arXiv:2501.15889v4 [cs.LG] 21 May 2025

Adaptive Width Neural Networks

Federico Errica
NEC Laboratories Europe

Henrik Christiansen
NEC Laboratories Europe

Viktor Zaverkin
NEC Laboratories Europe

Mathias Niepert
University of Stuttgart
NEC Laboratories Europe

Francesco Alesiani
NEC Laboratories Europe

Abstract

For almost 70 years, researchers have primarily relied on hyper-parameter tuning to select the width of neural networks' layers. This paper challenges the status quo by introducing an easy-to-use technique to learn an *unbounded* width of a neural network's layer *during training*. The method jointly optimizes the width and the parameters of each layer via standard backpropagation. We apply the technique to a broad range of data domains such as tables, images, text, sequences, and graphs, showing how the width adapts to the task's difficulty. A by product of our width learning approach is the easy *truncation* of the trained network at virtually zero cost, achieving a smooth trade-off between performance and compute resources. Alternatively, one can dynamically compress the network until performances do not degrade. In light of recent foundation models trained on large datasets, requiring billions of parameters and where hyper-parameter tuning is unfeasible due to huge training costs, our approach introduces a viable alternative for width learning.

1 Introduction

Since the construction of the Mark I Perceptron machine [48] the effective training of neural networks has remained an open research problem of great academic and practical value. The Mark I solved image recognition tasks by exploiting a layer of 512 *fixed* "association units" that in modern language correspond to the hidden units of a Multi-Layer Perceptron (MLP). MLPs possess universal approximation capabilities when assuming *arbitrary width* [10] and sigmoidal activations, and their convergence to good solutions was studied, for instance, in Rumelhart et al. [50] where the backpropagation algorithm was introduced as "simple, easy to implement on parallel hardware", and improvable by other techniques such as momentum that preserve locality of weight updates.

Yet, after almost 70 years of progress [34], the vast majority of neural networks, be they shallow or deep, still rely on a fixed choice of the number of neurons in their hidden layers. The width is typically treated as one of the many hyper-parameters that have to be carefully tuned whenever we approach a new task [62]. The tuning process has many names, such as model selection, hyper-parameter tuning, and cross-validation, and it is associated with non-negligible costs: Different architectural configurations are trained until one that performs best on a validation set is selected [41]. The configurations' space grows exponentially in the number of layers, so practitioners often resort to shortcuts such as picking a specific number of hidden units for *all* layers, which greatly reduces the search space together with the chances of selecting a better architecture for the task. Other techniques to tune hyper-parameters include constructive approaches [17, 64], which alternate parameter optimization and creation of new neurons, natural gradient-based heuristics that dynamically modify the network [40], bi-level optimization [19], and neural architecture search [61], which often requires separate training runs for each configuration.

Preprint. Under review.

The hyper-parameters' space exploration problem is exacerbated by the steep increase in size of recent neural architectures for language [9] and vision [68], for example, where parameters are in the order of billions to accommodate for a huge dataset. Training these models requires an amount of time, compute power, and energy that currently makes it unfeasible for most institutions to perform a thorough model selection and find good width parameters; the commonly accepted compromise is to stick to previously successful hyper-parameter choices. This may also explain why network pruning [6, 39], distillation [69] and quantization [39] techniques have recently been in the spotlight, as they trade-off hardware requirements and performance.

This work introduces a *simple* and *easy to use* technique to learn the width of each neural network's layer without imposing upper bounds (we refer to it as **unbounded width**). The width of each layer is *dynamically* adjusted during backpropagation [44], and it only requires a slight modification to the neural activations that does not alter the ability to parallelize computation. The technical strategy is to impose a soft ordering of hidden units by exploiting any monotonically decreasing function with unbounded support on natural numbers. With this, we do not need to fix a maximum number of neurons, which is typical of orthogonal approaches like supernetworks [61]. As a by-product, we can achieve a straightforward trade-off between parametrization and performance by deleting the last rows/columns of weight matrices, i.e., removing the "least important" neurons from the computation. Finally, we break symmetries in the parametrization of neural networks: it is not possible anymore to obtain an equivalent neural network behavior by permuting the weight matrices, which reduces the "jostling" effect where symmetric parametrizations compete when training starts [1].

We test our method on MLPs for tabular data, a Convolutional Neural Network (CNN) [33] for images, a Transformer architecture [58] for text, a Recurrent Neural Network (RNN) for sequences, for and a Deep Graph Network (DGN) [38, 52] for graphs, to showcase a broad scope of applicability. Empirical results show, as may be intuitively expected, that the width adapts to the task's difficulty with the same performance of a fixed-width baseline. We also investigate compressing of the network at training time while preserving accuracy, as well as a post-hoc truncation inducing a controlled trade-off at zero additional cost. Ablations suggest that the learned width is not influenced by the starting width (under bounded activations) nor by the batch size, advocating for a potentially large reduction of the hyper-parameter configuration space.

2 Related Work

Constructive methods dynamically learn the width of neural networks and are related in spirit to this work. The cascade correlation algorithm [17] alternates standard training with the creation of a new hidden unit minimizing the neural network's residual error. Similarly, the firefly network descent [64] grows the width and depth of a network every N training epochs via gradient descent on a dedicated loss. Yoon et al. [66] propose an ad-hoc algorithm for lifelong learning that grows the network by splitting and duplicating units to learn new tasks. Wu et al. [63] alternate training the network and then splitting existing neurons into offspring with equal weights. These works mostly focus on growing the neural network; Mitchell et al. [40] propose natural gradient-based heuristics to grow/shrink layers and hidden units of MLPs and CNNs. The main difference from our work is that we grow and shrink the network by simply computing the gradient of the loss, without relying on human-defined heuristics. The unbounded depth network of Nazaret and Blei [42], from which we draw inspiration, learns the number of layers of neural networks. Compared to that work, we focus our attention to the number of neurons, modifying the internals of the architecture rather than instantiating a multi-output one. Finally, we mention Bayesian nonparametrics approaches [43] that learn a potentially infinite number of clusters in an unsupervised fashion, as well as dynamic neural networks [24] that condition the architecture on input properties.

Orthogonal Methods Neural Architecture Search (NAS) is an automated process that designs neural networks for a given task [15, 61] and has been applied to different contexts [71, 35, 53]. Typically, neural network elements are added, removed, or modified based on validation performance [14, 60, 61], by means of reinforcement learning [70], evolutionary algorithms [47], and gradient-based approaches [35]. Typical NAS methods require enormous computational resources, sometimes reaching thousands of GPU days [70], due to the retraining of each new configuration. While recent advances on one-shot NAS models [8, 45, 2, 4, 55, 54] have drastically reduced the computational costs, they mostly focus on CNNs, assume a bounded search space, and do not learn the width. As

such, NAS methods are complementary to our approach. Bi-level optimization algorithms have also been used for hyper-parameter tuning [19], where hyper-parameters are the variables of the outer objective and the model parameters those of the inner objective. The solution sets of the inner problem are usually not available in closed form, which has been partly addressed by repeated application of (stochastic) gradient descent [11, 36, 18]. These methods are restricted to continuous hyper-parameters' optimization, and cannot therefore be applied to width optimization.
Finally, pruning [6] and distillation [29] are two methods that reduce the size of neural networks by trading-off performances; the former deletes neural connections [39] or entire neurons [57, 12], the latter trains a smaller network (student) to mimic a larger one (teacher) [22]. In particular, dynamic pruning techniques can compress the network at training time [23], by applying hard or soft masks [28]; for a comprehensive survey on pruning, please refer to [27]. Compared to most pruning approaches, our work can delete connections *and* reduce the model's memory, but also grow it indefinitely; compared to distillation, we do not necessarily need a new training to compress the network. These techniques, however, can be easily combined with our approach.

3 Adaptive Width Learning

We now introduce Adaptive Width Neural Networks (AWNN), a probabilistic framework that maximizes a simple variational objective via backpropagation over a neural network's parameters.

We are given a dataset of N *i.i.d.* samples, with input $x \in \mathbb{R}^F, F \in \mathbb{N}^+$ and target y whose domain depends on whether the task is regression or classification. For samples $X \in \mathbb{R}^{N \times F}$ and targets Y, the learning objective is to maximize the log-likelihood

$$\log p(Y|X) = \log \prod_{i=1}^{N} p(y_i|x_i) = \sum_{i=1}^{N} \log p(y_i|x_i) \tag{1}$$

with respect to the learnable parameters of $p(y|x)$.

We define $p(y|x)$ according to the graphical model of Figure 1 (left), to learn a neural network with **unbounded width** for each hidden layer ℓ. To do so, we assume the existence of an **infinite** sequence of *i.i.d.* latent variables $\boldsymbol{\theta}_\ell = \{\theta_{\ell n}\}_{n=1}^{\infty}$, where $\theta_{\ell n}$ is a multivariate variable over the learnable weights of neuron n at layer ℓ. However, since working with an infinite-width layer is not possible in practice, we also introduce a latent variable λ_ℓ that samples how many neurons to use for layer ℓ. That is, it *truncates an infinite width to a finite value* so that we can feasibly perform inference with the neural network. For a neural network of L layers, we define $\boldsymbol{\theta} = \{\boldsymbol{\theta}_\ell\}_{\ell=1}^{L}$ and $\boldsymbol{\lambda} = \{\lambda_\ell\}_{\ell=1}^{L}$, assuming independence across layers. Therefore, by marginalization one can write $p(Y|X) = \int p(Y, \boldsymbol{\lambda}, \boldsymbol{\theta}|X) d\boldsymbol{\lambda} d\boldsymbol{\theta}$.

We then decompose the joint distribution using the independence assumptions of the graphical model:

$$p(Y, \boldsymbol{\lambda}, \boldsymbol{\theta}|X) = \prod_{i=1}^{N} p(y_i, \boldsymbol{\lambda}, \boldsymbol{\theta}|x_i) \qquad p(y_i, \boldsymbol{\lambda}, \boldsymbol{\theta}|x_i) = p(y_i|\boldsymbol{\lambda}, \boldsymbol{\theta}, x_i) p(\boldsymbol{\lambda}) p(\boldsymbol{\theta}) \tag{2}$$

$$p(\boldsymbol{\lambda}) = \prod_{\ell=1}^{L} p(\lambda_\ell) = \prod_{\ell=1}^{L} \mathcal{N}(\lambda_\ell; \mu_\ell^\lambda, \sigma_\ell^\lambda) \qquad p(\boldsymbol{\theta}) = \prod_{\ell=1}^{L} \prod_{n=1}^{\infty} p(\theta_{\ell n}) = \prod_{\ell=1}^{L} \prod_{n=1}^{\infty} \mathcal{N}(\theta_{\ell n}; \mathbf{0}, \text{diag}(\sigma_\ell^\theta)) \tag{3}$$

$$p(y_i|\boldsymbol{\lambda}, \boldsymbol{\theta}, x_i) = \text{Neural Network as will be introduced in Section 3.1} \tag{4}$$

where $\sigma_\ell^\theta, \mu_\ell^\lambda, \sigma_\ell^\lambda$ are hyper-parameters. The neural network is parametrized by realizations $\boldsymbol{\lambda} \sim p(\boldsymbol{\lambda}), \boldsymbol{\theta} \sim \boldsymbol{\theta}$ – it relies on a finite number of neurons as detailed later and in Section 3.1 – and it outputs either class probabilities (classification) or the mean of a Gaussian distribution (regression) to parametrize $p(y_i|\boldsymbol{\lambda}, \boldsymbol{\theta}, x_i)$ depending on the task. Maximizing Equation (1), however, requires computing the evidence $\int p(Y, \boldsymbol{\lambda}, \boldsymbol{\theta}|X) d\boldsymbol{\lambda} d\boldsymbol{\theta}$, which is intractable. Therefore, we turn to mean-field variational inference [30, 7] to maximize an expected lower bound (ELBO) instead. This requires to define a variational distribution over the latent variables $q(\boldsymbol{\lambda}, \boldsymbol{\theta})$ and re-phrase the objective as:

$$\log p(Y|X) \geq \mathbb{E}_{q(\boldsymbol{\lambda}, \boldsymbol{\theta})} \left[\log \frac{p(Y, \boldsymbol{\lambda}, \boldsymbol{\theta}|X)}{q(\boldsymbol{\lambda}, \boldsymbol{\theta})} \right], \tag{5}$$

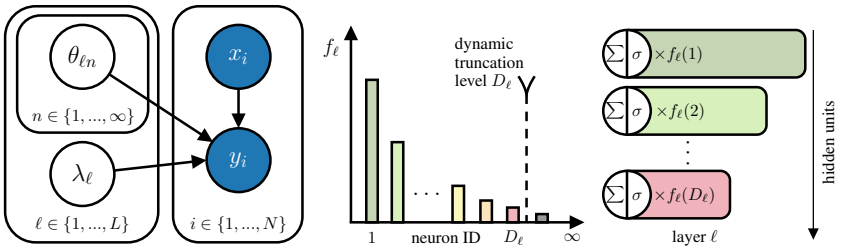

Figure 1: *(Left)* The graphical model of AWNN, with dark observable random variables and white latent ones. *(Middle)* The distribution f_ℓ over hidden units' importance at layer ℓ is parametrized by λ_ℓ. The width of layer ℓ is chosen as the quantile function of the distribution f_ℓ evaluated at k and denoted by D_ℓ. *(Right)* The hidden units' activations at layer ℓ are rescaled by their importance.

where $q(\boldsymbol{\lambda}, \boldsymbol{\theta})$ is parametrized by learnable *variational* parameters. Before continuing, we define the **truncated width** D_ℓ, that is the finite number of neurons at layer ℓ, as the quantile function evaluated at k, with k a hyper-parameter, of a distribution[1] f_ℓ with infinite support over $\mathbb{N}^+$ and parametrized by λ_ℓ; Appendix A and B provide formal requirements about f_ℓ. In other words, we find the integer such that the cumulative mass function of f_ℓ takes value k, and that integer D_ℓ is the truncated width of layer ℓ. We implement f_ℓ as a **discretized exponential distribution** adhering to Def. A.2, following the discretization strategy of Roy [49]: For every natural x, the discretized distribution relies on the exponential's cumulative distribution function:

$$f_\ell(x; \lambda_\ell) = (1 - e^{\lambda_\ell (x+1)}) - (1 - e^{\lambda_\ell (x)}). \tag{6}$$

We choose the exponential because it is a **monotonically decreasing** function and allows us to impose an ordering of importance among neurons, as detailed in Section 3.1.
Then, we can factorize the variational distribution $q(\boldsymbol{\lambda}, \boldsymbol{\theta})$ into:

$$q(\boldsymbol{\lambda}, \boldsymbol{\theta}) = q(\boldsymbol{\lambda}) q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \qquad q(\boldsymbol{\lambda}) = \prod_{\ell=1}^{L} q(\lambda_\ell) = \prod_{\ell=1}^{L} \mathcal{N}(\lambda_\ell; \nu_\ell, 1) \tag{7}$$

$$q(\boldsymbol{\theta}|\boldsymbol{\lambda}) = \prod_{\ell=1}^{L} \prod_{n=1}^{D_\ell} q(\theta_{\ell n}) \prod_{D_\ell+1}^{\infty} p(\theta_{\ell n}) \qquad q(\theta_{\ell n}) = \mathcal{N}(\theta_{\ell n}; \text{diag}(\rho_{\ell n}), \mathbf{I}). \tag{8}$$

Here, $\nu_\ell, \rho_{\ell n}$ are learnable variational parameters and, as before, we define $\boldsymbol{\rho}_\ell = \{\rho_{\ell n}\}_{n=1}^{D_\ell}$, $\boldsymbol{\rho} = \{\boldsymbol{\rho}_\ell\}_{\ell=1}^{L}$ and $\boldsymbol{\nu} = \{\nu_\ell\}_{\ell=1}^{L}$. Note that the set of variational parameters is **finite** as it depends on D_ℓ.

By expanding Equation 5 using the above definitions and approximating the expectations at the first order, *i.e.*, $\mathbb{E}_{q(\boldsymbol{\lambda})}[f(\boldsymbol{\lambda})]{=}f(\boldsymbol{\nu})$ and $\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}[f(\boldsymbol{\theta})] = f(\boldsymbol{\rho})$ as in Nazaret and Blei [42], we obtain the final form of the ELBO[2] (the full derivation is in Appendix C):

$$\sum_{\ell}^{L} \log \frac{p(\nu_\ell; \mu_\ell^\lambda, \sigma_\ell^\lambda)}{q(\nu_\ell; \nu_\ell)} + \sum_{\ell}^{L} \sum_{n=1}^{D_\ell} \log \frac{p(\rho_{\ell n}; \sigma_\ell^\theta)}{q(\rho_{\ell n}; \rho_{\ell n})} + \sum_{i=1}^{N} \log p(y_i | \boldsymbol{\lambda}{=}\boldsymbol{\nu}, \boldsymbol{\theta}{=}\boldsymbol{\rho}, x_i), \tag{9}$$

where distributions' parameters are made explicit to distinguish them. The first two terms in the ELBO regularize the width of the layers and the magnitude of the parameters when priors are informative, whereas the third term accounts for the predictive performance.
In practice, the finite variational parameters $\boldsymbol{\nu}, \boldsymbol{\rho}$ are those used to parametrize the neural network rather than sampling $\boldsymbol{\lambda}, \boldsymbol{\theta}$, which enables easy optimization via backpropagation. Maximizing Equation (9) will update each variational parameter ν_ℓ, which in turn will change the value of D_ℓ **during training**. If D_ℓ increases we initialize new neurons and draw their weights from a standard normal distribution, otherwise we discard the weights of the extra neurons. When implementing mini-batch training, the predictive loss needs to be rescaled by N/M, where M is the mini-batch size. From a Bayesian perspective, this is necessary as regularizers should weigh less if we have more data.

[1] General functions are allowed if a threshold can be computed.
[2] The higher k the better the ELBO approximation.

Compared to a fixed-width network with weight decay, we need to choose the priors' values of $\mu_\ell^\lambda, \sigma_\ell^\lambda$, as well as initialize the learnable ν_ℓ. The latter can be initially set to same value across layers since they can freely adapt later, or it can be sampled from the prior $p(\lambda_\ell)$. Therefore, we have two more hyper-parameters compared to the fixed-width network, but we make some considerations: **i)** it is always possible to use an uninformative prior over λ_ℓ, removing the extra hyper-parameters letting the model freely adapt the width of each layer (as is typical of frequentist approaches); **ii)** the choice of higher level of hyper-parameters is known to be less stringent than that of hyper-parameters themselves [21, 5], so we do not need to explore many values of μ_ℓ^λ and σ_ℓ^λ; **iii)** our experiments suggest that AWNN can converge to similar widths regardless of the starting point ν_ℓ, so that we may just need to perform model selection over one/two sensible initial values; **iv)** the more data, the less the priors will matter.

3.1 Imposing a Soft Ordering on Neurons' Importance

Now that the learning objective has been formalized, the missing ingredient is the definition of the neural network $p(y_i|\boldsymbol{\lambda}{=}\boldsymbol{\nu}, \boldsymbol{\theta}{=}\boldsymbol{\rho}, x_i)$ of Equation 4 as a modified MLP. Compared to a standard MLP, we use the variational parameters $\boldsymbol{\nu}$ that affect the truncation width at each hidden layer, whereas $\boldsymbol{\rho}$ are the weights. We choose a monotonically decreasing function f_ℓ, thus when a new neuron is added its relative importance is low and will not drastically impact the network output and hidden activations. In other words, we impose a soft ordering of importance among neurons.
We simply modify the classical activation h_j^ℓ of a hidden neuron j at layer ℓ as

$$h_j^\ell = \sigma\left(\sum_{k=1}^{D_{\ell-1}} w_{jk}^\ell h_k^{\ell-1}\right) f_\ell(j;\nu_\ell), \tag{10}$$

where $D_{\ell-1}$ is the truncated width of the previous layer, σ is a non-linear activation function and $w_{jk}^\ell \in \rho_{\ell j}$. That is, we rescale the activation of each neuron k by its "importance" $f_\ell(j;\nu_\ell)$. Note that the bias parameter is part of the weight vector as usual.

It is easy to see that, in theory, the optimization algorithm could rescale the weights of the next layer by a factor $1/f_\ell(j;\nu_\ell)$ to compensate for the term $f_\ell(j;\nu_\ell)$. This could lead to a degenerate situation where the activations of the first neurons are small relative to the others, thus breaking the soft-ordering and potentially wasting neurons. There are two strategies to address this undesirable effect. The first is to regularize the magnitude of the weights thanks to the prior $p(\theta_{\ell+1n})$, so that it may be difficult to compensate for the least important neurons that have a high $1/f_\ell(j)$. The second and less obvious strategy is to prevent the units' activations of the current layer to compensate for high values by bounding their range, e.g., using a ReLU6 or tanh activation [51]. We apply both strategies to our experiments, although we noted that they do not seem strictly necessary in practice.

3.2 Rescaled Weight Initialization for Deep AWNN

Rescaling the activations of hidden units using Equation 10 causes activations of deeper layers to quickly decay to zero for an AWNN MLP with ReLU nonlinearity initialized using the well known Kaiming scheme [25]. This affects convergence since gradients get close to zero and it becomes slow to train deep AWNN MLPs. We therefore derive a rescaled Kaiming weight that guarantees that the variance of activation across layers is constant at initialization.

Theorem 3.1. *Consider an MLP with activations as in Equation 10 and ReLU nonlinearity. At initialization, given* $\alpha_j^\ell = \sigma\left(\sum_{k=1}^{D_{\ell-1}} w_{jk}^\ell h_k^{\ell-1}\right), \mathrm{Var}[w_{j*}^\ell] = \frac{2}{\sum_{j=1}^{D_{\ell-1}} f_\ell^2(j)} \Rightarrow \mathrm{Var}[\alpha_j^\ell] \approx \mathrm{Var}[\alpha_j^{\ell-1}]$.

Proof. See Appendix E for an extended proof. □

The extended proof shows there is a connection between the variance of activations and the variance of gradients. In particular, the sufficient conditions over $\mathrm{Var}[w_{j*}^\ell]$ are identical if one initializes ν_ℓ in the same way for all layers. Therefore, if we initialize weights from a Gaussian distribution with standard deviation $\frac{\sqrt{2}}{\sqrt{\sum_{j=1}^{D_{\ell-1}} f_\ell^2(j)}}$, we guarantee that at initialization the variance of the deep network's gradients will be constant. The effect of the new initialization (dubbed "Kaiming+") can be seen in Figure 7,

where the distribution of activation values at initialization does not collapse in subsequent layers. This change drastically impacts overall convergence on the SpiralHard synthetic dataset (described in Section 4), where it appears it would be otherwise hard to converge using a standard Kaiming initialization. Algorithm 1 summarizes the main changes to the training procedure, namely the new initialization and the update of the model's truncated width at each training step.

Algorithm 1 AWNN Training Procedure

Input: Dataset $\mathcal{D}$, initialized AWNN model $\mathcal{M}$ (Section 3.2)
Output: Trained AWNN Model $\mathcal{M}$
for each training epoch **do**
  for batch in $\mathcal{D}$ **do**
    update_width($\mathcal{M}$)
    $\hat{y} \leftarrow \mathcal{M}$(batch)
    loss $\leftarrow$ ELBO($\mathcal{M}$, batch, $\hat{y}$) // Eq. 9
    $\mathcal{M} \leftarrow$ backpropagation($\mathcal{M}$, loss)
function update_width($\mathcal{M}$):
  for layer ℓ in $\mathcal{M}$.hidden_layers **do**
    $D_\ell \leftarrow$ quantile function of $f_\ell(\cdot; \nu_\ell)$ evaluated at k
    Use D_ℓ to update $\boldsymbol{\rho}_\ell, \boldsymbol{\rho}_{\ell+1}$ // add/remove neurons

3.3 Future Directions and Limitations

MLPs are ubiquitous in modern deep architectures. They are used at the very end of CNNs, in each Transformer layer, and they process messages coming from neighbors in DGNs. Our experiments focus on MLPs to showcase AWNN's broad applicability, but there are many other scenarios where one can apply AWNN's principles. For instance, one could impose a soft ordering of importance on CNNs' filters at each layer, therefore learning the number of filters during training.
From a more theoretical perspective, we believe one could draw connections between our technique and the Information Bottleneck principle [56], which seeks maximally representative (i.e., performance) and compact representations (e.g., width). Finally, the analysis of different functions f to soft-order neurons, for instance a power-law distribution or a sigmoid-like function, may have a different impact on convergence and performance and could be empirically investigated.

AWNN needs to change the network at *training* time with an associated overhead. Our current implementation has only a 3-4$\times$ overhead *on the smallest datasets*, the worst case scenario for our approach, where the inference cost is negligible compared to the network update. For more realistic tasks, we believe our method is advantageous compared to the cost of a grid-search or alternating training/updating approaches. In addition, further optimizations are possible: i) by updating the width every M mini-batches rather than at every step; ii) by allocating a larger network and dynamically grow it only if needed; iii) by creating ad-hoc support of AWNN on deep learning libraries.

4 Experiments and Setup

The purpose of the empirical analysis is not to claim AWNN is generally better than the fixed-width baseline. Rather, we demonstrate how AWNN overcomes the problem of fixing the number of neurons by learning it end-to-end, thus reducing the amount of hyper-parameter configurations to test. As such, due to the nature of this work and similarly to Mitchell et al. [40], we use the remaining space to thoroughly study the behavior of AWNN, so that it becomes clear how to use it in practice. We first quantitatively verify that AWNN does not harm the performance compared to baseline models and compare the chosen width by means of grid-search model selection with the learned width of AWNN. Second, we check that AWNN chooses a larger width for harder tasks, which can be seen as increasing the hypotheses space until the neural network finds a good path to convergence. Third, we verify that convergence speed is not significantly altered by AWNN, so that the main limitation lies in the extra overhead for adapting the network at each training step. As a sanity check, we study conditions under which AWNN's learned width does not seem to depend on starting hyper-parameters, so that their choice does not matter much. Finally, we analyze other practical advantages of training a neural network under the AWNN framework: the ability to compress information during training or post training, and the resulting trade-offs. Further analyses are in the Appendix.

We compare baselines that undergo proper hyper-parameter tuning (called "Fixed") against its AWNN version, where we replace any fixed MLP with an adaptive one. First, we train an MLP on 3 synthetic tabular tasks of increasing binary classification difficulty, namely a double moon, a spiral, and a double spiral that we call SpiralHard. A stratified hold-out split of 70% training/10% validation/20% test for risk assessment is chosen at random for these datasets. Similarly, we consider a ResNet-20

[26] trained on 3 image classification tasks, namely MNIST [32], CIFAR10, and CIFAR100 [31], where data splits and preprocessing are taken from the original paper and AWNN is applied to the downstream classifier. In the sequential domain, we implemented a basic adaptive Recurrent Neural Network (RNN) and evaluated on the PMNIST dataset [67] with the same data split as MNIST. In the graph domain, we train a Graph Isomorphism Network [65] on the NCI1 and REDDIT-B classification tasks, where topological information matters, using the same split and evaluation setup of Errica et al. [16]. Here, the first 1 hidden layer MLP as well as the one used in each graph convolutional layer are replaced by adaptive AWNN versions. On all these tasks, the metric of interest is the accuracy. Finally, for the textual domain we train a Transformer architecture [58] on the Multi30k English-German translation task [13], using a pretrained GPT-2 Tokenizer, and we evaluate the cross-entropy loss over the translated words. On tabular, image, and text-based tasks, an internal validation set (10%) for model selection is extracted from the union of outer training and validation sets, and the best configuration chosen according to the internal validation set is retrained 10 times on the outer train/validation/test splits, averaging test performances after an early stopping strategy on the validation set. Due to space reasons, we report datasets statistics and the hyper-parameter tried for the fixed and AWNN versions in Appendix F and G, respectively[3]. We ran the experiments on a server with 64 cores, 1.5TB of RAM, and 4 NVIDIA A40 with 48GB of memory.

5 Results

We begin by discussing the quantitative results of our experiments: Table 1 reports means and standard deviations across the 10 final training runs. In terms of performance, we observe that AWNN is more stable or accurate than a fixed MLP on DoubleMoon, Spiral, SpiralHard, and PMNIST; all other things being equal, it seems that using more neurons and their soft ordering are the main contributing factors to these improvements. On the image datasets, performances of AWNN are comparable to those of the fixed baseline but for CIFAR100, due to an unlucky run that did not converge. There, AWNN learns a smaller total width compared to grid search.

Table 1: Performances and total width of MLP layers for the fixed and AWNN versions of the various models used. The exact width chosen by model selection on the graph datasets is unknown since we report published results. "Linear" means the chosen downstream classifier is a linear model.

	Fixed		AWNN		Width (Fixed)	Width (AWNN)	
	Mean	(Std)	Mean	(Std)		Mean	(Std)
DoubleMoon	100.0	(0.0)	100.0	(0.0)	8	8.1	(2.8)
Spiral	99.5	(0.5)	99.8	(0.1)	16	65.9	(8.7)
SpiralHard	98.0	(2.0)	100.0	(0.0)	32	227.4	(32.4)
PMNIST	91.1	(0.4)	95.7	(0.2)	24	806.3	(44.5)
MNIST	99.6	(0.1)	99.7	(0.0)	Linear	19.4	(4.8)
CIFAR10	91.4	(0.2)	91.4	(0.2)	Linear	80.1	(12.4)
CIFAR100	66.5	(0.4)	63.1	(4.0)	256	161.9	(57.8)
NCI1	80.0	(1.4)	80.0	(1.1)	(96-320)	731.3	(128.2)
REDDIT-B	87.0	(4.4)	90.2	(1.3)	(96-320)	793.6	(574.0)
Multi30k (↓)	1.43	(0.4)	1.51	(0.2)	24576	123.2	(187.9)

Results on graph datasets are interesting in two respects: First, the performance on REDDIT-B is significantly improved by AWNN both in terms of average performance and stability of results; second, and akin to PMNIST, the total learned width is significantly higher than those tried in Xu et al. [65], Errica et al. [16], meaning a biased choice of possible widths had a profound influence on risk estimation for DGN models (i.e., GIN). This result makes it evident that it is important to let the network decide how many neurons are necessary to solve the task. Appendix H shows what happens when we retrain Fixed baselines using the total width as the width of each layer. Finally, the results on the Multi30k show that the AWNN Transformer learns to use 200x parameters less than the fixed Transformer for the feed-forward networks, achieving a statistically comparable test loss. This result is appealing when read through the lenses of modern deep learning, as the power required by some neural networks such as Large Language Models [9] is so high that cannot be afforded by most institutions, and it demands future investigations.

Adaptation to Task Difficulty and Convergence Intuitively, one would expect that AWNN learned larger widths for more difficult tasks. This is indeed what happens on the tabular datasets (and image datasets, see Appendix I) where some tasks are clearly harder than others. Figure 2 (left) shows that, given the same starting width per layer, the learned number of neurons grows according

[3]Code to reproduce results is in the supplementary material.

to the task's difficulty. It is also interesting to observe that the total width for a multi-layer MLP on SpiralHard is significantly lower than that achieved by a single-layer MLP, which is consistent with the circuit complexity theory arguments put forward in Bengio et al. [3], Mhaskar et al. [37]. It also appears that convergence is not affected by the introduction of AWNN, as investigated in Figure 2 (right), which was not obvious considering the parametrization constraints encouraged by the rescaling of neurons' activations.

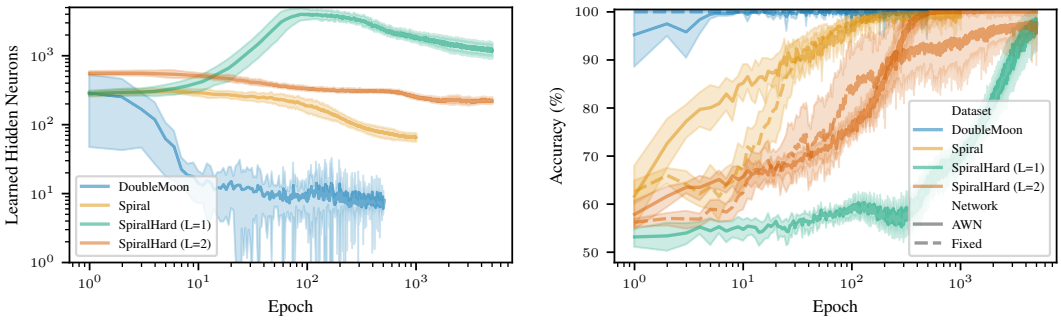

Figure 2: (Left) The learned width adapts to the increasing difficulty of the task, from the DoubleMoon to SpiralHard. (Right) AWNN reaches perfect test accuracy with a comparable amount of epochs on DoubleMoon and Spiral, while it converges faster on SpiralHard.

Training Stability Analysis To support our argument that AWNN can reduce the time spent performing hyper-parameter selection, we check whether AWNN learns a consistent amount of neurons across different training runs and hyper-parameter choices. Figure 3 reports the impact of the batch size and starting width averaged across the different configurations tried during model selection. Smaller batch sizes cause more instability, but in the long run we observe convergence to a similar width. Convergence with respect to different rates holds, instead, for the bounded ReLU6 activation; Appendix J shows that unbounded activations may cause the network to converge more slowly to the same width, which is in accord with the considerations about counterbalancing the rescaling effect of Section 3.1. Therefore, whenever possible, we recommend using bounded activations.

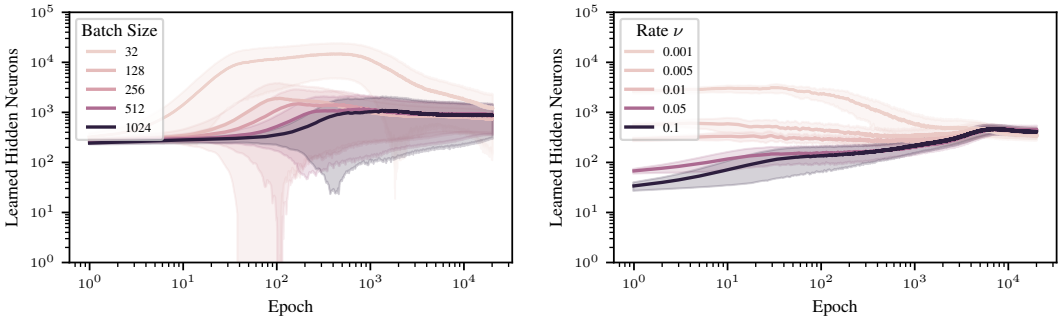

Figure 3: Training converges to similar widths on SpiralHard for different batch sizes (left) and starting rates ν, but the latter seems to require a bounded nonlinearity such as ReLU6 to converge in a reasonable amount of epochs (right).

Online Network Compression via Regularization So far, we have used an uninformative prior $p(\lambda)$ over the neural networks' width. We demonstrate the effect of an informative prior by performing a width-annealing experiment on the SpiralHard dataset. We set an uninformative $p(\boldsymbol{\theta})$ and ReLU6 nonlinearity. At epoch 1000, we introduce $p(\lambda_\ell) = \mathcal{N}(\lambda_\ell; 0.05, 1)$, and gradually anneal the standard deviation up to 0.1 at epoch 2500. Figure 4 shows that the width of the network reduces from approximately 800 neurons to 300 without any test performance degradation. We hypothesize that the least important neurons carry negligible information, therefore they can be safely removed without drastic changes in the output of the model. *This technique might be particularly useful to compress large models with billions of parameters.*

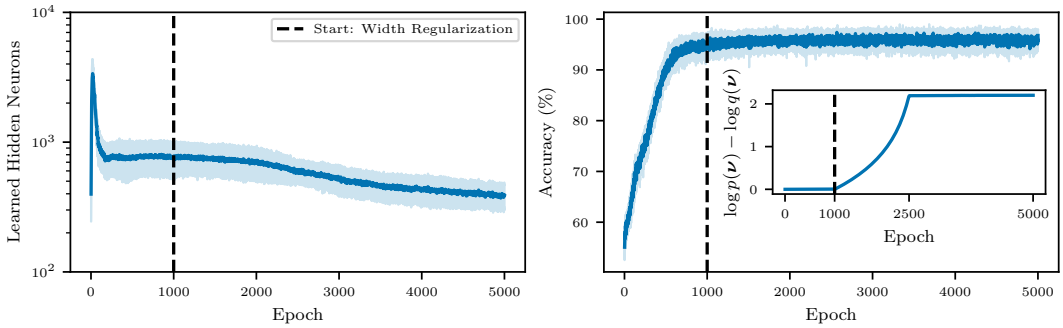

Figure 4: It is possible to regularize the width at training time by increasing the magnitude of the loss term $\log \frac{p(\boldsymbol{\nu})}{q(\boldsymbol{\nu})}$. The total width is reduced by more than 50% (left) while preserving accuracy (right). The inset plot refers to the loss term that AWNN tries to maximize.

Post-hoc Truncation Achieves a Trade-off between Performance and Compute Resources To further investigate the consequences of imposing a soft ordering among neurons, we show that it is possible to perform straightforward post-training truncation while still controlling the trade-off between performance and network size. Figure 5 shows an example for an MLP on the Spiral dataset, where the range of activation values (Equation 10) computed for all samples follows an exponential curve (right). Intuitively, removing the last neurons may have a negligible performance impact at the beginning and a drastic one as few neurons remain. This is what happens, where we are able to cut an MLP with hidden width 83 by 30% without loss of accuracy, after which a smooth degradation happens. If one accepts such a trade-off, this technique may be used to "distill" a trained neural network at virtually zero additional cost while reducing the memory requirements. Note that truncation heuristics that are either random or based on the magnitude of neurons' activations (i.e., excluding the rescaling term) do not perform as well.

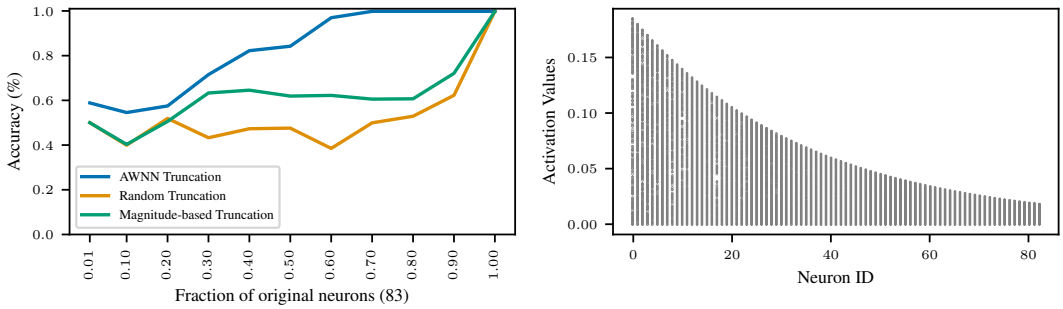

Figure 5: (Left) Thanks to the soft ordering imposed on the neurons, one can also truncate the neural network *after training* by simply removing the last neurons. (Right) The distribution of neurons' activations for all Spiral test samples follows an exponential-like curve.

6 Conclusions

We introduced a new methodology to learn an unbounded width of neural network layers within a single training, by imposing a soft ordering of importance among neurons. Our approach requires very few changes to the architecture, adapts the width to the task's difficulty, and does not impact negatively convergence. We showed stability of convergence to similar widths under bounded activations for different hyper-parameters configurations, advocating for a practical reduction of the width's search space. A by-product of neurons' ordering is the ability to easily compress the network during or after training, which is relevant in the context of foundational models trained on large data, which are believed to require billions of parameters. Finally, we have tested AWNN on different models and data domains to prove its broad scope of applicability: a Transformer architecture achieved a similar loss with 200x less parameters.