Using Neural Networks to Sanitize Compton Camera Simulated Data through the BRIDE Pipeline for Improving Gamma Imaging in Proton Therapy on the ada Cluster

REU Site: Online Interdisciplinary Big Data Analytics in Science and Engineering

Michael O. Chen[1], Julian Hodge[2], Peter L. Jin[3], Ella Protz[4], Elizabeth Wong[5], Ruth Obe[6], Ehsan Shakeri[2], Mostafa Cham[7], Matthias K. Gobbert[2], Carlos A. Barajas[2], Zhuoran Jiang[8], Vijay R. Sharma[9], Lei Ren[9], Sina Mossahebi[9], Stephen W. Peterson[10], and Jerimy C. Polf[11]

[1]Departments of Mathematics, Dartmouth College, USA
[2]Department of Mathematics and Statistics, University of Maryland, Baltimore County, USA
[3]James M. Bennett High School, Salisbury, MD, USA
[4]Department of Mathematics and Sciences, Florida Atlantic University, USA
[5]Department of Mathematics, Brookdale Community College, USA
[6] Department of Computer Science, University of Houston—Clear Lake, USA
[7]Department of Information Systems, University of Maryland, Baltimore County, USA
[8]Department of Radiation Oncology, Stanford University, USA
[9]Department of Radiation Oncology, University of Maryland School of Medicine, USA
[10]Department of Physics, University of Cape Town, South Africa
[11]M3D, Inc., USA

Technical Report HPCF–2024–5, hpcf.umbc.edu > Publications

Abstract

Precision medicine in cancer treatment increasingly relies on advanced radiotherapies, such as proton beam radiotherapy, to enhance efficacy of the treatment. When the proton beam in this treatment interacts with patient matter, the excited nuclei may emit prompt gamma ray interactions that can be captured by a Compton camera. The image reconstruction from this captured data faces the issue of mischaracterizing the sequences of incoming scattering events, leading to excessive background noise. To address this problem, several machine learning models such as Feedfoward Neural Networks (FNN) and Recurrent Neural Networks (RNN) were developed in PyTorch to properly characterize the scattering sequences on simulated datasets, including newly-created patient medium data, which were generated by using a pipeline comprised of the GEANT4 and Monte-Carlo Detector Effects (MCDE) softwares. These models were implemented using the novel 'Big-data REU Integrated Development and Experimentation' (BRIDE) platform, a modular pipeline that streamlines preprocessing, feature engineering, and model development and evaluation on parallelized GPU processors. Hyperparameter studies were done on the novel patient data as well as on water phantom datasets used during previous research. Patient data was more difficult than water phantom data to classify for both FNN and RNN models. FNN models had higher accuracy on patient medium data but lower accuracy on water phantom data when compared to RNN models. Previous results on several different datasets were reproduced on BRIDE and multiple new models achieved greater performance than in previous research.

Key words. Proton beam therapy, Compton camera, Classification, Recurrent neural network, PyTorch, Distributed Data Parallelism, ada

1 Introduction

In precision medicine, the medical records of a patient are evaluated as a tool for health care providers to deliver personalized treatment to meet individual needs of a patient. Having emerged as a common personalized treatment plan for cancer, radiotherapy involves delivery of a clinically determined dose of X-ray, electron, or proton radiation for the targeted destruction of a tumor. X-ray radiotherapy involves a high initial dose to the tumor target, which often results in side effects that impact adjacent normal tissue. Off-target effects could result in DNA damage, which may increase cell-aging termed senescence or undesirable symptoms such as inflammation, nausea, and vomiting [7].

A safety margin aims to deliver maximum exposure of treatment to all regions of the tumor but may include healthy tissue. Unlike the traditional radiotherapy options, proton beam therapy has shown promise in sharply reducing off-target effects of radiation due to the high energy proton beam ending at Bragg peak [17, 27]. Determination of the proton range for the Bragg peak of the prompt gamma radiation is needed for dose determination. In application, this has the potential to translate into real time images that will help clinicians avoid damaging healthy tissue. Uncertainty emerges due to the Compton camera being unable to identify the sequence of scattering interactions that result from the prompt gamma radiation [5]. To optimize the safety margin, deep learning has been used to address the background noise in the images reconstructed by the Compton camera.

Refs. [1], [12], and [25], each offer a unique contribution to the application of deep learning to the prompt gamma imaging problem. In [1], extensive hyperparameter studies demonstrate the efficacy of deep fully connected networks (FCN) in this application. [12] finds that recurrent neural networks (RNNs) offer a much faster and more compact solution, even if slightly less accurate. And [25] leveraged multi-process computation to dramatically increase the speed at which models are trained and deployed.

However, all 3 studies use the same dataset derived from Monte Carlo simulations of a *water phantom* (WP), a constant density water medium, in their training and testing of models. Recent improvements have allowed for *real patient* (RP) simulations to be generated, which can be processed into entirely new datasets to train and test on, as in Section 4.2. The value of using simulated data more consistent with reality is clear. It is not a given, though, that the results in [1], [12], and [25] will instantly transfer to this new data.

Hence, this work aims to test the extensibility of these previous results to the new RP data. In particular, we first improve the implementation of [25]'s multi-process training so rigorous tests on many datasets can be performed. Next, we refine the best models of [1] and [12] on WP data in our multi-process environments with improvements in architecture and model tuning. Finally, we perform an original hyperparameter study on the new RP data for both FCN and RNN models, and compare our results with WP data results.

The remainder of this report is organized as follows: Section 2 covers the background information on proton beam radiotherapy, Compton camera imaging, scatter types, and the need for machine learning models. Section 3 starts with technical background on the machine learning used in this work. It then does a deeper review of related works, focusing on the three aforementioned studies. Finally, it describes the physical and software resources used to execute the machine learning research. Section 4 covers how data was generated through GEANT4 and Monte-Carlo Detector Effects, and explains the significance of the new RP data. It then discusses preprocessing, our datasets, and feature engineering. Section 5 describes how we refined the multi-process implementation of [25]. It discusses key challenges of parallelizing training, and goes into detail about our solution, `BRIDE`. Section 6 covers the results of our tests on both WP and RP data, and provides a comparison of the two. Section 7 summarizes additional studies done in this research including

hybrid datasets and feature engineering. Section 8 concludes our findings, challenges encountered during our research, and potential clinical applications.

2 Prompt Gamma Imaging Background

2.1 Proton Beam Radiotherapy

Radiation therapy is one of the most common cancer treatments. X-ray therapy is a type of radiotherapy that delivers a high dosage of radiation to eradicate cancerous cells. However, the radiation dosage that is delivered to the tumor is often insufficient because the full dosage of radiation is delivered upon entering the body. Furthermore, X-rays will continue to travel posterior into the human body, thus causing excessive radiation exposure.

A relatively novel radiotherapy technique that can avoid the prior stated problems is proton beam therapy. Unlike X-ray therapy, proton beams in this type of radiotherapy deposit the vast majority of the radiation at the tumor site. Furthermore, the proton beam does not travel further than the tumor site, thus minimizing radiation exposure to healthy tissue. Hence, this treatment is widely considered to be more effective for some types of cancer. In proton beam therapy, ionizing radiation travels towards the target tissue; the energy level peaks at a location called the Bragg peak, and then sharply declines. Importantly, utilizing the Bragg peak allows for treatment plans that deliver radiation precisely to the tumor and avoid surrounding healthy tissue. However, the small distance between healthy and tumor tissue requires the Bragg peak to be accurately located. Clinicians need real-time information to determine the location of the Bragg peak so that it can be used to target the tumor as shown in Figure 2.1 [17].

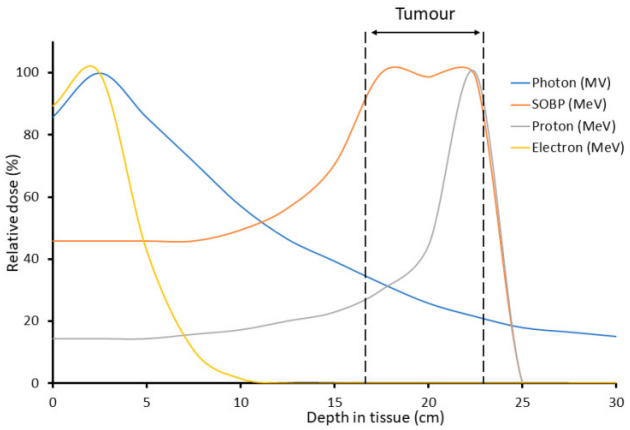


Figure 2.1: Spread-out Bragg peak (SOBP) [17]

During treatment, clinicians will create a safety margin, which is an enlarged treatment area which ensures that the entire tumor will receive the dosage of radiation. The safety margin will also account for any movement the patient may undergo during the treatment time, as well as the difference of positioning that the patient may have between treatment sessions. In Figure 2.2, both (a) and (b) represent two localized safety margins for treating a lung tumor, which minimize damage to the heart but may impact healthy lung tissue (b). At this point, clinicians would opt to safely proceed with the treatment with fewer proton beams and less damage to healthy tissue, resulting in a preference for (a), a single beam which avoids damage to healthy lung tissue over (b), two proton beams that unnecessarily damages the healthy lung tissue [31].

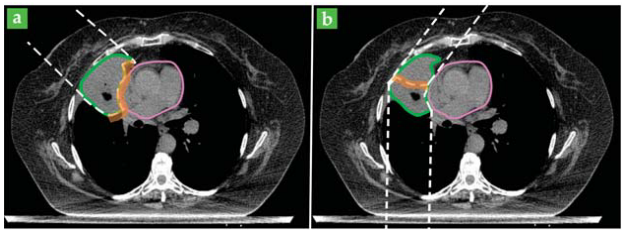


Figure 2.2: **a:** Optimal proton treatment beam, (dashed) targeting a tumor (green) with safety margin (orange) that overlaps with the healthy heart tissue (magenta). **b**: Suboptimal treatment plan of two beams which does not overlap with heart [31]

.

With real-time information about the trajectory of the proton beam, clinicians can then provide improved treatment to the patient. The safety margin could be smaller and the optimal path of Figure 2.2 (a) could be used. One of the proposed methods of real-time data acquisition is with the use of a detector called the Compton camera. When high energy proton beams collide with atomic nuclei and electrons in patient matter, radiation in the form of prompt gamma rays is emmitted; Compton cameras have the ability capture these prompt gamma rays [31]. This then leads to information about the path of the proton beams and the Bragg peak.

2.2 Compton Camera and Image Reconstruction

For the monitoring of proton radiotherapy treatment, prompt gamma radiation can be recorded using a Compton camera, a multistage detector that produces data to visualize prompt gamma radiation and scattering events. Prompt gamma radiation is emitted at a specific angle of displacement, which is determined from the energy levels of the proton collision with the nucleus. Prompt gamma rays interact with the camera; for each interaction, the camera will calculate (x_i, y_i, z_i) coordinates and the energy level e_i of the scatter. The Compton cone of emission is used to project the potential trajectories that this collision could occur as shown in 2.3 [28]. Using this cone of emission, the origin of the gamma is mathematically determined [30].

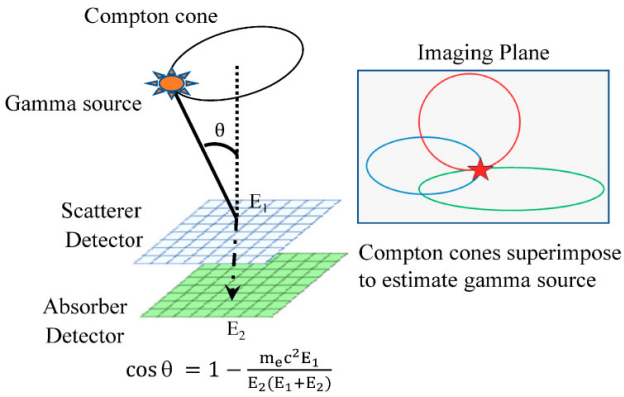


Figure 2.3: Compton cone of emission [28]

After the collection of the scattering events, image reconstruction algorithms recover a visualization of the path of the proton beam. However, the Compton camera has a major problem; due to its nature, the Compton camera has a finite time resolution. It does not explicitly record the sequential order of the prompt gamma rays, resulting in the camera being unable to detect the true ordering

of interactions. This causes noise in the reconstructed images, rendering them sometimes unusable in a medical setting [30].

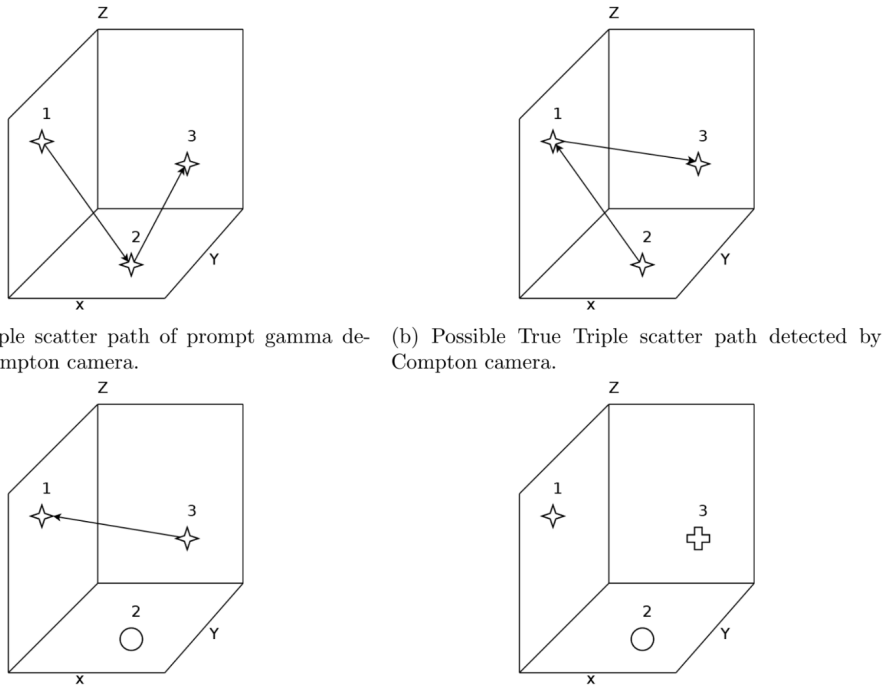
2.2.1 Scatter Types

Due to prompt gamma radiation emission at approximately the speed of light, the sequence of the scattering events becomes distorted, creating background noise that plagues image reconstruction. To help identify false events that creates noise within the image, scattering events are organized as scatter types. There are 13 types of scatterings, of which can be grouped into true triples, doubles to triples (DtoT), and false triples as displayed in Figure 2.4.

True Triples: True triples are three sequential interactions with the Compton camera. The ordering of the interactions can be one of six combinations: 123, 132, 213, 231, 312, and 321. Out of these, only the 123 combination is currently usable for image reconstruction purposes.

Doubles to Triples (DtoT): DtoT events are double and single interactions that occur independently of each other but are detected as one event by the Compton camera. There are six possible combinations of this event: 124, 134, 214, 234, 324, and 314, where the "4" refers to the second prompt gamma interaction in the misdetection events.

False Triples: False triples are events that are detected as a true triple, but in reality are comprised of three independent events. These false events may result in images with noise and must be discarded [5, 21, 30].



(a) True Triple scatter path of prompt gamma detected by Compton camera.

(b) Possible True Triple scatter path detected by Compton camera.

(c) Possible Double-to-Triple path of prompt gamma detected by Compton camera.

(d) False triple path of prompt gamma detected by Compton camera.

Figure 2.4: Scatter events [12]

2.2.2 The Need for Machine Learning

In order to make real-time proton radiotherapy more effective, real time imaging is required for treatment in order to verify the dosage of the proton beam and location of the Bragg Peak. There have been multiple different classical methods developed that attempt to "de-noise" or sanitize the reconstructed images. Applying a specific filter to the image to reduce blurring and artifacts is one such common technique; in [23], distance between each image pixel's center and the Compton cone was calculated, and pixel values and thresholds were then set to the respective distances, "cleaning" the image.

Machine learning in the field of computer vision, however, has the ability and potential to capture and exploit much deeper patterns than a filter. Machine learning can classify the different scatter events based on data from the Compton camera. The classified false events can then be removed, creating a sanitized image to be used for proton beam therapy treatment verification.

3 Machine Learning: A Solution

3.1 Technical Background on Machine Learning

Machine learning, a field of study in artificial intelligence, uses algorithms to analyze, identify, and generalize from specific trends and patterns within data. The main form of machine learning employed in this work are **Neural Networks** (NN). NNs have experienced an explosion of usage in multiple domains of science due to their ability to 'learn' to identify underlying trends and patterns in data and then extrapolate from data of the same form to perform sophisticated predictions. The specific type of machine learning used in this work is known as supervised learning, in which the NN is 'trained' on data that has a certain label of interest known as the **class**. It is then designed to predict this label through training, in which it may discover complex relationships between each observation of data and its label.

Neural networks were originally designed to emulate the ability of the human brains' neurons connections to memorize, recall, and adapt to novel information. They are composed of layers of neurons, which each take in a series of *weighted* inputs from training data and output a newly weighted transformation of input data, that is composed with a nonlinear *activation function* into the next layer of neurons. Thus, the layers of neurons learn from the training data by progressively updating the weights of the matrices that scale and shift the activation function output from each neuron in a layer unto the next neuron; this then continues into the next layer of neurons.

The composition of many such matrices that scale and shift the transformations of input data according to activation functions allow for the approximation of any continuous function which may exist as a relationship between a particular **feature** of the data and its output (in our case a **class** of particular scatter type). This idea is known as the Universal Approximation Theorem [14]. Though, it is not guaranteed that a particular algorithm can find the exact weights and sequence of necessary functional compositions that allow for a perfect deterministic relationship between input and output data. Even so, the **optimizer** algorithm and suitable **loss function** are designed to attempt to approximate this theoretical relationship to a numerical precision.

The two main types of neural networks applied in this work are Feed Forward neural networks and Recurrent neural networks, as they have shown some success in past work [34].

3.1.1 Feed Forward Neural Networks

The simplest deep learning model is the Feed Forward neural network (FNN). FNNs involve unidirectional flow of information, channeling the input through the hidden activation layer(s) to become output [33], as shown in Figure 3.1. Each neuron in one layer is connected to every neuron in the next layer, forming a fully-connected layer; in other words, the output of a neuron in a layer serves as the inputs for all of the neurons on the subsequent layer. These models are widely used for many machine learning applications, including search engines, image classification, economic forecasting, and language translation.

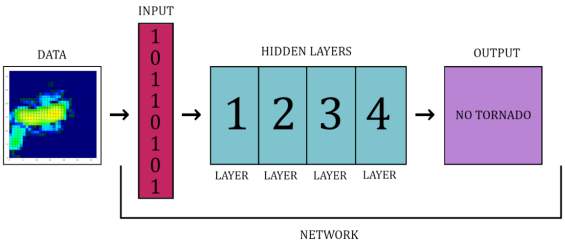


Figure 3.1: An example structure of a Feed Forward Neural Network [1]

3.1.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are multi-directional networks with recurrent units and equal weights in each layer. Each recurrent unit has forward activation units with a memory state needed to store information about the network at particular epoch and a backward propagation for training the network. While remembering the last input, the memory state is updated continuously with new information [34]. Some drawbacks to this type of model include that they are very sensitive to hyperparameter changes and are prone to exploding gradient issues, among others. A variety of models, including FNNs and RNNs, were tested on the simulated data [33]. Building upon Clark et al. research that has shown 74.6% accuracy using a LSTM with 256 fully connected layers, a plethora of models were implemented such as this RNN in hopes of achieving the clinical standards of 90% or higher validation accuracy [12]. This will be addressed in Section 6.1.

Long-Short Term Memory (LSTM) Neural Network A type of RNN is the Long Short Term Memory (LSTM) model, which features long-term dependencies. In addition to the defaults in RNNs, information of an LSTM can be sent to the input gate for memory, discarded in the forget gate, or produced as output from memory as shown in Figure 3.2 [34]. These three gates are part of a unique aspect of an LSTM model called a memory cell; this memory cell stores information that may be needed later in the model training process. Though LSTMs are typically used for natural language processing and time series forecasting, the model was thought to be able to perform well on our datasets due to its robust learning capabilities. The implementation of LSTM will be discussed in Section 6.2.2.

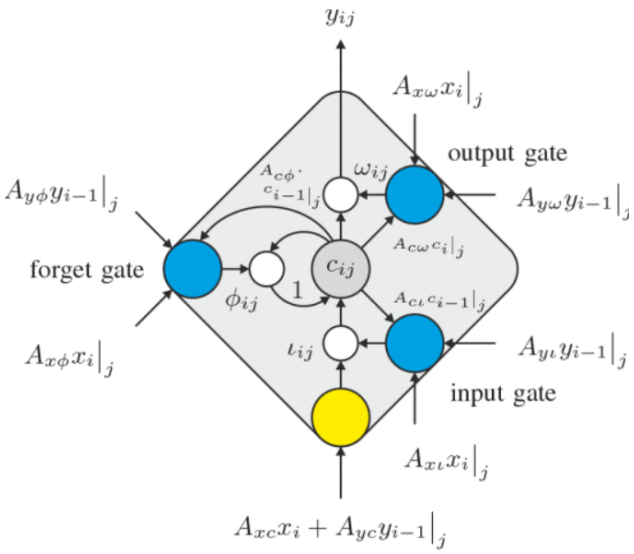


Figure 3.2: LSTM gates [34]

Gated Recurrent Neural Network As shown in Figure 3.3, the gated recurrent neural network (GRU), a type of RNN, factor in short-term dependencies with a reset gate by using a linear interpolation between the previous activation function value and the current one. GRUs are regulated by two gates: an update gate, which selects candidate hidden state activation function and continuously updates the network, and a reset gate, which sequesters unnecessary input that will not be used to update the hidden state [34]. A GRU is essentially a streamlined version of the LSTM model; however, unlike the LSTM, it does not contain separate memory cells. The implementation of the GRU will be discussed in Section 6.1.

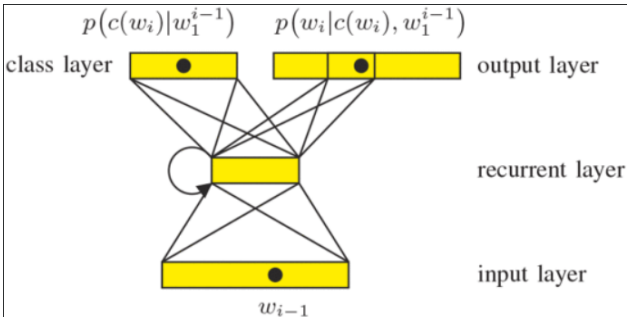


Figure 3.3: GRU diagram [34]

3.2 Related Works

In recent years, significant progress has been made in the application of machine learning to image reconstruction from prompt gamma radiation. Three relevant works are the research in [1], [12], and [25].

[1] focuses on optimizing deep fully connected neural networks (FCNs) for image reconstruction. The authors demonstrated that FCNs have a reasonable ability to classify events across various classes. They showed accuracy's that were at best 75%, which is not desirable for clinical use. In order the achieve this result, there was a comprehensive hyperparameter study involving 288 experiments. These studies identified promising configurations that could lead to better performance

in future application. There is also a strong emphasis on developing more compact neural networks that can achieve similar or superior performance while utilizing fewer parameters. RNNs were briefly tested. It is important to note that [1] utilized a *residual* architecture for their FCNs. This means they were able to train very deep networks with hundreds of layers without running into exploding gradients or other such problems. The best FCNs *without* residuals, which [3] explored, achieved a peak accuracy of 63.9%.

The research in [12] had a primary contribution related to RNNs. While deep residual FCNs achieved slightly higher accuracy, RNNs demonstrated comparable performance with a key advantage: they have a simpler architecture and fewer hidden layers. This allows for faster loading times and enhances usability. This also makes it more efficient for real time applications in a clinical setting. They completed a comprehensive hyperparameter study indicating that hybrid models, one specific example being a model with 4 LSTM layers and 2 FCN layers, could have load times as fast as 7 seconds but still achieve 73% accuracy. The team also implemented a learning rate scheduler which improved model accuracy and efficiency.

[25] builds upon the past 2 works and used distributed learning via PyTorch to improve training times. The peak accuracy of their models were notably lower than past years, which is understandable considering they started on a new code base. Their highest LSTM model was 4-layer model that reached 66.5% accuracy.

Finally, importantly, all three of the aforementioned studies were done using essentially the same simulated data originating from a water phantom. [25] and [12] used a slightly different preprocessing routine, leading to the the datasets they trained on being approximately 1.4 million records, while [1]'s data was approximately 1.8 million records long.

Hence, to summarize the key results from experiments using water phantom data:

- FCNs with residual blocks achieve 75% accuracy
- FCNs without residual blocks achieve 64% accuracy (3.1)
- RNNs achieve 73% accuracy but with much simpler model architecture

3.3 Hardware and Software

For this research, we utilized the Graphics Processing Units (GPU) in the ada cluster maintained by the UMBC High Performance Computing Facility (hpcf.umbc.edu). The ada GPU cluster consists of four nodes with eight 2018 Ti GPUs each, seven nodes with eight RTX 6000 GPUs each, and two nodes with eight RTX 6000 GPUs and an additional 384 GB of memory each.

The machine learning models were built and implemented using PyTorch v2.3.1 (https://PyTorch.org). For data preprocessing and manipulation, we used scikit-learn v1.3.0 (https://scikit-learn.org/stable/), pandas v2.2.2 (https://pandas.pydata.org/), and numpy v1.26.4 (https://numpy.org/). To visually display our results, we used matplotlib v3.8.4 (https://matplotlib.org/) and seaborn v0.13.2 (https://seaborn.pydata.org/). Our models were built inside of the python environment Anaconda3 (https://www.anaconda.com/).

3.3.1 Parallelization

Parallelization is widely used in various computing aspects, where multiple nodes work on distinct aspects of a problem simultaneously. Among several of the benefits of parallelization include speed and efficiency: a larger task is broken down into smaller tasks that are processed at the same time,

leading in an optimal situation to an approximately linear increase in speed. With parallelization, greater tasks can be tackled while keeping runtime to a minimum, an important advantage in its many usage areas including data processing, scientific computing, and specifically, machine learning and artificial intelligence. However, though, there are some limitations in implementing parallelization; parallel processing codes are more complex, leading to a higher bar for programmers and more errors, as is later discussed. This complexity may not required for more simple tasks where a serial system may be sufficient. In short, parallelization and parallel computing can be broken down into task parallelism and data parallelism. Task parallelism works by distributing tasks between processors while using the same data, where the various tasks run simultaneously. Data parallelism, however, is distinct from task parallelism. It distributes data between processors, rather than using the same data.

Massively parallel processing is a parallelization paradigm where a large number of processors perform coordinated computations at the same time. Using multiple GPUs is a massively parallel architecture, as they contain tens of thousands of threads. In modern machine learning, distributing data batches among several GPUs and training a singular machine learning model based on all of the devices is a popular way to increase the speed and efficiency of model implementation. Training a model over multiple GPUs is much faster compared to on a single GPU, as a clear benefit of parallelizing computations.

4 Real Patient Data: A New Challenge

Recently, new simulated data originating from patient tissue rather than water phantom has become available. A key contribution of this work is gaining insights on this new data. First, it is necessary to present background on the Process of how simulated data is created and transformed into ready-to-train data.

4.1 The Preprocessing Pipeline

Preprocessing, in machine learning, generally refers to the aspect of preparing input data and target classes to be fed into an specific machine learning algorithm [18]. As mentioned, [1], [12], and [3] all utilized the same preprocessing pipeline which can be divided into two broad steps: Data Generation and Further Preprocessing.

4.1.1 Data Generation

Due to the measurement limitations encountered during clinical proton beam delivery, the state-of-the-art GEANT4 toolkit is used to perform high fidelity simulations of the interactions of the protons in a proton beam with the target matter, which produces prompt gamma ray data and their energy deposition in a pre-clinical Compton camera, in lieu of direct measurement. After prompt gamma interactions with the Compton camera are simulated with GEANT4, the resulting data is then fed into the Monte-Carlo Detector Effects (MCDE) modeling package which employs probabilistic (Monte-Carlo) models of the prompt gamma rays, alongside known signal processing timing effects in the Compton camera. This is to help determine the most likely type of scattering that took place and the most likely order for the scattering (to then assign class labels for the machine learning model).

GEANT4 A virtual experimental setup is designed by incorporating CT images from Duke University into the GEANT4 simulation toolkit. As per design, a Compton camera detector placed below the patient bed captures prompt gamma events from the patient [30]. Data generation has been carried out at different energies and locations, referred to as layers and spots, respectively. For instance, layer 1 denotes a proton beam energy of 198.7 MeV, consisting of 16 spots. The data is stored in a tree named "scatterData", with 11 tuples packed in ROOT format. Furthermore, the prompt gamma events captured by the Compton camera are organized based on their energy deposition, distinguishing events that originate from positron annihilation (511 keV), and events from singles, doubles and triples.

Each spot generates two types of data files: Prompt Gamma Emissions and 511 keV Annihilation Gammas. Prompt Gamma Emissions result from the proton interactions with patient anatomy, while 511 keV Annihilation Gammas are produced when positrons annihilate with electrons, emitting gamma rays at 511 keV. The distribution of spots across items is as follows: Item 1 has 16 spots, Item 3 has 67 spots, Item 5 has 87 spots, Item 7 has 153 spots, Item 9 has 168 spots, Item 11 has 174 spots, Item 13 has 180 spots, Item 15 has 185 spots, Item 17 has 159 spots, Item 19 has 104 spots, and Item 21 has 13 spots.

To automate the submission of slurm jobs on the taki cluster for this data generation, we developed a Bash script named `automation_Sbatch_Final.sh`. This script processes up to 30 input files in one run, significantly streamlining the workflow. It reads from an input file, 'inputspots.txt', which lists the slurm job files to be submitted. The script logs progress and completion statuses in

'progress.log' and 'completed_jobs.log', respectively. This dual-logging mechanism ensures that we can track the progress of each job and identify any issues that may arise during execution.

The core functionality of the script is encapsulated in the `submit_job_with_retry` function; this is responsible for submitting jobs and is designed to be resilient against transient cluster issues. If a job submission fails due to the temporary unavailability of resources, the function waits for a specified period of time before retrying, up to 40 times. This "retry" mechanism is crucial for maintaining the continuity of the data generation process, as it ensures that temporary disruptions do not halt the workflow.

Logging and status tracking are integral to the scripts operation. Each submission attempt and its outcome are logged, providing a detailed record of the scripts' activities. Successful submissions are recorded in the 'completed_jobs.log' file to prevent reprocessing, ensuring that each job is only processed once. This feature not only prevents redundant processing but also allows for easy status checks through log files such as 'keepeye.log' and 'progress.log'.

To ensure uninterrupted execution, the script can be run in the background using the `nohup` command. This capability is crucial for long-running processes on shared computing resources as it allows the script to continue running even if the user disconnects from the terminal. This background execution is particularly important for high-performance computing environments, where long-running jobs are common.

By automating the submission of slurm jobs and incorporating robust retry mechanisms, the script effectively manages the generation of the large datasets needed for the training of machine learning models. Automation reduces manual intervention and optimizes resource usage on the HPCF cluster, facilitating efficient and reliable data processing. This detailed implementation and its role in our data generation pipeline underscore the importance of automation and error handling in high-performance computing environments, particularly for machine learning projects that require extensive computational resources.

Monte-Carlo Detector Effects As mentioned above, prompt gamma interactions in the Compton camera as simulated by GEANT4 are fed to the Monte-Carlo Detector Effects (MCDE) model. The interaction data is then post-processed to include the intrinsic detector and electronics readout properties of the prototype Compton camera. It has been demonstrated that the MCDE model provides a reasonable simulation of actual prompt gamma interactions and readout of the detector, making the simulated data a valid representation of the data recorded during clinical proton beam treatment [21]. In addition, one key advantage of utilizing simulated data is that we are able to generate large volumes of raw data using this GEANT4-MCDE pipeline. This has allowed us to create entirely new datasets this year, in addition to having access to past years' training data.

The final MCDE output are data files that includes single, double, and triple scatter events as they would have been captured by the Compton camera under the beam delivery conditions of the modeled experiment, such as irradiation field size, irradiation time, and beam intensity.

MCDE generated triples scatter data has 12 columns of data: 3 sets of energy and spatial coordinates data (e_i, x_i, y_i, z_i). The most important impact of MCDE modeling in our data generation is that it employs a model of the Compton scattering and signal processing timing effects to determine the most likely scatter ordering; therefore, our class labels are originally produced in this stage of data generation. Thus, the MCDE data is treated as our raw data for propagation through the preprocessing pipeline. This pipeline is due to the nature of the MCDE data as we cannot simply use it to train a machine learning model. First, every MCDE run does not generate the same amount of data; second, the proportion of true triple, DtoT, and false events vary at different dosages, resulting in imbalanced data which may not be appropriate for model training. In particular, a

higher dose rate means that the radiation dose is delivered more quickly, which can have different biological effects compared to a lower dose rate where the same total dose is delivered over a longer period.

4.1.2 Further Preprocessing

In this research, we focus only on the triple events. As mentioned, the raw MCDE scatter data has 12 columns, which are 3 sets of individual prompt-gamma ray detections of energy and 3D spatial coordinates (e_i, x_i, y_i, z_i). Each row of the data corresponds to one triple chain scattering interaction. We first unite each row of data with its proper class label as given by MCDE simulation. This gives a scattering label to each triples interaction that reveals its proper ordering in the case that it is a true triples event. With the DtoT event. the class label corresponds to the proper ordering of the double and the event that is a false detection. Lastly, in the case of a false triples event, the label shows that the scatter is actually a series of decoupled single events as opposed to one chain reaction. Overall, the MCDE data unified with the class label gives the initial data format as shown in Fig. 4.1.

Class	Interaction 1				Interaction 2				Interaction 3			
123	e_1	x_1	y_1	z_1	e_2	x_2	y_2	z_2	e_3	x_3	y_3	z_3
132	e_1	x_1	y_1	z_1	e_3	x_3	y_3	z_3	e_2	x_2	y_2	z_2
213	e_2	x_2	y_2	z_2	e_1	x_1	y_1	z_1	e_3	x_3	y_3	z_3
231	e_2	x_2	y_2	z_2	e_3	x_3	y_3	z_3	e_1	x_1	y_1	z_1
312	e_3	x_3	y_3	z_3	e_1	x_1	y_1	z_1	e_2	x_2	y_2	z_2
321	e_3	x_3	y_3	z_3	e_2	x_2	y_2	z_2	e_1	x_1	y_1	z_1
124	e_1	x_1	y_1	z_1	e_2	x_2	y_2	z_2	single			
214	e_2	x_2	y_2	z_2	e_1	x_1	y_1	z_1	single			
134	e_1	x_1	y_1	z_1	single				e_2	x_2	y_2	z_2
314	e_2	x_2	y_2	z_2	single				e_1	x_1	y_1	z_1
234	single				e_1	x_1	y_1	z_1	e_2	x_2	y_2	z_2
324	single				e_2	x_2	y_2	z_2	e_1	x_1	y_1	z_1
444	single				single				single			

Figure 4.1: Merged MCDE Data Format Interaction data creates 12 initial features while the class label is scatter type.

We then have to balance the dataset by classes in order to have viable training data, as most machine learning models generally perform poorly in a classification task with data whose classes are not properly balanced. The imbalance here is, for example, when there are more true triples than DtoT or no false event data. This imbalance of classes has primarily been observed to correlate with the simulated dosage rates. It has been seen that at lower dosage rates (less than 1kMU), the data is almost exclusively true triples, while at higher dosage rates, there is a much greater amount of doubles-triples and false events [4,5]. MCDE data was generated at multiple kMU dosage rates, including at 0kMU/min, the clinical minimum of 20kMU/min, 100kMU/min, and the clinical

maximum of 180kMU/min. For training purposes, we then combine a large volume of MCDE data at varying dosage rate with the same beam energy (150MeV), prune rows of the data so that we have an equal frequency of classes (according to the minimum occurring class), and then shuffle all the data, creating *merged, shuffled* data.

The last step in the preprocessing pipeline is to normalize the data as this presented numerical optimization advantages to the calculations done by neural networks. Neural networks generally perform best (converge more quickly) when the feature columns are normalized by scaling/centering the distribution of each feature, as well as removing outliers which may over-influence learned behavior. During normalization, each feature column was scaled/centered, outliers were removed, and the data was transformed to a normalized range (scaling to a mean of 0 and a unit standard deviation). To normalize the spatial variables i.e (x_i, y_i, z_i), this study used the MaxAbsScaler from the `sklearn` library; the PowerTransformer (Yeo-Johnson) was utilizted to standardize the energy deposition values [5].

The output of this 'Further Preprocessing' section of the pipeline is ready-to-train data.

4.2 Real Patient Data

The new "real patient" data is fed into the current preprocessing pipeline very early: instead of incorporating CT images constructed using a water phantom, CT images made using real patient tissue measurements are used. The 'water phantom' is the simulated experiment area that is filled with mass where the simulated protons are colliding. This simulated mass has the uniform density parameter of water. Contrarily, the real patient replacement has a simulated mass with variable density based on measurements of patient tissues. Moreover, the new post-MCDE data is thus divided into 'layers' which each correspond to different densities and energies, similar to the aspect that patient tissue varies [29].

Hence, the same basic architecture of preprocessing pipeline is used, but a change based on shifting from a water phantom to a real patient early in the pipeline creates a ready-to-train dataset that presumably has a substantially different underlying order.

4.2.1 Datasets

Below, we introduce the ready-to-train datasets that were utilized in this study. This, however, does come with an important and rather convoluted caveat: one of the key contributions of this work is producing an overhaul of the entire code base, which includes the preprocessing pipeline. Thus, the datasets below are not made using precisely the same pipeline that is described in the preceding sections[1]. However, the changes to the pipeline are *almost* entirely 'refactoring' changes, and so the datasets are nearly identical to the datasets that would have been made by the original pipeline.

1. `barajas`: this is data based off of a water phantom. It was originally created in [4] and was generated from preprocessing simulated experimental data of 20kMU, 100kMU, and 180kMU dosage rates with a 150MeV energy proton beam. It contains approximately 1.8 million observations.

2. `shakeri-obe`: this is data based off of real patients. This was generated using the process described in Subsection 4.2 with the parameters in Fig. 4.2[2]. It contains approximately 499,000 observations.

[1] It is different from both the pipeline before and after the real patient data changes

[2] It must be noted that during a few MCDE simulations for layers 3, 5, 7, and 9, the values 190, 192, 194, and 198 were mistakenly used for beam energy, which are not the same as the actual beam energies for these layers. The

3. `mothership`: this is a hybrid of the water phantom and real patient data. The dataset was constructed by combining all of the post-MCDE data that were preprocessed into the `barajas` and `shakeri-obe`, before *'Further Preprocessing'*. It contains approximately 3.8 million observations.

Layer	Beam Energy (MeV)	SimProtons	Dose Rate (kMU/min)	MU delivered
1	198	3.00E+08	20	5.00E+04
1	198	3.00E+08	20	5.00E+04
1	198	3.00E+08	180	5.00E+04
3	194	3.00E+08	20	5.00E+04
3	194	3.00E+08	20	5.00E+04
3	194	3.00E+08	60	5.00E+02
5	190	1.00E+10	100	5.00E+04
5	190	1.00E+10	100	5.00E+04
5	190	3.00E+09	60	5.00E+03
5	190	3.00E+08	20	5.00E+04
5	190	3.00E+09	140	5.00E+03
5	190	3.00E+09	100	5.00E+04
5	190	3.00E+10	40	5.00E+03
5	190	3.00E+09	140	5.00E+04
5	190	3.00E+10	20	5.00E+03
5	190	3.00E+10	60	5.00E+04
7	186	3.00E+09	100	5.00E+03
7	186	3.00E+10	100	5.00E+03
7	186	3.00E+10	100	5.00E+04
7	186	3.00E+09	20	5.00E+03
7	186	3.00E+09	140	5.00E+03
7	186	3.00E+09	140	5.00E+04
7	186	3.00E+10	40	5.00E+03
7	186	3.00E+09	40	5.00E+03
7	186	3.00E+09	40	5.00E+04
7	186	3.00E+10	60	5.00E+04
9	182	3.00E+08	20	5.00E+03
9	182	3.00E+08	60	5.00E+03
9	182	3.00E+08	20	5.00E+04
9	182	3.00E+09	40	5.00E+04

Figure 4.2: **MCDE Parameters**. The parameter layer represents the density of the tissues being irradiation by the gamma-ray. The beam energy (MeV) refers to the kinetic energy carried by protons when accelerated to high speeds. SimProtons represent the total number of simulated protons used in the simulation study. The dose rate refers to the amount of radiation dose delivered per unit of time. MU (Monitor Units) delivered are a measure used to quantify the amount of radiation delivered to a patient.

4.3 The Task at Hand

Given the new `shakeri-obe` dataset (based off of real patients), the task naturally arises to study it. We did the following:

simulation will be repeated with the actual beam energy values in our future works, and the table will be modified accordingly.

- We perform a dedicated hyperparameter study on both FCNs and RNNs with the real patient data to determine results analogous to "Equation" 3.1. This is Subsection 6.3.
- After attaining these results, we will compare them with those of previous works summarized in "Equation" (3.1). This is subsection 6.4.
- Finally, we found that the work in [25] in terms of its parallelized learning implementation had flaws, causing completion of the preceding 2 tasks impossible. Thus, we implemented an entirely new parallelized learning system, discussed in subsection 5.3.

We note at this point that we do not expect past results to transfer cleanly to the new data. Importantly, data based off of real patients is likely more complex than water phantom data; hence, models may have more trouble learning patterns in real patient data in general, and may not perform as well. Though, due to differences in underlying order, certain architectures may also be of advantage.

Though a water phantom has been used in some cases in research to simulate an actual patient, there is neither sufficient evidence in practice nor a mathematical proof that one can actually fully represent the other [2]. In a sense, two assumptions need to be taken to utilize a water phantom instead of a patient: a homogeneous water phantom fully represents a heterogeneous actual patient, and a water phantom can be "developed" into the shape of an actual patient. This is hence a key difference between using a water phantom versus real patient data.

5 Parallelized Learning

5.1 Challenges and Advantages of Parallelization

Distributed machine learning, as discussed in subsection 3.3, is one of the most exciting areas of development in machine learning today. The advantages of parallelizing computations in a machine learning context are many, but the most obvious one is the speed at which more complex models can be trained on increasingly large datasets (see [19]). For example, consider the design of the research in [1]: the hyperparameter study was necessarily split into two stages, the first with shorter baseline studies and the second with elongated studies because of the time and resources it took to perform runs for 4000 epochs. With our newly implemented parallelized system covered in subsection 5.3, we regularly completely studies with 8000 epochs with 4 GPUs, and could run jobs with 32 GPUs (given the node space). Though the resources constraint still exists, parallelized learning removes much of the time constraint that limits research, for example [1].

However, distributed learning comes with a host of challenges [35]. Many of these are related to low level implementation that a library such as PyTorch can wrap and "hide away" from the end user. [25] utilized PyTorch's Distributed Data Parallel (DDP) to train models across multiple GPUs. In fact, the ease of developing models over multiple processors in PyTorch is a significant benefit of the library over other deep learning libraries such as Tensorflow. The DDP process involves distributing the input data in unique subsets among the various GPUs; each processor then uses its given set of data to construct a model. With forward and backward passes, gradients among the GPUs are synchronized and averaged, and weights are updated on the entire model. Our parallelized learning implementation also uses DDP (albeit, one level up).

Unfortunately, challenges for the end user persist even with wrappers like DDP. In fact, the authors of [3] – citing a host of implementation problems including "configuration difficulties", "synchronization bottlenecks", and "complexities related to memory management and data distribution" – opted for single-GPU training, only setting up a working DDP system in [25]. The following subsection covers one such challenge in the form of an elusive bug, that, in fact, also plagued [25] unbeknownst to them; the discovery of this bug is a contribution of our work.

5.2 The Validation-Test Gap

We detected the 'Validation-Test Gap' immediately after implementing our new parallelized learning system, BRIDE, which is covered in subsection 5.3. In short, one of the key limitations of the code base we inherited from [25] was the lack of a *testing* set: there was only ever a validation set. Because no hyperparameter tuning was being performed, a validation set and test set should have been interchangeable, as both were simply holdout sets from the training data. (In other words, during hyperparameter tuning models may be selected based on validation set metrics, effectively invalidating their 'unseen' status, but in our case this does not apply.) For scalability and rigor, however, we implemented a test set.

But, what we noticed, among several models, was that the validation accuracy was very high (approximately 90%) on *both* BRIDE and the old code base. However, the testing accuracy on BRIDE was very low in comparison (around 60%). For instance, the same run with the accuracy learning curve in Figure 5.1 had the confusion matrix in Figure 5.2: validation accuracy is 88.2%, while testing accuracy is significantly lower at 62.6%. This difference in accuracy came to be known as the Validation-Test Gap. In our initial analysis of the bug, we accepted a 'dichotomy hypothesis': either the validation accuracy was correct and the testing accuracy was incorrect, or the testing accuracy was correct and the validation accuracy was incorrect (as any other situation was very unlikely). Importantly, our implementation of the validation-train split was completely

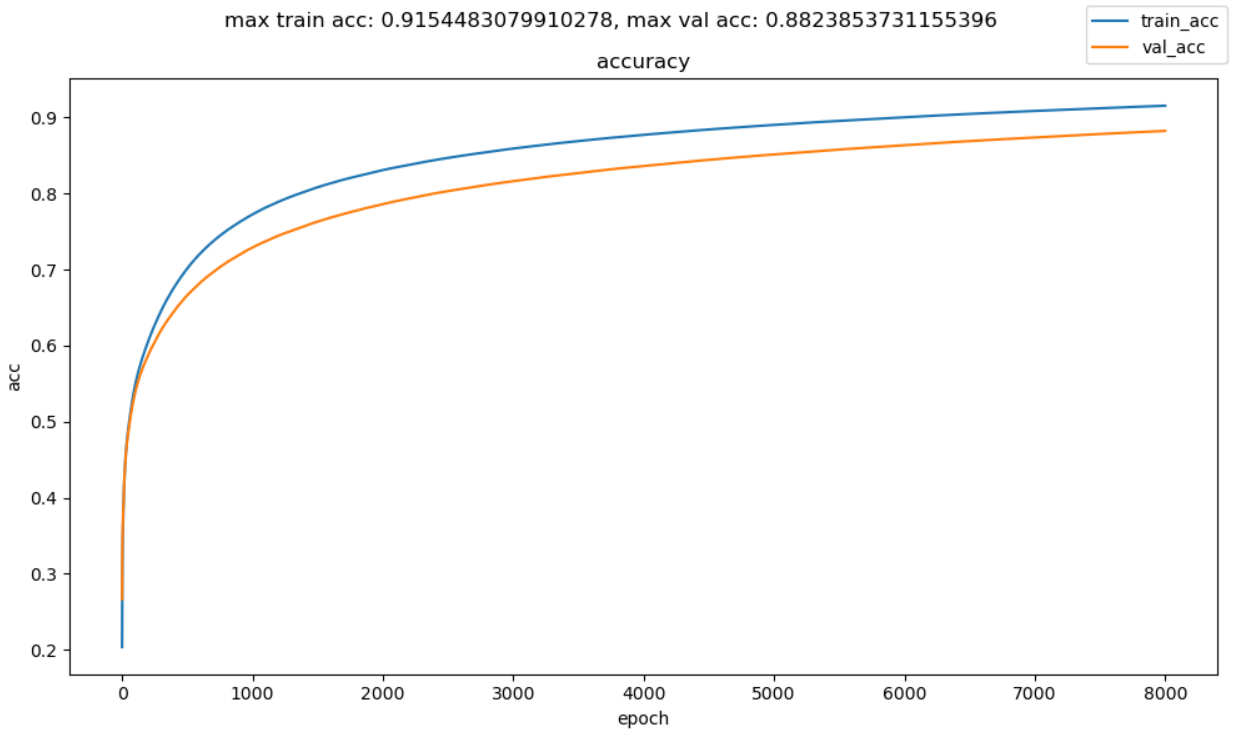


Figure 5.1: Accuracy training curve of a certain model. See Figure 5.2

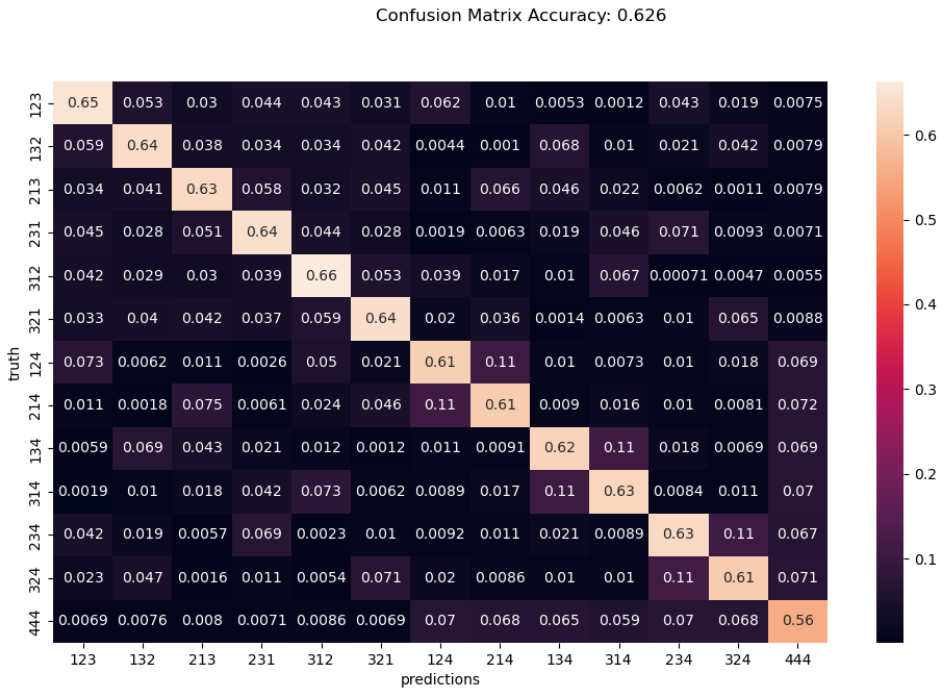


Figure 5.2: Confusion matrix of the model in Figure 5.1. There is a significant difference in accuracy between these two plots.

independent of the old code base's. Therefore, it seemed unlikely that the same bug would afflict two completely different implementations, which gave us false intuitions from the start of our experiments.

In the BRIDE platform, the train-test split happens in a file named `pp2.py` using sklearn's `train_test_split` method, while the train-validation split happens in a different file later in the pipeline called `utils.py` using PyTorch's `random_split` method. PyTorch Lightning, the high level wrapper we utilized, has a dedicated method to pass in the validation set and a another distinct method to pass in the test set. As previously stated, the two data splits taking place at different times and using different methods in the pipeline should not have greatly affected predictive performance, as both the validation and test sets were "held out". Our first experiments consisted of swapping these methods and testing different changes to these splits; for example, `pandas.DataFrame.reset_index` was called on the data after splitting (which also shuffles the data) [Experiment 1, variations 1-3]. These, however, led to no changes in behavior.

Our first productive experiments came when we had the two holdout sets 'swap places' [Experiment 2]. We simply passed the test set to the PyTorch Lightning method where the validation set was intended to go, and the validation set where the test set was intended to go. This resulted in a low validation accuracy and a high testing accuracy, as shown in Figures 5.3 and 5.4, which correspond to Figures 5.1 and 5.2. The results of these experiments suggested that the problem was not

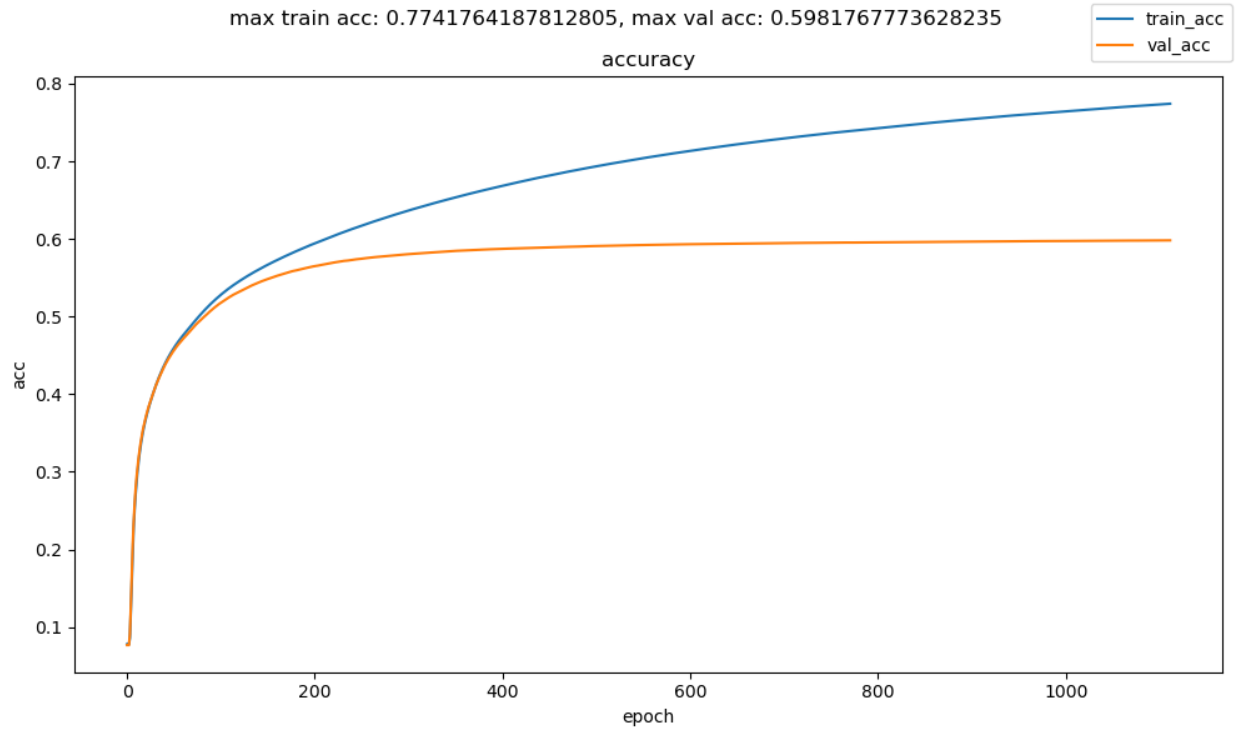


Figure 5.3: Accuracy training curve of a certain model. See Figure 5.4

with PyTorch Lightning. For example, we had theorized that Lightning may have been somehow leaking the validation data to the model, but the tests proved this to be false. This meant that the problem was with how the two data subsets were made.

Hence, we decided to cut the complexity of having two different files and did both splits using sklearn's `train_test_split` method within `utils.py`, in the same `setup` method [Experiment 3]. `setup` is a part of PyTorch Lightning's 'DataModule'; in Lightning documentation ([15]) it was

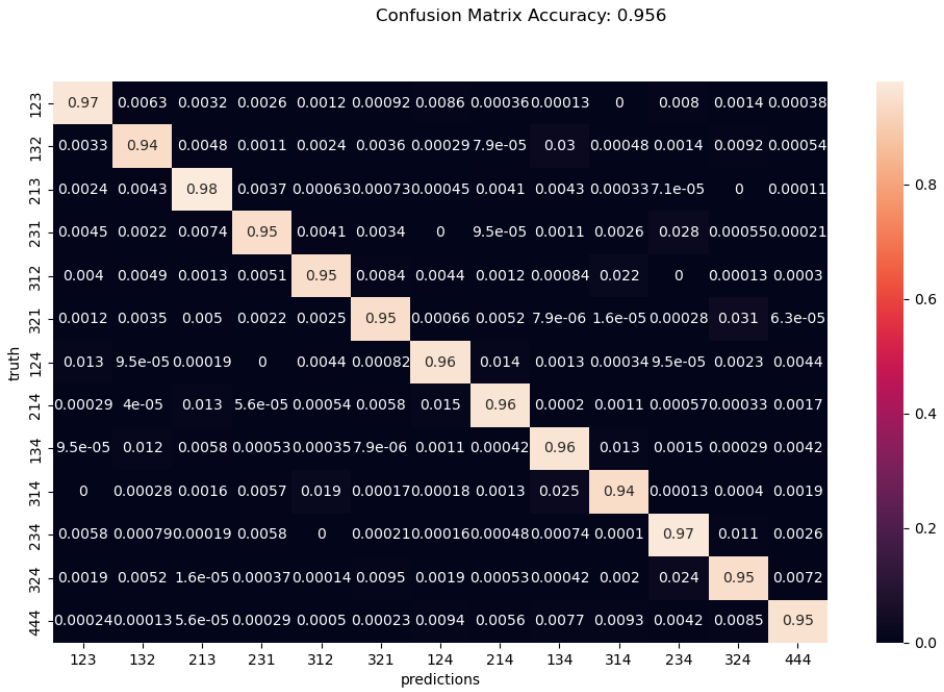


Figure 5.4: Confusion matrix representing 'testing' accuracy of the model in Figure 5.3. There is a 'reversed' (from the ones observed in Figures 5.1 and 5.2) significant discrepancy in accuracy between the two plots.

recommended to do train/test/validation splits in this method. Previously, only the train-validation split was happening in this method. But, this produced the same results as in Experiment 1: we performed the same operation in the same location to produce the test and validation splits, but a gap in accuracy still occurred.

After performing several more fruitless experiments, we finally performed the key experiment, Experiment 6, which was the 'converse' of Experiment 3: we implemented both dataset splits in `pp2.py` [Experiment 6]. The result was that validation accuracy decreased dramatically, as in Figure 5.3, and no change in behavior for the test accuracy was observed. This indicated to us that the issue may have been with the `setup` method. After doing deeper research into the `setup` method, we found that it was called *on every GPU*. This explained the seemingly lack of difference in the two sets in Experiment 3; testing is always done on one GPU, but training is performed using multiple.

We found a clear explanation for the validation-test gap: as `setup` was called once for every GPU, the (randomized) train-validation split was happening once on every GPU. This would cause an obvious 'data leak' as the model would gain exposure to (have its weights updated based upon and learn from) most of each of the validation sets, as the randomized split would cause some validation data to be part of the training data through the multiple GPUs; this hence would lead to a high validation accuracy since the model would have already trained on it. We had two immediate reservations with this explanation. First, how was the validation accuracy calculated when there were 4 different validation sets? Second, the PyTorch Lightning documentation specifically recommended that dataset splits using the `setup` method. For the first reservation, it was found that TorchMetrics ([24]), PyTorch Lightning's dedicated API to create custom metrics, wraps an averaging

calculation which does indeed take into account validation sets on different GPUs (TorchMetrics calls something similar to PyTorch's `torch.distributed.reduce` method which averages across GPUs). For the second reservation, we found that it is expected to set the seed of the random dataset splits in `setup`, which we did not do at first.

To confirm our explanation, we did two additional experiments. First, we ran two identical jobs, except one on 2 GPUs and the other on 1 GPU [Experiment 7]. The result was that the 1 GPU job had a validation accuracy similar to that in Figure 5.3, and the 2 GPU run produced a curve like that in Figure 5.1, which was consistent with our explanation. Finally, we tried setting the seed of the split in `setup` by placing an optional argument in the `random_split` method [Experiment 8]. This again produced a curve as the one in Figure 5.3 and a testing accuracy similar to Figure 5.4, where the training accuracy was high but both validation and testing accuracies were low. This final experiment was hence our "solution" to the Validation-Test Gap.

This work on the Validation-Test Gap thus constitutes both an important part of the research process but also a contribution to the scientific community. It demonstrates the difficulties of distributed learning and a specific complication, extending to a widely used machine learning library, that must be addressed when doing multiple-GPU training. Finally, it also explains a mystery in Ref. [25]. In Table IV, it can be seen that there is gap in validation and train accuracy disproportionately affecting the 1 GPU runs. The code base in [25] does not set a seed for the train-validation split; this gap can hence be explained by the data leak happening on their multiple-GPU runs, creating a falsely optimistic validation accuracy for those runs. The runs on 1 GPU, in a way, represent the "true" validation accuracy where data leak was not taking place. This also explains our initial observation that the bug was at furst in two independent code bases.

5.3 The BRIDE Platform

5.3.1 The Need for a New Development Platform

The need for a new development platform, encompassing both the scripts used to run tests and the theoretical structure underneath, became apparent from the code of [1], [12], and [3]. We have divided problems into 3 problem areas, each with multiple goals, which will be presented in descending order from high to low level.

Problem Area 1 (Discontinuity in the Big-Data REU Projects)

The Big-Data REU project's[3] nature can be described as a *discontinuous*, *collaborative* project. It is discontinuous in that substantive work is contained to 8-week periods each summer, and each period of work is completed by a unique team. It is collaborative because each of these separate teams are working towards the same goals. Any combined code platform to ensure progress must address these to aspects of the project.

Goal 1.1 (Minimum: reproducibility)

At a very minimum, a platform must allow for the efficient and exact reproduction of past years' results. This result should not be difficult to fill, hence its description as a 'minimum'. Any robust and well organized code base should achieve this objective.

Goal 1.2 (Expandability)

To effectively address the challenges posed by the discontinuity of the project, a platform must facilitate the subsequent years' teams building upon the work of previous years. The most simple version of this is Goal 1.1: copy-and-pasting past years' code. While this approach is not inherently incorrect, it is grossly inefficient compared to a truly expandable system.

From past years' work, there is a certain lack of continuity between teams. This, to an extent, is natural and necessary due to the new contributions each team is making. However, the project would benefit from the genuine reuse of code.

An example of this is the work in [3], in which there was a switch from Tensorflow to PyTorch. This constituted for an entirely new code base. However, a platform fulfilling Goal 1.2 would avoid this. Creating models is merely one part of the larger statistical learning workflow; ideally, the development platform would allow for the building upon of prior code, even if major updates such as a change of machine learning library are made (though this may not always be the case practically).

Problem Area 2 (Rigorous Testing and Flexible Experimentation)

From a design perspective, the central needs of the research tasks come together to form a set of guidelines for a statistical learning platform. Notably, as scientific research, the project necessitates meticulous record keeping and adherence to a rigorous set of machine learning principles. However, the researchers must also experiment with novel model architectures, which requires a degree of flexibility. This highly structured process on one hand prevents us from using any machine learning library "straight out of the box", which would otherwise be perfect for flexibility. On the other hand, the necessary flexibility of the project means we cannot "hard-code" a specific machine learning model.

[3]The single project [1], [12], [3], and this work all contributed to.

Goal 2.1 (Rigorous Testing)

The testing of a machine learning model involves evaluating performance and identifying an potential issues. The goal is to ensure that the model is reliable and "accurate".

1. Firstly, the code platform should enforce certain rigorous machine learning practices to ensure that testing is done correctly. Seemingly insignificant choices can become hugely damaging when data leaks happen, as in Subsection 5.2 in the form of a parallelization bug. There are also many other examples of this. For example, in subsection 5.2, it is mentioned that hyperparameter tuning based on a validation set effectively makes that data 'seen' by the model. Another example are data transformers: it is common to transform data during preprocessing using scalers. These scalers have to be fit to data; however, it is important that they are only fit to training data, and not testing data. Otherwise, a data leak would occur from this scaling transformation.

2. Second, data visualization and analysis can be key in gaining insight on different methods. An effective platform should wrap this functionality into one of convenience and high quality.

3. Third, documentation in the sense of keeping a record of all runs is vital; a second challenge is organizing records so that the most important ones are highly accessible. An automatic record-keeping system is not automatic in most platforms.

Goal 2.2 (Flexible Experimentation)

The structure of Goal 2.1 must be balanced with a certain flexibility to define and experiment with new models. This means "hard-coded" models such as those in the repository associated with Ref. [9] do not work. There are essentially two parts to this goal. First, the work of implementing a new model or preprocessing pipeline must be minimal; defining a new model should be as close to writing the code for just the model as possible. Second, a related but separate requirement that precedes the first is *modularity*. In order to be modular, a platform must separate at logical points the whole process of machine learning. This separation can also lead to minimal overhead, as a modularity will only mean changing one part of the code without affecting the rest.

Problem Area 3 (Readable Code)

This area less complex: low level code must follow correct practices.

Goal 3.1 (Readable Code) *As some code platforms can become extremely complex, code must be inherently "readable" in order to make changes in the future. This requires, as an example, an amount of comments directly in the code explaining each process, along with incorporating widely-used and understood coding practices. This also goes along with documentation of the research platform.*

5.3.2 Description of the BRIDE Platform

With the problem areas in Section 5.3.1 in consideration, this research designed and implemented the Big-Data REU Integrated Development and Experimentation (BRIDE) Platform. The BRIDE platform is coding workflow with an ontological structure; it can be described as having three "levels":

1. First, it is abstractly a set of guidelines defining a statistical learning workflow.

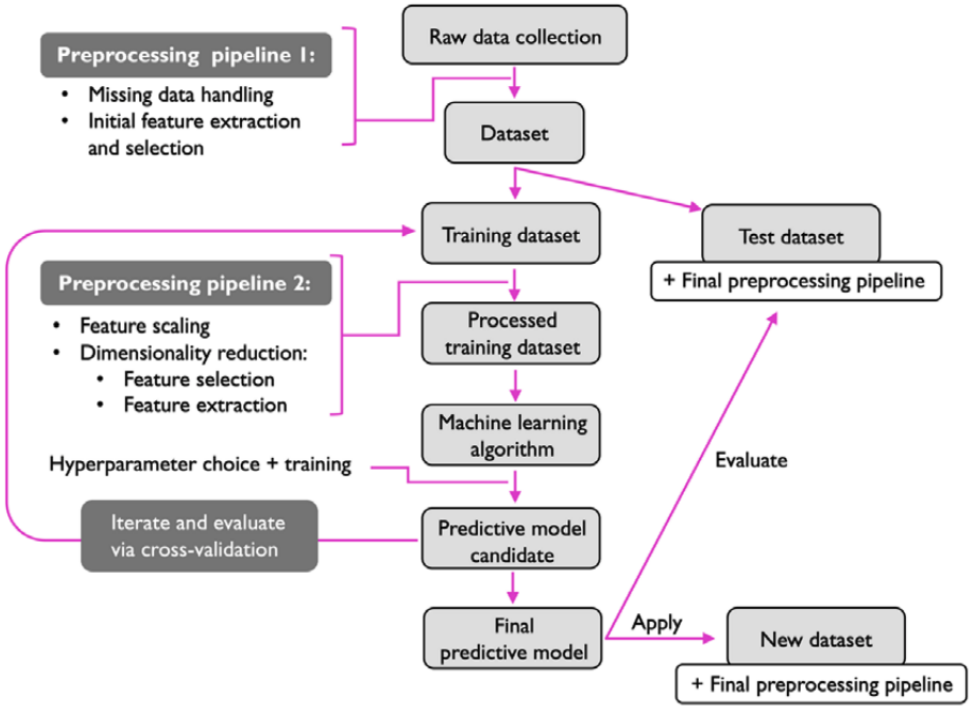


Figure 1.9: Predictive modeling workflow

Figure 5.5: A typical statistical learning workflow [32].

2. Second, at a general level it is a set of requirements defining a specific 'class' of code[4].
3. Third, at the physical level, it is code in the UMBC HPCF Ada cluster at the path `/nfs/rs/cybertrn/reu2024/team2/base/`.

The following sections describe BRIDE along these three levels of thinking.

Level of BRIDE 5.3.2.1 (High-Level Workflow)

Figure 5.5, from Ref. [32], depicts a standard workflow for applying statistical learning to inference tasks. This workflow was used as our guideline: it dictates an organized research process and enforces a certain rigor in the results. For example, BRIDE allows for a dedicated testing set, which is particularly important as reusing the same validation data subset multiple times to perform hyperparameter tuning causes it, in a way, to become part of the training set [8]

With these considerations in mind, the following platform structure, as displayed by Figure 5.6, was designed to provide a strong foundation for our work. This figure is a one-to-one description of BRIDE's existence as a workflow. Also, note that PP1 and PP2 are formally split by where the train-test split happens. PP1 contains preprocessing that is row independent (the transformations are independent for each row). In practice, this means PP1 will contain tasks such as converting raw data to a two dimensional array, class balancing, cleaning data, and making labels continuous, while PP2 will contain tasks including scaling, normalization, and feature selection and engineering.

There are some structures to note about the design of the BRIDE platform as a workflow:

[4]Note that this description may be misleading, as we don't expect the number of objects belonging to this 'class' to exceed 2 or 3.

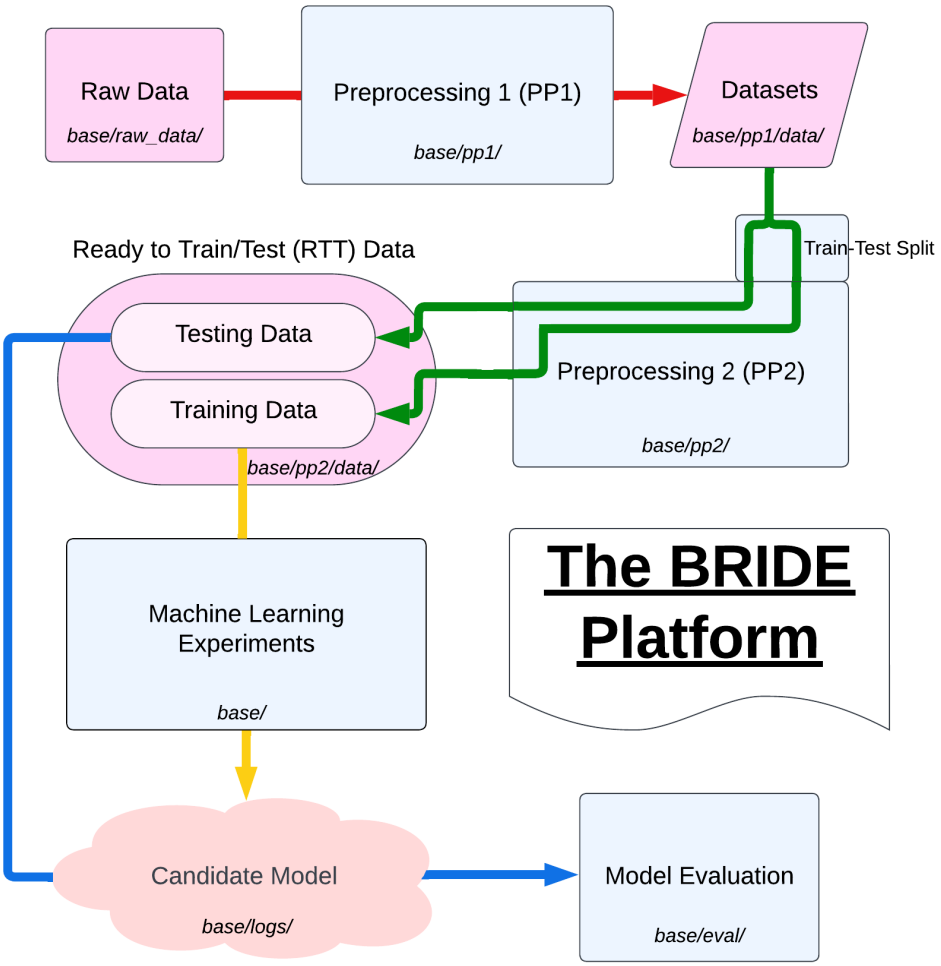


Figure 5.6: A flowchart of the structure of BRIDE.

- BRIDE has modularity, in that each process is standalone. This fulfills Goals 2.2 and 1.2. In particular, the latter goal is fulfilled because the modularity makes it so changes can be entirely focused on one section of the code base, making development and expandability easier. For example, as model evaluation is separated from model building, one could implement a new library wrapper for model building (e.g. from TensorFlow to PyTorch) without changing model evaluation.
- Rigorous generalization performance evaluation is built into BRIDE. This is related to Goal 2.1. By enforcing where the train-test dataset split happens, BRIDE prevents the issues discussed under Goal 2.1 from taking place.
- BRIDE also has "clean" and simple code, as the separation of processes is conducive to the systematic organization of code files into distinct folders. This fulfills Goal 3.1.

Level of BRIDE 5.3.2.2 (Modular Code System)

BRIDE's modular code system is essentially a set of requirements. BRIDE, on this level, is a

set of files organized in a Linux-like system with the aim of providing an effective place for research. The first and most important requirement is that each job, which corresponds to a particular run, has a unique identification code called a `run_id`, which is a string. The importance of unique `run_id`'s cannot be overstated; much of the functionality and record-keeping aspect of BRIDE is built around the `run_id`. The naming scheme used in Level 5.3.2.3 was `initialsMonthDigits.DateDigits_id-string`. For example, this could be `mc7.18_imFCNv2`.

It is assumed that there is a base directory in a Linux-like file system called `base/`, which contains an entire BRIDE object. In `base/`, there are a number of directories:

- `base/config/`: contains `run_id`.yaml files, each of which is the configuration for one run.
- `base/runs/`: contains `run_id` directories, each of which has a `train.sh` script for machine learning experiments, a `predict.sh` script for model evaluation, and a `slurm_output` directory for slurm output files.
- `base/logs/`: holds the logs of each run, the nature of which will depend on the implementation of the run.
- `base/eval/`: contains `run_id` directories, each of which has files from model evaluation depending on the implementation.

The `base/` directory must also include the below Python scripts:

- `model.py` contains the Python code to build models.
- `train.py` contains the Python code to do the machine learning experiments.
- `predict.py` contains the Python code to do model evaluation.
- `utils.py` contains all auxiliary Python code.

Finally, `base/` also contains the three directories `base/raw_data/`, `base/pp1/`, and `base/pp2/`, which comprise the BRIDE platform's preprocessing pipeline. This pipeline has multiple requirements relating to file types and data shapes:

- `base/raw_data/` contains directories, each of which has the raw data to be put through PP1 to make a dataset.
- Datasets are comma-separated values (.csv) files inside `base/pp1/data/`. They therefore must be 2 dimensional; they also must be a feature matrix with a rightmost target column attached. Labels in the target column must be continuous integers starting at 0. The labels should be balanced, and there must not be missing data.
- `base/pp1/` contains directories that hold PP1 routines. These directories contain Python scripts that are called by `pp1.sh` to run a PP1 routine which is one of only two files in `base/pp1/`. Intermediate files also reside in these directories. The other file in `base/pp1/` is `visualize.py`, which outputs visualizations to `base/pp1/visualize/`. This directory is filled with directories with the same name as the datasets (without the .csv extension).
- Ready-To-Train (RTT) data is organized in `base/pp2/data/`. This folder contains directories, each of which is the result of running a PP2 routine on a dataset. Each of these directories contains 2 directories, `base/pp2/data/name/train` and `base/pp2/data/name/test`. These directories have files the names `X.npy` and `y.npy`. These are the files that must be used by the machine learning experiments section code.

- `base/pp2/` is organized in exactly the same format as `base/pp1/`. However, the only additional requirement is that the train-test split must be the first event that happens in the PP2 routines. For this section, `sklearn`'s `Pipeline` class is recommended but not required. It is imperative that the transformers and scalers are fit to the training data and not the entire dataset. Also, tt is intended for the `pp1.sh` and `pp2.sh` scripts to be run from the command line.

The above restrictions enforce a well defined workflow in `base/`. New PP1 or PP2 routines are simply folders in `base/pp1/` or `base/pp2/`. Once a set of RTT data and a machine learning model have been defined, a `run_id` is made up by the user. The user makes a file in `base/config/` called `run_id`.yaml and completes it. They then construct a directory in `base/runs/` that is named the `run_id` and make a `train.sh` script (or, more likely, copy it). The user then submits the run. Once the run is finished, they make/copy a `predict.sh` script, get model files from `base/logs/`, and make them accessible to the prediction script (most likely through placing them in the .yaml file) to run prediction on the files in a `base/pp2/data/name/test/` directory. The results are saved in `base/eval/`.

We discuss how some of the goals from subsection 5.3.1 are fulfilled by this modular code system. By constraining file types and data shapes, each component of the platform can be adjusted without effect on the rest of the process. For example, since model evaluation always takes in a `y_pred.npy` file and a `y_truth` file, it can be rewritten once and both sklearn-based models and PyTorch neural networks can be used. This fulfills Goal 2.2. Next, the `run_id` requirement ensures that there is no overwriting, so records for runs are persistent as well as organized and accessible. This ensures Goals 1.1 and 2.1 are met. The ease of implementing new PP1 or PP2 routines as directories, or new models in `model.py`, fulfills the first part of Goal 2.2.

Level of BRIDE 5.3.2.3 (Actual Code Implementation)

The actual code, in the ada cluster, can be found at the GitHub repository [11]. The notebook at [10] contains a number of runs and the first major updates to the code base. The following list discusses a number of decisions regarding the code implementation for each process in the workflow:

- The rather involved preprocessing pipeline architecture was unfortunately not used to its full potential due to a lack of time. The pipeline in subsection 4.1 was used, with some minor changes. `base/raw_data/` contains MCDE output in appropriately named directories. PP1 is the conversion, shuffling, and merging steps from the original pipeline, along with a short script to make the labels continuous. PP2 is the transform step, after a split is done. The transformer is fit to the train data, and both the train and test data are transformed by it.

- As mentioned, PyTorch Lightning [15] is used for this machine learning process. This has made training on multiple GPUs seamless and virtually bug free. The file `model.py` contains classes, each of which inherits `lightning.LightningModule` and defines a model. Figure 5.9 contains an image of this. In addition to hyperparameters for models, yaml files in `base/config/` also contain fit and data parameters. Figure 5.7 contains an image of one such file. `utils.py` contains a single Lightning DataModule class. Tensorboard is used to visualize training curves real time, and uses `base/logs/tb_logs/run_id/`. An image of the interface is shown by Figure 5.8. There is also a CSVLogger and checkpointing call backs saving to `base/logs/.../run_id/`.

- `predict.sh`, as shown in Figure 5.10, calls both `predict.py`, which saves files to `base/eval/run_id/`, and `base/eval/eval.py`, which creates a confusion matrix, saving it in `base/eval/run_id/` and saves learning curves to `base/logs/csv_logs/run_id/...`

- We also utilized multiple `archive/` directories to keep the the directories organized and "clean". We also had up-to-date 'template' files for .yaml files or the training and prediction scripts for users to copy for runs.

Further technical code details can be found in [11]. A limitation of BRIDE, at this point, is a lack of a completely separate testing dataset (of which is not a data subset), though this is not required, or even used, in many cases. The simple train-test split is appropriate in most cases.

base > config > ! mc8.1_imFCNv8.yaml

```yaml
# FCNv19
run_id: 'mc8.1_imFCNv8'
pred_ckpt: ''
resume_ckpt: ''
mdl_key: 'impr_fcn'

data:
  train_data_path: '/nfs/rs/cybertrn/reu2024/team2/base/pp2/data/barajas/train/'
  test_data_path: '/nfs/rs/cybertrn/reu2024/team2/base/pp2/data/barajas/test/'
  batch_size: 4096
  val_split: 0.1

fit:
  max_epochs: 2000
  n_nodes: 1
  n_devices: 4
  patience:  500
  ckpt_freq: 100

model:
  input_size: 15
  num_classes: 13
  hidden_layers: [256, 256, 256, 256, 256, 128, 128, 128]
  activation: 'relu'
  lr: 0.001
  lr_step: 250
  lr_gam: 0.95
  penalty: 1.015
  dropout: 0.05
  l2: 0.01

```

Figure 5.7: An example .yaml file in`base/config/`

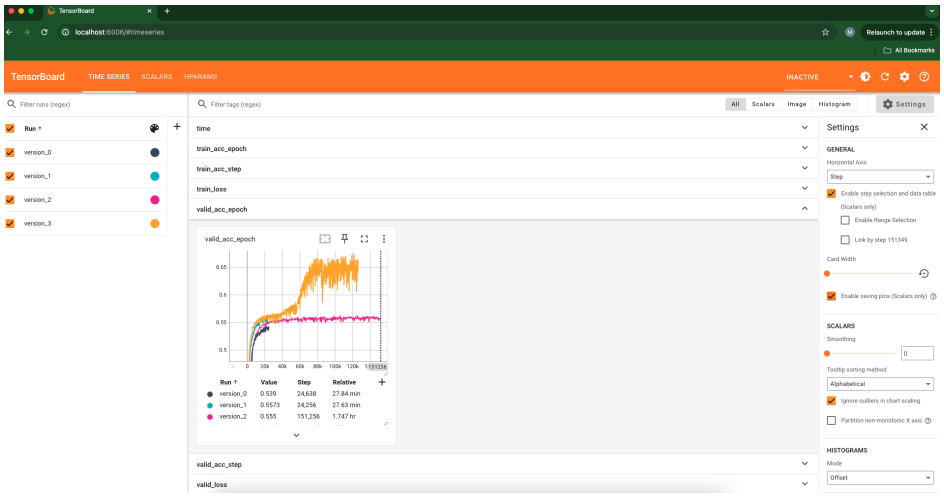
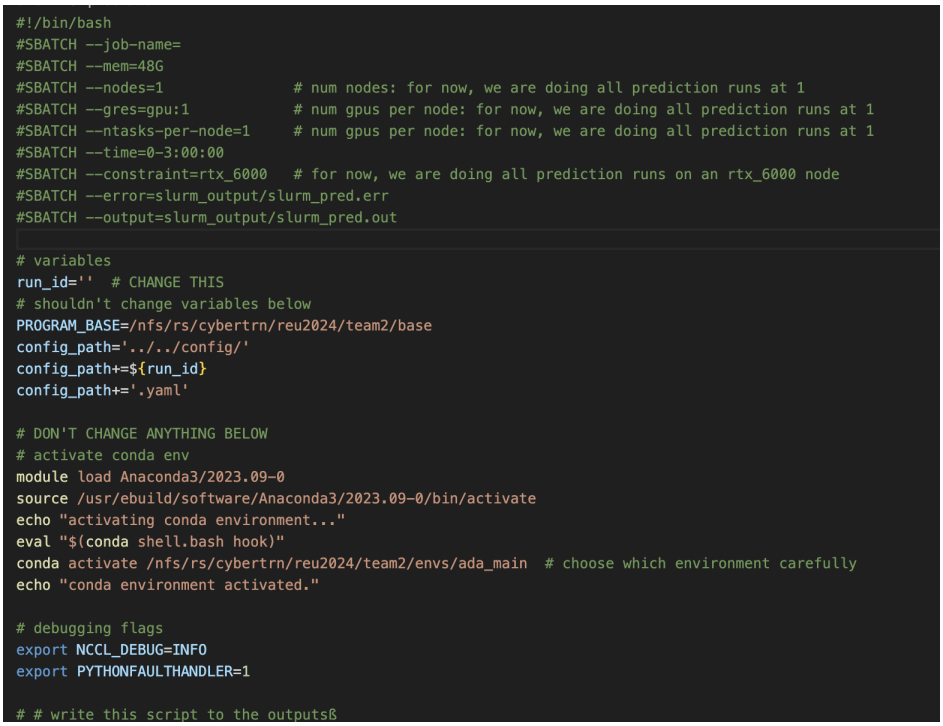


Figure 5.8: An image of Tensorboard in use.

```python
# FCN experimentation
class ImprovedFCN(pl.LightningModule):
    def __init__(self, input_size, num_classes, hidden_layers, activation, lr, lr_step, lr_gam, penalty, dropout=0, l2=0):
        super().__init__()
        self.save_hyperparameters()

        # PL attributes:
        self.train_acc = Accuracy(task='multiclass', num_classes=13)
        self.valid_acc = Accuracy(task='multiclass', num_classes=13)
        self.test_acc = Accuracy(task='multiclass', num_classes=13)

        # params
        self.input_size = input_size
        self.num_classes = num_classes

        self.hidden_layers = hidden_layers
        self.activation = utils.get_activation(activation)
        # old dropout
        # self.dropout = dropout
        # if dropout is not None: self.dropout_layer = nn.Dropout(dropout)
        # self.dropout = dropout
        # new dropout
        self.dropout_layer = nn.Dropout(dropout)

        self.lr = lr
        self.lr_step = lr_step
        self.lr_gam = lr_gam
        self.l2 = l2

        self.penalty = penalty
        self.optimizer_pointer = None
```

Figure 5.9: An example `model.py` class.



```
#!/bin/bash
#SBATCH --job-name=
#SBATCH --mem=48G
#SBATCH --nodes=1               # num nodes: for now, we are doing all prediction runs at 1
#SBATCH --gres=gpu:1            # num gpus per node: for now, we are doing all prediction runs at 1
#SBATCH --ntasks-per-node=1     # num gpus per node: for now, we are doing all prediction runs at 1
#SBATCH --time=0-3:00:00
#SBATCH --constraint=rtx_6000   # for now, we are doing all prediction runs on an rtx_6000 node
#SBATCH --error=slurm_output/slurm_pred.err
#SBATCH --output=slurm_output/slurm_pred.out

# variables
run_id=''  # CHANGE THIS
# shouldn't change variables below
PROGRAM_BASE=/nfs/rs/cybertrn/reu2024/team2/base
config_path='../../config/'
config_path+=${run_id}
config_path+='.yaml'

# DON'T CHANGE ANYTHING BELOW
# activate conda env
module load Anaconda3/2023.09-0
source /usr/ebuild/software/Anaconda3/2023.09-0/bin/activate
echo "activating conda environment..."
eval "$(conda shell.bash hook)"
conda activate /nfs/rs/cybertrn/reu2024/team2/envs/ada_main  # choose which environment carefully
echo "conda environment activated."

# debugging flags
export NCCL_DEBUG=INFO
export PYTHONFAULTHANDLER=1

# # write this script to the outputsß
```

Figure 5.10: An image of the `predict.sh` file.

6 Results

6.1 Initial Testing

We began by trying a wide range of machine learning models. These were broadly under the two categories of ensemble learning based off of decision trees and deep learning architectures designed for tabular data, as our data is essentially tabular in nature. However, our data is not heterogeneous, which could explain the relatively poor performance of these models, among other factors.

Random Forest, a machine learning method that utilizes an ensemble of decision trees to reach a single prediction, was found to have fair accuracy in previous research [6]. When using sklearn's RandomizedSearchCV for hyperparameter tuning, the most accurate Random Forest model, also applied in sklearn, obtained a validation accuracy of 54%. As these poor results may have been due to the model not being able to capture the complex patterns of our data, we turned to more intricate models, including gradient boosting algorithms. Gradient boosting algorithms are ensembles that also combine multiple decision trees; however, their sequential nature optimizes the model based on its previous iterations, leading to a more accurate model overall. We implemented three types of gradient boosting: XGBoost, LightGBM, and the Gradient Boosting Machine (GBM). Out of the three, an XGBoost model with tuned hyperparameters by RandomizedSearchCV achieved the best results; in 5-fold cross validation, validation accuracy was 64% on the feature engineered data. LightGBM validation accuracy was poorer, at 50%; however, this output was obtained on `barajas` data, which does not include several features that are present in the feature engineered dataset, and no hyperparameter tuning was utilized. Lastly, the GBM performed the poorest. Validation accuracy was 48% for both the `barajas` and feature engineered datasets, and the model took significantly longer to train than XGBoost and LightGBM.

As results for gradient boosting models were fairly poor, TabNet, a deep neural network architecture for tabular data, was implemented. TabNet has been shown to have comparable or even better performance than gradient boosting; in particular, it combines the benefits of both tree-based learning and neural networks. [22]. In the model, two feature transformers and a mask function prioritize features based on importance, not unlike that of which tree-machine learning algorithms. The outputs of these are then fed into a fully connected layer and attentive transformer, eventually leading to the final prediction. We utilized the TabNetClassifier from PyTorch-tabnet. Implemented with early stopping with `barajas` data, a TabNet model had a validation accuracy of 52%. In an attempt to improve upon this, RandomizedSearchCV to tune model hyperparameters was used; however, accuracy only increased to 55%. Though, early stopping of a few number of epochs was again utilized; training the model for a greater number of epochs with each random search iteration may increase accuracy, but would greatly increase the training time (for random search, runtime was already over 48 hours). As these TabNet results were uncharacteristically poor, we explored DANet, a deep abstract neural network for classification and regression with tabular data. The model framework combines several components called Abstract Layers, which do feature selection and abstraction, into a basic block; many basic blocks stack into a single DANet network. On the `barajas` dataset, DANet obtained a validation accuracy of 49%; this result, though, was lower than most tested of the other models.

As previously stated, a Gated Recurrent Unit (GRU) is a RNN layer that excels at capturing sequential patterns and in handling long-term dependencies. The gating mechanisms in a GRU reduce overfitting on model training data, which is especially beneficial for complex datasets such as ours. We implemented a neural network model composed of four GRU layers and 2 fully connected layers. Based on `shakeri-obe` data, there was a validation accuracy of 52%. However, on dataset of 8271 rows that was a subset of `shakeri-obe`, a much higher validation accuracy of 69% was

achieved (training accuracy was 80%), which is somewhat strange as more training data normally leads to greater accuracy, but this may be attributed to the model capacity being too large for the small dataset.

In addition to our other hybrid models, a neural network consisting of both convolutional and LSTM layers was performed. Convolutional neural networks (CNN), which are a type of feed-forward neural networks, consist of convolutional layers which combine input data with a convolution kernel to form a transformed feature map. These layers are arranged so that they usually learn simpler data patterns first, and more complex ones later. As CNNs are widely used for complicated tasks such as object recognition, we combined their convolutional layers and the intricate LSTM layers to form this CNN+LSTM hybrid model. Trained on a `barajas` dataset that was also feature engineered, 61% training and 62% validation accuracy was achieved. However, when trained on `barajas` that did not include the new features, there a significantly lower accuracy of 51% for both training and validation; this suggests that this type of model may have benefited a certain amount from the feature engineering, unlike our LSTM or FNN models.

6.2 Water Phantom Data

Instead of directly comparing our results from the study on RP data in subsection 6.3 with the past results on WP data (3.1), we opted to try to reproduce these WP results on BRIDE. There are clearly new challenges and subtle differences when using a new code implementation; we thought it would be best to compare RP results on BRIDE with WP results on BRIDE. Also, note that most of the initial results were tested on the water phantom dataset (`barajas`) as it has the most total samples of scatter data (1.8 million rows) in comparison to newly simulated real patient medium data (`shakeri-obe`) (499 thousand rows). It is a fundamental principle of machine learning that more training data usually gives neural networks better predictive power and better generalization ability [32]. The best performing models and architecture choices, when trained on the water phantom data, may be compared with their ability when trained and tested on the simulated patient data.

6.2.1 FCN Models

A naive approach to hyperparameter tuning was utilized to determine the best FCN model on the water phantom data, with heavy guidance from Section 3.1. We first implemented a baseline model based on Section 3.1, which was tuned to reach a model that achieved 65% accuracy, comparable to the nonresidual model in that Section [5]. The hyperparameters of that model are listed in Table 6.1.

[5] We were never able to implement a residual block model, so all FCN models in the results section are non-residual and will be compared to the corresponding result in Section 3.1.

Hyperparameter	Value
Hardware	4 rtx6000 GPUs
Validation split	0.1
Batch size	4096
FC layers	[15, 256, 256, 256, 256, 256, 128, 128, 128, 13]
Learning Rate	0.001
Learning Rate Change	0.95
Learning Rate Step	250
L2	0.01
Dropout	0.05
Loss Function	Cross Entropy + Custom Pairwise Loss (p=1.015)
Optimizer	AdamW
Activation Function	ReLU

Table 6.1: FCN Water Phantom Reproduction Hyperparameters.

The learning curves and confusion matrix of this model are shown in Figures 6.1 and 6.2.

Figure 6.1: Learning curves for FCN reproduction on the water phantom dataset.

Custom Pairwise Loss Function In Table 6.1, the model's loss function was listed as 'Cross Entropy + Custom Pairwise Loss (p=1.015)'. The Custom Pairwise Loss is a function we designed to boost the accuracy of models by penalizing wrong predictions *outside the correct event type* more than those inside the correct event type, with the three event types being true triple, DtoT, and false triple. It seems that an inherent flaw in the current PP1 is that the machine learning model does not "recognize" that of the 13 classes, the first 6, the 6 after that, and the last class form 3 defined groups. This is an order in the data that would be difficult for the model to learn; hence, it may be beneficial for the model to automatically know of the 3 groupings. We defined the custom pairwise loss term L_P to be

$$L_P = (1 + \text{avg}((D \cdot t) \cdot p))^h \tag{6.1}$$

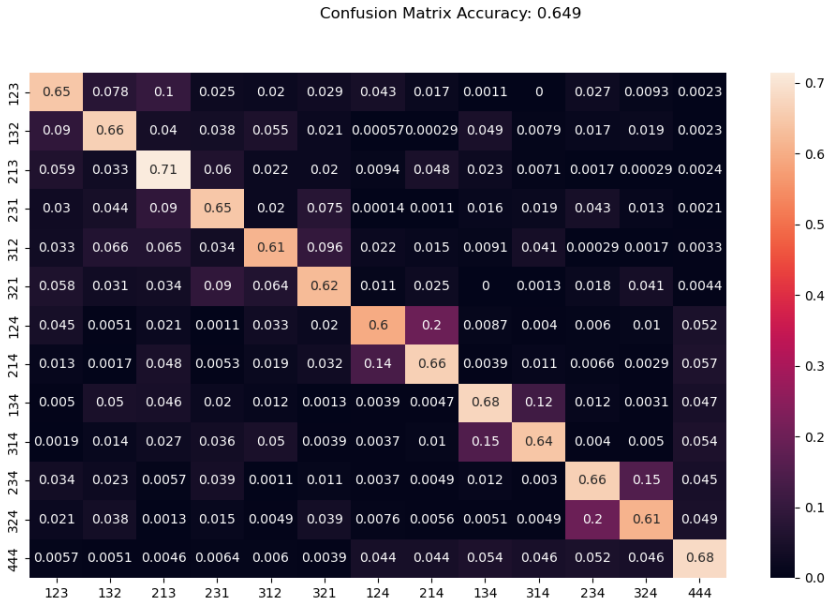


Figure 6.2: Confusion matrix for FCN reproduction on the water phantom dataset.

where h is called the penalty (and is a hyperparameter), D is a 13x13 matrix, and T and P are the target and prediction matrices, respectively. T and P are the one-hot encodings of target and prediction vectors, and are of shape 13x[batch size]. D_{ij} is the penalty factor for classifying class i as j. We used

$$D = \begin{bmatrix} 0 & 2 & 2 & 2 & 2 & 2 & 8 & 8 & 8 & 8 & 8 & 8 & 20 \\ 2 & 0 & 2 & 2 & 2 & 2 & 8 & 8 & 8 & 8 & 8 & 8 & 20 \\ 2 & 2 & 0 & 2 & 2 & 2 & 8 & 8 & 8 & 8 & 8 & 8 & 20 \\ 2 & 2 & 2 & 0 & 2 & 2 & 8 & 8 & 8 & 8 & 8 & 8 & 20 \\ 2 & 2 & 2 & 2 & 0 & 2 & 8 & 8 & 8 & 8 & 8 & 8 & 20 \\ 2 & 2 & 2 & 2 & 2 & 0 & 8 & 8 & 8 & 8 & 8 & 8 & 20 \\ 8 & 8 & 8 & 8 & 8 & 8 & 0 & 2 & 2 & 2 & 2 & 2 & 20 \\ 8 & 8 & 8 & 8 & 8 & 8 & 2 & 0 & 2 & 2 & 2 & 2 & 20 \\ 8 & 8 & 8 & 8 & 8 & 8 & 2 & 2 & 0 & 2 & 2 & 2 & 20 \\ 8 & 8 & 8 & 8 & 8 & 8 & 2 & 2 & 2 & 0 & 2 & 2 & 20 \\ 8 & 8 & 8 & 8 & 8 & 8 & 2 & 2 & 2 & 2 & 0 & 2 & 20 \\ 8 & 8 & 8 & 8 & 8 & 8 & 2 & 2 & 2 & 2 & 2 & 0 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 & 0 \end{bmatrix} \tag{6.2}$$

Hence, the quantity $\text{avg}((D \cdot t) \cdot p)$ will be greater if guesses outside the event type are made. This penalized more so for false triples. The syntax 'Cross Entropy + Custom Pairwise Loss (p=1.015)' is a bit misleading: L_P is typically somewhere between 1 and 2, so it is actually multiplied by the cross entropy loss function. From our tests p must be slightly greater than or equal to 1 for the loss to function properly. We believe that further adjustment and experimentation with this loss function could lead to a large improvement in model performance.

We tested the performance of the Custom Pairwise Loss function on an arbitrary run and found it led to an increase, although small, in training, validation, and testing accuracy of 1%, 1%, and

0.1%, respectively. See `mc7.27_imFCNv1` and `mc7.28_imFCNv1` in [11].

6.2.2 LSTM Models

Extensive model architectures and variable hyperparameter explorations were manually explored through training and testing versions of the LSTM-FCN on the `barajas` water phantom data. Constant hyperparameters throughout these trial runs are displayed in Table 6.2

Hyper-parameter	Value
Hardware	2 RTX6000 GPUs
Batch Size	4096
Loss Function	CrossEntropy
Optimizer	Adam
Activation Function	ReLU
LSTM Layers	4
Learning Rate	0.001
Learning Rate Change	0.95
Learning Rate Step	400
L2 Weight Decay	1×10^{-7}

Table 6.2: LSTM Parameters for `barajas` Water Phantom Dataset

Following the architecture from [3], a model with 4 LSTM layers followed by 12 linear fully connected layers (instead of 16 as to explore the results of some modest model simplification), with 256 neurons in all layers, was trained on the `barajas` dataset consisting of 1.8 million rows with a validation split of 20%. The variable parameters of this run are shown in Table 6.4:

Hyperparameter	Value
Validation split	0.2
LSTM Layers	4
FCN Layers	12
Neurons	256

Table 6.3: 4-LSTM 12-FCL Initial Testing Variable Parameters

The model achieved maximum training and validation accuracies of 76% and 64.3%, respectively, over 500 epochs; however, the model began severely overfitting around epoch 250, as observed by the validation loss and accuracy curves in Fig. 6.3. Thus, the run was ended at this relatively early epoch due to the intense overfitting observed.

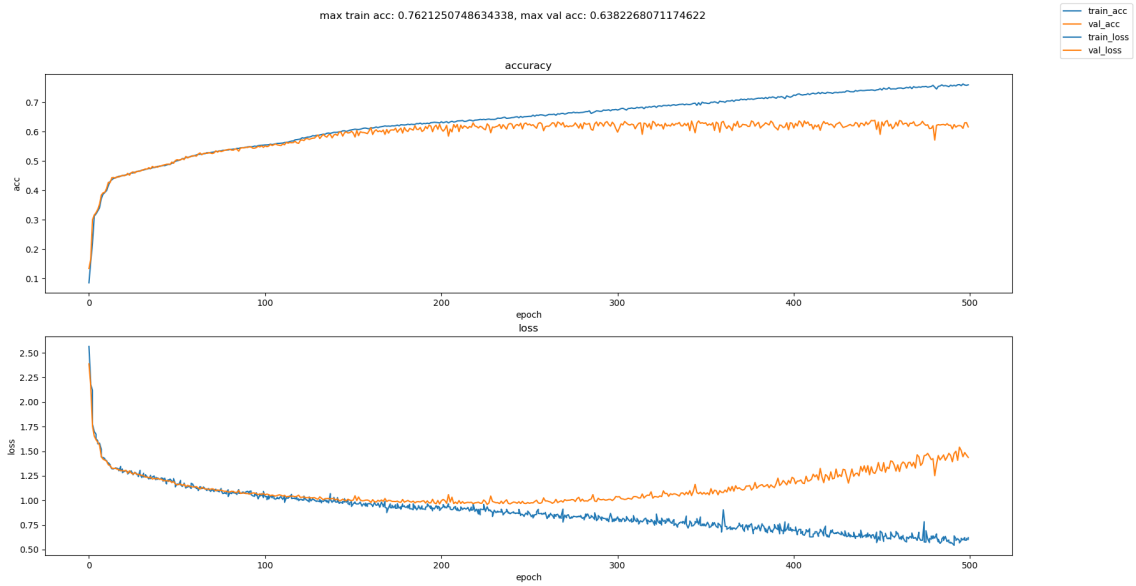


Figure 6.3: **Top:** Training (blue) and validation (orange) accuracies. **Bottom:** Loss over epochs for 4-LSTM 12-FCN 256 Neuron run showing clear overfitting and validation loss divergence.

In addition, the testing accuracy on a proper holdout set of size 10% was only 61% by the end of training, as displayed in the confusion matrix of Fig. 6.4. During prediction, the overfitting model correctly classifies both doubles and triples to similar percentages, although misclassifying adjacent doubles' classes to much larger degrees. In particular, it misclassifies class 6 doubles (124) as class 7 (214) 27% of the time. There is also similar behavior in the model misclassifying class 8 (314) as class 9 (134), class 10 (234) as class 11 (324), and vice versa. We can conclude that the model predicts the false event in the triple accurately, but often has trouble determining the correct ordering of the underlying doubles scatter.

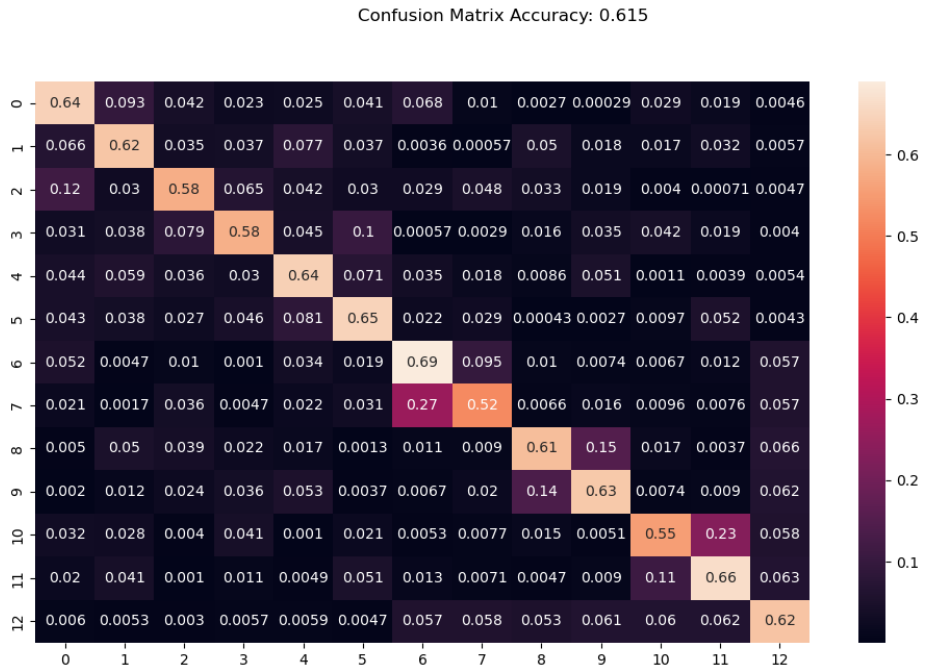


Figure 6.4: Confusion Matrix of 4-LSTM 12-FCN 256 Neuron run showing percentages of test set predictions in each class.

This suggested that simplifying the model architecture more could offer improvements in accuracy by decreasing overfitting. Despite the employment of dropout and weight decay, the dimensions of the model seemed to be too large, causing overfitting of training data. Improvements on the multi-layer LSTM-FCN model were made by simplifying the model architecture to 4 LSTM layers and 4 Linear Layers with 128 neurons per layer. In addition, the validation split was lowered as to supply the model more training examples to extract additional patterns in the dataset. These variable parameters are in Table 6.4:

Hyperparameter	Value
Validation split	0.1
LSTM Layers	4
FCN Layers	4
Neurons	128

Table 6.4: 4-LSTM 4-FCN 128 Neurons Per Layer run's variable parameters.

These parameter combinations and reductions in dense layer number, and neuron width supplied the current most successful PyTorch LSTM model on *exclusively* water phantom data as explored in this work. It achieved very similar results to [13] which had testing accuracies of 73%, compared to the present 72.5%. This can likely be attributed to using neuron linear layers of dimensions [128,128,128,128] as opposed to the [128,64] in [13]. We can surmise that additional reduction in complexity could increase testing accuracy even further. As observed in the training and validation accuracy/loss curves of Fig. 6.5, the maximum training and validation accuracies were 78.6% and 72.7%, on a validation split of 10% after 4000 epochs. Critically, reducing model complexity clearly

decreased the overfitting behavior (although not entirely) while increasing validation accuracy by more than 10%. The learning rate scheduler decreases the learning rate at 2000 epochs, reducing noise in accuracy and loss while also providing a sudden increase in accuracy.

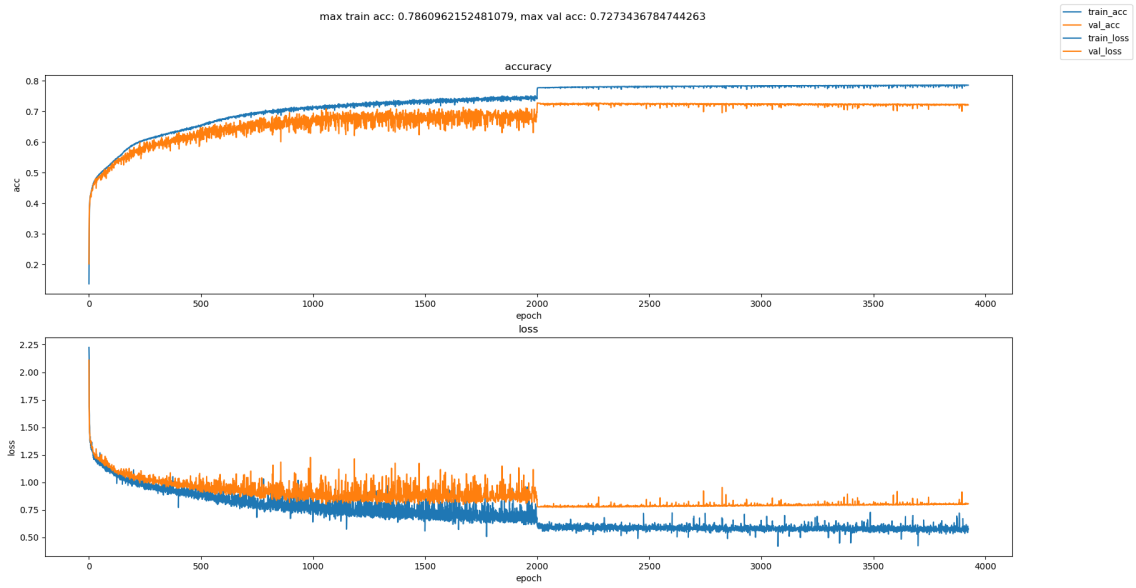


Figure 6.5: **Top:** Training (blue) and validation (orange) accuracies. **Bottom:** Loss over epochs for the 4-LSTM 4-FCN 128 Neuron run showing greatly reduced overfitting and higher accuracy.

Performing prediction of this model on the 10% split test set yielded a test accuracy of 72.5%. AS in the confusion matrix of the testing accuracy, there is 71-75% correct prediction in all triples except classes. As expected, due to the false single events which are difficult to predict, the doubles predictions are marginally worse in the 71-73% range. In summary, the network still correctly picks out the false event in doubles, but is more likely to misclassify the ordering of the doubles pairs. Finally the network also performs worse at false triples events at 67%.

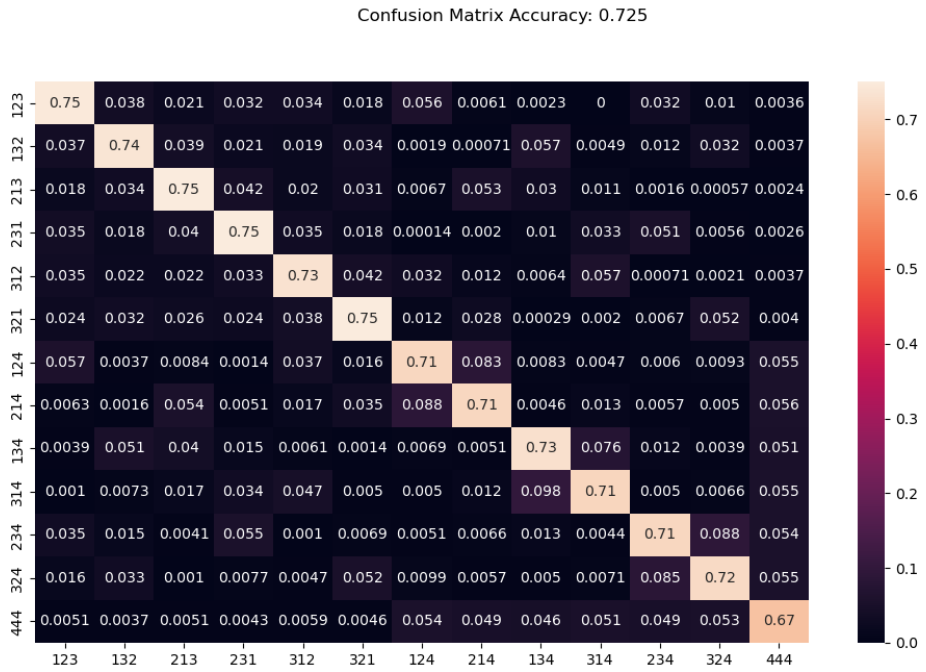


Figure 6.6: Confusion Matrix on 4-LSTM, 4-Linear, 128 Neurons Per Layer run, showing much higher percentage on predictions in all correct classes.

It must be noted, however, that this model is still overfitting as illustrated by the lack of clear convergence between the validation and test accuracies. Further experimentation should be performed in order to reduce overfitting, as well as aim to increase testing accuracy such as by applying stronger regularization techniques. But, to date, these are the most successful PyTorch mixed LSTM implementations on exclusively water phantom data, based on prior literature. These results serve as the foundation for the hyperparameter studies on the simulated real patient medium data.

6.3 Real Patient Data

In order to test the extensibility of machine learning results from water phantom to real patient data, and to gain further insights on real patient data, we conduct 2 initial hyperparameter studies – one for FCNs and another for LSTMs – on the new real patient data. Our studies are loosely based on those conducted in [12] and are divided into three stages:

- Stage 1: 'Spatter'. We run a multitude of tests to determine a reasonable set of constant starting parameters and identify 3 candidate hyperparameters where tuning could benefit. Our methodology here was very heuristic, so these runs will not be discussed. However, they can be found in [11].
- Stage 2: Hyperparameter Importance. For each of the 3 candidate hyperparameters, 2 values are chosen, and $(2)(2)(2) = 8$ tests are done to identify the 2 most influential parameters. The hyperparameter that is less influential is fixed at an optimal value.

- Stage 3: Final Tuning. For each of the 2 influential hyperparameters, 3 values are chosen, and (3)(3) = 9 tests are done to identify the optimal configuration of hyperparameters.

6.3.1 FCN Hyperparameter Study

Stage 2: Hyperparameter Importance The three candidate hyperparameters are listed in Table 6.5. An intuitive 'binary' labeling scheme was used to name tests: the two values of each of the candidate hyperparameters are assigned '0' and '1', and each run is a string of these binary digits in the order *batch size* | *dropout* | *neuron configuration*.

Hyperparameter	**0**	**1**
Batch Size	1024	4096
Neuron Configuration	18 layers, 582 ANL[6]	12 layers, 343 ANL[7]
Dropout	0.05	0.15

Table 6.5: FCN Stage 2 Candidate Hyperparameters.

Each test was run with periodic checkpointing (e.g. a checkpoint was saved every n epochs) and early stopping so that after `patience`=500[8] epochs without improvement in validation loss, the job would end. A checkpoint model from an epoch in the close vicinity of the maximum validation accuracy was tested to produce a testing accuracy. If a run showed erratic behavior or very low validation accuracy, it was not tested. The results of the 8 tests are shown in Table 6.6. Table 6.7 contains the constant hyperparameters.

Test	**Epoch**	**Max Train Acc., Train Curve**	**Max Val. Acc., Val. Curve**	**Test Acc.**
000	604	0.7005, concave	0.52, prominent peak	0.517
001	574	0.36, very noisy convex	0.36, very noisy convex	-
010	551	0.669, concave	0.53, small peak	0.523
100	658	0.7566, concave	0.53, prominent peak	0.515
011	645	0.5607, noisy concave	0.53, noisy peak	0.527
110	576	0.7316, concave	0.52, small peak	0.525
101	709	0.43, very noisy concave	0.44, very noisy concave	-
111	616	0.5975, concave	0.53, small peak	0.532

Table 6.6: FCN Stage 2 Tests

In order to evaluate how influential each of the candidate hyperparameters was, 'pairs' of runs were considered. That is, for every candidate hyperparameter, there are 4 sets of two runs in which, for those 2 runs, everything other than that candidate hyperparameter is held constant. For example, one of these 4 sets for batch size is highlighted in Table 6.8. We can see that the larger batch size (4096 vs 1024) appears to increase training accuracy but has minimal effect on testing accuracy between these two runs. The other 3 sets for batch size are highlighted in red, orange, and green in Table 6.9.

Completing similar analyses for the other candidate hyperparameters, the following conclusions were drawn about how influential each of the hyperparameters were:

[6] Average neurons per layer; full configuration is [4096, 2048, 1024, 512, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 128, 64, 32, 16].

[7] Average nuerons per layer; full configurations is [512, 1024, 512, 256, 256, 256, 256, 256, 256, 256, 256, 128].

[8] (chosen arbitrarily)

Hyperparameter	Value
Hardware	2 RTX6000 GPUs
Validation Split	0.1
Learning Rate	0.001
Learning Rate Change	0.95
Learning Rate Step	500
L2	0.01
Loss Function	Cross Entropy + Custom Pairwise Loss (p=1)
Optimizer	AdamW
Activation Function	ReLU

Table 6.7: FCN Stage 2 Constant Hyperparameters.

Test	Epoch	Max Train Acc., Train Curve	Max Val. Acc., Val curve	Test Acc.
000	604	0.7005, concave	0.52, prominent peak	0.517
001	574	0.36, very noisy concave	0.36, very noisy concave	-
010	551	0.669, concave	0.53, small peak	0.523
100	658	0.7566, concave	0.53, prominent peak	0.515
011	645	0.5607, noisy concave	0.53, noisy peak	0.527
110	576	0.7316, concave	0.52, small peak	0.525
101	709	0.43, very noisy concave	0.44, very noisy concave	-
111	616	0.5975, concave	0.53, small peak	0.532

Table 6.8: Set 1 for batch size importance analysis.

Test	Epoch	Max Train Acc., Train Curve	Max Val. Acc., Val. Curve	Test Acc.
000	604	0.7005, concave	0.52, prominent peak	0.517
001	574	0.36, very noisy concave	0.36, very noisy concave	-
010	551	0.669, concave	0.53, small peak	0.523
100	658	0.7566, concave	0.53, prominent peak	0.515
011	645	0.5607, noisy concave	0.53, noisy peak	0.527
110	576	0.7316, concave	0.52, small peak	0.525
101	709	0.43, very noisy concave	0.44, very noisy concave	-
111	616	0.5975, concave	0.53, small peak	0.532

Table 6.9: Sets 2, 3, and 4 for batch size importance analysis

- Batch size has little effect on validation and test accuracy. A larger batch size tends to overfit the data (increases training accuracy).
- Dropout has a small positive effect on validation and test accuracy if it doesn't cause the training to diverge. It decreases training accuracy significantly. This is expected, since dropout prevents overfitting. This trend is shown by Figure 6.7.
- The second neuron configuration performed much better than the first. With a slight decrease in training accuracy, it greatly increased validation and test accuracy.

It is also noted that a higher dropout with the more complex network caused training to diverge. Also, all models were overfitting to a degree except for the high dropout with a less complex network

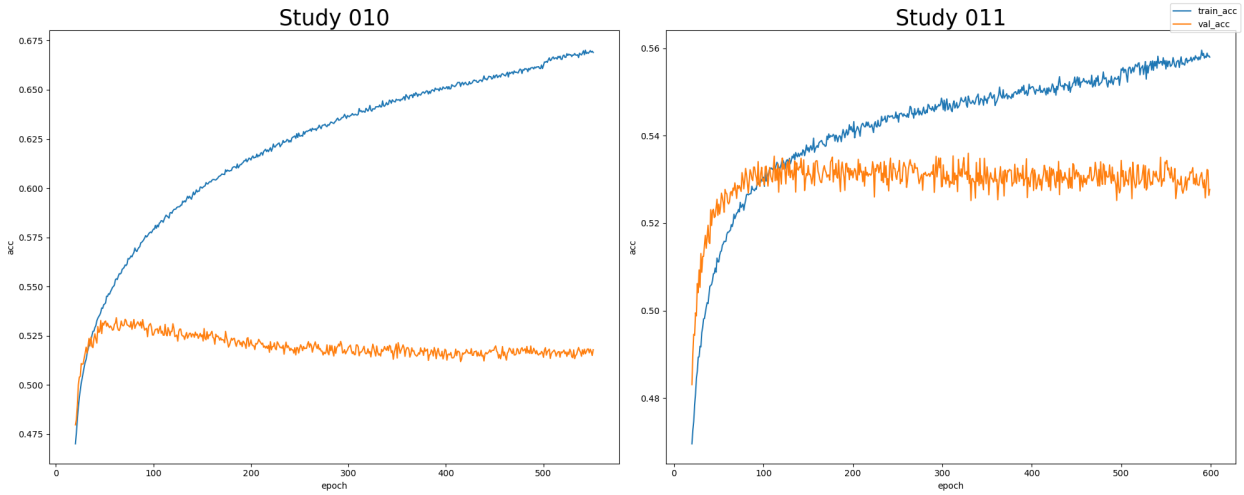


Figure 6.7: Dropout's effect on overfitting.

runs.

With these conclusions in mind, **dropout** and **neuron configuration** were chosen as the most influential hyperparamters to be studied in Stage 3. In addition, the constant hyperparameters were adjusted according to these conclusions, as shown in Table 6.11.

Stage 3: Final Tuning The values for dropout and neuron configuration studied in Stage 3 are shown in Table 6.10. A similar scheme to the binary naming in Stage 2 is used here, except in this stage the ternary digits 1, 2, and 3 are used. Table 6.11 contains the constant hyperparameters.

Hyperparameter	**1**	**2**	**3**
Neuron Configuration	12 layers, 315 ANL	8 layers, 272 ANL	8 layers, 68 ANL[9]
Dropout	0.05	0.2	0.4

Table 6.10: FCN Stage 3 Hyperparameters

Hyperparameter	**Value**
Hardware	2 RTX6000 GPUs
Validation split	0.1
Batch size	512
Learning Rate	0.0008
Learning Rate Change	0.95
Learning Rate Step	500
L2	0.01
Loss Function	Cross Entropy + Custom Pairwise Loss (p=1)
Optimizer	AdamW
Activation Function	ReLU

Table 6.11: FCN Stage 3 Constant Hyperparameters; changes are highlighted in yellow.

[9]Full configurations: [1024, 512, 256, 256, 256,256,256, 256, 256, 256, 128, 64], [512, 256, 256, 256, 256, 256, 256, 128], and [128, 64, 64, 64, 64, 64, 64, 32]

The 9 tests of this stage were conducted in the same manner as the 8 tests of Stage 2. The results are shown in Table 6.12. Also, note that several tests were stopped at 2000 epochs not because of early stopping, but due to an arbitrary constraint on epoch number. Some of these runs were resumed if accuracy appeared to be increasing, while the others were left as is. Note the lack of

Test	Epoch	Max Train Acc., Train Curve	Max Val. Acc., Val. Curve	Test Acc.
11	578	0.6068, concave	0.53, small peak	0.528
12	1531	0.51, very noisy concave	0.52, very noisy concave	0.514
13	581	0.39, noisy peak	0.41, small peak	-
21	564	0.6232, concave	0.56, noisy concave	0.550
22	3649	0.5387, noisy concave	0.55, noisy concave	0.540
23	1999	0.475, noisy concave	0.50, noisy concave	0.495
31	2061	0.5274, concave	0.54, noisy concave	0.534
32	1999	0.4698, noisy concave	0.48, noisy concave	0.483
33	1393	0.3306, noisy concave	0.35, noisy concave	-

Table 6.12: Results of the FCN Stage 3 tests; best accuracy result is highlighted in green.

"peaks" in the descriptions of the validation curves, when compared to the Stage 2 tests. This may be due to the measures designed to reduce overfitting implemented: instead of continuing to "learn" non-generalizable information about the training set, the models simply plateau in predictive performance.

Conclusions The best test is highlighted in green in Table 6.12. Hence, the optimal configuration for the FCN model found in this study is:

- Neuron Configuration: 8 layers, 272 ANL
- Dropout: 0.05
- Batch Size: 512
- L2: 0.01
- Loss Function: Cross Entropy + Custom Pairwise Loss (p=1)
- Optimizer: AdamW
- Activation: ReLU.

This model achieved an accuracy of 55% on the `shakeri-obe` dataset (the model at epoch 450 was used). The accuracy learning curves and confusion matrices for this model are included in Figures 6.9 and 6.8, respectively. From these figures, it is clear that the model is still overfitting, and that it does much better at classifying false triples compared to any other class.

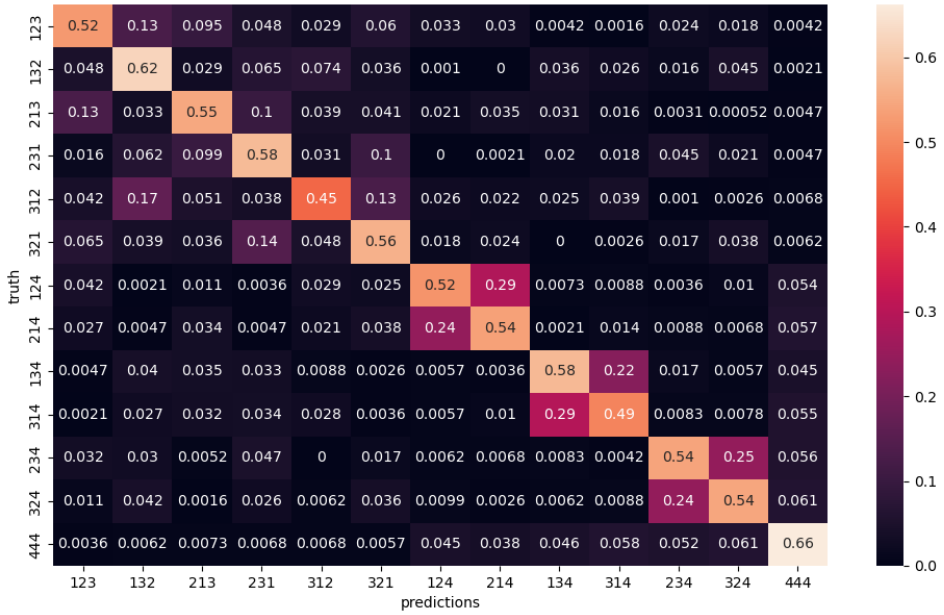


Figure 6.8: Test set confusion matrix for Test 21.

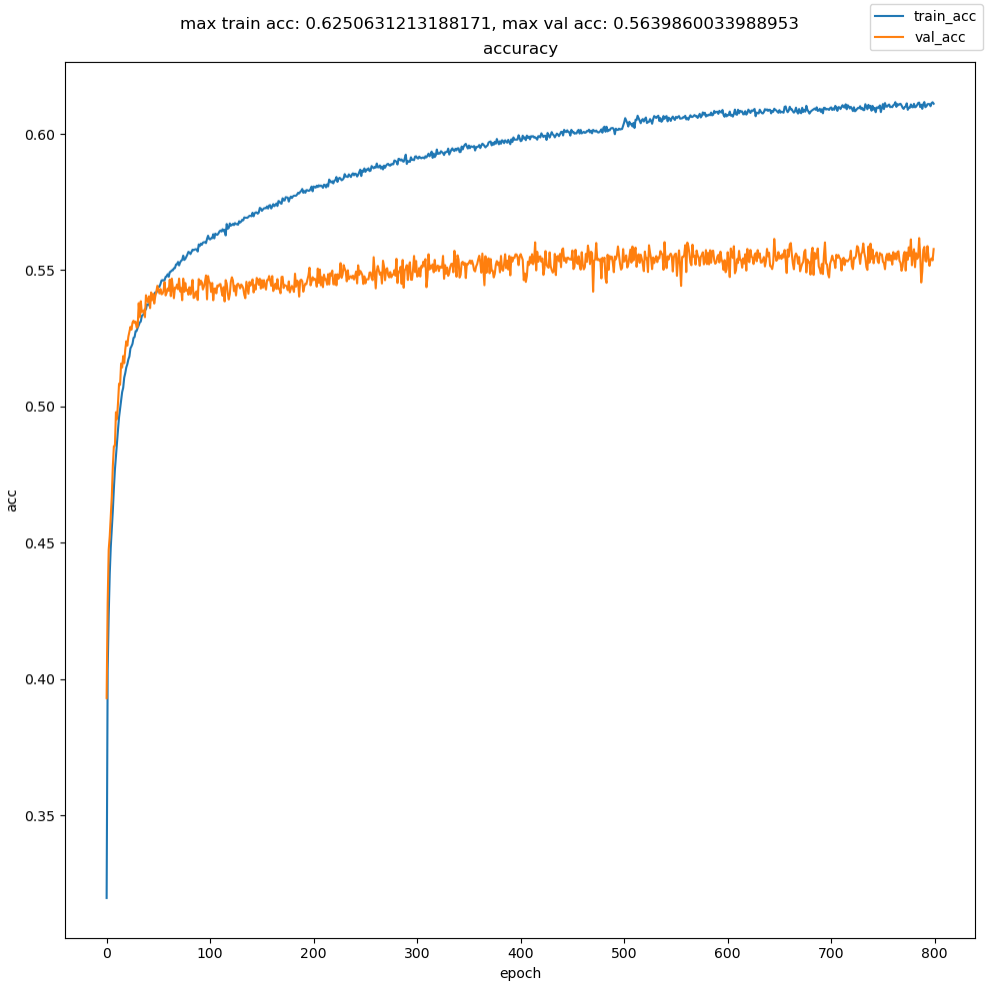


Figure 6.9: Accuracy learning curves for Test 21.

6.3.2 LSTM Hyperparameter Study

In light of the previous success of a 4 layer LSTM model with 2 dense layers in [12], versions of this model were implemented in PyTorch on BRIDE, on the simulated real patient data. Key differences that we elected to employ in this study are using the AdamW optimizer and the custom loss function implementation. The choice to use AdamW for L2-regularization purposes over the more common Adam optimizer with its sub-optimal form of weight decay is supported by [20], as well as by initial testing comparisons in which AdamW was observed on several datasets, especially `mothership`, to reduce overfitting. The second choice of using the custom loss function was included as it was expected that the custom loss would allow the models to generalize better. Table 6.13 contains the constant parameters of this LSTM hyperparameter study.

Hyperparameter	Value
Hardware	2 RTX6000 GPUs
Validation Split	0.1
Loss Function	Categorical CrossEntropy + Custom Pairwise Loss (p=1)
Optimizer	AdamW
Activation Function	Leaky ReLU
LSTM Layers	4
Learning Rate	0.001
Learning Rate Change	0.95
Learning Rate Step	100
L2	0.01
Custom Loss	True
Penalty	1

Table 6.13: LSTM Constant Parameters

Stage 2: Hyperparameter Importance The three LSTM candidate hyperparameters are listed in Table 6.14. The hidden layer values are the neuron dimensions of the hidden layers after the LSTM layers. Similar to before, a binary naming system from top to bottom of Table 6.14 was utilized.

Hyperparameter	**0**	**1**
Batch Size	2048	4096
Hidden Layers	[128, 64]	[128,128,128,128]
Dropout	0.15	0.45

Table 6.14: LSTM Stage 2 Candidate Parameters

The results from this stage of hyperparameter exploration are summarized in the Table 6.15.

Test	Epoch	Max Train Acc.	Max Val. Acc.	Val. Loss Minus Train Loss
000	671	64.2%	55.3%	0.89
001	683	61.8%	55.1%	1.02
010	712	62.5%	55%	0.67
011	699	61%	54.7%	0.83
100	967	60.5%	55.6%	0.429
101	705	62.5%	55.3%	0.77
110	902	60.1%	55.2%	0.35
111	765	60.5%	54.72%	0.74

Table 6.15: LSTM Stage 2 Tests results; best accuracy result is highlighted in green.

As displayed in Table 6.15, most results attain very close final training and validation accuracies of 60-61% and 54-55%, respectively. Among all runs, the patience ended training when validation accuracy remained stagnant for 500 epochs.

To first discuss differences in model accuracy as a result of dropout (where all other hyperparmeters were held constant), setting dropout to 0.45 generally resulted in the model stopping at an earlier epoch. In addition, it also lowered the gap between the max validation accuracy and max training accuracy for each run, yet counter-intuitively increased the difference between validation and training loss at the final epoch. The direct difference in accuracies of setting dropout higher indicates that it does have the intended effect of reducing overfitting. However, from the observation of final loss gaps being worse with higher dropout, it may indicate that once the model has started to already overfit, it will overfit worse (in terms of loss) when dropout is higher. In other words, as soon as the validation loss starts to increase, the divergence between training and validation losses and accuracies is worse with higher dropout. This may also be confirmed graphically comparing runs `100` and `101`. It can be observed from Figure 6.10 that on the plots with dropout 0.45, validation accuracy briefly surpasses the training accuracy within the initial 100 epochs; but, when the validation loss starts to increase, the overfitting behavior gets worse than the top plot where dropout is 0.15.

The next tuned hyperparameter to examine is the hidden layer input neuron sizes. The choice of comparing the four layer neuron architecture of [128,128,128,128] to [128,64] was used to compare whether simplifying the model by with less linear layers and less total neuron width would result in better accuracies and reduce over fitting. By directly comparing neuron choices with all other hyperparameters held constant, both the training and validation accuracy are marginally higher (1-2%) with the shallower models. However, the final loss gaps are greater for the shallower models. This may indicate that the shallower architectures could result in better accuracy when the models are performing at their best, but are prone to more severe overfitting once the models have begin to overfit. However, in the larger model architecture, a larger proportion of neurons must simultaneously begin overfitting to reach the same degree of loss gap as in the smaller model.

Finally, analyzing the contrasting batch sizes, the higher batch size runs resulted in marginally lower training accuracies among all runs (1-3%) and slightly higher validation accuracies (0.5%). Critically, among all runs, the higher batch size resulted in significantly lower final loss gaps in all cases. This does indicate that having a higher batch size seems to decrease the severity of overfitting in both the maximum gap of accuracies and losses between training and validation sets. This may suggest with these datasets that lower dropout and higher batch size are preferable for reducing overfitting (once the models are already overfitting). Neuron and layer adjustments have the most direct affect on increasing validation accuracy, as expected. However, none of the adjustments overall in hyperparameters offered a significantly higher validation accuracy, which was a key goal

of this hyperparameter study.

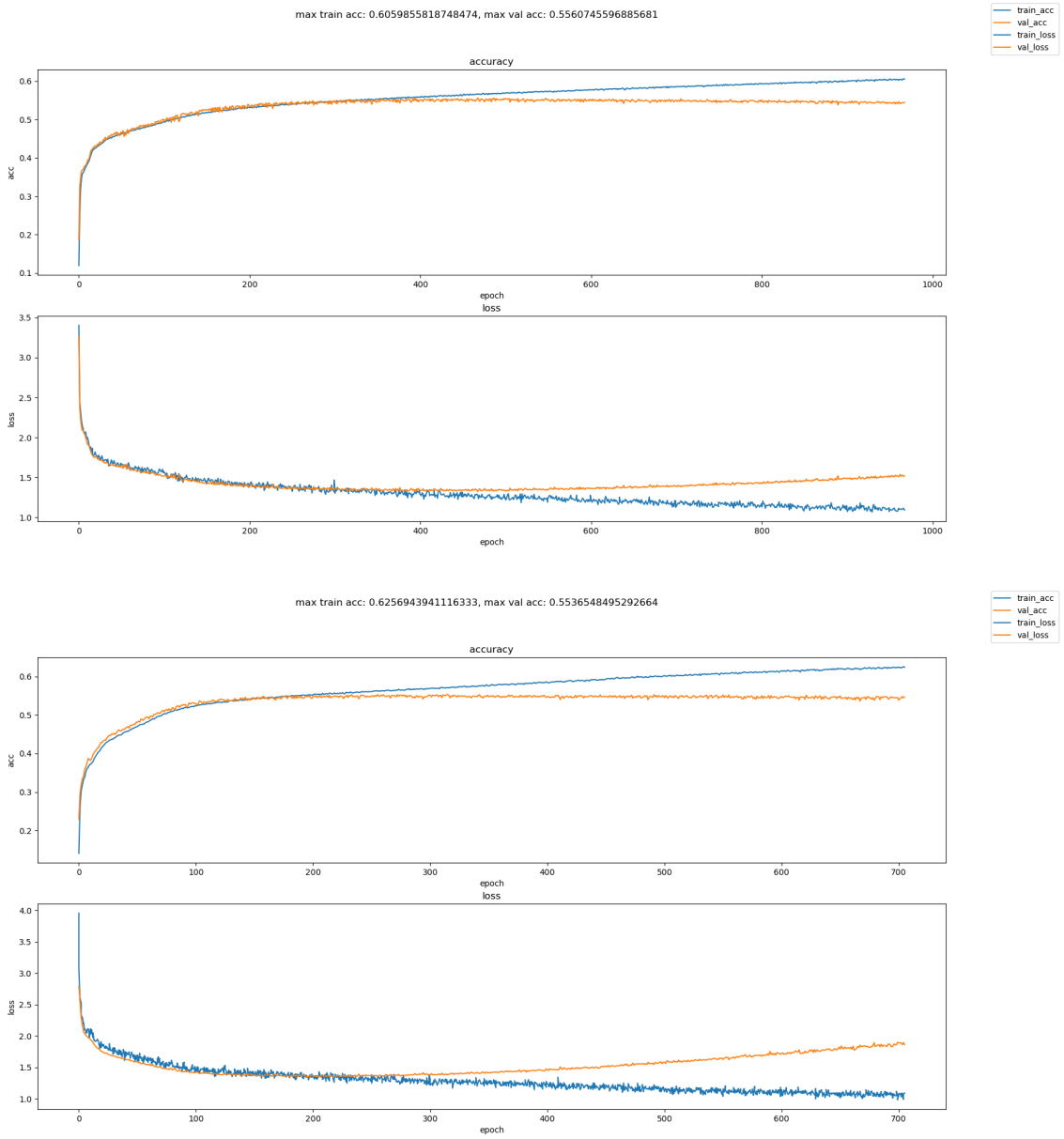


Figure 6.10: Training and loss learning curves for runs with 4096 batch size and hidden neurons [128,64] showing contrasting behavior of dropout. **Top:** dropout of 0.15. **Bottom:** dropout of 0.45.

6.4 Comparisons between Water Phantom and Real Patient Data

First, we note that the key results[10] on water phantom data in Section 3.1 were successfully reproduced on BRIDE, verifying the validity of experiments performed on BRIDE. First, a FCN with hyperparameters listed in Table 6.1 achieved a 65% accuracy, surpassing the corresponding

[10]Excluding the residual blocks model.

result in [3]. Its architecture was [512, 256, 256, 256, 256, 256, 256, 128]. Next, an LSTM with hyperparameters given in Section 6.3.2 achieved an accuracy of 72.5%, comparable to that in [12]. This model's architecture was 4 LSTM + 4x128 FCL.

The key results from our hyperparameter studies on RP data are summarized in Table 6.16. There are three salient comparisons to be made. First, the accuracies on the RP dataset are much

Model	**Accuracy**	**Architecture**
FCN	55.0%	[256, 256, 256, 256, 256, 128, 128, 128]
LSTM	55.6%	4 LSTM + 128,64 FCL

Table 6.16: Key results from RP hyperparameter studies.

lower, which confirmed our initial beliefs. Second, the LSTM does much better relative to the FCN on WP data, when compared to RP data. This implies that either the FCN architecture is specifically advantageous for RP data or LSTM architecture is particularly better to learn from WP data. Finally, we note that the architectures of best models were slightly different, but surprisingly similar. This may be because the data are ultimately of similar form.

7 Additional Studies

7.1 Mothership Dataset

As shown by Figure 7.1, a model containing 4 LSTM layers and 4 fully connected layers on the `mothership` data resulted in a training accuracy of 80%, a validation accuracy of 76%, and a test accuracy of 76%, our overall highest accuracies in this research. Our hyperparameters, listed in Table 7.1 for this model, were an adaptation of those of the 4 LSTM and 2 fully connected layers model of [13]. There are, however, some distinctions: in terms of parameters, this research's model's optimizer as Adam, instead of Nadam, and batch size was 4096 instead of 2048. ReLU activation was after all of the layers, instead of between the layers, and we had 4 fully connected layers, instead of 2. Importantly, though, our model was built using a more flexible PyTorch, compared to Tensorflow, and on the BRIDE platform.

Hyperparameter	Value
Hardware	4 RTX6000 GPUs
Batch Size	4096
Validation Split	0.1
Loss Function	Cross Entropy
Optimizer	Adam
Activation Function	ReLU
LSTM Layers	4
FCN Layers	4
Neurons	128
Learning Rate	0.001
Learning Rate Change	0.1
Learning Rate Step	2000
Dropout	0.0

Table 7.1: LSTM Parameters on the Mothership dataset

As most of our runs' accuracies were plateauing after a few hundred epochs, this current model implemented a different and distinct learning rate scheduler. A learning rate scheduler, in a neural network, is used to change the learning rate during the training process. Among several types of schedules is a step schedule, which involves adjusting the learning rate after a certain number of epochs. With this model, the learning rate began at 0.001; every 2000 epochs, the learning rate was multiplied by 0.1. This learning rate scheduler also involved running the model for thousands of epochs, a greater number than what we did previously. As displayed by Fig. 7.1, we notice a rise in accuracy after the learning rate is lowered at 2000 epochs, but no significant accuracy change at 4000 epochs. This may have been due to a very small learning rate by the 4000th epoch. There does seem to be a slight overfitting, which the validation accuracy approximately 4% lower than the training accuracy. This difference, however, is very prevalent in terms of machine learning, and is small considering that regularization was not applied. The model has a final testing accuracy of 76% as in Fig. 7.2 , which is a significant improvement from the previous studies. The cause of this improvement in accuracy may have been a greater volume of data; the `mothership` dataset consists of 3.8 million observations, compared to 1.4 million row data used in [25]. This conforms with the general trend in machine learning, in that a larger amount of data generally may lead to a better predictive performance. The `mothership` dataset, as stated in Subsection 4.2.1, consists of both simulated patient and water phantom data; hence, this result may be a more robust application in a medical situation.

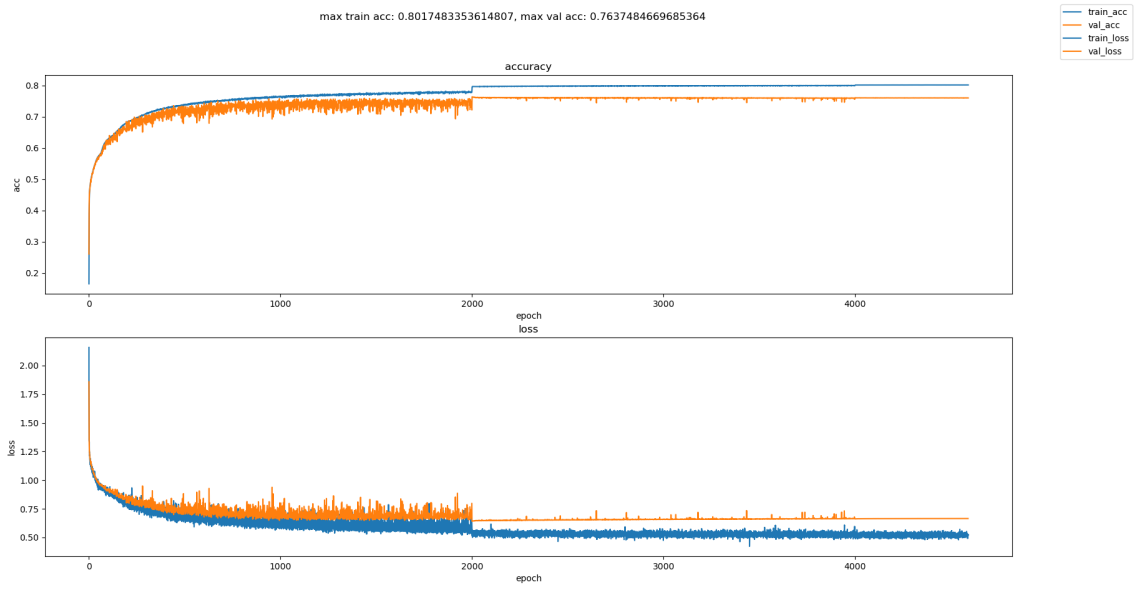


Figure 7.1: Accuracy and loss learning curve for LSTM `mothership` model

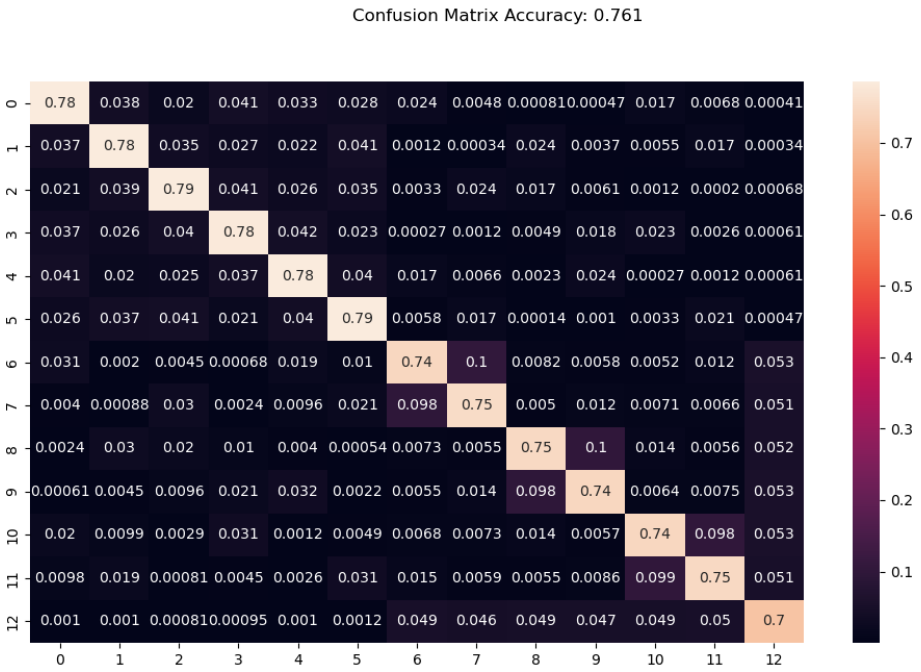


Figure 7.2: Testing set confusion matrix for the LSTM Mothership model

7.2 Effect of GPU Number on Performance

Another test that was explored was how utilizing different numbers GPUs affected the models. Based on past research [3], it is expected that increasing GPUs will decrease the time it take for the job to complete. Our results are consistent with past research. For example, with a particular fully connected neural network trained for 500 epochs, a 2 GPU run had a runtime of 1.32 hours, while that of 4 GPUs had a runtime of 0.66 hours. Doubling the number of GPUs from 2 to 4 in this case approximately halved the runtime; the linear speedup for this run is especially interesting, as a linear speedup is optimal but usually does not occur in practice [26]. A run on 8 GPUs was also implemented with the same hyperparameters as the runs above; the training time was approximately 0.30 hours. This speedup is not linear, though. Table 7.2 gives a summary of our comparisons of training time with different numbers of GPUs.

Number of GPUs	Runtime (Hours)
2	1.32
4	0.66
8	0.30

Table 7.2: Comparison of Runtime for Different Amount of GPUs for a Particular Neural Network

7.3 Feature Engineering

Feature engineering is a standard machine learning preprocessing technique that involves transforming data into a more effective format to be used as inputs. In particular, a type of feature engineering is creating new features, which is known to be sometimes helpful in the sense that more

information can be extracted from the existing features. This is especially so when working with a dataset such as the ones used in this research that have a relatively few amount of features. For example, many production machine learning models are developed on datasets with thousands of features. Adding new features using mathematical operations on existing features may only give marginal results, though, as these combinations of features should hypothetically also be inferred on through the composition of nonlinear activation functions. Thus, as part of a second preprocessing pipeline, we also explored feature engineering on the initial 12 features (e_i, x_i, y_i, z_i). Previous work introduced "Euclidean distances" as part of the first preprocessing pipeline [3, 5]. This euclidean distance calculates the 3 dimensional euclidean metric distance between the three points of gammas in one triples interaction. This generates 3 new features, which are used in all datasets. Mathematically, for each row

$$\begin{aligned}\text{euc1} &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \\ \text{euc2} &= \sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2 + (z_2 - z_3)^2} \\ \text{euc3} &= \sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2 + (z_3 - z_1)^2}\end{aligned}$$

The following features are exclusively used in the *feature engineered* datasets.

The next features added were the total energy and differences in energies for each row of interaction data. We believed that these features might give a very straightforward implication for the scatter class, as the energy in a true triple (1-2-3) is expected to decrease with each emission in the scattering event sequence at roughly $e_1 > e_2 > e_3$. This generates 4 additional features with the following relationships:

$$\begin{aligned}E_T &= e_1 + e_2 + e_3 \\ \Delta E_1 &= e_1 - e_2 \\ \Delta E_2 &= e_2 - e_3 \\ \Delta E_3 &= e_3 - e_1.\end{aligned}$$

The final features added are based off of the key relationship at the heart of Compton scattering: the Compton equation. The Compton equation defines the mathematical relationship between the angle of prompt-gamma emission, initial energy, and final energy. Given the electron rest energy keV, for any two prompt gamma rays the following relationship holds:

$$\cos(\phi_{ij}) = 1 - m_0c^2(\frac{1}{e_i} - \frac{1}{e_j}).$$

The cos of the angles of scatter where $m_0c^2 = 0.511$ keV and ϕ as the right hand side of the Compton equation were included as features. The actual angles were not included as arccos was out of argument for false interactions. This was done for every 2 combinations of scatter data, giving features $\cos(\phi_{12}), \cos(\phi_{21}), \cos(\phi_{23}), \cos(\phi)_{32}, \cos(\phi_{31}), \cos(\phi_{13})$ based on the energy values used for initial and final values in the Compton Equation [16].

8 Discussion and Conclusions

Subsection 6.3 contains the key results of our studies on the "real patient" data while Subsection 6.4 contains the key comparisons with analogous results on water phantom data. This research found

that the best FCN model on RP data had architecture [256, 256, 256, 256, 256, 128, 128, 128] and achieved a 55% testing accuracy while the best LSTM had architecture 4 LSTM + 128,64 FCL and achieved an accuracy of 55.6%. We also reproduced the best models of [1] and [12] on WP data. The architectures of the best RP data models were notably similar to the best performing models on WP data, but with significantly lower accuracy. Moreover, the gap in accuracies between FCNs and LSTMs was negligible for RP data, while LSTMs had significantly better predictive performance on WP data.

This indicates that the real patient data is (1) generally harder to predict on and (2) best suited for FCNs. Generalizing further, we find evidence that deep neural networks are very sensitive: even small changes in how data is simulated can completely change model performance. Researchers and professionals in industry must keep this in mind as a key part of scalability. These conclusions have the caveat in that there is a possibility that the gap in performance was due to the much smaller number of observations in the RP data, which is not an unlikely possibility.

This work also introduced the BRIDE platform, which addressed several problems in the areas of discontinuity in the Big-data REU Projects, rigorous testing and flexible experimentation, and readable, thoughtful code. We found the utility of this effective integrated development platform cannot be overstated. This is especially true to receive the significant benefits of parallelization, as there are substantial implementation challenges that parallelized learning comes with. These challenges are exemplified by the Validation-Test gap, a validation set data leak occurring when multiple-GPU training was used in PyTorch Lightning, which was exceptionally difficult to discover because of its embedding in the Lightning wrapper and unclear documentation.

Finally, this research explored several avenues in increasing the predictive performance and effectiveness of various machine learning models. Hybrid LSTM + FCL neural networks trained on a larger hybrid water phantom and real patient dataset achieved a testing accuracy of 76%. Novel feature engineering in multiple ways on the datasets were implemented in an attempt to increase accuracy and address overfitting. The effect of the number of GPUs on runtime was also investigated.

Acknowledgments

This work is supported by the grant "REU Site: Online Interdisciplinary Big Data Analytics in Science and Engineering" from the National Science Foundation (grant no. OAC–2348755). Undergraduate assistant co-author Obe acknowledges support from an REU Supplement. Co-authors Sharma and Ren acknowledge support from the NIH. The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, OAC–1726023, and CNS–1920079) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

References

[1] Alina M. Ali, David Lashbrooke, Rodrigo Yepez-Lopez, Sokhna A. York, Carlos A. Barajas, Matthias K. Gobbert, and Jerimy C. Polf. Towards optimal configurations for deep fully connected neural networks to improve image reconstruction in proton radiotherapy. Technical Report HPCF–2021–12, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2021.

[2] Manikandan Arjunan, Sureka Chandra Sekaran, Biplap Sarkar, and Sujatha Manikandan. A homogeneous water-equivalent anthropomorphic phantom for dosimetric verification of radiotherapy plans. *Journal of Medical Physics*, 43(2):100–105, 2018.

[3] Kaelen Baird, Sam Kadel, Brandt Kaufmann, Ruth Obe, Yasmin Soltani, Mostafa Cham, Matthias K. Gobbert, Carlos A. Barajas, Zhuoran Jiang, Vijay R. Sharma, Lei Ren, Stephen W. Peterson, and Jerimy C. Polf. Enhancing real-time imaging for radiotherapy: Leveraging hyperparameter tuning with PyTorch. Technical Report HPCF–2023–12, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2023.

[4] Carlos A. Barajas. *Neural Networks for the Sanitization of Compton Camera Based Prompt Gamma Imaging Data for Proton Radiotherapy.* Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2022.

[5] Carlos A. Barajas, Matthias K. Gobbert, and Jerimy C. Polf. Deep residual fully connected neural network classification of Compton camera based prompt gamma imaging for proton radiotherapy. *Front. Phys.*, 11:903929, 2023.

[6] Carlos A. Barajas, Gerson C. Kroiz, Matthias K. Gobbert, and Jerimy C. Polf. Classification of compton camera based prompt gamma imaging for proton radiotherapy by random forests. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 308–311, 2021.

[7] Rajamanickam Baskar, Kuo Ann Lee, Richard Yeo, and Kheng-Wei Yeoh. Cancer and radiation therapy: Current advances and future directions. *Int. J. Med. Sci.*, 9(3):193–199, 2012.

[8] Tyler J. Bradshaw, Zachary Huemann, Junjie Hu, and Arman Rahmim. A guide to cross-validation for artificial intelligence in medical imaging. *Radiology Artificial Intelligence*, 5(4), 2023.

[9] Jintai Chen, Kuanlun Liao, Yao Wan, Danny Z. Chen, and Jian Wu. Danets: Deep abstract networks for tabular data classification and regression, 2022.

[10] Michael Chen, Julian Hodge, Peter Jin, Ella Protz, and Elizabeth Wong. Bride off-cluster lab notebook, 2024.

[11] Michael Chen, Julian Hodge, Peter Jin, Ella Protz, and Elizabeth Wong. Github repository, 2024.

[12] Joseph Clark, Anaise Gaillard, Justin Koe, Nithya Navarathna, Daniel J. Kelly, Matthias K. Gobbert, Carlos A. Barajas, and Jerimy C. Polf. Sequence-based models for the classification of Compton camera prompt gamma imaging data for proton radiotherapy on the GPU clusters taki and ada. Technical Report HPCF–2022–12, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2022.

[13] Joseph Clark, Anaise Gaillard, Justin Koe, Nithya Navarathna, Daniel J. Kelly, Matthias K. Gobbert, Carlos A. Barajas, and Jerimy C. Polf. Multi-layer recurrent neural networks for the classification of Compton camera based imaging data for proton beam cancer treatment. In *9th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT 2022)*, in press (2022).

[14] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1984.

[15] William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019.

[16] Fernando Hueso-González, Fine Fiedler, Christian Golnik, Thomas Kormoll, Guntram Pausch, Johannes Petzoldt, Katja E. Römer, and Wolfgang Enghardt. Compton camera and prompt gamma ray timing: Two methods for in vivo range assessment in proton therapy. *Front. Oncol.*, 6(80), 2016.

[17] Jonathan R. Hughes and Jason L. Parsons. FLASH radiotherapy: Current knowledge and future insights using proton-beam therapy. *Int. J. Mol. Sci.*, 21(18):6492, 2020.

[18] M. Krebbs. *Deep Learning with Python.* Data Sciences Series. CreateSpace Independent Publishing Platform, 2018.

[19] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. arXiv preprint 2006.15704, 2020.

[20] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[21] Paul Maggi, Steve Peterson, Rajesh Panthi, Dennis Mackin, Hao Yang, Zhong He, Sam Beddar, and Jerimy Polf. Computational model for detector timing effects in Compton-camera based prompt-gamma imaging for proton radiotherapy. *Phys. Med. Biol.*, 65(12), 2020.

[22] Kevin McDonnell, Finbarr Murphy, Barry Sheehan, Leandro Masello, and German Castignani. Deep learning in insurance: Accuracy and model interpretability using TabNet. *Expert Systems with Applications*, 217(1):119543, 2023.

[23] Daniel W. Mundy and Michael G. Herman. An accelerated threshold-based back-projection algorithm for compton camera image reconstruction. *Medical Physics*, 38(1):15–22, 2011.

[24] Nicki Skafte Detlefsen, Jiri Borovec, Justus Schock, Ananya Harsh, Teddy Koker, Luca Di Liello, Daniel Stancl, Changsheng Quan, Maxim Grechkin, and William Falcon. TorchMetrics - Measuring Reproducibility in PyTorch, February 2022.

[25] Ruth Obe, Brandt Kaufmann, Kaelen Baird, Sam Kadel, Yasmin Soltani, Mostafa Cham, Matthias K. Gobbert, Carlos A. Barajas, Zhuoran Jiang, Vijay R. Sharma, Lei Ren, Stephen W. Peterson, and Jerimy C. Polf. Accelerating real-time imaging for radiotherapy: Leveraging multi-GPU training with PyTorch. In *2023 International Conference on Machine Learning and Applications (ICMLA 2023)*, pages 1735–1742, 2023.

[26] Peter S. Pacheco and Matthew Malensek. *An Introduction to Parallel Programming.* Morgan Kaufmann, 2021.

[27] Costanza M. V. Panaino, Ranald I. Mackay, Karen J. Kirkby, and Michael J. Taylor. A new method to reconstruct in 3D the emission position of the prompt gamma rays following proton beam irradiation. *Sci. Rep.*, 9(1):18820, 2019.

[28] Raj Kumar Parajuli, Makoto Sakai, Ramila Parajuli, and Mutsumi Tashiro. Development and applications of Compton camera —– a review. *Sensors*, 22:7374, 2022.

[29] Jerimy C Polf, Stephen Avery, Dennis S Mackin, and Sam Beddar. Imaging of prompt gamma rays emitted during delivery of clinical proton beams with a compton camera: feasibility studies for range verification. *Physics in Medicine & Biology*, 60(18):7085, 2015.

[30] Jerimy C. Polf, Carlos A. Barajas, Stephen W. Peterson, Dennis S. Mackin, Sam Beddar, Lei Ren, and Matthias K. Gobbert. Applications of machine learning to improve the clinical viability of Compton camera based in vivo range verification in proton radiotherapy. *Front. Phys.*, 10:838273, 2022.

[31] Jerimy C. Polf and Katia Parodi. Imaging particle beams for cancer treatment. *Phys. Today*, 68(10):28–33, 2015.

[32] Sebastian Raschka, YuXi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Expert Insight. Packt Publishing, 2022.

[33] Michael J. Smith and James E. Geach. Astronomia ex machina: a history, primer and outlook on neural networks in astronomy. *R. Soc. Open Sci.*, 10:221454, 2023.

[34] Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. From feedforward to recurrent LSTM neural networks for language modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):517–529, 2015.

[35] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2), mar 2020.