Performing Loop Integration Using Neural Networks

Ruben Bentley
Level 4 Project, MPhys Theoretical Physics
Supervisor: Professor D. Maître
Department of Physics, Durham University
(Dated: April 17, 2024)

Abstract: Neural network (NN) technology was used to integrate the Feynman parameterised integral for the 1 loop process of Higgs boson pair production, from a top loop, over a phase space region. Randomly sampled phase-space and Feynman parameters were used to obtain exact integrand values, that were then fitted to the derivative of the neural network. The neural network then evaluated these integrals over the trained region $\simeq 10$ times faster than the Monte Carlo integrator pySecDec, which integrates over specific phase-space configurations. Different activation functions were applied to the neural network to further the theoretical understanding and accuracy shown in the current literature. The performance of the architectures differed, because the shapes of their activation function's derivatives affected the behaviour of their backpropagated gradients during training. The GELU based architecture was the most accurate, with a mean of 3.9 ± 0.2 correct digits over the trained phase-space region, beating the tanh-based network 3.4 ± 0.2 digits (the most accurate in the original literature). Larger batch sizes improved the accuracy of architectures, as the GELU based network obtained an accuracy of 3.4 ± 0.3 digits, when trained on a batch size 25 times smaller. Deep GELU networks were slightly less accurate 3.8 ± 0.2 correct digits. GELU based networks had better generalisation for the boundaries of the sample space, than the softsign, sigmoid, and tanh. The shape of the GELU's first derivative made it less susceptible to dead node formation, than the other activation functions tested.

Keywords: Neural Network, Activation Function, GELU, Parametric integration, Backprop

Contents

1. INTRODUCTION

Due to the transition of the Large Hadron Collider (LHC) from a resonance discoverer to a precision machine, the demand for accurate and time effective calculations has never been more necessary [1]. An enormous amount of data is collected from the high energy proton-proton collisions at the LHC. This data is used to generate Feynman diagrams (graphical representations of scattering events), and their respective Feynman parameterised integrals, which are evaluated to obtain scattering amplitudes. Different techniques are implemented to perform these tasks, with the aim of collecting statistically large and reliable datasets. Therefore, it is important to develop existing techniques or introduce alternative ones entirely, to refine theoretical predictions.

This paper forwards the investigation of a new integration technique that evaluates Feynman parameterised integrals with neural network (NN) technology [2]. Where the derivative of the neural network (derivative neural network) is trained to the integrand of the integral, and then the neural network is used to evaluate that integral. This is explained in Section 2.2 and demonstrated schematically in Figure 1.

$$\begin{tikzcd}
\mathcal{NN} \arrow[r] & Integral \\
\frac{d\mathcal{NN}}{dx_1...dx_k} \arrow[u] \arrow[r, leftrightarrow] & Integrand
\end{tikzcd}$$

FIG. 1: Schematic diagram shows how the integrand of the Feynman parameterised integrals is used to train the derivative of the network and get the integral

The parametric integrals that can be solved with this technique have the following form below:

$$I(s_1, ..., s_m) = \int_0^1 dx_1 ... \int_0^1 dx_k f(s_1, ..., s_m; x_1, ..., x_k) \quad , \tag{1.1}$$

where s_i are phase-space parameters and x_i are the auxiliary variables (Feynman parameters) that are integrated. The performance of this network architecture was compared to the PySecDec [3] Monte Carlo (MC) integrator.

The PySecDec [3] MC integrator is one of many MC integrators used in industry to find the scattering amplitudes of scattering events. It is dynamic – being able to do this for many processes, when standard model (SM) or beyond standard model (BSM) physics is applied. It trains to specific phase-space configurations, so its accuracy scales by a factor of $\frac{1}{\sqrt{N}}$ [4], where N is the number of trained samples. However, because of the enormous amounts of Feynman diagrams being generated in the proton-proton inelastic collisions at the LHC, their speed is insufficient to produce reliable and statistically large datasets in the future. The progression of physics will be technologically limited by the speed of MC integrators [5], which integrate over specific phase-spaces, as shown in Figure 2a.

In this paper, we argue how this method is a necessary alternative to the predominantly used Monte Carlo (MC) integration programs, as the NN is trained over a bounded phase-space region, seen in Figure 2b. We sacrifice the accuracy of the integrals, for the speed of their

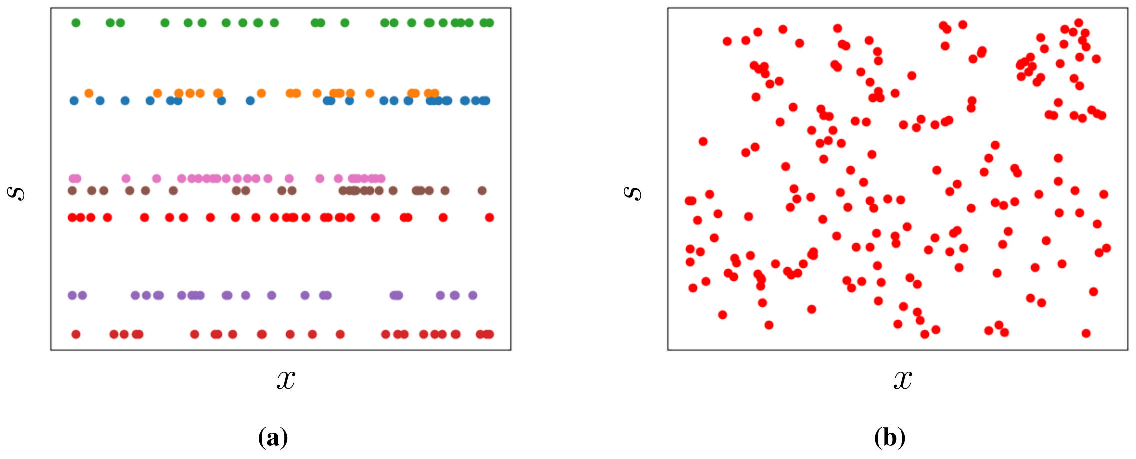


FIG. 2: (a) Schematic diagram of the MC sampling the Feynman parameters x_i for specific phase-spaces s_i. (b) Schematic diagram of the NN sampling the phase-space parameters s_i and the Feynman parameters x_i.

computation. The Feynman parameters and phase-space parameters are treated as equal contributions to network, unlike other NN based integration techniques [6–8]

Once trained, the NN can produce integrals of specified accuracy faster than the PySecDec MC integrator, since it is GPU compatible and can calculate a large number of integrals synchronously (see Table II). This NN approach lowers the strain on computational resources, resulting in an increased computational efficiency and added environmental benefits.

The preceding research paper, which introduced this architecture [2] showed superior speed to the PySecDec [3] MC integrator when solving Higgs boson pair production integrals over a phase-space region, to the same accuracy. The sigmoid and tanh activation functions were applied to the derivative neural network architecture, with the tanh outperforming the sigmoid for the 1 loop and 2 loop process.

In this study, the Gaussian error linear unit (GELU) [9], was applied to the architecture, and outperformed the classical activation functions. In addition, the softsign activation function was used to explain the relationship between the shapes of the activation function's derivatives and the networks performance, which was left for future research in the original literature [2].

Promising data is shown throughout this report to support future applications of NN technology in particle phenomenology. Neural network architectures have also been successful as event generators in the form of General Adversarial Networks (GANs) [5].

The organisation of this paper is as follows: Section 2 builds the understanding of the neural network architecture; Section 3 introduces back propagation and training; Section 4 goes over the theory of deep learning, which is required to explain and discuss the results shown in Section 5. This paper is concluded in Section 6.

1.1. The Parametric Integral

The Feynman parameterised integral, for a specific scattering event, is integrated to find the scattering amplitude of the event. The scattering amplitude describes how particles scatter for different phase-space configurations. In this project, neural networks were used to numerically integrate the Feynman parameterised integrals for Higgs boson pair production from a top quark loop. This process is shown in Figure 3, and the integral (for the 1 loop process) is:

$$I(s_{12}, s_{14}, m_H^2, m_t^2) = \int_0^1 dx_1 \int_0^1 dx_2 \int_0^1 dx_3 \frac{1}{\mathcal{F}_1^2} \quad . \tag{1.2}$$

The $\mathcal{F}_1$ is a second Synmanzik polynomial, a polynomial of the phase-space and auxiliary variable space:

$$\begin{aligned} \mathcal{F}_1 = m_t^2 + 2x_3m_t^2 + x_3^2m_t^2 + 2x_3m_t^2 - x_2s_{14} + 2x_2x_3m_t^2 \\ -x_2x_3m_H^2 + x_2^2m_t^2 + 2x_1m_t^2 + 2x_1x_3m_t^2 - x_1x_3s_{12} \\ +2x_1x_2m_t^2 - x_1x_2m_H^2 + x_12m_t^2 \quad . \end{aligned} \tag{1.3}$$

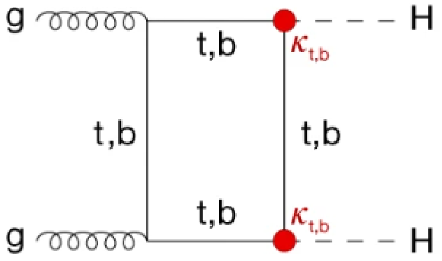


FIG. 3: Feynman diagram for Higgs boson pair production, from a virtual bottom/top quark loop. Figure 11 from the [10].

Eq. (1.2) cannot be solved analytically and was thus solved numerically, requiring the second Szymanzik polynomial $\mathcal{F}_1$ to be positive semi-definite. Therefore, a euclidean region of phase-space s_i was chosen [3] to make the second Szymanzik polynomial $\mathcal{F}_1$ positive semi-definite. This Feynman parameterised integral was also integrated in the preceding paper [2], along with the chosen phase-space Eq. (1.4), providing useful comparison:

$$m_t^2 \equiv 1 \quad , \quad -30 \leq \frac{s_{12}}{m_t^2} \leq -3 \quad , \quad -30 \leq \frac{s_{14}}{m_t^2} \leq -3 \quad , \quad -30 \leq \frac{m_H^2}{m_t^2} \leq -3 \quad , \tag{1.4}$$

where $\mathcal{F}_1 > 0$.

This region of phase-space is non-physical, so the scattering amplitudes that were calculated using this phase-space are not observed. The purpose of this investigation was to provide motivation for this next step of the research where the physical region is integrated. To integrate the physical phase-space region, the network must incorporate complex analysis, as this region may introduce poles to the integrals. Once this next step is successful, this method can be applied successfully to industry, but first its implementation needs to be tested at a simpler level of complexity.

1.2. On Higgs Pair Production

The discovery of the Higgs boson [11] at the LHC, provided experimental validation for the spontaneous symmetry breaking mechanism, that gives mass to the weak bosons. In the SM, the Higgs has a fixed set of scattering possibilities, which remain consistent and cannot be altered unless modifications beyond the standard model are introduced. The leading contribution for Higgs boson pair production, shown in Figure 3, is the loop-induced gluon-fusion process from virtual heavy quarks (predominately tops) [12]. Alternative Higgs pair-production scattering events have cross sections an order of magnitude smaller [13]. The observation of Higgs pair-production is rare, their cross sections are small comparatively to other processes [14], hence it is addressed theoretically in many papers [15, 16].
It is advantageous to transform scattering amplitude integrals from momentum space to Feynman parameter space, as it assists with numerical computation [3]. Feynman parameterisation is used to express the propagators, in the momentum-space integral, as terms in an integral over the Feynman parameters [17]. Allowing the momentum space to be integrated out, leaving the integration over the Feynman parameters:

$$I_i = \int_0^1 \sum_{j=1}^{N-1} dx_j x_j^{v_j - 1} \frac{\mathcal{U}_i(\overrightarrow{x})^{expo\mathcal{U}(\epsilon)}}{\mathcal{F}_i(\overrightarrow{x}, s_{ij})^{expo\mathcal{F}(\epsilon)}} \quad . \tag{1.5}$$

This is the form of a Feynman parameterised integral, shown in [3], with L loop and N propagators raised to the power of v_i. This method of integration can be applied to scattering processes with a Feynman parameterised integral of this form. The $\mathcal{U}$ is the first synmanzik polynomial, unlike the second Eq. (1.3), it is a polynomial of only the Feynman parameter space.

Feynman parameterised integrals are versatile, being applicable to various other fields of physics, such as gravitational wave physics [18].

2. ARCHITECTURE OVERVIEW

2.1. Deep Feedforward Networks

For this technique, deep feedforward neural networks (DNN) were implemented to calculate the integrals. The structure for the deep feedforward network is shown below in Figure 4. The network's structure is equivalent to a directed acyclic graph (DAG), because there are no loops in the graph and all the connections between the nodes are directed. The DAG structure assists the forward propagation of the network. In forward propagation, the information travels from the input layer, through the hidden layers and finally to the output layer.
At each node the incoming data undergoes a weighted sum Eq. (2.2), and is non-linearised with an activation function ϕ, Eq. (2.3). The activated node's output is then passed onto the subsequent layer. This process repeats until the final layer is reached, where the output of the network is defined by Eq. (2.4). The activation function allows the network to learn complex relationships with the training data (see Section 5.1).

The DNN has artificial neurons called nodes. The network's nodes are grouped into the

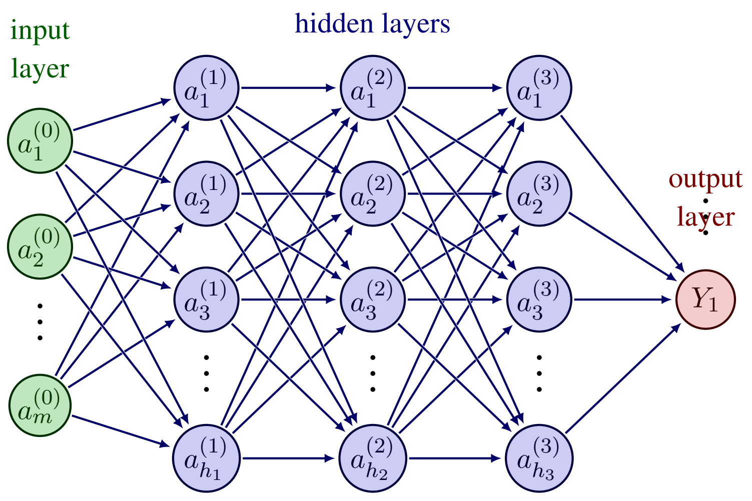


FIG. 4: Deep Neural Network structure for Deep Feedforward Network

layers as shown in Figure 4. The first layer of a network with n auxiliary variables x_i and m-n phase-space parameters s_i can be expressed as:

$$a_i^{(0)} = x_i \quad for \quad i \leq n \quad , \quad a_i^{(0)} = s_i \quad for \quad n < i \leq m \quad , \tag{2.1}$$

where the $a_i^{(l)}$ is the output for node i in layer l. The nodes of the network are given weights $w_{ij}^{(l)}$ for all their connections to subsequent layer nodes, which are represented as arrows in Figure 4. A bias $b_i^{(l)}$ is intrinsic to each of the layers:

$$z_i^{(l)} = \sum_j w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)} \quad , \tag{2.2}$$

$$a_i^{(l)} = \phi(z_i^{(l)}) \quad , \tag{2.3}$$

where the ϕ is an activation function. The final output of an L layer deep neural network is

$$Y_1 = \sum_j w_j^{(L+1)} a_j^{(L)} + b^{(L)} \quad . \tag{2.4}$$

In this method the Feynman parameters x_i and phase-space parameters s_i have an equal contribution to the output of the network, as they are both inputs to the network.
Once the network is successfully trained, it can integrate the Feynman parameterised integral over the trained phase-space region. Unlike the PySecDec Monte Carlo integrator, where specific Phase-spaces are integrated, Figure 2.
This DNN solves a large number of integrals synchronously over the trained phase-space, being as one forward pass is required to obtain the relevant terms. Hence, it is faster than the PySecDec MC integrator, which integrates over specific phase-spaces (see Table II).
This method is applicable to all integrals of this form, Eq. (1.1). The network parameters $w_{ij}^{(l)}$ and $b_i^{(l)}$ are initialised at the beginning of training (see Section 4.3), and were updated to better fit the training data. Throughout this paper the network parameters are grouped as:

$$(w_{ij}^{(l)}, b_i^{(l)}) \in \theta \quad , \quad \forall i, j, l \quad . \tag{2.5}$$

2.2. Derivative of the Neural Network

This subsection details the derivation of the derivative neural network. This structure can be seen in Figure 5. By Eq. (2.1), the derivative of an input node with respect to an auxiliary variable x_m is:

$$\frac{da_i^{(0)}}{dx_m} = \delta_{mi} \quad . \tag{2.6}$$

The subsequent layer first derivatives of the activation value with respect to x_m are shown to be:

$$\frac{da_i^{(l)}}{dx_m} = \phi'(z_i^{(l)})\frac{dz_i^{(l)}}{dx_m} = \phi'(z_i^{(l)})\left(\sum_j w_{ij}^{(l)}\frac{da_j^{(l-1)}}{dx_m}\right) \quad . \tag{2.7}$$

Where ϕ' is the derivative of the activation function. The necessary derivatives for the activation functions investigated are in Appendix A. To obtain the derivative network, each node was differentiated with respect to (w.r.t) all the auxiliary variables x_i present in the integral that was being solved.
For the I_1 integral there are three auxiliary variables. To calculate the derivative of a node in layer l with respect to two auxiliary variables x_m and x_r, chain rule is used and the following expression is achieved:

$$\begin{gathered}\frac{d^2a_i^{(l)}}{dx_rdx_m} = \phi''(z_i^{(l)})\frac{dz_i^{(l)}}{dx_r}\frac{dz_i^{(l)}}{dx_m} + \phi'(z_i^{(l)})\frac{d^2z_i^{(l)}}{dx_rdx_m} = \\ \phi''(z_i^{(l)})\left(\sum_j w_{ij}^{(l)}\frac{da_j^{(l-1)}}{dx_r}\right)\left(\sum_{j'} w_{ij'}^{(l)}\frac{da_{j'}^{(l-1)}}{dx_m}\right) + \phi'(z_i^{(l)})\left(\sum_j w_{ij}^{(l)}\frac{d^2a_j^{(l-1)}}{dx_rdx_m}\right) \quad .\end{gathered} \tag{2.8}$$

With the expression for the input layer being

$$\frac{d^2a_i^{(0)}}{dx_rdx_m} = 0 \quad . \tag{2.9}$$

The derivative of the network nodes for layer l with respect to 3 auxiliary variables x_1, x_2 and x_3 is expressed as a linear combination of activation function derivatives to the order of the integral:

$$\begin{gathered}\frac{d^3a_i^{(l)}}{dx_1dx_2dx_3} = \phi^{(3)}(z_i^{(l)})\frac{dz_i^{(l)}}{dx_1}\frac{dz_i^{(l)}}{dx_2}\frac{dz_i^{(l)}}{dx_3} \\ +\phi''(z_i^{(l)})\left(\frac{d^2z_i^{(l)}}{dx_1dx_2}\frac{dz_i^{(l)}}{dx_3} + \frac{d^2z_i^{(l)}}{dx_2dx_3}\frac{dz_i^{(l)}}{dx_1} + \frac{d^2z_i^{(l)}}{dx_1dx_3}\frac{dz_i^{(l)}}{dx_2}\right) \\ +\phi'(z_i^{(l)})\frac{d^3z_i^{(l)}}{dx_1dx_2dx_3} \quad .\end{gathered} \tag{2.10}$$

where

$$\frac{d^nz_i^{(l)}}{dx_1...dx_n} = \sum_j w_{ij}^{(l)}\frac{d^na_j^{(l-1)}}{dx_1...dx_n} \quad , \tag{2.11}$$

$$\alpha_i^{(l)} = \frac{d^n a_i^{(l)}}{dx_1...dx_n} \quad , \quad \gamma_n = \sum_j w_{ij}^{(L+1)} \frac{d^n a_j^{(L)}}{dx_1...dx_n} \tag{2.12}$$

$\alpha_i^{(l)}$ represents the derivative of the activation value $a_i^{(l)}$ (activation value of node i in layer l) w.r.t the Feynman parameters of the integral. γ_n is the output of the derivative neural network for integrals with n auxiliary variables.
The output of the 1 loop integral with 3 auxiliary variables is γ_3, and the output of the 2 loop integral in [2] with 6 auxiliary variables is γ_6. The output of the network is generalised as:

$$\gamma_n = \frac{d\mathcal{NN}(s_1, ..., s_m; x_1, ..., x_k)}{dx_1...dx_n} \quad . \tag{2.13}$$

To obtain accurate integral predictions the derivative network was trained to the integrand. Once trained successfully, the integral values are calculated using the forward pass of the network Figure 1, Eq.(2.14).

$$\begin{gathered} \int_0^1 dx_1 \int_0^1 dx_2 \int_0^1 dx_3 \frac{d\mathcal{NN}(s_{12}, s_{14}, m_H^2; x_1, x_2, x_3)}{dx_1 dx_2 dx_3} = \\ \mathcal{NN}(s_{12}, s_{14}, m_H^2; 1,1,1) - \mathcal{NN}(s_{12}, s_{14}, m_H^2; 1,1,0) \\ -\mathcal{NN}(s_{12}, s_{14}, m_H^2; 1,0,1) - \mathcal{NN}(s_{12}, s_{14}, m_H^2; 0,1,1) \\ +\mathcal{NN}(s_{12}, s_{14}, m_H^2; 0,0,1) + \mathcal{NN}(s_{12}, s_{14}, m_H^2; 0,1,0) \\ +\mathcal{NN}(s_{12}, s_{14}, m_H^2; 1,0,0) - \mathcal{NN}(s_{12}, s_{14}, m_H^2; 0,0,0) = I_1 \end{gathered} \tag{2.14}$$

This process can be generalised and continued for higher dimensional integrals, processes with more degrees of freedom, in particular the two loop integral with 6 degrees of freedom [2]. For each extra degree of freedom added to the integral, the number of terms in the derivative activation value grows immensely. Higher loop level processes will lower network performance, and the strain on computational resources will increase.

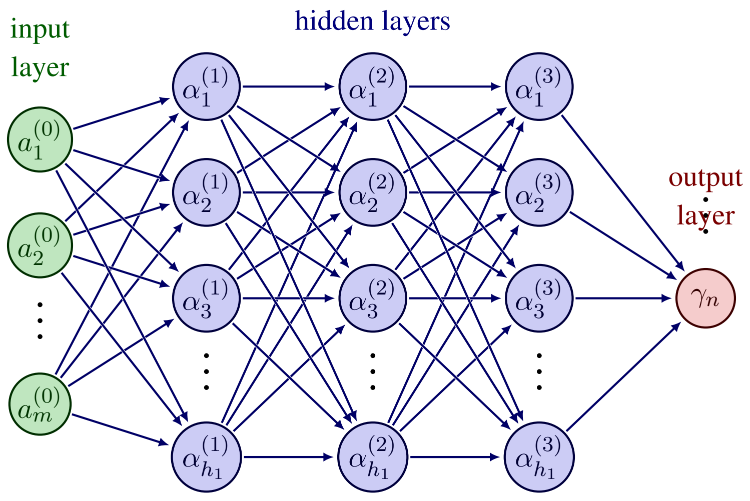


FIG. 5: Structure for the derivative of the Deep Neural Network with respect to the Feynman parameters x_i

2.3. Pre-processing

The parametric integral's phase-space and Feynman parameters are treated as equal contributions when modelling the network, this separates it from other integration approaches [19]. An important contribution for the convergence of the network is the form of the input data [20]. Unsuitable input data would inhibit the learning potential of the network. Therefore, it was vital to pre-process the input data to maximise the efficiency of the network.
Pre-processing is an important aspect of machine learning, and is used when the input data is in a form that makes it difficult for the NN architecture to learn. Eq. (2.2) shows that the inputs of the network $a_i^{(0)}$ are given an equal contribution to the weighted sum in the first layer. However, the s_i inputs have a larger magnitude than the x_i, which causes the network to be unstable.
Thus, the s_i inputs of the network where scaled linearly, so the inputs would have a similar magnitude - assisting network convergence. The new phase-space parameters $\tilde{s}_i$ were inputted into the derivative network to obtain predictions for the integrand $(\gamma_n)_i$. Their boundaries are defined as:

$$\tilde{s}_i = \frac{s_i}{30} \quad , \quad -1 \leq \tilde{s}_i \leq -0.1 \quad . \tag{2.15}$$

Further pre-processing included normalising the parametric integral from Eq. (1.1) at the centre of a hypercube $I \to \tilde{I}$ [2],

$$\tilde{I}(s_1, ..., s_m) = \frac{I(s_1, ..., s_m)}{f(s_1, ..., s_m; \frac{1}{2}, ..., \frac{1}{2})} = \int_0^1 dx_1 ... \int_0^1 dx_k \frac{f(s_1, ..., s_m; x_1, ..., x_k)}{f(s_1, ..., s_m; \frac{1}{2}, ..., \frac{1}{2})} \quad . \tag{2.16}$$

A Korobov transform [21, 22] was applied to improve the convergence of the integral, as it is shown to reduce the variance of multidimensional integrals [2, 3, 23]. Applying a Korobov transform to the integral from Eq. (1.1) gives

$$I(s_1, ..., s_m) = \int_0^1 dt_1 ... \int_0^1 dt_k w_1(t_1)...w_k(t_k) f(s_1, ..., s_m; x_1(t_1), ..., x_k(t_k)) \quad . \tag{2.17}$$

The weight of the transform w(t) is defined as

$$\int_0^1 w(t)dt = 1 \tag{2.18}$$

and gives

$$\int_0^t w(t')dt' = x(t) \quad . \tag{2.19}$$

The weight transformation chosen, was the same transformation present in [2].

$$x(t) = t^2(3 - 2t) \quad , \quad w(t) = 6t(1 - t) \quad . \tag{2.20}$$

This transformed the integral to

$$\tilde{I}(s_1, ..., s_m) = \int_0^1 dt_1 ... \int_0^1 dt_k \frac{w_1(t_1)...w_k(t_k) f(s_1, ..., s_m; x_1(t_1), ..., x_k(t_k))}{f(s_1, ..., s_m; \frac{1}{2}, ..., \frac{1}{2})} \quad . \tag{2.21}$$

This integral was then normalised between 0 and 1, as it increased the speed of network convergence because the elements of the s-x rank-1 lattice had a similar magnitude.

$$\hat{I} = \frac{\tilde{I}}{max(Integrand)} \quad . \tag{2.22}$$

3. BACK PROPAGATION AND GRADIENT DESCENT

When training the derivative of the network, a rank-1 lattice of randomly sampled s-x parameters was inputted in the forward direction of the derivative network to obtain a prediction for the integrand $(\gamma_n)_i$, by Eq. (2.13). The accuracy of the prediction $(\gamma_n)_i$ was dependent on the configuration of the network parameters θ, Eq. (2.5).
The network parameters θ were updated, Eq. (3.1), with the ADAM [24] gradient descent optimiser. Adam merges the best assets from adaptive learning rate and momentum based optimisers, making it highly effective for training deep neural networks. Its hyperparameters did not need to be tuned to achieve optimally trained networks (see Appendix B for additional information).

$$\theta + \Delta\theta \to \theta \tag{3.1}$$

The observed values $\hat{Y}_i$ were obtained when the rank-1 s-x parameter lattice was plugged into integrand ($\frac{1}{\mathcal{F}_1^2}$ integrand for the 1 loop integral). To test the accuracy of the predictions $(\gamma_n)_i$ with the observed values $\hat{Y}$, a mean squared error loss function was used:

$$loss_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - (\gamma_n)_i)^2 \quad = MSE(\hat{Y}_i, (\gamma_n)_i) \tag{3.2}$$

where N is the lattice size. A lower loss indicated that the network fitted to the training data better (see Section 4.1).

For the 1 loop integral, the derivative of the loss function w.r.t to an arbitrary network parameter θ, is expressed as a product of backpropagated gradients:

$$\frac{\partial L(X,\theta)}{\partial\theta} = \frac{\partial L(X,\theta)}{\partial\gamma_3}\frac{\partial\gamma_3}{\partial\alpha_i^{(l)}}\frac{\partial\alpha_i^{(l)}}{\partial z_i^{(l)}}\frac{\partial z_i^{(l)}}{\partial\theta} \quad . \tag{3.3}$$

Backpropagation (backprop)[25] allows the information from the loss to flow in the backwards direction of the network - from the network output, through the hidden layers, and to the input - to calculate the gradients of the loss with respect the network parameters. To lower the loss, the optimiser minimised these gradients (obtained from backprop) by updating the network parameters to get more accurate integrand predictions $(\gamma_n)_i$.
This form of the loss gradient, displays the significant impact that small changes in the network components can have on its overall performance. The second term is the partial derivative for the derivative activation value $\alpha_i^{(l)}$ Eq. (2.12) w.r.t the weighted sum at a node $z_i^{(l)}$ Eq. (2.2). This partial derivative can be expressed as a sum of the activation value derivatives, similar to Eq. (2.10), however it contains a derivative one order greater:

$$
\begin{aligned}
\frac{\partial \alpha_i^{(l)}}{\partial z_i^{(l)}} &= \phi^{(4)}(z_i^{(l)})\frac{dz_i^{(l)}}{dx_1}\frac{dz_i^{(l)}}{dx_2}\frac{dz_i^{(l)}}{dx_3} + \phi^{(3)}(z_i^{(l)})\frac{\partial}{\partial z_i^{(l)}}\Big(\frac{dz_i^{(l)}}{dx_1}\frac{dz_i^{(l)}}{dx_2}\frac{dz_i^{(l)}}{dx_3}\Big) \\
&+\phi^{(3)}(z_i^{(l)})\left(\frac{d^2z_i^{(l)}}{dx_1dx_2}\frac{dz_i^{(l)}}{dx_3} + \frac{d^2z_i^{(l)}}{dx_2dx_3}\frac{dz_i^{(l)}}{dx_1} + \frac{d^2z_i^{(l)}}{dx_1dx_3}\frac{dz_i^{(l)}}{dx_2}\right) \\
&+\phi''(z_i^{(l)})\frac{\partial}{\partial z_i^{(l)}}\left(\frac{d^2z_i^{(l)}}{dx_1dx_2}\frac{dz_i^{(l)}}{dx_3} + \frac{d^2z_i^{(l)}}{dx_2dx_3}\frac{dz_i^{(l)}}{dx_1} + \frac{d^2z_i^{(l)}}{dx_1dx_3}\frac{dz_i^{(l)}}{dx_2}\right) \\
&+\phi''(z_i^{(l)})\frac{d^3z_i^{(l)}}{dx_1dx_2dx_3} + \phi'(z_i^{(l)})\frac{\partial}{\partial z_i^{(l)}}\Big(\frac{d^3z_i^{(l)}}{dx_1dx_2dx_3}\Big) \quad .
\end{aligned}
\tag{3.4}
$$

This gradient demonstrates the consequential affect the shapes of activation functions and their derivatives have on network convergence. This will be discussed in Section 5.1.

The network parameters were trained for a fixed number of epochs. During each epoch a backwards computational graph was generated, with all the back-propagated gradients. At the end of each epoch, the gradients were zeroed. During training the randomly sampled rank-1 lattice of s-x parameters, was changed every 5 learning epochs, to prevent the network over training specific features of that lattice. This process was repeated in a training loop, until the loss plateaued, and the network converged to a final state.

Once the derivative network was optimally trained to the integrand of the integral, the forward pass of the network was used to calculate the integral, Eq. (2.14)

The pyTorch [26] python library was employed to code and train the neural network architecture, as it can perform complex tensor calculations, and can easily generate back-propagated computational graphs.

4. THEORY OF TRAINING DEEP NETWORKS

4.1. Loss Optimisation Landscape

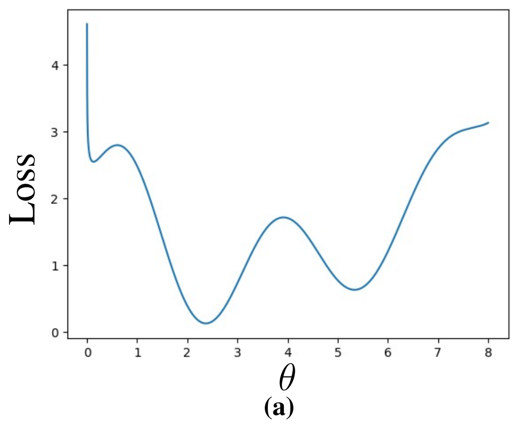
The MC integrator's loss scales by a factor of $\frac{1}{\sqrt{N}}$, where N is the number of trained samples. This allows the MC integrator to push its accuracy much further than the derivative neural network, as we sacrifice accuracy for speed. Hence the objective of this research is to push the loss of the derivative neural network to lower values.

The gradient descent training algorithm lowered the loss of the network, by producing the model which best represented the data. It is a regression task. In most applications of machine learning over-training can damage the accuracy of the model.

Usually, regularisation techniques are implemented to mediate the damaging affects over familiarity with training data has, such as limiting its ability to generalise unseen data. Training data is noisy in most instances of deep learning.

However the uniqueness of this application, is that the model does not require regularisation techniques to produce excellent results because the derivative neural network is fitted directly to the integrand [2] (zero noise).

Figure 6a shows a simplified schematic diagram for the loss surface. The loss surface is the



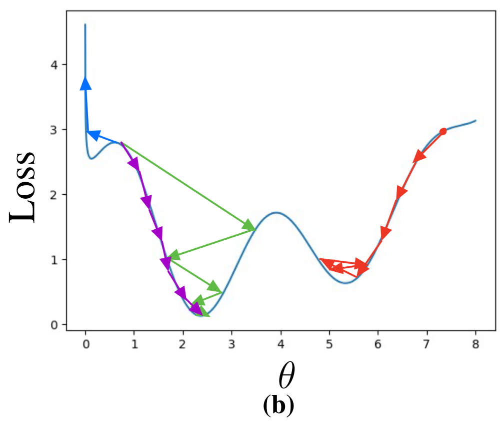


FIG. 6: (a) Schematic diagram of loss vs an arbitrary network parameter θ. (b) Schematic diagram of gradient descent.

topology of the loss function for different network parameter configurations θ of the architecture during training [27]. The points of zero gradient represent theoretical critical points that the network can successfully train to or get stuck at. The diagram clearly shows the presence of sub-optimal minima; areas with sharp gradients and the global minimum.

The global minimum is the lowest loss value that the network architecture can obtain. In deep networks, there are a high number of local minima with analogously low loss values, so the network does not need to be trained to the global minimum to produce sufficient accuracy[27–29].

The training of an arbitrary network is shown in Figure 6b. The starting points of the arrows denotes their initialisation and the arrow's size represents their learning rate, which is used to update the network parameters θ. The consequences of poor initialisation are prevalent in this diagram, as the network may train to worse minima (red) or diverge completely (blue) (see Section 4.3). The learning rate is equally important (see Section 4.4). The red arrow shows an oscillating θ around a local minima, this can happen if the learning rate doesn't change during training. The purple arrow shows a steady approach to an optimal local minimum. The green arrow displays how utilising an adaptive learning rate improves the speed of network convergence to a good local minimum, however there is the risk that the learning rate may be too large at certain points of the optimisation, and it may overshoot into worse local minima. The blue arrow shows an example of the network diverging, this can happen if the learning rate is too large and the θ hits a cliff, significantly worsening the network's performance. There may be flat regions in the loss topology, where the loss doesn't decrease until a significant amount of training occurs.

The loss surface evolves during training, hence there are many other occurrences that arise. To maximise the accuracy of the derivative neural network, strategies were implemented to reach optimal local minima: such as providing good initialisation, and optimally tuning learning rates.

4.2. Replicas

Separate networks, with the same architecture, were trained n times. Although the accuracy was similar, they were not repeats they were replicas R_i. Each replica experienced a different initialisation and converged to different final solutions, local minima (see Section 4.1).
The replicas were trained with the same methodology, so their integral estimates had similar means and variances σ^2. The replica's estimates were averaged $\tilde{R}$. Training more replicas reduces the error in the mean, provided the means are close to the true values . This method of averaging the replicas greatly increases the accuracy of the integrals result. This lowered the estimate's standard deviations σ by a factor of square root of n [4]
Therefore, it was ideal to have as many similar efficient tests as possible to get a more precise mean estimate:

$$\tilde{R} = \frac{R_1 + R_2 + ... + R_n}{n} \quad , \quad v = \frac{\sigma}{\sqrt{n}} \quad , \tag{4.1}$$

v is the standard deviation of the mean, which was taken as the uncertainty of the averaged replicas.
Figure 7 shows the significance of increasing the number of replicas n. In this experiment $3-5$ replicas were trained. The error for n averaged replicas decreased by a factor of 10 to the power of the y axis (decreased by a factor of the square root of n). For 100 replicas the error would decrease by a factor of 10, because the y coordinate is -1, and for 10000 replicas the error will decrease by a factor of 100 because the y coordinate is -2.

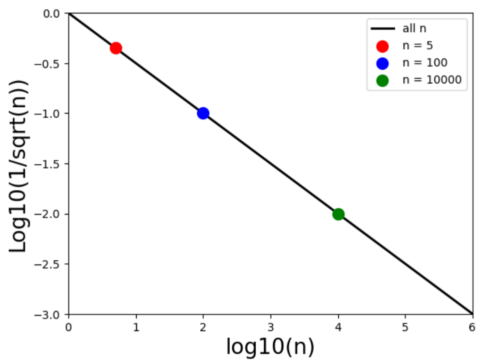


FIG. 7: The log10(error) of the replicas changing with the number of replicas

During training the model's accuracy was collected every 100 epochs, and the best model achieved during training was saved to be averaged with the replicas. Each replica took 7-10 hours to train.

4.3. Xavier Initialisation

The schematic diagram for gradients descent, Figure 6b demonstrates the importance of initialisation. The network bias parameters were initially set to zero, and the network weights were initialised with uniform Xavier initialisation [30]:

$$W \sim U\Big[-\sqrt{\frac{6 \cdot gain}{n_{in} + n_{out}}} \quad , \quad \sqrt{\frac{6 \cdot gain}{n_{in} + n_{out}}} \Big] \quad , \tag{4.2}$$

where n_{in} is the number of inputs and n_{out} is the number of outputs. The $gain$ factor was adjusted for different activation functions and network sizes, to ensure that the variance of the weights was large enough for successful learning, limiting the damaging affects of vanishing gradients or dead derivative activation node formation (see Section 5.3). Table I dipslays the $gain$ values for each architecture.

4.4. Learning Rate

The learning rate ϵ is the most important hyperparameter in deep learning (see Section 4.1). Learning rates that were too large caused network divergence, and learning rates that were too small - at the start of training - caused sub-optimal network convergence. The initial learning rate for each derivative network architecture was tuned iteratively, with the best ϵ being chosen for the subsequent replica network training.
The built in pyTorch ReduceLROnPlateau(optimizer, mode = 'min', factor, patience) scheduler was implemented to decrease the learning rate of the optimizer over training. The reasons are shown schematically with the purple arrows in Figure 6b, as it allows the derivative network to train to and reach lower loss in the optimisation surface.
The ReduceLROnPlateau scheduler, decreased the ϵ by a tuned factor, when the loss plateaued for a selected number of epochs (patience). It was found that a slower decrease in the learning rate, as seen in Figure 8, lead to increased accuracy. This was due to the network being able to train more s-x configurations.

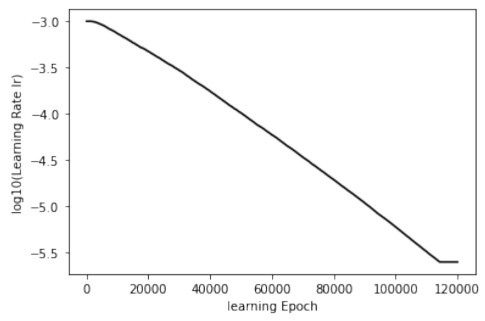


FIG. 8: The log10(learning rate) over training

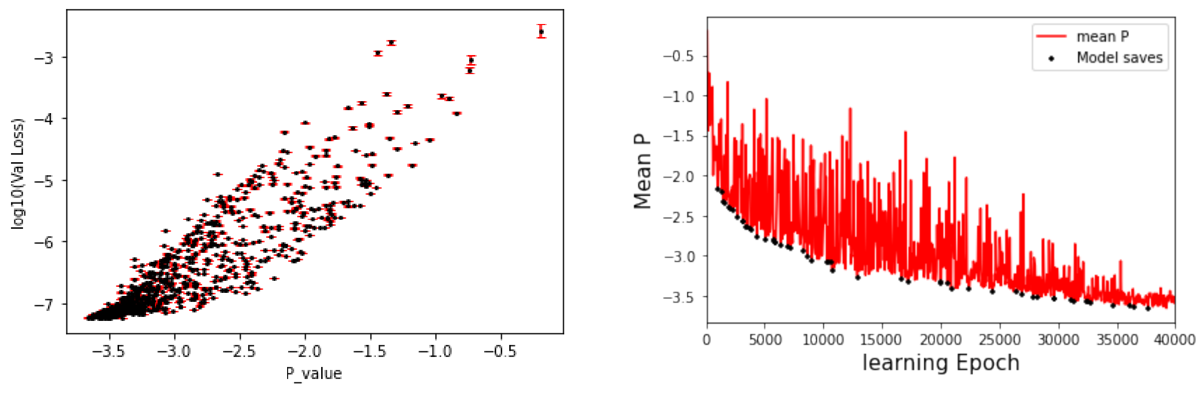
5. RESULTS

The P-value equation [2], Eq. (5.1), was used to compare the network's estimates e for the scattering amplitudes with the true values t from the Monte Carlo integrator, Pysecdec.

$$P = \log_{10}\left(\left|\frac{e-t}{t}\right|\right) \quad . \tag{5.1}$$

The negativity of the P-value is the number of correct digits the network had obtained for that specific phase-space configuration. For example, a P-value of $-n$ would indicate that the network was accurate for n digits in that phase-space.

P-value histograms were used to show the accuracy of the phase-space region trained, and to compare accuracy between the different architectures trained. The mean P-value, number of correct digits, was taken to be the networks measure of accuracy.
Figure 9a demonstrates the importance of effective training, as there is strong positive correlation between lower validation loss (unseen set) and more negative P-values. They are not directly proportional, as the validation set contains a handful of s-x combinations and does not represent the entire phase-space region.
Throughout training, the P-value was collected every 100 epochs. The P-value decreased throughout training with some noise, and eventually plateaued (reached a local minima). The network was saved when lower and lower P-values were achieved, Figure 9b. The state of the network with the best P-value was used as a replica (see Section 4.2).



(a) (b)

FIG. 9: (a) $\log_{10}$(Validation loss) vs the mean P-value, for a 3 layer 130 hidden unit GELU derivative network over training. (b) Mean P-value, for a 3 layer 130 hidden unit GELU derivative network over training (red), and model save points (black)

The logarithmic ratio (R-value) Eq.(5.2) shows the quality of the estimates. R-value histograms centered around 0, indicate that there is no damaging bias in the network. Thinner R-value histograms have more accurate integral predictions:

$$R = \log_{10}\left(\left|\frac{e}{t}\right|\right) \quad . \tag{5.2}$$

Once the replicas were averaged, the error of the mean v was taken as the networks' uncertainty, Eq. (4.1). Histograms of $\log_{10}\left(\frac{v}{|e-t|}\right)$, referred to as log10(est/true) on Figure 11a and Figure 21b, were used to compare the uncertainty v and true error present in the replica averaged architectures [2].
The straight lines (the zero of these histograms) represent the points where the true error was equal to the uncertainty v. Points lying to the right of this line, have a true error less than the uncertainty v and points to the left have a true error greater than the uncertainty.
Table I summarises the findings of this report. Figure 10 and Figure 11 show the P-value, R-value and log(est/true) histograms for the architectures trained with the GPU. Table II shows the superior speed of the neural network compared to the PySecDec integrator.
The GELU activation function was the most accurate activation function for the 1 loop integral.

	activation	layers	nodes	epochs	lattice size	gain	ϵ	machine	Replicas	P-value
I_1	softsign	3	130	2000	80000	1.5	0.01	GPU	3	-0.8 ± 0.2
I_1	sigmoid	3	130	15000	80000	2	0.001	GPU	3	-2.9 ± 0.3
I_1	tanh	3	130	10000	75000	1.5	0.001	GPU	5	-3.4 ± 0.2
I_1	GELU	3	130	10000	30000	1.5	0.01	GPU	3	-3.9 ± 0.2
I_1	GELU	10	60	10000	12500	1.5	0.01	GPU	3	-3.8 ± 0.2
I_1	GELU	3	100	10000	500	1.5	0.01	CPU	5	-3.1 ± 0.3
I_1	GELU	10	30	10000	500	1.5	0.01	CPU	5	-3.0 ± 0.3
I_1	GELU	10	60	10000	500	1.5	0.01	CPU	5	-3.4 ± 0.3

TABLE I: P-values for the following network configurations and parameters

The tanh activation function was more accurate than the sigmoid, displaying the repeatability of the previous investigation [2]. The softsign performed poorly. The reasons behind this are explained in the following subsections (See Appendix A for the activation function derivatives).

	$\simeq$ 4 digits
PySecDec	0.8ms
NN	0.05ms

TABLE II: Approximate run times to integrate 27000 integrals, to 4 digits of accuracy, over the trained phase-space region Eq. (1.4)

5.1. Activation Functions $\phi(x)$

Activation functions ϕ transform the weighted sums of neural networks, Eq. (2.2). If activation functions were omitted, the forward pass of the network would be linear. The architecture would have limited potential modelling complex behaviours, because the back propagated gradients would be constant. Activation functions and their derivatives introduce non-linearity to the weighted sums of the derivative neural network, Eq. (2.11).
The non-linear derivative activation values $\alpha_i^{(l)}$, were passed through the derivative network during forward propagation, and contributed to the subsequent layer nodes weighted sums, which were transformed non-linearly again. This process repeated until the output layer. Deep feed-forward networks have numerous hidden layers, the activation functions are used to calculate the outputs of the nodes and allow neural networks to learn sophisticated patterns and relationships with the training data. Figure 12 shows the shapes of the activation functions applied to the derivative neural network throughout this investigation.
The choice of activation function is an important factor for the success of deep learning architectures, as seen in Table I. Different activation functions greatly affect the architecture's

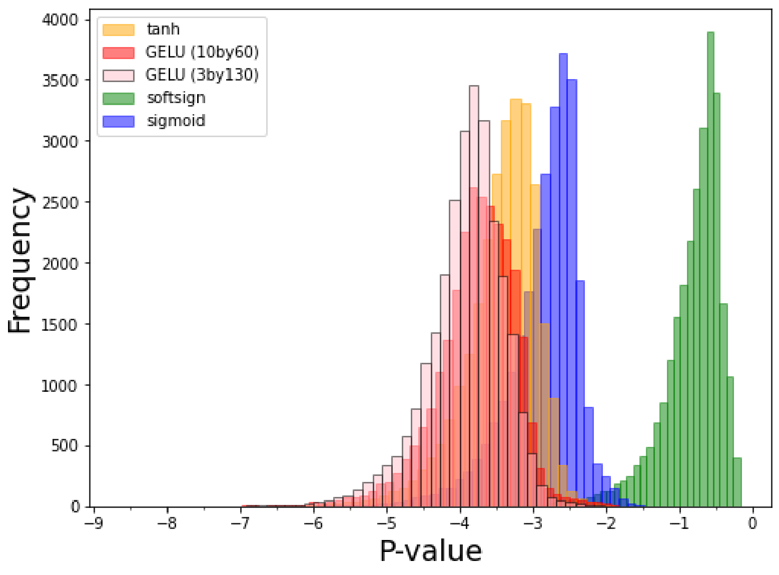


FIG. 10: (a) P-value histograms, for the five replica averaged architectures trained with the GPU(shown in Table I). Digits of accuracy for the different architecture.

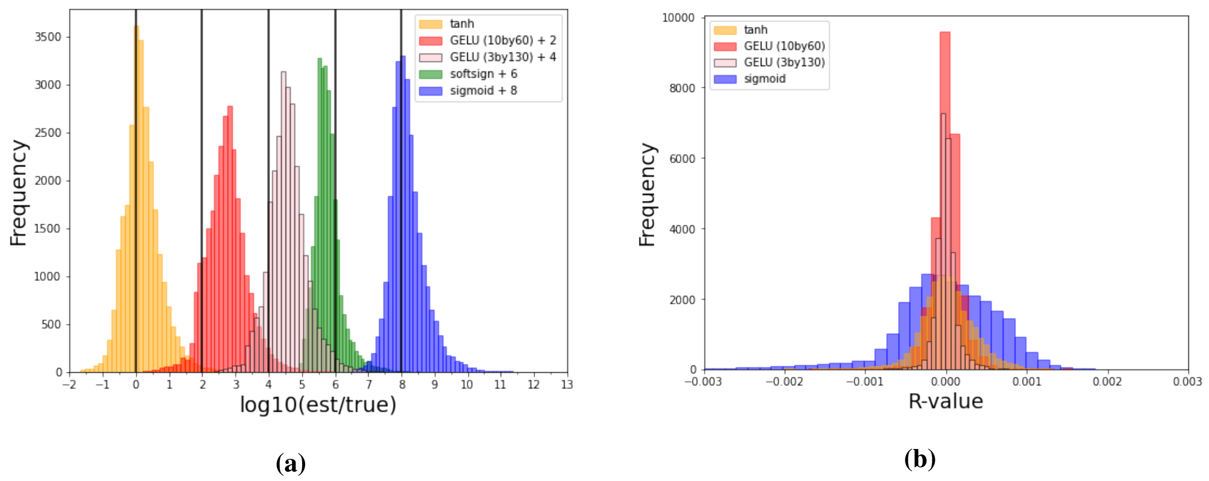


FIG. 11: (a) log10(est/true) histograms and (b) R-value histograms for the replica averaged architectures trained with the GPU, shown in Figure 10.

learning capacity, efficiency and stability due to their shapes[30].

In deep feed-forward networks, the most important derivative is the first derivative, as it is the only derivative that affects the size of the backpropagated gradients, in most cases. Whereas for the derivative neural network, the gradients of the derivative activation values contain a mixture of activation function derivatives (to order one more than the integral), Eq. (3.4). Therefore, when selecting activation functions for the derivative neural network, their higher order derivatives must possess favourable traits. The first four derivatives of the activation functions from Figure 12 are shown in Figure 13.

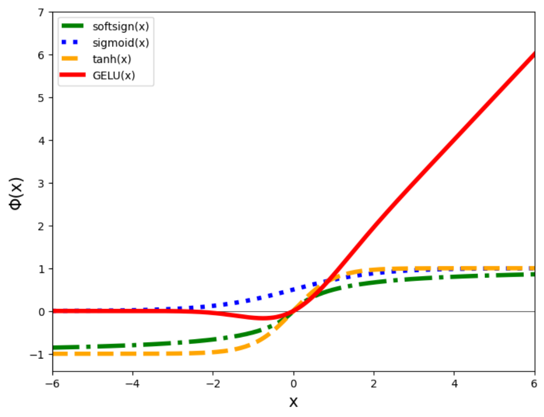


FIG. 12: Activation function ϕ(x) values for input x.

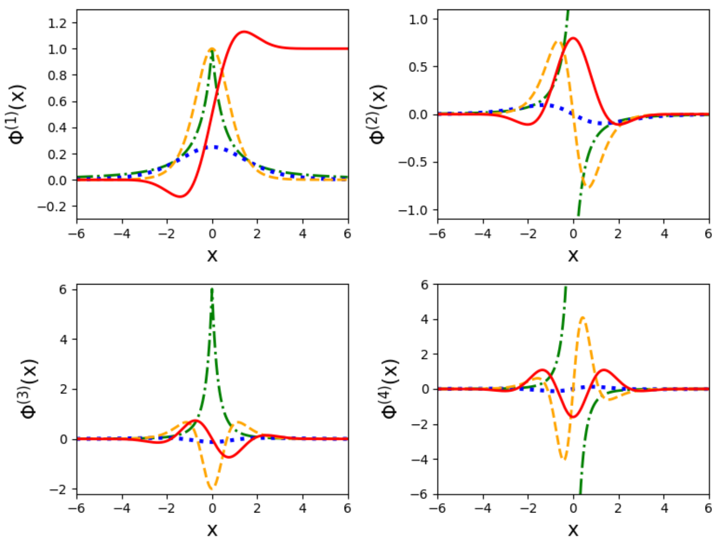


FIG. 13: First four derivatives of activation function ϕ(x) values for input x. Softsign (green), tanh (yellow), sigmoid (blue) and GELU (red).

5.2. Softsign

The softsign [31] is a classical activation function:

$$softsign(x) = \frac{x}{1 + |x|}. \tag{5.3}$$

The softsign demonstrated poor accuracy, 0.8 ± 0.2 correct digits, because its 2nd and 4th derivatives have jump discontinuities at $x = 0$, Figure 13. These jump discontinuities create an instability in the network derivative activation values, as 2nd and 4th derivative activation values that are around 0 would cause the back-propagated gradients Eq. (3.4) to be unstable. This can be seen in Figure 14, where the 2nd derivative of the activation value w.r.t to x_1 and x_2 and the derivative activation values are volatile, leading to poor P-values Figure 15. In addition, the softsign's errors were dominated by the true error, shown in Figure 11a (points mostly on the left of its zero line $x = 6$), indicating that a systematic variation is present due to the nature of its derivatives.

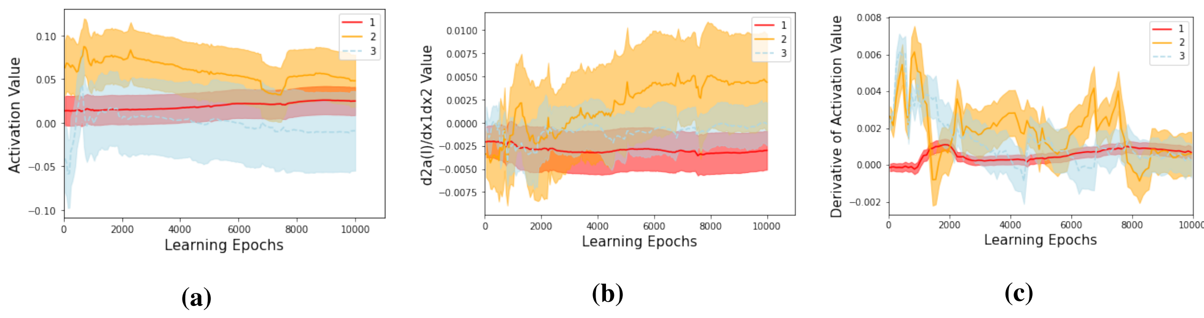


FIG. 14: Mean (central line) and standard deviations (std = boundary of shaded region), for each layer of the 3 layer 130 hidden softsign based derivative neural network, collected during the training (10000 epochs). The standard deviations (std) where divided by a factor. (a) Activation values (std/20). (b) Second derivative of activation value with respect to x_1 and x_2 auxiliary variables (std/20). (c) Derivative activation values (std/100)

Therefore, the activation function's derivatives must be continuous for successful network convergence.

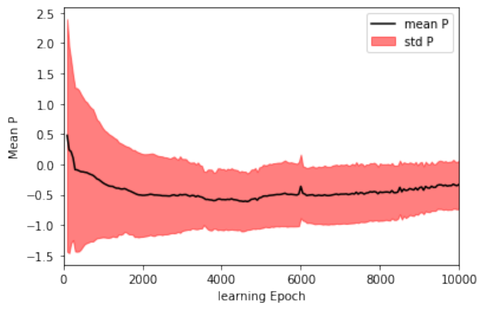


FIG. 15: Mean (black) and the standard deviation (red area boundary) of the P-value over training, for a 3 layer 130 hidden unit softsign based derivative neural network

5.3. The Logistic Sigmoid and the Hyperbolic Tangent

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5.4}$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5.5}$$

The tanh and sigmoid networks were more accurate than the softsign, because their derivatives are continuous, which lead to stable activation and derivative activation values, Figure 16.

Derivative networks with tanh units were approximately half a digit more accurate than the logistic sigmoid based networks, with 3.4 ± 0.2 correct digits compared to 2.9 ± 0.3. This superior accuracy was seen in [2], where Professor Maître's tanh based network, for the 1 loop integral, peaked at approximately 3.4 digits, Figure 17. The similar peaks of both 1 loop tanh based networks (see Figure 11) demonstrates the replicability of this integration technique.

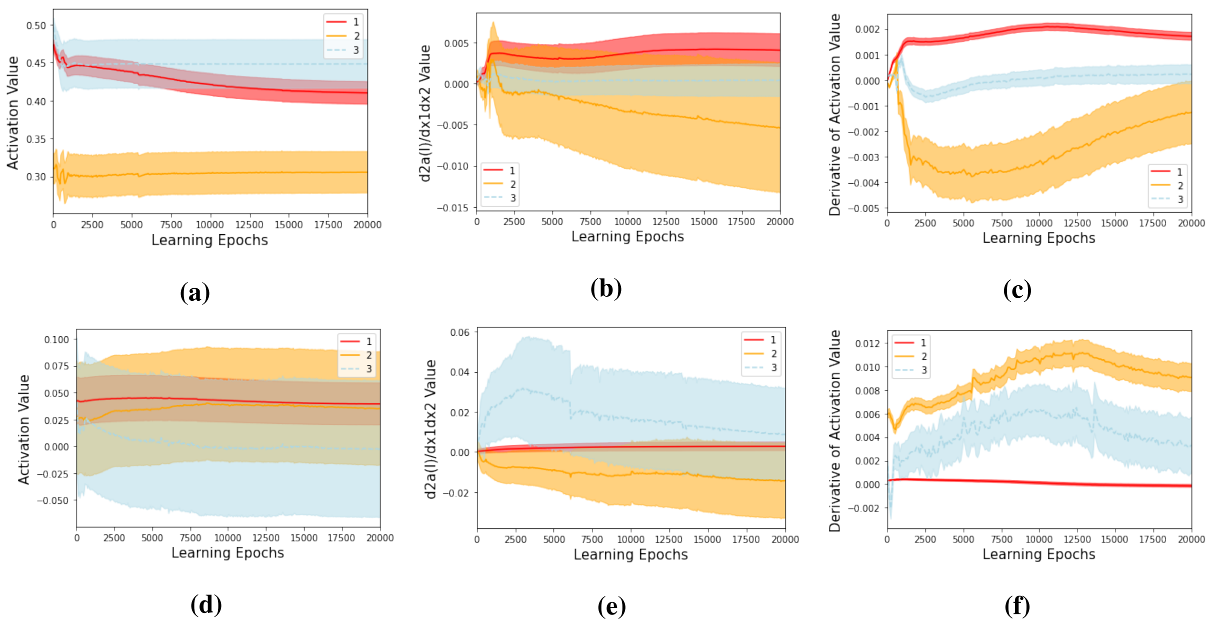


FIG. 16: Mean (central line) and standard deviations (std = boundary of shaded region), for each layer of the 3 layer 130 hidden derivative neural network, collected during the training. The standard deviations (std) where divided by a factor. Subplots (a), (b) and (c) were obtained from a sigmoid based network, and subplots (d), (e) and (f) were obtained from a tanh based network. (a)(d) Activation values (std/20). (b)(e) Second derivative of activation value with respect to x_1 and x_2 auxiliary variables (std/20). (c)(f) Derivative activation values (std/100). Sampled over 20000 epochs.

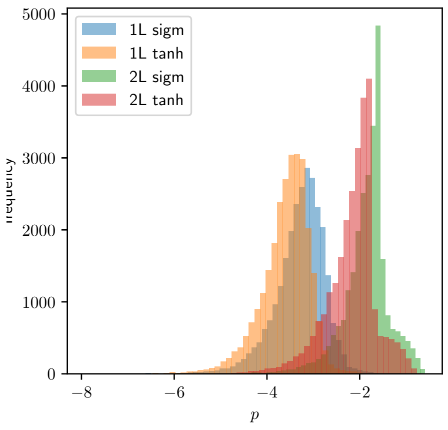


FIG. 17: Left panel in Figure 3 from [2], showing four P-value histograms for the tanh and sigmoid activation functions for the 1 loop (1L) and the 2 loop (2L) integrals, obtained from that previous investigation

Both of their errors were controlled by the network uncertainty v, shown in Figure 11a, as most of their points are on the right hand side of their zero lines ($x = 0$ and $x = 8$). This suggests that their errors are predominately determined by the noise of their estimates, meaning that averaging more replicas would increase their accuracy, as discussed in section 4.2.
The sigmoid based networks were less accurate than the tanh based networks, because their

gradients are more vulnerable to vanishing [30]. The sigmoid function has a range between zero and one, see Figure 12. For large negative values, the activation values saturate to zero, and no longer contribute to the weighted sums in the next layer, Eq. (2.2). In deep-feedforward networks this activation unit is referred to as a dead node, as it is effectively dead, no longer contributing to the network's learning.

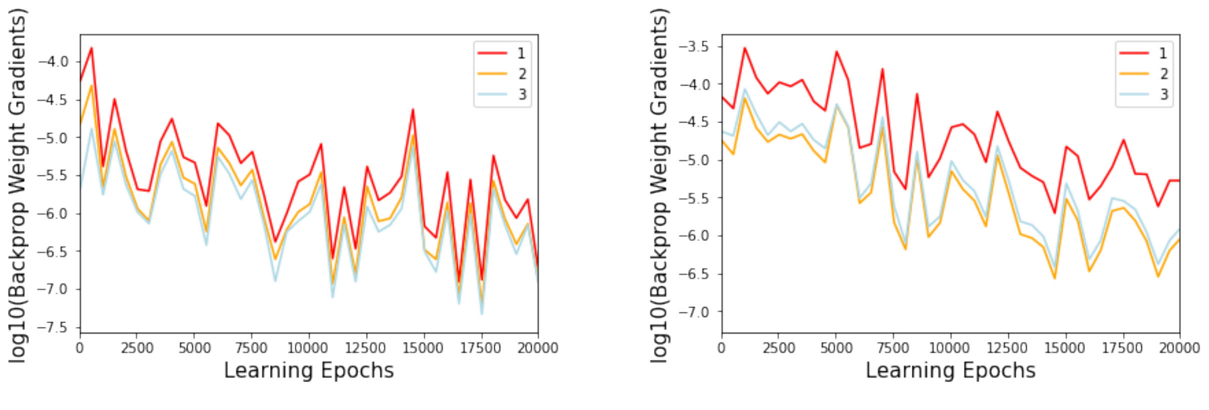
However, the derivative of the neural network was trained. The derivative activation values, Eq. (2.10), for the 1 loop integral, are written as a linear combination of the activation function's first three derivatives, and its backpropagated gradient w.r.t the weighted sum is written as a combination of its first four derivatives, Eq. (3.4). Therefore, for large input values the derivative activation values (and their gradients) will saturate to zero, as the sigmoid's derivatives are all flat. A derivative activation unit, with large inputs, can be referred to as a dead node.

A large number of dead nodes reduces the effective size of the network, lowers its representational capacity, and severely inhibits its performance. Thus, sigmoid based networks train to sub-optimal local minima.

The tanh function has a similar shape to the sigmoid, however it is bound between -1 and 1. Dead nodes are less prevalent in tanh based networks, on account of its derivatives shapes. They are larger in magnitude, and flatten less than the sigmoid for large values. Hence, tanh based networks train to better local minima, giving them higher accuracy.

When the networks were initialised, their parameters θ required significant adjustments to minimise the loss, so the variance of the backpropagated gradients was larger initially. The variance of the back-propagated gradients decreased during training, whilst the network was converging, requiring smaller modifications to its parameters θ [30]. A steady decrease in the variance indicated that the network was converging to an optimal solution.

The amount of averaged replicas was not the reason for the tanh's superior accuracy, although it assisted it. This was due to the decay of the backpropagated gradients variance being slower in the tanh than the sigmoid, shown in Figure 18. This implied that dead nodes were more common in the sigmoid based networks than the tanh based networks [30].



(a) **(b)**

FIG. 18: log10(standard deviations of the back-propagated gradients of weights), of each layer in the network, during training. (a) sigmoid and (b) tanh, sampled over 20000 epochs.

5.4. GELU

The GELU [9] activation function is superior to the classical activation functions (softsign, sigmoid and tanh) for many tasks [32], as shown through its application to the popular large language model GPT-3 [33]. It can be expressed as:

$$GELU(x) = x \cdot \Phi(x), \tag{5.6}$$

where the $\Phi(x)$ is a Gaussian cumulative distribution function:

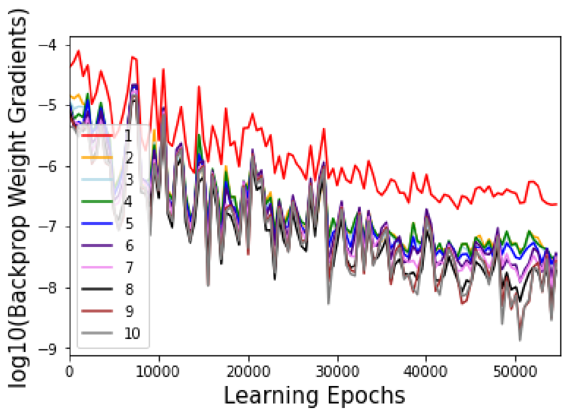
$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt = P(X \leq x) \quad , \quad X \sim N(0,1) \quad . \tag{5.7}$$

The GELU was approximated to:

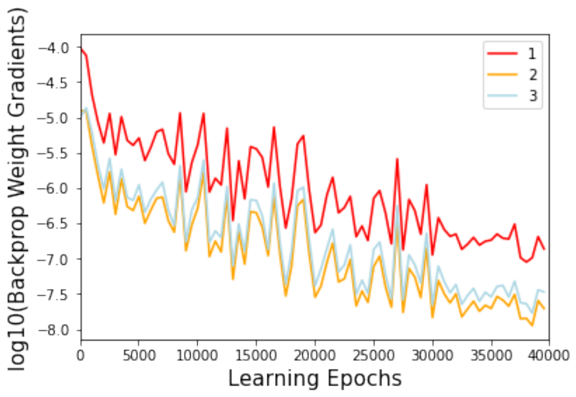
$$GELU(x) = 0.5x\Big(1 + tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\Big) \quad , \tag{5.8}$$

and its derivatives are present in Appendix A.
GELU based networks were the most accurate, with 3.9 ± 0.2 correct digits for the 3 layer 130 hidden unit network, and 3.8 ± 0.2 correct digits for the 10 by 60 hidden unit network. Their errors were dominated by the noise of their estimates, Figure 11a (most points on the right hand side), averaging more networks lowers their uncertainty.
Like the hyperbolic tangent, the higher order (2nd and above) derivative's tails approach zero slower than the sigmoid, and its derivatives also have large magnitudes. Its first derivative approaches one for large positive inputs and zero for large negative inputs, unlike the tanh, which approaches zero for large inputs that are positive or negative.
This makes it less susceptible to the dead node problem, as for large positive inputs the higher order derivatives will die, the first derivative terms remain, and still contribute to the learning. For large negative inputs the derivative activation unit will die, so there is a level of network effective size reduction, but not as much as the classical activation functions. Its derivative terms are more computational expensive than the derivatives of classical activation functions, so its batch sizes were smaller.
The slight difference in the performance of the two GELU architectures (3 layer 130 hidden units and 10 layer 60 hidden units) was due to the deeper network's higher proneness to forming dead nodes and smaller lattice size. It has less hidden units per layer, so dead nodes will have a greater affect on the subsequent layers learning, as the weighted sums have less terms contributing.
The size of the randomly sampled s-x rank-1 lattice given to the moderately sized network was approximately three times larger than the deep network's. They were trained for the same number of epochs, so the moderately sized network trained more points. The positive affects of larger batch sizes is discussed in Section 5.4.1
The behaviour of the backpropagated gradients in both architectures was similar, with the variance of the outer layers remaining smallest and the first layer remaining largest throughout training [30], Figure 19.

The activation values and derivative activation values in both architectures, produce stable networks, which easily converge, Figure 20.



(a)



(b)

FIG. 19: log10(standard deviations of the back-propagated gradients of weights), of each layer in the network, during training. (a) 10 layer 60 hidden units GELU based network. (b) 3 layer 130 hidden unit GELU based network.

5.4.1. GPU vs CPU

The training of the derivative neural networks was limited by the equipment's memory, and parallel processing capacity [34]. The 10 layer 60 hidden unit architecture trained with the central processing unit (CPU), had a batch size 25 times smaller than this architecture trained with the graphical processing unit (GPU). This CPU trained architecture had 3.4 ± 0.3 digits of accuracy, approximately half a digit less than the GPU trained architecture, 3.8 ± 0.2 digits.
GPUs have thousands of cores that can perform tasks synchronously. This property is desired for deep learning, as the GPU can perform large expensive tensor calculations more efficiently than CPUs [34]. The CPU parallel processes data with lower efficiency than GPU.
Consequently, derivative networks that were trained with the GPU could have larger s-x lattice sizes (batch sizes), exposing the network to more s-x points over training. The contrast between the 10 layer 60 hidden units accuracy trained with the different machines, provides evidence that the accuracy of the network increases with the size of the rank-1 s-x lattice.

5.4.2. CPU Trained GELU Architectures

The importance of network architecture was investigated when the batch size was fixed at 500 s-x combinations per s-x lattice, for the 3 layer 100 hidden unit architecture (moderately sized network) and the 10 layer 30 hidden unit architecture (deeper network). Like the GPU results, the moderately sized network slightly outperformed the deeper architecture (by 0.1 digits). Therefore, there is sufficient evidence that moderately sized architectures produce more accurate integrals than deeper architectures. The accuracy is enhanced with large batch sizes. Below are the histograms from the CPU investigation, Figure 21.
In particular, the R-value histogram, Figure 21c, of the moderately sized network (3 layer 100 hidden units) is more narrow, hence more accurate. The main source of error for the CPU trained architectures was the noise in their estimates, Figure 21b (most values on the right).

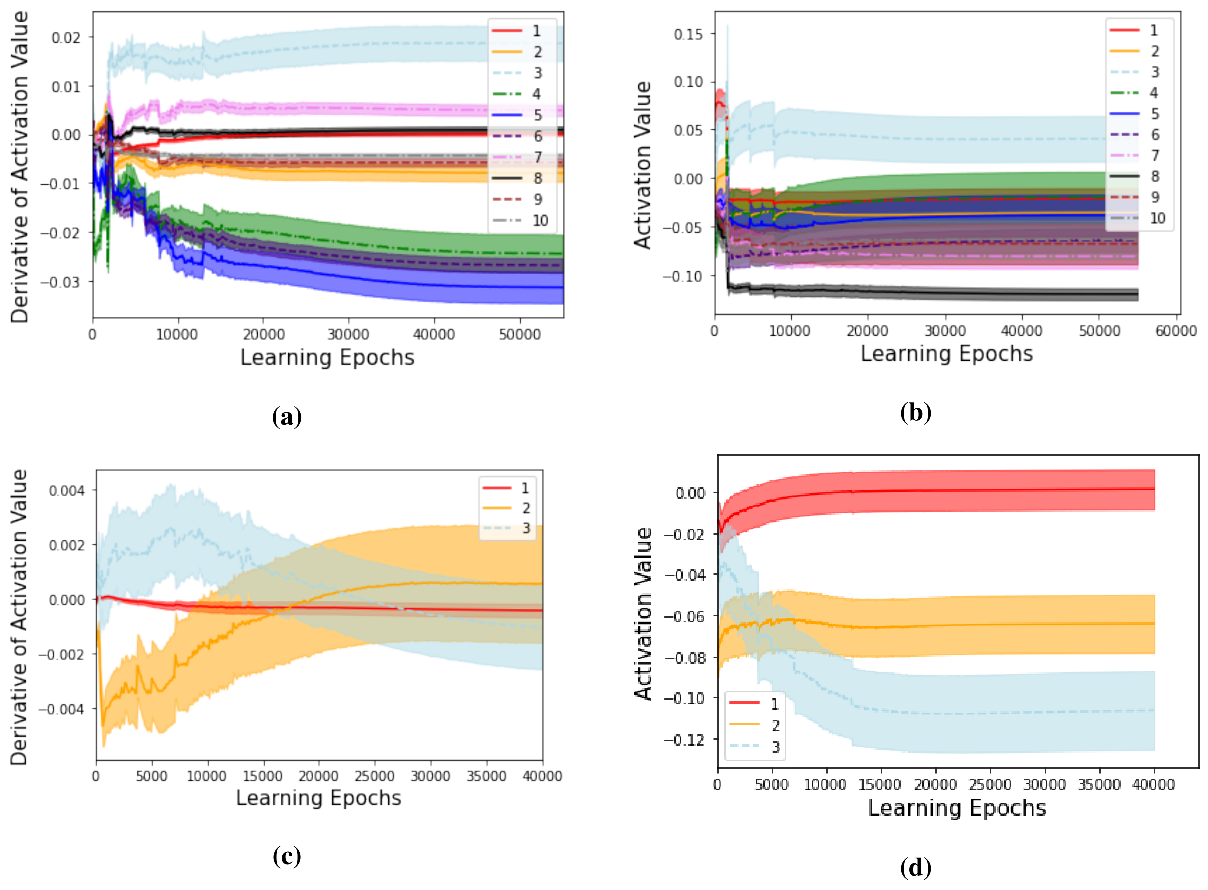


FIG. 20: Mean (central line) and standard deviations (std = boundary of shaded region), for each layer of the derivative neural network, collected during the training. The standard deviations (std) where divided by a factor. Subplots (a) and (b) were obtained from the 10 layer 60 hidden unit GELU based network, and subplots (c) and (d) were obtained from the 3 layer 130 hidden unit GELU based network. (a)(c) Activation values (std/20). (b)(d) Derivative activation values (std/100).

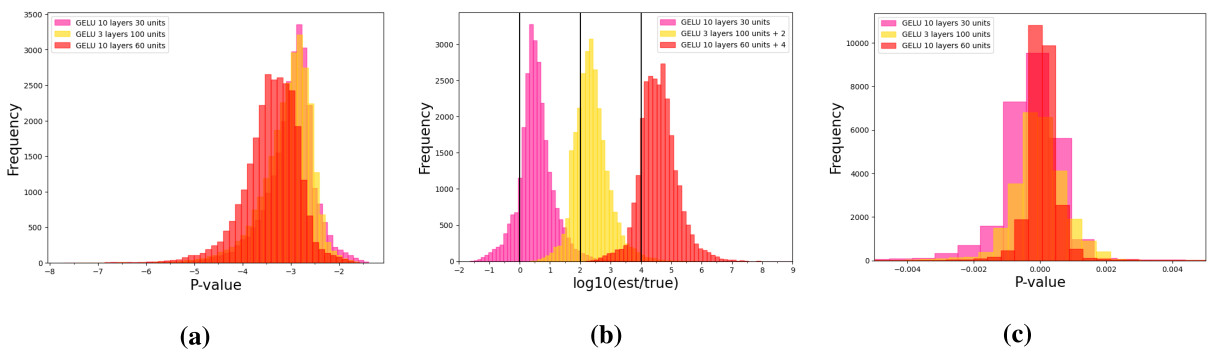
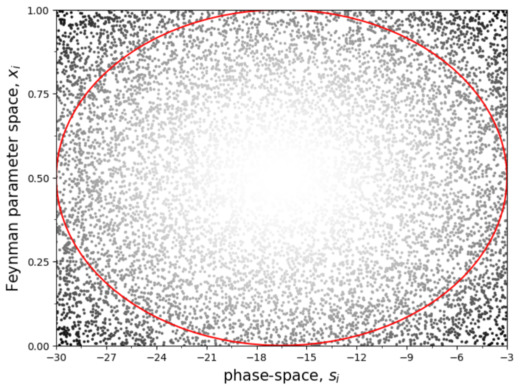


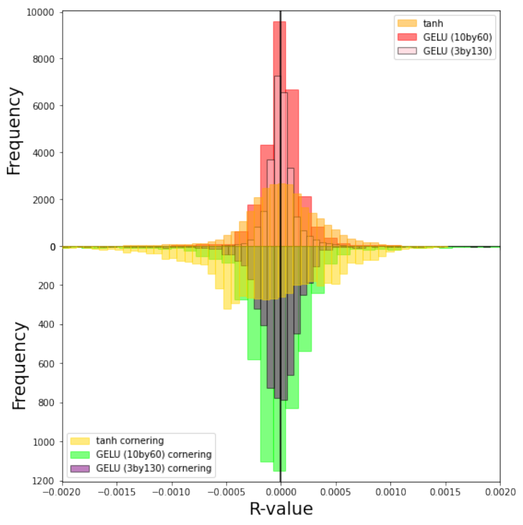
FIG. 21: CPU trained averaged GELU replicas Histograms. (details shown in the Table I) (a) P-value histogram, (b) Ratio of the uncertainty v and the true error. (c) R-value histograms

5.5. Cornering

The sampling of the s-x lattice was random. There was a higher density of sampled points at the central regions of the s-x space. Thus, the network experienced cornerness [2], where the network trained to the central region of the s-x space (sample space) more than the boundaries of the phase space, Figure 22a.



(a)



(b)

FIG. 22: (a)Schematic diagram showing cornering, were the brighter regions are the more densely sampled regions. The red ellipse is used as an aid to see the density (b) R-value histograms for the GPU trained tanh and GELU architectures for the whole phase space (top) and the outer values (bottom, inverted y axis) ($-30 < s_i < -29.5$ and $-3 > s_i > -3.5$)

Figure 22b shows the R-value histograms for the whole phase-space region (top) and the outer region (bottom, inverted y axis) for phase-space parameters lying in $-30 < s_i < -29.5$ and $-3 > s_i > -3.5$. The tanh outer region histogram is not centered at zero, so its estimates are significantly different from the true values.
The GELU demonstrates comparable levels of generalisation between the outer region and the whole phase-space region. It maintains its center (along 0), for the outer region, despite these points being trained less. However, its histograms are slightly broader. This indicates that the noise is high in these estimates, but their means are still accurate.

6. CONCLUSION

This paper continues the research of a new integration technique, that can solve Feynman parameterised integrals faster than the PySecDec MC integrator, by training over a phase-space region as well as the Feynman parameter space. It utilizes the parallel processing ability of GPU to produce integrals much faster than PySecDec. In this technique, the derivative of the network is fitted to the integrand of the integral, and the forward pass of the network was used to find the integral.
An emergent activation function, GELU was applied to the derivative network architecture. GELU based networks produced the most accurate integrals over the trained phase-space region (3.9 ± 0.2 digits), outperforming the previously tested tanh (3.4 ± 0.2 digits) and sigmoid

(2.9 ± 0.3 digits) activation functions. The shape of the GELU's first derivative made it less susceptible to dead node formation, than the classical activation functions. It also demonstrated robustness against the negative affects of cornering and lower batch sizes
The Deeper GELU architectures had lower accuracy than the moderately sized architecture. Further research is required to fully explain this, however it is potentially due to the higher prevalence of dead derivative activation nodes in the deeper structure.
The derivative neural network architecture is unique, as unlike many other neural networks, the shapes of its activation function's higher order derivatives also contribute to the learning of the network. This was supported by the softsign's poor accuracy (0.8 ± 0.2 digits), which demonstrated the importance that the higher order derivatives must be continuous for successful convergence.
The sigmoid, tanh, and GELU activation's errors were dominated by the noise present in integral estimates. Therefore, training more replicas would lower the uncertainty and improve the accuracy of the replica averaged architectures. Future research should focus on applying alternative activation functions, whose derivatives are less susceptible to vanishing and less computationally expensive.
Architectures achieved higher accuracy when trained on more s-x points (larger s-x training lattices). The accuracy was limited by the computational power of the training machines. Thus, further research into optimising the size of the network with the lattice size is required to find the perfect balance of phase-space exposure and representational capacity. The best sampling method should be investigated, as the central regions of the sample space are more densely populated with points (trained better). This technique can be applied to alternative research areas such as quantum circuits [35]. A new neural network structure that can integrate the physical region of phase-space should be investigated.

Acknowledgments

I would like to thank Professor Maître for our useful discussions on deep learning theories, alternative activation functions (GELU), and troubleshooting. Also, I give my thanks to Paul Clark, whom gave me access to the IPPP gpu machines.

References

[1] F. Bishara and M. Montull, "Machine learning amplitudes for faster event generation," *Physical Review D*, vol. 107, no. 7, p. L071901, 2023. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevD.107.L071901

[2] D. Maître and R. Santos-Mateos, "Multi-variable integration with a neural network," *Journal of High Energy Physics*, vol. 2023, no. 3, Mar. 2023. [Online]. Available: http://dx.doi.org/10.1007/JHEP03(2023)221

[3] S. Borowka, G. Heinrich, S. Jahn, S. P. Jones, M. Kerner, and J. Schlenk, "Numerical multiloop calculations: sector decomposition and qmc integration in pysecdec," *CERN Yellow Reports: Monographs*, vol. 3, pp. 185–185, 2020. [Online]. Available: https://doi.org/10.23731/CYRM-2020-003.185

[4] I. Hughes and T. Hase, *Measurements and their uncertainties: a practical guide to modern error analysis*. OUP Oxford, 2010. [Online]. Available: https://doi.org/10.1111/j.1751-5823.2011.00149_8.x

[5] R. Di Sipio, M. F. Giannelli, S. K. Haghighat, and S. Palazzo, "Dijetgan: a generative-adversarial network approach for the simulation of qcd dijet events at the lhc," *Journal of high energy physics*, vol. 2019, no. 8, 2019. [Online]. Available: https://doi.org/10.1007/JHEP08(2019)110

[6] E. Bothmann, T. Janßen, M. Knobbe, T. Schmale, and S. Schumann, "Exploring phase space with neural importance sampling," *SciPost Physics*, vol. 8, no. 4, p. 069, 2020. [Online]. Available: https://doi.org/10.21468/SciPostPhys.8.4.069

[7] M. D. Klimek and M. Perelstein, "Neural network-based approach to phase space integration," *SciPost Phys.*, vol. 9, p. 053, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1810.11509

[8] J. Bendavid, "Efficient monte carlo integration using boosted decision trees and generative deep neural networks," *arXiv preprint arXiv:1707.00028*, 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1707.00028

[9] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016. [Online]. Available: https://doi.org/10.48550/arXiv.1606.08415

[10] "A portrait of the higgs boson by the cms experiment ten years after the discovery," *Nature*, vol. 607, no. 7917, pp. 60–68, 2022. [Online]. Available: https://doi.org/10.1038/s41586-022-04892-x

[11] A. Collaboration *et al.*, "Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc," *arXiv preprint arXiv:1207.7214*, 2012. [Online]. Available: https://doi.org/10.1016/j.physletb.2012.08.020

[12] M. Mühlleitner, J. Schlenk, and M. Spira, "Top-yukawa-induced corrections to higgs pair production," *Journal of High Energy Physics*, vol. 2022, no. 10, pp. 1–15, 2022. [Online]. Available: https://doi.org/10.1007/JHEP10(2022)185

[13] J. Baglio, A. Djouadi, R. Gröber, M. M. Mühlleitner, J. Quevillon, and M. Spira, "The measurement of the higgs self-coupling at the lhc: theoretical status," *Journal of High Energy Physics*, vol. 2013, no. 4, pp. 1–40, 2013. [Online]. Available: https://doi.org/10.1007/JHEP04(2013)151

[14] T. Plehn, M. Spira, and P. Zerwas, "Pair production of neutral higgs particles in gluon-gluon collisions," *Nuclear Physics B*, vol. 479, no. 1-2, pp. 46–64, 1996. [Online]. Available: https://doi.org/10.1016/0550-3213(96)00418-X

[15] ——, "Pair production of neutral higgs particles in gluon-gluon collisions," *Nuclear Physics B*, vol. 479, no. 1-2, pp. 46–64, 1996. [Online]. Available: https://doi.org/10.1016/0550-3213(96)00418-X

[16] E. N. Glover and J. J. Van der Bij, "Higgs boson pair production via gluon fusion," *Nuclear Physics B*, vol. 309, no. 2, pp. 282–294, 1988. [Online]. Available: https://doi.org/10.1016/0550-3213(88)90083-1

[17] R. P. Feynman, "Space-time approach to quantum electrodynamics," *Phys. Rev.*, vol. 76, pp. 769–789, Sep 1949. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.76.769

[18] G. U. Jakobsen, G. Mogull, J. Plefka, B. Sauer, and Y. Xu, "Conservative scattering of spinning black holes at fourth post-minkowskian order," *Physical Review Letters*, vol. 131, no. 15, p. 151401, 2023. [Online]. Available: https://doi.org/10.1103/PhysRevLett.131.151401

[19] S. Lloyd, R. A. Irani, and M. Ahmadi, "Using neural networks for fast numerical integration and optimization," *IEEE Access*, vol. 8, pp. 84 519–84 531, 2020. [Online]. Available:

https://dx.doi.org/10.1109/ACCESS.2020.2991966

[20] N. M. Nawi, W. H. Atomi, and M. Z. Rehman, "The effect of data pre-processing on optimized training of artificial neural networks," *Procedia Technology*, vol. 11, pp. 32–39, 2013. [Online]. Available: https://doi.org/10.1016/j.protcy.2013.12.159

[21] D. P. Laurie, "Periodizing transformations for numerical integration," *Journal of computational and applied mathematics*, vol. 66, no. 1-2, pp. 337–344, 1996. [Online]. Available: https://doi.org/10.1016/0377-0427(95)00196-4

[22] F. Y. Kuo, I. H. Sloan, and H. Woźniakowski, "Periodization strategy may fail in high dimensions," *Numerical Algorithms*, vol. 46, pp. 369–391, 2007. [Online]. Available: https://doi.org/10.1007/s11075-007-9145-8

[23] G. Peter Lepage, "A new algorithm for adaptive multidimensional integration," *Journal of Computational Physics*, vol. 27, no. 2, pp. 192–203, 1978. [Online]. Available: https://doi.org/10.1016/0021-9991(78)90004-9

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014. [Online]. Available: https://doi.org/10.48550/arXiv.1412.6980

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986," *Biometrika*, vol. 71, pp. 599–607, 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1912.01703

[27] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," pp. 276–281, 2016. [Online]. Available: http://www.deeplearningbook.org

[28] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *arXiv preprint arXiv:1312.6120*, 2013. [Online]. Available: https://doi.org/10.48550/arXiv.1312.6120

[29] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," pp. 192–204, 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1412.0233

[30] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[31] J. Turian, J. Bergstra, and Y. Bengio, "Quadratic features and deep architectures for chunking," in *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers*, M. Ostendorf, M. Collins, S. Narayanan, D. W. Oard, and L. Vanderwende, Eds. Boulder, Colorado: Association for Computational Linguistics, Jun. 2009, pp. 245–248. [Online]. Available: https://aclanthology.org/N09-2062

[32] M. Lee, "Gelu activation function in deep learning: a comprehensive mathematical

analysis and performance," *arXiv preprint arXiv:2305.12073*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.12073

[33] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020. [Online]. Available: https://doi.org/10.1007/s11023-020-09548-1

[34] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," pp. 431–434, 2016. [Online]. Available: http://www.deeplearningbook.org

[35] J. M. Cruz-Martinez, M. Robbiati, and S. Carrazza, "Multi-variable integration with a variational quantum circuit," *arXiv preprint arXiv:2308.05657*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.05657

Summary For General Audience

The Large Hadron Collider (LHC) is transitioning from a particle discover to a precision machine. The LHC collides protons at very high energies. During these collisions lots of scattering events happen, where new particles are produced and interact with other particles. A large amount of data is collected during these collisions, and there are various techniques to calculate probabilities for specific processes occurring. One such event is Higgs boson pair production Figure 2, where two gluons fuse via a virtual top quark loop to produce two Higgs bosons.
This diagram has an associated scattering amplitude. Scattering amplitudes can be used to give probabilities with further computation. Currently, the most popular method to calculate scattering amplitudes is Monte Carlo (MC) integration. The MC integrators are slow. This paper introduces an alternative technique utilizing neural network technology. This approach can calculate scattering amplitudes approximately one hundred times faster than the popular MC integration program, PySecDec. There are added environmentally friendly benefits to this finding. It is still an ongoing area of research.

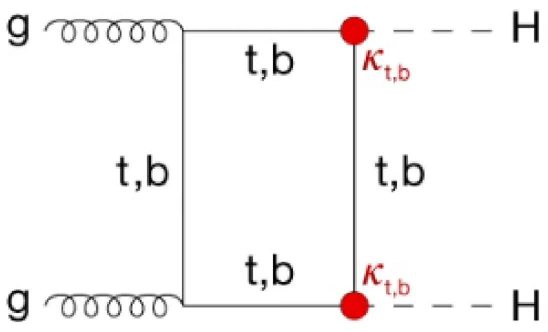


FIG. 2: Feynman diagram for Higgs boson pair production, from a virtual bottom/top quark loop. Figure 11 from the [10].

Appendix A: Activation Function Derivatives

Below the derivatives of the activation functions ϕ, used for the derivative neural network activation value differentials, are listed.
The derivatives for the Hyperbolic Tangent $(tanh(x) = T)$
$\phi(x) = T$
$\phi^{(1)}(x) = (1 - T)(1 + T)$
$\phi^{(2)}(x) = -2(1 - T)T(1 + T)$
$\phi^{(3)}(x) = -2(1 - T)(1 + T)(1 - 3T^2)$
$\phi^{(4)}(x) = 8(1 - T)T(1 + T)(2 - 3T^2)$
$\phi^{(5)}(x) = 8(1 - T)(1 + T)(2 - 15T^2 + 15T^4)$
$\phi^{(6)}(x) = -16(1 - T)T(1 + T)(17 - 60T^2 + 45T^4)$

The derivatives for the Logistic Sigmoid $(= \sigma)$
$\phi(x) = \sigma$
$\phi^{(1)}(x) = \sigma(1 - \sigma)$
$\phi^{(2)}(x) = \sigma(1 - \sigma)(1 - 2\sigma)$
$\phi^{(3)}(x) = \sigma(1 - \sigma)(1 - 6\sigma + 6\sigma^2)$
$\phi^{(4)}(x) = \sigma(1 - \sigma)(1 - 2\sigma)(1 - 12\sigma + 12\sigma^2)$
$\phi^{(5)}(x) = \sigma(1 - \sigma)(1 - 30\sigma + 150\sigma^2 - 240\sigma^3 + 120\sigma^4)$
$\phi^{(6)}(x) = \sigma(1 - \sigma)(1 - 2\sigma)(1 - 60\sigma + 420\sigma^2 - 720\sigma^3 + 360\sigma^4)$

The derivatives for the Softsign
$\phi(x) = \frac{x}{1+|x|}$
$\phi^{(1)}(x) = \frac{1}{(1+|x|)^2}$
$\phi^{(2)}(x) = -\frac{2x}{|x|(1+|x|)^3}$
$\phi^{(3)}(x) = \frac{6}{(1+|x|)^4}$
$\phi^{(4)}(x) = -\frac{24x}{|x|(1+|x|)^5}$

For the GELU:
$arg = \sqrt{\frac{2}{\pi}}(x + 0.044715x^3)$, $arg2 = \sqrt{\frac{2}{\pi}}(1 + 0.134145x^2)$,
$arg3 = \sqrt{\frac{2}{\pi}}(0.26829x)$, $arg4 = \sqrt{\frac{2}{\pi}}(0.26829)$
$\frac{d}{dx}\big(tanh(arg)\big) = arg2 \cdot sech^2(arg)$
$\frac{d^2}{d^2x}\big(tanh(arg)\big) = arg3 \cdot sech^2(arg) + arg2 \cdot \frac{d}{dx}\big(sech^2(arg)\big)$
$\frac{d^3}{d^3x}\big(tanh(arg)\big) = arg4 \cdot sech^2(arg) + 2 \cdot arg3 \cdot \frac{d}{dx}\big(sech^2(arg)\big) + arg2 \cdot \frac{d^2}{d^2x}\big(sech^2(arg)\big)$
$\frac{d^4}{d^4x}\big(tanh(arg)\big) =$
$arg4 \cdot \frac{d}{dx}\big(sech^2(arg)\big) + 2 \cdot arg4 \cdot \frac{d}{dx}\big(sech^2(arg)\big) + 2 \cdot arg4 \cdot \frac{d^3}{d^3x}\big(sech^2(arg) +$
$2 \cdot arg3 \cdot \frac{d^2}{d^2x}\big(sech^2(arg)\big)$

$\frac{d}{dx}\big(sech^2(arg)\big) = -2 \cdot arg2 \cdot tanh(arg) \cdot sech^2(arg)$
$\frac{d^2}{d^2x}\big(sech^2(arg)\big) =$
$-2\big(arg2 \cdot tanh(arg) \cdot \frac{d}{dx}\big(sech^2(arg)\big) + arg2 \cdot sech^2(arg) \cdot \frac{d}{dx}\big(tanh(arg)\big) + arg3 \cdot tanh(arg) \cdot$
$sech^2(arg)\big)$

$\frac{d^3}{d^3x}\big(sech^2(arg)\big) =$
$-\ 2\big(arg3 \cdot tanh(arg) \cdot \frac{d}{dx}\big(sech^2(arg)\big) + arg2 \cdot tanh(arg) \cdot \frac{d^2}{d^2x}\big(sech^2(arg)\big) + arg2 \cdot \frac{d}{dx}\big(tanh(arg)\big) \cdot \frac{d}{dx}\big(sech^2(arg)\big) + arg3 \cdot sech^2(arg) \cdot \frac{d}{dx}\big(tanh(arg)\big) + arg2 \cdot \frac{d}{dx}\big(sech^2(arg)\big) \cdot \frac{d}{dx}\big(tanh(arg)\big) + arg2 \cdot sech^2(arg) \cdot \frac{d^2}{d^2x}\big(tanh(arg)\big) + arg4 \cdot tanh(arg) \cdot sech^2(arg) + arg3 \cdot \frac{d}{dx}\big(tanh(arg)\big) \cdot sech^2(arg) + arg3 \cdot tanh(arg) \cdot \frac{d}{dx}\big(sech^2(arg)\big)\big)$

These are used to give the derivatives of the GELU:
$\phi(x) = 0.5x(1 + tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)))$
$\phi^{(1)}(x) = 0.5(1 + tanh(arg) + x\frac{d}{dx}\big(tanh(arg)\big)$
$\phi^{(2)}(x) = \frac{d}{dx}\big(tanh(arg)\big) + 0.5x\frac{d^2}{d^2x}\big(tanh(arg)\big)$
$\phi^{(3)}(x) = 1.5\frac{d^2}{d^2x}\big(tanh(arg)\big) + 0.5x\frac{d^3}{d^3x}\big(tanh(arg)\big)$
$\phi^{(4)}(x) = 2\frac{d^3}{d^3x}\big(tanh(arg)\big) + 0.5x\frac{d^4}{d^4x}\big(tanh(arg)\big)$

Appendix B: Optimiser

The network parameters θ were updated with the ADAM [24] optimiser, in a training loop. ADAM combines the best assets from adaptive learning rate and momentum based optimisers - making it highly effective for training deep neural networks. It is computationally efficient, widely used, and easy to implement. ADAM adapts the learning rate for individual network parameters θ, by tracking an exponential moving average for the first moment s_t, and second moment r_t. The moments are controlled by the exponential decay rates ρ_1 and ρ_2 respectively, and are bias corrected. The decay rate hyper-parameters did not require tuning for the network to converge to good local minima. The algorithm is shown below.

Algorithm 1 ADAM algorithm

```
Require: $\epsilon$: learning rate
Require: $\rho_1, \rho_2 \in$ [0,1): Exponential decay rates for moment estimates, set to 0.9 and 0.999 respectively
Require: small constant $\delta$ used for numerical stabilisation
Require: $\theta_0$: Initial parameters
  $s_0 = 0$ and $r_0 = 0$: Initialise 1st and 2nd moment variables
  $t = 0$: Initialise time step
  while $\theta_t$ not converged do
      $g_t \leftarrow \nabla_\theta L(\theta_t)$: compute gradients of the loss function
      $t \leftarrow t + 1$
      $s_t \leftarrow \rho_1 \cdot s_{t-1} + (1 - \rho_1) \cdot g_t$: update biased first moment estimate
      $r_t \leftarrow \rho_2 \cdot r_{t-1} + (1 - \rho_2) \cdot g_t^2$: update biased second moment estimate
      $\hat{s_t} \leftarrow \frac{s_t}{1-\rho_1^t}$: correct bias in first moment
      $\hat{r_t} \leftarrow \frac{r_t}{1-\rho_2^t}$: correct bias in second moment
      $\Delta\theta = -\epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$: operations applied element wise
      $\theta \leftarrow \theta + \Delta\theta$: update parameters
  end while
```

Appendix C: Machine Performance Parameters

Number of CPU Cores	Number of Threads	Base Clock Speed	Max Turbo Clockspeed	L3 cache
8	16	3.6 GHz	4.4 GHz	8MB

TABLE III: Paramters of the CPU AMD Ryzen 7 Processor