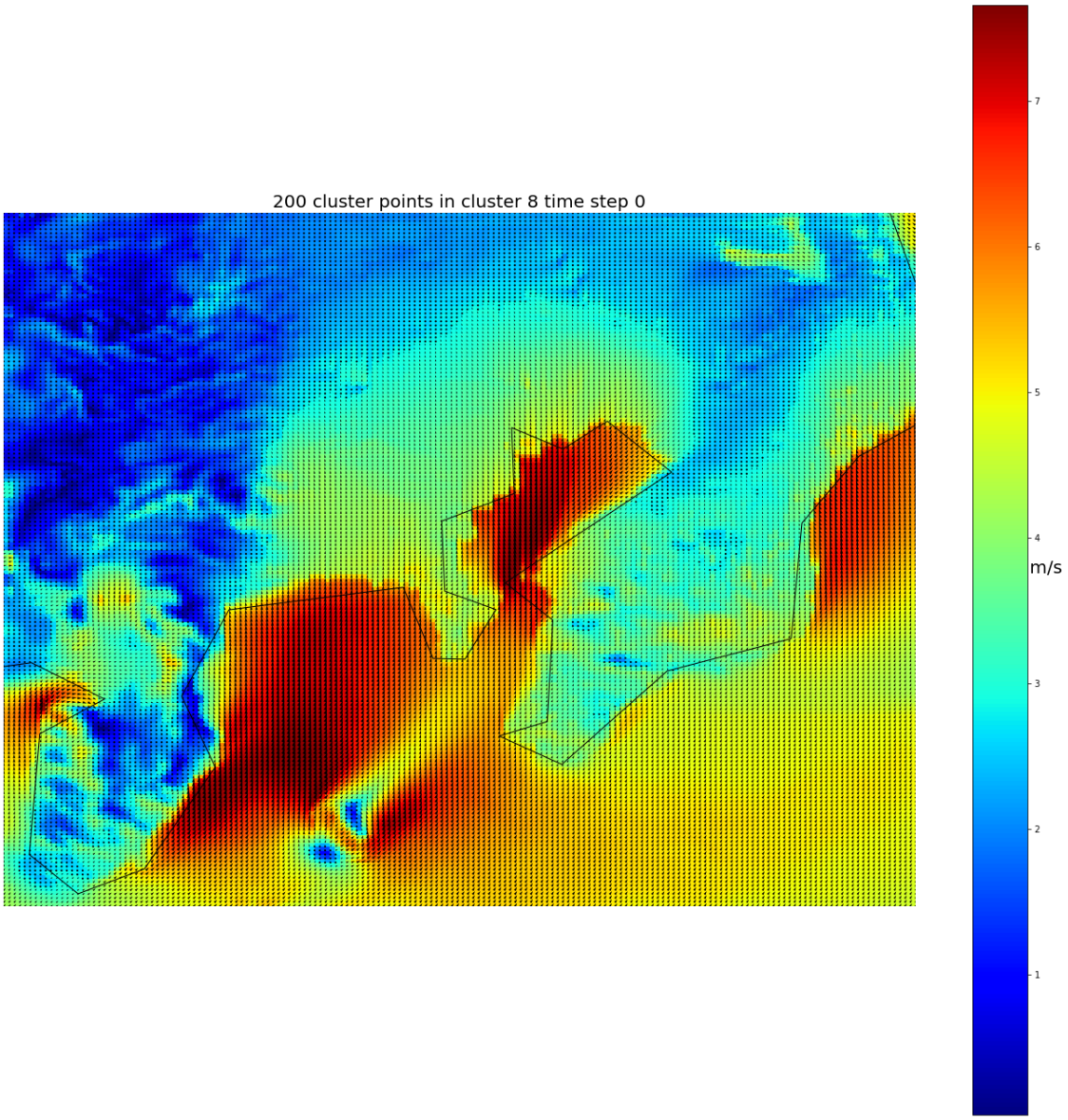


Figure C-19: Average wind speed of cluster 8 using the 3D CNN Autoencoder

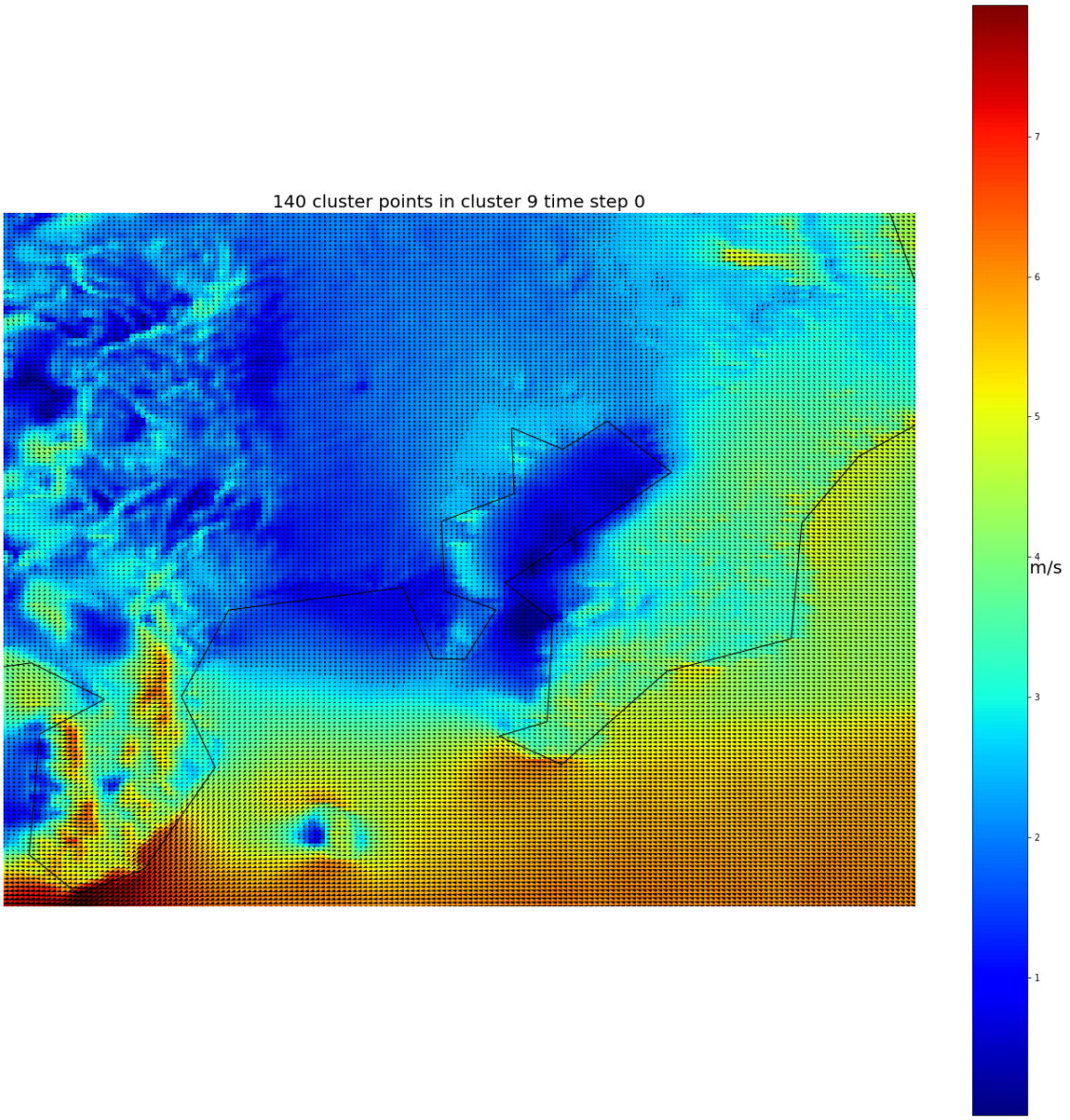


Figure C-20: Average wind speed of cluster 9 using the 3D CNN Autoencoder

Appendix D

Python Codes for Data Analysis

D-1 Imports and Data Preparation

D-1-1 Imports

```
# Import all the necessities

from keras.layers import Input, Dropout, Dense, Conv2D, Conv3D,
    Conv3DTranspose, MaxPooling3D, UpSampling3D, MaxPooling2D,
    UpSampling2D, InputLayer, Flatten, Reshape, Conv2DTranspose,
    AveragePooling2D, AveragePooling3D
from keras.models import Model, Sequential
from keras import backend as K
from keras.callbacks import TensorBoard
from keras.models import load_model
from keras.models import model_from_json

import netCDF4
import math
import matplotlib.pyplot as plt
import matplotlib
import matplotlib.gridspec as gridspec
import shapefile as shp
import seaborn as sns
import geopandas
import geoplot

from sklearn import metrics
from sklearn.cluster import KMeans, MiniBatchKMeans

from netCDF4 import num2date, date2num, date2index, MFDataset
import numpy as np

```

```
from mpl_toolkits.basemap import Basemap
import pandas as pd

from platform import python_version
print("Python version: "+python_version())
```

D-1-2 Data from Netcdf file

```
# Data
# Create vectors from all the .nc information

# Data is 2000-2018 data at 11 AM
data = netCDF4.Dataset('SFC_2000_2018_11.nc','r')

time_all = data.variables['time'][:]
lon = data.variables['longitude'][:]
lat = data.variables['latitude'][:]
u10 = data.variables['u10'][:]
v10 = data.variables['v10'][:]

# Date reconstruction from time vector
dates = num2date(time_all,data.variables['time'].units)

#To make the data (100,140) input
lon_cut=lon[1:]
lat_cut=lat[1:]
u10_cut = u10[:,1:u10.shape[1],1:u10.shape[2]]
v10_cut = v10[:,1:v10.shape[1],1:v10.shape[2]]

#Reshape for the use of CNN autoencoder (needs channels last)
u10_cut = u10_cut.reshape(u10_cut.shape[0], u10_cut.shape[1], u10_cut.
    shape[2], 1)

#Define training and testing data
lim = math.ceil(0.85*u10_cut.shape[0])

#Define amount of data that is train and test data
x_train = u10_cut[0:lim,:,:,:] #use the first 85% of the data for
    training
x_test = u10_cut[lim:,:,:,:] #use the last 15% of the data for testing
```

D-1-3 Make Basemap for Country Plots

```
fig=plt.figure()
ax=fig.add_axes([0.1,0.1,0.8,0.8])
# setup mercator map projection.
m = Basemap(llcrnrlon=np.amin(lon_cut),llcrnrlat=np.amin(lat_cut),
    urcrnrlon=np.amax(lon_cut),urcrnrlat=np.amax(lat_cut),\
            rsphere=(6378137.00,6356752.3142),\
            resolution='l',projection='gall')


```

```
x = np.linspace(0, m.urcrnrx, lon_cut.shape[0])
y = np.linspace(0, m.urcrnry, lat_cut.shape[0])[::-1] #Otherwise the plot is upside down

xx, yy = np.meshgrid(x,y)

m.drawcoastlines()
# draw borders
m.drawcountries()
# draw parallels
m.drawparallels(np.arange(30,90,15),labels=[1,1,0,1])
# draw meridians
m.drawmeridians(np.arange(-180,180,10),labels=[1,1,0,1])

plt.savefig('Large_Domain_nocolor.jpg')
plt.show()
```

D-2 Clustering

```
n_clus=50
kmeans = KMeans(n_clusters = n_clus)
clustered_data = kmeans.fit_predict(input_data)
```

D-3 Inertia Plot

```
# Calculate inertia for different cluster sizes

# fitting multiple k-means algorithms and storing the values in an empty list
SSE = []
for cluster in range(1,250):
    kmeans = KMeans(n_jobs = -1, n_clusters = cluster)
    kmeans.fit(R_dim)
    SSE.append(kmeans.inertia_)

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(1,250), 'SSE':SSE})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['SSE'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

D-4 Typhoon Jebi Comparison

```
#Find how many images in 1 cluster

amount_in_cluster=[0]*n_clus
for x in range(len(clustered_data)):
    for i in range(n_clus):
        if clustered_data[x]==i:
```

```
7                amount_in_cluster[i]=amount_in_cluster[i]+1
8
9   print(amount_in_cluster)
10
11  # Typhoon Jebi: 3rd of september
12  print(dates[6820])
13  clustered_data[6820]
14
15  c_test = clustered_data[6820]
16  vec_test = []
17
18  for n in range(len(clustered_data)):
19      if clustered_data[n] == c_test:
20          vec_test.append(n)      #Whenever a day is cluster c_test, it is
                appended in the empty vec_test list.
21
22  print(vec_test)
23
24  #Print the first 10 cases that are contained in the same cluster as Jebi
25  for n in vec_test[0:10]:
26      cs = plt.pcolor(xx, yy, u10_cut_reshape[n], cmap = 'jet')
27      cbar=plt.colorbar(cs)
28      cbar.set_label(' m/s',rotation=0)
29
30      m.drawcoastlines()
31      m.drawcountries()
32      plt.axis('off')
33      plt.title("Data point %i: %s in cluster %i" %(n, dates[n],
            clustered_data[n]))
34      plt.savefig("%04d.jpg" %n)
35      plt.show()
36
37
38  #Compare with typhoon Jebi
39  plt.title("Data point %i: %s \n %i points in cluster %i" %(6820, dates
        [6820], amount_in_cluster[clustered_data[6820]], clustered_data[6820])
        )
40  m.drawcoastlines()
41  m.drawcountries()
42  plt.axis('off')
43  cs = plt.pcolor(xx,yy,u10_cut_reshape[6820], cmap='jet')
44  cbar=plt.colorbar(cs)
45  cbar.set_label(' m/s',rotation=0)
46
47  plt.savefig("Jebi.jpg")
48  plt.show()
```

D-5 Python Codes for Autoencoders with u10 as input

D-5-1 No Autoencoder

```
#K-means needs at most 2 dimensions. Flatten u10 input to cluster
u10_flat = u10_cut.reshape(u10_cut.shape[0], u10_cut.shape[1]* u10_cut.
    shape[2])

#Compute K-means algorithm
n_clus = 50
kmeans = KMeans(n_clusters = n_clus)
clustered_data = kmeans.fit_predict(u10_flat)
```

D-5-2 Single layered autoencoder

```
# Autoencoder
# First try just the u10 and v10 data seperately
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],1))

#Encoder
x = Flatten()(input_img)
encoded = Dense(100, activation=None)(x)

#Decoder
x = Dense(14000, activation = None)(encoded)
decoded = Reshape((100,140,1))(x)

autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics=['accuracy'])
print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-5-3 Deep autoencoder

```
# Deep Autoencoder
# First try just the u10 and v10 data seperately
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],1))

#Encoder
x = Flatten()(input_img)
x = Dense(1000, activation = 'relu')(x)
x = Dense(500, activation = 'relu')(x)
encoded = Dense(100, activation=None)(x)

#Decoder

x = Dense(500, activation = 'relu')(encoded)
x = Dense(1000, activation = 'relu')(x)
x = Dense(14000, activation = None)(x)
```

```
decoded = Reshape((100,140,1))(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics=['accuracy'])

print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-5-4 CNN Autoencoder

```
#CNN Autoencoder

# First try just the u10 and v10 data seperately
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],1))

#Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
encoded = Dense(100, activation='linear')(x)

#Decoder

x = Dense(14000, activation = 'linear')(encoded)
x = Reshape((100,140,1))(x)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
decoded = Conv2DTranspose(1, (3, 3), activation = 'linear', padding='same
    ')(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])


print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-5-5 Deep CNN Autoencoder

```
#Deep CNN Autoencoder

# First try just the u10 and v10 data seperately
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],1))


#Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
x = Dense(500, activation='relu')(x)
x = Dense(250, activation='relu')(x)
encoded = Dense(100, activation='linear')(x)

#Decoder
x = Dense(250, activation= 'relu')(encoded)
x = Dense(500, activation= 'relu')(x)
x = Dense(14000, activation = 'linear')(x)
x = Reshape((100,140,1))(x2)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
decoded = Conv2DTranspose(1, (3, 3), activation = 'linear', padding='same
    ')(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-5-6 CNN Autoencoder with Max Pooling Layer

```
#CNN Autoencoder
# First try just the u10 and v10 data seperately
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],1))

#Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
    data_format=None)(x)
x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
```

```
x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
    data_format=None)(x)
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
encoded = Dense(100, activation='linear')(x)

#Decoder
x = Dense(875, activation = 'linear')(encoded)
x = Reshape((25,35,1))(x)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(
    x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(
    x)
decoded = Conv2DTranspose(1, (3, 3), activation = 'linear', padding='same
    ')(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-5-7 Deep CNN Autoencoder with Max Pooling Layer

```
# Deep CNN Autoencoder
# First try just the u10 and v10 data seperately
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],1))

#Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
    data_format=None)(x)
x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
    data_format=None)(x)
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
x = Dense(500, activation='relu')(x)
x = Dense(250, activation='relu')(x)
encoded = Dense(100, activation='linear')(x)

#Decoder

```

```
x = Dense(250, activation= 'relu')(encoded)
x = Dense(500, activation= 'relu')(x)
x = Dense(875, activation = 'linear')(x)
x = Reshape((25,35,1))(x)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(x)
decoded = Conv2DTranspose(1, (3, 3), activation = 'linear', padding='same')(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error', metrics = ['accuracy'])

print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-5-8 Image Construction for u10 as Input

```
# Make figure
#amount of images
amount = 2

#neuron value
encoded_imgs = encoder.predict(x_test)

#decoded picture
decoded_imgs = autoencoder.predict(x_test)

# Make subplot
for z in range(amount):
    f = plt.figure()
    f.set_figheight(10)
    f.set_figwidth(20)
    widths = [10, 2, 10]
    heights = [1]
    gs = f.add_gridspec(ncols=3, nrows=1, width_ratios = widths, height_ratios=heights)

#Original image
    f1 = f.add_subplot(gs[0,0])
    f1.set_title('Original', fontsize=20)
```

```
23      im1 = plt.pcolor(xx,yy,x_test[z].reshape(u10_cut.shape[1], u10_cut.
            shape[2]), cmap = 'jet')
24
25      # Plot wind speed between -15 and 15 m/s
26      plt.clim(-15,15)
27
28      # draw borders
29      m.drawcoastlines()
30      m.drawcountries()
31
32  #Neuron representation
33      f2 = f.add_subplot(gs[0,1])
34      f2.set_title('Code', fontsize=20)
35      im2 = plt.imshow(encoded_imgs[z].reshape([encoded_imgs.shape
            [-1]//4,-1]), cmap = 'jet')
36      im2.axes.get_xaxis().set_visible(False)
37      im2.axes.get_yaxis().set_visible(False)
38      plt.clim(-15,15)
39      cbar = plt.colorbar(im2)
40      cbar.set_label(' m/s', rotation=0, fontsize=18)
41
42  #Decoded image
43      f3 = f.add_subplot(gs[0,2])
44      f3.set_title('Decoded', fontsize=20)
45      im3 = plt.pcolor(xx,yy,decoded_imgs[z].reshape(u10_cut.shape[1],
            u10_cut.shape[2]), cmap = 'jet')
46
47      # Plot wind speed between -15 and 15 m/s
48      plt.clim(-15,15)
49
50      # draw borders
51      m.drawcoastlines()
52      m.drawcountries()
53
54
55  plt.savefig('single_autoencoder_performance_example.jpg')
56
57
58  #Find neuron value for entire input
59  R_dim = encoder.predict(u10_cut)
```

D-5-9 Image Construction for u10 and v10 as Input

```
1  # Print the X_test data and the output of the autoencoder respectively.
2  #neuron value
3  encoded_imgs = encoder.predict(x_test)
4
5  #decoded picture
6  decoded_imgs = autoencoder.predict(x_test)
7
8  z = 1 # data point
9
10 #U before autoencoder
```

```
cs = plt.pcolor(xx,yy,x_test[z,:,:,0], cmap = 'jet')
plt.clim(-15,15)
m.drawcoastlines()
m.drawcountries()
plt.savefig('u10v10_2DCNN_nomax_autoencoder_performance_example1.jpg')
plt.show()

#V before autoencoder
cs = plt.pcolor(xx,yy,x_test[z,:,:,1], cmap = 'jet')
plt.clim(-15,15)
m.drawcoastlines()
m.drawcountries()
plt.savefig('u10v10_2DCNN_nomax_autoencoder_performance_example2.jpg')
plt.show()

#Encoder
cs = plt.imshow(encoded_imgs[z].reshape([encoded_imgs.shape[-1]//5,-1]),
    cmap = 'jet')
plt.clim(-15,15)
plt.axis('off')
plt.colorbar(cs)
plt.savefig('u10v10_2DCNN_nomax_autoencoder_performance_example3.jpg')
plt.show()

#U After autoencoder
cs = plt.pcolor(xx,yy,decoded_imgs[z,:,:,0], cmap = 'jet')
plt.clim(-15,15)
m.drawcoastlines()
m.drawcountries()
plt.savefig('u10v10_2DCNN_nomax_autoencoder_performance_example4.jpg')
plt.show()

#V After autoencoder
cs = plt.pcolor(xx,yy,decoded_imgs[z,:,:,1], cmap = 'jet')
plt.clim(-15,15)
m.drawcoastlines()
m.drawcountries()
plt.savefig('u10v10_2DCNN_nomax_autoencoder_performance_example5.jpg')
plt.show()
```

D-6 Python Codes for Auteoncoders with u10 and v10 as input

D-6-1 Data Adjustments Necessary

```
image_input = np.zeros((u10_cut.shape[0], u10_cut.shape[1], u10_cut.shape
    [2], 2))
image_input[:,:,:,0] = u10_cut
image_input[:,:,:,1] = v10_cut

lim = math.ceil(0.85*image_input.shape[0])

#Define amount of data that is train and test data
```

```
x_train = image_input[0:lim,:,:,:]
x_test = image_input[lim:,:,:,:]
```

D-6-2 No Autoencoder

```
#K-means needs at most 2 dimensions. Flatten u10 input to cluster
image_input = image_input.reshape(image_input.shape[0], image_input.shape
    [1]*image_input.shape[2]*image_input.shape[3])
print(image_input.shape)

#Compute K-means algorithm
n_clus= 50
kmeans=KMeans(n_clusters=n_clus)
clustered_data=kmeans.fit_predict(image_input)
```

D-6-3 Single Layered Autoencoder

```
# Simple Autoencoder
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],2))

#Encoder
x = Flatten()(input_img)
encoded = Dense(100, activation=None)(x)

#Decoder

x = Dense(28000, activation = None)(encoded)
decoded = Reshape((100,140,2))(x)

autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics=['accuracy'])

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-6-4 Deep Autoencoder

```
# Deep Autoencoder
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],2))

#Encoder
x = Flatten()(input_img)
x = Dense(1000, activation = 'relu')(x)
x = Dense(500, activation = 'relu')(x)
encoded = Dense(100, activation=None)(x)

#Decoder
```

```
11
12 x = Dense(500, activation = 'relu')(encoded)
13 x = Dense(1000, activation = 'relu')(x)
14 x = Dense(28000, activation = None)(x)
15 decoded = Reshape((100,140,2))(x)
16
17 autoencoder = Model(input_img, decoded)
18 encoder = Model(input_img, encoded)
19 #decoder = Model(encoded, input_img)
20 autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
       metrics=['accuracy'])
21
22 print(autoencoder.summary())
23
24 autoencoder.fit(x_train, x_train,
25                 epochs=500,
26                 batch_size=64,
27                 shuffle=True,
28                 validation_data=(x_test, x_test))
```

D-6-5 CNN Autoencoder

```
1  #CNN Autoencoder
2  input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],2))
3
4  #Encoder
5  x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
6  x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
7  x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
8  x = Flatten()(x)
9  encoded = Dense(100, activation='linear')(x)
10
11 #Decoder
12
13 x = Dense(14000, activation = 'linear')(encoded)
14 x = Reshape((100,140,1))(x)
15 x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
16 x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
17 decoded = Conv2DTranspose(2, (3, 3), activation = 'linear', padding='same
       ')(x)
18
19 autoencoder = Model(input_img, decoded)
20 encoder = Model(input_img, encoded)
21 #decoder = Model(encoded, input_img)
22 autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
       metrics = ['accuracy'])
23
24 print(autoencoder.summary())
25
26 autoencoder.fit(x_train, x_train,
27                 epochs=500,
28                 batch_size=64,
29                 shuffle=True,
```

```
30                 validation_data=(x_test, x_test))
```

D-6-6 Deep CNN Autoencoder

```
1  #Deep CNN Autoencoder
2  input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],2))
3
4  #Encoder
5  x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
6  x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
7  x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
8  x = Flatten()(x)
9  x = Dense(500, activation='relu')(x)
10 x = Dense(250, activation='relu')(x)
11 encoded = Dense(100, activation='linear')(x)
12
13 #Decoder
14 x = Dense(250, activation= 'relu')(encoded)
15 x = Dense(500, activation= 'relu')(x)
16 x = Dense(14000, activation = 'linear')(x)
17 x = Reshape((100,140,1))(x)
18 x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
19 x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
20 decoded = Conv2DTranspose(2, (3, 3), activation = 'linear', padding='same
       ')(x)
21
22 autoencoder = Model(input_img, decoded)
23 encoder = Model(input_img, encoded)
24 #decoder = Model(encoded, input_img)
25 autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
       metrics = ['accuracy'])
26
27 print(autoencoder.summary())
28
29 autoencoder.fit(x_train, x_train,
30                 epochs=500,
31                 batch_size=64,
32                 shuffle=True,
33                 validation_data=(x_test, x_test))
```

D-6-7 CNN Autoencoder with Max Pooling Layer

```
1  # CNN Autoencoder
2  input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],2))
3
4  #Encoder
5  x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
6  x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
       data_format=None)(x)
7  x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
8  x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
       data_format=None)(x)
```

```
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
#x = Dense(500, activation='relu')(x)
#x = Dense(250, activation='relu')(x)
encoded = Dense(100, activation='linear')(x)

#Decoder

#x = Dense(250, activation= 'relu')(encoded)
#x = Dense(500, activation= 'relu')(x)
x = Dense(875, activation = 'linear')(encoded)
x = Reshape((25,35,1))(x)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(
    x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(
    x)
decoded = Conv2DTranspose(2, (3, 3), activation = 'linear', padding='same
    ')(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-6-8 Deep CNN Autoencoder with Max Pooling Layer

```
# Deep CNN Autoencoder
input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],2))

#Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
    data_format=None)(x)
x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
x = MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',
    data_format=None)(x)
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
x = Dense(500, activation='relu')(x)
x = Dense(250, activation='relu')(x)
encoded = Dense(100, activation='linear')(x)

```

```
#Decoder

x = Dense(250, activation= 'relu')(encoded)
x = Dense(500, activation= 'relu')(x)
x = Dense(875, activation = 'linear')(x)
x = Reshape((25,35,1))(x)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(
    x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
x = UpSampling2D(size=(2, 2), data_format=None, interpolation='nearest')(
    x)
decoded = Conv2DTranspose(2, (3, 3), activation = 'linear', padding='same
    ')(x)


autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

print(autoencoder.summary())

autoencoder.fit(x_train, x_train,
                epochs=500,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-7 Calculate Average Wind Speed in Cluster

```
#Find the U_mean, V_mean and abs_wind_average for every cluster

#make a matrix for every amount_in_cluster
#start with amount_in_clus,100,140,

u_sum_cluster = np.zeros((u10_cut.shape[1], u10_cut.shape[2], n_clus))
u_average_in_cluster = np.zeros((u10_cut.shape[1], u10_cut.shape[2],
    n_clus))

v_sum_cluster = np.zeros((v10_cut.shape[1], v10_cut.shape[2], n_clus))
v_average_in_cluster = np.zeros((v10_cut.shape[1], v10_cut.shape[2],
    n_clus))

abs_wind_average = np.zeros((u10_cut.shape[1], u10_cut.shape[2], n_clus))

for x in range(len(clustered_data)):
    for i in range(n_clus):
        if clustered_data[x]==i:
            u_sum_cluster[:,:,i]=u_sum_cluster[:,:,i]+u10_cut[x,:,:]
            v_sum_cluster[:,:,i]=v_sum_cluster[:,:,i]+v10_cut[x,:,:]

```

```
for p in range(n_clus):
    u_average_in_cluster[:,:,p] = u_sum_cluster[:,:,p]/amount_in_cluster[
        p]
    v_average_in_cluster[:,:,p] = v_sum_cluster[:,:,p]/amount_in_cluster[
        p]
    abs_wind_average[:,:,p] = np.sqrt(np.square(u_average_in_cluster[:,:,
        p]) + np.square(v_average_in_cluster[:,:,p]))
```

D-8 3D Autoencoder

D-8-1 Data preparation

```
#Calculate how many days
tt = 4 #Different time steps per day
days = int(len(u10)/tt) #data set divided by amount of data points per
    day
print(days)

# Make the data for 3D Autoencoder
# Data input
image_input = np.zeros((days, u10_cut.shape[1], u10_cut.shape[2], tt, 2))
     # 2 because of U10 and V10
for i in range(days):
    for t in range(tt): #4 time steps
        image_input[i,:,:,t,0] = u10_cut[i*4+t,:,:]
        image_input[i,:,:,t,1] = v10_cut[i*4+t,:,:]
print(image_input.shape)

# Define testing training data
lim = math.ceil(0.85*days) #85 percent of data is training data, 15%
    testing

x_train = image_input[0:lim,:,:,:,:] #use the first 85% samples for
    training
x_test = image_input[lim:,:,:,:,:] #use the last 15% samples for testing
```

D-8-2 3D CNN Autoencoder

```
#3DCNN Autoencoder

input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],tt,2))

#Encoder
x = Conv3D(64, (3, 3, 3), activation='relu', padding='same')(input_img)
x = Conv3D(16, (3, 3, 3), activation='sigmoid', padding='same')(x)
x = Conv3D(1, (3, 3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
encoded = Dense(100, activation='linear')(x)

#Decoder
x = Dense(56000, activation = 'linear')(encoded)
```

```
x = Reshape((100,140,4,1))(x)
x = Conv3DTranspose(16, (3, 3, 3), activation='relu', padding='same')(x)
x = Conv3DTranspose(64, (3, 3, 3), activation='sigmoid', padding='same')(
    x)
decoded = Conv3DTranspose(2, (3, 3, 3), activation = 'linear', padding='
    same')(x)

autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=16,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-8-3 3D CNN Autoencoder with Max Pooling Layers

```
#3DCNN Autoencoder

input_img = Input(shape=(u10_cut.shape[1],u10_cut.shape[2],tt,2))

#Encoder
x = Conv3D(64, (3, 3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling3D(pool_size=(2, 2, 1), strides=None, padding='valid',
    data_format=None)(x)
x = Conv3D(16, (3, 3, 3), activation='sigmoid', padding='same')(x)
x = MaxPooling3D(pool_size=(2, 2, 1), strides=None, padding='valid',
    data_format=None)(x)
x = Conv3D(1, (3, 3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
encoded = Dense(100, activation='linear')(x)

#Decoder
x = Dense(3500, activation = 'linear')(encoded)
x = Reshape((25,35,4,1))(x)
x = Conv3DTranspose(16, (3, 3, 3), activation='relu', padding='same')(x)
x = UpSampling3D(size=(2, 2, 1), data_format=None)(x)
x = Conv3DTranspose(64, (3, 3, 3), activation='sigmoid', padding='same')(
    x)
x = UpSampling3D(size=(2, 2, 1), data_format=None)(x)
decoded = Conv3DTranspose(2, (3, 3, 3), activation = 'linear', padding='
    same')(x)

autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

autoencoder.fit(x_train, x_train,
                epochs=100,
```

```
                batch_size=16,
                shuffle=True,
                validation_data=(x_test, x_test))
```

D-9 Case Study

D-9-1 2D CNN Autoencoder

```
f = netCDF4.Dataset('Worldcup.nc')

time_all = f.variables['Times'][:]
lat = f.variables['XLAT'][:]
lon = f.variables['XLONG'][:]
u10 = f.variables['U10'][:]
v10 = f.variables['V10'][:]

image_input = np.zeros((u10.shape[0], u10.shape[1], u10.shape[2], 2)) # 2 because of U10 and V10
image_input[:,:,:,0]=u10 #image_input (T,lon,lat,u10)
image_input[:,:,:,1]=v10 #image_input (T,lon,lat,v10)

print(image_input.shape)

lim = math.ceil(0.85*len(u10)) #85 percent of data is training data, 15% testing

x_train = image_input[0:lim,:,:,:] #use the first 85% samples for training
x_test = image_input[lim:,:,:,:] #use the last 15% samples for testing


#2D CNN Autoencoder

input_img = Input(shape=(u10.shape[1],u10.shape[2],2))

#Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = Conv2D(16, (3, 3), activation='sigmoid', padding='same')(x)
x = Conv2D(1, (3, 3), activation='linear', padding='same')(x)
x = Flatten()(x)
encoded = Dense(100, activation='linear')(x)

#Decoder
x = Dense(30276, activation = 'linear')(encoded)
x = Reshape((174,174,1))(x)
x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
x = Conv2DTranspose(64, (3, 3), activation='sigmoid', padding='same')(x)
decoded = Conv2DTranspose(2, (3, 3), activation = 'linear', padding='same')(x)


autoencoder = Model(input_img, decoded)
```

```
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
    metrics = ['accuracy'])

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=64,
                shuffle=True,
                validation_data=(x_test, x_test))

R_dim = encoder.predict(image_input, batch_size=1)

# Calculate inertia for different cluster sizes
# fitting multiple k-means algorithms and storing the values in an empty
    list
SSE = []
for cluster in range(1,50):
    kmeans = KMeans(n_jobs = -1, n_clusters = cluster)
    kmeans.fit(R_dim)
    SSE.append(kmeans.inertia_)


# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(1,50), 'SSE':SSE})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['SSE'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')

#K-means algorithm
n_clus= 10
kmeans = KMeans(n_clusters=n_clus)
clustered_data = kmeans.fit_predict(R_dim)
```

D-9-2 3D CNN Autoencoder

```
f = netCDF4.Dataset('Worldcup.nc')

time_all = f.variables['Times'][:]
lat = f.variables['XLAT'][:]
lon = f.variables['XLONG'][:]
u10 = f.variables['U10'][:]
v10 = f.variables['V10'][:]

tt = 6 #Different time steps per data sample

#Calculate how many hours

hours = int(len(u10)/tt) #data set divided by amount of data points per
    day. Rounded to whole
print(hours)
print(len(u10)/tt)

```

```
17 image_input = np.zeros((days, u10.shape[1], u10.shape[2], tt, 2)) # 2
       because of U10 and V10
18 for i in range(hours):
19     for t in range(tt): #6 time steps
20         image_input[i,:,:,t,0] = u10[i*tt+t,:,:]
21         image_input[i,:,:,t,1] = v10[i*tt+t,:,:]
22
23 print(image_input.shape)
24
25 lim = math.ceil(0.85*days) #85 percent of data is training data, 15%
       testing
26
27 x_train = image_input[0:lim,:,:,:,:] #use the first 85% samples for
       training
28 x_test = image_input[lim:,:,:,:,:] #use the last 15% samples for testing
29
30
31 #CNN Autoencoder
32
33 input_img = Input(shape=(u10.shape[1],u10.shape[2],tt,2))
34
35 # CNN Autoencoder from https://blog.keras.io/building-autoencoders-in-
       keras.html
36
37 #Encoder
38 x = Conv3D(64, (3, 3, 3), activation='relu', padding='same')(input_img)
39 x = Conv3D(16, (3, 3, 3), activation='sigmoid', padding='same')(x)
40 x = Conv3D(1, (3, 3, 3), activation='linear', padding='same')(x)
41 x = Flatten()(x)
42 encoded = Dense(100, activation='linear')(x)
43
44 #Decoder
45 x = Dense(181656, activation = 'linear')(encoded)
46 x = Reshape((174,174,6,1))(x)
47 x = Conv3DTranspose(16, (3, 3, 3), activation='relu', padding='same')(x)
48 x = Conv3DTranspose(64, (3, 3, 3), activation='sigmoid', padding='same')(
       x)
49 decoded = Conv3DTranspose(2, (3, 3, 3), activation = 'linear', padding='
       same')(x)
50
51 autoencoder = Model(input_img, decoded)
52 encoder = Model(input_img, encoded)
53 autoencoder.compile(optimizer='adamax', loss='mean_absolute_error',
       metrics = ['accuracy'])
54
55 autoencoder.fit(x_train, x_train,
56                 epochs=50,
57                 batch_size=4,
58                 shuffle=True,
59                 validation_data=(x_test, x_test))
60
61 R_dim = encoder.predict(image_input, batch_size=1)
62
```

```
63 #K-means algorithm
64 n_clus= 10
65 kmeans = KMeans(n_clusters=n_clus)
66 clustered_data = kmeans.fit_predict(R_dim)
```

Bibliography

[1] U. C. for Atmospheric Research (UCAR), "Comet meted program." https://www.meted.ucar.edu/. Accessed: 17-09-2019.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] J. Schmidhuber, "Deep learning in neural networks: an overview," *Neural Networks*, vol. 61, pp. 85–117, 2014.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[6] P. Ramachandran, B. Zoph, and Q. Le, "Searching for activation functions," *Google Brain*, 2017. arXiv:1710.05941v2.

[7] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

[8] B. Karlik and A. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems (IJAE)*, vol. 1, pp. 111–122, 2011.

[9] W. He, "Deep neural network based load forecast," *Computer Modelling & New Technologies*, vol. 18, pp. 258–262, 2014.

[10] D. Psaltis, A. Sideris, and A. Yamamura, "A multilayered neural network controller," *IEEE Control Systems Magazine*, vol. 8, pp. 17–21, 1988.

[11] S. Ruder, "An overview of gradient descent optimization algorithms," *https://arxiv.org/abs/1609.04747v2*, 2017.

[12] N. Qian, "On the momentum term in gradient descent learning algorithms," *The official journal of the international Neural Network Society*, vol. 12, pp. 145–151, 1999.

[13] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[14] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *3rd International Conference for Learning Representations*, 2015.

[15] G. Hinton, "Rmsprop." http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Visited on 28/02/2020.

[16] V. Bushaev. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimiza 2018. Visited on 28/02/2020.

[17] Y. Dauphin, R. Pascanu, C. Gulcehre, K. CHo, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," *Proceedings of the 27th International Conference on Neural Information Processing Systems*, vol. 2, pp. 2933–2941, 2014.

[18] A. Coates and Y. NG, *Learning Feature Representation with K-means*. Springer, 2012. Part of the book: "Neural Networks: Tricks of the Trade", written by G. Montavon, G.B. Orr and K. Müller.

[19] C. Zhang, X. Pan, H. Li, A. Gardiner, I. Sargent, J. Hare, and P. Atkinson, "A hybrid mlp-cnn classifier for very fine resolution remotely sensedimage classification," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 140, pp. 133–144, 2017.

[20] F. Huang and Y. LeCun, "Large-scale learning with svm and convolutional nets for generic object categorization," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2006.

[21] J. Ludwig, "Image convolution." http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf. Visited on 04-07-2019.

[22] A. Krizhevsky, "Convolutional deep belief networks on cifar-10," 2010. https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf.

[23] A.Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[24] Y. Wang, H. Yao, and S. Zhao, "Auto-encoder based dimensionality reduction," *Elsevier Neurocomputing*, vol. 184, pp. 232–242, 2015.

[25] G. Liu, H. Bao, and B. Han, "A stacked autoencoder-based deep neural network for achieving gearbox fault diagnosis," *Mathematical Problems in Engineering*, 2018.

[26] K. Fujimura, "2018: Japan's year of catastrophes." https://www.air-worldwide.com/publications/air-currents/2019/2018-japans-year-of-catastrophes/. Visited on 15/02/2020.

[27] "K-means clustering." https://scikit-learn.org/stable/modules/clustering.html#k-means. Visited on 27/02/2020.

[28] A. Amelia, "K-means clustering: From a to z." https://towardsdatascience.com/k-means-clustering-from-a-to-z-f6242a314e9a, 2018. Visited on 27/02/2020.

[29] J. M. Agency. https://www.jma.go.jp/jma/jma-eng/jma-center/rsmc-hp-pub-eg/besttrack_viewer.html. Visited on 29/02/2020.

[30] C. Patlolla, "Understanding the concept of hierarchical clustering technique." https://towardsdatascience.com/understanding-the-concept-of-hierarchical-clustering-technique-c6e8243758ec, 2018. Visited on 02/03/2020.

[31] D. Sculley, "Web-scale k-means clustering," *Proceedings of the 19th international conference on world wide web*, 2010.

[32] "Clustering scikit learn website." https://scikit-learn.org/stable/modules/clustering.html.