Evaluating Activation Functions with nAI and Quadrature Based Neural Networks

John C. Breedis[*]

Project Advisor: Dr. Tan Bui-Thanh[†]

Abstract. We provide a new perspective on comparing activation functions via neural network approximate identity (nAI), a family of bell-curve activation functions $\mathcal{B}_\theta$ with parameter $\theta > 0$. This was done by comparing the performance of several neural networks across multiple data sets, where each neural network has one hidden layer and uses nAI as activation functions. The data sets used include polynomial, discontinuous, and non-differentiable regression problems, as well as MNIST classification. Across all regression problems, hinge activation functions outperformed sigmoidal activation functions. For classification, sigmoidal activation functions perform slightly better than hinge functions. In addition to traditional neural networks, we develop a machine learning architecture that utilizes integral convolution and quadrature by fixing the initial weight and bias matrices of the network. Supplemental modifications were implemented to improve the accuracy of these networks. This modified architecture was used alongside standard Feedforward networks across each regression problem with similar efficacy.

1. Introduction.

Within the past decade machine learning has advanced tremendously, having the capability to identify trends in a variety of areas. Whether it be image classification, natural language processing, or scientific modeling, machine learning models are designed to best identify these trends and mimic them effectively. From the early Perceptron, to the modern Transformer models like ChatGPT, AI and machine learning models have the potential to mimic human capabilities.

One particular model that revolutionized the field is the artificial neural network (ANN). ANNs have found success in many fields, including image classification, model regression, PDEs, etc. In some of these cases, ANNs have proven to outperform more traditional methods.

The power behind an ANN derives from its composition of affine transformations and nonlinear, universal activation functions. A function is universal if it can be used to approximate any other function within a specified space. These few necessary conditions produce a wide variety of activation functions.

Due to their non-linear nature, activation functions are a critical component in each ANN model's ability to find solutions. However, with so many to choose from, the question often arises: which activation functions are better for each application? This has proven to be a difficult question, as most solutions are obtained via trial and error, or methods like Bayesian optimization.

While there have been several attempts in the past to compare activation functions on ANN models, each one has its own limitations. In this paper, we aim to contribute to this field of research by offering a more fair comparison between activation functions. This will be done utilizing a recent development by Bui-Thanh on the universality of several activation functions on shallow neural networks [1]. In his paper, he establishes the concept of a neural network approximate identity (nAI). We plan to use this structure and the theory involved to directly compare several traditional activation functions. This comparison involves applying shallow ANNs to several regression problems, an MNIST classification problem, as well as an introduction to an ANN architecture which utilizes quadrature methods to represent the method of approximation described in the paper. This Quadrature Based Neural Network (QBNN) will also be applied to the regression data as a means to measure the efficacy of each

[*]University of Texas at Austin, Department of Mathematics (jbreedis@utexas.edu)

[†]University of Texas at Austin, Oden Institute (tanbui@oden.utexas.edu)

activation function according to the quadrature method.

The paper is organized as follows. We will go into more detail regarding previous activation function comparisons, while primarily focusing on the theory of nAI in section 2. From there, we will describe the experiments used to compare our activation functions in section 3. This includes the activation functions being compared, the data sets our networks will approximate, as well as the different neural network architectures used. This will be immediately followed by section 4, where we will explore the QBNN architecture. In particular, we start by verifying that the quadrature rule from section 2 holds true. Afterwards, we generalize the quadrature rule to define a shallow ANN, apply this model to a particular regression problem, and conclude by introducing features to improve upon the QBNN's accuracy. This is once again done to compare the performance of nAI at approximating functions by their intended design. This is subsequently followed by results in section 5 for problems defined in section 3. Finally, we will give concluding remarks in section 6.

2. Literature Review.

There have been numerous experiments that compared activation functions over a variety of tasks. One such case is Pomerat, Segev, and Datta [5], who had the idea to generate a random polynomial function, and compared a handful of activation functions alongside optimizers. This paper, however, only considered ReLU, Sigmoid, and Tanh activation functions, as well as only considering smooth target functions. Another study by Eger, Youssef, and Gurevych [3] compared a wide variety of activation functions across several types of DNN, including Multi-layer Perceptron, Convolutional Neural Network, and Recurrent Neural Network. Others have considered uncommon activation functions, such as Radial Based Functions, Conic Section Functions, Exponential, Sine, etc. [4, 2]. All of these studies only compare a handful of activation functions, or evaluate them over very specific tasks. Our study hopes to give a more fair comparison over many activation functions via the introduction of nAI.

Most of the concepts used in this paper stem from Bui-Thanh [1], *A Unified and Constructive Framework for the Universality of Neural Networks*. In this paper, Bui-Thanh provides a unified and constructive framework for identifying when an activation function is universal using a neural network approximate identity (nAI).

An nAI is defined as a family of function $\mathcal{B}_\theta \in \mathcal{L}^1(\mathbb{R}^n)$ for $\theta > 0$ where

1. $\mathcal{B}_\theta$ is bounded in $\mathcal{L}^1$,
2. $\int_{\mathbb{R}^n} \mathcal{B}_\theta(x)dx = 1 \;\; \forall\theta > 0$, and
3. $\int_{||x||>\delta} |\mathcal{B}_\theta(x)|dx \to 0$ as $\theta \to 0$ for any $\delta > 0$

This is all analogous to say that the family of functions $\mathcal{B}_\theta$ converges to the Dirac delta function as $\theta \to 0$. This is best visualized by Figure 2.1, which displays some nAI at various θ values. One can see how $\mathcal{B}_\theta^{\tanh}$ becomes sharper as θ decreases. We can therefore refer to θ as a "spread parameter" of $\mathcal{B}_\theta$.

It is proven in Bui-Thanh [1] that $\forall f \in \mathcal{C}_0(\mathbb{R}^n)$,

$$\lim_{\theta\to 0} ||f * \mathcal{B}_\theta - f||_\infty = 0$$

where

$$(f * g)(x) := \int_{\mathbb{R}^n} f(x-y)g(y)dy \tag{2.1}$$

is the convolution of two functions f, g. The convolution integral $f * \mathcal{B}_\theta$ can be approximated via quadrature methods, therefore the quadrature must also approximate f as $\theta \to 0$ and the number of quadrature nodes $N \to \infty$. In one dimensional space, this quadrature

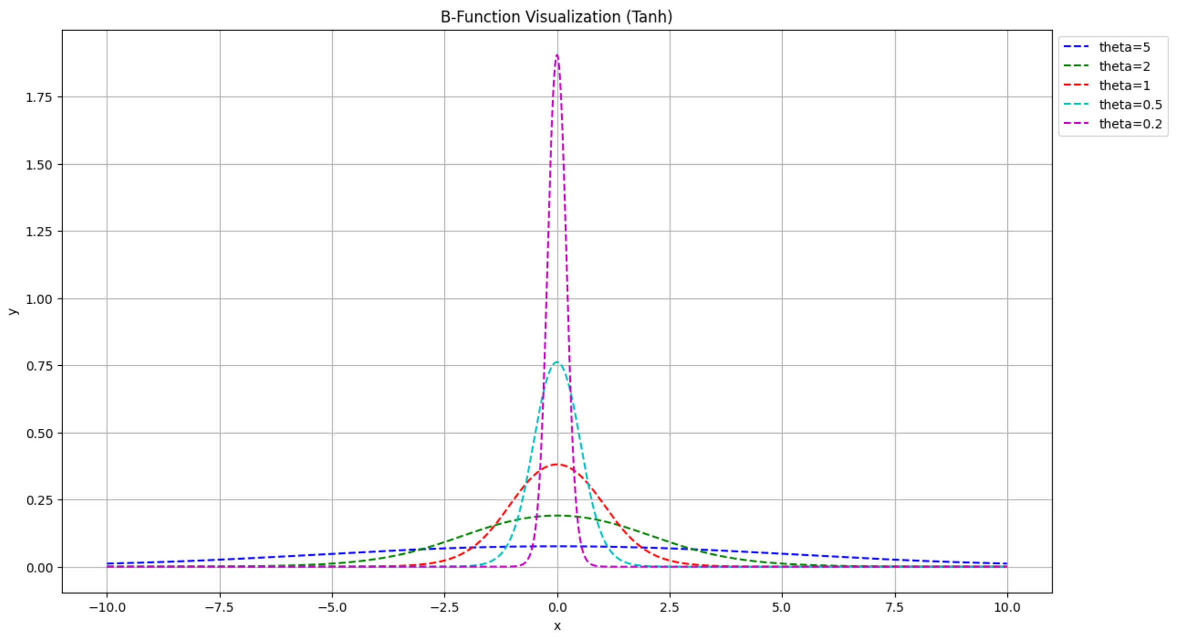


Figure 2.1: nAI of $\mathcal{B}_\theta^{\text{Tanh}}$ at various θ values

approximation becomes

$$f(x) \approx \sum_{i=1}^{N} q_i \mathcal{B}_\theta(x - y_i) f(y_i) \tag{2.2}$$

where $\{y_i\}_{i=1}^N$ are quadrature points with corresponding weights $\{q_i\}_{i=1}^N$.

One can show that many existing activation functions are universal by constructing a $\mathcal{B}_\theta$ function as a linear combination of such activation functions. We list the activation functions used during our experiments, as well as their nAI counterpart in subsection 3.1. By simplifying (2.2), we inevitably conclude with

$$f(x) \approx W_1\sigma(W_0 x + \vec{b}_0) + \vec{b}_1 \tag{2.3}$$

where W_0, b_0, W_1 and b_1 are arbitrary weight and bias vectors, and σ is our traditional activation function. In other words, we've shown that a neural network with one hidden layer can approximate any function $f \in \mathcal{C}_0(\mathbb{R})$ if the hidden layer is large enough and σ has an nAI representation.

Most traditional neural networks use the setup described in (2.3). In our experiments, we will use a similar structure to compare traditional activation functions using their nAI form. More details will be given in subsection 3.3.

3. Activation Function Comparison Methods.

In this section, we will discuss which activation functions we will be comparing, as well as the procedure used to measure each activation function's effectiveness.

3.1. Activation Function Types.

From Bui-Thanh [1] we know there are theoretically an endless number of activation functions. Most known activation functions can be separated into two groups: hinge and sigmoidal. Hinge functions are unbounded as $x \to \infty$ and have a descriptive "hinge" shape, while sigmoidal functions are always bounded and have an "S" shape. Both types are monotonic increasing. These categories are best visualized by ReLU and tanh respectively. For the purposes of our experiments, we limit the range of activation functions to those listed below:

$$\text{(3.1)}\quad \text{RePU}_q(x) = \begin{cases} x^q & \text{if } x\geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \qquad \text{ELU}(x) = \begin{cases} (e^x-1) & x\leq 0 \\ x & x>0 \end{cases}$$
$$\text{GELU}(x) = x\Phi(x) \qquad \text{SiLU}(x) = \frac{x}{1+e^{-x}} \qquad \text{Mish}(x) = x\tanh(\ln(1+e^x))$$

where $\Phi(x)$ is the Cumulative Distribution Function (CDF) of the standard normal function. Note that ReLU is equivalent to $\text{RePU}_{q=1}$. The activation functions listed above come from the distinct categorizations listed in Section 7 of Bui-Thanh [1].

In order to compare these traditional activation functions, we will use their nAI counterparts. If we denote hinge and sigmoidal activation functions as σ_h and σ_s respectively, we can define each nAI as follows in (3.2)–(3.4).

$$\text{(3.2)}\qquad \mathcal{B}_\theta^{\text{RePU}}(x) = \frac{1}{q!}\sum_{i=0}^{q+1}(-1)^i\binom{q+1}{i}\frac{\text{RePU}_q\left(x/\theta + \left(\frac{q+1}{2} - i\right)\right)}{\theta}$$

$$\text{(3.3)}\qquad \mathcal{B}_\theta^{\text{Sigmoid}}(x) = \frac{\sigma_s(x/\theta+1) - \sigma_s(x/\theta-1)}{2\theta}$$

$$\text{(3.4)}\qquad \mathcal{B}_\theta^{\text{Hinge}}(x) = \frac{\sigma_h(x/\theta+1) - 2\sigma_h(x/\theta) + \sigma_h(x/\theta-1)}{2\theta}$$

Equation (3.3) applies to tanh, (3.2) applies to RePU, while the remaining activation functions use (3.4). We can visualize and compare these functions with Figure 3.1.

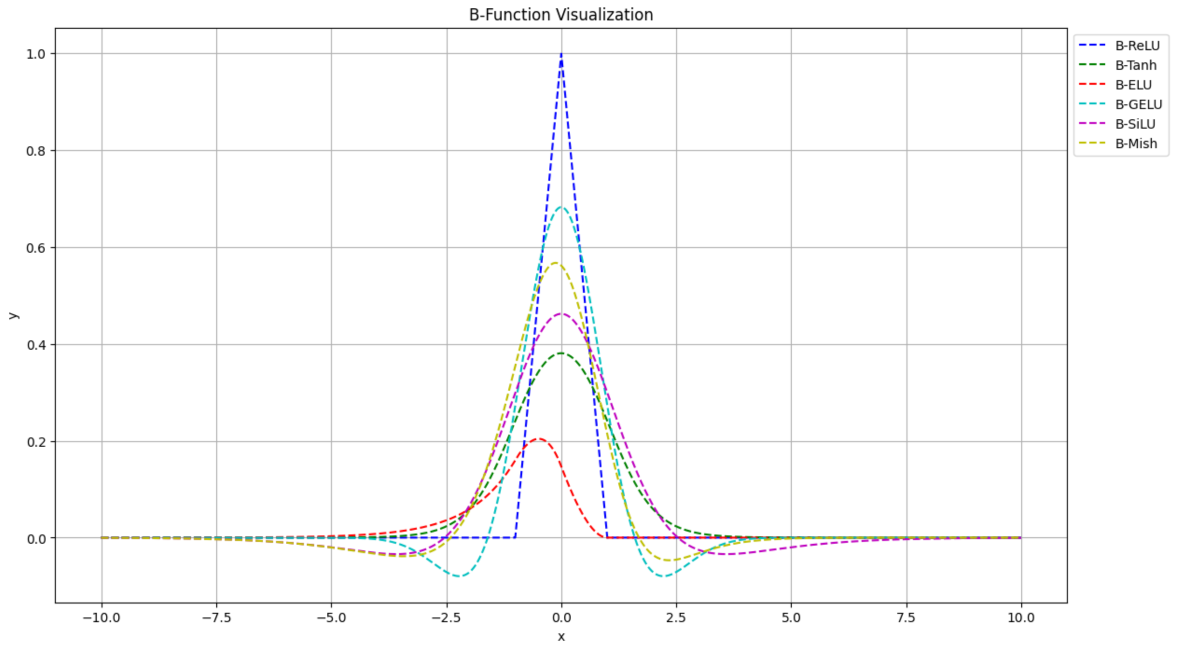


Figure 3.1: nAI of several traditional activation functions with $\theta = 1$

3.2. Problem Sets Being Used.

Regarding regression based problem, there are an endless range of functions to approximate. In order to measure how each activation function performs over a wide variety of tasks, one course of action is to generate many random target functions and average the results. This concept was used in Pomerat, Segev, and Datta [5] to generate smooth polynomial functions. In our experiments, we'll use a similar process, as well as modify it to also generate discontinuous and non-differentiable target functions. This is done to determine if such differences impact an activation function's efficacy.

To be specific, we can generate degree M polynomials, discontinuous functions, and non-differentiable functions. These are defined as follows:

$$p_{\text{poly}}(x) = a_0 + a_1 x + \ldots + \frac{a_M}{M!}x^M \tag{3.5}$$

$$p_{\text{disc}}(x) = a_0 + a_1 x + \ldots + \frac{a_M}{M!}x^M + \mathbb{1}_{f(x)>0} \cdot \left[b_0 + b_1 x + \ldots + \frac{b_M}{M!}x^M\right] \tag{3.6}$$

$$p_{\text{diff}}(x) = a_0 + a_1 x + \ldots + \frac{a_M}{M!}x^M + \left|b_0 + b_1 x + \ldots + \frac{b_M}{M!}x^M\right| \tag{3.7}$$

where $a_i, b_i \sim \mathcal{N}(0,1)$ are random and normally distributed $\forall i$. In (3.6), $\mathbb{1}_{f(x)>0} = \begin{cases} 1 & f(x)>0 \\ 0 & f(x)\leq 0 \end{cases}$ is an indicator function where $f(x)$ is designed to be some arbitrary, periodic function, in order to create multiple discontinuities at various points. In our experiments, $f(x) = \sin(q(x))$ where $q(x)$ is another arbitrary polynomial of degree M, as constructed in (3.5). The absolute value used in (3.7) is used to create random cusps that make the function non-differentiable, yet continuous. Unlike the functions in Pomerat, Segev, and Datta [5], the coefficients are scaled such that the higher powers do not dominate at extreme x values.

In our experiments, we evaluate each activation function over 50 random functions and average the resulting MSE for each activation function. This is done for each function type (3.5)–(3.7). The highest power $M = 7$ in every case. These 50 random functions of each category, with coefficients of terms as high of an order as 7, should collectively give a representation of real-valued functions found when designing a regression. Consequently, the resulting statistics should provide a robust metric of each activation function's efficacy at modeling most real-valued functions.

In order to compare these activation functions on the random functions established above, we need to train neural networks on artificial data sets that model these random functions. Similar to Pomerat, Segev, and Datta [5], the data sets are composed of 1000 features $x_i \sim \mathcal{N}(0,1)$. Each feature has a corresponding label $y_i = f(x_i) + \varepsilon_i$, where $f(x)$ is some random function defined in (3.5)–(3.7) and $\varepsilon_i \sim \mathcal{N}(0, 0.04)$ is random noise. The neural network architectures are established in subsection 3.3, as well as the training process used for these artificial data sets.

In terms of classification, we train a Feedforward model with nAI over the MNIST data set with a batch size of 128. More details about the network will follow in subsection 3.3.

3.3. Architecture. Every experiment conducted uses a multi-layer perceptron (MLP) architecture with one hidden layer. This layer will be composed of $N = 10$ neurons for regression. Each model will use the $\mathcal{B}_\theta$ form of each activation function listed in (3.1). Recall that $\mathcal{B}_\theta$ functions have θ as an additional parameter. We will initially train each model with $\theta = 0.1$ by default. Our first model (denoted Fixed θ's) keeps this parameter constant.

While $\theta = 0.1$ serves well as a default value, the accuracy should improve if θ can be optimized. This way, we can truly compare how each activation function performs at their best. A second model (denoted Trainable θ's) can train the θ_i of each $\mathcal{B}_{\theta_i}$ instance alongside the weights and biases of the previous model.

The ability to train parameters like θ is possible due to the JAX Python library, which is first tested and confirmed in subsection 4.3. The model described in that section is also used for comparison (denoted Quadrature Based or QBNN).

All of the coding is done in Python. The machine learning model architectures are coded manually using JAX to update the weights and biases directly. Of the three architectures, Trainable θ's is closest to the standard MLP architecture, while the other two are variations with fixed parameters. Every model trains for 200 epochs with a step size of 0.05 using the Adam optimizer. The loss function is Mean Squared Error (MSE).

Regarding classification problems, the data's shape and complexity require a few changes. The classification model will follow a $28 \times 28 \to 1024 \to 10$ standard Feedforward architecture. Additionally, θ parameters are used and are trained much like with the Trainable θ's model. Each classification model trained for 10 epochs with a batch size of 128, and a step size of 10^{-3}. The loss function for training is the multi-class cross-entropy loss function, while testing error will be calculated by both classification accuracy and F1 score.

4. Quadrature Based Models.

As mentioned in section 2, these nAI are designed to approximate functions via quadrature. This is displayed in (2.2). If nAI converge to a solution via quadrature, then a neural network that can mimic quadrature should be able to effectively model any target function.

In this section, we will consider the construction of Quadrature Based Neural Networks (QBNN). Initially, we will evaluate (2.2) directly on some example function. From there, we will construct the basic framework of these QBNN. We will conclude with several additions to the QBNN model with the purpose of increasing its accuracy. This includes the Quadrature Based architecture mentioned in subsection 3.3.

The example function used in this section is given by (4.1). The neural networks are trained over 2000 uniformly distributed feature points $x_i \sim [-1, 1]$, with $y_i = f(x_i)$ as corresponding labels. Unless stated otherwise, assume $\mathcal{B}_\theta = \mathcal{B}_\theta^{\text{Tanh}}$, which uses (3.3) for $\sigma_s = \tanh$.

$$f(x) = \sin(3.25\pi x)e^{2.59x} \tag{4.1}$$

4.1. Evaluating nAI Quadrature.

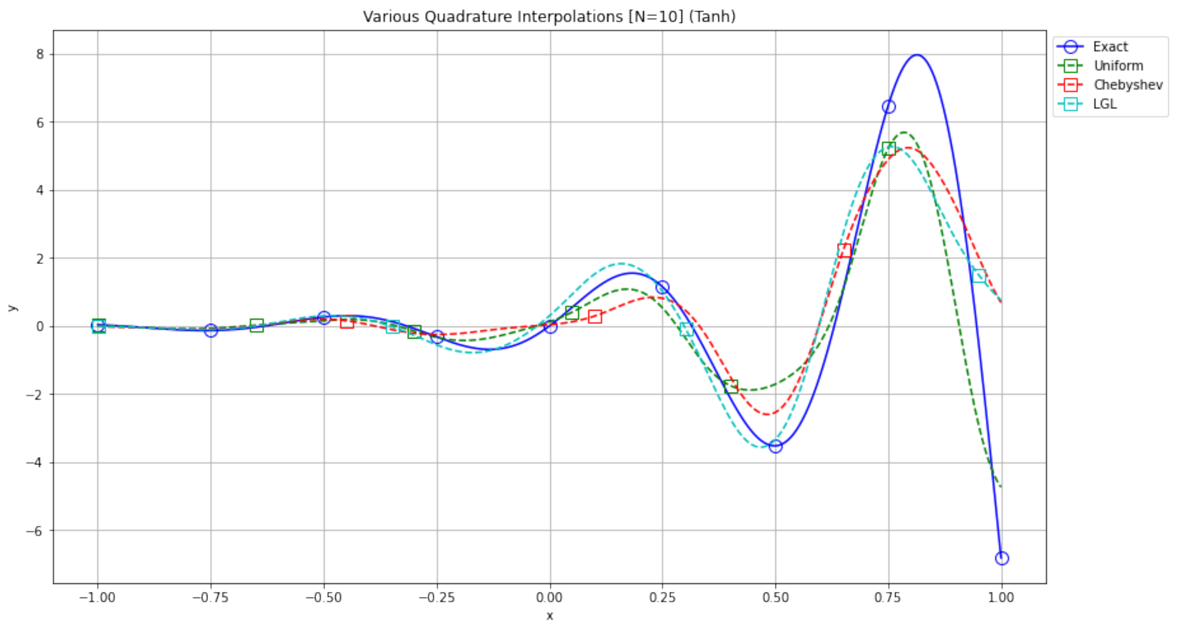
In order to evaluate the effectiveness of nAIs, we applied the quadrature approximation (2.2) to various quadrature rules and compared their performance to the function we are approximating. Our approximation metric is Mean Square Error (MSE).

The quadrature methods we chose were Uniform, Chebyshev, and Gaussian (LGL). Using each method's set of N nodes $\{y_i\}_{i=1}^N$ and weights $\{q_i\}_{i=1}^N$, we can evaluate the series approximation (2.2) across each method. Given some N quadrature points, we defined $\theta = 1/N$ as an effective spread constant. The theory states that as $N \to \infty$ and $\theta \to 0$, the series should converge to the stated function.

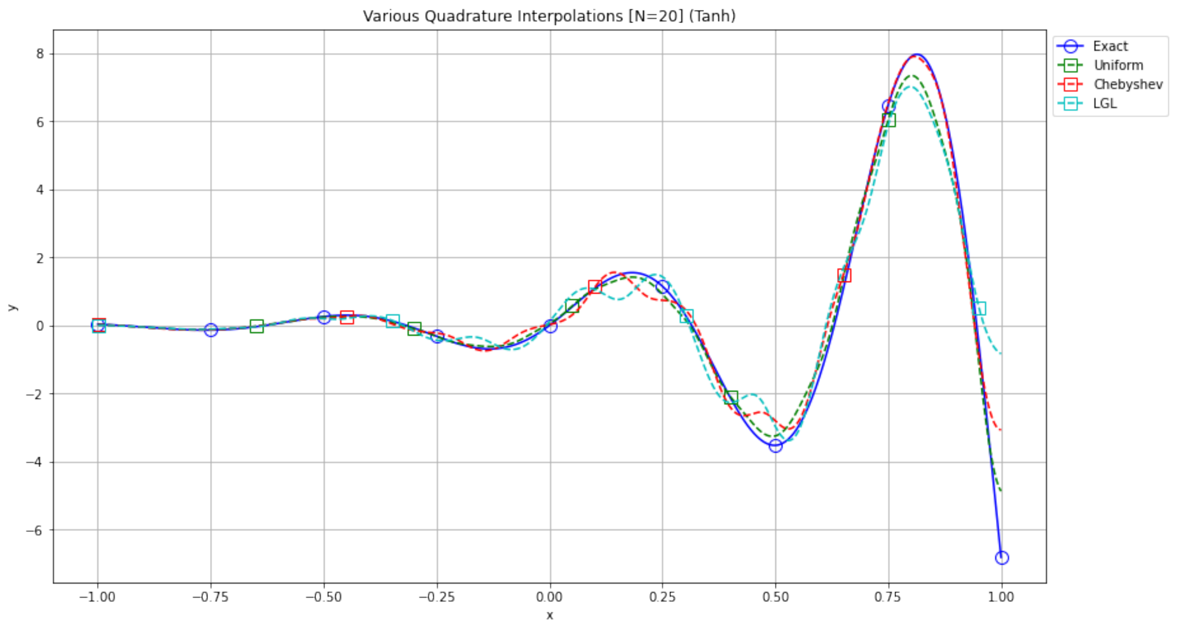
To test this theory, we compared these approximations to the true function for two different N values: $N = 10$ and $N = 20$. If the theory holds true, the $N = 20$ approximations should be better than the $N = 10$ approximations, regardless of the quadrature method. The results are given in Figures 4.1a and 4.1b. The solid blue line is the example function (4.1) being approximated, while the dashed lines correspond to Uniform, Chebyshev, and LGL quadrature methods.

In Figure 4.1a, the MSE ranges from <u>1.3075 to 1.5165</u>. As seen in the figure, the approximations capture the general shape of the target function, however they struggle to match the precise amplitudes.

Compare this to Figure 4.1b, where the MSE ranges from <u>0.0974 to 0.5188</u>. By doubling the number of quadrature points, the accuracy of the series approximations increases significantly. This leads us to confirm that the quadrature converges as the number of nodes increases and the spread θ deceases.



(a) Series Approximations ($N = 10$)



(b) Series Approximations ($N = 20$)

Figure 4.1: Using the series (2.2) to approximate $f(x)$ with each quadrature method

4.2. Training Simple Neural Net. In order to replicate the series approximation (2.2) with a neural network, we can re-imagine the sum as:

$$F(x) = \sum_{i=1}^{N} w_i \sigma(\vec{x}_i - \vec{b}_i) \tag{4.2}$$

where $\vec{x} = \begin{bmatrix} x & x & \cdots & x \end{bmatrix}^T$ is an N dimensional vector with x as each entry, $\vec{b} = [y_i]^T_{1\le i\le N}$ is the first bias vector and a vector of quadrature points, σ is our activation function $\mathcal{B}_\theta$, and w_i are the trainable weights. For our first model, only the w_i are trainable, since both $\vec{x}$ and $\vec{b}$ are fixed.

In this section we trained a model with $N = 10$ nodes, using $\mathcal{B}_\theta = \mathcal{B}_\theta^{\text{Tanh}}$ as defined in (3.3) as our activation function. Only LGL quadrature was used, as it performed the best compared to the other quadrature methods. The model was trained using the Adam optimizer with a step size of 10^{-2} over the course of 500 epochs. Since our model has $N = 10$ nodes, we fixed $\theta = 0.1$, much like the approximations in subsection 4.1. After training, the neural network returned its approximation in Figure 4.2. Much like the previous figures, the solid blue line represents the target function, while the dashed green line is the model's prediction.

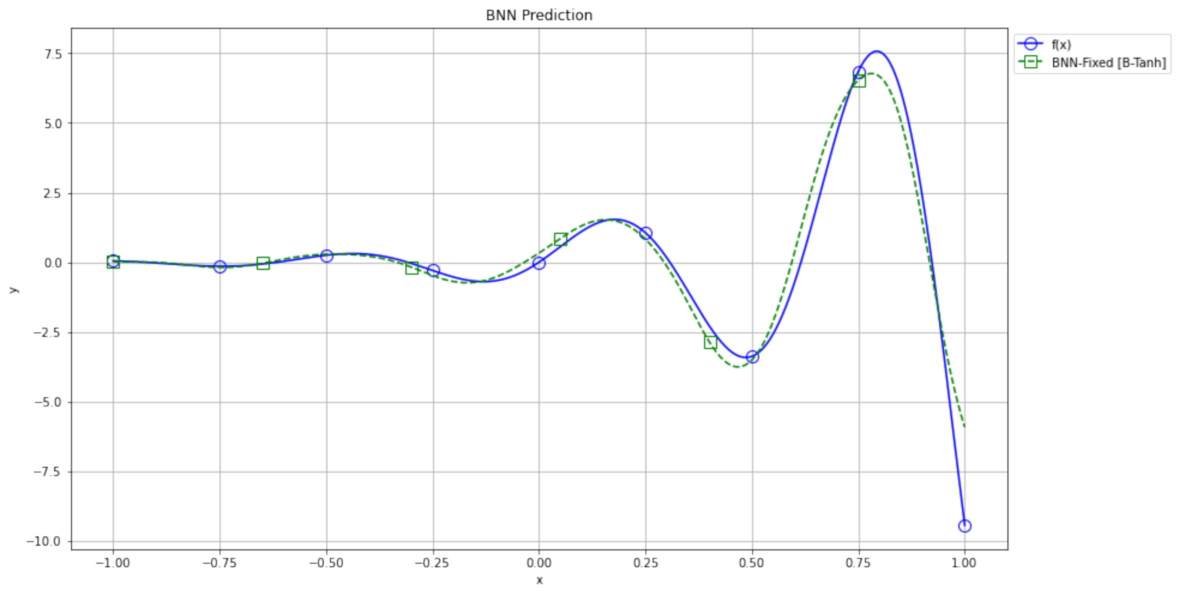


Figure 4.2: QBNN Model Predictions ($\theta = 0.1$ fixed)

The MSE of Figure 4.2 is <u>0.3010</u>. This surpasses the results of Figure 4.1a, which had a minimum MSE of 1.3075.

4.3. Implementing Spread Parameters. Throughout our past experiments, we've assumed that the spread parameter $\theta = 1/N$ is a good initial guess. Realistically, this parameter will vary depending on a variety of factors. If we can train θ alongside the weight vector, our model should improve.

Normally training a hyperparameter like θ is a difficult task, however by utilizing JAX's gradient method we can modify θ much like any other parameter. By using the gradient method such that we can update θ's alongside $\vec{w}$ as defined in (4.2), we obtain our new model variation.

By defining $\vec{\theta} = \left[\theta_i\right]_{1\le i\le N}$ as a trainable parameter, we can evaluate the Quadrature-Based Neural Network variation as given by (4.3).

$$F(x) = \sum_{i=1}^{N} w_i \mathcal{B}_{\theta_i}(x - \vec{b}_i) \tag{4.3}$$

This way, the spread parameter θ can vary for the nAI at each quadrature point, improving upon the potential accuracy of the neural network model. This new model variation will have the same properties as subsection 4.2, with the only difference being $\theta^0 = 0.1$ will be our initial value of each θ_i, rather than some fixed value.

After training this variation for 500 epochs, the model outputs its prediction in Figure 4.3. As is the case with the previous figures, the solid blue line is our target function (4.1), while the dashed line is the approximation.

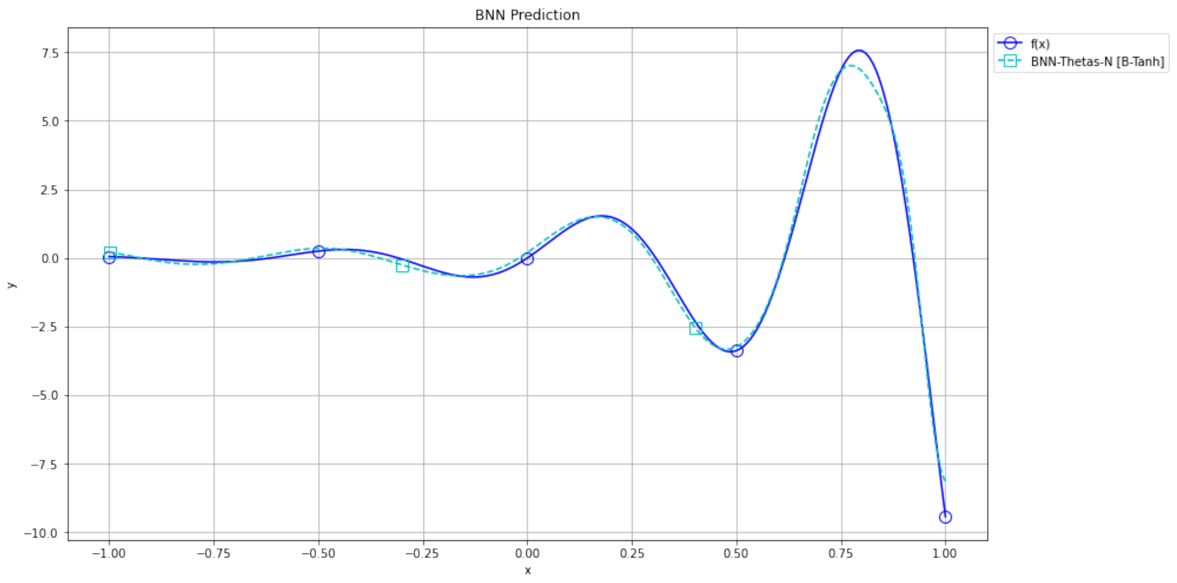


Figure 4.3: QBNN Model Predictions (θ's given in Table 4.1)

To get an idea of how the spread θ_i can vary across the domain, Table 4.1 displays each resulting θ_i with respect to the quadrature point where the corresponding nAI is evaluated. We observe how the spread θ_i decreases as $x \to 1$, causing the nAI to become sharper. This is due to how θ affects spread, as nAI are depicted in (3.3) and Figure 2.1.

Table 4.1: $\mathcal{B}^{\text{Tanh}}_{\theta_i}$ instances by their quadrature point x_i and spread θ_i

i	1	2	3	4	5	6	7	8	9	10
x_i	-1.000	-0.920	-0.739	-0.478	-0.165	0.165	0.478	0.739	0.920	1.000
θ_i	0.3472	0.2834	0.1877	0.2099	0.2058	0.1285	0.1192	0.0774	0.1096	0.0546

The MSE of Figure 4.3 is $\underline{0.0588}$. This is another significant step from Figure 4.2, which had a MSE of 0.3010.

4.4. Using More Flexible B-Functions. In addition to optimizing parameters like the amplitude and spread of nAI, there is still one factor we have yet to optimize: shape. Each preceding model was given an explicit activation function to use. Perhaps the model will improve if given the ability to reshape the activation function itself at each instance.

To do so, the activation function $\mathcal{B}^{\text{RePU}}_{\theta}$ will be used, which is formalized in (3.2). Much like how we trained θ, we can train the RePU parameter q, which modifies the shape of the nAI in a different manner. To understand how the parameter q changes the shape of the nAI, consider Figure 4.4. We can see how, as $q \to \infty$, the $\mathcal{B}^{\text{RePU}}_{\theta}$ function approaches a uniform distribution $u(x) \sim \begin{cases} 1/a & |x| \le a/2 \\ 0 & |x| > a/2 \end{cases}$ for some $a > 0$. By training both $\vec{\theta}$ and $\vec{q}$, the neural network should be able to improve its potential accuracy even further.

While (3.2) considers $q \in \mathbb{N}$, it can be generalized to non-integer q so long as we sum and

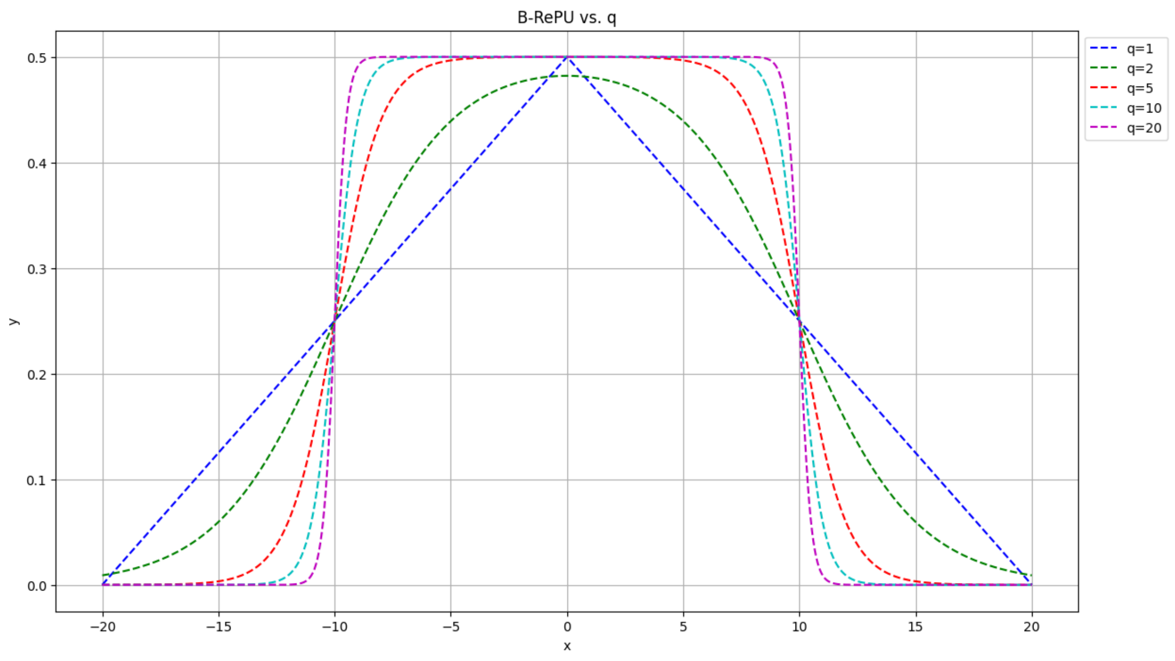


Figure 4.4: nAI of $\mathcal{B}_\theta^{\text{RePU}}$ at various q values

evaluate factorials where q is rounded up. In order for back propagation to work however, $q > 1$ is strictly enforced. Otherwise, the gradient involving RePU_q will be undefined at some points.

This neural network variation will use the same properties as subsection 4.3, except $\mathcal{B}_\theta^{\text{RePU}}$ will replace $\mathcal{B}_\theta^{\text{Tanh}}$, and each q_i will start with an initial value of 2. After training for 500 epochs, this model variation returns the approximation in Figure 4.5. As always, the solid blue line is the target function (4.1), while the dashed line is the approximation.

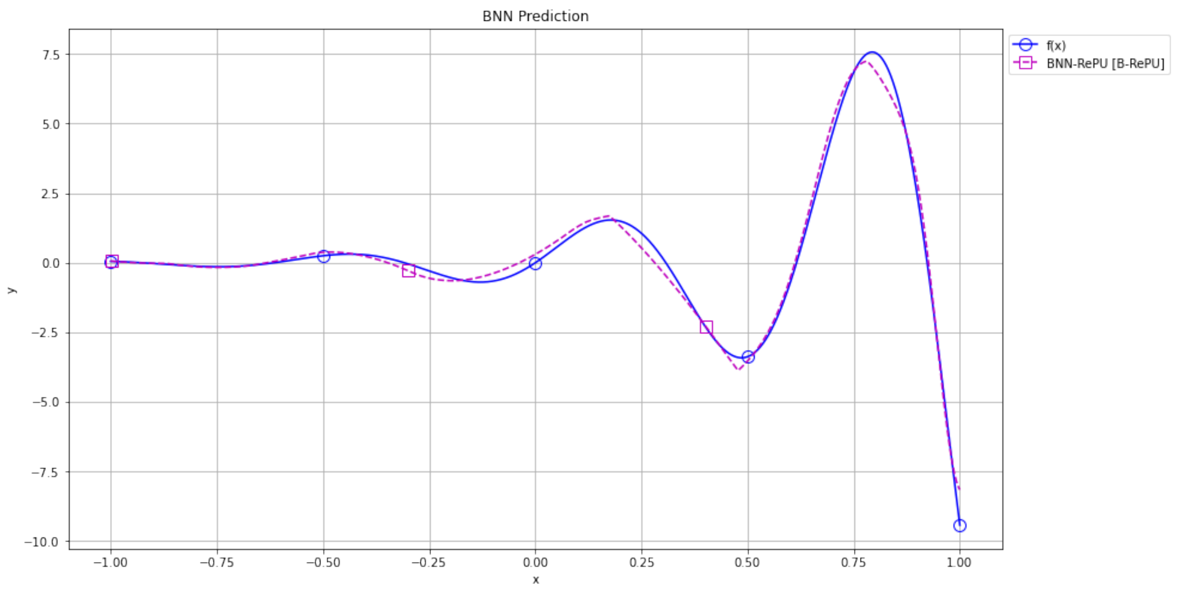


Figure 4.5: BNN Model Predictions (θ's and q's given in Table 4.2)

Table 4.2 displays the resulting θ_i and q_i of each nAI with respect to their quadrature point. Unlike Table 4.1, there is no consistent trend of either value, however the resulting prediction remains quite accurate.

Table 4.2: $\mathcal{B}_{\text{RePU}}$ instances by their quadrature point x_i, spread θ_i, and q_i

i	1	2	3	4	5	6	7	8	9	10
x_i	-1.000	-0.920	-0.739	-0.478	-0.165	0.165	0.478	0.739	0.920	1.000
θ_i	0.0783	0.1578	0.1261	0.3031	0.3573	0.3705	0.3039	0.3270	0.2050	0.1904
q_i	3.991	3.662	3.971	1.000	1.271	1.649	1.843	1.704	1.974	1.692

The MSE of Figure 4.5 is $\underline{0.0687}$. While not as accurate from Figure 4.3, which had a MSE of 0.0588, both models are very accurate, despite only having $N = 10$ nodes.

5. Comparison Results.

In this section, we evaluate the results of our experiments described in section 3. After iterating the models in subsection 3.3 over the problems in subsection 3.2, we obtain the results listed in the following subsections. Note that the $\mathcal{B}$-RePU activation function denotes the nAI where RePU has $q = 2$.

5.1. Fixed θs.

Given the more static structure of Fixed θs, the accuracy of the model is very dependent upon the nAI and θ value present when training. The model is able to train very quickly, however the accuracy of various activation functions may be misrepresented, and will possibly differ from the results in the following sections.

Table 5.1: Table documenting the average MSE of the Fixed θs architecture

		$\mathcal{B}$-ReLU	$\mathcal{B}$-RePU	$\mathcal{B}$-Tanh	$\mathcal{B}$-ELU	$\mathcal{B}$-GELU	$\mathcal{B}$-SiLU	$\mathcal{B}$-Mish
	Polynomials	**0.1078**	0.1207	0.2007	0.2541	0.1203	0.1539	0.1246
MSE	Discontinuous	0.4070	**0.4063**	0.5530	0.6266	0.4267	0.5213	0.4726
	Non-Diff.	**0.1631**	0.1936	0.3424	0.3836	0.1895	0.2601	0.2140

The results when training with Fixed θs is given by Table 5.1, which presents the average Mean Squared Error (MSE) over all corresponding functions. The bold numbers denote the minimum across the category.

Here we find that $\mathcal{B}$-ReLU and $\mathcal{B}$-RePU achieve the lowest MSE, with $\mathcal{B}$-GELU and $\mathcal{B}$-Mish trailing close behind. The remaining nAI struggle to match their results, with $\mathcal{B}$-ELU sometimes having more than twice the minimum MSE. This tells us that, when θ is fixed to $1/N$, $\mathcal{B}$-ReLU, $\mathcal{B}$-RePU, $\mathcal{B}$-GELU, and $\mathcal{B}$-Mish perform the best.

5.2. Trainable θs.

By giving the network the ability to train θs alongside other parameters, each nAI is able to better shape itself for the current function its approximating. It is for this reason that the results in this section are more significant and accurate to the truth, compared to our previous results in Table 5.1.

Table 5.2: Table documenting the average MSE of the Trainable θs architecture

		$\mathcal{B}$-ReLU	$\mathcal{B}$-RePU	$\mathcal{B}$-Tanh	$\mathcal{B}$-ELU	$\mathcal{B}$-GELU	$\mathcal{B}$-SiLU	$\mathcal{B}$-Mish
	Polynomials	0.0734	0.0814	0.0947	0.2669	0.0878	0.1171	**0.0707**
MSE	Discontinuous	0.3396	0.3214	**0.2864**	0.5643	0.3207	0.3385	0.3453
	Non-Diff.	0.1255	0.1618	0.1597	0.2107	**0.1100**	0.1367	0.1489

The results of this model are given by Table 5.2, which has the same setup as the previous table. In this table, the MSEs are much closer than they were with Fixed θs.

For polynomials, $\mathcal{B}$-ReLU, $\mathcal{B}$-RePU, $\mathcal{B}$-GELU, and $\mathcal{B}$-Mish attained the lowest MSE scores, with $\mathcal{B}$-Tanh and $\mathcal{B}$-SiLU trailing close behind. For Discontinuous, there is very little difference between each nAI, except for $\mathcal{B}$-ELU which has an MSE nearly double that of $\mathcal{B}$-Tanh. The non-differentiable results are similar to the polynomial errors, where $\mathcal{B}$-ReLU, $\mathcal{B}$-GELU, and $\mathcal{B}$-Mish have low MSE scores.

5.3. Quadrature Based. In section 2, we discussed how nAI are used with quadrature to prove how traditional activation functions are universal. The QBNN is designed to make use of this principle by fixing some parameters in order to simulate the quadrature approximation. The construction and details of this network are further discussed in subsection 4.3.

Table 5.3: Table documenting the average MSE of the Quadrature Based architecture

		$\mathcal{B}$-ReLU	$\mathcal{B}$-RePU	$\mathcal{B}$-Tanh	$\mathcal{B}$-ELU	$\mathcal{B}$-GELU	$\mathcal{B}$-SiLU	$\mathcal{B}$-Mish
	Polynomials	0.1627	0.1448	0.1101	0.5604	0.1004	**0.0844**	0.1175
MSE	Discontinuous	0.3919	0.3701	0.3695	0.9599	**0.2882**	0.3364	0.3410
	Non-Diff.	0.2280	0.2232	0.2297	0.8342	**0.1409**	0.1445	0.1738

The MSE from QBNN approximations are listed in Table 5.3. Here, we find that $\mathcal{B}$-GELU and $\mathcal{B}$-SiLU consistently perform the best, while $\mathcal{B}$-Tanh and $\mathcal{B}$-Mish are close behind. Unlike our previous results, $\mathcal{B}$-ReLU and $\mathcal{B}$-RePU do considerably worse with this architecture. Yet, much like our previous results, $\mathcal{B}$-ELU obtains MSE scores far larger than the rest.

5.4. Classification. Classification differs from regression in many ways. Instead of approximating some output on a continuous space, a classification problem involves a discrete output space. Perhaps the minor differences between activation functions plays a more significant role in classification as it does in regression.

Table 5.4: Table documenting the accuracy on the MNIST data set

	$\mathcal{B}$-ReLU	$\mathcal{B}$-RePU	$\mathcal{B}$-Tanh	$\mathcal{B}$-ELU	$\mathcal{B}$-GELU	$\mathcal{B}$-SiLU	$\mathcal{B}$-Mish
Accuracy	0.9791	0.9799	**0.9832**	0.9759	0.9822	**0.9839**	0.9827
F1	0.9789	0.9798	**0.9831**	0.9757	0.9821	**0.9838**	0.9825

The test accuracy and F1 score of each model is given in Table 5.4. Given the lowest test accuracy is 0.9759, we find that nAI can be applied to classification tasks much like regression. Comparatively, $\mathcal{B}$-Tanh and $\mathcal{B}$-SiLU perform quite well, while $\mathcal{B}$-ReLU, $\mathcal{B}$-RePU, and $\mathcal{B}$-ELU struggle the most.

6. Discussion.

In this work, we explored the concept of nAI in section 2, and how they can be used to approximate functions in section 4. In section 3, we were able to construct an nAI for several traditional activation functions, implement them into shallow ANN architectures, and apply them to a variety of tasks in order to present a fair comparison between these traditional activation functions. We can consider this a fair comparison as each nAI has a similar structure, and is commonly used by ANN to approximate target functions. The final results of these comparisons are displayed in Tables 5.1 to 5.4.

The results given by Tables 5.1 and 5.2 reveal that for shallow ANN, hinge activation functions like ReLU, RePU, GELU, and Mish are most effective at regression problems. By contrast, sigmoidal activation functions like Tanh perform best with classification problems, as found in Table 5.4. Finally, with regards to the Quadrature architecture discussed in section 4 and evaluated with Table 5.3, activation functions like GELU and SiLU achieve the best results.

In addition to comparing activation functions, we explored a modification to the generic neural network that utilizes nAI and quadrature methods to potentially improve upon the model's accuracy, known as QBNN. This concept comes from Bui-Thanh [1], where quadrature using nAI can effectively approximate continuous functions, as described in section 2.

We first confirmed that (2.2) held true for some target function, and observed how accuracy improved as the number of quadrature points increased. This led to the construction of the QBNN in subsection 4.2, which aims to mimic the quadrature approximation via iterative training. The further optimization of spread variables $\{\theta_i\}_{i=1}^N$ and RePU exponent parameters $\{q_i\}_{i=1}^N$ expand upon the QBNN model by enabling it to better predict the solution curve and identify details the current model could not. The use of θ_i enable the identification of both broad, long-spanning trends alongside short, brief changes in the label data. The use of q_i increase the variety of curve shapes the model can utilize when approximating the solution curve.

Acknowledgment. I would like to thank my mentor Prof. Tan Bui-Thanh from the Oden Institute at the University of Texas at Austin.

References.

[1] Tan Bui-Thanh. "A unified and constructive framework for the universality of neural networks". In: *IMA Journal of Applied Mathematics* (Nov. 2023), hxad032. ISSN: 0272-4960. DOI: 10.1093/imamat/hxad032. eprint: https://academic.oup.com/imamat/advance-article-pdf/doi/10.1093/imamat/hxad032/53969898/hxad032.pdf. URL: https://doi.org/10.1093/imamat/hxad032.

[2] Je-Chian Chen and Yu-Min Wang. "Comparing Activation Functions in Modeling Shoreline Variation Using Multilayer Perceptron Neural Network". In: *Water* 12.5 (2020). ISSN: 2073-4441. DOI: 10.3390/w12051281. URL: https://www.mdpi.com/2073-4441/12/5/1281.

[3] Steffen Eger, Paul Youssef, and Iryna Gurevych. *Is it Time to Swish? Comparing Deep Learning Activation Functions Across NLP tasks*. 2019. arXiv: 1901.02671 [cs.CL].

[4] Bekir Karlik and A Vehbi Olgac. "Performance analysis of various activation functions in generalized MLP architectures of neural networks". In: *International Journal of Artificial Intelligence and Expert Systems* 1.4 (2011), pp. 111–122.

[5] John Pomerat, Aviv Segev, and Rituparna Datta. "On Neural Network Activation Functions and Optimizers in Relation to Polynomial Regression". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 6183–6185. DOI: 10.1109/BigData47090.2019.9005674.