DEVELOPMENT OF NEURAL THERMAL SCATTERING (NeTS) MODULES FOR REACTOR MULTI-PHYSICS SIMULATIONS

C. A. Manring*, A. I. Hawari

Department of Nuclear Engineering, North Carolina State University, Raleigh, NC, USA 27695

camanrin@ncsu.edu, aihawari@ncsu.edu

ABSTRACT

Modern multi-physics codes, often employed in the simulation and development of thermal nuclear systems, depend heavily on thermal neutron interaction data to determine the space-time distribution of fission events. Therefore, the computationally expensive analysis of such systems motivates the advancement of thermal scattering law (TSL) data delivery methods. Despite considerable improvements on past strategies, current implementations are limited by trade-offs between speed, accuracy, and memory allocation. Furthermore, many of these implementations are not easily adaptable to additional input parameters (e.g., temperature), relying instead on various interpolation schemes. In this work, a novel approach to this problem is demonstrated with a neural network trained on beryllium oxide thermal scattering data generated by the *FLASSH* nuclear data code of the Low Energy Interaction Physics (LEIP) group at North Carolina State University. Using open-source deep learning libraries, this approach maps a unique functional form to the $S(\alpha, \beta, T)$ probability distribution function, providing a continuous representation of the TSL across the input phase space. For a given material, the result is a highly accurate, neural thermal scattering (NeTS) module that enables rapid sampling and execution with minimal memory requirements. Moreover, extension of the NeTS phase space to other parameters of interest (e.g., pressure, radiation damage) is highly possible. Consequently, NeTS modules for different materials under various conditions can be stored together in material “lockers” and accessed on-the-fly to generate problem specific cross-sections.

KEYWORDS: neutron, thermal scattering law, *FLASSH*, deep learning, NeTS

1. INTRODUCTION

The operation of nuclear systems (e.g., thermal reactors, critical assemblies) is ultimately characterized by a time and space-dependent population of neutrons distributed in energy and momentum. As neutrons lose energy (i.e., thermalize), their wavelengths approach the interatomic distances of their environment. In this thermal energy regime, the neutron scattering process is heavily dependent on the characteristic motion of scattering centers (i.e., nuclei) in the given medium. These material-dependent dynamics, which can be described by the probability distribution function (PDF), $S(\alpha, \beta)$ (also known as the dynamic structure factor or thermal scattering law (TSL)), alter and propagate the temporal scattering signal (i.e., neutron flux) in such a manner as to greatly influence fission.

High-fidelity multi-physics simulations often incorporate this information via a stochastic sampling process to determine the interaction outcome for each neutron in the system at a given timestep. Ideally, the

dynamical complexity described in the TSL would be represented by a relatively simple, analytical function for each material of interest. While this would be considerably beneficial, it is unfortunately not feasible in practice. However, arguably the next best approach is the use of appropriately sophisticated regression techniques. Historically, regression approaches have taken many forms (e.g., linear, logistic, polynomial, support vector machines, random forests, neural networks (NNs)) and have been applied in an equally impressive number of research and industrial endeavors. Using these techniques, it is possible to obtain a useful interpretation of an arbitrary, multivariate, continuous dataset. The use of a particular technique is often dictated by the inherent complexity (i.e., degree of linearity, number of dependent variables) of the dataset in question.

The TSL dataset is uniquely complex in both its nonlinearities and its natural dependencies (e.g., energy exchange, momentum exchange, temperature, pressure, radiation damage, porosity, composition). Thus, developing an accurate approximation of such a distribution relies on a similarly flexible regression approach. While many of the aforementioned strategies are not practically suitable to the task, NNs have proven themselves capable in similar applications [1]. Using this technique, the work discussed herein demonstrates a move beyond discrete TSL evaluations (often provided for only a handful of conditions, e.g., temperatures) and toward continuous evaluations, provided over a full range of relevant conditions. In this way, criticality and multi-physics code practitioners can access any subdomain of the distribution hypersurface and rapidly obtain an accurate, interpolation-free prediction of the TSL, all encompassed in a memory efficient module.

The data used to train the NeTS module was taken from a series of incoherent inelastic TSL evaluations for beryllium in beryllium oxide using advanced atomistic modeling techniques [2] and the Full Law Analysis Scattering System Hub (*FLASSH*) nuclear data code [3] currently under development in the Low Energy Interaction Physics (LEIP) group at the North Carolina State University (NCSU) Department of Nuclear Engineering.

2. METHODS

2.1. Overview of Artificial Neural Networks

As the name implies, artificial neural networks (ANNs) were originally abstracted from the network of neurons in living beings [4]. Due to their inherent ability to provide complex, nonlinear links/mappings/encodings relating various inputs and outputs, ANNs, specifically deep neural networks (DNNs), have become the most successful machine learning (ML) technique in addressing some of the most challenging applications (e.g., facial recognition, natural language processing (NLP)) [5]. Moreover, practitioners and researchers in the field have adopted and developed a host of different frameworks (e.g., TensorFlow, PyTorch, Caffe, Theano, Keras, MXNET) with which they can address these problems. Many of these platforms are open source, allowing for a large degree of collaboration among users. In addition to an ample set of tools, three architectures have become mainstays in the DL community: basic feedforward neural networks (FNNs) or multilayer perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). While FNNs are useful in a variety of classification and regression tasks, CNNs and RNNs are generally more adept at addressing problems with grid-like (e.g., images) and sequential (e.g., time series) inputs, respectively. Given the success of these architectures, many believe that the problem of supervised learning is largely solved. Many advanced architectures both in and outside of supervised learning are also being developed and utilized in the fields of gaming and robotics, among others. Some of these architectures include generative adversarial networks (GANs), variational autoencoders (VAEs), and deep residual networks (ResNets) [6-8]. Beyond these examples, there are many more architectures to choose from, and the number is growing rapidly.

2.2. Key Components

ANNs, in a most basic sense, rely on a series of connected layers (starting with inputs, X), each containing a set of individual neurons with associated weight (W) and bias (b) matrices:

$$output = X \cdot W + b. \quad (1)$$

Special “activation” functions are applied to the neuron outputs to provide the nonlinear mapping capabilities that help distinguish ANNs from other data fitting techniques. There are many different types of activation functions to choose from: rectified linear unit (ReLU), sigmoid, tanh (Eq. 2), softplus, softmax, etc., and each has its own properties and relevant use cases.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2)$$

Between the input and output layers of the network (Fig. 1), some number of hidden layers are added based on the desired fitting capacity. The number of neurons in the input and output layers depends on the number of input features and output type, respectively, while the number of neurons in the hidden layers is again related to model capacity. More advanced architectures combine these foundational elements, along with other features (e.g., recurrent loops, convolution layers, dropout layers, pooling layers, skip connections), to solve a wide range of problems.

Figure 1. Representative FNN in this work with three (or more) inputs, one output, and numerous hidden layers/neurons.

Once the model architecture is selected and the data is ready for training, the optimization algorithm (optimizer) and objective (loss) function are chosen. The optimizer, which is used to update the weights

and biases during training, is a very important factor in the model's performance. Selecting a proper loss function is also very important to model performance. The loss function serves as the judging metric during training and is generally minimized to obtain acceptable results. Among many others, mean squared error (MSE, Eq. 3) is useful for most regression tasks, where y_i , $\hat{y}_i$, and N represent the prediction, label (or "true" value), and number of samples, respectively.

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (3)$$

2.3. Using Neural Networks as Universal Function Approximators

One of the earliest and most intuitive applications for NNs came in the approximation of mathematical functions. The universal approximation theorem states that a NN with a single hidden layer comprised of a finite number of neurons with an appropriate activation function can approximate a continuous function on a subset of n -dimensional real coordinate space (i.e., $\mathbf{R}^n$) to arbitrary precision [9]. Two of the more popular activation functions used for this purpose are the sigmoid and tanh functions. A simple way to conceptualize the idea of approximating functions with NNs is the use of unit step functions with learned weights. Clearly, adding more step functions would increase the accuracy of the approximation. However, an optimal approach to this type of problem involves using a more complex set/arrangement of "basis" functions, and this is what is done in practice.

3. IMPLEMENTATION

3.1. General Approach

Among other reasons (e.g., dynamic model execution, Pythonic syntax), the PyTorch DL framework (version 1.0.1) [10] was selected for its ease of use and inherent interoperability with other useful Python libraries (i.e., Matplotlib, Pandas, NumPy, SciPy, Scikit-Learn). This decision significantly reduced the time spent on processing and manipulating the training data. In the approach to solution, a path of increasing complexity was put forth (Fig. 2) to aid in both the development and feasibility evaluation processes. The main goal of this approach is to minimize the complexity of the design process by gradually adjusting the network to solve successively more challenging regression problems.

Figure 2. Neural network function approximation development process.

Trial datasets were generated from quadratic functions (selected for their simplicity and relation to the Gaussian distribution (exponent), which can be a rough approximation of the TSL form) of one to three input variables over small, arbitrary subsets (i.e., [1, 3], [2, 4], and [3, 5] for x, y, and z, respectively). The subspace was then divided into $\sim 50 - 100$ points per input feature and scaled to fall in the range $[-1, 1]$ for training. After training, the trial networks, which contained only two hidden layers and less than 50 hidden neurons, were evaluated for accuracy using a separate, test dataset. Within $\sim 1 \times 10^4 - 1 \times 10^5$ training epochs, very high accuracy (i.e., low percent deviation, Eq. 4) levels were attained (i.e., $< 0.01\%$ maximum difference from the test data).

$$\text{PercentDeviation}_i = 100 \times \frac{|y_i - \hat{y}|}{\hat{y}} \quad (4)$$

Many permutations of the trial quadratic functions were also tested, and the robust shift and scale invariance characteristic of many NNs was aptly demonstrated. Representative results for the 1-D and 2-D cases (Fig. 3) reflect the aforementioned accuracy levels, providing the necessary confidence to proceed to TSL network development.

Figure 3. Fitting quadratic functions with one (left, clockwise from top left) and two (right, clockwise from top left) inputs.

3.2. Data Generation and Training Techniques

TSL data for beryllium in beryllium oxide was generated over a temperature range spanning 293 – 1293 K on a 200×550 point α/β grid using a custom Python script in conjunction with the *FLASH* nuclear data code. The total dataset contained $\sim 1 \times 10^8$ points and was reformatted, transformed (to log space), shifted, scaled, and divided (using another custom Python script) into training (70%), validation (15%), and test sets (15%). During the development cycle, many network iterations were constructed using up to five hidden layers and up to 500 neurons per hidden layer, and pre-trained networks/layers and layer freezing were utilized extensively to minimize the overall training time through a process known as transfer learning. The Adam [11] optimizer (with default PyTorch parameters) was used in batch mode with a learning rate decay scheme that reduced the learning rate by a constant factor (e.g., 0.5-0.9) when an arbitrary metric (e.g., loss, mean percent deviation) reached a plateau during training. The MSE loss function (Eq. 3) was also implemented, sometimes with L2 weight decay for its regularization effects. While the hyperbolic

tangent activation function (Eq. 2) was used in most cases, other activation functions (e.g., ReLU) often resulted in similar levels of performance.

While the inherent uncertainty in the TSL data may be $\sim 1\text{--}2\%$, the *FLASSH* output distribution is interpreted (in the scope of this work) as a smooth, continuous function without uncertainty. Thus, it is important to recognize this implicit zero noise assumption. While it eliminates the chance of overfitting to noisy individual data points, it does not preclude the possibility of excess capacity (overfitting) between points. To avoid this, the validation and test sets were used to check for such behavior. Another approach, using a very fine training grid, was also explored where the goal is to reach the interpolation limit between points, thus removing any spurious features reminiscent of overfitting.

3.3. Computational Details

During the development of the 1-D and 2-D trial networks, model training was performed on a desktop PC with an Intel Core i7-6700K CPU @ 4GHz and 24 GB of RAM. However, for the 3-D trial and all TSL networks, this hardware proved insufficient (i.e., prohibitively long training times), and training was transferred to a LEIP cluster node containing two Nvidia P100 GPUs. On the cluster, the models were allowed to train up to $\sim 1 \times 10^5$ epochs.

4. RESULTS

Various networks were evaluated (Table I) over the test set for select 1-D, 2-D, and 3-D cases (Fig. 4-5). In each case, very good agreement is observed over the input domain, particularly throughout the more feature-rich regions of β space. While the variations in α and T were relatively smooth, the opposite was true in β , owing to the complex distribution of beryllium energy states in beryllium oxide. Such complexity required a considerable number of training iterations to represent accurately. Percent deviation (Eq. 4) was used to quantify the difference between the TSL values predicted by the network and the “true” labeled data. The increasing complexity of the regression tasks is also reflected in the table values. Memory size requirements for the PyTorch models ranged between $\sim 100\text{--}700$ kB, providing a good estimate of necessary memory allocation in multi-physics applications. In Fig. 5 (right), the trivariate, interpolation-free TSL is displayed for the lowest and highest temperatures (spanning ~ 1000 K), demonstrating accuracy over a wide range of operating conditions.

Table I. Percent deviation between test grid labels and predictions

Case	Mean Dev. (%)	Max Dev. (%)
1-D TSL (α)	0.0062	0.0187
1-D TSL (β)	0.0897	0.7886
2-D TSL (α, β)	0.0825	0.8863
3-D TSL (α, β, T)	0.2061	4.4204

Figure 4. Representative 1-D (left, TSL v. α for arbitrary β/T , and right, TSL v. β for arbitrary α/T) model predictions (red line) for Be in BeO against a test evaluation grid (black dots).

Figure 5. Representative 2-D (left, TSL v. α, β for arbitrary T against a test evaluation grid (black dots)) and 3-D (right, for 293 K (bottom surface) and 1293 K (top surface)) model predictions for Be in BeO.

5. CONCLUSIONS

An accurate, memory efficient representation of a trivariate TSL has been created using an FNN. This first-of-a-kind demonstration of the utility of DL techniques in thermal neutron scattering data analysis presents new possibilities to the reactor multi-physics community and improves significantly on current TSL representation strategies. In addition to providing a range of TSL data over numerous dependent parameters, NeTS modules allow for the investigation of temporal phenomena (e.g., fuel burnup) effects on thermalization, which is not possible with current “static” TSL evaluations. Furthermore, NeTS modules would likely prove beneficial in various transient analysis scenarios where the material/reactor conditions should be closely reflected in the sampled TSL evaluations. Work to apply this approach to reactor materials

of high interest is currently underway, and further improvements in accuracy and memory size using pruning techniques and quantization are also under development.

ACKNOWLEDGMENTS

This work was partially supported by funding from the US Department of Energy (Office of Nuclear Energy) through the Nuclear Energy University Program (NEUP).

REFERENCES

1. G. Huang et al., "Trends in Extreme Learning Machines: A Review," *Neural Networks*, **61**, (2015).
2. A.I. Hawari, "Modern Techniques for Inelastic Thermal Neutron Scattering Analysis," *Nuclear Data Sheets*, **118**, pp. 172-175 (2014).
3. Y. Zhu and A.I. Hawari, "Full Law Analysis Scattering System Hub (FLASSH)," *PHYSOR 2018: Reactor Physics Paving the Way Towards More Efficient Systems*, Cancun, Mexico (2018).
4. M. Van Gerven, "Computational Foundations of Natural Intelligence," *Frontiers in Computational Neuroscience*, **11** (112), (2017).
5. Y. LeCun et al., "Deep Learning," *Nature*, **521**, pp. 436-444 (2015).
6. I. Goodfellow et al., "Generative Adversarial Nets," *arXiv preprint arXiv:1406.2661v1*, (2014).
7. C. Doersch, "Tutorial on Variational Autoencoders," *arXiv preprint arXiv:1606.05908v2*, (2016).
8. C. Szegedy et al., "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *arXiv preprint arXiv:1602.07261v2*, (2016).
9. G. Cybenko, "Approximations by Superposition of Sigmoidal Functions," *Mathematics of Control, Signals, and Systems*, **2**, pp. 303-314 (1989).
10. A. Paszke et al., "Automatic Differentiation in PyTorch," *Advances in Neural Information Processing Systems*, **30**, (2017).
11. D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *3rd International Conference for Learning Representations*, San Diego, USA (2015).