A Population Dynamics Inspired Constructive Algorithm for Growing Feedforward Neural Network Architectures

Matthew Ross

Thesis submitted to the University of Ottawa

in partial fulfillment of the requirements for the

PhD in Experimental Psychology

School of Psychology

Faculty of Social Sciences

University of Ottawa

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Sylvain Chartier, for his invaluable guidance, unwavering support, and exceptional mentorship throughout the entire duration of my thesis. His expertise, dedication, and commitment to my academic growth have been instrumental in shaping the direction of this research. I am truly grateful for all the unparalleled opportunities he has given me. He truly is a superlative supervisor.

I am also indebted to my thesis committee members, Dr. Jean-Philippe Thivierge, Dr. Pierre Payeur, Dr. Dennis Cousineau, and Dr. Michael Dawson. Their valuable insights, thoughtful suggestions, and rigorous examination of my thesis have immensely contributed to its refinement and completion. I am thankful for their time, expertise, and willingness to share their knowledge.

I would like to extend my heartfelt appreciation to my colleagues and fellow researchers who have made this academic journey enjoyable and memorable. From the enlightening and entertaining lab meetings to the unforgettable lab outings: Nareg Berberian, Albino Nikolla, Kinsey Church, Damiem and Thaddé Rollon-Mérette, André Cyr, Eric Kuebler, Marija Bolic, Melissa Johnson, Marc Dufour, and Matias Calderini.

This work would not have been possible without two extraordinary individuals who have sacrificed unfathomable amounts of time and energy, been my unwavering pillars of support, and believed in me, my parents, Douglas and Cheryl Ross. Finally, my partner Casey's understanding, and patience have been invaluable to me, especially during the challenging times when I felt overwhelmed and doubted my abilities. I am so grateful to have you by my side, and I look forward to celebrating this achievement and many more milestones together.

Contents

Table of Acronyms
(Sorted alphabetically)

ADAM:	Adaptive Moment Estimation
AI:	Artificial Intelligence
AGI:	Artificial General Intelligence
ANN:	Artificial Neural Network
BCW:	Breast Cancer Wisconsin
CC:	Cascade Correlation
CCG-DLNN	Cascade Correlation Growing-Deep Learning Neural Network
CNN:	Convolutional Neural Network
DNC:	Dynamic Node Creation
DNC-MLP:	Dynamic Node Creation-Multilayer Perceptron
EA:	Evolutionary Algorithms
EEG:	Electroencephalogram
fMLP:	Fixed Multilayer Perceptron
gMLP:	Growing Multilayer Perceptron
GP-DLNN:	Growing Pruning-Deep Learning Neural Network
GPU:	Graphics Processing Unit
MLP:	Multilayer Perceptron
MNIST:	Mixed National Institute of Standards and Technology
MSE:	Mean Squared Error
SLFN:	Single Layer Feedforward Network
SVD:	Singular Value Decomposition
UAW:	Update All Weights

List of Figures

List of Tables

Summary

The generalization ability of artificial neural networks (ANN) is highly dependent on their architectures and can be critical to solving a given problem. The current best practice uses fixed architectures determined via a trial-and-error approach. This process can be both computationally and temporally cumbersome and does not guarantee that an optimal topology will even be found. Replacing the user's role in designing topologies with methods that enable a system to manage its own growth can endow systems with adaptable learning.

Constructive algorithms offer the possibility of compact architectures as an alternative to the trial-and-error approach. This class of algorithms grows a network's topology by incrementally adding units during learning to match task complexity. However, the decision of when to add new units in constructive algorithms heavily depends on user-defined *a priori* hyperparameters, which can be task-specific. Contrary to having a user fine-tune hyperparameters that govern growth, the intrinsic population dynamics of an ANN could be used to self-govern the growing process. Theoretically, an ANN or each layer comprising the network can be viewed as a set of populations. From this perspective, a hidden layer can be considered the environment in which hidden units exist. In this work, we propose a novel, more self-governed growing algorithm inspired by population dynamics for determining near-optimal topologies of feedforward ANNs. This allows the inclusion of a carrying capacity, the maximum population of hidden units that can be sustained in a hidden layer. Including this constraint in combination with population dynamics provides a built-in mechanism for a dynamic growth rate. The proposed approach is used in parallel with direct performance feedback from the network to modulate the growth rate of the hidden layer, allowing the network to converge to smaller topologies based on the task's demands. More self-governed approaches reduce the number of finely-tuned hyperparameters required to decide when

to grow and put more control of the network's structure and representational capacities in the algorithms themselves, facilitating the emergence of inherent intelligent behaviour.

Chapter one introduces a dynamic, more self-governed growing algorithm inspired by population dynamics. Results show that compared to using fixed rules for determining hidden layer sizes; dynamic growth leads to smaller topologies than predicted while still being capable of solving the task. In chapter two, we investigate the algorithm's inherent properties to validate the more self-governed aspect. The results depict that the model's hyperparameters require less fine-tuning by the user and adhere more toward self-governance. Finally, in chapter three, we investigate the effects of growing hidden layers individually in a sequential fashion or simultaneously in a parallel fashion multilayer context. A modified version of the growing algorithm capable of growing parallel is proposed. Growing hidden layers in parallel resulted in comparable or higher performances than sequential approaches. The growing algorithm presented here offers more self-governed growth, which provides an effective general solution automatically tailored to the task.

General Introduction

The Need for Dynamic Architectures

Determining the optimal topology when designing an ANN can be critical to solving a given problem (Yao, 1999). However, there has yet to be a single agreed-upon method to accurately determine an optimal topology. This is partly due to most networks being domain-specific and having different task requirements (Thomas et al., 2016). It becomes increasingly difficult in the context of multilayered networks that ascertain their name from having one or more hidden layers comprised of hidden units. Concerning topology, if there are not enough hidden units, the network may fail to learn a given problem adequately, leading to underfitting. Underfitting is where the network fails to detect the complete signal in a dataset (Karsoliya, 2012; Naraan & Tagliarini, 2005). In contrast, if there are too many hidden units in the network, it may lead to overfitting as a result of too much flexibility and cause poor generalization (Boughrara et al., 2016; Curteanu & Cartwright, 2011; Kwok & Yeung, 1997a; Liu et al., 2002; Parekh et al., 2000).

Overfitting is one of the main problems for ANNs. It is when a model fits a dataset with almost zero error (Bilbao & Bilbao, 2017). Overfitting is problematic as there is a possibility that input data inherently contains some degree of noise. A consequence of a "perfect" fitting model will be that it also captures the noise in the data. This leads to poor generalization, or more specifically, the model "memorizes" noisy aspects of the data rather than "learning" to generalize (Bilbao & Bilbao, 2017; Thivierge et al., 2003; Ying, 2019). Another way to visualize the concept of overfitting is to consider curve fitting, in general, using polynomial functions. There is no limit to the order of a polynomial function that can be applied to a dataset. As such, any n^{th}-order polynomial function could be used to fit all points in a dataset perfectly. However, a trade-off

becomes apparent. Higher-order polynomials may fit one particular dataset perfectly but fail to generalize to a similar novel dataset (Bilbao & Bilbao, 2017; Ma & Khorasani, 2004a).

A parallel exists in machine learning, known as the *bias-variance* trade-off, which describes the trade-off between accuracy and consistency (Neal et al., 2018; Ying, 2019). The *bias* refers to the associated error when there is a mismatch between the model and the underlying data distribution, with a high bias causing underfitting. In contrast, the *variance* measures sensitivity to fluctuations in the training set (Yang et al., 2020), with a high variance causing overfitting. There are many ways to try to avoid overfitting, such as validation techniques, early stopping, regularization, and noise reduction (Bilbao & Bilbao, 2017; Ying, 2019). However, these techniques do not address one of the primary causes of overfitting, using a sub-optimal network size (Bilbao & Bilbao, 2017).

Determining Architectures

The generalization ability of ANNs is highly dependent on their architectures (Islam et al., 2009a). Therefore, finding an optimal topology large enough to learn an underlying function in the data to avoid underfitting while still being small enough to generalize effectively and avoid overfitting is highly desirable. For the bias-variance trade-off, this is akin to finding the lowest possible balance between bias and variance. For many years rules of thumb were used to determine the number of units that should be in the hidden layers. Some of these rules were:

- The number of hidden units should equal as many needed to capture 70-90% of the variance of the input data set (Boger & Guterman, 1997).
- The size of the hidden layer should be somewhere between the input layer size and the output layer size (Blum, 1992).

- The size of the hidden layer should never be more than twice as large as the input layer (Berry & Linoff, 1997).

However, these rules of thumb only generalize well in some cases. This is imparted mainly to most of these rules, not considering the size and complexity of the training data, which highly varies between tasks (Jinchuan & Xinzhe, 2008; Karsoliya, 2012; Thomas et al., 2016; Xu & Chen, 2008). Additionally, Fernandes and Lona (2005) proposed general guidelines for setting topologies *a priori* that are divided into three classes of ANNs. By their definition, Class I ANNs have more input features than outputs. They recommend for this class using a single layer with between 8-20 hidden units for increased accuracy and reduced training times, with the possibility of adding a second layer if there are more than four outputs that are dependent. Class II ANNs have the same number of input features as outputs. For one hidden layer, they recommend 20-40 hidden units; for two layers, they recommend 13-20 hidden units for the first layer and 18-25 hidden units for the second layer. Finally, Class III ANN have more outputs than input features; they recommend two to three hidden layers with 10-20, 15-20, and 15-20 hidden units for each layer, respectively. These guidelines will likely give good predictions but not an optimal ANN topology (Fernandes & Lona, 2005). However, it is essential to note that while these recommendations might guide a user when setting ANN topologies, they are derived from experience without reference to documented systematic experimental testing. Even with guidelines, determining an appropriate topology *a priori* that will converge and avoid overfitting is arduous (Boughrara et al., 2016).

Currently, ANN topologies are primarily fixed and manually set by the user *a priori,* where only weight connections are permitted to change during learning (Zemouri et al., 2018). However, many methods have been developed to find the optimal topology of ANNs. They can be divided

into four general categories: trial-and-error, statistical and empirical, evolutionary, and non-evolutionary approaches.

Trial-and-error Approach

The current best practice uses fixed architectures determined via a trial-and-error approach. This approach involves training and evaluating several different ANNs with different topologies and selecting the "best" for the given problem (Azimi-Sadjadi et al., 1993; Curteanu & Cartwright, 2011; Do Carmo Nicoletti et al., 2009; Fernandes & Lona, 2005). Typically, the selection is made by comparing each network's prediction error on the given task, with smaller prediction errors as indicators of potentially suitable topologies (Curteanu & Cartwright, 2011; Fernandes & Lona, 2005). However, this method has reliability issues as the topologies of the ANNs tested are usually not guided by anything other than the user's prior knowledge and intuition. Consequently, this process can be both computationally and temporally cumbersome and does not guarantee that an optimal or even near-optimal topology will even be found (Hernández-Espinosa & Fernández-Redondo, 2002; Khaw et al., 1995; Packianather et al., 2000; Parekh et al., 2000; Zemouri et al., 2018). Despite these drawbacks, this approach is still the most adopted as it is computationally and conceptually the most straightforward (Curteanu & Cartwright, 2011).

Statistical and Empirical Methods

Another avenue is to utilize statistical and empirical methods for determining ANN topologies. One approach is to use singular value decomposition (SVD). Psichogios and Ungar (1994) applied SVD to remove redundant hidden units as a post-learning method automatically. Similarly, Teoh et al. (2006) used SVD post-learning to estimate the number of hidden units and add or remove hidden units as needed in a single-layer feedforward network (SLFN). Alternatively, Cai et al. (2019) applied SVD directly to the training data to estimate the optimal

number of hidden nodes in a SLFN. Another statistical approach proposed by Rivals and Personnaz (2000) used least squares estimation to estimate the number of hidden units for a group of candidate models that were then filtered using Fisher tests to find the most optimal topology. Finally, a heavily statistics-based approach, known as Taguchi design of experiments (DoE), offers a systematic approach to determining optimal network structures and parameters for a given problem (for a review, see Curteanu & Cartwright, 2011). Taguchi DoE is typically used in product manufacturing as a method of quality control during design stages but has been extended to neural network optimization (Khaw et al., 1995; Packianather et al., 2000). In this method, measured network responses, such as accuracy and training time, are considered dependent variables. Independent variables or controllable variables, include things such as the number of hidden units per layer or learning rate. Finally, uncontrollable variables represent noise factors that impact network performance such as the random initialization of weight connections. This allows the user to examine the main effects and interactions that a controllable variable or combination of variables has on network performance for an experiment via a signal-to-noise ratio (Khaw et al., 1995; Packianather et al., 2000). To add to the level of optimization, orthogonal arrays are used to vary parameters allowing multiple experiments to be run in parallel (Packianather et al., 2000). An analysis of variance (ANOVA) can then be performed after all the experiments are complete to determine the best combination of variables (Packianather et al., 2000). While this method can provide optimal hidden layer sizes while considering other network parameters simultaneously, it inherently requires an enormous computational burden that must be performed *a priori*.

Evolutionary Approaches

One widely used ANN topology optimization technique utilizes an evolutionary approach. Inspired by Darwinism, this approach simulates natural selection and the evolutionary

process to stochastically search for the optimal ANN topology (Safi & Bouroumi, 2014). These processes are simulated using population-based algorithms known as evolutionary algorithms (EA). These algorithms include evolution strategies, evolutionary programming, and genetic algorithms (Branke, 1995; Yao, 1999). Broadly speaking, the overall process of each variant is similar. Each begins with a population of possible candidate solutions and undergoes an iterative, evolutionary process. New generations of candidate solutions are created through genetic operators, such as mutation and reproduction (Ding et al., 2013; Safi & Bouroumi, 2014). This evolutionary process is made possible by varying specialized encoded parameter details like the number of nodes into data structures that mimic chromosomes. The performances of each candidate can then be calculated and compared using a fitness function that assigns a corresponding fitness score. Fitness scores dictate the likelihood that the particular candidate will contribute to future generations of candidates (Curteanu & Cartwright, 2011).

EAs offer an attractive adaptative framework whereby topologies can be dynamically altered without direct user intervention. This allows near-optimal ANN architectures to be found automatically (Yao, 1999). A key feature of EAs is that they use population-to-population searches. This gives EAs a global search ability making them ideal for optimization and fitting an ANN's topology to the task (Ding et al., 2011, 2013). EAs have been applied to a myriad of topology optimizations for ANNs (for reviews, see Castillo et al., 2000; Curteanu & Cartwright, 2011; Ding et al., 2013; Yao, 1993).

For instance, a genetic algorithm was applied to determine the optimal number of hidden units in a multilayer perceptron (MLP) with a single hidden layer when identifying a 3^{rd}-order nonlinear system (Arena et al., 1992). In this scenario, the genetic encoding of the chromosome consisted of a binary string that indicated the presence or absence of a hidden unit in the network,

with a pre-defined maximum of 16 units. The genetic algorithm began with a population of neural networks with random binary strings creating different-sized hidden layers. After 2500 trials, each network's associated error value was used to measure that network's fitness. A higher error value resulted in a lower fitness score, and inversely a low error resulted in a high fitness score. Networks with higher fitness scores and high accuracy during testing were more likely to propagate their genetic encoding to the next generation through iterative reproduction. This resulted in a new population of networks with a new number of hidden units that had an increased probability of solving the task at hand. While a maximum of 16 hidden units was possible, after ten generations, it was determined that the optimal number of hidden units was 14 for a single hidden layer MLP for this particular task. Another attempt to find the optimal hidden layer size in a MLP combined a genetic algorithm with standard back-propagation (Castillo et al., 2000). The G-prop algorithm used a genetic algorithm to globally search and optimize the number of hidden units and initial weight settings. Then, back-propagation was used to search locally. The procedure is similar to the example mentioned above. An initial population of MLPs with random numbers of hidden units competed, and their fitness was evaluated based on the number of correct classifications on a breast cancer dataset test set and the number of hidden units. Those given a high fitness score were more likely to contribute to the next generation through various genetic operators, such as adding and eliminating hidden units. Overall, G-prop reached a better generalization to novel test sets than standard back-propagation (Castillo et al., 2000).

Inherent limitations accompanying an evolutionary approach have sparked debate surrounding their effectiveness for ANN topology optimization. Firstly, these methods can be very temporally demanding (Islam & Murase, 2001; Kwok & Yeung, 1997a). For example, a larger population will require more time to train. Additionally, the duration of the evolutionary process

is highly dependent on the number of generations set. Secondly, EAs typically require many hyperparameters to be defined that can impact their effectiveness (Islam & Murase, 2001). When approaching ANN topology optimization, an essential factor is determining the amount of information that should be encoded *a priori* into a chromosome. Encoding is divided into direct or indirect encoding (Ding et al., 2013). With direct encoding, all architecture details are encoded, like every connection to a unit (Yao, 1999). Direct encoding leads to very long genetic encoding, making it only feasible for small ANNs (Branke, 1995). The user typically sets the maximum possible topology to ensure networks do not grow outbound. However, this limits the search space of possible solutions and can exclude more optimal solutions (Branke, 1995). In contrast, only the most critical hyperparameters are encoded with indirect encoding, such as the number of units (Yao, 1999). However, the choice of information to encode by the user can also bias the search to exclude more optimal solutions (Branke, 1995). Finally, while the stochastic nature of EAs allows for a broader search of the search space, it also limits their explanatory power.

Non-Evolutionary Approaches

Non-evolutionary approaches can be divided into destructive, constructive, or hybrid methods for topology optimization in ANNs. The difference between these methodologies is straightforward. Destructive methods begin with a large initial network where units/connections are systematically removed. Inversely, constructive methods begin with a small initial network and systematically add units/connections. Hybrid methods combine constructive and destructive methods. These approaches use a destructive method either after the constructive method or simultaneously. Many variants exist within each of these methods; however, the underlying principle of incremental learning is universal. As the network learns to solve a given problem,

units/connections are incrementally removed or added to obtain the optimal topology that will yield the best performance.

Destructive Methods.

Systematically removing units or weight connections is known as network trimming or pruning. Pruning algorithms start with larger-than-necessary network topologies and trim or "prune" irrelevant and redundant connections/units. Commencing with a large network topology allows rapid learning with less sensitivity to initial conditions and increased odds of avoiding local minima (Augasta & Kathirvalavakumar, 2013; Reed, 1993). By pruning the network, these algorithms also benefit from smaller network topologies that avoid overfitting the data and obtain better generalization (Augasta & Kathirvalavakumar, 2013). The pruning process initially causes a decrease in the network's accuracy. As a result, after pruning the network, training is continued to mitigate this loss in a process known as fine-tuning (Blalock et al., 2020). The alternation between pruning and fine-tuning processes is repeated to reduce the network topology while minimizing network accuracy loss. Models that have been pruned obtain better generalizations with smaller topologies than their non-pruned counterparts (Augasta & Kathirvalavakumar, 2013; Blalock et al., 2020). Many pruning techniques have been proposed and fall into three broad methodological categories: sensitivity-based, penalty-term-based, and hybrid methods (for a survey of pruning methods, see Augasta & Kathirvalavakumar, 2013).

The most well-known pruning algorithms are sensitivity based. These algorithms calculate an estimate of the sensitivity of the cost function (Reed, 1993) to the removal of a unit/connection. The unit/connection with the most negligible effect on the cost function is subsequently pruned. The Optimal Brain Damage (Y. LeCun et al., 1990) and Optimal Brain Surgeon (Hassibi & Stork, 1993) methods are the most popular. The Optimal Brain Damage method starts with a larger

network topology than is needed and trains until a solution is obtained. Next, a measure of saliency that a unit/connection has in relation to the cost function is calculated by estimating the second derivative (LeCun et al., 1990). The units/connections with low saliency are pruned, and training is resumed until a solution is obtained. This process is repeated until a minimal topology is obtained that can still converge to a solution. Similarly, the Optimal Brain Surgeon method utilizes second derivatives to compute a measure of saliency. However, it avoids pruning the wrong weight connections by not assuming a diagonal Hessian matrix. The Hessian matrix provides information about the second-order derivatives of the loss function with respect to changes in the weight connections, with low valued elements indicating connections with low importance(Hassibi & Stork, 1993). Many pruning algorithms have been based on these two methods (Augasta & Kathirvalavakumar, 2013). Alternatively, a less popular approach is to estimate the sensitivity based on the output function (Engelbrecht & Cloete, 1996). For instance, Zeng and Yeung (2006) developed a sensitivity-based method that utilizes a novel sensitivity measure based on a hidden unit's output response to input deviations for pruning a MLP. Furthermore, they introduced a measure of a hidden unit's relevance and overall contribution to the network. With their novel technique, they could prune the least relevant hidden units to obtain a minimal MLP topology.

Penalty-term-based pruning methods modify the cost function directly by adding a penalty term that drives small or irrelevant weights toward zero (Augasta & Kathirvalavakumar, 2013; Reed, 1993). Setiono (1997) proposed a penalty function and magnitude-based weight elimination combination. The first part discouraged unnecessary connections, and the second part prevented weights from taking unnecessary large values. He was able to obtain smaller topologies in a SLFN successfully.

Constructive Methods.

Starting with a small network topology and incrementally adding connections, units, or layers is known as growing or constructing the network. This class of algorithms grows a network's topology to match task complexity while simultaneously learning (Bertini & Do Carmo Nicoletti, 2009; Boughrara et al., 2016). Constructive algorithms have been applied successfully to many recognition problems, including facial expression recognition (Boughrara et al., 2016; Ma & Khorasani, 2004b) and breast-cancer diagnosis (Zemouri et al., 2018), and many classification problems (for reviews and examples see, Bertini & Do Carmo Nicoletti, 2009; Do Carmo Nicoletti et al., 2009; Li et al., 2010; Parekh et al., 2000).

Advantages of Constructive Algorithms.

Using constructive algorithms has several advantages over pruning algorithms, as outlined by Kwok and Yeung (1997a, 1997b) in their survey on constructive algorithms for feed-forward ANNs. First, the initialization of the constructive approach is simple, using only a single unit. Whereas with the pruning approach, a guess must be made on how large the initial network should be. If the initial network is set too large, it could result in longer training times, as there are more redundant units and weights, with the possibility of the network converging on a much larger than-needed topology and possibly overfitting the data. If the initial network is too small, the network may not have an appropriate topology to adequately learn a given problem and lead to underfitting.

Second, the constructive approach is more computationally economical as it builds smaller network topologies first as a direct result of using additive incremental learning. The resulting constructed network topologies are directly related to the complexity of the given problem and the performance requirements (Ma & Khorasani, 2003). While with pruning, the majority of training time is spent on network topologies that are larger than required. This leads to potential wasted

computational effort and time spent removing unnecessary units and weight connections (Ma & Khorasani, 2003).

Third, constructive algorithms are more efficient in forward computations as they have fewer weights and other parameters to update initially, require less data for good generalization, are more easily described, and are more likely to find smaller topology solutions (Kwok & Yeung 1997a, 1997b). Additionally, several parameters used in pruning algorithms are task-specific and require careful specification to obtain an appropriate topology and good performance (Sharma & Chandra, 2010a). This task-specific parameter setting limits these algorithms' applicability to real-world applications.

Fourth, as previously explained, many pruning algorithms typically estimate the sensitivity of the error function when an element is removed; this estimate is for computational efficiency (Reed, 1993). However, only using estimations can introduce significant errors to the network (Kwok & Yeung 1997a, 1997b).

Finally, aside from Kwok and Yeung's (1997a, 1997b) survey, general advantages of constructive algorithms have been identified. Since constructive algorithms tend not to have an upper limit on the size that the network can grow, they can search the entire space of possible topologies (Curteanu & Cartwright, 2011). This endows these networks with almost guaranteed success when learning a given problem (Śmieja, 1993). Despite not having an upper size limit, these methods tend to grow near-minimal topologies required to solve given problems (Śmieja, 1993).

Taxonomy of Constructive Algorithms.

To differentiate between the different constructive algorithms in feedforward neural networks, Kwok and Yeung (1997a, 1997b) proposed a taxonomy based on: the state transition mapping, the training scheme used, and the given network architecture (see Figure 1).

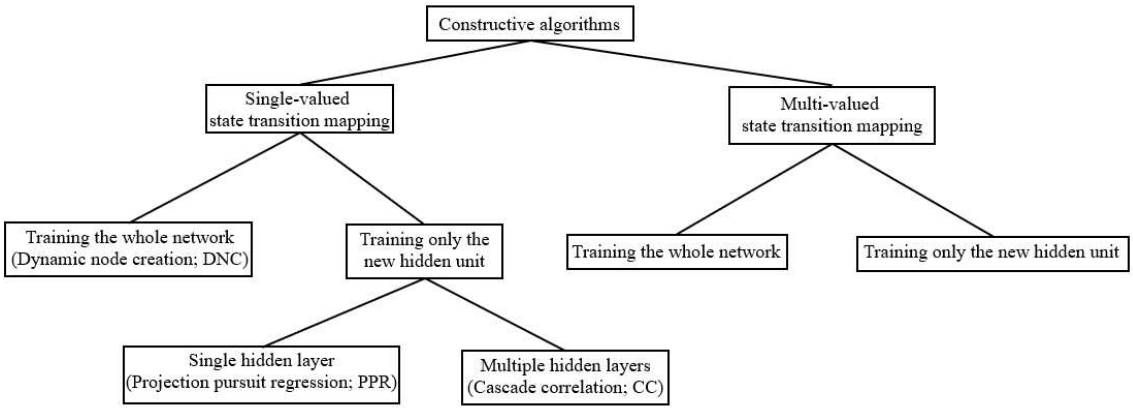

Figure 1. A taxonomy of constructive algorithms. Adapted and modified from Kwok and Yeung (1997a).

They consider the problem of constructing an ANN's architecture during learning as a search in the state space of all possible architectures (Aran et al., 2009; Kwok & Yeung, 1997a). Concerning constructive algorithms, state transition mapping maps the current state to the next state $\Delta : S_1 \rightarrow S_2$. The search between states is guided by conservation of the units and connections from one state to the next, a fundamental property of constructive algorithms, and by defining the Δ value. When Δ is single-valued, there is only one possible next state; if Δ is multivalued, there are several possible candidate states. The critical difference is connectivity. When Δ is single-valued, the number of incoming connections to each hidden unit is fixed and from the same source. Conversely, when Δ is multivalued, the number of incoming connections to each hidden unit is still fixed, but the source of the incoming connections can vary. The incoming connections can come from any combination of input or hidden units in the network. This gives rise to many possible candidate networks and, thus, multiple possible next states. As such, the taxonomy first

divides constructive algorithms into either single or multivalued state transition mapping (Kwok & Yeung, 1997a).

The categorization of constructive algorithms with respect to the training scheme is very straightforward. After adding a new hidden unit, the whole network is wholly trained, or only the new unit is trained. Originally, Kwok and Yeung (1997a) also included training with memorization as a possibility. However, they noted that while memorization of input patterns reduces computations during learning, the resulting growth is linearly related to the size of the training data set and results in large topologies. As such, this category of training schemes is not considered here, as the focus is on obtaining minimal topologies.

Each training scheme has its advantages and disadvantages. An advantage of retraining the whole network is its implementation simplicity (Islam et al., 2009a). One of the most influential constructive algorithms that use this learning scheme is Dynamic Node Creation (DNC). Proposed by Ash (1989), this model adds hidden units if the error curve flattens below an *a priori* set trigger slope value within an *a priori* determined time window. After each addition, the entire network is retrained and repeated until a minimum error is achieved. This simple algorithm takes advantage of the universal approximation capabilities of SLFN ANNS (Kwok & Yeung, 1997a) to obtain minimal topologies. Many variants inspired by DNC have been proposed (Azimi-Sadjadi et al., 1993; Bartlett, 1994; Hirose et al., 1991; Sharma & Chandra, 2010b; Zhang, 1994), however, the simplicity of this training scheme comes at a cost. The added computational load of retraining everything can lead to scale-up problems if the network grows very large (Kwok & Yeung, 1997a). Furthermore, each hidden unit sees a constantly changing environment, known as the moving target problem (Fahlman & Lebiere, 1990). Conversely, training only the newly added hidden unit involves fixing all connections into every other hidden unit, a process known as “freezing,” and

only training the connections of the new hidden unit. This training scheme dramatically reduces the computational load, achieves faster convergence, and avoids the "moving target problem" (Aran et al., 2009; Islam et al., 2009a; Kwok & Yeung, 1997a). One of the most influential constructive algorithms that follow this learning scheme is Cascade Correlation (CC; Fahlman & Lebiere, 1990). This model adds a hidden unit when there is no significant error reduction over a predetermined number of training epochs, known as a patience parameter, defined by the user *a priori.* When it is decided that a new hidden unit is needed, a new candidate unit is recruited. The candidate's input-side connection weights are then trained to maximize the covariance between residual network error and the candidate's output. The candidate with the highest covariance measure is then installed in the network. Aside from establishing connections to all inputs and outputs, it is also connected to every existing hidden unit in the network, creating a single hidden unit layer. Once the new unit is installed, its input connection weights are "frozen" to preserve the hidden unit's feature detection abilities (Thivierge et al., 2003). While the advantages make the choice of training scheme seem apparent, it also has several key disadvantages. This training scheme may result in ANNs with more hidden units than needed (Curteanu & Cartwright, 2011; Islam et al., 2009a) and fail to obtain optimal weights for the network as a whole (Kwok & Yeung, 1997a). By "freezing" the existing weights, the weight space is confined, limiting possible solutions (Aran et al., 2009), with new units usually being inefficient feature detectors leading to more units added and possible overfitting (Curteanu & Cartwright, 2011). There is still much debate surrounding which training scheme should be used, thus creating the second tier of the constructive algorithm taxonomy.

Finally, the taxonomy is broken down by the network architecture, which consists of single hidden layer networks vs. multiple hidden layers. The use of a single hidden layer or multiple

hidden layers is highly dependent on the given task (Safi & Bouroumi, 2014). Why is the taxonomy important? While it might seem obvious, to validate a constructive algorithm it should be compared to other growing methods. By understanding how constructive algorithms are categorized, appropriate and meaningful comparisons can be made.

Challenges and Limitations of Constructive Algorithms.

There are several limitations and challenges associated with constructive algorithms. One of the most acknowledged limitations is the possibility that the ANN will become trapped in a local minimum resulting in longer training times. This is primarily due to the initial small network being more susceptible to getting trapped (Augasta & Kathirvalavakumar, 2013; Yao, 1999). Adding new hidden units changes the shape of the weight space helping the training algorithm escape a local minimum or alternatively using techniques such as weight scaling can help (Liu et al., 2002). Another well-known limitation of constructive algorithms is their difficulties with noisy data, which can cause the network to grow too large and overfit the data (Aran & Alpaydın, 2003; José Luis Subirats et al., 2008). As with all ANNs, filtering or removing noisy inputs can improve learning speeds and reduce overfitting. Finally, it has been argued that constructive algorithms rarely result in a minimal or optimal topology but rather give a sub-minimal/sub-optimal topology (Kwok & Yeung, 1997a, 1997b; Ma & Khorasani, 2003, 2004b).

Several significant challenges were outlined by Sharma and Chandra (2010) that should be considered when employing constructive algorithms. These are: how to connect a newly added hidden unit, the activation function(s), the training scheme (re-train the whole network vs. "freezing"), which optimization technique should be used, and when to stop adding hidden units. While these are very important, one fundamental challenge is seldomly considered; *when should a new hidden unit be added*? This is because the methods for deciding when to add a new unit are

standard across many constructive algorithms (Sharma & Chandra, 2010b). The most common is when the error drops below or stops changing by a pre-set amount, usually over a given number of trials, a new hidden unit is added, often referred to as a "patience" parameter. Many approaches use this method, including the popular CC (Fahlman & Lebiere, 1990) and DNC (Ash, 1989). A similar technique can be found in Projection Pursuit Regression (Friedman & Stuetzle, 1981), where the user compares a criterion of fit for a smooth representation of residuals to a manually set threshold value. This adds an additional hyperparameter(s) that requires *a priori* setting and may require fine-tuning to meet a given task.

Hybrid Methods.

The combination of growing and pruning methods is known as a hybrid approach. Hybrid algorithms try to strike a balance between the network's complexity and computational resources and take advantage of both methods to obtain a parsimonious near-optimal topology. For example, Thivierge et al. (2003) introduced a dual-phase technique whereby the network grew by CC and pruned connection weights using the optimal brain damage technique during the input and output growing phases. Zemouri et al. (2020) proposed a growing and pruning algorithm for deep learning neural networks (GP-DLNN). During the growing phase, fully connected hidden units were added based on a performance degradation threshold. Post-growing, if an excess of units were statistically determined, the network would undergo a pruning phase which consisted of applying the iterative pruning algorithm and the statistical stepwise pruning algorithm to remove unnecessary units/connections. They showed that when both growing and pruning was used, the network had increased classification accuracy than when just growing alone was applied. Dai et al. (2019) proposed a more neuroscience approach where adding units/connections was based on the degree of correlation of pre and post-connections for growing in a convolutional neural network (CNN).

In the post-learning pruning phase, units/connections were pruned based on the magnitude of the weights as determined by a predefined threshold (for other examples of hybrid approaches, see Abd et al., 2021a; Ashfahani & Pratama, 2019; Han et al., 2017; Narasimha et al., 2008b; Pérez-Sánchez et al., 2018a; Pratama et al., 2020; Puma-Villanueva et al., 2012).

A More Self-Governed Growing Approach

As discussed, determining an appropriate number of hidden units *a priori* for a given task is very difficult. The reliance on artificially intelligent systems built "brick by brick" by a human designer has significantly limited their applicability to only particular tasks (Thórisson, 2012). To properly match task complexity, ANN topologies should be updated during learning (Zemouri et al., 2018). Many methods have been outlined in the previous sections that can effectively update ANN topologies during learning. The ability of these methods to add units/connections during learning represents a class of universal learners (Baum, 1989). However, the decision of when to add new units in constructive algorithms heavily depends on user-defined *a priori* hyperparameters. Replacing the user's role in designing ANN topologies with methods that enable a system to manage its own growth can endow systems with adaptable learning that can apply to many tasks across different environments, an artificial general intelligence (Thórisson, 2012).

Two self-growing learning algorithms were identified in the literature. The first algorithm based on heuristic terminal Attractor backpropagation, developed by Wang and Hsu (1991), uses a time-varying gains value incorporated into the weight update rule as a criterion to determine the number of hidden units in a single hidden layer MLP. The idea is that the next state selected during the search for topology should increase the system's energy. To achieve this, new hidden units need to be added if the error function does not converge to the minimum error. When the error function flattens, the value of the time-varying gains will become large. If this value surpasses a

manually set threshold, a new hidden unit can be added depending on the probability of adding a new unit. After growing, a hidden unit removal rule was incorporated to remove redundant units. The results showed that the proposed algorithm effectively automatically determined the appropriate number of hidden units. However, Wang and Hsu (1991) noted that without the hidden unit removal rule, the algorithm could grow to larger sizes with more redundant units. They suggest that high set threshold values can reduce the likelihood of this occurrence but can result in longer training times. While this approach is self-growing, the dependence on manually setting an appropriate threshold value can impact the size of the topology grown and cause overfitting that limits generalization. In the second, Huang et al. (2006) prove that SLFNs with randomly generated nodes are universal approximators. They then move from theory to application by creating a fully automatic SLFN. In this context, only the target minimum error and the maximum number of hidden nodes need to be defined by the user *a priori*. No other hyperparameters require manual tuning. New nodes are randomly added during learning if the maximum number of nodes and the minimum error has not been reached. Their results showed that a constructive algorithm based on randomly adding nodes results in good generalization performance on several classification datasets. They then extended their algorithm to construct a two-layered feedforward network. First, the training sample space is divided into subregions. Several SLFNs with additive nodes are trained on one subregion and share nodes in the second hidden layer. These results show that a two-layered feedforward network achieved better generalizations and learned faster. While this approach is self-governed, adding nodes in a stochastic way limits its explanatory power and can result in larger than necessary topologies resulting in sub-optimal topologies.

Population dynamics refers to a branch of ecology that involves quantifying and modeling the growth or decline in the size of a biological population and the investigation of the forces that

are responsible for fluctuations in size (Juliano, 2007). Theoretically, an ANN or each layer comprising the network can be viewed as a set of populations (Ross et al., 2020). From this perspective, a hidden layer can be considered the environment in which neuronal units exist. Contrary to having a user fine-tune hyperparameters that govern growth, the intrinsic population dynamics of an ANN could be used to self-govern the growing process. It is important to distinguish this from other methods, such as those outlined in the evolutionary approaches, that use a population-based approach. In these methods, population-based refers to training a population of stochastically generated candidate networks where the best-performing candidate is selected. Conversely, in the context of the proposed usage of population dynamics, only a single network is trained with a population(s) contained within. As such, to address the issue of self-growth, we propose a novel more self-governed growing algorithm that is inspired by population dynamics.

Overview of Chapters

The population dynamics inspired growing algorithm's properties and applications are investigated across three chapters. Chapter 1 introduces a novel dynamic more self-governed growing algorithm inspired by population dynamics. The algorithm is implemented in a single hidden layer multilayer perceptron (gMLP), whereby the dynamics of the algorithm are explained and tested on the classic *n*-Bit Parity problem compared to a fixed hidden layer network (fMLP). Chapter 2 investigates the algorithm's inherent properties to validate the more self-governed claim. To achieve this, the algorithm is implemented in the gMLP and tasked with classifying the complete MNIST data set of handwritten digits. The model is further validated through comparisons to an fMLP and another constructive algorithm with the same learning scheme, Dynamic Node Creation (DNC-MLP). Chapter 3 investigates the effects of growing sequentially

or in parallel in a multilayer context. A modified version of the growing algorithm capable of growing in a parallel fashion is proposed. Using benchmark classification tasks, the parallel version of the algorithm is compared to two sequential approaches in a multilayer context: the original sequential growing algorithm inspired by population dynamics and DNC.

Chapter 1

Should I Stay or Should I Grow? A Dynamic Self-Governed Growth for Determining Hidden Layer Size in a Multilayer Perceptron

Abstract

A novel dynamic self-governed growth algorithm inspired by population dynamics is introduced in a MLP. This allows the inclusion of a carrying capacity, the maximum population of hidden units that can be sustained in a single hidden layer. Including this constraint in combination with population dynamics provides a built-in mechanism for a dynamic growth rate. The proposed approach is used in parallel with direct performance feedback from the network to modulate the growth rate of the hidden layer. This algorithm incrementally adds units to the hidden layer until the complexity of the task no longer requires further addition. The MLP is extended with the growing algorithm, and its adaptability is tested by subjecting the network to increasing levels of task complexity for the *n*-bit problem. Using fixed rules that dictate both the size of a fixed layer MLP (fMLP) and the upper bound carrying capacity of the growing MLP (gMLP), the resulting topologies are directly compared on the *n*-bit problem. In short, the results suggest that even if a fixed rule sets an upper boundary of the carrying capacity, the growing algorithm can converge to less than the predicted number of units required for solving the given task, with the majority of trials growing to the same number of hidden units regardless of the rule used. This effect is consistent across the specified rules and levels of task complexity for the *n*-bit problem.

In multilayered ANNs, deciding the number of hidden units within the architecture can be crucial for adequately learning and solving a given task. Using too few units can result in underfitting, whereby the network may not adequately capture the input data, preventing proper learning or generalization. In contrast, too many units can lead to overfitting, whereby the input data is not enough to train all the hidden units properly and can therefore increase training time (Gaurang & Panchal, 2011; Kwok & Yeung, 1997a; Liu et al., 2002; Parekh et al., 2000). Unfortunately, there is no methodological consensus for determining the number of hidden units in a hidden layer. There are several rule-of-thumb methods (Gaurang & Panchal, 2011); however, typically, this number is found by extensive, time-consuming trial and error (Ash, 1989; Parekh et al., 2000).

This problem is critical when multilayered feedforward networks are used. The most popular is the multilayered perceptron (MLP) that uses the backpropagation algorithm developed by Rumelhart et al., (1986). The MLP includes internal hidden units that are not part of the input or output yet represent important features that capture task-specific regularities. As a result of the hidden layer extension of the standard perceptron, the MLP is capable of solving non-linearly separable tasks such as the XOR problem, as well as its higher dimensional extension known as the *n*-bit Parity problem (Wilamowski et al., 2003). The *n*-bit problem is one of the most widely used problems for testing the efficacy of training algorithms for ANNs (Lu et al., 2012; Setiono, 1997b). If the number of ones in the vector is even, then the output is 1, and if the number of ones is odd, then the output is 0 (Lu et al., 2012; Setiono, 1997b). The number of input vectors of an *n*-bit problem is given by the dimension parameter *m,* where *m* is given by 2^n, with $n \in \mathbb{N}_1$being the number of bits.

Previous approaches have formulated strict rules on the minimum number of hidden threshold units required to solve the *n*-bit Parity problem. For instance, in the case of a single hidden layer of the MLP, simulation experiments have shown that the minimum number of hidden units needed is equal to *n* in order to solve the *n*-bit problem (Fung & Li, 2001; Minnick, 2009), whereas others have proposed that the general solution for solving the *n*-bit problem is *n*+1 (Hunter et al., 2012). These formulations require that the architecture of the MLP remains purely feedforward, with no direct connections between the input and output layers. In contrast, if additional connections exist between the input and the output layers, as in the Bridged MLP, a generalized solution is 2*n*+1 (Hunter et al., 2012). Although these rules allow the characterization of the number of hidden units required in the hidden layer of an MLP, these established rules require a prior specification of the number of hidden units. They are only applicable if certain conditions are met.

A potential solution is to have the topology grow as the network learns through constructive algorithms that start with a few units and incrementally add units and connections (for a detailed review, see Curteanu & Cartwright, 2011; Kwok & Yeung, 1997a). This algorithm class offers flexible economic topologies that match task complexity (Kwok & Yeung, 1997a). The most popular among them include CC (Fahlman & Lebiere, 1990) which freezes the existing network and trains the newly added unit, and DNC (Ash, 1989) which retrains the entire network. Previous work under the DNC category has shown that the backpropagation algorithm can be used with varying numbers of hidden units to approach the XOR problem (Hirose et al., 1991). To accomplish incremental growth, the total error was checked every 100 epochs, and a new hidden unit was added if the total error remained higher than one percent. As a result, the size of the network increased outbound and therefore required intervention via removing a hidden unit and retraining until obtaining a more reasonable number of hidden units (Hirose et al., 1991).

When should a constructive algorithm add a new unit? Adding units randomly or on a fixed time schedule could lead to unnecessarily large topologies. If a user defines a maximum capacity size, it could prevent excessive growth. Nonetheless, added hidden units may not have enough time to be trained to impact the training error and could still lead to the addition of unnecessary units. The aforementioned constructive algorithms employ a user-defined hyperparameter whereby if the average training error drops below or stops changing by a pre-set amount, a new unit will be added (Ash, 1989; Fahlman & Lebiere, 1990; Hirose et al., 1991). While very effective, this requires some *a priori* hyperparameter setting.

We propose an extension of the MLP using a flexible growing algorithm inspired by population dynamics (Sun, 2016). Theoretically, a neural network or each layer comprising the network can be viewed as a set of populations. From this perspective, a hidden layer can be considered the environment in which neuronal units exist. This would endow the hidden layer with a carrying capacity. Put simply, the carrying capacity is the maximum population size that the environment can sustain, or in terms of ANNs, it is an upper bound on the possible number of hidden units. Giving the hidden layer this constraint allows the application of population dynamics when calculating the growth of the hidden unit population. This provides a built-in self-governed method for preventing the network from growing too large. In this version, we are only concerned with the carrying capacity of the hidden layer and no other environmental factors, such as resource availability (Sun, 2016). As such, we consider this population a "single species" of the neuronal unit within the hidden layer environment. For a given task, the algorithm allows the network to incrementally add units in the hidden layer until the complexity of the task no longer requires further addition in the number of hidden units.

The adaptability of the growing network (gMLP) is tested using increasing levels of task complexity for the *n*-bit problem and compared to an identical MLP network with a fixed hidden layer size (fMLP). In doing so, we show that fixed rules for the number of hidden units needed to solve *n*-bits are not ideal and that dynamic, flexible growth will lead to better-generalized approximations in size irrespective of the rule used. We do not focus on the total number of training epochs the network takes, but on the total number of hidden units, the network uses for each rule across different *n*-bits.

The remainder of the chapter is divided as follows: Section II introduces the model describing the network's architecture, activation function, learning algorithm, and learning procedure for a standard MLP with a single hidden layer. We then describe the growing algorithm inspired by population dynamics. Section III describes the results of the *n*-bit Parity problem using a standard fixed hidden-layer size (fMLP) and an extended version of the dynamic self-governed growth algorithm (gMLP). In section IV, we discuss and conclude the overall findings of our work.

Methods

Architecture

The MLP used is a simple feedforward network and is comprised of three layers: an input layer, a single hidden layer, and output layer containing a single unit (Figure 1.1). If the size of the hidden layer, s, is fixed then its size is dependent on the input vector dimension, n, and the selected rule. In this condition all the weight connections are present from the beginning. If the size of the hidden layer, s, is permitted to change across time as with growing, then new units are added to the end of the hidden layer. The weight connections associated to the new hidden unit are added from the input layer to the hidden and from the hidden to the output layer.

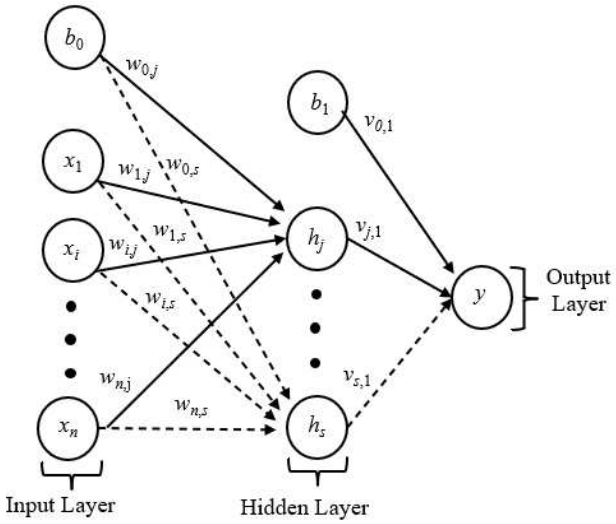

Figure 1.1 MLP architecture. In the above architecture x is a bit vector of length n, where n is the number of bits, s is the current number of hidden units, b the biases, and the dashed lines represent weight connections that are added during growing.

Activation Function

The activation function used for the MLP is the bipolar sigmoid function (1.1). This function ranges from -1 to 1, allowing a bipolar representation of the data (Figure 1.2). The derivative (1.2) is used when calculating the backpropagated error (Fausett, 1994).

$$f(x) = \frac{2}{1 + e^{-(x)}} - 1 \tag{1.1}$$

$$f'(x) = \frac{1}{2}(1 + f(x))(1 - f(x)) \tag{1.2}$$

Forward pass of the inputs through the hidden layer of the MLP network is derived according to:

$$h_j^{in} = b_0 + \sum_{i=1}^{n} x_i w_{ij} \tag{1.3}$$

$$h_j = f\left(h_j^{in}\right) \tag{1.4}$$

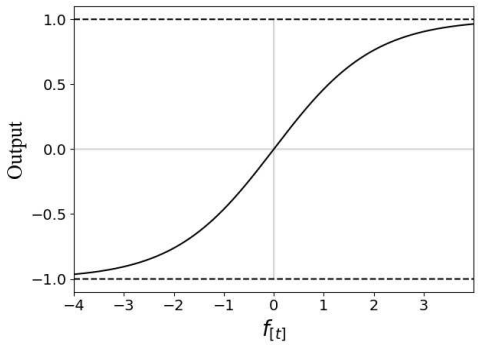

Figure 1.2 Bipolar sigmoid activation function.

where x_i is the input vector, w_{ij} are the weight connections between the input and the hidden layer, b_0 is the bias on the hidden layer, h_j^{in} is the activation of the hidden unit *j,* h_j is the output from the hidden unit, and f the bipolar activation function defined in (1.1).

Once the output from the hidden layer is obtained, this information is passed forward to the output layer according to:

$$y^{in} = b_1 + \sum_{j=1}^{s} h_j v_{j1} \tag{1.5}$$

$$y = f(y^{in}) \tag{1.6}$$

where v_{j1} is the weight connections between the hidden layer and the output unit, b_1 is the bias on the output layer, y^{in} is the activation of the output unit and y the output signal.

Learning Algorithm

The learning for the MLP network is based on standard backpropagation. Once an output is obtained from the MLP network, the error term (δ) is computed according to (1.7) from the output layer and back-propagated to the hidden layer.

$$\delta = (t - \mathrm{y}) f'(y^{in}) \tag{1.7}$$

where f' is the derivative of the activation function (1.2), and t is the associated target pattern. To improve convergence, targets were not set at the asymptotes of -1 and +1, but at -0.8 and +0.8

respectively (Fausett, 1994). The error term from the output layer is then used to calculate the error term from the hidden layer (δ_j) according to:

$$\delta_j^{in} = \delta v_{j1} \tag{1.8}$$

where δ_j^{in} is the summation of delta inputs from the output.

$$\delta_j = \delta_j^{in} f'\left(h_j^{in}\right) \tag{1.9}$$

Once the error terms have been calculated, the output unit updates its weight connections (1.10) and bias (1.11) by adding correction terms according to:

$$v_{j1}^{new} = v_{j1}^{old} + \eta(\delta h_j) \tag{1.10}$$

$$b_1^{new} = b_1^{old} + \eta\delta \tag{1.11}$$

where v_{j1}^{old} is the weight connections from the previous epoch, η the learning rate set to 0.1, v_{j1}^{new}represents the updated weight connections, b_1^{old} is the bias on the output layer from the previous epoch, and $b_1^{new}$is the updated bias term.

This process is then repeated by the hidden layer that uses its error term (δ_j) to create correction terms to update its own weights (1.12) and bias (1.13) according to:

$$w_{ij}^{new} = w_{ij}^{old} + \eta(\delta_j x_i) \tag{1.12}$$

$$b_0^{new} = b_0^{old} + \eta\delta_j \tag{1.13}$$

where w_{ij}^{old} is the weight connections from the previous epoch, $w_{ij}^{new}$is the updated weight connections, b_0^{old} is the bias on the hidden layer from the previous epoch, and $b_0^{new}$is the updated bias term.

The MLPs performance is calculated at the end of each epoch according to:

$$\text{MSE} = \frac{1}{m}\sum_{k=1}^{m}(t_k - \text{y}_k)^2 \tag{1.14}$$

where m is the input dimension (number of patterns) given by 2^n, with $n \in \mathbb{N}_1$ being the number of bits, and MSE is the mean-squared error of the MLP network.

Learning Procedure

Learning for both MLPs is conducted in the same manner whereby during each training epoch all input vectors are randomly shuffled and presented to the network. The learning is outlined by the following steps:

1) Initialization of weight connections at random values between -0.5 and 0.5.
2) Forward propagation, where the input vector is broadcasted to all hidden units that sum their weighted input signals according to (1.3) and apply the bipolar activation function (1.1) to compute each of their respective outputs (1.4).
3) Continuation of forward propagation where the output unit receives the summation of weighted input signals according to (1.5), over which the bipolar activation function is applied (1.1) to compute its output (1.6).
4) Each output unit receives the associated target patterns for each respective input training pattern and calculates its error term (δ) which is then backpropagated to the hidden layer (1.7).
5) The hidden layer uses the backpropagated error and calculates its own error information term δ_j according to (1.8) and (1.9).
6) Weights are then updated according to (1.10) for the output layer and (1.12) for the hidden layer. Biases are also updated according to (1.11) and (1.13).
7) Training continues until the maximum number of epochs is reached (set to 10^6) or if the MSE (1.14) falls below a minimum value set at 0.001.

The only exception between the fMLP and the gMLP learning procedures is the inclusion of the growing algorithm. The gMLP starts with only a single hidden unit and can only add units at the end of each epoch after the network has updated its weight connections. When adding a new unit, its new connections are also initialized at random values between -0.5 and 0.5. They are then updated along with all the other weight connections during the next epoch.

Growing Algorithm

The growth of the population of hidden units is dictated by the growing algorithm (1.15). This algorithm is adapted and modified from the Single-species model with the Allee effect outlined in (Sun, 2016). This algorithm is used during each epoch to calculate the change in the size of the hidden unit population (i.e., growth rate). Adding a new unit to the hidden layer only occurs when the growth rate reaches an integer value, as adding a proportion of a single unit is not plausible. The dynamics of the growing algorithm are characterized by the following:

$$\frac{\mathrm{d}h_s}{\mathrm{d}t} = MSE * \left(1 - \frac{h_s}{C}\right)(h_s - \alpha) \tag{1.15}$$

where C is the carrying capacity of the hidden layer environment (upper bound) determined by the fixed rule used, h_s is the current size of the hidden unit population, and α is a constant set to 0.2.

As the hidden unit population grows within the hidden layer, it receives direct performance feedback from the gMLP network through the MSE. The MSE modulates the rate at which the hidden unit population can grow (Figure 1.3). The MSE is inversely proportional to the growth rate. In this regard, as the gMLP learns, its performance can slow the hidden layer's growth rate. Concerning ANNs, during the learning process, the error is reduced and should approach zero. With a gradually declining error, the growth rate is in a state of constant change that is dependent on the network's performance. In this context, even though the hidden layer has a maximum

carrying capacity, the hidden unit population will grow toward the optimal number of hidden units required to reach an error close to zero. This is reflected in Figure 1.4, where curves: 10, 8, 6, 4, and 2; depict hypothetical situations where the MSE is initiated at a value of 1 and converges to 0.001 at different numbers of hidden units within the hidden layer. These curves reflect the fluctuation in growth rate due to error feedback from the network's performance. Hence, the minimum error may either be reached at the carrying capacity or at significantly lower hidden layer sizes, depending on task complexity or the defining characteristics of the network itself, such as the architecture or learning rule.

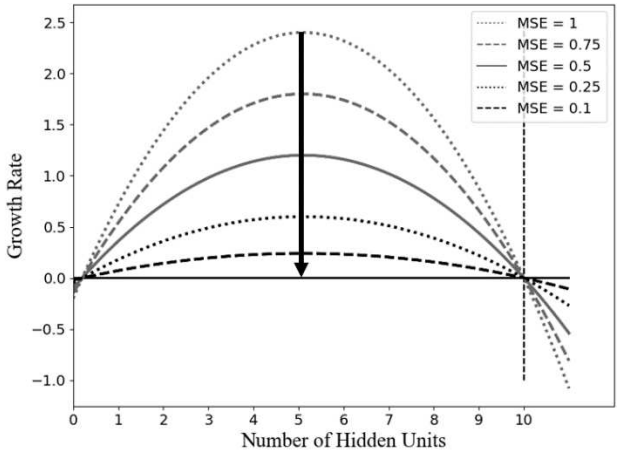

Figure 1.3. Hidden layer growth rate function with constant error. In the above figure the growth rate of the hidden unit population is shown with respect to the gMLP network's MSE. Here the carrying capacity (C) is fixed at 10 hidden units, and the MSE is held constant at values varied between values of 1 and 0.1.

Simulation: *n*-Bit Parity Problem

The *n*-bit parity vectors used in the following simulations were given a bipolar representation of -1 and 1. This set of simulations tested rules: 2^n, $2n+1$, and $n+1$; for determining hidden layer size for parity 2 to 7-bit problems, where n is the number of bits. These rules were implemented in two identical MLP networks. The fMLP with a fixed number

of hidden units given by the rules, and the gMLP uses the number of hidden units predicted by the rules as the carrying capacity.

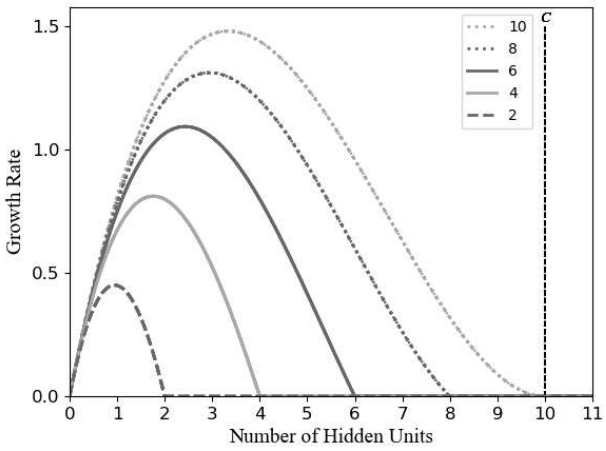

Figure 1.4. Hidden layer growth rate function with declining error. In the above figure the growth rate of the hidden unit population is shown with respect to the gMLP network's MSE. Here the carrying capacity (C) is fixed at 10 hidden units, α is set to 0.

Results

The effectiveness of the fMLP and the gMLP was evaluated for solving the n-bit parity problem for 2 to 7-bits across 100 trials per bit. To obtain an idea of just how the gMLP compared to the fMLP, a single trial of the 4-bit problem using the rule 2^n is examined in detail. Using the rule 2^n, the fMLP is set to 16 units. This means that this network has a total of 65 weight connections. As shown in Figure 1.5a, the MSE of the fMLP decreases gradually until reaching the pre-set minimum error of 0.001. In contrast, the gMLP uses the rule 2^n to set the carrying capacity to a value of 16 for the 4-bit problem. This network starts with a single hidden unit and grows in an ascending step-like fashion before converging at 5 hidden units (see Figure 1.6). Therefore, this network has a total of only 21 weight connections. As can be seen in Figure. 1.5b, the MSE of the network decreases in descending step-like fashion until it reaches the pre-set minimum error of 0.001. The plateaus in this figure indicate that the error is decreasing less and

less between epochs, and a new unit is needed. The spikes in error seen in Figure 1.5b at the end of the plateaus co-occur with adding a new unit to the network (see Figure 1.6) and consequently adding new randomized weight connections. This initial error spike is quickly reduced, allowing the network to continue to train and further lower the MSE. To summarize, using the same rule, 2^n, the fMLP uses more hidden units and weight connections than the gMLP but takes fewer training epochs.

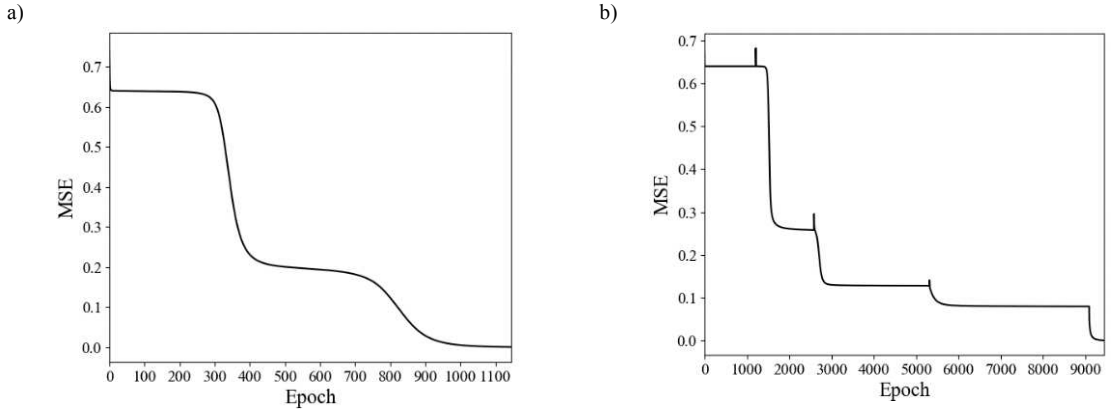

Figure 1.5. Mean squared error (MSE) across MLP training epochs for a 4-bit trial. a) MSE across fMLP training epochs when the rule 2^n determines a fixed hidden layer of 16 units. b) MSE across gMLP training epochs when the rule 2^n sets the carrying capacity (C) at 16, but the hidden layer only grows to 5 units.

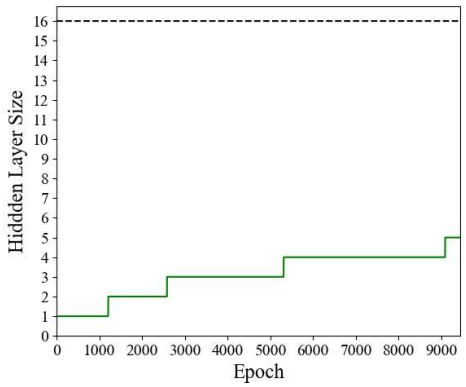

Figure 1.6. Growth of the hidden layer across epochs for a 4-bit trial. In the above figure the size of the hidden unit population is shown growing across gMLP training epochs. Here the carrying capacity, depicted by a dashed line, is determined by the rule 2^n to be 16 hidden units.

To verify that this result was consistent, both networks were subjected to 100 trials of each bit problem from 2 to 7-bits. This process was repeated across the rules: 2^n, $2n$+1, and n+1; for 18 conditions per MLP network. The results of the fMLP are shown in Table 1.1, and the results of the gMLP are shown in Table 1.2. The "Pred." (predicted) column gives the number of units the rule indicates should be used, and the "Actual" column gives the number of hidden units the network used across 100 trials. Due to the variability in gMLP performance, the number of hidden units in the hidden layer varies from trial to trial. As such, a range of hidden units is reported per bit in Table 1.2 from the least to most hidden units used across the 100 trials. To ensure full transparency, the frequency (percentage) of each hidden unit size for the 100 trials is reported across all the bit problems tested (Table 1.3).

Table 1.1

fMLP Hidden Layer Size (Averaged Over 100 Trials)

Bit	Rule for Number of Hidden Units								
	2^n			$2n$+1			n+1		
	Pred.	Actual	Epochs	Pred.	Actual	Epochs	Pred.	Actual	Epochs
2	4	4	789	5	5	721	3	3	918
3	8	8	315	7	7	334	4	4	434
4	16	16	1 174	9	9	1 041	5	5	5 358
5	32	32	461	11	11	514	6	6	8 596
6	64	64	6 693	13	13	3 967	7	7	10 535
7	128	128	703 721	15	15	12 917	8	8	140 490

Table 1.2

gMLP Hidden Layer Size (Averaged Over 100 Trials)

Bit	Rule for Number of Hidden Units								
	2^n			$2n$+1			n+1		
	Pred.	**Actual**	Epochs	Pred.	**Actual**	Epochs	Pred.	**Actual**	Epochs
2	4	**2-3**	2 685	5	**2-3**	2 463	3	**2**	7 674
3	8	**2-3**	3 173	7	**2-3**	3 276	4	**2-3**	3 888
4	16	**3-13**	25 678	9	**3-7**	14 084	5	**3-5**	26 056
5	32	**4-25**	22 941	11	**4-10**	50 334	6	**4-5**	103 412
6	64	**5-63**	60 598	13	**5-13**	93 686	7	**5-7**	261 749
7	128	**7-128**	321 232	15	**6-15**	527 059	8	**6-8**	631 603

Table 1.3

Frequency of Hidden Unit Sizes in gMLP Across 100 Trials

Bit	Rule for Number of Hidden Units		
	2^n	$2n+1$	$n+1$
2	**2:99%**, 3:1%	**2:94%**, 3:6%	**2:100%**
3	**2:83%**, 3:17%	**2:89%**, 3:11%	**2: 93%**, 3:7%
4	3:14%, **4:57%**, 5:25%, 6:3%, 13:1%	3:23%, **4:63%**, 5:13%, 7:1%	3:27%, **4:69%**, 5:4%
5	**4:73%**, 5:19%, 6:4%, 7:2%, 11:1%, 25:1%	**4:71%**, 5:22%, 6:2%, 7:1%, 10:4%	**4:72%**, 5:28%
6	5:17%, **6:43%**, 7:18%, 8:11%, 9:2%, 10:1%, 12:1%, 13:1%, 17:1%, 19:1%, 22:1%, 60:1%, 63:1%	5:16%, **6:46%**, 7:21%, 8:7%, 9:1%, 10:2%, 11:1%, 12:5%, 13:1%	5:22%, **6:68%**, 7:10%
7	**7:38%**, 8:16%, 9:2%, 11:2%, 12:3%, 13:4%, 23:1%, 25:1%, 27:1%, 54:1%, 82:1%, 119:1%, 125:1%, 126:2%, 127:10%, 128:16%	**7:42%**, 8:10%, 9:1%, 10:1%, 11:3%, 14:12%, 15:32%	6:3%, **7:50%**, 8:47%

Depending on the rule, the results of the fMLP for both 2 and 3-bit problems use between 3-8 hidden units (Table 1.1). In contrast, the gMLP uses between 2 and 3 hidden units regardless of the rule (Table 1.2). The percentages of each hidden unit size used across the 100 trials per each rule are shown in Table 1.3. These results show that 80-100% of the time, only two units are used for the 2 and 3-bit problems. A visual of the classification quality for a single trial of the 2-bit problem is shown in Figure 1.7. As can be seen in Figure 1.7a-c, the fMLP has clearly defined classification across the three rules, with the fMLP using: 4 units for 2^n, 5 units for $2n+1$, and 3 units for $n+1$. In contrast, the gMLP uses only 2 hidden units for all three rules. However, as seen in Figure 1.7d-f, the gMLP can still maintain clear distinct classification borders.

Examining the results of the 4-bit problem in depth using the rule 2^n, the fMLP used 16 hidden units (Table 1.1). While the gMLP varied from 3-13 units across the 100 trials (Table 1.2). However, examining these numbers closely, 99% of the time, the number of hidden units was between 3-6, and only 1% of the time was the hidden layer size 13 (Table 1.3). This is reflected in

Table 1.3 with a mode at 4 hidden units. Continuing to both 2*n*+1 and *n*+1 rules, the fMLP used 9 and 5 units, respectively (Table 1.1). The gMLP varied between 3-7 units, but closer examination reveals that only 1% of the time was 7 units used. Additionally, both distributions have a mode at 4 (Table 1.3).

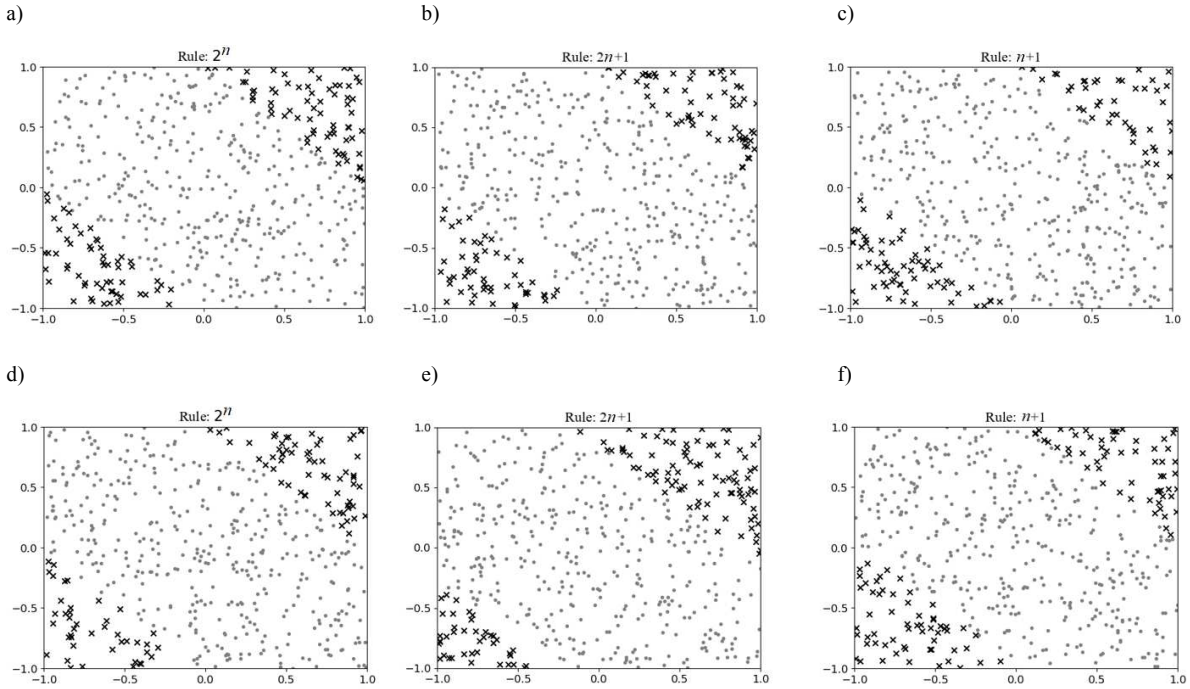

Figure 1.7. Classification borders of the MLP networks for a single trial of the 2-bit problem across the three different rules. a-c) fMLP classification quality across the three different rules for 500 random points. d-f) gMLP classification quality across the three different rules for 500 random points.

At 5-bit and higher, the number of predicted units shows distinct variation between rules. At 2^n the fMLP used 32 units, 11 units at 2*n*+1, and 6 units at *n*+1. Contrary to this, similarity can be seen across rules for the gMLP with a mode at 4 for all three distributions (Figure 1.8a-c). For 2^n, 98% of the trials used 7 units or less. Both the 11 and 25 units occur only 1% each. This effect is seen using the 2*n*+1 rule, with 96% of the trials using 7 units or less. These results are again continued with the *n*+1 rule, where 100% of results occur between 4-5 units. Similar results are observed for the 6-bit problem. Finally, for the 7-bit problem, the fMLP, dependent on the rule,

used 128, 15, and 8 hidden units. However, using the rule 2^n, which leads to using 128 hidden units, the network cannot converge during learning for many of the 100 trials. The gMLP showed variation compared to previous results with respect to the rules 2^n and $2n$+1. For 2^n in the gMLP, 65% of trials used between 7 and 13 hidden units; however, 30% of trials used between 119 and 128 hidden units (Table 1.3). Similar results are found using the rule $2n$+1, with 56% of trials between 7 and 12 and 44% of trials using 14 or 15 units. These two rules show evidence of bimodal distributions. Despite this, a mode of 7 hidden units is still observed across rules (Table 1.3).

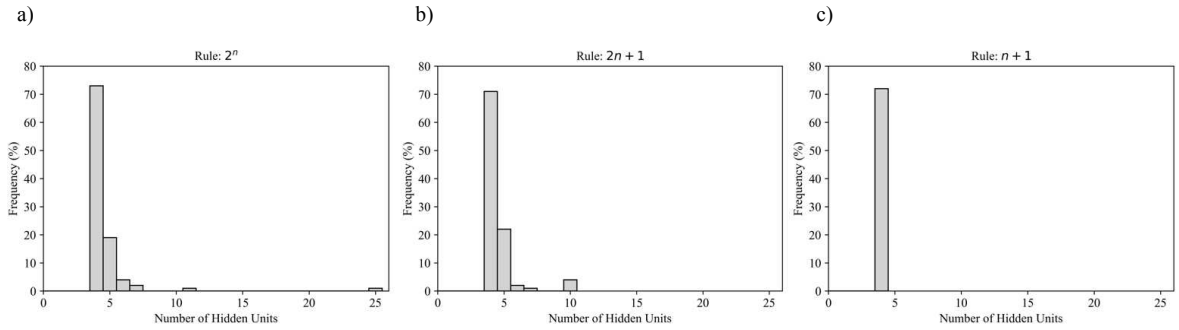

Figure 1.8. Frequency of hidden units needed by the gMLP across 100 trials. a-c) Final number of hidden units from 100 trials for the 5-bit problem using three different rules to determine the hidden layer's carrying capacity.

Discussion

In this study, we introduced a novel constructive algorithm inspired by population dynamics to incrementally add units to a MLP. This algorithm utilizes a carrying capacity hyperparameter that prevents the network from growing outbound. This endowed the network with a growth rate that allowed more rapid growth when the hidden layer size was far from capacity and a slower growth rate when it approached capacity. This carrying capacity was used in parallel with direct performance feedback from the network in the form of the MSE. The MSE scaled the growth rate in regards to the performance of the network. This allowed for dynamic self-governed growth, the rate of which was modulated both by network performance and hidden layer size.

To test the adaptability of this algorithm, we subjected the gMLP to increasing levels of task complexity for the *n*-bit problem and compared our results to an identical fMLP. We aimed to show that using fixed rules for hidden layer size, flexible, dynamic growth would lead to generalized approximations in hidden layer size regardless of the rule used. Even if a fixed rule sets the upper boundary of the carrying capacity, the growing algorithm can converge to several hidden units, falling within the lower range of units required for solving the given task. This effect is consistent regardless of the rule used for deciding on the upper boundary of the carrying capacity.

As illustrated in our results, the gMLP successfully showed a general hidden layer size regardless of the rule used. This is evident from the distributions for each of the three rules having the same mode for each bit problem. Additionally, as previously mentioned, only one of these rules, n+1, is an actual generalized rule for solving the *n*-bit problem with a feedforward architecture. The other two rules, 2^n and $2n$+1, were chosen to push both MLP networks to larger hidden layer sizes than the *n*-bit problem requires. Our results were clear for the fMLP. For the 7-bit problem, using too many hidden units, such as with the rule 2^n, caused the network during some trials to reach the maximum number of epochs set at one million to prevent network convergence. Contrary to this, the gMLP showed more robust qualities and, overall, across 100 trials, could converge to smaller topologies while still solving the task.

It was not until the 7-bit problem was reached that the gMLP experienced difficulties, as evident from the two bimodal distributions for the rules 2^n and $2n$+1. This problem is not new to growing methods. One of the limitations of growing methods is that the networks can sometimes get trapped in local minima (Curteanu & Cartwright, 2011; Narasimha et al., 2008). This, as well as slow convergence, are particularly pertinent problems with backpropagation. To escape the local

minima adding more than one unit at a time could solve this issue. To address slow convergence, using a modified error term (Bi et al., 2005) or avoid using standard backpropagation and use faster training methods like Quickprop in CC (Fahlman & Lebiere, 1990). Furthermore, for some of our results, the final hidden layer size is sometimes less than the minimum of n. For instance, with the 4-bit problem, the gMLP uses only 3 hidden units. Previous work using analytical solutions has shown that the number of hidden units required to solve n-bit problems is governed by $\left(\frac{n}{2}+1\right)$ if even and $\left(\frac{n+1}{2}\right)$ if odd (Setiono, 1997b). According to this metric, the 4-bit problem can be solved by a minimum of 3 hidden units.

Despite the model's success in solving the n-bit problem, it is important to note the variation in training times between the fMLP and the gMLP. The gMLP took more training epochs, growing from a single hidden unit to the amount needed to perform the task. However, as previously mentioned, optimizing application training times were not this study's objective. Instead, the primary objective was the proof of concept of a new growing algorithm inspired by population dynamics and the demonstration of its effectiveness and plausibility. The main novelty includes a carrying capacity and, by extension, a growth rate. Unlike previous methods, such as with the DNC (Ash, 1989) that calculates the average error in relation to a user-set minimum slope, the addition of new units to the hidden layer is not user determined. Instead, it is more self-governed based on the growth rate, which is directly scaled by the current size of the network in relation to the carrying capacity and simultaneously by the training performance of the network through the MSE. Even though a measure of error is used, there is no user-set criterion for how much the error stops changing by to then introduce a new unit to the network.

A similar mechanism is seen using evolutionary algorithms for growing the network structure. For instance, the inclusion of a growth probability operator and an accompanying growth

rate distribution to determine how many units the network grows by (Ang et al., 2008). This has been shown to exhibit higher growth probabilities at initial stages and lower growth probabilities at later stages, effectively speeding up the process of finding an optimal solution. Though this method is self-adaptive, it introduces randomness through mutations, requires an evaluation of fitness, and takes place across many networks, with only the best-performing ones chosen to "survive" (Ang et al., 2008). In the present algorithm, the growth rate is being utilized as a self-governed process without referring to processes governed by evolutionary dynamics. Additionally, only one network is grown, and the population refers only to the hidden layer size. Therefore, the current dynamic growth algorithm introduced here can be seen as a hybrid between evolutionary and constructive approaches.

To validate the observed results, replication with other tasks is needed, such as the Double Moon classification problem, where data point is distributed into two crescent "moon" shapes. The challenge is to classify the overlapping and non-linear shapes (Haykin, 2009). In this case, the gMLP would estimate the number of hidden units required to solve the task. This estimated value for the number of hidden units would provide an approximation of the general rule required to solve the Double Moon problem. In the same way that our approach provided approximations close to the rules mentioned for solving the *n*-bit Parity problem. If fixed rules that are task-specific can be generalized to other tasks and still provide accurate results, it would effectively validate the growing algorithm applied here. Additionally, it would be pertinent to test the generalization concerning the classification of the growing network to overfitting or underfitting the data. Such a validation could be achieved using a k-fold validation. Additionally, by combining this dynamic growth algorithm with more efficient learning approaches like Quickprop, we aim to reduce training times to allow a more direct comparison between growing and fixed networks. Future

work aims to refine this algorithm to incorporate neuronal pruning, allowing the algorithm to have a negative growth rate to decay the neuronal population and obtain an optimal minimal topology. To govern such a method, one potential avenue would be to re-introduce the possibility of local extinction as outlined in the original population dynamics model (Sun, 2016). Such a mechanism could create internal competition, potentially allowing a synergistic effect to reach optimal topologies. Finally, it would be very interesting to explore how a multilayered network with two hidden layers could be governed by two of these algorithms, whereby the growth of one layer could lead to pruning in another depending on the performance and needs of the network.

Conclusion

Dynamic growth inspired by population dynamics introduces a constructive approach that is self-governed. This approach can circumvent arbitrary trial and error to determine optimal hidden topologies. Furthermore, it also avoids the problem of growing outbound by converging to smaller topologies on its own, according to how both the network performance and the hidden layer size modulate the growth rate—the self-governed dynamics of this growing algorithm merit further investigation.

Chapter 2

A Constructive Algorithm for Deciding When to Grow: A Dynamic More Self-Governed Approach

Abstract

When employing constructive algorithms, deciding when to add a new hidden unit is an important challenge to be considered. The most common method is if the error stops changing by a pre-set amount over a given number of trials. However, this approach heavily depends on *a priori* fine-tuning of hyperparameters which can be task-specific. Alternatively, a dynamic, constructive growing algorithm inspired by population dynamics offers a more self-governed approach to this decision process. This approach is investigated to validate the algorithm and its self-governed properties. The inherent properties of the algorithm are examined in a single hidden layer multilayer perceptron (gMLP) tasked with classifying the MNIST data set. The number of units grown by the gMLP is validated and compared directly to identical MLPs, with fixed hidden layer sizes (fMLP) and another constructive method, Dynamic Node Creation (DNC-MLP). In short, the results depict that the model's hyperparameters require less fine-tuning by the user and adhere more toward self-governance. Furthermore, the number of hidden units grown by the gMLP is appropriate and consistent with our general benchmark predictions. The gMLP showed slightly lower performances than the fMLP and similar performances compared to the DNC-MLP. The distinction of when a new unit is added highlights the advantage of a more self-governed approach. Dynamic growth inspired by population dynamics offers a more self-governed alternative automatically tailored to a given task compared to the user finely-tuning hyperparameters for deciding when to add new hidden units.

Determining the number of hidden units to use in a multilayered feedforward network can be critical to solving a given problem. Not enough hidden units in the network may lead to underfitting and failing to learn a given problem adequately. In contrast, if there are too many hidden units in the network, it may lead to overfitting as a result of too much flexibility and cause poor generalization (Boughrara et al., 2016; Curteanu & Cartwright, 2011; Kwok & Yeung, 1997a; Liu et al., 2002; Parekh et al., 2000). Currently, there is no one agreed-upon method for determining the optimal topology, as it can vary drastically depending on the given problem or the network used. As such, the current practice is to use fixed architectures that are determined via a trial-and-error approach. This process is both computationally and temporally cumbersome and does not guarantee that an optimal topology will even be found (Hernández-Espinosa & Fernández-Redondo, 2002; Parekh et al., 2000; Zemouri et al., 2018). To address the issue of finding optimal topologies, several approaches have been proposed and can be broadly categorized into statistical and empirical, evolutionary, and non-evolutionary approaches.

Statistical and empirical methods have been used to determine the optimal number of hidden units and many other optimal network parameters (for a survey, see Curteanu & Cartwright, 2011). Of note is the use of singular value decomposition (SVD) for determining the optimal number of hidden units in single-layer feedforward networks. SVD has been applied as an ad hoc technique to automatically remove redundant hidden units (Psichogios & Ungar, 1994) and estimate the number of hidden units for a given task (Teoh et al., 2006). Recently, SVD has been applied *a priori* directly to the training data to estimate the number of hidden units in a single layer feedforward network for classification tasks (Cai et al., 2019). While this approach is practical, it is user-dependent and not self-governed.

Evolutionary approaches simulate natural selection and the evolutionary process to stochastically search for the optimal neural network topology. Evolutionary approaches offer an adaptive framework whereby topologies can be dynamically altered without direct user intervention (for reviews, see: Castillo et al., 2000; Curteanu & Cartwright, 2011; Ding et al., 2013; Yao, 1993). Conversely, non-evolutionary approaches use specific algorithms to determine optimal topologies automatically (Do Carmo Nicoletti et al., 2009). These approaches can be divided into either destructive or constructive methods. Destructive methods begin with a large initial network where units/connections are systematically pruned (for a survey of pruning methods, see Augasta & Kathirvalavakumar, 2013). Inversely, constructive methods begin with a small initial network and systematically add units/connections.

Constructive algorithms have several advantages over their destructive counterparts, as outlined by Kwok and Yeung (1997a, 1997b):

- Initialization of the constructive approach is simple, typically using only a single unit, whereas, with pruning, a guess must be made on how large the initial network should be.
- The constructive approach is more computationally economical as it builds smaller network topologies first, with the resulting constructed topologies being directly related to the complexity of the given problem and the performance requirements (Ma & Khorasani, 2003). Conversely, with pruning, most of the training time is spent on network topologies that are larger than required. This potentially wastes computational effort and time removing unnecessary units and weight connections (Ma & Khorasani, 2003).

- Constructive algorithms are more efficient in forward computations as they have less weights and other parameters to update initially, they require less data for good generalization, are more easily described, and are more likely to find smaller topology solutions.
- For computational efficiency, pruning algorithms typically estimate the sensitivity of the error function when an element is removed (Reed, 1993). However, using only estimations can introduce significant errors to the network (Kwok & Yeung 1997a, 1997b).

Constructive algorithms have been applied successfully to many problems, including facial expression recognition (Boughrara et al., 2016; Ma & Khorasani, 2004b), medical diagnosis (Kamruzzaman et al., 2004; Zemouri et al., 2018), modeling psychological development (Shultz, 2012), and even ship design optimization (Besnard et al., 2007). It has been well-documented that constructive algorithms are suitable for solving classification problems. The most well-known constructive algorithms are in the context of two-class classification problems (for a detailed review, see Do Carmo Nicoletti et al., 2009), such as the Tiling (Mézard, 1989), Tower (Gallant, 1990), and Upstart algorithms (Frean, 1990). Many of these algorithms have been extended to solve multiclass classification as well (for reviews and examples, see Bertini & Do Carmo Nicoletti, 2009; Do Carmo Nicoletti et al., 2009; Li et al., 2010; Parekh et al., 2000).

These algorithms can be generally classified according to the training scheme they employ. After a new hidden unit is added to the network, either only the new hidden unit is trained, or the whole network is completely trained (Kwok & Yeung, 1997a). For training only the newly added hidden unit, all connections for every other hidden unit are fixed in a process known as “freezing.” Then only the newly added unit’s connections are trained. This training scheme dramatically

reduces the computational load and achieves faster convergence (Aran et al., 2009; Islam et al., 2009a; Kwok & Yeung, 1997a). However, this training scheme may result in networks with more hidden units than needed (Curteanu & Cartwright, 2011; Islam et al., 2009a) and fail to obtain optimal weights for the network as a whole (Kwok & Yeung, 1997a). By "freezing" the existing weights, the weight space is confined, limiting possible solutions (Aran et al., 2009), with new units usually being inefficient feature detectors leading to more units added and possible overfitting (Curteanu & Cartwright, 2011). CC is the most popular constructive algorithm that employs this training scheme (Fahlman & Lebiere, 1990). Conversely, an advantage of retraining the whole network is its implementation simplicity (Islam et al., 2009a). One of the most influential constructive algorithms that use this learning scheme is DNC (Ash, 1989). This model adds hidden units if the error curve flattens below an *a priori* defined trigger slope value within a pre-determined time window. After each addition, the entire network is retrained, and the process is repeated until a minimum error is achieved. Many variants inspired by DNC have been proposed (Azimi-Sadjadi et al., 1993; Bartlett, 1994; Hirose et al., 1991; Sharma & Chandra, 2010b; Zhang, 1994). However, the simplicity of this training scheme comes at a cost. The added computational load of retraining everything can lead to scale-up problems if the network grows too large (Kwok & Yeung, 1997a). Furthermore, each hidden unit sees a constantly changing environment due to all the weights in the network changing at once; this is known as the "moving target problem" and can cause slower learning (Fahlman & Lebiere, 1990).

When employing constructive algorithms, Sharma and Chandra (2010) outlined several challenges that should be considered: how to connect a newly added hidden unit, the activation function(s), the training scheme (re-train the whole network vs. "freezing"), which optimization

technique should be used, and when to stop adding hidden units. However, an additional challenge should also be considered; when should a new hidden unit be added?

This has previously been approached using various methodologies, including statistical methods. Such as calculating network significance to estimate the network's generalization power with respect to bias and variance (Ashfahani & Pratama, 2019; Pratama et al., 2020) or calculating the weighted sum of non-extensive entropies (Susan & Dwivedi, 2014) and adjusting the hidden layer size accordingly. Alternatively, hidden units can be added randomly with an upper bound on the number of hidden units (Huang & Chen, 2008; Huang et al., 2006). Typically, however, the method for determining when to add a new hidden unit is generally standard across many constructive algorithms (Sharma & Chandra, 2010b). The most common method is if the error drops below or stops changing by a pre-set amount, usually over a given number of trials, a new hidden unit is added, often referred to as a patience parameter. Many approaches use this method, including the popular CC (Fahlman & Lebiere, 1990), DNC (Ash, 1989), and many of their variants (Bartlett, 1994; Hirose et al., 1991; Islam & Murase, 2001; Kwok & Yeung, 1993; Zhang, 1994), and others (Barakat et al., 2011; Qiao et al., 2016; Wu et al., 2015). This adds additional hyperparameter(s) that generally require(s) *a priori* fine-tuning to meet a given task's requirements.

Previously, we introduced a constructive growing algorithm that provides a more self-governed alternative for deciding when a new unit should be added (Ross et al., 2020). Inspired by population dynamics (Sun, 2016), this approach considers the hidden units as a population and the hidden layer as the environment they exist in. This provides the hidden layer with a carrying capacity, the maximum population the hidden layer environment can sustain, and allows the application of population dynamics to provide a population growth rate. Combining a carrying

capacity and direct performance feedback from the network incorporated into the algorithm creates a built-in dynamic self-governed method for growing the hidden layer, while simultaneously preventing growing outbound. To test this new approach, we implemented it in a MLP. We tested its adaptability on the *n*-bit Parity problem compared to an identical MLP with a fixed hidden layer size. Known topology rules determined the carrying capacity and fixed layer sizes. With *n* equal to the number of bits, these rules were: n+1 for a single hidden layer MLP (Hunter et al., 2012), $2n$+1 for a Bridged MLP that has direct connections from the input layer to the output layer (Hunter et al., 2012), and 2^n using the same number of hidden units as inputs. The results showed that regardless of the rule used to predict hidden layer size, dynamic growth leads to smaller topologies than predicted while still being capable of solving the task. This effect was consistent across various topology rules and various levels of task complexity. While this showed the initial success of the new approach, the self-governed dynamics of this growing algorithm still merit further investigation.

We propose investigating this approach further with three objectives to validate the efficacy of the technique and its self-governed properties. Within this context, we define more self-governed as a constructive approach that dictates when new hidden units should be added while limiting the user's role in fine-tuning hyperparameters *a priori*. In order to validate the self-governed aspect, first, we will examine the inherent properties of the algorithm. Second, we aim to show that the number of hidden units grown by our algorithm is minimal, appropriate, and suitable for the task. Finally, we will validate the approach used by the algorithm by comparing it directly to a fixed network and another constructive method, DNC, where the decision to add another unit is more dependent on *a priori* finely-tuned hyperparameters.

Single hidden layer feedforward networks with additive hidden units have been proven to be universal approximators (Hornik et al., 1989). These commonly used shallow networks require less computational resources and time than deeper networks (Tissera & McDonnell, 2016). For practicality, a simpler neural network is favoured for investigating the proposed growing algorithm and its inherent properties before examining it in more complex neural structures. To test the objectives outlined above, the growing algorithm is implemented in a single hidden layer multilayer perceptron (gMLP) and tasked with the commonly used MNIST (Mixed National Institute of Standards and Technology) data set of handwritten digits. The readily available MNIST data set has seen widespread use exhibiting its reliability and suitability for testing and benchmarking new models (Shamsuddin et al., 2018). The inherent properties of our algorithm are thus examined in detail using the MNIST data set. To determine if the growing algorithm is growing the hidden layer to an ideal size for learning the MNIST data set, a general benchmark hidden layer size is determined using both a fixed topology MLP (fMLP) that is manually set *a priori* and SVD. Finally, the classification performance and growing dynamics of the gMLP are directly compared to the fMLP and another identical MLP whose growth is governed by DNC (DNC-MLP). The generalized applicability of the gMLP is then considered by comparing the three approaches on a novel data set.

The remainder of the chapter is divided as follows: Section II introduces the model describing: the data set, the network's architecture, the activation function, the optimizer used, the learning procedure, the loss function, and the growing algorithm. Simulation I investigates properties of the growing algorithm, specifically the extinction constant, the effect of weight initialization, the carrying capacity, and the time step of the Euler approximation are examined using the gMLP. Simulation II establishes a general hidden layer size benchmark using the fMLP

and SVD. Simulation III compares the performances of the gMLP and fMLP networks in single trials. Simulation IV outlines the performance and growth dynamics of the DNC-MLP. Simulation V compares the average performance of all three networks across 25 trials. Simulation VI compares the average performance of all three networks on a novel data set. In Section III, we discuss and conclude the study's overall findings.

Methods

MNIST

The input used in all simulations is the MNIST data set. The MNIST data set consists of 70 000 black and white 28x28 pixel images of handwritten digits (LeCun et al., 1998; see Figure 2.1). The data set is divided into 60 000 images for training and 10 000 for testing the classifier. The dimensionality of the binary vector for each image totals 784 (n). The target labels for each image are one-hot encoded into a binary target vector, with each element representing a possible class. If an element has a value of 1, this indicates the correct associated class, also known as the ground truth.

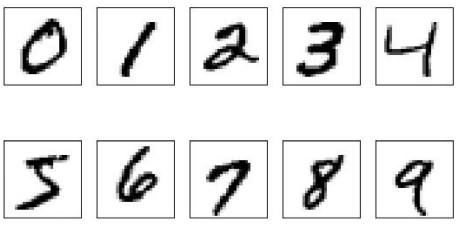

Figure 2.1. Sample of handwritten digits from the MNIST data base.

Architecture

A single hidden layer MLP is utilized, with the network architecture illustrated in Figure 2.2. The input and output layer sizes are fixed, while the current size of the hidden layer, *s*, is either

fixed or grows. If the hidden layer is fixed, it is manually pre-set, with all weight connections being present from the beginning. Suppose the size of the hidden layer is permitted to grow across time. In that case, the hidden layer begins with a single hidden unit and the associated weight connections from the input layer to the output layer. As learning progresses and growth occurs, new hidden units are added to the end of the hidden layer with all associated weight connections.

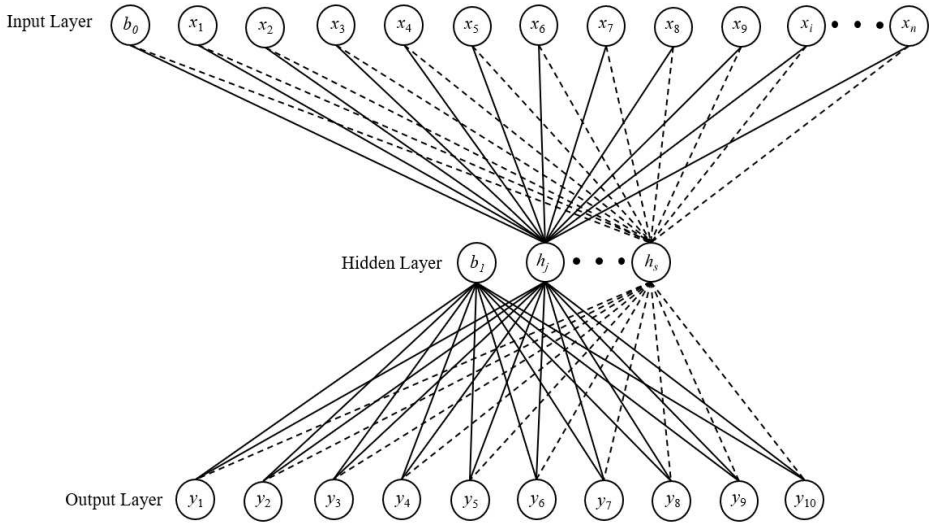

Figure 2.2. MLP architecture. In the above architecture x is vector of length n, where n, is the length (number of features) of a single number in the MNIST data set, s is the current number of hidden units, and the dashed lines are weight connections that are incrementally added during growing.

Activation Function

The activation function used for the hidden layer output of the MLP is the sigmoid activation function (2.1) (see Figure 2.3). The softmax function (2.2) is then employed to scale the final output of the network and generate interrelated probabilities for multiclass classification between 0 and 1.

$$h_j = \frac{1}{1 + e^{-(h_j^{in})}} \tag{2.1}$$

where h_j is the hidden layer output from unit j obtained in the usual way (for details see Ross et al., 2020) and h_j^{in} is its corresponding activation.

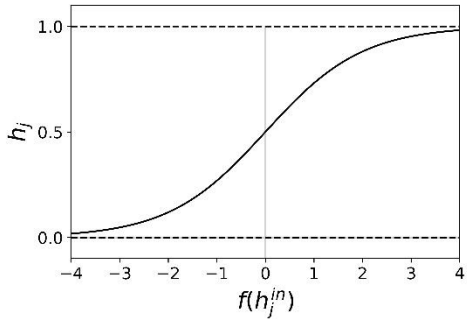

Figure 2.3. Sigmoid activation function.

$$\hat{y} = \frac{e^{y_i^{in}}}{\sum_{k=1}^{N} e^{y_k^{in}}} \tag{2.2}$$

where $\hat{y}$ is the categorical probabilistic output vector that sums to 1 and y_i^{in} is the input vector. The denominator is a normalization term across all classes (N). The derivative of the sigmoid function (2.3) is then used during backpropagation on the hidden layer activation.

$$f'(h_j^{in}) = \left(\frac{1}{1 + e^{-h_j^{in}}}\right)\left(1 - \frac{1}{1 + e^{-h_j^{in}}}\right) \tag{2.3}$$

Optimizer

ADAM (Adaptive Moment Estimation), a method for gradient-based stochastic optimization, was applied to help with learning the large MNIST data set (full details can be found in Kingma & Ba, 2015), with the gradients being obtained through backpropagation. ADAM uses moving averages to estimate the first (2.4) and second (2.5) moments.

$$m_{[t]} = \beta_1 m_{[t-1]} + (1 - \beta_1) g_{[t]} \tag{2.4}$$

where $m_{[t]}$ is the estimate of the first moment, β_1 is the exponential decay rate set at 0.9, and $g_{[t]}$ is the gradient.

$$v_{[t]} = \beta_2 v_{[t-1]} + (1 - \beta_2) g_{[t]}^2 \tag{2.5}$$

where $v_{[t]}$ is the estimate of the second moment, β_2 is the exponential decay rate set at 0.999, and $g_{[t]}^2$ is the current squared gradient.

Since $m_{[t]}$ and $v_{[t]}$ are initialized at zero, the estimates are therefore biased toward zero. To correct for this, bias-corrected estimates are computed for both the first (2.6) and second (2.7) moment estimates.

$$\widehat{m}_{[t]} = \frac{m_{[t]}}{1 - \beta_1^{[t]}} \tag{2.6}$$

where $\widehat{m}_{[t]}$ is the bias corrected first moment estimate.

$$\hat{v}_{[t]} = \frac{v_{[t]}}{1 - \beta_2^{[t]}} \tag{2.7}$$

where $\hat{v}_{[t]}$ is the bias corrected second moment estimate.

These bias-corrected estimates are then used to scale the learning rate during weight updates according to (2.8).

$$w_{[t]} = w_{[t-1]} - \eta \frac{\widehat{m}_{[t]}}{\sqrt{\hat{v}_{[t]}} + \varepsilon} \tag{2.8}$$

where $w_{[t-1]}$ is the weight connections from the previous epoch, η is the learning rate set to 0.01, and ε a constant set to 10^{-8}.

Loss Function

Since the MNIST data set is a multiclass classification problem, a categorical cross entropy loss function (2.9) was used.

$$J = -\frac{1}{M} \sum_{k=1}^{N} \sum_{q=1}^{M} y_q^k \log \hat{y}_q^k \tag{2.9}$$

where $\hat{y}_q^k$ is the k^{th} scalar value output from the network for an instance of q, y_q^k is the corresponding target value, and N is the total number of classes, and M is the total number of instances in the data.

Learning

The learning for the MLP networks follows the backpropagation algorithm developed by Rumelhart et al., (1986) with a slight modification. The output error term (δ_k) is calculated according to (2.10) since the derivative of the cross-entropy loss with softmax function simplifies to $(\hat{y}_k - y_k)$ (for the complete derivation, see Appendix A). As such, the hidden layer error term was calculated according to (2.11).

$$\delta_k = (\hat{y}_k - y_k)(h_j) \tag{2.10}$$

where δ_k is the error term for the output layer.

$$\delta_j = w_{jk}(\hat{y}_k - y_k)f'(h_j^{in}) \tag{2.11}$$

where δ_j is the error for the hidden layer and w_{jk} is the weight connections between unit j in the hidden layer and unit k in the output layer.

Learning was conducted using the ADAM optimizer as outlined by the following steps:

1) Initialization of weight connections at random values from a uniform distribution ranging from -0.1 to 0.1 and initialization of the biases, gradients, and moment estimates at zero.
2) Forward propagation of the input vectors to the hidden layer with the application of the sigmoid activation function (2.1).
3) Continuation of forward propagation to the output layer with the application of the softmax activation function (2.2) to obtain class probabilities.
4) Calculation of the gradients ($g_{[t]}$) through backpropagation using the derivative of sigmoid activation function (2.3).

5) Updating the moving averages of both the first (2.4) and second (2.5) moment estimates used by the ADAM optimizer.
6) Computation of the first (2.6) and second (2.7) moment bias-corrected estimates used by the ADAM optimizer.
7) Calculation of the categorical cross entropy (2.9) by comparing the output class probabilities to the ground truths.
8) Updating of the weights and biases (2.8).
9) Training continuation until one of the following conditions are met: the training categorical cross entropy falls below a minimum value set at 0.01, or the maximum allowed training epochs are reached (set at 100 000 epochs).

The only exception occurs when a growing algorithm is used. Adding a new unit is only permitted at the end of each epoch after the network has updated its respective weight connections. The newly added hidden unit's weight connections are initialized at random values from a uniform distribution ranging from -0.1 to 0.1. They are updated along with all the other weight connections during the next epoch. The decision of when to add a new hidden unit is outlined below for the gMLP and in Simulation IV for the DNC-MLP.

Growing Algorithm

The proposed growing algorithm is inspired by single-species model with Allee effect (Sun, 2016). The original model (2.12) was cubic function with stable fixed points at $x_0 = 0$ and $x_1 = K$, and an unstable fixed point at $x_1 = \alpha$. This gave rise to the population either reaching the upper bound set by the carrying capacity or falling below the Allee threshold and decreasing until reaching extinction. In this model the intrinsic growth rate (r) is a constant representing the growth rate per capita (Tsoularis & Wallace, 2002).

$$\frac{\mathrm{d}x}{\mathrm{d}t} = rx * \left(1 - \frac{x}{K}\right)(x - \alpha) \tag{2.12}$$

where K is the carrying capacity, α the Allee threshold, and r the intrinsic growth rate.

The adapted growing algorithm proposed here is a reduced quadratic form of the original single-species model with Allee effect. In this revised version, the constant intrinsic growth rate is replaced by time-dependent performance feedback from the network in the form of error. Consequently, the revised growing algorithm, resembles the classic Verhulst logistic growth equation (Tsoularis & Wallace, 2002).

Previously, we have shown that including direct performance feedback from the network in the form of error can be used as a more self-governed method to terminate the growing process (Ross et al., 2020). Essentially, as the network learns, its performance can slow the growth rate of the hidden layer, and as the error approaches zero, so too does the growth rate. Therefore, the growth rate is in a constant state of change directly related to the network's performance. As a result of this inherent property, even though the hidden layer has a maximum carrying capacity (upper bound), the hidden unit population will grow the number of hidden units needed to converge the error close to zero. This removes the need for the user to manually tune an *a priori* stopping condition and creates what we have previously suggested as a more self-governed growing algorithm. The growth of the hidden layer population in the gMLP is dictated by (2.13).

$$\frac{\mathrm{d}h_s}{\mathrm{d}t} = E * \left(1 - \frac{h_s}{C}\right)(h_s - \alpha) \tag{2.13}$$

where C is the carrying capacity of the hidden layer environment (upper bound), h_s is the size of the hidden unit population, E is the current global error of the network according to the categorical cross entropy loss function (2.9), and α is the extinction constant (lower bound) set to 0.2.

The roots of (2.13) are the fixed points of the system. When $E = 1$, the fixed points are:

$$\frac{\mathrm{d}h_s}{\mathrm{d}t} = f(h_s) = E * \left(1 - \frac{h_s}{C}\right)(h_s - \alpha)$$

$$= 1 * \left(1 - \frac{h_s}{C}\right)(h_s - \alpha)$$

$$= \left(1 - \frac{h_s}{C}\right)(h_s - \alpha)$$

$$h_{s_0} = \alpha \text{ and } h_{s_1} = C$$

The stability of the fixed point are determined by computing the derivative of (2.13).

$$f'(h_s) = E * \left(1 - \frac{h_s}{C}\right)(h_s - \alpha)$$

$$= \frac{d}{dh_s}\left[E * \left(1 - \frac{h_s}{C}\right)(h_s - \alpha)\right]$$

Pull out the constant.

$$= E * \frac{d}{dh_s}\left[\left(1 - \frac{h_s}{C}\right)(h_s - \alpha)\right]$$

Apply the product rule.

$$= E * \left(\frac{d}{dh_s}\left[\left(1 - \frac{h_s}{C}\right)\right] * (h_s - \alpha) + \left(1 - \frac{h_s}{C}\right) * \frac{d}{dh_s}[(h_s - \alpha)]\right)$$

$$= E * \left(\left(\frac{d}{dh_s}[1] - \frac{1}{C} * \frac{d}{dh_s}[h_s]\right) * (h_s - \alpha) + \left(1 - \frac{h_s}{C}\right) * \left(\frac{d}{dh_s}[h_s] + \frac{d}{dh_s}[-\alpha]\right)\right)$$

$$= E * \left(\left(0 - \frac{1}{C}\right) * (h_s - \alpha) + \left(1 - \frac{h_s}{C}\right) * (1 + 0)\right)$$

$$= E * \left(-\frac{h_s - \alpha}{C} + 1 - \frac{h_s}{C}\right)$$

Simplified to:

$$= -\frac{E * (2h_s - C - \alpha)}{C} \tag{2.14}$$

If the derivative (2.14) at the fixed point is greater than zero, then it is considered unstable. Conversely, if the derivative at the fixed point is less than zero, then it is considered stable. For example:

For $h_{s_0} = \alpha$:

When $E = 1$, $C = 100$, and $\alpha = 0$:

$$f'(\alpha) = -\frac{E*(2h_s - C - \alpha)}{C}$$

$$f'(0) = -\frac{(1)*(2(0)-(100)-(0))}{(100)}$$

$$= -\frac{-100}{(100)}$$

$$= 1$$

$$\therefore\ f'(\alpha)|f'(0) > 0$$

For $h_{s_1} = C$:

When $E = 1$, $C = 100$, and $\alpha = 0$:

$$f'(C) = -\frac{E*(2h_s - C - \alpha)}{C}$$

$$f'(100) = -\frac{(1)*(2(100)-(100)-(0))}{(100)}$$

$$= -\frac{100}{(100)}$$

$$= -1$$

$$\therefore f'(C)|\ f'(100) < 0$$

From this example, $h_{s_0} = \alpha$ is an unstable fixed point and $h_{s_1} = C$ is a stable fixed point (for a visualization see Figure 2.4). With $C > \alpha$ and $E > 0$, the hidden population will grow to the carrying capacity (C), $\lim_{t\to\infty} h_{s_{[t]}} = C$.

During each learning epoch, this algorithm is applied to calculate the growth rate of the hidden unit population. Adding a new hidden unit ($s + 1$) to the hidden layer occurs only when the growth rate (h_s) reaches an integer value. This constraint is applied, as adding a fraction of a unit is not plausible. Additionally, given that C is asymptotic, when $C - s = 1$ and $h_s > \frac{0.99}{1}$, h_s is rounded up to the next integer.

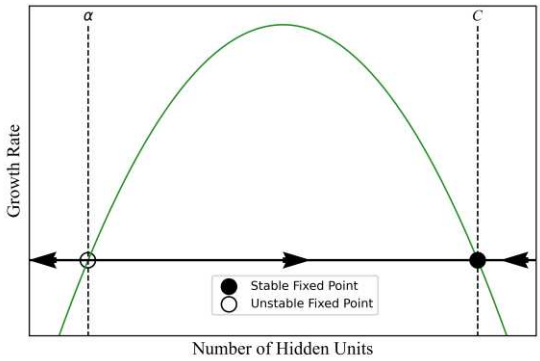

Figure 2.4. Fixed points of the growing function. With a stable fixed point at $h_s = C$ and an unstable fixed point at $h_s = \alpha$, when $E = 1$.

Simulation I: Inherent Properties of the Growing Algorithm

To validate this approach, the properties and effects of the algorithm's parameters (which are learned by the model) and hyperparameters (which are set by the user) need to be examined. The following simulations examine these effects to show the algorithm's inherent capabilities and independence from fine-tuning.

The Effect of the Extinction Constant (Lower Bound)

From the single-species model (Sun, 2016), where the algorithm is adapted, the constant α is known as the Allee threshold. If the population size falls below this threshold, the population will decrease and result in local extinction. Translated to applications in artificial neural networks, this hyperparameter endows the network with the ability to automatically prune units if the number of hidden units falls below this lower bound. In this context, a hidden layer population could go locally extinct if the hidden layer size is too small and thus falls below the threshold. To determine the exact nature of the extinction constant α, several tests were conducted on the effect of α in relation to hidden layer size initialization while trying to learn the MNIST data set. In these tests,

the extinction constant α is held constant at a value of 5, and the initial hidden layer size is varied to see the effects on the hidden layer population.

In the first test, the initial hidden layer size was set to 5 units, the same as α, and learning was conducted. This resulted in no change to hidden layer size (Figure 2.5a), a growth rate of zero throughout the training (Figure 2.5b), and the categorical cross entropy loss function to stop converging (Figure 2.5c). Next, the initial hidden layer size was set below α at 1 unit. Initializing the hidden layer population below the lower bound of 5 units set by the extinction constant results in immediate automatic pruning of the single hidden unit within 2 epochs (Figure 2.6a). Since the hidden layer now has a population size of zero, the learning is terminated. As depicted in Figure 2.6b, the growth rate becomes negative and switches to decay. This, in turn, causes the categorical cross entropy loss function to stop converging (Figure 2.6c). For the third and final test, the initial hidden layer size was set just above the 5-unit extinction constant at 6 units and manually dropped to 4 hidden units after 25 epochs. Once the hidden layer population fell below α, a gradual pruning occurred until the hidden layer population reached zero and became locally extinct (Figure 2.7a). The pruning of a unit coincides with negative growth (decay; Figure 2.7b) and the categorical cross entropy loss function to stop converging (Figure 2.7c).

To summarize, if the size of the hidden unit population is above the extinction constant (α), growth will occur as expected. If the hidden population size is on par with α, the growth rate will stay at zero, resulting in no change to the hidden layer size. Finally, if the hidden unit population is below α, local extinction will occur through gradual prunning until the hidden layer population reaches zero. In conclusion for growth to occur $\alpha \in \mathbb{R} | \ 0 \leq \alpha < h_s \ \cap \ \alpha < C$.

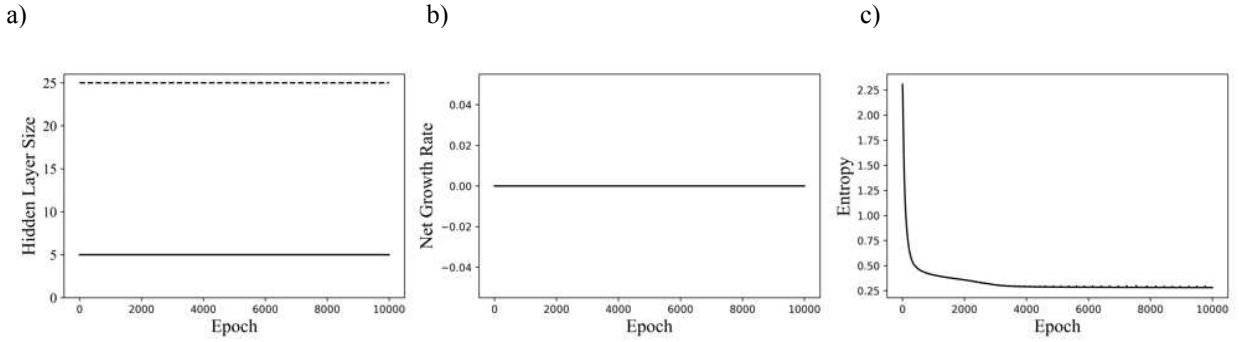

Figure 2.5. The effect of extinction (lower bound) on network hidden layer growth while learning the MNIST data set. Where C, the carrying capacity (upper bound) is set at 25, and both α (lower bound) and the initial hidden layer size are set to 5. a) Growth of the hidden layer across training epochs. b) The growth rate of the hidden layer across training epochs. c) Categorical cross entropy across training epochs.

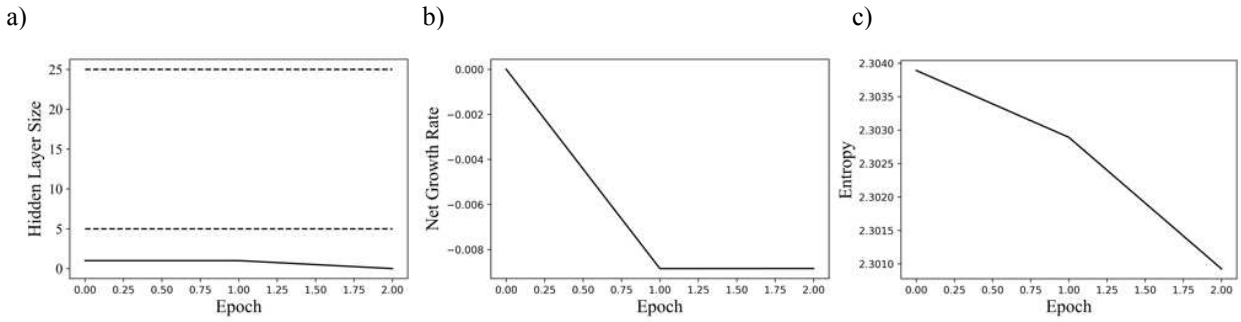

Figure 2.6. The effect of extinction (lower bound) on network hidden layer growth while learning the MNIST data set. Where C, the carrying capacity (upper bound) is set at 25, α (lower bound) is set at 5, and the initial hidden layer size is set to 1. a) Growth of the hidden layer across training epochs. b) The growth rate of the hidden layer across training epochs. c) Categorical cross entropy across training epochs.

The Effect of Weight Initialization

It is generally known that weight initialization has an impact on learning. However, the effect of weight initialization on growth is seldom examined. To determine the effect of weight initialization on the number of hidden units grown to solve the MNIST problem, various weight initializations drawn from different normal and uniform distributions were tested, and results were

averaged across 5 trials (see Table 2.1). Across all initialization methods tested, the training and testing accuracies remained high and similar. A noticeable difference can be observed in the average number of hidden units grown during learning. Specifically, smaller weight initializations lead to smaller topologies grown and fewer epochs needed. As such, weights were initialized using a uniform distribution ranging from -0.1 to 0.1, as it leads to the smallest topology with the least number of epochs needed on average.

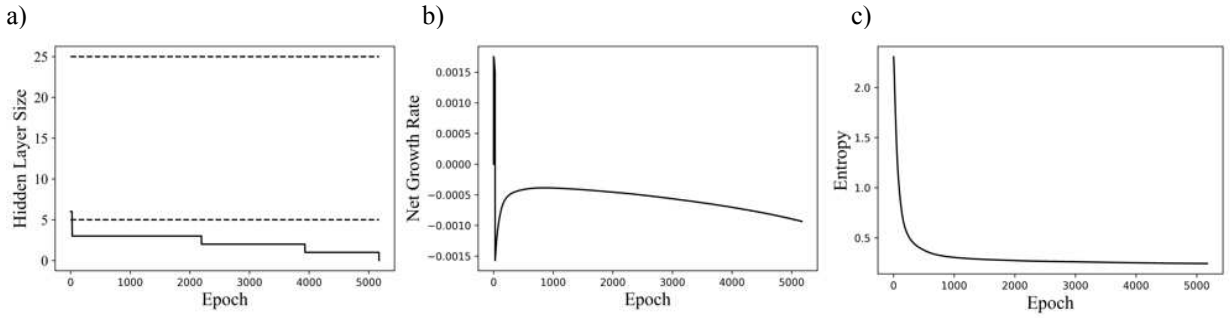

Figure 2.7. The effect of extinction (lower bound) on network hidden layer growth while learning the MNIST data set. Where C, the carrying capacity (upper bound) is set at 25, α (lower bound) is set at 5, and the initial hidden layer size is set to 6 and manually dropped to 4 after 25 epochs. a) Growth of the hidden layer across training epochs. b) The growth rate of the hidden layer across training epochs. c) Categorical cross entropy across training epochs.

To establish growth consistency by the growing algorithm while learning the MNIST data set, three trials were conducted with the random initialization of weight connections made deterministic through seeding. The network consistently grew to a final hidden layer size of 15 units in the same number of epochs (18 140) and reached the same training and testing accuracies of 94.15% and 88.35% across all three trials. These trials establish that any variability in the number of hidden units grown by the growing algorithm is due to the stochastic nature of the weight initialization.

Table 2.1

Average Results of the gMLP Using Varied Weight Initializations

Weight initialization method	Hidden layer size grown	Training accuracy (%)[a]	Testing accuracy (%)[a]	Epochs taken
Normal distribution (mean 0, std. dev. 1)	17.6 ±0.99	93.03	87.45	17 661.8 ±1837
Normal distribution (mean 0, std. dev. 0.5)	16.6 ±0.78	94.30	88.51	19 572.2 ±3663
Normal distribution (mean 0, std. dev. 0.1)	16.0 ±0.62	93.91	88.11	17 337.4 ±2789
Uniform (-1,1)	15.8 ±1.14	94.11	88.27	19 216.4 ±4235
Uniform (-0.5,0.5)	16.0 ±0.62	94.58	88.50	19 929.6 ±2583
Uniform (-0.1,0.1)	**15.4 ±0.48**	93.91	88.06	**17 509.6 ±1714**

Note: Averages were calculated across 5 trials with 95% confidence intervals. [a] When rounded to two decimals places, the 95% confidence interval was always 0.00, and as such were omitted.

Carrying Capacity (Upper Bound)

The carrying capacity (C) is the maximum hidden unit population the hidden layer can sustain (upper bound). Independent of the error feedback from the network, the carrying capacity has an expansive or compressive effect on the growth rate during learning that always converges the hidden layer size to the carrying capacity (Figure 2.8a). A lower carrying capacity results in a compressive effect on the growth rate, while in contrast, a larger carrying capacity results in an expansive effect on the growth rate. With error feedback from the network considered, a compressive effect is consistently observed. This leads to a distinct phenomenon whereby the network grows to similar final hidden layer sizes regardless of the carrying capacity used (Figure 2.8b). The average hidden layer size grown across 25 trials per carrying capacity is depicted in Figure 2.9. Within trials for each carrying capacity, it can be seen by the 95% confidence intervals that the variation of hidden layer size grown is approximately 1 unit. From this, it can be concluded that with the same carrying capacity, the network grows consistently to the same approximate hidden layer size. Furthermore, Figure 2.9 contains overlapping intervals when larger carrying capacities are used, showing that the network grows approximately to the same hidden layer size. Taken together, the carrying capacity hyperparameter does not require fine-tuning, but rather

setting it at an arbitrarily high value will result in approximately the same final hidden layer size. As such the carrying capacity is defined as $C \in \mathbb{N}_1 | \, C < \infty$.

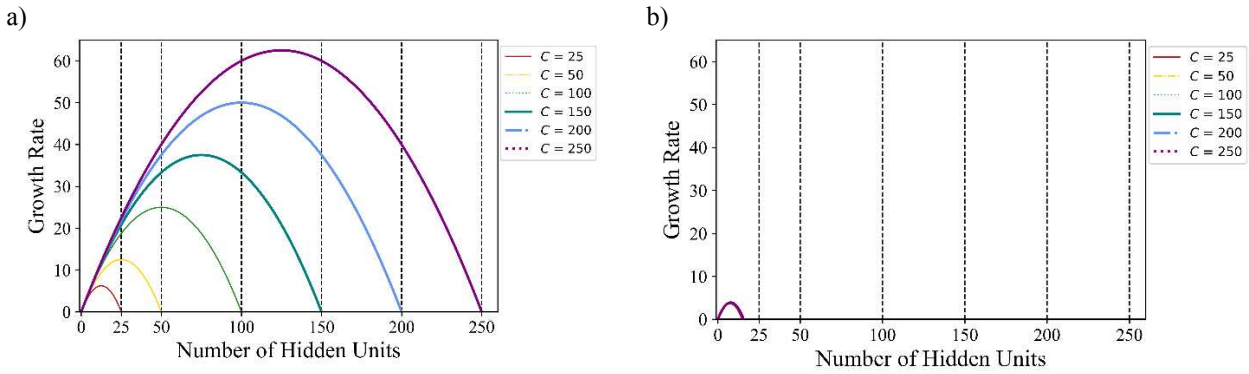

Figure 2.8. The effect of carrying capacity on the hidden unit growth rate. a) The effect of varied carrying capacities on the growth rate of hidden units when independent of error feedback from the network. b) The effect of varied carrying capacities on the growth rate of hidden units when dependent on error feedback from the network.

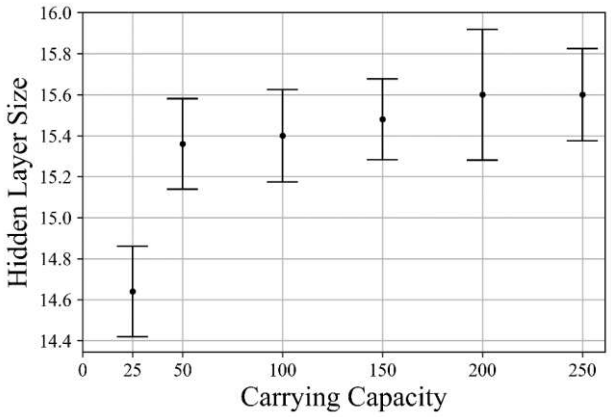

Figure 29. Average hidden layer sizes at varied carrying capacities. Averages were calculated across 25 trials per carrying capacity tested with 95% confidence interval bars depicted.

Euler Approximation - The Effect of dt

As the growing algorithm is a differential equation, a Euler approximation was applied. The Euler method employs step-wise calculations based on tangents that estimate the differential equation. This is accomplished using a time step hyperparameter (d*t*), the setting of which affects the accuracy of the estimation and the duration the model takes to run (Johnson & Chartier, 2017).

In the current estimation of the growing algorithm, the setting of this precision hyperparameter directly impacts the hidden layer size grown and the duration of training. Three dt values were tested to examine the impact of the Euler approximation on the growing algorithm while learning the MNIST data set.

An average of 25 trials for each dt tested are examined with a carrying capacity (C) of 100. A large dt of 0.1 leads to rapid growth during learning that saturates close to the carrying capacity (upper bound) with an average of 98 ±0.00 hidden units using 363.52 ±3.47 epochs. While the growth of the hidden layer size appears instantaneous, as depicted by a vertical line on this scale (see Figure 2.10), it is, in fact, a step-like function. Next, a dt of 0.01 was tested. This dt resulted in steady growth across an average of approximately 1 592 ±16.36 epochs with an average hidden layer size of 24.52 ±0.20 units. Finally, a dt of 0.001 leads to slow growth across 17 197 ±382.10 epochs, growing to an average of 15.36 ±0.25 hidden units. The effect of dt becomes clear; a smaller dt will lead the network to grow closer to the minimum number of hidden units it needs to solve a given task, in this case, the MNIST data set. In comparison, larger dt values result in rapid growth toward the maximum number of hidden units permitted by the carrying capacity. Therefore, when smaller steps are used by the Euler approximation, more accurate estimates of the growing algorithm are obtained.

Simulation II: Fixed MLP Establishing a Benchmark

In order to determine if the growing algorithm is growing the hidden layer to an ideal size for learning the MNIST data set, a proper benchmark needs to be established. To establish a proper benchmark, an MLP was tested with various fixed hidden layer sizes (fMLP). The fMLP was tested with fixed hidden layer sizes set at intervals of 10 and averaged across 25 trials (Figure 2.11). The main point of interest is the difference in epochs needed to learn the data set between

intervals. These differences establish a trade-off between the drop in training epochs from adding more hidden units. The objective is to establish when the network obtains the most significant drop in training epochs while being as conservative with hidden units as possible.

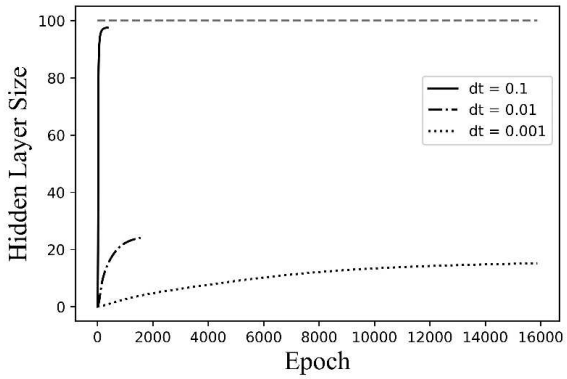

Figure 2.9. The effect of dt on the Euler approximation of the growing algorithm for a single trial. The carrying capacity (C) was set to 100 for each dt.

Instead of getting an approximation of the ideal hidden layer size from the curve alone, we attempt to establish a metric to measure this trade-off. The percent decrease in epochs taken from the start to the end of the intervals is calculated according to (2.15). This metric indicates at which interval the most significant decrease in epochs can be observed (Table 2.2). These values are then divided by the number of epochs at the start of the interval and multiplied by 100 to determine each range's overall contribution to the curve.

$$p_{decrease} = \left(\frac{t_{start} - t_{end}}{t_{start}}\right) \times 100 \tag{2.15}$$

where $p_{decrease}$ is the percent decrease in epochs, t_{start} is the average epochs taken to learn the MNIST at the start of the interval, and t_{end} is the average epochs taken to learn the MNIST at the end of the interval.

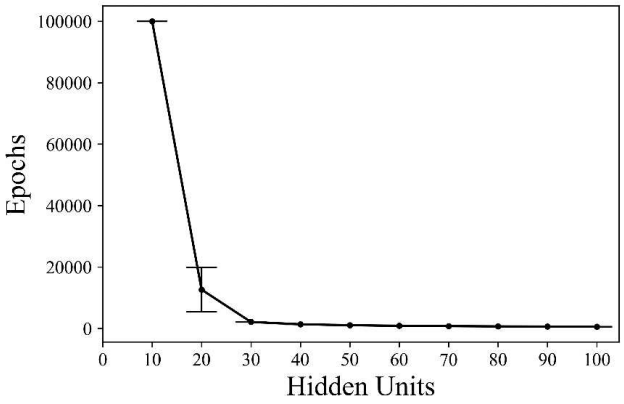

Figure 3. Average epochs taken using hidden layer size from 0-100 by intervals of 10. Averages were calculated across 25 trials for each interval tested, with 95% confidence interval bars depicted.

Table 2.2

Decrease in Epochs for Hidden Layer Size Ranges

Hidden layer size ranges	Percent decrease in epochs taken (%)	Overall contribution (%)
10-20	**87.34**	**30.73**
20-30	83.27	29.30
30-40	35.40	12.46
40-50	22.78	8.01
50-60	17.15	6.03
60-70	12.86	4.53
70-80	8.81	3.10
80-90	10.07	3.56
90-100	6.48	2.28

The most significant drop in epochs is observable from 10 to 20 hidden units (Figure 2.11; Table 2.2). The overall contribution values refer to the percent decrease in epochs taken across all intervals for the curve in general. According to these values, 30.73% of all decrease in epochs occur when going from 10 to 20 hidden units. As such, we expect the best trade-off for units added and learning epoch decrease to occur in the range of 10-20 hidden units.

In addition to our attempt to establish a general benchmark on the number of hidden units needed, we replicate the work of Cai et al., which utilized SVD to estimate the number of hidden units needed by a single hidden layer feedforward neural network for the MNIST data set. SVD is

applied directly to the input data set, and the main eigenvalues are used as the number of units in the hidden layer. Then the eigenvalue ratio is calculated according to (2.16) to indicate how much variance is explained by the number of eigenvalues selected (for full details, see Cai et al., 2019). Compared to our benchmark based on the decrease in epochs, according to SVD, using 10-20 hidden units should explain 69-78% of the variance in the MNIST data set (see Table 2.3).

$$\delta_j = \frac{|\lambda_j|}{\sum_{i=1}^{N} \lambda_i} \tag{2.16}$$

where λ is the main Eigenvalues

Table 2.3

Number of Different Main Eigenvalues and Associated Ratios - MNIST Data Set

Ratio	43%	69%	70%	74%	75%	76%	77%
Number of eigenvalues	1	10	11	15	16	17	18
Ratio	78%	80%	83%	85%	90%	95%	99%
Number of eigenvalues	20	23	29	34	53	103	281

Simulation III: Growing vs. Fixed- Single Trial Comparisons

To test if the established general benchmark gave a good approximation, a single trial was conducted using the gMLP with seeded weights. Based on the previous simulations, the carrying capacity (*C*) was set to a high value of 100, the initial hidden layer size was set to 1 unit, and the α hyperparameter was set to less than the initial hidden layer size at 0.2 to remove the possibility of extinction and promote growth.

As shown in Figure 2.12a, the hidden layer size grew gradually in a step-like fashion across 18 140 epochs, eventually growing to 15 hidden units (see Table 2.4). The observed steps in hidden layer size coincide with peaks in the network growth rate (Figure 2.12b) that drive a new unit's addition. Specifically, early during training, the growth rate is high, promoting rapid growth when the entropy is high to reduce this error rapidly. This is followed by a gradual downward slope of the growth rate with occasional spikes to drive the addition of new units to the network when

needed. Additionally, these steps coincide with almost "micro" spikes observable in the entropy loss function (Figure 2.13a). These "micro" spikes are expected and directly result from the added randomness to the network from the randomly initialized weight connections when a new hidden unit is added. Overall, the gMLP reached a training accuracy of 94.15% and a testing accuracy of 88.35% on the MNIST data set. To further assess the results of the single trial of the growing algorithm, the network was compared to single trials using a fMLP with the same seed for weights. The fMLP was tested using fixed hidden layer sizes of 100, 50, 25, and 15 units. The fMLP successfully learned the MNIST data minimizing the entropy loss function in all cases except using 15 hidden units (Figure 2.13b). Using 15 hidden units, the fMLP was just short of reaching the minimum entropy of 0.01 and reached the maximum number of training epochs. The results of the fixed trials can be found in Table 2.4.

From the initial prediction using the overall contribution metric of percent epoch decrease on fixed hidden layer intervals (see Simulation II), it was expected that the hidden layer should grow in the range of 10-20 hidden units to successfully learn the MNIST data set while accommodating the trade-off between units added and epochs taken and explaining 69-78% of the variance in the data set. The gMLP successfully grew to 15 hidden units, in line with our prediction. As indicated by the results in Table 2.4, the fMLP took fewer epochs to learn and reached higher training and testing accuracies.

Table 2.4

Comparison of Single Trial Results Between Fixed and Growing Hidden Layers

Network	Hidden layer size	Training accuracy (%)	Testing accuracy (%)	Epochs taken
fMLP	15	99.14	90.08	100 000
fMLP	25	98.46	94.15	3 049
fMLP	50	97.83	95.44	1 043
fMLP	100	97.47	95.93	584
gMLP	15	94.15	88.35	18 140

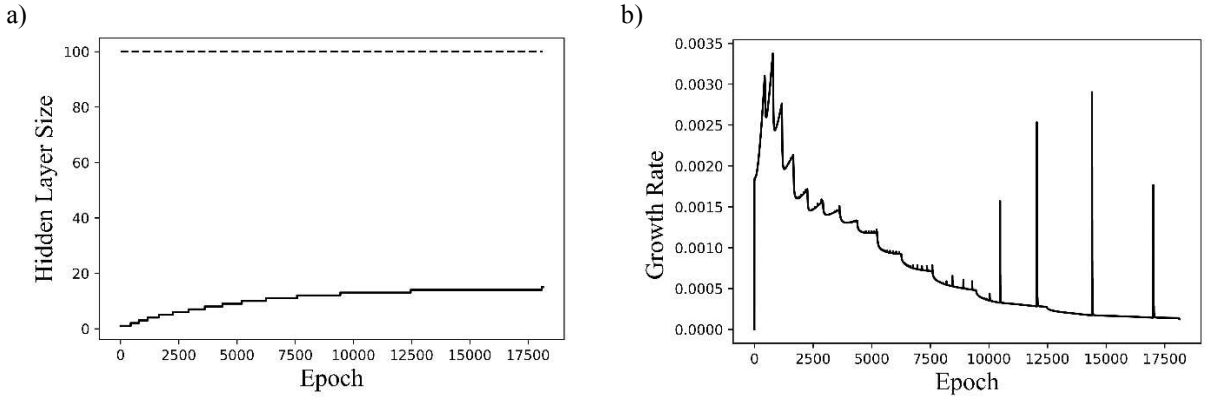

Figure 4.11. Single trial of the gMLP on the MNIST data set. a) The size of the hidden layer across training epochs. b) The overall growth rate across training epochs.

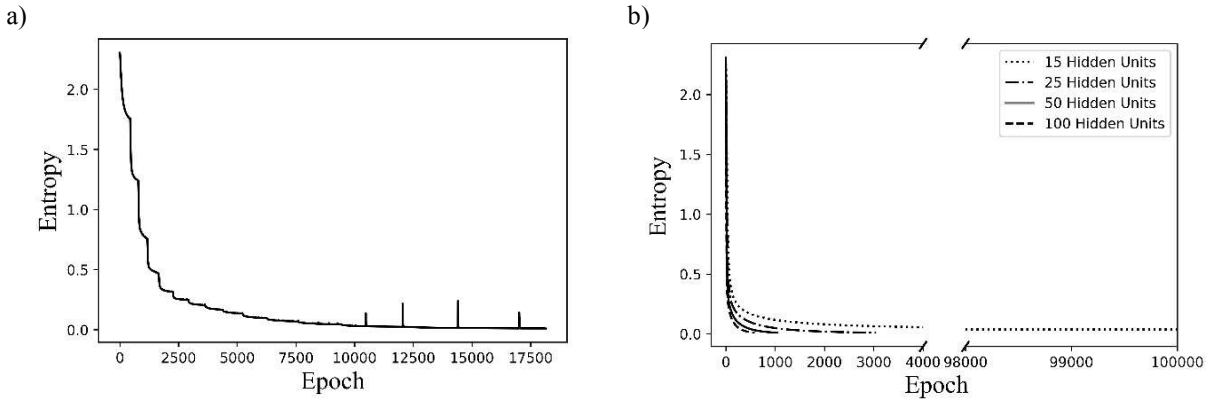

Figure 5.12. Categorical cross entropy across training epochs. a) Categorical cross entropy across training epochs for a single trial of the gMLP. b) Categorical cross entropy across training epochs for single trials with fixed hidden layers sizes of 15, 25, 50, and 100 units in the fMLP.

Simulation IV: Dynamic Node Creation (DNC)- Single Trials

The population dynamics inspired growing algorithm explored here is classified in the taxonomy of constructive algorithms as a single-valued state transition mapping that updates all weights after the addition of a new unit training scheme, the same as DNC. As such, the population dynamics inspired growing algorithm draws many parallels from DNC. With DNC, the network is initialized with a single hidden unit. New units are added iteratively when the error curve flattens

within a specific time frame (i.e., window width; Ash, 1989). The whole network is then retrained after the addition of a new unit. This requires an *a priori* finely-tuned trigger slope, which measures the error curve's flatness, whereby a new unit is added if the error falls below this value. Additionally, the user must finely tune the window width, the number of epochs over which the flatness of the error curve is compared to the trigger slope. According to these finely-tuned hyperparameters, a new unit can only be added to the network if the error has flattened and a minimum number of epochs have passed. While both DNC and our growing approach share fundamental similarities, it is the method whereby the decision of when to add a new unit that sets them apart. The growing algorithm tested here employs population dynamics to govern growth, while the DNC method employs systematic growth dictated by hyperparameters that require fine-tuning. As such, a comparison between fundamentally similar growing approaches that differ by methods that govern growth is explored.

To provide a better comparison, the original DNC approach was adapted to better suit the learning of the MNIST data set. This was primarily done to accommodate the shift to categorical cross entropy as a global error measure. The conditions that govern adding a new unit remain the same as the original (Ash, 1989). A new unit is added to the hidden layer if the categorical cross entropy loss curve flattens out below the trigger slope (Δ_T) and if enough epochs have passed since the last addition, determined by (2.17) and (2.18), respectively. The critical difference is determining when to cease adding new units to the network. The only way to terminate the iterative growing process is if one of the following conditions are met: the error falls below the desired cross entropy cut-off (2.19), or the maximum number of epochs set at 100 000 is reached. With this adaptation, the DNC approach was integrated into a single hidden layer MLP (DNC-MLP) and tasked with classifying the MNIST data set.

$$\frac{\rho_{[t]} - \rho_{[t-z]}}{\rho_{[t_0]}} < \Delta_T \tag{2.17}$$

where $\rho_{[t]}$ is the average entropy at time *t* across all output units, *z* is the window width in epochs over which the slope is determined, Δ_T is the trigger slope.

$$t - z \geq t_0 \tag{2.18}$$

where t_0 is the time in epochs when the last unit was added to the hidden layer.

$$\rho_{[t]} \leq c_\rho \tag{2.19}$$

where c_ρ is the desired cut-off for categorical cross entropy set at 0.01.

Since the hyperparameters for the trigger slope and window width require fine-tuning, any inappropriate values could lead to growing larger than the minimal number of units needed or prevent the network from growing beyond the minimal amount (Ash, 1989). As such, different trigger slopes and window widths were tested using the same random seed for weight initialization to determine appropriate values for these hyperparameters for the MNIST classification problem. The trigger slope (Δ_T) was set to either 0.05 or 0.01, and the window width (z) was set to either: 50, 100, 500, or 1000. The effects of window width on the number of hidden units grown for each trigger slope are depicted in Figure 2.14a and Figure 2.14b for a single trial, respectively. As can be seen by the results in Table 2.5, not many differences can be distinguished from a single trial.

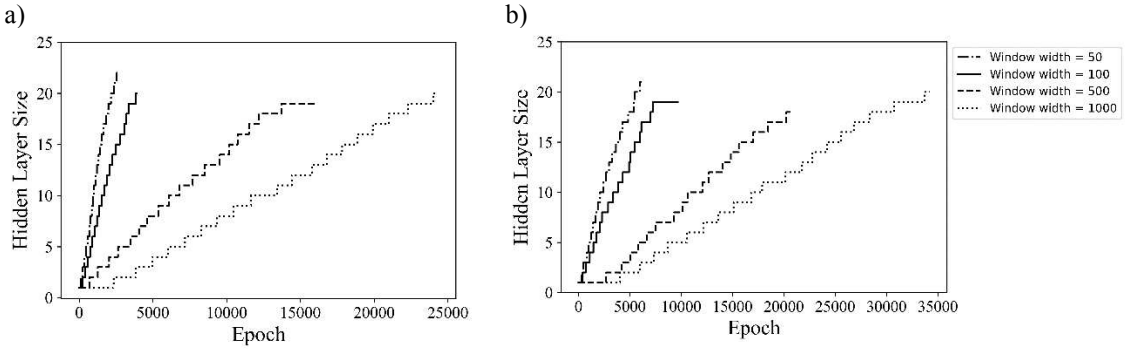

Figure 6.13. Changes in hidden layer size across training epochs using varied window widths for DNC in a MLP on the MNIST data set. a) Changes in hidden layer size across training epochs using different window widths with a

trigger slope of 0.05. b) Changes in hidden layer size across training epochs using different window widths with a trigger slope of 0.01.

Simulation V: ALL Avg (25 trials)

To ensure that the previous single trial simulations were consistent, an average of 25 trials was taken for the gMLP, fMLP, and DNC-MLP (see Table 2.6). Overall, the same conclusion as with the single trials can be drawn when comparing the average results of the gMLP compared to the fMLP. The fMLP took fewer epochs to learn, reaching higher training and testing accuracies. The only exception was with 15 hidden units, where the fMLP was just short of reaching the minimum entropy of 0.01 and reached the maximum number of training epochs. However, the hidden layer size used in the fMLP was still decided via a trial-and-error approach. Alternatively, the comparison of the gMLP to the DNC-MLP yields interesting results. The comparison of the two methods shows higher training and testing accuracies for the gMLP across all DNC runs. More notable is that the gMLP can grow a smaller topology than the DNC-MLP. Only afterward, when the window width was set to 3000 for a trigger slope of 0.01, did the DNC-MLP grow close to the gMLP at 16 hidden units. This highlights the reliance on fine-tuning hyperparameters with the DNC method. However, as shown, there are no clear methods other than a trial-and-error approach for tuning these hyperparameters that will maximize performance while growing to a minimal topology.

Simulation VI: Novel Data Set

To verify the conclusions drawn across the MLP models tested, a novel data set was introduced. Contrary to the multiclass classification of the MNIST data set, the Skin Segmentation data set represents a binary classification problem. The data set uses only three input features of RGB samples from face images to classify them as skin or non-skin samples. This significant

reduction of input features and class outputs compared to the MNIST should result in considerably smaller required topologies. A random subsample of 50 000 RGB samples with equal amounts of skin and non-skin classes was selected, with 40 000 samples used for training and 10 000 for testing. Aside from a novel data set, a decrease in the number of output units to two, and adjusted hidden layer sizes for the fMLP, no changes were made to any MLP or growing method.

Table 2.5

Results of Using DNC with Different Hyperparameters

Network	Measures				
	Window width	Hidden layer size	Training accuracy (%)	Testing accuracy (%)	Epochs taken
DNC-MLP (Δ_T= 0.05)	50	22	88.34	84.71	2 588
DNC-MLP (Δ_T= 0.05)	100	20	88.06	84.45	3 905
DNC-MLP (Δ_T= 0.05)	500	19	90.98	86.04	16 127
DNC-MLP (Δ_T= 0.05)	1000	20	87.59	83.11	24 333
DNC-MLP (Δ_T= 0.01)	50	21	89.98	85.67	6 164
DNC-MLP (Δ_T= 0.01)	100	19	91.11	86.18	9 671
DNC-MLP (Δ_T= 0.01)	500	18	85.47	81.60	20 837
DNC-MLP (Δ_T= 0.01)	1000	20	87.41	82.85	34 201

Table 2.6

Average Results of the gMLP, fMLP, and DNC-MLP-MNIST Data Set

Network	Measures				
	Window width	Hidden layer size	Training accuracy (%)[a]	Testing accuracy (%)[a]	Epochs taken
gMLP	**-**	**15.4 ±0.22**	**94.06**	**88.30**	**17 898.8 ±657**
fMLP	-	100	97.42	95.84	582.2 ±7
fMLP	-	50	97.82	95.43	1 055.4 ±8
fMLP	-	25	98.47	94.09	3 146.0 ±72
fMLP	-	**15**	99.24	90.60	**100 000±0**
DNC-MLP (Δ_T= 0.05)	50	21.72 ±0.26	88.15	84.65	2 580.5 ±21
DNC-MLP (Δ_T= 0.05)	100	19.04 ±0.21	88.78	84.78	4 227.7 ±59
DNC-MLP (Δ_T= 0.05)	500	19.36 ±0.45	89.07	84.54	14 521.7 ±330
DNC-MLP (Δ_T= 0.05)	1000	18.88 ±0.42	88.84	84.27	25 326.0 ±650
DNC-MLP (Δ_T= 0.01)	50	19.16 ±0.50	89.77	85.35	6 194.4 ±143
DNC-MLP (Δ_T= 0.01)	100	19.76 ±0.70	89.51	84.96	9 626.6 ±320
DNC-MLP (Δ_T= 0.01)	500	19.44 ±0.55	86.26	82.10	21 547.6 ±557
DNC-MLP (Δ_T= 0.01)	1000	**18.68 ±0.53**	86.10	82.04	32 680.4 ±712

Note: Averages were calculated across 25 trials with 95% confidence intervals. [a] When rounded to two decimals places, the 95% confidence interval was always 0.00, and as such were omitted.

As can be seen by comparing the average trial results of Table 2.7 using the Skin Segmentation data set and the results in Table 2.6 using the MNIST data set, there is not much

difference regarding the overall conclusion. The hidden layer size of fMLP was still determined by a trial-and-error approach, and concerning the DNC-MLP, larger window widths resulted in smaller topologies. However, a window width of 500 now achieves the same minimal size as a window with 1000 epochs. This highlights that a switch in the task could require re-tuning these hyperparameters.

Table 2.7

Average Results of the gMLP, fMLP, and DNC-MLP- Skin Segmentation Data Set

Network	Measures				
	Window width	Hidden layer size	Training accuracy (%)[a]	Testing accuracy (%)[a]	Epochs taken
gMLP	**-**	**3 ±0.00**	**97.63**	**97.65**	**13 237.6 ±6**
fMLP	-	5	98.06	98.08	3 431.8 ±53
fMLP	-	4	98.08	98.10	3 652.7 ±49
fMLP	-	3	98.06	98.07	3 908.7 ±52
DNC-MLP (Δ_T= 0.05)	50	8.8 ±0.66	97.03	97.12	1 701.6 ±35
DNC-MLP (Δ_T= 0.05)	100	5 ±0.00	97.02	97.13	1 996.5 ±22
DNC-MLP (Δ_T= 0.05)	**500**	**3 ±0.00**	97.51	97.62	3 434.1 ±22
DNC-MLP (Δ_T= 0.05)	**1000**	**3 ±0.00**	97.42	97.55	4 393.1 ±11
DNC-MLP (Δ_T= 0.01)	50	4 ±0.00	97.10	96.99	2 236.9 ±21
DNC-MLP (Δ_T= 0.01)	100	3.28 ±0.18	97.60	97.50	3 138.8 ±13
DNC-MLP (Δ_T= 0.01)	**500**	**3 ±0.00**	97.31	97.18	4 508.8 ±21
DNC-MLP (Δ_T= 0.01)	**1000**	**3 ±0.00**	97.23	97.09	6 061.0 ±40

Note: Averages were calculated across 25 trials with 95% confidence intervals. [a] When rounded to two decimals places, the 95% confidence interval was always 0.00, and as such were omitted.

Discussion

In this study, we sought to validate the efficacy of the self-governed nature of the growing algorithm inspired by population dynamics. We defined more self-governed as a constructive technique that dictates when new hidden units should be added while limiting the user's role in fine-tuning hyperparameters *a priori*. We first examined the inherent properties of the algorithm by conducting a series of simulations to determine the effect that varying hyperparameters had on the number of hidden units grown in a single hidden layer MLP using the MNIST data set.

The extinction constant, as previously discussed, does not require fine-tuning. This hyperparameter adds an intrinsic method to remove a population during learning, the value of

which only matters in relation to the current hidden layer size. If the value is less than the population size, growth will ensue, while greater than the population will result in population extinction. Next, it was shown that smaller weight initializations lead to smaller hidden layer topologies and that any variation in the number of hidden units grown directly results from the added randomness from the weight connections.

The carrying capacity, which acts as an upper bound on the population size, is the one hyperparameter that one would assume needs to be carefully tuned to a given problem. However, it was shown that fine-tuning the carrying capacity is arbitrary, as setting it to a high value will result in approximately the same number of hidden units grown. This is due to the incorporation of error feedback from the network that modulates the growth rate causing convergence to similar final hidden layer sizes regardless of the carrying capacity value. This adds the self-governance property to the algorithm, with the carrying capacity acting as a failsafe to prevent growing outbound.

Finally, smaller time steps of the Euler approximation result in smaller topologies. It is important to note that this time step hyperparameter is for altering the precision of the estimate made by the Euler approximation and not part of the growing algorithm itself. This value should be set in accordance with the desired objective but remain consistent across tasks. For instance, if the objective was to obtain a minimal topology, a smaller time step should be used. If size was no concern, larger values of dt could be used to promote rapid growth and reduce learning time. These findings suggest that the constructive algorithm applied here limits the user's role in fine-tuning hyperparameters *a priori* and, as such, by our definition can be considered more self-governed.

Next, the efficacy of the growing algorithm was evaluated. First, a general benchmark was established to determine if the number of hidden units grown was appropriate for classifying the

MNIST data set. Using the percent decrease in learning epochs derived from changes in hidden layer size and SVD as estimates, we found that the growing algorithm grew to an appropriate size of approximately 15 hidden units for solving the MNIST. Secondly, the algorithm's classification performance was directly compared to identical MLPs that used fixed topologies and growth governed by DNC. The growing algorithm had slightly lower performances and took longer than its fixed topology counterpart. However, no trial-and-error guesswork was required. Compared to the other constructive approach, the gMLP grew to a smaller topology and reached better performances. The distinction of when a new hidden unit is added highlights the advantage of a more self-governed approach. As shown, with DNC, there is still heavy fine-tuning of the hyperparameters *a priori*, which can yield different results and ultimately is still task-dependent.

The growing algorithm presented here offers more self-governed growth, which provides an effective general solution automatically tailored to the task. The benefits of a more self-governed constructive approach are self-evident. It can automatically grow a feedforward network to an appropriate size to solve a given task without tedious fine-tuning. Replacing the user's role in designing topologies with methods that enable a system to manage its own growth can endow systems with adaptable learning that can apply to many tasks across different environments, an artificial general intelligence (Thórisson, 2012).

In the current study, the stopping conditions were strict and limited to reaching a minimum entropy of 0.01 or reaching the maximum allowed training epochs of 100 000. This was done purposefully to allow a focused examination of growth. However, typically other methods, such as early stopping, are implemented to achieve the best performance while avoiding overfitting. In such a case, the fMLP that used only 15 hidden units would have successfully learned the MNIST in the allotted number of training epochs. This opens the possibility for the presented growing

approach to be used as a tool to determine an appropriate fixed topology without the need for repeated guesswork and trial and error. In this context, rather than perform trial-and-error when trying to pre-determine hidden layer size for a fixed network, the growing algorithm can be run once and automatically provide an appropriate number to initialize a fixed hidden layer size. Table 2.6 shows that setting the fMLP to 15 hidden units offers little change to training accuracies compared to 100, 50, or 25 hidden units. While there is an upward trend in training epochs, an advantage in conserving computational resources is evident. A drop from 100 units to 15 units saves 85 units and the 666 400 associated weight connections from the input layer, bias, and going to the output layer. For a drop from 50 to 15 units, 35 units, and 274 400 associated connections are conserved. Finally, a drop from 25 to 15 units, 10 units, and 78 400 associated connections are conserved. The conservation of computational resources could be even greater with other tasks that have even higher dimensional inputs.

A specific limitation to the self-governing property of the current approach is the reliance on a measure of global error from the network to mitigate growth rate. The exact nature of using different measures of global error on the growth rate remains to be investigated. Previously, we have shown success using mean-squared error (Ross et al., 2020) and categorical cross entropy in the current study. However, no comparison has yet been made. Furthermore, the use of global error limits the biological plausibility of the algorithm.

The inherent properties of the growing algorithm were investigated in the well-known MLP. As discussed previously, the complexity of the network topology was constrained to a single hidden layer to allow a detailed and controlled investigation. However, deeper neural networks can utilize more complex functions (Bianchini & Scarselli, 2014), have increased representational power, and are less prone to getting trapped in local minima (Lecun et al., 2015). An interesting

future step would be to investigate the extension of the growing algorithm presented here to not only grow network width but depth as well. This would allow the algorithm to be tested on more complex data sets and in deeper neural networks.

Conclusion

Dynamic growth inspired by population dynamics offers a more self-governed alternative to *a priori* hyperparameters requiring fine-tuning to decide when to add new hidden units. It can be used independently in a constructive approach or as a tool to estimate the number of hidden units for a fixed topology and circumvent the need for repeated trial-and-error. The potential applications of this more self-governed growing algorithm are far-reaching and merit further investigation.

Chapter 3

Dynamic Multilayer Growth: Parallel vs. Sequential approaches

Abstract

The decision of when to add a new hidden unit or layer is a fundamental challenge for constructive algorithms. It becomes even more complex in the context of multiple hidden layers. Growing both network width and depth offers a robust framework for leveraging the ability to capture more information from the data and model more complex representations. In the context of multiple hidden layers, should growing units occur sequentially with hidden units only being grown in one layer at a time or in parallel with hidden units growing across multiple layers simultaneously? The effects of growing sequentially or in parallel are investigated using a population dynamics-inspired growing algorithm in a multilayer context. A modified version of the constructive growing algorithm capable of growing in parallel is presented. Sequential and parallel growth methodologies are compared in a three-hidden layer MLP on several benchmark classification tasks. Several variants of these approaches are developed for a more in-depth comparison based on the type of hidden layer initialization and the weight update methods employed. Comparisons are then made to another sequential growing approach, Dynamic Node Creation. Growing hidden layers in parallel resulted in comparable or higher performances than sequential approaches. Growing hidden layers in parallel promotes growing narrower deep architectures tailored to the task. Dynamic growth inspired by population dynamics offers the potential to grow the width and depth of deeper neural networks in either a sequential or parallel fashion.

When deciding the architecture of an ANN, a fundamental question arises; is it better to use a shallow and wide network or deep and narrow? This problem is commonly known as the trade-off between *width* and *depth*. Where *width* refers to the number of units in a hidden layer and *depth* refers to the number of hidden and output layers (Zhong et al., 2019).

Previously, it was shown that multilayer feedforward networks with a single hidden layer and a large enough width could approximate any continuous function, making these shallow networks universal approximators (Funahashi, 1989; Hornik et al., 1989). Despite this, the ability of deeper architectures to learn more complex, distributed, and sparse representations make them more powerful than their shallow counterparts (Bianchini & Scarselli, 2014; Eldan & Shamir, 2016; Lecun et al., 2015; Zhong et al., 2019). With the increase in network depth, learning more abstract and complex representations can occur, allowing the network to discriminate inputs better (Bengio et al., 2013; Lecun et al., 2015). This brings us back to the trade-off, is depth more valuable than width? Bianchini and Scarselli (2014) used Betti numbers, a topological measure, to compare shallow and deep feedforward networks. They showed that a deep neural network with the same number of hidden units as its shallow counterpart could realize more complex functions. Eldan and Shamir (2016) demonstrated that for a fully connected feedforward network with a linear-output unit, "depth-even if increased by 1- can be exponentially more valuable than width for standard feedforward networks." With larger data sets and more powerful graphics processing units (GPUs), the increased representational power and faster training of deep neural networks have made them valuable tools. Deep neural networks have been applied to many different problems, including computer vision, object detection, and electroencephalogram (EEG) classification, to name a few (for reviews, see Craik et al., 2019; Guo et al., 2016; Lecun et al., 2015; Zhao et al., 2019).

A network's depth and width affect the ANN's ability to generalize. If the network architecture is too large, the model may overfit the data and cause poor generalization. Conversely, if the network architecture is too small, the model may underfit the data and cause over-generalization. (Curteanu & Cartwright, 2011; Islam et al., 2009b; Kwok & Yeung, 1997; Liu et al., 2002; Parekh et al., 2000). This reinforces the need to find optimal architectures that match a given task complexity. Currently, the best practice is to use fixed neural architectures found using a trial-and-error approach. This is due to its simplicity of implementation. However, this process can be temporally cumbersome and there is no guarantee that an optimal or near-optimal topology will be found (Curteanu & Cartwright, 2011; Parekh et al., 2000; Zemouri et al., 2018; Zemouri et al., 2020).

An alternative to using fixed architectures found by a trial-and-error approach is to use adaptive neural architectures. With adaptive neural architectures, the idea is to have a dynamic architecture that is updated during training (Pérez-Sánchez et al., 2018). Several strategies have been proposed and can be broadly classified into three categories: constructive algorithms, pruning algorithms, and hybrid methods.

Constructive algorithms involve starting with a small network architecture (typically one hidden unit) and gradually adding connections, units, or layers during training to match task complexity (Boughrara et al., 2016; Khan et al., 2020; Pérez-Sánchez et al., 2018). Conversely, pruning algorithms involve starting with a larger network where connections and units are pruned (removed) during the learning process to match task complexity (for surveys, see Augasta & Kathirvalavakumar, 2013; Blalock et al., 2020; Hoefler et al., 2021; Reed, 1993). Hybrid methods combine constructive and pruning algorithms. These methods typically involve using a constructive algorithm to grow the neural architecture first, then prune the subsequent architecture,

or simultaneously grow and prune during the learning process (Sharma & Chandra, 2010a). While using a hybrid approach is appealing and has had great success (Dai et al., 2019; Han et al., 2017; S. Mohamed et al., 2021; Narasimha et al., 2008a; Thivierge et al., 2003; Zemouri et al., 2020); the focus of the present article is on constructive algorithms.

Constructive algorithms offer the possibility of compact architectures as an alternative to the trial-and-error approach when designing architectures. The focus of constructive algorithms has primarily been on growing the width of a single hidden layer. Examples of applications with this focus include regression problems (Sadreddin & Sadaoui, 2021; Sharma & Chandra, 2010b), classification (Ash, 1989; Bertini & Do Carmo Nicoletti, 2009; Boughrara et al., 2016; Fontes & Embiruçu, 2021; Hernández-Espinosa & Fernández-Redondo, 2002; Islam et al., 2009a, 2009b; Kamruzzaman et al., 2004; Liu et al., 2002; Ma & Khorasani, 2004b; Masmoudi et al., 2011; Siddiquee et al., 2010; José L Subirats et al., 2010; Susan & Dwivedi, 2014; Wu et al., 2015; Young & Downs, 1998), and image segmentation (Ma & Khorasani, 2002) to name a few (for a list of more applications, see Khan et al., 2020). Conversely, approaches based on CC have offered a constructive approach that focused on growing network depth instead of width. These approaches use a cascaded architecture where the network grows many hidden layers the width of a single unit (Besnard et al., 2007; Fahlman & Lebiere, 1990; Qiao et al., 2016; Shultz, 2012; Thivierge et al., 2003; Wu et al., 2019).

Growing both network width and depth offers a robust framework for leveraging the ability to capture more information from the data and model more complex representations. There have been numerous approaches for determining when new hidden units and layers should be added (Aran & Alpaydın, 2003; Islam & Murase, 2001; Lehtokangas, 1999; Ma & Khorasani, 2003; Parekh et al., 2000; Puma-Villanueva et al., 2012). For instance, a measure of network

significance to quantify generalization power has been proposed for growing network width in combination with drift detection for growing depth (Ashfahani & Pratama, 2019; Pratama et al., 2020). Others have examined whether the error or loss has stopped changing by a predetermined threshold value. Baluja and Fahlman (1994) proposed a sibling/descendant CC, where once the error stops changing, candidate units can be added to the same layer (siblings) or a new layer (descendant). The candidate pool is made of both types of units, and whichever unit reduces the residual network error the most is added to the network. Zemouri et al., (2020) introduced the GP-DLNN that uses a penalty term to and compared to a threshold value. If the penalty is smaller than the threshold, a new unit is added, and a new layer is added if larger. Similarly, the CCG-DLNN, a constructive approach grounded in CC, also used the same approach to decide if candidate units should be added to the more recent layer or a new layer (Mohamed et al., 2021). Another approach is to set the maximum number of hidden units and layers in advance, as with the evolutionary algorithm for building deep neural networks (Zemouri et al., 2018; Zemouri, 2017). This approach calculates and compares the error at each step to a threshold value. Growth is complete if the error converges or the maximum allowed structure is reached. Only if the error convergence is not reached and the maximum number of hidden units for that layer is reached will a new layer be added (Zemouri et al., 2018; Zemouri, 2017).

When employing constructive algorithms, serval challenges have been identified and should be considered (Sharma & Chandra, 2010a; Zemouri et al., 2020):

- What are the criteria for adding a new unit or layer to the network?
- How to connect a newly added hidden unit?
- How should the weights be initialized?
- What training scheme should be used (re-train the whole network vs. freezing)?

- When to stop adding hidden units (What are the convergence criteria)?

Here, we focus on the first challenge: the criteria for adding a new unit or layer to the network. Specifically, we are interested in *when a new hidden unit or layer should be added*. In the context of multiple hidden layers, should growing units occur sequentially or in parallel across hidden layers? With a sequential growth methodology, we refer to only being able to grow hidden units in one hidden layer at a time. The examples of constructive algorithms that grow both width and depth mentioned previously fall under this methodology. Contrarily, with a parallel growth methodology, we refer to being able to grow hidden units across multiple layers simultaneously. This methodology is scarcely found in the literature, with a modular network approach being the most evident. Guan and Li (2002) used such an approach. They broke benchmark classification problems into simpler sub-problems using task decomposition and output parallelism. Modules were then grown and trained in parallel to solve each sub-problem. The resulting modules that could solve the sub-problems were merged into a modular network.

Previously, we introduced a constructive growing algorithm that provides a more self-governed alternative for deciding when a new unit should be added to a single hidden layer multilayer perceptron (Ross et al., 2020). This approach was inspired by population dynamics (Sun, 2016) and considered the hidden units as a population and the hidden layer as the environment they exist in. This endowed the hidden layer with a carrying capacity, the maximum population the hidden layer environment can sustain (an upper bound on the layer width). This allowed the application of population dynamics to provide a hidden unit population growth rate for the hidden layer. Combining the carrying capacity and direct performance feedback from the network created a built-in dynamic, more self-governed method for growing the hidden layer, simultaneously preventing growing outbound. A natural extension of this work would be to grow

both width and depth. Since each hidden layer is treated as having its hidden unit population, should the growth rate of each population be considered individually in a sequential fashion or simultaneously in a parallel fashion?

We propose investigating the effects of growing sequentially or in parallel using our population dynamics inspired growing algorithm in a multilayer context. To achieve this, we create a modified version of our constructive growing algorithm capable of growing in parallel. We then test the methodologies of sequential and parallel growth in a three-hidden layer multilayer perceptron on several benchmark classification tasks. We test several variants of these approaches for a more intricate comparison based on the hidden layer initialization and the training scheme. Comparisons are also made to another sequential growing approach, Dynamic Node Creation (DNC; Ash, 1989), that employs the common methodology of iteratively adding units based on the error curve flattening within a specific time frame.

The remainder of the chapter is divided as follows: Section II introduces the growing approaches employed (sequential growth, parallel growth, DNC growth), the network's architecture and variants, and the learning procedure. In Section III, we describe the results of several benchmark classification tasks. Finally, in Sections IV and V, we discuss and conclude the study's findings.

Methods

Growing

Sequential Growth.

The dynamics of the sequential growing algorithm are characterized by having only one hidden layer able to grow at a time. In this context, the network is not able to grow the subsequent layer ($\ell + 1$) until the current layer (ℓ) has reached the maximum number of hidden units that it

can sustain as dictated by the carrying capacity (C). A visualization of the sequential growing method's effect on growth rate and changes in hidden layer sizes for a three-hidden layer MLP is depicted in Figure 3.1. Sequential growth of the hidden layers is dictated by (3.1). This algorithm was inspired by the Single-species model with the Allee effect (Sun, 2016). It is used to calculate the change in the size of the hidden unit population (i.e., growth rate; $\frac{\mathrm{d}h_s}{\mathrm{d}t}$) for each hidden layer (ℓ). Incorporated into the algorithm is the global network error (E) as calculated by the loss function. This allows the network's performance to modulate the growth rate of the hidden unit population during learning. The result is that regardless of what the carrying capacity (C) is set to, the hidden unit population will grow based on the needs of the network. As such the carrying capacity is defined as $C \in \mathbb{N}_1 | \, C < \infty$. The addition of a new unit to the hidden layer only occurs when the growth rate reaches an integer value, as adding a proportion of a single unit is not plausible. The extinction constant (α) acts as a lower bound on the size of the hidden layer population. The population will go locally extinct and prune units if the population falls below this threshold value. As this study aims to examine growth, α is set at zero to remove the possibility of extinction.

$$\frac{\mathrm{d}h_s^\ell}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C}+1\right) * E * \left(1-\frac{h_s^\ell}{C}\right)(h_s^\ell - \alpha) \tag{3.1}$$

where C is the carrying capacity of the hidden layer environment (upper bound), ℓ is the hidden layer number, h_s is the growth of the hidden unit population, E is the current global error of the network, and α is the extinction constant (lower bound) set to 0.

Parallel Growth.

The dynamics of parallel growing algorithm are characterized by having all hidden layers in the network growing simultaneously. In this context, the network can grow the current layer (ℓ)

and all subsequent layers ($\ell \ldots m$) regardless if the carrying capacity (C) has been reached by any hidden layer. The growth rate of the first hidden layer (h_s^0) is calculated according to (3.1), while the growth rate of every subsequent layer (h_s^ℓ) is calculated according to (3.2). The growth rate of the first layer is independent of the size of any other layer, in contrast subsequent hidden layers are impacted by the size of the hidden layer that proceeds it ($h_s^{\ell-1}$). The effect of previous hidden layer sizes being considered allows for a more staggered growth rate for deeper layers. As the previous hidden layer increases in size and begins to approach the carrying capacity, where once it reached it will no longer be able to add any new hidden units, the current layers growth rate is dynamically increased to compensate for the continued need to add more hidden units to match the tasks complexity (see Figure 3.2). A visualization of the parallel growing method's effect on growth rate and changes in hidden layer sizes for a three-hidden layer MLP is depicted in Figure 3.3.

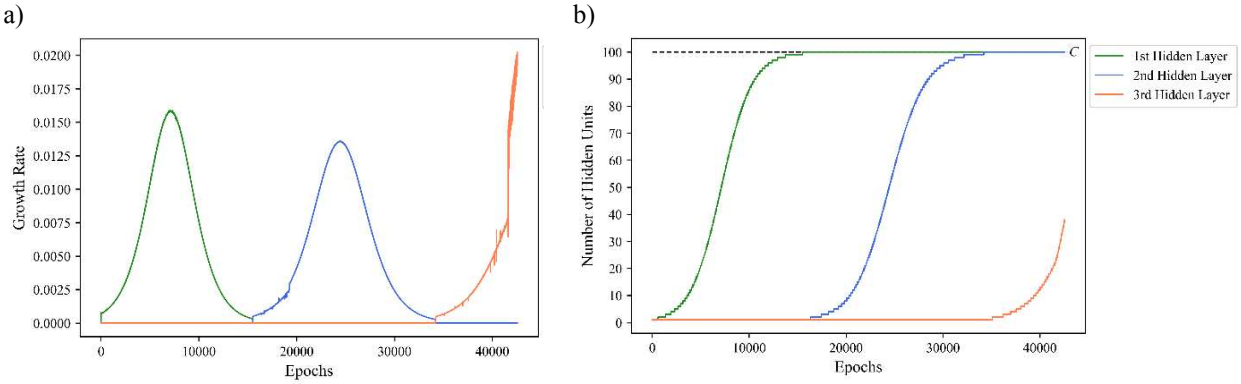

Figure 3.1. A sample of sequential growing dynamics for a three-hidden layer MLP. a) Sequential growth rate of three-hidden layers across training epochs. b) Changes in hidden layer size across three-hidden layers with a carrying capacity (C) of 100 during training.

$$\frac{\mathrm{d}h_s^\ell}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C - h_s^{\ell-1}} + 1\right) * E * \left(1 - \frac{h_s^\ell}{C}\right)(h_s^\ell - \alpha) \tag{3.2}$$

where C is the carrying capacity of the hidden layer environment (upper bound), h_s^{ℓ} is the growth of the hidden unit population of the current layer, $h_s^{\ell-1}$ is the growth of the hidden unit population of the previous layer, E is the current global error of the network, and α is the extinction constant (lower bound) set to 0.

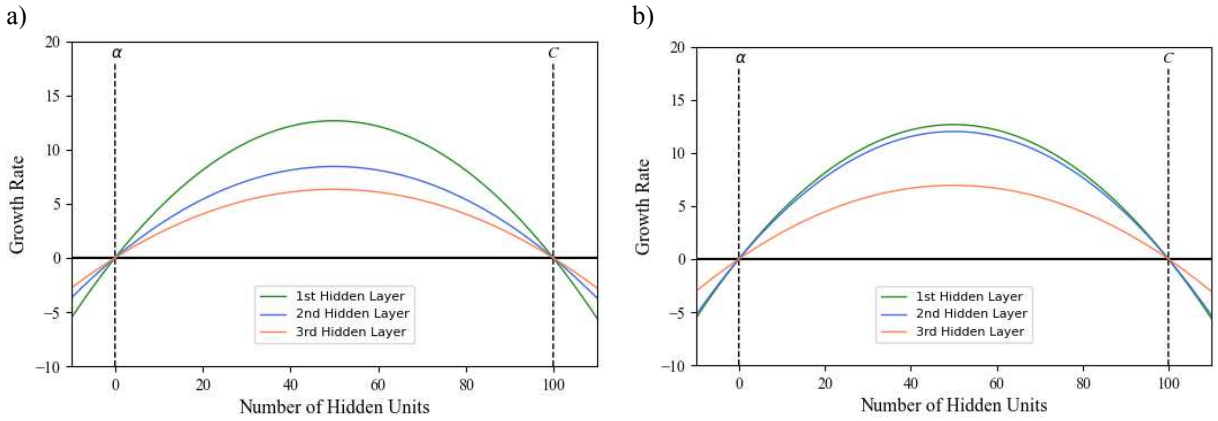

Figure 3.2. The effect of hidden layer size on the growth rate of subsequent layers. Where C is the carrying capacity and α is the extinction constant. a) Initial staggered growth rate of hidden layers, with deeper layers growing at a slower rate. b) Increase in the second layers growth rate as a response to the first hidden layer almost reaching the carrying capacity.

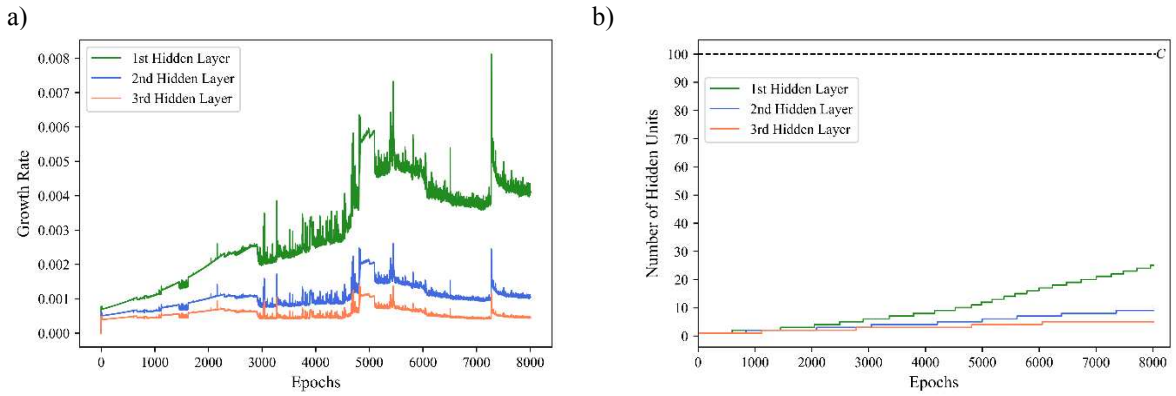

Figure 3.3. A sample of parallel growing dynamics for a three-hidden layer MLP. a) Parallel growth rate of three-hidden layers across training epochs. b) Changes in hidden layer size across three-hidden layers with a carrying capacity (C) of 100 during training.

DNC Growth.

Iteratively adding new units with DNC occurs when the error curve flattens within a specific time frame (i.e., window width; Ash, 1989). This requires an *a priori* finely-tuned trigger slope (Δ_T), which measures the error curve's flatness, whereby a new unit is added if the error falls below this value. Additionally, the user must finely tune the window width (z), the number of epochs over which the flatness of the error curve is compared to the trigger slope. If the error curve flattens out below the trigger slope (Δ_T), and if enough epochs have passed since the last addition, a new unit is added. This is determined by (3.3) and (3.4). The window width (z) was set to 1000 epochs and the trigger slope (Δ_T) was set to 0.01 to promote the construction of smaller architectures. To allow a comparison in the context of multiple hidden layers, here, DNC follows a sequential growth methodology. The network is not able to grow the subsequent layer ($\ell + 1$) until the current layer (ℓ) has reached the maximum number of hidden units that it can sustain as dictated by the carrying capacity (C).

$$\frac{\rho_{[t]} - \rho_{[t-z]}}{\rho_{[t_0]}} < \Delta_T \tag{3.3}$$

where $\rho_{[t]}$ is the average error at time t across all output units, z is the window width in epochs over which the slope is determined, Δ_T is the trigger slope.

$$t - z \geq t_0 \tag{3.4}$$

where t_0 is the time in epochs when the last unit was added to the hidden layer.

Architecture: A formal description

A multilayer perceptron (MLP) with m hidden layers is described by the number of units and the number of weight connections (see Figure 3.4). Let the size of layer ℓ at training epoch t

be denoted by $S_{[t]}^{\ell}$, with the size of the all the layers in the network at training epoch t be defined by vector $\psi_{[t]}$. The architecture of the network is thus represented as $\psi_{[t]} = \left(S_{[t]}^{0},\ S_{[t]}^{1},\ S_{[t]}^{2},\ \cdots S_{[t]}^{m},\ \cdots S_{[t]}^{m+1}\right)$. With m being the total number of hidden layers, $S_{[t]}^{0}$ being the size of the input layer (d), and $S_{[t]}^{m+1}$ the size of the output layer (k). The weights for each layer are defined by tensor Φ. With $\Phi = \left(W^{1},\ W^{2},\ \cdots W^{m} \cdots, W^{(m+1)}\right)$. Wherein each component W^{ℓ} of this tensor is a weight connection matrix:

$$W^{\ell} = \begin{pmatrix} w_{11}^{\ell} & \cdots & w_{1j}^{\ell} & \cdots & w_{1S_{(\ell-1)}}^{\ell} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{i1}^{\ell} & \cdots & w_{ij}^{\ell} & \cdots & w_{iS_{(\ell-1)}}^{\ell} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{S_{\ell}1}^{\ell} & \cdots & w_{S_{\ell}j}^{\ell} & \cdots & w_{S_{\ell}S_{(\ell-1)}}^{\ell} \end{pmatrix}$$

where w_{ij}^{ℓ} is the weight connection between the i^{th} unit of the layer ℓ and the j^{th} unit of layer $\ell - 1$.

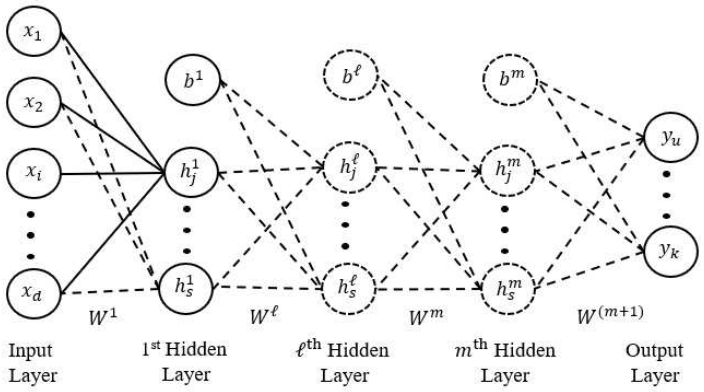

Figure 3.4. MLP architecture. In the above architecture *x* is vector of length d, where d, is the dimension the input vector, *s* is the current number of hidden units in that layer, *b* the biases, and the dashed lines and dashed circles are weight connections and hidden units that are incrementally added during growing.

The size of the input layer (d) and output layer (k) are fixed and dependent on the given task. In contrast, the width and depth of the hidden layers $(\psi_{[t]})$ are permitted to grow. For all

variants the maximum number of hidden layers is defined by Max_{ℓ} and the maximum number of hidden units per hidden layer is dictated by the carrying capacity (C). To simplify comparisons, Max_{ℓ} was fixed to three hidden layers, and C was fixed at 100 hidden units. The initialization condition gives rise to different network variants.

- In the first condition, the network is only initialized with one hidden layer with a single hidden unit (First-Init variant), $\psi_{[0]} = \left(S_{[0]}^{0},\ S_{[0]}^{1},\ S_{[0]}^{2}\right) = (d,\ 1,\ k)$, for a visualization see Figure 3.5a.
- Alternatively, the second condition initializes all three hidden layers with a single hidden unit (All-Init variant), $\psi_{[0]} = \left(S_{[0]}^{0},\ S_{[0]}^{1},\ S_{[0]}^{2},\ S_{[0]}^{3},\ S_{[0]}^{4}\right) = (d,\ 1,\ 1,\ 1,\ k)$, for a visualization see Figure 3.5b.

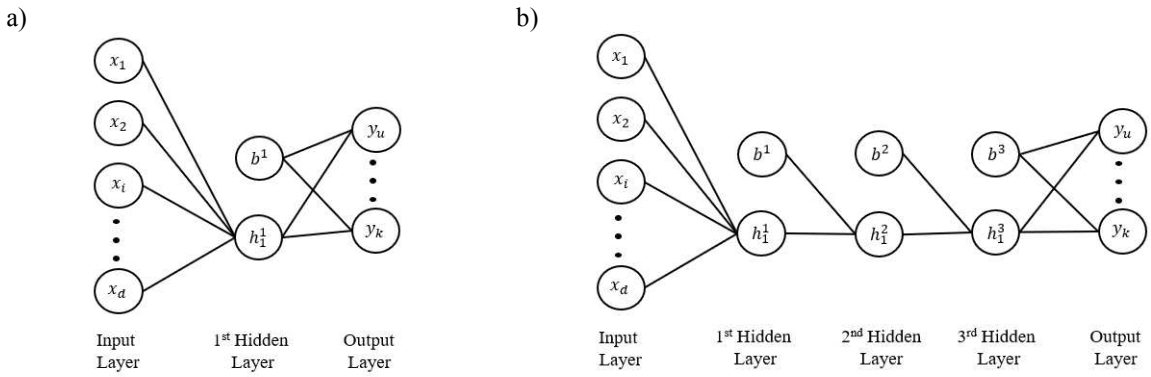

Figure 3.5. Network initialization conditions. a) Only one hidden layer initialized with a single hidden unit. b) All three hidden layers initialized with a single hidden unit.

When adding a new unit with any of the growing conditions (sequential, parallel, or DNC), the new unit is added to the end of the hidden layer with all associated incoming and outgoing weight connections in a fully connected fashion. These weight connections are randomly initialized. During learning, the weights are updated via batch stochastic gradient descent with momentum by two possible conditions:

- All weights in the network are updated at each training epoch t (UAW variant), for a visualization see Figure 3.6a.
- Only the incoming weights to the layer that is actively growing are updated and all other weights in the network are frozen (F variant), for a visualization see Figure 3.6b.

a)

b)

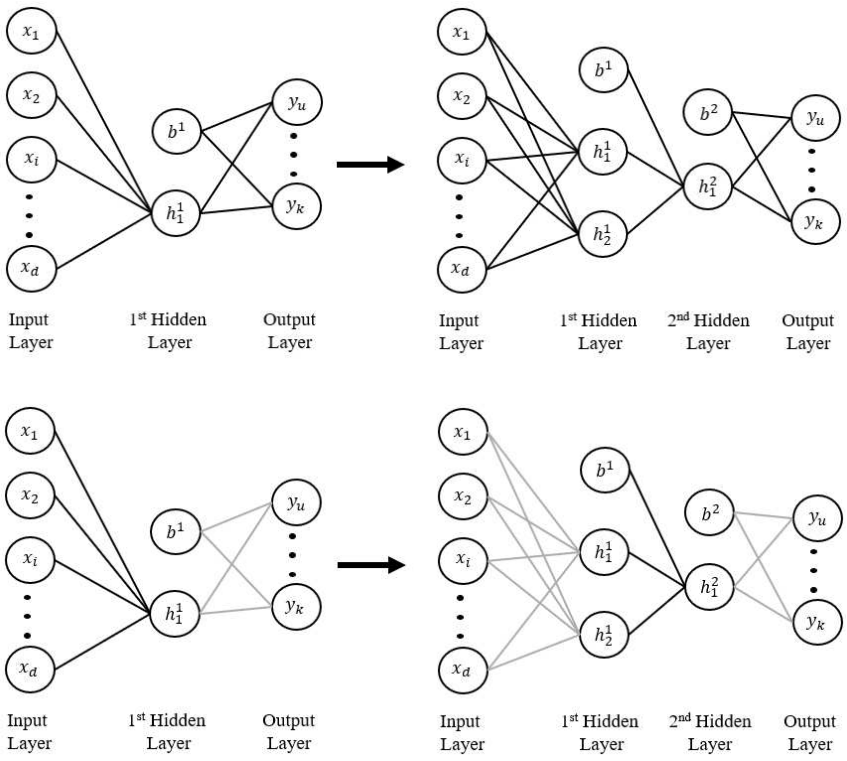

Figure 3.6. Weight update conditions. a) All network weights can be updated as the network grows (solid black lines). b) Only the incoming weights to the layer that is actively growing can be updated (solid black lines), all other weights in the network are frozen (light grey lines). In this example, initially the first hidden layer is active, then second hidden layer grows a new unit and becomes the active layer.

Learning

The following section outlines the learning process and the algorithms for each of the growing approaches. To measure network performance, the categorical cross entropy loss function

(3.5) is calculated at the end of each epoch as a measure of error with the addition of a weight penalty from L_2 regularization (ridge regression).

$$E_{[t]} = \left(-\frac{1}{M} \sum_{k=1}^{N} \sum_{q=1}^{M} y_k^q \log \hat{y}_k^q \right) + \left(\frac{\lambda}{2M} \sum_{i=1}^{d} {W_{[t]}^i}^2 \right) \tag{3.5}$$

where $\hat{y}_k^q$ is the k^{th} scalar value output from the network for example q, y_k^q is the corresponding target value, and N is the total number of classes, and M is the total number of instances in the data, d is the number of input features, and λ the regularization parameter tuned to 0.001.

At each training epoch t, R batch iterations of batch stochastic gradient descent with momentum are computed to update the weights W^ℓ. At each batch iteration $iter$ ($iter = 1\ to\ R$), momentum, the exponential moving average, is calculated according to (3.6).

$$m_{[t]} = \beta m_{[t-1]} + (1 - \beta) g_{[t]} \tag{3.6}$$

where $m_{[t]}$ is the velocity vector, β is the momentum term (friction coefficient) set at 0.9, and $g_{[t]}$ is the gradient at time t.

The weights are then updated according to (3.7).

$$W_{[t]}^\ell = W_{[t-1]}^\ell - \eta\left(m_{[t]}\right) \tag{3.7}$$

where η is the learning rate set to 0.01.

The sigmoid activation function (3.8) is used, with the softmax function (3.9) being used to scale the final output of the network and generate interrelated probabilities for multiclass classification between 0 and 1.

$$h_j^\ell = \frac{1}{1 + e^{-(h_j^{\ell\, in})}} \tag{3.8}$$

where h_j^{ℓ} is the hidden layer output from unit j for layer ℓ obtained in the usual way (for details, see Ross et al., 2020) and $h_j^{\ell\ in}$ is its corresponding activation.

$$\hat{y} = \frac{e^{y_u^{in}}}{\sum_{k=1}^{N} e^{y_k^{in}}} \tag{3.9}$$

where $\hat{y}$ is the categorical probabilistic output vector that sums to 1 and y_u^{in} is the input vector. The denominator is a normalization term across all classes (N).

With all approaches, training continues until one of the following conditions are met:

1) The maximum number of training epochs is reached, $Max_t = 100{,}000$.
2) The training categorical cross entropy falls below the convergence threshold, $Min_E = 0.01$.
3) The difference in testing accuracy over a 1000 epoch window is less than the testing accuracy threshold, $\theta = 0.01$.

The algorithms are described as follows:

1) Parallel Growth.

Part 1a Initialization: All Init-Variant
//Initialize all hidden layers.
$t = 0$, Starting epoch.
$Max_{\ell} = 3$, Maximum number of hidden layers.
$Init_{\ell} = 3$, Initial number of hidden layers.
$C = 100$, Maximum number of hidden units that a layer can sustain.
if F-Variant*//Only the incoming weights to active layer are updated and others frozen:*
$Activ_{\ell} = 1$, Initial active layer.
//Initialization of the network topology:
$\psi_{[t]} = \left(S_{[0]}^0 + S_{[0]}^1 + S_{[0]}^2 + S_{[0]}^3 + S_{[0]}^4\right) = (d, 1,\ 1,\ 1,\ y)$
//Random initialization of the weights from a uniform distribution ranging from -0.1 to 0.1:
$\Phi = (W^1 + W^2 + W^3 + W^4)$
// Initialization of the growth rate for the hidden layers:
$\varrho_{[t]} = \left(h_{s[0]}^1 + h_{s[0]}^2 + h_{s[0]}^3\right) = (0,\ 0,\ 0)$

Part 1b Initialization: First Init-Variant

//Initialize only the first hidden layer.

$t = 0$, Starting epoch.

$Max_\ell = 3$, Maximum number of hidden layers.

$Init_\ell = 1$, Initial number of hidden layers.

$C = 100$, Maximum number of hidden units that a layer can sustain.

if F-Variant*//Only the incoming weights to active layer are updated and others frozen:*

$Activ_\ell = 1$ Initial active layer.

//Initialization of the network topology:

$\psi_{[t]} = \left(S^0_{[0]} + S^1_{[0]} + S^2_{[0]}\right) = (d,\ 1,\ y)$

//Random initialization of the weights from a uniform distribution ranging from -0.1 to 0.1:

$\Phi = (W^1 + W^2)$

// Initialization of the growth rate for the hidden layers:

$\varrho_{[t]} = \left(h^1_{s[0]} + h^2_{s[0]} + h^3_{s[0]}\right) = (0,\ 0,\ 0)$

Part 2a Learning and Parallel Growing: UAW-Variant

//Update all weights.

$done = false$

$While\ t < Max_t\ and\ done = false$

$for\ iter = 1\ to\ R$

calculate $m_{[t]} = \beta m_{[t-1]} + (1-\beta) g_{[t]}$

calculate $W^\ell_{[t]} = W^\ell_{[t-1]} - \eta\left(m_{[t]}\right)$

end iter

calculate loss

$$E_{[t]} = \left(-\frac{1}{M}\sum_{k=1}^{N}\sum_{q=1}^{M} y_q^k \log \hat{y}_q^k\right) + \left(\frac{\lambda}{2M}\sum_{i=1}^{n} {W^i_{[t]}}^2\right)$$

calculate accuracy $TrainAcc_{[t]}$ and $TestAcc_{[t]}$

calculate growth rate

$for\ \ell = 1$ to m

$if\ \ell = 1$

$$\frac{\mathrm{d}h^\ell_s}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C} + 1\right) * E * \left(1 - \frac{h^\ell_s}{C}\right)(h^\ell_s - \alpha)$$

$else$

$$\frac{\mathrm{d}h^\ell_s}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C - h^{\ell-1}_s} + 1\right) * E * \left(1 - \frac{h^\ell_s}{C}\right)(h^\ell_s - \alpha)$$

//Update the growth rate

$$h^\ell_{s[t]} = h^\ell_{s[t-1]} + \frac{\mathrm{d}h^\ell_s}{\mathrm{d}t}$$

//Determine if a new unit is grown and has not reached the carrying capacity.

$if\ h^\ell_{s[t]} - h^\ell_{s[t-1]} = 1\ and\ h^\ell_{s[t]} < C$

// A new hidden unit has been grown.

$S^\ell_{[t]} = S^\ell_{[t-1]} + 1$

//Random initialization of the new unit's weights.

if $E_{[t]} < Min_E$ *or* $t = Max_t$ *or* $\left(TestAcc_{[t]} - TestAcc_{[t-1000]}\right) < \theta$

$done = true$

else

$t = t + 1$

Part 2b Learning and Parallel Growing: F-Variant

//Only the incoming weights to active layer are updated and others frozen.

$done = false$

$While\ t < Max_t\ and\ done = false$

$for\ iter = 1\ to\ R$

calculate $m_{[t]} = \beta m_{[t-1]} + (1-\beta) g_{[t]}$

//Only the incoming weights to the active layer are updated.

calculate $W_{[t]}^{Activ_\ell} = W_{[t-1]}^{Activ_\ell} - \eta\left(m_{[t]}\right)$

end iter

calculate loss

$$E_{[t]} = \left(-\frac{1}{M}\sum_{k=1}^{N}\sum_{q=1}^{M} y_q^k \log \hat{y}_q^k\right) + \left(\frac{\lambda}{2M}\sum_{i=1}^{n} {W_{[t]}^i}^2\right)$$

calculate accuracy $TrainAcc_{[t]}$ and $TestAcc_{[t]}$

calculate growth rate

$for\ \ell = 1$ to m

$if\ \ell = 1$

$$\frac{\mathrm{d}h_s^\ell}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C} + 1\right) * E * \left(1 - \frac{h_s^\ell}{C}\right)(h_s^\ell - \alpha)$$

$else$

$$\frac{\mathrm{d}h_s^\ell}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C - h_s^{\ell-1}} + 1\right) * E * \left(1 - \frac{h_s^\ell}{C}\right)(h_s^\ell - \alpha)$$

//Update the growth rate

$$h_{s[t]}^\ell = h_{s[t-1]}^\ell + \frac{\mathrm{d}h_s^\ell}{\mathrm{d}t}$$

//Determine if a new unit is grown and has not reached the carrying capacity.

if $h_{s[t]}^\ell - h_{s[t-1]}^\ell = 1\ \ and\ h_{s[t]}^\ell < C$

// A new hidden unit has been grown.

$S_{[t]}^\ell = S_{[t-1]}^\ell + 1$

//Random initialization of the new unit's weights.

// Has the new unit been grown in a different layer than the active layer?

if $\ell \neq Activ_\ell$

$Activ_\ell = \ell$ *//Update which is the active layer.*

if $E_{[t]} < Min_E$ *or* $t = Max_t$ *or* $\left(TestAcc_{[t]} - TestAcc_{[t-1000]}\right) < \theta$

$done = true$

else

$t = t + 1$

2) Sequential Growth and DNC Growth.

Part 1a Initialization: All Init-Variant
//Initialize all hidden layers.
$t = 0$, Starting epoch.
$Max_{\ell} = 3$, Maximum number of hidden layers.
$Init_{\ell} = 3$, Initial number of hidden layers.
$Current_{\ell} = 1$, Starting layer for sequential growth.
$C = 100$, Maximum number of hidden units that a layer can sustain.
//Initialization of the network topology:
$\psi_{[t]} = \left(S^0_{[0]} + S^1_{[0]} + S^2_{[0]} + S^3_{[0]} + S^4_{[0]}\right) = (d, 1,\ 1,\ 1,\ y)$
//Random initialization of the weights from a uniform distribution ranging from -0.1 to 0.1:
$\Phi = (W^1 + W^2 + W^3 + W^4)$
if Sequential*:*
// Initialization of the growth rate for the hidden layers:
$\varrho_{[t]} = \left(h^1_{s[0]} + h^2_{s[0]} + h^3_{s[0]}\right) = (0,\ 0,\ 0)$

Part 1b Initialization: First Init-Variant
//Initialize only the first hidden layer.
$t = 0$, Starting epoch.
$Max_{\ell} = 3$, Maximum number of hidden layers.
$Init_{\ell} = 1$, Initial number of hidden layers.
$Current_{\ell} = 1$, Starting layer for sequential growth.
$C = 100$, Maximum number of hidden units that a layer can sustain.
//Initialization of the network topology:
$\psi_{[t]} = \left(S^0_{[0]} + S^1_{[0]} + S^2_{[0]}\right) = (d,\ 1,\ y)$
//Random initialization of the weights from a uniform distribution ranging from -0.1 to 0.1:
$\Phi = (W^1 + W^2)$
if Sequential growth*:*
// Initialization of the growth rate for the hidden layers:
$\varrho_{[t]} = \left(h^1_{s[0]} + h^2_{s[0]} + h^3_{s[0]}\right) = (0,\ 0,\ 0)$

Sequential Growth.

Part 2a Learning and Sequential Growing: UAW-Variant
//Update all weights.
$done = false$
$While\ t < Max_t\ and\ done = false$
$for\ iter = 1\ to\ R$
calculate $m_{[t]} = \beta m_{[t-1]} + (1-\beta) g_{[t]}$
calculate $W^{\ell}_{[t]} = W^{\ell}_{[t-1]} - \eta\left(m_{[t]}\right)$
end iter
calculate loss

$$E_{[t]} = \left(-\frac{1}{M} \sum_{k=1}^{N} \sum_{q=1}^{M} y_q^k \log \hat{y}_q^k \right) + \left(\frac{\lambda}{2M} \sum_{i=1}^{n} {W_{[t]}^i}^2 \right)$$

calculate accuracy $TrainAcc_{[t]}$ and $TestAcc_{[t]}$

calculate growth rate

$for\ \ell = Current_\ell$

$\frac{\mathrm{d}h_s^\ell}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C} + 1\right) * E * \left(1 - \frac{h_s^\ell}{C}\right)(h_s^\ell - \alpha)$

//Update the growth rate

$h_{s[t]}^\ell = h_{s[t-1]}^\ell + \frac{\mathrm{d}h_s^\ell}{\mathrm{d}t}$

//Determine if a new unit is grown.

if $h_{s[t]}^\ell - h_{s[t-1]}^\ell = 1$

// Determine if the layer has reached the carrying capacity.

if $h_{s[t]}^\ell < C$

// A new hidden unit has been grown.

$S_{[t]}^\ell = S_{[t-1]}^\ell + 1$

//Random initialization of the new unit's weights.

elif $h_{s[t]}^\ell = C$

$Current_\ell = Current_\ell + 1$ *//Switch to the next layer.*

$S_{[t]}^{\ell+1} = 1$ *//Add the new unit to the next layer.*

if $E_{[t]} < Min_E$ *or* $t = Max_t$ *or* $\left(TestAcc_{[t]} - TestAcc_{[t-1000]}\right) < \theta$

$done = true$

else

$t = t + 1$

Part 2b Learning and Sequential Growing: F-Variant

//Only the incoming weights to active layer are updated and others frozen.

$done = false$

$While\ \ t < Max_t\ and\ done = false$

$for\ iter = 1\ to\ R$

calculate $m_{[t]} = \beta m_{[t-1]} + (1-\beta) g_{[t]}$

//Only the incoming weights to active layer are updated.

calculate $W_{[t]}^{Current_\ell} = W_{[t-1]}^{Current_\ell} - \eta\left(m_{[t]}\right)$

end iter

calculate loss

$$E_{[t]} = \left(-\frac{1}{M} \sum_{k=1}^{N} \sum_{q=1}^{M} y_q^k \log \hat{y}_q^k \right) + \left(\frac{\lambda}{2M} \sum_{i=1}^{n} {W_{[t]}^i}^2 \right)$$

calculate accuracy $TrainAcc_{[t]}$ and $TestAcc_{[t]}$

calculate growth rate

$for\ \ell = Current_\ell$

$\frac{\mathrm{d}h_s^\ell}{\mathrm{d}t} = \frac{1}{E+\ell} * \left(\frac{1}{C} + 1\right) * E * \left(1 - \frac{h_s^\ell}{C}\right)(h_s^\ell - \alpha)$

//Update the growth rate

$h_{s[t]}^\ell = h_{s[t-1]}^\ell + \frac{\mathrm{d}h_s^\ell}{\mathrm{d}t}$

//Determine if a new unit is grown.

if $h_{s[t]}^\ell - h_{s[t-1]}^\ell = 1$

// Determine if the layer has reached the carrying capacity.

if $h_{s[t]}^\ell < C$

// A new hidden unit has been grown.

$S_{[t]}^\ell = S_{[t-1]}^\ell + 1$

//Random initialization of the new unit's weights.

elif $h_{s[t]}^\ell = C$

$Current_\ell = Current_\ell + 1$ *//Switch to the next layer (active layer).*

$S_{[t]}^{\ell+1} = 1$ *//Add the new unit to the next layer.*

if $E_{[t]} < Min_E$ *or* $t = Max_t$ *or* $\left(TestAcc_{[t]} - TestAcc_{[t-1000]}\right) < \theta$

$done = true$

else

$t = t + 1$

DNC Growth.

Part 2a Learning and DNC Growing: UAW-Variant

//Update all weights.

$done = false$

$t_0 = 0$ *//Time in epochs when the last unit was added.*

$While\ t < Max_t\ and\ done = false$

$for\ iter = 1\ to\ R$

calculate $m_{[t]} = \beta m_{[t-1]} + (1-\beta) g_{[t]}$

calculate $W_{[t]}^\ell = W_{[t-1]}^\ell - \eta\left(m_{[t]}\right)$

end iter

calculate loss

$$E_{[t]} = \left(-\frac{1}{M}\sum_{k=1}^{N}\sum_{q=1}^{M} y_q^k \log \hat{y}_q^k\right) + \left(\frac{\lambda}{2M}\sum_{i=1}^{n} {W_{[t]}^i}^2\right)$$

calculate accuracy $TrainAcc_{[t]}$ and $TestAcc_{[t]}$

//Determine if enough time has passed and the error has stopped changing.

if $(t-z) \geq t_0\ and\ \frac{\rho_{[t]} - \rho_{[t-z]}}{\rho_{[t_0]}} < \Delta_T$

// Determine if the layer has reached the carrying capacity.

if $S_{[t]}^\ell < C$

// A new hidden unit is added.

$S_{[t]}^\ell = S_{[t-1]}^\ell + 1$

//Random initialization of the new unit's weights.

elif $S_{[t]}^{\ell} = C$

$Current_{\ell}$=$Current_{\ell} + 1$ *//Switch to the next layer.*

$S_{[t]}^{\ell+1} = 1$ *//Add the new unit to the next layer.*

$t_0 = t$

if $E_{[t]} < Min_E$ *or* $t = Max_t$ *or* $\left(TestAcc_{[t]} - TestAcc_{[t-1000]}\right) < \theta$

$done = true$

else

$t = t + 1$

Part 2b Learning and DNC Growing: F-Variant

//Only the incoming weights to active layer are updated and others frozen.

$t_0 = 0$ *//Time in epochs when the last unit was added.*

$While\ t < Max_t\ and\ done = false$

$for\ iter = 1\ to\ R$

calculate $m_{[t]} = \beta m_{[t-1]} + (1-\beta) g_{[t]}$

//Only the incoming weights to active layer are updated.

calculate $W_{[t]}^{Current_{\ell}} = W_{[t-1]}^{Current_{\ell}} - \eta\left(m_{[t]}\right)$

end iter

calculate loss

$$E_{[t]} = \left(-\frac{1}{M}\sum_{k=1}^{N}\sum_{q=1}^{M} y_q^k \log \hat{y}_q^k\right) + \left(\frac{\lambda}{2M}\sum_{i=1}^{n} {W_{[t]}^i}^2\right)$$

calculate accuracy $TrainAcc_{[t]}$ and $TestAcc_{[t]}$

//Determine if enough time has passed and the error has stopped changing.

if $(t - z) \geq t_0\ and\ \frac{\rho_{[t]} - \rho_{[t-z]}}{\rho_{[t_0]}} < \Delta_T$

// Determine if the layer has reached the carrying capacity.

if $S_{[t]}^{\ell} < C$

// A new hidden unit is added.

$S_{[t]}^{\ell} = S_{[t-1]}^{\ell} + 1$

//Random initialization of the new unit's weights.

elif $S_{[t]}^{\ell} = C$

$Current_{\ell}$=$Current_{\ell} + 1$ *//Switch to the next layer (active layer).*

$S_{[t]}^{\ell+1} = 1$ *//Add the new unit to the next layer.*

$t_0 = t$

if $E_{[t]} < Min_E$ *or* $t = Max_t$ *or* $\left(TestAcc_{[t]} - TestAcc_{[t-1000]}\right) < \theta$

$done = true$

else

$t = t + 1$

Experiments and Results

The proposed parallel growing algorithm was compared to the sequential version of our growing algorithm as well as DNC in the MLP on three benchmark classification data sets: the Breast Cancer Wisconsin (Diagnostic; BCW) data set (Wolberg et al., 1995), the Wine data set (Aeberhard & Forina, 1991), and the Fashion MNIST data set (Xiao et al., 2017). These data sets vary in the number of features, classes, and instances (see Table 3.1). For a more detailed comparison, four variants for each approach (Parallel, Sequential, and DNC) were considered. These variants were based on the number of layers pre-initialized; one (First Init) or all (All Init), and the type of weight update; updating all weights (UAW) or just the incoming connections to the active layer and freezing the others (F).

Table 3.1

Data Sets

Data set	Number of features	Number of classes	Number of instances
BCW	30	2	569
Wine	13	3	178
Fashion MNIST	784	10	70,000

The results of the binary classification of the BCW data set are shown in Table 3.2. With the All Init-UAW variant, all approaches had similar training and testing performances. Slight variations in network size were observed, with the DNC approach growing fewer units than the parallel. With the First Init-UAW variant, all approaches were almost identical in size, performance, and training epochs. Since only one hidden layer was initialized and the task was easier for the network, fewer hidden units were required; the parallel approach behaved like the sequential one with no additional hidden layers grown. All approaches had similar training and testing performances for the First Init-F variant, with DNC having smaller variations. However,

the parallel approach grew three layers and about 21 hidden units. Conversely, the sequential and DNC approaches required only one hidden layer and about 5 units. Here the sequential approach also took considerably fewer training epochs with a smaller standard deviation. Finally, for the All Init-F variant, the parallel approach notably outperformed both the sequential and DNC approaches—reaching near-perfect training and testing performances, compared to about 60% and 50% for the sequential and DNC approaches. Additionally, both the sequential and DNC approaches reached the maximum number of training epochs, while the parallel approach required much less. The parallel approach, on average, grew smaller (three layers and about 140.6 units) than the sequential (three layers and about 292.9 units) but grew larger than DNC (three layers and about 102.0 units). However, since the window width for DNC is set at 1000 epochs and the maximum number of hidden units is set at 100 per layer; DNC could not grow units past the First layer.

The three-class Wine data set showed similar results to the BCW (see Table 3.3). For the All Init-UAW variant, all the approaches had similar average accuracies. However, the sequential approach had a large standard deviation of about 6%. Regarding the average size, DNC was the smallest (24 units), then parallel (74.9 units), followed by sequential (101.7 units). Despite not being the smallest, the parallel approach took noticeably fewer training epochs with less variation. All approaches were almost identical in size for the First Init-UAW variant, with around one hidden unit. Like the BCW, since only one hidden layer is initialized and the task is easier for the network requiring fewer hidden units, the parallel approach behaves like the sequential one with no additional hidden layers grown. All approaches were also similar in performance, with DNC having a larger std. dev. of about 4 %. Of note is the difference in epochs taken. The parallel approach, on average, needed 110 training epochs, while the other approaches needed over 2,000.

For the First Init-F variant, the parallel approach grew larger (three hidden layers with an average of 166.8 hidden units) compared to the other approaches that only used one hidden layer (sequential: 23.2 hidden units, and DNC: 5.8 hidden units). While the parallel approach had, on average, 2% higher accuracies, it took more than double the training epochs. The parallel and sequential approaches had similar performances for the final variant, the All Init-F variant. Nevertheless, the sequential method grew to the maximum of 100 units per layer (300 units total) and reached, on average, over 90,000 training epochs. Comparatively, the parallel approach utilized only 248.3 units with an average of 18,7171 epochs. DNC encountered the same limitation and could not grow past the first layer. This resulted in poor performances and reaching the maximum number of training epochs.

The most challenging data set was the ten-class Fashion MNIST (see Table 3.4). The sequential approach had a better average performance accuracy for the All Init-UAW variant than the parallel, of about 5% for training and about 2.5% for testing. Regardless of the better performance, the parallel approach grew on average much smaller (22.6 hidden units) compared to the other approaches (sequential: 218.1 hidden units; DNC 57 hidden units) and took only 5,893 epochs to train (about 36,000-61,000 epochs less). For the First Init-UAW variant, both the parallel and sequential approaches had similar sizes, performances, and training epochs on average. Conversely, DNC grew smaller (about three fewer hidden units), had a lower training accuracy (about 5% less), but took significantly more epochs. The parallel approach had the highest performance for the First Init-F variant. However, on average, the parallel approach also grew significantly larger (three hidden layers with 215 hidden units) compared to the other two approaches (sequential: 51.6 hidden units; DNC 24.8 hidden units). The parallel approach vastly outperformed the other approaches for the final variant, the All Init-F. The parallel approach

Table 3.2

Average Results of the Parallel, Sequential, and DNC Variants - Breast Cancer Wisconsin

Approach	Measures				
	Layers	Hidden Units	Training Acc. (%)	Testing Acc. (%)	Epochs taken
			All Init-UAW		
Parallel	3	**7.1±3.75** **3.2±1.17** **2.4±0.66**	99.80±0.00	94.79±0.00	6,105.1±1,221.88
Sequential	3	**12.5±6.89** **1.0±0.00** **1.0±0.00**	99.80±0.00	93.90±0.00	7,406.4±1,801.80
DNC	3	**6.7±2.56** **1.0±0.00** **1.0±0.00**	99.80±0.00	96.50±0.00	8,807.8±2,524.43
			First Init-UAW		
Parallel	1	1.0±0.00	99.30±0.00	98.20±0.00	2,001.0±0.00
Sequential	1	1.0±0.00	98.70±0.00	100.00±0.00	2,001.0±0.00
DNC	1	1.0±0.00	99.10±0.00	96.50±0.00	2,001.0±0.00
			First Init-F		
Parallel	**3**	17.0±27.49 3.0±9.0 0.6±1.8	92.33±4.32	91.31±4.36	6,605.6±5,262.63
Sequential	1	**5.7±1.55**	92.19±4.61	92.91±3.24	**4,003.0±1,266.18**
DNC	1	**4.60±2.37**	93.29±1.77	92.80±1.75	7,206.2±3,519.19
			All Init-F		
Parallel	3	86.0±4.02 38.0±5.87 16.6±2.33	**98.11±0.23**	**96.68±0.38**	**16,015.0±895.32**
Sequential	3	100.0±0.00 100.0±0.00 92.9±0.54	63.30±0.00	60.50±0.00	100,000.0±0.00
DNC	3	100.0±0.00 1.0±0.00 1.0±0.00	47.64±12.19	46.66±17.25	100,000.0±0.00

Note: Averages were calculated across 10 trials with the std. dev. reported.

achieved approximately 40% higher training and testing accuracies than the sequential approach. It also took fewer training epochs and grew much smaller (170 vs. 300 hidden units). It should be noted that the sequential approach reached the maximum hidden layer size of 100 as set by the carrying capacity. The argument could be made that if this value was larger, it might be possible for the sequential approach to reach higher training and testing accuracies. However, the sequential

approach did approach the 100,000-training epoch limit. If it were permitted to continue to grow, it would have been stopped by this limit. The DNC approach encountered the same limitation as with other variants, and could not grow past the first layer. This resulted in abysmal performances and reaching the maximum number of training epochs.

Table 3.3

Average Results of the Parallel, Sequential, and DNC Variants - Wine

Approach	Measures				
	Layers	Hidden Units	Training Acc. (%)	Testing Acc. (%)	Epochs taken
			All Init-UAW		
Parallel	3	47.5±8.35 17.9±3.48 9.5±1.57	100.00±0.00	95.24±1.89	8,890.6±640.40
Sequential	3	97.7±2.97 3.3±3.20 1.0±0.00	96.91±6.46	96.39±6.13	21,999.9±5,145.26
DNC	3	22.8±7.70 1.0±0.00 1.0±0.00	100.0±0.00	99.44±1.18	25,124.1±7,862.15
			First Init-UAW		
Parallel	1	1.8±0.40	99.09±0.99	97.20±2.95	110.83±350.46
Sequential	1	1.5±0.50	99.02±1.10	97.20±2.95	2,208.8±224.04
DNC	1	1.0±0.00	95.21±4.21	93.02±4.36	2,101.1±300.3
			First Init-F		
Parallel	**3**	91.0±26.0 61.5±24.88 14.3±6.51	94.66±5.01	94.17±5.14	18,117.1±4,575.13
Sequential	1	23.2±12.79	92.48±5.04	91.39±5.91	6,205.2±1,779.42
DNC	1	5.8±2.52	92.05±6.20	92.49±7.31	8,707.7±4,076.61
			All Init-F		
Parallel	3	**99.1±0.30** **91.3±2.97** **57.9±4.99**	84.93±7.79	86.39±5.14	**18,717.7±318.12**
Sequential	3	100.0±0.00 100.0±0.00 100.0±0.00	86.26±8.97	88.34±9.86	90,659.9±6,462.79
DNC	3	100.0±0.00 1.0±0.00 1.0±0.00	31.70±3.38	29.76±1.35	100,000.0±0.00

Note: Averages were calculated across 10 trials with the std. dev. reported.

Concerning the variants, initializing only one hidden layer (First Init) led to smaller topologies, faster training times, and similar performances across all tasks compared to initializing all the hidden layers (All Init). Similarly, updating all weights (UAW) resulted in smaller topologies, faster training times, and similar performances across all tasks compared to using a

Table 3.4

Average Results of the Parallel, Sequential, and DNC Variants – Fashion MNIST

Approach	Measures				
	Layers	Hidden Units	Training Acc. (%)	Testing Acc. (%)	Epochs
			All Init-UAW		
Parallel	3	**12.9±2.07** **5.8±1.09** **3.9±0.60**	**89.43±1.43**	83.30±0.81	**5,893.7±66.91**
Sequential	3	**100.0±0.00** **100.0±0.00** **18.1±3.92**	**94.53±0.47**	85.7±0.46	**42,374.7±1,523.05**
DNC	3	55.0±37.66 1.0±0.00 1.0±0.00	11.99±4.20	11.98±4.17	67,763.9±39,695.49
			First Init-UAW		
Parallel	1	7.2±0.60	88.40±0.86	82.97±0.59	4,903.9±300.30
Sequential	1	7.8±1.08	89.04±0.83	83.23±0.28	5,304.3±458.72
DNC	1	**4.3±0.71**	83.39±4.89	83.39±4.89	**19,121.1±4,914.30**
			First Init-F		
Parallel	3	99.5±0.50 82.1±9.68 33.4±7.72	**93.31±2.85**	**87.47±2.37**	14,613.6±917.43
Sequential	1	51.6±12.53	85.08±0.70	83.67±0.68	6,805.8±872.65
DNC	1	**24.8±5.25**	83.82±0.98	81.04±0.91	24,823.8±5,255.01
			All Init-F		
Parallel	3	**82.9±8.43** **54.9±9.17** **32.3±5.83**	**94.73±3.95**	**85.98±2.42**	**9,008.0±775.37**
Sequential	3	100.0±0.00 100.0±0.00 100.0±0.00	45.63±11.11	45.46±11.10	89,268.3±14,322.36
DNC	3	100.0±0.00 1.0±0.00 1.0±0.00	10.00±0.00	10.00±0.00	100,000.0±0.00

Note: Averages were calculated across 10 trials with the std. dev. reported.

freezing approach where only the active layer's weights were updated (F). Overall, the parallel approach rivals its sequential counterpart. With the two and three-class problems, the parallel approach obtained similar performances with the possibility of using fewer training epochs. However, the improved training time usually coincided with larger architectures. With the more challenging 10-class problem, the parallel rivalled or outperformed the sequential approaches. Only with the First Init-F variant did the parallel approach result in larger structures and longer training times. For both the All Init variants, the parallel approach grew smaller, had high accuracies, and took less training epochs. For the First Init-UAW, the parallel approach performed the same as the sequential approach, as no additional hidden layers were needed to perform the task.

Multiply and Accumulate (MACs) were computed for the network weights during forward propagation for a single trial of the All Init UAW variant for the Fashion MNIST to get an idea of the computational load. MACs are when an addition operation follows a multiplication operation and gives an idea of the computational complexity of a model (Nahmias et al., 2020). MACs for each layer during forward propagation while learning the Fashion MNIST are calculated according to (3.10)

$$\text{MACs} = B * R * S_{[t]}^{\ell} * \varsigma \tag{3.10}$$

Where B is the batch size (set to 256) and R the batch iterations (set to 235) for the Fashion MINIST, $S_{[t]}^{\ell}$ is the hidden layer size at training epoch t, and ς the number of training epochs corresponding to that hidden layer size.

It is important to note that here MACs need to consider a dynamic hidden layer size that varies across time for growing networks. As such, ς is used instead of just the current training epoch (t). This is because, for example, in a growing network, one hidden unit may be used for

the first 500 training epochs, but then two hidden units for the following 300 training epochs. The results for a single trial of each growing approach are shown in Table 3.5. Results show that the parallel method uses ten billion fewer MACs for the All Init-UAW variant compared to sequential and DNC approaches. This is reflected by fewer hidden units being used across all three layers and fewer training epochs needed. The sequential approach takes approximately three billion fewer MACs than the DNC approach. While the sequential approach did use noticeably more hidden units (239 compared to 89), it took nearly half the number of training epochs. These sample results showcase the computational complexity of these approaches and highlight the economical computational complexity of growing in parallel.

Table 3.5

Single Trial Results of the Parallel, Sequential, and DNC All Init-UAW Variant – *Fashion MNIST*

	Measures					
Approach	Layers	Hidden Units	Training Acc. (%)	Testing Acc. (%)	Epochs	MACs
Parallel	3	**12.0** **5.0** **4.0**	90.3	83.4	**6,005**	**3,558,464,000**
Sequential	3	100.0 100.0 39.0	95.9	86.0	45,044	**16,446,901,760**
DNC	3	87.0 1.0 1.0	19.9	19.9	100,000	**19,677,012,480**

Discussion

This study focused on the challenge of *when a new hidden unit or layer should be added* when using a constructive algorithm. We proposed investigating the effects of growing sequentially or in parallel in a multilayer context. Sequential growth was characterized by only growing one hidden layer at a time. In contrast, parallel growth was characterized by growing all hidden layers simultaneously. To achieve this, we created a modified version of our population

dynamics inspired growing algorithm capable of growing in parallel. Several variants of these approaches based on the hidden layer initialization and the training scheme employed were used to ensure a more comprehensive comparison. The sequential and parallel approaches were tested on several benchmark classification tasks. Another sequential growing approach, DNC, was also compared as it used the standard methodology of iteratively adding units based on the error curve flattening within a specific time frame.

The results highlight two scenarios where parallel growing is more beneficial than sequential. The first is realized by comparing to the DNC approach, which iteratively adds units based on the error curve flattening within a specific time frame. The All Init-F variant highlighted the limitation of a constructive approach heavily reliant on fine-tuning hyperparameters. While a smaller trigger slope and a larger window width can lead to smaller topologies, it can prevent the network from sequentially growing multiple layers in a reasonable time frame. In this case, with a maximum of 100,000 training epochs and a higher maximum hidden units per layer, it is impossible to grow depth with DNC if the task requires it. If the hyperparameters are not finely-tuned to the specific task, the resulting architecture can grow larger than the minimal number of units needed or prevent the network from growing beyond the minimal amount (Ash, 1989). Secondly, for the more challenging 10-class problem, when multiple hidden layers were initialized, the parallel growing approach surpassed the sequential approaches. It offered smaller layer widths, faster training times, and comparable or higher performances.

The possibility of the resulting architecture being larger than required for a given task can be considered a limitation of growing in parallel. Increasing network complexity beyond what is required can lead to overfitting (Curteanu & Cartwright, 2011). A possible solution to this problem is to create a hybrid growing-pruning approach by further developing the population dynamics

inspired approach to include pruning. Including network pruning offers a way to reduce the structural complexity of the network systematically. One way to achieve this would be to have a negative growth rate to decay the neuronal population and obtain an optimal minimal topology. Including a way to prune the network would allow for comparisons to more state-of-the-art approaches, as hybrid methods appear to be more suitable in searching for optimal architectures (Zemouri et al., 2020).

In our population dynamics inspired growing algorithm, the global network error is used as a measure of performance feedback that modulates the growth rate of the hidden population. As the error converges toward zero, the growth rate likewise converges toward zero. This property gives rise to the algorithms' ability to grow near-optimal architectures based on task complexity. One potential issue is the notion of the network error not converging to a minimal value due to noise in the data, resulting in a continuously growing architecture. An alternative might be to use the network bias and variance as a built-in metric for scaling, growing and pruning, such as by calculating network significance (Ashfahani & Pratama, 2019; Pratama et al., 2020). Essentially, this could act as a form of early-stopping for the growing process.

The focus of this study was not to obtain the most optimal performance but to investigate the comparison between sequential and parallel growing methods. As such, no preprocessing methods or modifications to the data set were applied. Performing preprocessing techniques or processes such as data augmentation can introduce confounding variables, making meaningful comparisons between models challenging (Blalock et al., 2020). However, preprocessing allows using more real-world datasets and can lead to faster learning and higher classification accuracies (Asadi & Kareem, 2014). An interesting extension for comparing sequential and parallel growing would be to implement the constructive algorithm in a CNN to learn larger coloured images, such

as the CIFAR-10. CNNs are comprised of convolutional and pooling layers that are responsible for feature extraction, followed by a fully-connected layer that is responsible for classifying the image (for visualization, see Appendix B Figure B.1). Replacing the pre-designed architecture of the fully-connected layer with a constructive algorithm gives rise to a more adaptive CNN. Mohamed et al. (2020) replaced the fully-connected layer in a CNN with their cascade-correlation growing deep learning neural network algorithm (CCG-DLNN) to successfully classify lung cancer images. With the success of a sequential growing approach in a CNN, the question arises could a parallel growing approach be beneficial to reduce time spent growing?

Conclusion

The decision of when to add a new hidden unit or layer is a fundamental challenge for constructive algorithms. It becomes even more complex in the context of multiple hidden layers. Dynamic growth inspired by population dynamics offers the potential to grow the width and depth of deeper neural networks in either a sequential or parallel fashion. Growing hidden layers in parallel promotes growing narrower deep architectures tailored to the task. The application and testing of parallel growing methods, in general, merit further investigation.

General Discussion

Discussion

Constructive algorithms offer an alternative to designing architectures in a "brick by brick" fashion, allowing the network to tailor its architecture to the task demands. This thesis presented a dynamic, more self-governed growing algorithm inspired by population dynamics for determining hidden layer sizes in a feedforward ANN, focusing on addressing the challenge of when to add a new hidden unit. The proposed algorithm opts for increased self-governance by reducing the number of finely-tuned hyperparameters required to decide when to grow, putting more control of the network's structure and resulting capabilities in the algorithm itself. More self-governed growth has several advantages over constructive algorithms that rely on hyperparameters. It avoids manually fine-tuning hyperparameters, which can be time-consuming and susceptible to human bias. Additionally, as showcased in comparisons with DNC, algorithms that rely on hyperparameters for deciding when to grow may require re-tuning for different tasks. More self-governed growth offers a general solution that is less task-dependent. Furthermore, compared to self-governed stochastic growth, such as proposed by Huang et al. (2006), population dynamics-inspired growth is interpretable. The non-linear growth of the network is based on the network's needs during learning rather than just randomness.

Generalization

The growing algorithm's versatility extends beyond the confines of a single architecture, learning rule, or error measure. For example, self-organizing maps could benefit from the algorithm's capacity to dynamically grow the network topology to contribute to the efficient emergence of topographical representations. These networks typically comprise of a high-dimensional input space and a lower-dimensional grid, endowing them with dimension-reduction

properties and clustering capabilities (Kohonen, 2013; Ross et al., 2019). Rather than backpropagation, Hebbian learning is employed with a neighbourhood function to give rise to distinct clusters based on the input features. Although the output grid size is conventionally fixed, integrating growing mechanisms could facilitate the initialization of a small grid that can be dynamically grown according to the network's task-related demands. In such a case, the error could be represented by the quantization error, reflecting the difference in Euclidean distance between the input and weights at each iteration. Growth would then be modulated as the overall input representations in the output grid improve (shorter Euclidean distances). The carrying capacity could be viewed as a maximum grid size or a mechanism to constrain the grid to force regrouping and more general clustering. A noteworthy application of a more self-governed growing self-organizing map would be the application of robotic task allocation (Ross et al., 2019), where adequately setting grid size *a priori* may be challenging.

The growing algorithm could also be applied with principal component analysis for dimensionality reduction. In principal component analysis, the primary objective is to distill a reduced set of features that effectively represents the original dataset in a lower-dimensional subspace while maximizing the explained variance of the original dataset (Kherif & Latypova, 2020). To achieve this, the original inputs are transformed into linear uncorrelated combinations, called principal components. Subsequently, the original data is projected orthogonally onto the lower-dimensional subspace, aligning with the directions defined by the principal components. In this scenario, the growing algorithm could play a pivotal role by incrementally increasing the number of principal components, thereby explaining the most variance while minimizing the dimensionality. Here, the error in the algorithm could be represented by the reciprocal of the total variance explained, with increased growth corresponding to the extraction of more principal

components, resulting in more variance explained and modulation of the growing dynamics. The carrying capacity could aptly be interpreted as the maximum allowable number of principal components and set to the original dataset's dimensionality. Implementing such a framework could be realized within a multilayer feedforward ANN utilizing the nonlinear Generalized Hebbian Algorithm initially proposed by Sanger (1989).

Pruning

An intuitive progression would be to refine the growing algorithm to incorporate neuronal pruning. By incorporating a negative growth rate, it becomes feasible to reduce the neuronal population, a concept analogous to population decay. A straightforward way to achieve this would be to modify the fixed-point attractor at the carrying capacity. By reducing this upper bound value, it would be possible to introduce decay in a post-growing phase. However, introducing pruning brings two distinct challenges: Which unit should be pruned and how many?

Regarding the former, the simplest method, both computationally and to implement, would be to prune the last unit grown. Nevertheless, this approach can potentially harm overall network performance, as the last added unit might have played a crucial role in solving the task. A possible remedy would be implementing a sensitivity-based method, such as calculating the Optimal Brain Damage method to quantify a unit's importance relative to the cost function (LeCun et al., 1990; Reed, 1993). Concerning the latter challenge, how much to reduce the carrying capacity value to induce population decay (i.e., how many units to prune)—an incremental approach proves most pragmatic. This method entails gradually decreasing the carrying capacity, allowing the network time to adapt its weights, and subsequently comparing the performance of the pruned network (new state) with its unpruned counterpart (previous state). If performance remains the same or improves, the iterative pruning process continues until a diminished performance is observed.

When a diminished performance is observed, the network will revert to the previous higher-performing state (Chaber & Ławryńczuk, 2018). The integration of pruning alongside the presented growing algorithm gives rise to a hybrid strategy, which has been considered more suitable in the search strategy for the best ANN architecture (Zemouri et al., 2020).

Growth at Different Levels of Topology

In the presented work, we viewed the hidden layer as an environment wherein a hidden unit population exists. However, this theoretical perspective can be broadened to encompass additional prospective applications of the growing algorithm. In this broader perspective, each individual unit can be envisioned as an environment with a population of outgoing connections. Contrary to growing hidden units, outgoing connections could be grown based on a probabilistic approach, with higher probabilities favouring growing connections to more proximate units. Similarly, Dai et al. (2019) showed a successful approach to growing connections based on the degree of correlation between pre and post-connections in a CNN. Their method began with a sparsely connected seed architecture, where only 30% of connections were randomly activated. During the growth phase, dormant weight connections were activated based on the degree of correlation that reduced the loss quickly. Intriguingly, this growth policy led to higher connection densities grown at the center of images, corresponding to the placement of digits in the MNIST dataset. Alternatively, the network at large could be considered as the environment wherein the layers form the population. Instead of making predefined assumptions about the required number of layers, layers could be dynamically grown as needed. Furthermore, it is conceivable to apply the growing algorithm across multiple levels of network topology simultaneously, facilitating the growth of connections, units, and layers. If the available space was considered, dynamic growth could play a pivotal role in shaping the resulting architecture by adjusting the carrying capacity at

each level of the topology. For instance, in scenarios where depth is unconstrained, the network can grow sparser and deeper architecture solutions by constraining the carrying capacity at the level of connections and units. Conversely, the network can grow denser, wider architecture solutions when depth is constrained by constraining the carrying capacity at the layer level.

Lifelong Learning

The real-world environment is dynamic and in constant flux. The inherent flexibility brought on by constructive algorithms can help a system develop to better adapt to the continuous stream of information from a dynamic environment. Nevertheless, for a system to thrive in the real world, it must learn new things without forgetting what it has learned previously. This is known as continual or lifelong learning (Parisi et al., 2019).

Lifelong learning introduces the stability-plasticity dilemma. The system must be stable enough to consolidate what has been previously learned while plastic enough to accommodate new information (Parisi et al., 2019; Thomas et al., 2009). In this regard, ANNs are prone to "catastrophic forgetting." Catastrophic forgetting is when the network forgets the previously learned information when presented with different information (Kowaliw et al., 2014; Ribeiro et al., 1997). Constructive algorithms incorporate plasticity into ANN architectures to accommodate new information. However, these methods are still susceptible to catastrophic forgetting. Rusu et al. (2016) proposed using progressive networks that expand the existing architecture using sub-networks for new information. Alternatively, rather than changing the existing network architecture, Kirkpatrick et al. (2017) proposed the elastic weight consolidation algorithm. This algorithm acts as a synaptic consolidation for ANNs by slowing down the learning on weights that highly contribute to previously learned information—in essence, making aspects of the network temporarily less plastic. For a recent review of lifelong learning and methods for avoiding

catastrophic forgetting, see Parisi et al. (2019). Lifelong learning offers a unique challenge, showing the need for a cooperative interplay between static and adaptive architectures. While constructive algorithms offer flexibility that tailors ANN architectures to match task complexity, they must also strive to incorporate stability to avoid catastrophic forgetting.

Concluding Statement

This thesis presented a novel approach for determining near-optimal topologies of feedforward ANNs through a dynamic, more self-governed growing algorithm inspired by population dynamics. The proposed algorithm leveraged the intrinsic population dynamics of the ANN to adapt its architecture to match task complexity, offering the potential for more compact and efficient structures. Specifically, combining a carrying capacity with population dynamics and network feedback provided a built-in mechanism for dynamic growth, enabling the network to converge to smaller topologies based on the task's demands.

The experimental results showed the effectiveness of the dynamic, more self-governed growing algorithm. Chapter One showed that compared to fixed rules for determining hidden layer sizes, the proposed algorithm leads to smaller topologies while still achieving the desired task performance. Chapter Two showcased the algorithm's self-governance capabilities through systematic investigations and the reduced need for fine-tuning hyperparameters. Furthermore, in Chapter Three, examining the effects of growing hidden layers individually in a sequential or parallel fashion in a multilayer context revealed that growing layers in parallel can yield comparable or even superior performances to growing layers sequentially.

More self-governed growth offers a powerful and efficient solution to grow ANN architectures tailored to task demands automatically. Reducing hyperparameter fine-tuning *a priori* can facilitate the emergence of inherent intelligent behaviour within the network. This objective holds great promise for enhancing the generalization ability of ANNs, paving the way for more effective and adaptive problem-solving capabilities in various domains.

References

Aeberhard, S., & Forina, M. (1991). *Wine*. UCI Machine Learning Repository. https://doi.org/https://doi.org/10.24432/C5PC7J

Ang, J. H., Tan, K. C., & Al-Mamun, A. (2008). Training neural networks for classification using growth probability-based evolution. *Neurocomputing*, *71*(16–18), 3493–3508. https://doi.org/10.1016/j.neucom.2007.10.011

Aran, O., & Alpaydın, E. (2003). An incremental neural network construction algorithm for training multilayer perceptrons. *Artificial Neural Networks and Neural Information Processing. Istanbul, Turkey: ICANN/ICONIP*.

Aran, O., Yildiz, O. T., & Alpaydin, E. (2009). An incremental framework based on cross-validation for estimating the architecture of a multilayer perceptron. *International Journal of Pattern Recognition and Artificial Intelligence*, *23*(2), 159–190. https://doi.org/10.1142/S0218001409007132

Arena, P., Caponetto, R., Fortuna, L., & Xibilia, M. G. (1992). Genetic algorithms to select optimal neural network topology. *Midwest Symposium on Circuits and Systems*, 1381–1383. https://doi.org/10.1109/MWSCAS.1992.271082

Asadi, R., & Kareem, S. A. (2014). Review of feed forward neural network classification preprocessing techniques. *American Institute of Physics (AIP) Conference Proceedings*, *1602*(1), 567–573. https://doi.org/10.1063/1.4882541

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science*, *1*(4), 365–375. https://doi.org/10.1080/09540098908915647

Ashfahani, A., & Pratama, M. (2019). Autonomous deep learning: Continual learning approach for dynamic environments. *Proceedings of the 2019 SIAM International Conference on Data*

Mining, 666–674. https://doi.org/10.1137/1.9781611975673.75

Augasta, M. G., & Kathirvalavakumar, T. (2013). Pruning algorithms of neural networks - A comparative study. *Open Computer Science*, *3*(3), 105–115. https://doi.org/10.2478/s13537-013-0109-x

Azimi-Sadjadi, M. R., Sheedvash, S., & Trujillo, F. O. (1993). Recursive dynamic node creation in multilayer neural networks. *IEEE Transactions on Neural Networks*, *4*(2), 242–256. https://doi.org/10.1109/72.207612

Baluja, S., & Fahlman, S. E. (1994). *Reducing network depth in the cascade-correlation learning architecture*. https://doi.org/https://doi.org/10.21236/ada289352

Barakat, M., Druaux, F., Lefebvre, D., Khalil, M., & Mustapha, O. (2011). Self adaptive growing neural network classifier for faults detection and diagnosis. *Neurocomputing*, *74*(18), 3865–3876. https://doi.org/10.1016/J.NEUCOM.2011.08.001

Bartlett, E. B. (1994). Dynamic node architecture learning: An information theoretic approach. *Neural Networks*, *7*(1), 129–140. https://doi.org/10.1016/0893-6080(94)90061-2

Baum, E. B. (1989). A proposal for more powerful learning algorithms. *Neural Computation*, *1*(2), 201–207. https://doi.org/10.1162/neco.1989.1.2.201

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798–1828. https://doi.org/10.1109/TPAMI.2013.50

Berry, M. J., & Linoff, G. (1997). *Data mining techniques: For marketing, sales, and customer support.* John Wiley & Sons, Inc.

Bertini, J. R., & Do Carmo Nicoletti, M. (2009). A feedforward constructive neural network algorithm for multiclass tasks based on linear separability. *Constructive Neural Networks*,

258, 145–169. https://doi.org/10.1007/978-3-642-04512-7_8

Besnard, E., Schmitz, A., Hefazi, H., & Shinde, R. (2007). Constructive neural networks and their application to ship multidisciplinary design optimization. *Journal of Ship Research*, *51*(4), 297–312. https://doi.org/10.5957/jsr.2007.51.4.297

Bi, W., Wang, X., Tang, Z., & Tamura, H. (2005). Avoiding the local minima problem in backpropagation algorithm with modified error function. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, *E88-A*(12), 3645–3653. https://doi.org/10.1093/ietfec/e88-a.12.3645

Bianchini, M., & Scarselli, F. (2014). On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems*, *25*(8), 1553–1565. https://doi.org/10.1109/TNNLS.2013.2293637

Bilbao, I., & Bilbao, J. (2017). Overfitting problem and the over-training in the era of data: Particularly for artificial neural networks. *2017 IEEE 8th International Conference on Intelligent Computing and Information Systems, ICICIS 2017*, 173–177. https://doi.org/10.1109/INTELCIS.2017.8260032

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., & Guttag, J. (2020). What is the state of neural network pruning? *Proceedings of Machine Learning and Systems*, 129–146. .

Blum, A. (1992). *Neural networks in C++: An object-oriented framework for building connectionist systems*. John Wiley & Sons, Inc. https://dl.acm.org/doi/book/10.5555/129269

Boger, Z., & Guterman, H. (1997). Knowledge extraction from artificial neural networks models. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, *4*, 3030–3035. https://doi.org/10.1109/icsmc.1997.633051

Boughrara, H., Chtourou, M., Ben Amar, C., & Chen, L. (2016). Facial expression recognition

based on a mlp neural network using constructive training algorithm. *Multimedia Tools and Applications*, *75*(2), 709–731. https://doi.org/10.1007/s11042-014-2322-6

Branke, J. (1995). Evolutionary algorithms in neural network design and training-A review. *Proceedings of the 1st Nordic Workshop on Genetic Algorithms and Its Applictions*, 145–165.

Cai, G. W., Fang, Z., & Chen, Y. F. (2019). Estimating the number of hidden nodes of the single-hidden-layer feedforward neural networks. *Proceedings - 2019 15th International Conference on Computational Intelligence and Security, CIS 2019*, 172–176. https://doi.org/10.1109/CIS.2019.00044

Castillo, P. A., Merelo, J. J., Prieto, A., Rivas, V., & Romero, G. (2000). G-Prop: Global optimization of multilayer perceptrons using GAs. *Neurocomputing*, *35*(1–4), 149–163. https://doi.org/10.1016/S0925-2312(00)00302-7

Chaber, P., & Ławryńczuk, M. (2018). Pruning of recurrent neural models: An optimal brain damage approach. *Nonlinear Dynamics*, *92*(2), 763–780. https://doi.org/10.1007/S11071-018-4089-1/FIGURES/18

Craik, A., He, Y., & Contreras-Vidal, J. L. (2019). Deep learning for electroencephalogram (EEG) classification tasks a review. *Journal of Neural Engineering*, *16*(3). https://doi.org/10.1088/1741-2552/ab0ab5

Curteanu, S., & Cartwright, H. (2011). Neural networks applied in chemistry. I. Determination of the optimal topology of multilayer perceptron neural networks. *Journal of Chemometrics*, *25*(10), 527–549. https://doi.org/10.1002/cem.1401

Dahal, P. (2017). *Softmax and cross entropy loss*. https://www.parasdahal.com/softmax-crossentropy

Dai, X., Yin, H., & Jha, N. K. (2019). NeST: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, *68*(10), 1487–1497. https://doi.org/10.1109/TC.2019.2914438

Ding, S., Li, H., Su, C., Yu, J., & Jin, F. (2013). Evolutionary artificial neural networks: A review. *Artificial Intelligence Review*, *39*(3), 251–260. https://doi.org/10.1007/s10462-011-9270-6

Ding, S., Su, C., & Yu, J. (2011). An optimizing BP neural network algorithm based on genetic algorithm. *Artificial Intelligence Review*, *36*(2), 153–162. https://doi.org/10.1007/s10462-011-9208-z

Do Carmo Nicoletti, M., Bertini, J. R., Elizondo, D., Franco, L., & Jerez, J. M. (2009). Constructive neural network algorithms for feedforward architectures suitable for classification tasks. *Constructive Neural Networks*, *258*, 1–23. https://doi.org/10.1007/978-3-642-04512-7_1

Eldan, R., & Shamir, O. (2016). The power of depth for feedforward neural networks. *Conference on Learning Theory*, *49*, 907–940. https://proceedings.mlr.press/v49/eldan16.html

Engelbrecht, A. P., & Cloete, I. (1996). Sensitivity analysis algorithm for pruning feedforward neural networks. *IEEE International Conference on Neural Networks*, *2*, 1274–1278. https://doi.org/10.1109/icnn.1996.549081

Fahlman, S. E., & Lebiere, C. (1990). The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems*, 524–532.

Fausett, L. (1994). *Fundamentals of neural networks — architectures, algorithms, and applications* (pp. 289–302). Prentice-Hall Inc. https://doi.org/10.1177/002072099503200320

Fernandes, F. A. N., & Lona, L. M. F. (2005). Neural network applications in polymerization processes. *Brazilian Journal of Chemical Engineering*, *22*(3), 401–418.

https://doi.org/10.1590/S0104-66322005000300009

Fontes, C. H., & Embiruçu, M. (2021). An approach combining a new weight initialization method and constructive algorithm to configure a single feedforward neural network for multi-class classification. *Engineering Applications of Artificial Intelligence*, *106*, 104495. https://doi.org/10.1016/J.ENGAPPAI.2021.104495

Frean, M. (1990). The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, *2*(2), 198–209. https://doi.org/10.1162/neco.1990.2.2.198

Friedman, J. H., & Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American Statistical Association*, *76*(376), 817–823. https://doi.org/10.1080/01621459.1981.10477729

Funahashi, K. I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks*, *2*(3), 183–192. https://doi.org/10.1016/0893-6080(89)90003-8

Fung, H. K., & Li, L. K. (2001). Minimal feedforward parity networks using threshold gates. *Neural Computation*, *13*(2), 319–326. https://doi.org/10.1162/089976601300014556

Gallant, S. I. (1990). Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, *1*(2), 179–191. https://doi.org/10.1109/72.80230

Gaurang, P., & Panchal, D. (2011). Behaviour analysis of multilayer perceptrons with multiple hidden neurons and hidden layers. *International Journal of Computer Theory and Engineering*, *3*(2), 332–337. https://doi.org/10.7763/IJCTE.2011.V3.328

Guan, S. U., & Li, S. (2002). Parallel growing and training of neural networks using output parallelism. *IEEE Transactions on Neural Networks*, *13*(3), 542–550. https://doi.org/10.1109/TNN.2002.1000123

Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., & Lew, M. S. (2016). Deep learning for visual

understanding: A review. *Neurocomputing*, *187*, 27–48. https://doi.org/10.1016/J.NEUCOM.2015.09.116

Han, H. G., Zhang, S., & Qiao, J. F. (2017). An adaptive growing and pruning algorithm for designing recurrent neural network. *Neurocomputing*, *242*, 51–62. https://doi.org/10.1016/J.NEUCOM.2017.02.038

Hassibi, B., & Stork, D. (1993). Second order derivaties for network prunning: Optimal brain surgeon. *Proceedings of Neural Information Processing Systems, 5*, 164–171.

Haykin, S. (2009). Neural networks and Learning Machines Third Edition. In *Neural Networks and Learning Machines* (Third). Prentice Hall/Pearson. http://dspace.fue.edu.eg/xmlui/bitstream/handle/123456789/3421/5618.pdf?sequence=1

Hernández-Espinosa, C., & Fernández-Redondo, M. (2002). On the design of constructive training algorithms for multilayer feedforward. *Proceedings of the International Joint Conference on Neural Networks*, *1*, 890–895. https://doi.org/10.1109/ijcnn.2002.1005592

Hirose, Y., Yamashita, K., & Hijiya, S. (1991). Back-propagation algorithm which varies the number of hidden units. *Neural Networks*, *4*(1), 61–66. https://doi.org/10.1016/0893-6080(91)90032-Z

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., & Peste, A. (2021). Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *The Journal of Machine Learning Research*, *23*, 1–124. https://doi.org/10.5555/3546258.3546499

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Huang, G.-B., Chen, L., & Siew, C.-K. (2006). Universal approximation using incremental

constructive feedforward networks with random hidden nodes. *IEEE Transactions on Neural Networks*, *17*(04), 879--892. https://doi.org/10.1109/TNNLS.2017.2753725

Huang, G. B., & Chen, L. (2008). Enhanced random search based incremental extreme learning machine. *Neurocomputing*, *71*(16–18), 3460–3468. https://doi.org/10.1016/J.NEUCOM.2007.10.008

Hunter, D., Yu, H., Pukish, M. S., Kolbusz, J., & Wilamowski, B. M. (2012). Selection of proper neural network sizes and architectures-A comparative study. *IEEE Transactions on Industrial Informatics*, *8*(2), 228–240. https://doi.org/10.1109/TII.2012.2187914

Islam, M. M., & Murase, K. (2001). A new algorithm to design compact two-hidden-layer artificial neural networks. *Neural Networks*, *14*(9), 1265–1278. https://doi.org/10.1016/S0893-6080(01)00075-2

Islam, M. M., Sattar, A., Amin, M. F., Yao, X., & Murase, K. (2009a). A new constructive algorithm for architectural and functional adaptation of artificial neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *39*(6), 1590–1605. https://doi.org/10.1109/TSMCB.2009.2021849

Islam, M. M., Sattar, M. A., Amin, M. F., Yao, X., & Murase, K. (2009b). A new adaptive merging and growing algorithm for designing artificial neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *39*(3), 705–722. https://doi.org/10.1109/TSMCB.2008.2008724

Jinchuan, K., & Xinzhe, L. (2008). Empirical analysis of optimal hidden neurons in neural network modeling for stock prediction. *Proceedings - 2008 Pacific-Asia Workshop on Computational Intelligence and Industrial Application, PACIIA 2008*, *2*, 828–832. https://doi.org/10.1109/PACIIA.2008.363

Johnson, M. G., & Chartier, S. (2017). Spike neural models part I: The Hodgkin-Huxley model. *The Quantitative Methods for Psychology*, *13*(2). https://doi.org/10.20982/tqmp.13.2.p105

Juliano, S. A. (2007). Population dynamics. *Journal of the American Mosquito Control Association*, *23*(2), 265–275.

Kamruzzaman, S. M., Hasan, A. R., Siddiquee, A. B., & Mazumder, M. E. H. (2004). Medical diagnosis using neural network. *3rd International Conference on Electrical & Computer Engineering (ICECE)*. http://arxiv.org/abs/1009.4572

Karsoliya, S. (2012). Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture. *International Journal of Engineering Trends and Technology*, *3*(6), 714–717.

Khan, W. A., Chung, S. H., Awan, M. U., & Wen, X. (2020). Machine learning facilitated business intelligence (Part II): Neural networks optimization techniques and applications. *Industrial Management and Data Systems*, *120*(1), 128–163. https://doi.org/10.1108/IMDS-06-2019-0351

Khaw, J. F. C., Lim, B. S., & Lim, L. E. N. (1995). Optimal design of neural networks using the Taguchi method. *Neurocomputing*, *7*(3), 225–245. https://doi.org/10.1016/0925-2312(94)00013-I

Kherif, F., & Latypova, A. (2020). Principal component analysis. *Machine Learning: Methods and Applications to Brain Disorders*, 209–225. https://doi.org/10.1016/B978-0-12-815739-8.00012-2

Kingma, D. P., & Ba, J. L. (2015, December 22). Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. https://arxiv.org/abs/1412.6980v9

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences of the United States of America*, *114*(13), 3521–3526. https://doi.org/10.1073/PNAS.1611835114/-/DCSUPPLEMENTAL

Kohonen, T. (2013). Essentials of the self-organizing map. *Neural Networks*, *37*, 52–65. https://doi.org/10.1016/J.NEUNET.2012.09.018

Kowaliw, T., Bredeche, N., Chevallier, S., & Doursat, R. (2014). Artificial neurogenesis: An introduction and selective review. *Growing Adaptive Machines: Combining Development and Learning in Artificial Neural Networks*, *557*, 1–60. https://doi.org/10.1007/978-3-642-55337-0_1

Kwok, T. Y., & Yeung, D. Y. (1997a). Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE Transactions on Neural Networks*, *8*(3), 630–645. https://doi.org/10.1109/72.572102

Kwok, T. Y., & Yeung, D. Y. (1997b). Objective functions for training new hidden units in constructive neural networks. *IEEE Transactions on Neural Networks*, *8*(5), 1131–1148. https://doi.org/10.1109/72.623214

Kwok, T. Y., & Yeung, D. Y. (1993). Experimental analysis of input weight freezing in constructive neural networks. *IEEE International Conference on Neural Networks*, 511–513. https://doi.org/10.1109/icnn.1993.298610

Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature 2015 521:7553*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to

document recognition. *Proceedings of the IEEE*, *86*(11). https://doi.org/10.1109/5.726791

LeCun, Y., Denker, J. S., & Solla, S. A. (1990). Optimal brain damage. *Advances in Neural Information Processing Systems*, *2*, 598–605.

Lehtokangas, M. (1999). Modelling with constructive backpropagation. *Neural Networks*, *12*(4–5), 707–716. https://doi.org/10.1016/S0893-6080(99)00018-0

Li, Z., Cheng, G., & Qiang, X. (2010). Some classical constructive neural networks and their new developments. *ICENT 2010 - 2010 International Conference on Educational and Network Technology*, 174–178. https://doi.org/10.1109/ICENT.2010.5532201

Liu, D., Chang, T.-S., & Zhang, Y. (2002). A constructive algorithm for feedforward neural networks with incremental training. *IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications*, *49*(12). https://doi.org/10.1109/TCSI.2002.805733

Lu, Y., Yang, J., Wang, Q., & Huang, Z. J. (2012). The upper bound of the minimal number of hidden neurons for the parity problem in binary neural networks. *Science China Information Sciences*, *55*(7), 1579–1587. https://doi.org/10.1007/s11432-011-4405-6

Ma, L., & Khorasani, K. (2002). Application of adaptive constructive neural networks to image compression. *IEEE Transactions on Neural Networks*, *13*(5), 1112–1126. https://doi.org/10.1109/TNN.2002.1031943

Ma, L., & Khorasani, K. (2003). A new strategy for adaptively constructing multilayer feedforward neural networks. *Neurocomputing*, *51*, 361–385. https://doi.org/10.1016/S0925-2312(02)00597-0

Ma, L., & Khorasani, K. (2004a). New training strategies for constructive neural networks with application to regression problems. *Neural Networks*, *17*(4), 589–609. https://doi.org/10.1016/j.neunet.2004.02.002

Ma, L., & Khorasani, K. (2004b). Facial expression recognition using constructive feedforward neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *34*(3), 1588–1595. https://doi.org/10.1109/TSMCB.2004.825930

Masmoudi, S., Frikha, M., Chtourou, M., & Hamida, A. Ben. (2011). Efficient MLP constructive training algorithm using a neuron recruiting approach for isolated word recognition system. *International Journal of Speech Technology*, *14*(1), 1–10. https://doi.org/10.1007/s10772-010-9082-0

Mézard, M. (1989). Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A: Mathematical and General*, *22*(12), 2191–2203. https://doi.org/10.1088/0305-4470/22/12/019

Minnick, R. C. (2009). Linear-input logic. *IEEE Transactions on Electronic Computers*, *EC-10*(1), 6–16. https://doi.org/10.1109/tec.1961.5219146

Mohamed, S. A. E. M., Mohamed, M. H., & Farghally, M. F. (2020). Constructive learning of deep neural networks for bigdata analysis. *International Journal of Computer Applications Technology and Research*, *9*(12), 311–322. https://doi.org/10.7753/IJCATR0912.1001

Mohamed, S. A. E. M., Mohamed, M. H., & Farghally, M. F. (2021). A new cascade-correlation growing deep learning neural network algorithm. *Algorithms 2021, Vol. 14, Page 158*, *14*(5), 158. https://doi.org/10.3390/A14050158

Nahmias, M. A., De Lima, T. F., Tait, A. N., Peng, H. T., Shastri, B. J., & Prucnal, P. R. (2020). Photonic Multiply-Accumulate Operations for Neural Networks. *IEEE Journal of Selected Topics in Quantum Electronics*, *26*(1). https://doi.org/10.1109/JSTQE.2019.2941485

Naraan, S., & Tagliarini, G. (2005). An analysis of overfitting in MLP networks. *Proceedings of the International Joint Conference on Neural Networks*, *2*, 984–988.

https://doi.org/10.1109/IJCNN.2005.1555986

Narasimha, P. L., Delashmit, W. H., Manry, M. T., Li, J., & Maldonado, F. (2008). An integrated growing-pruning method for feedforward network training. *Neurocomputing*, *71*(13–15), 2831–2847. https://doi.org/10.1016/j.neucom.2007.08.026

Neal, B., Mittal, S., Baratin, A., Tantia, V., Scicluna, M., Lacoste-Julien, S., & Mitliagkas, I. (2018). A modern take on the bias-variance tradeoff in neural networks. *ArXiv Preprint ArXiv:1810.08591*. http://arxiv.org/abs/1810.08591

Packianather, M. S., Drake, P. R., & Rowlands, H. (2000). Optimizing the parameters of multilayered feedforward neural networks through Taguchi design of experiments. *Quality and Reliability Engineering International*, *16*(6), 461–473. https://doi.org/10.1002/1099-1638(200011/12)16:6<461::AID-QRE341>3.0.CO;2-G

Parekh, R., Yang, J., & Honavar, V. (2000). Constructive neural-network learning algorithms for pattern classification. *IEEE Transactions on Neural Networks*, *11*(2), 436–451. https://doi.org/10.1109/72.839013

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., & Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, *113*, 54–71. https://doi.org/10.1016/J.NEUNET.2019.01.012

Pérez-Sánchez, B., Fontenla-Romero, O., & Guijarro-Berdiñas, B. (2018). A review of adaptive online learning for artificial neural networks. *Artificial Intelligence Review*, *49*(2), 281–299. https://doi.org/10.1007/s10462-016-9526-2

Perrotta, P. (2020). *Killer combo: Softmax and cross entropy*. Medium. https://levelup.gitconnected.com/killer-combo-softmax-and-cross-entropy-5907442f60ba

Pratama, M., Za'in, C., Ashfahani, A., Ong, Y. S., & Ding, W. (2020). Automatic construction of

multi-layer perceptron network from streaming examples. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 1171–1180. https://doi.org/10.1145/3357384.3357946

Psichogios, D. C., & Ungar, L. H. (1994). SVD-NET: An algorithm that automatically selects network structure. *IEEE Transactions on Neural Networks*, *5*(3), 513–515. https://doi.org/10.1109/72.286929

Puma-Villanueva, W. J., dos Santos, E. P., & Von Zuben, F. J. (2012). A constructive algorithm to synthesize arbitrarily connected feedforward neural networks. *Neurocomputing*, *75*(1), 14–32. https://doi.org/10.1016/J.NEUCOM.2011.05.025

Qiao, J., Li, F., Han, H., & Li, W. (2016). Constructive algorithm for fully connected cascade feedforward neural networks. *Neurocomputing*, *182*, 154–164. https://doi.org/10.1016/J.NEUCOM.2015.12.003

Reed, R. (1993). Pruning algorithms—A survey. *IEEE Transactions on Neural Networks*, *4*(5), 740–747. https://doi.org/10.1109/72.248452

Ribeiro, J. N. G., Vasconcelos, G. C., & Queiroz, C. R. O. (1997). Comparative study of the cascade-correlation architecture in pattern recognition applications. *Proceedings of the Brazilian Symposium on Neural Networks, SBRN*, 31–40. https://doi.org/10.1109/sbrn.1997.645846

Rivals, I., & Personnaz, L. (2000). A statistical procedure for determining the optimal number of hidden neurons of a neural model. *Second International Symposium on Neural Computation (NC)*, 23–26.

Ross, M., Berberian, N., & Chartier, S. (2020). Should I stay or should I grow? A dynamic self-governed growth for determining hidden layer size in a multilayer perceptron. *Proceedings*

of the International Joint Conference on Neural Networks. https://doi.org/10.1109/IJCNN48605.2020.9207460

Ross, M., Payeur, P., & Chartier, S. (2019). Task allocation for heterogeneous robots using a self-organizing contextual map. *2019 IEEE International Symposium on Robotic and Sensors Environments (ROSE)*, 1–6. https://doi.org/10.1109/ROSE.2019.8790434

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., & Hadsell, R. (2016). Progressive neural networks. *ArXiv Preprint ArXiv:1606.04671*. https://arxiv.org/abs/1606.04671v3

Sadreddin, A., & Sadaoui, S. (2021). Incremental feature learning using constructive neural networks. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, *2021-Novem*, 704–708. https://doi.org/10.1109/ICTAI52525.2021.00111

Safi, Y., & Bouroumi, A. (2014). Evolutionary single hidden-layer feed forward networks. *International Journal Innovative Computing and Applications*, *6*(2), 73–86. https://doi.org/10.1504/IJICA.2014.066497

Sanger, T. D. (1989). Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, *2*(6), 459–473. https://doi.org/10.1016/0893-6080(89)90044-0

Setiono, R. (1997a). A penalty-function approach for pruning feedforward neural networks. *Neural Computation*, *9*(1), 185–204. https://doi.org/10.1162/neco.1997.9.1.185

Setiono, R. (1997b). On the solution of the parity problem by a single hidden layer feedforward neural network. *Neurocomputing*, *16*(3), 225–235. https://doi.org/10.1016/S0925-2312(97)00030-1

Shamsuddin, M. R., Abdul-Rahman, S., & Mohamed, A. (2018). Exploratory analysis of MNIST handwritten digit for machine learning modelling. *Communications in Computer and Information Science*, *937*, 134–145. https://doi.org/10.1007/978-981-13-3441-2_11

Sharma, S. K., & Chandra, P. (2010a). Constructive neural networks: A review. *International Journal of Engineering Science and Technology*, *2*(12), 7847–7855. https://www.researchgate.net/publication/50384469

Sharma, S. K., & Chandra, P. (2010b). An adaptive slope basic dynamic node creation algorithm for single hidden layer neural networks. *Proceedings - 2010 International Conference on Computational Intelligence and Communication Networks, CICN 2010*, 139–144. https://doi.org/10.1109/CICN.2010.38

Shultz, T. R. (2012). A constructive neural-network approach to modeling psychological development. *Cognitive Development*, *27*(4), 383–400. https://doi.org/10.1016/J.COGDEV.2012.08.002

Siddiquee, A. B., Mazumder, M. E. H., & Kamruzzaman, S. M. (2010). A constructive algorithm for feedforward neural networks for medical diagnostic reasoning. *ArXiv Preprint ArXiv:1009.4564*. https://doi.org/https://doi.org/10.48550/arXiv.1009.4564

Śmieja, F. J. (1993). Neural network constructive algorithms: Trading generalization for learning efficiency? *Circuits, Systems, and Signal Processing*, *12*(2), 331–374. https://doi.org/10.1007/BF01189880

Subirats, José L, Jerez, J. M., Iván Gómez, •, & Franco, L. (2010). Multiclass pattern recognition extension for the new C-Mantec constructive neural network algorithm. *Cognitive Computation*, *2*, 285–290. https://doi.org/10.1007/s12559-010-9051-6

Subirats, José Luis, Franco, L., Molina Conde, I., & Jerez, J. M. (2008). Active learning using a

constructive neural network algorithm. *Artificial Neural Networks-ICANN 2008: 18th International Conference, Prague, Czech Republic*, *5164 LNCS*(PART II), 803–811. https://doi.org/10.1007/978-3-540-87559-8_83

Sun, G. Q. (2016). Mathematical modeling of population dynamics with Allee effect. In *Nonlinear Dynamics* (Vol. 85, Issue 1). Springer Netherlands. https://doi.org/10.1007/s11071-016-2671-y

Susan, S., & Dwivedi, M. (2014). Dynamic growth of hidden-layer neurons using the non-extensive entropy. *Proceedings - 2014 4th International Conference on Communication Systems and Network Technologies, CSNT 2014*, 491–495. https://doi.org/10.1109/CSNT.2014.104

Teoh, E. J., Tan, K. C., & Xiang, C. (2006). Estimating the number of hidden neurons in a feedforward network using the singular value decomposition. *IEEE Transactions on Neural Networks*, *17*(6), 1623–1629. https://doi.org/10.1109/TNN.2006.880582

Thivierge, J. P., Rivest, F., & Shultz, T. R. (2003). A dual-phase technique for pruning constructive networks. *Proceedings of the International Joint Conference on Neural Networks*, *1*, 559–564. https://doi.org/10.1109/ijcnn.2003.1223407

Thomas, A. J., Petridis, M., Walters, S. D., Gheytassi, S. M., & Morgan, R. E. (2016). On predicting the optimal number of hidden nodes. *Proceedings - 2015 International Conference on Computational Science and Computational Intelligence, CSCI 2015*, 565–570. https://doi.org/10.1109/CSCI.2015.33

Thomas, M. S. C., McClelland, J. L., Richardson, F. M., Schapiro, A. C., & Baughman, F. D. (2009). Dynamic and connectionist approaches to development: Toward a future of mutually beneficial coevolution. In *Toward a Unified Theory of Development Connectionism and*

Dynamic System Theory Re-Consider (pp. 337–353). Oxford University PressNew York. https://doi.org/10.1093/acprof:oso/9780195300598.003.0017

Thórisson, K. R. (2012). A new constructivist AI: From manual methods to self-constructive systems. In *Theoretical Foundations of Artificial General Intelligence* (pp. 145–171). Atlantis Press, Paris. https://doi.org/10.2991/978-94-91216-62-6_9

Tissera, M. D., & McDonnell, M. D. (2016). Modular expansion of the hidden layer in single layer feedforward neural networks. *Proceedings of the International Joint Conference on Neural Networks*, 2939–2945. https://doi.org/10.1109/IJCNN.2016.7727571

Tsoularis, A., & Wallace, J. (2002). Analysis of logistic growth models. *Mathematical Biosciences*, *179*(1), 21–55. https://doi.org/10.1016/S0025-5564(02)00096-2

Wang, S. De, & Hsu, C. H. (1991). A self growing learning algorithm for determining the appropriate number of hidden units. *IEEE International Joint Conference on Neural Networks*, 1098–1104. https://doi.org/10.1109/ijcnn.1991.170543

Wilamowski, B. M., Hunter, D., & Malinowski, A. (2003). Solving parity-n problems with feedforward neural networks. *Proceedings of the International Joint Conference on Neural Networks*, *4*, 2546–2551. https://doi.org/10.1109/ijcnn.2003.1223966

Wolberg, W., Mangasarian, O., Street, N., & Street, W. (1995). *Breast Cancer Wisconsin (Diagnostic)*. UCI Machine Learning Repository. https://doi.org/https://doi.org/10.24432/C5DW2B

Wu, X., Rozycki, P., Kolbusz, J., & Wilamowski, B. M. (2019). Constructive cascade learning algorithm for fully connected networks. *Artificial Intelligence and Soft Computing: 18th International Conference, ICAISC 2019, Zakopane, Poland*, *11508 LNAI*, 236–247. https://doi.org/10.1007/978-3-030-20912-4_23

Wu, X., Rózycki, P., & Wilamowski, B. M. (2015). A hybrid constructive algorithm for single-layer feedforward networks learning. *IEEE Transactions on Neural Networks and Learning Systems*, *26*(8), 1659–1668. https://doi.org/10.1109/TNNLS.2014.2350957

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *ArXiv Preprint ArXiv:1708.07747*. https://doi.org/http://doi.org/10.48550/arXiv.1708.07747

Xu, S., & Chen, L. (2008, June 23). A novel approach for determining the optimal number of hidden layer neurons for FNN's and its application in data mining. *5th International Conference on Information Technology and Applications (ICITA 2008)*.

Yang, Z., Yu, Y., You, C., Steinhardt, J., & Ma, Y. (2020). Rethinking bias-variance trade-off for generalization of neural networks. *International Conference on Machine Learning*, 10767–10777. http://proceedings.mlr.press/v119/yang20j.html

Yao, X. (1993). A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, *8*(4), 539–567. https://doi.org/10.1002/int.4550080406

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447. https://doi.org/10.1109/5.784219

Ying, X. (2019). An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, *1168*(2), 22022. https://doi.org/10.1088/1742-6596/1168/2/022022

Young, S., & Downs, T. (1998). CARVE - A constructive algorithm for real-valued examples. *IEEE Transactions on Neural Networks*, *9*(6), 1180–1190. https://doi.org/10.1109/72.728361

Zemouri, R. (2017). An evolutionary building algorithm for deep neural networks. *12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization, WSOM 2017 - Proceedings*, 1–7.

https://doi.org/10.1109/WSOM.2017.8020002

Zemouri, R., Omri, N., Fnaiech, F., Zerhouni, N., & Fnaiech, N. (2020). A new growing pruning deep learning neural network algorithm (GP-DLNN). *Neural Computing and Applications 2019 32:24*, *32*(24), 18143–18159. https://doi.org/10.1007/S00521-019-04196-8

Zemouri, R., Omri, N., Morello, B., Devalland, C., Arnould, L., Zerhouni, N., & Fnaiech, F. (2018). Constructive deep neural network for breast cancer diagnosis. *IFAC-PapersOnLine*, *51*(27), 98–103. https://doi.org/10.1016/j.ifacol.2018.11.660

Zeng, X., & Yeung, D. S. (2006). Hidden neuron pruning of multilayer perceptrons using a quantified sensitivity measure. *Neurocomputing*, *69*(7-9 SPEC. ISS.), 825–837. https://doi.org/10.1016/j.neucom.2005.04.010

Zhang, B. T. (1994). Incremental learning algorithm that optimizes network size and sample size in one trial. *IEEE International Conference on Neural Networks - Conference Proceedings*, *1*, 215–220. https://doi.org/10.1109/icnn.1994.374165

Zhao, Z. Q., Zheng, P., Xu, S. T., & Wu, X. (2019). Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, *30*(11), 3212–3232. https://doi.org/10.1109/TNNLS.2018.2876865

Zhong, G., Ling, X., & Wang, L. N. (2019). From shallow feature learning to deep learning: Benefits from the width and depth of deep architectures. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *9*(1), e1255. https://doi.org/10.1002/widm.1255

Appendix A

The following derivations were adapted from Dahal (2017) and Perrotta (2020).

The Derivative of the Softmax Activation Function

The softmax function is:

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}} \quad \text{for } \forall j \in \{1, \dots, N\}$$

where a vector (a) of real-valued classification scores with dimension N (the number of classes) is taken as input and an N-dimensional vector with values squashed between zero and one, that sum to one, is outputted ($\hat{y}$). It maps $\hat{y}(a)$: $\mathbb{R}^N \to \mathbb{R}^N$.

$$\hat{y}(a): \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} \to \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{bmatrix}$$

For an arbitrary i and k, the $\frac{\partial \hat{y}_i}{\partial a_k}$ is as follows:

$$\frac{\partial \hat{y}_i}{\partial a_k} = \frac{\partial \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}}{\partial a_k}$$

Following the quotient rule for $f(x) = \frac{g(x)}{h(x)} : f'(x) = \frac{g\prime(x)h(x) - g(x)h\prime(x)}{h(x)^2}$

In the case of the softmax: $h_i = \sum_{j=1}^{N} e^{a_j}$ and $g_i = e^{a_i}$.

For $h_i{}'$ with respect to any a_k:

$$\frac{\partial}{\partial a_k} h_i = \frac{\partial}{\partial a_k} \sum_{j=1}^{N} e^{a_j}$$

$$= \sum_{j=1}^{N} \frac{\partial}{\partial a_k} e^{a_j}$$

$$= e^{a_k}$$

While for $g_i{}'$ there are two cases: $g_i{}'$ is only e^{a_k} if $i = k$, otherwise $g_k{}'$ is 0.

Following the quotient rule:

$if\ i = k,$

$$\frac{\partial \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}}{\partial a_k} = \frac{e^{a_i}\sum_{j=1}^{N} e^{a_j} - e^{a_k}e^{a_i}}{\left(\sum_{j=1}^{N} e^{a_j}\right)^2}$$

$$= \frac{e^{a_i}\left(\sum_{j=1}^{N} e^{a_j} - e^{a_k}\right)}{\left(\sum_{j=1}^{N} e^{a_j}\right)^2}$$

$$= \frac{e^{a_k}}{\sum_{j=1}^{N} e^{a_j}} \cdot \frac{\left(\sum_{j=1}^{N} e^{a_j} - e^{a_k}\right)}{\sum_{j=1}^{N} e^{a_j}}$$

$$= \hat{y}_i(1 - \hat{y}_k)$$

since $i = k$, it can be rewritten as:

$$= \hat{y}_k(1 - \hat{y}_k)$$

$if\ i \neq k,$

$$\frac{\partial \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}}{\partial a_k} = \frac{0 - e^{a_k}e^{a_i}}{\left(\sum_{j=1}^{N} e^{a_j}\right)^2}$$

$$= \frac{-e^{a_k}}{\sum_{j=1}^{N} e^{a_j}} \cdot \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$$

$$= \hat{y}_k\hat{y}_i$$

The derivative of the softmax is then:

$$\frac{\partial \hat{y}_i}{\partial a_k} = \begin{cases} \hat{y}_k(1 - \hat{y}_k), & if\ i = k \\ -\hat{y}_k\hat{y}_i\,, & if\ i \neq k \end{cases}$$

The Derivative of the Cross Entropy Loss Function with Softmax

The cross-entropy loss function provides a measure of distance between the probability distribution of the softmax output $\hat{y}$ (predicted classification), and the probability distribution of the correct classification given by y, a one-hot encoded vector of size N.

$$J(y, \hat{y}) = -\sum_{k=1}^{N} y_k \log \hat{y}_k$$

The derivative of the cross-entropy loss function with respect to the softmax function for a single output is as follows:

$$\frac{\partial J}{\partial a_k} = \sum_i -y_i \frac{\partial \log \hat{y}_i}{\partial a_k}$$

Apply Chain Rule:

$$= -\sum_i y_i \frac{\partial \log(\hat{y}_i)}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial a_k}$$

$$= -\sum_i y_i \frac{1}{\hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial a_k}$$

$$= -\sum_i \frac{y_i}{\hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial a_k}$$

Insert the derivative of the softmax function:

$$= -\sum_i \frac{y_i}{\hat{y}_i} \cdot [\hat{y}_k(1 - \hat{y}_k) - \hat{y}_k \hat{y}_i]$$

Separate the two cases of the softmax derivative $i = k$ and $i \neq k$:

$$= -\sum_{i=k} \frac{y_i}{\hat{y}_i} \cdot \hat{y}_k(1 - \hat{y}_k) + \sum_{i \neq k} \frac{y_i}{\hat{y}_i} \cdot (\hat{y}_k \hat{y}_i)$$

For the case of $i = k$, y_i and y_k are the same.

$$= -y_i + y_k\hat{y}_k + \sum_{i \neq k} y_i\,\hat{y}_k$$

Merge the two cases of the softmax derivative:

$$= -y_k \sum_{i} y_i\,\hat{y}_k$$

Since y is a one-hot encoded vector, $\sum_i y_i$ all it's elements are 0, except for one element that is 1. This means the sum simplifies to:

$$= -y_k + 1\hat{y}_k$$

Rewritten as:

$$= \hat{y}_k - y_k$$

Appendix B

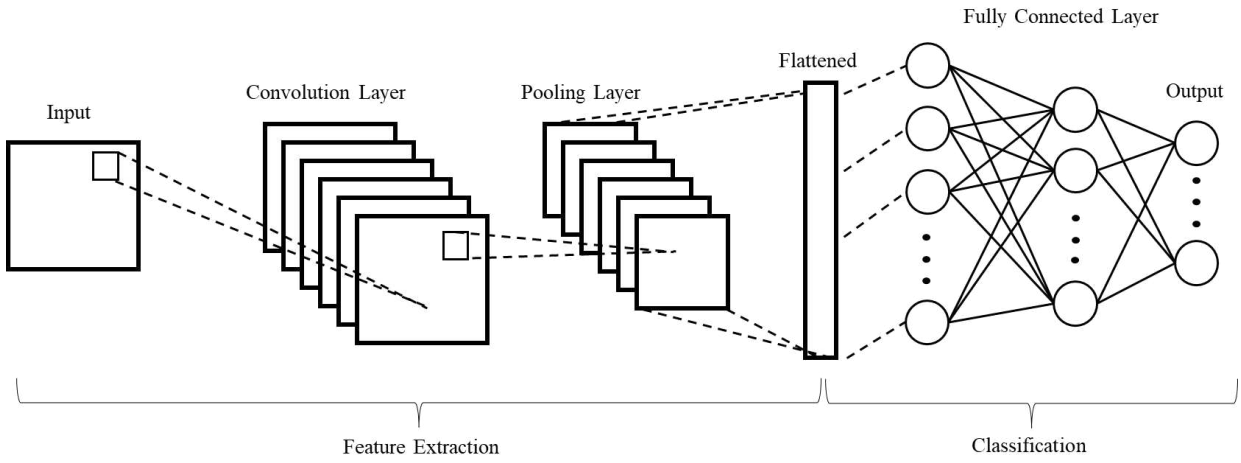

Figure B.1. Convolutional neural network (CNN) architecture.