The role of machine learning in closing the moments of the stationary radiative transfer equation

T.R. Veurink

April 2024

1 Abstract

When solving the radiative transfer equation, it is common to take the moments against the Legendre polynomials. However, solving this set of equations is difficult because it has n variables and $n+1$ unknowns. In this paper we take a look if this gap in knowledge can be overcome with a solution in the form of machine learning. We experiment with proper normalizations and try to offer good neural network structures to circumvent not knowing 1 of the moments.

We find that neural networks with the gradients, fluxes, and moments, all give good approximations to the fifth moment. Here the gradients perform best and the moments perform worst.
Even though a normalization with m_0 did significantly improve the performance of a neural network with moments as inputs, good normalization factor were not found for the other structures.

Contents

2 Introduction

The radiative transfer equation (RTE) is a widely used equation used in many branches of science like astrophysics, heat transfer, remote sensing, and medical imaging [2]. It describes how radiation propagates through a medium. However, solving this equation can give rise to problems. It contains an integral that is not analytically solvable. Throughout scientists have found various numerical ways to solve this. Methods have been used like the Direct Simulation Monte Carlo when trying to take a probabilistic approach and the moment method when taking a deterministic approach.

In this paper, we will take a closer look at the moment method. The idea of this method is to describe this integral as the sum of many different moments. To help build intuition for what these moments are, the zeroth moment is the intensity of the radiation and the first moment is the flux. What the flux exactly means in this context will be discussed in the theoretical background.

When trying to work with the moment method there is the problem that to calculate the $n'th$ moment, we need the $n+1'st$ moment. So this system is not closed. One is always one moment short. In this paper, we will take a look if we can find a machine learning setup to accurately predict this moment. In this paper, exclusively the fifth moment will be considered.

3 Theoretical Background

3.1 Neural Networks

3.1.1 What is a neural network?

When training a neural network, one solves a big optimization problem. Here one tries to minimize the error between the predictions the network makes and the known results. By training it against known data, we hope to achieve accurate predictions on data the network has not seen before. Data the network uses to train is the train data and data the network has not seen before is the test data.

The (simplified) neural network used in this report has the shape seen in figure 3.1.1[1].

This neural network is fully connected, meaning that every circle in a layer is connected to all circles of the layers beside it. Except for the circles in the input and output layer, every one of these circles represents a linear operation. When a circle receives n inputs, it multiplies these inputs with so-called weights and adds a bias. So if you have an input vector x, the circle returns $Wx + b$. Here W is a $n \times n$ matrix and b is $n \times 1$ vector. The weights and biases are also what we optimize when we talk about optimizing a neural network. We will also

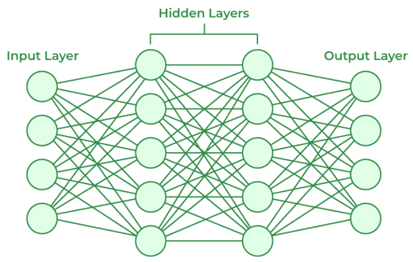

Figure 1: Layers of a neural network

call the weights and biases the parameters(θ) of the neural network. Yet, this linear approach is not sufficient for predicting a non-linear function. Therefore, we add an activation function (tanh) over the results, making the final function $tanh(Wx + b)$.

3.1.2 The optimization

We have established that we need to optimize the parameters to make accurate predictions on the test data. To understand the exact math behind the network, we will label the data used into 2 sets, the training data(Dtr) and the testing data(Dte). The training and testing data will also have a sub-class for the input and target data. This will be Dtr_i and Dtr_t or Dte_i and Dte_t respectively. To optimize our network, we try to minimize the error. We will consider the data in batches, in the code these have a size of 32 data pairs. The batch size(bs) can be considered the optimizer's jumpiness. If an optimizer is too jumpy, the neural network will never find the optimal minimum, because even when it would find it, it would jump straight out of it. However, when one uses a batch size that is too big, it barely jumps at all. This causes the optimizer to settle for the first local minimum available, while better solutions might exist. A batch size of 32 is generally considered to be good in most cases [3] and is used in this report. The error used in this report is the L2-squared error. Dte exists out of 1000 paired elements. Dtr exists out of 9000 paired elements of Dtr_i and Dtr_t. The $n'th$ pair is denoted as $Dtr_{i,n}$ and $Dtr_{t,n}$. The loss function is given by equation 1.

$$\mathcal{L}_{L2S}(k, Dtr_{i,n}, Dtr_{t,n}) = \frac{1}{bs} * \sum_{n=j}^{j+k} \frac{\sum(\mathcal{N}(Dtr_{i,n}) - Dtr_{t,n})^2}{\sum Dtr_{t,n}^2} \tag{1}$$

For $j < 9000 - k$, otherwise the equation changes to have the first sum go from j to 9000 and $\frac{1}{bs}$ becomes $\frac{1}{9000-j}$ This would only happen at the end of an iteration, as 32 does not divide 9000. An iteration through all data is called an epoch. The optimizer used in this report is a variant of the SGD optimizer described in following algorithm [3].

```
Init: Choose initial parameters $\theta^{(0)}$, batch size $bs \in \mathbf{N}$, learning rate $\eta$ and
number of epochs $k$. $K$ is the total number of epochs and r is the batchsize.
For $k$=1 up to $K$ do
    Mark entire dataset as unsampled.
    While dataset not fully marked as sampled do
        If $r$ unsampled points left in dataset then
            Sample $r$ points from dataset and mark them sampled.
        else
            Sample remaining points from dataset and mark them sampled.
            Compute $\mathcal{L}_{L2S}(\theta)$
            Set $\theta^{(k+1)} = \theta^{(k)} - \eta \nabla_\theta \mathcal{L}_{L2S}(\theta^{(k)})$
```

In this report we use a Nestorov accelerated variation of this algorithm where we use a momentum of 0.9. After the optimization has finished, we test the neural network on Dte and look at its performance. Every time we go through all the training data, we will have completed 1 epoch.

3.2 The RTE

The time-independent RTE for a gray medium in slab geometry has the form [2]:

$$v\partial_x f = \sigma_s \left(\frac{1}{2} \int_{-1}^{1} f dv - f \right) - \sigma_a f, \tag{2}$$

Where, $f = f(x, v)$ is the specific intensity of the radiation, v is the cosine of the angle between the photon velocity and the x-axis, and σ_a and σ_s are the absorption and scattering coefficients. They are assumed to be piece-wise constant functions, where these constants are taken from a uniform probability distribution. Here σ_a is made up from constants in $[0.1, 1.1]$ and σ_s is made up from constants taken from $[0.1, 20.1]$. These were used for the generation of the training and testing data.
It is common to take the moments of this differential equation against the Legendre polynomials. The k-th order polynomial is denoted as P_k. σ_a, σ_s and are dependent on x, but we will discard the (x) behind those variables for the sake of simplicity. These moments take the form:

$$m_k(x) = \frac{1}{2} \int_{-1}^{1} f(x, v) P_k(v) dv. \tag{3}$$

Now we can use Bonnet's recursion formula to derive the moment equations (4).

$$\begin{aligned}\partial_x m_1 &= -\sigma_a m_0 \\ \frac{1}{3}\partial_x m_0 + \frac{2}{3}\partial_x m_2 &= -(\sigma_s+\sigma_a) m_1 \\ &\vdots \\ \frac{N}{2N+1}\partial_x m_{N-1} + \frac{N+1}{2N+1}\partial_x m_{N+1} &= -(\sigma_s+\sigma_a) m_N\end{aligned} \tag{4}$$

With odd N. The last equation of (4) can be rewritten to:

$$m_N = \frac{N}{-(2N+1)(\sigma_s+\sigma_a)}\partial_x m_{N-1} + \frac{(N+1)}{-(2N+1)(\sigma_s+\sigma_a)}\partial_x m_{N+1}. \tag{5}$$

Here we can see an important relationship. The odd moments can be seen as a combination of the gradients of the even moments. The gradients themselves are then also dependent on the actual moment. The fluxes are the left sides of equation (4)
These equations can not be solved analytically since the $N'th$ moment is always dependent on the $N+1'st$ moment. This makes for a system of N equations and $N+1$ unknowns. In the PN-method this is solved by setting the last moment equal to zero. However, this needs a very large N to give accurate results in general. This paper researches if machine learning can offer a solution here by estimating one of the moments as a combination of other known properties of the system like the other moments, gradients or fluxes. This gives inspiration for neural networks of the following forms:

$$m_N \approx \mathcal{N}\left(m_0, m_2, \ldots, m_{N+1}\right) \tag{6}$$

$$m_N \approx \mathcal{N}\left(\partial m_0, \partial m_2, \ldots, \partial m_{N+1}\right) \tag{7}$$

$$m_N \approx \mathcal{N}\left(f_0, f_2, \ldots, f_{N+1}\right) \tag{8}$$

Here N is an odd integer and f_n is the flux of the $n'th$ moment.
These networks will be referenced as MO12MO1(Moment over 1 to Moment over 1), GO12MO1(Gradient over 1 to Moment over 1), and FO12MO1(Flux over 1 to Moment over 1) respectively. The results of these networks and possible optimization will be discussed later in this paper.

4 Network Architecture

4.1 The networks architecture

We will be looking at neural networks in the form of neural networks (6), (7) and (8). However, these might not be completely optimized yet, since no normalization is taking place. A method of normalization that has been used before in [2] is division by m_0. So as a starting point for designing the neural networks, we use the form:

$$\frac{m_N}{m_0} \approx \mathcal{N}\left(\frac{m_2}{m_0}, \frac{m_4}{m_0}, \ldots, \frac{m_{N+1}}{m_0}\right) \tag{9}$$

$$\frac{m_N}{m_0} \approx \mathcal{N}\left(\frac{\partial m_0}{m_0}, \frac{\partial m_2}{m_0}, \dots, \frac{\partial m_{N+1}}{m_0}\right) \tag{10}$$

$$\frac{m_N}{m_0} \approx \mathcal{N}\left(\frac{f_0}{m_0}, \frac{f_2}{m_0}, \dots, \frac{f_{N+1}}{m_0}\right) \tag{11}$$

These networks will be referenced as MOM2MOM(Moment Over Moment to Moment Over Moment), GOM2MOM and FOM2MOM respectively. Other interesting cases are the following:

$$\frac{m_N}{m_0} \approx \mathcal{N}\left(\frac{\partial m_2}{\partial m_0}, \frac{\partial m_4}{\partial m_0}, \dots, \frac{\partial m_{N+1}}{\partial m_0}\right), \tag{12}$$

$$\frac{m_N}{m_0} \approx \mathcal{N}\left(\frac{f_2}{f_0}, \frac{f_4}{f_0}, \dots, \frac{f_{N+1}}{f_0}\right), \tag{13}$$

$$\frac{m_N}{\partial m_0} \approx \mathcal{N}\left(\frac{\partial m_2}{\partial m_0}, \frac{\partial m_4}{\partial m_0}, \dots, \frac{\partial m_{N+1}}{\partial m_0}\right), \tag{14}$$

$$\frac{m_N}{f_0} \approx \mathcal{N}\left(\frac{f_2}{f_0}, \frac{f_4}{f_0}, \dots, \frac{f_{N+1}}{f_0}\right). \tag{15}$$

These networks will be referenced as GOG2MOM, FOF2MOM, GOG2MOG AND FOF2MOF respectively.

In this paper, we will be evaluating the neural networks by their performance in the L2 error defined as:

$$\mathcal{L}_{L2} = \sqrt{\mathcal{L}_{L2S}(k, Dte_{i,n}, Dte_{t,n})} \tag{16}$$

4.2 Determining the hyperparameters

To configure the optimal neural network, we will be using equation (9) as an initial condition. For simplicity, we will also solely be considering solving for m_5. Literature[2] found m_5 to be a moment that is not so high the solution becomes very small, but also not so small that neural networks can't approach them very well anymore. To find the best configuration for this neural network, we trained a series of them with different amounts of layers and neurons. Our methodology was to first train a neural network with various amount of layers, take the amount of layers that performed best, and then start to vary the amount of neurons per layer. This should lead to the best results with a limited amount of training. Below you can find the result of training the network with 512 neurons per layer and various amounts of layers. Here 512 was chosen because in this paper we will be estimating m_5 and the normalized variant will have 300 inputs. This makes 512 the first power of 2 which does not force the data to compress. Going over 300 also allows the network to express any additional complexity. The results of this testing you can seen in 4.2. in 4.2 we can see that 5 layers is the optimal amount. This will be the basis on which we test for the optimal amount of neurons per layer. This is plotted in 4.2. In figure:

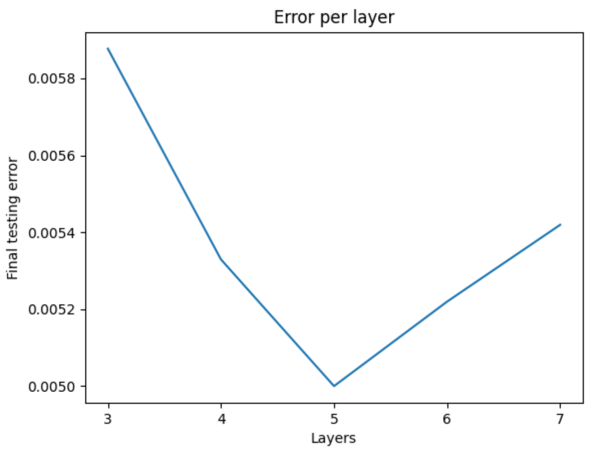

Figure 2: Error to layers

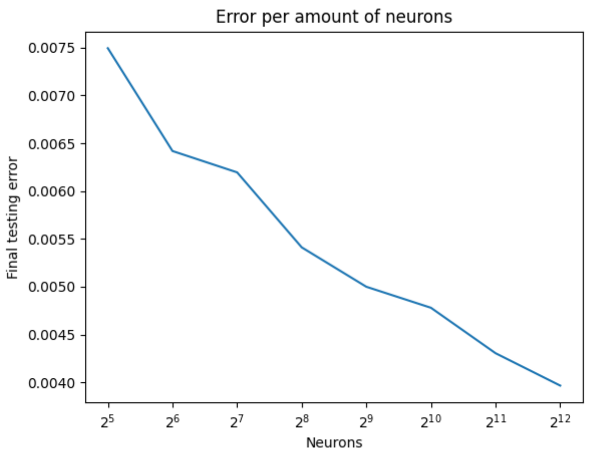

Figure 3: Error to neurons

4.2 we observe that this network benefits significantly from being wide. For computational reasons, we will train our network with 2048 neurons per layer. However, the network can most likely be significantly improved when expanded.

An important limitation of this method of determining the number of layers and neurons is that finding the optimum in 2 directions might not yield the global optimum. However, due to computational restrictions, it is a good approximation.

4.3 Power comparison

At last, we want to make an interesting observation. We found earlier that both the neural networks in equation (6) and (9) converge quite well. Given that not normalizing is the same as normalizing by ${m_0}^0$, it raises the question:
"What is the best value of x that normalizes a neural network of the form:"

$$\frac{m_N}{{m_0}^x} \approx \mathcal{N}\left(\frac{m_2}{{m_0}^x}, \frac{m_4}{{m_0}^x}, \ldots, \frac{m_{N+1}}{{m_0}^x}\right) \tag{17}$$

This question is difficult to answer and will not be answered in this paper. The problem is that when we talk about machine learning, normalization is quite closely bound to the learning rate, this causes the power of x to be dependent on the learning rate and in 3 simulations, no clear pattern was found. What makes this finding significant, is that many neural networks work with declining learning rates. This research shows that when one does this, a neural network might end up sub-optimal. In 4.3 We can see how the power develops throughout some learning rates. Over the course of this paper, we will use a learning rate

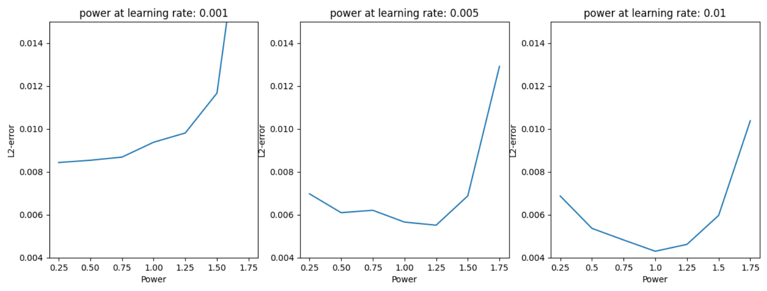

Figure 4: Error to power for different learning rates

of 0.001 and a normalization power of 1. This is normalizing with a power of 1 has been successful in the past[2] and we need a learning rate of 0.001 to be able to make proper comparisons later. Picking a learning rate of 0.01 will give normalization errors in the code. We also don't know exactly what the influence of these factors is when expanding our research to gradients and fluxes. Therefore taking a small learning rate is best for making good comparisons.

4.4 Other hyper parameters

This neural network was trained using the Python library 'pytorch' which has extensive documentation [5]. The training data for this neural network was provided by a numerical solver using the methodology of DSA preconditioned

source iteration for a DGFEM method for radiative transfer [4]. In the numerical solver, we solved for 1000 data points per test case and 10 different layers. Afterwards we took a pairwise average of the even moments to make them have an equal amount of points as the odd moment, followed by taking the average per 10 points to reduce the data to 100 data points. For the loss curves we did testing every 10 epochs.
During the training phase of the neural network, we would run the neural network for 1000 epochs. Even though the testing/training errors have not plateaued yet at this point, enough epochs have been done to give a clear insight into what machine learning method gives a lower testing error.
We used a plethora of learning rates while testing, but unless otherwise stated, 0.001 was used for everything except the neuron and layer determination, where a learning rate was used of 0.01. It has already been stated that we used the Nestorov variant of the SGD optimizer. Alternatives like the Adam optimizer and custom loss functions were tried, but none gave better results.

5 Results

We are splitting up the results into 3 sections. First, we'll evaluate our baseline by seeing how one can estimate the fifth moment using the even moments. After that, we will discuss the results of the neural networks using the gradient and eventually the fluxes.

5.1 Moments

When it comes to moments, we discuss the networks in equations (6) and (9). The results one can find in figure 5:

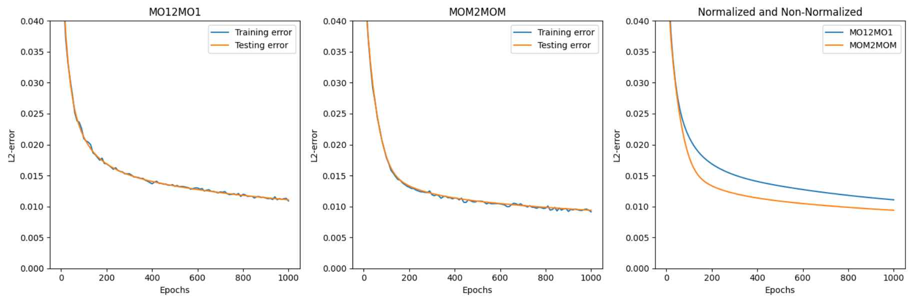

Figure 5: Error curves of MO12MO1 and MOM2MOM

We can clearly see that the NN with a normalized moment outperforms the NN without normalization. The errors on the training data and testing data are also very close, meaning that the NN regularizes well.

5.2 Gradients

Below, one can see the learning curves of the 4 gradient dependent neural networks, those of equation (7), (10), (12) and (14).

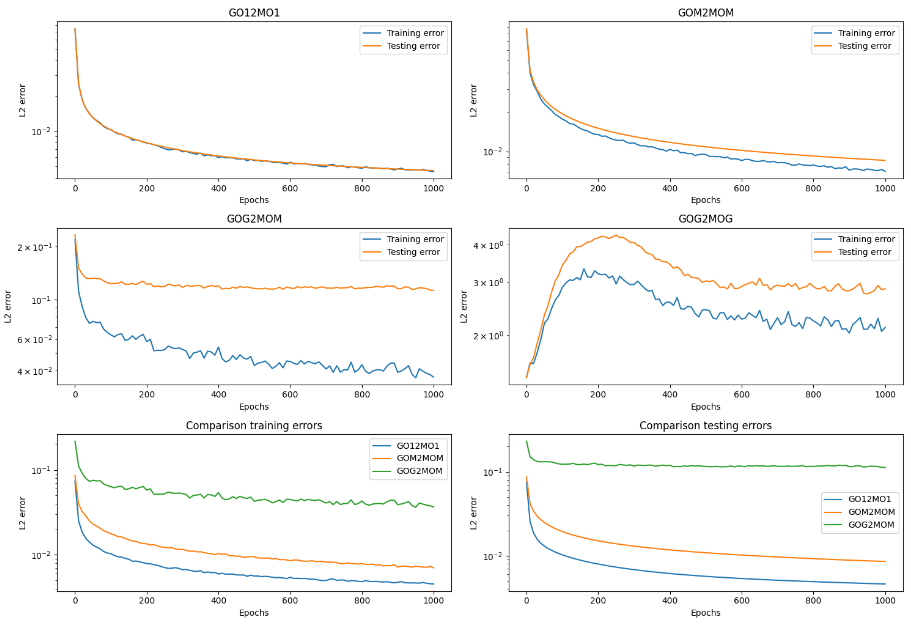

Figure 6: Error curves of estimations with the gradients

To be brief about the results of GOG2MOM and GOG2MOG. They don't converge very well. The random initialization of GOG2MOG was a better prediction than the solution it settled on eventually.
One might say that GOG2MOM might work better if the NN would be more regularized because of the big gap between the error of the training data and testing data. However, when we take a look at the comparison of training data errors of the 3 NN's, we see that even when the error of the testing data would perfectly follow that of the training data, GOG2MOM would still fall short.

5.3 Fluxes

Below one can find the results of the NN's using the fluxes as inputs, (8), (11), (13) and (15) as F2M, FOM2MOM, FOF2MOM and FOF2MOF.

The results of the NN's using the fluxes are very similar to those of the gradients. The exception is that in this case, FOM2MOM might have more potential for further decline since the tail of the neural network is declining steeply. Similar to the gradients, it is clear that no normalization works best for the fluxes

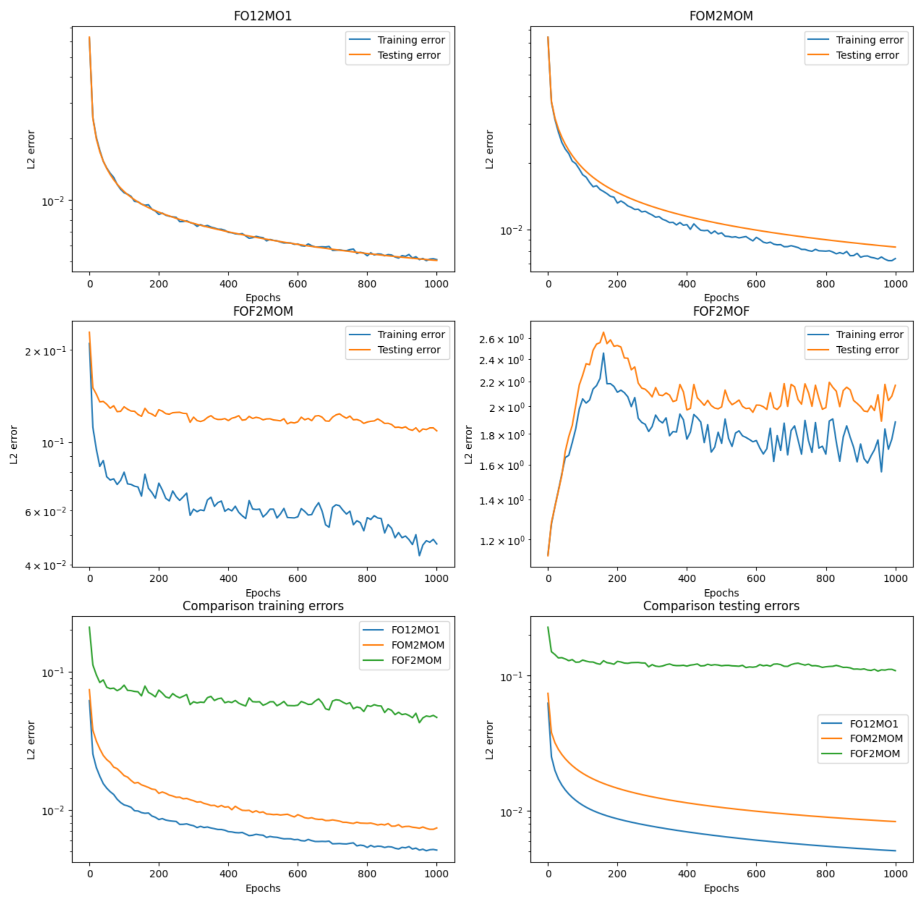

Figure 7: Error curves of estimations with the fluxes

5.4 Comparison of moments, gradients and fluxes

When comparing the NN's we decided that the regular moment, the normalized moments and the regular G2M and F2M will be compared. When plotting the curves against each other we get the following:
This data makes a lot of sense when compared to the moment's equations (4). We expect the fluxes to do best, then the gradients, and then the moments. We see that the gradients do outperform the fluxes by a small margin, but we also have to keep in mind that neural networks perform differently depending on how it was initialized, so differences can occur by chance. We also see that the normalization of the moments is very effective. It is however not effective enough to make it competitive with the gradients and fluxes.
Now we will look at the accuracy of these networks. We have been talking a lot about the L2 error, but what does that mean visually? For that reason,

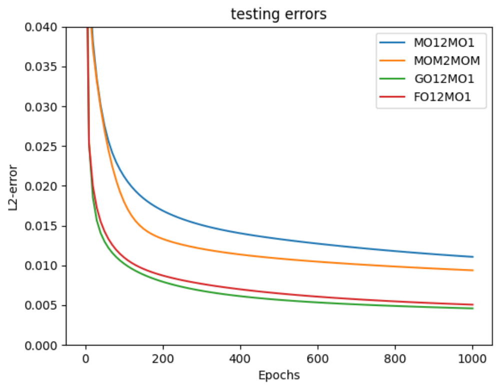

Figure 8: Comparison of error curves

we computed the median error of the testing set and found the test case that belongs to it. This was the same for all 3 NN's. That function being 5.4.
The errors are compressed on the right side of the graph because the magnitude

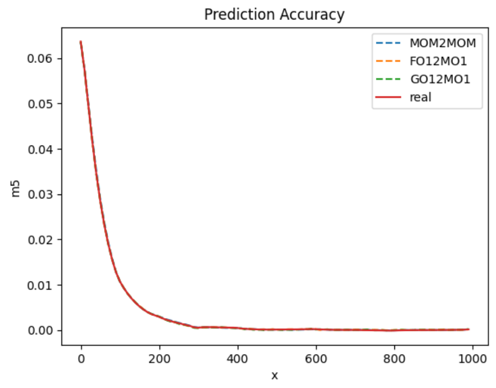

Figure 9: The fifth moment

is much bigger on the left. We can now observe these predictions where the left side is cut off. This graph is shown in 5.4.
There are a couple of things to notice here, the big thing is that GO12MO1 and FO12MO1 seem not to learn that moment values are close together and since the squared L2-error does not differentiate whether the prediction is below or above the target value, it doesn't train for it. This gives these graphs a very chaotic nature. However, one can see that the lines do follow the graph closely. Also, this might look worse than it is, because these peaks are just individual points, and if you look at the number of points that is a peak and that are

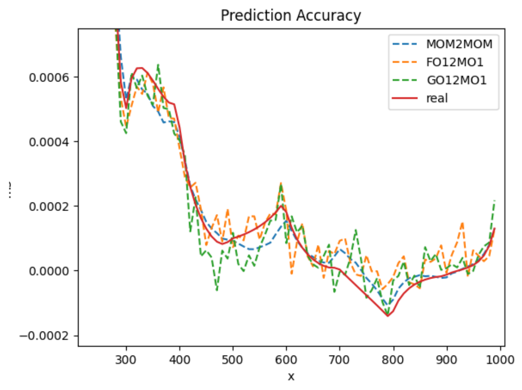

Figure 10: The right side of the fifth moment

close to the moment. there are many fewer peaks. It is also expected that the predictions of small points are worse. This is because a neural network finds it harder to approximate small values. The last thing to notice is that moments are not meant to become smaller than zero. This means that the error of the neural network to reality is primarily influenced by the error of the training data to reality. So for this model to improve, the biggest factor could be more precise training data.

6 Conclusion and Discussion

We believe that neural networks can play an important role in determining the moments when the physical properties σ_a and σ_s are unknown. M2M, MOM2MOM, G2M, GOM2MOM, F2M, and FOM2MOM all give good results, but our recommendation goes out to using MOM2MOM, G2M, and F2M. Normalization for the last 2 is not necessary since it does not significantly decrease the error. When given a more physical interpretation. When you can't learn the moment, but you can measure the gradients or fluxes, you probably also can't measure the zeroth moment used for normalization. So it is good to know that not being able to measure this will not lead to worse predictions of the odd moments.
Even though the moment has not been specifically closed in this paper, we hope that this may serve as a setup for future research.

For future research, we recommend to let the simulations run for longer and with more neurons per layer, since this seems to significantly improve the quality of the neural networks.
Another recommendation is to use dynamic normalization when it comes to the power of m_0 and the learning rate since the best optimization has been proven not to be constant.
It would also be useful to expand this paper to accommodate for other moments than just the fifth one.
It would also be interesting to see how the results in this paper stack up against the PN method both in terms of accuracy in predicting as in computational efficiency.
Another point that could improve the model is that in the end, the error was about $5 * 10^{-3}$. When squaring this, it would be $2.5 * 10^{-5}$. The code is using 32 bit floats, which gives precision to about 10^{-8}. This means that it is possible that the accuracy of the number representation has a negative effect on the training of the neural network and we recommend to train a neural network with 64 bit representation instead.

The last thing is, that this paper has delivered further proof that a normalization function can greatly improve the performance of a neural network. Yet, it is not known if the zeroth moment is even the optimal function to normalize with. It could be very interesting to see what would happen if one trains a neural network to find an optimal normalization function.

7 Acknowledgment

I want to thank Matthias Schlottbom for guiding me throughout this project and providing the numerical solver to generate the training data for the neural networks. I also would like to thank Harald Rutsch for providing the GPU that ran nearly all simulations.

References

[1] harkiran78. Artificial Neural Networks and its Applications - GeeksforGeeks, 6 2023.

[2] J. Huang, Y. Cheng, A.J. Christlieb, and L.F. Roberts. Machine learning moment closure models for the radiative transfer equation I: Directly learning a gradient based closure. *Journal of Computational Physics*, 453, 2022.

[3] Ruben Hoeksma Mengwu Guo, Tjeerd Jan Heeringa. *Non-linear optimisation and learning.* 3 2024.

[4] O. Palii and M. Schlottbom. On a convergent DSA preconditioned source iteration for a DGFEM method for radiative transfer. *Computers and Mathematics with Applications*, 79(12):3366–3377, 2020.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

8 Appendices

8.1 The python implementation

This is the code for the normalized moment, but the other neural networks had very similar architectures.

```
import torch
import torch.nn as nn
import numpy as np
import scipy.io as sio
from torch.utils.data import Dataset, DataLoader,
    random_split, Subset
import time
import random

# check if GPU is available and use it; otherwise use CPU
device = torch.device("cuda:0" if torch.cuda.is_available
    () else "cpu")
torch.set_default_device(device)


# Here we determine the properties of the dataset:
class MomentDataset(Dataset):
    def __init__(self):
        self.num_mom = 5  # Must be odd, this is the
            moment you determine
```

```
self.half_num_mom = int((self.num_mom+1)/2)   #
    Mostly used to determine the amount
self.num_train_data = 9000
self.num_test_data = 1000
data = sio.loadmat('results_3.mat')
even_m = torch.tensor(data['post_phie']).to(torch
    .float32)
odd_m = torch.tensor(data['post_phio']).to(torch.
    float32)
grads = torch.tensor(data['post_grads']).to(torch
    .float32)
fluxes = torch.tensor(data['post_fluxes']).to(
    torch.float32)

self.mode = "MOM2MOM"
chars = [*self.mode]

if chars[0] == "M":
     self.numerator_input = even_m[:(self.
         half_num_mom + 1) * 100, :]
elif chars[0] == "G":
     self.numerator_input = grads[:(self.
         half_num_mom + 1) * 100, :]
elif chars[0] == "F":
     self.numerator_input = fluxes[:(self.
         half_num_mom + 1) * 100, :]
elif chars[0] == "1":
     self.numerator_input = torch.ones(even_m[:(
         self.half_num_mom + 1) * 100, :].shape)

if chars[2] == "M":
     self.norm_fac = even_m[:100, :]
elif chars[2] == "G":
     self.norm_fac = grads[:100, :]
elif chars[2] == "F":
     self.norm_fac = fluxes[:100, :]
elif chars[2] == "1":
     self.norm_fac = torch.ones(even_m[:100, :].
         shape)

self.power = 1
self.inputs = self.numerator_input / torch.pow(
    self.norm_fac.repeat(self.half_num_mom + 1, 1)
    , self.power)
if chars[0] == chars[2]:   # Changed
     self.inputs = self.inputs[100:, :]
```

```
        self.inputs = torch.transpose(self.inputs, 0, 1)
            # We transpose so every different point is an
            input node

        if chars[6] == "M":
            self.norm_fac_output = even_m[:100, :]
        elif chars[6] == "G":
            self.norm_fac_output = grads[:100, :]
        elif chars[6] == "F":
            self.norm_fac_output = fluxes[:100, :]
        elif chars[6] == "1":
            self.norm_fac_output = torch.ones(even_m
                [:100, :].shape)

        self.outputs = odd_m[100 * (self.half_num_mom -
           1):100 * self.half_num_mom, :]  # This is the
           5th moment
        self.outputs = self.outputs / torch.pow(self.
           norm_fac_output, self.power)
        self.outputs = torch.transpose(self.outputs, 0,
           1)

        self.norm_fac_output = torch.transpose(self.
           norm_fac_output, 0, 1)

        self.n_samples = even_m.shape[0]  # Could also be
            odd_m doesn't realy matter
        self.len_input = self.inputs.shape[1]  # Used for
            number of input nodes
        self.len_output = self.outputs.shape[1]  # Used
           for number of output nodes

    def __getitem__(self, index):
        return self.inputs[index], self.outputs[index],
           self.norm_fac_output[index]

    def __len__(self):
        return self.n_samples


# Define the training and test datasets:
dataset = MomentDataset()
dataset.inputs.to(device)
dataset.outputs.to(device)
dataset.norm_fac.to(device)
```

```
train_dataset = Subset(dataset, range(dataset.
    num_train_data))
train_loader = DataLoader(dataset=train_dataset,
                          batch_size=32)    # This number
                              can be varied, but 32 is
                             generally quite good


def l2_error(x, y): # This function was provided by
    ChatGPT, but I understand and stand behind it
    squared_error = torch.sum((x - y) ** 2)
    y_squared = torch.sum(y ** 2)
    squared_error_divided_by_y_squared = squared_error /
        y_squared
    rmse_divided_by_y = torch.sqrt(
        squared_error_divided_by_y_squared)
    return rmse_divided_by_y


# Every epoch I test the NN on the test_set so I can see
    how the error develops
def verify(NN, dataset):
    global error_data
    global starttime
    test_dataset = Subset(dataset, range(dataset.
        num_train_data, dataset.num_train_data+dataset.
        num_test_data))
    test_loader = DataLoader(dataset=test_dataset)

    random_numbers = random.sample(range(0, dataset.
        num_train_data), dataset.num_test_data)
    train_test = Subset(dataset, random_numbers)
    train_test_loader = DataLoader(dataset=train_test)

    l2_errors = []
    ms_errors = []
    for inputs, outputs, norm_fac in train_test_loader:
        opt.zero_grad()
        pred = NN(inputs.to(device))
        norm_fac.to(device)
        l = loss(norm_fac.to(device)*pred, norm_fac.to(
            device)*outputs.to(device)).to("cpu").detach()
            .numpy()

        l2_errors.append(l2_error(norm_fac.to(device)*
            pred, norm_fac.to(device)*outputs.to(device)).
            to("cpu").detach().numpy())
        ms_errors.append(l)

    error_row1 = ms_errors + l2_errors
    error_row = error_row1
    l2_errors = []
    ms_errors = []
    for inputs, outputs, norm_fac in test_loader:
        opt.zero_grad()
        pred = NN(inputs.to(device))
        norm_fac.to(device)
        l = loss(norm_fac.to(device) * pred, norm_fac.to(
            device)*outputs.to(device)).to("cpu").detach()
            .numpy()

        l2_errors.append(l2_error(norm_fac.to(device)*
            pred, norm_fac.to(device)*outputs.to(device)).
            to("cpu").detach().numpy())
        ms_errors.append(l)

    error_row2 = ms_errors + l2_errors
    error_row = error_row + error_row2

    error_data.append(error_row)
    # Here we have the following errors: [mse_train,
        l2_train, mse_test, l2_test, runtime]
    return np.mean(ms_errors), np.mean(l2_errors)


# Define the NN architecture
num_neurons = 2048  # This is compressing the data, but
    even when you don't, it still gives very similar
    results
num_layers = 5
NN = nn.Sequential(nn.Linear(dataset.len_input,
    num_neurons), nn.Tanh(),  # num_mom-1+2
                    nn.Linear(num_neurons, num_neurons),
                        nn.Tanh(),
                    nn.Linear(num_neurons, num_neurons),
                        nn.Tanh(),
                    # nn.Linear(num_neurons, num_neurons),
                         nn.Tanh(),
                    # nn.Linear(num_neurons, num_neurons),
                         nn.Tanh(),
                    nn.Linear(num_neurons, dataset.
                        len_output))
```

```
NN.to(device)

# Define the optimizer and loss function:
lr = 0.001
opt = torch.optim.SGD(NN.parameters(), lr=lr, momentum
    =0.9, nesterov=True)

loss = nn.MSELoss()

# Now we start training
starttime = time.time()

error_data = []
epochs = 1000
for epoch in range(epochs):
    for inputs, outputs, norm_fac in train_loader:
        opt.zero_grad()
        pred = NN(inputs.to(device))
        l = loss(pred, outputs.to(device))/loss(torch.
            zeros((outputs.to(device)).shape), outputs.to(
            device))
        l.backward(retain_graph=True)
        opt.step()

    if epoch % 10 == 0 or epoch == epochs-1:
        mean_MSE, mean_rel_l2 = verify(NN, dataset)
        print("code2 epoch:", epoch, "mean error, 
            l2_error:", mean_MSE, mean_rel_l2, "epochtime 
            =", (time.time()-starttime)/60)

save_string = 'mom' + str(dataset.num_mom) + '_' + str(
    dataset.mode) + '_' + str(num_layers) + 'lay' + str(
    num_neurons) + 'neurons' + str(dataset.power) + 'pow'
torch.save(NN, save_string + '.pth') # Save the Neural
    Network
error_data = np.array(error_data)
np.save(save_string + '.npy', error_data)
```