A DEEP LEARNING FRAMEWORK FOR MUSICAL ACOUSTICS SIMULATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

The acoustic modeling of musical instruments is a heavy computational process, often bound to the solution of complex systems of partial differential equations (PDEs). Numerical models can achieve a high level of accuracy, but they may take up to several hours to complete a full simulation, especially in the case of intricate musical mechanisms. The application of deep learning, and in particular of *neural operators* that learn mappings between function spaces, has the potential to revolutionize how acoustics PDEs are solved and noticeably speed up musical simulations. However, such operators require large datasets, capable of exemplifying the relationship between input parameters (excitation) and output solutions (acoustic wave propagation) per each target musical instrument/configuration. With this work, we present an open-access, open-source framework designed for the generation of numerical musical acoustics datasets and for the training/benchmarking of acoustics neural operators. We first describe the overall structure of the framework and the proposed data generation workflow. Then, we detail the first numerical models that were ported to the framework. Finally, we conclude by sharing some preliminary results obtained by means of training a state-of-the-art neural operator with a dataset generated via the framework. This work is a first step towards the gathering of a research community that focuses on deep learning applied to musical acoustics, and shares workflows and benchmarking tools.

1 INTRODUCTION

The study of the acoustics of musical instruments is a challenging topic. Physics phenomena underlying music making are quite various and include excitation, resonant behavior, as well as the coupling and the dynamic modification of the involved mechanical parts. These make musical instruments remarkable examples of engineering, but also acoustic systems difficult to model. The most accurate simulations that exist today leverage the numerical solution of partial differential equations (PDEs), that are in turn designed to model the specific acoustic behavior of the targeted instruments (Bilbao, 2009). Unfortunately, the majority of the employed solvers are characterized by heavy computational requirements, often leading to restrictive implementation conditions (e.g., low spatio-temporal resolution, high degree of model simplification, non-interactive paradigms).

Recent advancements in deep learning have shown how neural networks may be used to enhance and even replace traditional PDE solvers (Bhatnagar et al., 2019), with the aim to improve performance. In particular, the use of *neural operators* has yielded promising results in fluids dynamics (Li et al., 2020), suggesting that their application may be successfully extended to revolutionize the simulation of the acoustics and the aeroacoustics of musical instruments. Being completely data-driven, neural operators could be trained to solve acoustics PDEs with synthetic datasets, generated via the large array of traditional numerical implementations that are available in the literature[1].

Although exciting, this scenario is hindered by a lack of common practices that are needed to bridge the domains of musical acoustics and deep learning. These include shared datasets, benchmarks, as well as general tools to help researchers categorize, manage and employ acoustics data for training and inference. The aim of our research is to foster the rapid growth of an active community where

[1]In this scenario the only constraint would be computational time—an affordable caveat when generating training sets.

these common practices could be discussed and formalized, along with the overall emerging field of deep learning-based musical acoustics. In line with this mission, in this work we present the *Neuralacoustics* framework, a collection of open-access/open-source scripts and tools designed to address the aforementioned needs. In particular, we provide an in-depth description of the dataset generation workflow proposed as part of the framework, and we introduce the first numerical models available in it. We also discuss preliminary results obtained by training a state-of-the-art neural operator for the solution of a simple acoustics problem, using exclusively the tools available in the framework.

2 BACKGROUND

Musical Acoustic Simulations. In the musical domain, the practice of designing mathematical models of instruments is often referred to with the term *physical modeling synthesis*. Common techniques include modal synthesis (Causse et al., 2011) and digital waveguides (Smith, 1992). Yet, the most precise techniques rely on numerical analysis (Castagné & Cadoz, 2003) (e.g., finite elements, finite differences). Numerical models implement solvers of PDE systems; they can finely simulate fundamental aspects of musical acoustics, like wave propagation and aeroacoustics, as well as physical phenomena beyond instruments and music (Yokota et al., 2002; Arnela & Guasch, 2014). The downside of numerical approaches lies in the computational load of the resulting models, as well as in the amount of parameters they have to comprise to properly simulate the instrument's behavior.

Of particular interest to our work is the case of time-domain simulations of musical instruments (Bilbao, 2009). In this context, the PDEs solved by the models describe the relationship between previous and next states of the instruments, organized over discrete time steps. Other than taking into account time-varying acoustic excitation of the instruments, this approach potentially enables the design of interactive models. Despite the high computational requirements of numerical analysis, real-time interactive models of musical instruments have been designed in recent years (Sosnick & Hsu, 2010; Allen & Raghuvanshi, 2015; Zappi et al., 2017). Unfortunately, this approach relies on expensive dedicated hardware (GPUs) and implementations are characterized by noticeable technical constraints, that limit access to models' parameters and constrains interaction (Renney et al., 2022). As a result, numerical analysis is employed for the greater part to model simple musical systems[2] (Bilbao et al., 2019), or for batch (i.e., non-real-time) simulations (Bilbao & Chick, 2013; Arnela & Guasch, 2014) that may require run-times of several hours. In both cases, the applicability as well as the intelligibility of the resulting models are heavily hindered.

Deep Learning and PDE Solvers. Recently, deep learning has been successfully explored for the generation of PDE solvers describing time-dependent problems (Blechschmidt & Ernst, 2021; Li et al., 2020). These neural solvers may reduce the overall computational requirements of traditional ones, while approximating their output with a remarkable degree of precision. One of the simplest examples of neural solvers consists of deep convolutional neural networks parametrizing the operator that maps inputs and outputs (i.e., solutions) of the PDEs (Bhatnagar et al., 2019; Khoo et al., 2021). The limitation to this approach lies in its dependence on the chosen mesh, meaning that it is not possible to compute solutions outside the discretization grid used for training. Physics informed neural networks solve this issue, as they are mesh-independent and designed to work alongside classical schemes (e.g., Runge-Kutta) (Raissi et al., 2019). Although capable of addressing problems in the small data setting and with high dimensionality (Blechschmidt & Ernst, 2021), they are often employed to solve time-dependent PDEs that share many similarities with the ones modeling musical acoustics—e.g., Navier-Stokes equations (Rudy et al., 2017; Font et al., 2021; Cai et al., 2022). Being only partially data-driven, this approach requires to tailor the network to a specific instance of the PDEs and to repeat training at any given new input.

Most of the individual advantages of the approaches introduced so far are collated in neural operators (Li et al., 2020). Neural operators are mesh-free operators that require no prior knowledge of the underlying PDEs. They learn mappings between infinite-dimensional spaces of functions relying only on a finite collection of observations; and they can be used without retraining to solve PDEs

[2]These numerical models can be deemed as "simple" only if compared to the complexity of actual acoustic instruments.

with different discretizations. Although very recent, they showed promising results not only in fluid dynamics (Li et al., 2020), but also in the solution of wave equations (Guan et al., 2021).

3 DATASET GENERATION FRAMEWORK

The Neuralacoustics framework stems from the necessity to generate musical acoustics datasets that could be easily employed in deep learning. It consists of a collection of Python implementations of numerical models of musical instruments, embedded in a modular structure that facilitates extensibility and allows for the application of a model-independent workflow. Its overall structure and some of its features were inspired by the repository that in 2021 accompanied the work of Li et al. (2020). While building on this previous work, we propose a framework that is specifically tailored to the case of musical instrument modeling and designed for extensibility. From a data-centric perspective, the design specifications of our framework adheres to the following constraints: the output of the acoustics simulations must be organized in data structures that are compatible with standard machine learning frameworks; such data structures must be easy to move between local and remote machines; and the output of each simulation must be easy to replicate.

The proposed framework can be accessed here[3]. It is written in Pytorch and requires the installation of a few additional libraries, mainly for the visualization of the acoustics simulations and for logging purposes. Pytorch was chosen for its flexibility and scalability on parallel architectures, yet the resulting datasets are not tied to this specific language (more details in Section 3.2). The dataset generation workflow that we propose acts upon three main types of components/scripts: solvers, numerical models and dataset generators. In the following subsections, we detail these components and then we introduce the workflow in each of its steps.

3.1 MAIN COMPONENTS OF SYNTHETIC MUSICAL ACOUSTICS DATASET GENERATION

Solvers. At its core, the Neuralacoustics framework contains solver scripts. Solvers implement numerical solutions of highly parametric PDE systems, capable of modeling entire families of musical instruments. They are stored in a dedicated directory of the framework, organized by the actual numerical method employed (e.g., finite difference, finite elements) and by the type of acoustic phenomena being modeled (e.g., surface transverse waves, air wave propagation, aeroacoustics). All solvers have both a set of specific parameters, that depend on the implementation details, and a set of common parameters. The latter include temporal and spatial resolution, size of the domain, duration of the simulation as well as the device where to run the code (CPU or GPU).

Currently, the framework includes three solvers, all based on finite-difference time-domain schemes. The first one was originally proposed by Adib (2000) and solves a PDE system capable of the simulation of damped transverse wave propagation in a two-dimensional medium. It can be used to model the basic acoustics of membranes, thin plates and rudimentary cymbals. The other two solvers tackle acoustic pressure propagation in 2D, and were ported from the OpenGL implementations proposed by Allen & Raghuvanshi (2015). The former is linear and it can be used to approximate woodwind bores; the latter includes non-linearities typical of brass instruments.

Numerical Models. Numerical models simulate specific musical instruments. To do so, each of these scripts loads a solver and sets some of its specific parameters, imposing for example constraints on the shape of the domain (i.e., spatial boundary conditions) or on the acoustic properties of the simulated materials. Furthermore, numerical models are characterized also by an excitation algorithm. Excitations can work as initial conditions simply aimed at setting the model into motion (e.g., initial displacement of a membrane), or as continuous models that excite the instrument throughout the whole duration of the simulation. Examples of continuous excitations may include basic sinusoidal pressure waves, precomputed glottal pulses Rosenberg (1971); Guasch et al. (2016) and reed models coupled with models of instruments' bores Bilbao et al. (2015); Allen & Raghuvanshi (2015). By setting the parameters of the underlying PDEs and defining an excitation input, numerical models can be used to simulate not only different instruments, but also different playing configurations of the same instrument (e.g., a membrane hit by a stick or by a mallet).

[3] ANONYMIZED FOR SUBMISSION

Each numerical model is composed of an implementation script and a configuration file. The configuration file exposes controllable parameters of the excitation algorithm implemented in the script, like the location of a hit or the area where an initial condition is applied. Along with these settings, the file may include any parameter of the solver that is not hard-coded by the model. These configurable parameters provide the ability to "tune" the behavior of the instrument and allow for the generation of datasets via the mechanism described in the next paragraph. The configuration file also includes general descriptions of the model and of its exposed parameters, along with the path to the solver in use. Both the implementation script and the configuration file of a model are hosted in the same directory, named after the simulated instrument.

For now, we included into the framework only four simple models, based on the aforementioned solvers. In particular, we implemented three models that simulate vibrating membranes, using as excitations impulses, noise and single spatial frequencies that match the horizontal modes of the surface, respectively. The fourth model leverages the linear acoustic pressure propagation solver to excite woodwinds bores with air wave pulses.

Dataset Generators. When used to train a network, a dataset needs to be as diverse as possible, to exemplify at best the mechanisms that must be learned/approximated (Goodfellow et al., 2016, Chapter 5). Dataset generators are designed to fulfill this need. They consist of algorithms that load a specific model and automatically sample its parameters' space, effectively running large numbers of simulations of the same instrument/configuration. Each of these simulations is characterized by different inputs to the same model/solver, leading to a diverse set of outputs (i.e., simulations). The framework includes two types of generators: random generators and grid-based generators. Random generators explore the parameters' space of the model using Python/Pytorch pseudo-random algorithms, driven by an arbitrary seed. This ensures determinism while avoiding clear patterns, and facilitates the reproducibility of results across different machines[4]. The number of random samples to take (i.e., the size of the resulting dataset) is passed to the generator as an input parameter. Grid-based generators do not rely on any random calculation, rather they sample the parameters' space in a linearly fashion. Each parameter to sample is assigned a range and a sampling step, then all the possible combinations of parameters are automatically computed—forming a "grid". Differently from the case of random generators, the total number of samples is not arbitrary but depends on the defined grid, and the resulting datasets need to be shuffled before being used for training.

Much like the case of numerical models, each dataset generator script is accompanied by a configuration file, including a path to the model that will be sampled. This file also permits to set the input parameters of the generator, effectively controlling how the model will be sampled. Example generator parameters include ranges and steps of the model's parameters to sample (for grid-based generators), the requested number of dataset entries and the current seed (for random generators), and the number of simulations that will be run at once as a batch. Configuration files also give the opportunity to hard-code some of the parameters the models expose, in case some features/properties need to be constant across the whole dataset. A practical example consists of precisely fixing the acoustic parameters of the instrument's material. In the simplest case, dataset generators are designed to explore a single instrument/playing configuration, and they are stored in a subdirectory of the model they are associated with. It is though straightforward to design dataset generators that sample more than one model and collate the results into a single dataset. This solution is quite convenient when a network has to learn the behavior of multiple configurations of the same instrument that are simulated by different models.

3.2 Dataset Generation Workflow

The main component scripts are not designed to be run directly. The framework includes instead an entry-level dataset generation script (*dataset_generate.py*), that allows for the correct use of generators, models and solvers, and partially hides the complexity of the underlying code and dependencies. This script can be accessed from the root directory of the framework. Once launched, it collects all the simulation data computed by a chosen generator script into an actual dataset. More specifically, dataset entries represent complete simulations, each associated to a different set

[4]We though noted that Pytorch deterministic algorithms may result in slightly different output numbers when the same generator is executed on different computer architectures.

of parameters; they consist of dictionaries, containing inputs to the model (excitations) and output solutions (acoustic waves) for every simulation time step.

The script is accompanied by an entry-level configuration script (*default.ini*) that can be used to choose: which dataset generator to run, the directory where the dataset will be saved, as well as the device where the code will run. The presence of the entry-level script permits to de-couple model-specific functions and parameters (accessible in the generator's configuration) from general/structural parameters that are applicable to the generation of any dataset (editable in the entry-level configuration), regardless of what instrument is exemplified and what low-level solver is used. Such an abstraction layer was introduced to make the framework more scalable and easier to extend.

The way in which datasets are structured and stored reflects the first two data-centric constraints we introduced at the beginning of this section, i.e., compatibility and portability. The entry-level dataset generation script outputs *MAT-files* (".mat" extension), a binary file standard introduced in MATLAB[5] to store workspace variables. We chose this specific format because it is often considered as one of the de-facto standards, and it is supported by a large variety of languages and frameworks used in machine learning—including Javascript, C/C++ and Python/Pytorch/Tensorflow. The generation of acoustics datasets may yield very large files, especially when simulations span big domains and long time windows. To avoid exceeding the maximum size supported by the native file system where the code runs, the entry-level dataset generation script is capable of splitting the dataset into several "chunks", each represented by an individual MAT-file. This solution comes in handy also when moving datasets between remote locations, for the transfer of large files may be subject to failures due to connection instability[6]. The number of chunks can be defined in the entry-level configuration file. The resulting dataset is structured as a directory, with inside all the chunk files sorted by order of creation. Eventually, when a dataset is loaded in memory (to train a network or to visualize its content), all the chunk files are transparently combined together back into a single dataset (more details in the next subsection).

The entry-level script compiles a log file, that is hosted in the dataset folder and contains a summary of the content of the dataset. Additionally, the log file reports the details of the generation process. More specifically, these details consist of all the parameters of the employed generator script, as well as those of the numerical model the generator is associated with. Thanks to this structure, any log file can function as a unique configuration file, that when passed to the entry-level script (in place of *default.ini*) sets both generator's and model's parameters, and allows users to automatically obtain an exact copy of the original dataset. This mechanism was designed to respect the third data constraint (replicability).

The complete dataset generation workflow is depicted in Figure 1 and can be summarized as follows. First, the user has to locate a numerical model that represents the specific instrument the dataset will exemplify. Descriptions and data available in the models' configuration files are in most cases enough to select the most appropriate script. Then, a dataset generator needs to be chosen, that samples the numerical model of interest. In this step, the user shall adjust the parameters exposed in the generator's configuration file, to make sure that the sampling procedure will result in data that well represent the instrument and its specific playing configuration[7]. Finally, the entry-level configuration file has to be edited with the path to the chosen generator, along with the other structural parameters. At this point, the user can run the entry-level dataset generation script and the resulting dataset will be computed and stored according to the requested settings. The compiled dataset log file can be shared with remote collaborators to replicate the dataset, avoiding the hassle of moving large files or going through the full workflow again; the only caveat is that the same version of the framework has to be installed on both ends (in particular, solvers, models and generators). Every step of the workflow can be carried out via command line, making it straightforward to check out the framework on remote machines and generate datasets on high performance computing clusters.

[5] `https://www.mathworks.com/`

[6] A common problematic scenario would be to move datasets to/from Google Drive when using Colab Notebooks.

[7] The specific metrics depends on the purpose and the application the dataset is computed for.

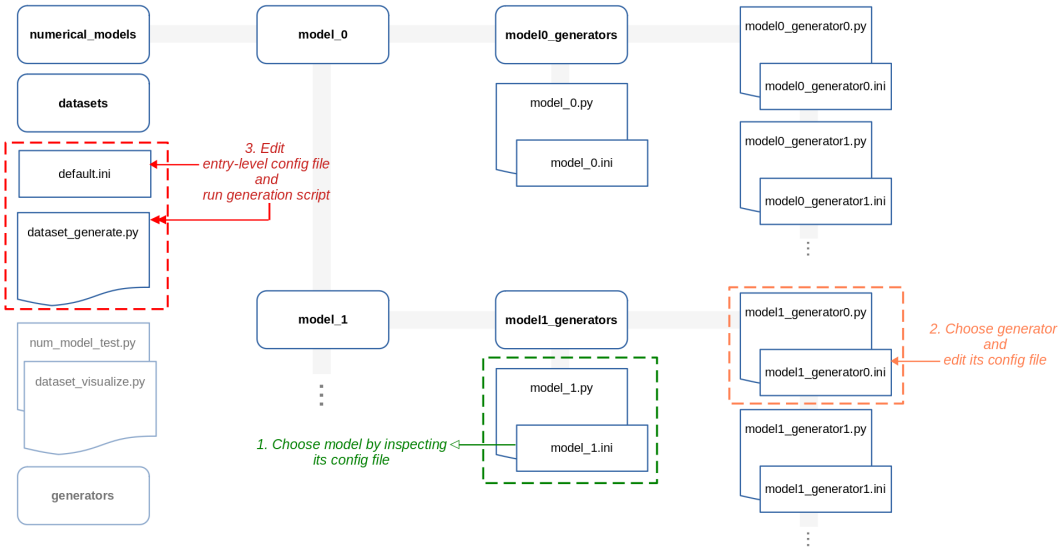

Figure 1: Diagram exemplifying the dataset generation workflow proposed within the framework. Folders' names are in bold. Only the generator's configuration file and the entry-level configuration file have to be edited by the user.

3.3 Utility Tools: Visualization and Data Points Extraction

In the root of the framework, the user can find two further entry-level scripts, *num_model_test.py* and *dataset_visualization.py*. Both can be set in the entry-level configuration file, which is organized in sections that gather the parameters of each specific script.

The *num_model_test.py* script is designed to test numerical models before they are used for dataset generation. Its mechanism is quite simple. A chosen model is run and all its outputs are sequentially visualized as plots, showcasing the propagating acoustic waves as an animation. Through the configuration file it is possible to choose the model to test and how long every "frame" of the animation (i.e., output) lasts. The actual parameters of the model are loaded from its configuration file, allowing for quick tuning and facilitating the comparison between different playing configurations.

Once a model is tuned and sampled with a generator, the content of the resulting dataset can be checked via the *dataset_visualization.py* script. Similarly to *num_model_test.py*, this script visualizes the outputs of the model as an animation, with the main difference that the frames are extracted from the dataset rather than being computed in real-time. Furthermore, *dataset_visualization.py* implements a windowing system that allows for the extraction of diverse sets of data points from the same dataset. To understand this mechanism, it is necessary to emphasize the difference between an "entry" within the dataset and a "data point" extracted from it. As detailed in Section 3.2, each entry in a Neuralacoustics dataset consists of a time series, representing the simulation of an instrument over a certain number of time steps. In the most general sense, a Neuralacoustics data point can be any sub-series of consecutive time steps found in a dataset entry. More than one data point can be extracted from a single dataset entry and the maximum size of a data point is equal to the total number of time steps of the simulation. In the latter case, only a single data point is extracted, which coincides with the full entry.

The windowing algorithm is part of a data point extraction tool, that retrieves data points by means of repeatedly windowing the entries' time series (collecting a data point per window). The process is depicted in Figure 2. The windowing parameters can be edited in the entry-level configuration file. The main ones are: the size of the windows, the stride applied between consecutive windows, and the dataset entry where the windows are applied. A further parameter allows to repeat the extraction over a number of consecutive entries, increasing the total number of frames visualized. To simply

visualize the full simulations within each entry, the user can set either the size of the window equal to the number of time steps of each entry, or the stride equal to the size of the window[8].

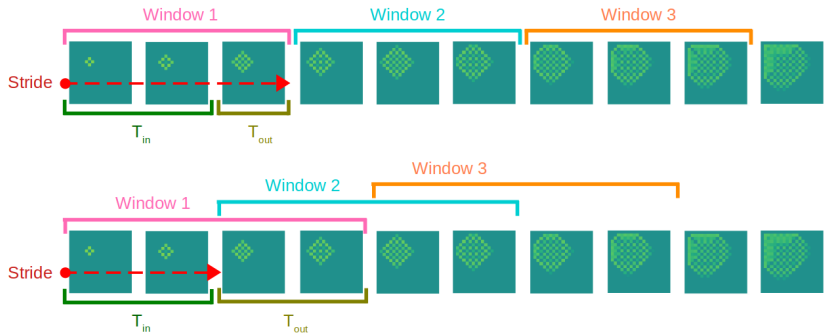

Figure 2: Windowing algorithm used to extract data points from Neuralacoustics datasets. The same dataset entry is processed via two different sets of parameters, to exemplify the extraction of different data. The diagram depicts only the solution time steps stored in the entry's dictionary.

The necessity to differentiate between dataset entries and data points, as well as to introduce a windowing algorithm¸ derives from neural operators and how they can be employed in the context of time-dependent problems. In particular, neural operators can be used to map the state of a numerical model across the last T_{in} time steps to its state across T_{out} future time steps; and during inference, they can be used auto-regressively to generate a continuous output. Yet, the number of data points used for training and the number of time steps (both input and output) each point is composed of have a strong impact on the accuracy of the operator Wang et al. (2020); Li et al. (2020). The data point extraction tool included in the Neuralacoustics framework addresses both these issues, by means of providing control over the size and the stride of the windows.

When used to collate a training set, the extraction tool keeps applying the windows across each entry of the dataset, until the number of data points requested by the user is obtained. It is important to notice, though, that the values returned by the tool contain more information than simple feature vectors, for two reasons. First, the size of the window is defined as $T_{win} = T_{in} + T_{out}$; hence, the extracted values contain both the states pertaining to the initial T_{in} time steps (i.e., the features) and the states pertaining to the following T_{out} time steps (i.e., the label to predict). Second, as described in Section 3.2, the values stored in each dataset entry include excitations to the model and resulting solutions. When extracted, these pieces of data may be combined in different ways to represent the actual state of the model in each time step, depending on the specifics of the problem and of the solver. This means that some extra logic is necessary to define features and labels and be able to employ the data points for training. While increasing the overall complexity of the framework, this design detail makes data handling as generic as possible, allowing for the application of the same workflow even when very different numerical models/simulations/solvers and neural operators are in use. We share the details of a simple example of data handling logic for training in the next section.

4 Training Neural Operators

The Neuralacoustics framework also aims to facilitate the design of neural operators capable of solving acoustics PDEs. Currently, this part of the framework is comparatively less developed than the dataset generation workflow introduced in Section 3, but basic training and evaluation pipelines have been completed. Besides, preliminary training results have been obtained, which will be discussed in Section 4.1.

Within the framework, neural operators are stored under the "networks" folder. Similarly to numerical models, each operator is composed of an implementation script and a configuration file. The script contains a highly-parameterized Pytorch implementation of the corresponding network, and

[8]A value of 0 or less will automatically set these parameters for these configurations.

the configuration file allows the users to easily control network-specific hyperparameters without looking into the implementation details. As of now, only a two-dimensional Fourier neural operator as introduced in (Li et al., 2020) is available in the framework. This was chosen as the first network to be ported into the framework because of its ability to solve time-based problems, as well as for its somewhat simple internal structure[9]. The port exposes a set of controllable hyperparameters, including spectral layer stack number, hidden size and Fourier transform modes. The network performs prediction one time step at a time and produces consecutive results in an auto-regressive manner.

In the root directory, the user can find a script (*train.py*) that carries out a full network training pipeline. Corresponding to this script is the training section in the *default.ini* file, where the user can specify a range of training configurations, including: dataset and network selection, size of training and validation sets, data points extraction details (e.g., T_{in} and T_{out}), learning and optimization parameters. This separation of training parameters and network hyperparameters ensures generalizability, potentially supporting the training of networks that diverge from the structure of the implemented Fourier neural operator. The training script operates on the dataset via the data points extraction tool described in Section 3.3; the logic used to combine excitations and solutions into state tensors is still hard-coded in the script. Once both data points and network are set, the script trains the neural model with the specified learning parameters. Weight checkpoints are also saved, using a customizable step interval.

Similarly to the data generation workflow, the training script compiles a log file that serves as a summary of the training details. All parameters related to the training process, alongside dataset generation parameters and network hyperparameters, are recorded so that the log file itself could be used as a unique configuration file for thoroughly replicating the neural model. Each time a training is performed, a corresponding directory is created under the "models" folder, where the weight checkpoints and the log file are stored for future reference.

Currently an entry-level evaluation script called *eval.py*, paired with the evaluation section in *default.ini*, provides intuitive insights into a trained network's performance. The users can specify the network checkpoint to evaluate, as well as the exact dataset and data entry for running inference on. The evaluation script visualizes the predicted domain state along with the ground truth domain state (computed by the original numerical model) and their difference, for a chosen number of time steps. An example of visualization is provided in next section.

4.1 Preliminary Results

We trained a two-dimensional Fourier operator network on a dataset generated with the membrane model including impulse excitation (Section 3.1, Numerical Models paragraph). We sampled the model with a generator that randomizes the location of the impulse across the domain (i.e., the area of the membrane), as well as the amplitude of the impulse. Each simulation spans 25 time steps. Both scripts are available in the framework. The width and the height of the 2D domain are both set to 64. The size of the training set is 25000 data points, while 5000 data points are used as validation set. Data points are extracted only from the first 20 time steps of each entry (this is a further setting available in the configuration file). The state of the model in each time step is chosen as the solution tensor in that step and the excitation tensors stored in the dictionary are ignored. The input step number T_{in} is 10 (the feature vector is composed of the previous 10 states) and the output time step T_{out} is 1 (the label is the next state). The neural operator has 4 spectral layers with 12 Fourier modes within each layer and a hidden size of 20. We trained the network for 800 epochs with Adam optimizer and an initial learning rate of 0.001.

The final loss amounts to 0.0355. Figure 3 shows two snapshots of the visualization generated via the evaluation script. The test set extracted by the dataset is not limited to the first 20 time steps, but spans the full duration of the simulations (25 steps). As can be observed, the 2D domain's state outputted by the trained network is a close estimate of the ground truth simulated via the numerical solver. Moreover, through an auto-regressive inference approach, the trained network is able to carry out a good prediction of how an impulse propagates beyond the first 20 time steps exemplified in the training set. This suggests that the learning captured—at least in part—the underlying physical mechanisms of wave propagation.

[9]The source code of the Fourier neural operators is available at `https://github.com/zongyi-li/fourier_neural_operator`

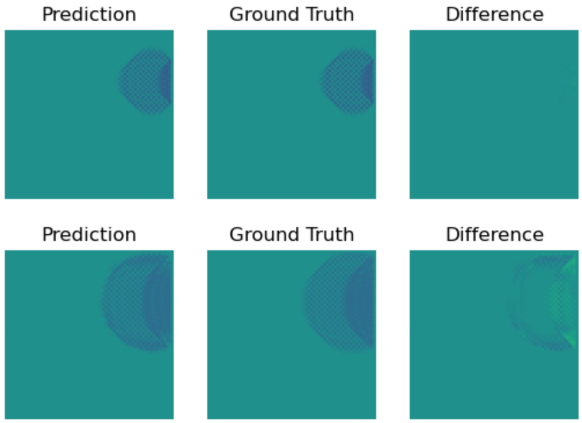

Figure 3: Evaluation visualization. From left to right, the three columns respectively shows the 2D domain state predicted by the trained network, the ground truth, and their difference. The top row shows the states at time step #15 (exemplified in the training set), and the bottom row shows the states at time step #25 (never reached in the training set).

Currently the visualization only provides a case-by-case qualitative evaluation of our deep learning network's performance. Quantitative evaluation methods on large datasets will be implemented in the near future, so that the framework can be used to precisely compare performance across multiple networks.

5 Conclusions and Future Work

In this paper we introduced the Neuralacoustics framework, an open access and open source collection of Python scripts and tools designed to facilitate the application of deep learning in the context of acoustics simulations and musical instrument modeling. In particular, the framework responds to the need for standards to combine the output of diverse acoustics simulations into datasets, and to use them for training.

The main components of the framework are numerical models and acoustic PDE solvers. These are arranged in a modular structure, that permits the application of a robust workflow for the generation of heterogeneous musical acoustics datasets. The generation process outputs data structures compatible with standard machine learning frameworks, and is designed to maximize portability and replicability. The Neuralacoustics framework features also a section dedicated to the training and the evaluation of neural operators using the generated acoustics datasets. While still in progress, this part of the workflow is functional and leverages a modular structure similar to the one proposed for the dataset generation process.

The framework is constantly improved and extended. At the current stage of development, it includes Pytorch implementations of four numerical models and three solvers, all previously introduced in the literature. As showcased by the preliminary data presented in this work, these are enough to train a state-of-the-art neural operator and test its performance in the context of basic musical acoustics. Furthermore, the current implementations are designed to work as blueprints for the porting of additional models and solvers. With the release of this work, we aim to empower all the researchers working in this emerging field with new tools for the development and the sharing of their own implementations. We believe that this in turn will foster the introduction of benchmarks and the development of novel deep learning approaches to acoustics modeling.

References

Artur B Adib. Study notes on numerical solutions of the wave equation with the finite difference method. *arXiv preprint physics/0009068*, 2000.

Andrew Allen and Nikunj Raghuvanshi. Aerophones in flatland: Interactive wave simulation of wind instruments. *ACM Transactions on Graphics (TOG)*, 34(4):1–11, 2015.

Marc Arnela and Oriol Guasch. Two-dimensional vocal tracts with three-dimensional behavior in the numerical generation of vowels. *The Journal of the Acoustical Society of America*, 135(1): 369–379, 2014.

Saakaar Bhatnagar, Yaser Afshar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, 64(2):525–545, 2019.

Stefan Bilbao. *Numerical sound synthesis: finite difference schemes and simulation in musical acoustics*. John Wiley & Sons, 2009.

Stefan Bilbao and John Chick. Finite difference time domain simulation for the brass instrument bore. *The Journal of the Acoustical Society of America*, 134(5):3860–3871, 2013.

Stefan Bilbao, Alberto Torin, and Vasileios Chatziioannou. Numerical modeling of collisions in musical instruments. *Acta Acustica united with Acustica*, 101(1):155–173, 2015.

Stefan Bilbao, Michele Ducceschi, and Craig Webb. Large-scale real-time modular physical modeling sound synthesis. In *Proc. Int. Conf. On Dig. Audio Eff.(DAFx 2019), Birmingham, UK*, 2019.

Jan Blechschmidt and Oliver G Ernst. Three ways to solve partial differential equations with neural networks—a review. *GAMM-Mitteilungen*, 44(2):e202100006, 2021.

Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: A review. *Acta Mechanica Sinica*, pp. 1–12, 2022.

Nicolas Castagné and Claude Cadoz. 10 criteria for evaluating physical modelling schemes for music creation. In *DAFx03*, pp. 7–p, 2003.

Rene Emile Causse, Joel Bensoam, and Nicholas Ellis. Modalys, a physical modeling synthesizer: More than twenty years of researches, developments, and musical uses. *The Journal of the Acoustical Society of America*, 130(4 journal=Computer music journal, volume=16, number=4,): 2365–2365, 2011.

Bernat Font, Gabriel D Weymouth, Vinh-Tan Nguyen, and Owen R Tutty. Deep learning of the spanwise-averaged navier–stokes equations. *Journal of Computational Physics*, 434:110199, 2021.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Steven Guan, Ko-Tsung Hsu, and Parag V Chitnis. Fourier neural operator networks: A fast and general solver for the photoacoustic wave equation. *arXiv preprint arXiv:2108.09374*, 2021.

Oriol Guasch, Marc Arnela, Ramon Codina, and Hector Espinoza. A stabilized finite element method for the mixed wave equation in an ale framework with application to diphthong production. *Acta Acustica united with Acustica*, 102(1):94–106, 2016.

Yuehaw Khoo, Jianfeng Lu, and Lexing Ying. Solving parametric pde problems with artificial neural networks. *European Journal of Applied Mathematics*, 32(3):421–435, 2021.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

Harri Renney, Silvin Willemsen, Benedict Gaster, and Tom Mitchell. Hypermodels-a framework for gpu accelerated physical modelling sound synthesis. In *NIME*, volume 22. PubPub, 2022.

Aaron E Rosenberg. Effect of glottal pulse shape on the quality of natural vowels. *The Journal of the Acoustical Society of America*, 49(2B):583–590, 1971.

Samuel H Rudy, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Data-driven discovery of partial differential equations. *Science advances*, 3(4):e1602614, 2017.

Julius O Smith. Physical modeling using digital waveguides. *Computer music journal*, 16(4):74–91, 1992.

Marc Sosnick and William Hsu. Efficient finite difference-based sound synthesis using gpus. In *Proceedings of the Sound and Music Computing Conference*, pp. 42–44, 2010.

Rui Wang, Karthik Kashinath, Mustafa Mustafa, Adrian Albert, and Rose Yu. Towards physics-informed deep learning for turbulent flow prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1457–1466, 2020.

Takatoshi Yokota, Shinichi Sakamoto, and Hideki Tachibana. Visualization of sound propagation and scattering in rooms. *Acoustical science and technology*, 23(1):40–46, 2002.

Victor Zappi, Andrew Allen, and Sidney S. Fels. The hyper drumhead: Making music with a massive real-time physical model. In *Proceedings of the 2017 International Computer Music Conference, ICMC 2017, Shanghai, China, October 16-20, 2017*. Michigan Publishing, 2017. URL `https://hdl.handle.net/2027/spo.bbp2372.2017.026`.