COMPONENT-WISE PERFORMANCE EVALUATION OF NEURAL NETWORKS USING THE SHAPLEY VALUE

INTRODUCTION TO GAME THEORY

Marco Bagatella **Gianluca Lain** **Samuele Giuliano Piazzetta** **Andrea Ziani**

ABSTRACT

Even though Artificial Neural Networks (ANN) are one of the most used models in machine learning, they are still too often treated as "black box" tools without a complete understanding of why some NN architectures work better than others. In this paper, we introduce a novel application of the Shapley Value in machine learning. While past approaches mainly target model interpretability and feature selection, we explore an orthogonal direction: we propose a method for measuring the contribution of single components to the performance of a neural network. By treating such components as players in a cooperative game, we can evaluate their utility by measuring the model's accuracy. We find that our results are consistent and coherent with common empirical findings in machine learning.

1 Introduction

Artificial neural networks are fundamental models in machine learning that can dramatically boost the efficacy of analytic tools. Nowadays, despite ANN being used more and more, understanding and finding the optimal structure of a multi-layer neural network is still an open problem. Frequently, the method used to evaluate the goodness of the elements present in a network is "trial and error". Data scientists try different architectures, train the resulting network, and evaluate training and testing errors. This technique is straightforward, but it is also time-consuming and relies on the data scientist's personal experience.

For this reason, researchers developed methods such as grid search, Bayesian optimization, and random search to overcome this cherry-picking approach programmatically. These techniques try to find the best architecture for a given problem, such as 'which type' and 'how many' components perform the best in that scenario. However, these methods fail to explicitly provide a higher-level understanding of how single components cooperate to increase the overall performance. Indeed, the risk is that, with a grid search approach, we can decide to add a specific component blindly, even if it does not significantly contribute to improving the overall performance in combination with all the other chosen components. In this way, neural networks are often taken as "black box" tools and are applied without understanding why a particular architecture is better than another.

Contribution
In this paper, we propose a new approach for evaluating the architecture of a neural network, which leverages the concept of cooperative games of game theory, namely the Shapley Value [1]. This method highlights the importance of single components in the network and tries to understand how a given component contributes to the overall performance while operating in a coalition with another subset of components.

Organization
In Section 2, we give an overview of the state-of-the-art methods that leverage the Shapley Value concept in the field of machine learning. In Section 3, we present the fundamental concepts of the Shapley Value and neural networks. Next, in Section 4, we describe our proposed method in detail and compare its benefits with the existing techniques. In Section 5, we give details about our test architecture and which experiments we conducted. In Section 6, we show the evaluation of our experiments, finding our results to be consistent and coherent with empirical findings in machine learning. Finally, in Section 7 we summarize our research and we propose possible future directions.

2 Related work

The Shapley Value was initially introduced in the field of game theory in relation to cooperative games. Recently, many researchers modeled various problems as if they were instances of games, using mathematical and economic theory to prove their hypotheses. In particular, the popularization of the field of machine learning saw the Shapley Value being used in different tasks.

From our research, feature selection was the first technique in which the Shapley Value was applied in the context of machine learning. As explained in [2, 3, 4], feature selection can be seen as a cooperative game in which the features are considered as players. At this point, the Shapley Value can help us understand which is the best-performing subset of features according to some performance metric used as a characteristic function of the game.

Subsequently, researchers thought to apply the Shapley Value to more complex models than classic linear regression, applying game theory concepts to artificial neural networks. For instance, as explained by F. Leon in [5], it is possible to employ the Shapley Value on neural networks by considering the hidden neurons as the players of a cooperative game. Leon explains how to optimize the topology of neural networks based on the Shapley Value, with an algorithm that prunes active neurons on the network simultaneously.

Another example where the Shapley Value was applied to NNs is [6]. In this paper, the author identifies an optimal feature subset using a novel algorithm called "Shapley Value Embedded Genetic Algorithm". Then, he shows how this method improves the neural network's accuracy using only the optimal, selected features. A completely new application of the Shapley Value was introduced by Mokdad et al. in [7]. The author proposed considering various feature selection methods as players. In this way, he was able to find the best subset of feature selection techniques that lead to the model's best overall performance.

As far as we know, no prior work proposed using the Shapley Value to evaluate the architecture of a neural network component-wise, i.e., seeing the components of the neural networks as players to find which combination of them gives the best contribution to the overall performance.

3 Background

In this section, we define the main idea behind cooperative games and the Shapley Value. We explain how these two concepts of game theory are mostly used in machine learning, highlighting advantages and disadvantages. Then, we give a brief overview of neural networks, which may help the reader to understand our research better.

3.1 Cooperative games and the Shapley Value

Cooperative games: In the game theory field [8], a cooperative game is a competition in which the primary units of decision-making are individuals (players) and coalitions (groups of players). The coalitions are defined by the signing of binding contracts between players.

Formally, a game can be defined by a 2-tuple $G(v, N)$ where:

- v is the characteristic function that maps subsets of players to the real numbers: $v : 2^N \rightarrow \mathbb{R}$, where $v(\emptyset) = 0$;
- $N = \{1, 2, ..., n\}$ indicates the indexes of the players in the game.

A coalition C_j is a specific subset of N, and the set of all coalitions is called *coalition structure* $\rho = \{C_1, C_2, .., C_k\}$. We define $k = 2^N$ as the total number of possible coalitions. Since $C_j \subseteq N$ with $j \in \{1, .., k\}$, we say that a coalition $C_j = N$ is the *grand coalition* and a coalition $C_j = \emptyset$ is an *empty coalition*.

Consider a game $G(v, N)$, where a coalition of players cooperates and obtains a certain overall gain from that cooperation. As some players may contribute more to the coalition than others, it may be useful to know how important is each player to the overall cooperation and what payoff can he reasonably expect. The Shapley Value provides one possible answer to this question. Furthermore, for a specific game $G(v, N)$, a "fair" allocation $x_i^*(v)$ should satisfy these four axioms:

- **Efficiency**: $\sum_{i \in N} x_i^*(v) = v(N)$;
- **Symmetry:** for any two players *i* and *j*, if $v(S \cup i) = v(S \cup j)$ for all *S* not including *i* and *j*, then $x_i^*(v) = x_j^*(v)$;
- **Dummy player:** for any *i*, if $v(S \cup i) = v(S)$ for all *S* not including i, then $x_i^*(v) = 0$;
- **Additivity:** If *u* and *v* are two characteristic functions, then $x^*(v + u) = x^*(v) + x^*(u)$.

The unique function which satisfies all four axioms for the set of all games is the Shapley Value.

The Shapley Value is a solution concept of fairly distributing gains and costs to several individuals working in a coalition. It applies primarily in situations in which each player's contributions are unequal, but they work in cooperation with each other to obtain the payoff.

Consider a set of N players and a characteristic function v that maps each subset $C_j \subseteq N$ of players to a real number, defining the payoff of a game when players in C_j take part in it. The Shapley Value is a way to quantify each individual's total contribution to the total reward $v(N)$ of the game when all players participate.

For a specific game $G(v, N)$, the Shapley Value of the *i*-th player can be computed as:

$$\phi_i(v, N) = \sum_{C_j \in N, i \in C_j} \frac{(|C_j| - 1)! \cdot (|N| - |C_j|)!}{|N|!} \cdot [v(C_j) - v(C_j \backslash \{i\})] \tag{1}$$

Hence, the Shapley Value for the *i*-th player is defined as the *average marginal contribution* of *i* to all possible coalitions C that can be formed without him. The marginal contribution is described by the second part of the equation: $[v(C_j) - v(C_j \backslash \{i\})]$. The sum in the equation defines that we have to sum up all the marginal values of the coalition, scaling each of them by a fixed factor. The value $(|C_j| - 1)! \cdot (|N| - |C_j|)!$ calculates how many permutations of each subset size we can have when we construct it out of all remaining coalition members excluding player i. The term $\frac{1}{|N|!}$ describes a scaling factor equal to the total number of players in the game. We use the two different scaling factors to average out the effects of the other coalition members for each subset size and to average the effect of the group size as well.

3.2 Common use of the Shapley Value in machine learning

Machine learning comprises a vast family of computational methods aiming to achieve good performance or predictions. One class of algorithms consists of learning a mapping $f : X \mapsto Y$ from an input x to an output y, starting from some input-output example pairs. This subset, which takes the name Supervised learning, includes tasks such as classification (i.e., assign data points a label) and regression (i.e., learn the real value mapping). Overall, the final goal is to obtain a model able to solve the task on new, unseen data.

In some simple tasks, the input x can be easily represented as a vector of features $\boldsymbol{x} \in \mathbb{R}^d$. In this scenario, the challenge comes down to selecting a hypothesis set (i.e., a family of mapping functions ϕ_j that characterizes the model) and finding suitable parameters w_j, which minimize a loss function ℓ. A loss function measures how much error we are committing while predicting labels $\hat{\boldsymbol{y}} \in \mathbb{R}^n$ from our model against the true labels $\boldsymbol{y} \in \mathbb{R}^n$. The learning process can be summarized with the following optimization problem:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \ell\Bigg(y_i; \sum_{j=1}^{d} w_j \phi_j(\boldsymbol{x}_i)\Bigg) \tag{2}$$

Therefore, while it is possible to tweak the weights w_j, the chosen model and the input features represent some constraints. Adopting the holdout method [9] or cross-validation [9, 10, 11] helps to solve the model selection problem, while feature engineering the initial data helps to obtain more relevant inputs. For the latter process, the following are the principal approaches:

- **Features extraction**: the process of composing and/or transforming the raw inputs to derive new and more effective features. Usually, hand-designing features have the drawback of requiring domain-knowledge;
- **Features selection**: the process of discarding the less relevant features, keeping only the ones containing the majority of the information. The application of this technique is useful to prevent overfitting: reducing the dimensions of the input reduces the complexity of the model, which often implies a reduction in the generalization error [2].

The most frequent use of the Shapley Value in the machine learning field is, indeed, to explain how much each feature influences the model prediction, i.e., understanding the importance of the features. As explained by Cohen et al. in [12], we can see a prediction of our machine learning model as an instance of a game by considering each feature value as a player in a game where the prediction is the payoff. The Shapley Values tell us how to distribute the payoff among the features fairly. Thus, a feature that is extremely useful for the prediction of the model will be associated with a high payoff. On the other hand, a feature that does not add much information to the model will be associated with a small payoff.

Since the Shapley Value satisfies the four axioms mentioned in Section 3.1, the difference between the prediction and the average prediction is fairly distributed among the feature values of the instance. This property distinguishes the Shapley Value from other methods of model explanation, such as LIME [13]. Furthermore, the Shapley Values may be defined on a global level, indicating how the model uses the features overall, and on a local level, indicating how the model decides for an individual data point. Thus, instead of comparing a prediction to the average prediction of the entire dataset, it is possible to compare it to a subset or a single data point. This contrastiveness is also something that local models like LIME do not have.

Despite the many advantages of the Shapley Value, all possible coalitions of feature values have to be evaluated with and without the *i*-th feature to calculate its exact contribution. Hence, for more than a few features, the exact solution becomes problematic as the number of possible coalitions increases exponentially with the number of features. For this reason, different methods to compute an approximation of the Shapley Value, which leverage Montecarlo sampling, are often used [14]. Another method, proposed by Lundberg and Lee [15], evaluates the model over randomly sampled subsets and uses a weighted linear regression to approximate the Shapley Values. However, these sampling-based approximations can suffer from high variance when the number of samples to be collected per instance is limited. For large-scale models, where the number of features is large, the number of samples required to obtain a stable estimate can be prohibitively computationally expensive.

3.3 Neural Networks

In some scenarios, the complexity of the input (e.g., an image) makes guessing the correct hypothesis set, and hand-crafting features a tough, if not impossible, job. To face this problem, it is necessary to introduce a different family of models: neural networks.

The main idea behind neural networks is parameterizing features mappings ϕ and learn the related parameters θ_j instead of creating features by hand. In this way, during the training process, the model will learn, along with the weights of the features, also the suitable feature mappings capable of creating some more powerful representation of the features. The optimization problem is now the following:

$$\mathbf{w}^* = \arg\min_{\mathbf{w},\boldsymbol{\theta}} \sum_{i=1}^{n} \ell\Big(y_i; \sum_{j=1}^{m} w_j \phi(\boldsymbol{x}_i, \theta_j)\Big) \tag{3}$$

One common approach is coding these new feature maps ϕ as the application of a non linear function φ applied to a linear combination of the features $\boldsymbol{x}$: $\phi(\theta^T\boldsymbol{x}) = \varphi(\theta^T\boldsymbol{x})$. The function φ takes the name of *activation function*; further details will be provided later in this section.

Artificial Neural Network: With the term Artificial Neural Network (ANN) we refer to any function of the form:

$$f(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{\theta}) = \sum_{j=1}^{m} w_j \phi(\theta_j^T \boldsymbol{x}) \tag{4}$$

However, this term is generally adopted to express non-linear functions that are nested compositions of linear functions, in turn, composed with non-linear ones. In other words, we can express the idea of a neural network as a set of perceptron units [16], the "neurons", organized in layers (see Fig. 1 for a simplified schema of a generic neural network architecture). The first layer of the neural network is the input layer, represented by the input vector $\boldsymbol{x}$ itself. L "hidden" layers follow: every single unit in the layer l, is connected to previous layer's $(l-1)$ units and the ones belonging to the following layer $(l+1)$. Finally, the last hidden layer, L, is considered the output of the model. Overall, this general model can be viewed as a combination of function composition and matrix multiplication:

$$g(\boldsymbol{x}) = \varphi^L(\boldsymbol{W}^L \varphi^{L-1}(\boldsymbol{W}^{L-1} ... \, \varphi^1(\boldsymbol{W}^1 x))) \tag{5}$$

where $\boldsymbol{W}^l$ is the matrix containing all the weights corresponding to the connections between neurons of the layer $l-1$ and l, and φ^l is the non-linear function used in the l^{th} layer, called *activation function*.

Main components of a neural network: In this paragraph, we list the main components, techniques, and methods about neural networks necessary for a better understanding of our work:

- **Fully connected layer**: is the basic idea in the structure of simple ANN, and consists of a layer having each neuron connected with all the neurons of the previous layer. Connections at each layer are parameterized with weights $\boldsymbol{W}^l$, where w_{jk}^l is the weight of the connection between the j^{th} neuron in the layer $l-1$ and the k^{th} neuron in the layer l;

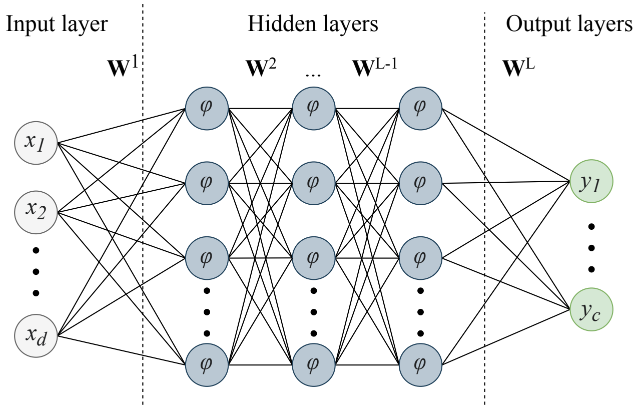

Figure 1: Simplified schema representing the architecture of a generic neural network.

- **Activation function**: an activation function φ is a function, applied at the neuron level, used to introduce non-linearities in the network. As described above, the introduction of non-linearities increases the expressiveness of the model. The most commonly used functions comprise the sigmoid function [17], the hyperbolic tangent, and the most recently widely adopted rectified linear unit (ReLU) [18];
- **Optimizer**: the optimization problem that we try to solve while training a neural network is the one given in Equation (3), and generally, it is a non-convex problem. Therefore, instead of solving the problem finding a closed-form solution as in convex optimization problems, it is commonly used an iterative method called "Stochastic gradient descent"(SGD). An SGD iteration consists of selecting a small part of the training data (starting from a single sample, up to considering a mini-batch of fixed size) and evaluating the loss function of the model on these data points. It follows an update of the model weights towards the opposite direction of the gradient of the loss function, and with a step size η (learning rate), aiming to reach a point of minimum (global or local). Each iteration can be summarized in the following expression:

  $$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \nabla_{\boldsymbol{W}} \ell(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x}) \tag{6}$$

  SGD learning rate η must be carefully tuned since an inappropriate value can compromise the convergence of the method or slow the learning process significantly. Some extensions and improvements of SGD tackle these problems by introducing techniques such as the adaptive learning rate and momentum. Specifically, in this work we will compare vanilla SGD implementation with Adam optimizer [19];

A well-known risk deriving from the usage of expressive models like neural networks is incurring in overfitting: limited training data and high model capacity contribute to learning the wrong relationships between data, which are just a result of sample noise. Then, the model will perform better and better on the training data, while worsening the generalization performance on new unseen data. Two techniques that help to face this problem are:

- **Regularization**: The idea behind regularization is that a high magnitude of weights usually implies more complexity of the model, and thus, a higher risk of overfitting. It is sufficient to introduce a penalty term that discourages the model from keeping high weights to prevent this problem. Equation 2 can be rewritten in the following way, where $\boldsymbol{W}$ indicates the matrix containing all the weights of the network and λ controls the contribution of the regularizing term in the overall objective:

  $$\mathbf{W}^* = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\boldsymbol{W}; \boldsymbol{x}_i, y_i) + \lambda \|\boldsymbol{W}\|_F^2 \tag{7}$$

- **Dropout**: Dropout is a technique proposed by Srivastava et al. [20] consisting of temporarily freezing some neurons from the network during part of the training process, with a probability p, along with all its incoming and outgoing connections. The overall effect is equivalent to adopting an ensemble of smaller capacity models, that both prevents overfitting and usually increases the total accuracy.

Further problems can arise while dealing with data that are not properly formatted. A first example is computing calculations with very high/small numbers. A frequent consequence is a numerical instability, often producing wrong results that generally slow down the convergence of the network. Moreover, a significant difference in magnitude between features can compromise SGD convergence: the weights' update in the iterative method is driven by the loss function gradient's direction, strongly determined by highly significant features. Low magnitude features will be hence neglected, possibly precluding the correct convergence to a local minimum.

Normalization is a frequently adopted solution to address these problems. This technique consists of scaling features to a common range, e.g., by subtracting their mean and dividing by their standard deviation. Two types of normalization can be employed in ANN:

- **Input normalization**: normalization is applied to the training data. More precisely, the mean and the standard deviation used for applying the transformation are calculated from all the training data points;
- **Batch normalization**: the same concept can be applied to every layer's output, to prevent facing this problem in the deeper layers of the network. Since the input to these layers varies along the training process and is usually processed in mini-batches, the mean and standard deviations must be calculated using different measures. We redirect the interested reader to the original work proposed by Ioffe and Szegedy in [21].

3.4 Convolutional Neural Network

In some scenarios, we might want a model that can perform a non-trivial task (e.g., image classification) and ensure some degree of shift, scale, and distortion invariance. These motives fostered the introduction of Convolutional Neural Networks (CNN). Such a type of ANN leverages the ideas of *local receptive field*, *shared weights* and *spatial sub-sampling*. The main idea is to compute a learnable and local filtering operation on the input image to leverage local and spacial information. In this context the following layers are commonly used:

- **Convolutional layer**: a convolutional layer is derived from a fully connected layer, operates with particular constraints on two consecutive 2D layers of neurons of the same size. It only connects each neuron to a localized set of neurons in the previous layers, usually in a $N \times N$ grid centered around it. Moreover, the weights of the connections to the previous layer are forced to be shared across neurons. This allows a convolutional layer to learn and represent a particular feature map, called "filter", that can identify structural patterns in the original image (e.g., edges and corners), as was often done by hand in the field of computer vision. By repeatedly stacking such filters on each other and connecting several convolutional layers, it is possible to extract richer features from the original image, with deeper layers being able to learn a high-level representation of complex objects;
- **Pooling layer**: In order to aggregate feature maps and to enable learning at different scales, pooling layers are usually inserted in CNNs. Following the idea that nearby inputs are strongly dependent on each other, it could make sense in some applications (e.g., image classification), to aggregate some units to decrease the width of the network and, thus, reduce the number of parameters. Pooling, indeed, consists of summarizing the convolutional layer's output, creating a reduced representation in which each unit is the result of a function applied to the referring unit block. Examples of such functions are Max Pooling, which maps a block to the maximum value contained, and Average Pooling, that instead takes the average of all the values.

LeNet-5: One simple and historical example of CNN was presented in LeCun et al. [22], under the name of LeNet-5. In this work, the network was successfully applied for handwritten character recognition in the MNIST dataset (see Sec. 5.1). The architecture of the network, as shown in Fig. 2, presents three convolutional layers (CONV), two pooling layers (POOL), and two fully connected layers (FC), in the following order: CONV-POOL-CONV-POOL-CONV-FC-FC.

4 Proposed method

In this section, we describe in detail how to consider the components of a neural network as they are players in a cooperative game. From that, we explain: how to compute the Shapley Value, why such an application makes sense, and which caveats should not be overlooked.

4.1 Overview

As discussed in previous sections, the necessity to understand and motivate the performance of a particular model is among the main challenges in modern machine learning. This has proved to be especially hard to do when the model is very complex or over-parametrized, as is the case with most neural network-based architectures. The principles motivating the surprising effectiveness of some techniques have often been found or proven years after their deployment in real-world applications [23]. Many applications are nowadays developed by relying on best practices or even on

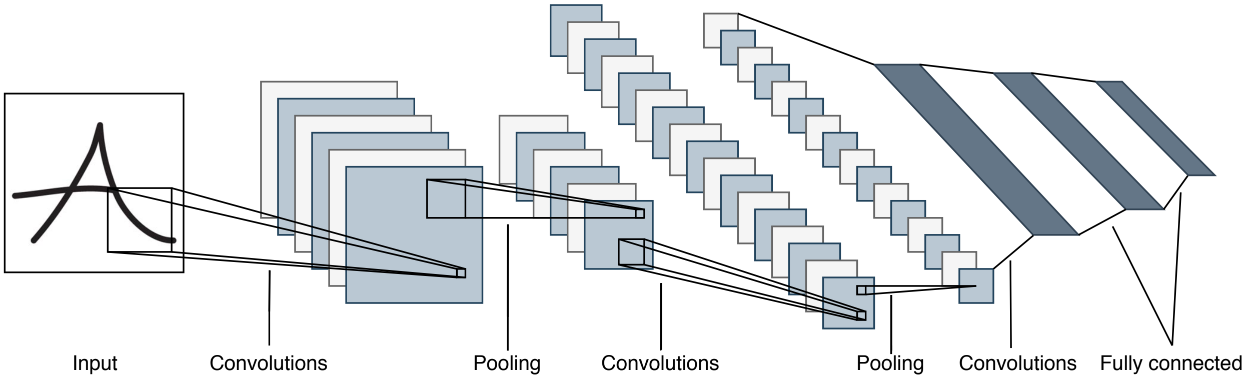

Figure 2: Simplified schema of the CNN “LeNet-5”, presented in LeCun et al. [22]

empirical experience. For this reason, we believe that it is of great importance to propose a method for evaluating the overall contribution of a single module or component to the performance of a model. In this paper, we will use the word $component$ to indicate building blocks or techniques used in conjunction with a model.

The main idea we are introducing is using the Shapley Value to understand the contribution of specific sets of components of a neural network to the overall performance of the model. We propose this method to investigate and measure which ones among these components are mainly responsible for the effectiveness of the network and, possibly, which ones are important in a vacuum and should be considered when designing a model for similar applications.

We believe our method is proposing a more structured approach compared to how such questions are often tackled. Today, this is often attempted empirically, usually by manually adding or removing components to a backbone architecture and comparing the model’s performance on a separate test set or by cross-validation. For computational reasons, when more than one component is to be evaluated, the procedure is often iterative or incremental, gradually adding the components that seem beneficial to the overall performance. Therefore, the contribution of a single component is often dependent on the particular order of components chosen in this routine.

We identify the main drawback of our method in its high computational costs, making it hardly feasible for more complex models. However, if we can overcome this problem, our method can give a more comprehensive evaluation on each component by considering its contribution not concerning a particular subset of other components, but in relation to all possible combinations. Therefore, this evaluation should be more general and robust, possibly even capable of answering broader questions regarding the effectiveness or the weaknesses of neural network-based approaches in general. For this reason, we believe it can still be beneficial and reasonable to apply it to simpler and faster models.

In the context of this paper, we will focus on neural networks-based models, and in particular, in the task of image classification. However, we believe that in principle, these ideas can be adapted to different models and, indeed, to different tasks.

4.2 Definition

We will now more formally introduce our method. Let us fix a machine learning task and a dataset split into training and test subsets. Given a backbone architecture A, let us define a $component$ as any modification to the model that can be added or removed in its entirety. For instance, the model’s learning rate cannot be defined as a $component$ since it is always required to be present. On the other hand, a technique such as dropout can be considered a $component$, as it can be arbitrarily activated or deactivated. Let us assume that all relevant hyper-parameters are fixed to reasonable values.

We define a set N of components, and a scoring function $v : \mathcal{P}(N) \to R$, where $\mathcal{P}(N)$ stands for the power set of N. The function v associates a subset of N to the test set accuracy of the fully trained backbone architecture A where only components in the subset of N were activated.

Since each component contributes to the overall performance of the network, we can also formally define N as a set of players in a cooperative game $G(v, N)$ where:

- v is the scoring function of our network;
- $N = \{1, 2, ..., n\}$ indicates the indexes of the components in our network.

A coalition C_j is a specific subset of active components, and the set of all coalitions includes every possible combination of network architecture that we can build from N. Since $C_j \subseteq N$ with $j \in \{1, .., k\}$, we say that the *grand coalition* corresponds to the architecture where all the n components are active and the *empty coalition* corresponds to a network with no active component. Therefore, we can then compute the Shapley Value ϕ_i of the i-th component as in Equation (1) and the results will reflect how each component contributes to the overall performance of the network.

5 Experiments

In this section, we describe our experimental setup and our choices regarding the process we adopted to test the proposed method. We will focus on a simple yet very common task, that is image classification, although, as we argued in the previous section, this method can also be applied to regression problems or generative models. Similarly, we will leverage a supervised approach, but our method is also feasible for unsupervised learning problems.

5.1 Datasets

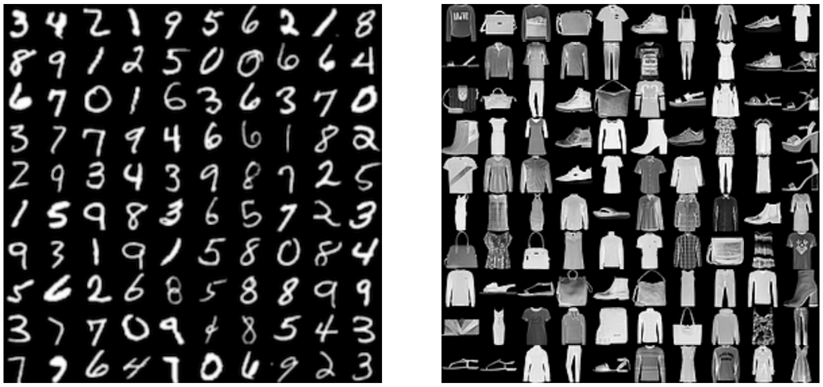

Figure 3: Samples from the MNIST (left) and FashionMNIST (right) datasets.

We test our assumptions on two datasets of great historical significance: MNIST[22] and FashionMNIST[24]. The MNIST dataset consists of 70.000 28x28 pixel black and white images of handwritten digits, 10.000 of which make up a test set. It was introduced and popularized by LeCun in 1998 and has been one of the first datasets to show the effectiveness of neural network-based approaches in image classification. On the other hand, FashionMNIST is a more recent collection of data that shares most characteristics with the former (resolution, color, and the number of classes), but represents arguably richer and more complex objects, as it contains images of different pieces of clothing, shoes and such articles. Samples from both datasets are shown in Figure 3.

We choose these two datasets mainly for their simplicity and low resolution, in order to enable a quick experimental turnover. We cannot, unfortunately, adopt massive and more modern datasets such as ImageNet[25] because of prohibitively long training time, but we suggest it as a good candidate for further experiments.

5.2 Architecture

As our backbone architecture A, we choose LeNet-5[22], which was purposely designed to tackle the MNIST dataset. As we already introduced the main ideas behind this architecture in Section 3.3, we will now mainly focus on the modifications we introduce. We mainly remove convolutional constraints on all layers as well as activation functions. Such changes are motivated by the adoption of particular components as players and are reversed when specific coalitions are selected.

To clarify what we mean by this, we will now introduce the seven components we have chosen. They range from modules to techniques and optimizer algorithms. All components apart from Adam, as well as the backbone architecture, can be seen in Figure 4.

- **Convolutional layers**: this component enables the convolutional structure in the first three layers of the network. When disabled, all layers in the network are fully connected. Enabling this component forces weights to be shared across filters, sets the weights of connections to distant elements of the previous layer to zero, lowers the complexity, and allows for spacial information to be gathered;

- **Activation**: a fundamental concept behind the functioning of neural networks is the introduction of non-linearities in order to fit complex functions. This component adds a sigmoid activation function after every layer for this very reason;
- **Normalization**: inputs are fed to the model as 2D arrays with 8-bit integer values. This component acts before the data is fed into the network and scales the input to be between 0 and 1.
- **Batch Normalization**: adds a batch normalization layer after each of the first three blocks of the network;
- **Dropout**: if enabled, dropout is applied after each of the first three blocks. The dropout probability is set to 0.2, a value that is often used in practice and that we empirically selected;
- **Regularization**: adds an L1 regularizer to the kernel of all layers, when activated;
- **Adam**: by default, our network is trained using SGD. Adam is considered an incremental upgrade as it uses the same principles but slightly corrects the gradient direction via moment estimation, as well as deploying additional techniques. For this reason, when we activate this component, we are toggling such improvements.

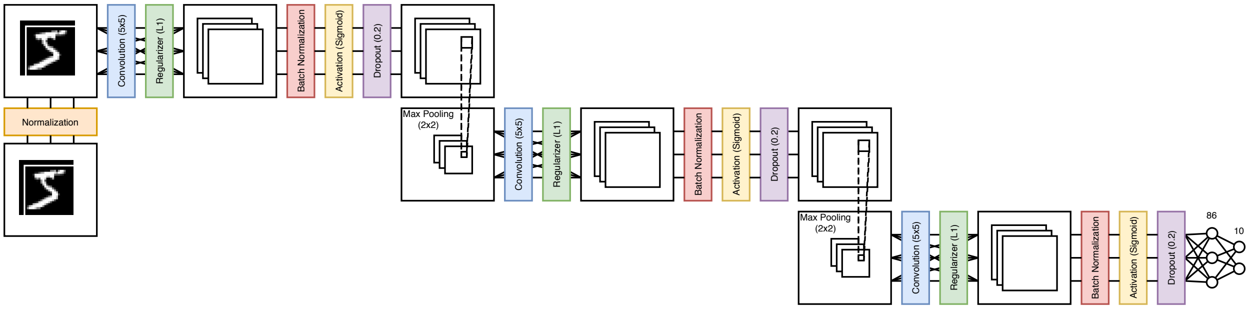

Figure 4: A simplified diagram of our adopted architecture. Blocks of the same color make up a component. Note that the last fully connected layers host more neurons than shown, having as many output neurons as the number of classes.

5.3 Practical choices

In order to get reliable results, we fix hyper-parameters across all runs. We selected such hyper-parameters to achieve high test set accuracy on the complete model, in which all components are activated, while not requiring an impractical training time.

From a practical standpoint, we implement the model described above in Python using the TensorFlow2[26] library. Our implementation will be packaged as a Jupyter Notebook and made available at `github.com/OsD977/shapley-value-neural-network`.

We retrieve each possible coalition by computing the power set of N, which is the set of the seven components. For each coalition C_j, we build a backbone model in which we only activate the components present in the coalition. We train each model with early stopping and compute the test set accuracy when the validation loss plateaus or the maximum number of epochs is reached. We, then, save this value for all coalitions.

This procedure requires training 2^7 models, each using a different combination of components. In total, around an hour is required to complete the training on an Nvidia Tesla K80 Graphical Processing Unit.

Once we collect the accuracy value of each coalition, we compute the Shapley Value for each component using the formula in Equation 1. This global procedure is repeated ten times for each dataset to ensure the robustness of the method.

6 Results

In this section, we give general considerations regarding the behavior of the accuracy and the Shapley Value obtained from our tests. We also highlight the contribution that each component of our network has concerning the accuracy of our model predictions.

6.1 General observations

We show our results averaged over the runs in Figure 5 for each component, and extensively in Tables 2 and 3 in the Appendix. Note that, although the values computed across all runs show a stochastic behavior, the rankings remain, for the most part, consistent. Therefore, it is possible to draw conclusions on the effectiveness and on the findings of our method. To maintain a reasonable runtime, we chose not to have more runs, although that could have given more significant results. This remains a direction in which future works can expand.

In Figure 6, we report on the relationship between the test accuracy of the neural network and the cardinality of the subset of active components (the values used to produce the histogram are listed in Table 1 in the Appendix). We note that the trend is generally increasing, with diminishing returns as we approach the grand coalition of size seven. As a matter of fact, in most cases, this coalition represents an over-engineered approach, which fails to improve accuracy and is detrimental. Overall, the plot is a good indicator of the validity of our experimental pipeline.

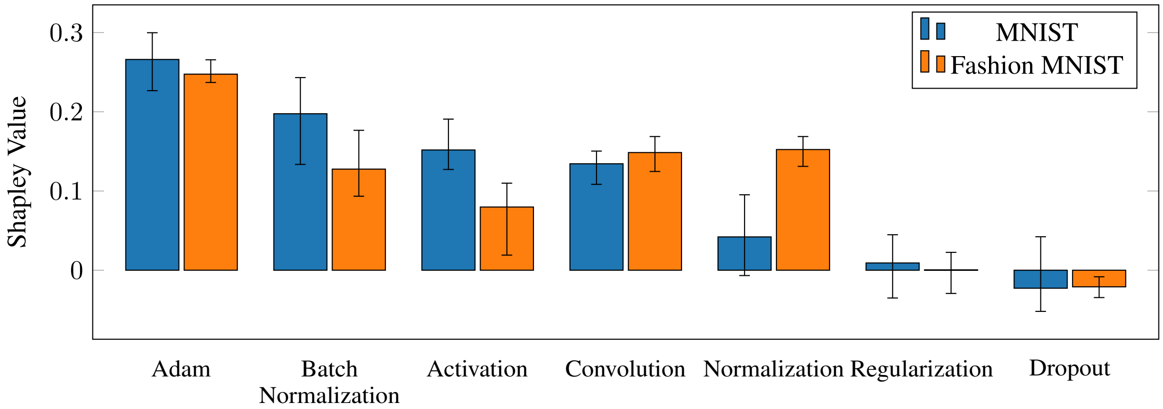

Figure 5: Average Shapley Value of each component of the two datasets. Error bars show the minimum and maximum Shapley Value across runs.

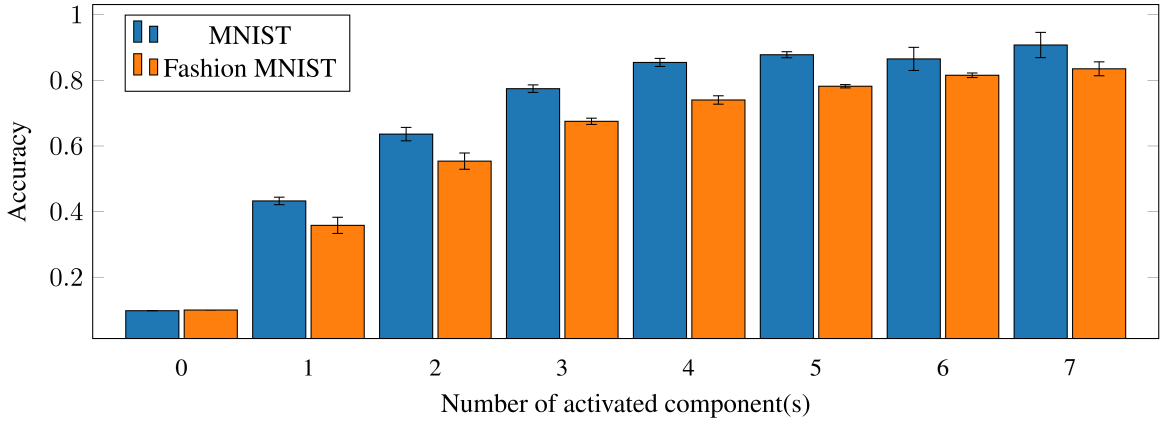

Figure 6: Average test accuracy for the set of models with a precise number of active components, from zero (no active component) up to seven (every component is active). Errors bars show the standard deviation across five runs.

6.2 Component-wise considerations

We can now analyze the general trends across both datasets, starting from the ones with the highest Shapley Value, and we discuss the obtained results.

Surprisingly the highest Shapley Value across both datasets is obtained by the *Adam* component. This result is likely because the loss function for this task is strictly not convex due to the lack of strong regularization. We must remember that considering half of the runs does not force a convolutional structure in the first layers. The fact that *Adam* has the highest Shapley Value shows how fundamental the choice of an optimizer is in this application and how reliant on it the overall performance remains.

As expected, the component enabling convolutions also gets significant credits for the performance of the network. Many modern achievements in computer vision and image processing have strong foundations in the usage of convolutional operators and filters to reconstruct local patterns and create rich representation while limiting the model's complexity. Conversely, MNIST datasets are relatively simple and can also be tackled with decent results by fully connected networks or simpler methods. On the other hand, the slightly higher Shapley Value on FashionMNIST hints that the importance of convolutions would be even more clear on more complex datasets.

The Shapley Value also attributes great significance to normalizing agents (both in preprocessing and in batch normalization layers). This trend has often been observed in practice and shows the benefits of regularization on the model. Normalizing signals corresponds to performing a form of alignment across all different samples, enabling the classifier to focus on similar representations. Of course, *Batch Normalization* has many other benefits, such as enabling easier and faster training, as already mentioned.

On the negative side of the spectrum we find *Regularization* and *Dropout*. Both techniques are used in practice to fight overfitting, which usually arises with more powerful models, but they are recently used less and less. While both datasets have a considerable number of samples while being quite simple, this also contributes to preventing overfitting. For these reasons, we believe that especially *Dropout* does not serve its purpose in most coalitions, as it is shown to be detrimental to the final result.

The main difference across the two datasets is that the *Normalization* component is given much higher importance for the second dataset. This may be related to the fact that FashionMNIST represents a harder task with richer information, while MNIST is simple enough not to benefit from some preprocessing.

We finally note that the sum of Shapley Values across the set of components corresponds to the difference in accuracy between the backbone model (when all components are not active) and the model associated with the grand coalition. This is one significant way our approach differs from the straightforward application of the Shapley Value in game theory. In our experiments in general $v(\emptyset) \neq 0$. As a matter of fact, it is to be expected that the backbone model's accuracy will be greater than zero even when all components are not active, as predictions of an overly simple model can hardly be worse than random guesses. However, this does not hurt our general assumptions, as we do not consider absolute values as much as rankings.

7 Conclusion and further developments

In this paper, we introduced the usage of Shapley Value for evaluating the contribution of single components to the overall performance of a neural network-based model. Our method builds on game-theoretic foundations to provide a comprehensive analysis of the utility of components concerning the surrounding architecture. We designed a single pipeline and applied this method in practice to show its effectiveness in distributing credits. Our results were consistent and significant despite the limited experimentation we were able to carry out due to computational constraints. Thus, we are confident that our method can be applied to produce useful insights and to provide a more general understanding of what parts of neural network architectures are the most effective for their success.

In conclusion, we reckon that this method should be investigated more in detail. There are many different directions, among which we would like to highlight some. There is definitely room for experimentation on larger datasets (such as ImageNet) or more complex models (e.g., those belonging to the ResNet[27] family). As a last note, we believe it would also be of great interest to explore whether it is possible to transfer some form of mathematical guarantees on the Shapley Value to the field of machine learning.

References

[1] Lloyd S Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317, 1953.

[2] Jianyu Miao and Lingfeng Niu. A survey on feature selection. *Procedia Computer Science*, 91:919–926, 12 2016.

[3] Shay Cohen, Gideon Dror, and Eytan Ruppin. Feature selection via coalitional game theory. *Neural Computation*, 19(7):1939–1961, 2007.

[4] Jihong Liu and Soo-Young Lee. Study on feature select based on coalitional game. In *2008 International Conference on Neural Networks and Signal Processing*, pages 445–450. IEEE, 2008.

[5] F. Leon. Optimizing neural network topology using shapley value. In *2014 18th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 862–867, 2014.

[6] S Sasikala, S Appavu alias Balamurugan, and S Geetha. Improving detection performance of artificial neural network by shapley value embedded genetic feature selector. *Neural Network World*, 26(2):175, 2016.

[7] Fatiha Mokdad, Djamel Bouchaffra, Nabil Zerrouki, and Azzedine Touazi. Determination of an optimal feature selection method based on maximum shapley value. In *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 116–121. IEEE, 2015.

[8] John Von Neumann and Oskar Morgenstern. Theory of games and economic behavior, 2nd rev. 1947.

[9] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[10] Jun Shao. Linear model selection by cross-validation. *Journal of the American statistical Association*, 88(422):486–494, 1993.

[11] Michael W Browne. Cross-validation methods. *Journal of mathematical psychology*, 44(1):108–132, 2000.

[12] Shay Cohen, Eytan Ruppin, and Gideon Dror. Feature selection based on the shapley value. pages 665–670, 01 2005.

[13] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Igor Kononenko et al. An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research*, 11(Jan):1–18, 2010.

[15] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in neural information processing systems*, pages 4765–4774, 2017.

[16] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[17] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[18] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[22] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. 1998.

[23] Robert Schapire. *Explaining AdaBoost*, pages 37–52. 10 2013.

[24] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[25] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[26] Various authors. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

A Experimental data

Table 1: Accuracy of our model averaged on subsets of increasing size, evaluated over five different runs on both datasets.

	MNIST					**FashionMNIST**				
	Run 1	Run 2	Run 3	Run 4	Run 5	Run 1	Run 2	Run 3	Run 4	Run 5
0 component(s)	0.098	0.098	0.098	0.098	0.098	0.100	0.100	0.100	0.100	0.100
1 component(s)	0.417	0.426	0.434	0.451	0.436	0.316	0.372	0.369	0.347	0.387
2 component(s)	0.640	0.624	0.611	0.634	0.672	0.537	0.580	0.514	0.562	0.576
3 component(s)	0.759	0.776	0.794	0.776	0.768	0.692	0.673	0.665	0.668	0.678
4 component(s)	0.863	0.852	0.837	0.849	0.873	0.721	0.759	0.748	0.735	0.739
5 component(s)	0.860	0.879	0.886	0.885	0.881	0.779	0.787	0.789	0.780	0.776
6 component(s)	0.844	0.823	0.898	0.916	0.847	0.820	0.816	0.816	0.803	0.823
7 component(s)	0.848	0.926	0.877	0.945	0.942	0.843	0.860	0.803	0.851	0.818

Table 2: Shapley Values of the different components of our architecture, evaluated over ten different runs on MNIST dataset.

	MNIST									
	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Activation	0.144	0.136	0.170	0.139	0.127	0.129	0.145	0.173	0.191	0.164
Adam	0.227	0.271	0.293	0.275	0.256	0.256	0.300	0.268	0.239	0.276
Batch Norm.	0.183	0.134	0.228	0.206	0.236	0.190	0.152	0.194	0.210	0.243
Convolution	0.108	0.143	0.150	0.146	0.126	0.128	0.127	0.136	0.129	0.148
Dropout	-0.052	-0.018	-0.037	-0.032	-0.031	-0.022	0.042	-0.034	-0.003	-0.040
Normalization	0.010	0.095	-0.007	0.011	0.029	0.034	0.083	0.046	0.063	0.056
Regularization	-0.035	0.045	0.020	0.028	0.009	0.035	-0.021	-0.004	0.015	0.001

Table 3: Shapley Values of the different components of our architecture, evaluated over ten different runs on FashionMNIST dataset.

	FashionMNIST									
	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Activation	0.071	0.073	0.091	0.097	0.019	0.056	0.096	0.092	0.110	0.094
Adam	0.249	0.240	0.251	0.247	0.266	0.255	0.238	0.247	0.245	0.237
Batch Norm.	0.107	0.177	0.127	0.129	0.139	0.119	0.146	0.096	0.093	0.143
Convolution	0.143	0.125	0.147	0.154	0.161	0.156	0.132	0.161	0.169	0.138
Dropout	-0.014	-0.034	-0.012	-0.008	-0.035	-0.012	-0.020	-0.027	-0.026	-0.022
Normalization	0.131	0.168	0.157	0.138	0.169	0.158	0.155	0.163	0.154	0.132
Regularization	0.023	-0.010	-0.005	-0.008	0.006	0.012	0.012	-0.029	0.005	-0.003