*minerals*

MDPI

*Article*

# Enhancing Deep Learning and Computer Image Analysis in Petrography through Artificial Self-Awareness Mechanisms

**Paolo Dell'Aversana**

Eni S.p.A., San Donato Milanese, 20097 Milan, Italy; paolo.dell'aversana@eni.com

**Abstract:** In this paper, we discuss the implementation of artificial self-awareness mechanisms and self-reflection abilities in deep neural networks. While the current limitations of research prevent achieving cognitive capabilities on par with natural biological entities, the incorporation of basic self-awareness and self-reflection mechanisms in deep learning architectures offers substantial advantages in tackling specific problems across various scientific fields, including geosciences. In the first section, we outline the foundational architecture of our deep learning approach termed Self-Aware Learning (SAL). The subsequent part of the paper highlights the practical benefits of this machine learning methodology through synthetic tests and applications addressed to automatic classification and image analysis of real petrological data sets. We show how Self-Aware Learning allows enhanced accuracy, reduced overfitting problems, and improved performances compared to other existing methods.

**Keywords:** deep learning; self-awareness; adaptive learning; classification; petrography



**Citation:** Dell'Aversana, P. Enhancing Deep Learning and Computer Image Analysis in Petrography through Artificial Self-Awareness Mechanisms. *Minerals* **2024**, *14*, 247. https://doi.org/10.3390/min14030247

Academic Editors: Wenlei Wang, Shuyun Xie and Zhijun Chen

Received: 23 January 2024
Revised: 19 February 2024
Accepted: 26 February 2024
Published: 28 February 2024



## 1. Introduction

In philosophy of mind and in neurosciences, self-awareness refers to the capacity of an individual to introspectively perceive and recognize her/his own existence, thoughts, feelings, and experiences as separate from the surrounding world and from other individuals [1–5]. It encompasses a range of aspects, including self-perception, self-identity, self-consciousness, and self-reflection. It allows individuals to have a sense of their own individuality and to recognize themselves as distinct entities with their own thoughts, emotions, and perspectives. Self-awareness also involves being aware of how one's actions, behaviors, and decisions may affect oneself and others. Inspired by the above neuroscientific concepts, in this paper, we discuss how to implement a rudimental form of self-awareness and self-reflection in artificial neural networks. We emphasize the term "rudimental" because, at the present status of this research, to our knowledge, it is not possible to develop any self-awareness in artificial agents that is minimally comparable with natural awareness in biological entities. However, although we recognize that limitation, we start from the assumption that enhancing deep learning architecture with rudimentary self-awareness and self-reflection abilities can bring important benefits in solving specific technical problems. Several studies and applications in artificial intelligence and robotics support this intriguing idea [6–17]. The following are some of the possible expected advantages motivating the present research.

Improved Adaptability: By incorporating basic self-awareness abilities, artificial neural networks can autonomously (without any direct human intervention) adapt the hyperparameters of their codes and architectures to dynamic conditions (in the AI context) and evolving data sets. This is relevant, for instance, in Earth disciplines as well as in medical sciences and in many other areas, where data patterns and features are often unbalanced and may vary over time. In principle, self-awareness enables the network to monitor its own performance and adjust its behavior accordingly.

Enhanced Error Detection and Handling: Neural networks with basic self-awareness and self-reflection mechanisms can detect and handle own errors more effectively. They can

evaluate the confidence or uncertainty of their predictions and identify situations where their outputs may be unreliable. For instance, in Earth science applications, this capability is crucial for critical decision-making processes.

Greater Robustness: Self-awareness, even if implemented through rudimentary forms, allows neural networks to recognize their own limitations and respond appropriately. For example, in Earth disciplines, where data can be noisy, incomplete or unbalanced, having a network that can autonomously evaluate the quality and reliability of its results can lead to more robust and accurate outcomes, including classification and prediction of complex data sets in both terms of data series and images, as well as other mixed information. This is particularly relevant in a variety of fields such as mineralogy and petrography images analysis; geological/geochemical/hydrocarbon exploration; hydrology; gas/water/geothermal reservoir characterization; weather, environmental, seismic and volcanic hazards forecasting.

Efficient Computing Resource Allocation: Neural networks with basic self-awareness capabilities can optimize the allocation of computational resources. They can dynamically assess the complexity of a task, allocate resources accordingly, and prioritize the most important computations. This can lead to significant efficiency gains in computationally demanding Earth science and/or medical applications, such as large-scale simulations, multi-physics geophysical joint inversion, geospatial data analysis, climate modeling, biological data fusion, and advanced imaging (combining data from different experimental techniques, such as genomics, proteomics, metabolomics, and high-resolution imaging).

Explainability: Self-awareness mechanisms can provide insights into the decision-making process of neural networks, making their outputs more explainable and interpretable. This is critical in almost all the scientific and financial domains, where scientists, analysts, stakeholders and managers need to understand the underlying factors, reasoning, and bias and error propagation behind predictions or classifications. Self-awareness can enhance the transparency and trustworthiness of the models, enabling better collaboration and informed decision-making.

In summary, by introducing rudimentary self-awareness and self-reflection mechanisms in artificial neural networks, we can leverage these benefits to tackle complex problems in many fields of industrial, medical, financial, and academic interest, including Earth disciplines, health sciences, business sectors, and so on. Having these goals in mind, in this paper, we introduce a novel approach for incorporating some basic self-awareness and self-reflection mechanisms into artificial neural networks, improving their performance and adaptability in specific tasks. In the following, this type of Self-Aware artificial network is indicated as SA-net, and the approach is indicated as Self-Aware Learning, or briefly, SAL.

In the next sections, we introduce the basic methodological aspects of the SAL approach and explain how to implement its architecture and its workflow. Next, we show how SAL can solve classification and prediction problems through tests on synthetic data. Finally, we will apply the SAL approach to real petrological data sets, discussing its benefits and limitations.

## 2. SAL Methodology, Adaptive Learning, and Self-Monitoring

Adaptive learning is an educational method that utilizes computer algorithms and artificial intelligence to enhance learner engagement and provide personalized learning materials and activities tailored to meet the distinct requirements of individual learners. This learner can be a human or an artificial agent. If we consider an artificial learner represented by a neural network model (see Appendix B for details about the key terminology related to Artificial Neural Network Models), we propose the following comprehensive iterative formula for expressing the key idea of artificial adaptive learning:

$$\theta(t+1;\Omega) = \theta(t;\Omega) - \alpha \cdot \nabla L\left(\theta(t;\Omega)\right) \quad (1)$$

In this formula, $\theta(t;\Omega)$ represents the network model's parameters and its architecture (hyper-parameters) at time step $t$, $\alpha$ denotes the learning rate, and $\nabla L\left(\theta(t;\Omega)\right)$ denotes

the gradient of the loss function with respect to the network parameters and its hyper-parameters (defining its architecture). The parameters of a "standard" neural network (outside the SAL paradigm) correspond typically to the weights of the connections, and they are learned during the training stage (see Appendix B for details about the terminology used in this section). Instead, in the SAL methodology, $\theta(t;\ \Omega)$ include hyper-parameters too (the key architectural elements of the network itself), here denoted with the generic symbol $\Omega$. These hyper-parameters are typically the batch size, the number of epochs, the number of hidden layers, the number of neurons for each layer, and other architectural elements (Appendix B). They influence the performances of the network and how its parameters (connection weights) will be learned. Commonly, we optimize these hyper-parameters through techniques like grid search or random search, as well as by trial and error, through direct human intervention. In the SAL approach, parameters, hyper-parameters and the entire architecture are learned through self-monitoring and self-adaptive learning. The network model updates its entire functional structure autonomously (in the sense clarified below) by subtracting the scaled gradient, allowing it to adapt its internal representations based on the feedback from the loss function. We recall that, in a generic deep neural network, the loss function is a mathematical function that quantifies the discrepancy between the predicted output and the true output. It measures the error, or "loss", of the model's predictions, indicating how well or poorly the network is performing. One commonly used loss function is the mean squared error (MSE). Another commonly used loss function in deep neural networks is the cross-entropy loss function (see Appendix B). It is often employed in classification tasks where the goal is to assign input data to different categories or classes. The general expression of the cross-entropy loss function for a multi-class classification problem is as follows:

$$L = -\sum_{i=1}^{N}[y_i \cdot log(p_i)] \tag{2}$$

where:

$L$ represents the cross-entropy loss;
$\sum$ denotes the summation over all $N$ classes;
$y_i$ is the true label of class $i$ (0 or 1 depending on whether the sample belongs to that class or not); and
$p_i$ is the predicted probability of class $i$.

In the SAL approach, the learning rate $\alpha$ controls the step size of both the parameter and hyper-parameter updates. A higher learning rate allows for larger adjustments, while a lower learning rate ensures smaller, more cautious updates. Hence, the loss function also depends on $\Omega$.

The gradient $\nabla L\ (\theta(t;\ \Omega))$ is computed using techniques such as backpropagation, which calculates the derivative of the loss function with respect to each parameter and hyper-parameter in the model. It provides information on how the loss function changes as the entire network structure/architecture vary, enabling the model to adjust its internal representations accordingly.

In the context of the SAL approach, both self-adaptive learning and self-monitoring mechanisms play a crucial role in evaluating the performance of the network model at each time step. The self-monitoring function, denoted as $M()$, takes the model's output $O(t)$ as input and computes a performance measure, which provides an assessment of how well the model is performing on a specific task. Formally, we can express this as:

$$P(t;\ \Omega) = M\ (O(t;\ \Omega)) \tag{3}$$

Here, $P(t;\ \Omega)$ represents the performance measure at time step $t$ (epoch), for a given parameter and hyper-parameter set, $\Omega$. The performance measure can vary depending on the task and the desired evaluation criteria. For example, in classification tasks, the

performance measure could be training and/or validation accuracy, which indicates the proportion of correctly classified/predicted samples. In regression tasks, the performance measure could be the mean squared error, which quantifies the average squared difference between the predicted and target values. The self-monitoring function $M\ (O(t;\ \Omega))$ encapsulates the calculation of the performance measure. It takes the model's output $O(t;\ \Omega)$ as its input and computes the desired performance metric.

By incorporating self-monitoring abilities in the network, the model gains the ability to track its own performance over time. We will discuss these mechanisms later on, in a specific section dedicated to "self-reflection mechanisms". The performance measure $P(t;\ \Omega)$ provides feedback to the model, enabling it to assess its current capabilities and potentially trigger adaptive changes in its learning process based on the performance feedback. These adaptive changes can include adjusting the learning rate, modifying the model's hyper-parameters and architecture, or updating the training data distribution.

Overall, self-monitoring mechanisms allow the model to evaluate its own performance, providing valuable information for decision-making processes in adaptive learning. By continuously monitoring its performance (through the epochs), the model can adapt its learning process, parameters and hyper-parameters to improve its performance.

## 3. Architecture, Workflow and Functionalities

In the following three sections, we introduce, respectively, the main architectural aspects and the key steps of the SAL workflow, the crucial functionalities of the SAL network and, finally, the concept of "self-reflection mechanisms" applied to deep neural networks (see the Appendix A, for a didactical Python block about the implementation of self-reflection model updating).

### *3.1. Architecture and Workflow*

Figure 1 is a block diagram showing the workflow's key steps of SAL. On the left side of the scheme, we see that "standard" networks, such as Convolutional or Residual networks [18,19], initially process the input data, depending on specific tasks. These networks generate a preliminary output, which can be the classification of image data sets or the prediction of time-series data, for example. Throughout and after the network performance, all the key hyper-parameters are analyzed using dedicated self-aware mechanisms (described below). The goal is to autonomously evaluate the network's performance in real-time, without external intervention from the user, and update the network's architecture to improve the results. This means maximizing all the accuracy criteria and, at the same time, minimizing the validation loss function, limiting overfitting effects. The performance is iteratively compared to the results obtained in the previous step, and the hyper-parameters are adjusted accordingly (see Equation (1)). This loop continues until one or more of the following stopping criteria are met:

- Maximum number of iterations: we define in advance a maximum number of iterations to prevent the algorithm from running indefinitely. Once this limit is reached (fixed empirically), the training process can be stopped.
- Convergence of hyper-parameters: the network self-monitors the convergence of the hyper-parameters by tracking their changes between iterations. If the changes fall below a predefined threshold, it can be an indication that the network has reached a stable configuration (for instance, we can set a threshold based on the relative change in hyper-parameter values between iterations).
- Performance improvement: the network self-monitors the performance metric of interest (e.g., accuracy or loss) on a validation set or during cross-validation. If the performance metric does not show significant improvement over a certain number of iterations, it may indicate that further updates to the hyper-parameters are unlikely to yield significant benefits.
- Resource constraints: if the training process exceeds these pre-set timing constraints without substantial improvements in performance, it automatically stops the loop.

- Early stopping: an early stopping mechanism based on a predefined criterion is set in advance, such as the performance on a validation set. If the performance does not improve or starts to deteriorate after a certain number of iterations, the training can be stopped early to avoid overfitting or wasting computational resources.

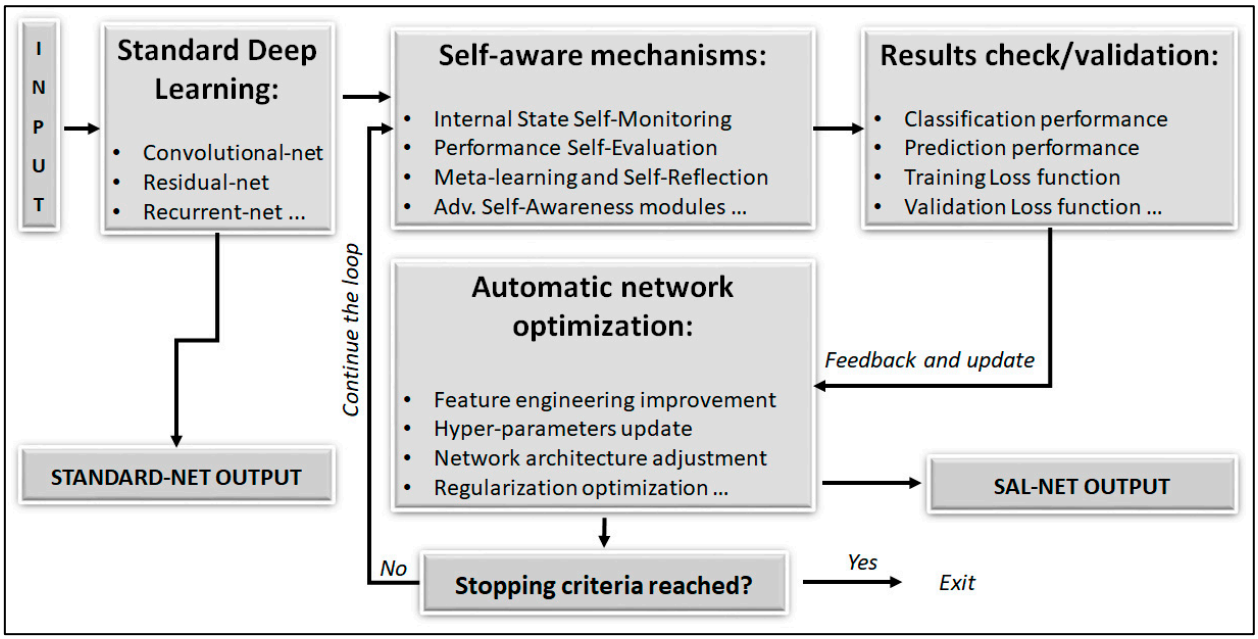


**Figure 1.** Scheme of the SAL architecture and workflow (for terminology and additional details, see Appendix B).

All of these stopping criteria are tailored to the specific Self-Awareness network and the problem being addressed. It is important to balance the desire for further optimization with the practical limitations and the risk of overfitting.

### *3.2. SAL Functionalities*

The following are the basic functions of SAL:

- Internal State Monitoring: Mechanisms within the neural network architecture that monitor its internal state, including neuron activations and flow of information through the neural layers that define the network architecture. This will provide the network with a basic control of its own activity, through continuous/autonomous analysis of the fundamental hyper-parameters.
- Performance Self-Evaluation: Mechanisms within the neural network architecture that enable the neural network itself to evaluate its own performance and recognize errors. These are techniques such as loss prediction and confidence estimation that assess the network's uncertainties in its predictions and that identify possible convergence problems (for additional details, see Appendix B).
- Metacognition: Integrated metacognitive mechanisms, allowing the network to assess its own knowledge and monitor the learning process. This will help the network identify gaps in its knowledge and make more informed decisions. In other words, one of the key functions of metacognition in SAL is to identify gaps or deficiencies in the network's knowledge. This involves recognizing situations where the network is uncertain or where its predictions are unreliable. By identifying knowledge gaps, the network can prioritize learning in those areas or seek additional data or training to improve its understanding. An important aspect of metacognition is self-reflection. We are going to discuss this in detail in a dedicated section below.
- Continual Adaptation and Learning: The neural network is designed to support continual adaptation to new situations and changes in the environment. This may in-

volve architecture updates (here named "artificial plasticity mechanisms") to facilitate ongoing learning over time.

- Advanced Neural Functions: These include, for instance, "attention mechanisms". Dedicated modules enable the network to focus attention on specific aspects of its internal state or the surrounding environment. Other functions are pre-processing self-optimization, automatic features extraction, automatic features ranking, and optimal data normalization. Pre-processing involves preparing the input data before feeding them into the neural network. This can include tasks such as resizing images to a standard size, normalizing pixel values, or applying data augmentation techniques to increase the diversity of training examples. For example, in a computer vision task where the neural network is trained to classify images of handwritten digits, pre-processing might involve resizing all input images to a fixed size (e.g., $28 \times 28$ pixels) and normalizing pixel values to lie within a certain range (e.g., zero to one).

### *3.3. Self-Reflection*

A crucial meta-cognitive aspect of SAL architecture is the possibility to implement rudimental mechanisms of Self-reflection in deep neural networks. Self-reflection mechanisms, also known as self-supervision or self-training, are techniques employed in neural networks to enhance their learning and performance. They involve introducing additional tasks or objectives to the network's training process, aiming to improve its representation learning capabilities and generalization abilities. Compared to standard neural network functions, self-reflection mechanisms offer several benefits, such as Auxiliary Classifiers, Adaptive Loss Weights, and Adaptive Gradient-Based Regularization. The Auxiliary Classifiers are additional neural networks that help improve the training of the self-reflection model. These auxiliary classifiers are typically inserted at intermediate neural layers of the main network architecture. Each auxiliary classifier learns to predict the same categories as the main classifier but from a different representation of the input data. This approach can lead to improved generalization performance and robustness, especially in complex classification tasks.

The Adaptive Loss Weights allow the self-reflection model to assign different weights to the losses from the main classifier and the auxiliary classifiers. Adaptive Gradient-Based Regularization (AGBR) combines the principles of gradient-based optimization and adaptive regularization. It dynamically adjusts the regularization strength during the training process based on the gradient information of the model's parameters. It aims to strike a balance between reducing the model's complexity and preserving important patterns in the data. By adaptively modifying the regularization strength, AGBR helps prevent overfitting by effectively controlling the model's capacity and avoiding excessive reliance on noisy or irrelevant features.

Dropout and Momentum are two additional important hyper-parameters that are iteratively adjusted during the SAL workflow. Dropout is a further regularization technique that helps prevent overfitting in neural networks. It works by creating an ensemble of smaller subnetworks within the main network, making it harder for the network to memorize the training data. During testing or inference, no units are dropped out, and the entire network is used for prediction. The Dropout rate is an adjustable hyper-parameter that determines the fraction of units that drop out during training. Momentum is a technique used in optimization algorithms, such as Stochastic Gradient Descent (SGD), to speed up convergence and overcome local minima. In the context of SAL networks, Momentum is a hyper-parameter that controls the update of network weights during training. It adds a fraction of the previous weight update to the current update, allowing the optimizer to maintain a certain velocity or momentum in the weight updates. Momentum helps the optimizer escape from sharp local minima and converge to global solutions. The Momentum value is a hyper-parameter that determines the fraction of the previous weight update to add to the current update.

## 4. A Synthetic Test

In order to show SAL effectiveness, we compared, through synthetic tests, the performances of the SAL approach with a "standard" Deep Neural Network (here briefly indicated as ST-net) based on two or more hidden layers. This was built by using the well-known "Keras" Python library. In the first test, the first hidden layer consists of a fully connected (dense) layer with 16 units (neurons). It applies a rectified linear unit (ReLU) activation function (see Appendix B) to introduce non-linearity in the network. The input to this layer is the output from the previous layer (the input layer). The second hidden layer is another fully connected layer with 8 units and ReLU activation. It takes the output from the previous layer as input. Finally, the output layer represents the final layer of the network; it is responsible for producing the output predictions. It is also a fully connected layer, but the number of units is determined by the number of output classes or categories. The activation function used in this standard model is Softmax, which normalizes the outputs into probabilities, allowing the model to predict the class probabilities for each input sample. The "Sequential container" from the "Keras" library is used to stack these layers in a sequential manner, creating the "standard" neural network model.

This type of test is aimed at classifying synthetically generated data (created through computer simulation) in two or more classes based on five features (this is just an arbitrary choice to make the test relatively simple), and to compare the accuracy of the results with that of those obtained through the SAL approach. The challenge, in this specific case, consists in the fact that the data show a strong overlap in the feature space, as shown in Figure 2.

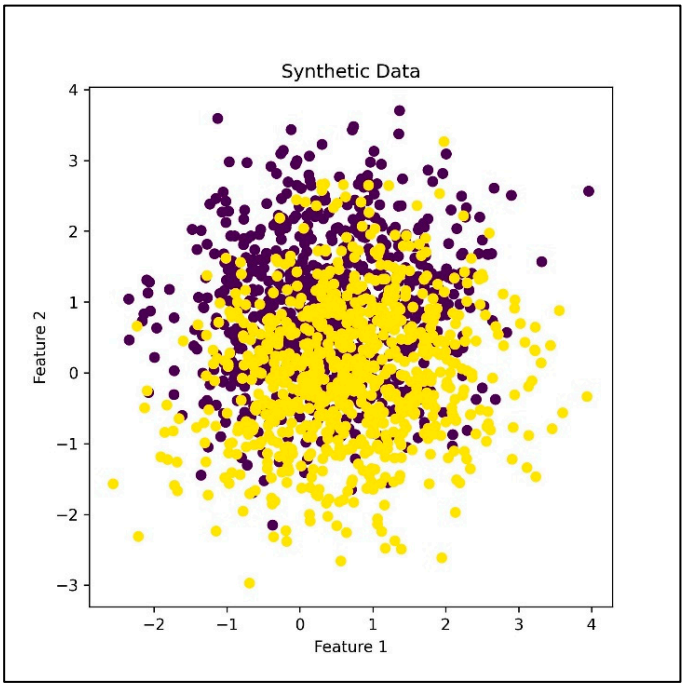


**Figure 2.** Example of synthetic (simulated) data distribution in the feature space, in this case determined by the first two features (purple and yellow indicate the two different classes).

Following the scheme of Figure 1, we implemented the SAL architecture (here indicated as Self-Reflection Model) by introducing self-reflection mechanisms in the workflow. We used a genetic algorithm for iteratively (and without human intervention) updating such self-reflection mechanisms. Our (proprietary) code generates a population of candidate network architectures, trains and evaluates them, selects the best-performing models, and performs mutations to create the next generation of hyper-parameters. The process is repeated for a specified number of generations (epochs). The best architecture found during the process is continuously evaluated on the test set. The code then plots the

training accuracy and loss, as well as the validation accuracy and loss, for both the best SAL architecture found and the Standard Model.

The Self-Reflection Model starts with just one or two hidden layer(s) and 16 neurons (this is an arbitrary choice to start with a simple network architecture). After each epoch, the model evaluates the training and validation accuracy and the training and validation loss. At each iteration, the model autonomously updates its hyper-parameters (number of hidden layers, neurons, connection weights, regularization, Dropout, Momentum and so forth) and continues training.

This updated code aims to improve the performance of the SAL model by adapting its architecture and hyper-parameters during training based on the self-reflection mechanisms.

Figure 3 shows the comparison between the Standard Model and the Self-Reflection Model. Looking at the training accuracy plots (top graphs), the Standard Model shows a higher training accuracy trend and a lower training loss than the Self-Reflection Model. This means that the training process seems to be more effective for the Standard Model than the Self-Reflection Model. However, when comparing the performances of the two models on the validation data set (bottom graphs), we can see that the standard network is affected by strong overfitting for increasing iterations (epochs). Instead, we do not see these overfitting effects in the case of the Self-Reflection Model. Furthermore, the validation accuracy of the Standard Model is generally lower than the Self-Reflection Model accuracy. We remark that validation accuracy and loss trends are measures of how well the model generalizes to new data. Validation accuracy represents the percentage of correctly predicted samples in the validation set. A stable and high validation accuracy suggests that the model has learned meaningful patterns and is able to make accurate predictions about unseen data.

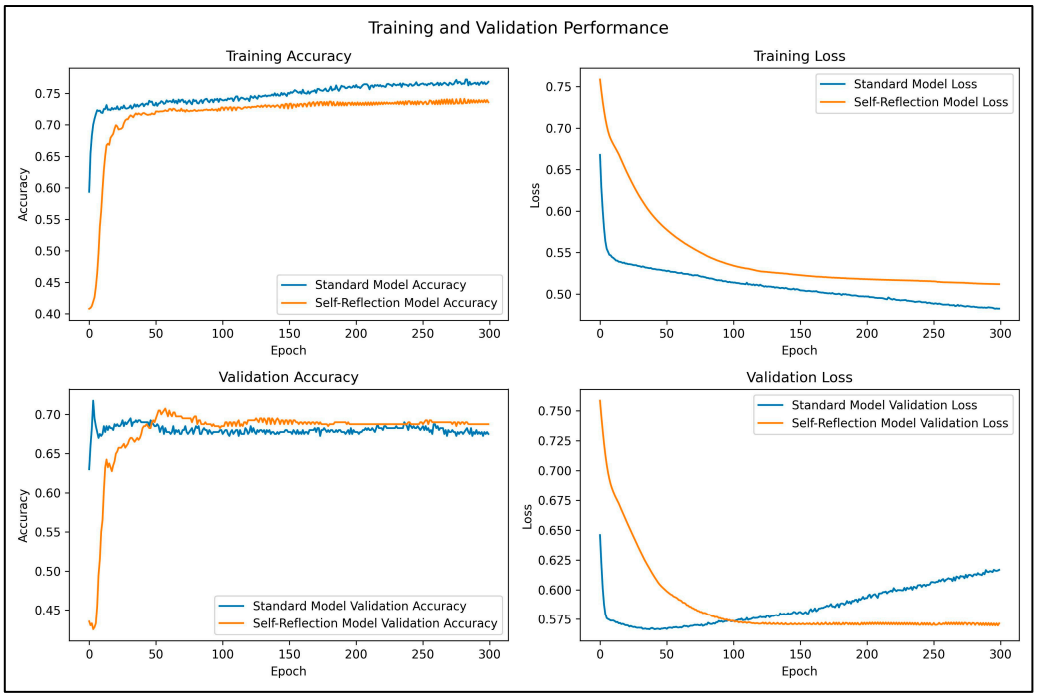


**Figure 3.** Training (**upper panels**) and validation (**lower panels**) accuracy and loss comparison between a Standard Model (blue curves) and a Self-Reflection Model (orange curves).

In other words, the Standard Model is well trained, but shows poor generalization abilities due to clear overfitting problems. In contrast, the Self-Reflection Model shows a more stable trend on validation data and small or null overfitting effects, demonstrating higher generalization capabilities. This is one of the benefits provided by the SAL approach, which incorporates self-reflection mechanisms.

## 5. Test on Real Petrological Data

In this section, we present an application aimed at classifying a data set of about 1500 rock samples, based on 10 chemical features (major oxides) and 8 rock classes, totaling about 15,000 instances. The following is the list of the oxides considered in this application: $SiO_2$, $TiO_2$, $Al_2O_3$, $Fe_2O_3$, MnO, MgO, CaO, $Na_2O$, $K_2O$, and $P_2O_5$ (in weight %).

We have used the public data set available on the GEOROC (Geochemistry of Rocks of the Oceans and Continents) available on the website (http://georoc.mpch-mainz.gwdg.de/georoc/, accessed on 15 December 2023). In particular, we have (partially) used the files available at the following link: https://georoc.mpch-mainz.gwdg.de/georoc/webseite/Expert_Datasets.htm, accessed on 15 December 2023. The samples have been analyzed through in-situ geochemistry techniques.

Among these classes, some rock types are predominant, like Andesite, Basaltic Andesite, Rhyolite, and Dacite. Additionally, there are some "minor" classes with significantly fewer samples. The chemical/mineralogical compositions of these classes partially overlap, as depicted in Figure 4.

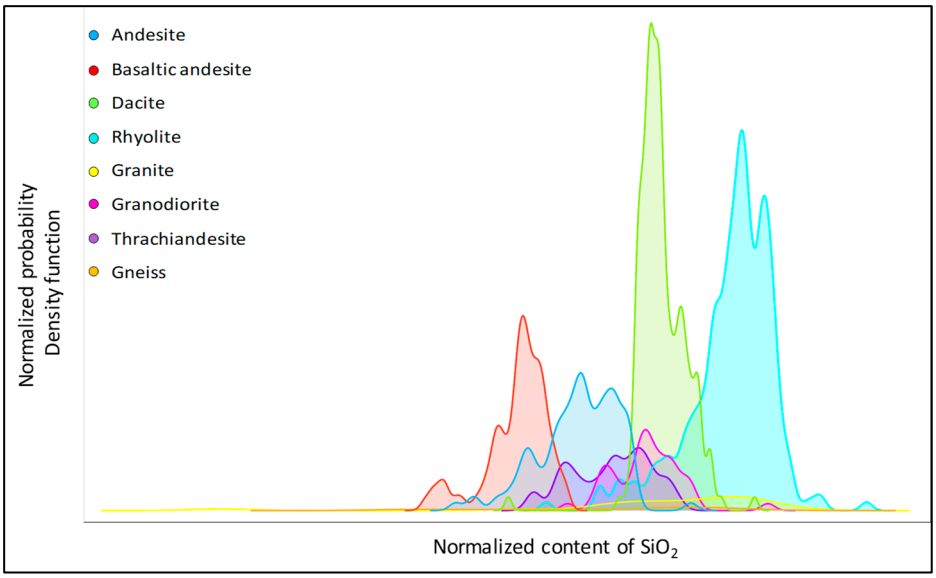


**Figure 4.** Example of rock class distribution based, in this case, on normalized content of $SiO_2$.

Such an evident overlap in the feature space is conceptually similar to the overlap shown in the synthetic test discussed in the previous section, clarifying better the sense and the motivations of the test itself. In other words, our goal is to show how the SAL approach can be more effective than standard machine learning methodologies for classifying complex data sets that show significant overlap in the feature space, as happens in the case of different rock typologies.

The limited number of samples belonging to classes other than Rhyolite, Dacite, Andesite, and Basaltic Andesite create a sort of "background noise" that negatively affects the classification process. In other words, these samples have a chemical composition partially similar to one or more of the four/five major classes, which makes classification more challenging. We applied a SAL model for classifying the rock types using all the available chemical features. We started with a very simple deep neural network model including just two hidden layers with 50 neurons for each layer. Then, using the SAL iterative approach, the algorithm autonomously updated the network architecture by increasing the number of hidden layers and the number of neurons for each layer. Moreover, the model iteratively re-adapted the other hyper-parameters (learning rate, number of iterations, regularization factor, type of solver, and so forth) with the final aim of increasing classification accuracy and other performances indexes. The updated SAL model included five hidden layers,

with a number of neurons ranging between 300 and 500, a low regularization factor ($\alpha$, ranging between 0.0005 and 0.001) an ADAM solver, and a RELU activation function.

Finally, we compared the performances of the updated SAL model with those of other classification methods based on different machine-learning algorithms. Table 1 shows the classification accuracy and the precision values for the SAL model, Decision Tree, Random Forest, Naïve Bayes, Logistic Regression, CN2 Rule Inducer and Adaptive Boosting algorithms, all trained on the same labelled data set (about 20% of the total data samples).

**Table 1.** Comparison of performance indexes of SAL Neural Network and other machine learning algorithms (see Appendix B for a brief explanation of these alternative algorithms).

| Model | Classification Accuracy | Precision |
|---|---|---|
| **SAL Neural Network** | **0.705** | **0.735** |
| Decision Tree | 0.579 | 0.529 |
| Random Forest | 0.684 | 0.691 |
| Naive Bayes | 0.56 | 0.583 |
| Logistic Regression | 0.613 | 0.546 |
| CN2 Rule Inducer | 0.421 | 0.28 |
| Adaptive Boosting | 0.664 | 0.672 |

Looking at Table 1, we can see that the SAL model shows a generally higher performance than the other approaches. Finally, it achieves satisfactory classification results by separating the main classes into distinct feature areas, with only partial and unavoidable overlap (Figure 5). Specifically, there is some expectable overlap between Basaltic Andesite, Trachyandesite, and Andesite rocks.

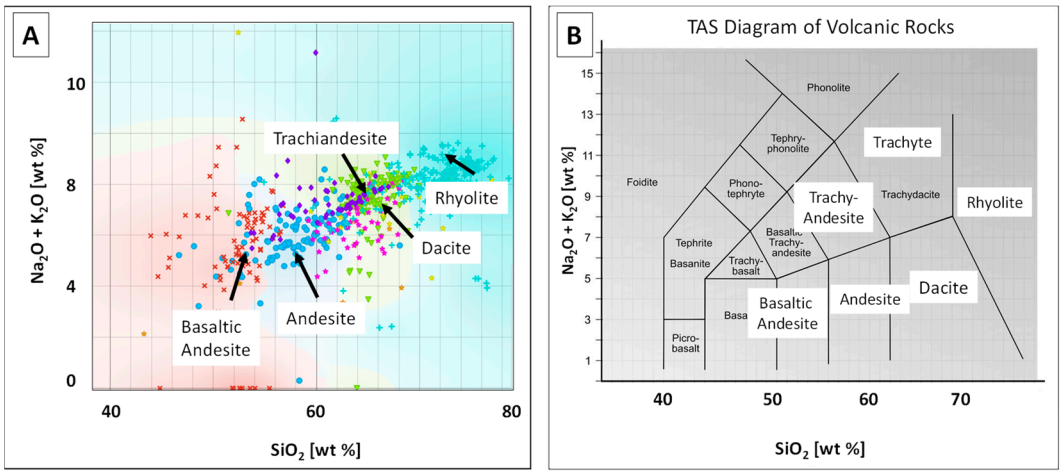


**Figure 5.** Classification example through SAL model network, using Total Alkali vs. Silica as diagnostic features. Panel (**A**) indicates the classification result (using different colors for the different rock classes), while panel (**B**) is the TAS (Total Alkali-Silica) Diagram.

We remark that Figure 5 illustrates the plot generated through the SAL classifier, using the sum of $Na_2O$ and $K_2O$ oxides vs. $SiO_2$ (weight %) as examples of diagnostic features. However, the classification process utilizes all ten chemical features available in the database.

## 6. Image Analysis and Classification Using SAL

In previous works [18], we have described the entire workflow of different types of deep learning methodologies for classifying mineralogical thin sections. Now, we apply our SAL approach for classifying an experimental set of thin section images representing

various types of magmatic rocks. The images of the samples analyzed in this test come from a different database with respect to the test discussed in the previous section. Specific references are detailed in the "data availability" section at the end of the paper. For our test, we selected both types of images at NX as well as N//; magnification: 2× (long side = 7 mm).

In this case, our methodology adds image pre-processing and image embedding method optimization to the self-reflection optimization workflow described in the previous sections. In this test, as in the previously described applications, the key innovation lies in the incorporation of self-reflection mechanisms within the deep neural network architecture to enhance the model's interpretability and classification process. The following are the main steps involved in the workflow.

First, we created a labeled data set containing images and corresponding labels for training and evaluation. Second, we explored and analyzed the data set to understand its characteristics. The data set consists of several hundred images of thin sections of four types of rocks, stored as low-resolution jpeg files. Then, we applied various data pre-processing techniques like resizing, normalization, and augmentation to improve the model's generalization. Resizing involves adjusting the dimensions of the input images to a standard size. This ensures uniformity in the input data, which can be beneficial for neural networks since they often require inputs of consistent dimensions. Resizing is particularly common in computer vision tasks, where images may have varying sizes but need to be fed into the model with fixed dimensions.

Normalization is the process of scaling the numerical values of the input images to a standard range. It helps in stabilizing and accelerating the training process, as it ensures that features contribute equally to the model's learning process. For example, in an image data set, normalization might involve scaling pixel values from their original range (e.g., 0–255 for grayscale images) to a range between 0 and 1, by dividing each pixel value by 255.

Data augmentation involves generating new training examples by applying transformations or perturbations to existing data. These transformations can include rotations, translations, flips, changes in brightness or contrast, and more. Augmentation helps in increasing the diversity of the training data set, which can improve the model's ability to generalize about unseen data and reduce overfitting.

Next, we iteratively optimized our Deep Neural Network Architecture, including the self-reflection Mechanisms described in this paper. In particular, these were addressed by Image Embedding Method Selection and Optimization. We reiterate that image embedding is a technique used in computer vision and machine learning to represent images as feature vectors, enabling various tasks such as image classification, object detection, and image retrieval. In our SAL approach for classification of mineralogical thin sections, the network iteratively tests multiple embedding algorithms, ultimately adopting the optimal technique (the one that produces the highest performance parameters). In this specific case, the SqueezeNet embedding method was the most effective. This is a computationally efficient neural network architecture, combining convolutional layers and modules to extract image features while maintaining high accuracy. In particular, the SqueezeNet embedding method leverages a combination of efficient architectural design choices, such as model compression techniques, to achieve a good balance between computational efficiency and accuracy in image classification tasks. This makes it particularly well-suited for deployment on resource-constrained devices or in scenarios where computational efficiency is critical.

The SAL network also explored other techniques, including VGG-16 and VGG-19, and "Painters", a method synthesizing images to create embedding. Additionally, it examined "DeepLoc", an algorithm designed for protein subcellular localization tasks, utilizing multiple layers of a convolutional neural network for image embedding.

VGG-16 and VGG-19 are well-known convolutional neural network architectures that allow the achievement of higher accuracy on image classification tasks. However, this comes at the cost of increased computational resources and memory requirements.

The "Painters" method synthesizes images to create embeddings, likely utilizing generative adversarial networks (GANs) or similar techniques. While it may produce

visually appealing results, its accuracy for image classification tasks may vary depending on factors such as the quality of the synthesized images and the effectiveness of the embedding process. It may not achieve the same level of accuracy as traditional convolutional neural network-based methods like SqueezeNet, VGG-16, or VGG-19.

"DeepLoc" is an algorithm specifically designed for protein subcellular localization tasks, utilizing multiple layers of a convolutional neural network for image embedding. While its accuracy may vary depending on the specific task and data set, "DeepLoc" is optimized for this particular biological application and may achieve high accuracy in protein localization tasks. However, its performance may not directly translate to other image classification tasks.

The next step was focused on Overfitting Reduction; our SAL model includes techniques to reduce overfitting, such as dropout, regularization, early stopping, and so forth.

We recall that both dropout and regularization help improve the model's generalization performance by promoting simpler, more robust representations of the data and preventing the model from memorizing noise in the training data. By incorporating these techniques, machine-learning practitioners can build models that perform well not only on the training data but also on new, unseen data.

The SAL network evaluated the results on a separate test data set to get an unbiased estimate of its performance. Finally, it measured classification accuracy, precision, recall, F1-score, and other relevant metrics to quantitatively assess the model's performance. After proper training, the SAL model performed inference on new, unseen images to classify the entire data set.

Figure 6 shows an example of classified images, with an accuracy close to 80%. The limited number of labelled examples used for training causes about 20% of the misclassified images (not shown in the figure). We expect that by increasing the size of the training data set, the classification results can improve significantly.

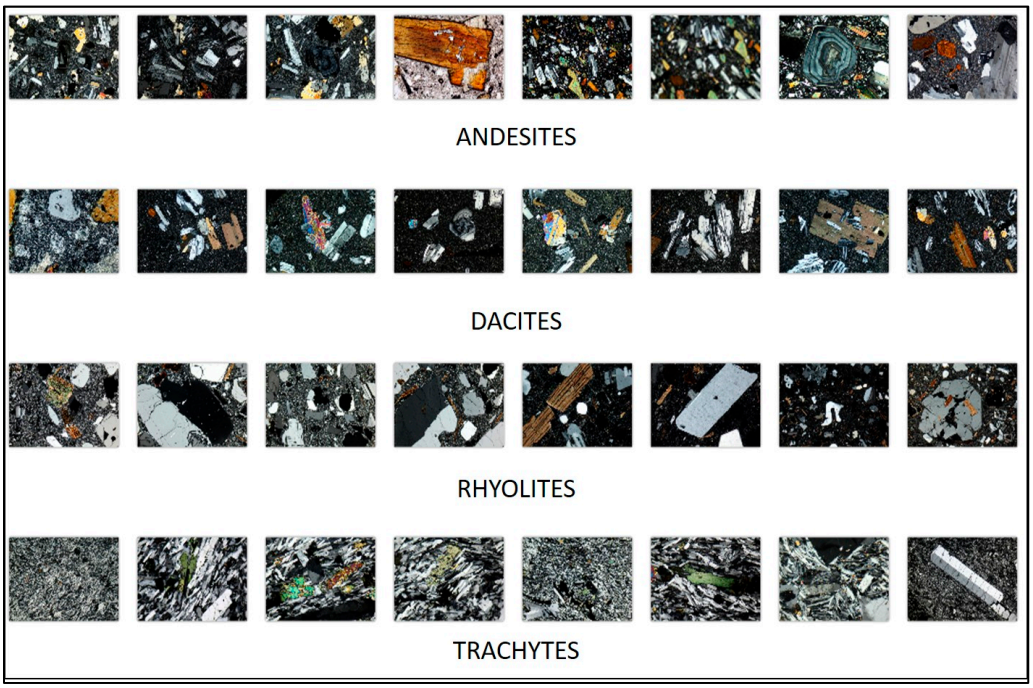


**Figure 6.** Classification examples through SAL model network of different types of magmatic rocks.

As in the previous test, we compared the performances of the updated SAL model with those of other classification methods based on different machine-learning algorithms. Table 2 shows the classification accuracy and the precision values for the SAL model, Decision Tree, Random Forest, Logistic Regression, CN2 Rule Inducer and Adaptive Boosting algorithms, all trained on the same labelled data set.

**Table 2.** Comparison of performance indexes (like in Table 1) of SAL Neural Network and other machine learning algorithms (see Appendix B for a brief explanation of these alternative algorithms).

| Model | Classification Accuracy | Precision |
|---|---|---|
| **SAL Neural Network** | **0.781** | **0.817** |
| Decision Tree | 0.625 | 0.681 |
| Random Forest | 0.719 | 0.71 |
| Logistic Regression | 0.618 | 0.551 |
| CN2 Rule Inducer | 0.428 | 0.311 |
| Adaptive Boosting | 0.625 | 0.655 |

## 7. Conclusions

Self-Awareness Learning (SAL) based on deep neural networks updated by self-reflection and self-adapting mechanisms can significantly improve the performance of standard deep neural networks. In fact, these models can continuously monitor their own performance and adapt their hyper-parameters accordingly. This adaptability allows the network to dynamically adjust its structure/architecture and optimize its behavior based on the variations of the environment and of the input data set. One significant advantage of self-awareness networks is their ability to mitigate overfitting issues. By continuously re-adapting their hyper-parameters, these models can dynamically control the complexity of the network, preventing it from memorizing noise or irrelevant patterns in the data and, finally, improving performance. Tests on both synthetic and real petrological data have confirmed the benefits of self-awareness networks. In particular, we have discussed how the SAL approach can improve accuracy and general performances of deep neural networks for classification of petrological data based on both chemical attributes and thin section images. We discussed these applications separately, but it is clear that the same SAL model can be applied simultaneously to numerical and image attributes, with the purpose of using a complete multi-feature matrix for automatic classification. In other words, the performance of the SAL approach can be improved further by combining multiple and complementary types of attributes, enhancing the capability of this approach for classification and prediction tasks. All our tests have demonstrated improved performance, better generalization, and the ability to handle varying data characteristics. The adaptive nature of these models allows them to automatically adjust hyper-parameters such as learning rate, regularization parameters, or dropout rates, reducing the need for manual tuning.

In conclusion, self-awareness networks contribute to more efficient and effective training processes by optimizing themselves based on encountered data and task complexity. They offer a flexible and autonomous approach to hyper-parameter tuning, leading to improved overall performance without the need for extensive manual experimentation.

**Funding:** This research received no external funding.

**Data Availability Statement:** Public data set available on the GEOROC (Geochemistry of Rocks of the Oceans and Continents) website (http://georoc.mpch-mainz.gwdg.de/georoc/ accessed on 15 December 2023). For this paper, we selected we selected some of the data referenced as [20]. All the images (microscope mineral thin sections) discussed and shown in this paper have been obtained courtesy of Alessandro Da Mommio. Link: http://www.alexstrekeisen.it/index.php, accessed on 21 July 2023. The specific jpeg files used in this paper can be obtained upon request by writing an email to dellavers@tiscali.it.

**Conflicts of Interest:** The author declares no conflicts of interest.

## Appendix A. A Simplified Example of Code for Self-Reflection Model Update

The following Python code shows, just for didactical purposes, the training loop for a machine learning self-reflection model using the TensorFlow and Keras libraries. The users can modify, expand and/or re-adapt the code for their own purposes, applying it to a specific data set given in input. Let us describe it step by step:

- The code starts with a "for loop" iterating over the number of epochs specified by the "epochs" variable.
- Within each epoch, a tf.GradientTape() context is created. This context is used to compute the gradients of the trainable variables with respect to a given loss function. It enables automatic differentiation in TensorFlow.
- The model "self_reflection_model" (this can be any initial deep learning model properly set by the user) is called with the input data X_train to obtain the model's predictions (outputs).
- The "sparse_categorical_crossentropy" loss function from Keras is applied to calculate the loss value between the predicted outputs and the true labels y_train.
- The gradients of the loss with respect to the trainable variables of the model are computed using tape.gradient().
- The computed gradients are then used to update the model's trainable variables using the optimizer's apply_gradients() method.
- The model is evaluated on the training set (X_train and y_train) to compute the training loss and accuracy.
- The model is also evaluated on the validation set (X_test and y_test) to compute validation loss and accuracy.
- The computed losses and accuracies are appended to their respective lists in the self_reflection_history dictionary.
- If the current validation accuracy is higher than the previous best accuracy, the best accuracy and the configuration of the current model are updated.
- The self-reflection hyper-parameters (in this simplified example, just learning rate, momentum, and dropout rate) are appended to their respective lists in the self_reflection_hyperparameters dictionary.
- Self-reflection mechanisms are applied to adjust the hyper-parameters based on the performance of the validation set. If the current validation accuracy is lower or equal to the previous epoch's validation accuracy, the hyper-parameters are updated.

Code "building blocks":

```
# First, import all the necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras

# Assuming that all the previous building blocks have been implemented (data loading,
# initial network model properly set, initial training loop, result plotting block and so
# forth), the following is the core block for the self-reflection model update).

    for epoch in range(epochs):
        with tf.GradientTape() as tape:
            outputs = self_reflection_model(X_train, training = True)
            loss_value = keras.losses.sparse_categorical_crossentropy(y_train, outputs)
            gradients = tape.gradient(loss_value, self_reflection_model.trainable_variables)
        self_reflection_model.optimizer.apply_gradients(zip(gradients, \
        self_reflection_model.trainable_variables))

        train_loss,train_accuracy=self_reflection_model.evaluate(X_train,y_train,verbose=0)
        val_loss, val_accuracy = self_reflection_model.evaluate(X_test, y_test, verbose = 0)

        self_reflection_history['loss'].append(train_loss)
        self_reflection_history['accuracy'].append(train_accuracy)
        self_reflection_history['val_loss'].append(val_loss)
```

```
self_reflection_history['val_accuracy'].append(val_accuracy)

if val_accuracy > best_accuracy:
    best_accuracy = val_accuracy
    best_architecture = self_reflection_model.get_config()

# Update self-reflection hyper-parameters
self_reflection_hyperparameters['learning_rate'].append(learning_rate)
self_reflection_hyperparameters['momentum'].append(momentum)
self_reflection_hyperparameters['dropout_rate'].append(dropout_rate)

# Self-reflection mechanisms
if epoch > 0 and val_accuracy <= self_reflection_history['val_accuracy'][epoch-1]:
    self_reflection_model.compile(optimizer = 'adam', \
    loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    print("Updating hyperparameters...")
    learning_rate * = 0.9 # Decrease learning rate by 10%
    momentum * = 1.1 # Increase momentum by 10%
    dropout_rate += 0.5 # Increase dropout rate by 0.5
```

In summary, this simplified block code implements a training loop that combines basic (extremely simplified) self-reflection mechanisms with model training, performance evaluation, and best model tracking. It enables the model to adapt its own learning process, assess its performance, and make adjustments to improve its accuracy and overall performance in the context of self-learning and self-awareness deep learning.

We remark that this code is very simple because it has just illustrative purposes. The reader can ask directly the author for more complex and complete codes (Jupyter notebooks) regarding the SAL approach at the following email address: dellavers@tiscali.it.

## Appendix B. An Overview of Artificial Neural Networks, Other Machine Learning Methods and Key Terminology

In this appendix, we provide a simple and didactical introduction to the key terminology of Artificial Neural Networks (ANN), together with a conceptual scheme of ANN. Our purpose is to allow geoscientists who are not necessarily experts in Machine Learning to understand the fundamentals and the key aspects of Neural Networks, Machine Learning, and Deep Learning techniques frequently mentioned in this paper. With the help of this tutorial appendix, the reader should be able to follow without any difficulty the methodological discussion and the innovations introduced in our work. Additional details can be found in Dell'Aversana [19]. In the appendix, we do not follow any alphabetical order of terminology, but rather a logical order that can be useful for clarifying the most important aspects of the learning process and the architecture of Artificial Neural Networks.

Artificial Neural Network Model. An Artificial Neural Network (ANN) model is a computational framework inspired by the structure and functioning of biological neural networks in the human brain. It consists of interconnected nodes, or neurons, organized in layers. Information is processed through these interconnected nodes, where each node performs a simple mathematical operation on its input and passes the result to the next neuronal layer. Through a process of training using input-output pairs, ANNs can learn complex patterns and relationships within data, enabling tasks such as classification, regression, and pattern recognition. ANN models have found applications in various fields including image and speech recognition, natural language processing, and predictive analytics. Figure A1 shows one of the first conceptual ANN models, called "ADALINE", short for "Adaptive Linear Neuron" or later "Adaptive Linear Element". This is an early single-layer Artificial Neural Network. Its name derives from the physical device that implemented this network, developed by Professor Bernard Widrow and his doctoral

student Ted Hoff at Stanford University in 1960. It is useful for didactical purposes because it includes some of the crucial aspects of ANNs, such as "weights", a "bias", a "summation", and "activation functions".

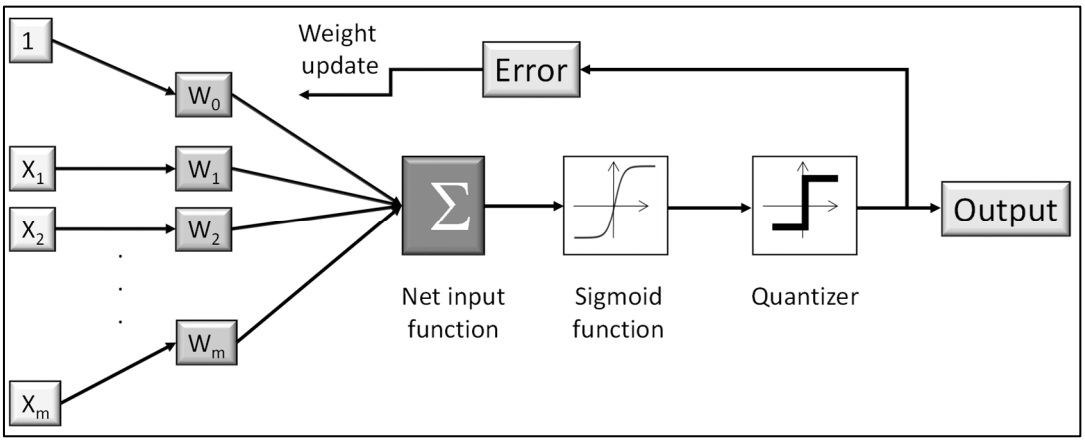


**Figure A1.** Scheme of the "ADALINE" workflow using the sigmoid (Logistic) activation function (see text below for explanation of this terminology).

Weights, Neurons Bias, Summation and Activation Functions: In an Artificial Neural Network, weights ($W_i$) represent the strength of connections between neurons. Neurons are the basic computational units of a neural network. They receive inputs, perform a computation, and produce an output. The weights determine the impact of one neuron's output on another neuron's input. Each connection between neurons is associated with a weight, which is essentially a parameter that the neural network learns during the training process. When the network receives an input ($X_i$), each input is multiplied by its corresponding weight ($W_i$), and these weighted inputs are then summed, through a "summation function" ($\Sigma$, in the figure) and passed through an "activation function" (such as the Sigmoid function in the figure) to produce the output of the neuron. Finally, bias in an ANN is an additional parameter that allows neurons to have flexibility in their activation functions, influencing when they activate and improving the network's ability to model complex relationships in the data. Below, we are going to explain, briefly, what the role of these weights, neurons, activation functions and so forth is during the training process of a generic ANN, and what the final goal of this type of architecture is.

Learning process in ANNs: Adjusting the weights during training is how the neural network learns to make accurate predictions or classifications based on the input data. In simple words, an ANN functions by mimicking the biological structure of the brain through interconnected nodes, or neurons. Its goal is to learn patterns from training data by adjusting the weights of connections between neurons, ultimately producing an expected output when presented with new input data.

Activation functions: ANNs often use activation functions to introduce non-linearity into the model, such as the Sigmoid. Quantizing ("Quantizer" in the figure) these activation functions can involve reducing the precision of the values they produce, typically by rounding or truncating them to a certain number of bits. This process can help in reducing the computational complexity and memory requirements of the network.

An example of an activation function is the Rectified Linear Unit (ReLU); this is one of the simplest activation functions and is defined as follows: If the input is positive, ReLU returns the input value unchanged. If the input is negative, ReLU returns zero.

ANN Architecture: With reference to Formula (1) in this paper, the architecture of an ANN refers to its overall structure or organization, including the arrangement of neurons, the number of layers, and the connections between neurons. The architecture defines how information flows through the network and determines its computational capabilities. Key components of an ANN's architecture include the following:

Neuron Structure and Layer Organization: ANN architectures typically consist of multiple layers of neurons arranged in a sequential manner. Common types of layers include input layers, hidden layers, and output layers. The architecture specifies the number of layers and the number of neurons in each layer. Deep neural networks have multiple hidden layers, enabling them to learn complex hierarchical representations of data.

Connectivity Patterns: The architecture defines how neurons are connected within and between layers. In feedforward neural networks, neurons are typically fully connected, meaning each neuron in one layer is connected to every neuron in the subsequent layer. Other architectures, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have specialized connectivity patterns tailored to specific types of data (e.g., images, sequences).

Topology: The arrangement of neurons and connections forms the network's topology. This includes the overall shape and structure of the network, which can vary widely depending on the specific architecture and task requirements.

Training and Learning Mechanisms: The architecture may also specify the training algorithm used to optimize the network's parameters (e.g., weights and biases) based on input–output pairs. Common training algorithms include backpropagation and its variants, which adjust the network's parameters to minimize a specified loss function (see below).

Learning rate: The learning rate is a hyper-parameter that controls the step size at which the weights of a neural network are updated during the training process. It determines how much the model's parameters (weights and biases) are adjusted in response to the estimated error each time the model undergoes an optimization step, typically through techniques like gradient descent. Indeed, the gradient of the loss function with respect to the network parameters is a fundamental concept in training neural networks, particularly through techniques like "gradient descent" and its variants. Here is a detailed explanation that requires the brief introduction of further basic terminology widely mentioned in the paper:

Loss Function: The loss function quantifies the discrepancy between the predicted output of the neural network and the actual target values in the training data. It represents the objective that the network aims to minimize during training. Common loss functions include mean squared error (MSE) for regression tasks and categorical cross-entropy for classification tasks.

Gradient: The gradient of a function represents the rate of change of the function with respect to its parameters. In the context of neural networks, the gradient of the loss function with respect to the network parameters (typically weights and biases) indicates the direction and magnitude of the steepest ascent of the loss function in the parameter space. In other words, it tells us how the loss would change if we were to make small adjustments to the parameters.

Backpropagation: Computing the gradient of the loss function with respect to the network parameters is typically performed using the backpropagation algorithm. Backpropagation efficiently computes the gradients layer by layer, starting from the output layer and moving backward through the network. It utilizes the "chain rule" of calculus to propagate gradients backward through the network, efficiently computing the gradients of the loss function with respect to the parameters of each layer.

Parameter Update: Once the gradients of the loss function with respect to the network parameters are computed, they are used to update the parameters in the direction that minimizes the loss. This update is typically performed iteratively using optimization algorithms like gradient descent, which adjust the parameters by subtracting a fraction of the gradient scaled by the learning rate.

Optimization: By iteratively updating the network parameters based on the gradients of the loss function, the network gradually learns to minimize the loss and improve its predictive performance on the training data. This process of optimizing the network parameters based on the gradients of the loss function is at the core of training neural networks.

Convergence in Neural Network Training: Convergence refers to the point in the training process of a neural network where the model parameters stabilize, and further training iterations do not significantly improve the performance of the model on the training data. In simpler terms, it is when the model has learned as much as it can from the data and has reached its optimal performance level.

Local Minima: In optimization problems, like training a neural network, the goal is to find the lowest point (minimum) of the loss function which measures how well the model's predictions match the actual targets. However, there can be many low points in the landscape of the loss function, and not all of them are the lowest possible point. Local minima are points in this landscape where the loss function is lower than its value in the surrounding area but may not be the absolute lowest point.

Cross-entropy: The cross-entropy loss function (Formula (2) in the paper), often used in classification tasks, measures how well the predicted probabilities match the actual outcomes. In essence, cross-entropy loss punishes wrong predictions more severely when they are confident (high probability) and rewards correct predictions more when they are confident. It is like a scorecard that tells you how well your predictions match the actual outcomes, encouraging your model to improve its predictions over time.

Epoch: In the context of training an Artificial Neural Network (ANN), an epoch refers to one complete pass of the entire training data set through the neural network. During each epoch, the neural network undergoes all the necessary steps, including the "Forward Pass" (each training sample in the data set is fed forward through the network, layer by layer, to generate predictions or outputs), calculation of Loss, Backward Pass (Backpropagation), Parameter Update, and so forth.

Performance of ANN: The performance of an Artificial Neural Network model, mentioned in Formula (3) of the paper, refers to how well it accomplishes its intended task, such as classification, regression, or pattern recognition. Measuring the performance of an ANN is crucial for evaluating its effectiveness and determining whether it meets the requirements of the problem at hand. Here is how we typically measure the performance of an ANN:

Accuracy: Accuracy is a common metric for classification tasks and represents the proportion of correctly classified instances out of the total number of instances. It is calculated as the number of correct predictions divided by the total number of predictions.

Precision, Recall, and F1 Score: These metrics are commonly used in binary and multiclass classification tasks. Precision measures the proportion of true positive predictions out of all positive predictions, recall measures the proportion of true positive predictions out of all actual positive instances, and the F1 score is the harmonic mean of precision and recall. These metrics provide a better understanding of the performance of the ANN, particularly in imbalanced data sets.

ROC Curve and AUC: Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) are used to evaluate the performance of binary classifiers. The ROC curve plots the true positive rate against the false positive rate at various threshold settings, and AUC represents the area under the ROC curve. A higher AUC indicates better discrimination between positive and negative instances.

Mean Absolute Error (MAE) and Mean Squared Error (MSE): These metrics are commonly used for regression tasks. MAE measures the average absolute difference between the predicted and actual values, while MSE measures the average squared difference between them.

Confusion Matrix: A confusion matrix provides a tabular summary of the performance of a classification model. It shows the number of true positive, true negative, false positive, and false negative predictions, allowing for a more detailed analysis of the model's performance across different classes.

Overfitting: In Artificial Neural Networks, overfitting refers to a situation where a model learns to perform very well on the training data but fails to generalize its performance

to new, unseen data. In other words, the model captures noise and randomness in the training data as if it were real patterns, leading to poor performance on unseen data.

A brief explanation of additional Machine Learning methods used in the paper:

The CN2 Rule Induction consists of an algorithm designed for the efficient induction of simple rules of form "if condition, then predict class". Its main advantages are that it works properly even in the presence of significant noise, and the classification rules can be easily understood.

The Naïve Bayes classifier works using a Bayesian approach. A probabilistic classifier estimates conditional probabilities of the dependent variable from training data. Then it applies the posterior probabilities to the classification of new data instances. A key advantage offered by this approach is that it is fast for discrete features; in contrast, it is less efficient for continuous features.

A support vector machine (SVM) works on a different principle with respect to the previous algorithms. In fact, it splits the attribute space with a hyper-plane, and tries to maximize the margin between the instances of different classes or class values.

The Decision Tree algorithm is a technique that works by separating the data into two or more homogeneous sets (or sub-populations). The separation criteria are based on the most significant features in input variables. It is a precursor to Random Forest.

Random Forest is an ensemble learning method that uses a set of Decision Trees. Each Tree is developed from a sample extracted from the training data. When developing individual Trees, an arbitrary subset of attributes is drawn (hence the term "Random"). The best attribute for the split is selected from that arbitrary subset. The final model is based on the majority vote from individually developed Trees in the Forest.

Adaptive Boosting: Similar to Random Forest, Adaptive Boosting consists of multiple classifiers; the final output is the combination of the outputs of those algorithms. The final goal is to create a strong classifier as a linear combination of "weak" classifiers.

Additional concepts and definitions used in the paper:

The normalized probability density distribution, also known as the normalized probability density function (PDF), represents the probability distribution of a continuous random variable. "Normalized" in this context means that the area under the probability density curve sums to 1, ensuring that the total probability of all possible outcomes is equal to 1.

## References


1. Damasio, A. *Self Comes to Mind: Constructing the Conscious Brain*; Pantheon: New York, NY, USA, 2010.
2. Edelman, G.M. *Neural Darwinism: The Theory of Neuronal Group Selection*; Basic Books: New York, NY, USA, 1987; ISBN 0-19-286089-5.
3. Edelman, G.M. *Bright Air, Brilliant Fire: On the Matter of the Mind*; Reprint Edition 1993; Basic Books: New York, NY, USA, 1992; ISBN 0-465-00764-3.
4. Tononi, G.; Boly, M.; Massimini, M.; Koch, C. Integrated information theory: From consciousness to its physical substrate. *Nat. Rev. Neurosci.* **2016**, *17*, 450–461. [CrossRef] [PubMed]
5. Tononi, G.; Edelman, G.M. Consciousness and complexity. *Science* **1998**, *282*, 1846–1851. [CrossRef] [PubMed]
6. Chella, A.; Frixione, M.; Gaglio, S. A cognitive architecture for robot self-consciousness. *Artif. Intell. Med.* **2008**, *44*, 147–154. [CrossRef] [PubMed]
7. Chella, A.; Lanza, F.; Pipitone, A.; Seidita, V. Knowledge acquisition through introspection in human-robot cooperation. *Biol. Inspir. Cogn. Arc.* **2018**, *25*, 1–7. [CrossRef]
8. Dehaene, S.; Lau, H.; Kouider, S. What is consciousness, and could machines have it? *Science* **2017**, *358*, 486–492. [CrossRef] [PubMed]
9. Gorbenko, A.; Popov, V.; Sheka, A. Robot self-awareness: Exploration of internal states. *Appl. Math. Sci.* **2012**, *6*, 675–688.
10. Graziano, M.S. The attention schema theory: A foundation for engineering artificial consciousness. *Front. Robot. AI* **2017**, *4*, 60. [CrossRef]
11. Holland, O. (Ed.) *Machine Consciousness*; Imprint Academic: New York, NY, USA, 2003.
12. Kinouchi, Y.; Mackin, K.J. A basic architecture of an autonomous adaptive system with conscious-like function for a humanoid robot. *Front. Robot. AI* **2018**, *5*, 30. [CrossRef]

13. Lewis, P.; Platzner, M.; Yao, X. An Outlook for Self-Awareness in Computing Systems. Self Awareness in Autonomic Systems Magazine. 2012. Available online: https://www.researchgate.net/publication/263473254_An_Outlook_for_Self-awareness_in_Computing_Systems (accessed on 15 December 2023).
14. Novianto, R. Flexible Attention-Based Cognitive Architecture for Robots. Ph.D. Thesis, Open Publications of UTS Scholars, University of Technology, Sydney, NSW, Australia, 2014.
15. Reggia, J.A. The rise of machine consciousness: Studying consciousness with computational models. *Neural Netw.* **2013**, *44*, 112–131. [CrossRef] [PubMed]
16. Scheutz, M. Artificial emotions and machine consciousness. In *The Cambridge Handbook of Artificial Intelligence*; Frankish, K., Ramsey, W., Eds.; Cambridge University Press: Cambridge, UK, 2014; pp. 247–266. [CrossRef]
17. Winfield, A.F.T. Experiments in artificial theory of mind: From safety to story-telling. *Front. Robot. AI* **2018**, *5*, 75. [CrossRef] [PubMed]
18. Dell'Aversana, P. An Integrated Deep Learning Framework for Classification of Mineral Thin Sections and Other Geo-Data, a Tutorial. *Minerals* **2023**, *13*, 584. [CrossRef]
19. Dell'Aversana, P. *Artificial Neural Networks and Deep Learning: A Simple Overview*; Research Gate: Berlin, Germany, 2019. [CrossRef]
20. Mamani, M.; Wörner, G.; Sempere, T. Geochemical variations in igneous rocks of the Central Andean orocline (13° S to 18° S): Tracing crustal thickening and magma generation through time and space. *Bull. Geol. Soc. Am.* **2010**, *122*, 162–182. [CrossRef]