

A Data, Data Processing, Management, and Storage

Converting the output data from the split-step Fourier method simulation for the nonlinear optics process to a form that is both usable by a machine learning network and that is appropriate for long-term storage and management is a nontrivial task. The output from the original simulation includes, among other information, 3 full fields—SHG 1, SHG 2, and SFG—in both time and frequency domains for every slice in the crystal. Each element in the array is natively stored as a complex-128 bit number, so the total storage requirements for each run of the simulation, assuming all of this information must be stored, is $3 * (32768 * 128) / 8 = 1.57$ MB for one slice in one domain, then $1.57 \text{ MB} * 2 = 3.15$ MB for time and frequency domains, and finally $3.15 \text{ MB} * 100 = 315$ MB for all the slices in the crystal. Even for our modest 10,000 unique examples for simulation runs, this would yield nearly 3.15 TB of data and would require a network to have an input size of 98,304 complex parameters.

Instead, we reduce the data size and set up a data repository for this reduced data. While the main manuscript briefly describes the data preprocessing steps, here we provide more details and describe where and how the data is stored, as well as how to access and process it.

A.1 Data

All data is generated using our start-to-end (S2E) laser modeling framework [1]. As discussed in the main manuscript, our model of the photoinjector laser system consists of a mode-locked oscillator, pulse shaper, amplifier, and the dispersion-controlled nonlinear synthesis (DCNS) nonlinear optical (NLO) upconversion scheme. For data generation, we alter the pulse shaper parameters in order to modulate the laser pulses fed into the amplifier and then eventually fed into the DCNS section. The pulse shaper controls the spectral amplitude and phase of the signal. Our total parameter space allows for second-order dispersion (SOD) to range from $-1e4 \text{ fs}^2$ to $1e4 \text{ fs}^2$, third-order dispersion (TOD) to range from $-1e5 \text{ fs}^3$ to $1e5 \text{ fs}^3$, hole width to span 0.1 nm to 4 nm, hole position to span 1022 nm to 1036 nm, and hole depth from 0 to 95%. This combination covers a significant portion of the total parameter space for our pulse shaper used in lab and is based on its physical limitations. We randomly select parameter combinations within this space, ensuring that all endpoints are covered and that half of the samples have no hole (depth of 0).

All other model parameters (for oscillator, amplifier, and NLO DCNS method) are held constant in this simulation. We offer two separate repositories of data based on user preference. One contains files that have only undergone stage 1 preprocessing (as described in the main text and which will be elaborated on in the next section) and which contains the pulse shaper parameters. We also provide access to unified H5 files that have undergone both stages of preprocessing and are directly ready for the model described in our paper. This set, however, does not include the pulse shaper parameters. Section A.3 provides more details on how to access this data. Furthermore, we can work with any group interested in using our S2E framework to apply it to a new laser system to generate data.

A.2 Data Preprocessing

The data preprocessing steps are shown in fig. 1. We start by taking the original fields (displayed in intensity—solid line—and phase—dashed line— form) and downsampling and then cutting to arrive at a reduced amount of data. These are what are stored in the numpy files in one of the data repositories. Then these three fields (downsampled and cut for SFG, SHG 1, and SHG 2) are then concatenated according to version 2 of the second stage preprocessing where the real components of the three fields come first and then the imaginary components. These are then scaled using the MinMaxScaler from scikit-learn¹ such that each component in the full concatenated vector is between 0 and 1. Finally, these vectors are saved as 32-bit floating point format. This stored data is what we provide access to in the H5 files. The next section describes how to access both of these repositories. Our primary focus is on the ones containing the H5 files, though, as they are more straightforward for quickly implementing our model.

¹<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>

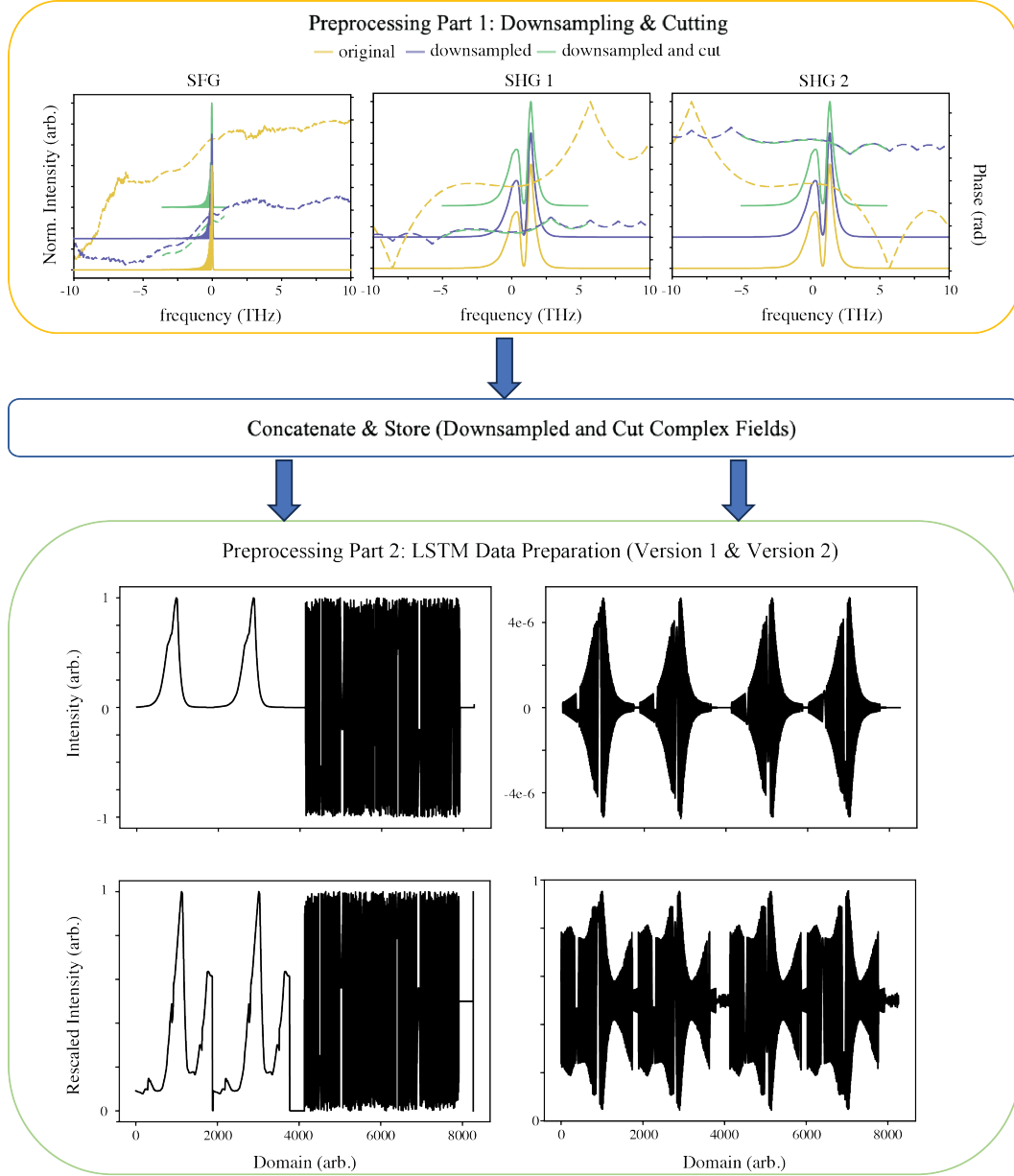


Figure 1: Shows the data preprocessing steps from original data reduction, to storage, and finally reformatting for the LSTM. In the first set of three plots, the solid lines are intensity and the dashed lines are phase.

A.3 Data Repository Usage

All data is stored under the Stanford Digital Repository (SDR), which guarantees secure, archival storage with persistent access via a persistent URL (PURL) as well as a DOI ². The data in both repositories is stored under a Creative Commons license CC0 1.0 Universal, and we (the authors) bear all responsibility in case of violation of rights. For convenience we give access to two formatted versions of our data. One is the data exactly prepared for the long short-term memory (LSTM) model described in the manuscript using H5 files. The other uses a combination of numpy files for data and pickle files with parameters.

The first can be accessed at <https://purl.stanford.edu/nf288ry2198> and <https://doi.org/10.1364/OE.520542>. Here there are two H5 files, one labeled X for input data to the network and one labeled Y for truth outputs. The X data H5 file has 100 datasets. Each dataset has dimension (10,000, 10, 8264). The 10,000 comes from taking 100 unique examples of simulation through the NLO process, which takes 100 steps through a crystal medium. Since each step in the crystal is usable training data, then this yields $100 \times 100 = 10,000$. Then for the LSTM which expects 10 inputs as described in the main manuscript, the 10 previous slices in the crystal are stacked. Finally, the 8264 comes from the second stage preprocessing using version 2 which yields the vectors with real and imaginary for the three fields. The Y H5 file is similar except its dimension is (10,000, 8264) since the 10 slices used in X are used to predict one output from Y.

Now all of this data is scaled and ready for the LSTM. However, to analyze the data or reformat, one must first inverse scale. The repository includes a pickle file of the scaler. The analysis code (described in section B) uses this scaler to inverse transform the data and then combines the real and imaginary portions of each field back together in order to extract the full fields and eventually plot the output in intensity/phase format. Similarly, if one wanted to try version 1 preprocessing with the intensity, phase, energy arrangement using these H5 files, one would need to apply this same process to retrieve the fields and then re-separate them by intensity, phase, and energy. The fourth file in the repository is the model weights file for the LSTM presented in the paper.

The alternate repository takes the data after stage one preprocessing and thus keeps them in complex form and does not scale the data. This repository, found at <https://doi.org/10.25740/fv412tg4309>, contains 20,001 files—10,000 “.npz” files for the primary data, 10,000 “.pkl” files containing simulation parameters, and 1 text file for a data description that matches the abstract on the SDR. Each of these 10,000 numpy files contain data for each 100 slices in the crystal. This data requires further preprocessing to be compatible with the LSTM. The preprocessing functions are supplied in the Github repository.

For both versions of the data repositories, accessing the data is as simple as visiting the above links and downloading the files to one directory. The authors can offer additional support should a user encounter difficulties.

B Reproducibility

This section describes how to access our code and replicate our procedure.

B.1 Code Organization and Preprocessing

The code can be found at https://github.com/jhirschm/DCNS_LSTM_Public. This Github repo contains 5 directories:

- Data: vectors required in some preprocessing and analysis
- LSTM: code for the definition and initialization of models (discussed in section B)
 - model.py
 - main_fn.py
- Analysis: code for analyzing trained network (discussed in section B)
 - analyze_reim.py

²<https://library.stanford.edu/research/stanford-digital-repository>

- util.py
- Preprocessing: code for preparing downloaded SDR data for LSTM (discussed in section A)
 - preprocessing_version1_intPhEn.ipynb
 - preprocessing_version2_reIm.ipynb
 - data_visualizer.ipynb
 - util.py
- Utilz: code for the loading the dataset, defining the loss functions, training the models and running inference and time analysis for the models.
 - data.py
 - loads.py
 - losses.py
 - main_fn.py
 - training.py
- main.py
- requirements.txt
- README.md

The Data directory contains several important vectors for analysis and data conversion. The Analysis folder contains the functions for inspecting the test output the LSTM and replicating the analysis from the manuscript. The Preprocessing folder contains the notebooks one can use for taking the numpy files from the alternate data repository or (with some custom code) back converting the H5 files to arrive at the two second stage preprocessing versions. However, these preprocessing steps are not necessary if the H5 files are used directly. The Utilz directory contains various helper functions and models. The “main.py” file runs the training by taking in command line parameters. The README file contains additional instructions and examples how to run the code. The requirements file details which packages are required.

While we recommend using the H5 files from the <https://purl.stanford.edu/nf288ry2198> and <https://doi.org/10.1364/OE.520542> repository, if one chooses to use the alternative repository then preprocessing will be needed. In Preprocessing, the two preprocessing files are version 1 and version 2 methods for the second stage of data preprocessing (described in the main manuscript). Running either of these Jupyter Notebooks requires the data from the SDR to be downloaded (and the directory location to be known) and an output directory to be created where the processed data will be stored. The script will then save the data (X_i and y_i), the MinMax normalized version of the data (as X_{new_i} and y_{new_i}) in the output directory. The scaler for the normalization will also be saved (both during the data processing itself and as a backup after it finishes processing). There is an option to stop the generation early if you need to run a few to just test the functionality. Beware that running both data preprocessing files with the same output directory will overwrite that data, so make sure to make unique directories. Lastly, the data_visualizer file allows for quick checks of the generated data. Ensure that directories point to the correct locations. Running the entire notebook will plot an example chosen from the data saved from preprocessing (chosen via file name and the with variables “ii” and “jj”). It will also separate the fields so that they can be viewed in a more natural way as intensity and phase.

The LSTM directory contains the model file which defines several LSTM-based neural network architectures using PyTorch, as the proposed architectures to solve our defined problem. These models feature a variety of configurations such as bidirectional LSTMs, varying dropout options, and different numbers of hidden layers and post-LSTM linear transformations. The class called LSTMModel is the one that we chose to train and present for this paper. In the same directory, the main_lstm function in main_fn.py acts as an entry point for setting up and training different LSTM-based neural network models defined in LSTM/model.py. It imports the model classes and utility functions for dataset loading and the main training function (elaborated on in section B.2). The function begins by configuring a dictionary with model parameters such as input size, hidden layer size, number of layers, and dropout rates, which are taken from the input arguments. It then selects the appropriate model based on the specified model type in the arguments. After initializing the model with these parameters, it loads training, validation, and testing datasets using a utility function that accepts the input arguments. Finally, it calls a primary function that handles the training process,

passing the model along with the datasets and additional model configurations. This setup allows for flexible experimentation with different LSTM configurations and datasets, streamlining the process of training and evaluating models.

In Utilz, the CustomSequence class defines a custom dataset class for handling data stored in our H5 files. This class allows for the dynamic loading of data points and labels based on specified indices, supporting different modes of operation such as training, testing, and analysis through the load_mode parameter. It also provides functionality to load the data either entirely or in parts depending on the mode, and it optionally loads data directly into GPU memory if specified. There's also an extended class, CustomSequenceTiming, which adds the capability to time the data loading and inference processes by loading and maintaining the entire dataset in memory for faster access. The constructors of these classes take in the path to the data directory, the desired files indices, and a boolean flag called "load_mode".

The Utilz/loads.py file is designed to facilitate the initialization of custom loss functions and datasets. It defines a function get_custom_loss that maps string identifiers to actual loss function implementations such as mean squared error (MSE) [defined in Utilz/losses.py] and weighted MSE losses. It allows the configuration of these loss functions based on parameters passed through to it. The get_datasets function dynamically constructs datasets for training, validation, and testing phases. It leverages the CustomSequence and CustomSequenceTiming classes to handle different modes of data loading, supporting features like loading directly to GPU.

The main_function in the Utilz/main_fn.py file acts as a centralized control hub for executing various operations related to model training, analysis, and prediction, using configurations passed via an args namespace, which is directly set from the command line parameters the users sends to the program at the time of execution. It begins by fetching a custom loss function tailored to the model's needs using the get_custom_loss function. Then, depending on the flags set in the args, the function can execute different paths.

For timing analysis (section B.4), setting args.custom_code to 1 or 2 will either time previous models or load current model parameters and time them. If set to 3, it calls the functions to calculate and plot the histograms for our mixed domain evaluation metric. For data analysis (sections B.3 and D), if args.do_analysis is enabled, it performs an analysis-only mode that logs and processes data for a specified model, leveraging the do_analysis function. If prediction is enabled (args.do_prediction), it predicts using the model and performs analysis post-prediction, which is useful for validation or real-world application scenarios. If none of the special flags are set, it decides between hyperparameter-tuning and training or straightforward training and testing based on args.tune_train. This decision dictates the use of either tune_and_train or train_and_test functions, setting the course for model optimization and evaluation. The flags set in the "args" provide us with the options for choosing CPU/GPU, whether to use validation set (during training), whether and where to save checkpoints and model weights (for training), and where and how to output the predicted results (for predicting)

The Utilz/training.py file provides a comprehensive suite of functions designed to facilitate various stages of training and evaluating our machine learning models.

B.2 Training and Prediction

Training and prediction are accessed from the main.py script and altered via command line parameters (see the README file in the repo for examples). This main file uses the functions described previously. For predicting, there are two primary modes. During inference where one wants to run the LSTM for the entire nonlinear process through the whole crystal, CustomSequence has to be created with "load_mode=True" so that only every other 100 samples are used for predicting. This way the prediction process starts with the input to the crystal and uses that to propagate to the final output. For training, the CustomSequence must be created with "load_mode=False" to load every single sample since each slice in the crystal can be used for training the LSTM.

The training and predicting modules are all provided, together with the options for choosing CPU/GPU, whether to use validation set (during training), whether and where to save checkpoints and model weights (for training), and where and how to output the predicted results (for predicting). During predicting, one can pass in either a newly-initialized model to train from scratch or a path to the location where the weights of a pre-trained model are saved, giving more flexibility in checking the results of some previous epochs, or even training a previous model for additional epochs.

B.3 Analysis

The primary analysis involves inspecting efficacy of model to replicate the simulation results. We explored several loss functions and evaluation metrics tailored to our application. A user can find these in Utilz/losses.py where we define the two main important loss metrics (weighted_MSE and “calculate_mixed_MSE_metric”). It also includes the function “visualize_MSE_errors” which visualizes the histograms of the mixed MSE metric as seen in fig. 4.

B.4 Timing Analysis

To run the timing analysis, one first needs to use the CustomSequenceTiming class instead of reusing the CustomSequence class with test_mode=True. Note this class loads all the prediction instances into memory at once. This is feasible because only 1 in every 100 instances is used during the prediction phase, so we can fit the entire prediction dataset into memory. This entire process takes only around 0.46 seconds when using the .h5 file format.

The time_prediction function gives users choices of using CPU/GPU and batch size for predicting. Both the predicted results and the time elapsed (in seconds) are generated as output. Similarly, one can pass in either a newly-initialized model architecture and the path to its weights or a model with weights loaded.

C Additional LSTM Results

The manuscript detailed our best results that were a result of model architecture tuning, including exploring bidirectional LSTMs, varying dropout, various numbers of hidden layers and post-LSTM linear transformations, as well as finding loss functions that led to desired results. Fig. 2 shows the training and validation loss for this model.

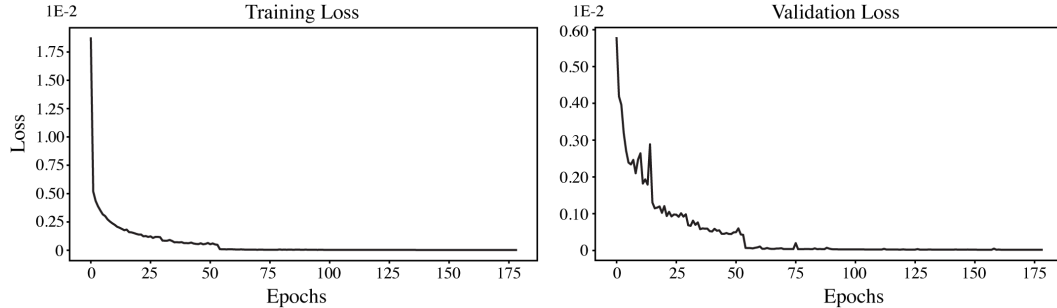


Figure 2: Shows train and validation loss for the LSTM model from manuscript using version 2 data preprocessing.

Before this fine-tuned search for the model trained on version 2 second-stage preprocessing, we also explored some early models based on version 1 preprocessing. In this exploratory initial study, which was only trained over 30 epochs, the best results had a model consisting of a single LSTM layer with two linear layers using ReLu and Sigmoid, respectively. However, there were early indications that the models were not learning as fast on this version 1 preprocessing data format compared to version 2, likely because of the noisy nature of the phase. Fig. 3 shows some of these early test set predictions training on version 1 preprocessing. Here, both the predicted SFG and SHG 1 spectral intensities (solid lines) for minimal and large hole and for both time and frequency examples don’t match the true values. This is especially true for the SFG magnitude where the intensity profile is nearly five times smaller and for the SHG 1 hole where a hole is predicted in fig. 3c when no hole should be present. Furthermore, the learned phase for the SFG cases in the frequency domain only matches near the peak intensity. For SHG 1, the phase in the frequency domain matches, but, in the time domain, the phase begins to diverge. In early exploratory studies of similar models but for version 2 preprocessing, the results were more promising and therefore we continued with version 2 preprocessing.

This version 1 data preprocessing seemingly led to worse results for several reasons. The phase, as shown in manuscript fig. 4a and 4b, is a difficult function to learn because of the large jumps from 0 to 2π . Furthermore, we pre-normalized the intensities (main manuscript fig. 4a) to be between 0 and 1 by normalizing them to energy. While this still preserves all information, it does compact the information that represents the magnitude of the profiles into one value for each field and then relies on the network learning these energy values properly. If the energy value for the SFG process is not learned successfully, then the scaling at the output will be wrong as well. This is a potential problem since we see the SFG predicted values are too small. Likely, all of these issues can be addressed with proper weighting of loss function and model hyperparameter adjustments. Additionally, one might be able to apply a smoothing function to the phase based on knowledge that the value of phase has no physical importance in regions where intensity is low. This would remove many of the oscillations found in wrapped phase. It should be noted that when we plot phase, we always plot unwrapped phase; however, it is difficult to re-wrap unwrapped phase and therefore only the initial wrapped phase should be used in training the models. Because these early results were not as promising as the results from stage-two preprocessing version 2, version 1 was not further explored in this study.

D Data Evaluation

To the best of our knowledge, there is no single, adopted evaluation metric in the optics and photonics machine learning community as the desired outcome is application dependent and diagnostic instrument dependent. In simulation, often the entire field information is known in both domains. However, in experimental setups, measuring laser pulses is more complicated. Spectrometers can provide information about the spectral intensity of the pulse but will not provide any information about the phase. Autocorrelators and cross-correlators can obtain temporal intensity measurements but without temporal phase. More advanced measurement techniques like Frequency-Resolved Optical Gating (FROG) and Spectral Phase Interferometry for Direct Electric Field Reconstruction (SPIDER) produce spectrograms of the pulses and, through post-processing algorithms, obtain the intensities and phases in both time and frequency domains [2]. However, FROG and SPIDER measurements are limited on some of the pulse attributes. Other characteristics of the pulses, like average power and repetition rate, can usually be obtained with ease. Moreover, different applications emphasize different measurement techniques. For instance, spectroscopy experiments tend to be most concerned with the spectrum of the pulse. In our use case for the photoinjector laser, we are primarily focused on the final output temporal intensity. However, since we want to keep our simulation studies more broadly applicable, we emphasize the importance of both domains.

We trained in frequency, as we could reduce data size more effectively in frequency and thus reduce total number of model parameters. Our mean-squared error (MSE) results calculated during training for the loss function all operated on the full-field, specifically on the reformatted data that separated the three fields' real and imaginary contributions. This method still presents some ambiguities since the phase of the field has no significance when the intensity of the field is near 0. There might be a way to boost the significant portions by separating the fields into intensity and phase and then multiply the phase vector by a filter function that takes the intensity vector as input, essentially using the intensity to roll-off the phase during training.

For testing though, we inspect both the frequency and temporal intensities since they are closely tied to physical observables. Since the prediction is in frequency domain, transforming the output to time provides some indication that the frequency intensity and phase (or real and imaginary portions) were captured correctly. As discussed in the manuscript, we then create a weighted normalized MSE (NMSE) function so that specific applications can adjust the emphasis on frequency versus temporal domains, making it a more comprehensive evaluation metric for the optics and photonics community that can adapt to the application requirements.

Fig. 4 shows the distribution of errors using the even weighting of the NMSE metric for the test data for running the LSTM through the entire crystal, meaning errors on prediction slice-by-slice accumulate. The distributions for each field are skewed with the mean higher than the median for each.

Table 2 in the main manuscript shows the errors for SFG, SHG 1, and SHG 2 for both examples for the even weighting case ($c_t = c_f = 0.5$). The minimal hole example was in the 5th for all fields with an error 3 orders of magnitude less than the mean SFG error and 2 orders of magnitude less

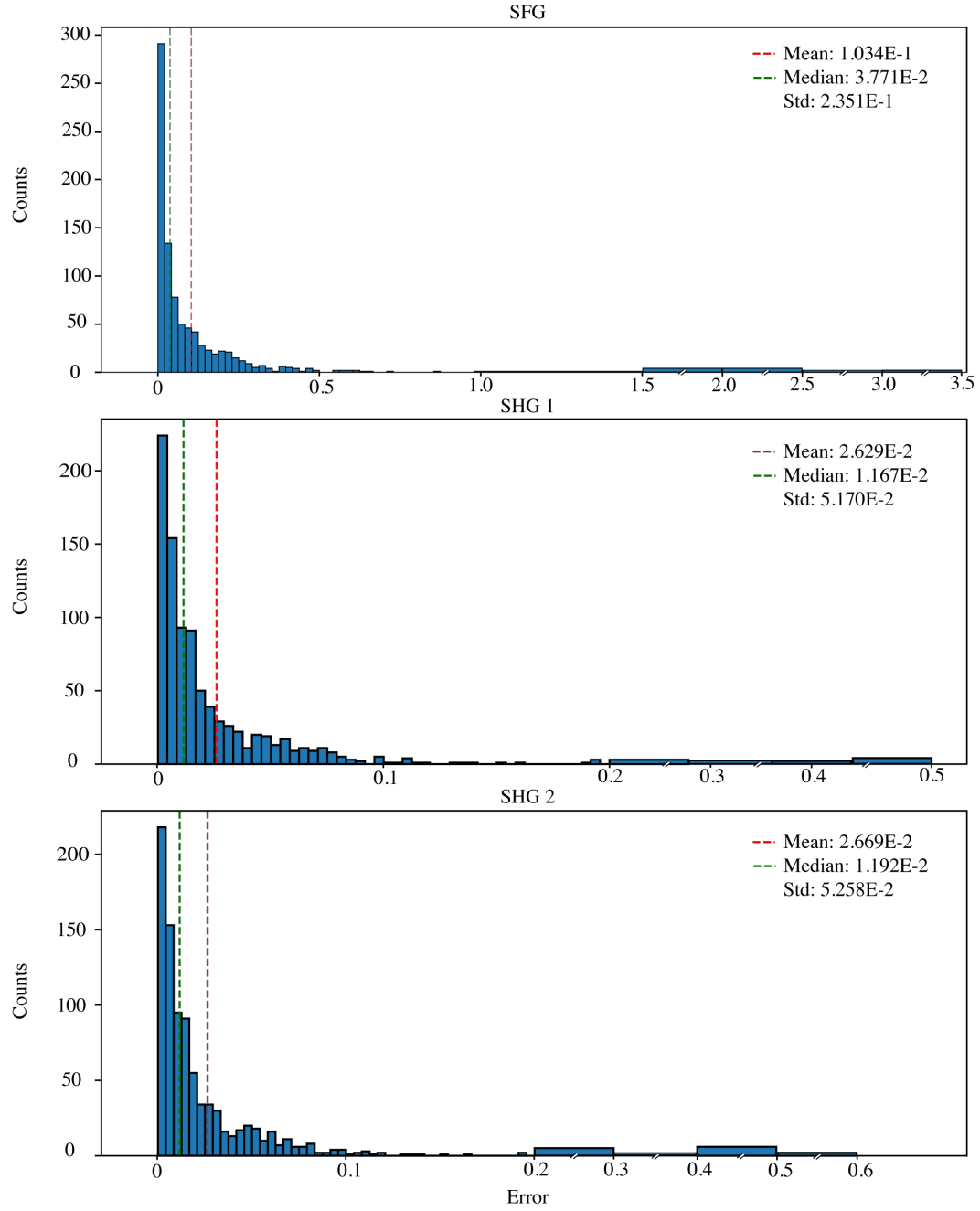


Figure 4: Distribution of errors for the weighted NMSE for SFG, SHG 1, and SHG 2. Red line indicates mean and green indicates median. The higher error bins are sized larger for better readability and the jump in the x-axis is demarcated with parallel lines.

than the SHG 1 and SHG 2 mean errors. The large hole example performed worse, being above 50th for all fields with the error being just 1 order of magnitude below the SFG mean and being about 3 times larger than the SHG 1 and SHG 2 means. Nonetheless, the large hole example still captured the dynamics adequately, speaking to the strength of the model.

In the future, more advanced versions of this metric can be developed that can account for type and sampling rate of diagnostics used in the experiment that the simulation model is trying to mimic. In most cases, the lab diagnostic resolution is significantly less than what is achievable in simulation, so even the fight for perfection in simulation cannot always be fully confirmed by experiment.

Besides accuracy, we see substantial speed-up using the LSTM over traditional numerical methods, especially when a GPU is available. As discussed in manuscript, we achieve a 93% reduction in total simulation time for the photoinjector laser model, averaged over 1,000 instances. The portions of the simulation prior to the DCNS NLO section take approximately 0.13s out of the entire 1.98s simulation time. Running 1,000 instances in batch sizes of 200 for the LSTM replacing the DCNS section takes about 7.43s. The final reduction is given by

$$\frac{1.98 - \frac{(0.13 \times 1000 + 7.43)}{1000}}{1.98} \times 100 = 93.06\% \quad (1)$$

to yield the roughly 93% reduction in time. This speed-up is essential in cases where one wants to generate a large amount of data or rapidly explore a local parameter space for real-time control.

E Data Applicability and Ethics

To the best of the authors’ knowledge, our data does not pose any major ethical risk or negative societal impact. Our data was generated from our own simulation models and the intended use of the results is to speed-up and enhance future simulations so that resource usage in lab studies can be reduced (and instead can be explored in simulation first). Out of the listed societal impacts and potential harmful consequences from the NeurIPS ethics guidelines³, the environmental impact is the only relevant one for our work. We are generating data, storing data, and training models on the SLAC Shared Scientific Data Facility (S3DF). Server facilities, like this one, run 24-7 and thus, via energy consumption, have a negative impact on the environment. By requesting resources on S3DF, we contribute to this impact. While this sort of effect is unavoidable, it still must be addressed explicitly.

The framework and dataset we introduce is tailored in this paper to a specific application case, the LCLS photoinjector laser. The framework can be used to model many other systems, though. The LSTM we train is, thus, specifically addressing this same photoinjector laser because the generated data comes from the tailored version of the simulation framework. Furthermore, in this study, which we are using as a proof-of-concept to continue more in-depth studies, we only alter the pulse shaper control parameters for generating data. There are many other parameters that can be adjusted that will alter the entire parameter space and thus improve the robustness of the LSTM model. This is being addressed as an ethical consideration because we do not want to be making any claims this particular model we trained will be able to broadly replace all nonlinear optical simulations using the upconversion techniques we employ. Instead, we set the stage for training larger, more broadly applicable models that we are now beginning to explore (discussed in section 5 of the main manuscript).

References

- [1] Jack Hirschman, Randy Lemons, Minyang Wang, Peter Kroetz, and Sergio Carbajo. Design, tuning, and blackbox optimization of laser systems. *Optics Express*, 32(9):15610–15622, 2024.
- [2] Ursula Keller and R Paschotta. *Ultrafast Lasers*. Springer, 2021.

³<https://nips.cc/public/EthicsGuidelines>