CNN-Based Detection of Venusian Volcanoes in Magellan SAR Data

G. Durante, P. A. Lopez Torres, E. Ottoboni

December 13 2024

Abstract

This paper evaluates the detection of Venusian volcanic formations in Magellan Synthetic Aperture Radar (SAR) data using Convolutional Neural Networks (CNNs). Two models were implemented: Model A, a baseline architecture, achieved a test accuracy of 95.31% and an AUC of 0.97, while Model B, incorporating dropout layers and improved preprocessing, reached similar accuracy with enhanced loss stability. The preprocessing steps included the removal of corrupted images, Principal Component Analysis (PCA) for dimensionality reduction, and SMOTE to address class imbalance. Comparisons with Feedforward Neural Networks (FFNNs), Logistic Regression and Random Forest showed CNNs' superior performance in capturing spatial patterns, with alternative methods achieving lower F1-scores and recall.

INTRODUCTION

The automated classification of visual data has become a cornerstone in the application of machine learning, particularly for high-resolution imagery from space exploration missions. This study focuses on evaluating various classifiers, including Logistic Regression, Random Forest, Feedforward Neural Networks (FFNNs), and Convolutional Neural Networks (CNNs), for classifying and detecting volcanic formations in images of Venus's surface. These images, captured by NASA's Magellan spacecraft during its radar-mapping mission from 1990 to 1994, present a unique challenge due to their high dimensionality, inherent noise, and uncertainties in ground truth labeling.

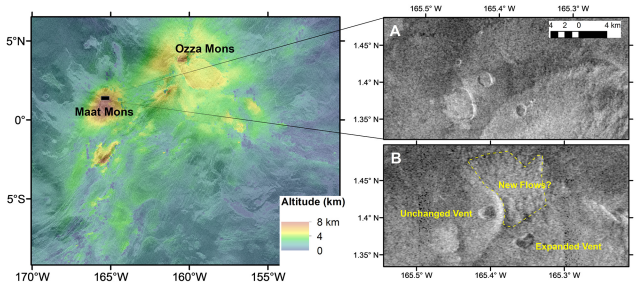


Figure 1: Altitude data for the Maat and Ozza Mons shield volcanoes on the Venus surface on the left, with the area of study indicated by a black box. On the right, before (A) and after (B) Magellan observations of the expanded vent on Maat Mons, with possible new lava flows after an eruptive event. Credit: Jet Propulsion Laboratory (NASA).

The ultimate goal of this analysis is to contribute to the development of accurate automated systems capable of assisting planetary scientists in cataloging and analyzing Venusian volcanic features. Moreover, the broader context of this study aligns with the objectives of NASA's upcoming VERITAS mission, which will deploy state-of-the-art radar and spectroscopic technologies to map Venus' surface with unprecedented detail. This mission is expected to complement Magellan's legacy by enabling further application of machine learning techniques for planetary exploration

Finally, the structure of this paper is organized as follows: The first section outlines the theoretical background of the methods and models employed. The second section describes the implementation of these concepts in detail. The results section presents a comparison between the implemented classifiers based on the obtained experimental results. The discussion section examines the existing literature relevant to the analyzed topic. Lastly, the conclusion highlights the key findings, acknowledges limitations, and proposes potential directions for future research.

THEORY

Principal Component Analysis

Dimensionality reduction techniques are crucial for high-dimensional datasets, especially when the number of features (p) greatly exceeds the number of samples (n). This is precisely the case for the dataset analyzed in this study, as discussed in detail in the Methods section. In such cases, several issues can arise. For instance, in methods like Ordinary Least Squares (OLS), the inversion of a nearly singular covariance matrix can lead to numerical instability. Moreover, high feature-to-sample ratios increase the risk of overfitting, as the model can adapt too closely to the training data, capturing noise instead of general patterns. Optimization meth-

ods, such as those used in logistic regression, may also suffer from slow convergence due to redundant or highly correlated features [1].

Principal Component Analysis (PCA) stands out as one of the most popular and effective dimensionality reduction techniques, addressing these challenges by transforming the data into a lower-dimensional space while preserving as much variability as possible. PCA achieves this by identifying a set of orthogonal directions, known as principal components, that maximize the variance of the data.

Mathematically, let the dataset be represented as a design matrix $X \in \mathbb{R}^{n \times p}$, where n is the number of samples and p the number of features. PCA begins by centering the data, which involves subtracting the mean of each feature to ensure the data is mean-centered:

$$\tilde{X} = X - \mu,$$

The covariance matrix C of the centered data is then computed as:

$$C = \frac{1}{n} \tilde{X}^\top \tilde{X},$$

where $C \in \mathbb{R}^{p \times p}$ is symmetric and encapsulates the variance of each feature along its diagonal and the covariances between features in the off-diagonal elements.

PCA relies on the eigendecomposition of C, solving the eigenvalue equation:

$$C\mathbf{v}_i = \lambda_i \mathbf{v}_i,$$

where λ_i are the eigenvalues representing the variance explained by the corresponding eigenvectors $\mathbf{v}_i$. The eigenvectors are orthogonal due to the symmetry of C, and they define the principal components. The eigenvalues are ordered such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p$.

To reduce dimensionality, the first k eigenvectors, corresponding to the largest k eigenvalues, are selected to form the projection matrix $V_k \in \mathbb{R}^{p \times k}$. The data is then projected onto this lower-dimensional subspace as:

$$Z = \tilde{X} V_k,$$

where $Z \in \mathbb{R}^{n \times k}$ represents the transformed data in the reduced-dimensional space.

By retaining only the components with the largest eigenvalues, PCA effectively reduces dimensionality while minimizing the loss of information. This not only mitigates overfitting and numerical instability but also accelerates optimization processes in subsequent modeling tasks [2].

Logistic Regression

Logistic regression is widely used for binary classification problems, where the goal is to classify data points into one of two categories. The method leverages the *sigmoid (logistic) function*, defined as:

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

The sigmoid function $S(x)$ maps any real-valued number to the range $(0, 1)$, making it useful for estimating probabilities.

In logistic regression, the loss function commonly used is the *log-loss* (also known as the *binary cross-entropy loss*), which measures the performance of the classification model. It is defined as:

$$L(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right],$$

where y_i is the true label (0 or 1), $\hat{y}_i = S(\theta^\top x_i)$ is the predicted probability for the positive class, and θ represents the model parameters.

Minimizing the *negative log-likelihood* (NLL) of the data under the model is equivalent to maximizing the *log-likelihood* because these operations are inverses of each other. Maximizing the likelihood aligns the model's predictions with the observed data, while minimizing the NLL penalizes deviations from the true labels. Since optimization routines often aim to minimize a cost, the NLL is used as the loss function [2].

For a deeper theoretical understanding of logistic regression, refer to the 'Theory' section, specifically the 'Logistic Regression' subsection, in our previous report, *Evaluation of Feedforward Neural Network Performance* [3].

Random Forest

Random forests represent valuable classifiers to consider when addressing a classification task. Before examining them, we need to introduce the concept of decision trees, particularly classification trees.

Classification trees are a type of supervised learning algorithm used for classification tasks, although a related variant, regression trees, is used for predicting continuous outcomes. The process begins with a root node that contains the entire dataset. From this root, the algorithm recursively splits the data into smaller subsets based on decision rules, creating interior nodes (also called decision nodes). This splitting continues until a specified stopping criterion is met, such as a minimum number of samples per node or a maximum tree depth, resulting in terminal nodes, known as leaf nodes, which represent the final classification outcomes.

More specifically, from the root node, the tree develops using *recursive binary splitting*—a greedy, top-down approach that partitions the dataset into M distinct, non-overlapping regions $(R_1, R_2, \ldots, R_M)$. For each region R_m, the value p_{mk} is defined as the proportion of observations in region R_m that belong to class k, and is given by:

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k),$$

where N_m is the total number of observations in R_m, and $I(y_i = k)$ is an indicator function that equals 1 if observation x_i belongs to class k, and 0 otherwise.

The splits that generate these M regions happen based on impurity measures, such as:

- Misclassification error:

$$1 - \max_k \{p_{mk}\},$$

  which measures the proportion of observations in R_m that do not belong to the most common class. A low misclassification error indicates that the region is "pure," dominated by a single class.

- Gini index g:

$$g = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}).$$

- Information entropy (or just entropy) s:

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}.$$

Here, $m \in \{1, 2, \ldots, M\}$ indexes the regions $(R_1, R_2, \ldots, R_M)$, and $k \in \{1, 2, \ldots, K\}$ indexes the K possible class labels.

We denote the resulting tree as T_0. However, this approach often results in overfitting, producing unnecessarily large and complex trees that memorize noise in the training data rather than generalizing well to unseen data. To address this issue, pruning is a critical next step. In particular, the *cost complexity pruning* algorithm serves as an efficient method to simplify the tree while retaining its predictive performance. This method balances the complexity of the tree with its classification performance. For each value of the complexity parameter α, a subtree $T \subseteq T_0$ is selected to minimize the following criterion:

$$\text{Cost}(T) = \sum_{m=1}^{|T|} \text{Impurity}(R_m) + \alpha|T|,$$

where $|T|$ is the number of terminal nodes in the subtree T, Impurity(R_m) measures node impurity (e.g., misclassification error, Gini index, or entropy) for region R_m, and α is a tuning parameter controlling the trade-off between tree size and impurity.

Pruning involves gradually increasing α to remove splits that contribute little to reducing impurity, resulting in a simpler tree that avoids overfitting and generalizes better to unseen data. Nevertheless, even with pruning, classification trees often struggle with overfitting, particularly in cases with high data variance or limited training samples. For this reason, random forests are widely regarded as a superior method for classification tasks.

Random forests improve upon bagging (Bootstrap Aggregating) by introducing additional randomness. Like bagged trees, random forests build multiple decision trees using bootstrap samples of the training data. However, during tree construction, a random subset of m predictors is selected at each split from the total p predictors, and the best split is chosen only from this subset. Typically, m is set to $\sqrt{p}$ for classification tasks. This random feature selection reduces correlation between the trees, improving their variance-reduction capability when predictions are aggregated.

The final prediction is made by combining the outputs of all trees: by majority voting for classification tasks and by averaging for regression tasks. By decorrelating the trees, random forests mitigate overfitting more effectively than bagged trees, making them a robust and widely used method for both classification and regression [1] [2].

Feedforward Neural Networks

A Feedforward Neural Network (FFNN) is a type of artificial neural network where connections between nodes do not form cycles. It is commonly used in supervised learning tasks such as regression and classification due to its simplicity and ability to model complex nonlinear relationships.

A FFNN consists of three primary components: an input layer, hidden layers, and an output layer. The input layer corresponds to the features of the dataset, with each node representing one input dimension. Hidden layers process inputs through weighted connections and nonlinear activation functions, enabling the network to learn hierarchical feature representations. Stacking multiple hidden layers creates deep networks capable of capturing intricate data patterns. The output layer provides predictions, with its structure determined by the task. For instance, a single node with a sigmoid activation is common in binary classification, while softmax and linear activations are used for multi-class classification and regression, respectively.

Forward propagation involves the sequential flow of data through the network. Each neuron computes a weighted sum of its inputs, adds a bias, and applies an activation function:

$$z_i^{(l)} = \sum_{j=1}^{n} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)},$$

$$a_i^{(l)} = \phi(z_i^{(l)}),$$

where $z_i^{(l)}$ is the pre-activation value, ϕ the activation function, $w_{ij}^{(l)}$ the weight of the connection, $b_i^{(l)}$ the bias, and $a_j^{(l-1)}$ the activation from the previous layer. This process continues iteratively until predictions are produced at the output layer.

Learning in a FFNN involves adjusting weights and biases to minimize a predefined loss function, such as mean squared error for regression or cross-entropy for classification. Using gradient-based optimization algorithms like stochastic gradient descent or Adam, the network updates its parameters iteratively. Gradients are computed through backpropagation, which applies the chain rule to propagate errors backward through the network. Parameters are then updated as:

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

where η is the learning rate, $\mathcal{L}$ the loss function.

FFNNs are versatile and capable of approximating any continuous function, as guaranteed by the universal approximation theorem. They can handle high-dimensional data effectively. However, they can be computationally intensive, especially with large datasets or deep architectures, and are prone to overfitting. Techniques like dropout and L_2-norm

regularization are often employed to mitigate these issues. For a more detailed discussion on this topic, please refer to our previous paper [3].

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are among the most significant advancements in deep learning, particularly in the field of image and video analysis. These networks are designed to process structured data by exploiting spatial hierarchies in images through mathematical operations like convolution, pooling, and fully connected transformations. The ability of CNNs to automatically learn and extract features from raw data has led to their widespread use in computer vision tasks, including image classification, object detection, and segmentation. To implement the neural networks in Python we will be using different libraries: Tensorflow[4], Pytorch[5] and Keras[6].

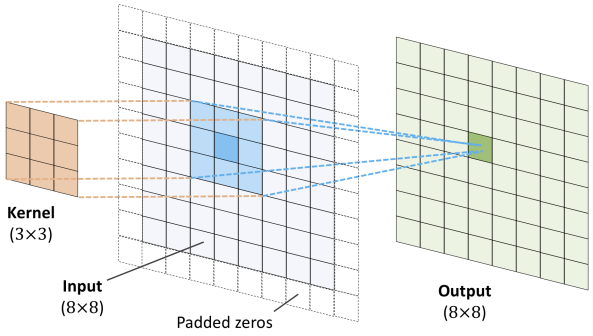


Figure 2: Convolution operation. A 3×3 kernel slides over an 8×8 input matrix, padded with zeros. The operation produces an output of the same size by computing element-wise multiplications and summing the results. Ref: Chapter 13 S. Raschka et al.[7]

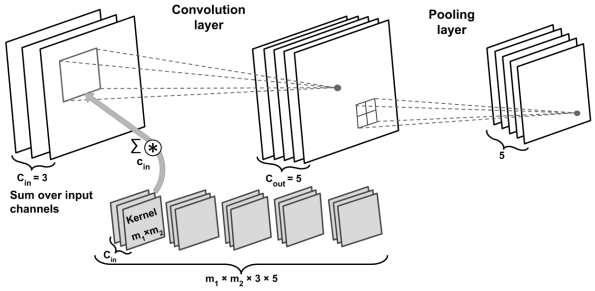


Figure 3: Multi-channel convolution applied to three input channels (e.g., RGB). Each kernel spans all input channels, producing a single output feature map. Using multiple kernels, five distinct feature maps are created. Ref: Chapter 13 S. Raschka et al.[7]

The convolution operation, shown in Figure **2**, lies at the heart of CNNs. In its simplest form, convolution involves sliding a small matrix, called a kernel, over the input data and computing a weighted sum at each position. Mathematically, for an input matrix $X \in \mathbb{R}^{H \times W}$ and a kernel $K \in \mathbb{R}^{m \times n}$, the convolution operation produces an output feature map Y as follows:

$$Y(i,j) = \sum_{p=0}^{m-1} \sum_{q=0}^{n-1} X(i+p, j+q) \cdot K(p,q).$$

Here, (i,j) represents the spatial location in the output feature map. The kernel slides over the input data in fixed increments, known as the stride, and optionally includes zero-padding around the borders to preserve spatial dimensions. The dimensions of the output feature map are determined by the relationship:

$$H_{\text{out}} = \frac{H - m + 2p}{s} + 1, \quad W_{\text{out}} = \frac{W - n + 2p}{s} + 1,$$

where p is the padding size, s is the stride, and m, n are the dimensions of the kernel.

In the case of multi-channel inputs, such as RGB images, the convolution operation is extended to accommodate multiple input channels. As shown in Figure **3**, the kernel also spans the depth of the input, producing a single feature map by summing over the contributions from all input channels. If the input data has C_{in} channels and the kernel has C_{out} filters, the output will consist of C_{out} feature maps, with each filter learning to detect specific patterns.

After the convolution layers, pooling layers, depicted in Figure **4**, are applied to reduce the spatial dimensions of the feature maps, thereby improving computational efficiency and reducing overfitting. Max-pooling, one of the most commonly used pooling methods, selects the maximum value within a small region of the feature map, while mean-pooling computes the average value. For an input feature map X, max-pooling with a filter size $m \times n$ and a stride s is given by:

$$Y(i,j) = \max_{p=0,q=0}^{m-1,n-1} X(i \cdot s + p, j \cdot s + q).$$

Pooling layers are vital in preserving the most salient features while discarding less relevant details.

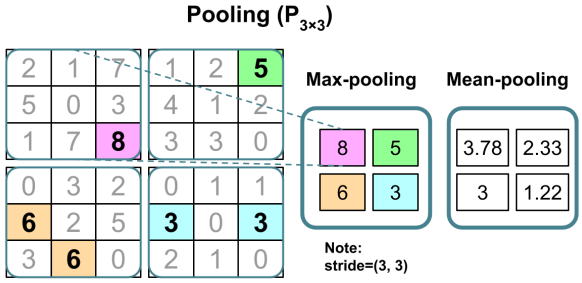


Figure 4: Max-pooling and mean-pooling. Max-pooling selects the largest value, and mean-pooling computes the average within a region. A 3×3 pooling filter with stride 3 is used, reducing spatial dimensions. Ref: Chapter 13 S. Raschka et al.[7]

The final stages of a CNN involve flattening the multidimensional feature maps into a one-dimensional vector, followed by fully connected layers, as shown in Figure **5**. These layers operate similarly to traditional neural networks. For an

input vector $\mathbf{x}$, a fully connected layer computes:

$$\mathbf{y} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}),$$

where $\mathbf{W}$ represents the weight matrix, $\mathbf{b}$ is the bias vector, and σ is an activation function such as ReLU or sigmoid. In the final layer, a softmax activation function is often used to produce probabilities for classification tasks:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}},$$

where z_i represents the raw score for class i.

The architecture illustrated in Figure **5** showcases a typical CNN structure applied to image classification. Starting with an input image of dimensions $28 \times 28 \times 1$ (e.g., a grayscale image), the network applies a series of convolutional layers (using 32 and 64 filters), followed by max-pooling operations to downsample the feature maps. After flattening the feature maps, the fully connected layers culminate in a 10-class output layer, suitable for tasks like handwritten digit recognition.

By combining these operations with fully connected layers, CNNs can represent increasingly abstract patterns, enabling high accuracy in tasks requiring spatial understanding. The visual illustrations provided elucidate the core operations and their roles within the broader CNN architecture.

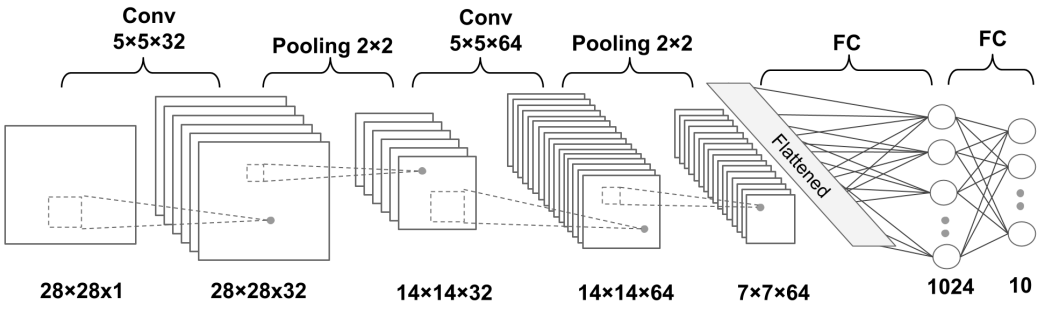
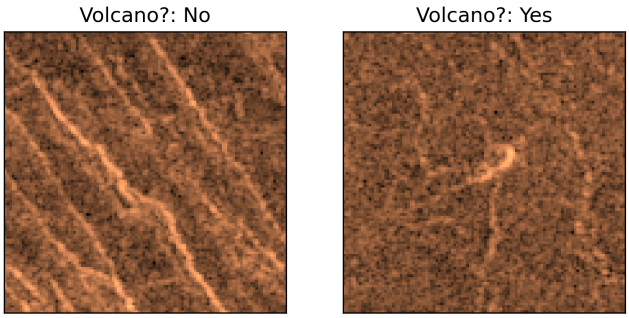


Figure 5: Architecture of a Convolutional Neural Network (CNN) designed for classification. It consists of two convolutional layers with 5×5 kernels and pooling layers, followed by fully connected layers. The input is a grayscale image of size $28 \times 28 \times 1$, and the final output is a 10-class classification. Ref: Chapter 13 S. Raschka et al.[7]

METHODS

Dataset



Volcano?	0.0
Type	NaN
Radius	NaN
Number Volcanoes	NaN

Volcano?	1.0
Type	4.0
Radius	2.24
Number Volcanoes	2.00

Figure 6: Comparative visualization of two images: one classified as a volcano (right) and one not classified as a volcano (left). The accompanying data tables provide metadata for each image.

The dataset used, derived from the JARtool experiments, consists of 1024x1024-pixel grayscale images reduced to 110x110-pixel inputs for computational feasibility. Each image captures radar backscatter intensities that reflect surface characteristics, such as roughness and topography, providing a foundation for identifying volcanic features. Labels include binary indicators for the presence of volcanoes, alongside supplementary information such as volcano type, radius, and count, with varying degrees of labeling confidence. In Figure **6** we can se two examples of how data look like in the dataset.

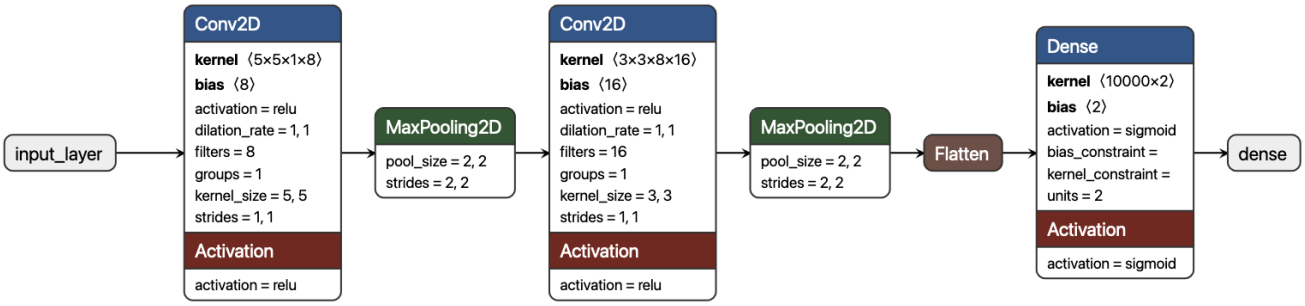
Convolutional Neural Network

The first model employed in this study, referred to as Model A, is a Convolutional Neural Network (CNN) designed as a baseline classifier (Figure **7**a). The architecture of this model is relatively simple and consists of two convolutional layers, each followed by a max-pooling layer, a flattening layer to convert the multidimensional output into a one-dimensional array, and a final dense layer for classification. The dense layer uses a sigmoid activation function, allowing the model to predict binary outcomes, specifically whether a volcano is present or absent. This architecture was chosen to provide an initial assessment of the dataset and its suitability for binary classification tasks.

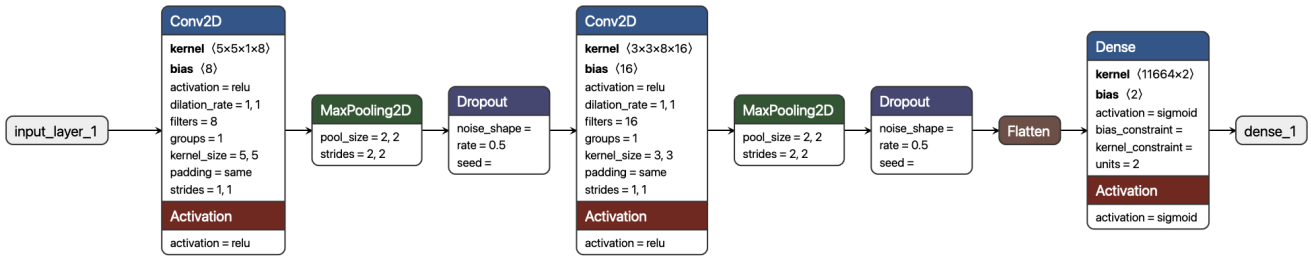
To further improve the model's performance, a preprocessing step was implemented before training the second model, Model B (Figure **7**b). This preprocessing focused on cleaning the dataset by identifying and removing corrupted images.

Corruption in this context referred to images that were entirely black or partially damaged. Fully black images were straightforward to detect by summing their pixel values. Partially corrupted images were identified using a more nuanced approach that involved summing pixel values across fixed intervals within the image data. If the sum remained zero over these intervals, the image was flagged as corrupted. This process aimed to eliminate data that could adversely affect the training process, though limitations in detecting certain corruption patterns were acknowledged.

Following the data cleaning phase, Model B was developed with an enhanced architecture. This model introduced dropout layers to address overfitting, a common issue in neural networks, particularly when working with smaller datasets. The dropout layers were added after each convolutional and max-pooling layer to randomly deactivate a fraction of the neurons during training, forcing the model to generalize better. Model B retained the overall structure of two convolutional layers followed by max-pooling layers but incorporated these dropout layers and used `"SAME"` padding in the convolutional layers to preserve spatial dimensions.



(a) CNN Model A architecture



(b) CNN Model B architecture

Figure 7: Architectures of two Convolutional Neural Networks, Model A and Model B. ModelA comprises a series of convolutional and max-pooling layers, culminating in a dense output layer with sigmoid activation. ModelB extends this design by incorporating dropout layers after each max-pooling stage to enhance generalization.

Both Model A and Model B were trained using the Adam optimizer (algorithm discussed in our previous report – *Evaluation of Feed Forward Neural Network Performance* [3]), a widely utilized optimization algorithm in deep learning, known for its adaptive learning rate and ability to efficiently handle sparse gradients. The update rule for Adam combines the benefits of momentum and RMSprop and is given by the equations:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

where g_t is the gradient of the loss function with respect to the model parameters θ, m_t and v_t are the exponentially decaying averages of past gradients and squared gradients, respectively, β_1 and β_2 are hyperparameters controlling the decay rates, and η is the learning rate. These features make Adam particularly effective in ensuring stable and efficient convergence.

For the loss function, both models employed binary crossentropy, appropriate for the binary classification task of determining the presence or absence of volcanic features. Binary crossentropy is defined as:

$$
\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i) \right]
$$

where y_i represents the true label (0 or 1) for the i-th sample, $\hat{y}_i$ is the predicted probability of the positive class, and N is the total number of samples. This loss function quantifies the difference between the predicted probabilities and the actual labels, guiding the optimization process to mini-

mize errors during training. The combination of the Adam optimizer and binary crossentropy ensured effective learning and accurate classification in both models.

Feedforward Neural Networks

We looked at two different frameworks for implementing Feedforward Neural Networks (FFNNs): PyTorch and Keras. The PyTorch-based approach focused on designing a simple yet effective architecture, starting with an input layer that incorporated 12100 features from the dataset. The data was then propagated through two fully connected hidden layers consisting of 64 and 32 neurons respectively. Each hidden layer used the ReLU activation function to introduce non-linearity and included a 50% probability dropout mechanism, effectively reducing the risk of overfitting [8]. Finally, the output layer contained a number of neurons equal to the total number of classes in the dataset, using a softmax activation function to generate a probability distribution across the classes [9]. The training phase involved extensive experimentation with optimisation algorithms, including Adam, SGD with Momentum, and RMSprop. The Adam optimiser, a widely used adaptive gradient method, combines the advantages of RMSprop and momentum, making it well suited to tasks requiring efficient and stable convergence [8]. SGD with momentum extends the basic stochastic gradient descent approach by incorporating a momentum term, which helps accelerate convergence towards consistent gradients and dampens oscillations [8]. RMSprop, on the other hand, adapts the learning rate for each parameter individually, maintaining a moving average of the squared gradients to achieve a robust balance between speed and stability during training [7]. The loss function chosen was the Cross-Entropy Loss, which is specifically tailored for multi-class classification problems as it measures the divergence between the predicted probability distribution and the true class labels. Hyperparameter optimisation played a central role, exploring different learning rates from 10^{-7} to 10^{-3}, and testing different configurations for the number of epochs (20, 30, and 40). Each combination of optimiser, learning rate, and number of epochs was systematically evaluated through iterative training on the training set and subsequent performance testing on the test set. The evaluation process captured accuracy metrics and loss values for each hyperparameter configuration. These results were visualised using heat maps, allowing clear identification of the optimal settings that maximised accuracy on the test set [10]. This comprehensive approach not only identified the best performing hyperparameter combinations, but also provided valuable insight into the model's behaviour under varying training conditions.

In the Keras implementation, the data preprocessing involved normalising the input values using StandardScaler to ensure a standardised distribution with a mean of zero and a standard deviation of one. At the same time, the target labels were encoded into a one-hot format using OneHotEncoder, a crucial step to ensure compatibility with the categorical loss function employed. The neural network architecture was designed with an input layer sized to match the number of features in the dataset. This was followed by two fully connected hidden layers consisting of 64 and 32 neurons respectively, both using the ReLU activation function. The output layer, which was tailored to the number of classes in the problem, used a softmax activation function to generate normalised probabilities for each class. For training, Categorical Crossentropy was chosen as the loss function, as it is well suited to multi-class classification tasks. The Adam optimisation algorithm was used with an initial learning rate of 0.001 [7]. To monitor the progress of the model, 20% of the training set was used for validation during training. This approach allowed continuous evaluation of performance on the validation set at the end of each epoch. Training was performed using mini-batches of size 32 over 25 epochs. Finally, during evaluation, the test set was preprocessed to ensure that it was fully aligned with the input data and labels, eliminating any discrepancies in the number of samples between input and output. The performance of the model was assessed based on its accuracy on the test set.

Logistic Regression

A simpler classifier evaluated in this analysis is logistic regression. As with any model implementation, data preprocessing plays a critical role in determining its performance. One of the first steps in our preprocessing pipeline was the removal of corrupted images from the dataset, as retaining such data could negatively impact model performance and the reliability of the results. Normalization was the second step, ensuring that all features were on the same scale. This prevents any feature from dominating due to large numerical ranges, which is important because logistic regression is sensitive to feature scaling. In this analysis of image data, where pixel values ranged from 0 to 255, dividing all values by 255 normalized the features to a [0, 1] range. This approach was computationally efficient and produced the same result as using the `MinMaxScaler` from `scikit-learn` [11].

The next step in our data preprocessing was Principal Component Analysis (PCA). In our dataset, we had 6791 training samples (after having removed the corrupted images), each consisting of 12100 features (flattened image pixels). The number of features p far exceeded the number of samples n, which increased the risk of overfitting and slow convergence for gradient descent methods. To address this, we applied PCA to reduce the dimensionality of the data while preserving as much variance as possible. Specifically, we set a goal of retaining 95% of the total variance. This approach is considered preferable to arbitrarily choosing the number of dimensions [2]. This process resulted in the retention of 2069 principal components, significantly reducing the feature space ($p << n$) while maintaining the most important information for classification.

Additionally, we applied an oversampling technique to address the class imbalance present in the dataset. In fact, after having removed the corrupted images, the training set consisted of 6791 images, out of which only 987 were labeled as volcanoes. This imbalance could have led to a biased model. To counter this, we used SMOTE (Synthetic Minority Over-sampling Technique), an advanced oversampling method that generates synthetic examples rather than simply duplicating the minority class instances. After applying

SMOTE, we ended up with 5804 volcano images and 5804 non-volcano images, achieving a properly balanced dataset. This allowed the model to learn more effectively from both classes, improving its generalization and classification performance.

For the implementation of the logistic regression model, we developed a custom class that utilized stochastic gradient descent (SGD). The model employed mini-batch gradient descent, which processed small subsets of the dataset to improve computational efficiency while maintaining stable gradient updates. A momentum term was included to accelerate convergence and reduce oscillations during training. To enhance efficiency and prevent overfitting, we incorporated early stopping, which terminated training if the loss failed to improve within a specified patience period, avoiding unnecessary computations. The hyperparameter values used for these features were consistent with those applied in the logistic regression implementation from our second report, *Evaluation of Feed Forward Neural Network Performance* [3].

Between its parameters, the class also included the *learning rate* used for the SGD and the *regularization parameter* (L2). Both hyperparameters were fine-tuned through a grid search approach [7], using the F1-score as the evaluation metric. For the learning rate, candidate values were logarithmically sampled from 10^{-4} to 10^{0}. We found that a value of 0.006 yielded the best results. The regularization parameter, on the other hand, was evaluated over a range of values from 10^{-3} to 10^{1} and was finally set to 0.0028, balancing the model's complexity and its ability to generalize. This first analysis showed an F1-score of 0.55 for the model's ability to identify volcanoes. The precision of 0.39 indicated that the model was overestimating the number of volcano images, as many of its predictions for the `Volcano` class were incorrect (high number of false positives). However, the high recall of 0.95 showed that the model was able to correctly identify most images containing volcanoes. This combination of high recall and low precision highlighted that, while the model was good at identifying volcanoes, it tended to incorrectly classify many non-volcano images as volcanoes.

To further improve performance, we focused on fine-tuning the *classification threshold*. Specifically, we identified the optimal decision boundary for classifying the presence or absence of a volcano in the given image. Using again a grid search approach, we determined that the optimal threshold was 0.79 (up from the default of 0.5). After fine-tuning the threshold, we observed improvements in both the model's precision and recall for the `Volcano` class. This final optimization produced the best results, as announced and discussed in the subsection related to logistic regression results.

Random Forest

Among the various classifiers considered for this analysis, a tree-based model, such as Random Forest, was included due to its robustness and ability to handle complex classification tasks effectively.

Before implementing the model, we followed the same data preprocessing procedure applied for logistic regression. This began with the removal of corrupted images, followed by normalization. It is worth noting that, unlike gradient-based models, Random Forest is not sensitive to feature scaling, meaning that steps like normalization are not strictly required. However, given the high dimensionality of the dataset (12100 input features) and its unbalanced nature, we opted to apply both PCA and SMOTE, which required normalized data. The normalization process divided each feature by the highest pixel intensity value, 255, consistent with the preprocessing applied to the other models.

We implemented the Random Forest model using the `RandomForestClassifier` class from the `sklearn.ensemble` module of the Scikit-learn library [11]. For the initial implementation, we applied minimal regularization to observe the model's behavior without strong constraints. The key hyperparameters for this setup were:

- `n_estimators`: Number of decision trees in the ensemble (`1000`).
- `max_depth`: Maximum depth of each decision tree (`None`, allowing full growth).
- `max_features`: Number of features randomly selected at each split (`sqrt`, the default setting).
- `criterion`: Impurity measure used for evaluating splits (`gini` and `entropy` were tested).

By analyzing the metrics obtained from the model's performance on both the train and test datasets, we observed that, regardless of whether `gini` or `entropy` was used as the impurity measure, the model performed exceptionally well on the train data. However, its results on the test data, particularly in identifying the presence of volcanoes, were only moderate. To address this, we introduced constraints by setting the `max_depth` parameter to `10`, significantly limiting the model's complexity. While this adjustment led to slightly reduced metrics on the train data, it mitigated overfitting and improved the model's ability to generalize to the test data. However, the test set results, particularly for identifying the presence of volcanoes, highlighted the need for a more thorough analysis. (see Table **1**).

Precision	Recall	F1-Score
0.47	0.69	0.56

Table 1: Metrics for the Random Forest model with `max_depth=10` in identifying the `Volcano` class.

To further refine the model, we conducted a grid search using the `GridSearchCV` class from the Scikit-learn library [11], with 5-fold cross-validation. The search explored tree depths from 1 to 15 and used an ensemble size of 500 trees (reduced from 1000 to improve computational efficiency), optimizing for the F1-score as the primary metric. The results of this analysis are presented in Figure **8**, showing a consistent improvement in both training and test scores, indicating that the model effectively learns useful patterns from the data while maintaining its ability to generalize to unseen examples.

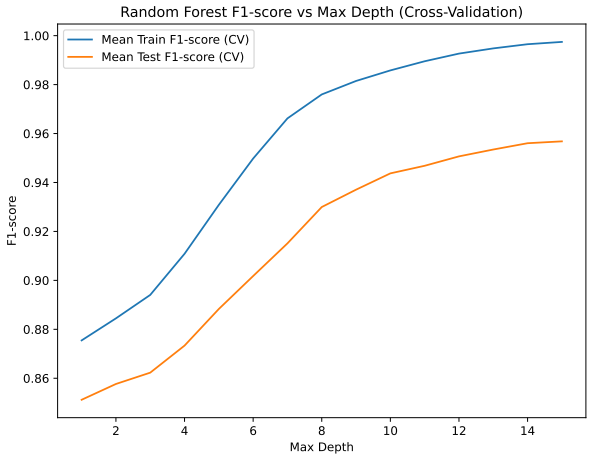


Figure 8: Comparison of the mean train and test F1-scores for the Random Forest model across maximum tree depths ranging from 1 to 15, evaluated using 5-fold cross-validation. The lack of a clear divergence between the training and test metrics as the maximum depth increases suggests that the model has not yet reached a point of significant overfitting.

Following the grid search procedure, we incorporated the best `max_depth` parameter, found to be 15 with an F1-score of 0.96, into the final Random Forest model. The detailed results are presented in the subsection on random forest results.

RESULTS

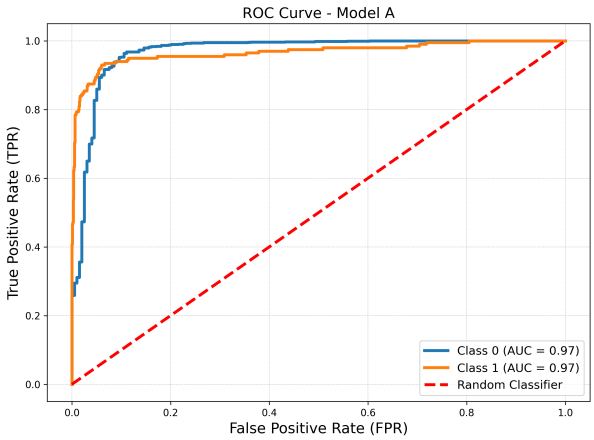
Convolutional Neural Network

Model A served as a foundational effort to understand the dataset's characteristics, while Model B refined this approach by addressing data quality issues and incorporating techniques to improve generalization. In the Table **2** we can analyze the metrics of the two models trained over 25 epochs (Figure **13**b: Training and Evaluation of Model B). Model A achieved a test loss of 0.1794, a test accuracy of 95.31%, and an AUC of 0.97. Conversely, Model B, which incorporates improved data quality and generalization techniques, obtained a marginally lower test loss of 0.1695 and a test accuracy of 95.02%, with a slightly reduced AUC of 0.96.

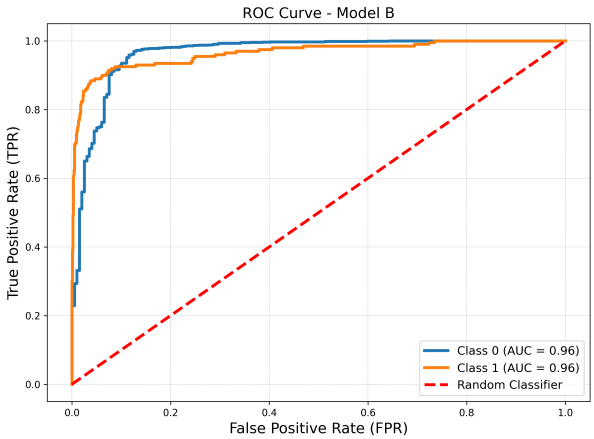
Model	Test Loss	Test Accuracy	AUC
A	0.1794	0.9531	0.97
B	0.1695	0.9502	0.96

Table 2: Final metrics for CNN's models to correctly identify the correct class: Test loss and Accuracy.

The ROC curves for Model A and Model B, shown in Figure **13**, provide a detailed view of each model's discriminative capability. The Area Under the Curve (AUC) quantifies the overall performance of the model, with values closer to 1.0 indicating near-perfect classification and values near 0.5 representing random guessing. As depicted in Figure **13**a, the ROC curve for Model A demonstrates a smooth and steep rise towards the upper-left corner of the plot. This behavior corresponds to a high AUC value of 0.97 for both classes. The steepness near the origin implies that Model A achieves a low FPR even at higher thresholds, thereby minimizing false positives. For Model B (Figure **13**b), the ROC curve shows a similar trend but with minor deviations that reflect slightly reduced performance. The AUC for both classes is 0.96, indicating a slight loss in discriminative power compared to Model A. While Model B maintains a curve shape similar to Model A, its trajectory flattens marginally earlier, suggesting a slightly higher FPR at certain thresholds.



(a) ROC Curve for Model A. The model achieves an AUC of 0.97 for both classes, indicating high discriminative performance.



(b) ROC Curve for Model B. The model achieves an AUC of 0.96 for both classes, showing a slightly reduced but still strong performance compared to Model A.

Figure 9: Comparison of ROC Curves for Models A and B. The curves demonstrate the trade-off between the true positive rate (TPR) and the false positive rate (FPR) across different threshold values, with both models exhibiting strong classification capabilities.

The AUC difference between Model A and Model B is small, but notable. Model A achieves marginally higher performance, particularly in scenarios requiring strict false positive

control. Model B, though slightly less precise, still demonstrates robust performance and may generalize better in unseen or noisier data due to the additional optimization techniques.

The training and validation performance of both models over 25 epochs is depicted in Figure **13**. For Model A (Figure **13**a), the training loss exhibits a consistent downward trend, indicating effective learning. Validation loss stabilizes after approximately 10 epochs, suggesting a good balance between optimization and generalization with minimal overfitting. Training accuracy improves steadily, converging around 95.3%, while validation accuracy follows a similar trend. Model B (Figure **13**b) shows a consistent decrease in training loss, similar to Model A. However, validation loss decreases sharply during the initial 10 epochs before stabilizing with minor fluctuations. Training and validation accuracies converge to values near 95%, slightly lower than Model A. This trend highlights the influence of additional techniques incorporated in Model B, which, while improving generalization, result in marginally reduced accuracy. A comparative analysis between the two models reveals key insights. First, the loss dynamics show that Model A achieves a more stable and smooth decrease in both training and validation loss, highlighting superior convergence and reduced overfitting. In contrast, Model B demonstrates minor fluctuations in the validation loss after epoch 10, suggesting a slight compromise in stability. Second, the accuracy trends indicate that while both models achieve high training accuracy (>95%), Model A maintains a higher validation accuracy (0.94 compared to 0.93) and exhibits a narrower gap between training and validation performance over time. Finally, regarding convergence speed, both models show rapid improvement in loss and accuracy during the initial 10 epochs. However, Model A stabilizes earlier and maintains consistency throughout the training process, while Model B requires additional epochs to converge and shows minor variations.

In conclusion, Model A demonstrates smoother convergence, lower validation loss, and more stable generalization performance, making it more reliable for unseen data. Model B, while effective and achieving comparable accuracy, shows minor instability in validation loss, which slightly impacts its generalization capability.

Feedforward Neural Networks

The entire training process was closely monitored, with both loss and accuracy trends analysed on a period basis. Of the optimisers tested, Adam showed rapid convergence, characterised by a significant reduction in loss and a steady increase in accuracy. The key results can be summarised as follows:

Optimiser	Learning Rate	Epochs	Accuracy (%)
Adam	0.0006667	20	94.33
SGD	0.0001667	20	93.96
RMSprop	0.0003334	20	94.33

Table 3: Comparison of the performance of different optimisers

Adam stood out as the best optimiser, achieving 94.33% accuracy, due to its ability to effectively balance stability and speed of convergence. Its optimal learning rate of 0.0006667 allowed a consistent reduction in loss and increase in accuracy during training. SGD, although achieving a slightly lower accuracy of 93.96%, performed well. However, it requires a more precise tuning of the learning rate, which in this case was 0.0001667, to approach Adam's results. Finally, RMSprop matched Adam's results, also achieving 94.33% accuracy with a learning rate of 0.0003334. This makes it an equally valid alternative, particularly useful in similar configurations. These results highlight how robust Adam is in image classification scenarios, but also show that RMSprop can be an equally effective choice. The effectiveness of SGD, on the other hand, is more dependent on the choice of parameters.

To better understand and compare the performance of the optimisers, two summary graphs have been created:

1. A heatmap showing accuracy as a function of number of epochs and learning rate, providing an overview of performance (see Figure **10**).
2. A comparative graph showing the trend in loss and accuracy for the best results obtained with each optimiser.

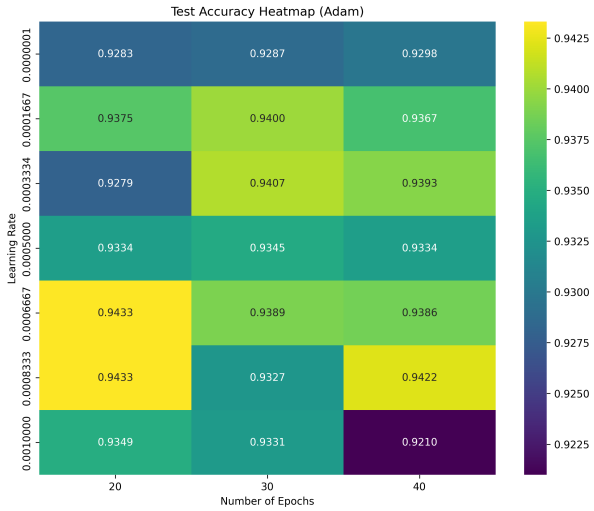


Figure 10: Heatmap of test accuracy for a neural network trained with the Adam optimizer across learning rates (10^{-7} to 10^{-3}) and epochs (20, 30, 40). Optimal accuracy (94.33%) is achieved at a learning rate of 0.00006667 with 20 epochs, highlighting the best generalization performance.

The analysis highlights Adam's effectiveness in balancing accuracy and stability during training, making it a robust choice for complex image classification tasks. RMSprop emerges as an equally strong alternative, particularly under similar configurations. While slightly less powerful in this scenario, SGD remains a competitive option, especially when fine-tuned for specific datasets. For a detailed comparison of training and testing performance using both PyTorch

and Keras implementations, please refer to the accompanying notebook available on GitHub.

Logistic Regression

The final metrics obtained from the fully optimized logistic regression for the correct identification of the `Volcano` class are shown in Table **4**.

Precision	Recall	F1-Score	Accuracy
0.78	0.79	0.78	0.93

Table 4: Final metrics for the fully optimized logistic regression in terms of its ability to correctly identify the `Volcano` class.

Building on the final metrics presented above, we now examine the performance of the logistic regression model in more detail. The confusion matrix for the model is shown in Figure **11**. This matrix provides a detailed breakdown of the model's performance in terms of true positives, false positives, true negatives, and false negatives. As shown in the confusion matrix, we achieved a relatively low number of false positives, which is critical for applications where incorrectly classifying a non-volcano as a volcano could have serious consequences.

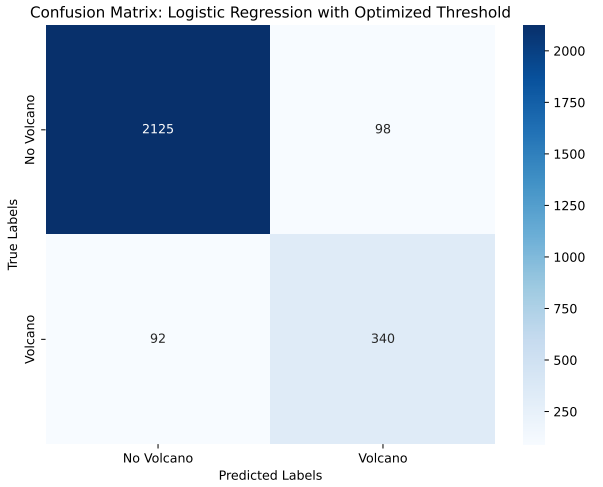


Figure 11: Confusion matrix for the fully optimized logistic regression model, highlighting its performance in classifying the `Volcano` class.

Despite the progress made with logistic regression, its key limitation remained the assumption of linearity. This assumption prevented the model from fully capturing complex patterns in the data. To overcome this, exploring non-linear classifiers, such as decision trees and random forests, could provide significant performance improvements by better modeling the underlying relationships. In the following section, we examine the performance of the Random Forest classifier.

Random Forest

The results obtained from the implementation of the random forest model fell short of initial expectations. Even after optimizing the `max_depth` parameter for each tree, the performance was mediocre. Table **5** provides an overview of the key metrics.

Precision	Recall	F1-Score	Accuracy
0.55	0.66	0.60	0.86

Table 5: Final metrics for the optimized random forest model (`max_depth=15`) in terms of its ability to correctly identify the `Volcano` class.

As shown, the random forest model's ability to correctly identify volcanic images was underwhelming. A possible explanation for this could be overfitting caused by the tree depth (`max_depth=15`), though this is unexpected given the use of grid search with cross-validation to optimize the parameter. The confusion matrix in Figure **12** offers deeper insights into the model's classification performance.

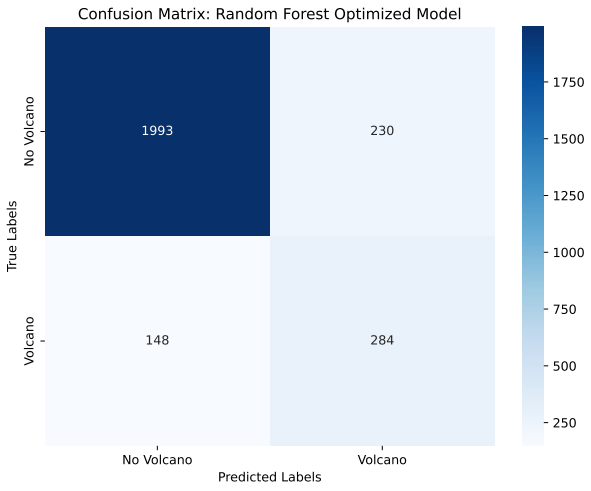
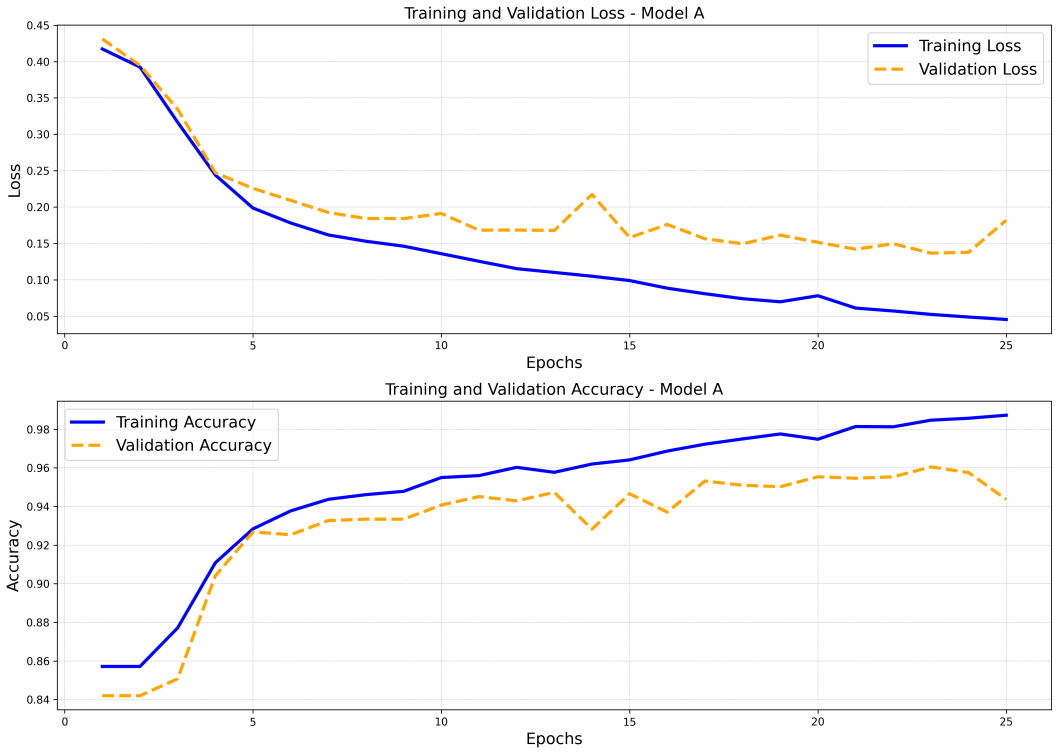
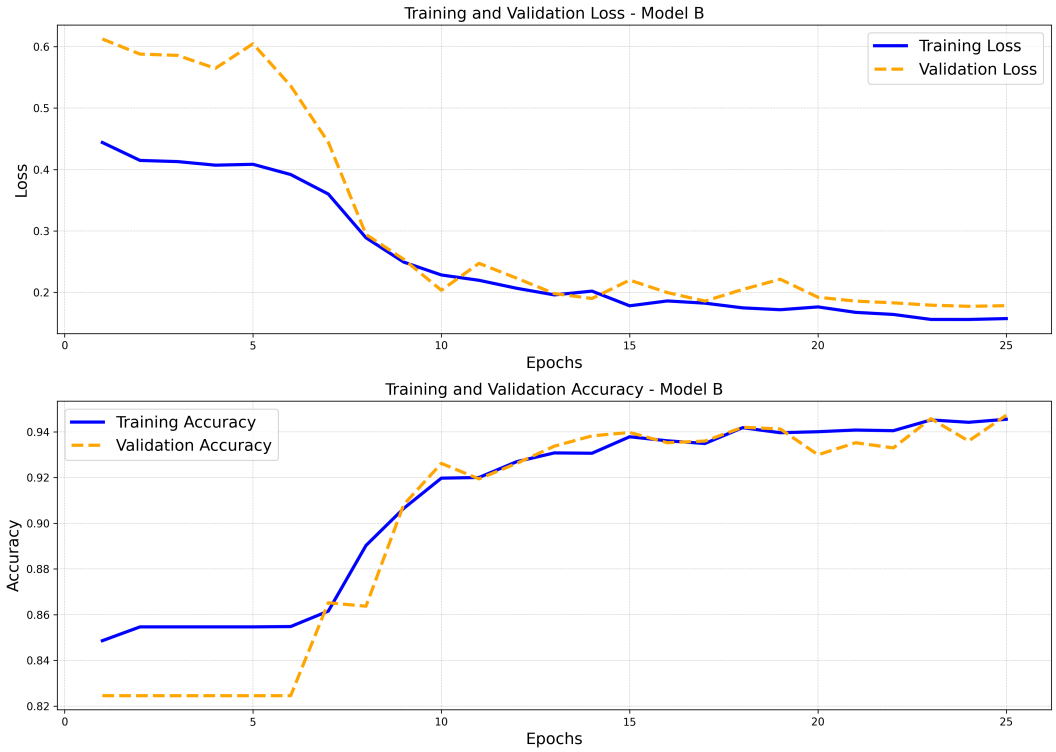


Figure 12: Confusion matrix for the optimized random forest model (`max_depth=15`), highlighting its performance in classifying the `Volcano` class.



(a) Training and validation performance of CNN Model A. The top plot represents the training and validation loss as a function of epochs. The training loss (solid blue line) steadily decreases, demonstrating effective optimization, while the validation loss (dashed orange line) stabilizes after epoch 10, indicating the absence of significant overfitting. The bottom plot shows training accuracy (solid blue line) and validation accuracy (dashed orange line).



(b) Training and validation performance of CNN Model B. In the top plot, the training loss (solid blue line) consistently decreases, indicating effective model learning, while the validation loss (dashed orange line) decreases initially and stabilizes after approximately 10 epochs with minor fluctuations. The bottom plot depicts training accuracy (solid blue line) and validation accuracy (dashed orange line).

Figure 13: Training and validation performance of CNN Models A and B, over 25 epochs.

DISCUSSION

Now, we compare our results with findings from the existing literature, particularly the benchmark study on classifying and locating Venusian volcanoes using YOLOv5 [12]. This comparison provides valuable insights into the performance of different methods, the challenges posed by limited datasets, and the impact of data augmentation techniques in machine learning applications for image detection. Figure **14** illustrates the key differences between the two object annotation formats. In the Magellan format, the center coordinates of the annotated objects are expressed in absolute pixel values, whereas in the YOLO format, these coordinates are normalized relative to the image dimensions.

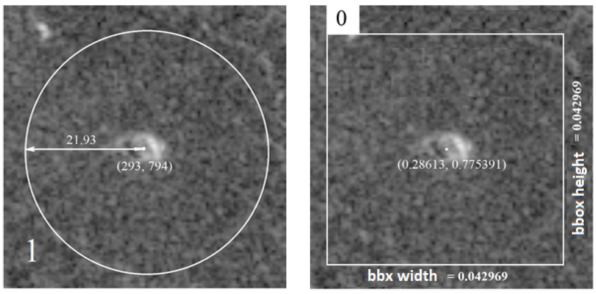


Figure 14: Difference between Magellan (left) and YOLO (right) annotations.

Our study employs a multi-model framework, including Convolutional Neural Networks, Feedforward Neural Networks, Random Forest, and Logistic Regression, to address the task of identifying volcanic formations within SAR images of Venus. CNN models, particularly Model A and the enhanced Model B, were central to our methodology. Model A, with its simple two-layer architecture followed by max-pooling and a dense classification layer, served as a baseline. In contrast, Model B introduced critical improvements such as dropout layers to mitigate overfitting and padding techniques to preserve spatial resolution during convolutional operations. The Random Forest model, implemented through an ensemble of decision trees, and Logistic Regression, optimized using stochastic gradient descent (SGD), provided alternative perspectives for classification with a focus on interpretability and computational efficiency.

In contrast, the YOLOv5 [13] study adopts a single-stage object detection paradigm where localization and classification are seamlessly integrated into a unified pipeline (Figure **15**). YOLOv5 operates by dividing the input image into a grid, predicting bounding boxes and class probabilities for each grid cell, effectively addressing both tasks simultaneously. Its architecture relies on a convolutional backbone for feature extraction, a neck for multi-scale feature fusion, and a detection head for bounding box regression and class prediction. The model's objective function includes two primary components: a regression loss L_{box} for bounding box accuracy and a classification loss L_{cls} for object categorization, which are optimized simultaneously during training. Mathematically, the total loss can be expressed as:

$$L_{\text{total}} = \lambda_{\text{box}} L_{\text{box}} + \lambda_{\text{cls}} L_{\text{cls}} + \lambda_{\text{obj}} L_{\text{obj}}, \tag{1}$$

where L_{obj} penalizes the confidence score for incorrect object predictions, and λ values are weight parameters balancing these terms.

One fundamental divergence between the two studies lies in their treatment of the dataset. Our study addresses data limitations through Principal Component Analysis (PCA) for dimensionality reduction and Synthetic Minority Oversampling Technique (SMOTE) to combat class imbalance. PCA reduces the high-dimensional pixel space into a lower-dimensional subspace while preserving 95% of the variance, significantly mitigating overfitting risks in models trained on relatively small datasets. SMOTE, on the other hand, synthesizes new examples of minority-class instances to ensure balanced learning, particularly for volcanic images underrepresented in the dataset. These strategies improve the generalizability of the models but do not fundamentally alter the number or diversity of input images.

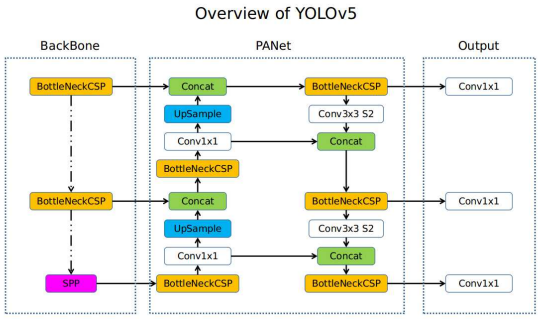


Figure 15: Overview of model structure about YOLOv5. GitHub

In contrast, the YOLOv5 study emphasizes data augmentation as a critical strategy to enhance performance. While classical augmentations such as rotation, mirroring, and contrast adjustments contribute to modest improvements, the introduction of mosaic augmentation proves transformative. The mosaic technique generates composite images by combining multiple input samples into a single image (Figure **16**), effectively increasing both dataset size and variability. By exposing the model to a wider range of spatial contexts, mosaic augmentation mitigates overfitting and enhances generalization. The results underscore its effectiveness, as the mean Average Precision (mAP) improves dramatically from an initial 28.4% to 83.5% following the application of advanced augmentation strategies. This highlights the importance of dataset diversity in deep learning models, particularly those requiring large-scale inputs for optimal training.

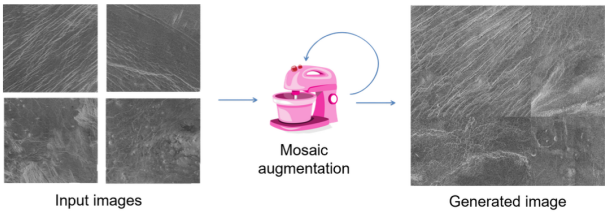


Figure 16: Concept of mosaic augmentation technique.

The performance metrics further delineate the comparative advantages and limitations of each approach. In our study, CNN Model B demonstrates the best results among the tested models due to the introduction of dropout layers, which regularize the network and reduce overfitting. However, despite the inclusion of SMOTE and PCA, the overall classification performance remains sensitive to the dataset's limited size and heterogeneity. Logistic Regression and Random Forest models, though more interpretable, lack the capability to capture complex spatial hierarchies inherent in volcanic images, limiting their accuracy relative to CNNs.

YOLOv5, by contrast, benefits from its unified detection framework, which inherently accounts for both localization and classification tasks. The single-pass detection strategy enables the model to process images more efficiently than two-stage frameworks, reducing computational overhead while maintaining high accuracy. However, YOLOv5's reliance on data-intensive training presents a challenge for small datasets such as the Magellan SAR set. Without augmentation, the model initially suffers from underfitting, as evidenced by low mAP scores. This observation aligns with theoretical expectations, as deep learning models require large-scale datasets to approximate the true underlying data distribution effectively. The model's improvement following augmentation further reinforces the relationship between dataset size, diversity, and generalization capability.

From a computational perspective, the trade-offs between our approach and YOLOv5 are noteworthy. Our framework, relying on PCA and SMOTE, is computationally efficient and scalable for small datasets but lacks the end-to-end optimization of YOLOv5. CNNs, while effective for classification, do not inherently address object localization, requiring additional methodologies for bounding box prediction. YOLOv5, on the other hand, excels in tasks requiring simultaneous detection and classification, making it a superior choice for real-time and large-scale applications despite its higher computational demands.

YOLOv5 achieves superior accuracy by leveraging advanced augmentation techniques to compensate for data limitations, while our approach emphasizes preprocessing, dimensionality reduction, and class balancing to enhance model performance. The results underscore the critical importance of dataset size, augmentation diversity, and methodological alignment with task requirements in achieving high accuracy for image-based classification and detection tasks.

CONCLUSION

This study investigated different machine learning methods, focusing on the use of convolutional neural networks (CNNs) to classify volcanic formations in SAR images of Venus. The results show that the 'Model B' of CNNs achieved more robust generalisation by introducing techniques such as dropout and padding. However, the limitations of the dataset, both in terms of size and heterogeneity, negatively affected the overall performance. This suggests that advanced approaches such as data augmentation could further improve the results.

Despite the demonstrated effectiveness in classifying SAR images, the tested models showed some limitations in terms of accuracy compared to more advanced frameworks such as YOLOv5 trained on large-scale datasets like COCO, which uses innovative augmentation techniques such as mosaicking. The results highlight the importance of combining traditional preprocessing approaches such as PCA and SMOTE with more sophisticated augmentation strategies to address complex tasks such as simultaneous object detection and classification.

The CNN-based approach provides a solid starting point for more advanced analysis. To improve performance, future studies could include the use of more diverse datasets, transfer learning techniques and integration with specific object detection models. In addition, the adoption of optimisation strategies for CNN architectures, such as the search for hyperparameters or the use of pre-trained models, could significantly increase the ability to generalise to new data.

References

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

[2] M. Hjorth-Jensen. *MachineLearning*. https://github.com/CompPhysics/MachineLearning. 2024.

[3] P. A. Lopez Torres G. Durante E. Ottoboni. *Project1*. https://github.com/Elisaottoboni/Machine-Learning-University-of-Oslo/tree/main/Project2. 2024.

[4] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[5] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] François Chollet et al. *Keras*. https://keras.io. 2015.

[7] S. Raschka et al. *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Expert insight. Packt Publishing, 2022. ISBN: 9781801819312. URL: https://books.google.no/books?id=UHbNzgEACAAJ.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Chapters 2-14 are highly recommended; lectures follow this text to a large extent. MIT Press, 2016. URL: https://www.deeplearningbook.org/.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Main textbook covering chapters 1-7, 11, and 12. Available for free PDF download at https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf. Springer, 2006. URL: https://www.springer.com/gp/book/9780387310732.

[10] Aurélien Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. Contains many code examples and hands-on applications of algorithms discussed in the course. O'Reilly Media, 2019. URL: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/.

[11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[12] Daniel Đuranović et al. "Localization and Classification of Venusian Volcanoes Using Image Detection Algorithms." In: *Sensors* 23.3 (2023). ISSN: 1424-8220. DOI: 10.3390/s23031224. URL: https://www.mdpi.com/1424-8220/23/3/1224.

[13] Glenn Jocher. *YOLOv5 by Ultralytics*. Version 7.0. 2020. DOI: 10.5281/zenodo.3908559. URL: https://github.com/ultralytics/yolov5.

[14] P. A. Lopez Torres G. Durante E. Ottoboni. *Project3*. https://github.com/Elisaottoboni/Machine-Learning-University-of-Oslo/tree/main/Project3. 2024.