ARTIFICIAL NEURAL NETWORK PERFORMANCE MODEL FOR PARALLEL PARTICLE TRANSPORT CALCULATION

A Thesis

by

J. DILLON HERRING

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Marvin L. Adams
Committee Members,	Jean C. Ragusa
	Jim E. Morel
	Andrea Bonito
Head of Department,	Michael Nastasi

August 2021

Major Subject: Nuclear Engineering

ABSTRACT

There is a need to improve the predictive capability of high-fidelity simulations of physical phenomena that include the transport of thermal radiation and/or other subatomic particles. There are many ingredients of improved capability, including solution algorithms that more efficiently use modern massively parallel computers. The most time-consuming element of many widely used particle-transport methods is the *transport sweep*, in which the particle intensity—a function of position, energy, and direction—is calculated given the most recent estimate for the collisional source. The intensity in a given spatial cell depends on the intensity entering from neighboring cells in the given direction, which imposes restrictions on the order of calculations and implies that cells must communicate exiting intensities to their downstream neighbors. Such dependencies and communication requirements make parallel execution more difficult. A parallel transport calculation in Texas A&M's state-of-the-art PDT code *partitions* the spatial domain across processors as directed by partitioning parameters. It *aggregates* spatial cells into cellsets, directions into anglesets, and energy groups into groupsets, as directed by aggregation parameters. A single work unit during a sweep calculates particle intensities in a single cellset/angleset/groupset combination. At each "stage" of the sweep every processor with available work executes one work unit and communicates outflow intensities to processors responsible for adjacent downstream cellsets. The ingredients of the "optimal sweep" methodology developed by Texas A&M in collaboration with the NNSA labs are: (i) a provably optimal scheduling algorithm, which executes the sweep in the minimum possible number of stages for any given partitioning and aggregation factors; (ii) a performance model that predicts sweep time for that execution; and (iii) an algorithm that chooses partitioning and aggregation factors that minimize sweep time. Here we explore the use of Artificial Neural Networks (ANNs) for such a model, and its memory-use counterpart, and compare against our previous models. We design simple networks that have the ability to replicate previous models but also to augment those models with nonlinear corrections if this better fits the data. These simple nonlinear ANNs outperform our previous models, reducing average prediction

errors from $\approx 41\%$ to $\approx 21\%$ for some problems of interest, although large maximum errors are observed for both models. Additionally PDT reports unexpected results for parallel problems, possibly contribution to the large maximum observed errors. Despite this observation, both the ANN based nonlinear model and our previous model show signs of fruitful practical use for an algorithm such as the one described in (iii). The memory-usage model shows promising results predicting memory usage within ≈ 0.024 GB for out of sample data points.

DEDICATION

To my Mother and Father, for which this would not be possible without. To my lovely Fiancé, Kylie, for her love and support through the long nights and missed dates.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Marvin Adams, advisor, Professors Jean Ragusa and Jim Morel, of the Department of Nuclear Engineering and Professor Andrea Bonito of the Department of Mathematics. We thank Dr. Mauricio Tano-Retamales and Daryl Hawkins for many helpful discussions and patient tutorials. All other work conducted for the thesis was completed by the student independently.

Funding Sources

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002376. Established by Congress in 2000, NNSA is a semi-autonomous agency within the U.S. Department of Energy responsible for enhancing national security through the military application of nuclear science. NNSA maintains and enhances the safety, security, reliability and performance of the U.S. nuclear weapons stockpile without nuclear testing; works to reduce global danger from weapons of mass destruction; provides the U.S. Navy with safe and effective nuclear propulsion; and responds to nuclear and radiological emergencies in the U.S. and abroad.

NOMENCLATURE

ReLU	Rectified Linear Activation Function
PDT	Texas A&M University's state-of-the-art Particle Transport Code
ANN	Artificial Neural Network

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

*

1.1 Background and Motivation

There are many challenges in the field of computational physics. One of them is simulating the interactions between radiation and matter. The linearized Boltzmann equation describes the transport of particles through background media and is widely used to model the transport of neutrons, photons, electrons, and other subatomic particles ([1],[2]). Applications range from nuclear reactor design to photon distributions in the atmosphere. The linearized Boltzmann equation takes the following form:

$$\frac{1}{v(E)}\frac{\partial\psi(\vec{r},E,\hat{\Omega},t)}{\partial t}+\hat{\Omega}\cdot\vec{\nabla}\psi(\vec{r},E,\hat{\Omega},t)+\sigma_t(\vec{r},E,t)\psi(\vec{r},E,\hat{\Omega},t)=Q(\vec{r},E,\hat{\Omega},t). \quad (1.1)$$

Equation (1.1) is a statement of particle conservation, expressing that change rate equals gain rate minus loss rate. The quantity of interest, $\psi(\vec{r},E,\hat{\Omega},t)$, is known as the angular flux. Assuming cartesian geometry, $\vec{r}=<x,y,z>$, the angular flux can be thought of as the total particle tracks at position $\vec{r}$, generated from particles moving in direction $\hat{\Omega}$, per unit volume xyz, per unit energy E, per steradian sr, per unit time t. Commonly, the angular flux has units of $\frac{n-cm}{cm^3-MeV-sr-s}$ (we will use n to represent particles). The dependencies of the angular flux describe what is known as phase space. This is the 7 dimensional space required to fully describe the transport of particles, composed of space (x,y,z); direction ($sin\theta cos\phi$, $sin\theta sin\phi$, $cos\theta$); energy E; and time t. ϕ is the azimuthal angle of particle travel and θ is the polar angle of particle travel. It is regularly referred to as a 6 dimensional phase space at time t as opposed to a 7 dimensional phase space. This is seen in Figure (1.1). It is convenient to refer to the angular flux as a particle track length rate density. Here, particle track length refers to the $n-cm$ units. We add in rate because it is per

*Reprinted with permission from "Artificial Neural Network Performance Models for Parallel Particle Transport Calculation" by James D. Herring, 2021. M&C 2021, Copyright 2021 by M&C 2021.

unit time and density because it is per unit volume, per unit steradian, per unit energy. Notice, this is per unit phase space. When dividing a quantity by a unit of phase space we will describe the ratio as 'quantity rate density'. This nomenclature will be adopted for the remainder of this thesis.

$\sigma_t(\vec{r}, E, t)$ is the total macroscopic interaction cross section, and varies with location, particle

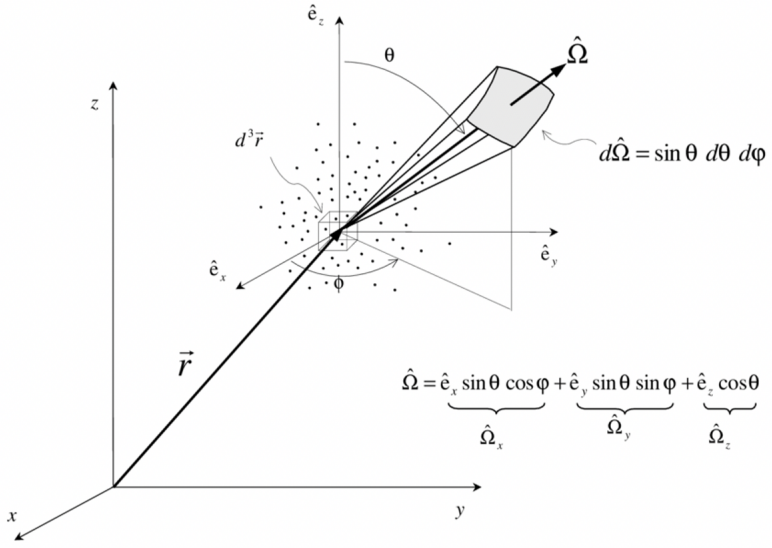

Figure 1.1: Particles at location $\vec{r}$, moving in direction $\hat{\Omega}$ [3].

energy, and time.

It will be helpful to go through each term of Equation (1.1) and formulate a description. Each term is a rate density. It will be convenient to explain each rate density in terms of its integral over an interval of phase space, $\Delta x, \Delta y, \Delta z, \Delta E, \Delta\hat{\Omega}, \Delta t$. Starting on the left hand side of Equation (1.1) we see $\frac{1}{v(E)}\frac{\partial\psi(\vec{r},E,\hat{\Omega},t)}{\partial t}$. This term is the particle change rate density with corresponding units of $\frac{n}{cm^3-MeV-sr-s}$. When integrated over an interval of phase space it is the number of particles in the 6 dimensional phase space interval at time $t + \Delta t$ minus the number of particles at time t. Next we have $\hat{\Omega} \cdot \vec{\nabla}\psi(\vec{r}, E, \hat{\Omega}, t)$. This term is often referred to as the streaming term. It is a particle rate density with units of $\frac{n}{cm^3-MeV-sr-s}$. When integrated over an interval of phase space

it is the number of particles leaving the 6 dimensional phase space interval through its spatial surfaces over the time interval minus the number of particles entering the 6 dimensional phase space interval through its spatial surfaces over the time interval. Then we have $\sigma_t(\vec{r}, E, t)\psi(\vec{r}, E, \hat{\Omega}, t)$, the collision rate density. When integrated over a phase space interval it is the number of collisions between particles and the medium in the 6 dimensional phase space interval over the time interval. $Q(\vec{r}, E, \hat{\Omega}, t)$ is the source rate density with units of $\frac{n}{cm^3-MeV-sr-s}$ and when integrated over a phase space interval is the number of particles born in the 6 dimensional phase space interval over the time interval.

The source term $Q(\vec{r}, E, \hat{\Omega}, t)$ commonly has the form:

$$Q(\vec{r}, E, \hat{\Omega}, t) = \int_0^\infty \int_{4\pi} \sigma_s(\vec{r}, \hat{\Omega}^{'} \rightarrow \hat{\Omega}, E^{'} \rightarrow E, t)\psi(\vec{r}, E^{'}, \hat{\Omega}^{'}, t)d\hat{\Omega}^{'}dE^{'} + Q_{ext}(\vec{r}, E, \hat{\Omega}, t). \tag{1.2}$$

Here $\sigma_s(\vec{r}, \hat{\Omega}^{'} \rightarrow \hat{\Omega}, E^{'} \rightarrow E, t)$ is the differential scattering cross section. It represents the probability that a particle moving in direction $\hat{\Omega}^{'}$ at energy $E^{'}$ undergoes a scattering interaction and ends up in direction $\hat{\Omega}$ at energy E. "Scattering" is used here to include all interactions from which one or more particles emerge. This cross section has units of $\frac{1}{cm-sr-MeV}$. Consequently $\int_0^\infty \int_{4\pi} \sigma_s(\vec{r}, \hat{\Omega}^{'} \rightarrow \hat{\Omega}, E^{'} \rightarrow E, t)\psi(\vec{r}, E^{'}, \hat{\Omega}^{'}, t)d\hat{\Omega}^{'}dE^{'}$ is the total number of particles (including within group scattering) that scatter into direction $\hat{\Omega}$ and energy E per unit volume, per unit energy, per unit time. Typically the differential scattering cross section is expressed as a combination of Legendre polynomials:

$$\sigma_s(\vec{r}, \hat{\Omega}^{'} \rightarrow \hat{\Omega}, E^{'} \rightarrow E, t) = \sum_{n=0}^{N} \frac{2n+1}{4\pi}\sigma_{s,n}(\vec{r}, E^{'} \rightarrow E, t)P_n(\hat{\Omega}^{'} \cdot \hat{\Omega}). \tag{1.3}$$

The $n'th$ Legendre polynomial P_n can be expressed as:

$$P_n(\hat{\Omega}^{'} \cdot \hat{\Omega}) = \frac{4\pi}{2n+1} \sum_{m=-n}^{n} Y_{n,m}(\hat{\Omega})Y^*_{n,m}(\hat{\Omega}^{'}). \tag{1.4}$$

Here $Y_{n,m}$ is the spherical harmonic function of polar order n and azimuthal order m and $Y^*_{n,m}$ is

its complex conjugate [4]. When Equation (1.4) is substituted into Equation (1.3) we end up with:

$$\sigma_s(\vec{r}, \hat{\Omega}^{'} \rightarrow \hat{\Omega}, E^{'} \rightarrow E, t) = \sum_{n=0}^{N} \sum_{m=-n}^{n} \sigma_{s,n}(\vec{r}, E^{'} \rightarrow E, t) Y_{n,m}(\hat{\Omega}) Y^{*}_{n,m}(\hat{\Omega}^{'}). \tag{1.5}$$

Consequently Equation (1.2) can be expressed as:

$$Q(\vec{r}, E, \hat{\Omega}, t) = \int_0^{\infty} \sum_{n=0}^{N} \sum_{m=-n}^{n} \sigma_{s,n}(\vec{r}, E^{'} \rightarrow E, t) Y_{n,m}(\hat{\Omega}) \phi_{n,m}(\vec{r}, E^{'}, t) dE^{'} + Q_{ext}(\vec{r}, E, \hat{\Omega}, t), \tag{1.6}$$

with $\phi_{n,m}(\vec{r}, E^{'}, t) = \int_{4\pi} Y^{*}_{n,m}(\hat{\Omega}^{'}) \psi(\vec{r}, E^{'}, \hat{\Omega}^{'}, t) d\hat{\Omega}^{'}$.

For problems of interest Equation (1.1) does not have an analytic solution, which means we must obtain an approximate solution through the use of discretization methods. First we will tackle the angular dependence $\hat{\Omega}$. Often known as the S_N method, we approximate $\int_{4\pi} Y^{*}_{n,m}(\hat{\Omega}^{'}) \psi(\vec{r}, E^{'}, \hat{\Omega}^{'}, t) d\hat{\Omega}^{'}$ with a quadrature. Applied to Equation (1.1) we have:

$$\begin{aligned} &\frac{1}{v(E)} \frac{\partial \psi(\vec{r}, E, \hat{\Omega}_d, t)}{\partial t} + \hat{\Omega}_d \cdot \vec{\nabla} \psi(\vec{r}, E, \hat{\Omega}_d, t) + \sigma_t(\vec{r}, E, t) \psi(\vec{r}, E, \hat{\Omega}_d, t) = \\ &\int_0^{\infty} \sum_{n=0}^{N} \sum_{m=-n}^{n} \sigma_{s,n}(\vec{r}, E^{'} \rightarrow E, t) Y_{n,m}(\hat{\Omega}_d) \phi_{n,m}(\vec{r}, E^{'}, t) dE^{'} + Q_{ext}(\vec{r}, E, \hat{\Omega}_d, t), \end{aligned} \tag{1.7}$$

and

$$\phi_{n,m}(\vec{r}, E^{'}, t) = \sum_{d=1}^{D} Y^{*}_{n,m}(\hat{\Omega}^{'}_d) \psi(\vec{r}, E^{'}, \hat{\Omega}^{'}_d, t) \Delta\Omega^{'}_d. \tag{1.8}$$

In Equation (1.8) we have approximated the angular integral inside the definition of $\phi_{n,m}$ with a quadrature, with quadrature points $\hat{\Omega}^{'}_d$ and corresponding quadrature weights $\Delta\Omega^{'}_d$. Next we take care of the energy dependence E. First we break all possible energies that particles can have into G intervals or groups. Then we define the g^{th} angular and scalar fluxes as $\int_{\Delta g} \psi(\vec{r}, E, \hat{\Omega}_d, t) dE$ and $\int_{\Delta g} \phi(\vec{r}, E, t) dE$. If we integrate Equation (1.7) over Δg we recognize that

$$\sigma_{t,g}(\vec{r}, t) = \frac{\int_{\Delta g} \sigma_t(\vec{r}, E, t) \psi(\vec{r}, E, \hat{\Omega}_d, t) dE}{\int_{\Delta g} \psi(\vec{r}, E, \hat{\Omega}_d, t) dE}$$

with a similar definition for $\sigma_{s,n,g'\to g}(\vec{r},t)$, in order to obtain Equations (1.9) and (1.10):

$$\frac{1}{v_g}\frac{\partial\psi_g(\vec{r},\hat{\Omega}_d,t)}{\partial t}+\hat{\Omega}_d\cdot\vec{\nabla}\psi_g(\vec{r},\hat{\Omega}_d,t)+\sigma_{t,g}(\vec{r},t)\psi_g(\vec{r},\hat{\Omega}_d,t)= \sum_{g'=1}^{g'=G}\sum_{n=0}^{N}\sum_{m=-n}^{n}\sigma_{s,n,g'\to g}(\vec{r},t)Y_{n,m}(\hat{\Omega}_d)\phi_{n,m,g'}(\vec{r},t)+Q_{ext,g}(\vec{r},\hat{\Omega}_d,t), \tag{1.9}$$

and

$$\phi_{n,m,g'}(\vec{r},t)=\sum_{d=1}^{D}Y^*_{n,m}(\hat{\Omega}'_d)\psi_{g'}(\vec{r},\hat{\Omega}'_d,t)\Delta\Omega'_d. \tag{1.10}$$

Now we will show that time dependent problems can be solved as sequential steady state problems. We will use the backward Euler time discretization method here however the same conclusions can be drawn using other time discretization methods. Applying backward Euler to Equation (1.9) gives us:

$$\frac{1}{v_g}(\frac{\psi_g(\vec{r},\hat{\Omega}_d,t+\Delta t)-\psi_g(\vec{r},\hat{\Omega}_d,t)}{\Delta t})+\hat{\Omega}_d\cdot\vec{\nabla}\psi_g(\vec{r},\hat{\Omega}_d,t+\Delta t)+\sigma_{t,g}(\vec{r},t+\Delta t)\psi_g(\vec{r},\hat{\Omega}_d,t+\Delta t)= \sum_{g'=1}^{g'=G}\sum_{n=0}^{N}\sum_{m=-n}^{n}\sigma_{s,n,g'\to g}(\vec{r},t+\Delta t)Y_{n,m}(\hat{\Omega}_d)\phi_{n,m,g'}(\vec{r},t+\Delta t)+Q_{ext,g}(\vec{r},\hat{\Omega}_d,t+\Delta t). \tag{1.11}$$

If we move $\psi(t)$ to the right hand side we have:

$$\hat{\Omega}_d\cdot\vec{\nabla}\psi_g(\vec{r},\hat{\Omega}_d,t+\Delta t)+(\sigma_{t,g}(\vec{r},t+\Delta t)+\frac{1}{v_g\Delta t})\psi_g(\vec{r},\hat{\Omega}_d,t+\Delta t)= \sum_{g'=1}^{g'=G}\sum_{n=0}^{N}\sum_{m=-n}^{n}\sigma_{s,n,g'\to g}(\vec{r},t+\Delta t)Y_{n,m}(\hat{\Omega}_d)\phi_{n,m,g'}(\vec{r},t+\Delta t)+Q_{ext,g}(\vec{r},\hat{\Omega}_d,t+\Delta t)+\frac{\psi_g(\vec{r},\hat{\Omega}_d,t)}{v_g\Delta t}. \tag{1.12}$$

Recognize that Equation (1.12) is recovered from Equation (1.9) if a steady state approximation is made with two additional terms: a $\frac{1}{v_g\Delta t}$ added to the total cross section and the flux at the previous time step acting as a source. Since these quantities are known Equation (1.12) demonstrates that

time dependent problems can be solved from sequential steady state problems. Consequently, for the remainder of this thesis we will consider steady state problems of the form:

$$\hat{\Omega}_d \cdot \vec{\nabla}\psi_g(\vec{r},\hat{\Omega}_d) + \sigma_{t,g}(\vec{r})\psi_g(\vec{r},\hat{\Omega}_d) = \sum_{g'=1}^{g'=G}\sum_{n=0}^{N}\sum_{m=-n}^{n}\sigma_{s,n,g'\to g}(\vec{r})Y_{n,m}(\hat{\Omega}_d)\phi_{n,m,g'}(\vec{r}) + Q_{ext,g}(\vec{r},\hat{\Omega}_d). \tag{1.13}$$

To deal with the spatial dependence $\vec{r}$ we adopt a Discontinuous Finite Element Method (DFEM). To begin we assume the spatial domain is broken into I cells with the i^{th} cell having volume V_i. We define a set of J basis functions $b_{i,j}$ for cell i. If we approximate the angular flux in cell i to be a linear combination of the basis functions in that cell and the flux evaluated at support points j we have:

$$\psi_i(\vec{r},E_g,\hat{\Omega}_d) = \sum_{j=1}^{J} b_{i,j}(\vec{r})\psi_i(\vec{r_j},E_g,\hat{\Omega}_d). \tag{1.14}$$

For the remainder of this section we will be dropping the full dependency expression. For example we will refer to $\psi_{i,g}(\vec{r_j},\hat{\Omega}_d)$ as $\psi^i_{j,g,d}$. As is customary in finite element analysis, we will multiply by a set of K weight functions $w_{i,k}$ for cell i, and integrate over cell i. Then we insert Equation (1.14) into Equation (1.13). We begin with the second term on the left hand side of Equation (1.13).

$$\sum_{j=1}^{J}\int_{V_i} dV\sigma_{t,g,i}\psi^i_{j,d,g}b_{i,j}(\vec{r})w_{i,k}(\vec{r}), \tag{1.15}$$

where $\sigma_{t,g,i}$ is the total macroscopic interaction cross section in cell i for energy group g. Next we consider the first term on the right hand side of Equation (1.13). We insert the DFEM approximation into our definition of $\phi_{n,m,g}(\vec{r})$:

$$\phi^i_{n,m,g}(\vec{r}) = \sum_{j=1}^{J}\sum_{d'}^{D} Y^*_{n,m,d'}\psi^i_{j,d',g}b_{i,j}(\vec{r})\Delta\Omega_{d'}. \tag{1.16}$$

If we use Equation (1.16) in the first term on the right hand side of Equation (1.13), multiply by

our K weight functions, and integrate over the volume of cell i we get,

$$\sum_{j=1}^{J}\sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G}\int_{V_i} dV w_{i,k}(\vec{r})\sigma_{s,n,g'\rightarrow g,i}Y_{n,m,d}\sum_{d'=1}^{D} Y^{*}_{n,m,d'}\psi^{i}_{j,d',g'}b_{i,j}(\vec{r})\Delta\Omega_{d'} = \sum_{j=1}^{J}\sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G}\int_{V_i} dV w_{i,k}(\vec{r})\sigma_{s,n,g'\rightarrow g,i}Y_{n,m,d}\phi^{i}_{n,m,j,g'}. \tag{1.17}$$

Or written as a vector equation:

$$\sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G}\mathbf{S}_{i,g'\rightarrow g,n}Y_{n,m,d}\vec{\phi}^{i}_{n,m,g'}, \tag{1.18}$$

where element k,j of the matrix $\mathbf{S}_{i,g'\rightarrow g,n}$ is:

$$S^{k,j}_{i,g'\rightarrow g,n} = \int_{V_i} dV w_{i,k}(\vec{r})\sigma_{s,n,g'\rightarrow g,i}b_{i,j}(\vec{r}). \tag{1.19}$$

$\vec{\phi}^{i}_{n,m,g'}$ is the scattering source vector for cell i, polar scattering order n, azimuthal scattering order m, scattering group $g^{'}$, of length J where element j is

$$\phi^{i}_{j,n,m,g'} = \sum_{d'=1}^{D} Y^{*}_{n,m,d'}\psi^{i}_{j,d',g'}\Delta\Omega_{d'}. \tag{1.20}$$

We note that Equation (1.20) expresses part of the right-hand side in terms of the solution, but defer the explanation of how this is treated for later in this section. Next we discretize the source term $Q_{ext}(\vec{r}, E_g, \hat{\Omega}_d)$. If we assume a form similar to Equation (1.14) we can write:

$$Q^{i}_{ext}(\vec{r}, E_g, \hat{\Omega}_d) = \sum_{j=1}^{J} b_{i,j}(\vec{r})Q^{i}_{ext}(\vec{r}_j, E_g, \hat{\Omega}_d). \tag{1.21}$$

Substituting back into Equation (1.11), multiplying by our K weight functions, and integrating over cell i:

$$\sum_{j=1}^{J}\int_{V_i} dV w_{i,k}(\vec{r})b_{i,j}(\vec{r})Q^{i}_{ext}(\vec{r}_j, E_g, \hat{\Omega}_d) = \mathbf{Q}_i\vec{q}^{i}_{d,g}, \tag{1.22}$$

where the element k, j of the matrix $\mathbf{Q}_i$ is:

$$Q_i^{k,j} = \int_{V_i} dV w_{i,k}(\vec{r}) b_{i,j}(\vec{r}), \tag{1.23}$$

and $\vec{q}_{d,g}^{\,i}$ is the extraneous source vector for cell i, group g, and direction d, of length J. The streaming term requires some extra steps. First we get the term in its weak form.

$$\int_{V_i} w_{i,k}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}\psi_{d,g}(\vec{r}))dV. \tag{1.24}$$

Here we have multiplied the streaming term in Equation (1.11) by the set of weight functions and integrated it over cell i. Gausses Divergence Theorem says:

$$\int_V dV \vec{\nabla} \cdot \vec{F} = \int_{\partial V} dA(\hat{e}_n \cdot \vec{F}). \tag{1.25}$$

If we let $F = \hat{\Omega}_d w_{i,k}(\vec{r})\psi_{d,g}(\vec{r})$ then the left hand side of Equation (1.25) becomes:

$$\begin{aligned} \int_{V_i} dV \vec{\nabla} \cdot \hat{\Omega}_d w_{i,k}(\vec{r})\psi_{d,g}(\vec{r}) = \\ \int_{V_i} dV[\Omega_{d,x}\frac{\partial}{\partial_x}(w_{i,k}(\vec{r})\psi_{d,g}(\vec{r})) + \Omega_{d,y}\frac{\partial}{\partial_y}(w_{i,k}(\vec{r})\psi_{d,g}(\vec{r})) + \Omega_{d,z}\frac{\partial}{\partial_z}(w_{i,k}(\vec{r})\psi_{d,g}(\vec{r}))]. \end{aligned} \tag{1.26}$$

Here, $\hat{\Omega}_d$ has been removed from the gradient operator, as it does not depend on space. We recognize that Equation (1.26) can be expressed as:

$$\int_{V_i} dV(\hat{\Omega}_d \cdot \vec{\nabla}(w_{i,k}(\vec{r})\psi_{d,g}(\vec{r}))). \tag{1.27}$$

If we expand the gradient operator using the product rule we get:

$$\begin{aligned} \int_{V_i} dV(\hat{\Omega}_d \cdot (\psi_{d,g}(\vec{r})\vec{\nabla}w_{i,k}(\vec{r}) + w_{i,k}(\vec{r})\vec{\nabla}\psi_{d,g}(\vec{r}))) = \int_{V_i} dV(\psi_{d,g}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}w_{i,k}(\vec{r}))) + \\ \int_{V_i} dV(w_{i,k}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}\psi_{d,g}(\vec{r}))). \end{aligned} \tag{1.28}$$

We recognize that the second term on the right hand side is Equation (1.24). Also, notice that the left hand side is equal to the left hand side of Equation (1.26). Thus we get:

$$\int_{V_i} dV(w_{i,k}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}\psi_{d,g}(\vec{r}))) = \int_{V_i} dV\vec{\nabla} \cdot \hat{\Omega}_d w_{i,k}(\vec{r})\psi_{d,g}(\vec{r}) - \int_{V_i} dV(\psi_{d,g}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}w_{i,k}(\vec{r}))). \tag{1.29}$$

Applying Divergence Theorem to the first term on the right hand side:

$$\int_{V_i} dV(w_{i,k}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}\psi_{d,g}(\vec{r}))) = \int_{\partial V_i} dA(\hat{e}_n \cdot \hat{\Omega}_d w_{i,k}(\vec{r})\psi_{d,g}(\vec{r})) - \int_{V_i} dV(\psi_{d,g}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}w_{i,k}(\vec{r}))). \tag{1.30}$$

Inserting the DFEM approximation characterized by Equation (1.14) we have:

$$\begin{aligned}\sum_{j=1}^{J} \int_{V_i} dV(w_{i,k}(\vec{r})\psi^i_{j,d,g}(\hat{\Omega}_d \cdot \vec{\nabla}b_{i,j}(\vec{r}))) = \sum_{j=1}^{J} \int_{\partial V_i} dA\psi^{\tilde{i}}_{j,d,g}(\hat{e}_n \cdot \hat{\Omega}_d w_{i,k}(\vec{r})b_{\tilde{i},j}(\vec{r})) - \\ \sum_{j=1}^{J} \int_{V_i} dV(\psi^i_{j,d,g} b_{i,j}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla}w_{i,k}(\vec{r}))).\end{aligned} \tag{1.31}$$

The consequences of Equation (1.31) have immediate impact on many widely used radiation transport solution algorithms. Notice the first term on the right hand side (surface term) is a surface integral. That is, $b_{i,j}$ and $w_{i,k}$ will only be evaluated at the surface of cell i in that term. This presents a problem because the DFEM approximation has allowed the angular flux to be discontinuous at cell boundaries [5]. To deal with the discontinuities we have denoted $\tilde{i}$ to be the index of the cell upstream of the surface under consideration. That is

$$\psi_{d,g}(\vec{r}_l) = \begin{cases} \psi_{d,g}(\vec{r}_l^{\,-}) & \text{if } \hat{\Omega}_d \cdot \hat{e}_n > 0 \\ \psi_{d,g}(\vec{r}_l^{\,+}) & \text{if } \hat{\Omega}_d \cdot \hat{e}_n < 0. \end{cases} \tag{1.32}$$

Here we have said that the angular flux evaluated at the surface of cell i characterized by $\vec{r}_l$ is equal to the flux evaluated just inside the surface (i.e. just inside cell i), $\psi_{d,g}(\vec{r}_l^{\,-})$, if the direction of particle travel is outgoing, $\hat{\Omega}_d \cdot \hat{e}_n > 0$. If the direction of particle travel is incoming, $\hat{\Omega}_d \cdot \hat{e}_n < 0$,

then the flux is taken as the value evaluated just outside (i.e just inside the *upstream* cell) the surface $\psi_{d,g}(\vec{r_l}^{+})$. With the discontinuities handled we split the surface integral into the sum of two integrals:

$$\sum_{j=1}^{J}\int_{\partial V_i} dA\tilde{\psi}^{i}_{j,d,g}(\hat{\Omega}_d w_{i,k}(\vec{r})b_{\tilde{i},j}(\vec{r})\cdot\hat{e}_n) = \sum_{j=1}^{J}\int_{\partial V_i^{+}} dA\psi^{i}_{j,d,g}(\hat{\Omega}_d w_{i,k}(\vec{r})b_{i,j}(\vec{r})\cdot\hat{e}_n) + \sum_{j=1}^{J}\int_{\partial V_i^{-}} dA\tilde{\psi}^{i}_{j,d,g}(\hat{\Omega}_d w_{i,k}(\vec{r})\tilde{b}_{i,j}(\vec{r})\cdot\hat{e}_n), \tag{1.33}$$

where

$$\partial V_i = \begin{cases} \partial V_i^{+} & \text{if } \hat{\Omega}_d\cdot\hat{e}_n > 0 \\ \partial V_i^{-} & \text{if } \hat{\Omega}_d\cdot\hat{e}_n < 0, \end{cases}$$

and $\tilde{\psi}^{i}_{d,g}$ is the angular flux evaluated at direction d, group g, in the cell upstream of the surface on cell i being considered. Similarly $\tilde{b}_{i,j}(\vec{r})$ is the $j'th$ basis function in the cell upstream of the surface on cell i being considered. We can now express the right hand side of Equation (1.31) vectorially as:

$$\mathbf{L}_i^{+}\vec{\psi}^{i}_{d,g} + \sum_{f=1}^{f=F}\mathbf{L}^{-}_{i,f}\vec{\tilde{\psi}}^{i}_{f,d,g} - \mathbf{L}_i\vec{\psi}^{i}_{d,g}. \tag{1.34}$$

Here $\mathbf{L}_i^{+}$ represents the outgoing surface matrix for cell i, where element k,j is

$$L^{+}_{i,k,j} = \int_{\partial V_i^{+}} dA(\hat{\Omega}_d w_{i,k}(\vec{r})b_{i,j}(\vec{r})\cdot\hat{e}_n). \tag{1.35}$$

$\mathbf{L}^{-}_{i,f}$ represents the incoming surface matrix for surface f on cell i, where element k,j is

$$L^{-}_{i,f,k,j} = \int_{f} dA(\hat{\Omega}_d w_{i,k}(\vec{r})\tilde{b}_{i,j}(\vec{r})\cdot\hat{e}_n). \tag{1.36}$$

$\vec{\tilde{\psi}}^{i}_{f,d,g}$ is the flux vector for the cell upstream of surface f on cell i. $\mathbf{L}_i$ represents the remaining

terms of Equation (1.31) for cell i, where element k, j is

$$L_{i,k,j} = \int_{V_i} dV(b_{i,j}(\vec{r})(\hat{\Omega}_d \cdot \vec{\nabla} w_{i,k}(\vec{r}))). \tag{1.37}$$

Equation (1.34) produces a crucial result. Notice the first and last term in Equation (1.34) are operating on the same vector, $\vec{\psi}^i_{d,g}$. This is the vector of length J that corresponds to the angular flux in cell i for quadrature direction d and energy group g, where element j is the angular flux at support point j in cell i. The matrix for face f operates on $\vec{\tilde{\psi}}^i_{f,d,g}$. This is the vector of length J whose $j'th$ element is the angular flux for quadrature direction d, energy group g, at support point j in the cell upstream of surface f on cell i. Additionally, we can express Equation (1.15) as

$$\mathbf{M}_{i,g}\vec{\psi}^i_{d,g}, \tag{1.38}$$

where $\mathbf{M}_{i,g}$ is referred to as the mass matrix. Collecting Equations (1.34, 1.38, 1.18, 1.22) we can write the discretized steady state transport equation as:

$$(\mathbf{L}^+_i - \mathbf{L}_i + \mathbf{M}_{i,g})\vec{\psi}^i_{d,g} + \sum_{f=1}^{f=F} \mathbf{L}^-_{i,f}\vec{\tilde{\psi}}^i_{f,d,g} = \sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G} \mathbf{S}_{i,g'\to g,n} Y_{n,m,d}\vec{\phi}^i_{n,m,g'} + \mathbf{Q}_i\vec{q}^i_{d,g}. \tag{1.39}$$

Equation (1.39) is the fully discretized neutron transport equation for cell i. In order to solve Equation (1.39) one must know the incoming angular flux values for cell i, $\vec{\tilde{\psi}}^i_{f,d,g}$ for all F incoming surfaces. One method is to solve each cell sequentially, starting from the boundary (the boundary cell's incoming flux values are determined by the boundary conditions). This technique is known as a *transport sweep* [6]. Furthermore, in the definition of $\vec{\phi}^i$ seen in Equation (1.20), we have a dependence on the solution $\vec{\psi}^i_{d,g}$. To deal with this we use a common technique called source iteration [6]. Applying the iterative technique into Equation (1.39) we get:

$$(\mathbf{L}^+_i - \mathbf{L}_i + \mathbf{M}_{i,g})\vec{\psi}^{i,\ell+1}_{d,g} = -\sum_{f=1}^{f=F} \mathbf{L}^-_{i,f}\vec{\tilde{\psi}}^{i,\ell+1}_{f,d,g} + \sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G} \mathbf{S}_{i,g'\to g,n} Y_{n,m,d}\vec{\phi}^{i,\ell}_{n,m,g'} + \mathbf{Q}_i\vec{q}^i_{d,g}, \tag{1.40}$$

and the j^{th} element of $\vec{\phi}^{i,\ell}_{n,m,g'}$ is

$$\phi^{i,\ell}_{j,n,m,g'} = \sum_{d'=1}^{D} Y^{*}_{n,m,d'} \psi^{i,\ell}_{j,d',g'} \Delta\Omega_{d'}. \tag{1.41}$$

where ℓ is the iteration index. Equations (1.40) and (1.41) tell us that for every cell, direction, energy group combination, there are J unknowns to solve for. Consequently there are $I \times D \times G \times J$ unknowns in the problem. Furthermore, for accurate results in many interesting and difficult problems typical values of I, D, G, and J are $O(10^6)$, $O(10^3)$, $O(10^2)$, and $O(10^1)$ [6]. This leads to $O(10^{12})$ unknowns for typical problems and more for some problems of particular interest. To deal with the size of these problems massively parallel computers are usually employed.

1.2 Transport Sweeps

Equations (1.40) and (1.41) characterize the system of linear equations to be solved on each cell for each energy group and each quadrature direction. Imagine a sequential series of rectangular cells labeled cell 1 to cell I. If particles are restricted to movement in two directions, from cell 1 to cell I and from cell I to cell 1, then its trivial to see that the order in which particles enter each cell is different for each direction. We will call this the sweep ordering. Consequently, solving Equations (1.40) and (1.41) depend on the sweep ordering. A simple sequential solution algorithm to solve these equations can be seen in Algorithm (1).

$\vec{\phi}^{\ell} = \vec{\phi}_0$

while $||\vec{\phi}^{\ell+1} - \vec{\phi}^{\ell}|| > \epsilon$ **do**

  for *d = 1 to D* **do**

    $\vec{\phi}^{i,\ell+1} = \vec{0}$

    for *all i in sweep order d* **do**

      for *g = 1 to G* **do**

        for *all faces f, on cell i* **do**

          if *f is a boundary face for direction* d **then**

            $\vec{\tilde{\psi}}^{i,\ell+1}_{f,d,g} = \vec{\psi}_{BC,f,d,g}$

          else

            use the incoming flux values from upstream cells

          end

        end

        $\vec{\psi}^{i,\ell+1}_{d,g} = (\mathbf{L}_i^{+} - \mathbf{L}_i + \mathbf{M}_{i,g})^{-1}(-\sum_{f=1}^{f=F} \mathbf{L}^{-}_{i,f}\vec{\tilde{\psi}}^{i,\ell+1}_{f,d,g} +$
        $\sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G} \mathbf{S}_{i,g'\rightarrow g,n} Y_{n,m,d}\vec{\phi}^{i,\ell}_{n,m,g'} + \mathbf{Q}_i\vec{q}^{\,i}_{d,g})$

        $\vec{\phi}^{i,\ell+1}_{n,m,g'} = \vec{\phi}^{i,\ell+1}_{n,m,g'} + Y^{*}_{n,m,d}\vec{\psi}^{i,\ell+1}_{d,g'}\Delta\Omega_d$

        Send all outgoing face flux values to downstream cells

      end

    end

  end

  $\vec{\phi}^{\ell} = \vec{\phi}^{\ell+1}$

end

Algorithm 1: Simple transport solution algorithm.

A transport sweep on a single processor machine is characterized by the steps inside the while loop in Algorithm (1). The idea is to solve each cell, direction, group combination sequentially, passing outgoing flux information downstream once it is known. Notice the next iteration of the scattering source vector for cell i, $\vec{\phi}^{i,\ell+1}$, is updated as the angular flux solutions in cell i for direction d and group g become available. Algorithm (1) reveals an important detail. A downstream cell has all the

information it needs to solve for $\vec{\psi}_{d,g}$ once its upstream cells have finished solving Equation (1.40) for direction d and group g. Consequently, it is not necessary to iterate through the entire g for loop in Algorithm (1) before the sweep moves to the next cell in the sweep order.

Imagine a brick shaped spatial domain with brick shaped cells. On such a grid, all directions within a given octant of directional space will have the same sweep ordering, which will begin with a cell in one corner of the domain and end with the cell in farthest corner [6]. We can collect those directions and call them an *angle set*. Angle sets are not required to contain all the angles that have the same sweep ordering, however all angles in an angle set must have the same sweep ordering. Previously, we stated that a downstream cell is ready to compute the angular flux for direction d and group g once its upstream counterpart has finished computation. We can bundle all of the groups an upstream cell computes before communicating results to downstream cells into a *group set*. Similar logic can be applied to cells. The subset of cells in a specific sweep ordering computed before communicating results to downstream cells in the sweep ordering can be bundled into a *cell set*. Consider a problem with multiple angle sets, group sets, and cell sets with 2 angles per angle set and 1 group per group set, $A_n = 2$ and $A_g = 1$. Additionally, unique processors own all the cells in each cell set. The downstream cells in the sweep ordering (owned by another processor) can begin solving Equation (1.40) for the first angle and first group while their upstream counterparts are solving Equation (1.40) for the first angle and the second group. This logic, as seen from a single processor is illustrated in Algorithm (2).

```
for all angle sets do
    for all group sets do
        for all cell sets in current sweep order do
            if proc owns cells in cell set and boundary info is ready then
                for i in cell set do
                    for all d in angle set do
                        for g in group set do
                            for all faces f, on cell i do
                                if f is a boundary face for direction d then
                                    ψ̃^{i,ℓ+1}_{f,d,g} = ψ⃗_{BC,f,d,g}
                                else
                                    use the incoming flux values from upstream cells
                                end
                            end
                            Solve for ψ⃗^{i,ℓ+1}_{d,g} and φ⃗^{i,ℓ+1}_{n,m,g'} using Equations (1.40) and (1.41)
                        end
                    end
                end
                Communicate outgoing face flux values to the processor that owns the next cell set
            else if proc owns cells in cell set, but boundary info is not ready then
                wait, then check again
            else
                continue
            end
        end
    end
end
```

Algorithm 2: Simple parallelized sweep algorithm as seen from one processor.

It is demonstrated in [7] that the time it takes to execute a transport sweep, which we will call the *sweep time*, depends on the sizes of the cell, angle, and group sets. The number of cells, angles, and groups in a set are known as the *aggregation parameters*. Understanding the relationship between sweep time and aggregation parameters is a key ingredient to solving radiation transport problems efficiently. If we are able to predict the sweep time based on the aggregation parameters then we can select the parameters that minimize sweep time. This provides motivation to produce a sweep time model.

It was mentioned earlier that these problems are typically solved on massively parallel machines. In practice, jobs are submitted and placed in a queue. If your job has errors or violates one of the machines operation constraints it will not run and the next job in the queue is executed. One of these constraints to be aware of is memory limitations as it is frustrating to waste time in the queues. As such there is a need to develop a model to predict the memory usage of a transport problem as well.

This work explores the use of Artificial Neural Networks (ANNs) to model two features of the transport sweep: (i) Sweep time; and (ii) Memory usage. As part of the exploration, ANN-based models are compared against traditional analytic models. Sweep-time and memory models are two critical ingredients in the development of optimal sweep algorithms. More details about optimal transport sweeps are provided in [6].

2. ANALYTIC PERFORMANCE MODELS

* This section characterizes PDTs current sweep time model. The concept of “stages" is introduced and used to simplify our sweep algorithm. Then we use that algorithm to derive a linear-combination sweep time model. Researchers previously developed a scaling suite that executes 384 serial PDT runs and records their aggregation parameters, sweep time, and memory usage in .csv format. Here we extend the suite to produce a set of parallel runs in addition to the serial runs. The method of least squares is employed to determine the set of empirical time constants that minimize the squared difference between what the linear combination model predicts and reality. Equation (2.1) is then demonstrated to produce better results if we divide the entire relationship by the number of unknowns in the problem, producing a relationship between the aggregation parameters and the *grind time*.

2.1 PDT's Current Analytic Performance Model

Consider again Algorithm (2). In the previous section we saw that using aggregation parameters allows us to parallelize the sweep algorithm. That is, the next cell set can begin working before all the group sets are finished. Let's define A_{cells}, A_n, and A_g as the aggregation parameters for cells, angles, and groups. Furthermore, let's assume the job is running on N_{procs} processors. If we partition N_{cells} spatial cells evenly amongst the processors, then each processor owns $\frac{N_{cells}}{N_{procs}}$ cells. If we extend our definition of aggregation parameters from the last section to our current example then each processor will have to solve Equation (1.40) $A_{cells} \times A_n \times A_g$ times before it communicates outgoing face flux values to downstream cells. We will call each $A_{cells} \times A_n \times A_g$ block of solutions to Equation (1.40) a *task*. Then, each processor must complete $\frac{N_{cells} N_n N_g}{A_{cells} A_n A_g N_{procs}}$ tasks to solve for all of its unknowns. That is the number of angles sets time the number of group sets times the number of cell sets. We also realize that processors that own downstream cells begin their tasks later than their upstream counterparts. As such they will sit idly until the boundary flux

*Reprinted with permission from “Artificial Neural Network Performance Models for Parallel Particle Transport Calculation" by James D. Herring, 2021. M&C 2021, Copyright 2021 by M&C 2021.

information they need is available. Moreover, the processors that own upstream cells will have to wait idly after they solve for all of their unknowns due to downstream cells beginning their tasks later than the upstream cells. It is demonstrated in [6] that the minimum number of idle tasks required in a sweep for problems of interest is $N_{idle} = P_x + \delta_x - 2 + P_y + \delta_y + N_k(P_z + \delta_z - 2)$. Here P_i is the number of processors along the axis i^{th} coordinate axis, δ_u is 0 or 1 for P_u even or odd, respectively, $N_k = \frac{N_z}{P_z A_z}$, and A_z is the aggregation parameter for the number of cells along the z coordinate axis. We now define $N_{stages} = N_{task} + N_{idle}$. The number of stages is how many times we iterate through the angle sets, group sets, and cell sets, plus the number of times the processor doesn't have boundary information ready and must wait and check again, as illustrated in Algorithm (2). With these definitions we can simplify our sweep algorithm seen in Algorithm (3).

for *stages = 1 to* N_{stages} **do**
  if *cell* i *has boundary information available* **then**
    for *i in cell set* **do**
      Fetch data and prep
      for *d in angle set* **do**
        Form **L** matrices
        for *g in group set* **do**
          $\vec{\psi}_{d,g}^{i,\ell+1} = (\mathbf{L}_i^+ - \mathbf{L}_i + \mathbf{M}_{i,g})^{-1}(-\sum_{f=1}^{f=F} \mathbf{L}_{i,f}^- \vec{\tilde{\psi}}_{f,d,g}^{i,\ell+1} +$
          $\sum_{n=0}^{N} \sum_{m=-n}^{n} \sum_{g'}^{G} \mathbf{S}_{i,g' \to g,m,n} \vec{\phi}_{n,m,g'}^{i,\ell} + \mathbf{Q}_i \vec{q}_{d,g}^{i})$
          $\vec{\phi}_{n,m,g'}^{i,\ell+1} = \vec{\phi}_{n,m,g'}^{i,\ell+1} + Y_{n,m,d}^{*} \vec{\psi}_{d,g'}^{i,\ell+1} \Delta\Omega_d$
        end
      end
    end
  else
    wait
  end
  Communicate outgoing face flux values to downstream cells
end

Algorithm 3: Simple sweep algorithm with aggregation parameters.

From Algorithm (3), we can write down an intuitive relationship for how long a sweep should take. Starting inside the stages for loop there will be some time consumed from initializing all the data structures to house the fluxes and matrices needed. We will call this T_{wf}. We also see that at the end of the stages loop we communicate all outgoing face flux values. The time associated with this communication will be called T_{comm}. We define T_{cell}, T_n, and T_g to be the time it takes to execute each loop respectively. T_g includes the arithmetic operations associated with solving the matrix equation for $\vec{\psi}_{d,g}^{i,\ell+1}$. T_n is the time it takes to form the L matrices. Lastly, we will define T_m as the time it takes to form the scattering source, $\sum_{n=0}^{N}\sum_{m=-n}^{n}\sum_{g'}^{G}\mathbf{S}_{i,g'\rightarrow g,m,n}\vec{\phi}_{n,m,g'}^{i,\ell}$. With these definitions we can write down a relationship for sweep time and aggregation parameters.

$$T_{sweep} = N_{stages}(T_{comm} + T_{wf} + A_{cells}(T_{cell} + A_n(T_n + A_g(T_g + T_m N_m)))), \tag{2.1}$$

where T_{sweep} is the time it takes to sweep across the problem domain, solving for the angular flux in each cell, $N_{stages} = \frac{N_{cells}N_nN_g}{A_{cells}A_nA_gP_xP_yP_z} + N_{idle}$, N_{cells} is the number of spatial cells in the problem, N_n is the number of directions in the problem, N_g is the number of energy groups in the problem, A_{cells} is the number of cells in a cell set, A_n is the number of angles in an angle set, A_g is the number of groups in a group set, P_i is the number of processors along the i^{th} coordinate axis, and N_m is the number of scattering moments in the problem. T_{comm}, T_{wf}, T_{cell}, T_n, T_g, and T_m are empirically determined constants, (using the method of least squares. See Section (2.3)) unique to the machine on which the problem is being run.

Two adjustments are made to the sweep time model above to improve performance. More details are provided in Section (2.3). The first adjustment introduces *grind time*. Grind time is the sweep time per unknown in the problem. When the method of least squares is applied to the model after this adjustment, the squared difference between grind times is minimized as opposed to sweep times like before. The second adjustment is a normalization step. When the method of least squares is applied after this adjustment the squared difference between the sweep time per unknown per grind time and one is minimized.

2.2 PDT Performance Suite

There exists a suite of PDT problems used to study the code's performance. This consists of 384 PDT runs, each one with a different combination of N_g, N_m, A_{cells}, and A_n. Each problem has $16 \times 16 \times 16$ cells and 168 total directions. The suite uses A_{cells} values of 2, 4, 8, and 16 and A_n values of 1, 3, 7, and 21. All angles in a given angle set are in the same octant, and each octant has $\frac{168}{8} = 21$ angles. The suite includes problems with 1, 3, 27, and 99 energy groups, N_g. In Equation (1.41) there is a summation over N scattering moments. PDT sets this quantity using the *scattering order* S_o, where $N = (S_o + 1)^2$. The suite varies scattering order from 0 (isotropic scattering) to 5. A python script launches all 384 jobs sequentially and writes the problem parameters (total cells, total angles, scattering moments, number of processors, aggregation parameters, and sweep time) to a .csv file.

As part of this work we develop a suite of PDT problems to study the codes parallel performance. The problems cover the following $P_x \times P_y \times P_z$ processor arrangements:

- $2 \times 2 \times 2$
- $4 \times 4 \times 2$
- $8 \times 2 \times 2$
- $8 \times 4 \times 4$
- $8 \times 8 \times 2$
- $16 \times 4 \times 2$
- $16 \times 8 \times 8$
- $16 \times 16 \times 4$
- $32 \times 2 \times 2$
- $32 \times 16 \times 2$

- $128 \times 4 \times 2$
- $256 \times 2 \times 2$

Each processor arrangement contains 5 *suites*, where each suite varies the number of cells per processor and the number of angles. Furthermore, each suite contains *variants* that vary the number of scattering moments and groups. Lastly, each variant has *members* that vary the number of cells per cell set along the z axis and the number of angles per angle set. This gives us $P_a \times suites \times variants(P_i) \times members(P_i)$ PDT problems in the data set, where P_a is the number of processor arrangements and P_i is the i^{th} processor arrangement. Notice the number of variants and members depends on the processor arrangement. The suites for each processor arrangement are described below.

- $2 \times 2 \times 2$
  - S1
    - $(N_x, N_y, N_z) : (12, 12, 12)$
    - N_n : 2048 (16 polar, 16 azimuthal)
    - $A_g, S_o : (1, (0, 1, 3, 4, 5)); (3, (0, 4)); (27, (0, 2, 5))$
    - $A_z, A_n : (1, (256, 128, 64, 32, 16, 8));$
      $(2, (256, 128, 32, 8, 4)); (3, (256, 128, 16, 4, 2)); (6, (256, 64, 1))$
  - S2
    - $(N_x, N_y, N_z) : (12, 12, 12)$
    - N_n : 512 (8 polar, 8 azimuthal)
    - $A_g, S_o : (1, (0, 1, 3, 5)); (3, (0, 2)); (27, (0, 2, 5)); (99, 0)$
    - $A_z, A_n : (1, (64, 32, 16, 8, 4)); (2, (32, 8, 2)); (3, (16, 4, 1)); (6, (16))$
  - S3
    - $(N_x, N_y, N_z) : (12, 12, 12)$

* N_n : 128 (4 polar, 4 azimuthal)
* A_g, S_o : $(3, (0, 4)); (27, (0, 2, 5)); (99, 0)$
* A_z, A_n : $(1, (16, 8, 4, 2, 1)); (2, (8, 2)); (3, (16, 4, 1)); (6, (16, 1))$

– S4

* $(N_x, N_y, N_z) : (12, 12, 12)$
* N_n : 128 (4 polar, 4 azimuthal)
* A_g, S_o : $(3, (0, 2, 4)); (27, (1, 5)); (99, 0)$
* A_z, A_n : $(1, (16, 8, 4, 2)); (2, (8, 4)); (4, (16, 4, 2)); (8, (16, 8, 1));$ $(16, (16, 4, 1)); (32, (16, 1)); (64, (16, 1)); (128, (16, 4, 1))$

– S5

* $(N_x, N_y, N_z) : (12, 12, 12)$
* N_n : 32 (2 polar, 2 azimuthal)
* A_g, S_o : $(3, (0, 4)); (27, (1, 5)); (99, 0)$
* A_z, A_n : $(1, (4, 2, 1)); (2, (4, 1)); (3, (4, 1)); (6, (4, 1))$

• $4 \times 4 \times 2$

– S1

* $(N_x, N_y, N_z) : (24, 24, 12)$
* N_n : 2048 (16 polar, 16 azimuthal)
* A_g, S_o : $(1, 1); (3, 1); (27, 1); (99, 1)$
* A_z, A_n : $(1, (16, 64, 32)); (3, (16, 64, 32)); (2, (16, 64, 32))$

– S2

* $(N_x, N_y, N_z) : (48, 48, 12)$
* N_n : 512 (8 polar, 8 azimuthal)
* A_g, S_o : $(1, 3); (3, 3); (27, 3); (99, 3)$

    - $A_z, A_n : (6, (2, 64, 16)); (3, (2, 64, 16)); (1, (2, 64, 16))$

  - S3

    - $(N_x, N_y, N_z) : (96, 96, 12)$
    - $N_n : 128$ (4 polar, 4 azimuthal)
    - $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
    - $A_z, A_n : (2, (2, 4, 8)); (3, (2, 4, 8)); (1, (2, 4, 8))$

  - S4

    - $(N_x, N_y, N_z) : (24, 24, 256)$
    - $N_n : 128$ (4 polar, 4 azimuthal)
    - $A_g, S_o : (1, 5); (3, 5); (27, 5); (99, 5)$
    - $A_z, A_n : (1, (16, 2, 1)); (2, (16, 2, 1)); (4, (16, 2, 1)); (8, (16, 2, 1));$ $(16, (16, 2, 1)); (32, (16, 2, 1)); (64, (16, 2, 1)); (128, (16, 2, 1))$

  - S5

    - $(N_x, N_y, N_z) : (96, 96, 12)$
    - $N_n : 32$ (2 polar, 2 azimuthal)
    - $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
    - $A_z, A_n : (3, (2, 4, 1)); (1, (2, 4, 1)); (2, (2, 4, 1))$

- $8 \times 2 \times 2$

  - S1

    - $(N_x, N_y, N_z) : (48, 12, 12)$
    - $N_n : 2048$ (16 polar, 16 azimuthal)
    - $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
    - $A_z, A_n : (1, (2, 128, 4)); (3, (2, 128, 4)); (2, (2, 128, 4))$

  - S2

* $(N_x, N_y, N_z) : (96, 24, 12)$
* $N_n : 512$ (8 polar, 8 azimuthal)
* $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
* $A_z, A_n : (6, (32, 2, 4)); (2, (32, 2, 4)); (3, (32, 2, 4))$

– S3

* $(N_x, N_y, N_z) : (192, 48, 12)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
* $A_z, A_n : (3, (8, 2, 4)); (1, (8, 2, 4)); (6, (8, 2, 4))$

– S4

* $(N_x, N_y, N_z) : (48, 12, 256)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 5); (3, 5); (27, 5); (99, 5)$
* $A_z, A_n : (1, (2, 8, 16)); (2, (2, 8, 16)); (4, (2, 8, 16)); (8, (2, 8, 16));$ $(16, (2, 8, 16)); (32, (2, 8, 16)); (64, (2, 8, 16)); (128, (2, 8, 16))$

– S5

* $(N_x, N_y, N_z) : (192, 48, 12)$
* $N_n : 32$ (2 polar, 2 azimuthal)
* $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
* $A_z, A_n : (6, (2, 4, 1)); (2, (2, 4, 1)); (1, (2, 4, 1))$

• $8 \times 4 \times 4$

– S1

* $(N_x, N_y, N_z) : (48, 24, 24)$
* $N_n : 2048$ (16 polar, 16 azimuthal)

* $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
* $A_z, A_n : (1, (64, 256, 2)); (2, (64, 256, 2)); (3, (64, 256, 2)); (6, (64, 256, 2))$

– S2

* $(N_x, N_y, N_z) : (96, 48, 24)$
* $N_n : 512$ (8 polar, 8 azimuthal)
* $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
* $A_z, A_n : (1, (8, 32, 16)); (2, (8, 32, 16)); (3, (8, 32, 16)); (6, (8, 32, 16))$

– S3

* $(N_x, N_y, N_z) : (192, 96, 24)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
* $A_z, A_n : (1, (4, 8, 2)); (2, (4, 8, 2)); (3, (4, 8, 2)); (6, (4, 8, 2))$

– S4

* $(N_x, N_y, N_z) : (48, 24, 512)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 1); (3, 1); (27, 1); (99, 1)$
* $A_z, A_n : (1, (4, 8, 16)); (2, (4, 8, 16)); (4, (4, 8, 16)); (8, (4, 8, 16));$ $(16, (4, 8, 16)); (32, (4, 8, 16)); (64, (4, 8, 16)); (128, (4, 8, 16))$

– S5

* $(N_x, N_y, N_z) : (192, 96, 24)$
* $N_n : 32$ (2 polar, 2 azimuthal)
* $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
* $A_z, A_n : (1, (1, 2, 4)); (2, (1, 2, 4)); (3, (1, 2, 4)); (6, (1, 2, 4))$

• $8 \times 8 \times 2$

- S1
  - $(N_x, N_y, N_z) : (48, 48, 12)$
  - $N_n : 2048$ (16 polar, 16 azimuthal)
  - $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
  - $A_z, A_n : (6, (16, 128, 2)); (3, (16, 128, 2)); (1, (16, 128, 2))$
- S2
  - $(N_x, N_y, N_z) : (96, 96, 12)$
  - $N_n : 512$ (8 polar, 8 azimuthal)
  - $A_g, S_o : (1, 1); (3, 1); (27, 1); (99, 1)$
  - $A_z, A_n : (6, (2, 4, 16)); (1, (2, 4, 16)); (3, (2, 4, 16))$
- S3
  - $(N_x, N_y, N_z) : (192, 192, 12)$
  - $N_n : 128$ (4 polar, 4 azimuthal)
  - $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
  - $A_z, A_n : (3, (16, 8, 2)); (1, (16, 8, 2)); (2, (16, 8, 2))$
- S4
  - $(N_x, N_y, N_z) : (48, 48, 256)$
  - $N_n : 128$ (4 polar, 4 azimuthal)
  - $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
  - $A_z, A_n : (1, (2, 1, 8)); (2, (2, 1, 8)); (4, (2, 1, 8)); (8, (2, 1, 8));$ $(16, (2, 1, 8)); (32, (2, 1, 8)); (64, (2, 1, 8)); (128, (2, 1, 8))$
- S5
  - $(N_x, N_y, N_z) : (192, 192, 12)$
  - $N_n : 32$ (2 polar, 2 azimuthal)

    - $A_g, S_o : (1,5); (3,5); (27,5); (99,5)$
    - $A_z, A_n : (6,(2,4,1)); (2,(2,4,1)); (1,(2,4,1))$

- $16 \times 4 \times 2$
  - S1
    - $(N_x, N_y, N_z) : (96, 24, 12)$
    - N_n : 2048 (16 polar, 16 azimuthal)
    - $A_g, S_o : (1,2); (3,2); (27,2); (99,2)$
    - $A_z, A_n : (6,(8,128,2)); (3,(8,128,2)); (1,(8,128,2))$
  - S2
    - $(N_x, N_y, N_z) : (192, 48, 12)$
    - N_n : 512 (8 polar, 8 azimuthal)
    - $A_g, S_o : (1,0); (3,0); (27,0); (99,0)$
    - $A_z, A_n : (3,(8,2,1)); (1,(8,2,1)); (2,(8,2,1))$
  - S3
    - $(N_x, N_y, N_z) : (384, 96, 12)$
    - N_n : 128 (4 polar, 4 azimuthal)
    - $A_g, S_o : (1,5); (3,5); (27,5); (99,5)$
    - $A_z, A_n : (3,(8,4,2)); (1,(8,4,2)); (2,(8,4,2))$
  - S4
    - $(N_x, N_y, N_z) : (96, 24, 256)$
    - N_n : 128 (4 polar, 4 azimuthal)
    - $A_g, S_o : (1,4); (3,4); (27,4); (99,4)$
    - $A_z, A_n : (1,(2,1,8)); (2,(2,1,8)); (4,(2,1,8)); (8,(2,1,8));$
      $(16,(2,1,8)); (32,(2,1,8)); (64,(2,1,8)); (128,(2,1,8))$

- S5
    - $(N_x, N_y, N_z) : (384, 96, 12)$
    - N_n : 32 (2 polar, 2 azimuthal)
    - $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
    - $A_z, A_n : (6, (2, 4, 1)); (2, (2, 4, 1)); (1, (2, 4, 1))$

- $16 \times 8 \times 8$
    - S1
        - $(N_x, N_y, N_z) : (96, 48, 48)$
        - N_n : 2048 (16 polar, 16 azimuthal)
        - $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
        - $A_z, A_n : (6, (2, 256, 16)); (3, (2, 256, 16)); (1, (2, 256, 16)); (2, (2, 256, 16))$
    - S2
        - $(N_x, N_y, N_z) : (192, 96, 48)$
        - N_n : 512 (8 polar, 8 azimuthal)
        - $A_g, S_o : (1, 5); (3, 5); (27, 5); (99, 5)$
        - $A_z, A_n : (3, (4, 16, 2)); (1, (4, 16, 2)); (2, (4, 16, 2)); (6, (4, 16, 2))$
    - S3
        - $(N_x, N_y, N_z) : (384, 192, 48)$
        - N_n : 128 (4 polar, 4 azimuthal)
        - $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
        - $A_z, A_n : (3, (1, 16, 2)); (1, (1, 16, 2)); (2, (1, 16, 2)); (6, (1, 16, 2))$
    - S4
        - $(N_x, N_y, N_z) : (96, 48, 1024)$
        - N_n : 128 (4 polar, 4 azimuthal)

* $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
* $A_z, A_n : (1, (1, 4, 16)); (2, (1, 4, 16)); (4, (1, 4, 16)); (8, (1, 4, 16));$ $(16, (1, 4, 16)); (32, (1, 4, 16)); (64, (1, 4, 16)); (128, (1, 4, 16))$

– S5

* $(N_x, N_y, N_z) : (384, 192, 48)$
* $N_n : 32$ (2 polar, 2 azimuthal)
* $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
* $A_z, A_n : (6, (2, 4, 1)); (2, (2, 4, 1)); (1, (2, 4, 1)); (3, (2, 4, 1))$

• $16 \times 16 \times 4$

– S1

* $(N_x, N_y, N_z) : (96, 96, 24)$
* $N_n : 2048$ (16 polar, 16 azimuthal)
* $A_g, S_o : (1, 1); (3, 1); (27, 1); (99, 1)$
* $A_z, A_n : (6, (256, 4, 32)); (3, (256, 4, 32)); (1, (256, 4, 32)); (2, (256, 4, 32))$

– S2

* $(N_x, N_y, N_z) : (192, 192, 24)$
* $N_n : 512$ (8 polar, 8 azimuthal)
* $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
* $A_z, A_n : (3, (8, 1, 64)); (1, (8, 1, 64)); (2, (8, 1, 64)); (6, (8, 1, 64))$

– S3

* $(N_x, N_y, N_z) : (384, 384, 24)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 5); (3, 5); (27, 5); (99, 5)$
* $A_z, A_n : (3, (16, 4, 2)); (1, (16, 4, 2)); (2, (16, 4, 2)); (6, (16, 4, 2))$

- S4
    - $(N_x, N_y, N_z) : (96, 96, 512)$
    - $N_n : 128$ (4 polar, 4 azimuthal)
    - $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
    - $A_z, A_n : (1, (2, 8, 16)); (2, (2, 8, 16)); (4, (2, 8, 16)); (8, (2, 8, 16)); (16, (2, 8, 16)); (32, (2, 8, 16)); (64, (2, 8, 16)); (128, (2, 8, 16))$
- S5
    - $(N_x, N_y, N_z) : (384, 384, 24)$
    - $N_n : 32$ (2 polar, 2 azimuthal)
    - $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
    - $A_z, A_n : (6, (2, 4, 1)); (2, (2, 4, 1)); (1, (2, 4, 1)); (3, (2, 4, 1))$

- $32 \times 2 \times 2$
    - S1
        - $(N_x, N_y, N_z) : (192, 12, 12)$
        - $N_n : 2048$ (16 polar, 16 azimuthal)
        - $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
        - $A_z, A_n : (6, (128, 64, 32)); (3, (128, 64, 32)); (2, (128, 64, 32))$
    - S2
        - $(N_x, N_y, N_z) : (384, 24, 12)$
        - $N_n : 512$ (8 polar, 8 azimuthal)
        - $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
        - $A_z, A_n : (3, (4, 32, 1)); (1, (4, 32, 1)); (6, (4, 32, 1))$
    - S3
        - $(N_x, N_y, N_z) : (768, 48, 12)$

* N_n : 128 (4 polar, 4 azimuthal)
* A_g, S_o : $(1, 5); (3, 5); (27, 5); (99, 5)$
* A_z, A_n : $(3, (16, 4, 1)); (1, (16, 4, 1)); (6, (16, 4, 1))$

– S4

* (N_x, N_y, N_z) : $(192, 12, 256)$
* N_n : 128 (4 polar, 4 azimuthal)
* A_g, S_o : $(1, 1); (3, 1); (27, 1); (99, 1)$
* A_z, A_n : $(1, (2, 4, 16)); (2, (2, 4, 16)); (4, (2, 4, 16)); (8, (2, 4, 16)); (16, (2, 4, 16)); (32, (2, 4, 16)); (64, (2, 4, 16)); (128, (2, 4, 16))$

– S5

* (N_x, N_y, N_z) : $(768, 48, 12)$
* N_n : 32 (2 polar, 2 azimuthal)
* A_g, S_o : $(1, 2); (3, 2); (27, 2); (99, 2)$
* A_z, A_n : $(6, (2, 4, 1)); (1, (2, 4, 1)); (3, (2, 4, 1))$

• $32 \times 16 \times 2$

– S1

* (N_x, N_y, N_z) : $(192, 96, 12)$
* N_n : 2048 (16 polar, 16 azimuthal)
* A_g, S_o : $(1, 4); (3, 4); (27, 4); (99, 4)$
* A_z, A_n : $(1, (8, 32, 16)); (3, (8, 32, 16)); (2, (8, 32, 16))$

– S2

* (N_x, N_y, N_z) : $(384, 192, 12)$
* N_n : 512 (8 polar, 8 azimuthal)
* A_g, S_o : $(1, 5); (3, 5); (27, 5); (99, 5)$

* $A_z, A_n : (2, (8, 16, 4)); (1, (8, 16, 4)); (6, (8, 16, 4))$

- S3
  * $(N_x, N_y, N_z) : (768, 384, 12)$
  * $N_n : 128$ (4 polar, 4 azimuthal)
  * $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
  * $A_z, A_n : (3, (16, 8, 1)); (2, (16, 8, 1)); (6, (16, 8, 1))$
- S4
  * $(N_x, N_y, N_z) : (192, 96, 256)$
  * $N_n : 128$ (4 polar, 4 azimuthal)
  * $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
  * $A_z, A_n : (1, (2, 1, 16)); (2, (2, 1, 16)); (4, (2, 1, 16)); (8, (2, 1, 16)); (16, (2, 1, 16)); (32, (2, 1, 16)); (64, (2, 1, 16)); (128, (2, 1, 16))$
- S5
  * $(N_x, N_y, N_z) : (768, 384, 12)$
  * $N_n : 32$ (2 polar, 2 azimuthal)
  * $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
  * $A_z, A_n : (6, (2, 4, 1)); (2, (2, 4, 1)); (3, (2, 4, 1))$

- $128 \times 4 \times 2$
  - S1
    * $(N_x, N_y, N_z) : (268, 24, 12)$
    * $N_n : 2048$ (16 polar, 16 azimuthal)
    * $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
    * $A_z, A_n : (1, (8, 1, 16)); (3, (8, 1, 16)); (2, (8, 1, 16))$
  - S2

* $(N_x, N_y, N_z) : (1536, 48, 12)$
* $N_n : 512$ (8 polar, 8 azimuthal)
* $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
* $A_z, A_n : (3, (8, 2, 32)); (1, (8, 2, 32)); (6, (8, 2, 32))$

– S3

* $(N_x, N_y, N_z) : (3072, 96, 12)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
* $A_z, A_n : (1, (16, 2, 4)); (2, (16, 2, 4)); (6, (16, 2, 4))$

– S4

* $(N_x, N_y, N_z) : (768, 24, 256)$
* $N_n : 128$ (4 polar, 4 azimuthal)
* $A_g, S_o : (1, 5); (3, 5); (27, 5); (99, 5)$
* $A_z, A_n : (1, (8, 2, 4)); (2, (8, 2, 4)); (4, (8, 2, 4)); (8, (8, 2, 4)); (16, (8, 2, 4)); (32, (8, 2, 4)); (64, (8, 2, 4)); (128, (8, 2, 4))$

– S5

* $(N_x, N_y, N_z) : (3072, 96, 12)$
* $N_n : 32$ (2 polar, 2 azimuthal)
* $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
* $A_z, A_n : (6, (2, 4, 1)); (2, (2, 4, 1)); (1, (2, 4, 1))$

• $256 \times 2 \times 2$

– S1

* $(N_x, N_y, N_z) : (1536, 12, 12)$
* $N_n : 2048$ (16 polar, 16 azimuthal)

* $A_g, S_o : (1, 0); (3, 0); (27, 0); (99, 0)$
* $A_z, A_n : (1, (16, 128, 256)); (3, (16, 128, 256)); (6, (16, 128, 256))$

- S2
  * $(N_x, N_y, N_z) : (3072, 24, 12)$
  * $N_n : 512$ (8 polar, 8 azimuthal)
  * $A_g, S_o : (1, 2); (3, 2); (27, 2); (99, 2)$
  * $A_z, A_n : (3, (8, 2, 32)); (1, (8, 2, 32)); (2, (8, 2, 32))$
- S3
  * $(N_x, N_y, N_z) : (6144, 48, 12)$
  * $N_n : 128$ (4 polar, 4 azimuthal)
  * $A_g, S_o : (1, 1); (3, 1); (27, 1); (99, 1)$
  * $A_z, A_n : (1, (8, 16, 1)); (2, (8, 16, 1)); (3, (8, 16, 1))$
- S4
  * $(N_x, N_y, N_z) : (1536, 12, 256)$
  * $N_n : 128$ (4 polar, 4 azimuthal)
  * $A_g, S_o : (1, 3); (3, 3); (27, 3); (99, 3)$
  * $A_z, A_n : (1, (8, 2, 16)); (2, (8, 2, 16)); (4, (8, 2, 16)); (8, (8, 2, 16));$ $(16, (8, 2, 16)); (32, (8, 2, 16)); (64, (8, 2, 16)); (128, (8, 2, 16))$
- S5
  * $(N_x, N_y, N_z) : (6144, 48, 12)$
  * $N_n : 32$ (2 polar, 2 azimuthal)
  * $A_g, S_o : (1, 4); (3, 4); (27, 4); (99, 4)$
  * $A_z, A_n : (3, (2, 4, 1)); (2, (2, 4, 1)); (1, (2, 4, 1))$

Similarly, a data set is constructed to study PDTs memory usage. Training and testing problems were serial (memory per processor is not a strong function of processor count and we wished to

begin exploring memory models in a relatively simple setting). Training runs varied the following parameters:

$$A_g = \{1, 3, 27, 99\} \; ; \qquad N_n = \{48, 168\} \; ; \tag{2.2}$$

$$N_m = \{1, 9, 25\} \; ; \qquad N_x = N_y = N_z = \{10, 16\} \; ; \tag{2.3}$$

$$(A_x, A_y, A_z) = \text{ prime factors of } \{10, 16\} \; . \tag{2.4}$$

Testing runs used $N_x = A_x = N_y = A_y = N_z = 14$ and varied the following:

$$A_g = \{3, 99\} \; ; \qquad N_n = \{120, 360\} \; ; \tag{2.5}$$

$$N_m = \{4, 16\} \; ; \qquad A_z = \{1, 2, 7, 14\} \; . \tag{2.6}$$

2.3 Finding the Time Constants

The method of Least Squares is applied to Equation (2.1) to obtain the time constants that minimizes the squared difference between what the linear combination model predicts and what the data set reports for the sweep time. For a given set of tabulated PDT runs the problem parameters and the sweep times are known. The proceeding characterization assumes use of the parallel sweep time training set described in Section (2.2). For the i^{th} PDT run, Equation. (2.1) can be written as $T_{sweep,i} = N_{stages,i}(T_{comm} + T_{wf} + A_{cells,i}(T_{cell} + A_{n,i}(T_n + A_{g,i}(T_g + T_m N_{m,i}))))$. Notice there is no subscript i on any of the time constants. As such they can be factored out of all equations and we have a system of linear equations:

$$\mathbf{A}\vec{t} = \vec{T}_{sweep} \tag{2.7}$$

where $\mathbf{A}$ is a matrix with each row representing the problem parameters of each PDT run, $\vec{t}$ is the unknown vector of time constants ($T_{latency}$, T_{db}, T_{wf}, T_{cell}, T_n, T_g, T_m), and $\vec{T}_{sweep}$ is the vector of all the tabulated sweep times. The step described above is necessary to compute the vector of time constants such that sweep times that have not been tabulated can be computed. Some elaboration on T_{comm} is needed. T_{comm} is the time for processor-to-processor communication per stage. It

is modeled as $3T_{latency} + N_{outcs}T_{db}$ [6]. This is the communication time to three downstream processors as latency time plus the time to send N_{outcs} values to those processors, where N_{outcs} is the number of unknowns on the exiting surfaces of a cellset for one angleset and groupset. With this definition Equation (2.1) becomes:

$$T_{sweep} = N_{stages}((3T_{latency} + N_{outcs}T_{db}) + T_{wf} + A_{cells}(T_{cell} + A_n(T_n + A_g(T_g + T_m N_m)))) \quad (2.8)$$

In [6] it is shown that $N_{outcs} = N_g A_n N_\alpha (A_x A_y + A_x A_z + A_y A_z)$. N_α is the number of spatial degrees of freedom communicated per cell face and depends on the basis functions selected during the DFEM process described in Section (1.1). For the linear discontinuous spatial discretization, which is used here, $N_\alpha = 4$. Because our data set contains serial and parallel runs special treatment has to be employed such that the elements of the matrix A that multiply $T_{latency}$ and T_{double} are correct for serial and parallel runs. To accomplish this Equation (2.8) is written as:

$$\begin{aligned} T_{sweep} = N_{stages}(((min(1, P_x - 1) + min(1, P_y - 1) + min(1, P_z - 1))T_{latency} + \\ (N_g A_n N_\alpha (min(P_z, P_z - 1)A_x A_y + min(P_y, P_y - 1)A_x A_z + min(P_x, P_x - 1)A_y A_z))T_{db}) + \\ T_{wf} + A_{cells}(T_{cell} + A_n(T_n + A_g(T_g + T_m N_m)))) \end{aligned} \quad (2.9)$$

Equation (2.7) is built using Equation (2.9), where A has dimensions 816×7, $\vec{t}$ has dimension 7, and $\vec{T}_{sweep}$ has dimension 816. The system of linear equations in matrix Equation (2.7) represents an over determined system leaving A non-invertible. Thus a least squares solution is employed to compute $\vec{t}$. It is demonstrated in [12] that

$$\vec{t} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^T \cdot \vec{T}_{sweep}) \quad (2.10)$$

yields the $\vec{t}$ that minimizes the squared difference between $\vec{T}_{sweep}$ and $\mathbf{A} \cdot \vec{t}$. Equation (2.10) is known as the Method of Least Squares. To test the predictive capability of this method it is applied

to the serial data set described in Section (2.2). The results can be seen in Figure (2.1). Greater than two thirds of the data points are between -50% and 50% of reality. However, these results are not sufficiently accurate for our purposes. All of the runs with a percent error greater than 100% come from PDT runs with one energy group. That is, they are the smallest problems relative to the data set. Consequently, they have the smallest sweep time. The 99 energy group problems are the largest problems in the data set. Consequently they have the largest sweep time. The largest problems sweep times are $\approx 3000\%$ bigger than the smallest problems. As a result they hold more weight in the least squares process. This model fits large problems well but misses smaller problems. To remedy this we attempt to fit against a metric that is not weighted too much to one side. We chose grind time as opposed to sweep time. Grind time is the sweep time divided by the total number of unknowns in the problem per processor, thus accounting for increased sweep times in larger problems. The adjustments to our least squares procedure are simple. The i^{th} sweep time from Equation (2.7) is written as:

$$\frac{\vec{A}_i}{N_{cells,i}N_{g,i}N_{n,i}N_{\alpha}N_{p,i}}\vec{t} = T_{grind,i}. \tag{2.11}$$

Here, Equation (2.11) is simply the i^{th} row from **A** in Equation (2.7) divided by the total number of unknowns per sweep per processor for the i^{th} PDT run, where $N_{p,i}$ is the number of processors in the i^{th} PDT run. Our least squares procedure applied to Equation (2.11) produces the results seen in Figure (2.2). These results are significantly improved over those in Figure (2.1), with approximately 82% of the runs having less than or equal to 20% error. Equation (2.11) is minimizing the square of the relative difference between the linear combination models prediction of the grind time and the observed grind time. However, if we divided Equation (2.11) by $T_{grind,i}$ we will have eliminated all possible weighting skews during the least squares process. This gives us

$$\frac{\vec{A}_i}{N_{cells,i}N_{g,i}N_{n,i}N_{\alpha}T_{grind,i}N_{p,i}}\vec{t} = 1. \tag{2.12}$$

Figure (2.3) reports the percent difference between Equation (2.12) and tabulated grind times. We

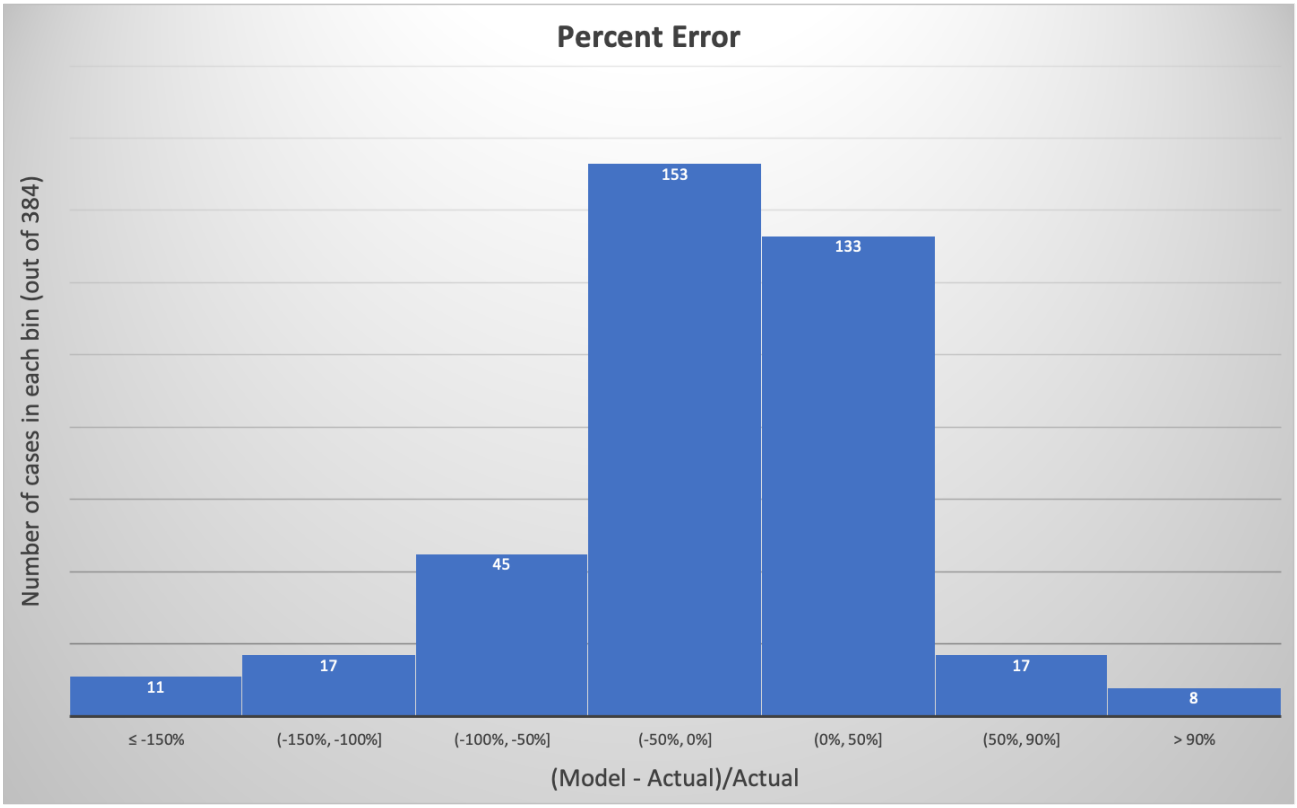

Figure 2.1: Percent Error between Equation (2.7) and tabulated serial sweep times.

do see an improvement from the previous model described by Equation (2.11), with approximately 93% of the runs having less than or equal to 20% error. This model does perform well enough for practical use but higher accuracy is desired.

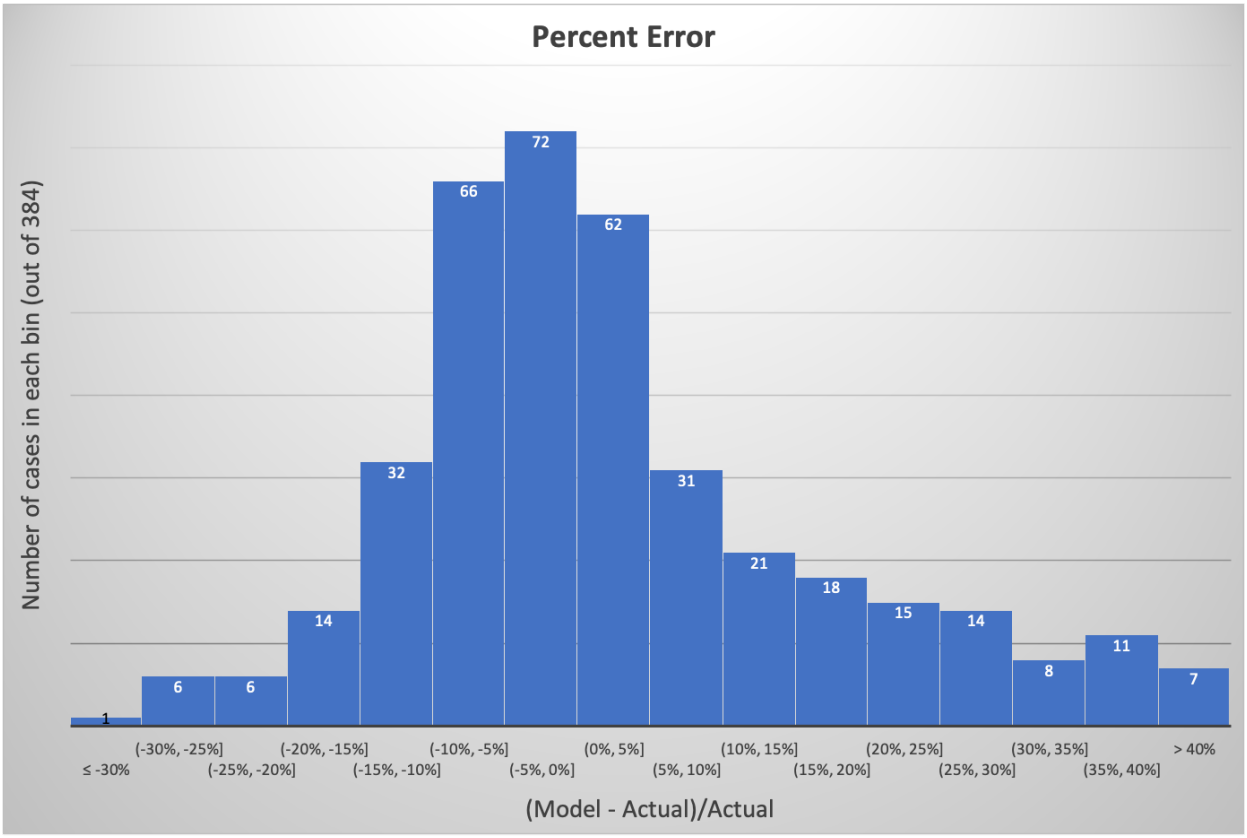

Figure 2.2: Percent Error between Equation (2.11) and tabulated serial grind times.

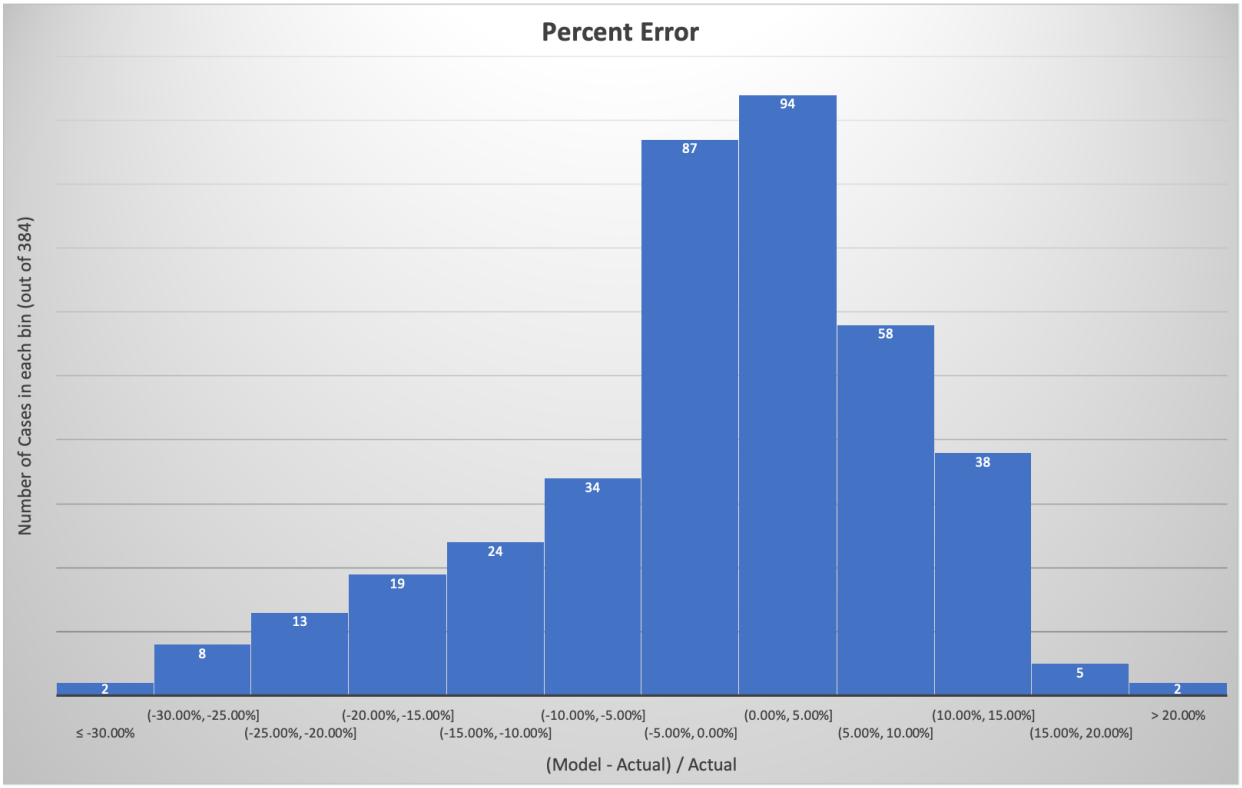

Figure 2.3: Percent error between Equation (2.12) and tabulated serial grind times.

3.1 Densely Connected Neurons

There is a need for an algorithm to accurately predict transport sweep times and memory usage as a function of partitioning and aggregation parameters for a given problem on a given machine with a given number of processors. We will investigate the ability of ANNs to make these predictions.

We begin by giving a more detailed explanation of ANNs and how they learn. Much of this explanation can be found in [8] and [9]. ANNs are composed of layers each with a specified number of neurons, as seen in Figure (3.1). Each neuron has its own *activation* which is just a real

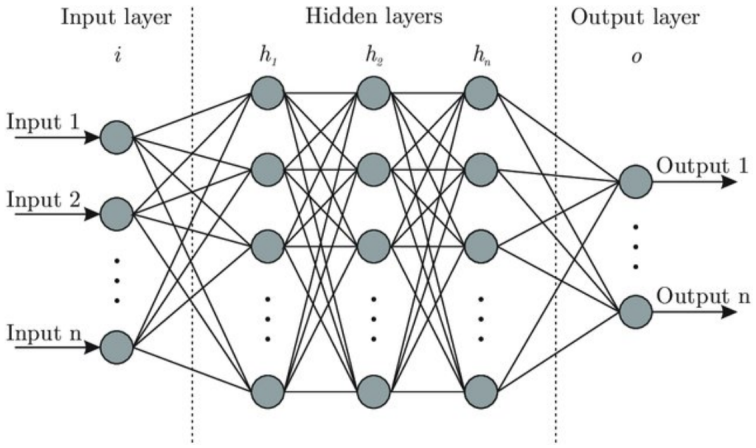

Figure 3.1: Example ANN architecture.

number. The first layer of the network is the input layer. The last layer contains the output. This work uses a pseudo-densely connected network. In a densely connected network each neuron in a layer is connected to all the neurons in the previous layer. Mathematically this looks like:

$$a_l^i = \sigma(\Sigma_j^{N_{l-1}}(a_{l-1}^j w_{ij}) + b_l^i), \tag{3.1}$$

where σ is the activation function, a_l^i is the activation for the i^{th} neuron in layer l, w_{ij} is the weight between the i^{th} neuron in layer l and the j^{th} neuron in layer $l-1$, and b_l^i is the bias for the i^{th} neuron in layer l. Activation functions are applied to each neuron to add nonlinearity or prevent unwanted activations (vanishing or exploding activations).

3.2 Weight and Bias Tuning

ANNs require data sets. Let $\mathbf{X}$ be a matrix containing M observations for N independent variables. Moreover, $\vec{y}$ is the set of M dependent variables. That is, $\mathbf{X} \in \mathbb{R}^{M\times N}$ and $\vec{y} \in \mathbb{R}^M$. We call this a data set. ANNs are trained on data sets by tuning the weights and biases seen in Equation (3.1) via optimizing an objective function. The network used for this work seeks to minimize a loss function. To accomplish this mission we employ stochastic gradient descent (SGD). Generally, SGD is performed by selecting the next evaluation point in the negative direction of the gradient. That is, for some function $f(x)$, we search for a solution iteratively of the form $x_{i+1} = x_i - \alpha\frac{df(x)}{dx}$. This is the simple case where f only depends on one input, x.

Define $L(\vec{\hat{y}}(\mathbf{X}, \vec{w}, \vec{b}), \vec{y})$ to be a loss function. Qualitatively, it is a measure of how close the ANNs predictions are to the training set. L depends on $\hat{y}_i$ and y_i, where $\hat{y}_i$ is the i^{th} output of the ANN and y_i is the i^{th} element in the training set. Our goal, is to find the weights and biases that minimize L. The weights and biases are tuned from the derivative of the loss function with respect to the weight or bias being updated. However, those gradients depend on each data point. Common ANN implementations use reverse mode automatic differentiation to compute the gradients of the ANNs activations with respect to the weights and biases for each data point. These gradients are then propagated through the network (called back propagation) to compute the gradient of the loss function with respect to the weights and biases. Let O signify the output layer. Back propagation begins in layer O.

$$\frac{\partial L}{\partial w_j^{O-1}} = \sum_i \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial w_j^{O-1}}, \tag{3.2}$$

$$\frac{\partial L}{\partial b_y^{O}} = \sum_i \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial b_y^{O}}, \tag{3.3}$$

$$\frac{\partial L}{\partial a_j^{O-1}} = \sum_i \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial a_{j,i}^{O-1}}. \tag{3.4}$$

Equations (3.2-3.4) are summing the contributions of all the data points, i, to the gradient of the loss function with respect to: 1) the weight between the output neuron and the j^{th} neuron in the layer preceding the output layer; 2) the output neuron bias; 3) as well as the activation of the j^{th} neuron in the layer preceding the output layer. It's important to note that $\hat{y}_i$ is just the activation of the output neuron. These derivatives are stored and the algorithm moves to the next layer (the next closest layer to the inputs). It is common to see a $\frac{\partial \hat{y}_i}{\partial \sigma} \frac{\partial \sigma}{\partial x}$ instead of $\frac{\partial \hat{y}_i}{\partial x}$, for some general model parameter x.

$$\frac{\partial L}{\partial w_{j^{O-1}k^{O-2}}} = \frac{\partial L}{\partial a_j^{O-1}} \sum_i \frac{\partial a_{j,i}^{O-1}}{\partial w_{j^{O-1}k}^{O-2}}, \tag{3.5}$$

$$\frac{\partial L}{\partial b_j^{O-1}} = \frac{\partial L}{\partial a_j^{O-1}} \sum_i \frac{\partial a_{j,i}^{O-1}}{\partial b_j^{O-1}}, \tag{3.6}$$

$$\frac{\partial L}{\partial a_k^{O-2}} = \sum_{j=1}^{N_{O-1}} \frac{\partial L}{\partial a_j^{O-1}} \sum_i \frac{\partial a_{j,i}^{O-1}}{\partial a_{k,i}^{O-2}}. \tag{3.7}$$

Here $\frac{\partial L}{\partial w_{j^{O-1}k^{O-2}}}$ is the derivative of the loss function with respect to the weight between the j^{th} neuron in layer $O-1$ and k^{th} neuron in layer $O-2$, $\frac{\partial L}{\partial b_j^{O-1}}$ is the derivative of the loss function with respect to the j^{th} neuron's bias in layer $O-1$, $\frac{\partial L}{\partial a_k^{O-2}}$ is the derivative of the loss function with respect to the k^{th} neuron's activation in layer $O-2$, and N_{O-1} is the number of neurons in layer $O-1$. Equation (3.4) is used in order to compute Equations (3.5-3.7). The process continues until all the derivatives are computed *for all weights and biases in the network*. This means the weight updates are made after one pass over the entire data set (i.e. there will be 1 weight update in a single epoch):

$$w_{j^\ell k^{\ell-1},new} = w_{j^\ell k^{\ell-1},old} - lr \frac{\partial L}{\partial w_{j^\ell k^{\ell-1},old}}. \tag{3.8}$$

Here $w_{j^\ell k^{\ell-1},new}$ is the new weight between the j^{th} neuron in layer ℓ and the k^{th} neuron in layer $\ell-1$, lr is the learning rate, and $\frac{\partial L}{\partial w_{j^\ell k^{\ell-1},old}}$ is the derivative of the loss function with respect to $w_{j^\ell k^{\ell-1},old}$. The learning rate is a specified constant that represents how much of the derivative is

actually used to update the weight. In numerical analysis literature it is referred to as the step size [9].

In SGD, the process described above is executed for b randomly sampled data points from the set, called a batch (i.e. there is one weight update per batch and $\frac{M}{b}$ batches per epoch):

$$w_{j^\ell k^{\ell-1},new} = w_{j^\ell k^{\ell-1},old} - lr\frac{\partial L_b}{\partial w_{j^\ell k^{\ell-1},old}}, \tag{3.9}$$

where $\frac{\partial L_b}{\partial w_{j^\ell k^{\ell-1},old}}$ is the derivative of the loss function computed over b randomly sampled data points with respect to the weight between the j^{th} neuron in layer ℓ and the k^{th} neuron in layer $\ell - 1$, $w_{j^\ell k^{\ell-1},old}$. One advantage of SGD is we do not have to compute the gradients of the network activation for every point in the data before weight updates are made. Rather we compute the network gradients for b data points before an update is made. Consequently, SGD converges faster than vanilla gradient descent. Because SGD does more frequent updates, the variance is higher. The leads to fluctuations in the loss as training proceeds and the potential to move to new local minimums on the object surface. This also leads to convergence complications as SGD will overshoot the minimum. However, it has been shown that when the learning rate is decayed SGD shows the same convergence behavior as vanilla gradient descent [11].

The training process described above is executed iteratively until the loss function converges to a minimum. During iteration it is important to check for overfitting as well as convergence. Overfitting occurs when the optimization of the loss function converges to a point that is specific to the data set rather than the general trend of the phenomena under consideration. One way to monitor overfitting is to reserve a subset of the data, a validation set, that will not be used in the optimization step of the training process. While iterating, the loss function is reported for each point in the training data set and the validation data set. Evidence of overfitting is said to occur when the loss reported for the validation data points is increasing while the loss reported for the training data points is decreasing.

3.3 Activation Functions

The concept of activation functions was mentioned briefly in Section (3.1). Their usage is seen in Equation (3.1). There are many possible activation functions, each with its own benefits and drawbacks. For example the sigmoid activation function has the form:

$$\sigma = \frac{1}{1 + e^{-x}}. \tag{3.10}$$

Neurons activations that possess this activation function are bound to the interval $[0, 1]$. This is widely used in the field of reinforcement learning when the network outputs probabilities or characteristics of Gaussian distributions.

We discuss two activation functions in this thesis: the exponential and the rectified linear activation functions. The exponential was chosen during preliminary testing for this work where it was found to outperform other typical activation functions for this problem. The rectified linear activation function was selected due to its ability to approximate highly nonlinear functions while minimizing additional model complexity. More details are provided in the following section.

3.4 Rectified Linear 1D Lagrange Interpolation Polynomials

The goal of this section is to show that ANNs using rectified linear (ReLU) activation functions can replicate continuous piecewise linear (CPWL) interpolation polynomials. This is useful for problems of this type since often times the functional form of the phenomena being modeled is not known and can have high degrees of nonlinearity. The ReLU activation function can be defined as:

$$f(x) = max(0, x). \tag{3.11}$$

More details of ReLU functions and their ability to replicate interpolation polynomials are given in [10]. Define an interpolation scheme as a set of CPWL polynomials:

$$f(x) = \sum_{i=1}^{i=N} P_i(x) f_i, \tag{3.12}$$

where $P_i(x)$ is the i^{th} first order Lagrange polynomial. If $x \in [x_{i-1}, x_{i+1}]$ and $N = 3$ Equation (3.12) can be written as:

$$f(x) = P_{i-1}(x)f_{i-1} + P_i(x)f_i + P_{i+1}(x)f_{i+1}, \tag{3.13}$$

where

$$P_{i-1}(x) = \frac{x_i - x}{x_i - x_{i-1}}; x \in [x_{i-1}, x_i],$$

$$P_{i-1}(x) = 0; otherwise,$$

and

$$P_i(x) = \frac{x - x_{i-1}}{x_i - x_{i-1}}; x \in [x_{i-1}, x_i],$$

$$P_i(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i}; x \in [x_i, x_{i+1}],$$

$$P_i(x) = 0; otherwise,$$

and finally,

$$P_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}; x \in [x_i, x_{i+1}],$$

$$P_{i+1}(x) = 0; otherwise.$$

Suppose

$$\hat{P}_i(x) = \frac{1}{h_{i-1}} ReLU(x - x_{i-1}) - (\frac{1}{h_{i-1}} + \frac{1}{h_i}) ReLU(x - x_i) + \frac{1}{h_i} ReLU(x - x_{x_{i+1}}),$$

where

$$h_i = x_{i+1} - x_i; x \in [x_{i-1}, x_{i+1}].$$

It's trivial to see:

$$\hat{P}_i(x) = 0; x \leq x_{i-1}.$$

Furthermore, $\hat{P}_i(x)$ increases linearly for $x_{i-1} < x < x_i$. Evaluating at x_i takes a little algebra.

$$\hat{P}_i(x_i) = \frac{x_i - x_{i-1}}{h_{i-1}} - (\frac{0}{h_{i-1}} + \frac{0}{h_i}).$$

When h_{i-1} is inserted we get the 1 we are looking for. If $x_i < x < x_{i+1}$, then $\hat{P}_i(x)$ decreases linearly. Next, we evaluate at x_{i+1}.

$$\hat{P}_i(x_{i+1}) = \frac{x_{i+1} - x_{i-1}}{h_{i-1}} + \frac{x_i - x_{i+1}}{h_{i-1}} + \frac{x_i - x_{i+1}}{h_i},$$

$$\hat{P}_i(x_{i+1}) = \frac{x_i - x_{i-1}}{h_{i-1}} - 1,$$

$$\hat{P}_i(x_{i+1}) = 0.$$

Lastly, we require $\hat{P}_i(x) = 0; x > x_{i+1}$.

$$\hat{P}_i(x) = \frac{x - x_{i-1}}{h_{i-1}} + \frac{x_i - x}{h_{i-1}} + \frac{x_i - x}{h_i} + \frac{x - x_{i+1}}{h_i},$$

$$\hat{P}_i(x) = \frac{x_i - x_{i-1}}{h_{i-1}} + \frac{x_i - x_{i+1}}{h_i}.$$

If

$$\frac{x_{i+1} - x_i}{h_i} = 1,$$

then

$$-1\frac{x_{i+1} - x_i}{h_i} = -1 * 1 = -1 = \frac{x_i - x_{i+1}}{h_i}.$$

Thus,

$$\hat{P}_i(x) = \frac{x_i - x_{i-1}}{h_{i-1}} + \frac{x_i - x_{i+1}}{h_i} = 1 - 1 = 0.$$

This means that any first order interior Lagrange interpolation polynomial can be written as the sum of three ReLU neurons. Next we consider a polynomial who's support point lies on the left edge of the 1D mesh:

$$\hat{P}_i(x) = \frac{1}{h_i} ReLU(x_{i+1} - x_i); x \in [x_i, x_{i+1}].$$

It is trivial to see that $\hat{P}_i(x) = 1; x = x_i$, $\hat{P}_i(x)$ linearly decreases for $x_i < x < x_{i+1}$, and $\hat{P}_i(x) = 0; x \geq x_{i+1}$. Finally we consider a polynomial who's support point lies on the right boundary of the 1D mesh:

$$\hat{P}_i(x) = \frac{1}{h_{i-1}} ReLU(x - x_{i-1}); x \in [x_{i-1}, x_i].$$

Again the following evaluations are trivial; $\hat{P}_i(x) = 0; x \leq x_{i-1}$, $\hat{P}_i(x)$ linearly increases for $x_{i-1} < x < x_i$, and $\hat{P}_i(x) = 1; x = x_i$. As a consequence a 1D, 1^{st} order Lagrange polynomial interpolation scheme with M intervals can be represented as one layer in an ANN with $3M - 1$ rectified linear neurons.

Another important thing to see is inside the ReLU function evaluations is an arithmetic operation. Consequently, the neural net must have the form

$$f(x) = \sum_{i=1}^{i=N} \alpha_i ReLU(w_i x + \beta_i),$$

where N is the number of neurons, α_i is the weight between the output and the i^{th} neuron in the hidden layer, β_i is the bias applied to the i^{th} neuron the hidden layer, and w_i is the weight between the input and the i^{th} neuron in the hidden layer.

The discussion above establishes that an ANN can replicate an M interval CPWL interpolation scheme using $3M - 1$ neurons with ReLU activation functions. Furthermore during the training process the optimization step must be allowed to tune the biases as well as the weights of the network. Piecewise interpolation schemes are attractive because they are capable of approximating highly nonlinear functions without knowledge of the functional form. Furthermore, linear combinations of 1^{st} order polynomials are readily understood, keeping the models complexity in check.

4. IMPLEMENTATION OF ANN PERFORMANCE MODELS

* In this section, we develop a simple, readily understood, ANN based sweep time model. Rather than employ black box ANN methods we demonstrate that the ANN is performing the same mathematics as the linear-combination grind time model. After this is established we add modest complexity to the ANN model in an attempt to improve predictive performance. Lastly, ANN based models are developed for memory usage. Several architectures are developed and tested.

4.1 Grind time models

Artificial Neural Networks were introduced in Section (3.1). In it, the structure of the neurons and layers and their relationship with stochastic gradient descent was discussed. Algorithms to develop optimal network architectures are now a topic of intense research. Rather than using a "black box" design we employ networks executing readily understood mathematics and modest numbers of degrees of freedom. First we ensure that our ANN can do no worse than Equation (2.11). We use Equation (2.11) instead of Equation (2.12) because we are training an ANN to predict the grind time. This is different than the goal of using the method of least squares described in Section (2.3) where the objective was to find the T terms. One can construct a simple ANN whose output has the form

$$T_{grind}^{linear}(\vec{x}) = \sum_{i=1}^{i=7} x_i w_i + b \; . \tag{4.1}$$

where the inputs $\{x_i\}$ are the row vectors of the matrix form of Equation (2.11) and the constants $\{w_i\}$ are the ANN's estimate of the T terms themselves. Verifying that this simple ANN reproduces the results obtained from Equation (2.11) lets us know that the ANN is performing the mathematics we think it is. The next step is to add modest complexity to capture any non linearities that may be present in the data. The proposed "nonlinear" ANN is a combination of two quantities:

*Reprinted with permission from "Artificial Neural Network Performance Models for Parallel Particle Transport Calculation" by James D. Herring, 2021. M&C 2021, Copyright 2021 by M&C 2021.

(1) a linear combination of inputs as in Equation (2.11), and (2) a nonlinear function of a potentially different linear combination of inputs with a different $\{w_i\}$ and b. The nonlinear function is applied by a layer of neurons that apply rectified linear (ReLU) functions [10], which can generate piecewise linear approximations to any function, as discussed in Section (3.4). All nonlinear ANNs in this study used $M = 5$ intervals. This is seen in Figure (4.1) and Equation (4.2).

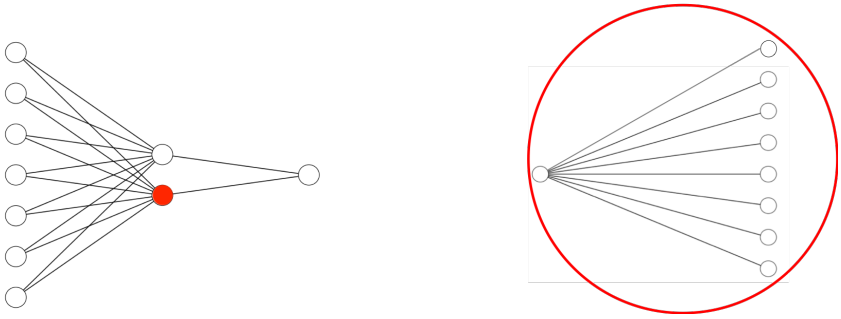

Figure 4.1: ANN that adds a nonlinear correction to the original linear model. Left: simplified view. Right: detail "inside" the red neuron for the example of 7 ReLU neurons.

$$T_{grind}(\vec{x}) = w_1^{out}\left(\sum_{i=1}^{i=7} w_i^{in} x_i\right) + \sum_{j=1}^{j=3M-1} w_j^{out} ReLU\left(w_j^{red}\left[\sum_{k=1}^{k=7} \tilde{w}_k^{in} x_k\right] + b_j\right) , \qquad (4.2)$$

4.2 Implementation of ANNs

To implement the ANNs Tensorflow [13] and Keras are employed. Each layer makes use of the keras.layer.Dense() function. In the case of the nonlinear ANN described above, the linear neuron in the hidden layer is a Dense layer connected to the inputs, while the second nonlinear neuron is two Dense layers. A Dense layer, denoted layer α here for simplicity, is connected to the inputs, and a Dense layer with ReLU activation functions is connected to the layer α. Then the ReLU layer and the linear layer are concatenated using keras.layers.Concatenate(). The output layer is then connected to the concatenated layer. Lastly the Adam optimizer is used with a learning rate of

$1e^{-3}$, which determines how far (in the direction negative of the gradient) the network parameters are moved from iteration to iteration.

To train the networks the batch size is set to 30% of the number of training data points. All ANNs use either mean percentage error as the loss function or mean squared error. Training executes for 30×10^6 epochs with an early stopping criterion of 1×10^6 epochs.

The model is saved via .hdf5 format using keras.callbacks.ModelCheckpoint(). Each time the loss reaches a new minimum the weights are saved, not to be overridden until a new minimum is observed. Validation data is assessed by loading the trained model and passing out of sample data points into the network. Then the output is measured against the validation data sets' known answers.

4.3 Memory usage models

In addition to sweep time models we devise memory-usage models as well. The inputs to this model are chosen based on our knowledge about what the code needs to store and what variables determine the size of the largest arrays. The inputs used take the following form:

1. N_{cells}^{norm}. This input is chosen because there are many quantities stored for every cell, including geometric information, cross sections, and matrices.

2. $D_v N_{cells}^{norm} N_g^{norm}$. Here D_v = the average number of spatial degrees of freedom per cell. This input combination is chosen because some quantities, such as a fixed isotropic source strength, are stored in the code by spatial degree of freedom and energy group.

3. $D_v N_{cells}^{norm} N_g^{norm} N_m^{norm}$. This input is chosen because some quantities, such as the scattering source strength, are stored in the code by spatial degrees of freedom, energy group, and scattering moment.

4. $4C_{plane} N_n^{norm} N_g^{norm}$. Here C_{plane} is the normalized number of cells in the sweep plane. This input was chosen because information such as the number of incident fluxes being stored depends on the number of angles, the number of energy groups, and the number of cells in the sweep plane.

5. $4C_{plane}A_n^{norm}N_g^{norm}$. This input was chosen because boundary information requires storage proportional to the number of cells on the boundary, the number of directions, the number of groups, and the number of spatial degrees of freedom on the boundary face. The total number of angles is needed here because PDT stores outgoing information as well as incoming.

6. $\frac{N_n^{norm}N_{cells}^{norm}}{A_n^{norm}A_{cells}^{norm}}$. Here the first fraction is the normalized number of angle sets and the second is the normalized number of cell sets. This input is chosen because the information communicated to downstream cells depends on the number of angle sets and the number of cell sets.

7. $2N_g^{norm}\sqrt{N_m^{norm}}$. This input is chosen because of the term $\mathbf{S}_{i,g'\to g,n}Y_{n,m,d}\vec{\phi}^{i,\ell}_{n,m,g'}$ in Equation (1.40).

The superscript $norm$ indicates that the variables were normalized by dividing the quantity by a 'typical' value. We chose the following:

1. $N_{cells}^{norm} = \frac{N_{cells}}{1000}$
2. $N_g^{norm} = \frac{N_g}{10}$
3. $N_m^{norm} = \frac{N_m}{15}$
4. $N_n^{norm} = \frac{N_n}{200}$
5. $A_{cells}^{norm} = \frac{A_{cells}}{27}$
6. $A_n^{norm} = \frac{A_n}{25}$

Additionally, the units of the memory usage data was switched from mega-bytes to giga-bytes.

Normalized inputs were used to ensure that the inputs and outputs of the data set were both $O(1)$ such that the weights of the network would also be $\approx O(1)$. If this normalization isn't performed the weights are orders of magnitude smaller than the loss function evaluations resulting in massive gradients, causing oscillations in gradient descent. Figure 4.2 shows the loss function

evaluations as a function of epoch number for a minimization process using normalized and unnormalized variables.

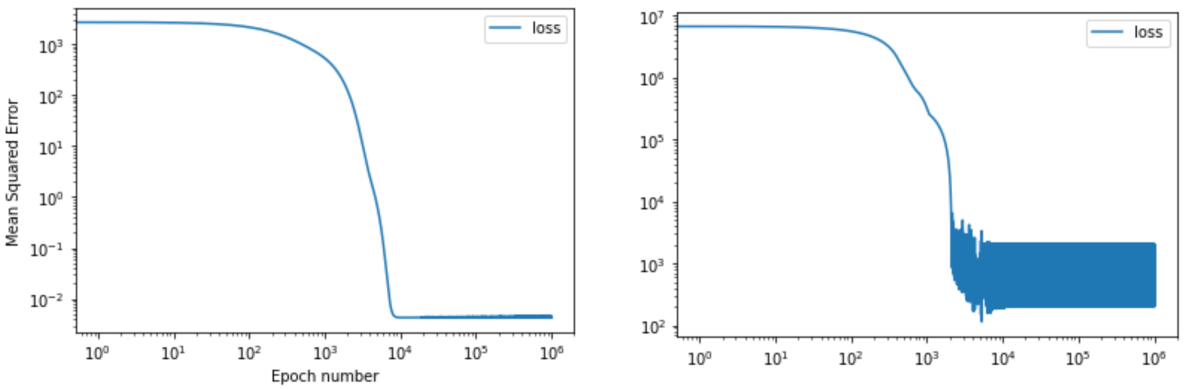

Figure 4.2: Mean squared error vs epoch number. Left: Normalized variables. Right: Unnormalized variables and data

The first ANN tested has the form:

$$M_{usage}(\vec{x}) = w_1^{out}\left(\sum_{i=1}^{i=7} w_i^{in}x_i + b\right) + \sum_{j=1}^{j=3M-1} w_j^{out} ReLU\left(w_j^{red}\left[\sum_{k=1}^{k=7}\tilde{w}_k^{in}x_k\right] + \tilde{b}_j\right) \quad , \qquad (4.3)$$

where the $\{x_i\}$ represent the 7 inputs defined above. Figure (4.3) provides visual clarity. This structure was motivated by the results seen in the nonlinear grind time model, characterized by Equation (4.2). The other ANN considered is a linear combination of the seven normalized inputs developed above. These ANN structures are compared in Section (5).

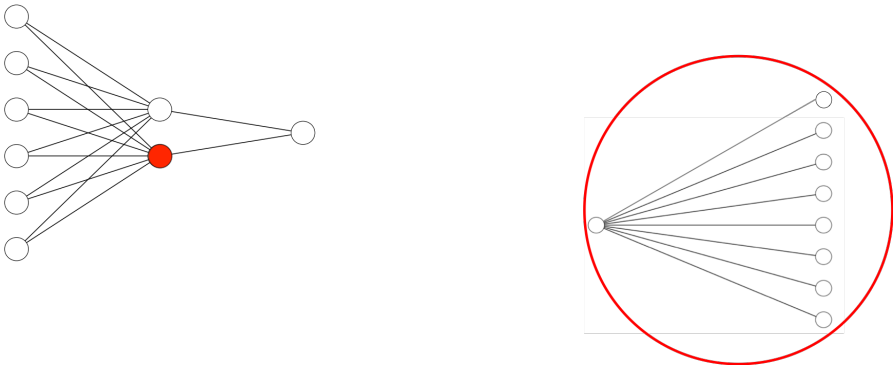

Figure 4.3: ANN structure for PDT ReLU memory usage model. Left: simplified view. Right: detail “inside" the red neuron.

5. RESULTS

*

5.1 Replicating PDT's Current Sweep Time Model

To begin we used an ANN to reproduce the results of the readily understood mathematics of the linear least squares grind time model described by Equation (2.11). This establishes that the machinery used is doing the mathematics we think it is. Furthermore, the nonlinear ANNs proposed here contain a linear combination model within them. Consequently, the nonlinear Tensorflow models can do no worse (on the training data) than the linear model, if the optimization process is executed correctly. Table (5.1) reports the T coefficients from performing the Method of Least Squares on Equation (2.11) and the $\{w_i\}$ coefficients from performing SGD with the linear ANN characterized by Equation (4.1) using the L1 norm of the squared difference as the loss function, discussed in Section (3.2). The data set used here is the 384 serial data set described at the beginning of Section (2.2). The small differences between the two can be explained by the iterative nature of SGD. The software used to minimize functions via SGD is Tensorflow. The convergence criterion used is known as "early stopping". This routine will stop SGD if the loss function evaluations do not decrease within a user specified number of epochs. Consequently, obtaining a precise minimum becomes non trivial. Here, training consisted of 30×10^6 epochs with an early stopping criterion of 1×10^6 for all ANNs. The differences observed in Table (5.1) are small enough to give us confidence that the ANN is doing the mathematics we expect it to.

5.2 Details of Grind Time Model Testing for Parallel Runs

We observed previously that the linear least-squares performance model is reasonably accurate for serial problems, and thus for serial problems there is little motivation to use ANNs to improve on the linear model. However, when we consider parallel problems, which are of much greater

*Reprinted with permission from "Artificial Neural Network Performance Models for Parallel Particle Transport Calculation" by James D. Herring, 2021. M&C 2021, Copyright 2021 by M&C 2021.

Table 5.1: **Time constants produced from the linear ANN and the linear least squares (LS) model.**

Time Constant	Linear LS model produced constants (ns)	Linear combination ANN constants (ns)
$T_{latency}$	8320	8330
T_{db}	-4	-5
T_{wf}	4604	4586
T_{cell}	1664	1659
T_n	138	142
T_g	155	160
T_m	6	6

practical interest, we find that the linear least-squares model performs relatively poorly, as we shall show below. We began our study of nonlinear ANNs by developing a dataset that represented the space of *realistic* parallel PDT problems (see Section (2.2)). Recall from Section (2.2) that within each processor arrangement, there are 5 problem suites that vary the number of cells per processor and the number of angles. Furthermore, each suite contains variants that vary the number of scattering moments and groups. Across all processor arrangements there are 256 total variants totaling to 3425 individual PDT runs. The dataset was split randomly 5 times into training and testing subsets. 51 of the 256 variants (80/20 split) were selected at random to be included in the testing set. The other 205 variants served as the training set. The model is trained and tested on each of the 5 randomly sampled subsets.

5.3 Performance of Grind Time Models

In this section we present and discuss the performance of the linear model and the $M = 5$ nonlinear model on the training and testing suites described in the previous section. We will see that the nonlinear model reduces prediction error relative to the linear model, but the nonlinear model still produces large errors on some problems. Because both models are built, at least in part, from our expectation about how PDT runs, the source of these large errors is identified as a discrepancy between our expectation and reality. Despite these large errors both models show signs of fruitful practical use as the behavior of problems with strong grind time dependence on aggregation

parameters is captured. Lastly, the nonlinear model shows signs of being more effective as the optimization problem increases in complexity.

Table (5.2) reports the average and maximum percent errors for all 5 testing splits. The nonlinear ANN achieves significantly lower training and prediction (testing) errors relative to the linear model. Average errors are reduced by a factor of two and maximum errors by large factors. However, the nonlinear ANN still exhibits high maximum errors. The large disparity between maximum and average errors is evidence that only a small fraction of problems are giving the models trouble.

Table 5.2: **Performance of Linear and Nonlinear Grind Time Models**

Split Number	Model Type	$E^{avg}_{training}$	$E^{max}_{training}$	$E^{avg}_{testing}$	$E^{max}_{testing}$
1	Linear ANN	45%	641%	43%	557%
1	Nonlinear ANN	20%	113 %	18%	105%
2	Linear ANN	41%	567%	40%	283%
2	Nonlinear ANN	18%	213 %	21%	141%
3	Linear ANN	47%	725%	48%	512%
3	Nonlinear ANN	26%	276 %	25%	255%
4	Linear ANN	43%	540%	45%	475%
4	Nonlinear ANN	20%	196 %	22%	220%
5	Linear ANN	45%	672%	43%	581%
5	Nonlinear ANN	19%	100 %	19%	105%

Figure (5.1) shows the PDT reported grind times and the linear model's predictions of the grind times for the second testing subset. The red circles encapsulate all member problems of the same variant. For some variants, the linear model captures PDT's behavior, as seen in the first 8 variants of the top right plot in Figure (5.1). However, there are other variants where the model is unable to capture the behavior. The variance of the grind time observed, for example, in the last variant of the middle right plot in Figure (5.1) is expected by our understanding of the code. However, in other variants, such as the ones seen in the top left plot of Figure (5.1), there is little grind time change inside each red circle. This behavior, observed on LLNL's Quartz machine (which uses Intel Xeon

processors), was unexpected based on our understanding of PDT and on PDT's behavior on the IBM Blue Gene / Q machine. Consequently the linear model, which is built on our expectation about the code, is unable to capture the behavior. An interesting step for future research would be to investigate the origin of these PDT results on the Quartz architecture.

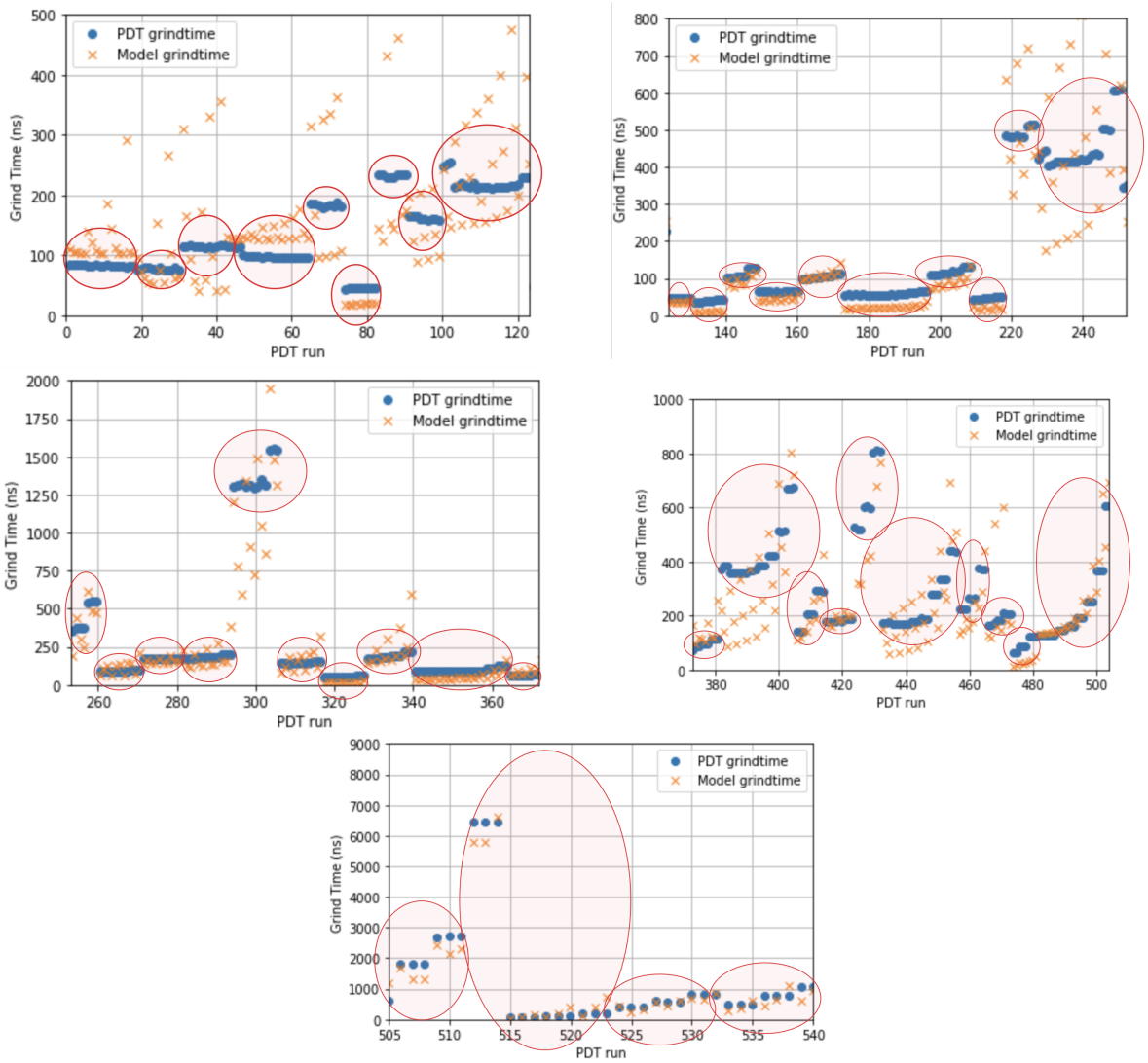

Figure 5.1: Zoomed views of linear model predictions vs PDT reported grind times for testing data from second random split. Points inside red circles are members of the same variant.

Figure (5.2) shows the PDT reported grind times and the nonlinear model's predictions of the

grind times for the testing subset in second random partitioning of the data. Again, the red circles encapsulate all member problems of the same variant. As expected, the nonlinear model captures more of PDTs behavior, but still shows errors among some variants. Again, the model misses some of the unexpected PDT grind times like the ones observed in the latter variants of the top left plot in Figure (5.2).

Despite the large maximum errors both models show signs of fruitful practical use. We have mentioned that inside each circle are the member problems unique to a specific variant in the data set. Within each variant the number of groups per group set and the scattering order is held constant while the number of cells per cell set along the z axis and the number of angles per angle set vary. Suppose we have an algorithm that selected A_n and A_z based on the model's predictions. For variants that show significant grind time dependence on A_n and A_z the algorithm would select a near optimal A_n and A_z using either model. For other variants, any choice of A parameters would lead to about the same grind time, and thus the algorithm would provide a near optimal run time in any problem in the dataset.

In reality problem parameter selection will be more complex than just selecting an optimal A_n and A_z, so a more complex ANN might be helpful. Figure (5.3) shows the predicted grind times for a nonlinear model with 4 'red neurons' from Figure (4.1) and the PDT reported grind times. Figure (5.4) shows the predicted grind times for a nonlinear model with 1 red neuron and the PDT reported grind time. Table (5.3) shows the performance statistics for the linear model, the 1 red neuron nonlinear model and the 4 red neuron nonlinear model.

Table 5.3: **Performance of Linear and Nonlinear Grind Time Models on the 1st Test-Train Split**

Split Number	Model Type	$E^{avg}_{training}$	$E^{max}_{training}$	$E^{avg}_{testing}$	$E^{max}_{testing}$
1	Linear ANN	45%	641%	43%	557%
1	Nonlinear ANN (1 red neuron)	20%	113 %	18%	105%
1	Nonlinear ANN (4 red neurons)	12%	74 %	16%	161%

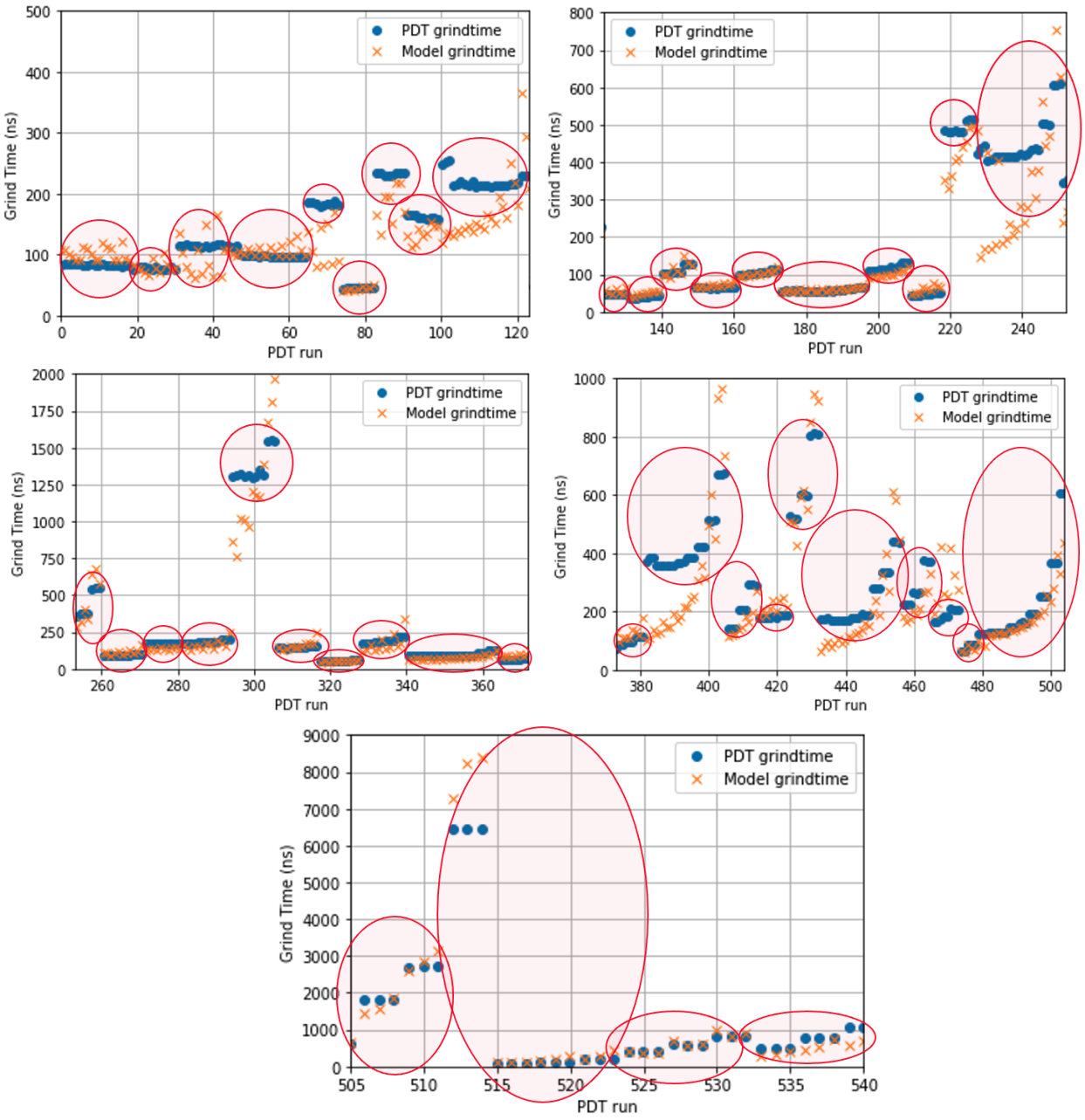

Figure 5.2: Zoomed views of nonlinear model predictions vs PDT reported grind times for testing data from second random split. Points inside red circles are members of the same variant.

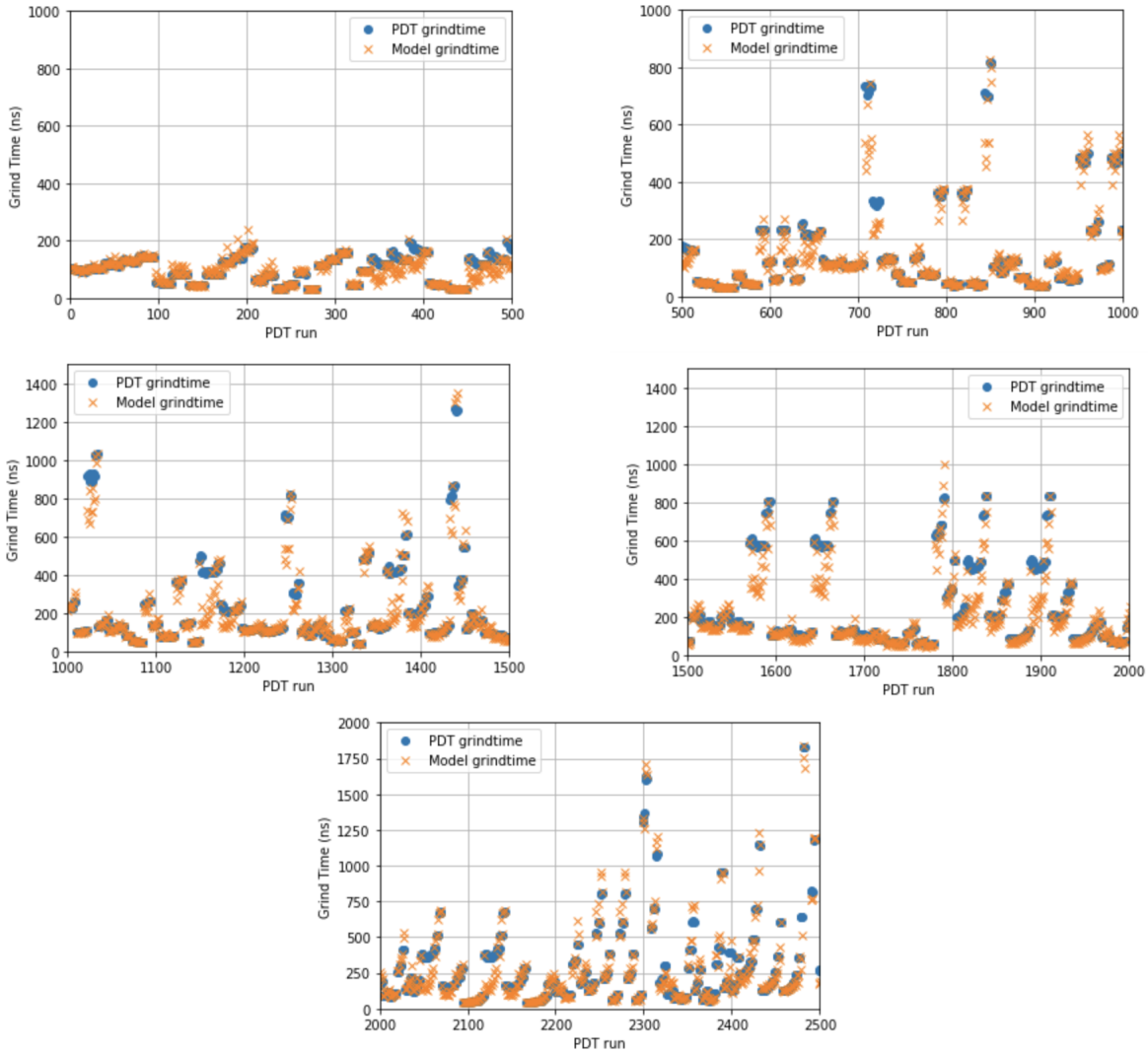

Figure 5.3: Zoomed views of nonlinear model with 4 ‘red neurons’ predictions vs PDT reported grind times for training data from first random split.

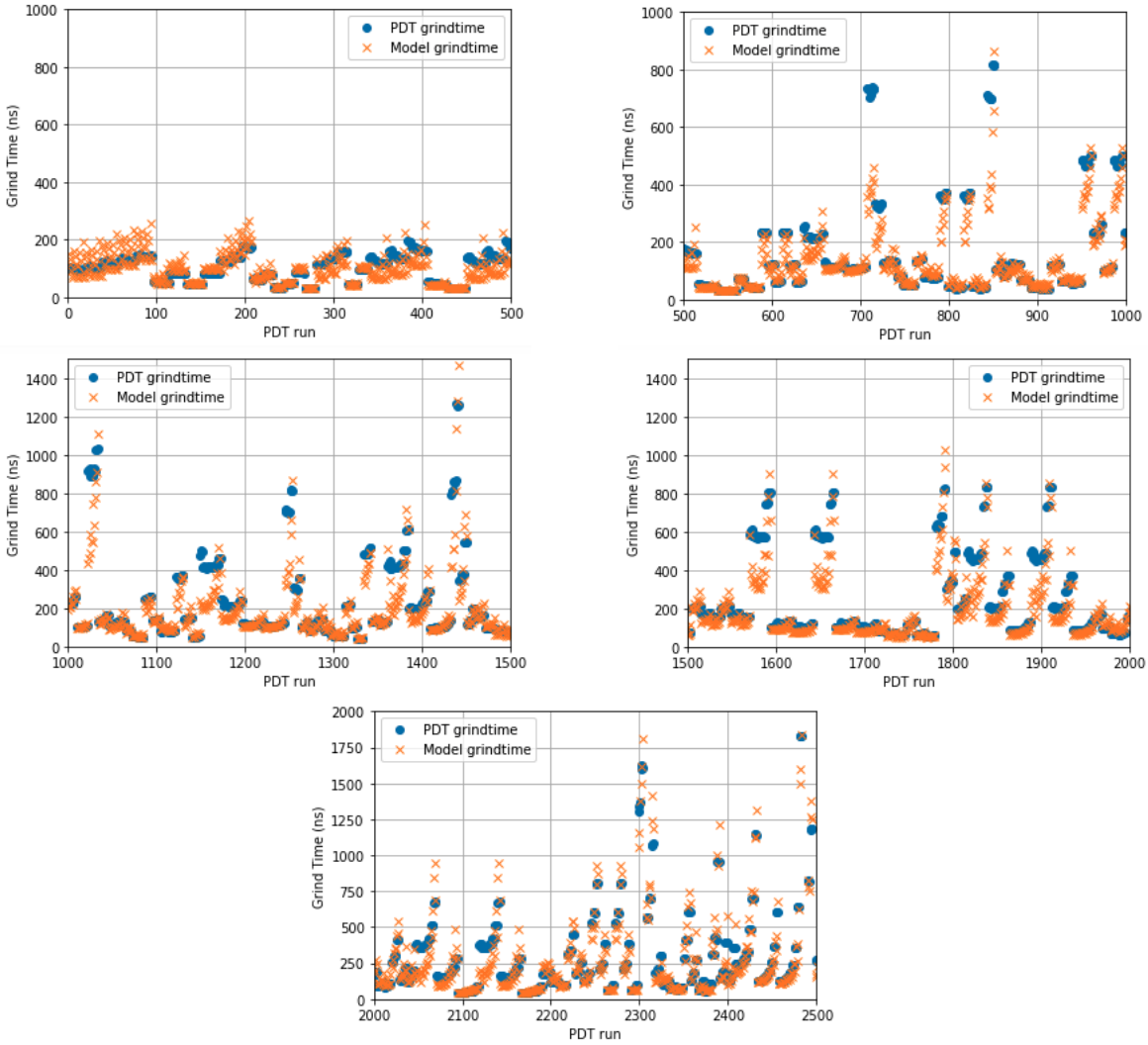

Figure 5.4: Zoomed views of nonlinear model with 1 ‘red neurons’ predictions vs PDT reported grind times for training data from first random split.

As expected the training maximum and average errors decrease, falling to 74% and 12% respectively. Interestingly the average error on the testing set also decreases to 16%. The increased maximum error of 161% could be evidence of overfitting. These results open the door for future research to find the optimal number of degrees of freedom to reduce in and out of sample prediction errors. We did not do this as our goal was to investigate the feasibility of *simple* nonlinear ANNs for predicting grind times. With this in mind, the nonlinear model shows signs of being more effective than the linear model as the optimization problem complexity increases.

5.4 ANN-Based Memory Models

Given the potential for the simple linear-plus-ReLU ANN of Figure (4.1) for predicting grind times, we applied similar architecture, seen in Equation (4.3) and Figure (4.3), to predicting PDT's memory use.

The desired use of a model influences how it should be trained. With the grind-time model, it is important to have low *percentage* error, at least in parameter ranges of interest for efficient solutions. With the memory model, it is important that the model be most accurate for the highest-memory problems, because these problems pose the risk of crashing due to insufficient memory. We therefore used mean squared error (GB^2)—not percent error—as the error metric, which gives higher importance to problems with higher memory requirements in the training set.

The linear memory model's performance is shown for the training set (left) and testing set (right) in Figure (5.5). Results are good, with average and maximum training-set errors of 19% and 51%, and with smaller errors for high-memory problems. Performance is also good on the test suite, with average and maximum errors of 6% and 35%—even lower than on the training suite.

The nonlinear memory model's performance is shown in Figure (5.6). We see what we hoped for when we added a simple correction network to the linear-model network: the nonlinear correction has reduced training and testing errors, which were already tolerably small. Table (5.4) summarizes the performance statistics for the memory models.

An interesting trend is observed in both of the right plots in Figures (5.6) and (5.5). There are clearly 6 groupings in the data. Moving from left to right within each group we see 4 subgroups that

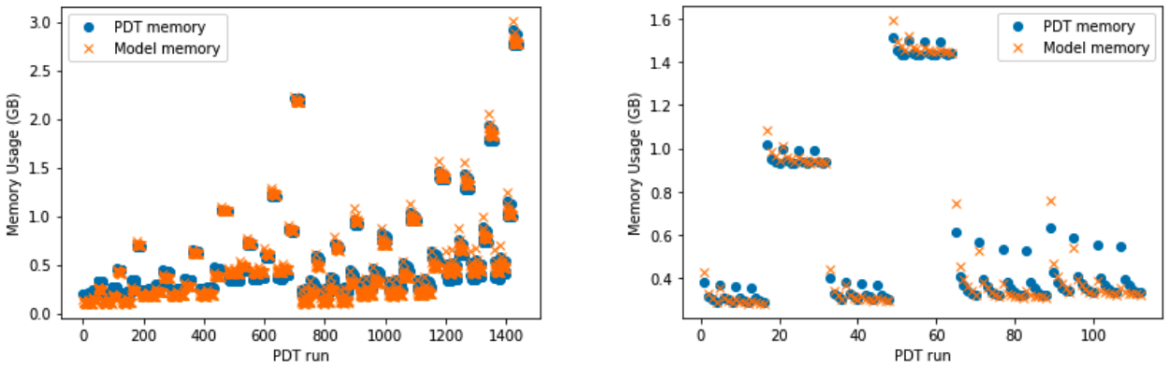

Figure 5.5: PDTs reported memory usage v Linear ANN predicted memory usage. Left: training data. Right: testing data

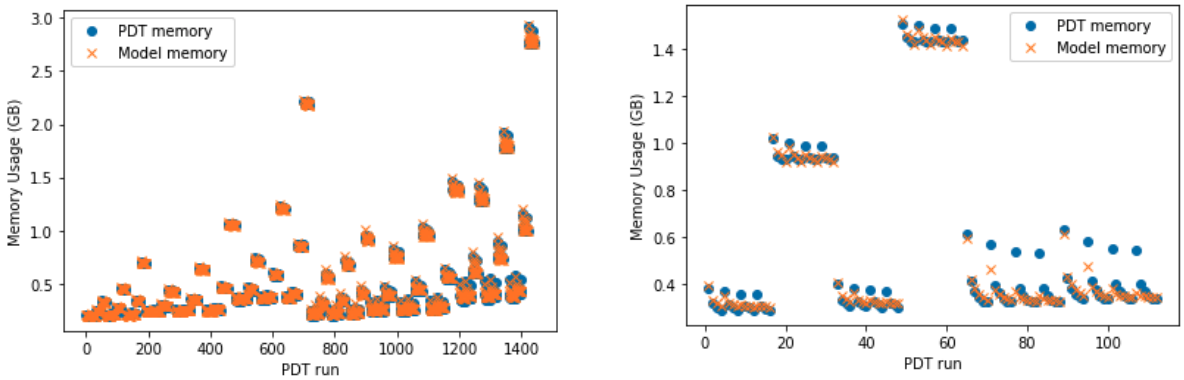

Figure 5.6: PDTs reported memory usage v Nonlinear ANN predicted memory usage. Left: training data. Right: testing data

are quadratic-like in appearance. Moving from the left most subgroup to the right most subgroup we increase A_z. Moving from the left most point within a subgroup to the rightmost point we are increasing A_n. This trend is also observed in the grind time model results, though it is weaker.

Table 5.4: **Performance of Linear and Nonlinear ANN memory models**

Model Type	$E^{avg}_{training}$	$E^{max}_{training}$	$E^{avg}_{testing}$	$E^{max}_{testing}$
Linear ANN	0.059 GB	0.13 GB	0.028 GB	0.19 GB
Nonlinear ANN	0.026 GB	0.106 GB	0.024 GB	0.16 GB

6. SUMMARY AND CONCLUSIONS

6.1 Conclusion

Our goal in this work was to explore the use of relatively simple ANNs in modeling the performance of particle-transport codes. We have presented an exceedingly simple ANN that replicates a linear analytic model characterized by Equation (2.11). Models of this form have been used to predict sweep times for many years in the PDT code [6]. For serial problems, these simple linear analytic models perform well. However for parallel problems, which are of much more practical interest, performance is relatively poor. Furthermore, we developed a slightly more complex ANN that combines the linear model and a nonlinear function of a combination of inputs, seen in Equation (4.2). Grind time models of this form do significantly reduce prediction error for parallel problems, but still post relatively high maximum errors for a handful of cases.

Despite these high errors, both models capture PDT performance behavior *when grind time is a strong function of* A_n *and* A_z. A likely application for these performance models is in an algorithm that selects the aggregation and partitioning parameters that will minimize grind time. With this in mind, these models show signs of fruitful practical performance. Further, we show that slightly increasing the complexity of our nonlinear ANNs decreases performance error on training data and average performance error on out of sample problems. However, the maximum error does seem to increase with model complexity. This could be a sign of overfitting and should be studied further. We have not attempted to make the best possible ANN for this application; rather, we have attempted to demonstrate that very simple networks can be designed that are easy to train, easy to use, and are guaranteed to outperform simple analytic models of the form of Equation (2.1). The results shown here successfully demonstrate this.

Additionally, we have explored using ANNs to predict the memory usage of particle-transport codes. Of the ANN architectures tested, the nonlinear ReLU model of Equation (4.3) demonstrated the best results, predicting memory usage within ≈ 0.02GB for out of sample data points.

Predicting the grind time and memory usage are ultimately used for two tasks:

1. To select aggregation parameters that ensure the fastest possible sweep time which will lead to the fastest execution time, for a given problem on a given machine with a given set of processors.

2. To ensure ahead of time that the memory usage will not exceed the memory limits of the machine the job is being submitted too.

With these goals in mind, the ANNs presented here are reasonable candidates for integration into an optimization tool, and our results suggest paths toward improved predictive capability.

This work has developed simple ANNs that can be readily used in transport applications. The results shown here also point the way toward improved ANN-based models if higher accuracy is desired. One avenue to pursue is identifying why, in some isolated cases, PDT execution time is independent of A_n and A_z. Such information could lead to more intuitive model input features and thus improved predictive capability. Another avenue is exploring the optimal model complexity to reduce training and testing errors.

REFERENCES

[1] Bell G.I. and Glassstone S. (1970) *Nuclear Reactor Theory,* Van Nostrand Reinhold, New York.

[2] Marvin L. Adams and Edward W. Larsen, *Fast iterative Methods for Discrete-Ordinates Particle Transport Calculations*, Progress in Nuclear Energy, Volume 40, Issue 1, DOI: 10.1016/S0149-1970(01)00023-3 2002

[3] Kowalok, Michael. (2004). *Adjoint methods for external beam inverse treatment planning*

[4] Mauricio E. Tano and Jean C. Ragusa, *Sweep-Net: An Artificial Neural Network for radiation transport solves*, Journal of Computational Physics, DOI: https://doi.org/10.1016/j.jcp.2020.109757, 2021

[5] Marvin L. Adams, *Discontinuous Finite Element Transport Solutions in Thick Diffusive Problems*, Nuclear science and engineering: the journal of the American Nuclear Society, DOI: 10.13182/NSE00-41, March, 2001

[6] M.P. Adams, et al., *Provably optimal parallel transport sweeps on semi-structured grids*, Journal of Computational Physics, DOI: 10.1016/j.jcp.2020.109234, January 2020.

[7] Mathis, Mark Michael, *A general performance model for parallel sweeps on orthogonal grids for particle transport calculations*, Master's thesis, Texas A&M University, 2000

[8] LeCun, Y., Bengio, Y. and Hinton, G. E. *Deep Learning*, Nature, Vol. 521, pp 436-444, DOI:10.1038/nature14539, 2015

[9] Y. Lei, T. Hu, G. Li and K. Tang, *Stochastic Gradient Descent for Nonconvex Learning Without Bounded Gradient Assumptions*, in IEEE Transactions on Neural Networks and Learning Systems, vol. 31, no. 10, pp. 4394-4400, Oct. 2020, DOI: 10.1109/TNNLS.2019.2952219

[10] Juncai He, et al., *ReLU Deep Neural Networks and Linear Finite Elements*, Journal of Computational Math, DOI: 10.4208/jcm.1901-m2018-0160 July 2018.

[11] *An overview of gradient descent optimization algorithms*, https://arxiv.org/abs/1609.04747, Ruder, Sebastian, 2017, June

[12] Chunyan Wang, et al., "A simple method for processing data with least squares method", Proc. SPIE 10452, 14^{th} *Conference on Education and Training in Optics and Photonics, ETOP 2017*, 104523A (16 August 2017)

[13] Abadi, Mart, et al., *Tensorflow: A system for large-scale machine learning*, 12th USENIX {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 265-283, 2016.