

---

# BIO-INSPIRED ADAPTIVE NEURONS FOR DYNAMIC WEIGHTING IN ARTIFICIAL NEURAL NETWORKS \*

---

**Ashhadul Islam**

Hamad Bin Khalifa University  
Doha, Qatar  
[asislam@hbku.edu.qa](mailto:asislam@hbku.edu.qa)

**Abdesselam Bouzerdoum**

University of Wollongong, Australia  
Hamad Bin Khalifa University, Doha, Qatar  
[a\\_bouzerdoum@uow.edu.au](mailto:a_bouzerdoum@uow.edu.au), [abouzerdoum@hbku.edu.qa](mailto:abouzerdoum@hbku.edu.qa)

**Samir Brahim Belhaouari\***

Hamad Bin Khalifa University  
Doha, Qatar  
[sbelhaouari@hbku.edu.qa](mailto:sbelhaouari@hbku.edu.qa)

## ABSTRACT

Traditional neural networks employ fixed weights during inference, limiting their ability to adapt to changing input conditions, unlike biological neurons that adjust signal strength dynamically based on stimuli. This discrepancy between artificial and biological neurons constrains neural network flexibility and adaptability. To bridge this gap, we propose a novel framework for adaptive neural networks, where neuron weights are modeled as functions of the input signal, allowing the network to adjust dynamically in real-time. Importantly, we achieve this within the same traditional architecture of an Artificial Neural Network, maintaining structural familiarity while introducing dynamic adaptability. In our research, we apply Chebyshev polynomials as one of the many possible decomposition methods to achieve this adaptive weighting mechanism, with polynomial coefficients learned during training. Out of the 145 datasets tested, our adaptive Chebyshev neural network demonstrated a marked improvement over an equivalent MLP in approximately 8% of cases, performing strictly better on 121 datasets. In the remaining 24 datasets, the performance of our algorithm matched that of the MLP, highlighting its ability to generalize standard neural network behavior while offering enhanced adaptability. As a generalized form of the MLP, this model seamlessly retains MLP performance where needed while extending its capabilities to achieve superior accuracy across a wide range of complex tasks. These results underscore the potential of adaptive neurons to enhance generalization, flexibility, and robustness in neural networks, particularly in applications with dynamic or non-linear data dependencies.

**Keywords** Adaptive neural networks · Dynamic weighting · Input-dependent weights · Neural network generalization · Chebyshev polynomials

## 1 Introduction

In conventional neural networks [3], the weights associated with each neuron remain constant during inference, meaning that the output is determined by a fixed set of weights that were learned during training. While this method has been successful in numerous applications, it does not mirror the adaptive behavior observed in biological neurons. In biological systems, neurons adjust the strength of the signals they transmit in response to the intensity of the input stimuli, leading to dynamic weighting [39, 53, 8]. This discrepancy between artificial neurons and their biological counterparts may limit the flexibility and adaptability of current neural networks, particularly in environments with variable input conditions.

---

\* *Citation: Authors. Title. Pages.... DOI:000000/11111.*

\* Corresponding author: [sbelhaouari@hbku.edu.qa](mailto:sbelhaouari@hbku.edu.qa)

To address this, we propose a bio-inspired approach that incorporates adaptive neurons using Chebyshev polynomials [30]. In this framework, instead of having fixed weights, the weight of each connection dynamically adjusts based on the input signal. Specifically, the weight is expressed as a sum of Chebyshev polynomials, where the coefficients of the polynomials are learnable parameters optimized during training.

Let  $x_i$  represent the input to the neuron and  $w_i$  represent the adaptive weight associated with the input  $x_i$ . The adaptive weight is defined as:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (1)$$

where  $c_{i,j}$  are the learnable coefficients,  $T_j(x_i)$  represents the Chebyshev polynomial of the first kind of order  $j$  evaluated at  $x_i$ , and  $k$  is the polynomial order. The neuron's output is then computed as:

$$\text{Neuron Output: } y = \sum_{i=1}^n x_i \cdot w_i(x_i) = \sum_{i=1}^n x_i \cdot \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (2)$$

By making the weights dependent on the input, the neuron can dynamically adjust its output, allowing it to better mimic the adaptive behavior seen in biological neurons. This results in a more flexible and biologically plausible artificial neuron model.

This approach introduces several key innovations:

1. Dynamic weighting, enabling neuron output to reflect the input strength in a biologically plausible manner
2. Mathematical flexibility provided by Chebyshev polynomials, enabling nuanced and complex weight dynamics
3. Enhanced generalization in tasks with complex or non-linear input-output relationships.

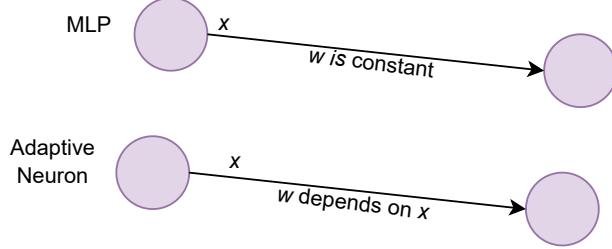


Figure 1: Illustration of fixed versus adaptive weights in neuron connections. The top connection represents a conventional neuron with static weights post-training, unaffected by input variations. The lower connection demonstrates an adaptive neuron, where the weight is a dynamic function of the input data, modeled through different decomposition functions.

This concept is illustrated in Figure 1. The top connection in the figure represents a conventional neuron, where the weights are fixed after training and do not vary with the input. This standard approach limits the neuron's adaptability to different input conditions. In contrast, the lower connection demonstrates an adaptive neuron, where each weight dynamically changes as a function of the input data. Using decomposition function like Chebyshev polynomials to model the weight as a sum of polynomial terms, this adaptive mechanism allows the neuron to adjust its response to varying input levels, making it more flexible and responsive. By incorporating adaptive weights, this framework brings neural networks closer to the dynamic behaviors observed in biological neurons.

Our method bridges the gap between biological and artificial neurons, improving adaptability and robustness in neural networks, with potential applications in areas like bio-inspired computing, brain-computer interfaces, and pattern recognition. This research paves the way for new architectures that incorporate dynamic weight adjustment mechanisms, advancing the field of neural network design.

## 1.1 Biological Basis of Neurons and Their Inspiration for Artificial Neural Networks

### 1.1.1 Human Neuron Structure and Function

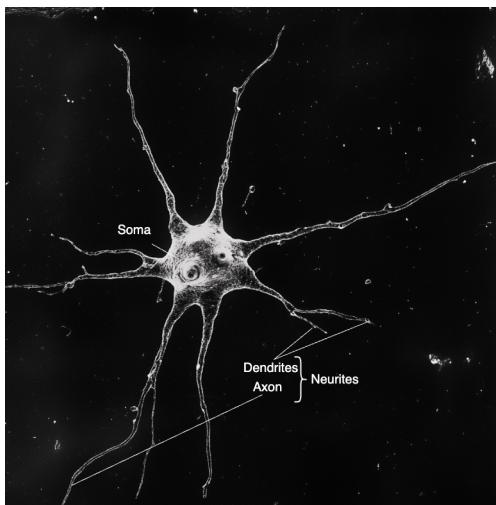


Figure 2: Structure of a human neuron, depicting the soma (cell body), dendrites, and axon, which are essential for transmitting neural signals.

Human neurons are specialized cells responsible for transmitting signals within the brain and nervous system, enabling cognition, sensation, and motor control. As shown in Figure 2, a neuron consists of three primary components: the soma, dendrites, and axon [24]. The soma, or cell body, contains the nucleus and is responsible for maintaining the neuron's structure and performing essential metabolic processes [1]. Dendrites are branch-like structures that receive chemical signals (neurotransmitters) from other neurons and convert these chemical signals into electrical impulses that travel toward the soma [44]. The axon is a long projection that transmits these electrical signals away from the soma, culminating at the synapse, where neurotransmitters are released to communicate with other neurons [23].

### 1.1.2 Neuron Communication and Nonlinearity

Neurons communicate via electrical and chemical signals, enabling rapid and complex information processing across the brain [2]. When a neuron is activated, it generates an action potential—an electrical impulse initiated when the neuron's membrane depolarizes due to the influx of sodium ions ( $\text{Na}^+$ ), allowing the neuron to reach a critical threshold [21]. This action potential travels along the axon and is converted into a chemical signal at the synapse, where neurotransmitters are released and bind to receptors on the dendrites of a postsynaptic neuron, generating an electrical response in the receiving neuron [24].

The response of neurons to input signals is inherently nonlinear. Neurons exhibit threshold behavior, where they fire action potentials only when the input surpasses a certain threshold [26]. Below this threshold, the neuron remains inactive, regardless of slight variations in input. Furthermore, neurons integrate signals over time through temporal summation, where multiple inputs arriving within a short time window can summate, potentially pushing the neuron past its threshold. Spatial summation also occurs, where inputs from different locations on the dendrites have varying effects on the soma, depending on their proximity and strength [4]. Another important factor contributing to nonlinearity is synaptic plasticity, where the strength of synaptic connections changes based on prior activity, thus modifying how neurons communicate over time.

### 1.1.3 Inspiration for Artificial Neural Networks

Artificial neural networks (ANNs) are inspired by biological neurons but simplify their functions to make them computationally efficient [43]. While artificial neurons process input via linear combinations of weights and biases, biological neurons exhibit more dynamic and nonlinear behaviors. Understanding these nonlinear properties in biological neurons offers valuable insights for improving ANN design [27].

In biological neurons, synaptic plasticity allows connections to strengthen or weaken over time, adapting to stimuli dynamically [4]. Similarly, ANNs adjust synaptic weights during training to minimize error, but without the temporal

and spatial complexity of biological learning. While activation functions in ANNs (such as ReLU or sigmoid) introduce nonlinearity, they are a simplified abstraction of the more complex processes that govern biological neuron responses [14].

Recent advancements such as spiking neural networks (SNNs), which more closely model biological neurons by simulating spike-timing dynamics, introduce temporal nonlinearity into the network [34]. Hebbian learning, based on the principle that "cells that fire together, wire together," is another biologically inspired mechanism that could improve how ANNs adapt to changing environments [20]. Models such as recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks are used to capture temporal dependencies, mimicking the recurrent connections in biological circuits [22]. These architectures bring ANNs closer to the dynamic, feedback-rich environment of biological neurons.

#### 1.1.4 Relevance to Adaptive Neural Networks

The nonlinear, dynamic behavior of biological neurons, particularly through mechanisms like synaptic plasticity and action potentials, has inspired our approach to adaptive neurons. Biological neurons adjust their responses based on the input signal's intensity and timing, a process we aim to emulate in artificial neurons using Chebyshev polynomials. By incorporating input-dependent adaptive weights, we propose a model that mirrors the biological neuron's ability to integrate complex signals nonlinearly. This provides the flexibility needed to capture intricate patterns in data, much like how biological systems process information.

### 1.2 Orthogonal Decomposition Techniques and Their Application to Adaptive Neurons

In various mathematical and computational problems, approximating a function  $f(x)$  using orthogonal basis functions is a widely used method for capturing the complexity of a function while ensuring computational efficiency. This technique, known as **orthogonal decomposition** [7], involves representing a function as a sum of orthogonal basis functions  $\{\phi_n(x)\}$ , where each term in the series contributes uniquely to different features of the function [35]. The general form of the approximation is given by:

$$f_N(x) = \sum_{n=0}^N c_n \phi_n(x)$$

where  $\phi_n(x)$  are orthogonal basis functions, and  $c_n$  are coefficients determined by projecting  $f(x)$  onto the basis functions.  $N$  represents the degree of the approximation or the highest order of the orthogonal basis functions used in the summation

### 1.3 Different Mathematical Formulations for Orthogonal Decomposition

Several families of orthogonal functions have been developed to handle specific kinds of problems. Each of these families has its own properties and is suited to different types of function approximation tasks. A few notable examples include:

- **Legendre Polynomial Approximation:** For functions defined on finite intervals, such as  $[-1, 1]$ , Legendre polynomials are commonly used as the orthogonal basis [32]. The approximation takes the form:

$$f_N(x) = \sum_{n=0}^N c_n P_n(x)$$

where  $P_n(x)$  are the Legendre polynomials. The coefficients  $c_n$  are calculated by projecting  $f(x)$  onto the Legendre polynomials:

$$c_n = \frac{\int_{-1}^1 f(x) P_n(x) dx}{\int_{-1}^1 P_n(x)^2 dx}$$

These polynomials are orthogonal over the interval  $[-1, 1]$ , and their orthogonality condition is given by:

$$\int_{-1}^1 P_m(x) P_n(x) dx = 0 \quad \text{if } m \neq n$$

- **Chebyshev Polynomial Approximation:** Chebyshev polynomials are particularly useful when minimizing errors and oscillations in function approximation, especially for smooth functions [30]. The approximation is given by:

$$f_N(x) = \sum_{n=0}^N c_n T_n(x)$$

where  $T_n(x)$  are the Chebyshev polynomials of the first kind. The coefficients  $c_n$  are calculated using:

$$c_n = \frac{2}{\pi} \int_{-1}^1 \frac{f(x)T_n(x)}{\sqrt{1-x^2}} dx \quad \text{for } n \geq 1$$

The orthogonality condition for Chebyshev polynomials is:

$$\int_{-1}^1 \frac{T_m(x)T_n(x)}{\sqrt{1-x^2}} dx = 0 \quad \text{if } m \neq n$$

These polynomials are ideal for approximating functions that are defined over the interval  $[-1, 1]$  and need to minimize the maximum deviation from the true function.

- **Wavelet Decomposition:** In situations where both time and frequency localization are important, wavelets provide an orthogonal decomposition [46, 45]. A function  $f(x)$  is approximated using wavelet basis functions  $\psi_n(x)$ :

$$f_N(x) = \sum_{n=0}^N c_n \psi_n(x)$$

Here,  $\psi_n(x)$  are wavelet functions, and  $c_n$  are coefficients found by projecting  $f(x)$  onto these wavelets:

$$c_n = \int_{-\infty}^{\infty} f(x)\psi_n(x)dx$$

- **Eigenfunction Decomposition:** Eigenfunction decomposition is a method used to express a function in terms of the eigenfunctions of a linear operator  $L$  [12, 48]. The function  $f(x)$  is decomposed as:

$$f(x) = \sum_{n=0}^N c_n \phi_n(x)$$

where  $\phi_n(x)$  are the eigenfunctions of  $L$ , and  $c_n$  are coefficients found by projecting  $f(x)$  onto the eigenfunctions. This technique is particularly useful in solving problems governed by differential or integral operators.

#### 1.4 Adaptive Weighting through Chebyshev Polynomial Decomposition

In the context of our proposed adaptive neuron model, we leverage **Chebyshev polynomials** for orthogonal decomposition of the neuron weights. By representing the weight as a sum of Chebyshev polynomials, we introduce dynamic, input-dependent behavior that enables the neuron to adjust its output based on the input signal. The adaptive weight is expressed as:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i)$$

This method allows the network to dynamically adjust its weights based on the input, leading to better flexibility and generalization, particularly in tasks where the input-output relationship is non-linear or complex.

By incorporating orthogonal decomposition into the design of the neuron model, we ensure that the learned weights are mathematically well-behaved and capable of capturing diverse patterns in the input data. Chebyshev polynomials, with their orthogonality and efficient approximation properties, play a central role in making this approach both computationally feasible and effective.

## 1.5 Properties of Chebyshev Polynomials

Chebyshev polynomials are a sequence of orthogonal polynomials that arise in various fields of applied mathematics, including approximation theory and numerical analysis. They are particularly useful in scenarios that involve minimizing the maximum error between a function and its polynomial approximation, making them well-suited for tasks requiring efficient and accurate representations of complex functions. The key properties of Chebyshev polynomials, particularly those of the first kind  $T_j(x)$ , which are used in our adaptive neuron model [30], include the following:

### 1.5.1 Recursive Definition

Chebyshev polynomials of the first kind, denoted  $T_j(x)$ , can be defined recursively. The first two polynomials in the sequence are:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \end{aligned}$$

For  $j \geq 2$ , the polynomials follow the recursive relation:

$$T_j(x) = 2x \cdot T_{j-1}(x) - T_{j-2}(x)$$

This recursive definition [30] allows for efficient computation of higher-order polynomials and facilitates their integration into the adaptive neuron framework without significantly increasing computational overhead.

### 1.5.2 Orthogonality

One of the key features of Chebyshev polynomials is their orthogonality with respect to the weight function  $\frac{1}{\sqrt{1-x^2}}$  over the interval  $[-1, 1]$  [30]. Specifically, for  $j \neq m$ :

$$\int_{-1}^1 T_j(x) T_m(x) \frac{dx}{\sqrt{1-x^2}} = 0$$

Orthogonality ensures that the polynomials represent independent components of the input, which can lead to better representation and learning of complex, high-dimensional input data in neural networks. In the context of our model, the orthogonality property aids in reducing redundancy in the neuron's response to varying inputs [30].

### 1.5.3 Extremal Properties

Chebyshev polynomials are known for minimizing the maximum deviation from zero over the interval  $[-1, 1]$ . This extremal property makes them particularly effective in approximating functions with high accuracy while keeping the coefficients well-behaved. This feature is highly desirable in neural networks, as it can help prevent overfitting by providing smooth, well-distributed weights across inputs [30, 49, 38].

### 1.5.4 Equioscillation and Roots

The roots of the Chebyshev polynomials of the first kind,  $T_j(x)$ , are located at:

$$x_k = \cos\left(\frac{(2k-1)\pi}{2j}\right) \quad \text{for } k = 1, 2, \dots, j$$

These roots are distributed symmetrically in the interval  $[-1, 1]$  and are used in approximation theory to achieve optimal interpolation. In our adaptive neuron model, the roots of the Chebyshev polynomials provide a natural way to discretize and sample the input space, allowing for dynamic adjustment of weights based on input intensities [30, 49, 38].

### 1.5.5 Rapid Growth Outside of $[-1, 1]$

While Chebyshev polynomials are well-behaved within the interval  $[-1, 1]$ , their values grow rapidly for inputs outside this interval. This behavior can be controlled within our model by normalizing the inputs or using a transformation that ensures the input remains in the range where the polynomials are stable. This is critical in ensuring that the adaptive

weights do not become overly sensitive to out-of-range inputs, thereby preserving the stability and robustness of the neuron's output [7, 30, 49, 38].

### 1.5.6 Approximation Power

Chebyshev polynomials are widely recognized for their superior approximation properties, especially when compared to other polynomials such as Legendre or Laguerre. The series expansion in terms of Chebyshev polynomials is often used in Chebyshev approximation to achieve a near-optimal polynomial approximation of continuous functions. In our model, this approximation power translates into the ability to represent complex, non-linear relationships between the input and the neuron's response, enabling more nuanced and flexible learning [30, 49, 38].

## 1.6 Chebyshev Polynomials in Adaptive Neural Networks

The properties of Chebyshev polynomials make them an excellent choice for adaptive neural networks. Their orthogonality, extremal properties, and efficient recursive computation allow neurons to dynamically adjust their weights based on input signals, enhancing the network's ability to generalize and adapt to diverse input conditions. The integration of Chebyshev polynomials in the weight computation, as proposed in this research, leads to several advantages:

- **Dynamic and Adaptive Weighting:** Unlike traditional neurons with fixed weights, neurons in our model can adjust their weights on-the-fly based on the intensity of the input, providing more flexibility and adaptability to changing environments.
- **Improved Generalization:** The orthogonality and approximation power of Chebyshev polynomials help in reducing overfitting and improving generalization to unseen data, especially in tasks with complex input-output relationships.
- **Efficient Computation:** The recursive formulation of Chebyshev polynomials ensures that the additional computational overhead introduced by dynamic weighting remains manageable, making the approach feasible for real-world applications.

These properties, along with the dynamic nature of the proposed adaptive neuron model, open up new possibilities for designing neural networks that are more flexible, robust, and capable of handling non-linear and complex data patterns effectively.

## 2 Related Work

This section provides an overview of recent developments in neural network architectures that enhance adaptability and efficiency through various approaches. We discuss methods that improve inference efficiency, introduce dynamic weighting, leverage Chebyshev polynomials, and incorporate spiking mechanisms, all contributing to the evolution of neural networks toward more adaptive and biologically plausible models. Additionally, we explore the Kolmogorov–Arnold Network (KAN) as a foundation for developing adaptive neuron models.

### 2.1 Adaptive Neural Networks for Efficient Inference

Recent advancements in adaptive neural networks have focused on enhancing computational efficiency by modifying network evaluation based on the complexity of individual examples. One notable approach introduces a method to selectively activate network components, allowing early exits for instances that are accurately classified within initial layers, thus avoiding full model evaluation [5]. Additionally, this method introduces a mechanism for adaptive network selection, where a lightweight model is chosen for simpler examples, while more complex networks are reserved for challenging cases. By formulating this process as a policy learning task, the approach optimizes layer-wise or model-level selection through weighted binary classification, significantly reducing inference time while maintaining accuracy.

### 2.2 Dynamic Weighting in Neural Networks

Another prominent direction in adaptive neural networks involves dynamically adjusting weights [18] based on input characteristics during inference. This concept is utilized in various forms, such as attention mechanisms, conditionally parameterized convolutions, and deformable convolutions, each catering to specific model adaptation needs.

For instance:

- **CondConv:** Conditionally parameterized convolutions employ customized convolutional filters for each example, thus enhancing network capacity while maintaining efficient inference [52].
- **Dynamic Convolution:** Here, multiple convolutional kernels are aggregated dynamically using input-dependent attention, allowing the model to maintain a low computational footprint while increasing representational flexibility [9].
- **Segmentation-aware CNNs:** These networks use local attention masks that selectively attend to region-specific inputs, improving spatial precision for tasks such as semantic segmentation and optical flow [19].
- **Deformable Convolutional Networks (DCNs):** By adjusting convolutional sampling locations, these networks adapt their receptive fields to object shapes, which improves performance in tasks requiring spatial sensitivity [10].

### 2.3 Chebyshev Polynomials in Neural Networks

The study by Troumbis et al. [50] introduces a neural network architecture that leverages Chebyshev polynomials for effective modeling of complex, non-linear environmental data. This network architecture applies Chebyshev polynomials through a layered feedforward structure, with the network parameters optimized via differential evolution, a population-based optimization algorithm.

The network is composed of four sequential layers, each performing a distinct operation:

- **Layer 1:** Generates linear combinations of the input variables  $x_i$  to form  $L_i$  as follows:

$$L_i = \sum_{j=1}^p a_{i,j} \cdot x_j$$

where  $a_{i,j}$  represents the learned coefficients, and  $p$  denotes the number of input variables.

- **Layer 2:** Scales each  $L_i$  to the interval  $[-1, 1]$  to fit within the domain of Chebyshev polynomials, using the following transformation:

$$\tilde{L}_i = \frac{2(L_i - L_{min})}{L_{max} - L_{min}} - 1$$

where  $L_{min}$  and  $L_{max}$  denote the minimum and maximum values, respectively.

- **Layer 3:** Computes truncated Chebyshev series expansions for each normalized  $\tilde{L}_i$ , represented as:

$$T_i(\tilde{L}_i) = \sum_{n=0}^N c_{i,n} \cdot T_n(\tilde{L}_i)$$

where  $c_{i,n}$  are coefficients specific to each input,  $T_n(\tilde{L}_i)$  represents the Chebyshev polynomial of order  $n$ , and  $N$  is the truncation order.

- **Layer 4:** Linearly combines the truncated Chebyshev series from each node to produce the final network output:

$$y = \sum_{i=1}^c w_i \cdot T_i(\tilde{L}_i)$$

where  $w_i$  represents the weights applied to each truncated series, and  $c$  is the number of hidden nodes.

The training of this Chebyshev polynomial-based network is accomplished using the Differential Evolution (DE) algorithm, which optimizes the network parameters through mutation, crossover, and selection phases. DE is particularly suited for this application because of its ability to handle non-linear optimization problems without relying on gradient-based methods, thus providing robustness against local minima and ensuring convergence to an optimal solution.

In experiments, this Chebyshev polynomial network outperformed other architectures, including networks using Hermite polynomials, radial basis functions, and Takagi-Sugeno-Kang neuro-fuzzy models, across diverse environmental datasets. This performance was assessed using standard metrics such as Root Mean Square Error (RMSE) and Mean Absolute Error (MAE), demonstrating the network's competitive edge in complex, data-intensive tasks.

Our approach diverges from the work of Troumbis et al. [50] by introducing Chebyshev polynomials not as fixed components in a pre-defined network structure, but as adaptable elements that dynamically adjust weights based on input signals. This adaptability provides greater flexibility and allows our network to generalize beyond specific environmental

datasets. Additionally, while Troumbis et al. focus solely on Chebyshev polynomials, our framework is designed to accommodate any decomposition function, using Chebyshev polynomials merely as one possible implementation. This extensibility positions our model as a more versatile, bio-inspired alternative that can tailor its behavior to a broader range of complex tasks.

## 2.4 Spiking Neural Networks and Adaptation

Spiking Neural Networks (SNNs) [16] represent a brain-inspired computational model that utilizes binary spike-based communication, allowing for efficient information processing through event-driven and spatio-temporal mechanisms. These properties make SNNs particularly effective in handling temporal data and enable energy-efficient computations, aligning with biological neural networks' sparse firing behavior. However, the discontinuous nature of spike-based information transmission introduces challenges for training deep SNNs, as conventional gradient-based optimization methods cannot be directly applied due to the non-differentiable spiking mechanism.

One approach to overcome this challenge is *Spike-Timing-Dependent Plasticity* (STDP) [29], a biologically inspired mechanism for weight updates in SNNs. Despite its grounding in biology, STDP alone is not sufficient for training large-scale networks, limiting its applicability in practical, complex tasks.

Two prevalent methods have been developed to achieve more effective training of deep SNNs:

1. **ANN-to-SNN Conversion:** This method involves training an Artificial Neural Network (ANN) with standard continuous activation functions (such as ReLU) and subsequently converting it into an SNN by replacing the activations with spike-based mechanisms [17]. This process retains the original network's learned representations and is straightforward to implement. However, it is typically constrained to rate-coding, ignoring the potential for more dynamic temporal behaviors within SNNs.
2. **Surrogate Gradient (SG) Approach:** The SG approach introduces a differentiable surrogate function that approximates the spiking neuron's non-differentiable firing activity, enabling backpropagation in SNNs [13]. This technique has shown significant promise in handling temporal data and achieving competitive performance with few time steps on large-scale datasets. The SG method allows SNNs to utilize the spatio-temporal dynamics of spikes effectively, thereby enhancing the network's ability to process temporal information while maintaining efficiency.

In recent studies, including Wu et al. [51], various enhancements to direct learning-based SNNs are discussed, categorized broadly into accuracy improvement methods, efficiency improvement methods, and methods that exploit temporal dynamics. These categories are further divided based on specific objectives:

- *Accuracy Improvement:* Focused on increasing representational capacity and alleviating training challenges [51].
- *Efficiency Improvement:* Involves techniques such as network compression and sparse connectivity to reduce computational costs [51].
- *Temporal Dynamics Utilization:* Exploits sequential learning and integration with neuromorphic sensors to harness SNNs' unique temporal characteristics [51].

Adaptive SNNs (AdSNNs) further advance SNNs [54] by incorporating spike frequency adaptation, a phenomenon observed in biological neurons that dynamically modulates firing rates to encode input intensity effectively. The adaptation can be modeled through the dynamic threshold mechanism, where the threshold  $\vartheta(t)$  for spike generation adapts based on previous spiking activity. This adaptation can be expressed as:

$$\vartheta_j(t) = \vartheta_0 + \sum_{t_j} m_f \vartheta_0 \gamma(t - t_j),$$

where  $\vartheta_0$  is the resting threshold,  $m_f$  controls adaptation speed, and  $\gamma(t)$  is an adaptation kernel.

This adaptive mechanism allows for spike-based coding precision to be adjusted according to the input's dynamic range. By modulating firing rates, AdSNNs achieve efficient encoding while remaining energy-conscious, paralleling biological neural behavior. The flexibility in adjusting neural coding precision offers advantages in tasks that require varying levels of attentional focus, making AdSNNs suitable for temporally continuous, asynchronous applications.

## 2.5 Distinctive Features of Our Approach

While each of the methods discussed above enhances neural network adaptability and efficiency through unique mechanisms, our approach introduces a novel adaptive framework by dynamically adjusting neuron weights based on input signals. Unlike the *adaptive neural networks for efficient inference* that rely on selecting pre-trained sub-networks, our model directly modifies weights at the neuron level to achieve real-time adaptability. In contrast to *dynamic weighting in neural networks*, which generally incorporates fixed mechanisms like attention weights or conditionally parameterized convolutions, our model treats weights as functions of the input signal itself, enabling a more fluid response to input variations.

Our use of *Chebyshev polynomials* is distinct from previous methods, such as that of Troumbis et al. [50], as it serves as an example within a broader framework that can integrate various orthogonal functions, not just Chebyshev polynomials. This design allows the network to be configured with any decomposition function, thereby expanding its flexibility across diverse applications.

Compared to *spiking neural networks* (SNNs), which rely on binary spikes and temporal dynamics to mimic biological neurons, our approach achieves bio-inspired adaptability without the need for spike-based communication. Our model continuously adjusts weights as opposed to relying on discrete events, providing a smoother and potentially more computationally efficient pathway for tasks that do not necessitate spike-based signaling.

## 2.6 Inspiration from Kolmogorov–Arnold Networks

Multi-Layer Perceptrons (MLPs) [42, 40, 31], commonly referred to as fully-connected feedforward neural networks, serve as essential components in modern deep learning architectures. MLPs consist of multiple layers of neurons, where each neuron applies a predefined activation function to the weighted sum of its inputs. This architecture enables MLPs to approximate a wide variety of nonlinear functions, a capability supported by the universal approximation theorem. Consequently, MLPs are widely employed in diverse deep learning tasks, including classification, regression, and feature extraction. However, MLPs are not without limitations, such as challenges in interpreting their learned representations and constraints in scaling the network.

Kolmogorov–Arnold Networks (KANs) [28] offer a novel alternative to conventional MLPs by utilizing the Kolmogorov–Arnold representation theorem. In contrast to MLPs, which use fixed activation functions for neurons, KANs introduce learnable activation functions on the edges, replacing linear weights with univariate functions modeled as splines.

The concept of adaptive neurons using Chebyshev polynomials was inspired by the Kolmogorov–Arnold Network (KAN) architecture, particularly its innovative approach of replacing fixed neuron activation functions with learnable functions on the edges. The key difference, however, lies in the representation of these adaptive weights. While KANs employ univariate spline functions to parameterize the weights, the proposed adaptive neurons use Chebyshev polynomials to model dynamic, input-dependent weights. This allows for a richer, more flexible representation of weight dynamics, enabling the neurons to adjust their output based on the input signal, thus offering a bio-inspired approach that more closely mirrors the behavior of biological neurons.

## 3 Proposed Architecture

In this section, we detail the architecture of the proposed adaptive neuron model using Chebyshev polynomials.

### 3.1 Simple Neuron Model

In a traditional neuron within a neural network, the process of generating the output relies on combining the inputs and their associated weights through a linear combination. The neuron takes a set of input values  $x_1, x_2, \dots, x_n$ , where each input  $x_i$  is multiplied by a corresponding weight  $w_i$ . The weight represents the strength of the connection between the input and the neuron, determining how much influence that particular input will have on the final output.

Mathematically, the neuron computes a weighted sum of its inputs, which can be represented as:

$$y = \sum_{i=1}^n w_i \cdot x_i \quad (3)$$

This equation describes a linear combination of the inputs, where each input  $x_i$  contributes to the output proportionally to its corresponding weight  $w_i$ . The output of the neuron is simply the result of this linear combination, which is often passed through an activation function to introduce non-linearity and allow the network to learn more complex patterns.

In this basic model, the behavior of the neuron is limited to a linear combination of the inputs, meaning the output is only capable of representing linear functions of the inputs. To capture more complex relationships and patterns in the data, this output is typically passed through a non-linear activation function, such as the sigmoid or ReLU, which transforms the linear output into a non-linear space, allowing the neural network to model more sophisticated patterns.

This form of linear combination is the foundational operation in artificial neural networks, forming the basis for deeper architectures such as multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), and other complex models.

### 3.2 Decomposition of Weights Using Chebyshev Polynomials

In traditional neural networks, the weights associated with each input to a neuron are fixed after training. However, biological neurons exhibit dynamic behavior where the strength of the connections (or synaptic weights) can adapt in response to stimuli. To better mimic this adaptive nature, we introduce a decomposition of the fixed weights into dynamic, input-dependent weights using Chebyshev polynomials.

Chebyshev polynomials are a family of orthogonal polynomials that provide a mathematically efficient way to approximate complex functions. By leveraging these polynomials, we enable the weights in our model to vary as a function of the input, making them more flexible and capable of capturing intricate, non-linear relationships between inputs and outputs. Specifically, instead of assigning a fixed weight  $w_i$  to an input  $x_i$ , we represent  $w_i$  as a sum of Chebyshev polynomials, with the coefficients of these polynomials being learned during training.

For example, when using Chebyshev polynomials of order 3, the weight  $w_i$  corresponding to input  $x_i$  can be expressed as a sum of Chebyshev polynomials of increasing order. This is given by:

$$w_i(x_i) = \sum_{j=0}^2 c_{i,j} T_j(x_i) \quad (4)$$

Here,  $T_j(x_i)$  represents the Chebyshev polynomial of the first kind of degree  $j$ , evaluated at input  $x_i$ . The coefficients  $c_{i,j}$  are the learnable parameters that are optimized during training. By adjusting these coefficients, the network can adaptively alter the weight  $w_i$  based on the input  $x_i$ .

#### 3.2.1 Chebyshev Polynomials of the First Kind

Chebyshev polynomials of the first kind,  $T_j(x)$ , are defined as:

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_2(x) = 2x^2 - 1, \quad T_3(x) = 4x^3 - 3x, \dots$$

These polynomials have useful properties, such as orthogonality, which allows for efficient representation and approximation of functions. When used to decompose the weights, these polynomials allow the model to dynamically adapt the contribution of each input in a non-linear and flexible manner.

#### 3.2.2 Generalization of the Decomposition

We generalize the above formulation by extending it to all inputs in the network. For a set of inputs  $\{x_1, x_2, \dots, x_n\}$ , the adaptive weight for each input is expressed as:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (5)$$

Thus, the neuron's output, which is a weighted sum of all inputs, is given by:

$$Y = \sum_{i=1}^n x_i w_i(x_i) = \sum_{i=1}^n x_i \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (6)$$

This generalized formulation allows the weights for each input to vary dynamically, and the overall output reflects this flexibility.

### 3.3 Chebyshev Polynomial Representation of Weights

In this architecture, the input signals  $\{x_1, x_2, \dots, x_n\}$  are transformed through Chebyshev polynomials  $T_j(x_i)$ , and the adaptive weights are computed as a weighted sum of these polynomials. Each weight is expressed as:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (7)$$

This representation allows each input weight to be dynamically adjusted based on the current input, thus enabling the network to adapt its responses.

### 3.4 Swapping the Chebyshev Operation

For computational efficiency, the Chebyshev operation can be swapped outside the neuron. This reorganization of the Chebyshev polynomial transformation allows for better modularity in the computation. The general form for the transformation can be written as:

$$T_j(x_i) \quad \text{can be precomputed and stored before weighting operations,} \quad w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (8)$$

### 3.5 Encapsulating the Transformation

To encapsulate the Chebyshev transformation into a reusable function, the model applies the Chebyshev transform to each input before it reaches the neuron. Mathematically, the transformation function can be represented as:

$$\text{Chebyshev Transform}(x_i) = [T_0(x_i), T_1(x_i), \dots, T_k(x_i)] \quad (9)$$

### 3.6 Fully Connected Network with Chebyshev Transformation

Finally, we integrate the Chebyshev transformation into a fully connected network. The network processes inputs  $\{x_1, x_2, \dots, x_n\}$  through the Chebyshev transformation, and the resulting adaptive weights are used for the final classification or regression tasks:

$$Y = \sum_{i=1}^n x_i \left( \sum_{j=0}^k c_{i,j} T_j(x_i) \right) \quad (10)$$

### 3.7 Adaptive Neuron Architecture with Chebyshev Polynomial Transformation

Figure 3 presents the architecture of the proposed adaptive neuron model, which incorporates Chebyshev polynomial transformations. In this design, each input feature  $x_i$  undergoes a Chebyshev decomposition, expanding it into a set of polynomial functions  $T_1(x_i), T_2(x_i), \dots, T_k(x_i)$ . This transformation allows for adaptive weighting of inputs, enhancing the model's flexibility to capture complex patterns.

In the adaptive architecture, the Chebyshev decomposition is applied at the input layer, generating multiple transformed features for each input variable. This decomposition captures non-linear interactions, improving the model's robustness. The polynomial terms are then combined through a set of learned coefficients, allowing the output  $y$  to adapt based on the input structure dynamically.

### 3.8 Optimization of Adaptive Neural Networks

Optimizing adaptive neural networks with decomposition-based weights introduces unique challenges, particularly in managing the number of parameters associated with each decomposition strategy. When using Gaussian decomposi-

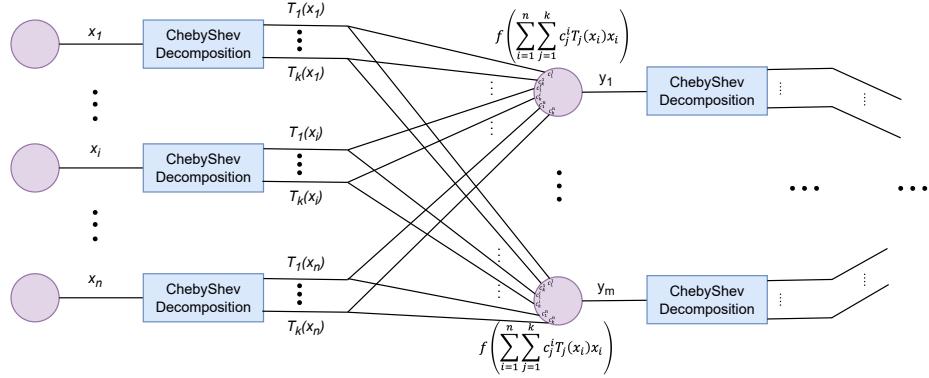


Figure 3: Experiment architecture with Chebyshev polynomial decomposition applied to each input  $x_i$ . This transformation enables the model to adapt its responses dynamically by expanding each input feature through Chebyshev polynomial terms, improving the representation capability of the neuron model.

tion, for example, two parameters—mean and standard deviation—are required for each decomposition component. Conversely, with Chebyshev polynomial decomposition, each term in the polynomial has only a single associated parameter, resulting in fewer parameters to optimize. This simplicity is one reason we favor Chebyshev polynomials in our adaptive neuron model.

The Gaussian decomposition for weight  $w_i(x_i)$  can be represented as:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} \exp\left(-\frac{(x_i - \mu_{i,j})^2}{2\sigma_{i,j}^2}\right) \quad (11)$$

where  $c_{i,j}$  is the weight coefficient,  $\mu_{i,j}$  is the mean, and  $\sigma_{i,j}$  is the standard deviation. This requires the optimization of both  $\mu_{i,j}$  and  $\sigma_{i,j}$  for each term  $j$ , increasing the parameter count and computational load.

For comparison, the Chebyshev decomposition has only one parameter per term:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i) \quad (12)$$

To illustrate the differences, Table 1 provides a generalized comparison of the parameter requirements for different decomposition methods, assuming  $n$  features and polynomial order  $k$ .

Decomposition Type	Parameters per Term	Total Parameters (n features, order k)
Chebyshev Polynomial	$c_{i,j}$	$n \times (k + 1)$
Gaussian	$c_{i,j}, \mu_{i,j}, \sigma_{i,j}$	$n \times 3(k + 1)$
Fourier	$a_{i,j}, b_{i,j}$	$n \times 2(k + 1)$
Legendre Polynomial	$c_{i,j}$	$n \times (k + 1)$

Table 1: Parameter requirements for different decomposition methods used in adaptive neural networks, with  $n$  as the number of features and  $k$  as the decomposition order.

### 3.8.1 Optimization of an Adaptive Neuron in Chebyshev Network

In our adaptive Chebyshev neural network, each weight  $w_i$  is expressed as a function of Chebyshev coefficients, allowing dynamic adjustments based on the input. This decomposition provides the network with the flexibility to capture complex, non-linear relationships. Specifically, each weight  $w_i$  is represented as:

$$w_i(x_i) = \sum_{j=0}^k c_{i,j} T_j(x_i)$$

where  $c_{i,j}$  represents the Chebyshev coefficient for the  $j$ -th term,  $T_j(x_i)$  denotes the Chebyshev polynomial of order  $j$  evaluated at the input  $x_i$ , and  $k$  is the highest polynomial order used. This decomposition makes each weight a flexible, input-dependent function, enhancing the model's adaptability.

To optimize these coefficients, we calculate the gradient of the loss  $L$  with respect to each Chebyshev coefficient  $c_{i,j}$ , denoted as  $\Delta L_c$ . This gradient calculation is structured in terms of two components:  $\Delta L_w$  and  $\Delta W_i$ .

1. Gradient of the Loss with respect to each Weight: The vector  $\Delta L_w$  represents the partial derivatives of the loss  $L$  with respect to each weight  $w_i$ :

$$\Delta L_w = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \end{pmatrix}$$

Here,  $\frac{\delta L}{\delta w_i}$  denotes how sensitive the loss function  $L$  is to changes in the weight  $w_i$ . This term captures the impact of each weight on the overall loss.

2. Gradient of each Weight with respect to Chebyshev Coefficients: For each weight  $w_i$ , the vector  $\Delta W_i$  represents the partial derivatives of  $w_i$  with respect to each Chebyshev coefficient  $c_{i,j}$ :

$$\Delta W_i = \begin{pmatrix} \frac{\delta w_i}{\delta c_{0,i}} \\ \frac{\delta w_i}{\delta c_{1,i}} \\ \vdots \\ \frac{\delta w_i}{\delta c_{k,i}} \end{pmatrix}$$

Here,  $\frac{\delta w_i}{\delta c_{j,i}}$  represents how sensitive the weight  $w_i$  is to changes in the  $j$ -th Chebyshev coefficient  $c_{i,j}$ . This term quantifies how each coefficient contributes to the value of the weight  $w_i$ .

Each term in  $\Delta L_c$  is then computed as the product of the corresponding elements from  $\Delta L_w$  and  $\Delta W_i$ , as shown below.

Thus,  $\Delta L_c$  can be expressed as:

$$\Delta L_c = \begin{pmatrix} \frac{\delta L}{\delta w_1} \cdot \Delta W_1 \\ \frac{\delta L}{\delta w_2} \cdot \Delta W_2 \\ \vdots \\ \frac{\delta L}{\delta w_n} \cdot \Delta W_n \end{pmatrix} = \begin{pmatrix} \frac{\delta L}{\delta w_1} \cdot \frac{\delta w_1}{\delta c_{0,1}} \\ \frac{\delta L}{\delta w_1} \cdot \frac{\delta w_1}{\delta c_{1,1}} \\ \vdots \\ \frac{\delta L}{\delta w_1} \cdot \frac{\delta w_1}{\delta c_{k,1}} \\ \frac{\delta L}{\delta w_2} \cdot \frac{\delta w_2}{\delta c_{0,2}} \\ \vdots \\ \frac{\delta L}{\delta w_2} \cdot \frac{\delta w_2}{\delta c_{k,2}} \\ \vdots \\ \frac{\delta L}{\delta w_n} \cdot \frac{\delta w_n}{\delta c_{0,n}} \\ \vdots \\ \frac{\delta L}{\delta w_n} \cdot \frac{\delta w_n}{\delta c_{k,n}} \end{pmatrix}_{n(k+1) \times 1}$$

This formulation allows us to compute  $\Delta L_c$  as a column vector of size  $n(k + 1)$  by 1. The resulting gradient vector  $\Delta L_c$  can be utilized in standard backpropagation, enabling efficient training of the adaptive Chebyshev neural network.

By decomposing weights in terms of Chebyshev polynomials, this method offers a flexible and computationally efficient approach, reducing the number of parameters while enhancing the model's adaptability and capacity to represent complex, non-linear relationships.

### 3.8.2 Preprocessing with Chebyshev Transformation

In our model, the Chebyshev decomposition can be treated as a preprocessing step before each layer, which simplifies the backpropagation process. This approach enables us to leverage standard backpropagation techniques, as the Chebyshev transformation is applied to the input matrix before it is passed through the neuron layer. Figure 4 shows how this transformation can be applied as a preprocessing step in a single operational unit, while Figure 5 depicts the transformation at a layer-wise scale in an MLP architecture.

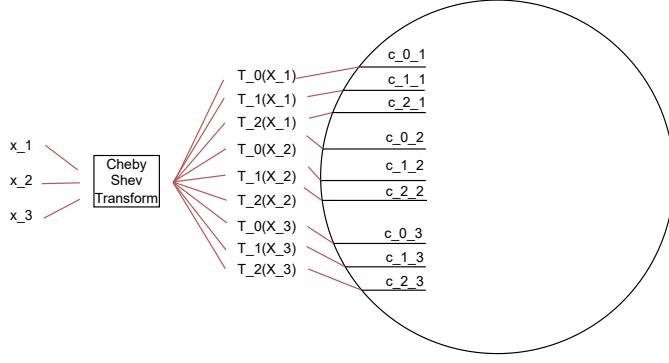


Figure 4: Illustration of a single operation unit where the Chebyshev transformation is applied as a preprocessing step to the input, preserving gradients and enabling efficient backpropagation.

### 3.9 Optimization Techniques for Adaptive Neural Networks

Optimizing adaptive neural networks requires efficient techniques to handle the complexity introduced by decomposition-based adaptive weights. Various optimization methods have been proposed to enhance the convergence and stability of neural networks. **Gradient Descent** [15] is a fundamental approach, adjusting weights by taking steps proportional to the negative gradient of the loss function. **Stochastic Gradient Descent (SGD)** [6] updates weights after each training example, speeding up convergence but increasing update variance. **Mini-Batch Gradient Descent** [41] balances batch gradient descent and SGD by updating parameters after each mini-batch, improving convergence speed and reducing variance. Advanced techniques, such as **Momentum** [36], add a momentum term to reduce oscillations

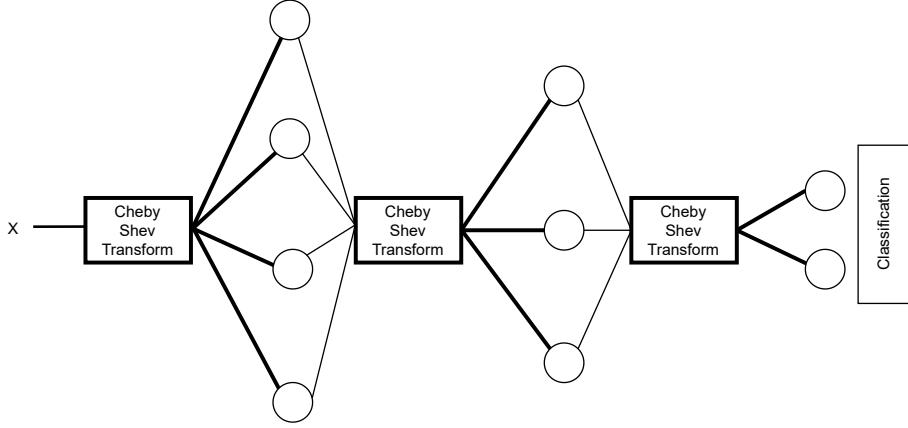


Figure 5: Full architecture demonstrating layer-wise preprocessing with Chebyshev transformation in an MLP model, facilitating standard backpropagation.

and accelerate convergence. **Nesterov Accelerated Gradient (NAG)** [47] builds upon Momentum by calculating gradients at a future point, enhancing convergence smoothness. **Adagrad** [11] adapts learning rates individually for each parameter, though it suffers from decaying learning rates, which **AdaDelta** [55] addresses by accumulating only recent gradients. **Adam (Adaptive Moment Estimation)** [25] combines aspects of both Momentum and Adagrad, maintaining exponentially decaying averages of past gradients and squared gradients, which improves convergence and robustness to hyperparameter tuning [37]. In our experiments, we employ the Adam optimizer, given its adaptability and efficiency, which are essential for training complex adaptive neural networks like our Chebyshev-based model.

## 4 Data And Experiment

### 4.1 Datasets used

For our experiments, we used 145 datasets from the Penn Machine Learning Benchmarks (PMLB) [33], a curated collection of benchmark datasets designed for evaluating and comparing supervised machine learning algorithms. These datasets cover a wide range of applications, including binary and multi-class classification, and they feature combinations of categorical, ordinal, and continuous features. For this study, we focused exclusively on the tabular classification datasets.

To provide an overview of the dataset characteristics, we generated visualizations that depict the distribution of various dataset properties, such as the number of rows, columns, and classes. These charts allow us to gain insights into the diversity and scale of the datasets used, helping us to understand the environments in which our proposed adaptive neuron model will be evaluated.

Figure 6a illustrates the distribution of datasets by the number of rows, showing the frequency of datasets across different dataset sizes. This variation in dataset length provides a robust testing ground, especially for evaluating model scalability with larger datasets. Figure 6b shows the distribution by the number of columns, representing the diversity in feature space dimensionality. This variety tests the model's capacity to handle datasets with different levels of complexity. Figure 6c depicts the distribution by the number of classes, covering both binary and multi-class classification tasks. This diversity in target classes enables assessment of the model's performance across varying classification complexities. The datasets used in our experiments exhibit considerable diversity, making them well-suited for a comprehensive evaluation of the proposed adaptive neuron model. The range in dataset sizes is quite broad, with the largest dataset containing 105,908 rows and the smallest consisting of only 42 rows. Feature counts also vary significantly, spanning from 2 to 1000 features, while the number of target classes ranges from 2 to 26. This variety in dataset properties ensures that our model is tested across a wide spectrum of scenarios, from low-dimensional to high-dimensional feature spaces and from binary to complex multi-class classification tasks. Such diversity strengthens the evaluation, highlighting the model's adaptability and robustness across diverse applications.

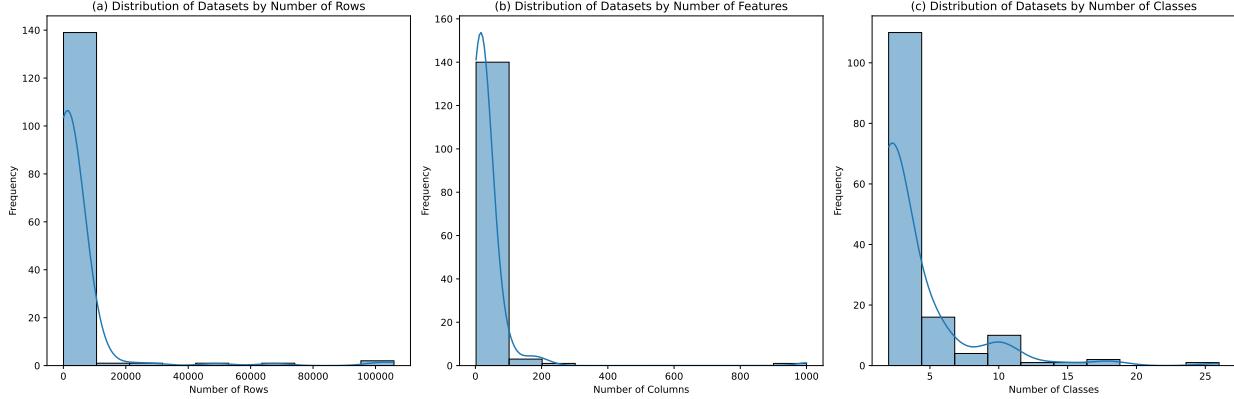


Figure 6: Overview of PMLB dataset distributions. (a) Distribution of datasets by the number of rows, highlighting the variability in dataset size, with most datasets containing a moderate number of rows suitable for robust model evaluation. (b) Distribution of datasets by the number of features, showcasing the range of dimensionalities that test the model's adaptability to various feature spaces. (c) Distribution of datasets by the number of classes, illustrating the diversity in classification tasks from binary to multi-class.

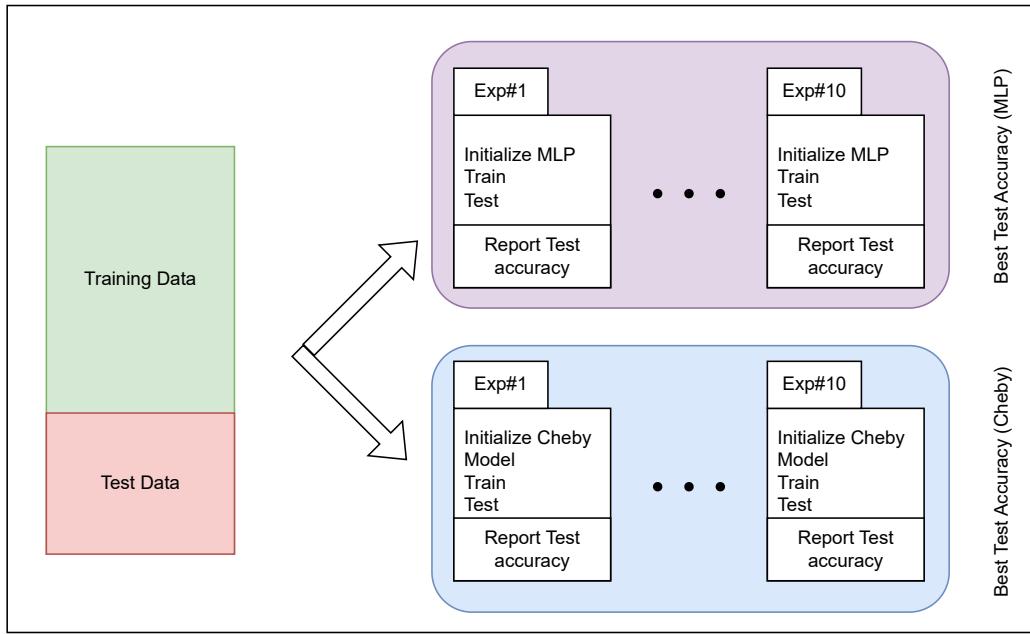


Figure 7: Workflow of the experiment showing the repeated training and evaluation steps for the MLP and Chebyshev models.

## 4.2 Experiment Framework

The experiment evaluates the performance of a Multi-Layer Perceptron (MLP) and a Chebyshev-enhanced model across various datasets to compare their classification accuracies. We utilize a consistent model architecture for each dataset to ensure comparability.

For each dataset, we split the data into training and testing sets, followed by the training of both the MLP and Chebyshev models. Each model is independently trained 10 times on the same data split, and the best accuracy on the test set for each model type.

### 4.2.1 Model Architectures

Both the MLP and Chebyshev-enhanced models share a consistent three-layer architecture for comparability. Each model has an initial layer with 4 neurons, followed by a second layer of 2 neurons, and an output layer where the number of neurons aligns with the number of classes in each dataset. The Chebyshev model applies a Chebyshev transformation to the input features before each layer, enhancing feature representation according to the Chebyshev order, but otherwise maintains the same structure as the MLP. The Chebyshev order, unless otherwise specified is set at 3.

### 4.2.2 Training Parameters

Both models are trained using consistent hyperparameters to ensure a fair comparison. A learning rate of 0.001 is applied, with the ReLU activation function guiding non-linear transformations within each layer. Optimization is handled by the Adam optimizer, chosen for its adaptive learning capabilities, and each model undergoes training over 500 epochs to allow sufficient convergence on the dataset patterns.

## 4.3 Pruning to Leverage Higher-Order Chebyshev Terms

We also explored the use of higher-order Chebyshev terms for model training, followed by parameter pruning to reduce model size while maintaining or even improving performance. We initially trained models with higher-order Chebyshev terms, up to degree 6, then applied pruning strategies to optimize model efficiency. Notably, the pruned models often achieved higher accuracy than the unpruned versions, as reported in the results section.

### 4.3.1 Pruning Strategy 1: Parameter Thresholding

Our initial pruning approach, denoted by the function **Prune**, involves setting a threshold to selectively remove parameters with values below a specific cut-off. In this straightforward method, each parameter's value in the model is evaluated independently, without regard to its relationship within the Chebyshev decomposition structure. The pruning function is expressed as:

$$W^{(1^*)} = \text{Prune} \left( W^{(1)} \mid W^{(2)}, \dots, W^{(m)} \right)$$

where  $W^{(i)}$  represents the weights of layer  $i$  before pruning and  $W^{(i*)}$  represents the weights of layer  $i$  after pruning. Thus  $W^{(1^*)}$  represents the weights of layer 1 after pruning, and  $m$  is the total number of layers in the network. This equation indicates that the pruning of  $W^{(1)}$  is performed independently of the other layers, focusing solely on the values within layer 1 that fall below the threshold.

To further improve the pruning process, we define a function called **ForwardPrune** for subsequent layers. This function prunes weights in the current layer while taking into account the pruning state of previous layers. For example,

$$W^{(2^*)} = \text{ForwardPrune} \left( W^{(2)} \mid W^{(1^*)}, W^{(3)}, \dots, W^{(m)} \right)$$

where  $W^{(2^*)}$  represents the pruned weights of layer 2, taking into account that layer 1 has already been pruned. For any pruned layer (denoted with an asterisk \*), weights set to zero remain frozen in the fine-tuning process, meaning they are not allowed to change, while non-zero weights continue to be fine-tuned. This approach ensures that once weights are set to zero, they stay inactive in subsequent training.

In general, the pruning process for any layer  $l$  can be expressed as:

$$W^{(l^*)} = \text{Prune} \left( W^{(l)} \mid W^{(1^*)}, W^{(2^*)}, \dots, W^{((l-1)^*)}, W^{(l+1)}, \dots, W^{(m)} \right) \quad (13)$$

where  $m$  is the total number of layers in the network. This equation signifies that when pruning layer  $l$ , the method considers all previously pruned layers to ensure a structured, layer-wise pruning approach. This method allows us to progressively prune the network, layer by layer, while taking into account the structure of previously pruned layers, thereby optimizing parameter reduction without disrupting the network's overall architecture.

### 4.3.2 Pruning Strategy 2: Grouped Parameter Pruning for Chebyshev Decomposition

In our experiments, we also introduced a novel pruning method tailored to the Chebyshev decomposition, which considers the structure of parameters within each Chebyshev polynomial expansion. For each feature  $i$ , we compute a composite weight  $w_i$  that reflects the influence of all coefficients associated with that feature's Chebyshev terms. This is achieved by calculating the square root of the sum of squares for each coefficient  $c_{i,j}$  as follows:

$$\|w_i(x)\|_2 := \sqrt{\sum_{j=0}^k c_{i,j}^2} \quad (14)$$

where  $i$  is the feature index,  $j$  ranges from 0 to  $k$  (the highest Chebyshev polynomial degree), and  $c_{i,j}$  represents the coefficient for the  $j$ -th Chebyshev term in feature  $i$ 's expansion. Note that this calculation does not involve the input  $x$ , focusing solely on the coefficients.

Following the calculation of  $w_i$  values, we apply the same threshold-based pruning function as in the first strategy. If any  $w_i$  in equation 14 falls below the threshold, all coefficients  $c_{i,j}$  corresponding to that feature are pruned from the model. This approach ensures that low-impact features (as determined by their cumulative Chebyshev contributions) are removed in their entirety, enhancing the model's efficiency without compromising interpretability.

### 4.3.3 Evaluation Methodology

For each dataset, the following steps are conducted:

1. The dataset is split into training and testing subsets.
2. The MLP model is trained on the training subset 10 times, and the highest accuracy achieved on the test subset is recorded.
3. The Chebyshev model undergoes the same training process on the same data split, with the highest accuracy on the test subset also recorded.

Finally, the test accuracies for both the MLP and Chebyshev models are compared across all datasets, allowing us to assess the relative performance gains introduced by the Chebyshev enhancement.

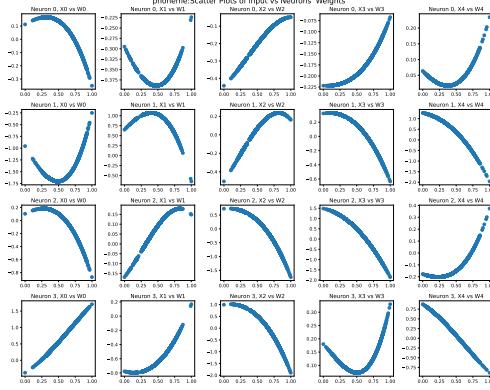


Figure 8: Scatter plots illustrating the relationship between inputs and adaptive weights for each neuron in the first layer, using the 'phoneme' dataset. The rows represent the neurons. At each row, the columns represent the weight distribution with respect to feature values. The smoothness in the weight transitions reflects the dynamic adaptability of weights based on varying input values.

## 4.4 Analysis of Weight Distribution in Adaptive Neurons

In this section, we examine how the weights in the adaptive neurons vary with changing input values, focusing on the first layer, which consists of 4 neurons. The weight distributions are analyzed and plotted for two representative datasets: 'phoneme' (Figure 8) and 'yeast' (Figure 9). Each plot demonstrates the relationship between the input features and their corresponding adaptive weights, revealing the smooth transitions as inputs change. This behavior showcases the

adaptability of the model in modulating weights based on input, which is instrumental in capturing non-linear patterns within the data.

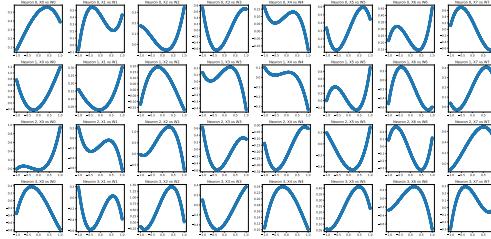


Figure 9: Scatter plots illustrating the relationship between inputs and adaptive weights for each neuron in the first layer, using the 'yeast' dataset. At each row, the columns represent the weight distribution with respect to feature values.. These plots further demonstrate the model's capacity to adjust weights smoothly according to input changes.

This adaptive behavior enhances the network's ability to generalize across various datasets, enabling improved performance particularly in scenarios where input data exhibits complex, non-linear dependencies.

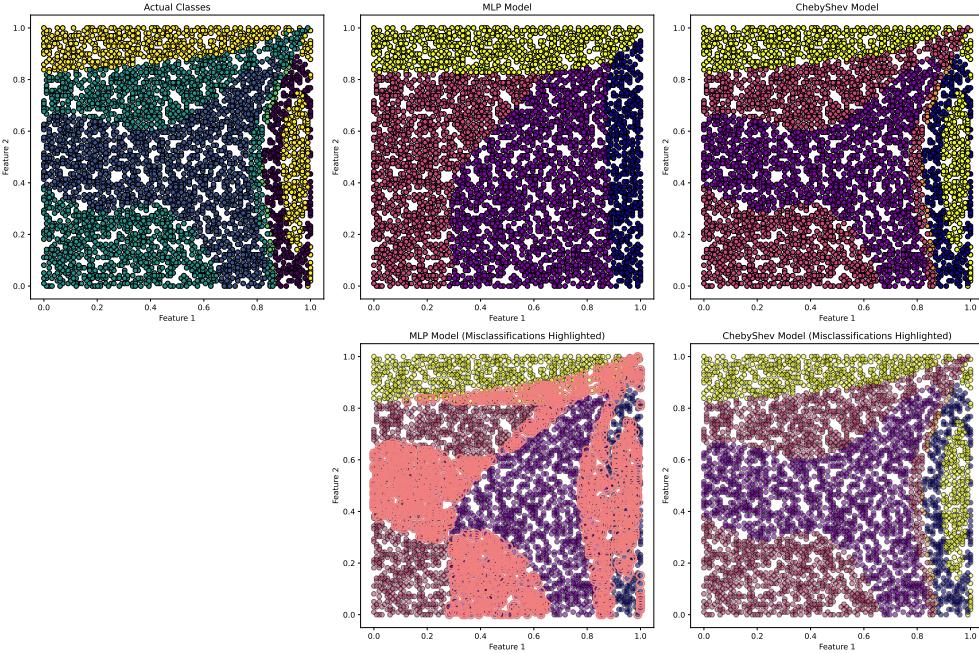


Figure 10: Comparison of decision boundaries on the "solar\_flare\_1" dataset with 5 classes. The first plot in the top row shows the actual class distribution. The second plot shows the MLP's decision boundaries, followed by Chebyshev's. The bottom row highlights misclassifications for each model, with pink areas indicating errors.

#### 4.5 Analysis of Decision Boundaries

To evaluate the ability of Chebyshev adaptive neural networks in learning decision boundaries, we analyzed their performance on challenging multiclass datasets. We trained both the Chebyshev model and the MLP on these datasets and observed the decision boundaries each model learned. Figure 10 presents an example with the "solar\_flare\_1" dataset, which consists of 5 non-linearly distributed classes. The first plot in the first row illustrates the actual class distribution. The second and third plots in the first row show the decision boundaries learned by the MLP and Chebyshev models, respectively. Below each model's boundary plot, the corresponding misclassifications are highlighted in pink, providing insights into the models' abilities to correctly classify the data. The Chebyshev adaptive neural network demonstrates a clear advantage in capturing non-linear boundaries and reducing misclassifications.

The superiority of the Chebyshev model over the MLP is further illustrated in Figure 11. Here, the decision boundary is highlighted in black. Figure 11a shows the original data scatter plot with the true decision boundary, while Figure 11b depicts the MLP's decision boundary, which struggles to capture the non-linearities. In contrast, Figure 11c shows the decision boundary learned by the Chebyshev model, which effectively captures the complex, non-linear boundaries.

These visualizations highlight the Chebyshev model's superior ability to learn complex, non-linear decision boundaries, underscoring its advantage over traditional MLPs in handling intricate data distributions.

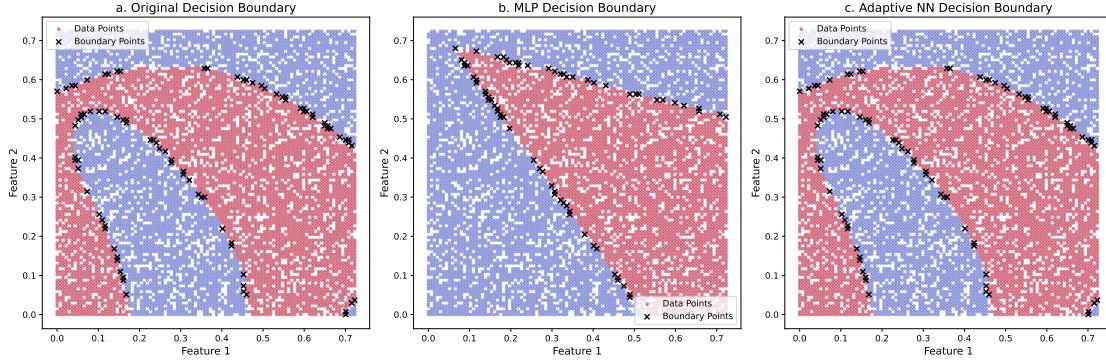


Figure 11: Decision boundary analysis on the "Hill Valley without noise" dataset. (a) Original scatter plot with decision boundary highlighted. (b) Decision boundary learned by the MLP, showing limited capacity to capture non-linearities. (c) Decision boundary learned by the Chebyshev model, effectively capturing non-linear boundaries.

Overall, the Chebyshev adaptive neural network demonstrates a strong capability in learning the decision boundaries of complex, non-linear datasets, outperforming the MLP in terms of boundary precision and reducing classification errors. Additional examples are provided in the supplementary section.

## 5 Results and Discussion

### 5.1 Results

To assess the effectiveness of Chebyshev neural networks with adaptive weights compared to traditional Multi-Layer Perceptrons (MLP), we conducted a series of experiments across 145 datasets. Table 2 presents a summary of the top 10 datasets where Chebyshev networks demonstrated the greatest improvement over MLP. The table displays the accuracy achieved by the Chebyshev Adaptive model, followed by the MLP accuracy, and finally the difference between them. On average, the Chebyshev model achieved a mean accuracy of 84.13%, outperforming the MLP's mean accuracy of 80.87%. Furthermore, Chebyshev networks outperformed MLP in 108 out of 145 datasets, amounting to 74% of cases, indicating a notable advantage in consistency and accuracy. Notably, 5% of the datasets exhibited accuracy gains exceeding 15%, while 10% showed improvements above 10%. Additionally, 22% of datasets observed a difference in accuracy of over 5%, underscoring the model's broad applicability across varied datasets. The highest recorded gain was 26.83%, observed in the "auto" dataset, where the MLP's accuracy of 48.78% was improved to 75.61% by the Chebyshev model. The complete comparison table is provided in the supplementary section.

These findings underscore the potential of the Chebyshev neural network with adaptive weights, demonstrating its enhanced accuracy across a diverse set of datasets, especially in cases with non-linear patterns that may not be effectively captured by traditional MLP architectures.

### 5.2 Performance Gains with Chebyshev Adaptive Networks

Figure 12 displays a bar plot of delta values, representing the difference between Chebyshev and MLP accuracies, sorted in descending order. This plot emphasizes the Chebyshev model's substantial positive impact across multiple datasets, with positive deltas highlighted to indicate cases where it outperformed the MLP. As the Chebyshev adaptive neural network generalizes the MLP architecture, its performance is either equivalent to or better than that of the MLP. In 108 cases, the Chebyshev model achieved higher accuracy than the MLP, while in the remaining cases, it matched the MLP performance with a delta of zero.

Dataset	Chebyshev Accuracy	MLP Accuracy	Improvement
auto	75.61	48.78	26.83
vowel	86.87	60.61	26.26
analcatdata_fraud	88.89	66.67	22.22
soybean	84.44	62.96	21.48
tic_tac_toe	98.44	79.17	19.27
letter	54.48	36.05	18.43
movement_libras	58.33	43.06	15.28
ring	97.70	84.66	13.04
car	96.53	83.82	12.72
hayes_roth	81.25	68.75	12.50

Table 2: Top 10 datasets with the largest positive delta in accuracy, showcasing cases where Chebyshev networks significantly outperformed MLPs.

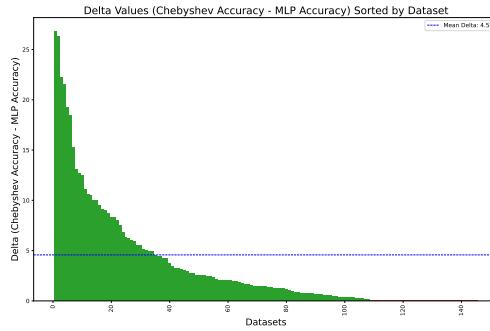


Figure 12: Bar plot of delta values (Chebyshev accuracy - MLP accuracy) sorted by dataset, illustrating significant improvements achieved by the Chebyshev model across various datasets.

### 5.3 Accuracy Comparison

The supremacy of the Chebyshev adaptive network is evident in Figure 13, which presents a scatter plot comparing Chebyshev and MLP accuracies for each dataset. A 45-degree reference line demarcates the datasets; points above this line correspond to cases where Chebyshev outperformed MLP, underscoring the widespread instances of Chebyshev's enhanced accuracy.

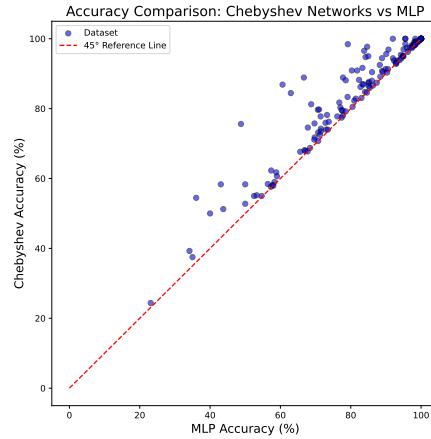


Figure 13: Scatter plot that compares the accuracy rates between Chebyshev and MLP methods across various datasets

The results highlight the effectiveness of the Chebyshev neural network with adaptive weights as a robust alternative to traditional MLP architectures. By leveraging Chebyshev polynomials, the adaptive model dynamically adjusts weights in response to input variations, capturing complex, non-linear patterns that MLPs often miss that will be discussed in the following subsection. The Chebyshev model consistently achieved higher or equivalent accuracy compared to MLP across a diverse set of 145 datasets, with notable gains in 74% of cases. These results suggest that adaptive networks, particularly those using polynomial transformations, offer substantial improvements in predictive accuracy and generalization, making them valuable in diverse real-world applications where traditional neural networks may be limited.

#### 5.4 Performance after Pruning

The results in Table 3 illustrate the performance of the pruned Chebyshev adaptive neural network across various datasets, with datasets ordered by decreasing compression levels. We show the 7 datasets where maximum compression could be achieved. The rest of the table can be found in the appendix section. Remarkably, the pruning process allowed for up to 90% compression in model size, accompanied by an increase in accuracy compared to the original MLP performance. In cases where the unpruned Chebyshev model accuracy matched that of the MLP, pruning enabled a boost in accuracy, demonstrating the effectiveness of this approach in both reducing model complexity and enhancing performance.

<b>Dataset</b>	<b>MLP Accuracy</b>	<b>ChebyShev Adaptive Model Accuracy (Pruned)</b>	<b>Compression</b>
wdbc	98.246	99.123	89.2
credit a	84.057	84.783	89.1
tokyo1	93.229	93.75	86.3
breast	100	100	85.8
pima	78.571	79.221	82.9
cmc	57.288	57.627	80.3
cleveland nominal	59.016	60.656	79.7

Table 3: Performance of pruned Chebyshev adaptive neural networks compared to standard MLPs across various datasets. The table highlights improvements in accuracy achieved by pruning, with the datasets arranged in decreasing order of model compression. For datasets where the unpruned Chebyshev model matched MLP performance, pruning not only increased accuracy but also resulted in substantial model compression, achieving up to 90% reduction in model size while enhancing accuracy. The table columns display the dataset names, MLP accuracy, pruned Chebyshev model accuracy, and the compression percentage.

## 6 Future Directions

Future work could explore the application of adaptive weights in out-of-distribution scenarios, focusing on their potential to enhance model resilience and improve generalization beyond standard datasets. Additionally, alternative decomposition methods, such as Fourier, Legendre, and Hermite polynomials, could be investigated to assess how different decomposition techniques impact model performance across various dataset characteristics. By examining these alternatives, research could identify which decomposition types are best suited for specific dataset attributes, including class count, class imbalance, and data distribution properties, thereby enabling more targeted applications of adaptive models.

For functions involving multiple variables,  $f(x_1, x_2, \dots, x_d)$ , Chebyshev decomposition can be extended to higher dimensions through a multivariate Chebyshev series expansion. This approach approximates  $f(x_1, x_2, \dots, x_d)$  using products of univariate Chebyshev polynomials for each variable, providing a powerful tool for multivariate function approximation.

The following is a step-by-step outline for decomposing a multivariate function using Chebyshev polynomials, which forms the foundation for future exploration in high-dimensional adaptive networks.

### 6.1 Steps for Multivariate Chebyshev Decomposition

To extend Chebyshev decomposition to multivariate functions, we start by defining the multivariate Chebyshev polynomial basis. For  $d$  dimensions, the basis functions are products of univariate Chebyshev polynomials for

each variable, so for a given degree  $m_i$  in each dimension  $x_i$ , the basis function is  $T_{m_1, m_2, \dots, m_d}(x_1, x_2, \dots, x_d) = T_{m_1}(x_1)T_{m_2}(x_2) \cdots T_{m_d}(x_d)$ , where  $T_{m_i}(x_i)$  represents the Chebyshev polynomial of degree  $m_i$  in the  $i$ -th variable, defined on  $[-1, 1]$ . If the domain of  $f(x_1, x_2, \dots, x_d)$  lies outside  $[-1, 1]^d$ , we map each variable  $x_i$  from  $[a_i, b_i]$  to  $[-1, 1]$  using  $x'_i = \frac{2x_i - (a_i + b_i)}{b_i - a_i}$ , scaling and shifting  $x_i$  so that  $f(x_1, x_2, \dots, x_d)$  is transformed to  $f(x'_1, x'_2, \dots, x'_d)$  on  $[-1, 1]^d$ . We then approximate  $f(x_1, x_2, \dots, x_d)$  by a finite multivariate Chebyshev series:

$$f(x_1, x_2, \dots, x_d) \approx \sum_{m_1=0}^{M_1} \sum_{m_2=0}^{M_2} \cdots \sum_{m_d=0}^{M_d} c_{m_1, m_2, \dots, m_d} T_{m_1}(x_1) T_{m_2}(x_2) \cdots T_{m_d}(x_d),$$

where  $c_{m_1, m_2, \dots, m_d}$  are the Chebyshev coefficients. These coefficients are calculated as

$$c_{m_1, m_2, \dots, m_d} = \frac{2^d}{\pi^d} \int_{-1}^1 \cdots \int_{-1}^1 \frac{f(x_1, x_2, \dots, x_d) T_{m_1}(x_1) \cdots T_{m_d}(x_d)}{\sqrt{1-x_1^2} \cdots \sqrt{1-x_d^2}} dx_1 \cdots dx_d,$$

but due to computational difficulty, these integrals are usually approximated using Chebyshev nodes and discrete transforms. For each variable  $x_i$ , we choose  $M_i + 1$  Chebyshev nodes as  $x_i^{(k)} = \cos\left(\frac{(2k+1)\pi}{2(M_i+1)}\right)$ , for  $k = 0, 1, \dots, M_i$ , and evaluate  $f$  at each point on the resulting  $(M_1 + 1) \times (M_2 + 1) \times \cdots \times (M_d + 1)$  grid, yielding  $f_{k_1, k_2, \dots, k_d} = f(x_1^{(k_1)}, x_2^{(k_2)}, \dots, x_d^{(k_d)})$ . To find the coefficients  $c_{m_1, m_2, \dots, m_d}$ , we then apply a multidimensional Discrete Cosine Transform (DCT) to these sampled values. Finally, the multivariate Chebyshev polynomial approximation for  $f(x_1, x_2, \dots, x_d)$  is constructed as

$$f(x_1, x_2, \dots, x_d) \approx \sum_{m_1=0}^{M_1} \sum_{m_2=0}^{M_2} \cdots \sum_{m_d=0}^{M_d} c_{m_1, m_2, \dots, m_d} T_{m_1}(x_1) T_{m_2}(x_2) \cdots T_{m_d}(x_d).$$

## 6.2 Example: 3D Chebyshev Decomposition with Pairwise Combinations

For a function of three variables,  $f(x, y, z)$ , a pairwise Chebyshev decomposition can help in capturing interactions between pairs of variables. This involves decomposing  $f(x, y, z)$  as a sum of functions of pairs of variables:

$$f(x, y, z) \approx \sum_{m=0}^M \sum_{n=0}^N c_{m,n}^{(x,y)} T_m(x) T_n(y) + \sum_{m=0}^M \sum_{p=0}^P c_{m,p}^{(x,z)} T_m(x) T_p(z) + \sum_{n=0}^N \sum_{p=0}^P c_{n,p}^{(y,z)} T_n(y) T_p(z),$$

where  $c_{m,n}^{(x,y)}$ ,  $c_{m,p}^{(x,z)}$ , and  $c_{n,p}^{(y,z)}$  are the Chebyshev coefficients for each pairwise combination.

## 6.3 Efficient Multivariate Function Approximation Using Pairwise Chebyshev Decomposition

We aim to classify a dataset with features  $x_1, x_2, \dots, x_d$ , with the objective of predicting  $y$  based on the function  $y = f(x_1, x_2, \dots, x_d)$ . The question is how to estimate the function  $f(x_1, x_2, \dots, x_d)$ . Using function decomposition, we can represent  $f(x_1, \dots, x_d)$  with orthogonal functions  $T_{m_1, m_2, \dots, m_d}(x_1, \dots, x_d)$  as follows:

$$f(x_1, x_2, \dots, x_d) = \sum_{m_1=0}^{\infty} \sum_{m_2=0}^{\infty} \cdots \sum_{m_d=0}^{\infty} c_{m_1, m_2, \dots, m_d} T_{m_1, m_2, \dots, m_d}(x_1, x_2, \dots, x_d),$$

or

$$f(x_1, x_2, \dots, x_d) \approx \sum_{m_1=0}^k \cdots \sum_{m_d=0}^k c_{m_1, m_2, \dots, m_d} T_{m_1, \dots, m_d}(x_1, x_2, \dots, x_d).$$

Thus, we have  $(k + 1)^d$  parameters to estimate. While this estimation is feasible in low dimensions, the problem becomes significantly more difficult as the dimensionality  $d$  increases.

## 6.4 Pairwise Decomposition for Dimensionality Reduction

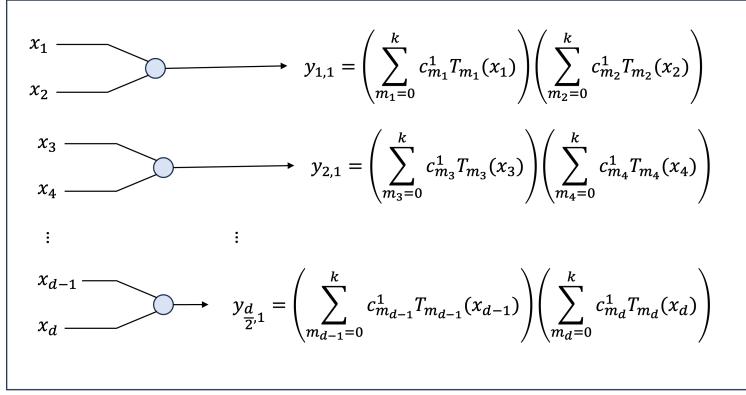


Figure 14: Initial layer of the pairwise decomposition, illustrating the pairwise approximation of variables  $x_1, x_2, \dots, x_d$ .

In high-dimensional cases, direct multivariate Chebyshev decomposition can lead to an excessive number of parameters. To address this, we propose a pairwise decomposition strategy to reduce the complexity of approximation. Suppose we have a function  $f(x_1, x_2, \dots, x_d)$  and aim to classify or predict based on the features  $x_1, x_2, \dots, x_d$ . The goal is to approximate  $f(x_1, x_2, \dots, x_d)$  through orthogonal decomposition using Chebyshev polynomials. Using Chebyshev decomposition, we approximate  $f(x_1, x_2, \dots, x_d)$  as:

$$f(x_1, \dots, x_d) \approx \sum_{m_1=0}^k \cdots \sum_{m_d=0}^k c_{m_1, \dots, m_d} T_{m_1}(x_1) \cdots T_{m_d}(x_d),$$

where  $T_{m_i}(x_i)$  denotes the Chebyshev polynomial of degree  $m_i$  for each variable  $x_i$ . This expression involves  $(k+1)^d$  parameters, which becomes computationally challenging as  $d$  increases. To reduce dimensionality, we assume pairwise combinations of variables, simplifying the expression by decomposing each pair:

$$T_{m_1, m_2, \dots, m_d}(x_1, x_2, \dots, x_d) = \prod_{i=1}^d T_{m_i}(x_i).$$

We then approximate  $f(x_1, \dots, x_d)$  by summing over pairwise terms, effectively reducing the parameter space. We calculate intermediate outputs for each pairwise combination:

$$y_{1,1} = \sum_{m_1=0}^k c_{m_1} T_{m_1}(x_1), \quad y_{2,1} = \sum_{m_2=0}^k c_{m_2} T_{m_2}(x_2),$$

and so forth, until the final output is obtained by aggregating these pairwise decomposed terms. Through backpropagation, we estimate each coefficient  $c_j^i$ , balancing approximation accuracy with computational feasibility. This method is particularly suitable for high-dimensional data where computational efficiency is critical.

In summary, Figures 14, 15, and 16 illustrate the use of pairwise decomposition at each layer, which helps to handle the explosion of parameters in high-dimensional approximation. This approach leads to a final decomposition depth  $r+1 = \log_e(d)$ , and backpropagation is used to estimate each coefficient  $c_j^i$ , thereby enabling efficient and accurate high-dimensional function approximation.

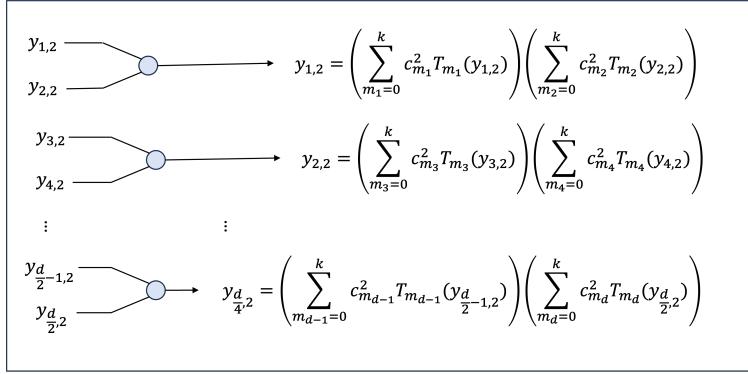
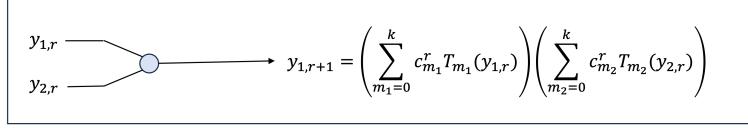


Figure 15: Intermediate layer showing further pairwise combinations to reduce parameter space complexity.

Figure 16: Final aggregation layer where pairwise approximations are combined to form the complete approximation of  $f(x_1, x_2, \dots, x_d)$ .

## 7 Conclusion

This paper introduces an adaptive neural network model utilizing Chebyshev polynomials to achieve input-dependent weighting, emulating biological adaptability and enhancing the network's ability to capture complex, non-linear patterns. Empirical evaluation on 145 datasets from the PMLB benchmark reveals that the Chebyshev model outperforms traditional Multi-Layer Perceptrons (MLP) in 74% of cases, achieving a mean accuracy of 84.13% versus 80.87% for MLP, with the highest accuracy gain reaching 26.83%. Pruning further enhances efficiency, achieving up to 90% compression without sacrificing performance. This adaptable framework shows promise in handling high-dimensional and out-of-distribution data, making it broadly applicable in areas with complex data dependencies, like healthcare and finance. Overall, this Chebyshev-based adaptive approach provides a flexible and efficient alternative to fixed-weight neural architectures, paving the way for future research into other orthogonal decompositions and high-dimensional adaptive models.

## Acknowledgments

This was supported in part by.....

## References

- [1] Bruce Alberts. Molecular biology of the cell 4th edition. (*No Title*), 2002.
- [2] Mark F Bear, Barry W Connors, Michael A Paradiso, MF Bear, BW Connors, and MA Neuroscience. Exploring the brain. *Neuroscience 3ra. Ed. Baltimore*, 541, 2007.
- [3] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

- [4] Tim VP Bliss and Graham L Collingridge. A synaptic model of memory: long-term potentiation in the hippocampus. *Nature*, 361(6407):31–39, 1993.
- [5] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017.
- [6] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.
- [7] John P Boyd. *Chebyshev and Fourier spectral methods*. Courier Corporation, 2001.
- [8] Carlos Stein Naves de Brito and Wulfram Gerstner. Learning what matters: Synaptic plasticity with invariance to second-order input correlations. *PLOS Computational Biology*, 20(2):e1011844, 2024.
- [9] Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. Dynamic convolution: Attention over convolution kernels. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11030–11039, 2020.
- [10] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 764–773, 2017.
- [11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.
- [12] Andrew Duggleby, Kenneth S Ball, Mark R Paul, and Paul F Fischer. Dynamical eigenfunction decomposition of turbulent pipe flow. *Journal of Turbulence*, (8):N43, 2007.
- [13] Wei Fang, Zhaofei Yu, Yanqi Chen, Tiejun Huang, Timothée Masquelier, and Yonghong Tian. Deep residual learning in spiking neural networks. *Advances in Neural Information Processing Systems*, 34:21056–21069, 2021.
- [14] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.
- [15] Ian Goodfellow. Deep learning, 2016.
- [16] Yufei Guo, Xuhui Huang, and Zhe Ma. Direct learning-based deep spiking neural networks: a review. *Frontiers in Neuroscience*, 17:1209795, 2023.
- [17] Bing Han and Kaushik Roy. Deep spiking neural network: Energy efficiency through time based coding. In *European conference on computer vision*, pages 388–404. Springer, 2020.
- [18] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2021.
- [19] Adam W Harley, Konstantinos G Derpanis, and Iasonas Kokkinos. Segmentation-aware convolutional networks using local attention masks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5038–5047, 2017.
- [20] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.
- [21] Bertil Hille. Ion channels of excitable membranes third edition. (*No Title*), 2001.
- [22] S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.
- [23] Ronald R Hoy. Principles of neurobiology. *Journal of Undergraduate Neuroscience Education*, 15(1):R22, 2016.
- [24] Eric R Kandel, James H Schwartz, Thomas M Jessell, Steven Siegelbaum, A James Hudspeth, Sarah Mack, et al. *Principles of neural science*, volume 4. McGraw-hill New York, 2000.
- [25] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [26] Christof Koch. *Biophysics of computation: information processing in single neurons*. Oxford university press, 2004.
- [27] Christof Koch and Idan Segev. The role of single neurons in information processing. *Nature neuroscience*, 3(11):1171–1177, 2000.
- [28] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks. *arXiv preprint arXiv:2404.19756*, 2024.
- [29] Sergey A Lobov, Alexey N Mikhaylov, Maxim Shamshin, Valeri A Makarov, and Victor B Kazantsev. Spatial properties of stdp in a self-learning spiking neural network enable controlling a mobile robot. *Frontiers in neuroscience*, 14:88, 2020.

- [30] John C Mason and David C Handscomb. *Chebyshev polynomials*. Chapman and Hall/CRC, 2002.
- [31] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.
- [32] Zaid Odibat. On legendre polynomial approximation with the vim or ham for numerical treatment of nonlinear fractional differential equations. *Journal of Computational and Applied Mathematics*, 235(9):2956–2968, 2011.
- [33] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(36):1–13, Dec 2017.
- [34] Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: opportunities and challenges. *Frontiers in neuroscience*, 12:409662, 2018.
- [35] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.
- [36] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.
- [37] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.
- [38] Theodore J Rivlin. *Chebyshev polynomials*. Courier Dover Publications, 2020.
- [39] Tim Rohe and Marc L Zeise. Inputs, outputs, and multisensory processing. *Neuroscience for Psychologists: An Introduction*, pages 153–192, 2021.
- [40] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [41] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [42] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- [43] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
- [44] Nelson Spruston, Greg Stuart, and Michael Häusser. Dendritic integration. *Dendrites*, pages 231–271, 1999.
- [45] Jean-Luc Starck, Jalal Fadili, and Fionn Murtagh. The undecimated wavelet decomposition and its reconstruction. *IEEE transactions on image processing*, 16(2):297–309, 2007.
- [46] Mallat Stephane. A wavelet tour of signal processing, 1999.
- [47] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.
- [48] Gerald Teschl. *Ordinary differential equations and dynamical systems*, volume 140. American Mathematical Society, 2024.
- [49] Lloyd N Trefethen. Approximation theory and approximation practice. In *Book of abstracts*, page 21, 2009.
- [50] Ioannis A Troumbis, George E Tsekouras, John Tsimikas, Christos Kalloniatis, and Dias Haralambopoulos. A chebyshev polynomial feedforward neural network trained by differential evolution and its application in environmental case studies. *Environmental Modelling & Software*, 126:104663, 2020.
- [51] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, Yuan Xie, and Luping Shi. Direct training for spiking neural networks: Faster, larger, better. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 1311–1318, 2019.
- [52] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. Condconv: Conditionally parameterized convolutions for efficient inference. *Advances in neural information processing systems*, 32, 2019.
- [53] Xiaoyu Yang and Giancarlo La Camera. Co-existence of synaptic plasticity and metastable dynamics in a spiking model of cortical circuits. *PLOS Computational Biology*, 20(7):e1012220, 2024.
- [54] Davide Zambrano, Roeland Nusselder, H Steven Scholte, and Sander M Bohté. Sparse computation in adaptive spiking neural networks. *Frontiers in neuroscience*, 12:987, 2019.
- [55] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

# Appendix

## A Complete Dataset description

Dataset Name	Size	Number of Features	Number of Classes
adult	29305	14	2
agaricus lepiota	4887	22	2
allbp	2263	29	3
allhyper	2262	29	4
allhypo	2262	29	3
allrep	2263	29	4
analcatdata aids	30	4	2
analcatdata asbestos	49	3	2
analcatdata author-ship	504	70	4
analcatdata bankruptcy	30	6	2
analcatdata boxing1	72	3	2
analcatdata boxing2	79	3	2
analcatdata cred-itscore	60	6	2
analcatdata cyy-oung8092	58	10	2
analcatdata cyy-oung9302	55	10	2
analcatdata dmft	478	4	6
analcatdata fraud	25	11	2
analcatdata ger-mangss	240	5	4
analcatdata happiness	36	3	3
analcatdata japansolvent	31	9	2
analcatdata lawsuit	158	4	2
ann thyroid	4320	21	3
appendicitis	63	7	2
australian	414	14	2
auto	121	25	5
backache	108	32	2
balance scale	375	4	3
biomed	125	8	2
breast	419	10	2
breast cancer	171	9	2
breast cancer wisconsin	341	30	2
breast w	419	9	2
buggyCrx	414	15	2
bupa	207	5	2
calendarDOW	239	32	5
car	1036	6	4
car evaluation	1036	21	4
cars	235	8	3
chess	1917	36	2
churn	3000	20	2
clean1	285	168	2
clean2	3958	168	2
cleve	181	13	2
cleveland	181	13	5

<b>Dataset Name</b>	<b>Size</b>	<b>Number of Features</b>	<b>Number of Classes</b>
cleveland nominal	181	7	5
cmc	883	9	3
coil2000	5893	85	2
colic	220	22	2
collins	291	23	13
connect 4	40534	42	3
contraceptive	883	9	3
corral	96	6	2
credit a	414	15	2
credit g	600	20	2
crx	414	15	2
dermatology	219	34	6
diabetes	460	8	2
dis	2263	29	2
dna	1911	180	3
ecoli	196	7	5
fars	60580	29	8
flags	106	43	5
flare	639	10	2
GAMETES Epistasis	960	1000	2
2 Way 1000atts 0.4H			
EDM 1 EDM 1 1			
GAMETES Epistasis	960	20	2
2 Way 20atts 0.1H			
EDM 1 1			
GAMETES Epistasis	960	20	2
2 Way 20atts 0.4H			
EDM 1 1			
GAMETES Epistasis	960	20	2
3 Way 20atts 0.2H			
EDM 1 1			
GAMETES Heterogeneity	960	20	2
20atts 1600 Het 0.4 0.2 50 EDM 2 001			
GAMETES Heterogeneity	960	20	2
20atts 1600 Het 0.4 0.2 75 EDM 2 001			
german	600	20	2
glass	123	9	5
glass2	97	9	2
haberman	183	3	2
hayes roth	96	4	3
heart c	181	13	2
heart h	176	13	2
heart statlog	162	13	2
hepatitis	93	19	2
Hill Valley with noise	727	100	2
Hill Valley without noise	727	100	2
horse colic	220	22	2
house votes 84	261	16	2
hungarian	176	13	2
hypothyroid	1897	25	2
ionosphere	210	34	2
iris	90	4	3
irish	300	5	2

<b>Dataset Name</b>	<b>Size</b>	<b>Number of Features</b>	<b>Number of Classes</b>
kr vs kp	1917	36	2
krkopt	16833	6	18
led24	1920	24	10
led7	1920	7	10
letter	12000	16	26
mfeat factors	1200	216	10
mfeat fourier	1200	76	10
mfeat karhunen	1200	64	10
mfeat morphological	1200	6	10
mfeat zernike	1200	47	10
mofn 3 7 10	794	10	2
monk1	333	6	2
monk2	360	6	2
monk3	332	6	2
movement libras	216	90	15
mushroom	4874	22	2
new thyroid	129	5	3
optdigits	3372	64	10
page blocks	3283	10	5
parity5+5	674	10	2
penguins	199	7	3
phoneme	3242	5	2
pima	460	8	2
prnn crabs	120	7	2
prnn fglass	123	9	5
prnn synth	150	2	2
profbc	403	9	2
ring	4440	20	2
saheart	277	9	2
satimage	3861	36	6
schizo	204	14	3
segmentation	1386	19	7
sleep	63544	13	5
solar flare 1	189	12	5
solar flare 2	639	12	6
sonar	124	60	2
soybean	405	35	18
spambase	2760	57	2
spect	160	22	2
spectf	209	44	2
splice	1912	60	3
tae	90	5	3
texture	3300	40	11
threeOf9	307	9	2
tic tac toe	574	9	2
tokyo1	575	44	2
twonorm	4440	20	2
vehicle	507	18	4
vote	261	16	2
vowel	594	13	11
waveform 21	3000	21	3
waveform 40	3000	40	3
wdbc	341	30	2
wine quality red	959	11	6
wine quality white	2938	11	7
wine recognition	106	13	3
xd6	583	9	2
yeast	887	8	9

## B Complete Results

Table 5: Comparison of Chebyshev Adaptive Model and MLP Accuracy Across Datasets

Dataset Name	Chebyshev Model Accuracy	Adaptive	MLP Accuracy	Difference (Chebyshev - MLP)
auto	75.61		48.78	<b>26.829</b>
vowel	86.869		60.606	<b>26.263</b>
analcatdata fraud	88.889		66.667	<b>22.222</b>
soybean	84.444		62.963	<b>21.481</b>
tic tac toe	98.438		79.167	<b>19.271</b>
letter	54.475		36.05	<b>18.425</b>
movement libras	58.333		43.056	<b>15.278</b>
ring	97.703		84.662	<b>13.041</b>
car	96.532		83.815	<b>12.717</b>
hayes roth	81.25		68.75	<b>12.5</b>
analcatdata boxing2	88.889		77.778	<b>11.111</b>
ecoli	90.909		80.303	<b>10.606</b>
analcatdata cyyoung9302	94.737		84.211	<b>10.526</b>
analcatdata cyyoung8092	95		85	<b>10</b>
analcatdata aids	50		40	10
sonar	88.095		78.571	9.524
analcatdata japansolvent	90.909		81.818	9.091
connect 4	79.677		70.634	9.044
mfeat zernike	79.75		71	8.75
analcatdata boxing1	91.667		83.333	8.333
analcatdata happiness	58.333		50	8.333
monk1	100		91.964	8.036
calendarDOW	51.25		43.75	7.5
hungarian	74.576		67.797	6.78
solar flare 1	77.777		71.429	6.349
collins	96.907		90.722	6.186
glass2	75.758		69.697	6.061
analcatdata asbestos	88.235		82.353	5.882
backache	94.444		88.889	5.556
texture	95.636		90.091	5.545
krkopt	39.273		34.141	5.132
mfeat karhunen	81.75		76.75	5
Hill Valley with noise	78.189		73.251	4.938
cleveland	62.295		57.377	4.918
appendicitis	100		95.455	4.545
parity5+5	100		95.556	4.444
splice	90.439		86.05	4.389
phoneme	83.349		79.093	4.255
mfeat factors	92.5		88.25	4.25
spect	87.037		83.333	3.704
breast cancer	86.207		82.759	3.448
cleve	86.885		83.607	3.279
heart c	80.328		77.049	3.279
GAMETES Epistasis 2 Way	75.938		72.813	3.125
20atts 0.4H EDM 1 1				
penguins	98.507		95.522	2.985
schizo	61.765		58.824	2.941
flags	52.777		50	2.777
mfeat morphological	74.25		71.5	2.75
GAMETES Epistasis 2 Way	55		52.5	2.5
1000atts 0.4H EDM 1 EDM				
1 1				

<b>Dataset Name</b>	<b>Chebyshev Model Accuracy</b>	<b>Adaptive</b>	<b>MLP Accuracy</b>	<b>Difference (Chebyshev - MLP)</b>
GAMETES Heterogeneity	76.25		73.75	2.5
20atts 1600 Het 0.4 0.2 75				
EDM 2 001				
analcatdata germangss	37.5		35	2.5
optdigits	87.633		85.142	2.491
glass	73.171		70.732	2.439
balance scale	94.4		92	2.4
satimage	87.334		85.004	2.33
saheart	79.569		77.419	2.15
led24	67.656		65.625	2.031
yeast	58.446		56.419	2.027
mfeat fourier	71.75		69.75	2
prnn synth	88		86	2
credit g	73.5		71.5	2
segmentation	96.97		95.022	1.948
wine quality white	55.204		53.265	1.939
heart statlog	90.741		88.889	1.852
vehicle	79.412		77.647	1.765
monk2	100		98.347	1.653
cleveland nominal	60.656		59.016	1.639
solar flare 2	77.77		76.19	1.58
profb	71.111		69.629	1.482
crx	91.304		89.855	1.449
car evaluation	97.399		95.954	1.445
spectf	82.857		81.429	1.429
dna	93.73		92.32	1.411
colic	90.541		89.189	1.351
horse colic	86.486		85.135	1.351
cars	82.278		81.013	1.266
analcatdata dmft	24.375		23.125	1.25
GAMETES Heterogeneity	72.5		71.25	1.25
20atts 1600 Het 0.4 0.2 50				
EDM 2 001				
GAMETES Epistasis 2 Way	68.125		66.875	1.25
20atts 0.1H EDM 1 1				
vote	95.402		94.252	1.15
sleep	74.011		72.982	1.029
threeOf9	99.029		98.058	0.971
wdbc	99.123		98.246	0.877
Hill Valley without noise	67.901		67.078	0.823
allrep	98.14		97.35	0.79
credit a	84.783		84.057	0.726
australian	89.13		88.406	0.725
buggyCrx	91.304		90.58	0.724
breast w	99.286		98.571	0.714
contraceptive	58.983		58.305	0.678
allbp	97.483		96.821	0.662
pima	79.221		78.571	0.649
adult	85.935		85.393	0.543
tokyo1	93.75		93.229	0.521
german	74		73.5	0.5
twonorm	98.243		97.837	0.406
churn	93.2		92.8	0.4
cmc	57.627		57.288	0.339
hypothyroid	97.946		97.63	0.316
GAMETES Epistasis 3 Way	55		54.688	0.313
20atts 0.2H EDM 1 1				

<b>Dataset Name</b>	<b>Chebyshev Model Accuracy</b>	<b>Adaptive</b>	<b>MLP Accuracy</b>	<b>Difference (Chebyshev - MLP)</b>
wine quality red	58.125		57.813	0.313
led7	68.75		68.438	0.313
waveform 40	86.6		86.3	0.3
allhypo	95.093		94.828	0.265
dis	98.278		98.013	0.265
fars	78.028		77.795	0.233
page blocks	94.977		94.795	0.183
ann thyroid	98.888		98.75	0.138
new thyroid	97.674		97.674	0
biomed	92.857		92.857	0
flare	84.579		84.579	0
bupa	57.971		57.971	0
house votes 84	98.851		98.851	0
analcatdata lawsuit	96.226		96.226	0
mofn 3 7 10	100		100	0
heart h	83.051		83.051	0
agaricus lepiota	100		100	0
analcatdata bankruptcy	100		100	0
ionosphere	92.958		92.958	0
irish	100		100	0
monk3	98.198		98.198	0
kr vs kp	99.531		99.531	0
wine recognition	100		100	0
hepatitis	90.323		90.323	0
iris	100		100	0
clean2	100		100	0
tae	67.742		67.742	0
prnn fglass	70.732		70.732	0
corral	100		100	0
xd6	100		100	0
coil2000	93.893		93.893	0
breast cancer wisconsin	98.246		98.246	0
chess	99.531		99.531	0
analcatdata creditscore	95		95	0
haberman	77.419		77.419	0
prnn crabs	100		100	0
allhyper	98.278		98.278	0
mushroom	100		100	0
breast	100		100	0
waveform 21	87.4		87.4	0
spambase	94.354		94.354	0
diabetes	80.519		80.519	0
analcatdata authorship	99.408		99.408	0
clean1	100		100	0
dermatology	97.297		97.297	0

## C Performance of Pruned ChebyShev model

Comparison of pruned Chebyshev adaptive neural networks with standard MLPs across a variety of datasets. Table 6 showcases accuracy gains achieved through pruning, with datasets organized by decreasing levels of compression. Pruning was specifically applied to networks where the unpruned Chebyshev model matched the accuracy of the MLP, leading to notable improvements in accuracy and significant model compression, reaching up to 90% reduction in model size. The columns present dataset names, MLP accuracy, pruned Chebyshev model accuracy, and the achieved compression percentage.

<b>Dataset</b>	<b>MLP Accuracy</b>	<b>ChebyShev Adaptive Model Accuracy (Pruned)</b>	<b>Compression</b>
wdbc	98.246	99.123	89.2
credit a	84.057	84.783	89.1
tokyo1	93.229	93.75	86.3
breast	100	100	85.8
pima	78.571	79.221	82.9
cmc	57.288	57.627	80.3
cleveland nominal	59.016	60.656	79.7
GAMETES Heterogeneity 20atts 1600 Het 0.4 0.2 75 EDM 2 001	73.75	76.25	77.5
GAMETES Epistasis 2 Way 20atts 0.1H EDM 1 1	66.875	68.125	75.3
contraceptive	58.305	58.983	74.2
buggyCrx	90.58	91.304	74
heart statlog	88.889	90.741	73.8
profb	69.629	71.111	69.5
cleve	83.607	86.885	66.7
flags	50	52.777	66.6
colic	89.189	90.541	61
GAMETES Epistasis 2 Way 1000atts 0.4H EDM 1 EDM 1 1	52.5	55	57.6
GAMETES Heterogeneity 20atts 1600 Het 0.4 0.2 50 EDM 2 001	71.25	72.5	56.7
Hill Valley with noise	73.251	78.189	55.2
saheart	77.419	79.569	52
analcatdata asbestos	82.353	88.235	49
german	73.5	74	48.1
breast cancer	82.759	86.207	46

Table 6: Performance of pruned Chebyshev adaptive neural networks compared to standard MLPs across various datasets. The table highlights improvements in accuracy achieved by pruning, with the datasets arranged in decreasing order of model compression. For datasets where the unpruned Chebyshev model matched MLP performance, pruning not only increased accuracy but also resulted in substantial model compression, achieving up to 90% reduction in model size while enhancing accuracy. The table columns display the dataset names, MLP accuracy, pruned Chebyshev model accuracy, and the compression percentage.

## D Distribution of Weight for Different Datasets

In this section, we examine the adaptive value of the neural network parameters with changing inputs. After training a Chebyshev adaptive model, we pass input values within the range of -1 to 1 and observe the corresponding adaptive weight values. These values are then plotted to illustrate how the weights respond to varying inputs, enhancing the model's capacity for tasks such as classification. The curve shapes in these plots indicate that the adaptive weights effectively respond to input variations, enabling improved performance in downstream tasks.

Figures 17, 18, and 19 depict the adaptive weight distribution for the *contraceptive*, *led7*, and *mfeat\_morphological* datasets, respectively. These figures highlight the smooth response of weights to the input variations, supporting the model's improved classification capabilities and flexibility.

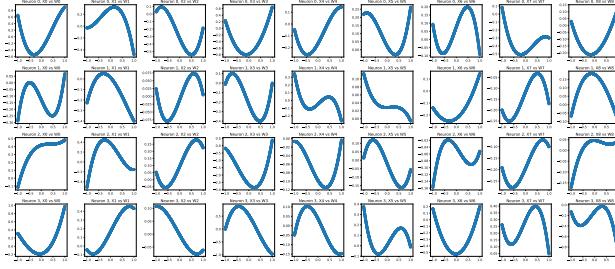


Figure 17: Adaptive weight distribution for the *contraceptive* dataset, showing the weight’s smooth variation with input values, reflecting its adaptive behavior.

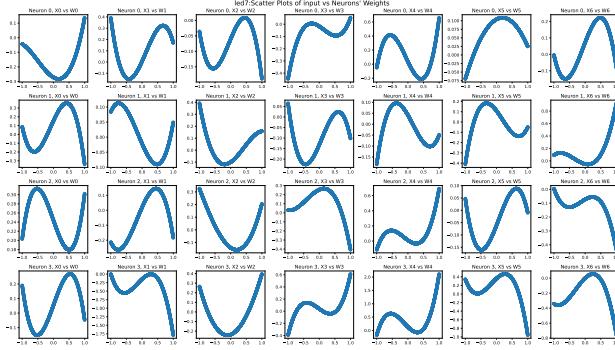


Figure 18: Adaptive weight distribution for the *led7* dataset, illustrating the responsive nature of weights to input changes within the Chebyshev adaptive model.

## E Decision Boundaries for Different Datasets

In addition to the decision boundaries shown in the main paper, we provide a comparison of decision boundaries between the MLP and Chebyshev adaptive neural networks across various datasets. We include both binary and multiclass datasets to illustrate the Chebyshev model’s superior ability to capture complex, non-linear boundaries. This is evident in the visualizations where the Chebyshev adaptive neural network significantly outperforms the MLP, particularly in datasets with intricate class distributions. The following figures illustrate these boundaries for each dataset.

These figures collectively demonstrate the Chebyshev model’s superiority in developing precise decision boundaries, especially in datasets with complex, non-linear separations between classes.

## F Samir input

For functions of multiple variables,  $f(x_1, x_2, \dots, x_d)$ , Chebyshev decomposition can be extended to higher dimensions by using a multivariate Chebyshev series expansion. This approach involves approximating  $f(x_1, x_2, \dots, x_d)$  using products of univariate Chebyshev polynomials for each variable.

Here’s a step-by-step outline for decomposing a multivariate function using Chebyshev polynomials.

### Step 1: Define the Multivariate Chebyshev Polynomial Basis

For  $d$  dimensions, the Chebyshev polynomial basis functions are products of univariate Chebyshev polynomials for each variable. For a given degree  $m_i$  in each dimension  $x_i$ , the basis function is:

$$T_{m_1, m_2, \dots, m_d}(x_1, x_2, \dots, x_d) = T_{m_1}(x_1)T_{m_2}(x_2) \cdots T_{m_d}(x_d),$$

where  $T_{m_i}(x_i)$  is the Chebyshev polynomial of degree  $m_i$  in the  $i$ -th variable. This polynomial is defined on the interval  $[-1, 1]$ .

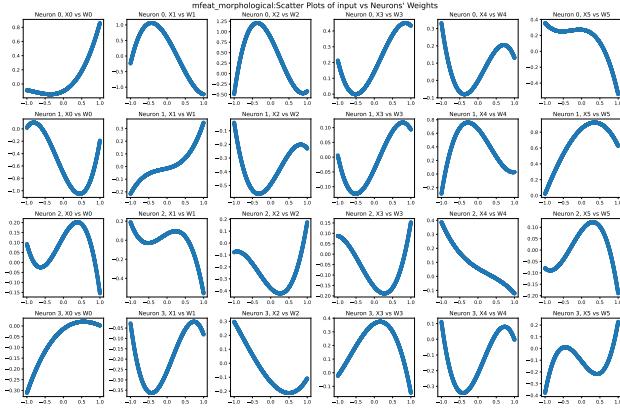


Figure 19: Adaptive weight distribution for the *mfeat\_morphological* dataset, demonstrating the input-dependent weight adaptation in the model.

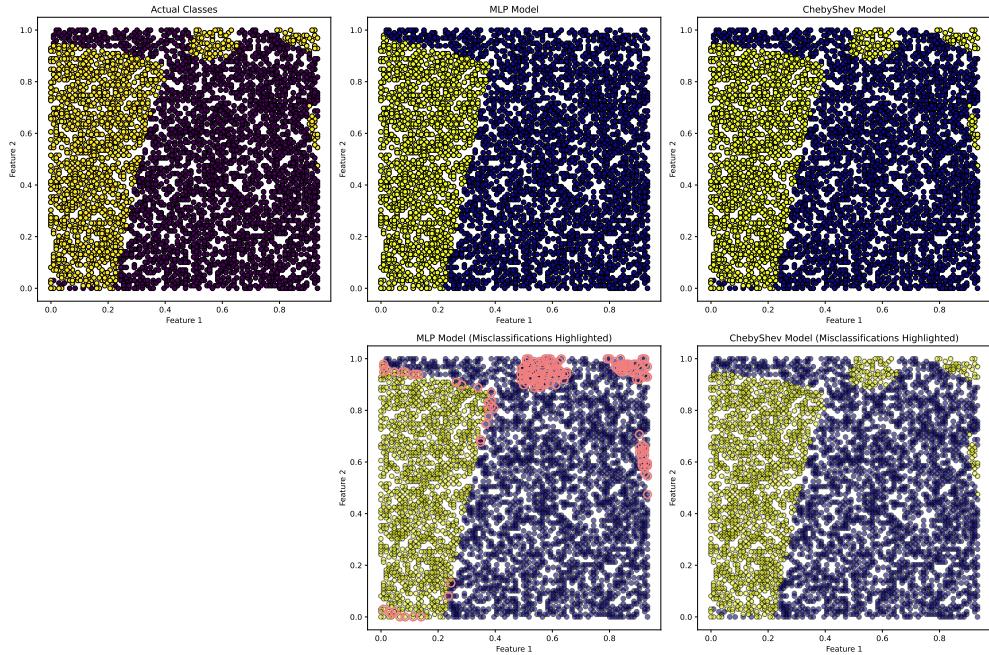


Figure 20: Comparison of decision boundaries on the "Tokyo1" dataset. The first plot in the top row shows the actual class distribution. The second plot shows the decision boundaries learned by the MLP, followed by those of the Chebyshev model. Below each plot, the misclassifications by each model are highlighted in pink.

Step 2: Define the Function Domain and Map to  $[-1, 1]^d$

Chebyshev polynomials are naturally defined on  $[-1, 1]$ . If the domain of  $f(x_1, x_2, \dots, x_d)$  is outside  $[-1, 1]^d$ , map each variable  $x_i$  from  $[a_i, b_i]$  to  $[-1, 1]$ :

$$x'_i = \frac{2x_i - (a_i + b_i)}{b_i - a_i}.$$

This transformation scales and shifts  $x_i$  to  $x'_i \in [-1, 1]$ , so  $f(x_1, x_2, \dots, x_d)$  becomes  $f(x'_1, x'_2, \dots, x'_d)$  on  $[-1, 1]^d$ .

Step 3: Express  $f(x_1, x_2, \dots, x_d)$  as a Multivariate Chebyshev Series

We approximate  $f(x_1, x_2, \dots, x_d)$  by a finite sum:

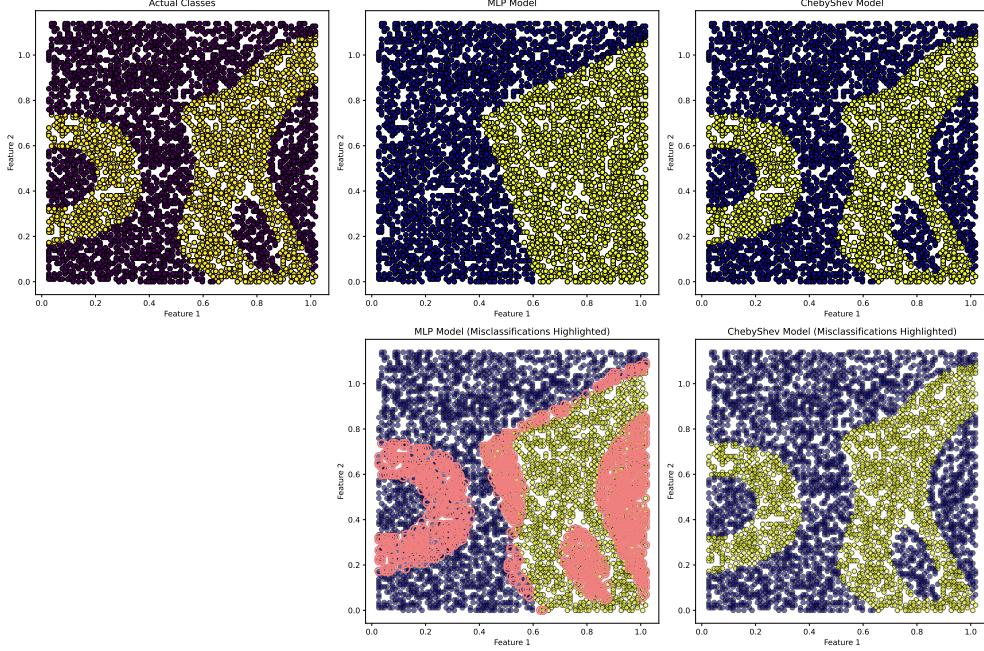


Figure 21: Comparison of decision boundaries on the "SAHeart" dataset. The top row displays the actual class distribution, followed by the decision boundaries of the MLP and Chebyshev models. The bottom row shows misclassifications for each model, with pink regions indicating errors.

$$f(x_1, x_2, \dots, x_d) \approx \sum_{m_1=0}^{M_1} \sum_{m_2=0}^{M_2} \cdots \sum_{m_d=0}^{M_d} c_{m_1, m_2, \dots, m_d} T_{m_1}(x_1) T_{m_2}(x_2) \cdots T_{m_d}(x_d),$$

where  $c_{m_1, m_2, \dots, m_d}$  are the Chebyshev coefficients.

**Step 4: Calculate the Multivariate Chebyshev Coefficients**

The Chebyshev coefficients  $c_{m_1, m_2, \dots, m_d}$  are given by:

$$c_{m_1, m_2, \dots, m_d} = \frac{2^d}{\pi^d} \int_{-1}^1 \cdots \int_{-1}^1 \frac{f(x_1, x_2, \dots, x_d) T_{m_1}(x_1) \cdots T_{m_d}(x_d)}{\sqrt{1-x_1^2} \cdots \sqrt{1-x_d^2}} dx_1 \cdots dx_d.$$

In practice, calculating these integrals directly is impractical, so they are typically approximated using Chebyshev nodes and discrete transforms.

**Step 5: Sample  $f(x_1, x_2, \dots, x_d)$  at the Chebyshev Nodes**

For each variable  $x_i$ , choose  $M_i + 1$  Chebyshev nodes:

$$x_i^{(k)} = \cos \left( \frac{(2k+1)\pi}{2(M_i+1)} \right), \quad k = 0, 1, \dots, M_i.$$

Evaluate  $f$  at each point in the resulting  $(M_1 + 1) \times (M_2 + 1) \times \cdots \times (M_d + 1)$  grid of Chebyshev nodes:

$$f_{k_1, k_2, \dots, k_d} = f \left( x_1^{(k_1)}, x_2^{(k_2)}, \dots, x_d^{(k_d)} \right).$$

**Step 6: Use a Discrete Cosine Transform (DCT) to Approximate  $c_{m_1, m_2, \dots, m_d}$**

To find the coefficients  $c_{m_1, m_2, \dots, m_d}$ , apply a multidimensional Discrete Cosine Transform (DCT) to the sampled values  $f_{k_1, k_2, \dots, k_d}$ .

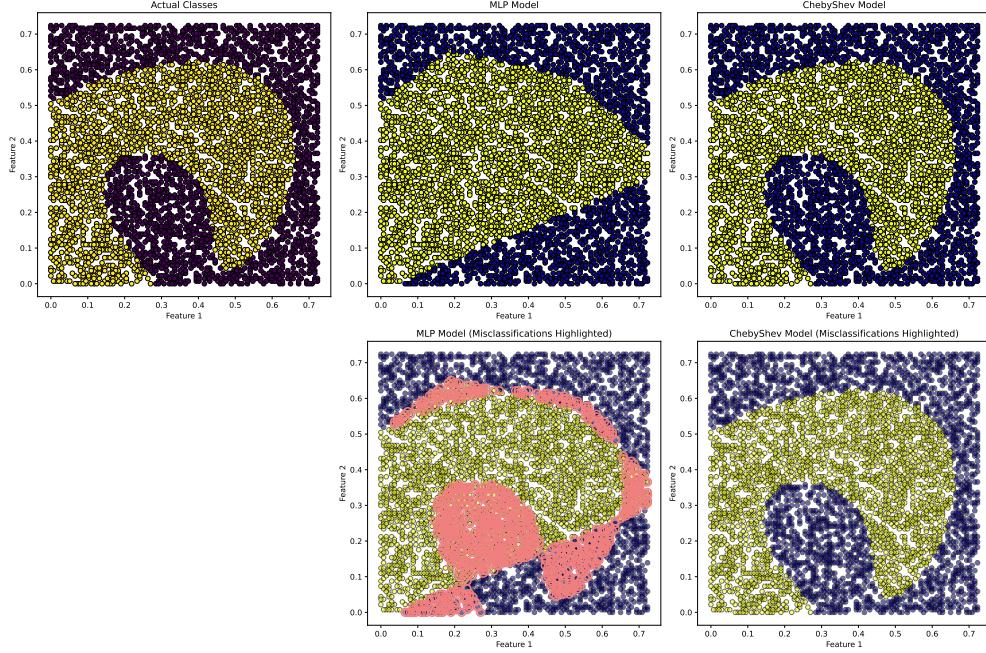


Figure 22: Decision boundaries comparison on the "Hill Valley without Noise" dataset. The first plot shows the actual classes, followed by the MLP and Chebyshev model boundaries. Misclassifications are highlighted in the bottom row, with pink areas indicating errors for each model.

Each coefficient  $c_{m_1, m_2, \dots, m_d}$  can then be obtained from the DCT, and you may need to scale some coefficients based on the indices  $m_1, m_2, \dots, m_d$  (e.g., halving for certain indices).

**Step 7: Form the Multivariate Chebyshev Polynomial Approximation**

The Chebyshev approximation for  $f(x_1, x_2, \dots, x_d)$  is then:

$$f(x_1, x_2, \dots, x_d) \approx \sum_{m_1=0}^{M_1} \sum_{m_2=0}^{M_2} \cdots \sum_{m_d=0}^{M_d} c_{m_1, m_2, \dots, m_d} T_{m_1}(x_1) T_{m_2}(x_2) \cdots T_{m_d}(x_d).$$

#### Example: 3D Case

For a function of three variables,  $f(x, y, z)$ , we have:

1. Nodes: Choose Chebyshev nodes  $x_i^{(k)}, y_j^{(l)}, z_k^{(m)}$  in each variable.
2. Sampling: Evaluate  $f(x, y, z)$  at each point on a 3D Chebyshev grid.
3. Coefficients: Apply a 3D DCT to get  $c_{m,n,p}$ .
4. Approximation:

$$f(x, y, z) \approx \sum_{m=0}^M \sum_{n=0}^N \sum_{p=0}^P c_{m,n,p} T_m(x) T_n(y) T_p(z).$$

This method generalizes for higher dimensions, allowing efficient approximation of functions with multiple variables using Chebyshev polynomials. This approach is particularly valuable in multidimensional interpolation, regression, and approximation tasks where high accuracy is required.

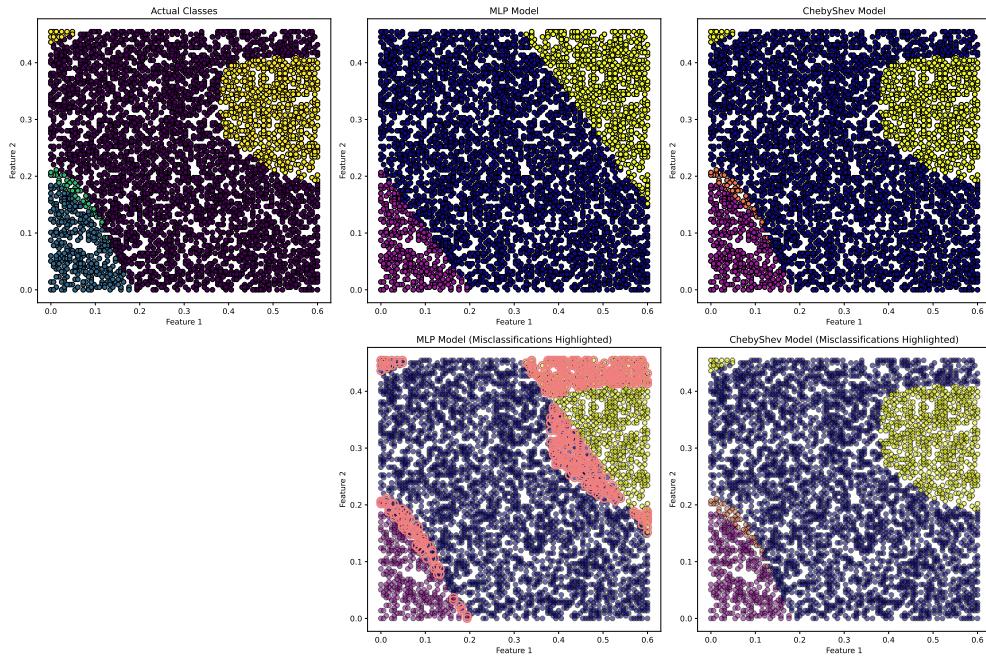


Figure 23: Decision boundaries on the "Flags" dataset. The initial plot displays the actual class distribution, followed by MLP and Chebyshev model boundaries. Misclassifications are presented in the bottom row for each model, with pink regions highlighting errors.

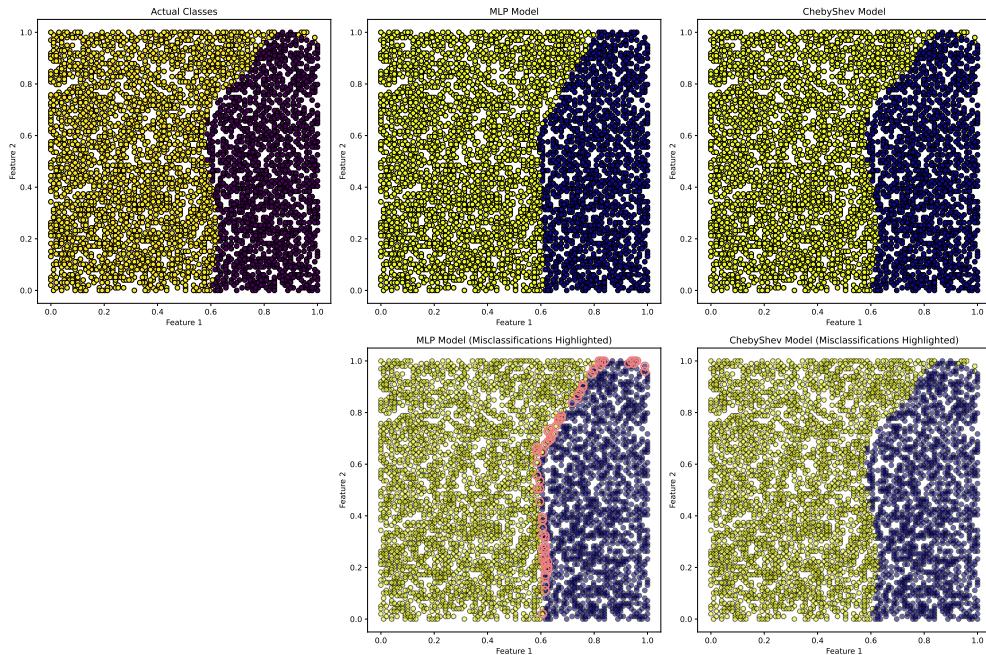


Figure 24: Decision boundary comparison on the "CMC" dataset. The top row contains the actual distribution and the decision boundaries of MLP and Chebyshev models. The misclassifications are shown in the bottom row for each model, with pink areas indicating errors.

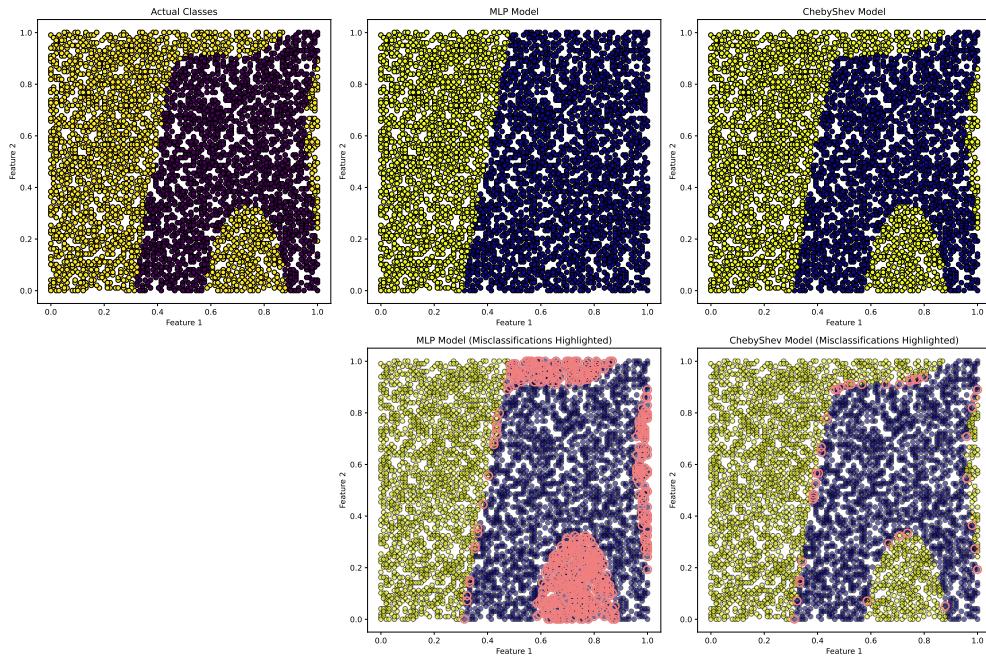


Figure 25: Comparison of decision boundaries on the "Clean2" dataset. The first plot shows the actual classes, followed by the boundaries learned by MLP and Chebyshev models. Misclassifications are highlighted in pink for each model in the bottom row.

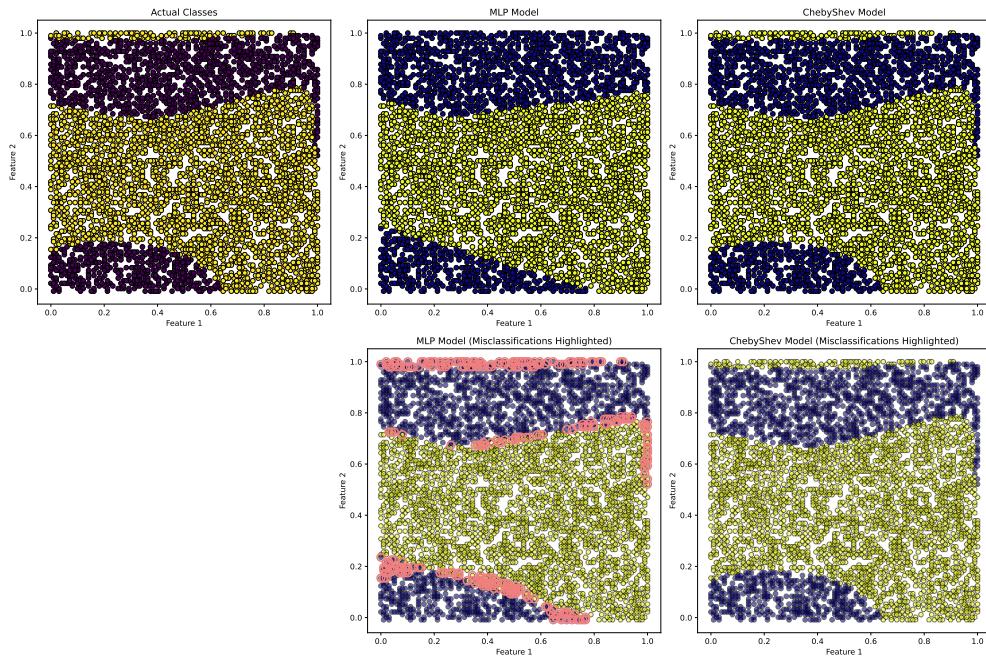


Figure 26: Decision boundaries for the "BuggyCrx" dataset. The top row displays the actual class distribution, followed by the decision boundaries of the MLP and Chebyshev models. Misclassifications are shown in pink in the bottom row for each model.