IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 22 March 2024, accepted 14 April 2024, date of publication 22 April 2024, date of current version 29 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3391889

RESEARCH ARTICLE

GenericSNN: A Framework for Easy Development of Spiking Neural Networks

ALBERTO MARTIN-MARTIN[1,2], MARTA VERONA-ALMEIDA[1], RUBÉN PADIAL-ALLUÉ[2], JAVIER MENDEZ[1], ENCARNACIÓN CASTILLO[2], AND LUIS PARRILLA[2], (Member, IEEE)

[1]eesy-Innovation, 82008 Unterhaching, Germany
[2]Department of Electronics and Computer Technology, Faculty of Sciences, University of Granada, Granada, 18071 Andalusia, Spain

Corresponding author: Alberto Martin-Martin (alberto.martin@eesy-innovation.com)

This work is part of the project TED2021-129938B-I00, funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/ PRTR. At the same time, part of this work has been developed in the context of the project "ANDANTE" (European project number 876925) as part of the European call H2020-ECSEL-2019-2-RIA and also partially financed by the German Federal Ministry of Education and Research (BMBF).

ABSTRACT Spiking Neural Networks (SNNs) have emerged as a prominent paradigm for brain-inspired computing, capable of processing temporal information and event-driven data in an efficient and biologically plausible manner. However, their revolutionary and complex nature is one of the key reasons why SNNs are not yet a widely used approach in contrast to traditional Artificial Neural Networks (ANNs). In this paper, we present a comprehensive SNN framework that offers user-friendly implementation. It has been designed so that it is compatible with other well-known software tools for data science, being easy to integrate with them. We showcase the versatility of the framework by applying it to various well-known benchmarking datasets, including image processing of handwritten numbers, time-series forecasting and an advance use case for speech recognition, achieving competitive results compared to traditional ANNs. Our SNN framework aims to bridge the gap between neuroscience and artificial intelligence, empowering researchers and practitioners with an accessible tool to explore the potential of neuro-inspired computing in advancing the field of AI.

INDEX TERMS Deep neural networks, edge computing, framework, spiking neural networks.

I. INTRODUCTION

In recent years, the utilization of Spiking Neural Networks (SNNs) has gained substantial popularity across diverse research fields such as Edge Computing (EC) and Internet of Things (IoT). SNNs have demonstrated the capability to achieve key milestones such as enhanced power efficiency, reduced latency, and high-performance computing while maintaining the accuracy achievements of conventional Deep Neural Networks (DNNs) [1], [2], [3], [4], [5].

SNN models adopt an approach similar to the biological processes in the human brain, encoding information into discrete spikes that are subsequently propagated through the network. In the case of SNN models, the network processes these spike signals on its structural configuration until the cumulative internal values reach a defined threshold. At this point, an internal neuron spike is triggered. This mechanism aligns more closely with the information processing paradigm in biological brains compared to the conventional Artificial Neural Networks (ANNs) [6], [7].

However, due to their revolutionary and complex nature, these networks require a deep comprehension of Deep Learning (DL) techniques to accurately define the SNN network architecture as well as its internal parameters. These SNN parameters include variables such as neuron activation thresholds, activation functions within neurons, the selected number of temporal samples for the inference, and temporal characteristics of neurons among many other parameters [8], [9]. These parameters are not present in the context of traditional DNN models, leading to an increase of the complexity for non-expert users who may struggle to understand the complex relationships among these

The associate editor coordinating the review of this manuscript and approving it for publication was Ludovico Minati.

variables. As a result, this high-complexity can lead to wrong configurations.

This complexity is one of the key reasons why, despite their efficacy, SNNs are not used yet as the default approach for DNN implementation at the network edge in contrast to alternative techniques including DNN quantization, pruning, or knowledge distillation. To address this issue, several software packages have been created recently, aiming to facilitate SNN development. However, most of these frameworks have a task-oriented structure and workflow, such as the ones proposed in [10], [11], and [12]. Nevertheless, these frameworks still require expert knowledge for their use due to the highly specific case-oriented style of them. Another important difficulty of these frameworks lies in their complexity to configure the SNN as well as the simulator to run these intricate networks.

In this work, a novel framework to design, evaluate the viability of the designs, train and evaluate SNN models based on the Nengo [13] and NengoDL [14] packages is depicted. These mentioned packages provide a simulator and utilities to build and deploy SNNs. Nevertheless, they are designed for experts, thus being difficult its use by non-experienced users. The framework proposed in this paper has been developed aiming to ease the usage of SNN models by non-specialist users. To achieve this goal, the steps to generate a SNN model have been simplified to replicate the same steps required when defining a traditional ANN. Furthermore, the internal names of the proposed framework follow the naming convention used in the TensorFlow package [15] in order to increase their interoperability capabilities with other Machine Learning (ML) packages such as SciPy [16].

For clarification, we summarize the contributions of this research paper as follows:

- An easy framework to design SNN is proposed. The workflow pipeline when using this framework is highly similar to popular frameworks such as Keras [17] or TensorFlow [15] to ease the usage of SNN by non-expert users.
- The proposed framework was developed taking into consideration its compatibility with external popular ML Python packages. This enables the usage of state-of-the-art techniques with the proposed framework.
- This framework has been evaluated in three different use cases. These use cases have been selected to show how this framework can be easily used to design classification-oriented as well as prediction-oriented SNN models.

The remaining sections of this work are structured as follows: Section II presents the most relevant software tools in the literature for similar tasks. After this, Section III discusses the proposed framework for a deeper understanding of its capabilities. In Section IV, three use cases are presented in order to further explain how the proposed framework can be used to design SNN models for multiple applications. Finally, Section V focuses on conclusions of this work.

II. STATE OF THE ART

In this section, some of the most relevant frameworks for SNN model creation are discussed, aiming to present the current state of the art of this field. Furthermore, these frameworks are compared with the one proposed in this paper regarding their capabilities and limitations.

As indicated in Section I, various other publications have also proposed similar tools or frameworks for SNN model creation. For instance, C. Li introduced a framework in [10] that mainly focuses on optimizing the conversion process from ANNs to SNNs while reducing the time to represent the information of the SNN by quantifying the ANN network. Nevertheless, this tool is intended for specific applications such as image processing (it reports 70.18% accuracy on the ImageNet [18] dataset), and does not face neither the complexity of the network generation process nor the compatibility with external tools. As a consequence, it is highly restrictive regarding both the techniques and the creation process for SNN models.

Similarly, in [12] a framework focused on direct SNN quantization rather than relying on a quantized ANN model is described. In this proposal, all parameters involved in the SNN configuration for the parameter quantization are taken into consideration. In practical experiments, the framework demonstrated a remarkable memory footprint reduction of four times in comparison to baseline models while facing only a 1% accuracy drop on the MNIST [19] dataset. However, despite its high-performance results regarding memory optimization, this framework remains centered on a specific task of the SNN model creation without addressing general challenges of SNN model development.

Further contributing to this research field, Zhou proposed a novel approach in [11]. In this paper, the author proposed a framework to convert Transformer Networks into SNN models. This approach combines the latest advances in DL with biologically plausible structures, aiming to balance the SNN's energy-performance benefits with the robust accuracy of Transformers. As a result of this approach, this framework achieves a classification accuracy of 74.81% on ImageNet, surpassing state-of-the-art SNN results. However, it is important to point that this technique aims to improve the performance without considering the complexity or the final user of the framework. This set it apart from the user-centric approach embodied by the framework proposed in this paper.

Differently from the previous papers, other authors faced the SNN generation complexity problem by developing general frameworks, similar to the followed approach in this paper. Notably, the company SynSense introduced the "Rockpool" tool [20], which aims to increase the SNN usage among non-expert users. Rockpool has numerous examples of how to use this framework for audio classification tasks as well as adversarial training. At the same time, all the documentation and code to use this framework can be easily found on the website of the company, leading to an easy to understand package. However, while Rockpool aims

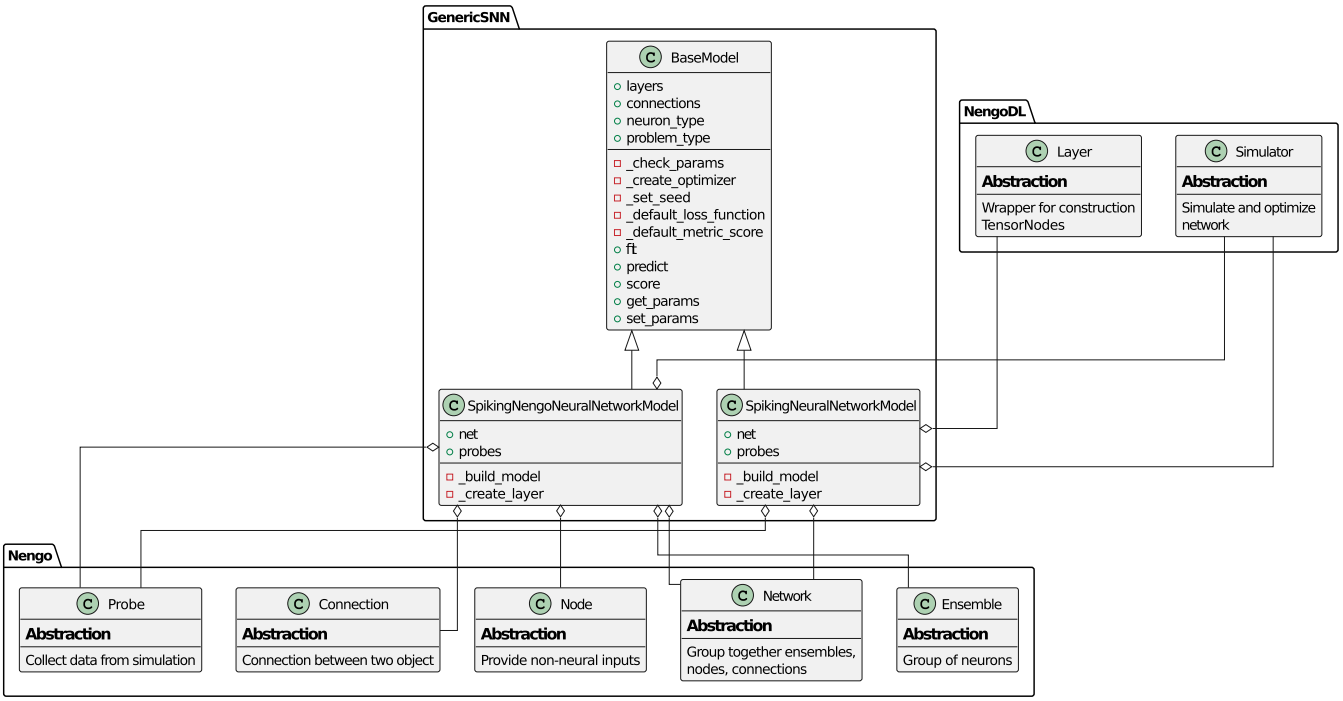


FIGURE 1. ClassDiagram of the proposed framework.

to simplify the SNN utilization, it diverges from standard naming conventions used in other relevant AI packages. This creates a bottleneck when trying to use SNN for state-of-the-art applications.

Additionally, Liu presented a framework for SNN generation in [21]. This framework not only adapts an ANN's activation function for a spike-approach activation but also aims to reduce information loss resulting from the ANN-SNN conversion, thus leading to a reduction of the accuracy degradation. However, it is essential to acknowledge that this framework offers only a description of the approach followed by the author and not a full framework. At the same time, this tool is not intended for general users but expert ones who can replicate the approach proposed by the author neither consider the integration of the framework with other external packages.

Finally, one of the frameworks worth mentioning within the state-of-the-art is the one developed by Fang, called SpikingJelly [22]. One notable feature of SpikingJelly is, as well as other frameworks like Nengo, its focus on automatic differentiation, a crucial component for gradient-based training of deep learning models. This allows to optimize the parameters of the SNN and thus to solve complex traditional deep learning problems by using this type of networks. The framework also includes two separated modules, a very extensive and detailed one for activation based learning, and another one for timing based learning, which only includes the encoding process and a neuron model. The type of neurons that could be implemented in SpikingJelly are more detailed than the simplified neurons from Nengo and the framework proposed in this paper. As an example, its documentation shows how to build and train a spiking convolutional network to solve the Fashion-MNIST problem [23]. However, the necessary amount of lines of code for this application is larger compared to those required by our proposed framework, as it will be shown in Section IV. The reason behind this is the high level of detail required by the configuration of SpikingJelly since it aims at developing application specific state-of-the-art techniques in comparison with the proposed framework. This requires a deep understanding of SNNs, in contrast with the simplified configuration available in our framework, aiming at non-expert user, where these configuration steps are not strictly necessary.

As a summary, it is important to remark that these pre-existing frameworks have not fully addressed the discussed challenges. One of the reasons for this is their approaches are oriented to a specific application or platform, thus leading to limitations on their generality and user-friendliness. In contrast, the proposed framework in this paper aims to solve the limitations in existing solutions by offering a user-friendly approach by reducing the complexities of SNN model creation while integrating capabilities for compatibility with existing relevant AI packages.

In the following section, the proposed framework will be described in a detailed level. This will allow readers to better understand the framework's capabilities in comparison with the proposal found in the state of the art.

III. GENERICSNN FRAMEWORK

The present section introduces the proposed GenericSNN package, which comprises three essential classes, each

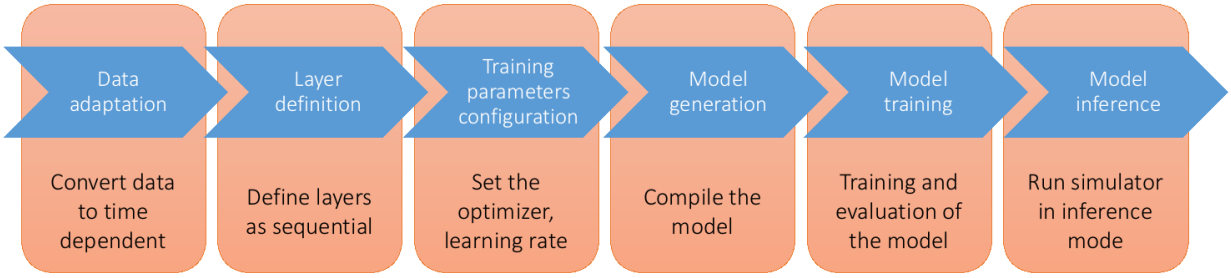


FIGURE 2. Workflow of the proposed framework for SNN model generation, training and inference.

designed to facilitate specific functionalities within SNN modeling. Firstly, the core of the package is the *BaseModel* class, serving as the parent class from which the other classes inherit crucial utility methods. These methods play a vital role in tasks such as creating the optimizer for the training process and visualizing spiking patterns. Moreover, the package encompasses a dedicated class aimed at creating SNN models exclusively employing Nengo [13] objects (except for the Simulator which is common for both classes and taken from NengoDL [14]), thus enhancing model construction efficiency. Lastly, an analogous class is included, allowing for the creation of models utilizing NengoDL objects, and providing an alternative approach for network design and evaluation. Figure 1 shows a class diagram where the relationship between Nengo and NengoDL with our classes is detailed. Through this package, researchers and practitioners can effectively develop and analyze SNN models in a modular and flexible manner.

Despite the different classes, all of them follow a common flow regarding the implementation and execution of an SNN. Figure 2 shows the proposed diagram flow, which is highly similar to other common deep learning frameworks such as Keras or TensorFlow [15], [17]. The first step is *Data adaptation* since this type of model requires the input data to have a temporal dimension. Therefore, the user has to convert its data to this format. There are different options to accomplish this step, and we have selected for our framework a technique consisting on dividing the samples using a time window in order to generate multiple time samples of a standardized size. However, this is only possible when the data has a temporal evolution. When working with individual samples, the network still requires them to have a temporal dimension. Our approach for this case is to repeat the same sample for the time window length.

Apart from the data conversion to time domain data, data must be converted into spikes, a fundamental aspect of SNN modeling, for the correct processing of the data when implementing SNNs models. This, as well as the remaining steps of our flow, will be further explained in the following subsections.

A. NEURAL CODING SCHEMES

Neural coding schemes are used to convert input analog or digital data (e.g. the value of the pixel of an image) into spikes that are transmitted to the excitatory neurons [24]. Following biological brain approaches for information codification, there are various encoding schemes used to represent environmental information such as acoustic, visual, or somatic data into electrical spikes.

Between all the different approaches, two main coding schemes can be differentiated: rate coding and temporal coding. The first one, rate encoding, has been proven to be the predominant technique to transmit information within nervous systems [25]. Rate encoding can be further divided into three subcategories: count, density, and population rate encoding, being the last one the coding schemes used by Nengo [13].

Following this approach, the information is encoded by a population of neurons whose firing rate is described by a tuning curve. The superposition in a large population can encode single numbers, vectors, or even function fields [26]. In the decoding process, the different trains of spikes are first filtered by an exponentially decaying filter accounting for the process of a spike generating a postsynaptic current, and then they are all summed together [13]. Figure 3 summarizes this encoding and decoding approach. The left column shows the encoding process, in which a signal is encoded by a population of eight neurons whose firing rate depends on the tuning curve configured using the *uniform* option provided by the framework. The top graph in that column shows the tuning curves of the different neurons, where each of the colors is associated with each of the neurons, using the same colors for the rest of the graphs. The middle graph of the encoding column shows the input signal, which is a simple sinusoidal wave. Finally, the bottom graph of that column represents the spikes of each neuron, and it can be seen how, when the value of the input signal approaches one, the firing rate of the neuron represented in blue is maximum meanwhile the one from the neuron represented in gray is null. The right column shows the decoding process applied to reconstruct the original signal. The top graphs of the figure shows the same train of spikes of each neuron generated in the enconding process explained before. The middle graph shows the same train of spikes after apply a low-pass filter of the synapses. Finally, the bottom graph shows how, by adding up in a weighted way the components from the different neurons, the original signal is reconstructed. Those neurons which increasing tuning curve have a positive weight, while the others have a negative one.

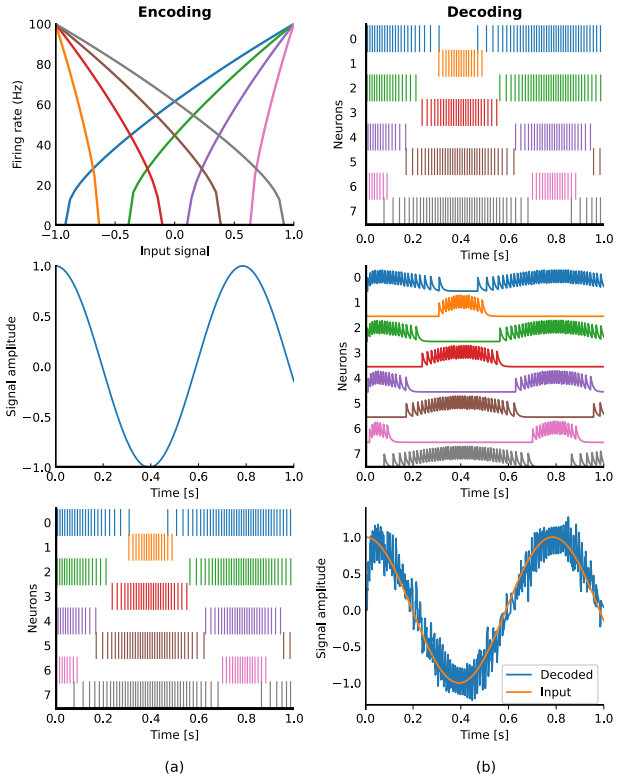


FIGURE 3. Spikes coding scheme, a) encoding process, b) decoding process. In all subfigures, except in the left-middle and right-bottom ones, each color is associated with a neuron.

In our framework, the number of neurons in the population of each layer is specified by the parameters *n_neurons* when defining the layer itself. As well, the maximum firing rates of the neurons are specified when the *SpikingNengoNeuralNetworkModel* class is instantiated. In the case of the interception of the tuning curves (the point when the activity of each neuron turns to zero), the framework allows the possibility to choose between two options: setting all the curves to zero interception, or distributing the interceptions uniformly between -1 and 1. However, the framework provides default values for these parameters, in case the user is not sure about how to configure them. The values of these parameters have been selected based on a prior study to ensure they fit general applications, as it will be shown in Section IV.

B. NETWORK DEFINITION

The way of defining the SNN will depend on which class the user is implementing. As previously commented, the class defined in *SpikingNengoNeuralNetworkModel* uses purely Nengo objects. This implies that the user must specify the layers in terms of Ensemble object parameters (where an Ensemble is a population of neurons) and the connection between each Ensemble in terms of the parameters of the operation that is performed between two populations of neurons (e.g. the parameters of a convolution).

As a consequence of this structure for the layer definition, the user has a deeper control over the internal configuration of the connections among the layers and neurons. This leads to the possibility of further optimizing and adapting the SNN topology to a specific use case. However, a deeper knowledge of the topic is required in contrast with the *SpikingNeuralNetworkModel* class.

When using *SpikingNeuralNetworkModel* class, only a definition of the layer structure is required for the construction of the network topology, in a more similar way to a conventional ANN when defined with Keras. Specifically, the layer structure definition is based on a list creation where each element of the list is a layer. These layers only require traditional parameters such as the activation function or the number of neurons.

After the layer definition, as shown in Figure 2, the internal parameters for the training of the model must be configured. It would be necessary to specify neuron-related parameters, the optimizer to use as well as the loss function, among other training parameters. One of those configurable parameters is *problem_type*, which will be used to set the internal parameters to fit the specific use case, based on the input provided by the user. The options for these parameters as well as the effect on other internal parameters are shown in Table 1.

TABLE 1. Internal configuration of the training parameters based on *problem_type* variable.

Problem type	Loss function	Metric
Regression	Mean Squared Error	Mean Squared Error
Binary classification	Binary crossentropy	Binary accuracy
Multiclass classification	Sparse categorical crossentropy	Sparse categorical accuracy

Once the training parameters are defined, the user can instantiate the class to generate the SNN object for the training, evaluation, and inference. This will trigger the internal generation of the model without having to deal with complex syntax. Additionally, Nengo Probe objects will be instantiated, whose purpose is to collect data from the simulation (spike data, represented values, neuron voltages, etc.), for every layer automatically. This will allow the user to debug the network in case any problem is detected during the training phase. In the following subsection, details about the training and evaluation of the SNN models will be provided.

C. TRAINING AND EVALUATION

For the training, the two mentioned classes of this framework lie on NengoDL which sits at the intersection between deep learning and neuromorphic modeling tools, combining Nengo and TensorFlow [14]. Due to the use of TensorFlow behind the hood, it is possible to train the network in both CPUs and GPUs, thus allowing users to train their models in different scenarios. When high-performance resources are available, such as large GPU servers, the training phase can be executed in the GPUs in order to parallelize this process,

leading to a time reduction for this step. In other scenarios, the training could also be executed in CPUs without having to adapt the script.

NengoDL provides a simulator that allows optimizing the network parameters. Most of the optimization methods require the model to be differentiable, which can represent an issue in this kind of models. However, in most of the cases good results could be achieved by rate-based approximation of the spiking neuron model, which is the approach followed by the NengoDL simulator [14]. By employing this approximated model it is possible to use automatic differentiation methods following the same techniques as those utilized in a traditional deep learning model.

To use this NengoDL simulator, it is necessary to compile the network, provide a valid optimizer, as well as provide a large number of configuration values. All of this is simplified in our framework, creating automatically a configuration for the simulator based on the arguments provided when the class is instantiated. Besides, all the data produced during the simulation will be gathered making use of the probes explained before. This information can be later accessed easily by calling the methods defined in our classes.

The parent class of our framework, *BaseModel*, provides several methods that allow plotting the data generated during training. The data gathered vary from the voltage of the spikes, the number of spikes that emits the population of neurons of the different layers, as well as the temporal decision results based on the spikes. By providing the layer index as an argument for these methods, it is also possible to access the temporal results in each layer and not only in the last one.

Upon optimization, both classes incorporate two essential methods, namely *predict* and *score*. In order to get the prediction and the loss function score of the model, it is also necessary to use the NengoDL simulator, but in this case with a different configuration that allows making predictions instead of optimizing the parameters. This configuration is also provided automatically, making it transparent to the user, based on the argument *problem_type* mentioned before.

D. COMPATIBILITY WITH EXTERNAL PACKAGES

This framework was initially planned to be fully compatible with other well-known data science frameworks. This implies that a naming convention of the different methods that the class implements should be followed. Some of the methods are standard in all the frameworks, for example, the aforementioned *fit*, *predict*, and *score* methods. This must be taken into consideration because some utility functions make use of them to provide external capabilities.

This is especially relevant when aiming to increase the popularity of SNN models for emerging research topics. This approach enables the user not only to make use of external packages for hyper-parameter optimization but also integrate new techniques into these models, promoting broader usability within the machine learning community.

For the first case, the hyper-parameter optimization process, numerous approaches can be followed. The goal of this process is to search for the parameters that minimize the loss function implemented for the user's application. Some of these parameters are not easy to study, such as the number of neurons of each layer, or even the proper loss function. However, the selection of these parameters has a high impact on the performance results of the resulting SNN model.

One of the external packages that can be used for this task is *scikit-learn* [16]. This package, as previously commented, makes use of some of the methods defined in the proposed frameworks such as *score*, *get_params* or *set_param* in order to provide capabilities to study the hyper-parameter configuration, such as the *GridSearch* functionality. Apart from this, this adherence to scikit-learn's interface specifications renders the class fully compatible with its estimator and predictor interfaces.

Regarding the integration of emerging AI techniques in this framework, due to its dependency with TensorFlow, the usage of novel layer configurations as well as loss functions or metrics will be direct.

Some of these capabilities, as well as the implementation of the discussed classes of this framework, will be shown in Section IV. There, two different well-known applications will be covered for a deeper understanding of the coding structure of our framework. The achieved results will also be compared with the state-of-the-art results based on traditional ANN models.

IV. USE CASES

In this section, three different applications based on the proposed SNN framework will be explained and discussed to further understand how the framework is used in comparison with other approaches previously commented in Section II. In these use cases, not only the direct application of the framework to generate SNN models will be shown but also how it can be used with other external AI packages to integrate external capabilities. The two first examples will introduce the usage of the proposed framework for simple applications while the last example will focus on a more advanced use case.

The examples with the full scripts that will be explained in the following subsections can be found in the same GitHub repository where the full framework code is located.

A. MNIST CLASSIFICATION USE CASE

In this subsection, an application based on the proposed framework for classification will be commented. In order to better demonstrate the capabilities of the framework, a commonly used classification dataset has been selected. The selected dataset is the Modified National Institute of Standards and Technology (MNIST) handwritten digital dataset [27]. This dataset contains handwritten digits (from 0 to 9) in grayscale images of 28×28 pixels. These images include samples from different people to ensure a

```
layers_ = [
    dict(ensemble=dict(n_neurons=1 * 28 * 28,
        dimensions=1)),
    dict(ensemble=dict(n_neurons=6 * 13 * 13,
        dimensions=1)),
    dict(ensemble=dict(n_neurons=24 * 6 * 6,
        dimensions=1)),
    dict(ensemble=dict(n_neurons=10,
        dimensions=10))]

connections = [
    dict(conv=dict(conv=dict(n_filters=1,
        input_shape=(1, 28, 28),
         channels_last=False,
         kernel_size=(1, 1),
         init=np.ones((1, 1, 1, 1))),
         conn={})),
    dict(conv=dict(conv=dict(n_filters=6,
         input_shape=(1, 28, 28),
         channels_last=False,
         kernel_size=(3, 3),
         strides=(2, 2)),
         conn={})),
    dict(conv=dict(conv=dict(n_filters=24,
         input_shape=(6, 13, 13),
         channels_last=False,
         kernel_size=(3, 3),
         strides=(2, 2)),
         conn={})),
    dict(conn=dict(transform=Glorot()))]
```

LISTING 1. SNN layer definition

large variety among the data that can enable the models trained on this data to achieve a high generality for the task.

This dataset also contains classification labels for the data that can be used for supervised model training. At the same time, the dataset has a train sub-dataset, which contains 60,000 samples, as well as a test sub-dataset, which contains 10,000 samples. This way it is possible to calculate the accuracy in the test dataset to ensure the achieved accuracy is general and not only for the training data. Some examples of this digit data are shown in Figure 4.

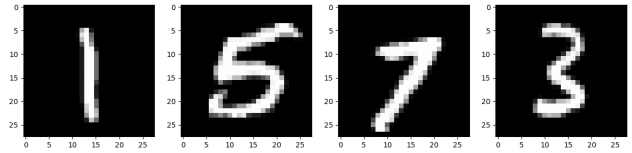


FIGURE 4. Examples of MNIST sample images.

The goal when using this dataset is the classification of individual images into their corresponding classes using an SNN based on the proposed framework. To achieve this objective, a series of preprocessing steps are required to align the input data with the prerequisites imposed by the SNN model. Since SNN requires temporal processing of spikes to provide a stable outcome, the first preprocessing step involves a transformation of the input data into a time-series input. However, in cases where data remains static across time, rather than concatenating different samples to include temporal features, the same image can be concatenated multiple times to ensure static information across temporal steps. Consequently, the input data shape would be (batch size, sample, time steps, features) where the number of features will be set to 784 since the images are converted from 28×28 into flat vectors of 784 features. Converting the data into a flat vector is always required when using the *SpikingNengoNeuralNetworkModel* class. Subsequently, this class will execute a reshaping process to adapt the data to the shape specifications of the selected layer.

Following the data preprocessing phase, the data is ready for the training. However, the SNN architecture must be defined before starting the training process. As with all neural network models, the first step for the network structure definition involves the selection of parameters such as the number of layers, layer types and number of neurons among other parameters. Since the input data are images, convolutional layers can be used to first extract relevant features that can be later used to classify the data. To define the layer structure, we need to create two lists, similar to how it is done in Nengo, to encapsulate both the layers themselves and the interconnections with their respective properties. An example of this process is depicted in Listing 1.

Once the network structure has been defined, the parameters for the training and neuron attributes must be defined to finish generating the full SNN model. Among the parameters to define in this point, as in general ANNs, are the numbers of epochs, batch size, optimizer, learning rate and input shape as well as the specific parameters for the SNN such as the neuron type, simulation time steps, maximum firing rates of the neurons, membrane RC time constant and absolute refractory period, among others. However, the class will set some default parameters in case the user does not specify them. This strategy allows users to easily define a first network configuration whose results can be used later to refine the network parameters. The code to be executed in this case is shown in Listing 2.

During the generation of the SNN model, if the user has set some parameters that are incompatible among themselves, such as wrong values for the RC time constant and the absolute refractory period, the framework will return an error message indicating which parameters must be adjusted and how this can be performed.

Once the network has been correctly generated, the user can start the training phase. For this, the method *SpikingNengoNeuralNetworkModel.fit* can be used as when using TensorFlow [15] or Keras [17]. This call will trigger the training phase, which will use the internal parameters previously defined such as number of epochs or batch size. Once the training phase finishes, it is possible to execute a prediction using the method *SpikingNengoNeuralNetwork-Model.predict* or calculating the accuracy of the model on a set of data and labels by calling the method *SpikingNengoNeuralNetworkModel.score*. With the previously defined

```
minibatch_size = 200
presentation_time = 0.2
dt = 0.001
n_steps = int(presentation_time / dt)

arg = dict(layers=layers_,
      connections=connections,
      optimizer='adam',
      lr=0.001,
      spike_func_mode='random',
      input_size=28 * 28,
      epochs=15,
      presentation_time=presentation_time,
      dt=dt,
      problem_type='classification-multiclass'
      )

model = SpikingNengoNeuralNetworkModel(**arg)
```

LISTING 2. SNN model creation.

```
model.fit(train_images, train_labels)
y_pred = model.predict(test_images)
score = model.score(test_images, test_labels)
```

LISTING 3. SNN training and evaluation.

SNN, the achieved test accuracy is 95.90%. This result has been calculated as shown in Listing 3.

The achieved performance is similar to other implementations based on traditional ANNs, as shown in Table 2. Some authors, such as Kadam et al. [28] compare the performance of multiple Convolutional Neural Networks (CNNs) to further understand the effect of modifying the hyper-parameters of the network in the final result. This author achieves a test accuracy of 98.27% using the same batch size as the SNN defined in this paper and a similar architecture. Following this topic, there are numerous tutorials for other frameworks that use MNIST handwritten digits dataset to demonstrate the workflow of the framework, such as Keras or TensorFlow. One of these examples can be found in [29], where a train accuracy and test accuracy of 98.87% and 99.04% were achieved, respectively. Similarly, it was shown in a TensorFlow example [30] how a simple ANN of 2 layers was able to achieve a train accuracy of 98.18% and a test accuracy of 97.28%. As a summary, we can conclude that the SNN defined with the proposed framework achieves results of similar magnitude order to the rest of the implementations since the maximum accuracy difference is 3.14% when comparing it with the results from [29].

TABLE 2. Comparison of accuracy results based on our framework and similar implementations.

Algorithm	Train accuracy	Test accuracy
Proposed SNN	97.02%	96.50%
Kadam S.S. [28]	99.93%	98.27%
Keras [29]	98.87%	99.04%
TensorFlow [30]	98.18%	97.28%

The previously implemented methods, as commented in Section III, were designed taking into consideration the compatibility with other packages. Because of this, our framework has these common names and executing pipelines. Nevertheless, after the training, it is possible to access some specific information of SNN models. One of these options is extracting the output spikes to visualize the temporal evolution of the classification results. This information can be used to better set the number of time steps to include in each input data sample. Some examples of the output spikes as well as the temporal classification output decision are shown in Figure 5.

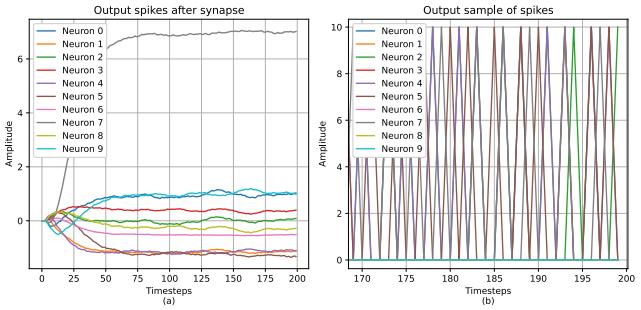


FIGURE 5. Output spikes from SNN trained with MNIST data, a) output spikes after synapse over timesteps, b) raw spikes over timesteps. Notice that in b) a zoom has been applied in order to better visualize the spikes.

From Figure 5, one of the optimizations the user could make is to reduce the number of time steps required to provide a prediction. As an example, in the left image of the figure, the value of the synapse associated with the correct class for this input data is highly superior to the rest of the synapse values since the time step 75. Therefore, the time steps could be reduced to improve the energy consumption, memory consumption and the latency of the model without affecting the classification accuracy of the SNN model.

As a summary of this use case, a list of the relevant features of the framework are provided below:

- The structure of the pipeline does not require a deeper understanding of SNN to be implemented since it can be used following the same steps of Keras or TensorFlow.
- The user does not need to configure the simulator for the SNN since the framework will configure it based on the parameters of the SNN.
- The framework includes methods to extract the metric results with the SNN as well as methods to access spike results that can be used to further optimize the designed SNN.
- SNN models trained with the proposed framework achieve similar results to traditional ANNs taking into consideration the large difference in the data processing between SNNs and ANNs.

B. MILES PER GALLON PREDICTION USE CASE

In this use case, the auto-mpg dataset [31] will be used to present how the proposed framework can be used to generate

```
dt = 0.001
minibatch_size = 10
tau_ref = 0.002

layers_ = [
    dict(dense=dict(units=300)),
    dict(dense=dict(units=200)),
    dict(dense=dict(units=100)),
    dict(dense=dict(units=1),
    neuron_type=Sigmoid(tau_ref=tau_ref))
]

arg = dict(layers=layers_,
      lr=0.001,
      optimizer='rmsprop',
      input_size=n_features,
      loss_function='mean_absolute_error',
      metric_score='mean_absolute_error',
      minibatch_size=minibatch_size,
      epochs=300,
      dt=dt,
      tau_ref=tau_ref,
      max_rate=300,
      problem_type='regression'
      )

model = SpikingNeuralNetworkModel(**arg)
```

LISTING 4. Definition of SNN for auto-mpg.

a SNN model for regression tasks. Simultaneously, within this use case, external capabilities from the *scikit* package [16] will be incorporated into the network structure definition and training phases to facilitate the identification of optimal model configurations.

This dataset contains information regarding multiple vehicle features such as number of cylinders, horsepower, weight and acceleration among others. Some of these features are discrete values such as the number of cylinders while the dataset also contains continuous values such as vehicle weight or acceleration. Based on these features, the aim when using this dataset is predicting the fuel efficiency in terms of miles per gallon. For illustrative purposes, some sample data extracted from this dataset is presented in Table 3. It is important to note that this table does not contains the entirety of features within the dataset; rather, its purpose is to offer a brief introduction to the data.

TABLE 3. Data sample from some features of the auto-mpg dataset.

MPG	Horsepower	Weight	Acceleration
18.0	130.0	3504.0	12.0
15.0	165.0	3693.0	11.5
18.0	150.0	3436.0	11.0
16.0	150.0	3433.0	12.0

First of all, the data has to be preprocessed in order to fit the data structure that can be used with the SNN. In this case, first of all the categorical features have to be converted into numerical values. Accordingly, the *'Origin'* feature must be converted to integer values. Furthermore, a normalization procedure is applied to all features to ensure the network takes into consideration all features equally. Otherwise, due to the large range of variation between features, this could not be ensured. After this, as in the previous use case, the data must be converted into a temporal series that the SNN can study over time to provide a stable output. However, this data does not have any temporal evolution so the same sample will be repeated a number of times to maintain it long enough for the network to provide a consistent output.

```
param_grid = {'lr': [0.001, 0.0005],
              'minibatch_size': [10, 20, 40]}

grid_model = GridSearchCV(model,
                          param_grid,
                          cv=3)

grid_model.fit(x_train, y_train)
```

LISTING 5. Grid search for learning rate and batch size.

In this particular scenario, the *SpikingNeuralNetwork-Model* class will be used in order to showcase the usage of this class. Consequently, the definition of the SNN model varies with respect to the previous use case. Here, the layer structure is defined using only one list, assuming a linear interconnection among these layers with preceding layers for data propagation. The remaining parameter configuration is similar to the previous use case and can be seen in detail in Listing 4.

At this point, the model is ready for the training phase. However, rather than assuming a network structure directly, an alternative approach emerges when using the *GridSearch* method from the *scikit-learn* library [16]. This approach facilitates the evaluation of numerous model configurations based on the combination of a set of desired values for some of the parameters of the model. This enables the user to keep track of all the evaluated configurations to avoid human errors in this step. Moreover, this approach reduces the time to configure and train all these models thanks to the optimizations of this method.

Therefore, the learning rate and batch size parameters will be evaluated using this approach to find the best configuration following the Listing 5. In this case, *'lr'* refers to the learning rate and *'minibatch size'* to the desired batch size.

Using this code, the model will be trained using all the possible combinations of the learning rate and batch size to find the optimal combination. Similarly, the number of neurons per layer can also be evaluated following the same approach. However, an additional auxiliary function is required to generate all the possible combinations of neurons per layer, since the framework input must be the whole network structure. Nevertheless, with the proposed function this issue is solved and can be used in combination with other parameters such as learning rate, as depicted in Listing 6.

As a result of this study, the best configurations, among the considered ones, for the SNN model are a learning rate

```
def gen_layer(struct):
  layer = [
      dict(dense=dict(units=struct['neu_1'])),
      dict(dense=dict(units=struct['neu_2'])),
      dict(dense=dict(units=struct['neu_3'])),
      dict(dense=dict(units=9,
           activation='sigmoid'),
           neuron_type=Sigmoid(tau_ref= 0.003))
      ]
  return layer

def gen_layer_struct(neu_1, neu_2, neu_3):
  param_grid = dict(neu_1= neu_1,
                    neu_2= neu_2,
                    neu_3= neu_3)

  keys, value = zip(*param_grid.items())
  permutations = [dict(zip(keys, v))
                  for v in it.product(*value)]

  layers = []
  for i in range(len(permutations)):
    layers.append(gen_layer(permutations[i]))

  return layers

neu_1 = [400, 300, 200]
neu_2 = [300, 200, 100]
neu_3 = [200, 100]

layers_perm = gen_layer_struct(neu_1,
                               neu_2,
                               neu_3)

param_grid = {'layers': layers_perm,
              'lr': [0.001, 0.0005]}

grid_model = GridSearchCV(model,
                          param_grid,
                          cv=3)

grid_model.fit(x_train, y_train)
```

LISTING 6. Grid search for learning rate and number of neurons.

TABLE 4. Comparison of results using auto-mpg dataset.

Algorithm	Train MAE	Test MAE
Proposed SNN	0.04	0.04
TensorFlow [32]	1.87	1.84
M. Rahman [33]	3.25	3.28

of 0.001, a batch size of 10 samples, and a layer structure of 400, 300 and 200 neurons respectively in the hidden layers. The performance results with this configuration can be found in Table 4 where the Mean Absolute Error (MAE) is used as a metric. At the same time, some results using other frameworks that can be used as a reference for performance comparison are included in this table to show how the proposed network outperforms these other approaches regarding MAE results.

When using this class of the framework, it is still possible to access the internal parameters of the SNN model apart from

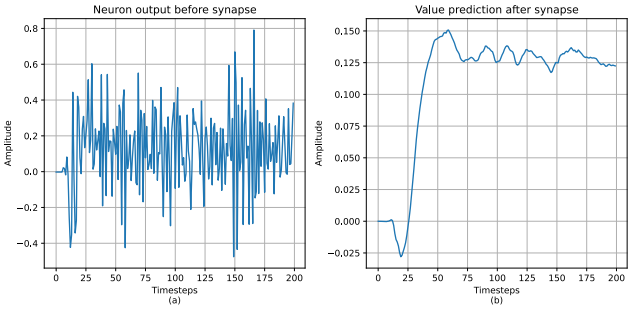


FIGURE 6. Output from SNN trained with Aut-MPG data, a) output of the neuron without synapse, b) output of the neuron after synapse, which correspond to the estimated value.

the traditional output value and loss. The spikes, as in the last use case, can be accessed to further understand the temporal component of the decision making process. However, since in this case the output is a single regression value, there is only one output neuron in contrast with the previous use case as shown in Figure 6.

In the left side of Figure 6 it is possible to observe the temporal evolution of the spikes of the output neuron as well as how this also affects the variation of the final output of the neuron, which is shown in the right side of the image. When studying the temporal evolution of this last signal, it is possible to observe how, after 45 time steps, the value starts to oscillate around a central value. Consequently, by studying only this sample it is not possible to further optimize the temporal components of the network for this use case.

As a summary of this use case, a list of the relevant features of the framework presented in this subsection are provided below:

- The class *SpikingNeuralNetworkModel* has been presented to better understand how to use it to define an SNN model.
- It has been shown how the proposed framework can be used with sklearn packages such as *GridSearch*.
- The performance results achieved with the proposed framework reach a lower MAE than the rest of the approaches used for comparison and based on traditional ANNs.

C. GOOGLE SPEECH COMMANDS

Another cutting-edge use case that can be effectively addressed through the use of SNNs, making this framework highly valuable, is keyword spotting. The objective here is to identify voice commands, for which a dataset of single spoken English words has been used. More precisely, the Google Speech Commands v0.02 dataset has been used [34]. It contains a set of 30 target words spoken by a variety of different speakers, as well as background noise, adding a total of 65,727 one-second labeled audio files with a sampling rate of 16 KHz.

In this use case, the raw data by itself cannot be used directly for the training of the model due to its complexity.

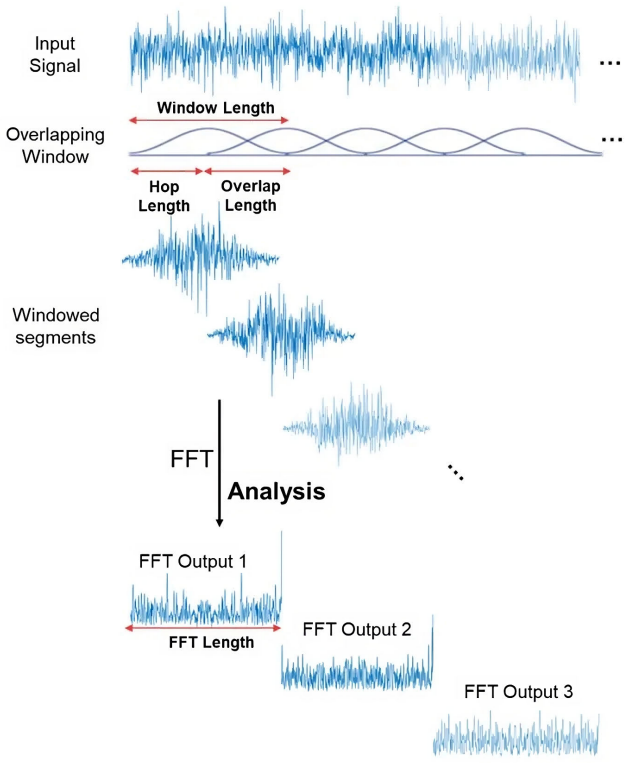


FIGURE 7. Visual explanation of the spectrogram calculation [36].

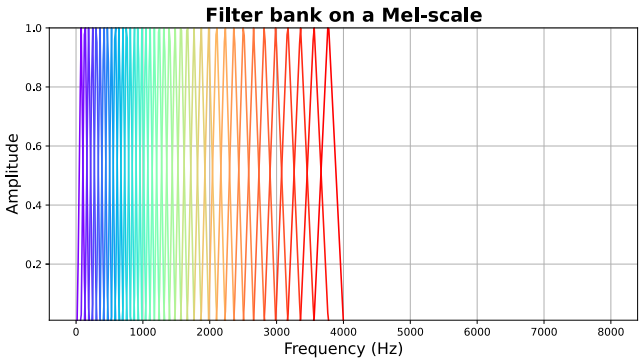


FIGURE 8. Filter bank used to project FFT bins onto Mel-frequency bins. Each filter is represented using a different color, with a total of 40 filters represented in this figure.

Consequently, a set of preprocessing techniques have to be applied to extract relevant features. These techniques are used to generate the so-called Mel-Spectogram [35], which provides information about the frequency content of a non periodic signal in a scale more similar to how humans perceive the sound. The Mel-Spectogram is derived from the normal spectrogram of the signal. Therefore, the first step to generate it is the calculation of the signal spectrogram. Figure 7 shows how this spectrogram is computed by applying the Fast Fourier Transform (FFT) on a set of overlapping window segments of the signals.

In this example, a window length of 480 samples (30 ms) and a hop length of 240 samples were selected to compute the spectrogram. This configuration was selected empirically after a research of different numbers of samples for the window and hop length based on their final model performance. These preprocessing parameters could be further optimized. However, in this example we want to demonstrate the capabilities of the proposed framework rather than further diving in the data preprocessing optimization.

After this, in order to compute the Mel-Spectrogram it is necessary to convert the frequencies to Mel-scale [37]. The human ears do not perceive frequencies on a linear scale, being the lower frequencies better differentiated than the higher ones. The Mel-scale aims to mimic the non-linear human ear perception of sounds. To perform this conversion, a triangular filter bank is applied.

Each of the filters of this filter bank has a response of 1 at the center frequency and decreases linearly towards 0 till it reaches the center frequencies of the two adjacent filters of both sides. Figure 8 shows an example of a filter bank with 40 filters. This number of filters was the one selected for the preprocessing. This could be further optimized as previously commented.

In summary, by applying this preprocessing, the initial 16,000 samples audio signals are turned into images of 40×67 pixels, where each row corresponds to a "mel-frequency" and each column corresponds to one of the previously commented data windows. Figure 9 shows an example of the computed Mel-Spectrogram.

To facilitate the comparison of results with other models in the literature, one option is to utilize a reduced version of the dataset instead of the entire dataset. Since this is a demonstratory use case to show the framework functionality, it has been decided to use just two speech commands, which are *'left'* and *'right'*.

Since this is a benchmark dataset, the train, test and validation splits are already provided, so different models could be compared in the same conditions. The dataset contains a list with the names of the audio files corresponding to each subset. Specifically, after reducing the dataset to use only the two mentioned words, the train set contains 3,318 samples, while the test and validation ones contain 464 and 466 samples respectively, which is approximately a 11% of the train size.

In a similar way to how it was done in the previous MNIST example, it is also necessary to add a temporal dimension to the input data after the described preprocessing. For that purpose, a vectorization has to be applied to the Mel-Spectrogram matrices, adding also a new dimension for the time. As elucidated in the Subsection III, for static data, a common technique involves concatenating the same input multiple times along the time dimension. As a result of this, the input data shape will be (batch size, sample, time steps, features), with the number of features in this case being 2,680. This number comes from the flattening process of the images of shape 40×67 pixels.

Once the data preprocessing has been applied, the only thing that remains is to define the network architecture and

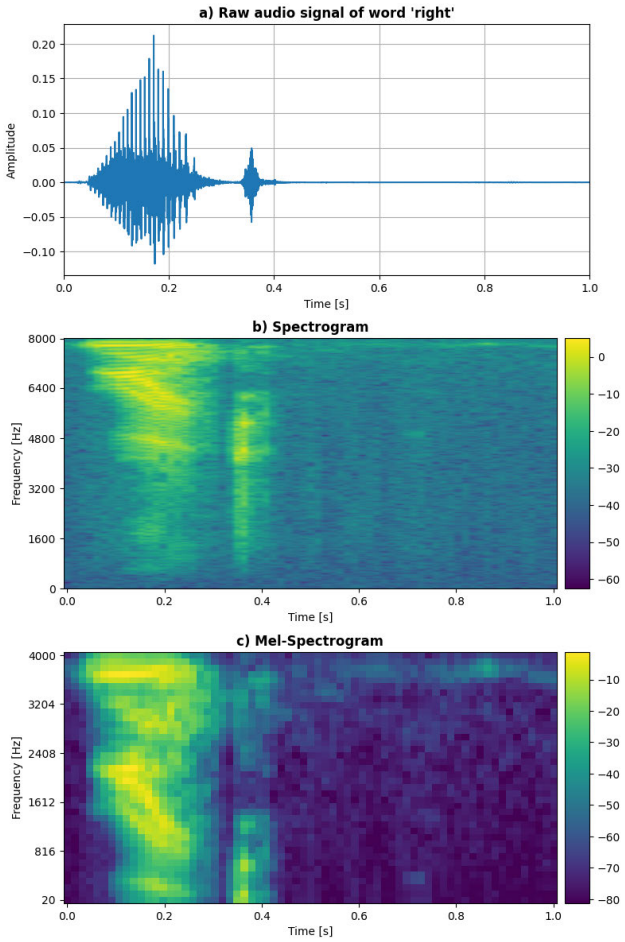


FIGURE 9. Visual representation of the preprocessing applied to the raw audio signal.

to train it. In this use case, the *SpikingNeuralNetworkModel* class will be used. Therefore, the SNN definition is similar to the one described in the auto-mpg example of Subsection IV-B, using only one list to specify the layer structure. Since here the input data are images, the most suitable layers that could be applied are convolutional layers to further extract relevant features. Nevertheless, it is always advisable to try other architectures to make sure which one is the most convenient in each case. This use case will demonstrate how to achieve it using the proposed framework.

In Listing 7 it is shown how to configure the initial model with this type of layers. However, the number of filters of these layers, the number of layers and the learning rate are hyperparameters that have to be set based on empirical experimentation. This is an iterative process that could be highly time consuming. To speed up this process and reduce possible human errors, the *scikit-learn* library can be integrated in the usage of the proposed framework to execute a grid search of these empirical hyperparameters. The *GridSearch* method will internally change the hyperparameters of the initial model shown in Listing 7 based on the specified combinations

```
layers_ = [
    dict(dense=dict(units=500)),
    dict(dense=dict(units=100)),
    dict(dense=dict(units=num_labels))
]

snn_params = dict(
    layers=layers_,
    lr=0.001,
    optimizer='adam',
    neuron_type='SoftLIF',
    input_size=40 * 67,
    minibatch_size=100,
    epochs=50,
    amp=0.01,
    dt=0.001,
    tau_rc=0.02,
    max_rate=100,
    radius=1,
    synapse_post_trainig=0.001,
    problem_type='classification-multiclass',
    seed_=0
)

model = SpikingNeuralNetworkModel(**arg)
```

LISTING 7. Definition of SNN for Google Speech Commands.

of them. By the end of this automatized iterative process, the best combination of hyperparameters will be provided.

In Subsection IV-B, a grid search looking for the best combination of number of neurons of each layer, learning rate and batch size was shown. However, this technique will be used this time to study different SNN architectures that could be applied to study the Mel-Spectrogram images. For demonstration purposes, three different options of architectures will be taken into consideration, but a more complex set of combinations could be used following the same approach. In contrast with the auto-mpg example, the hyperparameters of each layer will remain constant, but the number of layers or the type of them will change between iterations. As it is shown in Listing 8, a fully connected network as well as two convolutional network are tested, in combination with different learning rates and batch sizes.

After performing this study, the best combination of hyperparameters is a batch size of 100, a learning rate of 0.001 in combination with the third network option, which is actually the network with no convolutional connections among the selected ones. Using these hyperparameters, the performance obtained can be found in Table 5, where the accuracy of the model is used as comparison metric. Since this is a well known benchmark dataset, other results can be found in the literature for comparison purposes. In this table, it could be seen that the results that the network yields using our framework are close to the top ones found in the literature. The reference founded does not provide the train metrics, that is why they are not included in Table 5. Those top results are obtained using networks with complex layers, while the one shown in this use case is a simple one and it still produces comparable results.

```
snn_arch1 = [
    dict(dense=dict(units=500)),
    dict(dense=dict(units=100)),
    dict(dense=dict(units=num_labels))
]

snn_arch2 = [
    dict(conv=dict(filters=4,
                   kernel_size=3,
                   padding='valid'),
                   shape_in=(40, 67, 1)),
    dict(dense=dict(units=500)),
    dict(dense=dict(units=100)),
    dict(dense=dict(units=num_labels))
]

snn_arch3 = [
    dict(conv=dict(filters=4,
                   kernel_size=3,
                   padding='valid'),
                   shape_in=(40, 67, 1)),
    dict(conv=dict(filters=8,
                   kernel_size=5,
                   padding='valid'),
                   shape_in=(38, 65, 4)),
    dict(dense=dict(units=500)),
    dict(dense=dict(units=100)),
    dict(dense=dict(units=num_labels))
]

param_grid = {'layers': [snn_arch1,
                         snn_arch2,
                         snn_arch3],
              'lr': [0.001, 0.0005],
              'minibatch_size': [50, 100, 200]}

grid_model = GridSearchCV(model,
                          param_grid,
                          cv=3)

grid_model.fit(x_train, y_train)
```

LISTING 8. Grid search for different architectures, learning rate and batch size.

TABLE 5. Comparison of results using Google Speech Commands dataset.

Algorithm	Train accuracy	Test accuracy
Proposed Arch. 1	99.91%	96.08%
Proposed Arch. 2	97.18%	91.52%
Proposed Arch. 3	96.79%	91.96%
Attention RNN [38]	-	99.20%
Attention RNN v2 [38]	-	99.40%

Through this use case, it has been shown how to solve a state of the art problem with SNN using our framework. Moreover, various types of architectures have been presented, along with a pipeline that implements *GridSearch* to identify the optimal combination of architectures and two crucial hyperparameters. In summary, a list of the relevant features that have been tested to be effective in this use case is provided.

- The class *SpikingNeuralNetworkModel* has been presented to better understand how to use it to define an SNN model using convolutional layers.
- It has been shown how to perform *GridSearch* using this framework to find the best candidate among different types of architectures.
- The performance results achieved in this use case are comparable to the top one found in the literature.

V. CONCLUSION

In this paper, we have presented a comprehensive SNN framework and showcased its successful application to three use cases. The SNN framework, inspired by the biological neural processes, offers a unique approach to neural computation, particularly in handling temporal and event-driven data. Through a detailed exploration of the framework's architecture, learning mechanisms, and simulation techniques, we have highlighted its potential to revolutionize the field of artificial intelligence and advance our understanding of brain-inspired computing.

By applying the SNN framework to three use cases, including image processing of handwritten numbers, time-series forecasting and speech recognition, we have demonstrated its effectiveness in well-known benchmarking databases that include state-of-the-art applications. In these applications, the SNN framework has shown remarkable capabilities in capturing temporal dynamics and processing asynchronous input streams, having a similar Key Performance Indicator (KPI) to other models.

However, as with any emerging technology, challenges remain in optimizing the SNN framework's training procedures, scalability, and computational efficiency. Further research is necessary to explore advanced spike-based learning algorithms and to optimize hardware implementations to fully unlock the framework's potential.

In conclusion, the GenericSNN framework proves to be user-friendly and highly applicable to real-world scenarios. Researchers and practitioners can seamlessly integrate the SNN framework into their existing workflows, thanks to its accessible implementation and compatibility with widely-used programming libraries. Moreover, the framework demonstrates exceptional performance and generalization capabilities when applied to standard benchmarking datasets in various domains. As technology progresses, we look forward to witnessing the broader adoption and transformative impact of the SNN framework in shaping the future of AI and cognitive computing.

The scripts of the proposed framework, as well as the examples, can be accessed in: https://gitlab.eesysmart.de/public-ai-team/genericsnn

REFERENCES

[1] Q. T. Pham, T. Q. Nguyen, P. C. Hoang, Q. H. Dang, D. M. Nguyen, and H. H. Nguyen, "A review of SNN implementation on FPGA," in *Proc. Int. Conf. Multimedia Anal. Pattern Recognit. (MAPR)*, Oct. 2021, pp. 1–6.
[2] S. Wang, T. H. Cheng, and M. H. Lim, "A hierarchical taxonomic survey of spiking neural networks," *Memetic Comput.*, vol. 14, no. 3, pp. 335–354, Sep. 2022.
[3] Q. Liu and Z. Zhang, "Ultralow power always-on intelligent and connected SNN-based system for multimedia IoT-enabled applications," *IEEE Internet Things J.*, vol. 9, no. 17, pp. 15570–15577, Sep. 2022.

[4] A. R. Zarzoor, N. A. S. Al-Jamali, and I. R. K. Al-Saedi, "Traffic classification of IoT devices by utilizing spike neural network learning approach," *Math. Model. Eng. Problems*, vol. 10, no. 2, pp. 639–646, Apr. 2023.

[5] Q. Jiang and J. Sha, "The use of SNN for ultralow-power RF fingerprinting identification with attention mechanisms in VDES-SAT," *IEEE Internet Things J.*, vol. 10, no. 17, pp. 15594–15603, 2023, doi: 10.1109/JIOT.2023.3264715.

[6] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Netw.*, vol. 111, pp. 47–63, Mar. 2019.

[7] J. Vreeken, "Spiking neural networks, an introduction," Universiteit Utrecht, Utrecht, The Netherlands, Tech. Rep., 2003. [Online]. Available: https://scholar.google.es/scholar?q=%E2%80%98Spiking+neural+networks,+an+introduction,&hl=en&as_sdt=0&as_vis=1&oi=scholart

[8] K. D. Carlson, J. M. Nageswaran, N. Dutt, and J. L. Krichmar, "An efficient automated parameter tuning framework for spiking neural networks," *Frontiers Neurosci.*, vol. 8, p. 10, Feb. 2014.

[9] S. Schliebs, M. Defoin-Platel, S. Worner, and N. Kasabov, "Integrated feature and parameter optimization for an evolving spiking neural network: Exploring heterogeneous probabilistic models," *Neural Netw.*, vol. 22, nos. 5–6, pp. 623–632, Jul. 2009.

[10] C. Li, L. Ma, and S. Furber, "Quantization framework for fast spiking neural networks," *Frontiers Neurosci.*, vol. 16, Jul. 2022, Art. no. 918793.

[11] Z. Zhou, Y. Zhu, C. He, Y. Wang, S. Yan, Y. Tian, and L. Yuan, "Spikformer: When spiking neural network meets transformer," 2022, *arXiv:2209.15425*.

[12] R. V. W. Putra and M. Shafique, "Q-SpiNN: A framework for quantizing spiking neural networks," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–8.

[13] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: A Python tool for building large-scale functional brain models," *Frontiers in Neuroinform.*, vol. 7, p. 48, Jan. 2014.

[14] D. Rasmussen, "NengoDL: Combining deep learning and neuromorphic modelling methods," 2018, *arXiv:1805.11144*.

[15] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," Google, Mountain View, CA, USA, Tech. Rep. 1603.04467, 2015. [Online]. Available: https://arxiv.org/abs/1603.04467

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.

[17] F. Chollet. (2015). *Keras*. [Online]. Available: https://github.com/fchollet/keras

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[19] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Oct. 2012.

[20] D. R. Muir, F. Bauer, and P. Weidel, "Rockpool documentaton," SynSense, Zürich, Switzerland, Tech. Rep. 10.5281/zenodo.3773845, Sep. 2019. [Online]. Available: https://rockpool.ai/about.html

[21] F. Liu, W. Zhao, Y. Chen, Z. Wang, and L. Jiang, "SpikeConverter: An efficient conversion framework zipping the gap between artificial neural networks and spiking neural networks," in *Proc. AAAI Conf. Artif. Intell.*, vol. 36, 2022, pp. 1692–1701.

[22] W. Fang, Y. Chen, J. Ding, Z. Yu, T. Masquelier, D. Chen, L. Huang, H. Zhou, G. Li, and Y. Tian, "SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence," *Sci. Adv.*, vol. 9, no. 40, Oct. 2023, Art. no. eadi1480.

[23] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," 2017, *arXiv:1708.07747*.

[24] W. Guo, M. E. Fouda, A. M. Eltawil, and K. N. Salama, "Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems," *Frontiers Neurosci.*, vol. 15, Mar. 2021, Art. no. 638474.

[25] D. Auge, J. Hille, E. Mueller, and A. Knoll, "A survey of encoding techniques for signal processing in spiking neural networks," *Neural Process. Lett.*, vol. 53, no. 6, pp. 4693–4710, Dec. 2021.

[26] C. Eliasmith and C. H. Anderson, *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MA, USA: MIT Press, 2003.

[27] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, 2012.

[28] S. S. Kadam, A. C. Adamuthe, and A. B. Patil, "CNN model for image classification on MNIST and fashion-MNIST dataset," *J. Sci. Res.*, vol. 64, no. 2, pp. 374–384, 2020.

[29] S. Mahapatra. *A Simple 2D CNN for MNIST Digit Recognition*. Accessed: Jul. 21, 2023. [Online]. Available: https://towardsdatascience.com/a-simple-2d-cnn-for-mnist-digit-recognition-a998dbc1e79a

[30] TensorFlow. *Training of a Neural Network With MNIST and Keras*. Accessed: Jul. 21, 2023. [Online]. Available: https://www.tensorflow.org/datasets/kerasexample?hl=es-419

[31] R. Quinlan, "Auto MPG," in *UCI Machine Learning Repository*, 1993. [Online]. Available: https://archive.ics.uci.edu/dataset/9/auto+mpg, doi: 10.24432/C5859H.

[32] TensorFlow. *Training of a Neural Network With Auto-MPG With TensorFlow*. Accessed: Jul. 24, 2023. [Online]. Available: https://www.tensorflow.org/tutorials/keras/regression?hl=es-419

[33] M. Rahman and M. Asadujjaman, "Implementation of artificial neural network on regression analysis," in *Proc. 5th Annu. Syst. Model. Conf. (SMC)*, Sep. 2021, pp. 1–7.

[34] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.

[35] D. O'Shaughnessy, *Speech Communications: Human and Machine (IEEE)*. New York, NY, USA: Universities Press, 1987. [Online]. Available: https://ieeexplore.ieee.org/book/5263814

[36] The MathWorks. *Sfft*. Accessed: Jan. 21, 2024. [Online]. Available: https://de.mathworks.com/help/dsp/ref/dsp.stft.html

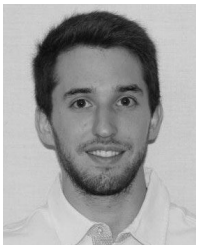
[37] S. S. Stevens and J. Volkmann, "The relation of pitch to frequency: A revised scale," *Amer. J. Psychol.*, vol. 53, no. 3, pp. 329–353, 1940.

[38] D. Coimbra de Andrade, S. Leo, M. Loesener Da Silva Viana, and C. Bernkopf, "A neural attention model for speech command recognition," 2018, *arXiv:1808.08929*.

ALBERTO MARTIN-MARTIN received the B.Sc. degree in electronic engineering from the University of Granada (UGR), Spain, in 2019, and the M.Sc. degree in data science and computer engineering from UGR, where he is currently pursuing the Ph.D. degree in the field of implementation of artificial neural networks on reconfigurable hardware. In January 2021, he moved to Munich to work as an Artificial Intelligence Engineer with eesy-Innovation GmbH. His current research interests include deep learning, FPGAs, and embedded systems.

MARTA VERONA-ALMEIDA received the double bachelor's degree in mathematics and computer science from the University of Granada, in 2018, and the master's degree in data science and computer engineering from the University of Granada. She is currently pursuing the Ph.D. degree in collaboration with the Politechnical University of Madrid (UPM) in the field of data-driven modeling methodology for identifying the implication of sleep quality on health. After finishing her studies, she was a Research Assistant with the University of Granada, contributing to the development of KnowSeq package. In August 2020, she moved to Munich to work as an Artificial Intelligence Engineer with eesy-Innovation GmbH. Her current research interests include deep learning, health care, and genomics.

RUBÉN PADIAL-ALLUÉ received the B.Sc. degree in industrial electronics engineering from the University of Granada, Spain, in 2016, and the M.Sc. degree in microelectronics from the University of Seville, Spain, in 2020. In February 2023, he commenced collaboration with the University of Granada as part of a research project that encompasses his Ph.D. studies focused on the implementation and optimization of artificial neural networks for hardware acceleration on FPGA devices. His current research interests include embedded systems, FPGAs, and their applications in the field of artificial intelligence.



JAVIER MENDEZ received the B.Sc. degree in electronics engineering from the University of Granada, in 2018, the M.Sc. degree in electronics, robotics and automatics engineering from the University of Seville, Spain, in 2019, and the Ph.D. degree (cum laude) in artificial intelligence from the University of Granada, in 2022. After the M.Sc. degree, he moved to Munich to start the Ph.D. degree with Infineon Technologies. Since May 2022, he has been with eesy-Innovation. His current research interests include emerging deep learning trends, edge computing, and sensor fusion. During the master's degree, he received the award as the Top Student of the Graduating Class.

ENCARNACIÓN CASTILLO received the M.A.Sc. and Ph.D. degrees in electronic engineering from the University of Granada, in 2002 and 2008, respectively. From 2003 to 2005, she was a Research Fellow with the Department of Electronics and Computer Technology, University of Granada, where she is currently a tenured Professor. All this work has led to the publication of 50 articles in indexed journals and more than 60 contributions to international conferences. She has coauthored two patents in Spain and contributed to 13 projects in the several national and regional programs and 11 technology-transfer contracts. Her current research interests include cryptoprocessors on FPGA, memory devices, smart instrumentation, and acquisition and processing of biosignals. She serves as a reviewer and a guest editor for several journals.

LUIS PARRILLA (Member, IEEE) received the M.Sc. degree in physics (majoring in electronics), the M.A.Sc. degree in electronic engineering, and the Ph.D. degree in physics from the University of Granada, Granada, Spain, in 1993, 1995, and 1997, respectively. In 1995, he joined the Department of Electronics and Computer Technology, University of Granada, where he has been a Professor, since 2000. He is the author of more than 70 technical papers in international journals and conferences and serves regularly as a reviewer and a guest editor for several journals. His current research interests include the protection of IP cores on VLSI and FPGA-based systems, the development of high-performance arithmetic and algebraic circuits for IoT and cryptographic applications, the design of specific architectures for cryptographic processors, and hardware implementation of artificial neural networks.