UNIVERSITA' DEGLI STUDI DI PARMA

DOTTORATO DI RICERCA IN
Ingegneria Civile e Architettura

CICLO 37°

Forecasting landslide behavior with Deep Learning algorithms for Early Warning purposes

Coordinatore:
Chiar.mo Prof. Andrea Spagnoli

Tutore:
Chiar.mo Prof. Andrea Segalini

Correlatore:
Dott. Ing. Alessandro Valletta

Dottorando: Marco Conciatori

Anni Accademici 2021/2022 – 2023/2024

Table of Contents

List of Figures ... 4

List of Tables ... 6

1. Introduction ... 8

2. State of the Art in Machine Learning ... 9

2.1. Supervised Learning ... 11

2.1.1. Linear and Logistic Regression ... 12

2.1.2. Support Vector Machines (SVM) ... 12

2.1.3. Decision Trees and Ensemble Methods ... 13

2.2. Unsupervised Learning ... 13

2.2.1. Clustering ... 14

2.2.2. Dimensionality Reduction ... 15

2.3. Reinforcement Learning ... 16

2.3.1. Q-learning ... 16

2.3.2. Policy Gradient Methods ... 17

2.4. Deep Learning ... 17

2.4.1. Feed-forward Neural Networks ... 17

2.4.2. Convolutional Neural Networks ... 20

2.4.3. Transformers and attention mechanisms ... 22

2.4.4. Recurrent Neural Networks ... 24

2.4.5. Transfer Learning and Fine-tuning ... 25

2.5. Emerging Trends ... 27

2.6. ML in the Geotechnical field ... 27

2.6.1. Soil Classification and Property Prediction ... 27

2.6.2. Settlement Prediction and Foundation Design ... 28

2.6.3. Underground excavation and Support Design ... 28

2.6.4. Geotechnical Instrumentation and Monitoring ... 28

2.6.5. Slope Stability and Landslide forecasting ... 28

3. Data ... 33

3.1. Landslide forecasting ... 33

3.1.1. Study sites ... 33

3.1.2. Data structure ... 39

3.1.3. Sensors and data collection ... 41

3.1.4. Data preprocessing ... 44

3.2. Tree species classification ... 50

3.2.1. Study site ... 50
3.2.2. Data structure ... 52
3.2.3. Sensors and data collection ... 53
3.2.4. Data preprocessing ... 54
4. Algorithms development ... 56
4.1. Landslide forecasting ... 56
4.1.1. Comparison algorithm (Baseline) ... 56
4.1.2. Original algorithm ... 57
4.1.3. Vertical Array as input ... 60
4.1.4. Addition of rainfall data ... 61
4.1.5. Custom loss function ... 63
4.1.6. Neural Network's architecture expansion ... 66
4.1.7. Data balancing and data augmentation ... 69
4.1.8. Condensation and reduction of input features ... 71
4.1.9. Switch from regression to classification ... 72
4.1.10. Conversion of time series to images ... 73
4.2. Tree species classification ... 76
4.2.1. Motivation ... 76
4.2.2. Original algorithm ... 78
4.2.3. Orthomosaic input ... 79
4.3. Custom metrics development ... 83
4.3.1. Background and Motivation ... 83
4.3.2. Metrics for landslide forecasting and tree species recognition ... 84
4.3.3. Metrics for tree species recognition with orthomosaic input ... 87
4.4. Interpolation and graphical representation of displacements ... 88
4.5. Hyperparameter space exploration ... 89
4.6. Implementation details and code availability ... 92
5. Results ... 94
5.1. Landslide forecasting ... 94
5.1.1. Original algorithm ... 94
5.1.2. Vertical Array as input ... 97
5.1.3. Addition of rainfall data ... 98
5.1.4. Custom loss function ... 100
5.1.5. Neural Network's architecture expansion ... 104
5.1.6. Full dataset available (all four sites) ... 107
5.1.7. Data balancing and data augmentation ... 109

5.1.8. Condensation and reduction of input features 111
5.1.9. Switch from regression to classification... 112
5.1.10. Conversion of time series to images 114
5.1.11. Relationship between data quantity and model performance 116
5.1.12. Runtime considerations 118
5.1.13. Overall comparison... 119
5.2. Tree species classification 120
5.2.1. Original algorithm 120
5.2.2. Relationship between data quantity and model performance 124
5.2.3. Runtime considerations 126
5.2.4. Orthomosaic input.. 127
6. Discussion.. 127
7. Conclusions 129
7.1. Future developments 129
8. Bibliography.. 130

List of Figures

Figure 1: Max Pooling examples (1D left, 2D right). 21
Figure 2: Recurrent Neural Network (RNN) architecture. 24
Figure 3: transfer learning example. It is the procedure employed in the tree species recognition and the last version of the landslide forecasting algorithms. Image adapted from (Mukhlif et al., 2023).. 26
Figure 4: input image composed of a rainfall scalogram (upper half) and velocity scalogram (lower half). Source: (Teza et al., 2022). 32
Figure 5: retaining wall and monitoring devices placement. Source: (Segalini et al., 2019)....... 37
Figure 6: Measurement devices placement. V = DT0103; A = DT0003 + 1 Crack Link of DT0002; K = 1 Crack Link of DT0002. Source: (Bertolotti, 2022). 39
Figure 7: MUMS displacement calculation (left) and composition of individual inclinometer (right). 42
Figure 8: MUMS Measurement axis, side view (left) and top view (right)................................ 42
Figure 9: example of dataset subdivision. Two thresholds separate data into three categories. 46
Figure 10: example of dataset subdivision with thresholds based on standard deviation instead of absolute values. 46
Figure 11: example of observation-target pairs extraction from time series............................ 47
Figure 12: Zao mountains location, zoom of site 1. Source: (Conciatori et al., 2024). 51
Figure 13: drone DJI Mavic 2 Pro. 54
Figure 14: Observation, target, and prediction. Only x displacement is plotted. 58
Figure 15: Observation, target, and prediction. Plotting x displacement, water level and atmospheric pressure at ground level (no y displacement). 59
Figure 16: architecture of the NNs used in the earlier iterations of the software. 60

Figure 17: whole Vertical Array as input. 61
Figure 18: output for the algorithm that takes a whole Vertical Array as input. 61
Figure 19: displacements and precipitations measured in Site A (January 2021). It is possible to see both phenomena: displacement provoked by rainfall, and the delay of ~2-5 days between cause and effect. 63
Figure 20: example of typical bad prediction. The model ignores an evident trend because it learned to always predict that the future displacement will be very similar to the last observed displacement. 65
Figure 21: graphical explanation for the TSM loss function. Green and red demarcate the two areas in which predictions are considered “good” or “bad” respectively. Where good means that the prediction is closer to the real displacement (green dot) than to the last observation (rightmost purple dot), and the opposite for bad predictions. This is achieved by calculating the midpoint (black dot) between real displacement and last observation, which is the watershed between the two areas. 66
Figure 22: Dense NN modular architecture. The red part consists of layers that can be modified by the user: the number of layers and the number of neurons in each is customizable. 67
Figure 23: Convolutional Neural Network modular architecture. Red layers can be customized by the user at creation time. 69
Figure 24: example of input for the algorithm described in point 2. Two plots containing displacements and precipitations respectively are stacked to create a single input image. 74
Figure 25: summary of the Gramian Angular Field conversion procedure. Source: (Z. Wang & Oates, 2015) 75
Figure 26: a, b, and c are examples of observations converted to images with the GAF method. Data comes from real displacements measured in site A. Images constructed in this way are always symmetrical along the main diagonal. 75
Figure 27: input/output example of the second version of tree species classification algorithm. It takes a whole orthomosaic as input, and outputs an image with the same shape color-coded by species distribution. 82
Figure 28: ML-related articles published. Source: (Conciatori, Valletta, et al., 2024). 83
Figure 29: Confusion Matrix example. The main diagonal is highlighted with light-blue color. ... 85
Figure 30: Standard and custom Recall values for varying degrees of imbalance (left). Standard and custom F1 values for varying degrees of imbalance (right). Source: (Conciatori, Valletta, et al., 2024). 87
Figure 31: site A map with interpolated displacement overlay between three sites. Source: (Conciatori et al., 2022). Red dots mark the position of Vertical Arrays. 89
Figure 32: example of hyperparameter search results with two hyperparameters variable (Learning Rate and ndi). All other hyperparameters are fixed on their initial values. The height of a bar represents the performance of a model trained with the corresponding LR and ndi. Since loss is an error with respect to true displacements, shorter bars mean better models. 92
Figure 33: performance of original algorithm and baseline on the test set. MAE = Mean Absolute Error, MSE = mean Squared Error. 96
Figure 34: real VS predicted displacements over a month. Both x (above) and y (below) are displayed. 96
Figure 35: performance of original algorithm and the new version which takes vertical arrays as inputs. 98
Figure 36: results of the three experiments with rainfall data. The original algorithm, which is the same but without access to precipitation information, is added for comparison. 99

Figure 37: comparison of the eight algorithms tested. As in the previous sections, the metrics displayed here are MAE and MSE. Better performance corresponds to lower values of these metrics. ... 102
Figure 38: comparison of the eight algorithms tested. The metrics developed in Section "4.3.2. Metrics for landslide forecasting and tree species recognition" are used here. Better performance corresponds to higher values of these metrics. ... 103
Figure 39: confusion matrix of one of the five experiments that were conducted to calculate the average performance of TSM (no rain, c=10). The red cell is the intersection between the main diagonal (correct predictions) and the first column (this algorithm biased preference). ... 104
Figure 40: performance of the models obtained with the architecture expansion modification. FFNN is the Feed-Froward Neural Network used in all previous experiments, but now with more dense layers, 1D CNN is the new architecture introduced: one-dimensional Convolutional Neural Network. ... 106
Figure 41: performance of the two models described in the previous section, trained using data from all four sites. ... 108
Figure 42: three models are tested, with increasing degree of data balancing and augmentation. Now F1 Score metric is also implemented and shown in the results. ... 110
Figure 43: best model from the previous section (balanced dataset, 5X data augmentation) compared with new version ("Features reduction"). ... 112
Figure 44: performance of the new model that predicts the class of future displacements, compared to the best models of the previous two sections. ... 113
Figure 45: confusion matrix of the test results for one of the Classification NNs trained and tested for this experiment. ... 114
Figure 46: comparison between "plain graph" model and GAF model. Both use images as input in place of time series. ... 116
Figure 47: (a) Loss function variation with respect to the number of training observations; (b) Mean Squared Error (MSE) and Mean Absolute Error (MAE) for each dataset size. ... 118
Figure 48: summary of all tests conducted. This figure allows to compare any version with any other. ... 120
Figure 49: models' comparison – detailed. ... 121
Figure 50: models' comparison - summary. ... 122
Figure 51: confusion matrix of the best model. The cells on the main diagonal represent the correct predictions. ... 123
Figure 52: species classification results depending on training data quantity. The "Total" label indicates the number of patches that compose the current dataset, the "Train" label is number of patches effectively used for training. ... 124
Figure 53: species classification results depending on training data quantity with a different test set specifically prepared. The "Total" label indicates the number of patches that compose the current dataset, the "Train" label is number of patches effectively used for training. ... 125

List of Tables

Table 1: Number of observations collected from each site. This is with 12-days observations (the most common choice in the experiments) and after removing missing or erroneous instrument readings. ... 33
Table 2: site A's MUMS composition. ... 34

Table 3: site B’s MUMS composition. 35
Table 4: site C’s MUMS composition. 36
Table 5: site D’s MUMS composition. 38
Table 6: example of input data structure. 39
Table 7: example of output data structure. 40
Table 8: input data structure. 40
Table 9: output structure used. 40
Table 10: plant species count, divided by site and species. 52
Table 11: example of input data for the tree classification model. 52
Table 12: original algorithm input. 57
Table 13: original algorithm output. 58
Table 14: composition of the input for the algorithm that also uses rainfall. 62
Table 15: “reduced” input, format of the input for the landslide forecasting algorithm after feature condensation. 72
Table 16: “reduced” output, format of the output for the landslide forecasting algorithm after feature condensation. 72
Table 17: example outputs for which knowing the uncertainty (instead of just the top class) is useful. 79
Table 18: hyperparameters used to test the original algorithm. 95
Table 19: hyperparameters selected for testing the algorithm that takes a whole vertical array as input and the model from the previous section. 97
Table 20: hyperparameters used for the three tests of the algorithm that uses rainfall data. Most hyperparameters remain unchanged, which is indicated with gray cells. Original algorithm added as comparison. 99
Table 21: hyperparameters used for the first four tests (no rainfall data) of the algorithm that uses the TSM custom loss function. Most hyperparameters remain unchanged, which is indicated with gray cells. 100
Table 22: hyperparameters used for the other four tests (with rainfall data) of the algorithm that uses the TSM custom loss function. Most hyperparameters remain unchanged, which is indicated with gray cells. 101
Table 23: hyperparameters used for the two experiments. 105
Table 24: additional hyperparameters specific to the CNN. 106
Table 25: hyperparameters used for the two experiments. 107
Table 26: additional hyperparameters specific to the CNN. Since “Pooling operation” is null, the pooling parameters are not used in this configuration, so they are marked by gray background. This configuration will be used for most future experiments that for simplicity will refer to this table instead of repeating it. 108
Table 27: hyperparameters used for the three experiments with data balancing and data augmentation. 109
Table 28: hyperparameters for the feature reduction experiments. 111
Table 29: hyperparameters for the classification experiments. 112
Table 30: hyperparameters for the two algorithms described in the corresponding development section (“4.1.10. Conversion of time series to images”). 115
Table 31: data quantity and corresponding performance. 117
Table 32: hardware specifications of the computer used for execution-time measurements. 118
Table 33: execution time required by each task using the computer described above. 119
Table 34: best hyperparameters found with automatic hyperparameter space exploration. ... 122
Table 35: metric evaluation of the best model found. 123

Table 36: hardware specifications of the two computers used for execution-time measurements.. 126
Table 37: execution time required by each task depending on the computer used. 126

1. Introduction

Landslides represent a relevant hazard in terms of risk for humans and damage to infrastructure. Landslides are also particular in that the threat they pose has been constantly increasing by the year for multiple reasons: expansion of human presence in the environment (not only for habitation but also for roads, agriculture, and industries), anthropogenic influences on the environment such as deforestation, and climate change impact.

Knowledge and instruments to prepare for, mitigate, and respond to such events also improve constantly. With the substantial advancement that took place in the sensors' technology, it is possible to collect vast amounts of data from monitored sites with high frequency and reliability, often with near real time availability. This information can include a variety of different physical characteristics: land displacement, inclination, water level, temperature, pressure...

This study proposes a method for predicting the near future behavior of landslides, so that dangerous events can be identified with more margin on the time to alert and prepare. It leverages this wealth of information and the developments in the field of Machine Learning to generate predictions of displacements with adaptability in regard to the type of data used. The software developed can be used to create and train models with customizable specifications, and subsequently test and utilize them. A model consists of a Neural Network plus some additional functions to prepare and standardize the input data. The model trains using known past data, and afterwards takes as inputs the measured physical characteristics of a site and predicts the displacement expected for the following days.

Additionally, a separate Deep Learning model that recognizes plant presence and species from drone photos has been implemented. Its purpose is to utilize this new information as extra input for the landslide forecasting model.

Artificial Intelligence and Machine Learning registered an explosion of interest in research and have found many real-world applications, mainly thanks to Deep Neural Networks. The most important algorithms are discussed in the next section, with a focus on the more relevant applications of ML in the Geotechnical field.

In the “3. Data” Section all records used for this study are presented, with detailed descriptions of their sources’ location, and collection instruments. This and the following sections are divided into landslide forecasting and tree species recognition, since they are effectively two different algorithms.

The fourth Section (“4. Algorithms development”) describes the ideas, formulas, and code implementations behind the two algorithms, and the different versions created. Other important code-related considerations are added here, for example the explanation of the custom metrics used, or the principles behind the search for optimal hyperparameters.

In the “5. Results” Section, the performance of each major version of the landslide forecasting algorithm is shown. They are compared with previous/similar versions and the results are discussed one by one. The same is done for the tree species recognition algorithm, but it only has two versions to test. Experiments are also run to evaluate the importance of data quantity and time consumption.

In the “6. Discussion” Section, the results obtained by the various versions are evaluated in the complete context and their implications are considered. The problem of landslide forecasting is also discussed from a more abstract standpoint.

The last Section summarizes the work done for this study, the results obtained and proposes future research directions given the knowledge and experience accumulated.

2. State of the Art in Machine Learning

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that focuses on the development of algorithms that allow computers to learn from and make predictions or decisions based on data. Unlike traditional programming, where explicit instructions are coded by humans, machine learning algorithms identify patterns in data and make decisions with minimal human intervention. The field has witnessed exponential growth due to advancements in computational power, the availability of large datasets, and the development of sophisticated algorithms.

The following are definitions of key concepts of ML algorithms (Goodfellow et al., 2016).

Data: the foundation of Machine Learning is data. It can be in the form of numbers, text, images, or any other type of information that can be processed by a computer. Data can be structured or

unstructured. Structured data is highly organized and easily searchable (e.g., databases, spreadsheets), while unstructured data is less organized (e.g., text, images, audio).

Features:

- Features are individual measurable properties or characteristics of the data. For example, in a dataset of house prices, features might include the size of the house, number of bedrooms, and location.
- Feature Selection: identifying the most relevant features for model training to reduce dimensionality and improve model performance. Methods include filter methods (e.g., correlation, mutual information) (Croxton & Cowden, 1939), wrapper methods (e.g., recursive feature elimination) (F. Li & Yang, 2005), and embedded methods (e.g., LASSO) (Ranstam & Cook, 2018), tree-based feature importance (Zhou & Hooker, 2021).
- Feature Engineering: the process of selecting, modifying, or creating new features to improve the performance of an ML model. Techniques include normalization, standardization, and encoding categorical variables.

Labels (or targets): the variables that we want to predict. In supervised learning, each data point has a corresponding label. For example, for a house price prediction model, the label would be the price of the house. It can be continuous (regression) or categorical (classification).

Model (or algorithm): a mathematical representation that maps input data (features) to output data (labels). Models are trained to learn the relationships between inputs and outputs.

Parameters (or weights): they are the internal variables that allow models to train. They change automatically during training and encode the model's accumulated knowledge of the task. Recent models can have a very high number of weights and biases, for example Llama 3.1 has 405×10^9 parameters (Dubey et al., 2024), while ChatGPT-4 is estimated to have more than 10^{12} ('GPT-4', 2024; OpenAI et al., 2023).

Hyperparameters: they are variables that control the learning process (e.g., learning rate, number of hidden layers, number of epochs). They are manually set before training and selecting the best values for them is really important for the quality of the resulting trained model.

Training:

- Training a model involves feeding it a large amount of data and adjusting the model's parameters to minimize errors in predictions. This process is iterative and continues until the model performs well on the training data.
- Loss Function: a mathematical function that measures the difference between the model's predictions and the actual values. Common loss functions include Mean Squared Error (MSE) (Goodfellow et al., 2016) for regression and Cross Entropy (Mao et al., 2023) for classification.
- Optimization Algorithms: methods to converge on the best parameters (parameters that minimize the loss function). Common algorithms include Gradient Descent, Stochastic Gradient Descent (SGD), and Adam (Kingma & Ba, 2017; Lecun et al., 1998).

Testing:

- After training, the model is evaluated on a separate dataset (testing data) to assess its performance and ability to generalize to new, unseen data.
- Evaluation Metrics: metrics used to assess the performance of an ML model. For regression, common metrics include Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared (Goodfellow et al., 2016). For classification, metrics include accuracy, precision, recall, and F1-score (Powers, 2008; Sasaki, 2007).
- Cross-Validation: a technique to assess the model's performance by dividing the dataset into multiple folds and training/testing the model on different subsets. Common methods include k-fold cross-validation and leave-one-out cross-validation (Berrar, 2019).

The rest of the section delves into the current state of Machine Learning, along with its applications and emerging trends. Afterwards, there is a more in-depth examination of algorithms and studies closely related to this thesis. This includes Deep Learning, currently the most successful and promising subfield of ML, and the state-of-the-art on the two applications presented in this thesis: landslide forecasting, and tree species classification.

2.1. Supervised Learning

Supervised learning involves training a model on a labeled dataset, where the desired output is known. It encompasses a wide range of algorithms such as linear regression, logistic regression, support vector machines (SVM), decision trees, and ensemble methods like random forests and gradient boosting machines.

2.1.1. Linear and Logistic Regression

Linear regression (Su et al., 2012) is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. The goal of linear regression is to find the values of the coefficients that minimize the Sum of Squared Errors (SSE) between the predicted and actual values of the dependent variable. Linear regression assumes a linear relationship between the dependent and independent variables, making it suitable for continuous outcome variables. Logistic regression (LaValley, 2008), on the other hand, is used when the dependent variable is categorical, particularly binary (having two possible outcomes). Instead of predicting a continuous outcome, logistic regression models the probability that a given input point belongs to a particular class. The logistic regression model uses the logistic function (also known as the sigmoid function) to map predicted values to probabilities. The coefficients are estimated using Maximum Likelihood Estimation (MLE), which maximizes the likelihood that the observed data is accurately predicted by the model. Logistic regression is widely used for binary classification problems and can be extended to multiclass classification through techniques like One-vs-Rest (OvR) or multinomial logistic regression.

2.1.2. Support Vector Machines (SVM)

Support Vector Machines (SVM) (Suthaharan, 2016) are supervised learning methods used for classification, regression, and outlier detection. In classification tasks, SVM aims to find the optimal hyperplane that best separates data points of different classes in a high-dimensional space, maximizing the margin between the closest points (support vectors) of the classes to the hyperplane. The decision function is $f(x) = w \cdot x + b$, with the goal of minimizing $\|w\|^2$ while ensuring each data point is correctly classified with a margin of at least 1.

When data is not linearly separable, SVMs map input features into a higher-dimensional space, enabling linear separation. Common kernels include polynomial, radial basis function (RBF), and sigmoid. SVMs are effective in high-dimensional spaces and versatile, applicable to text classification, image recognition, and more. They can also handle regression problems (Support Vector Regression, SVR) and multi-class classification using strategies like one-vs-one or one-vs-rest. In summary, SVMs are robust classification techniques that find optimal hyperplanes to separate classes, leveraging kernel methods for non-linear data.

2.1.3. Decision Trees and Ensemble Methods

Decision Trees (SONG & LU, 2015) are a type of supervised learning algorithm used for both classification and regression tasks. They work by recursively splitting the data into subsets based on the value of input features, creating a tree-like model of decisions. Each internal node represents a feature test, each branch represents an outcome of the test, and each leaf node represents a class label or continuous value. The goal is to create a model that predicts the target variable by learning simple decision rules inferred from the data features. Decision Trees are easy to understand and interpret, but they are prone to overfitting, especially with complex datasets.

Ensemble Methods (Dietterich, 2000) enhance the performance of single models like Decision Trees by combining multiple models to produce a more robust and accurate predictive model. There are various ensemble techniques, but two of the most popular are Bagging and Boosting. Bagging, or Bootstrap Aggregating, involves training multiple Decision Trees on different random subsets of the training data and then averaging their predictions to reduce variance and prevent overfitting. Random Forest is a well-known example of a bagging method, where a large number of Decision Trees are trained on bootstrapped samples of the data, and the final prediction is made by majority voting in classification or averaging in regression.

Boosting, on the other hand, builds models sequentially, each new model attempting to correct the errors of the previous ones. It focuses on difficult cases by giving them more weight in subsequent models. AdaBoost and Gradient Boosting are prominent boosting techniques. AdaBoost adjusts the weights of incorrectly classified instances, while Gradient Boosting builds models in a stage-wise fashion.

Ensemble methods generally provide better performance than individual models because they combine the strengths of multiple models and mitigate their weaknesses.

2.2. Unsupervised Learning

Unsupervised learning (Goodfellow et al., 2016) deals with unlabeled data, aiming to find hidden patterns or intrinsic structures. Common algorithms include clustering techniques like k-means, hierarchical clustering, and density-based spatial clustering of applications with noise (DBSCAN), as well as dimensionality reduction methods such as principal component analysis (PCA) and t-distributed stochastic neighbor embedding (t-SNE). Having labeled data is much better for ML algorithms, the importance of Unsupervised Learning comes from the fact that the

amount of unlabeled data is many orders of magnitude higher than that of labeled data. Any algorithm that can extract useful information from unlabeled data is therefore important.

2.2.1. Clustering

The goal of Clustering is to group a set of objects in such a way that objects in the same group (or cluster) are more similar to each other than to those in other groups. Unlike supervised learning, clustering deals with unlabeled data, finding inherent structures within the data set. There are several popular clustering algorithms, each with its own approach to defining clusters.

- K-Means (Ahmed et al., 2020): this algorithm partitions the data into K clusters. It starts by initializing K centroids randomly and iteratively assigns each data point to the nearest centroid, then recalculates the centroids as the average of all points in a cluster. This process repeats until the centroids no longer change significantly.
- Hierarchical Clustering (Nielsen, 2016): this method creates a tree-like structure of clusters by either iteratively merging smaller clusters (agglomerative) or splitting larger clusters (divisive). The result is a dendrogram, which can be cut at different levels to produce different numbers of clusters.
- Density-Based Spatial Clustering of Applications with Noise (DBSCAN) (Ester et al., 1996): this algorithm groups together points that are closely packed, marking points in low-density regions as outliers. It defines clusters based on the density of data points in a region, making it particularly useful for discovering clusters of arbitrary shapes.
- Gaussian Mixture Models (GMM) (Reynolds & others, 2009; Viroli & McLachlan, 2019): this probabilistic model assumes that the data is generated from a mixture of several Gaussian distributions with unknown parameters. It uses the Expectation-Maximization algorithm to find the best fit of the Gaussian distributions to the data.

Clustering has a wide range of applications, including social network analysis, image segmentation, and anomaly detection. It helps in identifying patterns and structures in data, enabling better understanding and decision-making. The effectiveness of clustering depends on the choice of the algorithm, the nature of the data, and the distance metric used to measure similarity between data points.

2.2.2. Dimensionality Reduction

Dimensionality reduction (Jia et al., 2022) is a technique used in Machine Learning and data analysis to reduce the number of features or dimensions in a dataset while retaining as much relevant information as possible. It simplifies data, making it easier to visualize, interpret, and process, and can help improve the performance of Machine Learning algorithms by reducing overfitting and computational cost. There are two main approaches to dimensionality reduction: feature selection and feature extraction. Feature Selection involves selecting a subset of the most important features from the original dataset, based on specific criteria or algorithms. Examples include correlation coefficients and mutual information. Feature Extraction creates new features by transforming the original data into a lower-dimensional space. Popular techniques include the following algorithms.

- Principal Component Analysis (PCA) (Jia et al., 2022): this method transforms the original features into a set of linearly uncorrelated components, ordered by the amount of variance they explain in the data. PCA finds the directions (principal components) that maximize the variance and projects the data onto these directions, reducing dimensionality while preserving as much variability as possible.
- Linear Discriminant Analysis (LDA) (Jia et al., 2022): primarily used for classification tasks, LDA finds the linear combinations of features that best separate different classes. It maximizes the ratio of between-class variance to within-class variance, making it useful for reducing dimensionality while enhancing class separability.
- t-Distributed Stochastic Neighbor Embedding (t-SNE) (Linderman et al., 2019): this non-linear technique is particularly effective for visualizing high-dimensional data in 2 or 3 dimensions. It minimizes the divergence between two distributions: one that measures pairwise similarities of the input objects in high-dimensional space and one that measures pairwise similarities of the corresponding low-dimensional points.

Dimensionality reduction techniques are widely used in various applications, including data visualization, noise reduction, and improving the efficiency of machine learning models. By focusing on the most informative features or creating new, more compact representations, these techniques help uncover underlying patterns and structures in the data, making analysis more manageable and insightful.

2.3. Reinforcement Learning

Reinforcement learning (RL) (Wiering & Van Otterlo, 2012) is a paradigm where agents learn to make decisions by interacting with the environment. An agent knows the current state of the environment and takes actions based on that and its goal. Each action changes the state of the environment and provides a reward to the agent. The agent tries to maximize cumulative rewards over time. Key algorithms in RL include Q-learning and policy gradient methods. RL has seen applications in fields like robotics, autonomous driving, and recommendation systems, where it helps solve sequential decision-making problems by learning optimal policies through trial and error.

2.3.1. Q-learning

Q-learning (Watkins & Dayan, 1992) is a model-free reinforcement learning algorithm used to determine the optimal action-selection policy for a given finite Markov Decision Process (MDP). It focuses on learning the quality of actions, represented by the Q-value, to help an agent make decisions that maximize its cumulative reward over time.

The Q-value $Q(s, a)$ represents the expected future rewards of taking action a in state s, followed by the optimal policy thereafter. This value is updated iteratively using the Bellman equation (Eq. 1):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q\,(s', a') - Q(s,a) \right] \tag{1}$$

where α is the learning rate, r is the reward, γ is the discount factor, s' is the new state, and a' is the new action.

The Q-learning process begins by initializing Q-values for all state-action pairs. The agent then selects actions based on an ϵ-greedy policy, which balances exploration and exploitation. After executing an action and observing the resulting state and reward, the agent updates the Q-value for the state-action pair. This cycle repeats until the Q-values stabilize or for a set number of episodes. Q-learning is advantageous because it does not require a model of the environment and can handle unknown environments, but it is unable to solve continuous environments.

2.3.2. Policy Gradient Methods

Policy gradient methods (Sutton et al., 1999) are a class of reinforcement learning algorithms that optimize the policy directly rather than estimating value functions. They are particularly useful for problems with continuous action spaces and where the policy needs to be stochastic.

In reinforcement learning, the policy defines the behavior of an agent, mapping states to actions. Policy gradient methods seek to find the optimal policy by adjusting the parameters of the policy to maximize the expected cumulative reward. This is achieved by using gradient ascent on the expected reward.

The key idea is to parameterize the policy by a set of parameters θ. The policy, denoted as $\pi_\theta(a|s)$, gives the probability of taking action a in state s under parameters θ. The objective is to maximize the expected return, $J(\theta)$, which is the expected cumulative reward.

To optimize $J(\theta)$, we compute the gradient $\nabla_\theta J(\theta)$ and update the parameters in the direction of this gradient. This is done using the policy gradient theorem (Equation 2), which states:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s|a)] \quad (2)$$

where $Q^\pi(s|a)$ is the action-value function, representing the expected return for taking action a in state s under policy π. In practice, the gradient is estimated using samples from the environment.

Policy gradient methods have several advantages: they can handle high-dimensional and continuous action spaces, and they naturally support stochastic policies, which are beneficial for exploration. Additionally, they directly optimize the performance measure of interest.

2.4. Deep Learning

Deep learning (Goodfellow et al., 2016) employs Neural Networks with many layers (Deep Neural Networks) to model complex patterns in data. Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and transformers are some of the most prominent architectures in deep learning. Since the algorithms developed in the course of this thesis fall in this category, the explanations presented here will be more in-depth than the previous ones.

2.4.1. Feed-forward Neural Networks

A feed-forward neural network (Svozil et al., 1997) is a fundamental type of artificial Neural Network used in Machine Learning and pattern recognition tasks. It is characterized by the

absence of feedback connections between its neurons, meaning the information flows in a single direction—from input nodes through intermediate nodes (hidden layers) to output nodes. The architecture of a feed-forward neural network consists of multiple layers of neurons, typically organized into three types of layers: input layer, hidden layers, and output layer. Each layer transforms the input data through a sequence of operations. The simplest case is when each hidden layer is a Dense, or fully connected layer. The term “Feed-Forward Neural Network” (FFNN) is commonly used to indicate such an occurrence, and that is the case for this thesis also, even though the definition technically has a broader meaning (for example CNNs are also feed-forward neural networks because there is no recursion in their architecture). Here's a breakdown of its components and operations.

- Input Layer: the input features are denoted as $\mathbf{x}$, where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ represents n features. The input layer consists of n neurons, each corresponding to one feature x_i.
- Hidden Layers: A feed-forward NN typically has one or more hidden layers. Let's denote the number of hidden layers as L. Each hidden layer l (where $l = 1,2,\ldots,L$) consists of neurons that perform the operations expressed by Equations 3 and 4:
    1. Compute the weighted sum of inputs:

    $$z_i^{(l)} = \sum_{j=1}^{m^{(l-1)}} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)} \tag{3}$$

    where $z_i^{(l)}$ is the weighted sum for neuron i in layer l, $w_{ij}^{(l)}$ are the weights connecting neuron j in layer $l-1$ to neuron i in layer l, $a_j^{(l-1)}$ is the activation of neuron j in layer $l-1$, $b_i^{(l)}$ is the bias term for neuron i in layer l, and $m^{(l-1)}$ is the number of neurons in layer $l-1$.
    2. Apply an activation function $f^{(l)}$:

    $$a_i^{(l)} = f^{(l)}(z_i^{(l)}) \tag{4}$$

    The choice of activation function $f^{(l)}$ (e.g., sigmoid, ReLU, tanh) introduces non-linearity into the network, enabling it to learn complex relationships in the data.
- Output Layer: The output layer produces the final predictions or outputs of the network. If the network is performing a regression task, the output layer might consist of a single neuron. For a classification task with K classes, the output layer typically consists of K neurons. The output $\hat{y}_i$ (prediction for the i-th output neuron) is computed according to Eq. 5:

$$\hat{y}_i = a_i^{(L)} \tag{5}$$

where $a_i^{(L)}$ is the activation of neuron i in the output layer L.

The functioning of a feed-forward neural network involves the propagation of data through the network in a forward direction. Each neuron in a layer receives inputs from all neurons in the previous layer, computes a weighted sum of these inputs, adds a bias term, and then applies an activation function to produce an output. This output becomes the input to the next layer of neurons, continuing until the final output layer is reached.

Training a feed-forward neural network typically involves three main phases: forward propagation, loss calculation, and backpropagation. During forward propagation, the input data is fed through the network to generate predictions. These predictions are compared with the actual target values to compute a loss function, which quantifies the network's performance. Backpropagation then adjusts the weights and biases of the neurons in the network to minimize this loss function, using optimization techniques such as gradient descent.

1. Forward Propagation: computes the outputs of each layer sequentially from the input layer to the output layer using Equations 3-5.
2. Loss Function: compares the predicted outputs $\hat{\boldsymbol{y}}$ with the actual target values $\boldsymbol{y}$ using a loss function $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$, which measures the difference between predicted and true values.
3. Backpropagation: adjusts the weights (Equation 6) and biases (Equation 7) of the network to minimize the loss. This involves computing gradients of the loss function with respect to the network parameters (weights and biases) and updating them using an optimization technique:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \tag{6}$$

$$b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b_i^{(l)}} \tag{7}$$

where η is the learning rate controlling the size of the updates.

In summary, a feed-forward neural network processes input data through multiple layers of neurons, applying weighted sums and activation functions to produce outputs. It is trained using

forward propagation to compute predictions and backpropagation to adjust parameters, aiming to minimize a specified loss function.

2.4.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) (Gu et al., 2018; Z. Li et al., 2022) are a specialized kind of neural network designed primarily for processing structured grid-like data such as images. CNN is a Neural Network in which at least some of the layers are convolutional. Like the original concept of Neural Networks, CNNs were also inspired by biology: the source was the organization of neurons in the animal's visual cortex.

Convolutional Neural Networks have many advantages over simple FFNNs, the most important being their weight efficiency. The number of weights they use to examine vectors and matrices is much smaller than Fully Connected Networks because each neuron of a convolutional layer is connected with only a small subset of neurons of the previous layer, relying on the assumption that the closer the elements the more correlated they are. CNN are also shift invariant (or space invariant) and their efficacy has been proved by many studies (Z. Li et al., 2022). CNNs have become the cornerstone of modern computer vision, enabling significant advancements in tasks like image recognition, object detection, and image segmentation. Their architecture, moreover, works not only for 2D data, but also for inputs with any number of dimensions, as long as the elements present more correlation with each other the closer they are along any of the axis.

"Convolutional layer" is a type of NN layer which performs the convolution operation. A convolution operation involves a filter or kernel sliding over the input data to produce feature maps. Each filter extracts specific features such as edges, textures, or patterns from the input data. Mathematically, for a 2D convolution, the operation can be expressed with Equation 8:

$$(I * K)(i,j) = \sum_{m} \sum_{n} I(i+m, j+n) \cdot K(m,n) \tag{8}$$

where I is the input image, K is the kernel, and (i,j) denotes the position in the output feature map. This operation helps in capturing spatial hierarchies in the data. After each convolution operation, an activation function is applied elementwise to introduce non-linearity into the model. The most commonly used activation function in CNNs is the Rectified Linear Unit (ReLU) (Apicella et al., 2021; Sharma et al., 2020), defined in Equation 9:

$$\text{ReLU}(x) = \max(0, x) \tag{9}$$

ReLU helps in overcoming the vanishing gradient problem and accelerates the convergence of the training process.

Pooling layers are typically used after convolutional layers to reduce the spatial dimensions of the feature maps, thereby decreasing the computational load and controlling overfitting. They also help in making the representations invariant to small translations and distortions in the input data. The most common form of pooling is max pooling, which takes the maximum value in each window (analogous to the previous kernel) of the feature map to down-sample it (Eq. 10):

$$\text{MaxPooling}(x) = \max_{i \in \text{window}} x_i \tag{10}$$

Figure 1 presents examples with 1-dimensional and 2-dimensional arrays. The windows used have dimension 2 for the first and 2 × 2 for the second.

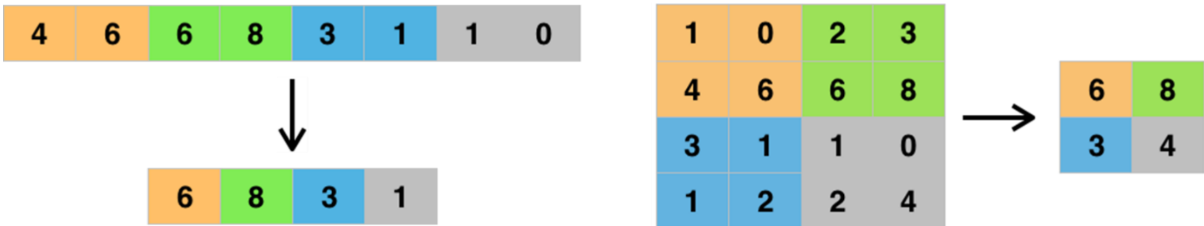

Figure 1: Max Pooling examples (1D left, 2D right).

A typical CNN architecture starts with an input layer, followed by a sequence of convolutional and pooling layers. These layers are often repeated multiple times to form deep networks capable of learning high-level features. Finally, the network ends with one or more fully connected layers, used to combine the extracted features, and an output layer, which provides the final prediction.

CNNs have revolutionized the field of computer vision with their ability to automatically learn hierarchical features from raw image data. They are widely used in applications such as image classification, object detection, face recognition, and medical image analysis. Beyond vision, CNNs have also been adapted for tasks like speech recognition and natural language processing, demonstrating their versatility. One of the key advantages of CNNs is their ability to learn spatial hierarchies of features, making them highly efficient for image-related tasks. They are also less prone to overfitting compared to fully connected networks, thanks to the shared weights in convolutional layers and the reduction in the number of parameters. However, CNNs have some limitations: designing and tuning CNN architectures can be complex and time-consuming, often requiring domain expertise and extensive experimentation.

2.4.3. Transformers and attention mechanisms

Transformers, introduced in the seminal paper "Attention is All You Need" (Vaswani et al., 2017), have fundamentally changed Natural Language Processing (NLP) and found applications in various other domains like computer vision and protein folding.

At the heart of transformers lies the self-attention mechanism which allows them to effectively handle long-range dependencies. This begins with the scaled dot-product attention, which computes attention scores using the queries (Q), keys (K), and values (V). The scores are derived from the dot product of the queries and keys, scaled by the square root of the dimension of the keys, Equation 11:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{11}$$

This computation results in a weighted sum of the values, where the weights indicate the relevance of the corresponding keys to the queries.

Building on this, the transformer employs multi-head attention to capture different aspects of the input. This involves projecting the queries, keys, and values into multiple subspaces and applying the attention mechanism in parallel. Each head performs attention independently, and their outputs are then concatenated and linearly transformed, see Equation 12:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \tag{12}$$

Following the multi-head attention, the transformer applies a position-wise feed-forward network to each position in the sequence independently. This network comprises two linear transformations with a ReLU activation in between, Equation 13 below:

$$\text{FFNN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{13}$$

These feed-forward networks enhance the model's capacity to process complex transformations of the input data.

Transformers, unlike RNNs and LSTMs, do not have an inherent sense of the order of the input sequence. To address this, positional encodings are added to the input embeddings, providing the model with information about the position of each token in the sequence. These encodings are necessary for maintaining order information within the data.

The transformer architecture follows an encoder-decoder structure. The encoder consists of a stack of identical layers, each containing two sub-layers: multi-head self-attention and a position-wise feed-forward network. Each sub-layer is wrapped with a residual connection and followed by layer normalization.

The decoder mirrors this structure but adds an additional sub-layer for multi-head attention over the encoder's output. Thus, each decoder layer comprises three sub-layers: masked multi-head self-attention, multi-head attention over the encoder's output, and a position-wise feed-forward network.

Transformers are typically trained using a cross-entropy loss function for tasks like language modeling and sequence-to-sequence prediction. The optimization process often employs the Adam optimizer with a warmup phase for the learning rate. This means the learning rate increases linearly for a set number of steps and then decreases proportionally to the inverse square root of the step number. To prevent overfitting, techniques such as dropout are used to regularize the model.

Transformers have achieved exceptional success in a wide array of NLP tasks, including machine translation, text summarization, and sentiment analysis. Their impact has extended beyond NLP, with adaptations like Vision Transformers (ViTs) applying the architecture to image classification and other vision tasks. Moreover, transformers have shown promise in fields like speech recognition and protein folding, as demonstrated by models like AlphaFold.

One of the primary advantages of transformers is their ability to capture long-range dependencies without the vanishing gradient problems of RNNs. Additionally, transformers are highly parallelizable during training, making them faster to train on modern hardware compared to RNNs and LSTMs. However, they are computationally expensive, with quadratic complexity concerning the input sequence length. Finally, transformers require substantial amounts of data and computational resources for training, even compared to other DL algorithms.

Transformers have become a cornerstone in deep learning, especially for sequence-to-sequence tasks. Their flexibility and effectiveness in handling long-range dependencies have made them suitable for a broad spectrum of applications. As research continues, improvements and adaptations of the transformer architecture will likely expand its impact across various fields, solidifying its role in the advancement of artificial intelligence.

2.4.4. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) (Salehinejad et al., 2018) are a type of artificial neural network designed to process sequential data, such as time series, speech, and text. Unlike traditional Feed-Forward Neural Networks, RNNs have connections that loop back on themselves, enabling them to maintain a form of memory and capture temporal dependencies. The cyclic nature of their connections makes them fundamentally different from all other networks, in which information always flows from input to output layers. This unique characteristic, in theory, provides RNNs with various interesting capabilities (for example handling tasks where the order of inputs is relevant). It is difficult, however, to set up the task and RNN architecture in such a way as to make use of them.

An RNN operates by taking an input and passing it through a hidden layer, which updates its state based on both the current input and the previous hidden state. This process can be mathematically represented as follows: the hidden state at a given time step is a function of the input at that time step and the hidden state from the previous time step. The output at each time step is then derived from this hidden state (Figure 2).

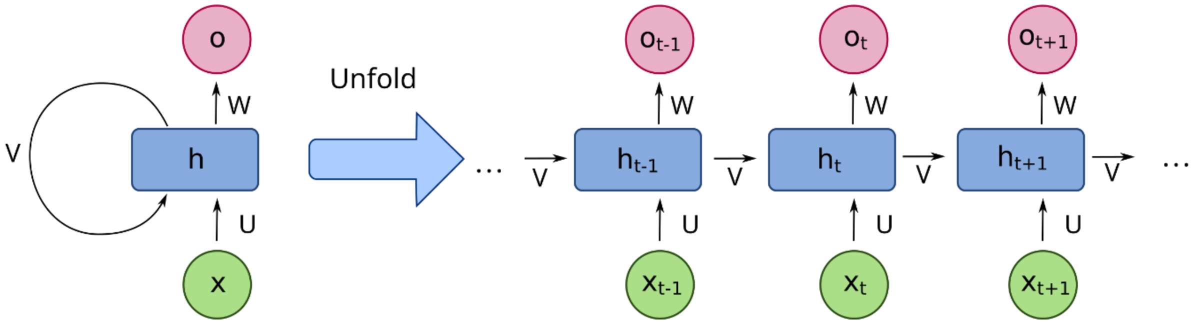

Figure 2: Recurrent Neural Network (RNN) architecture.

There are different types of RNNs, each designed to address specific challenges. The simplest form is the vanilla RNN, which has a single hidden layer. However, vanilla RNNs often struggle with the vanishing gradient problem, where gradients become too small during training, making it difficult to learn long-term dependencies. To overcome this, more advanced architectures like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) were developed (Salehinejad et al., 2018). LSTMs introduce gates to control the flow of information, effectively managing the vanishing gradient problem. GRUs offer a simpler alternative to LSTMs with fewer gates, making them computationally more efficient while still addressing the core issue.

Despite their strengths, RNNs face challenges, particularly in training. The sequential nature of RNNs makes training computationally intensive and time-consuming. Techniques like truncated

BackproPagation Through Time (BPTT) help mitigate these issues by making training more feasible. Another challenge is capturing long-term dependencies, which LSTMs and GRUs are specifically designed to handle more effectively.

In conclusion, Recurrent Neural Networks are powerful tools for modeling sequential data and capturing temporal dependencies. While they come with challenges such as vanishing gradients, advancements like LSTMs and GRUs have made them more effective and widely applicable across various domains.

2.4.5. Transfer Learning and Fine-tuning

It has been demonstrated many times in different contexts that when Deep Neural Networks don't produce good results (and the task was set up appropriately), it is often a matter of using bigger Networks and training them with more data. The drawbacks are that much more training data is required, and the training time increases geometrically, because a NN with more weights is slower and it is trained with more data. However, the more complex the problem, the more likely our only solution is a DNN approach (es. text generation with ChatGPT, image generation with Dall-e, Go board game with AlphaGo Zero...). For very complex problems the quantity of parameters and training data necessary for NNs to reach satisfactory performances is so high that the computations can only be performed in dedicated structures like datacenters - or even cluster of datacenters - over months, which is not something that is available to most researchers.

The solution is Transfer Learning (Mukhlif et al., 2023; Weiss et al., 2016), coupled with fine-tuning (Church et al., 2021). Transfer Learning is a ML technique where a trained model (called "pre-trained model" in this context), initially developed for a different but related task, is repurposed as the starting point for a new task (Figure 3). Instead of training a model from scratch, Transfer Learning leverages the knowledge embedded in an existing model trained on a large dataset, usually from a similar domain. This is possible because the early layers of a Neural Network capture generic features such as edges, textures, and patterns, while the later layers capture task-specific features. By reusing the early layers of a pre-trained model, we can transfer the general features learned from a large and diverse dataset to a new task. This approach significantly reduces the amount of data and computational resources required to train the model. It requires pre-trained models, but many of them have been released publicly online.

Fine-tuning is a specific process within Transfer Learning: the pre-trained model is further trained on the new task's dataset. This involves adjusting the model weights by continuing the

backpropagation process, but typically with a lower Learning Rate to make finer updates. The goal of fine-tuning is to adapt the pre-trained model to the specific nuances of the new task, ensuring better performance and accuracy.

The last layer (or layers) of a NN is called "decision layer" because it receives the elaboration of the input performed by all the previous layers and generates the output. It also holds the information about the expected output shape. Substituting this layer with a new one before fine-tuning the model, ensures the new decision layer learns the correlation between the elaborated data and the final output (see Figure 3, right downward arrow).

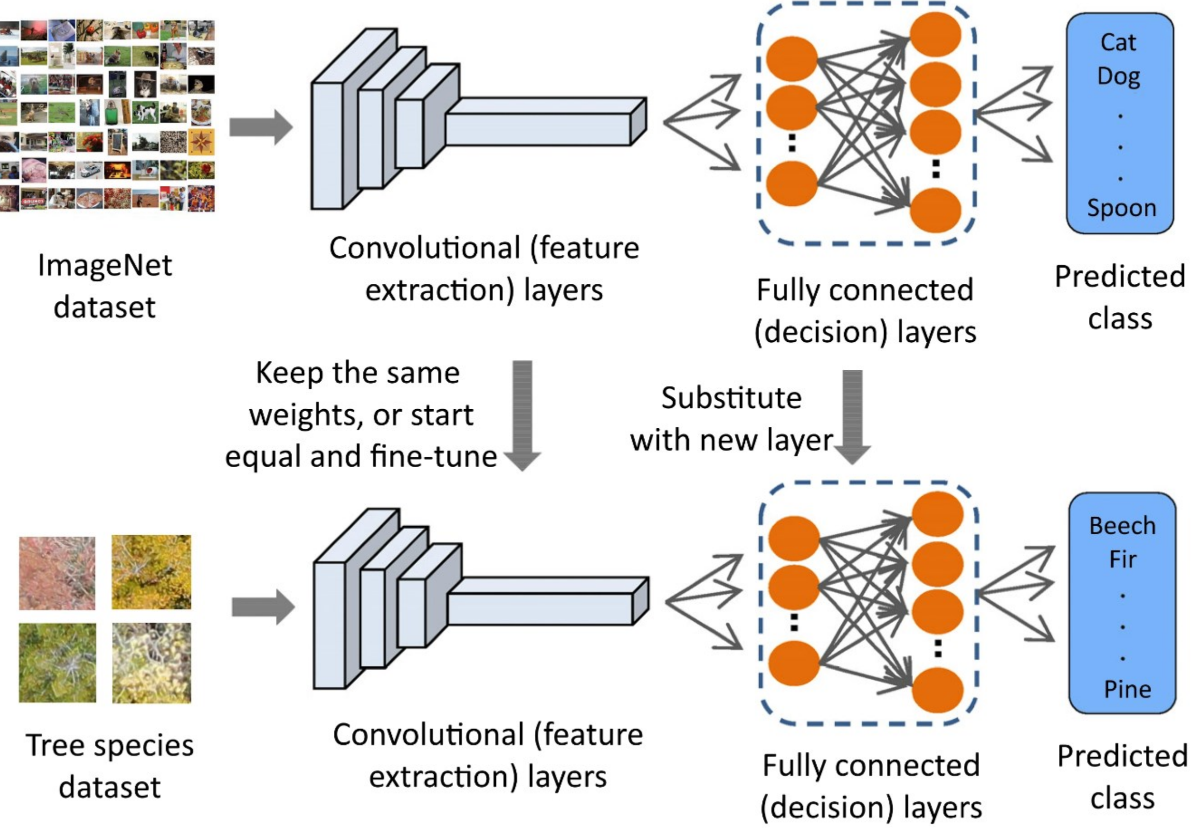

Figure 3: transfer learning example. It is the procedure employed in the tree species recognition and the last version of the landslide forecasting algorithms. Image adapted from (Mukhlif et al., 2023).

Training models with a very big and generic dataset and only afterwards fine-tuning them with the relatively few task-specific data, often results in the NNs learning more robust internal representations and reaching higher final accuracy.

All the pretrained models used in this study (for both landslide forecasting and tree species identification) were trained on the ImageNet dataset (Deng et al., 2009) (https://www.image-net.org/).

"ImageNet is an image dataset organized according to the WordNet hierarchy. Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "synset". There are more than 100,000 synsets in WordNet; the majority of them are nouns (80,000+). In ImageNet, we aim to provide on average 1000 images to illustrate each synset. Images of each concept are quality-controlled and human-annotated. In its completion, we hope ImageNet will offer tens of millions of cleanly labeled and sorted images for most of the concepts in the WordNet hierarchy."

ImageNet has become a well-established standard for training or pretraining image-based DL models, with its vast number of images and different categories manually labeled. They also introduced a yearly challenge (Russakovsky et al., 2015) based on this dataset, to monitor the development of Computer Vision through the years.

2.5. Emerging Trends

Recent trends in machine learning include federated learning, explainable AI (XAI), and quantum machine learning. Federated learning enables collaborative model training across decentralized devices while preserving data privacy. XAI focuses on making AI systems transparent and interpretable, addressing the "black box" nature of deep learning models. Quantum machine learning explores the integration of quantum computing with ML to potentially solve complex problems more efficiently.

2.6. ML in the Geotechnical field

Machine learning (ML) applications in the geotechnical field are diverse and have significantly enhanced various aspects of geotechnical engineering by improving prediction accuracy, decision-making processes, and efficiency. Some notable applications will be discussed in the following sections.

2.6.1. Soil Classification and Property Prediction

Machine learning models can be used to classify soil types and predict their properties based on input data such as grain size distribution, moisture content, and other physical characteristics. Techniques such as Support Vector Machines (SVM), Random Forests, and Neural Networks can be trained to predict soil properties like shear strength, permeability, and compressibility from laboratory and field data.

2.6.2. Settlement Prediction and Foundation Design

Accurate prediction of ground settlement due to construction activities, such as excavation, tunneling, or loading, is crucial. ML algorithms can be trained on historical settlement data to predict future settlement behavior. This also aids in optimizing foundation design by predicting the load-bearing capacity and settlement of different foundation types (e.g., shallow foundations, pile foundations). Models trained with historical foundation performance data help in selecting the appropriate foundation type and design parameters for specific site conditions. Algorithms like Neural Networks (NNs) and Regression Models are employed to improve prediction accuracy over traditional empirical methods.

2.6.3. Underground excavation and Support Design

In tunnel engineering, ML models predict ground behavior and stability during and after excavation. These predictions help in designing support systems such as shotcrete, rock bolts, and lining. Models analyze geological conditions, excavation methods, and support performance to optimize tunnel safety and work efficiency.

2.6.4. Geotechnical Instrumentation and Monitoring

ML techniques enhance the interpretation of data from geotechnical instrumentation (e.g., inclinometers, piezometers, strain gauges). They are able to objectively analyze measurements and identify patterns, anomalies, and trends. Those algorithms can process very large amounts of data in real time for time-critical applications such as Early Warning, allowing for timely intervention and mitigation.

2.6.5. Slope Stability and Landslide forecasting

Landslide forecasting using machine learning has undergone significant advancements, largely due to the integration of diverse data sources and the development of sophisticated algorithms. This innovative approach leverages both historical and real-time data to predict landslide occurrences, providing critical information for disaster management and risk mitigation.

At the core of landslide forecasting lies the integration of various types of data. Topographical data, including Digital Elevation Models (DEMs) that provide information on slope, aspect, and elevation, are fundamental. Steeper slopes are more prone to landslides, making slope gradient a critical factor. Geological data, such as soil type and properties, and lithology, which studies the physical characteristics of rocks, are also important. Vegetation data, including land cover types and the health and density of vegetation assessed through metrics like the Normalized

Difference Vegetation Index (NDVI), affect soil cohesion and slope stability. This data helps understand the inherent stability of the ground.

Hydrological data plays a crucial role as well, since rainfall is a primary trigger for landslides. Soil moisture and groundwater levels further influence slope stability. For instance, higher moisture content can indicate soil saturation, increasing the risk of landslides. Seismic data, which records earthquake activity and ground shaking intensity, is another potential trigger, in already susceptible areas. Finally, anthropogenic data, such as land use changes due to urbanization, deforestation, and infrastructure development, can destabilize slopes and increase landslide risk.

Real-time monitoring and early warning systems have become increasingly sophisticated. Sensor networks comprising IoT devices monitor soil moisture, rainfall, and ground movement, feeding data into machine learning models for real-time risk assessments. Satellite-based systems, using data from satellites like Sentinel-1 and Sentinel-2, monitor changes in land cover and soil moisture, integrating this information with machine learning models to provide early warnings.

Geographic Information Systems (GIS) are used to visualize landslide susceptibility maps, integrating various data layers and model outputs. This aids in spatial analysis and decision-making for urban planning and disaster management. Moreover, quantifying uncertainty in predictions is another important factor for effective risk management. Ensemble methods and Bayesian Neural Networks are employed to estimate confidence levels, aiding in risk assessment and decision-making under uncertainty.

However, challenges remain. High-resolution and accurate data are needed for training reliable models, and efforts are ongoing to improve data collection methods and integrate diverse data sources. Ensuring that models trained in one region can generalize to other regions with different geological and climatic conditions remains a challenge. Transfer learning and domain adaptation techniques are being explored to address this issue. Interdisciplinary approaches that combine machine learning with traditional geotechnical and hydrological models leverage the strengths of both methods, enhancing model robustness. Additionally, developing systems capable of processing real-time data and providing timely alerts requires advancements in hardware and software infrastructure. Cloud computing and edge computing are potential solutions for handling large-scale, real-time data processing.

Here are a few examples of studies that, while addressing the same subject, differ significantly in terms of data, methodology or objectives. These examples illustrate how different approaches to the same problem can yield diverse procedures and results.

"Landslide Displacement Prediction of Shuping Landslide Combining PSO and LSSVM Model" (Jia et al., 2023) studies the Three Gorges Reservoir, created by the Three Gorges Dam, a massive Chinese project for reducing Yangtze River's floodings and generate hydroelectric power. When the water level is at its maximum of 175 m above sea level, the dam reservoir is on average about 660 km in length, 1,12 km in width and contains 39.3 km^3 of water. Due to environmental characteristics and constant water infiltration and weight, there have been many landslides and other concerns since its construction.

The paper analyzes the Shuping landslide and tries to predict the cumulative displacement (CD) by decomposing it: $CD = TTD + PTD + RTD$, where $TTD =$ Trend Term Displacement controlled by the geological conditions, $PTD =$ Periodic Term Displacement controlled by external factors, $RTD =$ Random Term Displacement. This approach is appropriate in this case because reservoir water level and seasonal rainfall have a very significant impact on displacement, and they exhibit periodic regular fluctuations in the Three Gorges Reservoir. Thus, the paper differs in the way it is calculates the displacement: it uses two different sets of data for predicting the two main components of CD (geological conditions for TTD, rain and water level for PTD) and combine them to compute the CD prediction.

"Landslide hazard and susceptibility maps derived from satellite and remote sensing data using limit equilibrium analysis and machine learning model" (Dashbold et al., 2023) uses satellite and remote sensing data to generate Landslide Susceptibility Mapping (LSM) and Landslide Hazard Mapping (LHM) of a region. LSM expresses landslide risk over a large area (regional scale in the paper), it is calculated with a logistic regression model and the risk is the probability of a landslide occurring. LHM works on a local scale, but it contains temporal information (time of the expected occurrence of the landslide). LHM is calculated with the limit equilibrium factor of safety (Equation 14) (Lu & Godt, 2008):

$$\mathrm{FS} = \frac{\tan(\phi')}{\tan(\beta)} + \frac{2c'}{\gamma H_{ss}\sin(2\beta)} + \frac{\sigma^s}{\gamma H_{ss}}[\tan(\beta) + \cot(\beta)]\tan(\phi') \quad (14)$$

where FS is the factor of safety; ϕ' is the soil friction angle; c' is the effective soil cohesion; β is the slope angle; γ is the soil total unit weight; H_{ss} is the depth to bedrock; σ^s is the suction stress. The study combines the two maps by first generating the LSM and using its information to find

locations where to calculate LHMs. The model was validated with data from northern Kentucky where several landslides were registered, and the necessary data was available. In this case the output of the model are LSMs and LHMs instead of future displacements, and input data are also different.

"Wadenow: A Matlab Toolbox for Early Forecasting of the Velocity Trend of a Rainfall-Triggered Landslide by Means of Continuous Wavelet Transform and Deep Learning" (Teza et al., 2022). In this paper the authors convert displacements and rainfall data into scalograms with Continuous Wavelet Transform (CWT) (Aguiar-Conraria & Soares, 2014; Rioul & Duhamel, 1992), then generate a single image from the two scalograms (see Figure 4 as an example). Equations 15 and 16 describe the most important concepts of wavelets and this signal transformation. A "mother" wavelet is a function $\psi(t) \in L^2(\mathbb{R})$ which respects the condition $0 < C_\psi < \infty$, with

$$C_\psi = \int_{-\infty}^{\infty} \frac{|\Psi(\omega)|}{|\omega|} d\omega \tag{15}$$

Now that we defined $\psi(t)$, the CWT of a time series $x(t)$ is

$$X_\omega(a,b) = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} x(t) \overline{\psi}\left(\frac{t-b}{a}\right) dt \tag{16}$$

With $\overline{\psi}$ complex conjugate of ψ. a and b represent scaling and translation factors respectively.

A CNN is used for learning the relationship between images and future velocity trends. The velocity trends are three velocities (low, mid, high) plus the four transitions between those (low to mid, mid to high, high to mid, mid to low).

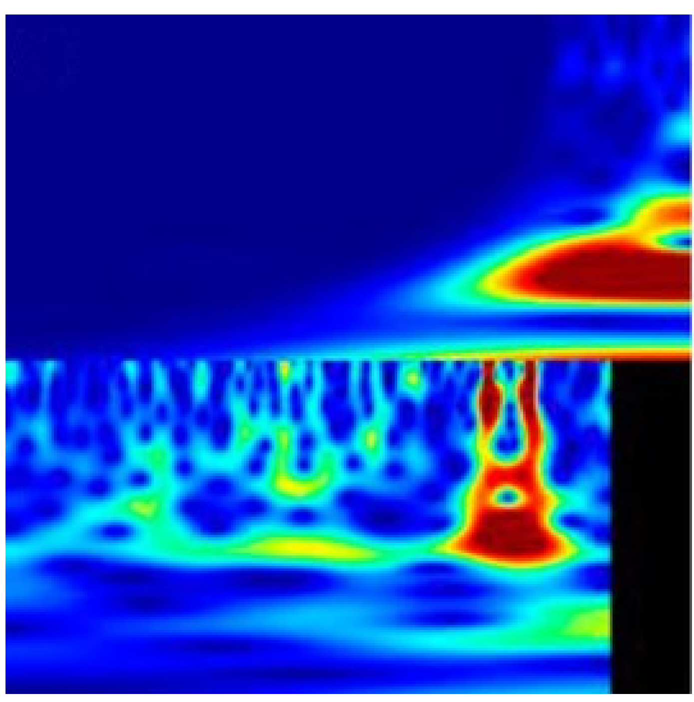

Figure 4: input image composed of a rainfall scalogram (upper half) and velocity scalogram (lower half). Source: (Teza et al., 2022).

The CNN is a pretrained VGG19 model (Simonyan & Zisserman, 2015), and they test two versions: one using displacements and rainfall, one using only rainfall data.

"Impact of Vegetation Differences on Shallow Landslides: A Case Study in Aso, Japan" (Asada & Minagawa, 2023) studies the relationship between vegetation coverage and landslide occurrences. The algorithms (generalized linear model and random forest) use many consolidated parameters to assess slope failure risk, like elevation, slope angle, slope aspect, undulation, topographic wetness index, and geology. They introduced vegetation among the primary factors and noticed that not only the quantity was relevant, even forest structure and type of vegetation had significant impacts on the predictions.

The study focuses on shallow landslides triggered by rainfall. The authors trained and tested algorithms with data from Aso region's caldera, which was historically covered by grassland but has seen artificial forests planted since 1960. The area was divided into slope units (Asada et al., 2020), so that they also had the negative cases where landslides didn't occur.

The results showed that "the probability of shallow landslides in secondary grasslands was approximately three times that of coniferous and broadleaf forests, and approximately nine times that of broadleaf secondary forests. The landslide probability of shrubs was approximately four times that of coniferous and broadleaf forests, and approximately ten times that of broadleaf

secondary forests" (Asada & Minagawa, 2023). Which confirms that vegetation coverage and species have both influence over slope stability.

3. Data

3.1. Landslide forecasting

3.1.1. Study sites

The data available for training and testing the landslide forecasting algorithms are collected over many years from four different sites (called A, B, C, and D from now on). The monitoring data and site names cannot be disclosed, but their relevant characteristics will be described in this section. For the earlier versions of the algorithm only the first of the four case studies was available, which reduces training data, but simplifies the problem (the test data should be very similar to the training data). After all four datasets were available, a function for reading data from multiple sources was added to the algorithm.

From the four datasets combined, after cleaning and formatting the data, ~250,000 observations can be extracted. With the setup used for most of the later versions of the algorithm (shift thresholds = [0.60, 0.85], data balancing = True, data augmentation = 5), and reserving 20% of data for validation and test purposes, the available training observations become ~1,800,000.

Table 1: Number of observations collected from each site. This is with 12-days observations (the most common choice in the experiments) and after removing missing or erroneous instrument readings.

Site A	Site B	Site C	Site D	Total
104,456	98,715	13,620	30,732	247,523

Site A: Landslide Monitoring in Northern Italy

The first monitoring site (Valletta, 2022) is situated in a region of Northern Italy, where a major construction project for a new transport infrastructure is underway. This area is characterized by a complex landscape of mountains and hills, which raises concerns about potential environmental impacts. To address these concerns and ensure that the new infrastructure interacts safely with its surroundings, the site has been equipped with a comprehensive monitoring system, utilizing advanced MUMS instrumentation.

Specifically, four Vertical Arrays (VA) were strategically placed across various parts of the area to monitor key factors that could affect the stability of the site. These VAs track ground displacements, variations in water levels, and fluctuations in atmospheric pressure, providing critical data for assessing the site's stability. Each Vertical Array was custom designed to suit its location, incorporating a unique combination and number of sensors based on the overall monitoring strategy detailed in the site's plan. The specific characteristics of each device installed at the site are outlined in Table 2.

Table 2: site A's MUMS composition.

Array ID	**Installation date [dd/mm/yyyy]**	**Sensors number and typology**	**Length [m]**
DT0099	06/03/2019	20x Tilt Link HR 3D V 2x Piezo Link	20
DT0100	05/12/2018	20x Tilt Link HR 3D V 2x Piezo Link 1x Baro Link	20
DT0101	06/03/2019	20x Tilt Link HR 3D V 1x Piezo Link 1x Baro Link	20
DT0102	06/03/2019	20x Tilt Link HR 3D V 1x Piezo Link	20

The Vertical Arrays were installed between December 2018 and March 2019, marking the beginning of the continuous monitoring effort. To this day, the monitoring system remains active, capturing and recording data six times daily, offering a high-resolution insight into the evolving conditions of the landslide-prone area.

Site B: landslide in Southern Italy

This case study (Valletta, 2022) details the monitoring activities conducted on a slope in Southern Italy, where a high-speed railway tunnel was being constructed. After identifying a dormant landslide within the area, it was decided to implement an automatic monitoring system. This system was designed to track any deformations that might occur as a result of the ongoing excavation activities related to the tunnel's construction.

The monitoring system employed in this region comprises four Vertical Array MUMS inclinometers, each with different specifications depending on the unique requirements of the location. The inclinometers range in length from 30 to 80 meters and contain between 31 and 81 3D MEMS sensors. The spacing between the sensor nodes also varies according to the vertical positioning: in the presumed stable zones, the distance between nodes is set at 2 meters, while in the proximity of the sliding surface, the distance is reduced to 0.5 meters to ensure more precise measurements.

In addition to the inclinometers, the system incorporates four Piezo Arrays, each composed of two analog piezometers. These devices are used to monitor changes in water levels over time, which could potentially impact the slope's stability. A more detailed breakdown of the characteristics of each Array is available in Table 3.

Table 3: site B's MUMS composition.

Array ID	Installation date [dd/mm/yyyy]	Sensors number and typology	Length [m]
DT0004	18/02/2020	2x Piezo Link	45
DT0111	23/01/2020	73x Tilt Link V	69
DT0005	18/02/2020	2x Piezo Link	31
DT0112	24/01/2020	81x Tilt Link V	80
DT0006	18/02/2020	2x Piezo Link	52
DT0113	14/01/2020	66x Tilt Link V	80
DT0007	22/01/2020	2x Piezo Link	30
DT0114	24/01/2020	31x Tilt Link V	30

Throughout the monitoring period, the data collected by the Vertical Arrays indicated a consistent level of activity at the site, suggesting that the slope remains dynamically engaged. However, no critical instabilities or alarming deformations have been detected within the area of concern.

Site C: retaining wall on the French Alps

This case study (Segalini et al., 2019) focuses on monitoring a 12-meter-high geogrid-reinforced retaining wall located in the French Alps (Figure 5). The wall showed signs of instability, prompting displacement measurements using manual inclinometer probes and topographic surveys. Based on these findings, reinforcement work was undertaken, and an automated monitoring system was installed. The system, using Modular Underground Monitoring System (MUMS) technology,

included two 15-meter-long inclinometer arrays equipped with multi-parametric sensors to track displacements, pore pressure, and temperature (see Table 4 and Figure 5). The system allowed remote control and data transmission to a centralized elaboration unit, where raw data were processed into physical measurements.

Table 4: site C's MUMS composition.

Array ID	Installation date [dd/mm/yyyy]	Sensors number and typology	Length [m]
DT0080	29/08/2017	15 Tilt Link HR 3D V 1 Piezo Link 1 Baro Link	15
DT0081	29/08/2017	15 Tilt Link HR 3D V 1 Piezo Link 1 Therm Link	15

The installed Vertical Arrays comprised Tilt Link HR 3D V sensors, piezometers, and additional sensors, placed at different depths to monitor movements. A displacement threshold was set for early warning procedures, with an alert triggered if a displacement of 1 mm/day was exceeded over seven consecutive days. Data were collected for 17 months until 08/01/2019. Although no alerts were triggered, critical structural movements were detected, particularly significant displacements at depths of 5 meters in one VA and at both 5 and 14 meters in the second Array. For this reason, the wall was demolished.

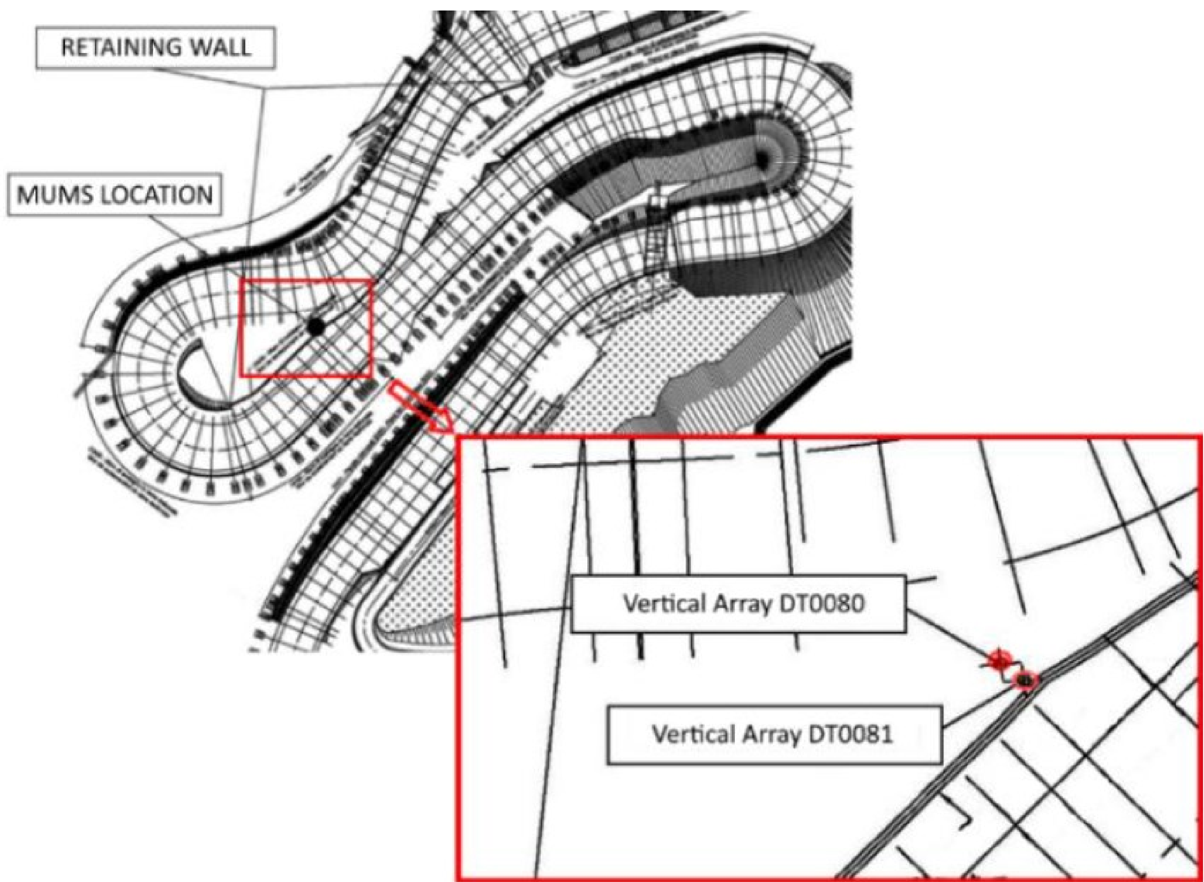

Figure 5: retaining wall and monitoring devices placement. Source: (Segalini et al., 2019).

Site D: landslide in the Apennines, Italy

This landslide (Bertolotti, 2022) is located in the municipality of Corniglio, within the Parma Apennines at an altitude of 788 meters above sea level.

The landslide is part of a geologically sensitive area characterized by a complex interaction between ancient landslide deposits and deep-seated gravitational slope deformation (DGSD). These deposits mainly consist of fine-grained debris, such as pelitic and sandy material, with embedded rock fragments. Historically, the northern slope has been subject to active landslides, and it has been the focus of consolidation efforts in the past.

The landslide itself extends in a southwest-northeast direction, spanning about 140 meters in length and 175 meters in width, covering an area of approximately 18,000 square meters (Figure 6). It is classified as a complex and composite phenomenon, featuring different sections with rotational-translational movements, rockfalls, and debris flows that converge into a river at the foot of the slope.

The area's geology is dominated by the Monte Caio Flysch Formation, which consists of heavily fractured calcareous-marly rocks that retain their stratigraphic coherence despite being affected

by tectonic stresses. This structural weakness, combined with continuous erosion by the flow of the river, has destabilized the slope over time. The river's erosion at the base of the slope has weakened the material that supports the hillside, contributing to the landslide's occurrence.

The landslide, which activated in May 2019, posed a significant risk and required urgent intervention. A monitoring system was implemented to assess the situation and quantify the risks. Historical records from the region's landslide inventory reveal several similar events in the area, with notable occurrences in 1997 and 1999, both linked to the erosive action of the river at the base of the slope. These past events highlight the ongoing vulnerability of the area to landslide hazards.

The MUMS system features three different Arrays installed on-site to monitor various parameters, including ground tilt, water pressure, and structural cracks (see Table 5 and Figure 6).

Table 5: site D's MUMS composition.

Array ID	**Installation date [dd/mm/yyyy]**	**Sensors number and typology**	**Length [m]**
DT0103	19/07/2019	29x Tilt Link HR 3D V 1x Piezo Link 1x Baro Link	30
DT0003	19/07/2019	1x Klino Link HR 3D	-
DT0002	19/07/2019	2x Crack Link	-

Figure 6: Measurement devices placement. V = DT0103; A = DT0003 + 1 Crack Link of DT0002; K = 1 Crack Link of DT0002. Source: (Bertolotti, 2022).

3.1.2. Data structure

The state of a site in a particular moment is represented by time series of physical characteristics (displacements, rain, soil water level...) measured by sensors in the ndi (Number Days Input) days preceding said moment (see Table 6 as an example).

Table 6: example of input data structure.

		Inclinometer			**Piezometer**		**Barometer**
		x displacement	**y displacement**	**Depth**	**Water level**	**Depth**	**Pressure**
Days	**1**						
	2						
	3						
	...	...					
	ndi						

The data represented in Table 6 can be referred to as “observation”, because if is the state of the world captured by the sensors, and “input”, because it is the format of input data expected by the Neural Networks used in this study. Note that certain values have a depth associated with them,

because, for example, MUMS have many inclinometers that are all in the same location when viewed from above, but different depths. The depth information is important because it allows to find the height of the sliding surface.

The NN takes as input the site state and outputs the displacement predicted for the ndo (Number Days Output) days $ndi + 1, ndi + 2, \dots\ ndi + ndo$, with $1 \leq ndo \leq ndi$ (Table 7).

Table 7: example of output data structure.

		Inclinometer	
		x displacement	**y displacement**
Days	$ndi + 1$		
	$ndi + 2$		
	....	...	
	$ndi + ndo$		

ndi was often kept at a value of 12. For this reason, to simplify tables and explanations, it will be often substituted by this number (Table 8). In all the experiments conducted, for simplicity, ndo was always kept at 1, so that the NN predicts only the single day following the last observed day (Table 9).

Table 8: input data structure.

		Inclinometer			**Piezometer**		**Barometer**
		x displacement	**y displacement**	**Depth**	**Water level**	**Depth**	**Pressure**
Days	**1**						
	2						
	3						
	...	...					
	12						

Table 9: output structure used.

		Inclinometer	
		x displacement	**y displacement**
Days	**13**		

Data in the above format are called “output” when they are produced by the NN. Data in the same format are also extracted from the time series, one for every observation, and they represent the true evolution of the site that the NN should predict from the corresponding observation. In this second case they are called “targets”.

The columns of Table 6-8, in ML terminology, are called features: the properties of the input that the NN will learn to associate to output values. Similarly to *ndi* and *ndo*, the number of columns of inputs and outputs are referred to as *nfi* (Number of Input Features) and *nfo* (Number of Output Features). This is useful because the number of input and output days (and in later version even the number of features) can be modified by the user while input and output layer of a NN cannot be changed once the model has been created. For this reason, the algorithm must dynamically create the NN appropriate for the settings chosen by the user.

3.1.3. Sensors and data collection

From the data available for this study, the sensors used, and their respective outputs are:

- Inclinometer
    - x displacement
    - y displacement
    - depth
- Piezometer
    - soil water level
    - depth
- Barometer
    - atmospheric pressure
- Rain gauge
    - Precipitation

Inclinometers and piezometers are part of an instrument called MUMS (Modular Underground Monitoring System, Figure 7) (Segalini et al., 2014) hence every measurement produced by one of the sensors is also associated with the depth of the sensor with respect to ground level (Figure 8). MUMS are composed of a series of IP69 sealed units linked together with a Kevlar cable for support and connected with a four-core signal cable for data transmission. Each node contains an inclinometer (high-resolution 3D MEMS sensor, Figure 7 right) and the number of such units and their spacing can be adjusted according to site characteristics and project requirements.

Other sensors can be added to the nodes, like piezometers or thermometers. For an example of real MUMS monitoring setups see Table 2 - Table 5 from the "3.1.1. Study sites" Section.

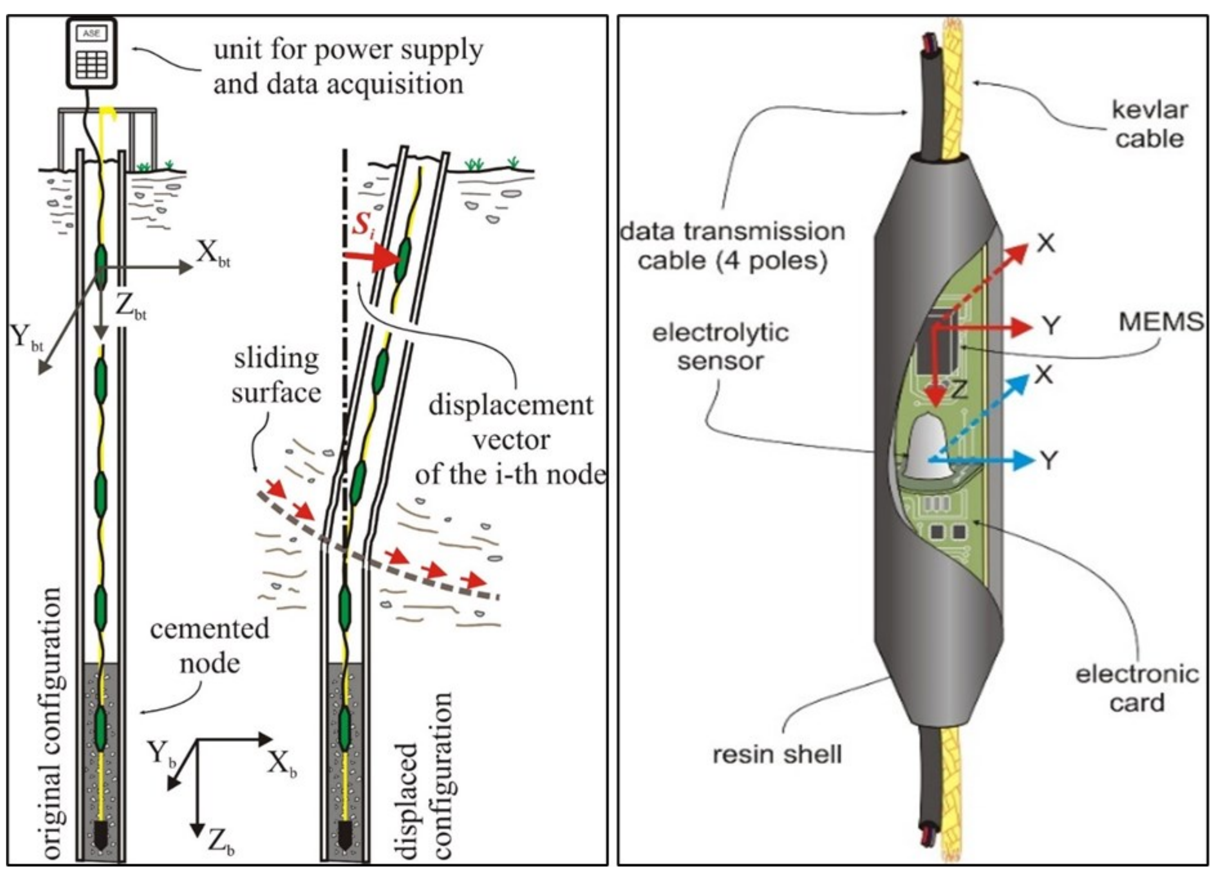

Figure 7: MUMS displacement calculation (left) and composition of individual inclinometer (right).

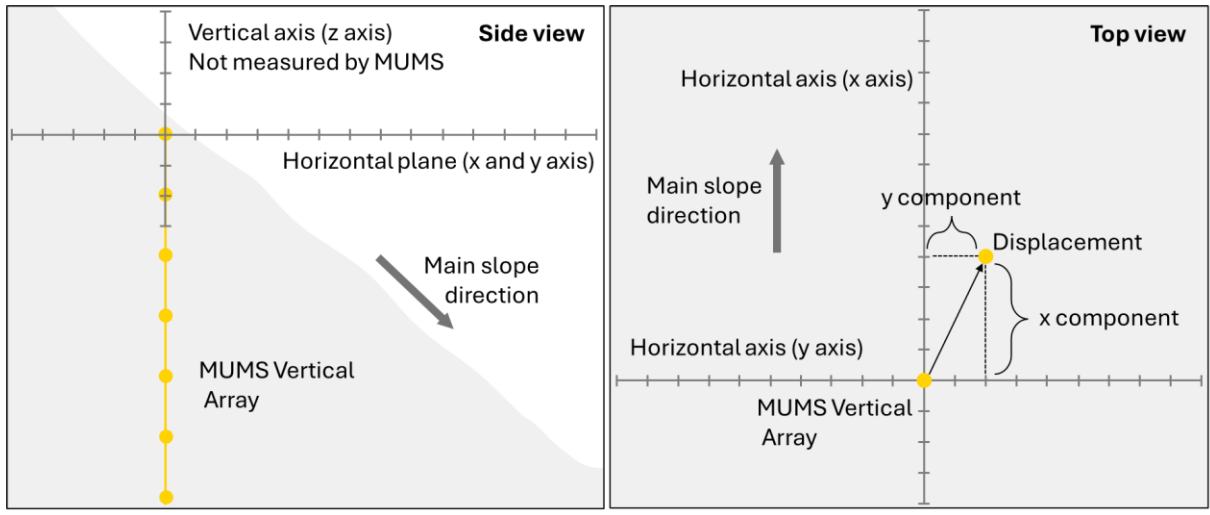

Figure 8: MUMS Measurement axis, side view (left) and top view (right).

X and y displacements are the movements that occurred between the last and current measurements. They are horizontal displacements calculated along two perpendicular vectors,

typically the main slope direction or the magnetic north direction and its orthogonal direction on the horizontal plane (Figure 8).

The individual sensors inside the Vertical Array used to calculate the displacements are inclinometers, which can directly measure only their inclination with respect to the vertical direction (z axis). Terrain movements in the x-y horizontal plane causes inclinometers at different heights to measure different inclination levels (Figure 7 left) and this information can be used to calculate backwards the displacement and its depth.

Piezometers are normally one (sometimes two) per Vertical Array. If more than one is present in a VA, the algorithm finds the piezometer with the least errors and missing data in its history and discards all the others. Then its measurements are broadcasted to all the Vertical Array inclinometers. Barometers are also present on site, but normally there is only one, or at least less than one per VA. For this reason, the algorithm finds the barometer with the least errors and missing data in its history and discards all the others. Its measurements are then broadcasted to all the inclinometers of all the MUMS of the site. There are no Rain gauges in the study sites, rainfall data was obtained from publicly available records of nearby meteorological stations.

- Site A: https://ambientepub.regione.liguria.it/SiraQualMeteo/script/PubAccessoDatiMeteo.asp
- Site B: http://www.agricoltura.regione.campania.it/meteo/archivio_meteo.html
- Site C: pieced together from multiple sources.
- Site D: https://simc.arpae.it/dext3r/

The model generated by this framework is agnostic to the type of input information it can use to produce a prediction, it will try to find correlations between examples of input-output pairs without explicit knowledge of what those numeric data represent in the physical world. Once the input format has been established, however, it must remain consistent. This means that even if not all sensors are available, the model can be trained with only a subset of the above list, and more or different sensors can be used, as long as the data they produce can be represented with time series. The only limitation is that the format of data used during the training phase becomes the only one accepted by the model to subsequently generate predictions. In theory the more information the model knows about the site, the better it can predict the future displacement, but using input with many different sensors' data has serious drawbacks in practice: first the larger and more complex the input data are, the bigger the network needs to be to be able to learn useful patterns. Second, if part of the input has low or no correlation with the output it can make the relationship between input and output harder to learn for the model, because it introduces noise.

Both problems significantly increase the quantity of training examples needed to obtain the same results, and their effect compounds geometrically. Finally, the richer and more specific the input data are, the less training data is available: for example, it is possible to find many public records of displacement time series for landslides, or the precipitation in the landslide area in the required period. It is not so common, however, to have information like soil water level or atmospheric pressure, and the more uncommon data are used, the more difficult it becomes to find enough training data. Once trained, the model can then be used only for predictions on sites where all the required information is known. For these reasons, it is worthwhile to also test models that only use a subset of the available sensor data, trying to keep only the most relevant information.

Many different implementations of the landslide forecasting algorithm were tested, with different types and combinations of input/output data.

3.1.4. Data preprocessing

Data preprocessing is important in general, and fundamental when dealing with deep learning algorithms, because they are very sensitive to the format choice for the input. The software developed includes functions to read the data from file and prepare it for training and testing purposes.

The preprocessing sequence consists of the following operations:

1. From every input file, which corresponds to one site, the available sensors are found.
2. The algorithm computes the subset of sensors that are present in every site.
3. All data from sensors outside the common subset is discarded.
4. Optionally, the first and last days of every time series is removed because it is often incomplete.
5. Precipitation data can be shifted by *rds* (Rainfall Days Shift) days with respect to the other time series because of their delayed effect on displacement.
6. It is assumed that the sampling frequency is at least one in a 24-hour period. If it is higher than that, the data is aggregated to have exactly that frequency.
7. From user-defined displacement categories, thresholds are computed which divide displacement data in the required way.
8. The time series of every site are separated into lists of observations (the format of Table 6 and Table 8 expected by the Neural Network) and corresponding targets (Table 7 and Table 9).

9. Each observation-target pair is checked and any that contains too many missing or corrupted values is discarded.
10. All the inputs from every site are aggregated into a single dataloader.
11. Data are optionally shuffled.
12. The dataloader is split into training, validation, and test dataloaders according to user-defined proportions (by default it is respectively 80%, 10%, 10%).
13. Data are normalized. This proportionally scales the values to fit in the [0, 1] interval.
14. Data balancing and data augmentation are optionally performed, only on the training data.

The algorithm has been created to be as general as possible, which is why it can use data from multiple sites and tries to find all the expected measurements in the data, but it adjusts automatically to any missing device and creates a NN that does not expect that as input (points 1-3).

The influence of rainfall on landslides has been studied and it is understood to be a very important factor in its evolution (Kuradusenge et al., 2020; Teza et al., 2022; Zhu et al., 2017), but it is also known that depending on many factors, the effect of precipitation on a site can manifest with a significant delay. Point 5 in the previous list allows us to manually adjust the data and account for this behavior: by aligning the precipitations with their effect on the site, the NN can more easily identify the correlation between cause and effect.

The metrics used and some of the versions of the algorithm tested have to divide the displacements into categories. Since the division is arbitrary, the user can choose and modify this parameter at any time. The division is based on the intensity of the displacement, and the default categories are low displacements (bottom 84% of the data), mid displacements (data in the range 84%-97%), and high displacements (top 3% of data). It is possible to change not only the range of each interval, but also the number of intervals. For a more fine-grained division it is possible to have five, or even ten categories. The thresholds are a list of integer numbers which represent separators for the data, so that, for example, two thresholds separate the data into 3 categories (see Figure 9).

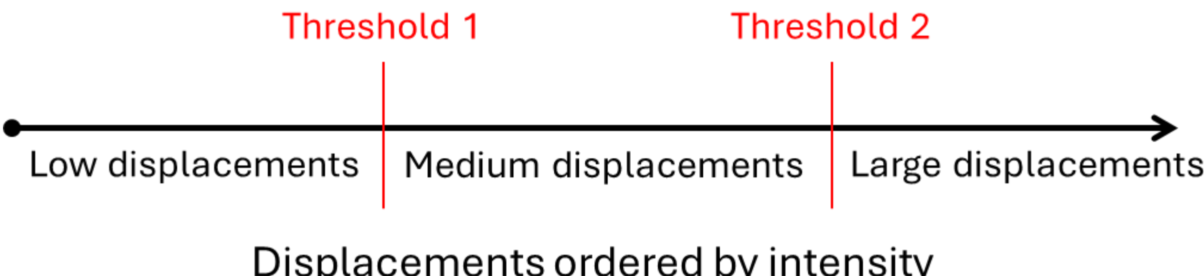

Figure 9: example of dataset subdivision. Two thresholds separate data into three categories.

Since it is impossible to know a priori the range of values of a landslide dataset, the thresholds are not distances directly compared to the measured displacements. Each threshold represents the distance in number of standard deviations (or σ) from the center of the distribution. This requires measurements to be normally distributed, which is a very reasonable assumption. As an example, shift thresholds = [0, 2], means that the first threshold is 0 standard deviations above the mean (which is exactly the mean), while the second threshold is 2 standard deviations above the means. The data would be divided into the 3 categories listed below and in Figure 10.

1. Low displacements: the lower 50% of the measurements (below threshold 1).
2. medium displacements: measurement between 50% and 97% (between threshold 1 and threshold 2).
3. High displacements: the top 3% of the measurements (above threshold 2).

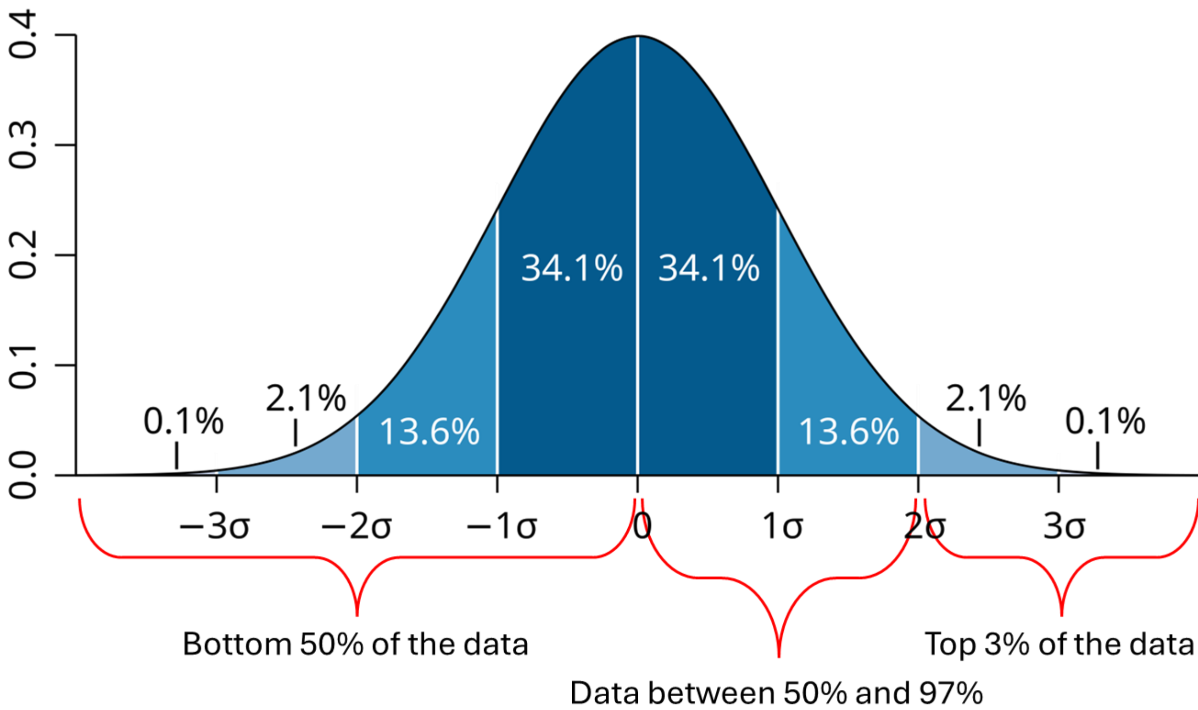

Figure 10: example of dataset subdivision with thresholds based on standard deviation instead of absolute values.

This method of division has the advantage of being independent from the values of the measurement of any particular dataset. In later versions, the thresholds can be given directly as percentages, which is much more intuitive, and the algorithm handles the necessary conversion and computations.

From a single time series, the observations and targets created in point 8 can be partially overlapping to maximize the information extracted from the measurements, as shown in Figure 11 below.

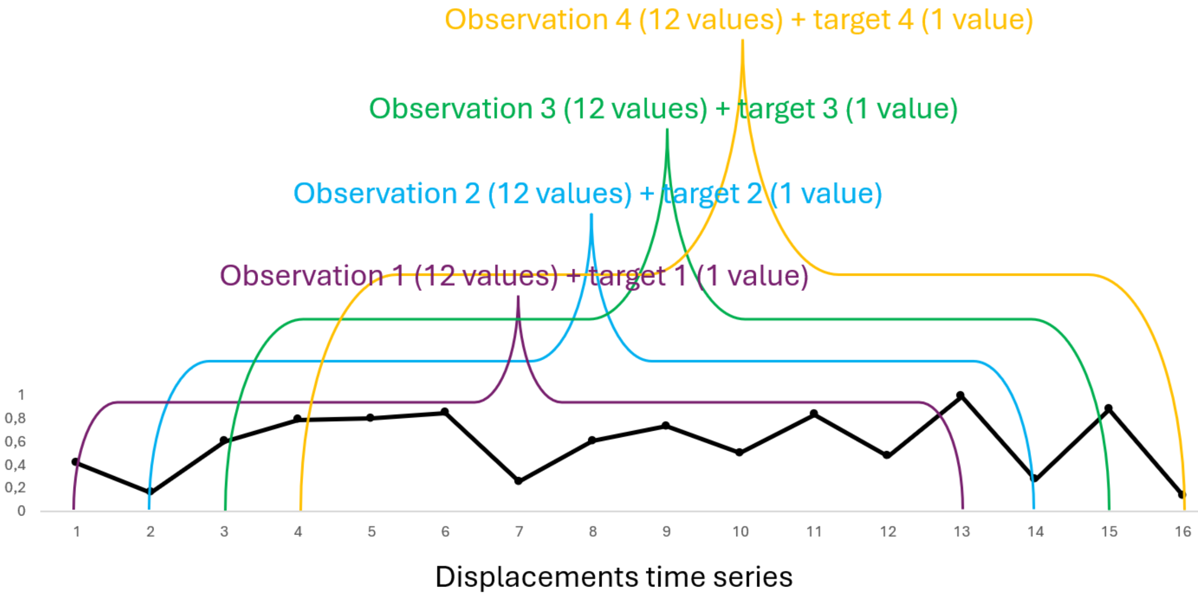

Figure 11: example of observation-target pairs extraction from time series.

A standard observation contains 12 days of displacements (one value per day), while a target is 1 day (the day after the last value in the observation), so that a pair consists of 13 days. Even though two pairs are 26 days, from this 16-days example four pairs are extracted, by allowing them to partially overlap with one another.

In point 9 the algorithm ensures that each observation and target fulfil two conditions: having at maximum pna (Proportion of Nans Allowed) missing values, and no time series with more than mdl (Missing Days Limit) consecutive missing values. Both parameters can be modified, but their default values are $pna = 8\%,\ mdl = 4$.

The data used for this study have a frequency of one measurement every 4 hours, the aggregation is necessary to have the desired 1/day sampling. The operation described in point 10 must process data differently depending on the type. For example, atmospheric pressure is the average of all the day's measurements, while displacement is calculated by subtracting from the

last measurement of the day, the last measurement of the previous day, which ensures that all the displacement occurred in the day of interest is accounted for.
Up to this point, each measurement in the data is associated with many additional information: measuring instrument, feature, date and time of the collection, site of origin… The NN, however, should only see the pure time series of values and nothing else. Simply discarding all the extra information will render us unable to visualize in any meaningful way the predictions of the NN (even though the metrics will still work normally). For this reason, in point 10 when the data is collected in the dataloader, a corresponding dataset is also created. The dataset contains all the information that would have been discarded for all the values: the dataset has the same exact shape of the dataloader and for each value inside the dataloader, the cell in the same position in the dataset contains the metadata for that value. Finally, it is necessary to ensure that each transformation applied to the dataloader is also applied to the dataset, for example when data are shuffled in point 11, the exact same operation must be executed on the dataset, so that the correspondence of cell position remains intact.
Training, validation, and test data (point 12) have different roles inside the algorithm. Training data is used to train the DL model, so that the more examples it sees, the more it becomes able to give correct outputs. They are also used to compute metrics about the whole dataset, so that information about validation and test data does not reach the model. For example, to normalize data in the [0, 1] interval it is necessary to know, for each time series, the maximum and minimum values. They are computed from training data only, instead of using the whole dataset. They are used multiple times during the training phase, one for Epoch. Validation data is used at the end of each epoch to test the model: if it does not improve, the changes of the weights from the current epoch are discarded. They can also be used as a condition to interrupt the training: for example, if the NN goes through six epochs without improvement, the training is halted. Even though validation data is never directly used for training the model, it still has an indirect influence on the weights by discriminating between changes to keep or discard at the end of each epoch, and potentially the length of the training. After the training phase, the model has to be tested on completely new data, which had no part in the training process: this is where test data are used.

Neural networks work better with values with mean 0 and variance 1 for various reasons (Bengio, 2012; Huang et al., 2023).

- Improved gradient descent convergence: in training deep neural networks, optimization is often done using gradient-based methods like gradient descent. If input features have varying scales, the gradient descent steps may be imbalanced, causing slower

convergence or making it harder to find the optimal parameters. Standardization helps ensure that the learning process is more stable and converges faster.

- Avoiding saturation in activation functions: many common activation functions, such as the sigmoid and tanh functions, can saturate (i.e., their gradients approach zero) when inputs are far from 0. Normalizing inputs helps keep activations within the range where these functions have significant gradients, which improves learning.
- More effective weight initialization: normalizing data allows for more effective weight initialization. Many initialization techniques (like He or Xavier initialization) assume that the input data has 0 mean and 1 variance. If the data is not normalized, these initializations might not work as intended.
- Balanced contribution of features: without normalization, features with larger scales may dominate the learning process, potentially overshadowing the importance of other features. Standardization ensures that all features contribute equally to learning.

The passage described in point 13 is obtained with Equation 17, it ensures that data are in a small range close to 0.

$$x_{norm} = \frac{x - T_{min}}{T_{max} - T_{min}} \tag{17}$$

Where x and x_{norm} are the original value and the normalized value respectively. T_{min} and T_{max} are the minimum and maximum values among the training data to avoid the validation and test sets to affect the normalization (by transmitting information). This means that in the validation and test sets there could be values that when normalized become lower than 0 or greater than 1, but this is not a problem because they should be unusual and still be close to the expected range. It is important to note that when choosing to use this method, T_{min} and T_{max} must be saved permanently alongside the model, because after training with normalized data, every new input must also be normalized in this same way before showing it to the NN. It is possible to obtain back the original values by applying the inverse formula (Eq. 18) to normalized values:

$$x = x_{norm} \times (T_{max} - T_{min}) + T_{min} \tag{18}$$

Data augmentation consists of increasing the training data by applying some kind of transformation to them. In this case the transformation is the addition of noise to the original

data. The numbers added to the data are generated from a gaussian distribution with $\mu = 0$, and σ equal to displacements in training data σ according to Equation 19:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{19}$$

The observations can be multimodal (displacements, precipitation, soil water level...) but this transformation is applied only to displacements, so the training data σ is also computed using only displacement data. Users can also choose between two implementations: in the first a single number (for type of data) is generated from the distribution and applied to an observation and the corresponding target. With the other option a different number for each day of observation and target are generated.

Data balancing uses data augmentation to even out the number of observation-target pairs belonging to all the classes. The algorithm finds the class with the highest number of elements and increases the quantity of pairs belonging to the other classes (with the data augmentation procedure described above) until each class has the same number of elements.

3.2. Tree species classification

3.2.1. Study site

The vegetation classification algorithm (Conciatori et al., 2024) requires its own separate data and preprocessing phase. The site chosen for this part of the research is in the Zao Mountains, a mountain range of volcanic origin, located in Japan, between the Yamagata Prefecture, at the border with Miyagi Prefecture.

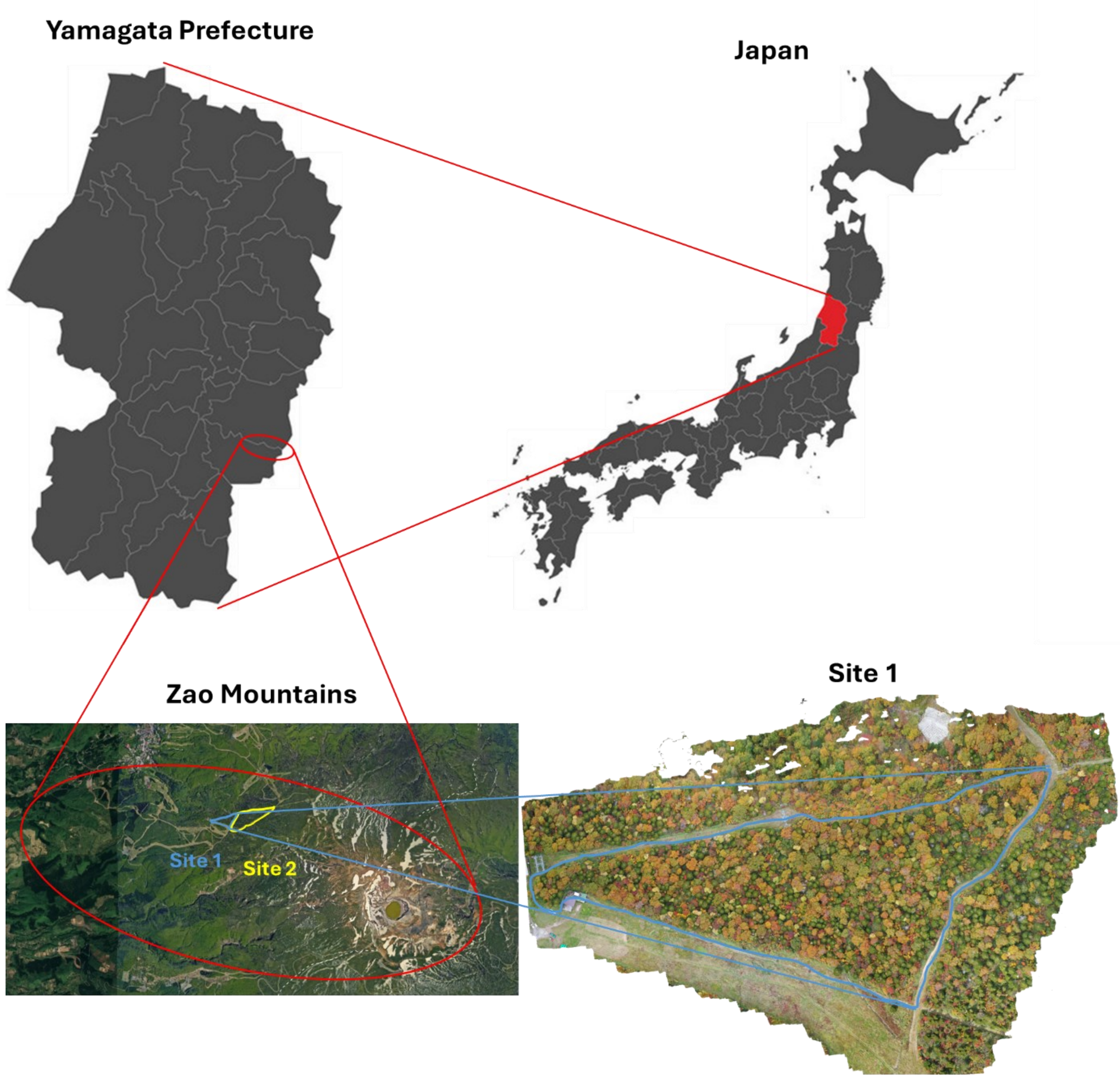

Figure 12: Zao mountains location, zoom of site 1. Source: (Conciatori et al., 2024).

The surveyed area has a surface of ~20 Ha and the altitude varies from 1,334 m at its lowest point to a maximum of 1,733 m (Figure 12), which influences the type and proportion of species found. The five species chosen for this study (see Table 10 and Table 11) are the most abundant, though there is a significant difference in diffusion with Fir (*Abies mariesii*) occupying more than 85% of the total area.

Table 10: plant species count, divided by site and species.

		Species				
		Beech	Pine	Fir	Japanese Rowan	Koshiabura
Sites	1	341	2	339	173	116
	2	131	63	3182	773	195
Total per Species		472	65	3521	946	311

3.2.2. Data structure

The expected inputs are images, square patches of $\sim 100 \times 100$ pixels that, given altitude and camera resolution, correspond to $\sim 2 \times 2$ meters of ground. Each patch is selected so that it contains only one species of vegetation (see examples in Table 11). The images are taken with drones and the perspective is always from above, perpendicular to the terrain.

Table 11: example of input data for the tree classification model.

Square patch sample	Species
	English common name: Beech Japanese common name: *Buna* Scientific name: *Fagus crenata*
	English common name: Fir Japanese common name: *Aomoritodomatsu* Scientific name: *Abies mariesii*
	English common name: Koshiabura Japanese common name: *Koshiabura* Scientific name: *Chengiopanax sciadophylloides*

Square patch sample	Species
	English common name: Japanese Rowan Japanese common name: *Nanakamado* Scientific name: *Sorbus commixta*
	English common name: Pine Japanese common name: *Matsu* Scientific name: *Pinus spp.*

The season for data collection has a big impact on the final result, in particular for deciduous trees, because they exhibit significant changes over 1-year cycles (color, presence of leaves). So much so that a NN trained with data from a single period can only be used for images collected in the same part of the year. The other possibility is training a model with images collected regularly during the year, to create a more general NN that can be employed for images from any season. For this study, however, the first approach was chosen because the data available for training was only from the autumn season.

Weather and lighting conditions can also affect the quality of the images, for example if the sun is very strong images are very crisp, but many tree crowns can be in the shade of tallest trees, which alters their color.

3.2.3. Sensors and data collection

A DJI Mavic 2 Pro Drone was employed for data collection. The UAV is equipped with a nadir-pointing camera (Hasselblad L1D-20c) with a maximum resolution of 20-megapixel and RGB channels (Figure 13).

Figure 13: drone DJI Mavic 2 Pro.

These settings were selected to obtain a ground sampling distance of 1.5-2.1 cm/pixel, which corresponds to the 2 m per 100 pixels mentioned at the beginning of this section.

- S priority (ISO-100).
- Shutter speed at 1/80 - 1/120 s in case of favorable conditions.
- Shutter speed at 1/240 - 1/320 s in case of worse conditions (strong sun or wind).
- 90% image overlap in each direction.

Since the area is on the side of a mountain, the altitude varies constantly from point to point. The drone was able to keep a constant altitude of 90 m during all the flights thanks to its height auto-regulation function, which preserves the relative proportions the of trees in the images. During a flight, drones go over the selected area in a grid pattern, taking pictures very often to ensure a high degree of overlapping, which is required by successive steps.

3.2.4. Data preprocessing

Using specialized software, *Agisoft Metashape* for this study (https://www.agisoft.com/), the collected images are composed to create a single orthomosaic image. The overlapping regions help the software align correctly the images with one another, but the process is not a simple cut-and-paste: they are also transformed so that in the resulting orthomosaic each pixel is shown as if seen exactly from above, perpendicular to the terrain. The 100x100 patches are extracted from this orthomosaic. Since each patch is required to contain only one species (it could be just part of a single tree if it is bigger than 2x2 m), they must be manually extracted from the UAV images. The patches used for training must also be manually classified, meaning that a forestry expert

has to identify the single species of tree in each patch and name the file accordingly. The actual process is faster: using software like *ArcGIS* or *QGIS* the expert can simply select points (one per tree) for each species from the orthomosaic. The program automatically extracts patches around each point according to predefined characteristics (shape, dimension…) and saves them with the name of the species and followed by progressive numbers. The expert's knowledge to recognize the tree species from the orthomosaic is still required for this operation.

The tree classification algorithm then automatically performs further elaboration on the patches obtained from the previous step (since it is a classification task, the tree species are also called classes).

1. The patches are loaded from memory and their class (species of the tree) is inferred from the name of the file or from the folder tree structure (inside the data directory there is one folder per species).
2. The number and types of classes present is also inferred from the training data.
3. The patches are split into training, validation, and test datasets according to user-defined proportions (by default it is respectively 80%, 10%, 10%).
4. Data balancing and data augmentation are optionally performed on the training data.
5. If the NN employed is a pre-trained model with custom transformations, they are applied at this moment.
6. The datasets are converted to dataloaders.

Data augmentation mentioned in point 4 is different from the one described earlier, because the data used here are images instead of time series. A set of transformations are defined:

- Horizontal flip
- Vertical flip
- Rotate 90 degrees clockwise
- Rotate 180 degrees
- Change brightness (randomly between 0.4 and 1.8 of the original brightness)

The algorithm cycles through the original images and creates new ones by applying to them each transformation with a probability chosen by the user.

Pre-trained NNs can be extremely useful because they are trained with an amount of data that is normally not available to researchers, which also requires very high computational capabilities for training (datacenter level). They are created for a specific problem but can be adapted to

similar tasks through a procedure called transfer learning. The end result is often much better than creating and training a new NN because of the limitation in training data and computation time. Pre-trained NNs, however, are often very sophisticated, and their structure and parameters are adapted specifically to the problem they have been designed for. The models employed in this algorithm are pretrained on ImageNet (Deng et al., 2009) and expect images with particular requirements. To simplify their usage, those NNs come with a set of functions called custom transformations, which can be applied to images to conform them to the expected input for the models (point 5).

4. Algorithms development

In this section the two principal projects produced during the PhD will be presented, not entering into the details of the code, but examining their high-level functions and capabilities. There will also be subsections dedicated to auxiliary functions and algorithms which are not directly part of the two main projects but were developed for them and are relevant for the thesis.

4.1. Landslide forecasting

This algorithm predicts the future displacements of a landslide given past displacements and possibly other sensors' signals.

The same problem can be approached in many different ways, and for landslide forecasting alternative versions of the algorithm were implemented and tested. The sequence of attempts is presented in chronological order. As stated before, the user can choose how many future days the NN will try to predict for all the tests, however, this value was always kept at 1, since the task becomes more difficult otherwise.

4.1.1. Comparison algorithm (Baseline)

In the beginning, a very simple model was added for comparison, also called baseline model. The idea is that this very simple model would provide a fixed baseline to measure the improvement of the various versions of the algorithm. This comparison model takes the same input as the normal model (for compatibility reasons), but only reads the displacement of the last day of the input and outputs it as a prediction without doing any actual computations. In summary it always "predicts" that the new displacement will be exactly the same as the last measurement.

This comparison model was later discarded because, even though it is extremely simple, it is, on average, much more accurate than most sophisticate prediction algorithms, in particular for

landslides with very low activity (es. Site A: Landslide Monitoring in Northern Italy). More importantly, it also suffers from the same problem discussed in Section “4.3.2. Metrics for landslide forecasting and tree species recognition”. Since landslides show low activity for most of the time, the algorithm seems to have very good results, but it will never be able to predict an acceleration event. Hence the metric used to measure its performance is not appropriate for this combination of algorithm and task, since it should not be so positive. To summarize, it was not a good comparison because it was a bad predictor, but appeared to have very good performance nonetheless.

4.1.2. Original algorithm

The first implementation of the algorithm is the most direct for the given task: it uses the data obtained from on-site instruments and outputs the expected future displacements. Since precipitations are not obtained from MUMS, they are not used in this version. The input of the NN is (see also Table 12):

- Inclinometer
  - x displacement
  - y displacement
  - depth
- Piezometer
  - soil water level
  - depth
- Barometer
  - atmospheric pressure

Table 12: original algorithm input.

<table>
<tr><th colspan="2"></th><th colspan="3">Inclinometer</th><th colspan="2">Piezometer</th><th>Barometer</th></tr>
<tr><th colspan="2"></th><th>x displacement</th><th>y displacement</th><th>Depth</th><th>Water level</th><th>Depth</th><th>Pressure</th></tr>
<tr><td rowspan="5">Days</td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>…</td><td colspan="6">…</td></tr>
<tr><td>12</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

The displacements measured each day are one for each inclinometer inside each Vertical Array, per number of MUMS in the site. The algorithm ensures there is exactly one piezometer measurement per VA, so that this value has to be copied and associated with all the displacements of that VA. In a similar way, the algorithm ensures that there is exactly one barometer for the entire site, which means that its measurement must be cloned and associated with all the displacements of all the MUMS of the site.

The target and NN output consist only of the predicted displacement with the format of Table 13:

Table 13: original algorithm output.

		Inclinometer	
		x displacement	**y displacement**
Days	**13**		

Input, target and output can also be represented graphically, Figure 14 and Figure 15 below are examples. The first plot shows only displacement, while the second also contains all the auxiliary information. For clarity only one axis is represented (x axis).

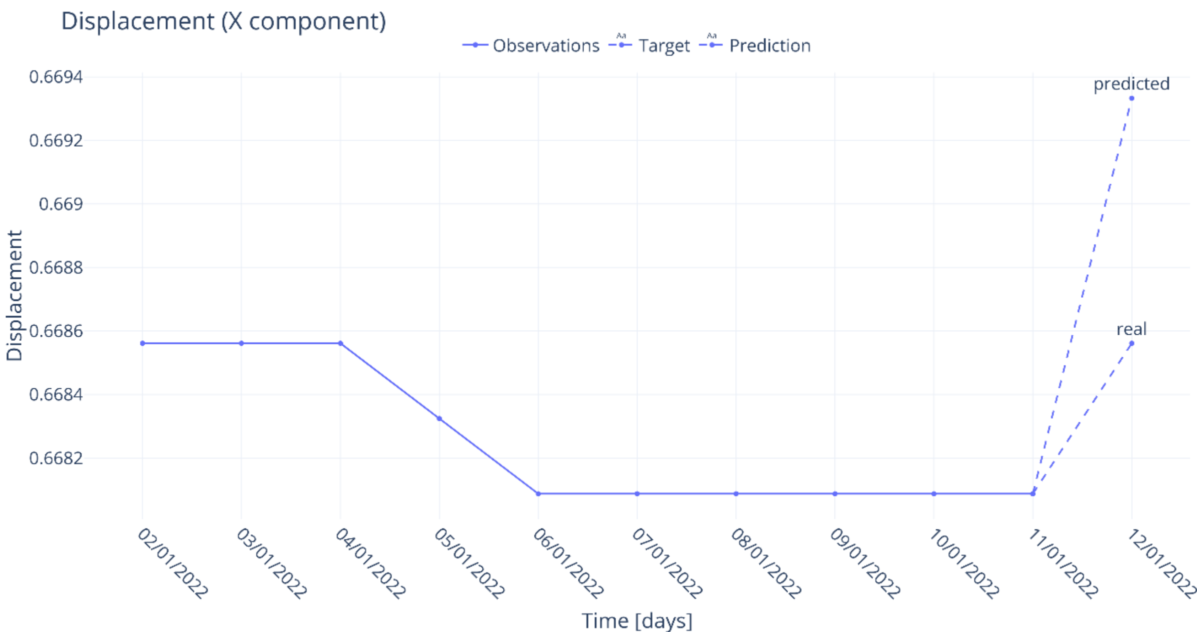

Figure 14: Observation, target, and prediction. Only x displacement is plotted.

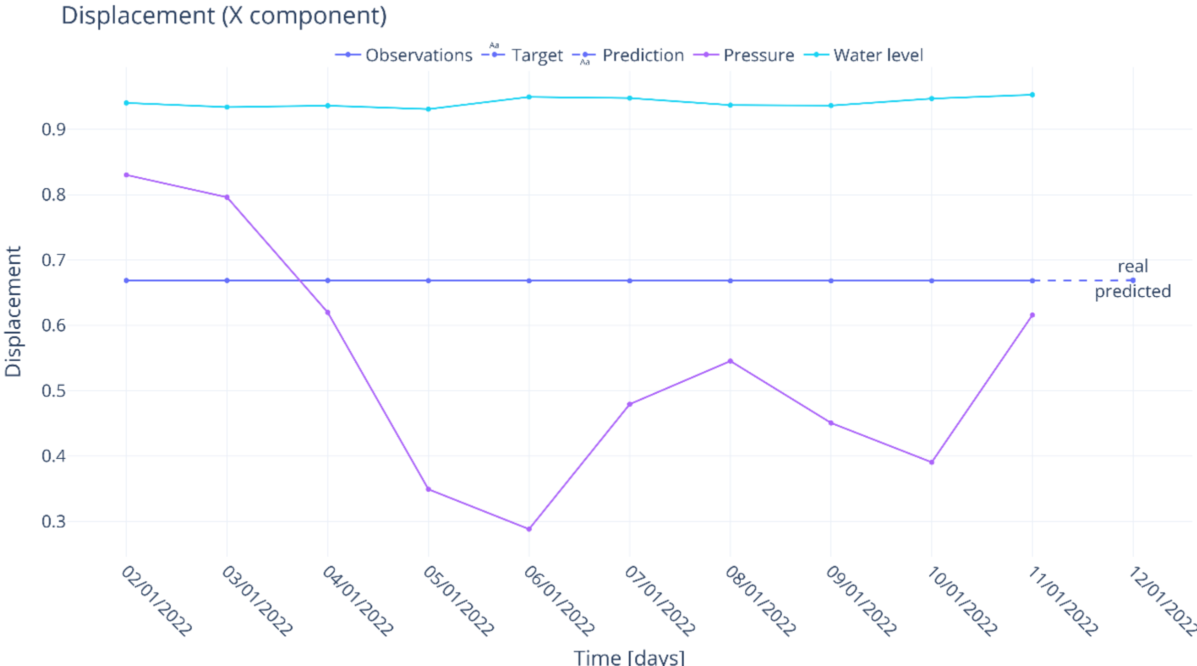

Figure 15: Observation, target, and prediction. Plotting x displacement, water level and atmospheric pressure at ground level (no y displacement).

Figure 15 also highlights the importance of normalizing data: the displacement information in the two images is exactly the same, but since water level and pressure have much greater oscillations, the scale of the second plot is such that the displacements appear as a flat line.

The Neural Networks used before the change introduced in Section “4.1.6. Neural Network’s architecture expansion” have a semi-fixed structure, shown in Figure 16.

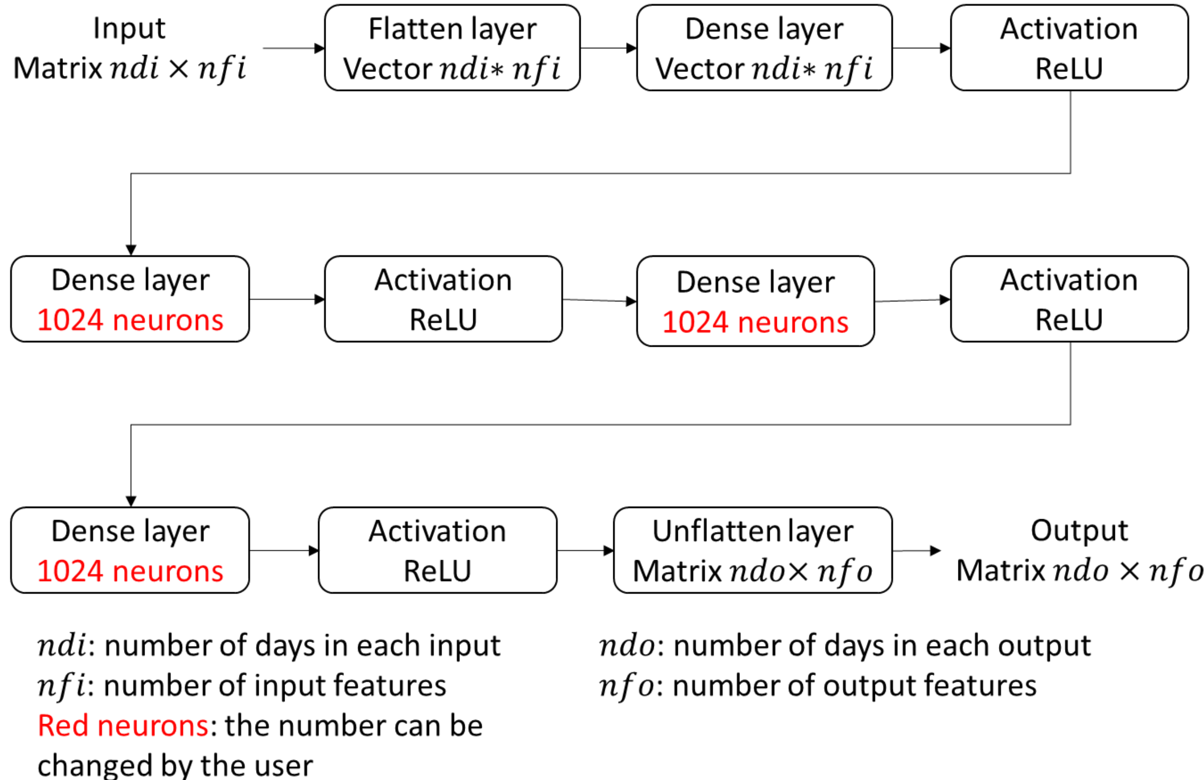

Figure 16: architecture of the NNs used in the earlier iterations of the software.

This is a simple architecture, the type and number of layers of this FFNN is fixed, but the number of neurons in the dense layers can be modified by the user. As explained in "3.2.2. Data structure" Section, the shape of the input and output layers cannot be changed once decided. But they also cannot be fixed because their shape depends on settings the user can change (es. the number of days an observation contains). For these reasons when the software needs to create the NN for a new training, it calculates the correct shape for the input and output layers by reading the current settings configuration.

As anticipated at the beginning of "3.1.1. Study sites" Section, this earlier implementation of the model can only read data from a single source because only site A data were available at this time. This is also the only version that uses TensorFlow 2 as the underlying ML framework, every subsequent iteration is based on PyTorch.

4.1.3. Vertical Array as input

Each MUMS' VA is composed of sensors at constantly increasing depths, and in the original version of the algorithm each one is examined separately. This implementation uses as input a 3D matrix (Figure 17) which is equivalent to Table 12 repeated for each inclinometer of a single MUMS, which gives the 3rd dimension to the matrix. This has the advantage of giving the NN more

context about the state of the landslide but has two significant disadvantages: it produces a model that can only work with data obtained by MUMS with the same exact length as the ones used in the training. It also reduces the number of training data by a factor equal to the length of the Vertical Arrays.

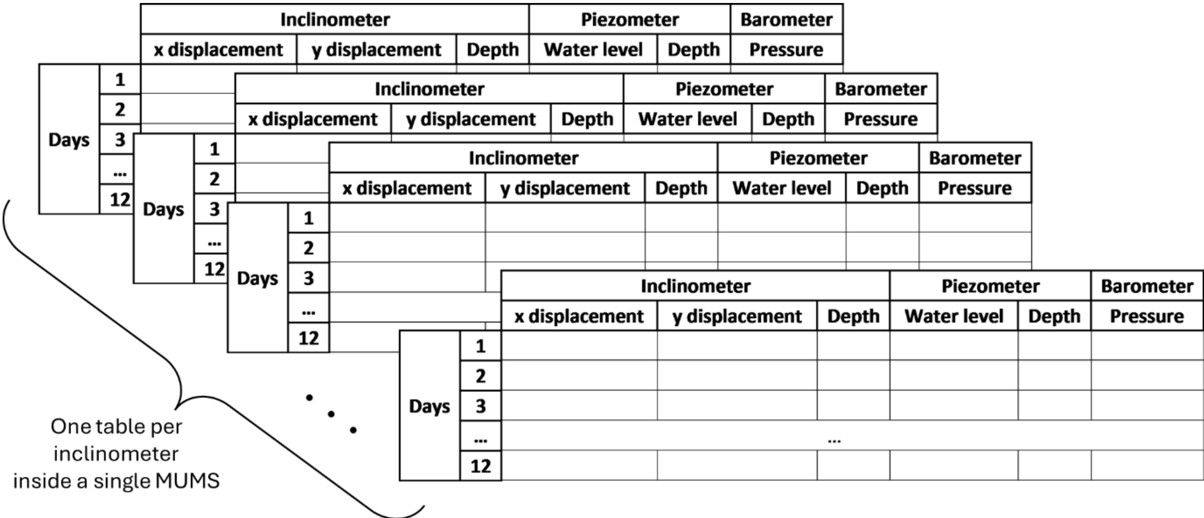

Figure 17: whole Vertical Array as input.

The target and output also gain the same additional dimension, as shown in Figure 18.

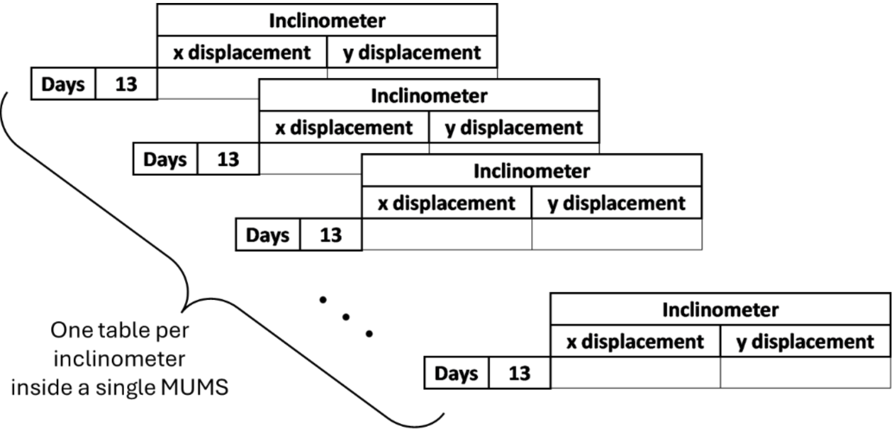

Figure 18: output for the algorithm that takes a whole Vertical Array as input.

4.1.4. Addition of rainfall data

There are drawbacks to adding more information to a NN input, as summarized by the list below.

- The bigger the input, the bigger a model needs to be to understand it and draw useful conclusions, which in turn requires bigger training datasets and longer training times.
- A network that is trained with a lot of information can be used to make predictions on a new site only if it is provided with the same amount and type of information, which can limit its applicability.

- In this particular case, the MUMS already measured soil water levels.

The effect of precipitation on landslides, however, is well known and documented, in fact it is one of the most impactful external events on their behavior (Iverson, 2000; Polemio et al., 2000). Rainfall information is also quite easy to obtain, even for past periods, because there are many meteorological stations which provide publicly their weather measurements. Considering all these reasons, it was decided to add rainfall data to the input of the original algorithm, which has the format of Table 14.

Table 14: composition of the input for the algorithm that also uses rainfall.

		Inclinometer			Piezometer		Barometer	Rain gauge
		x displacement	**y displacement**	**Depth**	**Water level**	**Depth**	**Pressure**	**Precipitation**
Days	**1**							
	2							
	3							
	…	…						
	12							

The output does not change, it is the predicted displacement of the next day, see Table 13.

Even though precipitations influence landslides, the effect can have a significant time delay (Iverson, 2000; Polemio et al., 2000), depending on terrain, vegetation coverage and landslide type. To account for that, along with rainfall data, a setting is added to the algorithm. The parameter *rainfall_days_shift (rds)* controls the shift (in days) between the precipitation time series and all other time series. For example, if *rainfall_days_shift* = 1 and the displacement data are 12 days from **02**/01/2021 to **13**/01/2021, the rainfall data would be from **01**/01/2021 to **12**/01/2021. Adjusting this parameter can help the model to understand the relationship between precipitation and landslide movements.

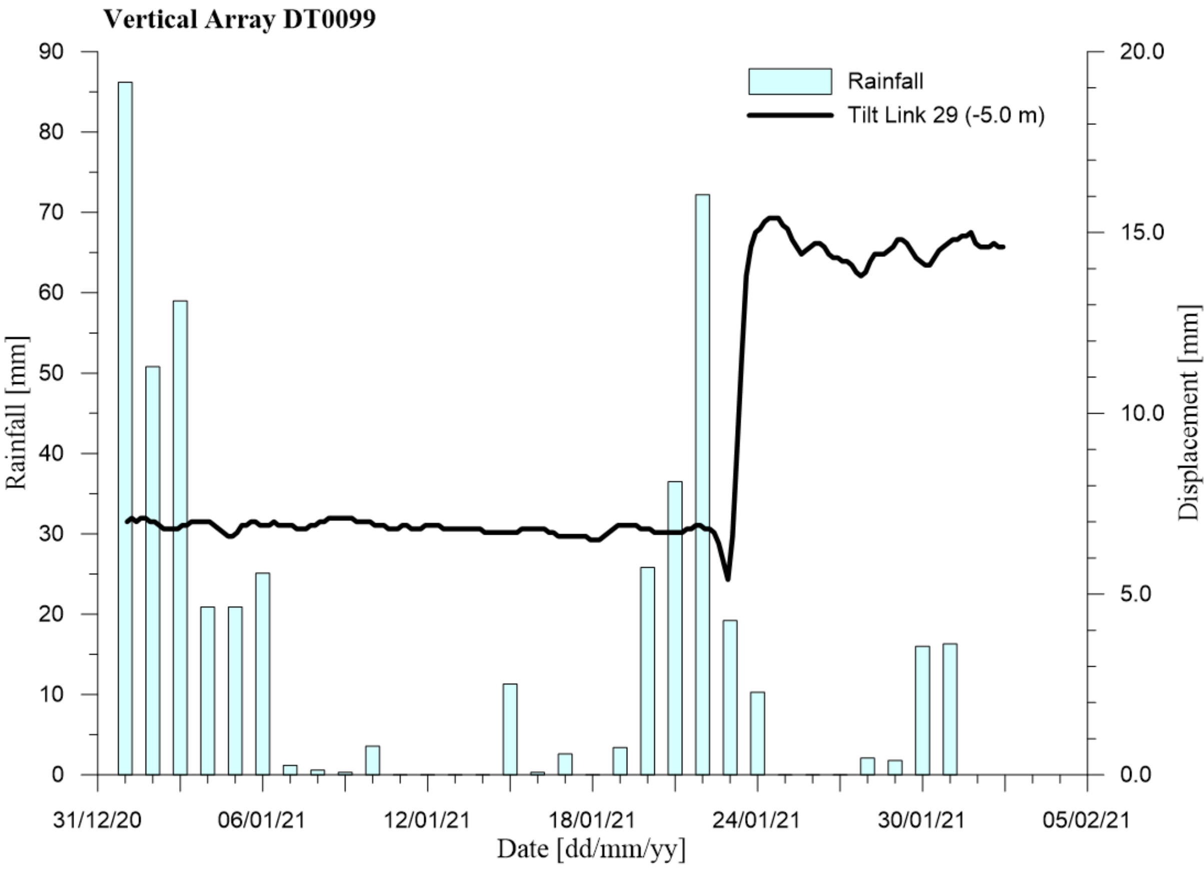

Figure 19: displacements and precipitations measured in Site A (January 2021). It is possible to see both phenomena: displacement provoked by rainfall, and the delay of ~2-5 days between cause and effect.

Figure 19 plots displacement and rainfall data from Site A (Section "Site A: Landslide Monitoring in Northern Italy") in the same period. Intense precipitations from 20/01/2021 to 24/01/2021 cause an abrupt displacement between 23/01/2021 and 24/01/2021. This is an example of the relationship between precipitations and landslide activations, and also shows the time lag that occurs between the two events.

4.1.5. Custom loss function

Training data is not the only factor that controls what a NN learns in the training phase. The loss function is what ML algorithms try to optimize during that phase: their sole objective is to give answers that will minimize the value of the loss, which represent the "distance" from the correct answer. *Loss* is the operation by which the output of a model is compared to the ground truth during training, and it depends on output type and problem definition. The loss function can be very simple: for example, for a network that takes two numbers as input and must learn to output their sum, the loss can be the absolute difference (MAE) between output and correct sum. In other cases, very specific and complicated functions must be created.

- How to measure numerically the “loss” of a sentence written by ChatGPT in response to user request?
- How “good” is an image generated with stable diffusion (Rombach et al., 2022) form a text description (Borji, 2023; Stöckl, 2023)?

Up to this point the algorithm used *MSELoss* loss function (or simply MSE), which means it calculated the Mean Squared Error (MSE) between predicted displacements and measured displacements. It is a standard loss function for regression algorithms, expressed by Equation 20 below.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \tag{20}$$

In statistics, Y and $\hat{Y}$ are used to refer to true and predicted measurements vectors respectively, while n is the length of the vector of displacements (both vectors Y and $\hat{Y}$ must have the same length) $n = |Y| = |\hat{Y}|$. Y and $\hat{Y}$ are vectors and not single numbers because the loss is designed to work on many inputs at once, mainly for efficiency reasons.

We noticed a problem with the standard MSE loss for our particular case. For a large majority of the time, the displacements change very slowly over time: in the available data typical daily oscillations were in the order of 10^{-4} meters. The models learned to minimize the errors by simply “predicting” that the displacement of the next day would be the same as the displacement of the last day of the input (displacement of day 13 = displacement of day 12), see Figure 20 for an example. This is concerning because the whole purpose of the study is to predict sudden dangerous activities of a landslide to launch early warnings, which the models were unable to do.

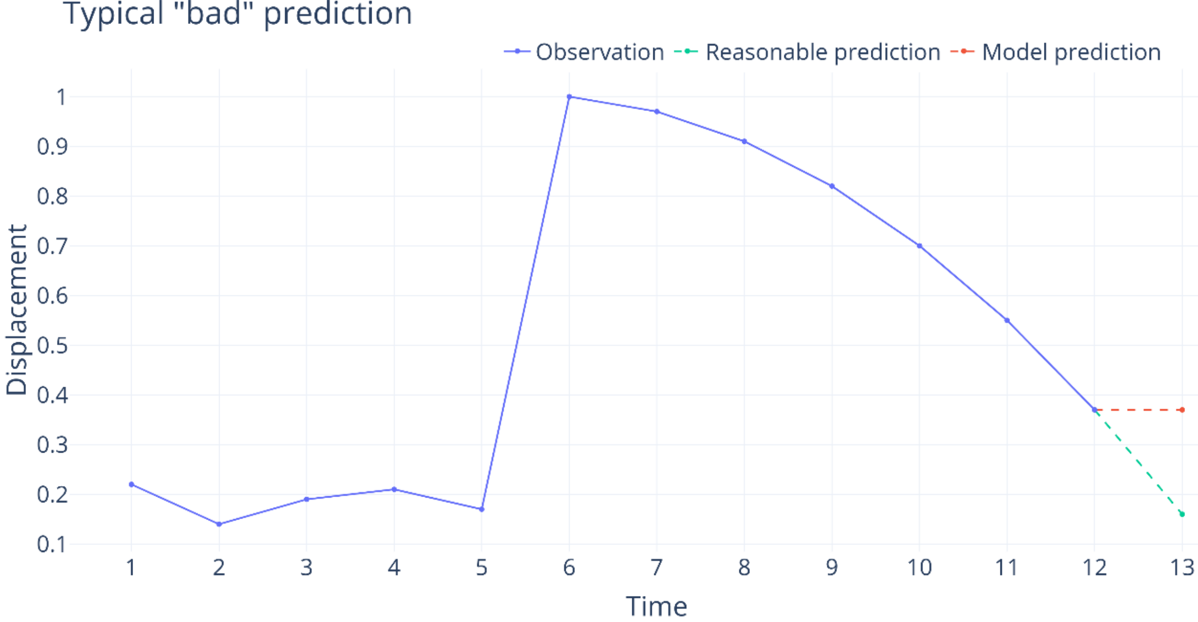

Figure 20: example of typical bad prediction. The model ignores an evident trend because it learned to always predict that the future displacement will be very similar to the last observed displacement.

To address this problem the custom loss function “Time Series Midpoint” or TSM is developed (Eq. 21). The target Y used by this function has an added dimension of length 2, because to evaluate a prediction, it needs to know two values: the true displacement and also the last day of the corresponding input. This changes Y from a vector of length n into a matrix with dimensions $n \times 2$ so that $Y_{1-n,1}^{new,\ day_k} = Y_{1-n}^{old,\ day_{k-1}}$ and $Y_{1-n,2}^{new,\ day_k} = Y_{1-n}^{old,\ day_k}$ where Y^{old} and Y^{new} are the targets used by MSE and TSM respectively. The new target is the union of two consecutive “old” targets.

$$TSM = \begin{cases} \frac{1}{n}\sum_{i=1}^{n}\left(Y_{i,2} - \hat{Y}_i\right)^2, & \text{if } \left|Y_{i,2} - \hat{Y}_i\right| < \left|Y_{i,1} - \hat{Y}_i\right| \\ \frac{1}{n}\sum_{i=1}^{n}\left(Y_{i,2} - \hat{Y}_i\right)^2 c, & \text{otherwise} \end{cases} \tag{21}$$

TSM is equivalent to MSE if the displacement prediction $\hat{Y}_i$ is more similar to the ground truth $Y_{i,2}$ than to the last value seen in the input $Y_{i,1}$, which is expressed by the formula “if $\left|Y_{i,2} - \hat{Y}_i\right| < \left|Y_{i,1} - \hat{Y}_i\right|$” (Figure 21). If the prediction is not a real prediction but a guess using the past value, this loss function computes the MSE and multiplies it by the coefficient $c \geq 1$. In this second case the TSM gives a result that is greater than MSE (greater loss values correspond to worse predictions). This should encourage the algorithm to give predictions that are closer to the real displacement to predict than to the past displacement.

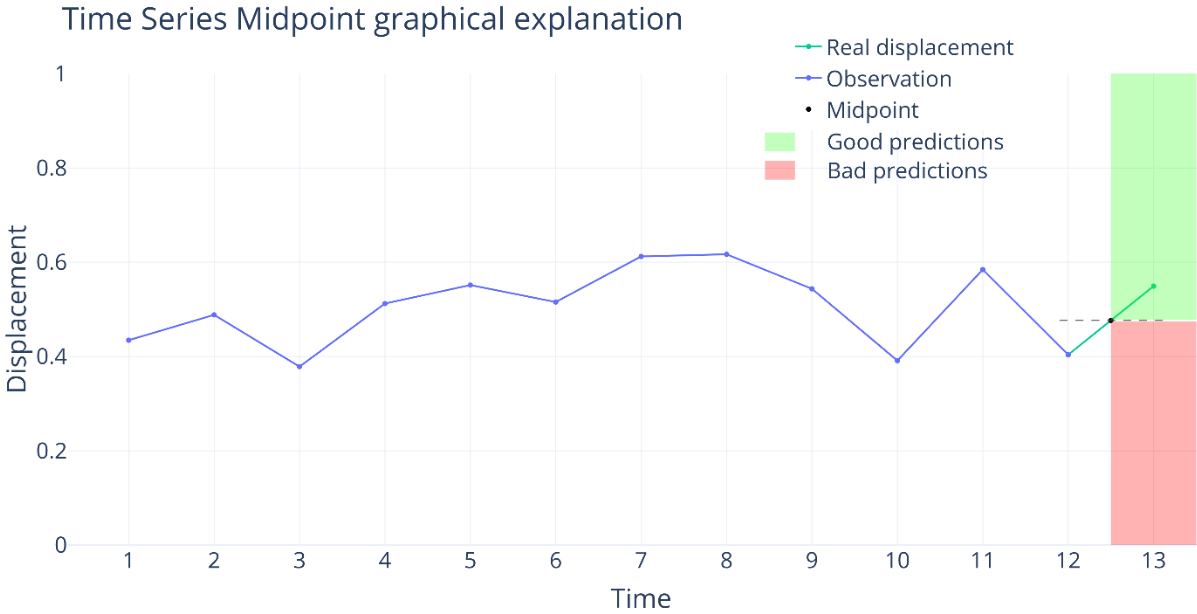

Figure 21: graphical explanation for the TSM loss function. Green and red demarcate the two areas in which predictions are considered "good" or "bad" respectively. Where good means that the prediction is closer to the real displacement (green dot) than to the last observation (rightmost purple dot), and the opposite for bad predictions. This is achieved by calculating the midpoint (black dot) between real displacement and last observation, which is the watershed between the two areas.

The parameter c controls how much TSM "punishes" bad predictions compared to MSE, if $c = 1$ then TSM = MSE.

4.1.6. Neural Network's architecture expansion

The software developed allows for the automatic generation and training of customizable Neural Network models. The NN architecture can be configured before training by specifying the number of layers and the number of neurons per layer, while the shape of the input and output layers is automatically inferred from inputs and targets respectively, as shown in Figure 22.

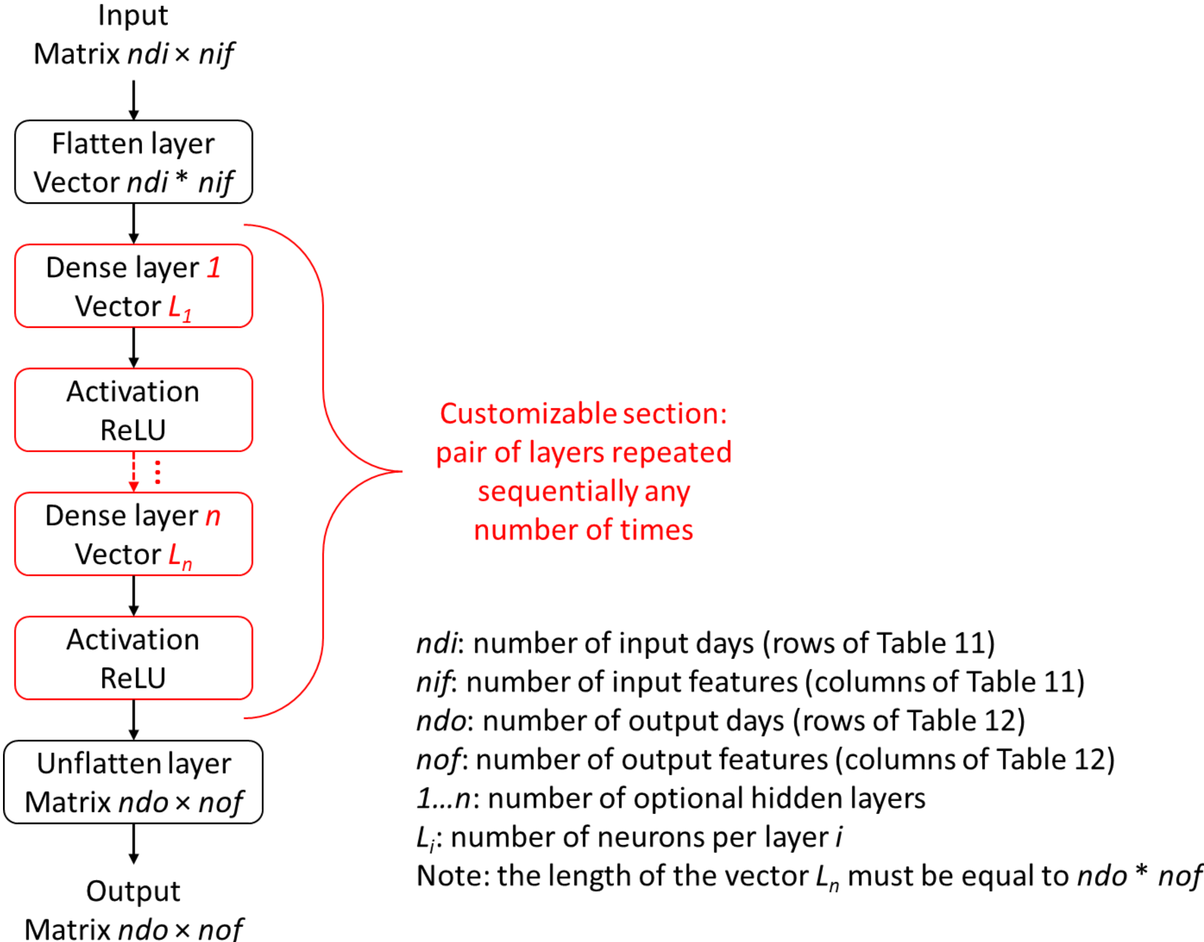

Figure 22: Dense NN modular architecture. The red part consists of layers that can be modified by the user: the number of layers and the number of neurons in each is customizable.

This allows the creation of custom networks of any shape, but it has one notable limitation: all the hidden layers are dense (or linear) layers, which are the most basic building block of NNs, but not the only one. A NN with this type of architecture is called Feed-Forward Neural Network (Svozil et al., 1997).

The following is a description of the structure of the Neural Networks created with this procedure.

- Input layer: accepts only inputs with the correct shape, matrices $ndi \times nif$ (Table 6).
- Flatten layer: reshapes the matrices to 1D vectors with length $ndi * nif$.
- Dense layer: connects every neuron of the previous layer with every neuron of his layer, which can result in a very high number of connections. For example, if the two interested layers have 200 and 1,000 neurons respectively, the number of connections between the two is $200 \times 1{,}000 = 200{,}000$. The strength of each connection is updated during the training phase, and it is the process by which the NN tries to learn correlations between the inputs.

- The Rectified Linear Unit (ReLU) layer is an activation function. The purpose of activation functions is to improve results and introduce desired properties like non-linearity or finite range. ReLU performs the operation $f(x) = \max(0, x)$ for each neuron individually, see Equation 9 in Section "2.4.2. Convolutional Neural Networks".
- There must be at least one pair of dense-ReLU layers, but the user can add any number of them and can choose the quantity of neurons of each of those layers.
- Unflatten layer: reshapes the 1D vectors from the last ReLU layer into $ndo \times nof$ matrices. The length of the input vector must be $ndo * nof$, otherwise the operation would fail.
- Output layer: accepts only inputs with the correct shape, matrices $ndo \times nof$ (Table 7), and renders them available outside the Network.

A more sophisticated approach to examine time series, or any sequence in which close elements are more correlated with each other than far away elements, are Convolutional Neural Networks (see Section "2.4.2. Convolutional Neural Networks").

Two-dimensional convolutions are often employed to examine images, but since our data consists of many mono-dimensional time series, we utilize 1D convolutional layers. The input is a group of time series (x and y displacements, precipitations, soil water level...), each one of them is examined by a dedicated sequence of convolutional layers, which should convert the input into a more compressed and meaningful set of numbers. The result of the convolutional layers is then passed to the "decision layers" (one or a few dense layers), which make use of the information extracted by the convolutional layers.

The following is a description of the structure of the Neural Networks created with this procedure, also illustrated in Figure 23.

- Input layer: accepts only inputs with the correct shape, matrices $ndi \times nif$ (Table 6).
- Many parallel 1D convolutional layers: each layer examines one of the input time series (12 days of x displacements, 12 days of y displacements, 12 days of precipitation...).
- Optional Max Pooling layers: each convolutional layer can be followed by a pooling layer which down-sample the previous layer's neurons using the maximum operation (see Equation 10 and Figure 1 left).
- The user can define any number of convolutional and pooling layers and can fully specify any of their parameters.

- Flatten layer: recombines the output of each parallel convolution into a single 1D vector. The final part of the Network is exactly the same as the previously described Dense NN (see "4.1.2. Original algorithm" Section).

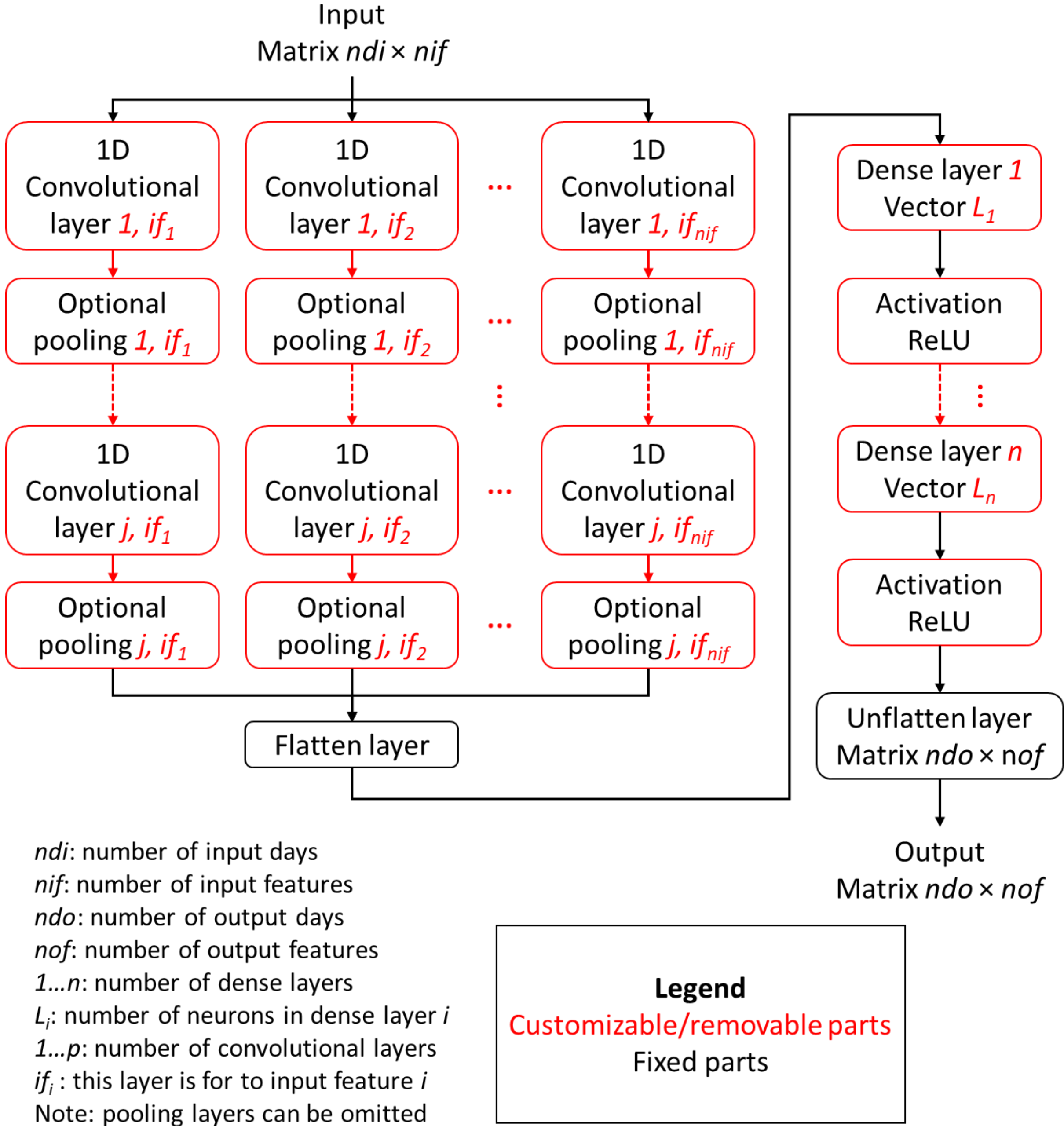

Figure 23: Convolutional Neural Network modular architecture. Red layers can be customized by the user at creation time.

4.1.7. Data balancing and data augmentation

Deep Learning algorithms have the advantage of learning from data, even very complex functions or behaviors that we do not know or would not be able to describe formally. They are, in turn, very dependent on the quantity and quality of training data, even the perfect DL model, if trained with

a poor-quality dataset or insufficient data would not yield good results. Data distribution and pre-processing operations on datasets can also have a big influence on DL algorithms.

Considering these factors, functions for data augmentation (Maharana et al., 2022; Mumuni & Mumuni, 2022; Rebuffi et al., 2021) and data balancing (Batista et al., 2004; Jadhav et al., 2022) were implemented. Both are standard techniques, the first involves the creation of "new" data by applying random noise or similar operations to the original data. The specific operation depends on the structure of the original data (random alterations of an image or a text string are very different). Data augmentation gives the model more training data, which is expected to improve the performance, and can also help reduce overfitting. Data balancing uses data augmentation to reduce the disparity of sample quantity between each class. This improves the recognition of underrepresented classes and ensures the model predicts based on useful features and not probability distribution. DL algorithms tend to notice if a class appears much more frequently or rarely than others in the training and try to use this knowledge in the predictions. The desired behavior, however, is that each prediction only depends on the information in the input, not on prior expected probabilities. When all classes are balanced, this problem is removed.

The augmentation function executes the following steps for each observation-target pair of the training dataset (augmentation and balancing are always performed only on the training dataset).

- Generation of a random number $noise$ according to a gaussian distribution with $\mu = 0$, and σ equal to displacements in training data σ (Equation 19).
- $noise$ is kept inside the range $[-is, +is]$, (is is inclinometer sensibility) with a modulo operation.
- $noise$ is added to the displacements of the observation and target.
- If the addition of $noise$ caused the target to change class, this pair is discarded.

Changing a setting it is possible to generate and use one random number per displacement, instead of one for all the displacements of the observation and target pair.

If data balancing is required, the previous operations are continuously used on the smallest class until all classes have the same number of elements, after which, if data augmentation is also required, it is performed on the new balanced dataset.

Note that landslide classification has been treated as a regression problem, which means that the true results and the output of the models are real numbers, not classes. In this subsection the classes mentioned are calculated by applying user defined thresholds to the numeric values

to discretize them into a finite and small number of classes. See Section “3.1.4. Data preprocessing” for an exhaustive explanation.

4.1.8. Condensation and reduction of input features

DL algorithms learn faster and better if the input data is both informative and relevant. This is because, with more information available the models can learn more precise and robust solutions, as long as the information presented is correlated with the output and linearly independent. If part of the information is redundant (not linearly independent) or it is noise (not or weakly correlated with the output), a DL algorithm can still reach a good solution by learning to ignore these parts of the input, but it requires more training data, and potentially more weights, to achieve the same results. Another practical consideration is that if the algorithm is trained to predict the next displacement using many different sources (like atmospheric pressure, soil water level, temperature...), to use the model on a new site, there is the need to collect all those types of data. For these reasons, the next step was to reduce the input dimensions while preserving the information content as much as possible.

Each displacement measurement obtained from MUMS consists of two numeric values: displacement along the x and y axis of the horizontal plane (Figure 8, right panel). In this version of the software, they are combined into a single number which represents the total horizontal displacement using Equation 22:

$$h_disp = \sqrt[2]{x_disp^2 + y_disp^2} \quad (22)$$

In this way two values are reduced to one, with the only information lost being the direction of the movement, which the algorithm was not able to use anyway because it has no information about the orientation of the x axis with respect to the main slope direction. It is sufficient that some of the training data uses different conventions, or that new inputs use a different convention from the training data to make this information non-useful and detrimental instead.

The main purpose of the barometer was to provide indirect information about precipitations to the algorithm. Since now the input data contains rainfall information, the barometer is excluded to reduce redundant inputs. It is also tested the removal of the soil water level for the same reason.

The input changes from Table 14 to the new, greatly simplified, Table 15.

Table 15: "reduced" input, format of the input for the landslide forecasting algorithm after feature condensation.

		Inclinometer	Rain gauge
		h displacement	**Precipitation**
Days	**1**		
	2		
	3		
	…	…	
	12		

The output is also changed accordingly from the two values of Table 13 to Table 16.

Table 16: "reduced" output, format of the output for the landslide forecasting algorithm after feature condensation.

		Inclinometer
		h displacement
Days	**13**	

4.1.9. Switch from regression to classification

Once a research problem has been selected, its framing is one of the most important decisions to make. What exactly are we trying to calculate or predict, what is considered a correct solution, how are data represented… For DL, this has consequences on what algorithms can be applied to the problem, what loss function, optimizer, and metrics are available, and so on. Up to now landslide classification was tackled as a regression problem, which means that the real displacements measured were considered the targets, and models tried to output values similar to those targets. It also means it is impossible to expect an exact answer since the solution space is $\mathbb{R}$. This is the reason the loss function computes the difference between prediction and target: the algorithm tries to minimize that difference, and the problem is considered solved if the algorithm consistently gives predictions close enough to their targets.

We experimented by reframing this problem as a classification task. This involves extending what was already done for the metrics (see Section "4.3. Custom metrics development") to all the calculations: user defined thresholds are used to change all the targets from real displacements to their corresponding classes. The loss function is changed from MSE or TSM to Cross Entropy. The algorithm's output is changed to a vector of length $n =$ number of classes, in which element

$i \in [1, n]$ corresponds to the confidence of the model in the future displacement belonging to class i. The output is transformed into a probability distribution ρ by applying a $softmax$ or σ operation (Goodfellow et al., 2016), Equation 23 below:

$$\rho_i = \sigma(output, i) = \frac{e^{output_i}}{\sum_{j=1}^{n} e^{output_j}} \quad (23)$$

With $e =$ Euler's number. This operation ensures that each element is in the valid probability range $\rho_i \in [0, 1]$, and that they sum up to 1, $\sum \rho = 1$. The answer is calculated by taking the class which is associated with the highest probability in ρ (Eq. 24):

$$\text{Predicted class} = arg\ max(\rho) \quad (24)$$

Another way to describe this change is to say that now the correct solution has a small number of possible discrete values (for example 1 = small displacements, 2 = medium displacements, 3 = high displacements), and we don't measure a continuous distance from the truth like before, the model will either give the correct answer or not.

4.1.10. Conversion of time series to images

Some papers use various techniques to convert time series data into images and use those images as input for their models. There are a few reasons to go through this process:

- Computer Vision is currently one of the most studied and advanced subfield of DL, which produced very sophisticated but specific architectures.
- As of now, image recognition and classification tasks boast a much greater success than time series predictions.
- For the task of image classification there are many - very successful and robust - pre-trained models available. They can be adapted with fine-tuning to specific problems (detailed explanation in Section "2.4.5. Transfer Learning and Fine-tuning").

When applying this strategy, the most important design choice is how to represent time series with images. For this study, three different methods are tested.

1. Each feature time series is plotted into its own graph. The graphs are then stacked vertically into a single image (Figure 24).

2. All features are converted into graphs using Gramian Angular Field (Figure 25). The image is obtained by stacking the graphs horizontally (see Figure 26 for examples of displacement graphs).

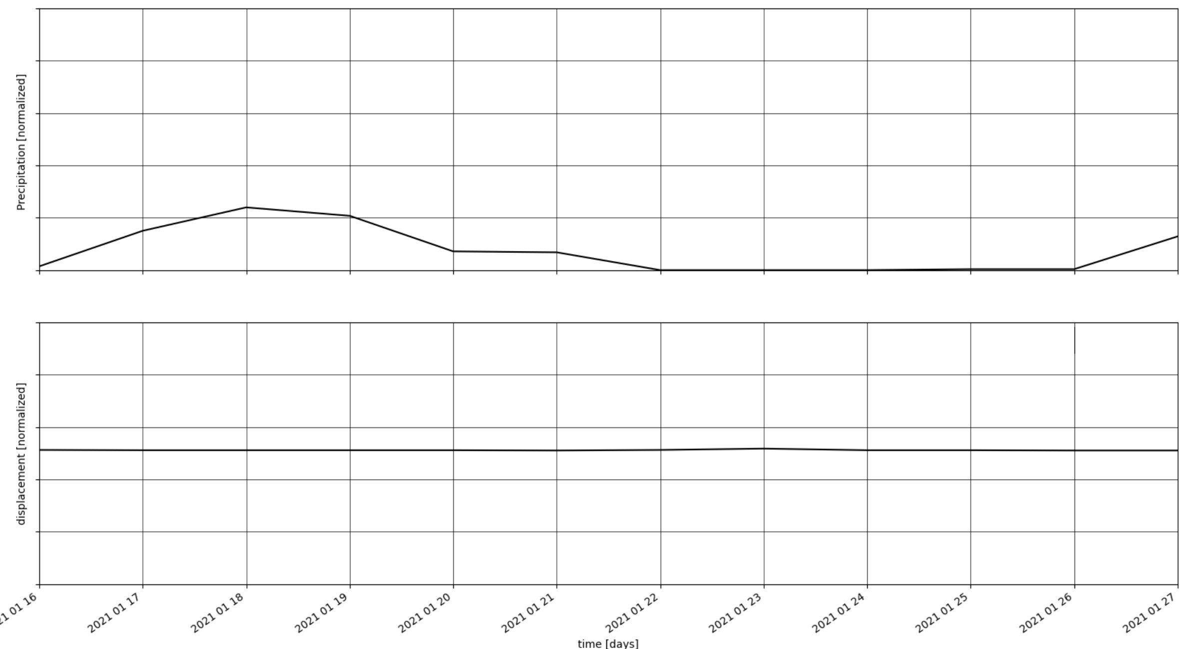

Figure 24: example of input for the algorithm described in point 2. Two plots containing displacements and precipitations respectively are stacked to create a single input image.

For point 1 method the values used must be normalized in the [0,1] interval, so that features with different scales can be represented simultaneously in a meaningful way. This is because the number of pixels in an image has much less precision than what floating numbers allow. As an example, in the displacement dataset there are many values $< 10^{-4}$m, a sequence of those values in the same cartesian plane with rainfall data would result in them being represented by a flat horizontal line $y = 0$. Depending on the characteristics of each feature, however, some time series may appear flatter, since min and max values used for the axis are calculated from the whole time series, not from the single observation. This is necessary to maintain fixed proportions between observations, otherwise the NNs would not be able to learn anything.

The main idea for point 2 method comes from the paper (Teza et al., 2022), while the code implementation for converting images is from (Z. Wang & Oates, 2015). It uses Gramian Angular Field (GAF) (Figure 25), which consists of applying the following operations to observations in sequence:

1. Rescaling
2. Change to polar coordinates

3. Calculate the Gramian matrix
4. Smooth out with Piecewise Aggregation Approximation

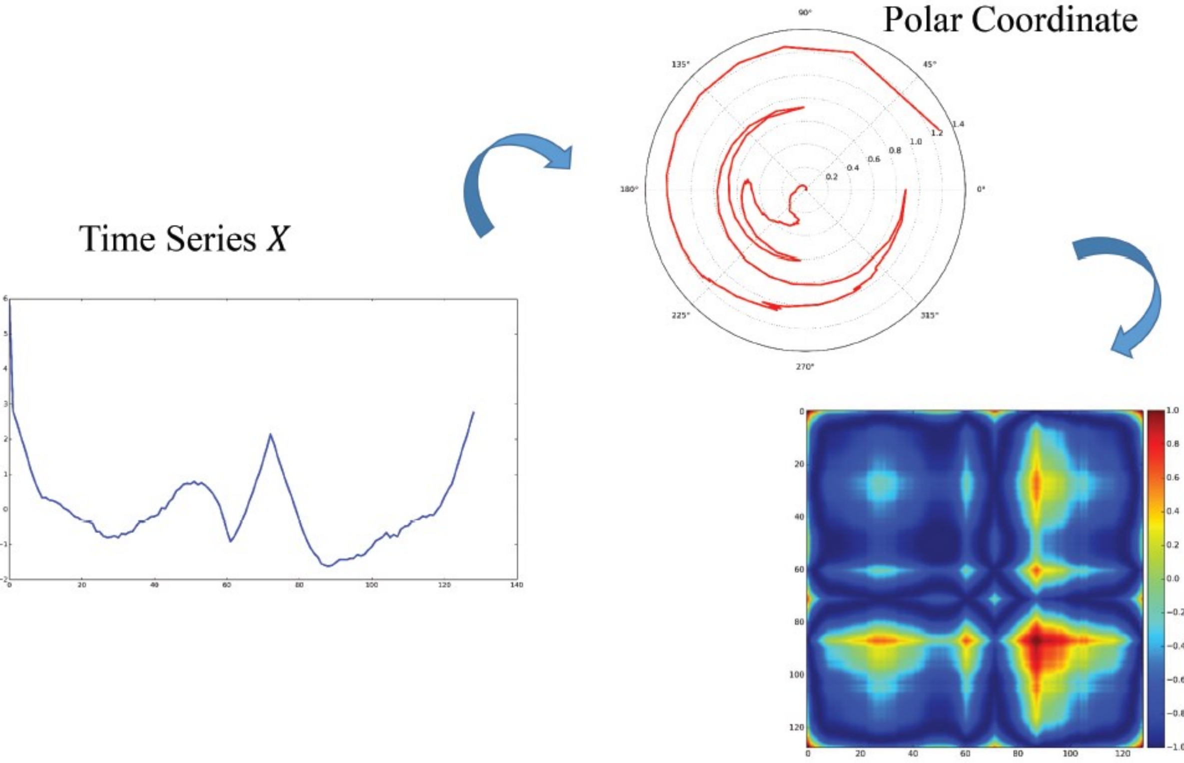

Figure 25: summary of the Gramian Angular Field conversion procedure. Source: (Z. Wang & Oates, 2015)

The results are images that can be examined with large pretrained Computer Vision models, and they preserve the original information and time relationships. Figure 26 provides an example of three different displacement-only observations.

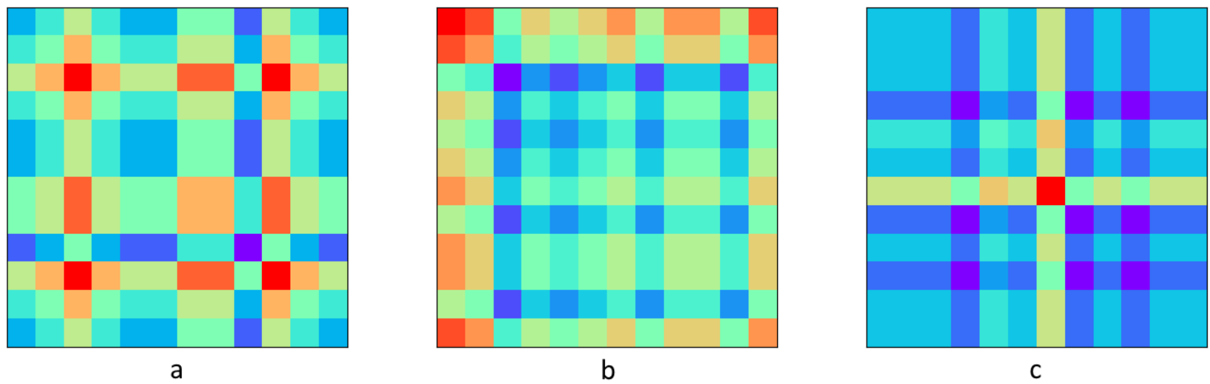

Figure 26: a, b, and c are examples of observations converted to images with the GAF method. Data comes from real displacements measured in site A. Images constructed in this way are always symmetrical along the main diagonal.

One disadvantage is that they are not a compact way to represent information, moreover only half of the image is needed because it is specular along the main diagonal. The image is created in

this way because NNs need square or rectangular images, so even if half is technically redundant, it is kept.

We were not able to find appropriate pre-trained NN for time series prediction, but there are many pre-trained image-classification models (for explanation and motivations for pre-trained NNs see Section "2.4.5. Transfer Learning and Fine-tuning"). Two NNs, RegNet (Radosavovic et al., 2020) (pretrained implementation at the link https://pytorch.org/vision/stable/models/regnet.html) and Swin Transformer (Liu et al., 2021; Liu, Hu, et al., 2022) (pretrained implementation at the link https://pytorch.org/vision/stable/models/swin_transformer.htmlhttps://pytorch.org/vision/stable/models/regnet.html) are selected among Torchvision models. Torchvision (https://pytorch.org/vision/stable/index.html) is a computer vision library, part of PyTorch (https://pytorch.org/), which is an open-source Machine Learning framework. The criterion for the choice is the two models' excellent performance on the ImageNet Challenge (Russakovsky et al., 2015), for which they are trained.

The decision layer is replaced with a new layer that changes the output from ImageNet categories to displacements.

4.2. Tree species classification

This part of the project was developed in collaboration with the Smart Forest Lab, from the Faculty of Agriculture of Yamagata University (Japan). The algorithm and results obtained have also been published as a journal article "Plant Species Classification and Biodiversity Estimation from UAV Images with Deep Learning" (Conciatori, Tran, et al., 2024).

This algorithm examines drone images of trees and identifies the tree's species. This is a classification problem, meaning that the correct answer, or target, is a discreet value, in this case the classes are the five known tree species.

4.2.1. Motivation

The mutual interaction between plants and landslides is a well-known phenomenon, but in recent years some studies were able to better understand and quantify it. The interaction goes both ways, but this study is concerned with landslide forecasting, so the objective is only obtaining and using information about vegetation coverage for landslides prediction. Here's a breakdown of the key mechanisms that influence landslide susceptibility:

1. Mechanical Reinforcement:

- Root systems act as a mechanical reinforcement for the soil, increasing shear strength and reducing susceptibility to downslope movement. (S. Wang et al., 2020). Dense root networks bind soil particles together, preventing erosion and enhancing soil cohesion (Schmidt et al., 2001).

2. Hydrological Regulation:

- Vegetation intercepts rainfall and roots modify water infiltration and flow dynamics. This changes pore pressure within the slope, a trigger for landslides (Lann et al., 2024).
- Transpiration by plants draws water from the soil, further lowering pore pressure and promoting slope stability (Chirico et al., 2013).

3. Slope Morphology:

- Vegetation cover reduces the impact of raindrop splash erosion, which can weaken the soil surface and increase susceptibility to landslides (Lann et al., 2024).

4. Additional Considerations:

- The type and density of vegetation plays a significant role. Deep-rooted trees have different effects water infiltration and mechanical reinforcement on slope stability compared to shallow-rooted ones (Lann et al., 2024; Y. Li et al., 2021).
- Conversely, in some cases, dense vegetation increases slope weight, potentially contributing to instability (Lann et al., 2024).

This connection means that the presence (and characteristics) or absence of vegetation can be used as additional information for evaluating landslide susceptibility.

The relationship between vegetation cover and landslides is especially interesting when compared to other factors like terrain geology or slope angle, because it can be modified by people. This means that with an in-depth understanding of the phenomenon it is possible to reduce landslide risks by planning in advance for the appropriate presence of plants in hazardous zones.

There are also interesting studies on the opposite effect: the influence of landslides and terrain instability on vegetation species and biodiversity (Asada et al., 2020; Hu et al., 2018; Myster &

Walker, 1997; Restrepo & Vitousek, 2001). However the objective of this research is to understand and predict landslide collapse, so this side of the connection is not examined in depth.

4.2.2. Original algorithm

Similarly to the landslide forecasting, the first attempt was made with custom NNs trained from scratch on the tree patches available, but this attempt was abandoned very early in favor of pre-trained models. This is why, even though the developed code allows for the creation of custom CNNs (which are very adept at image examination), all the experiments and considerations are about pre-trained models.

This project uses pre-trained models available through Torchvision, however, since there are too many to test exhaustively, four are selected:

1. ResNet (He et al., 2015) (https://pytorch.org/vision/stable/models/resnet.html)
2. RegNet (Radosavovic et al., 2020) (https://pytorch.org/vision/stable/models/regnet.html)
3. ConvNext (Liu, Mao, et al., 2022) (https://pytorch.org/vision/stable/models/convnext.html)
4. Swin Transformer (Liu et al., 2021; Liu, Hu, et al., 2022) (https://pytorch.org/vision/stable/models/swin_transformer.html)

ResNet is the oldest but still relevant, and it is a good baseline because it has been used in many other similar studies. The other three models were chosen because they obtained some of the highest scores on the ImageNet Challenge (Russakovsky et al., 2015), an important benchmark for image classification. ResNet, RegNet and ConvNext are Convolutional Neural Networks, which were developed exactly for image elaboration. Swin belongs to the Transformer family, which were designed for NLP (natural Language Processing) tasks but found many other applications, including image classification.

The project is structured to be as general as possible: even though these four architectures are used, any other image classification NN from Torchvision can be added. The only information required for the project to work with a new model is a pointer to its decision layer, because that is the part of the Network that must be modified, and it is architecture-specific (see Section "2.4.5. Transfer Learning and Fine-tuning").

The model requires specific characteristics for the input patches: square shape, top-down viewpoint, must contain mostly one species, this species must be among the ones the model

knows. The preprocessing described in Section "3.2.4. Data preprocessing" ensures the input data fulfills those conditions.

The output is a vector of length $n = 5$ (number of species), it is a probability distribution over the five known classes. Since a single-class answer is required by the loss function, the class with the highest probability is interpreted as the model's output with Equation 24.

Knowledge of the probability distribution gives much more information, and it helps understand how confident the model is in its predictions. For example, both outputs of Table 17 are converted into the same answer "*Fagus crenata*" because $arg\ max(\text{Output 1}) = arg\ max(\text{Output 2}) = 1$, which corresponds to the first class of Table 11, "*Fagus crenata*". In the case of Output 1, however, it is possible to see the algorithm is almost certain in its answer, while for Output 2 all the species have very similar probabilities.

Table 17: example outputs for which knowing the uncertainty (instead of just the top class) is useful.

Class	**Output 1**	**Output 2**
Fagus crenata	99.6%	20.4%
Abies mariesii	0.1%	19.9%
Chengiopanax sciadophylloides	0.1%	19.9%
Sorbus commixta	0.1%	19.9%
Pinus spp.	0.1%	19.9%

The algorithm is also able to automatically calculate the biodiversity of a group of input images according to three different standards: Species Richness, Gini-Simpson Index, and Shannon-Wiener Index (Jost, 2006). This is not directly related to landslides, it is a function required by the researchers of Yamagata University as part of the collaboration for the development of this algorithm.

4.2.3. Orthomosaic input

The previous approach tries to automate vegetation information from UAV photos to ultimately use it in landslide forecasting. The problem with this method is that the species recognition algorithm, unlike landslide predictions, requires human intervention for two specific sub-tasks in order to work.

1. Extraction and classification of patches from orthomosaics for training the NN.
2. Extraction of patches on which to use the trained model.

The first point is not a significantly limiting factor because manually creating the training dataset is a one-time effort. We also conducted experiments to gauge the minimum quantity of patches needed, and it is in the order of tens per species (see “5.2.2. Relationship between data quantity and model performance”), a very low amount for typical DL applications.

The second point means that once a model has been trained and is ready, it is not sufficient to create an orthomosaic of the site of interest with drones to use it. There is also the need to manually extract patches from the orthomosaic, making sure this subsample is representative of the vegetation of the area. Manual extraction is required because each patch should be centered on a single tree (or contain mostly one species when trees are difficult to distinguish) for the algorithm to give reliable results.

To overcome this limitation a different version of the tree species classification algorithm was developed. The most significant difference is that the inputs for the new version are not patches but a whole orthomosaic (or any sub-part of it), which removes all the need for patch extraction of point 2. The output is also different: it is a 3-dimensional matrix Y of size $w \times h \times n^{+}$, $w =$ *input orthomosaic width*, $h =$ *input orthomosaic height*, $n^{+} = 1 +$ *number of classes the model is trained on*. This means that for each pixel with coordinates $(x, y), x \in \{0,1, \dots w\}, y \in \{0,1, \dots h\}$, $Y_{x,y}$ is a mono-dimensional vector with one element per class, $|Y_{x,y}| = n^{+}$. The value of each element of this vector is the probability that the pixel belongs to a tree of the corresponding class. The same concept can be expressed as: the value of element $Y_{x,y,c+1}$ is the probability that the pixel with coordinates (x, y) in the orthomosaic is part of a tree of species c. The extra class represents pixels that do not belong to any known class, either because they are species that are not known by the model, or because they are not vegetation at all. This addition is made because in a whole orthomosaic it is unreasonable not to expect any extraneous elements. For an input-output example see Figure 27.

This algorithm internally uses the model developed for the original version: the training is unchanged, and it still uses manually extracted and labelled patches. Since this is the case, in this subsection “algorithm” and “model” will not be used interchangeably like in the rest of the thesis. “Algorithm” will be used to indicate the whole procedure, while “model” will be the NN that is used inside the algorithm. For the same reason, there is a distinction between input and output of the algorithm (respectively orthomosaic and 3-dimensional matrix), and “partial” input and output of the model (respectively patch and 1-dimensional vector).

Only at inference time inputs and outputs are different. This is achieved by adding a layer of software that automatically extracts patches from the input orthomosaic and passes those to the

underlying model. The model only gives good results for patches that contain one species, and there is no way of extracting only those without human supervision. We address this problem by systematically making the model look at all the orthomosaic through a sliding window with the size of a normal patch (100×100 pixels). The window flows over the orthomosaic, each time moving by a distance that is less than its length, so that each pixel is examined more than one time. The partial output of the model is recorded for each window position by adding 1 to the values of the corresponding window in the tridimensional output matrix. For example, let's imagine the 100×100 sliding windows is in position [30, 50] (the top-left corner of the sliding window is 30 pixels to the right and 50 pixels below the top-left corner of the orthomosaic) and the model predicts class 3 for this patch. Then all the elements of Y in this range are increased by 1: $Y_{30-130,50-150,3} \coloneqq Y_{30-130,50-150,3} + 1$. In more general terms, when the sliding window has coordinates x_1, y_1 and the prediction is class c, then Y is updated with the rule expressed by Equation 25:

$$Y_{x_1-x_1+100,y_1-y_1+100,c} \coloneqq Y_{x_1-x_1+100,y_1-y_1+100,c} + 1 \tag{25}$$

The predictions (or partial outputs) given by the model for each position of the sliding window are in the same format of probability distribution as explained in the original algorithm. With the definition of a confidence threshold ct, it is possible to separate cases in which the model was certain of the answer from cases in which it was less confident. If the highest probability of the partial output is lower than the confidence threshold $max(\rho) < ct$, then the algorithm considers this as belonging to the extra "unknown" class. Which means that the operation to compute the predicted class is no more Equation 24, it uses Equation 26:

$$c = \begin{cases} arg\ max(\rho), & \text{if } max(\rho) \geq ct \\ n^+, & \text{otherwise} \end{cases} \tag{26}$$

With $c =$ predicted class, $\rho =$ probability distribution of the partial output, $n^+ = |\rho| + 1 =$ number of classes $+1, ct =$ confidence threshold.

Finally, the values inside Y are normalized by dividing each 1-D vector $Y_{x,y}$ by the number of times the corresponding pixel has been part of a sliding windows. This passage is necessary because pixels near the borders are part of sliding windows less frequently, so without proper normalization their contribution would not be weighted correctly.

The output can be represented graphically to see the species distribution over the input orthomosaic (Figure 27).

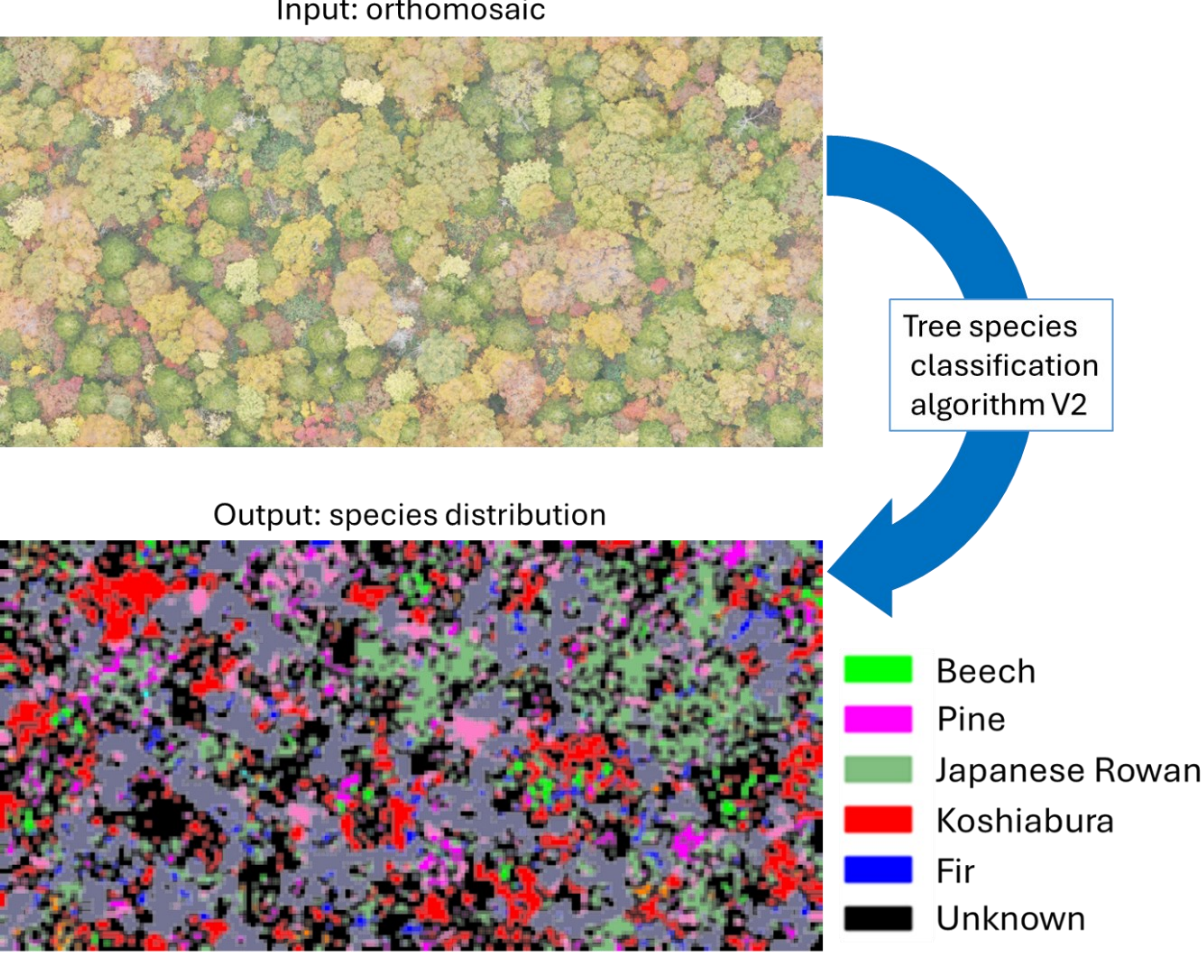

Figure 27: input/output example of the second version of tree species classification algorithm. It takes a whole orthomosaic as input, and outputs an image with the same shape color-coded by species distribution.

The real advantages, however, are the following two.

(1) The input is an orthomosaic, which can be obtained relatively easily given the appropriate equipment.

(2) The output is the species distribution over the orthomosaic area, in a format (3-D matrix) that can be directly used by other algorithms and NNs.

Since input and output are custom complex data, metrics to evaluate this version of the algorithm had to be designed ad-hoc and are presented in Section "4.3.3. Metrics for tree species recognition with orthomosaic input".

4.3. Custom metrics development

4.3.1. Background and Motivation

In Machine Learning, the selection of appropriate metrics is crucial for evaluating model performance accurately. Metrics provide a quantifiable means to assess how well a model performs and can highlight strengths and weaknesses that guide further model development and optimization. Metrics are the cornerstone of model evaluation in machine learning, serving as the primary tools for assessing and comparing algorithm performance. Good metrics offer insights into the effectiveness of models, inform decisions about model adjustments, and ultimately contribute to the success of machine learning projects. However, selecting inappropriate metrics can lead to misleading conclusions and suboptimal models, adversely affecting real-world applications.

The success of Machine Learning is relatively recent, and since then it has been developing at a very fast pace. As can be seen from Figure 28, the number of publications related to ML produced until 2012 are less than the papers on the same argument of the single year 2022.

Figure 28: ML-related articles published. Source: (Conciatori, Valletta, et al., 2024).

Many different fields tried to apply newly proposed algorithms and techniques to a variety of real-world applications, including geotechnics and civil engineering. One of the problems that emerged in the rush to apply new ML models in other domains is the lack of rigorous testing with appropriate metrics, and the absence of common benchmarks for consistent cross-work comparison. These considerations led to the conceptualization and publication of “Improving the quality evaluation process of machine learning algorithms applied to landslide time series

analysis” (Conciatori, Valletta, et al., 2024). The paper addresses the lack of standardized evaluation metrics for the specific problem of landslide forecasting. So, the solution takes into account common problems for this subject like strong dataset imbalances and the possibility of having both classification and regression algorithms. Imbalance is not limited to landslide-related tasks, but it is doubly important in those cases because typically the vast majority of observations are of static or slow-moving terrain, the proportion of measurements of pre-collapse displacements are a very small proportion of the total (if they are present at all). So, landslide dataset present strong imbalances, which in general leads to the smaller classes being under-predicted or not recognized well. The whole point of landslide forecasting and Early Warning Systems, however, is exactly being able to recognize pre-collapse signals.

4.3.2. Metrics for landslide forecasting and tree species recognition

The paper introduces modified metrics designed to be sensitive to class imbalance and applicable to both classification and regression models. The primary metrics discussed are Accuracy, Precision, Recall, and F1-Score, each highlighting different aspects of model performance. Those are metrics typically employed for classification algorithms. To enable consistent comparison between regression and classification algorithms, the output and targets of regression algorithms are discretized into a finite set of classes. This is done after each prediction, so that the algorithm itself is not altered in any way. Afterwards a confusion matrix records the true vs. predicted occurrences for each class, enabling the calculation of the aforementioned metrics.

A multi-class confusion matrix (Ting, 2017) is an extension of the binary confusion matrix used for evaluating the performance of a classification algorithm when there are more than two classes. It summarizes the prediction results of a classification model by comparing the actual and predicted classes for each instance in the dataset. For a classification problem with n classes, the confusion matrix is an $n \times n$ matrix, where n is the number of classes. Each row represents the instances of an actual class, while each column represents the instances of a predicted class. Figure 29 is an example of multiclass Confusion Matrix for a problem with three classes.

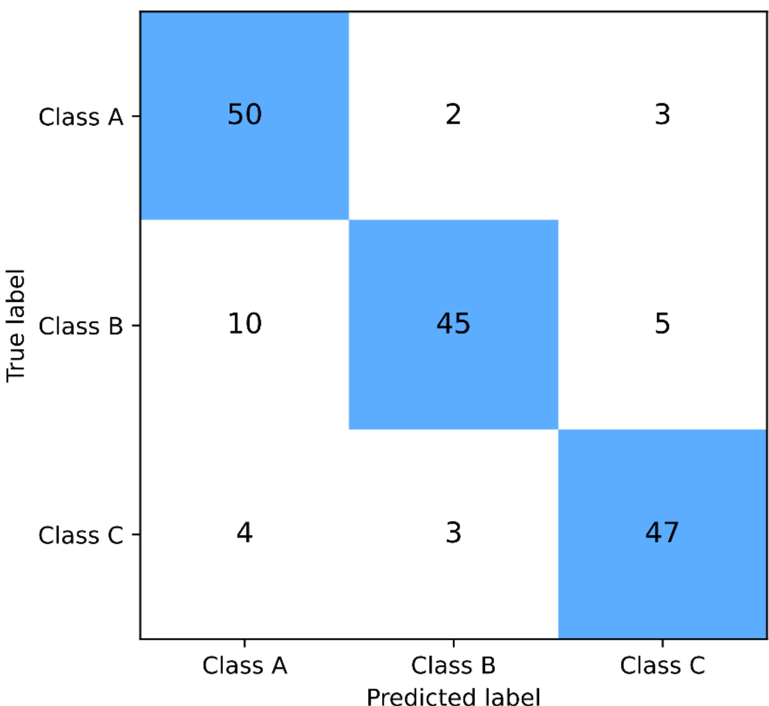

Figure 29: Confusion Matrix example. The main diagonal is highlighted with light-blue color.

The number 50 in the first row and column represents 50 instances in which the models correctly recognized class A. The number "3" in row 1, column 3, means that there were three cases in which the model wrongly classified class A elements as class C elements. In general, all the elements on the main diagonal (light-blue background) are correct classifications, while all the others are misclassifications.

From the confusion matrix True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) (les, 1986; Rothman, 2010) can be computed on a by class basis.

- TP_i = elements of class *i* that are rightfully classified by the model.
- FN_i = elements of class *i* that are not recognized by the model (they are predicted as a different class).
- TN_i = elements not of class *i*, correctly not predicted as class *i* data.
- FP_i = elements not of class *i*, incorrectly predicted as belonging to class *i*.

Note that depending on the class currently considered, the same cell can hold a different meaning. For example, the "10" in the second row, first column, represents 10 False Positives for class A (the model classified them as class A but they are not), but 10 False Negatives for class B (they are of class B but the model gave as output a different class).

Finally, the actual metrics can be calculated using Equations 27-30:

$$\text{Accuracy} = \frac{\text{correct predictions}}{\text{total predictions}} \tag{27}$$

$$\text{Precision} = \frac{1}{n}\sum_{k=1}^{n}\frac{\text{TP}_k}{\text{TP}_k + \text{FP}_k} \tag{28}$$

$$\text{Recall} = \frac{1}{n}\sum_{k=1}^{n}\frac{\text{TP}_k}{\text{TP}_k + \text{FN}_k} \tag{29}$$

$$F_1 = \frac{1}{n}\sum_{k=1}^{n}\frac{2 \times \text{TP}_k}{2 \times \text{TP}_k + \text{FP}_k + \text{FN}_k} \tag{30}$$

n is the number of classes. Accuracy is the only metric that does not use the intermediate results for its calculation, nor the class division, it is just the proportion of correct answers given by the model. It is useful to get a rough assessment of the model's performance but can miss subtle problems. The other metrics are computed by taking the average of their class-wise results. Precision, Recall, and F_1 calculated in this way are sensitive to imbalances in the training dataset and help identify if this causes issues in the trained models. This concept can be illustrated by an example. For hypothesis, the dataset has two classes, with 99% of the elements belonging to class A, 1% to class B. Any metric that is not calculated per-class and averaged out will be almost insensible to how well the models recognize class B. An algorithm that, instead of computing a prediction, always says "class A" with any input, will get an Accuracy of 99%, which is extremely high, considering it is not very good. Recall from equation 29, however, would make clear that this model has problems, giving an evaluation of 50%. In general, the left graph of Figure 30 show how standard Recall and Recall from equation 29 react to imbalance in the dataset. The right graph of Figure 30 displays the same information for F_1 Score (Equation 30).

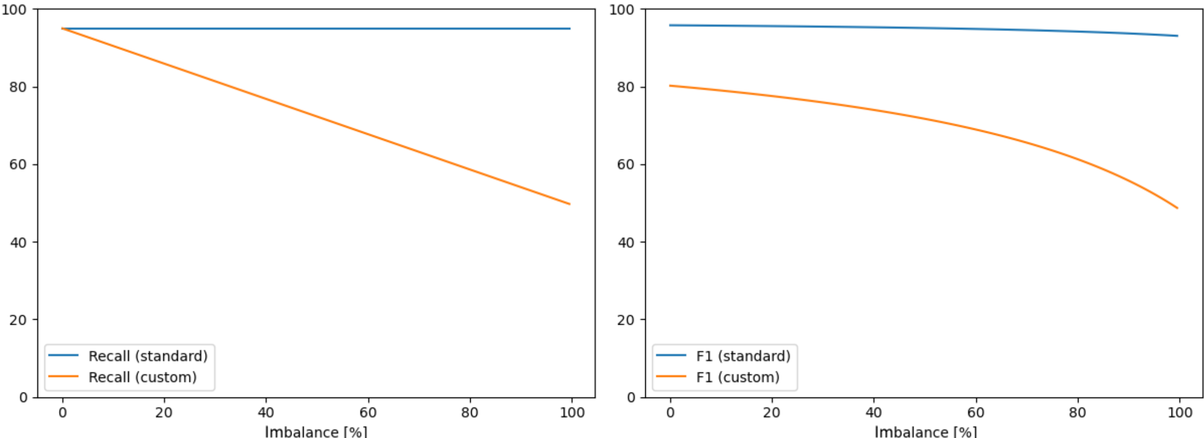

Figure 30: Standard and custom Recall values for varying degrees of imbalance (left). Standard and custom F1 values for varying degrees of imbalance (right). Source: (Conciatori, Valletta, et al., 2024).

For an in-dept explanation and the quantification of the imbalance used in x axis see (Conciatori, Valletta, et al., 2024). Calculating the metrics class by class and averaging the results (not weighted by the number of elements of the classes) produces evaluations that are useful for detecting problems in classification of the underrepresented classes.

4.3.3. Metrics for tree species recognition with orthomosaic input

This algorithm outputs a 3-dimensional matrix Y of dimensions $w \times h \times n^{+}$, $w =$ *input orthomosaic width*, $h =$ *input orthomosaic height*, $n^{+} = 1 +$ *number of classes the model is trained on*. To evaluate its performance, two ingredients are necessary:

(1) the targets which are manually identified species in orthomosaics.

(2) Metrics able to compare those targets with the 3-dimensional output.

The custom metrics calculate True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) once for each class with Equations 31-34.

$$TP_c = \sum_{x=1}^{w} \sum_{y=1}^{h} \begin{cases} 1, if \left(BMT_{x,y,c} \text{ AND } BMP_{x,y,c}\right) = \text{TRUE} \\ 0, \text{otherwise} \end{cases} \tag{31}$$

$$TN_c = \sum_{x=1}^{w} \sum_{y=1}^{h} \begin{cases} 1, if \left((\text{NOT } BMT_{x,y,c}) \text{ AND } (\text{NOT } BMP_{x,y,c})\right) = \text{TRUE} \\ 0, \text{otherwise} \end{cases} \tag{32}$$

$$FP_c = \sum_{x=1}^{w} \sum_{y=1}^{h} \begin{cases} 1, if \; \big((\text{NOT } BMT_{x,y,c}) \text{ AND } BMP_{x,y,c}\big) = \text{TRUE} \\ 0, \text{otherwise} \end{cases} \quad (33)$$

$$FN_c = \sum_{x=1}^{w} \sum_{y=1}^{h} \begin{cases} 1, if \; \big(BMT_{x,y,c} \text{ AND } (\text{NOT } BMP_{x,y,c})\big) = \text{TRUE} \\ 0, \text{otherwise} \end{cases} \quad (34)$$

With c current species, w and h width and height of the orthomosaic, (x, y) position of the element, $x = \{1,2, \dots w\}, y = \{1,2, \dots h\}$. BMT means Boolean Mask of the Target, and it is a 3-D matrix in which the first two dimensions are the same as the target orthomosaic w, h. The third dimension has size n^+. Each element has a Boolean value: element (x, y, c) is TRUE if the pixel of coordinates (x, y) is of class c, FALSE otherwise. BMP means Boolean Mask of the Prediction, and it is analogous to BMT. “NOT”, and “AND” are standard logical operators.

While the calculations for TP, TN, FP, and FN are different from before, the actual metrics derived from them are the same as the ones described previously, and they are calculated using the same Equations 27-30.

4.4. Interpolation and graphical representation of displacements

Depending on its extension and importance, a monitored site can have many MUMS and other measurement devices. To help visualize and interpret the displacements, both measured and predicted, a specific tool was developed. This visualization tool takes displacements measured from three different points on a site, along with the coordinates of these points, and creates a semi-transparent overlay on a map of the site (Figure 31).

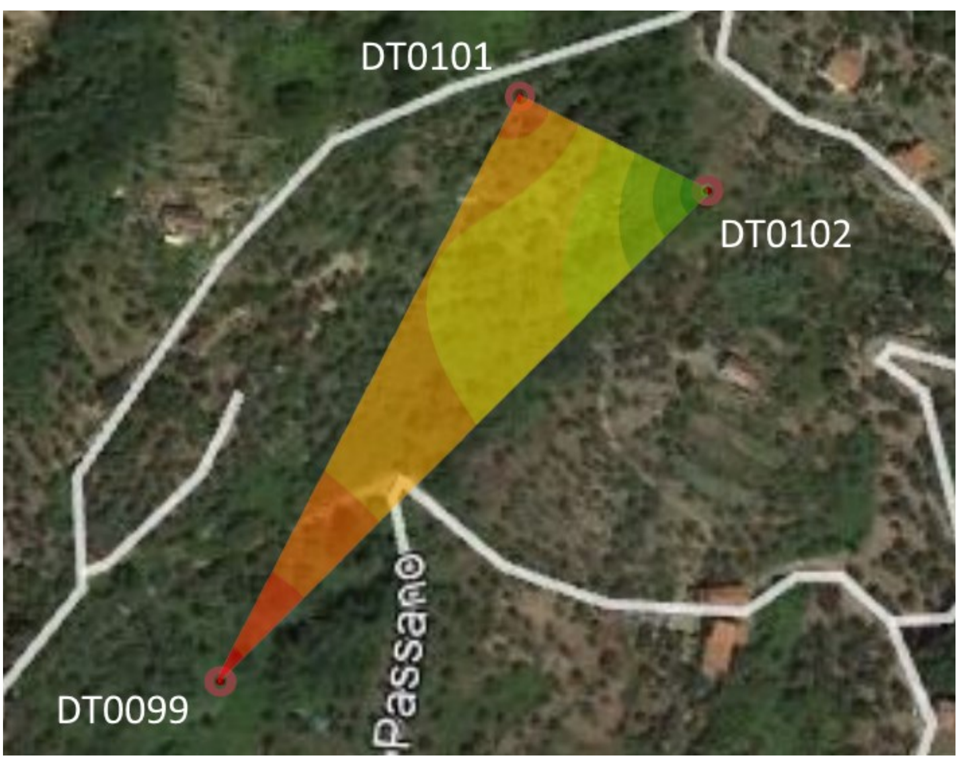

Figure 31: site A map with interpolated displacement overlay between three sites. Source: (Conciatori et al., 2022). Red dots mark the position of Vertical Arrays.

The overlay changes color from red to green depending on the magnitude of the displacements and automatically interpolates the unknown values between the three points. By repeating this operation all the surface of a site can be colored with this procedure. The project up to his point, including the graphical visualization tool, was presented at the 14th International Conference on Geostatistics for Environmental Applications (GeoENV 2022) (Zanini & D'Oria, 2022) in Parma with the title "Importance of multi-parameter approaches in the development of Machine Learning algorithms for landslide displacement forecasting" (Conciatori et al., 2022).

4.5. Hyperparameter space exploration

A hyperparameter in Machine Learning is a configuration parameter set before the learning process begins, which determines the behavior and performance of the model. It differs from model parameters (also called weights), which are learned from the training data (Goodfellow et al., 2016). Hyperparameters are set manually and include aspects like learning rate, number of epochs, batch size, and architecture-specific settings such as the number of layers or neurons in a neural network. Depending on the task, hyperparameters can be difficult to calibrate a-priori but can have large consequences on the resulting model (Claesen & De Moor, 2015; Hutter et al., 2019; L. Yang & Shami, 2020). For this reason, the proposed framework includes a function that performs a grid search on all the desired hyperparameters, by specifying the list of values to try

for each of them. This means that the function automatically trains a model for each combination of values and confronts their performances.

Since grid search tries all the possible combinations of values for each hyperparameter, this number grows faster than the exponential function $f(x) = h^x$ where x is the number of hyperparameters and h is the number of values of the hyperparameter with the least number of values. This problem of exponential growth in complexity with the linear growth of the number of variables is called in ML and related fields "curse of dimensionality" (S. Bengio & Bengio, 2000; Kuo & Sloan, 2005).

For example, let's consider a simple case with three hyperparameters: *learning rate* (with values to try $v_{lr} = \{10^{-5}, 10^{-4}, 10^{-3}\}$), *number of epochs* (with values to try $v_e = \{10, 15, 20\}$), and *batch size* (with values to try $v_{bs} = \{16, 32\}$). If only the learning rate was specified, the function would train three models, because $|v_{lr}| = 3$. By adding number of epochs, the number of models to train becomes $|v_{lr}| \times |v_e| = 9$. Searching over all three hyperparameters the number of combinations is $|v_{lr}| \times |v_e| \times |v_{bs}| = 18 \geq h^x = 2^3$, $\min(|v_{lr}|, |v_e|, |v_{bs}|) = 2$, number of hyperparameters $= 3$.

To use this type of search effectively, first a coarse exploration is conducted, so that the number of values for each hyperparameter is kept low, but their range is high. Afterwards, a second search is conducted, by restricting the ranges of each hyperparameter to what produced the best results in the previous one. If necessary, this process can be used multiple times further reducing the searchable space with each iteration, but in our case two steps were sufficient to find the optimal combination of hyperparameters.

The list of hyperparameters (and corresponding description) considered in this study is the following.

- Number of training epochs: in each epoch all the training data are fed to the NN and used to improve its performance. The number of training epochs is the number of times all the training data were used in such a fashion.
- Optimizer: it is the algorithm that changes the weights of the NN during training based on the difference between model's predictions and true predictions
- Learning Rate: it is the speed with which the optimizer alters the weights. A high LR produces faster initial convergence but may lead to fluctuations and difficulty in fine-tuning. Conversely low LR requires many training samples to converge.

- Weight Decay: allows us to combine the best characteristics of high and low Learning Rates. The model starts with a high LR, which is progressively reduced according to Weight Decay.
- Data Augmentation: techniques by which the quantity of training data can be increased with slightly altered copies of themselves.
- Data Balancing: procedure for compensating imbalanced datasets. The quantity of data belonging to smaller classes is increased with the Data Augmentation techniques.
- Frozen Layers: this hyperparameter is only used for fine-tuning pretrained NNs. It controls whether all the layers or only the decision layer are changed during the fine-tuning.
- Rainfall data (and *rds*): presence and eventual shift of precipitation information.
- *ndi* and *ndo*: length of input and output time series respectively.
- Batch size: how many observations are examined at a time.
- Loss function: which function will be used to measure the error between NN predictions and true results.
- Network shape: depending on the version of the program, there can be multiple hyperparameters with varying levels of control.
- Shift thresholds: thresholds used to separate the displacements in distinct classes.
- Data split: percentage of data to use as training, validation, and test sets.
- Max loss coefficient: only valid for the TSM loss function, determines how much it differs from MSE.

The framework supports the graphical visualization of results with 1-hyperparameter of 2-hyperparameter combinations (only one or two hyperparameters are searched) because they require 2D and 3D graphs respectively. Figure 32 is a 3D graph example. When more hyperparameters are varied simultaneously, it is not useful to represent the results as N-dimensional graphs, so they are simply printed as text information.

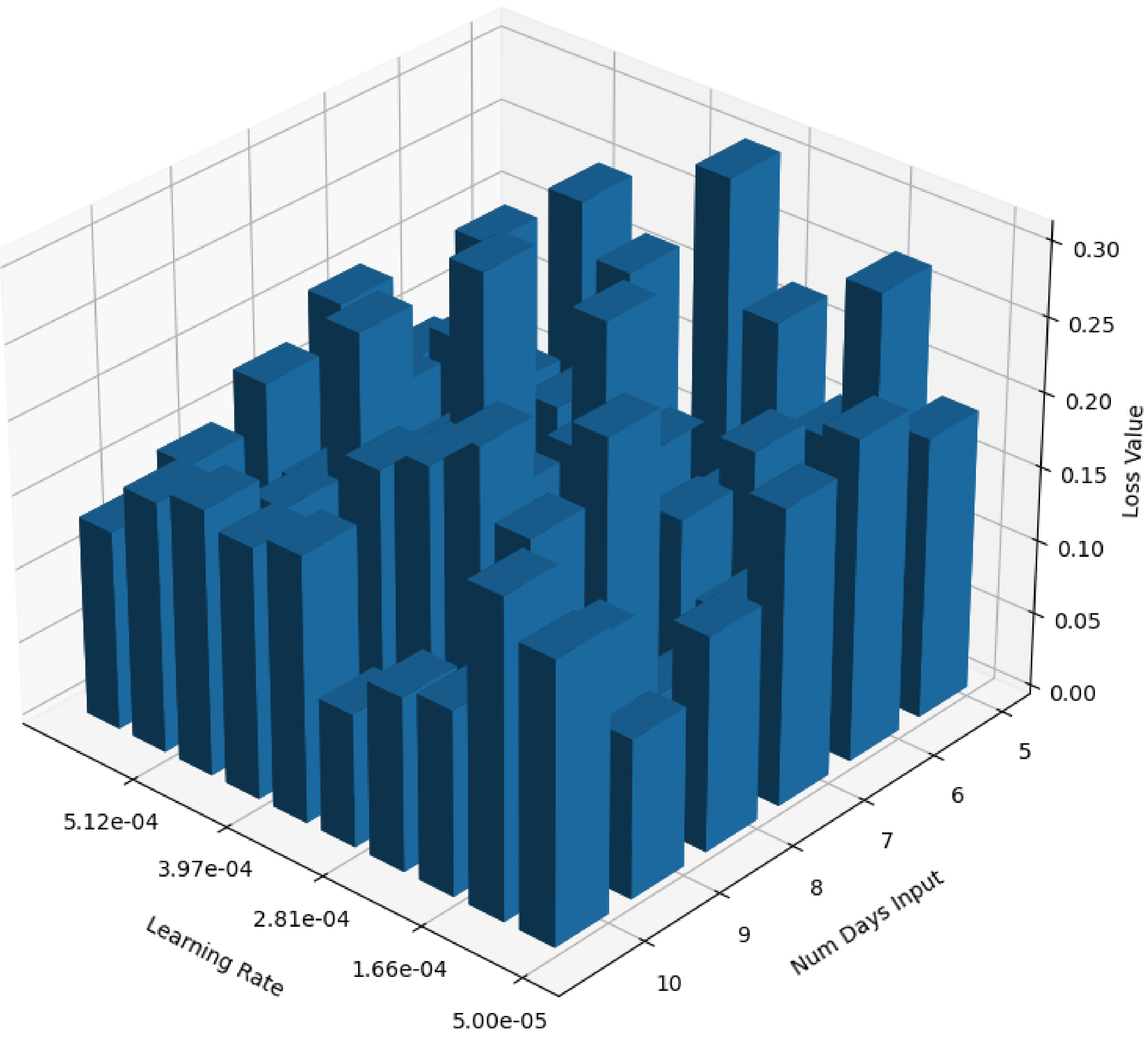

Figure 32: example of hyperparameter search results with two hyperparameters variable (Learning Rate and ndi). All other hyperparameters are fixed on their initial values. The height of a bar represents the performance of a model trained with the corresponding LR and ndi. Since loss is an error with respect to true displacements, shorter bars mean better models.

4.6. Implementation details and code availability

All the code is written using Python programming language (Van Rossum & Drake, 2009) (https://www.python.org/) with the addition of libraries for expanding some functionalities. PyTorch (Imambi et al., 2021; Paszke et al., 2019) (https://pytorch.org/) is the most important one as it handles ML-related operations. The earliest versions of this project relied on TensorFlow 2 (Abadi et al., 2016) (https://www.tensorflow.org/), but a change was required because PyTorch allows for more control over the tensors while training, which was necessary. Pandas (Mckinney, 2011) (https://pandas.pydata.org/) is used to import and manipulate data stored in *csv* files,

NumPy (Harris et al., 2020) (https://numpy.org/) and SciPy (Virtanen et al., 2020) (https://scipy.org/) implement complex mathematical functions.

All data collected for tree species classification project is shared through Dropbox at the link: https://www.dropbox.com/scl/fo/q532q2irrb5jv35s5it6h/AIKRoDgS0-HuJnvsxGIdpWQ/paper_data?rlkey=ntuy2ph8sdg5mtttnk5v8339x&subfolder_nav_tracking=1&st=ifgvvsq5&dl=0

There are also examples of trained models (one per architecture) provided with the same platform: https://www.dropbox.com/scl/fo/q532q2irrb5jv35s5it6h/ABGd_YH3QKAs2Eb9LuQed00/models?rlkey=ntuy2ph8sdg5mtttnk5v8339x&subfolder_nav_tracking=1&st=yvz8n208&dl=0

The code is hosted on the GitHub (https://github.com/) platform https://github.com/marco-conciatori-public, separated into three projects.

- Landslide forecasting, which is not public.
- The code for the custom metrics ("4.3. Custom metrics development") is part of landslide forecasting, but it is still publicly accessible at https://github.com/marco-conciatori-public/ml_algorithms_evaluation as part of the publication "Improving the quality evaluation process of machine learning algorithms applied to landslide time series analysis" (Conciatori, Valletta, et al., 2024).
- Tree species classification, which is publicly accessible (https://github.com/marco-conciatori-public/tree_classification) as part of the publication "Plant Species Classification and Biodiversity Estimation from UAV Images with Deep Learning" (Conciatori, Tran, et al., 2024).

Setting up the environment necessary to run the code, even when it is fully available and well documented, is not trivial and requires programming expertise. For this reason, a Google Colab notebook is also available for tree species classification at https://colab.research.google.com/drive/1Pe9zkwzts3JIi80xxRfrBa-2Pqe87FlK. Colab notebooks (https://colab.research.google.com/) provide computing power (the code runs remotely) and the necessary environment so that a piece of code can be executed without installing anything on the local computer. The plant species classification Colab notebook automatically imports the code from GitHub, downloads the data from Dropbox, trains and tests a Swin transformer model. It is possible to run the training with different hyperparameters, and to execute other experiments and tests on the trained models.

5. Results

This section will present the results obtained by the algorithms described previously. For both landslide forecasting and tree species classification it is also studied the tradeoff between training data quantity and performance, and the runtime depending on the hardware used.

Training NNs involve random elements from multiple sources: weights initialization, composition and order of training, validation, and test dataset from the source data, data augmentation operations... Because of this, each training gives different results, even when the hyperparameters are the same. Thus, when possible, each experiment is executed multiple times and the results shown and discussed are the average of those runs.

5.1. Landslide forecasting

Since the landslide forecasting algorithm went through the significant changes described in Section "4.1. Landslide forecasting", here results are presented for each of the major versions separately. The settings used can change by version, based on what gives the best results for that implementation. Since it is impractical to show all the tests conducted and the combinations of hyperparameters tried, the following sections will only display the configurations that gave the best results, or that are relevant for the discussion. Note also that available settings and hyperparameters change with the versions, since new functions and customizations are gradually added with time.

The earlier versions only use site A data because the other sites were not available from the beginning but were added during the course of the study.

Note that each version of the algorithm in the following sections is displayed with the best hyperparameters found, but that is no guarantee that they are the best possible hyperparameters, and consequently that the results shown are the best possible for that particular implementation. Exploring all the possibilities however is an intractable problem, as explained thoroughly in the "4.5. Hyperparameter space exploration" Section, so this kind of approximation is the best that can be achieved.

5.1.1. Original algorithm

The original algorithm is trained and tested on site A data with the settings and hyperparameters of Table 18.

Table 18: hyperparameters used to test the original algorithm.

Hyperparameters	**Original algorithm**
Epochs	10
ndi	10
ndo	1
Batch size	64
Loss function	MSE
Learning Rate	0.01
Optimizer	Adam
Hidden units	1024, 1024, 1024
Shift thresholds	[1, 2]
Data split	80%, 10%, 10%

ndi = number of days in each observation, *ndo* = number of future days to predict.

"Hidden units" represent the modifiable part of the NN in the earlier versions: how many neurons each dense layer has.

Two shift thresholds means that the data are divided into 3 categories (small, medium, and high displacements) which are dynamically calculated so that:

- small displacements contain 84% of the data (the 84% smallest displacements)
- medium displacements contain 13% of the data (displacements between 84% and 97%)
- high displacements contain the remaining 3% of the data (displacements above 97%)

It uses the standard dataset division of 80% training data, 10% validation data, 10% test data.

Figure 33 shows the performances of the model and baseline algorithm, explained in Section "4.1.1. Comparison algorithm (Baseline)". Only this first version of the landslide prediction algorithm uses the baseline algorithm as comparison. All subsequent implementations use this or later versions as comparison, which are more relevant.

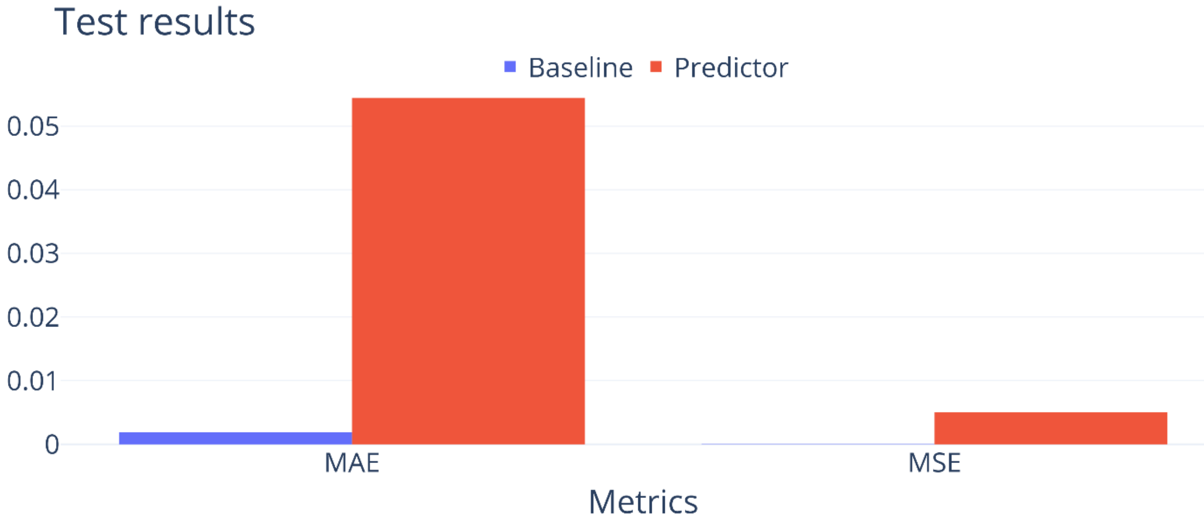

Figure 33: performance of original algorithm and baseline on the test set. MAE = Mean Absolute Error, MSE = mean Squared Error.

The results are the average of 5 independent runs, which is more informative. As we can see, even though the baseline is not a real prediction algorithm, the landslide changes so slowly that, on average, it performs much better than the predictor (original algorithm).

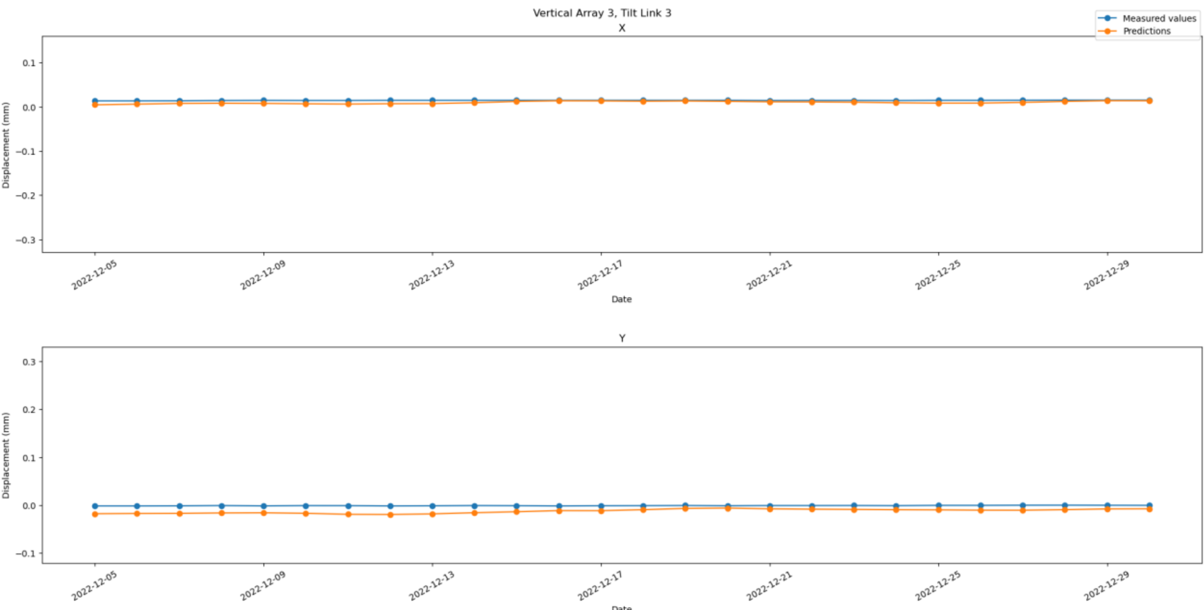

Figure 34: real VS predicted displacements over a month. Both x (above) and y (below) are displayed.

Figure 34 is an example of a comparison of the model's predictions against ground truth over a longer period.

5.1.2. Vertical Array as input

This experiment consists of expanding the input of the algorithm, to give it more context to the detriment of generality. This is because to be used after training, the algorithm needs 20X more data, and more importantly, it requires this data to be provided in the exact same format as the vertical array the algorithm was trained on.

The algorithm is trained on site A data. Table 19 shows the hyperparameters used for this algorithm, which includes a new hyperparameter (in bold) to switch between using or not the new input format.

To highlight the difference between the settings of the two versions, unchanged hyperparameters are represented with a grey-colored empty cell in the right column.

Table 19: hyperparameters selected for testing the algorithm that takes a whole vertical array as input and the model from the previous section.

Hyperparameters	**VA as input**	**Original algorithm**
Epochs	10	
ndi	12	10
ndo	1	
Batch size	64	
Loss function	MSE	
Learning Rate	$5 * 10^{-5}$	0.01
Optimizer	Adam	
Hidden units	1024, 512, 256	1024, 1024, 1024
Shift thresholds	[1, 2]	
Data split	80%, 10%, 10%	
Use single displacements	False	True

Figure 35 compares the performance of this implementation with the original algorithm from the previous section. Even though the calculation necessary to obtain the MAE and MSE are slightly different for the outputs of this algorithm, once obtained, those values can be directly confronted with the other versions'.

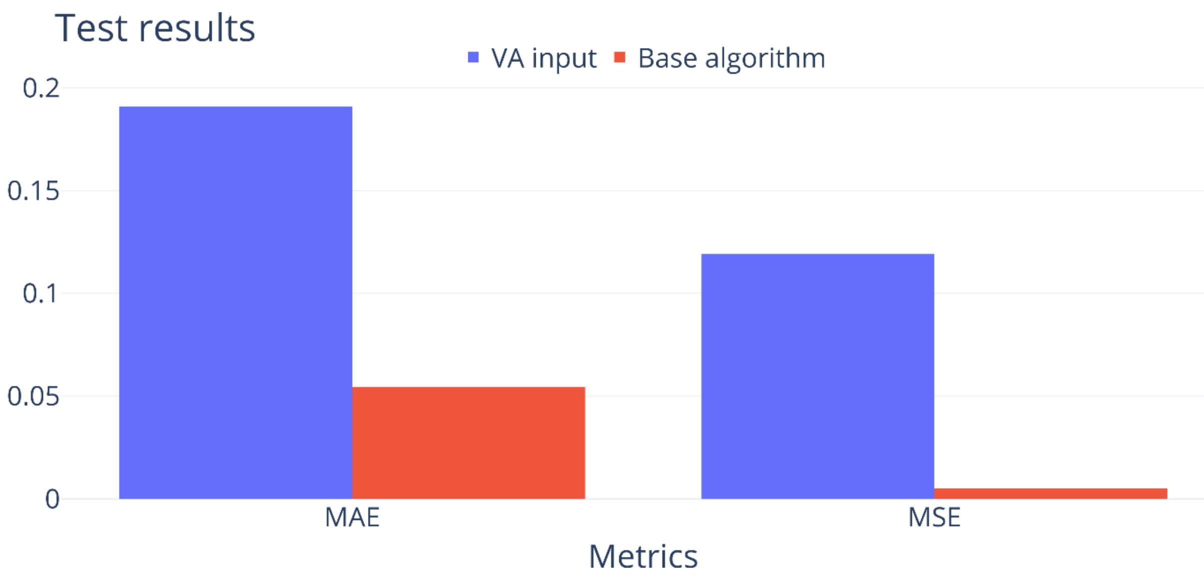

Figure 35: performance of original algorithm and the new version which takes vertical arrays as inputs.

Since the results were worse than the original model, and it had more restricting conditions for its usage, this type of model was abandoned going forward. For the same reason, the related hyperparameter "Use single displacements" will be removed.

5.1.3. Addition of rainfall data

This version of the algorithm adds rainfall data to the input, and also new hyperparameters highlighted in bold. "Use rainfall data" simply controls whether we use the original algorithm "4.1.2. Original algorithm" or the new one with access to precipitation information. Rainfall Days Shift (*rds*) controls the number of days by which precipitation measurements are anticipated with respect to all the other time series. This configuration helps the NN align the cause (rainfall) with the effect (displacement) which can be delayed by many days. By default, it is set to 0, so that no shift is performed, but we also test if using it gives better results.

Three tests are conducted with rainfall data, and for reference, the original algorithm, which does not use rainfall data, is also computed and displayed. The hyperparameters for all three experiments are in Table 20. They all use the same configuration (which is represented by cells with a gray background) except for *rds*.

This also uses only measurements from site A.

Table 20: hyperparameters used for the three tests of the algorithm that uses rainfall data. Most hyperparameters remain unchanged, which is indicated with gray cells. Original algorithm added as comparison.

Hyperparameters	**Test rainfall 1**	**Test rainfall 2**	**Test rainfall 3**	**Original algorithm**
Epochs	10			
ndi	12	12	12	10
ndo	1			
Batch size	64			
Loss function	MSE			
Learning Rate	0.00774	0.00774	0.00774	0.01
Optimizer	Adam			
Hidden units	1024, 256, 16	1024, 256, 16	1024, 256, 16	1024, 1024, 1024
Shift thresholds	[1, 2]			
Data split	80%, 10%, 10%			
Use rainfall data	True	True	True	False
rds	0	3	6	-

Figure 36 shows the results of the three experiments. The plotted values are obtained by running each experiment five times and averaging the results.

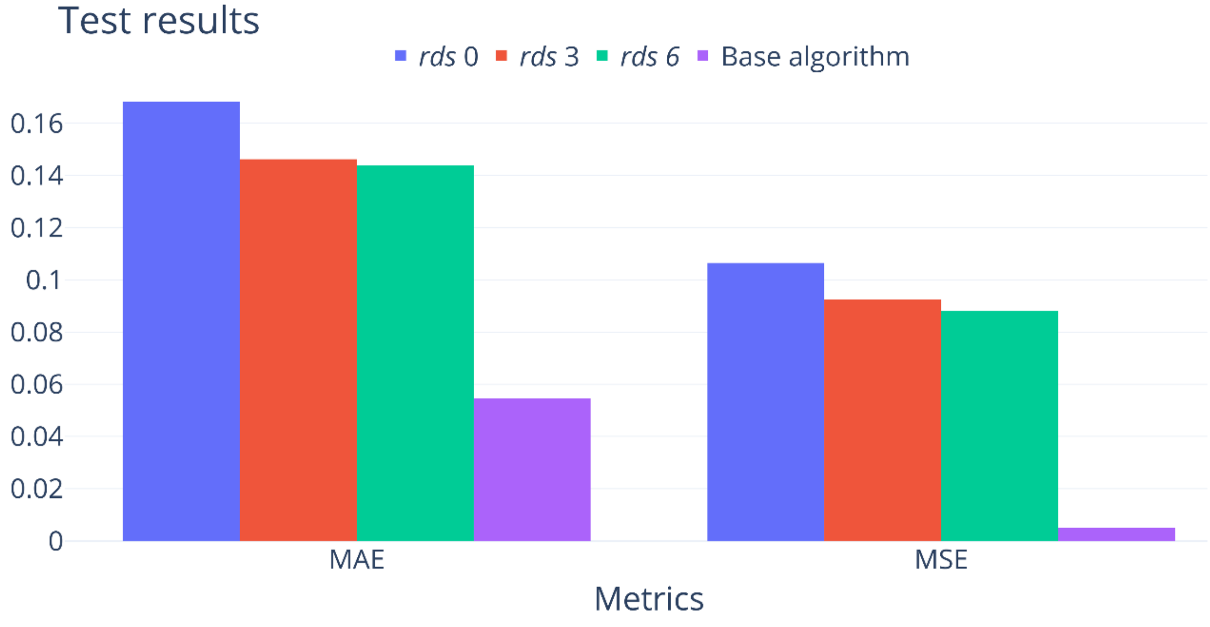

Figure 36: results of the three experiments with rainfall data. The original algorithm, which is the same but without access to precipitation information, is added for comparison.

We see that shifting the rainfall time series has a positive effect on the results, but the original algorithm, without knowledge of precipitation has a better performance.

5.1.4. Custom loss function

This version implements the custom loss function Times Series Midpoint (TSM), which can be used by changing the "loss function" hyperparameter. "Max loss coefficient" is added to the hyperparameters, highlighted in bold. It controls how much TSM punishes bad predictions compared to MSE.

Between the previous and this version of the algorithm, the custom metrics explained in Section "4.3.2. Metrics for landslide forecasting and tree species recognition" were also developed, so from now on they will be used to evaluate the performance alongside the previous MAE and MSE.

Since many variations of the original algorithm are independent from one another (es. access or not to rainfall data and usage of MSE or TSM as loss function) they can be applied in any linear combination, and that has been tried as much as possible. Since the number of configurations grows exponentially, it was impossible to test all combinations, but by using heuristics and trial and errors, all the most promising ones are explored.

In this case the eight following combinations are tested (Table 21 and Table 22). Many more tests were actually conducted to find the best values for the other hyperparameters that here appear fixed, but displaying all those combinations would be impractical and not focus on the last modifications.

Table 21: hyperparameters used for the first four tests (no rainfall data) of the algorithm that uses the TSM custom loss function. Most hyperparameters remain unchanged, which is indicated with gray cells.

Hyperparameters	**Test 1**	**Test 2**	**Test 3**	**Test 4**
Epochs	10			
ndi	12			
ndo	1			
Batch size	64			
Loss function	MSE	TSM	TSM	TSM
Max loss coefficient (*c*)	-	2	5	10
Learning Rate	$5 * 10^{-5}$			
Optimizer	Adam			
Hidden units	1024, 512, 256			

Shift thresholds	0.70, 0.90			
Data split	80%, 10%, 10%			
Use rainfall data	False	False	False	False
rds	-	-	-	-

Table 22: hyperparameters used for the other four tests (with rainfall data) of the algorithm that uses the TSM custom loss function. Most hyperparameters remain unchanged, which is indicated with gray cells.

Hyperparameters	**Test 5**	**Test 6**	**Test 7**	**Test 8**
Epochs	10			
ndi	12			
ndo	1			
Batch size	64			
Loss function	MSE	TSM	TSM	TSM
Max loss coefficient (*c*)	-	2	5	10
Learning Rate	$5 * 10^{-5}$			
Optimizer	Adam			
Hidden units	1024, 512, 256			
Shift thresholds	0.70, 0.90			
Data split	80%, 10%, 10%			
Use rainfall data	True	True	True	True
rds	3	3	3	3

From this version onward there is also another minor change. *Shift thresholds* are now expressed directly in percentages (0.70 and 0.90 means 70% and 90%) instead of number of sigmas from the mean. This does not change the computations made by the algorithm, but it is easier to understand and modify for the user. For example in this case it means that, of all the displacements, the smaller 70% are considered “low displacements”, the 20% between 70% and 90% are considered “medium displacements”, the 10% above 90% are considered “large displacements”.

The hyperparameter *rds* is only relevant when precipitation information is used, and technically changes (from “3” to “undefined”), but in practice it stays the same and is ignored when not necessary, so it is not a variable in this series of experiments. This is represented with the gray

background in Table 21 and Table 22, even though the values are written inside and appear to change.

Figure 37 and Figure 38 show the comparison between the eight algorithms and, as always, the results are the average of 5 experiments for each single category. The first image displays the performance of the models according to the same metrics used up to this point (MAE and MSE). The second image employs the custom metrics developed to better evaluate this kind of algorithm.

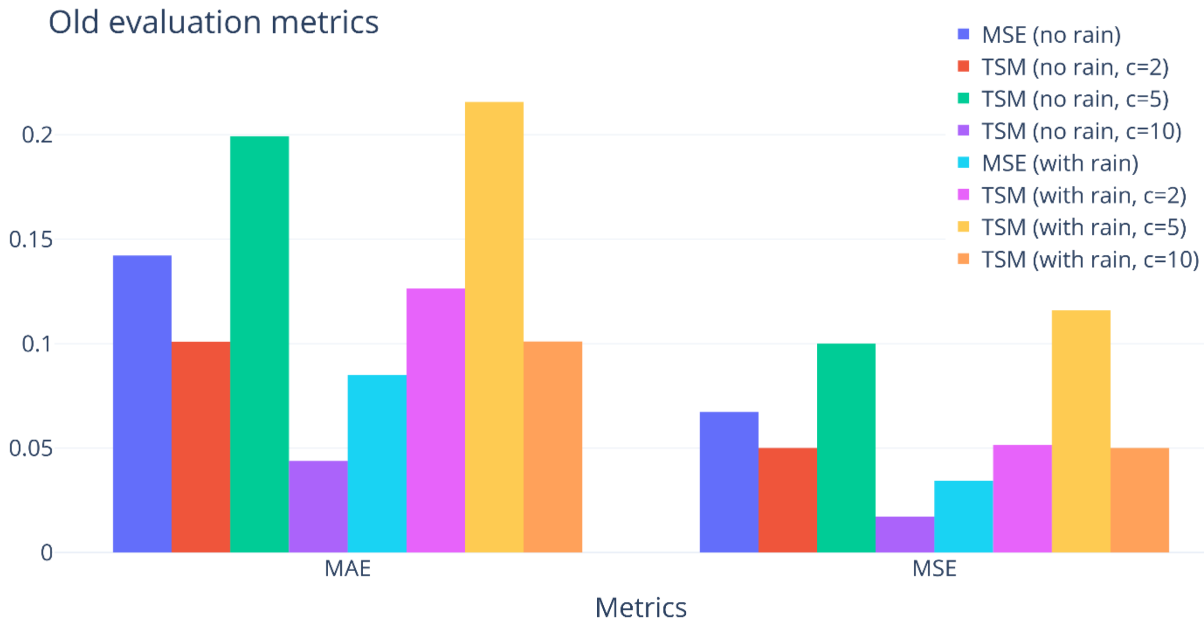

Figure 37: comparison of the eight algorithms tested. As in the previous sections, the metrics displayed here are MAE and MSE. Better performance corresponds to lower values of these metrics.

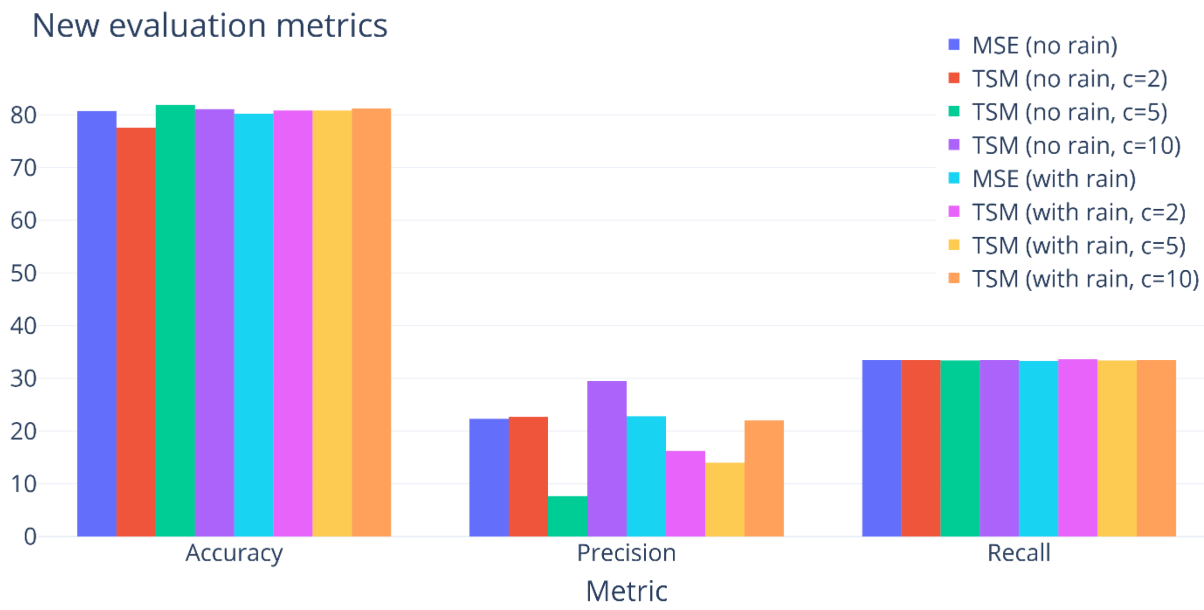

Figure 38: comparison of the eight algorithms tested. The metrics developed in Section "4.3.2. Metrics for landslide forecasting and tree species recognition" are used here. Better performance corresponds to higher values of these metrics.

From Figure 37, we can see that TSM with $c = 10$ appears to be the best configuration, with and without rainfall data. From this image we would also get the impression that, besides being the best relative to the other versions, this model is also performing very well in absolute terms (low average errors on the predictions). If we examine the second image (Figure 38), however, we see different results. Accuracy is a global metric, and it is in accord with MAE and MSE. Most targets to predict are very similar to their corresponding last observed displacements, and most targets to predict fall into the category of low displacements. Most models instead of becoming good general predictors, learn that they can get good results by "predicting" that the new displacement will be roughly the same as the last observed one. This causes them to get good evaluations from MAE and MSE. For a similar reason it also gives high accuracy: the majority of "last observations" fall within the "low displacements" class, so most models' predictions also are in this category. Since also most targets are in this class, on average the models get the right answer.

Precision and recall help in distinguishing between bad and good predictors because they are calculated on a per-class basis and are not influenced by the numerosity of the classes. So, from Figure 38, thanks to precision and recall, we can see that, while TSM with $c = 10$ (with and without rain) are still the best models, in absolute terms they are not performing very well.

This is even more evident if we examine the confusion matrix (Figure 39) from which precision and recall are computed.

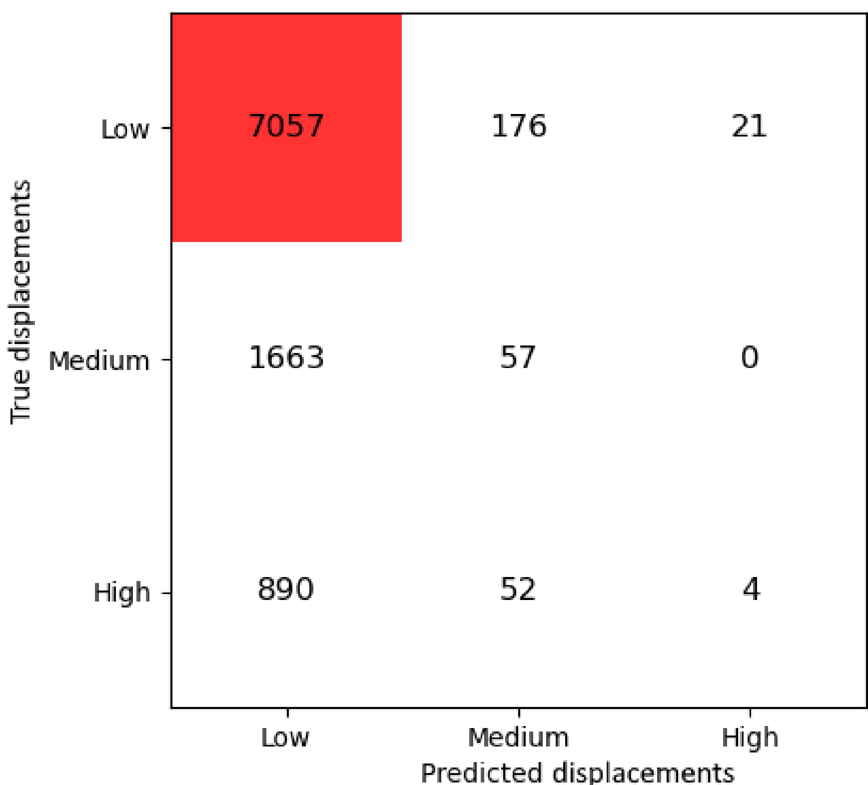

Figure 39: confusion matrix of one of the five experiments that were conducted to calculate the average performance of TSM (no rain, c=10). The red cell is the intersection between the main diagonal (correct predictions) and the first column (this algorithm biased preference).

The predictions are distributed randomly, mostly on low displacements (left column) more than according to ground truth (main diagonal). The top-left cell (in red), however, is part of both the leftmost column and the main diagonal, and it also contains most of the total targets. So, if we just calculate the proportion of correct predictions (the sum of the cells on the main diagonal) with respect to the total (the sum of all the cells), it is a high number. If, on the other hand, we examine class by class, the true high displacements (last row) are mostly predicted as low and medium displacements (bottom-left and bottom-middle cells). Only 4 of the $946 = 890 + 52 + 4$ high displacements are correctly recognized. And the same happens with medium displacements.

5.1.5. Neural Network's architecture expansion

The last improvement to the software allows to specify and modify the structure of the Feed-Froward NN that has been used up to now. More importantly, a new architecture is also added, which can be used in place of the FFNN, a 1D Convolutional NN. The complete explanation and implementation details are in the corresponding algorithm-development Section "4.1.6. Neural Network's architecture expansion".

From this version, only the custom metrics will be used, since in the previous section we have shown that MAE and MSE are not a good representation of the models' performance.

Two experiments are conducted, one with the newly added 1D CNN, one with a FFNN deeper than what was used in the previous sections (because it was fixed at 3 dense layers). The hyperparameters are in Table 23 below.

Table 23: hyperparameters used for the two experiments.

Hyperparameters	**Deeper FFNN**	**1D CNN**
Epochs	20	10
ndi	12	20
ndo	1	
Batch size	64	
Loss function	TSM	
Max loss coefficient (*c*)	10	
Learning Rate	$5 * 10^{-5}$	
Optimizer	Adam	
NN type	FFNN	1D CNN
Hidden units	2048, 2048, 512, 128, 32	256, 256, 128, 64
Shift thresholds	0.70, 0.90	
Data split	80%, 10%, 10%	
Use rainfall data	True	
rds	3	

The new hyperparameter (in bold) "NN type" controls whether we use the newly introduced 1D CNN or the FFNN.

Note that the number of days in input has to be increased for the CNN, specifically *ndi* has to grow proportionally to the number of Convolutional and pooling layers. This is for a technical reason: each 1D Convolutional layer shortens the input vector by 2, and even more if a pooling operation is performed afterwards (which is common and useful).

The FFNN can be fully described by the "Hidden units" hyperparameter coupled with the description given in the "4.1.2. Original algorithm" Section. The CNN also uses this hyperparameter to define the number and dimension of the decision layers. Those are one or more dense layers placed after the convolutional layers, basically a small FFNN inside the CNN

(which is why it can be described with the same hyperparameter). To fully describe a CNN, however, there is need for many more hyperparameters, which are listed in Table 24 below.

Table 24: additional hyperparameters specific to the CNN.

1D CNN parameters		**values**
Number of convolutional layers		3
Pooling operation		*MaxPool1d*
Convolutional parameters	Kernel size	3
	Stride	1
	Padding	0
Pooling parameters	Kernel size	3
	Stride	1
	Padding	0

For the meaning and explanation of those parameters see the "4.1.6. Neural Network's architecture expansion" Section.

The experiments' results are represented in Figure 40.

Figure 40: performance of the models obtained with the architecture expansion modification. FFNN is the Feed-Froward Neural Network used in all previous experiments, but now with more dense layers, 1D CNN is the new architecture introduced: one-dimensional Convolutional Neural Network.

Accuracy is high, but Precision and Recall are still quite low, meaning that the algorithms have trouble predicting rare events (medium and high displacements).

5.1.6. Full dataset available (all four sites)

This section does not have a corresponding one in "4. Algorithms development" because it didn't involve significand code modifications, the change is that, up to the last version, only data from site A was used. From this moment forwards all four sites described in "3.1.1. Study sites" are available, which increases observations from ~100,000 to ~250,000 (Table 1).

This significantly increases available training data, but since no modifications have been made to the algorithm, the experiments conducted will be the same as the previous section, except for a few hyperparameter adjustments. The pooling operation of the CNN before was "max pooling 1D" (see Table 24), while now it is set to "null" (Table 26), which means no pooling layer will be inserted in the NN and the corresponding operations will be skipped. This, in turn, allows us to use the same *ndi* for both models. The new hyperparameters are listed in Table 25 and Table 26.

Table 25: hyperparameters used for the two experiments.

Hyperparameters	**Deeper FFNN**	**1D CNN**
Epochs	10	
ndi	12	
ndo	1	
Batch size	64	
Loss function	TSM	
Max loss coefficient (*c*)	10	
Learning Rate	$5 * 10^{-5}$	
Optimizer	Adam	
NN type	FFNN	1D CNN
Hidden units	2048, 2048, 512, 128, 32	256, 256, 128, 64
Shift thresholds	0.70, 0.90	
Data split	80%, 10%, 10%	
Use rainfall data	True	
rds	3	

Table 26: additional hyperparameters specific to the CNN. Since "Pooling operation" is null, the pooling parameters are not used in this configuration, so they are marked by gray background. This configuration will be used for most future experiments that for simplicity will refer to this table instead of repeating it.

<table>
<tr><th colspan="2">1D CNN parameters</th><th>values</th></tr>
<tr><td colspan="2">Number of convolutional layers</td><td>3</td></tr>
<tr><td colspan="2">Pooling operation</td><td>null</td></tr>
<tr><td rowspan="3">Convolutional parameters</td><td>Kernel size</td><td>3</td></tr>
<tr><td>Stride</td><td>1</td></tr>
<tr><td>Padding</td><td>0</td></tr>
<tr><td rowspan="3">Pooling parameters</td><td>Kernel size</td><td>-</td></tr>
<tr><td>Stride</td><td>-</td></tr>
<tr><td>Padding</td><td>-</td></tr>
</table>

Note that in general, having more training data is better for NNs, and also introducing multiple sources of information help reduce overfitting. In this case, however, it also makes the problem harder to solve: with only site A, training and test data all exhibit very similar characteristics, but with the introduction of three more datasets this is not true anymore.

The results are shown in Figure 41.

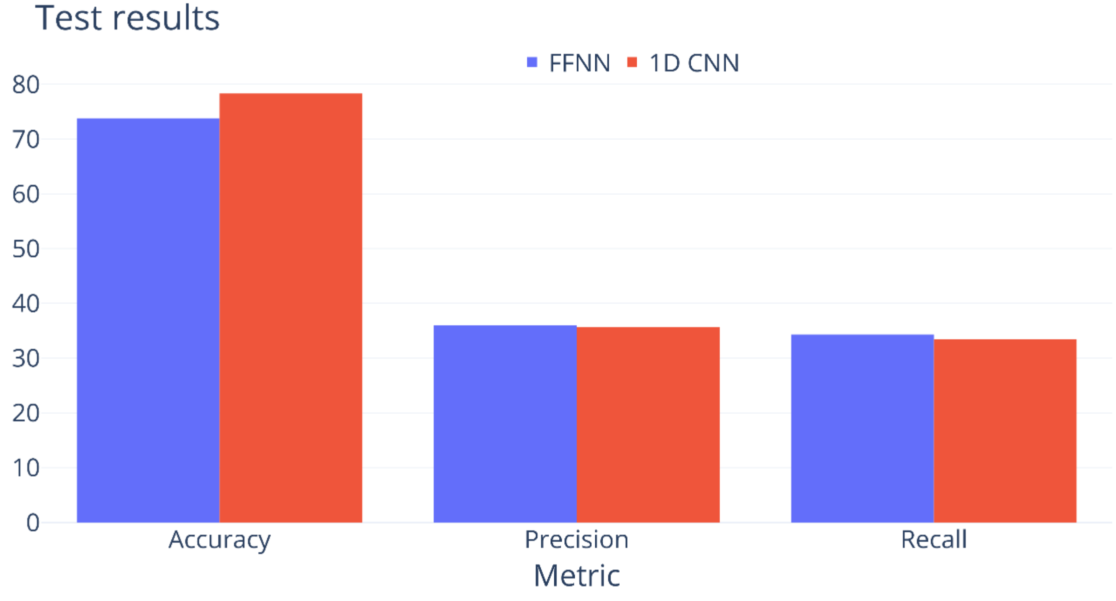

Figure 41: performance of the two models described in the previous section, trained using data from all four sites.

5.1.7. Data balancing and data augmentation

This addition to the code introduces two new hyperparameters highlighted with bold text in the table below (Table 27). For the CNN-specific parameters refer to Table 26 from the previous section.

Table 27: hyperparameters used for the three experiments with data balancing and data augmentation.

Hyperparameters	**Test 1**	**Test 2**	**Test 3**
Epochs	10		
ndi	12		
ndo	1		
Batch size	64		
Loss function	TSM		
Max loss coefficient (*c*)	10		
Learning Rate	$5 * 10^{-5}$		
Optimizer	Adam		
NN type	1D CNN		
Hidden units	256, 256, 128, 64		
Shift thresholds	0.60, 0.85		
Data split	80%, 10%, 10%		
Use rainfall data	True		
rds	3		
Data balancing	False	True	True
Augmentation proportion	1	1	5

The precise effect of the new hyperparameters is explained in "3.1.4. Data preprocessing" and "4.1.7. Data balancing and data augmentation" Sections. In summary data balancing controls whether the number of elements of each class is leveled out by adding new observations to the smaller classes until all of them have the same number of elements of the largest class. Augmentation proportion can only be specified if data balancing is also applied, and both operations are performed only on the training dataset. Augmentation proportion is a number ≥ 1 that tells the algorithm to generate new data until $|new\ dataset| = |balanced\ dataset| \times augmentation\ proportion$. Note that on the right side of the equation there is the balanced dataset, not the original dataset, which already has more elements than the training dataset.

For example, if we examine the last column, the original dataset has 247,523 observations. From this, we can extract the training dataset, which contains 80% of the original observations: $|training\ dataset| = 247{,}523 \times 0.8 = 198{,}018$. With the shift thresholds used (0.60, 0.85) the original three classes will have respectively:

1. low displacements $ld = 198{,}018 \times 0.6 = 118{,}811$ observations
2. medium displacements $md = 198{,}018 \times (0.85 - 0.6) = 49{,}504$ observations
3. high displacements $hd = 198{,}018 \times (1 - 0.85) = 29{,}702$ Observations.

After applying data balancing all three classes will have the same number of elements as the biggest one (low displacements), which means $ld = md = hd = \max(ld, md, hd) = 118{,}811$.

This increases the balanced dataset's observations to $118{,}811 \times 3 = 356{,}433 \geq 198{,}018$. If we also add data augmentation according to the rightmost column of Table 27 ("Test 3"), the number of elements of each class will increase by a factor of 5, $118{,}811 \times 5 = 594{,}055$, which brings the size of the augmented dataset to $594{,}055 \times 3 = 1{,}782{,}165$.

The comparison between the three models tested is summarized by Figure 42.

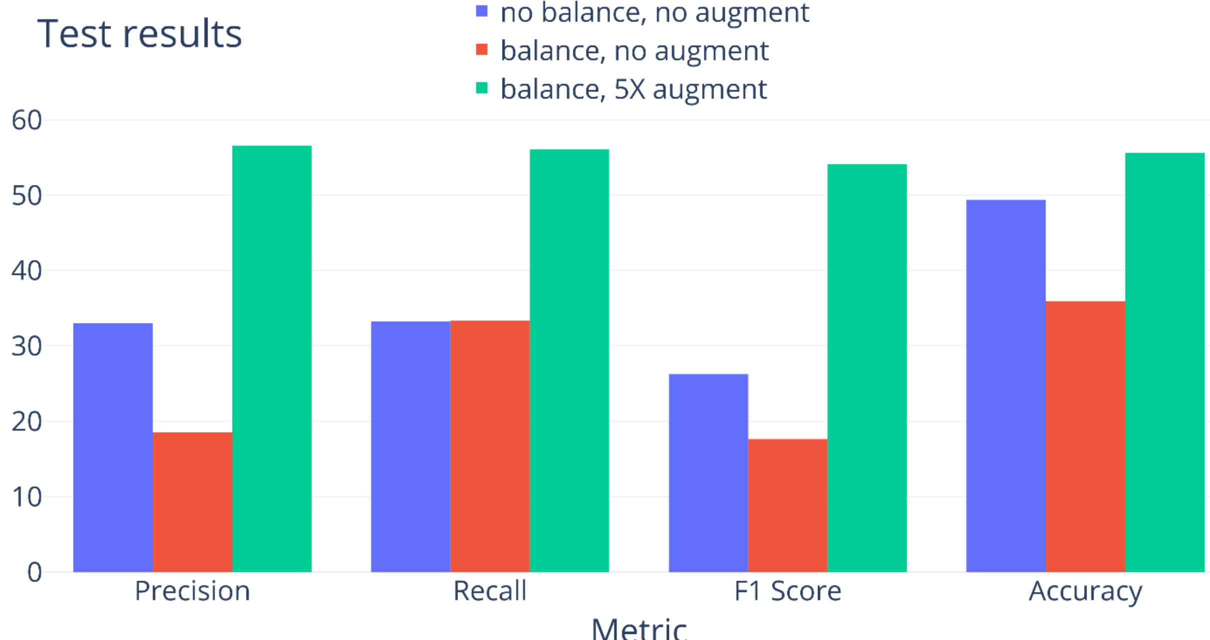

Figure 42: three models are tested, with increasing degree of data balancing and augmentation. Now F1 Score metric is also implemented and shown in the results.

Only applying data balancing does not appear to produce good results while adding both balancing and augmentation gives better results than the other two tests.

5.1.8. Condensation and reduction of input features

For this version only one experiment with the new code is made, and it is compared with the previous version. Both use the configuration that had the best performance in the earlier test (balanced dataset, 5X data augmentation). For the complete set of hyperparameters used, see Table 28.

Table 28: hyperparameters for the feature reduction experiments.

Hyperparameters	**Previous version**	**Features reduction**
Epochs	10	
ndi	12	
ndo	1	
Batch size	64	
Loss function	TSM	
Max loss coefficient (c)	10	5
Learning Rate	$5 * 10^{-5}$	10^{-5}
Optimizer	Adam	
NN type	1D CNN	
Hidden units	256, 256, 128, 64	
Shift thresholds	0.60, 0.85	
Data split	80%, 10%, 10%	
Use rainfall data	True	
rds	3	
Data balancing	True	
Augmentation proportion	5	

CNN hyperparameters are the same as Table 26. The results are shown below (Figure 43).

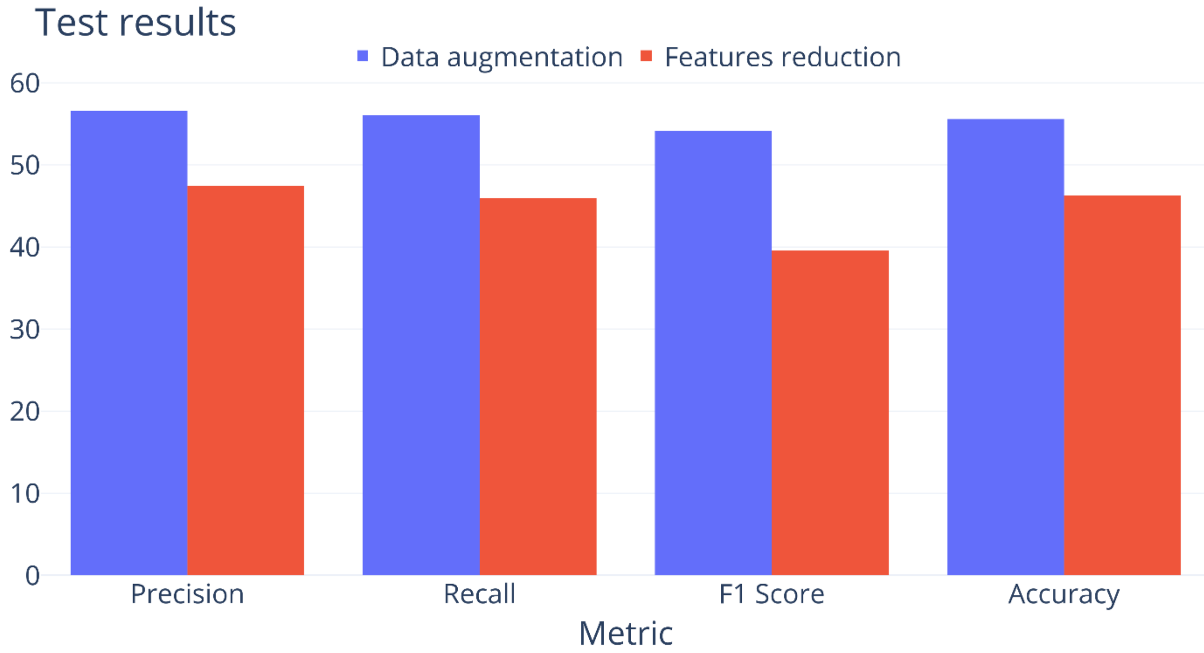

Figure 43: best model from the previous section (balanced dataset, 5X data augmentation) compared with new version ("Features reduction").

This new version seems to be a downgrade from the previous best model, but it is faster to train, requires less data and, most importantly, it only requires displacements and precipitations to operate.

5.1.9. Switch from regression to classification

With this change, the algorithms do not try to predict the exact future displacements, only their category. This means the output is no longer a real number, but an integer between 1 and *n*, with *n* = number of classes.

This version builds on top of the previous one, and currently the best model is the one from "5.1.7. Data balancing and data augmentation", so all three NNs will be compared. The hyperparameters are listed in Table 29 while the experiment's outcome is shown in Figure 44.

Table 29: hyperparameters for the classification experiments.

Hyperparameters	Data augmentation	Features reduction	Classification
Epochs	10		
ndi	12	12	14
ndo	1		
Batch size	64		
Loss function	TSM	TSM	Cross Entropy

Max loss coefficient (c)	10	5	-
Learning Rate	$5 * 10^{-5}$	10^{-5}	10^{-5}
Optimizer	Adam		
NN type	1D CNN		
Hidden units	256, 256, 128, 64		
Shift thresholds	0.60, 0.85		
Data split	80%, 10%, 10%		
rds	3		
Data balancing	True		
Augmentation proportion	5		

Note that "Use rainfall data" hyperparameter has been removed: from this version precipitations are automatically included if present in the training dataset.

Another change is that with categorial data the loss function is no more the custom TSM, with the associated "Max loss coefficient (c)" hyperparameter, but the standard "Cross Entropy", as explained in "4.1.9. Switch from regression to classification".

Figure 44: performance of the new model that predicts the class of future displacements, compared to the best models of the previous two sections.

The classification model gives only a coarse prediction (the velocity category expected) instead of the precise displacement velocity. Thanks to this simplification, however, it is able reach a significantly higher performance.

For example, Figure 45 is a confusion matrix that represent the result of one of the five NNs trained and tested for the "Classification" version in the previous plot (Figure 44). From this matrix we can see that most of the real high displacements (last row) were correctly predicted: $\frac{59{,}756}{4{,}051+16{,}836+59{,}756} = 74.1\%$. At the same time, the false alarms (top two cells of the right-most column) are few: $\frac{3{,}718+1{,}139}{3{,}718+1{,}139+59{,}756} = 7.5\%$.

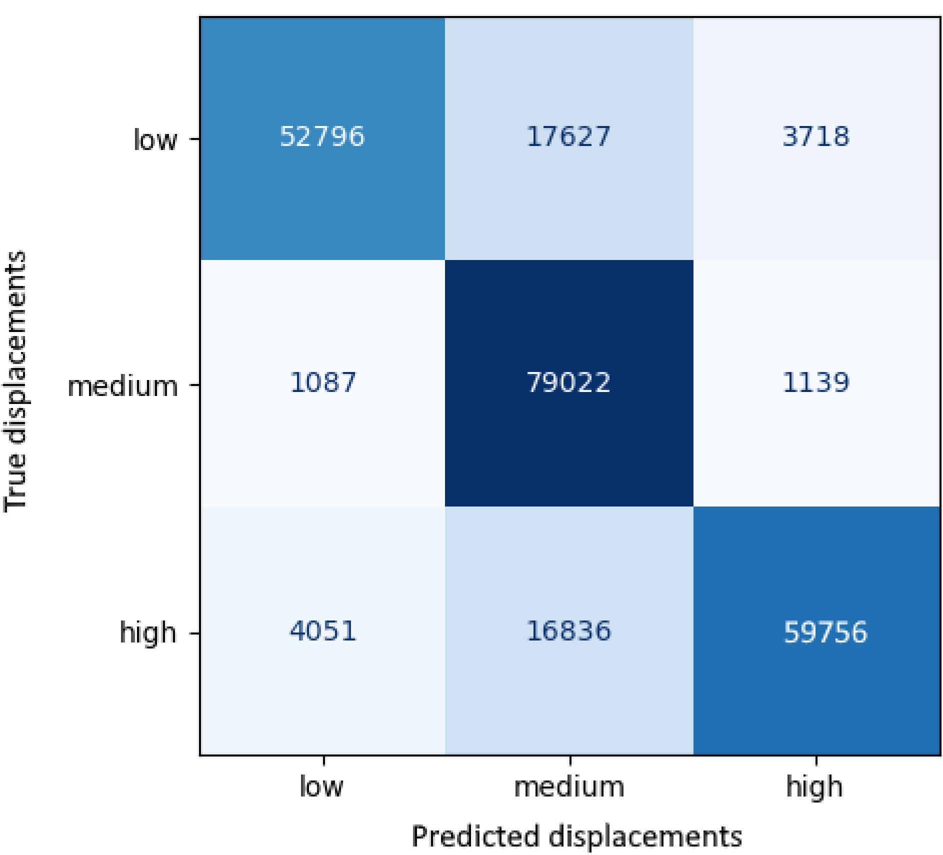

Figure 45: confusion matrix of the test results for one of the Classification NNs trained and tested for this experiment.

5.1.10. Conversion of time series to images

This version introduces the most drastic change tried in the course of this study: inputs are no longer time series, but images that encode the same information. Moreover, the NN employed are big, complicated, pretrained models.

Their hyperparameter configuration is in the following Table 30.

Table 30: hyperparameters for the two algorithms described in the corresponding development section ("4.1.10. Conversion of time series to images").

Hyperparameters	**Plain graph**	**GAF**
Epochs	10	
ndi	12	
ndo	1	
Batch size	64	
Loss function	MSE	
Learning Rate	5×10^{-5}	
Optimizer	Adam	
NN type	Swin small	
Shift thresholds	0.60, 0.85	
Data split	80%, 10%, 10%	
rds	3	
Data balancing	True	
Augmentation proportion	1	

Now the algorithms do not use custom NNs but pretrained models, so the hyperparameter "Hidden units" is not necessary anymore; the same holds for the "1D CNN parameters" (Table 24 and Table 26).

The main difference between the two algorithms (which is not captured by the hyperparameters) is the method used to represent the times series data as images:

- **"Plain graph"** converts observations into images by simply drawing a plot of the time series.
- **"GAF"** converts observations into images using the GAF method.

The performance of the new models are displayed in Figure 46.

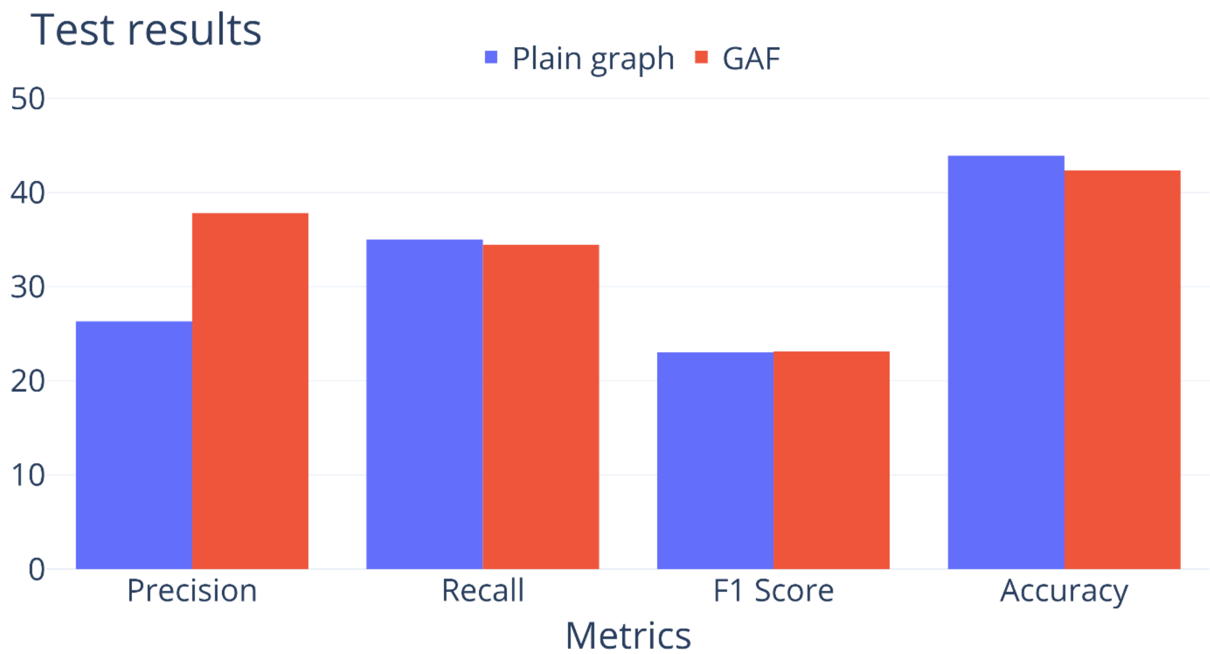

Figure 46: comparison between "plain graph" model and GAF model. Both use images as input in place of time series.

The GAF model appears to have a significantly higher precision, but slightly lower recall and overall accuracy. This means that it gives significantly fewer false alarms, at the price of rarely missing a real alarm. This is with respect to the plain graph model; in absolute terms both have quite low performances.

5.1.11. Relationship between data quantity and model performance

This section has already been published as part of the conference article “Improving Geotechnical Monitoring System Performances by Integrating Advanced Technologies for Data Acquisition and Elaboration” (Conciatori, Valletta, et al., 2024).

As previously mentioned, ML-based algorithms depend on the availability of previously collected monitoring data to train neural networks, enhancing their capacity to produce reliable results. It is especially critical to work with large datasets during the training phase, as they provide a diverse range of examples and variations, allowing the model to capture a broader representation of underlying patterns and relationships. In this context, the ability of automated systems to achieve high sampling frequencies plays a central role in gathering substantial amounts of data within a relatively short time.

The following example emphasizes the significance of these principles, demonstrating the evolution of neural network reliability when trained on datasets of varying sizes. The case study

focuses on the monitoring of a retaining wall protecting a road leading to a tunnel in the French Alps. Two vertical array automatic inclinometers, incorporating both MEMS sensors and piezometers, were installed on-site to gather information about structural displacement and water level variations over time (Segalini et al., 2019). The maximum available dataset for this analysis consisted of 21,621 records collected throughout the monitoring period. For training, 70% of the total dataset, amounting to 15,134 records, was used. Four additional scenarios were defined for comparison, where the number of available records was halved iteratively, resulting in five datasets of sizes ranging from 800 to 15,000 observations.

Performance evaluation for each training process was based on three metrics: the loss function, Mean Squared Error (MSE), and Mean Absolute Error (MAE). For each dataset 10 models were independently trained, and the average values from the two best-performing trials were considered. Table 31 summarizes the dataset sizes and the corresponding performance outcomes.

Table 31: data quantity and corresponding performance.

Dataset ID	Number of training observations	Loss function	MSE	MAE
1	15,134	0.00143	0.00075	0.00895
2	7,567	0.00195	0.00102	0.01172
3	4,305	0.01495	0.00794	0.06855
4	1,782	0.02107	0.01133	0.08128
5	808	0.03574	0.02079	0.07634

A sharp increase in the loss function is observed as the dataset size decreases, with Dataset 1 and Dataset 5 showing a difference of one order of magnitude. As shown in Figure 47a, the loss function follows an exponential trend with a correlation coefficient of 0.89. Since the training process aims to minimize the loss function, the limited data availability clearly hinders the process, resulting in a neural network with lower reliability and precision.

A similar pattern is evident in the analysis of the MSE and MAE, as displayed in Figure 47b. The reduction in available data significantly affects the neural network's ability to make accurate predictions. MSE values between Datasets 1 and 5 differ by two orders of magnitude, mirroring the loss function trend. For MAE, the most notable difference is between Datasets 2 and 3, with values of 0.01172 and 0.06885, respectively, as shown in Table 31.

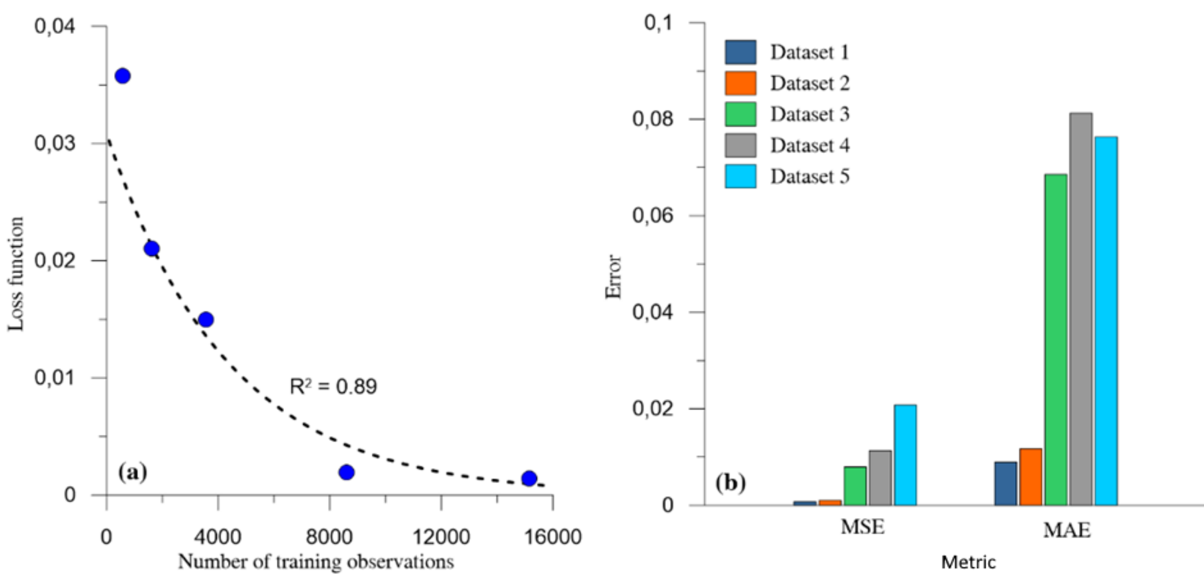

Figure 47: (a) Loss function variation with respect to the number of training observations; (b) Mean Squared Error (MSE) and Mean Absolute Error (MAE) for each dataset size.

5.1.12. Runtime considerations

The tests for the landslide forecasting algorithms speed are conducted on the MS-7E07, a high-performance computer. The most relevant details are listed in Table 32 below.

Table 32: hardware specifications of the computer used for execution-time measurements.

Name	**Processor**	**Graphic card**	**RAM**	**Operating System**
MS-7E07	Intel Core i9-13900KF 3.00 GHz	NVIDIA GeForce RTX 4090	128 GB	Windows 11 Home

The MS-7E07 is very powerful compared to normal personal computers, but it still is a general-purpose single pc, not a specialized machine for ML tasks. The second table (Table 33) shows the time required for the tasks involved in landslide forecasting. The times depend on the hyperparameters selected and other details, so below is also provided all the relevant information:

- Data: all four sites
- Features: only displacements
- Data balancing: True
- Data augmentation: 1 (which means no data augmentation)

- Epochs: 10
- Batch size: 64

Table 33: execution time required by each task using the computer described above.

Task	Time
Data preprocessing and balancing	13 minutes, 28 seconds
Conversion of time series to plain graphs	13 hours, 16 minutes
Conversion of time series to GAF images	17 hours, 37 minutes
Training time (no images)	3 minutes, 38 seconds
Training time (with plain graphs)	5 hours, 17 minutes
Training time (with GAF images)	6 hours, 44 minutes
Inference time (100 time series observations)	2.1 milliseconds
Inference time (100 plain graphs)	0.23 seconds
Inference time (100 GAF images)	0.73 seconds

The image conversion in particular can be very slow and require a lot of memory while active. Note, however, that ML algorithms in most cases are trained once and only used afterwards, so that only the very fast time of the last column is relevant.

Even if the objective is to run multiple trainings to explore the effects of various hyperparameters or to search for the best model, preprocessed data and images (first two columns) can be generated only once and reused. So even to run multiple experiments, it is not necessary to redo the first two operations, only the training (third column).

The inference time is the time that a trained model requires to give a prediction when presented with an input. This interval is very small, so trained models can be exported to low performance devices and still provide real-time responsivity. This is also the reason to calculate it on 100 inputs rather than 1, it is more accurate and less influenced by other processes, like data loading or the time measurement itself.

5.1.13. Overall comparison

To predict the future landslide displacements many versions of the algorithm were developed and tested. In this section all the results are summarized in a single plot (Figure 48), so that all versions can be directly compared. The plot uses accuracy, precision, and recall, which means that the algorithm of Section “5.1.2 Vertical Array as input” cannot be represented. The same

should be true for the original algorithm, but it has been evaluated later as comparison for other versions (Test 1 in Section “5.1.4 Custom loss function”), so there are appropriate values to display. F1 is not used because it was introduced later, thus not all evaluations have it.

The “Categorial target” version on the right is separated from the others by a dashed line. This is because it basically solves a different (and easier) problem. It just tries to predict whether a displacement will be low, medium, or high, instead of its physical magnitude. So, even thought is uses the same metrics, it is not directly comparable to the other versions.

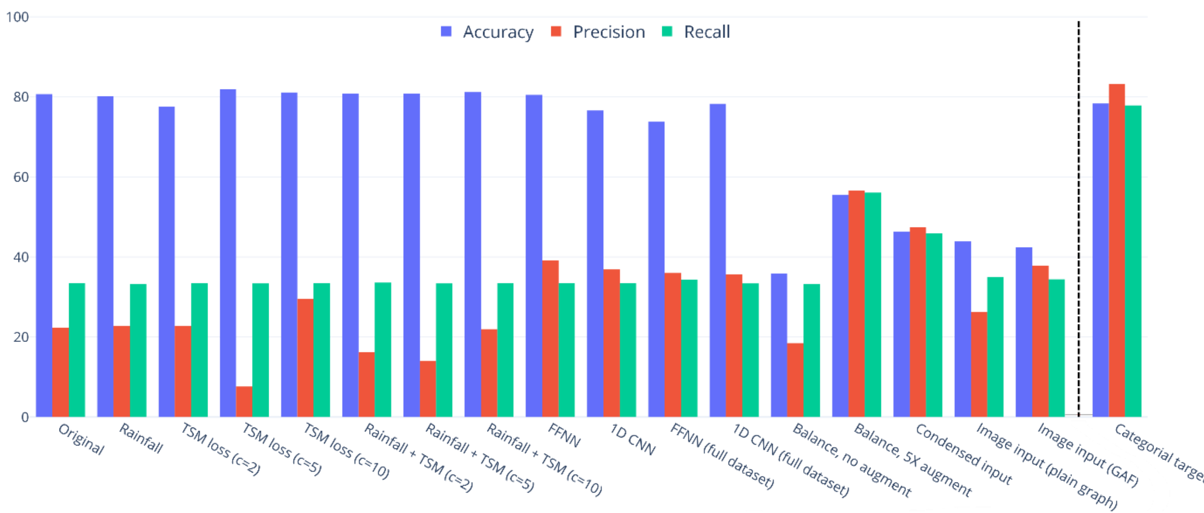

Figure 48: summary of all tests conducted. This figure allows to compare any version with any other.

Many of the earlier versions have high accuracy but low precision and recall, which we were able to notice thanks to the custom metrics. As discussed previously, this means the algorithm is good for common cases but struggles with unusual inputs, and for this particular application it is a bad combination since the “unusual inputs” are exactly the dangerous events we want to identify. The efforts to improve the algorithm resulted in the version that use data balancing and augmentation, and the “condensed input” version, both of which significantly raise precision and recall.

5.2. Tree species classification

5.2.1. Original algorithm

The four pretrained NN tested obtained very good results in patches recognition in the majority of hyperparameter combinations tested, indicating that the chosen algorithms were fitting for the problem.

The first part of the tests consisted of finding the best set of hyperparameters for each architecture independently. For this purpose, a specific script was created: it takes as input a model architecture and ranges of values for the hyperparameters and it automatically trains and tests models with all the possible combinations of values, as explained in Section "4.5. Hyperparameter space exploration". After two phases of this process (coarse and fine-grained) the set of hyperparameters which produced the best results were identified.

With this information five models were trained for each architecture, to finally identify the most appropriate NN configuration (Figure 49). This was necessary because each training has some elements of randomness, which means that the average result of multiple trainings is more reliable.

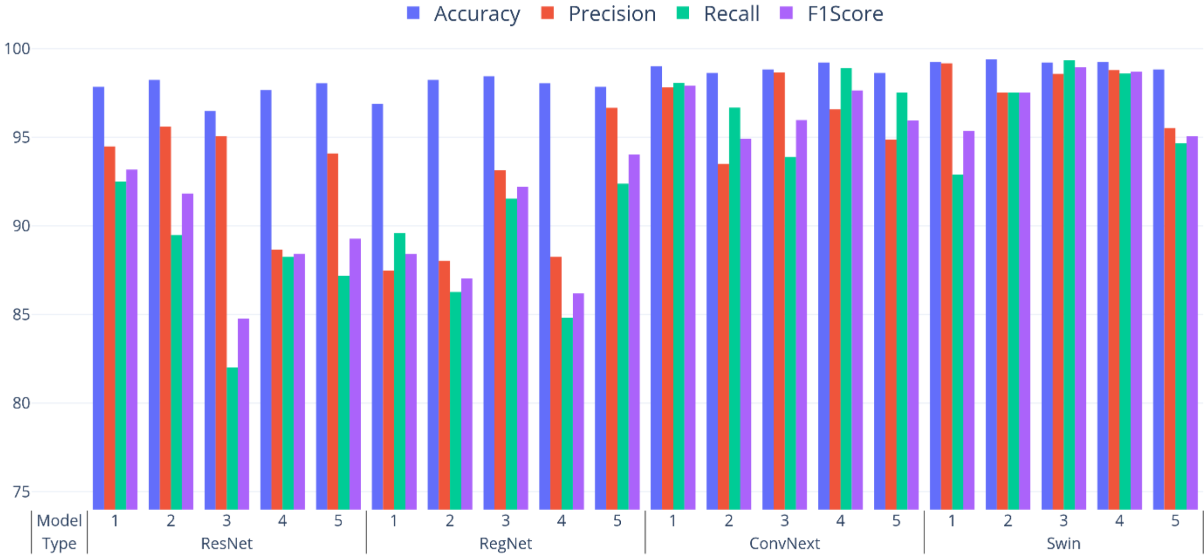

Figure 49: models' comparison – detailed.

Figure 50 presents a more compact representation of those results, from which it is apparent that Swin transformer was the best model, followed closely by ConvNext.

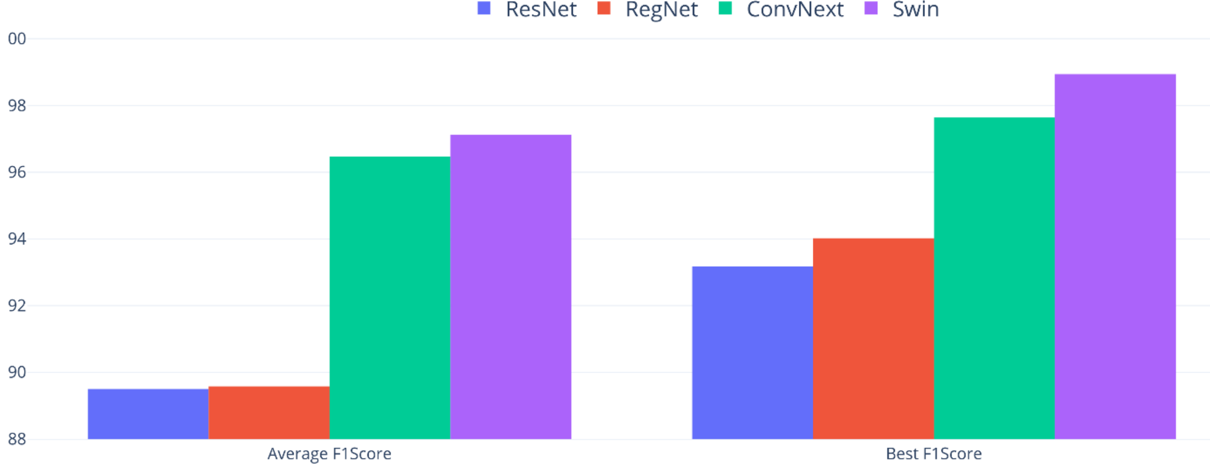

Figure 50: models' comparison - summary.

We decided to consider not only the average results but also the top results because our models trained relatively fast. Since for practical usage NNs are trained once and queried many times, it is worth doing more than one training run and choosing the best model obtained.

The conclusion of all these steps is that the best model is Swin transformer (tiny version) with the following hyperparameters (Table 34):

Table 34: best hyperparameters found with automatic hyperparameter space exploration.

Hyperparameters	**Original algorithm**
epochs	20
Optimizer	Adam
Learning Rate	10^{-5}
Weight Decay	10^{-3}
Data Augmentation	13
Data Balancing	No
Frozen Layers	No

Up to this point, only training and validation data were used, and all the results are computed using the validation dataset. This is to ensure that test data had no influence on the models, even by indirectly deciding which architecture is selected or which model is the best one.

The best model performance is summarized by the four metrics in Table 35, while Figure 51 shows the confusion matrix with complete, class-by-class, information of correct predictions and errors.

Table 35: metric evaluation of the best model found.

Accuracy	Precision	Recall	F1 Score
99.22%	98.58%	99.35%	98.94%

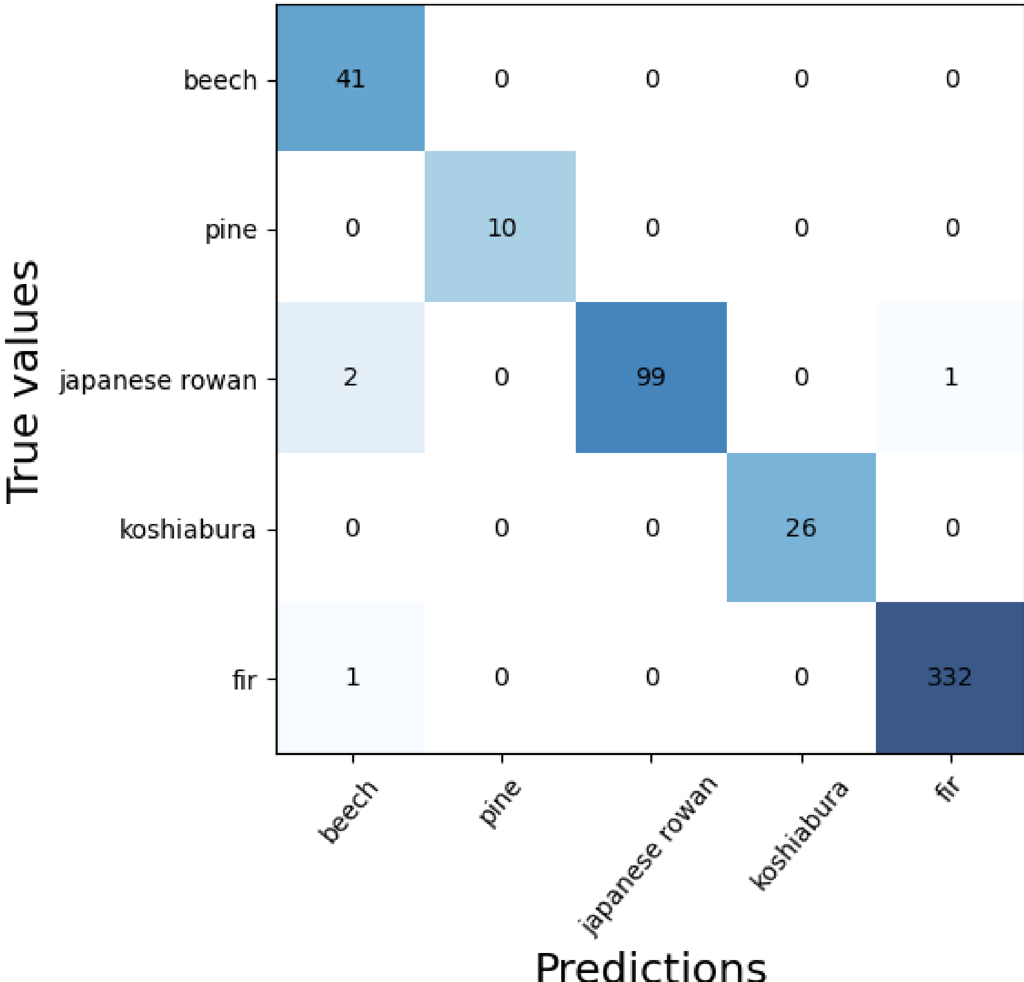

Figure 51: confusion matrix of the best model. The cells on the main diagonal represent the correct predictions.

It is important to note that the problem faced by our models is easier than a real case scenario because the training, validation, and test set have all the same species distribution. Also, all the patches come from the same environment, were taken in the same period of the year, and have the same lighting conditions. Nonetheless the results obtained by the best model are impressive: from the confusion matrix we can see that it only made 4 errors (0.78% of the test set), 3 of which are different plants misclassified as Beech.

5.2.2. Relationship between data quantity and model performance

As done for the landslide forecasting algorithm, the relationship between availability of training data and quality of the resulting model was explored. The experiment consisted of training models with the same setup that produced the best model (see Table 34), progressively reducing the training data. For each step of Figure 52, three models were trained, and the metrics show the average obtained.

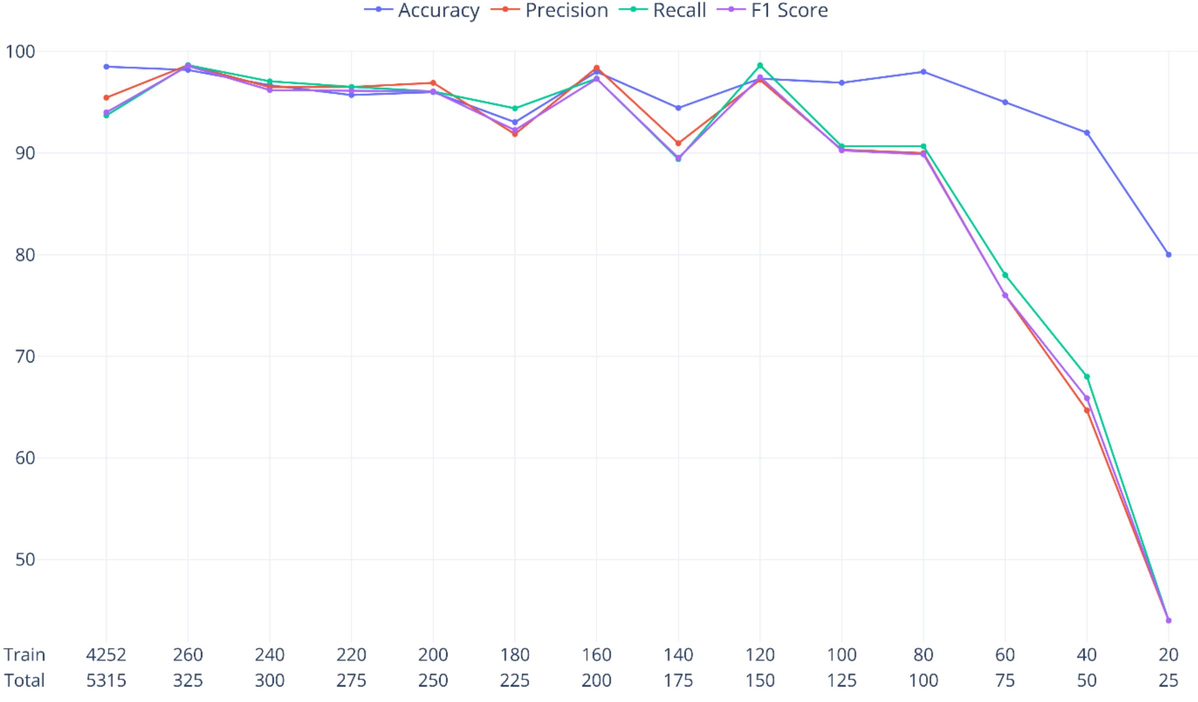

Figure 52: species classification results depending on training data quantity. The "Total" label indicates the number of patches that compose the current dataset, the "Train" label is number of patches effectively used for training.

Since the dataset presents a strong imbalance, the first reduction, from complete dataset (5,315 patches) to only 325 patches, was used to even out all the tree species. The smallest class is pine with 65 images, so every other class was reduced to the same dimension, which resulted in a dataset of $65 \times 5 = 325$ elements. Afterwards, 5 patches from each species are removed at each iteration, resulting in a reduction of 25 elements each time.

The two rows of x labels of Figure 52 show the total number of available patches for that step (lower row, "Total") and the subset of them that were used for training (upper row, "Train"), which is 80% of the total. The second consideration is that the best model hyperparameters include a data augmentation value of 13. For example, the point on the x axis identified by 100 "Total" patches has $100 \times 0.8 = 80$ training patches, which will become $80 \times 13 = 1{,}040$ after data augmentation.

Even though there are some fluctuations, the metrics remain near or above 90% from the start up to a 100 images dataset, while for smaller datasets there is a rapid decline in performance. This is a notable result for practical application and other studies because building a dataset with more than 5,000 elements can be prohibitive, but we show that good results can still be reached with a much smaller dataset (only 100 patches).

The point of this test is to demonstrate that, even with a much lower number of available patches, pretrained model are still able to become good classifier, that is why the previous test only used the images from the "Total" row for everything with the standard division of 80% training, 10% validation, and 10% test. With very small datasets, however, there is a concern that training and validation sets would not be enough for proper training and at the same time the test set would be too small to notice possible problems.

For this reason, a second test was conducted: the models trained for the previous experiment were assessed against a new patch collection created ad-hoc and not part of the original dataset. This also made the problem harder and more realistic because the new dataset differed in species distribution and other characteristics from the training data.

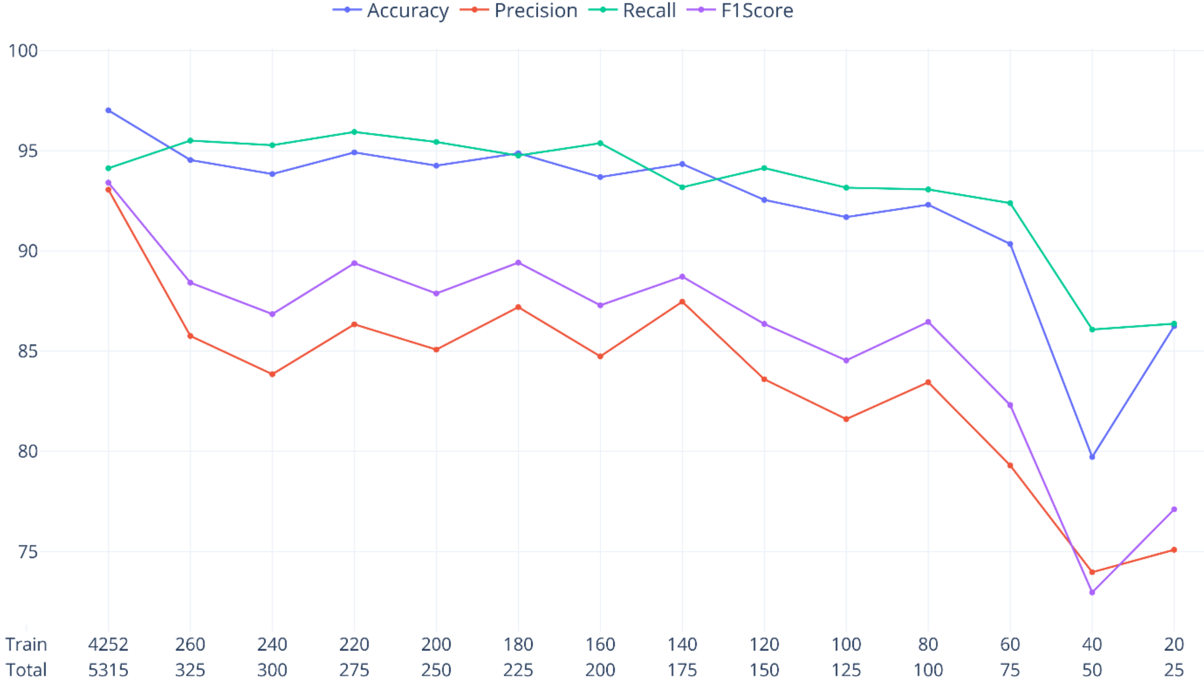

Figure 53: species classification results depending on training data quantity with a different test set specifically prepared. The "Total" label indicates the number of patches that compose the current dataset, the "Train" label is number of patches effectively used for training.

As expected, the results shown in Figure 53 are lower compared to Figure 52, but they exhibit the same general trend and the 100-patches dataset is the last before the performance starts

degrading. This confirms that the models are perfectly viable even with a dataset 50 times smaller than what we had available.

5.2.3. Runtime considerations

Runtime depends on the hardware used, for this reason the algorithm is executed on two quite different computers. Table 36 lists the specs of the two machines, while Table 37 shows the time required by the three major procedures involved.

Table 36: hardware specifications of the two computers used for execution-time measurements.

Name	Processor	Graphic card	RAM	Operating System
Asus Zenbook UX535LH	Intel Core i7-10870H 2.20GHz	NVIDIA GeForce GTX 1650 Max-Q	16 GB	Windows 11 Pro
MS-7E07	Intel Core i9-13900KF 3.00 GHz	NVIDIA GeForce RTX 4090	128 GB	Windows 11 Home

Table 37: execution time required by each task depending on the computer used.

Name	Data preprocessing and augmentation time (full dataset)	Training time (full dataset)	Inference time (100 images)
Asus Zenbook UX535LH	13 minutes	10 hours, 26 minutes	~2.12 seconds
MS-7E07	2 minutes, 15 seconds	43 minutes	~0.73 seconds

Table 37 contains the computation time needed to perform various tasks. The data processing column and training column are calculated using the best configuration, which means all the 5,315 patches of Table 10 and data augmentation set to 13 (the standard setting for the other experiments). The last column is the time required for the trained network to evaluate 100 patches and guess the species.

The Asus ZenBook has characteristics that are suited for ML applications, but except for that it is a normal laptop. Nonetheless it is able to complete a full training in less than 1 day, which is not very long, considering this process should be executed only one time.

The MS-7E07 has better components, and with the training times it sports it is possible to schedule multiple trainings in sequence or try many different hyperparameter configurations progressively adjusting them. While the second computer is superior, it is still far from datacenters or supercomputers, the combination of the code developed and data available does not require so much time and computation to complete.

Inference time for most NNs is very small even on slow hardware. For this reason, unless the algorithm is intended for the examination of billions of preexisting records, this phase is never a problem, regardless of the computer. This, in turn, allows users to train the algorithm on a fast machine and then distribute the trained NN on less powerful edge devices.

5.2.4. Orthomosaic input

One of the problems encountered during the development of the advanced version of the tree species recognition algorithm was the lack of suitable targets for ground truth comparison during the test phase. This is due to the fact that the training phase is the same as the original version, and thus uses the species labels of the patches as targets. The output, in contrast, is a whole orthomosaic, and to evaluate it ad-hoc metrics were created (see "4.3.3. Metrics for tree species recognition with orthomosaic input" Section). The problem with this approach is that the metrics need the ground truth corresponding to the model output, which in this case means the same input orthomosaic with each species highlighted with a different color. This type of data was not available, so it was not possible to apply the metrics to evaluate the algorithm's performance. The alternative is for an expert to manually check the results, which is less precise and objective, but nonetheless insightful.

6. Discussion

The landslide forecasting algorithm has gone through many different implementations to test many variations:

- Input and output content
- Input and output representation
- NN architecture

- Data balancing and augmentation
- Hyperparameter combinations

The primary reason for creating and trying all these algorithms was that they didn't produce very good results, even though it became clearer only after the introduction of the custom metrics. Consider that, since there are 3 classes (low, medium, and high displacements) even a completely random model would get 33% of the answers right.

The most notable exceptions are the algorithm that uses data balancing and augmentation, and the algorithm that switches from regression to classification. In the first case, this algorithm has access to ~15 times more data than its predecessors, in the second case, the problem to solve is easier. However, there is no guarantee that they are really generally able to predict the evolution of new landslides: it is possible they only overfitted the training data and the test data are similar enough to be handled reasonably well.

One of the reasons for the difficulty of forecasting events like landslides with this type of algorithm is that they behave like complex systems, in the formal sense (Ladyman et al., 2013; Tordesillas et al., 2018). So, landslides are subject to phenomena like non-linear evolution and butterfly effect (Chen et al., 2023; Ghys, 2015), which means that two almost identical initial states can lead to very different future displacements. Moreover, there is no guarantee that the available sensors can capture the small differences that give rise to different evolutions.

The algorithm for vegetation recognition is extremely accurate, but in the first version (presented in Section "4.2.2. Original algorithm") it is impractical to use for gaining additional information for landslide predictions. This is because plant images have to be extracted manually from the orthomosaic both for training and usage of the model, which can be very time consuming and requires a certain degree of expertise. The second version ("4.2.3. Orthomosaic input" Section) solves the problem, but it was not possible to meaningfully evaluate its performance for lack of appropriate expert-labeled data.

Since both algorithms are intended for practical applications, it is important that the requirements in terms of time and hardware are reasonable. This was the reason for including the runtime tests. The only long-time requirement is for the landslide forecasting algorithm when used with time series converted to images, but since the relative experiments didn't produce good results, it is not important. In general, depending on the available hardware, training any of the models developed is a matter of hours (or days if a search for the best hyperparameters is

conducted). The very small amount of time required to use trained models makes them viable on almost any device, and also quite cheap in terms of energy consumption.

7. Conclusions

During the course of the study, an algorithm for landslide forecasting and one for tree species recognition were developed. This includes a full set of functions for auxiliary but necessary operations:

- data processing
- hyperparameter space search
- custom evaluation metrics and loss function
- visualization of the results with graphs and 3D plots

The complete code for the tree species recognition and part of the algorithm for landslide forecasting are openly accessible on GitHub as part of their relative publications (Conciatori, Tran, et al., 2024; Conciatori, Valletta, et al., 2024). The full dataset of plant images and orthomosaics generated in collaboration with Yamagata University is freely available.

The landslide forecasting algorithm is not application-ready in terms of both accuracy and reliability, but two of the versions tested (see Sections "4.1.7. Data balancing and data augmentation" and "4.1.9. Switch from regression to classification") show promising results.

The tree species recognition algorithm works very well, and it can be used separately to identify plant species and to also calculate the biodiversity of a dataset. However, to use it in conjunction with the landslide prediction algorithm, the second version is required ("4.2.3. Orthomosaic input"), which we were not able to test.

7.1. Future developments

The tree species classification algorithm only needs suitable test data for the evaluation of the second version, which, in turn, would make its information available for the landslide forecasting algorithm.

Other interesting directions to predict the evolution of slopes and detect dangerous situations are

- Building a very large (possibly global) dataset of all measurements obtained from landslide sensors and consequent displacements. It has been shown in practice that

increasing both size of NNs and training datasets can result in large qualitative improvements. This would also include a significant effort to standardize and clean data.

- Build extremely accurate (but slow) physical simulations and use them to train ML models. This enables the models to approximate the simulations with arbitrary precision, with the advantage that they run much faster.
- Use different data sources and procedures, like satellite images to identify dangerous areas.

8. Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2016, March 14). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. arXiv.Org. https://arxiv.org/abs/1603.04467v2

Aguiar-Conraria, L., & Soares, M. J. (2014). The Continuous Wavelet Transform: Moving Beyond Uni- and Bivariate Analysis. *Journal of Economic Surveys*, *28*(2), 344–375. https://doi.org/10.1111/joes.12012

Ahmed, M., Seraj, R., & Islam, S. M. S. (2020). The k-means Algorithm: A Comprehensive Survey and Performance Evaluation. *Electronics*, *9*(8), Article 8. https://doi.org/10.3390/electronics9081295

Apicella, A., Donnarumma, F., Isgrò, F., & Prevete, R. (2021). A survey on modern trainable activation functions. *Neural Networks*, *138*, 14–32. https://doi.org/10.1016/j.neunet.2021.01.026

Asada, H., & Minagawa, T. (2023). Impact of Vegetation Differences on Shallow Landslides: A Case Study in Aso, Japan. *Water*, *15*(18), Article 18. https://doi.org/10.3390/w15183193

Asada, H., Minagawa, T., Koyama, A., & Ichiyanagi, H. (2020). Factor analysis of surface collapse on slopes caused by the July 2017 Northern Kyushu Heavy Rain. *Ecology and Civil Engineering*, *23*(1), 185–196. https://doi.org/10.3825/ece.23.185

Batista, G. E. A. P. A., Prati, R. C., & Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, *6*(1), 20–29. https://doi.org/10.1145/1007730.1007735

Bengio, S., & Bengio, Y. (2000). Taking on the curse of dimensionality in joint distributions using neural networks. *IEEE Transactions on Neural Networks*, *11*(3), 550–557. IEEE Transactions on Neural Networks. https://doi.org/10.1109/72.846725

Bengio, Y. (2012). Practical Recommendations for Gradient-Based Training of Deep Architectures. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural Networks: Tricks of the Trade: Second Edition* (pp. 437–478). Springer. https://doi.org/10.1007/978-3-642-35289-8_26

Berrar, D. (2019). Cross-Validation. *Encyclopedia of Bioinformatics and Computational Biology*, *1*(April), 542–545.

Bertolotti Matteo. (2022). *La frana di Agna: Modellazione numerica e confronto con i dati di monitoraggio* [Master Thesis]. Università degli studi di Parma. Scienze e Tecnologie Geologiche.

Borji, A. (2023). *Generated Faces in the Wild: Quantitative Comparison of Stable Diffusion, Midjourney and DALL-E 2* (No. arXiv:2210.00586). arXiv. https://doi.org/10.48550/arXiv.2210.00586

Chen, N., Tian, S., Wang, F., Shi, P., Liu, L., Xiao, M., Liu, E., Tang, W., Rahman, M., & Somos-Valenzuela, M. (2023). Multi-wing butterfly effects on catastrophic rockslides. *Geoscience Frontiers*, *14*(6), 101627. https://doi.org/10.1016/j.gsf.2023.101627

Chirico, G. B., Borga, M., Tarolli, P., Rigon, R., & Preti, F. (2013). Role of Vegetation on Slope Stability under Transient Unsaturated Conditions. *Procedia Environmental Sciences*, *19*, 932–941. https://doi.org/10.1016/j.proenv.2013.06.103

Church, K. W., Chen, Z., & Ma, Y. (2021). Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering*, *27*(6), 763–778. https://doi.org/10.1017/S1351324921000322

Claesen, M., & De Moor, B. (2015, February 7). *Hyperparameter Search in Machine Learning*. arXiv.Org. https://arxiv.org/abs/1502.02127v2

Conciatori, M., Tran, N. T. C., Diez, Y., Valletta, A., Segalini, A., & Lopez Caceres, M. L. (2024). Plant Species Classification and Biodiversity Estimation from UAV Images with Deep Learning. *Remote Sensing*, *16*(19), Article 19. https://doi.org/10.3390/rs16193654

Conciatori, M., Valletta, A., & Segalini, A. (2022). *Importance of multi-parameter approaches in the development of Machine Learning algorithms for landslide displacement forecasting*.

Conciatori, M., Valletta, A., & Segalini, A. (2024). Improving the quality evaluation process of machine learning algorithms applied to landslide time series analysis. *Computers & Geosciences*, *184*, 105531. https://doi.org/10.1016/j.cageo.2024.105531

Croxton, F. E., & Cowden, D. J. (1939). *Applied General Statistics*. Prentice-Hall, Incorporated.

Dashbold, B., Bryson, L. S., & Crawford, M. M. (2023). Landslide hazard and susceptibility maps derived from satellite and remote sensing data using limit equilibrium analysis and

machine learning model. *Natural Hazards*, *116*(1), 235–265. https://doi.org/10.1007/s11069-022-05671-7

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

Dietterich, T. G. (2000). Ensemble Methods in Machine Learning. *Multiple Classifier Systems*, 1–15. https://doi.org/10.1007/3-540-45014-9_1

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., … Zhao, Z. (2024). *The Llama 3 Herd of Models* (No. arXiv:2407.21783). arXiv. https://doi.org/10.48550/arXiv.2407.21783

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., & others. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *Kdd*, *96*(34), 226–231.

Ghys, É. (2015). The butterfly effect. *The Proceedings of the 12th International Congress on Mathematical Education: Intellectual and Attitudinal Challenges*, 19–39.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

GPT-4. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=GPT-4&oldid=1242777230

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., & Chen, T. (2018). Recent advances in convolutional neural networks. *Pattern Recognition*, *77*, 354–377. https://doi.org/10.1016/j.patcog.2017.10.013

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

He, K., Zhang, X., Ren, S., & Sun, J. (2015, December 10). *Deep Residual Learning for Image Recognition*. arXiv.Org. https://arxiv.org/abs/1512.03385v1

Hu, S., Jiao, J., García-Fayos, P., Kou, M., Chen, Y., & Wang, W. (2018). Telling a different story: Plant recolonization after landslides under a semi-arid climate. *Plant and Soil*, *426*(1), 163–178. https://doi.org/10.1007/s11104-018-3612-y

Huang, L., Qin, J., Zhou, Y., Zhu, F., Liu, L., & Shao, L. (2023). Normalization Techniques in Training DNNs: Methodology, Analysis and Application. *IEEE Transactions on Pattern Analysis and*

Machine Intelligence, *45*(8), 10173–10196. IEEE Transactions on Pattern Analysis and Machine Intelligence. https://doi.org/10.1109/TPAMI.2023.3250241

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated Machine Learning: Methods, Systems, Challenges*. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5

Imambi, S., Prakash, K. B., & Kanagachidambaresan, G. R. (2021). PyTorch. In K. B. Prakash & G. R. Kanagachidambaresan (Eds.), *Programming with TensorFlow: Solution for Edge Computing Applications* (pp. 87–104). Springer International Publishing. https://doi.org/10.1007/978-3-030-57077-4_10

Iverson, R. M. (2000). Landslide triggering by rain infiltration. *Water Resources Research*, *36*(7), 1897–1910. https://doi.org/10.1029/2000WR900090

Jadhav, A., M. Mostafa, S., Elmannai, H., & Karim, F. K. (2022). An Empirical Assessment of Performance of Data Balancing Techniques in Classification Task. *Applied Sciences*, *12*(8), Article 8. https://doi.org/10.3390/app12083928

Jia, W., Sun, M., Lian, J., & Hou, S. (2022). Feature dimensionality reduction: A review. *Complex & Intelligent Systems*, *8*(3), 2663–2693. https://doi.org/10.1007/s40747-021-00637-x

Jia, W., Wen, T., Li, D., Guo, W., Quan, Z., Wang, Y., Huang, D., & Hu, M. (2023). Landslide Displacement Prediction of Shuping Landslide Combining PSO and LSSVM Model. *Water*, *15*(4), Article 4. https://doi.org/10.3390/w15040612

Jost, L. (2006). Entropy and diversity. *Oikos*, *113*(2), 363–375. https://doi.org/10.1111/j.2006.0030-1299.14714.x

Kingma, D. P., & Ba, J. (2017). *Adam: A Method for Stochastic Optimization* (No. arXiv:1412.6980). arXiv. https://doi.org/10.48550/arXiv.1412.6980

Kuo, F. Y., & Sloan, I. H. (2005). Lifting the curse of dimensionality. *Notices of the AMS*, *52*(11), 1320–1328.

Kuradusenge, M., Kumaran, S., & Zennaro, M. (2020). Rainfall-Induced Landslide Prediction Using Machine Learning Models: The Case of Ngororero District, Rwanda. *International Journal of Environmental Research and Public Health*, *17*(11), Article 11. https://doi.org/10.3390/ijerph17114147

Ladyman, J., Lambert, J., & Wiesner, K. (2013). What is a complex system? *European Journal for Philosophy of Science*, *3*(1), 33–67. https://doi.org/10.1007/s13194-012-0056-8

Lann, T., Bao, H., Lan, H., Zheng, H., Yan, C., & Peng, J. (2024). Hydro-mechanical effects of vegetation on slope stability: A review. *Science of The Total Environment*, *926*, 171691. https://doi.org/10.1016/j.scitotenv.2024.171691

LaValley, M. P. (2008). Logistic Regression. *Circulation*, *117*(18), 2395–2399. https://doi.org/10.1161/CIRCULATIONAHA.106.682658

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. Proceedings of the IEEE. https://doi.org/10.1109/5.726791

les, leventhal. (1986). Type 1 and type 2 errors in the statistical analysis of listening tests. *Journal of the Audio Engineering Society*, *34*(6), 437–453.

Li, F., & Yang, Y. (2005). Analysis of recursive feature elimination methods. *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 633–634. https://doi.org/10.1145/1076034.1076164

Li, Y., Satyanaga, A., & Rahardjo, H. (2021). Characteristics of unsaturated soil slope covered with capillary barrier system and deep-rooted grass under different rainfall patterns. *International Soil and Water Conservation Research*, *9*(3), 405–418. https://doi.org/10.1016/j.iswcr.2021.03.004

Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2022). A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*, *33*(12), 6999–7019. IEEE Transactions on Neural Networks and Learning Systems. https://doi.org/10.1109/TNNLS.2021.3084827

Linderman, G. C., Rachh, M., Hoskins, J. G., Steinerberger, S., & Kluger, Y. (2019). Efficient Algorithms for t-distributed Stochastic Neighborhood Embedding. *Nature Methods*, *16*(3), 243–245. https://doi.org/10.1038/s41592-018-0308-4

Liu, Z., Hu, H., Lin, Y., Yao, Z., Xie, Z., Wei, Y., Ning, J., Cao, Y., Zhang, Z., Dong, L., & others. (2022). Swin transformer v2: Scaling up capacity and resolution. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 12009–12019.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., & Guo, B. (2021, March 25). *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows*. arXiv.Org. https://arxiv.org/abs/2103.14030v2

Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T., & Xie, S. (2022, January 10). *A ConvNet for the 2020s*. arXiv.Org. https://arxiv.org/abs/2201.03545v2

Lu, N., & Godt, J. (2008). Infinite slope stability under steady unsaturated seepage conditions. *Water Resources Research*, *44*(11). https://doi.org/10.1029/2008WR006976

Maharana, K., Mondal, S., & Nemade, B. (2022). A review: Data pre-processing and data augmentation techniques. *Global Transitions Proceedings*, *3*(1), 91–99. https://doi.org/10.1016/j.gltp.2022.04.020

Mao, A., Mohri, M., & Zhong, Y. (2023). Cross-Entropy Loss Functions: Theoretical Analysis and Applications. *Proceedings of the 40th International Conference on Machine Learning*, 23803–23828. https://proceedings.mlr.press/v202/mao23b.html

Mckinney, W. (2011). pandas: A Foundational Python Library for Data Analysis and Statistics. *Python High Performance Science Computer*.

Mukhlif, A. A., Al-Khateeb, B., & Mohammed, M. A. (2023). Incorporating a Novel Dual Transfer Learning Approach for Medical Images. *Sensors*, *23*(2), Article 2. https://doi.org/10.3390/s23020570

Mumuni, A., & Mumuni, F. (2022). Data augmentation: A comprehensive survey of modern approaches. *Array*, *16*, 100258. https://doi.org/10.1016/j.array.2022.100258

Myster, R. W., & Walker, L. R. (1997). Plant successional pathways on Puerto Rican landslides. *Journal of Tropical Ecology*, *13*(2), 165–173. https://doi.org/10.1017/S0266467400010397

Nielsen, F. (2016). Hierarchical Clustering. In F. Nielsen (Ed.), *Introduction to HPC with MPI for Data Science* (pp. 195–211). Springer International Publishing. https://doi.org/10.1007/978-3-319-21903-5_8

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., … Zoph, B. (2023, March 15). *GPT-4 Technical Report*. arXiv.Org. https://arxiv.org/abs/2303.08774v6

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, *32*. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

Polemio, M., Petrucci, O., & others. (2000). Rainfall as a landslide triggering factor an overview of recent international research. *Landslides in Research, Theory and Practice*.

Powers, D. (2008). Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. *Mach. Learn. Technol.*, *2*.

Radosavovic, I., Kosaraju, R. P., Girshick, R., He, K., & Dollár, P. (2020, March 30). *Designing Network Design Spaces*. arXiv.Org. https://arxiv.org/abs/2003.13678v1

Ranstam, J., & Cook, J. A. (2018). LASSO regression. *British Journal of Surgery*, *105*(10), 1348. https://doi.org/10.1002/bjs.10895

Rebuffi, S.-A., Gowal, S., Calian, D. A., Stimberg, F., Wiles, O., & Mann, T. A. (2021). Data Augmentation Can Improve Robustness. *Advances in Neural Information Processing Systems*, *34*, 29935–29948. https://proceedings.neurips.cc/paper/2021/hash/fb4c48608ce8825b558ccf07169a3421-Abstract.html

Restrepo, C., & Vitousek, P. (2001). Landslides, Alien Species, and the Diversity of a Hawaiian Montane Mesic Ecosystem1. *Biotropica*, *33*(3), 409–420. https://doi.org/10.1111/j.1744-7429.2001.tb00195.x

Reynolds, D. A. & others. (2009). Gaussian mixture models. *Encyclopedia of Biometrics*, *741*(659–663).

Rioul, O., & Duhamel, P. (1992). Fast algorithms for discrete and continuous wavelet transforms. *IEEE Transactions on Information Theory*, *38*(2), 569–586. IEEE Transactions on Information Theory. https://doi.org/10.1109/18.119724

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). *High-Resolution Image Synthesis with Latent Diffusion Models* (No. arXiv:2112.10752). arXiv. https://doi.org/10.48550/arXiv.2112.10752

Rothman, K. J. (2010). Curbing type I and type II errors. *European Journal of Epidemiology*, *25*(4), 223–224. https://doi.org/10.1007/s10654-010-9437-5

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, *115*(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

Salehinejad, H., Sankar, S., Barfett, J., Colak, E., & Valaee, S. (2018). *Recent Advances in Recurrent Neural Networks* (No. arXiv:1801.01078). arXiv. https://doi.org/10.48550/arXiv.1801.01078

Sasaki, Y. (2007). The truth of the F-measure. *Teach Tutor Mater, 1, 5*.

Schmidt, K. M., Roering, J. J., Stock, J. D., Dietrich, W. E., Montgomery, D. R., & Schaub, T. (2001). The variability of root cohesion as an influence on shallow landslide susceptibility in the Oregon Coast Range. *Canadian Geotechnical Journal*, *38*(5), 995–1024. https://doi.org/10.1139/t01-031

Segalini, A., Chiapponi, L., Pastarini, B., & Carini, C. (2014). Automated Inclinometer Monitoring Based on Micro Electro-Mechanical System Technology: Applications and Verification. In K. Sassa, P. Canuti, & Y. Yin (Eds.), *Landslide Science for a Safer Geoenvironment* (pp.

595–600). Springer International Publishing. https://doi.org/10.1007/978-3-319-05050-8_92

Segalini, A., Valletta, A., Carri, A., & Cavalca, E. (2019). Monitoring of a retaining wall with innovative multi-parameter tools. *Proceedings of the 4th Regional Symposium on Landslides in the Adriatic - Balkan Region*, 31–36. https://doi.org/10.35123/ReSyLAB_2019_5

Sharma, S., Sharma, S., & Athaiya, A. (2020). ACTIVATION FUNCTIONS IN NEURAL NETWORKS. *International Journal of Engineering Applied Sciences and Technology*, *04*(12), 310–316. https://doi.org/10.33564/IJEAST.2020.v04i12.054

Simonyan, K., & Zisserman, A. (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition* (No. arXiv:1409.1556). arXiv. https://doi.org/10.48550/arXiv.1409.1556

SONG, Y., & LU, Y. (2015). Decision tree methods: Applications for classification and prediction. *Shanghai Archives of Psychiatry*, *27*(2), 130–135. https://doi.org/10.11919/j.issn.1002-0829.215044

Stöckl, A. (2023). Evaluating a Synthetic Image Dataset Generated with Stable Diffusion. In X.-S. Yang, R. S. Sherratt, N. Dey, & A. Joshi (Eds.), *Proceedings of Eighth International Congress on Information and Communication Technology* (pp. 805–818). Springer Nature. https://doi.org/10.1007/978-981-99-3243-6_64

Su, X., Yan, X., & Tsai, C.-L. (2012). Linear regression. *WIREs Computational Statistics*, *4*(3), 275–294. https://doi.org/10.1002/wics.1198

Suthaharan, S. (2016). Support Vector Machine. In S. Suthaharan (Ed.), *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning* (pp. 207–235). Springer US. https://doi.org/10.1007/978-1-4899-7641-3_9

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems*, *12*. https://proceedings.neurips.cc/paper_files/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html

Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, *39*(1), 43–62. https://doi.org/10.1016/S0169-7439(97)00061-0

Teza, G., Cola, S., Brezzi, L., & Galgaro, A. (2022). Wadenow: A Matlab Toolbox for Early Forecasting of the Velocity Trend of a Rainfall-Triggered Landslide by Means of

Continuous Wavelet Transform and Deep Learning. *Geosciences*, *12*(5), Article 5. https://doi.org/10.3390/geosciences12050205

Ting, K. M. (2017). Confusion Matrix. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of Machine Learning and Data Mining* (pp. 260–260). Springer US. https://doi.org/10.1007/978-1-4899-7687-1_50

Tordesillas, A., Zhou, Z., & Batterham, R. (2018). A data-driven complex systems approach to early prediction of landslides. *Mechanics Research Communications*, *92*, 137–141. https://doi.org/10.1016/j.mechrescom.2018.08.008

Valletta, A. (2022). *Automatic detection of landslide events for risk management and early warning procedures* [Doctoral thesis, Università degli studi di Parma. Dipartimento di Ingegneria e architettura]. https://www.repository.unipr.it/handle/1889/4822

Van Rossum, G., & Drake, F. L. (2009). *Introduction To Python 3: Python Documentation Manual Part 1*. CreateSpace.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. ukasz, & Polosukhin, I. (2017). Attention is All you Need. *Advances in Neural Information Processing Systems*, *30*. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

Viroli, C., & McLachlan, G. J. (2019). Deep Gaussian mixture models. *Statistics and Computing*, *29*(1), 43–51. https://doi.org/10.1007/s11222-017-9793-z

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … van Mulbregt, P. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Wang, S., Zhao, M., Meng, X., Chen, G., Zeng, R., Yang, Q., Liu, Y., & Wang, B. (2020). Evaluation of the Effects of Forest on Slope Stability and Its Implications for Forest Management: A Case Study of Bailong River Basin, China. *Sustainability*, *12*(16), Article 16. https://doi.org/10.3390/su12166655

Wang, Z., & Oates, T. (2015). Encoding time series as images for visual inspection and classification using tiled convolutional neural networks. *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279–292. https://doi.org/10.1007/BF00992698

Weiss, K., Khoshgoftaar, T. M., & Wang, D. (2016). A survey of transfer learning. *Journal of Big Data*, *3*(1), 9. https://doi.org/10.1186/s40537-016-0043-6

Wiering, M., & Van Otterlo, M. (Eds.). (2012). *Reinforcement Learning: State-of-the-Art* (Vol. 12). Springer. https://doi.org/10.1007/978-3-642-27645-3

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316. https://doi.org/10.1016/j.neucom.2020.07.061

Zanini, A., & D'Oria, M. (2022). *14TH INTERNATIONAL CONFERENCE ON GEOSTATISTICS FOR ENVIRONMENTAL APPLICATIONS*. *14*. https://2022.geoenvia.org/wp-content/uploads/sites/7/2023/01/Proceedings_of_geoENV2022.pdf

Zhou, Z., & Hooker, G. (2021). Unbiased Measurement of Feature Importance in Tree-Based Methods. *ACM Trans. Knowl. Discov. Data*, *15*(2), 26:1-26:21. https://doi.org/10.1145/3429445

Zhu, X., Xu, Q., Tang, M., Nie, W., Ma, S., & Xu, Z. (2017). Comparison of two optimized machine learning models for predicting displacement of rainfall-induced landslide: A case study in Sichuan Province, China. *Engineering Geology*, *218*, 213–222. https://doi.org/10.1016/j.enggeo.2017.01.022