Enhancing Real-Time Imaging for Radiotherapy: Leveraging Hyperparameter Tuning with PyTorch

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

Kaelen Baird[1], Sam Kadel[2], Brandt Kaufmann[3], Ruth Obe[4], Yasmin Soltani[5], Mostafa Cham[6], Matthias K. Gobbert[7], Carlos A. Barajas[7], Zhuoran Jiang[8], Vijay R. Sharma[9], Lei Ren[9], Stephen W. Peterson[10], and Jerimy C. Polf[11]

[1]Departments of Computer Science and of Mathematics, Skidmore College, USA
[2]Departments of Computer Science and of Psychology, Mount Holyoke College, USA
[3]Department of Mathematics and Statistics, University of San Francisco, USA
[4]Department of Computer Science, University of Houston—Clear Lake, USA
[5]Department of Biomedical Engineering, University of Houston, USA
[6]Department of Information Systems, University of Maryland, Baltimore County, USA
[7]Department of Mathematics and Statistics, University of Maryland, Baltimore County, USA
[8]Medical Physics Graduate Program, Duke 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]H3D, Inc., USA

Technical Report HPCF–2023–12, hpcf.umbc.edu > Publications

Abstract

Proton beam therapy is an advanced form of cancer radiotherapy that uses high-energy proton beams to deliver precise and targeted radiation to tumors, mitigating unnecessary radiation exposure to surrounding healthy tissues.

Utilizing real-time imaging of prompt gamma rays can enhance the effectiveness of this therapy. Compton cameras are proposed for this purpose, capturing prompt gamma rays emitted by proton beams as they traverse a patient's body. However, the Compton camera's non-zero time resolution results in simultaneous recording of interactions, causing reconstructed images to be noisy and lacking the necessary level of detail to effectively assess proton delivery for the patient.

In an effort to address the challenges posed by the Compton camera's resolution and its impact on image quality, machine learning techniques, such as recurrent neural networks, are employed to classify and refine the generated data. These advanced algorithms can effectively distinguish various interaction types and enhance the captured information, leading to more precise evaluations of proton delivery during the patient's treatment.

To achieve the objectives of enhancing data captured by the Compton camera, a PyTorch model was specifically designed. This decision was driven by PyTorch's flexibility, powerful capabilities in handling sequential data, and enhanced GPU usage, accelerating the model's computations and further optimizing the processing of large-scale data. The model successfully demonstrated faster training performance compared to previous approaches and achieves an overall fair accuracy with so far limited hyperparameter tuning, highlighting its effectiveness in advancing real-time imaging of prompt gamma rays for enhanced evaluation of proton delivery in cancer therapy.

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

1 Introduction

Proton beam therapy is an advanced form of cancer radiotherapy that uses high-energy proton beams to deliver precise and targeted radiation to tumors mitigating unnecessary radiation exposure. Unlike X-ray therapies, which go through the entire body, proton beams release the majority of their energy in a more localized area. This localized release of energy is the *Bragg peak*, which allows more precise radiation delivery. Since proton beam therapy applies more radiation in a smaller radius, it is especially important to know where the beam is in relation to tumors. In order to take full advantage of proton beam therapy and prevent damaging healthy tissue when patients move, clinicians need an efficient technique to image prompt gamma rays in real time.

Utilizing real-time imaging of prompt gamma rays can enhance the effectiveness of this therapy. Compton cameras are proposed for this purpose, capturing prompt gamma rays emitted by proton beams as they traverse a patient's body. However, the Compton camera's non-zero time resolution results in the simultaneous recording of interactions, causing reconstructed images to be noisy and lacking the necessary level of detail to assess proton delivery for the patient effectively. The noise in the Compton camera causes uncertainty about the location of the proton beam when radiating tissues which can cause healthy tissue to be radiated possibly leading to future complications.

To address the challenges posed by the Compton camera's resolution and its impact on image quality, machine learning techniques, such as recurrent and deep neural networks, are employed to classify the prompt gamma event ordering to improve the clarity of the proton beam during treatment. These trained models clean the raw Compton camera data by identifying and removing false data before image reconstruction. These advanced learning algorithms can effectively distinguish various interaction types and enhance the captured information while reducing external noise in the imaging. This type of real-time imaging leads to more precise evaluations of the radiation delivery during the patient's treatment. It ensures the whole tumor gets the appropriate radiation levels for successful treatment.

We design deep and recurrent neural network models in PyTorch to enhance data captured by the Compton camera. The decision to develop these models with the PyTorch library over the commonly used Tensorflow library is driven by PyTorch's flexibility, powerful capabilities in handling sequential data, enhanced GPU usage, and simple Python-like syntax, accelerating the model's computations and further optimizing the processing of large-scale data and allowing for faster development and training of new model types. The model successfully demonstrates speedier training performance than previous approaches and achieves fair accuracy with so far limited hyperparameter tuning. This highlights its effectiveness in advancing real-time imaging of prompt gamma rays for enhanced evaluation of proton delivery in cancer therapy.

The remainder of this report is organized as follows: Section 2 covers proton beam radiation and the background information on imaging, prompt-gamma interactions, and machine learning models. Section 3 covers our translation of the existing models to PyTorch, configuration of distributed data parallel (DDP) training, and hardware and software used for model training. Section 4 covers the results of our hyperparameter study on our deep and recurrent neural networks. Section 5 wraps up our research, concluding our findings, successes, and struggles during our study.

2 Application Background

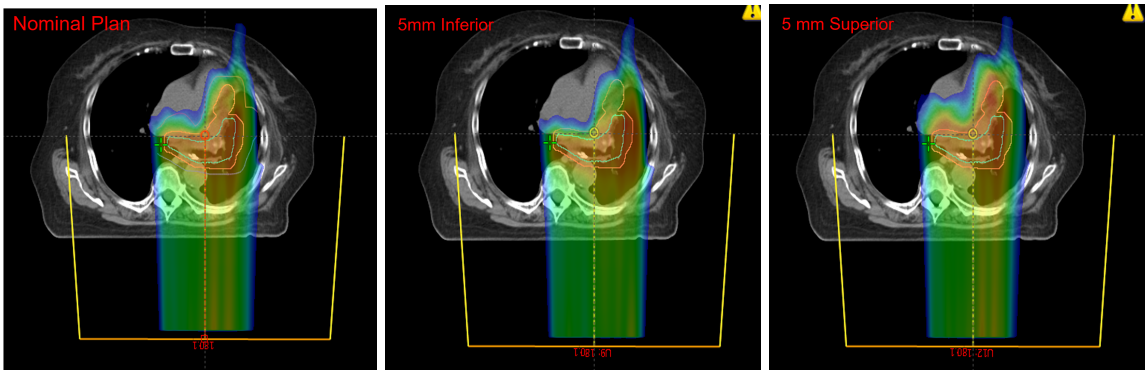
2.1 Proton Beam Therapy

Radiation therapy is an effective approach for treating cancer by utilizing powerful radiation to eradicate cancer cells. X-ray therapy is frequently utilized in cancer treatment. However, a large portion of the radiation is delivered as it enters the body. Unfortunately, radiation therapy often fails to provide a sufficiently concentrated dose to the tumor while providing a similar dose to healthy tissue. Moreover, X-rays pass through the entire body, causing unavoidable radiation exposure. In contrast, proton therapy; another type of radiation treatment; offers enhanced efficiency in addressing these issues [14].

In contrast to X-ray therapy, proton therapy concentrates most of the radiation dosage at the tumor site instead of the point of entry. This precise of targeting significantly improves the treatment's effectiveness. Moreover, proton therapy outperforms X-ray therapy by limiting the penetration of the proton beam to the tumor area, thereby reducing the exposure of surrounding tissues.

The tumor size determines whether a step-by-step elimination of tumor cells using the radiation beam is required. To guarantee that every part of the tumor receives the necessary radiation dosage, healthcare professionals create a safety margin. This margin expands the treatment area and accounts for the patient's slight movements or positional variations during several weeks of treatment, to guarantee that all of the tumor is treated. But this safety margin may encroach on healthy tissue that should not receive treatment. Figure 2.1 visualizes the issue. Figure 2.1a shows the desired outcome of the treatment by a proton beam from the bottom of the scan. Figure 2.1b illustrates the situation of the patient moving slightly upward during the treatment, cause an undershoot of the proton beam from the bottom of the scan. While Figure 2.1c illustrates an overshoot because of the patient moving upward during the treatment.

Having real-time information about the path of the proton beam inside the patient's body during treatment could reduce the safety margin's size and protect healthy tissue better. One proposed solution for obtaining such information is using a Compton camera, which can capture immediate images of prompt gamma rays emitted by the proton beams as they pass through the body. These cameras offer valuable insights into the beam's location, facilitating more precise and efficient treatment.



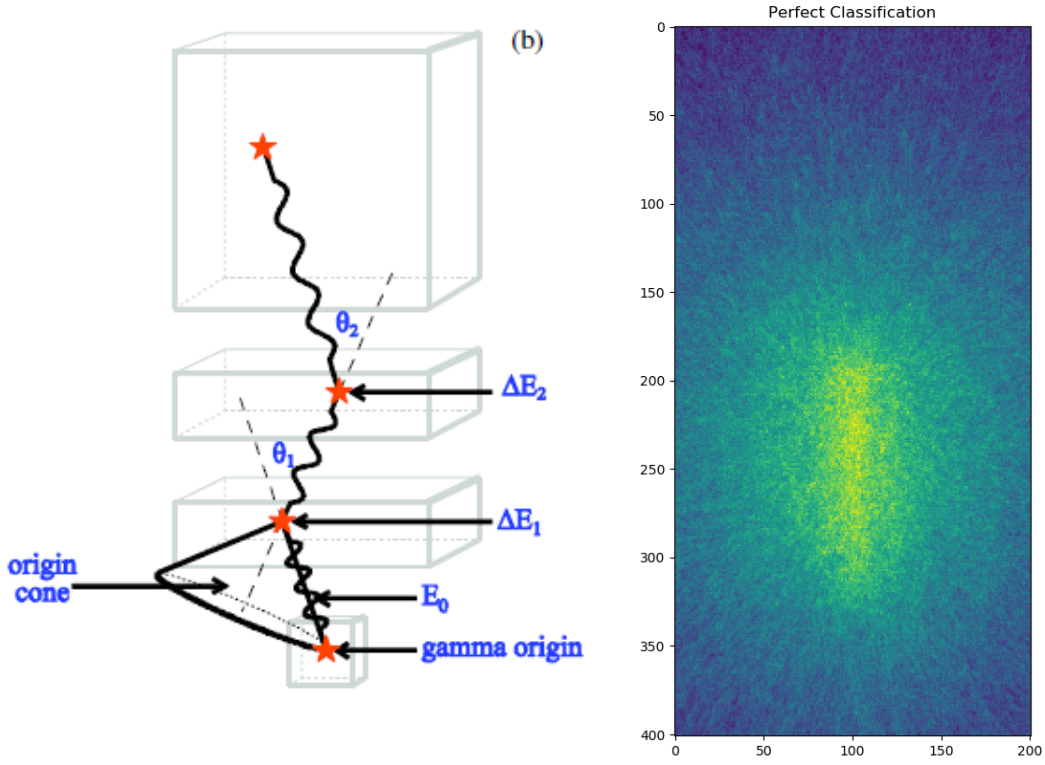
(a) normal (b) undershoot (c) overshoot

Figure 2.1: Distal range uncertainties of beam therapy.

2.2 Compton Camera and Image Reconstruction

Compton cameras are advanced multistage detectors used to image proton beams used in proton beam therapy. When protons pass through the human body, they interact with atoms, resulting in the emission of prompt gamma rays. As these gamma rays exit the body, some of them collide with the modules in the Compton camera. In Figure 2.2 (a), the gamma ray is shown coming from the bottom and bouncing through three modules of the Compton camera. The camera's modules then measure the energy and position of the prompt gamma rays as they traverse different detection stages. Each recorded Compton scatter includes x-, y-, and z-coordinates, as well as the corresponding energy level. These recorded interactions, known as *events*, provide raw output data in the form (e_i, x_i, y_i, z_i), where $i = 1, 2, 3$, and e_i represents the energy level.

Sophisticated algorithms exist for reconstructing the path of the proton beam, as in Figure 2.2 (b), based on the data obtained from the Compton camera. By utilizing the camera's ability to generate complete 3D images of the proton beam's range, it becomes possible to compare the planned treatment dose with the patient's CT scan and make any necessary adjustments. In radiotherapy, it is crucial to ensure conformity between the treatment plan and its execution, ensuring that the patient's bone and soft tissue landmarks are aligned as intended during treatment planning. Even minor movements such as changes in position, fidgeting, scratching, or looking away can disrupt the treatment plan. By obtaining reliable information about the patient's condition from the reconstructed images, clinicians have a better chance of ensuring that the entire tumor receives the precise dose planned while ensuring the safety of surrounding healthy tissues.



(Phys. Med. Biol., 2010.)

(a) (b)

Figure 2.2: Proton beam scatters.

2.3 Scatter Types

When imaging the proton beam, prompt gammas are emitted at speeds close to the speed of light. Since all of the interactions in an event happen almost simultaneously, the Compton camera cannot decode the proper ordering of the interaction in an event. This is where scatter types help to identify false events causing noise in the image. There are 13 classes of scatterings, in three groups by type:

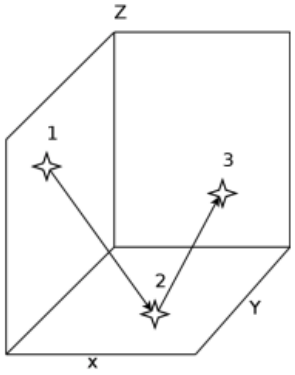
True Triple: The true triples events happen when the camera captures the path of the prompt gamma with a triple interaction in the event. Notice that the ordering of the interactions can be one of six combinations of true triple scattering: 123, 132, 213, 231, 312, 321. As the data is currently implemented, only the 123 ordering of the triple event is usable for image reconstruction. Figure 2.3 (a) and Figure 2.3 (b) are both True Triple events, but only (a) matches the usable ordering subclass.

Double-to-Triples (DtoT): The DtoT events happen when the Compton camera detects an event composed of double and single interaction that are independent of each other, as shown in Figure 2.3 (c). There are six possible ordering of DtoT events: 124, 134, 214, 234, 324, 314. Interaction "4" in the ordering refers to the second prompt gamma interaction in the misdetection events.

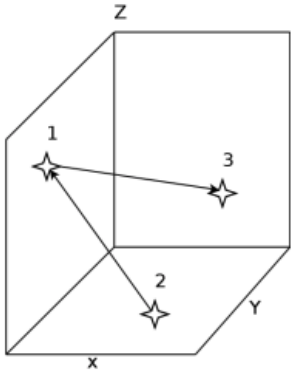
False Triple: The false triple events happen when the Compton camera detects a true triple Figure 2.3 (a) when in reality, the event is composed of three single independent interactions as seen in Figure 2.3 (d).

2.4 Machine Learning in Image Reconstruction

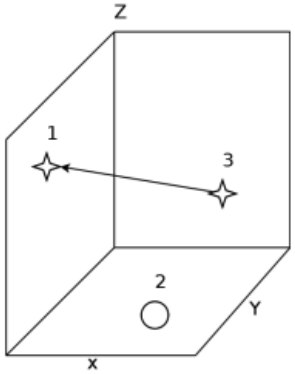
In our case — and as it often is in medical imaging — our data is extremely noisy, and getting significantly more raw data is not cheap or efficient. This is where machine learning comes in. Machine learning has a rich history in image processing and reconstruction. Since 2006, deep learning has been known to have the capability to recognize target objects in images, and since has been used and improved upon. The data obtained from the Compton cameras are extremely noisy, as there are limitations in the imaging provided, primarily due to their inability to gauge the timing of interactions of gamma particles. Because of the noise, the raw images from the cameras are far from the required accuracy for use in proton beam therapy. They must be thoroughly cleaned to be precise enough to be used and trusted when using proton beams. This cleaning process uses machine learning to predict the initial proton beams based on the messy result returned by the Compton cameras.



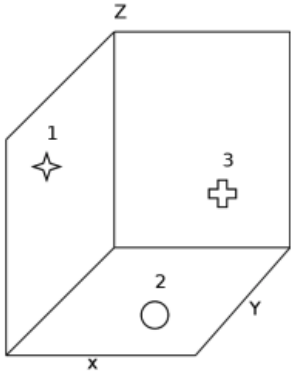
(a) True Triple scatter path of prompt gamma detected by Compton camera.



(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 2.3: Scatter events.

3 Methodology

3.1 Background on Artificial Intelligence/Machine Learning/Neural Networks

The field of artificial intelligence was born in the 1950s with a generation of scientists, mathematicians, and philosophers aiming to answer fundamental questions about intelligence. Turing's work on computation led him to ponder the ability of machines to have intelligence in his 1950 paper, Computing Machinery and Intelligence. The first implementation of artificial intelligence (AI) was presented in 1956 by Allen Newell, Cliff Shaw, and Herbert Simon at the Dartmouth Summer Research Project on Artificial Intelligence (DSRPAI) [3]. This conference spurred the scientific communities interest in intelligence and spread the sentiment that AI was achievable.

In recent decades, with the explosion of big data, AI has caught the attention of the media and general public with some of the advances in machine learning (ML), artificial neural networks (ANN), and deep learning (DL). Even with significant advancements in all these studies, more clarity is needed around the distinction between AI, ML, ANN, and DL, which are different even though they are highly associated.

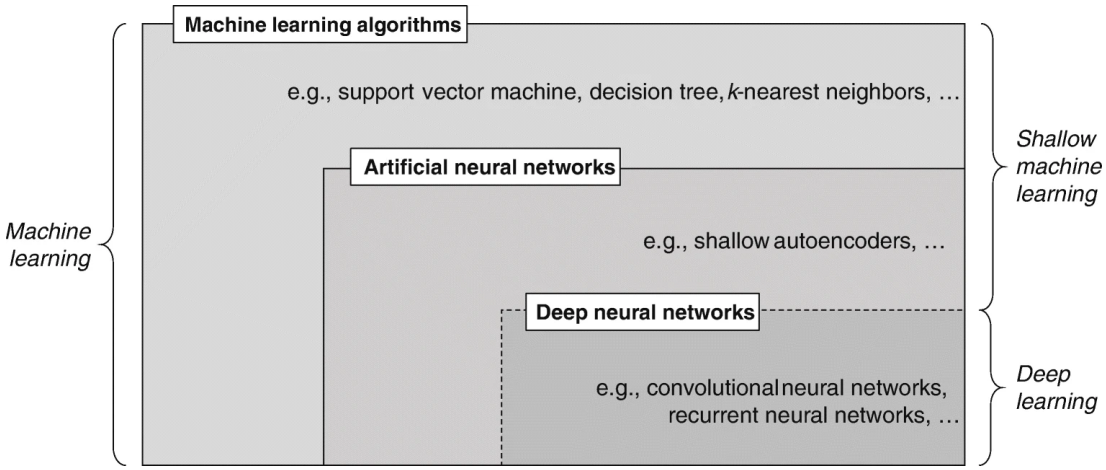


Figure 3.1: Venn diagram of artificial intelligence concepts [9].

Figure 3.1 shows a Venn diagram that shows the interconnectivity between several levels of these concepts. Broadly defined, AI is the field of techniques to allow machines to mimic the behavior and decision-making of humans to automate and solve complex tasks with little to no human interaction. The diagram shows that ML is a sub-field of AI, ANN is a method of ML, and DL is a type of ANN.

Unlike conventional algorithmic approaches, where an algorithm is explicitly developed with specific features in mind, ML is a field that focuses on learning through the development of algorithms to best represent a set of data. Within ML, there are four standard methods for tackling a problem: supervised, unsupervised, semi-supervised, and reinforcement learning [15].

Depending on the learning task, ML has various algorithms to address the different categories of problems. A few of these algorithms include regression models, decision trees, Bayesian methods, and artificial neural networks. ANNs are a network of artificial neurons, loosely modeling the neurons and connections in biological brains, allowing the ML method to have a flexible structure for modeling a wide range of data distributions. Figure 3.2 shows the schematic of a simple ANN with three layers from left to right: the input, hidden, and output layers.

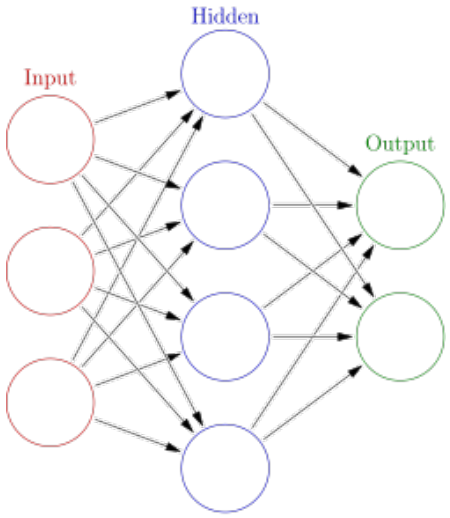


Figure 3.2: Graph of artificial neural network [1].

3.1.1 Layer Types

ANNs are frequently used with many different layer types, among them are Gated Recurrent Units (GRU), Long Short-Term Memory (LSTM), Dense, and Fully Connected Layers(FCL). GRU and LSTM layers were created as a way to combat the short term memory of ANNs. Oftentimes, learning networks would "forget" or not accurately apply earlier data when working through a lengthy dataset. If you passed a long essay, the network may not remember the thesis, as it weighs more recent information heavier. LSTM layers introduce a few key components: the cell state, input gate, forget gate, and output gate. The cell state takes information throughout the entire chain of LSTM cells, and helps to preserve older information. The input gate determines how information from the input is passed through the layer. Either a hyperbolic tangent or sigmoid function is used to determine how much of the input should go to the cell state [13]. The forget gate controls what information is thrown away or kept. A sigmoid function is used to generate values from zero to one, and values closer to 0 are forgotten. finally the output gate takes the values that are remembered, and passes them through the layer. GRU layers have only 2 gates, a update gate and a reset gate. The update gate is responsible for keeping or forgetting information, and the reset gate is responsible for distinguishing what past information should be forgotten.

3.1.2 Deep Neural Networks

A neural network with more than three hidden layers is called a Deep Neural Network (DNN). Deep neural networks employ deep learning techniques to generate a suitable representation of the input data required for a specific task. However, one significant challenge posed by the implementation of these deep neural networks is the diminishment of human interpretability. The decision-making processes of these sophisticated methodologies are primarily non-transparent and obscure, thereby creating a "black box" scenario [5]. Despite the drawback of reduced interpretability in deep learning methodologies, they are crucial when handling vast and high-dimensional datasets. Traditional machine learning techniques can struggle to find meaningful patterns in such complex data due to their limitations in managing multidimensionality and sheer volume. Therefore, the capability of deep learning to model intricate and nuanced data structures makes it indispensable in advancing machine learning applications, despite the trade-off in human interpretability.

3.1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of artificial neural network commonly used in tasks involving sequential data, such as speech recognition, natural language processing, and time series analysis. Unlike traditional neural networks where inputs and outputs are independent of each other, RNNs have connections that allow information to flow in a loop, enabling them to capture and utilize sequential dependencies in the data. The key feature of RNNs is their ability to maintain an internal memory or hidden state that allows them to process inputs in a sequential manner. At each time step, the output from the previous step is fed as input to the current step, allowing the network to retain information about the past and make predictions based on the context of the sequence. RNNs are composed of interconnected layers of artificial neurons, or nodes, with weighted connections between them. These connections allow the network to learn and update its parameters, such as weights and biases, through a process called backpropagation. During training, the network compares its output to the desired output and adjusts its parameters to minimize the error [11, 17]. The specific implementation of the recurrence block distinguishes several types of RNNs. RNNs are a helpful model when classifying Compton camera events because of the ability of the model to encode information about previous events into the evaluation of another event.

Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) are recurrent neural network (RNN) layers that offer advantages in capturing sequential information and handling long-term dependencies. They are widely used in various deep learning models, particularly in natural language processing, time series analysis, and other sequential data tasks. These layers possess gating mechanisms that control the flow of information through the network, enabling them to focus on relevant information and discard irrelevant or redundant information. This feature enhances the model's ability to retain essential information over time. In previous years, the gating mechanisms in GRU and LSTM layers have been used to help prevent overfitting by regulating the flow of information and controlling the network's capacity to memorize training data. This property is particularly beneficial when dealing with large and complex datasets, as it improves the model's generalization capability [6].

3.1.4 Feedforward Neural Networks

Feedforward Neural Networks (FNNs) are a type of artificial neural network where information flows in only one direction, from the input layer through the hidden layers to the output layer. FNNs are also known as multilayer perceptrons (MLPs) and are characterized by the absence of cycles or loops in the network structure. In an FNN, each neuron in a layer is connected to every neuron in the subsequent layer, forming a fully connected layer. This means that the output of each neuron in one layer serves as input to all neurons in the next layer. The connections between neurons are weighted, and each neuron applies an activation function to the weighted sum of its inputs to produce an output [5].

FNNs are powerful models that can approximate any continuous function to arbitrary precision, thanks to the universal approximation theorem. They have been successfully applied in various domains, including image classification, natural language processing, and financial forecasting. Training an FNN involves adjusting the weights and biases of the network using optimization algorithms such as gradient descent to minimize the difference between the predicted outputs and the true outputs [5].

3.1.5 Transformer Network

Transformer neural networks are a machine learning architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies. To capture the long-range dependencies, the network uses a self-attention mechanism to relate the different orderings of the sequence to output an attention representation of the sequence. This mechanism was first proposed in [16] for the purpose of introducing the attention mechanism aimed to solve the vanishing gradient issue encountered in more traditional machine learning techniques that perform gradient-based learning methods and backpropagation through the network. The problem is that with each iteration of training, all of the weights in the network update their values proportional to the partial derivatives of the error function with respect to the current weight. The gradient will vanish, so the weights are effectively prevented from changing. The self-attention mechanism works by representing inputs as an attention vector. For example, given a sentence, the mechanism will focus on how relevant a particular word is with respect to the other words in the sentence. In sequence modeling, the transformer network uses stack self-attention layers and point-wise, fully connected layers to create an encoder and decoder structure. The full structure of a transformer network can be seen in Figure 3.3.

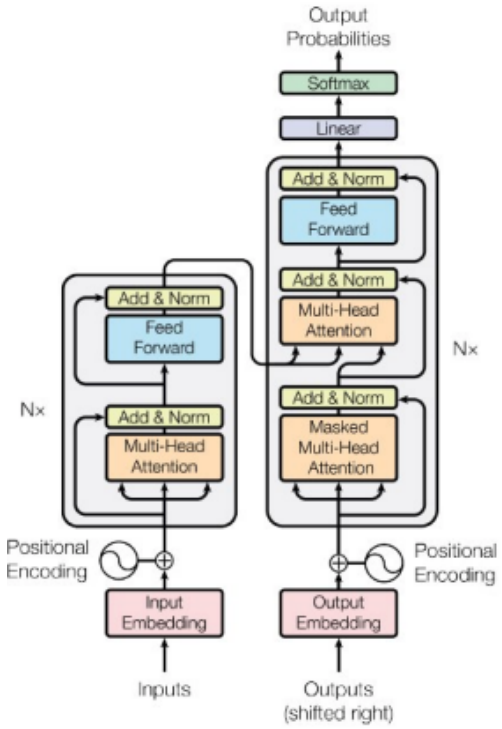


Figure 3.3: Transformer neural network architecture [16].

3.2 Translation to PyTorch

PyTorch and TensorFlow are both powerful, open-source deep learning frameworks, however, there are some general differences that might make one more appealing than the other for certain use cases. Previous works on this project used TensorFlow and Keras as ML library [2,7,8,18]. Our attempt to improve accuracy values includes translating the code base from TensorFlow to PyTorch. There are several reasons for refactoring the legacy code base into PyTorch such as python-like syntax, easy setup of distributed and paralleled training, and easy GPU optimization and configuration. In recent years, the percentage of research papers using PyTorch over TensorFlow has continued to grow reaching 75% of all new data science research papers using the PyTorch library [12], visually represented in Figure 3.4. We created a PyTorch sequential model and normalized the activation of the layers to adjust and scale the inputs using batch normalization. Still focusing on single dense layer neural network, we generated residual blocks to map and capture the difference between the desired output and the input before obtaining the final output.

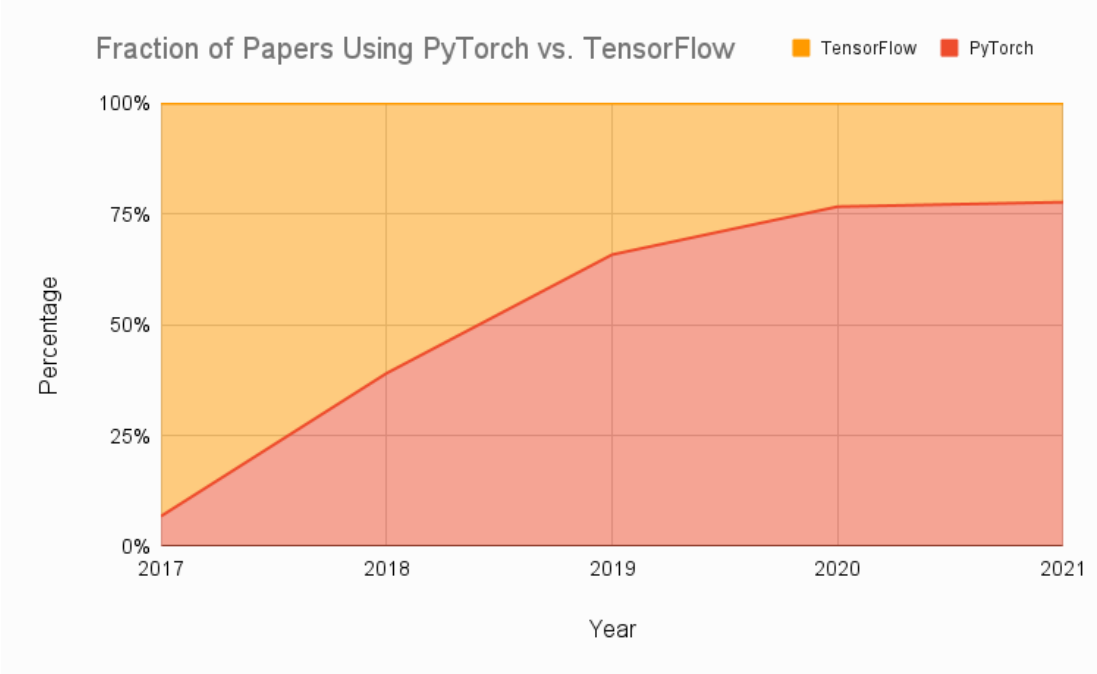


Figure 3.4: PyTorch prevalence over time [12].

We constructed fully connected layers for each residual block with new layers added to the previous layers and specific number of neurons for each layer created. The dropout rate is defaulted to 0 and batch normalization to False. The regularization selects the appropriate loss function of the neural network, which are either nn.L2Loss (Mean Squared Error loss) or nn.L1Loss (Mean Absolute Error loss). The activation function is default to None otherwise it applies the activation function to the layer. The activation function is either LeakyReLU or PReLU. We used PyTorch optim module to initialize the optimizer to Adam, Nadam, or SDG, which uses traditional or Nesterov momentum.

During the process of converting models from Keras to Pytorch, several challenges were encountered due to the differences in layer parameterization and compatibility between the two frameworks. PyTorch's nn.Linear method requires the specification of input and output feature dimensions as integers, representing the dimensions of the input and output spaces, while Keras'

equivalent Dense layer only requires the output space dimension. Additionally, the tensor layers used in PyTorch posed compatibility issues when attempting to pass them into the nn.Sequential() container. Further disparities were observed in terms of layer initialization, with PyTorch employing the default initialization scheme and Keras utilizing Glorot uniform initialization by default. Furthermore, activation functions in PyTorch are instantiated as modules (e.g., nn.ReLU()), while in Keras, they are specified as strings (e.g., 'relu'). Model definition also varies, as PyTorch involves subclassing nn.Module and overriding the forward method, while Keras provides the Sequential or functional API. It is important to note that both frameworks offer a range of loss functions and optimizers; however, syntax and availability may differ between the two. These differences and challenges encountered during the PyTorch to Keras model conversion process underscore the need to carefully account for the dissimilarities in layer parameterization and compatibility between the frameworks. The compatibility issues with the layers being passed out as tensors instead of linear layers kept persisting for several days. To resolve this issue, we opted to replace the original implementation of the model.py code with an alternative approach. While most of the initial challenges were successfully addressed through this code replacement, we subsequently faced new obstacles. One particular issue involved the usage of Distributed Data Parallel (DDP), which caused additional errors. We performed debugging and resolved these issues by identifying and fixing the root causes.

3.2.1 PyTorch Distributed Data Parallel (DDP)

Threading is a standard solution to carrying out parallel tasks allowing tasks to be distributed across multiple threads. A benefit of using PyTorch is implementing a similar technique to train different machine learning models in parallel across multiple GPUs [10]. We use the DataParallel function to perform this DDP to inform PyTorch that the model must be parallelized. By adding this wrapper over our model, PyTorch will automatically use multiprocessing and data slicing to achieve parallelism over multiple GPUs. DDP in PyTorch uses multiprocessing to avoid well-known issues of threading in Python that arise due to the Global Interpreter Lock (GIL). The Global Interpreter Lock only allows one program thread to use the Python Interpreter, locking all of the others. This is the issue with threading since the GIL prevents perfect parallelism [4]. Implementing DDP will allow for faster training and better scaling as the training data size grows. DDP reduces the memory footprint by sharing parameters and ensures consistency across the model during training. Another benefit of implementing DDP into our research is that it provides a fault tolerance mechanism to handle failures during distributed training.

3.2.2 PyTorch DDP Configuration

To setup the distributed data parallelism, we used the torch.distributed package [10] that provides the functionalities for distributed training. To initialize the distributed environment, we used dist.init_process_group() [10]. The backend is specified as 'nccl' [10] which is commonly used for GPU-based distributed training. The dist.get_rank() [10] and dist.get_world_size() [10] methods get the current process rank and the total number of processes, respectively. To get the available port, we used the getPort package for the free port. In the main() method where we created and trained the model, the model is wrapped with DistributedDataParallel, which enables synchronized gradient updates across multiple processes. Each process will operate on its own portion of the dataset. For multiprocessing, we used PyTorch's torch.multiprocessing package to launch the train function across multiple processes. The 'nprocs' argument specifies the number of processes to create, which matches the desired world size for distributed training.

3.2.3 PyTorch Model

In PyTorch, models are not compiled like in the Keras frameworks. Instead, we defined the model architecture using PyTorch's nn.Module class, implemented the forward pass in the forward method of the model, defined the loss function using nn.CrossEntropyLoss() [10], and chose an optimizer using optim.SGD [10] with the model parameters and learning rate. In the training loop, we performed the forward pass, calculated the loss perform backward pass, and updated the model parameters using the optimizer.

The getActivator() method is setup to return the appropriate function based on the name of the activator used, which is the LeakyReLU or the PReLU. The getOptimizer() method returns the PyTorch optimizer based on the provided optimizer name loading the parameters from params.json file.

3.2.4 PyTorch CSVLogger

In PyTorch, there is no direct equivalent to the CSVLogger like in Keras. To save the training process to the CSV files, we created our own custom CSVLogger class, which checks if the training_log.csv already exists or not using os.path.exists(). If it does not exists, it creates an empty file in write mode with the specified filename that logs the desired information in training_log.csv. The TimeHistory() method defines on_epoch_begin and on_epoch_end() methods, which are called at the beginning and end of each epoch, respectively. The information logged into training_log.csv include the epoch time, loss, accuracy, validation loss, and validation accuracy for each epoch.

3.2.5 PyTorch Tensors

The design and implementation of layers differ between Keras and PyTorch. In Keras, the Dense layer is a high-level abstraction that encapsulates the weights, biases, and operations required for a fully connected layer. When creating a Dense layer in Keras, it returns a KerasTensor object, which is a symbolic representation of a tensor in the Keras computational graph. It allows Keras to track the operations and dependencies between layers. However, in PyTorch, the nn.Linear module represents a linear transformation layer. When we created an nn.Linear layer, it returned an instance of the Linear class, which is a subclass of Module. The Linear class provides methods for forward pass computation of the layer and manages weights and biases internally. We used nn.Linear(nNeurons, nNeurons) to create a linear layer with nNeurons input features and nNeurons output features. Since leaky_relu() expects a Tensor type and not Linear type as input, we applied the activation function to the output of the Linear module to ensure that a tensor is passed as input to the activation function. In PyTorch, the input to a neural network layer is typically a torch.Tensor object. The input tensor represents the data that is passed as input to the neural network layer and it is a multidimensional array or tensor containing the input values for a batch of samples.

3.2.6 Training Model Across Single/Multiple GPU(s)

DistributedDataParallel is a PyTorch module used for distributed training across multiple devices or machines [10]. It is designed to work with GPU and CPU modules where model is trained across multiple devices. The DDP wrapper requires that the underlying module supports parallel execution. The device type we train our model on is CUDA, which means our model was on a GPU device. The DDP wrapper takes a device_ids array argument which specifies how many devices to use for parallel training. If the array contains only one element, the training is done on a single

GPU but if the array contains more than one element, the training is performed on multiple GPUs. To ensure that all tensors used by the model are on CUDA devices, we checked that all the model parameters are on CUDA and verified that the model inputs during training are also on CUDA devices.

3.3 Hardware and Software

We used the Graphics Processing Unit (GPU) clusters in the ada system in the UMBC High Performance Computing Facility (hpcf.umbc.edu) for our hyperparameter studies. The ada system has 3 distinct node types: four nodes with 8 NVIDIA RTX 2080 Ti GPUs each with 11 GB GPU memory; seven nodes with 8 NVidia Quadro RTX 6000 GPUs each with 24 GB of GPU memory; two nodes with 8 NVidia Quadro RTX 8000 GPUs each with 48 GB GPU memory. Each node has 384 GB of CPU memory (12×32 GB DDR4 at 2933 MT/s), except the two RTX 8000 nodes, which have 768 GB of CPU memory (12×64 GB DDR4 at 2933 MT/s).

Networks built on ada were built using PyTorch v1.12.1 (https://pytorch.org/). We also used scikit-learn v0.23.dev0 (https://scikit-learn.org/stable/) to preprocess and normalize the data. Moreover pandas v1.1.0 (https://pandas.pydata.org/) and numpy v1.25.1 (www.numpy.org) were also used to help preprocess the data. Finally, we used the matplotlib v3.5.1 (www.matplotlib.org) library to graph our results. Networks built on ada were built inside the python virual environment package Anaconda3 (https://www.anaconda.com/) v4.8.3.

4 Results

In our research, we trained a neural network using a dataset created by a Monte Carlo simulation, containing 1,443,993 records across 16 columns. This dataset includes 13 different categories, with a roughly even number in each. Of the 15 features, two are empty, specifically numbers 6 and 7. These features detail spatial coordinates, distances measured in Euclidean terms, and energy levels for each interaction, with distances given in centimeters (cm). An interaction is made up of three spatial points and an energy reading. Each row of data can be a triple, double-to-triple, or false triple, and contains three of these interactions. The data used for training included only True Triples, Double-to-Triple scatter, and False events.

We separated the data into training and validation parts, using 20% for validation, and reorganized it to make sequential reading easier. Each initial record of 15 features was changed into three interactions, and each of these interactions was made up of five features: three spatial points, distance, and energy. Thus, each record was inputted into the neural network as three sequential interactions.

This year, we encountered a significant change when using the cross-entropy loss function in PyTorch. This function doesn't work with a data format called one-hot encoded labels. So, we didn't use these labels during training. We made this specific change to make sure our training with PyTorch's cross-entropy loss function was efficient and accurate.

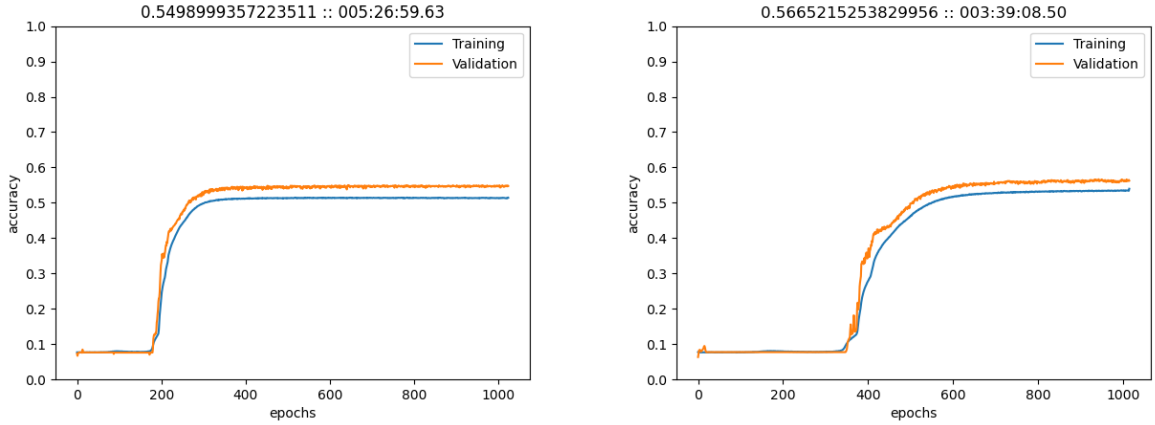
4.1 Inital Testing

During the initial stages of our research, we conducted comprehensive testing on the Keras code provided from the previous year's study. This existing code served as a valuable benchmark, offering insights into the baseline performance for our project. During training, the beginning and end times spent on each epoch was tracked and recorded in time history. The results from the training were

Class	0	1	2	4	5	8	9	10	11	12	13	14
Count	59891	60306	60097	60313	60240	60171	60267	60151	60087	60223	60165	60398

Table 4.1: Class distribution.

stored in a log file called training-log-00.csv file. Figure 4.1 shows the plot of the accuracy from the data generated.



(a) Model trained With 1 GPU

(b) Model trained With 2 GPUs

Figure 4.1: Accuracy of previous model by epoch With timestamp.

In Figure 4.1, the entire training took about 5 hours and 27 minutes to complete on a single gpu ada cluster. About 180 epochs into the training the accuracy was about 10%, but at 200 epochs there was a rapid increase in the accuracy up to about 350 epochs which produced an accuracy of approximately 55%. From this point on, the accuracy appears to be consistent throughout the rest of the training up to 1024 epochs.

The class distribution provides insights into the number of instances or samples belonging to each class in the dataset. In this case, it appears that there is a relatively balanced distribution across the classes, as shown in Table 4.1

4.2 Hyper-parameter Study

Like previous research, we explore various networks for classification using the Dense, LSTM, and GRU layers [2, 7, 8, 18]. We begin by examining the number of epochs, the batch size, and the learning rate. We then explore the number of layers, neurons, and the dropout rate to determine a promising configuration for the network. Table 4.2 shows the constant parameters for all model studies. In order to mitigate overfitting, we then test the model using a clip gradient, batch normalization, and L1 and L2 regularization to improve the model. The varying hyperparameters that are tuned throughout this study can be seen in Table 4.3.

4.2.1 Number of Layers

When starting this study, we observe the parameters used for the dense models runs of previous TensorFlow models [18]. We first investigated the number of layers, starting with 64, 128, and 256 layers. All other parameters were kept constant. See Table 4.2 for these parameters. These runs

Hyper-parameter	*Value*
Indim	15
Outdim	13
Optimizer	Adam
Loss Function	Categorical Crossentropy

Table 4.2: Constant model parameters.

Hyperparameter	*Value*
Layers	256
Neurons	256
Batch Size	8192
Learning Rate	1.0e-3
Train/Validation	0.8/0.2
Dropout	0.45
Inter-activation	leakyrelu
Clip Gradient	0
Layer Type	Dense

Table 4.3: Variable Model Parameters

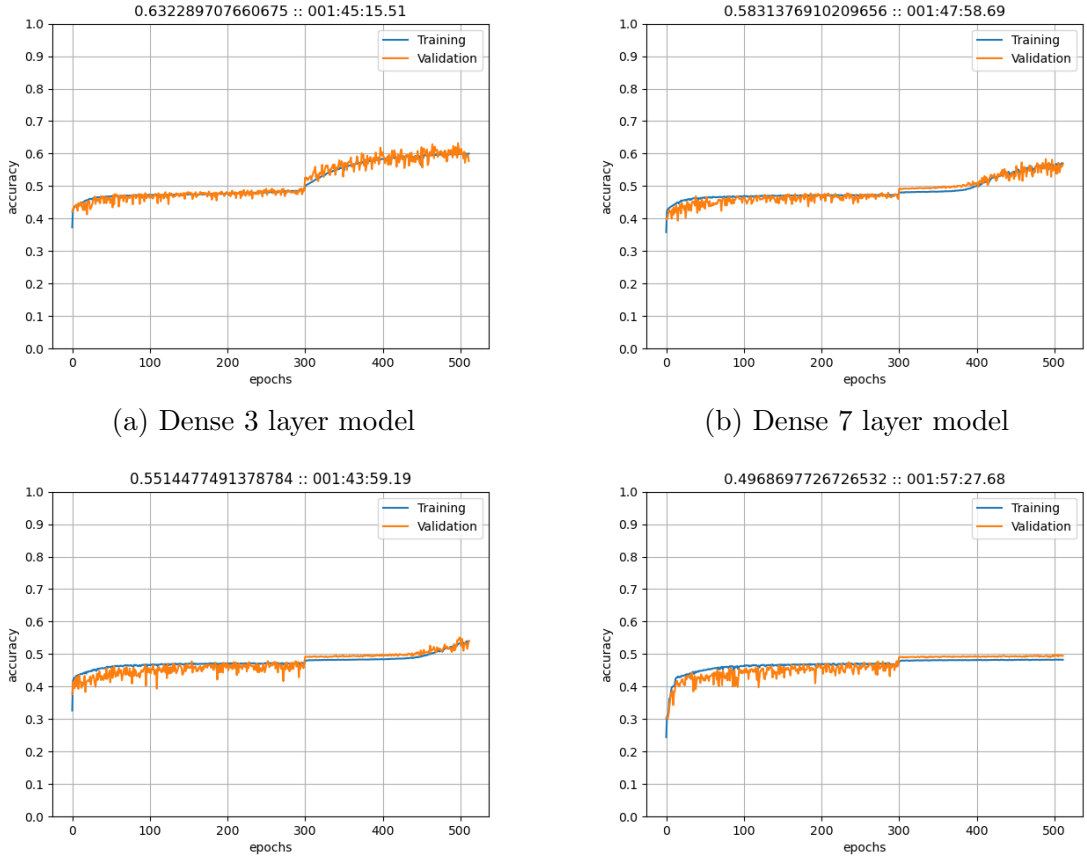
stagnated at roughly 7% accuracy with little to no improvement with increasing epochs. We then tested a model with only 16 layers, which saw a significant increase in accuracy from 7% to 49.7% as shown in Figure 4.2, indicating that fewer layers leads to higher accuracy in our dense model. We then conducted 2048 epoch runs with 1–16 layers. We found that if there were too few layers, the model would become severely over fitted. 11 layers produced the best accuracy while limiting overfitting.

4.2.2 Learning rate

After finding that a lower number of dense layers produced better results, the next step was to adjust our learning rate hyperparameters. We trained 2, 4, 6, and 11 layer dense models with constant hyperparameters defined in Table 4.4. We used a learning rate of 1e-3, which was multiplied by of 1e-1, 3e-1, and 5e-1 every 682 epochs and 341 epochs. From this experiment, we found better results from changing the learning rate every 341 epochs. When we multiplied the learning rate by 1e-1, 3e-1, and 5e-1, we achieved an accuracy of 63.9%, 63.5%, and 59.8% and 5e-1 had an accuracy of 59.8%. Decreasing the learning step from 682 to 341 improve all models independent of learning change by a couple of percentage points, as shown in Figure 4.3.

4.2.3 Train-Validation Split

We trained dense models with 11 layers with train-validation splits of 0.9/0.1, 0.85/0.15, 0.8/0.2, 0.75/0.25, 0.7/0.3. We found that our train-validation split had an insignificant effect on our model as all of the tested models were within 1% accuracy of each other with around 64% accuracy. Table 4.5 shows the various validation split percentages with the corresponding train and validation accuracies.



(a) Dense 3 layer model

(b) Dense 7 layer model

(c) Dense 11 layer mdoel

(d) Dense 16 layer model

Figure 4.2: Accuracy of 3, 7, and 16 dense layer models.

4.2.4 Batch Size

Larger batch sizes can improve model performance by providing more stable weight updates, efficient parallel processing, and better generalization. However, extremely large batch sizes can lead to memory issues and slower convergence.

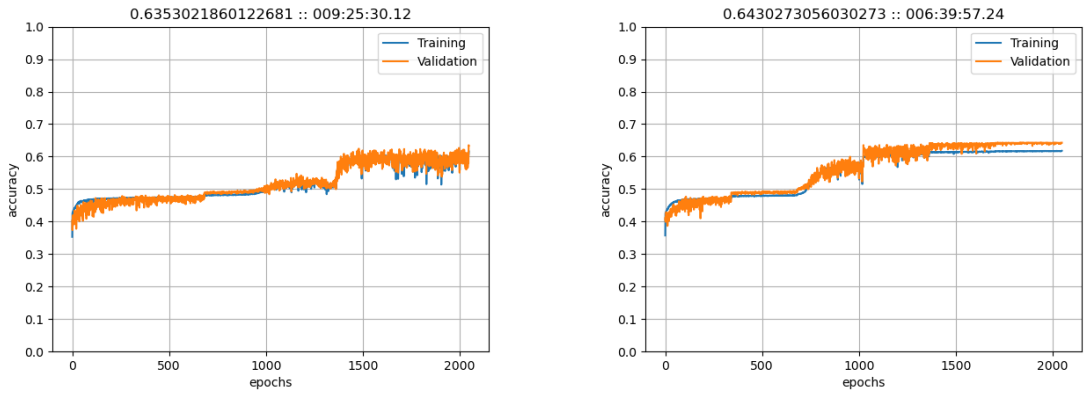
To determine the best batch size for the data we trained models using batch sizes of 8192, 16384, 32768, and 65536 as shown in Table 4.6. Based on our training of these model we concluded that a batch size of 8192 has the best accuracy of any of the models with the least training time. As the batch size increase the training time increased while the accuracy of the model significantly decrease.

4.2.5 Clip Gradient

Clip Gradient has little to no change on peak accuracy, producing results that are less than .0005 different then our preset clip gradient. That being said, one run with 1.0e-4, 10x smaller than our default performed slightly better (+.0004 peak accuracy). Other runs with higher clip gradients–1.0e-3, 5.0e-4, 5.0e-4–performed slightly worse, but only brought down peak accuracy by 0.0006, as shown in Table 4.7. Concluding testing, 1.0e-4 seems to be the best value out of values testing in a similar range, but this is not a very powerful variable in changing peak accuracy.

Hyperparameter	*Value*
Learning Rate	1.0e-3
Train/Validation	0.8/0.2
Batch Size	8192
Neurons	256
Number of Layers	11
Dropout	0.45
Inter-activation	leakyrelu
Clip Gradient	1.0e-3
Layer Type	Dense

Table 4.4: Hyperparameters in learning rate model



(a) (b)

Figure 4.3: Graphs of model accuracy: dense 11 layer model with 3e-1 learning change and (a) 682 and (b) 341 learning steps.

4.2.6 Dropout

Dropout hyper parameterization during training can improve model performance by reducing overfitting. By randomly deactivating neurons, dropout prevents certain neurons from relying too heavily on specific inputs, forcing the network to learn more robust and generalized features. This regularization technique helps the model generalize better to unseen data, enhancing its ability to generalize to new examples during inference. To determine the right dropout rate for Dense layers

Validation Split	*Accuracy*	*Val. Accuracy*
10%	62.25	64.29
15%	62.32	64.54
20%	61.45	64.08
25%	61.80	63.99
30%	61.67	64.02

Table 4.5: Hyperparameters in Train-Validation split model

Batch Size	*Accuracy*	*Val. Accuracy*
8192	61.63	63.96
16384	61.73	63.89
32768	49.87	52.74
65536	48.75	50.32

Table 4.6: Batch size on 11-layer dense model accuracy

Clip Gradient	*Accuracy*	*Val. Accuracy*
5.0e-3	61.44	63.98
5.0e-4	61.41	64.10
1.0e-4	61.77	63.98

Table 4.7: Clip gradient on 11-layer dense model accuracy

in the model we trained models with constant parameters defined in Table 4.2 and variable dropout rates of 15, 20, 30, 40, 60% dropout. As demonstrated in Table 4.8, we found that lowering the dropout percentage increase the accuracy of our model with the best performing training being the model with 15% dropout rate seen in Figure 4.4. In the GRU layers, we observed that a dropout rate of 10% led to notable performance, achieving an accuracy of approximately 43%. However, when the dropout rate was increased to 60%, the model's performance deteriorated significantly, with accuracy remaining below 7%. This could be due to their simple architecture: When the dropout rate is set too high, it might cause the GRU to lose too much information during training, leading to poor performance. On the other hand, the LSTM layers showed an intriguing behavior. The training performance remained unchanged between a 10% dropout rate and a 70% dropout rate. It appears that LSTM's architecture is better suited to handle higher dropout rates, and thus, the performance remains stable even with a 70% dropout rate.

4.2.7 Layer Type

Besides fully connected layers, in our experiment, we employed LSTM and GRU layers for our classification task. In PyTorch, both GRU and LSTM layers are provided as part of the torch.nn module, specifically as nn.GRU and nn.LSTM. PyTorch provides the num_layers parameter for these modules, allowing users to specify the number of stacked layers within the model architecture. During our RNN training utilizing LSTM layers, we conducted evaluations across models with 2,

Dropout	*Accuracy*	*Val. Accuracy*
60%	59.27	62.76
55%	60.02	63.17
40%	62.04	64.31
30%	63.29	65.17
20%	64.81	66.37
15%	65.28	66.85

Table 4.8: Dropout rate on accuracy in 11-layer dense model

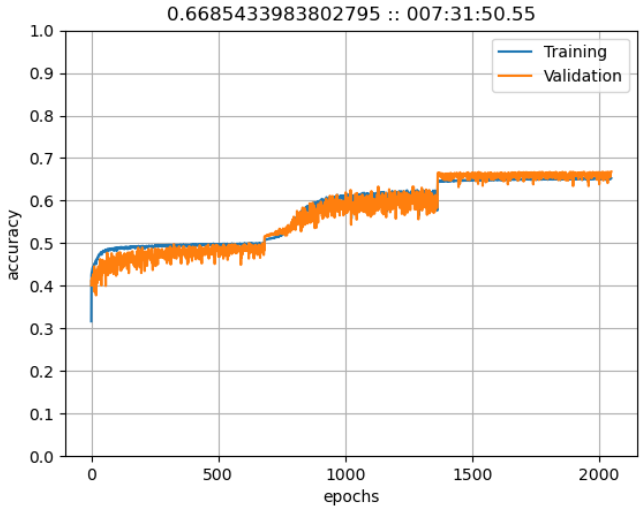


Figure 4.4: Accuracy graph of dense 11 layer model with 15% dropout rate.

4, 16, and 128 LSTM layers. The models with 16 and 128 LSTM layers underperformed, yielding accuracies not exceeding 7.73%. With 4 LSTM layers, the model achieved accuracies of 66.5%, 63.7%, and 56.4% under L2 regularization values of 1e-2 and 1e-3. In contrast, the 2 LSTM layer configuration resulted in accuracies of 56.6%, 54.1%, 63.7%, and 63.3% with L2 regularization values of 1e-2 and 1e-4.

Furthermore, we assessed the performance of our model using GRU layers, specifically with 2, 4, and 6 layers. The 6 GRU layer configuration, when paired with 16 and 64 dense layers, yielded accuracies of 7.7% and 7.6% respectively. The 4 GRU layer model achieved a 55.1% accuracy with 16 dense layers, but a significantly lower 8.4% accuracy with 64 dense layers. As shown in Figure 4.5, the most optimal performance was observed with the 2 GRU layer model, which registered accuracies of 69.08% when combined with 10 fully connected layers.

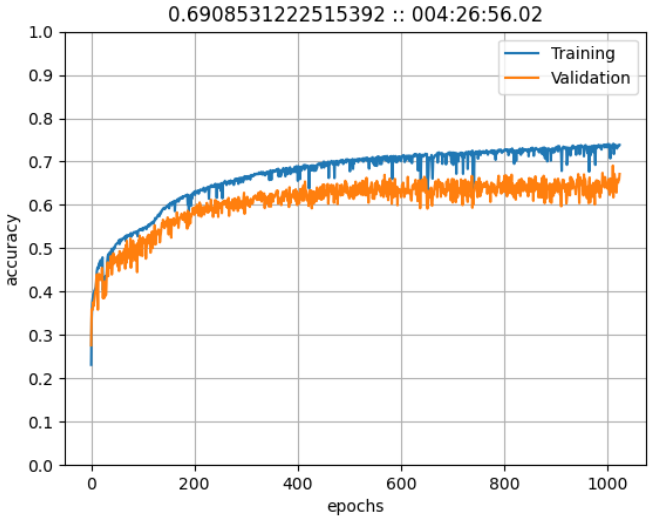


Figure 4.5: Accuracy graph of our 2 GRU + 10 FCL model

4.3 Distributed Data Parallelization

In our experimentation process, we aimed to leverage PyTorch's Distributed Data Parallel (DDP) to enhance the training efficiency of our models across multiple GPUs. DDP, a model parallelism technique provided by PyTorch, promises to distribute input data across various GPUs, allowing each to process a unique subset of the input data. This approach replicates the model on each processor, with every model processing a different partition of the input data. After the forward and backward passes, DDP is designed to synchronize the gradients across all GPUs and update the model. However, despite our efforts, we encountered challenges in integrating DDP into our training pipeline. These challenges encompassed a variety of issues, starting with configuration difficulties where aligning the DDP settings with our existing infrastructure proved more intricate than anticipated. Additionally, we faced synchronization bottlenecks, particularly when attempting to consolidate gradients across the GPUs. This not only hindered the expected performance improvements but also occasionally led to inconsistent model states across the devices. Such inconsistencies risked the integrity of our training process. Furthermore, we encountered complexities related to memory management and data distribution, which added layers of troubleshooting that diverted our focus from the primary objectives. Despite the promise that DDP holds in terms of distributed training and potential speed-ups, these challenges compelled us to revert to single-GPU training for the scope of this report. However, we remain optimistic about revisiting and harnessing the power of DDP in future iterations of our work, armed with the insights gained from this experience.

5 Conclusions and Future Work

Our research focuses on the methodology of defining and training DNNs in PyTorch, which provides the advantage of dynamic computational graphs. This dynamic nature allows for greater flexibility in model development and debugging. Unlike TensorFlow, which uses static computational graphs, PyTorch constructs the computational graph on-the-fly during runtime. This dynamic approach enables easier modifications to the model and supports dynamic control flow. In contrast, TensorFlow requires the entire graph structure to be defined before execution. The dynamic computational graph feature in PyTorch enhances its flexibility and ease of use in model development and debugging compared to TensorFlow's static computational graph approach.

In this study, we explored various networks, including Dense, LSTM, and GRU layers, for classification to optimize the performance of recurrent neural networks (RNNs). To facilitate our experiments and monitor the training process effectively, we implemented custom utilities in PyTorch. One of these utilities is the custom CSVLogger class, which allowed us to log essential training details, including epoch number, loss, and accuracy. Additionally, we used the TimeHistory() method to measure the duration of each epoch during the training process. By recording the epoch times in the TimeHistory object, we gained valuable insights into the training progress and performance.

To determine the optimal network configurations, we conducted an extensive hyperparameter study, considering factors such as the number of epochs, batch size, learning rate, layer sizes, learning change, learning step, train-validation split, clip gradient, and dropout rate. We first explored the impact of the number of layers in the Dense models. Initial tests with higher layer counts (64, 128, and 256) resulted in stagnant accuracy of around 7%. However, a model with 16 layers exhibited a notable increase in accuracy, and 11 layers were found to be optimal, achieving the best accuracy while effectively preventing overfitting. The examination of learning change and learning step hyperparameters revealed that a learning change of 1e-1 resulted in the most accurate

model with 63.9% accuracy. The choice of train-validation split ratios had minimal impact on model performance, with all tested models achieving similar accuracies of around 64%. Additionally, we discovered that a batch size of 8192 yielded the highest accuracy while minimizing training time, making it the favored option for our model. The influence of clip gradient on peak accuracy was marginal, with 1.0e-4 displaying the most favorable outcomes among the tested values in a similar range. Regarding dropout rates, we observed that lower percentages led to improved accuracy, with the best model achieving a dropout rate of 15%.

Moreover, we evaluated models with varying LSTM and GRU layers. Models with 16 and 128 LSTM layers showed limited performance, with accuracies below 7.73%. However, the 4-layer LSTM model achieved up to 66.5% accuracy, while the 2-layer configuration reached 63.7%. For GRU layers, the 2-layer model combined with 10 fully connected layers stood out, achieving an accuracy of 69.08%. In contrast, configurations with more GRU layers, especially with 64 dense layers, generally underperformed.

Furthermore, we aimed to utilize PyTorch's Distributed Data Parallel (DDP) for efficient multi-GPU training. While DDP offers the potential for enhanced parallelism by distributing data across GPUs, we faced challenges in its integration. Issues ranged from configuration difficulties to synchronization problems and memory management complexities. As a result, we opted for single-GPU training in this report. Nevertheless, our experience with DDP has provided valuable insights, and we are hopeful about its potential benefits in future work.

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–2050943). Co-author Cham additionally acknowledges support as HPCF RA. Co-author Polf acknowledges 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] By Glosser.ca - Own work, Derivative of File:Artificial neural network.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24913461.

[2] 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.

[3] Rockwell Anyoha. The history of artificial intelligence, 2017. It can be found at the URL https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/.

[4] Jason Brownlee. Threadpool and the global interpreter lock, 2021. It can be found at the URL https://superfastpython.com/threadpool-gil/.

[5] Rene Y. Choi, Aaron S. Coyner, Jayashree Kalpathy-Cramer, Michael F. Chiang, and J. Peter Campbell. Introduction to Machine Learning, Neural Networks, and Deep Learning. *Translational Vision Science & Technology*, 9(2):14–14, 02 2020.

[6] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[7] 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.

[8] 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).

[9] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electron Markets*, 31:685–695, 2021.

[10] Shen Li. Getting started with distributed data parallel. It can be found at the URL https://pytorch.org/tutorials/intermediate/ddp_tutorial.html.

[11] Javaid Nabi. Recurrent neural networks (rnns), 2019.

[12] Ryan O'Connor. Pytorch vs tensorflow in 2023, 2023. it can be found at the URL https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/.

[13] Michael Phi. Illustrated guide to lstm's and gru's: A step by step explanation, 2018.

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

[15] Ravish Raj. Supervised, unsupervised and semi-supervised learning with real-life usecase. URL can be found at https://www.enjoyalgorithms.com/blogs/supervised-unsupervised-and-semisupervised-learning.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[17] Chi-Feng Wang. The vanishing gradient problem: The problem, its causes, its significance, and its solutions. It can be found at the URL https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.

[18] Sokhna A. York, Alina M. Ali, David C. Lashbrooke Jr, Rodrigo Yepez-Lopez, Carlos A. Barajas, Matthias K. Gobbert, and Jerimy C. Polf. Promising hyperparameter configurations for deep fully connected neural networks to improve image reconstruction in proton radiotherapy. In *2021 IEEE International Conference on Big Data (Big Data 2021)*, pages 5648–5657, 2021.