GRADMAX: GROWING NEURAL NETWORKS USING GRADIENT INFORMATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The architecture and the parameters of neural networks are often optimized independently, which requires costly retraining of the parameters whenever the architecture is modified. In this work we instead focus on growing the architecture without requiring costly retraining. We present a method that adds new neurons during training without impacting what is already learned, while improving the training dynamics. We do this by maximizing the gradients of the new neurons and find an approximation to the optimal initialization by means of the singular value decomposition (SVD). We call this technique Gradient Maximizing Growth (GradMax) and demonstrate its effectiveness in variety of vision tasks and architectures.

1 INTRODUCTION

The architecture of deep learning models influences a model's inductive biases and has been shown to have a crucial effect on both the training speed and generalization (dÁscoli et al., 2019; Neyshabur, 2020). Searching for the best architecture for a given task is an active research area with diverse approaches, including neural architecture search (NAS) (Elsken et al., 2019), pruning (Liu et al., 2018), and evolutionary algorithms (Stanley & Miikkulainen, 2002). Most of these approaches are costly, as they require large search spaces or large architectures to start with. In this work we consider an alternative approach: Can we start with a small network and learn an efficient architecture without ever needing a large network or exceeding the size of our final architecture?

The idea of incrementally increasing the size of a model has been used in many settings such as boosting (Friedman, 2001), continual learning (Rusu et al., 2016), architecture search (Elsken et al., 2017), optimization (Fukumizu & Amari, 2000), and reinforcement learning (Berner et al., 2019). Despite the wide range of applications of growing neural networks, the initialization of newly grown neurons is rarely studied. Existing work on growing neural networks either adds new neurons randomly (Chen et al., 2016; Berner et al., 2019) or chooses them with the aim of decreasing the training loss (Bengio et al., 2006; Liu et al., 2019; Wu et al., 2020a). In this work we take a different approach. Instead of improving the training objective immediately through growing, we focus on improving the subsequent training dynamics. As we will show, this has a longer-lasting effect than the greedy approach of improving loss during growing.

Our main contribution is a new growing method that maximizes the gradient norm of newly added neurons, which we call GradMax. We show that this gradient maximization problem can be solved directly using the singular value decomposition (SVD) and initializing the new neurons using the top k singular vectors. We show that the increased gradient norm persists during future steps, which in return yields faster training.

2 GROWING NEURAL NETWORKS

Neural network learning is often formalized as the following optimization problem:

$$\arg\min_{f \in S} \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim D} \left[L(f(\boldsymbol{x}), \boldsymbol{y}) \right] , \tag{1}$$

where S is a set of neural networks, L is a loss function and D is a data set consisting of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$). Most often the set S is constrained to a single architecture (e.g., ResNet-18(Zagoruyko & Komodakis, 2016)) and only the parameters of the model are learned. However, hand-crafted

architectures are often suboptimal and various approaches aim to optimize the architecture itself together with its parameters.

Methods such as random search (Bergstra & Bengio, 2012), NAS, and pruning search a larger space that contains a variety of architectures. Growing neural networks has the same goal: It learns the architecture and weights jointly. It aims to achieve this by incrementally increasing the size of the model. Since this approach never requires training a large model from scratch it needs less memory and compute than pruning methods (Yuan et al., 2021). Reducing the cost of finding efficient architectures is important given the ever-growing size of architectures and the accompanying increase in environmental footprint (Thompson et al., 2021).

Growing neural networks: When, where and how? Algorithms for growing neural networks often start training with a small *seed architecture*. Then over the course of the training new neurons are added to the seed architecture, either increasing the width of the existing layers or creating new layers. Algorithms for growing neural networks should address the following questions:

1. **When** to add new neurons? For instance, some methods require the training loss to plateau before growing, whereas others grow using a predefined schedule.
2. **Where** to add new capacity? We can add new neurons to the existing layers or create new layers among the existing ones.
3. **How** to initialize the new capacity?

In this work we mainly focus on the question of **how** and introduce a new initialization method for the new neurons. In order to make our comparison with other initialization methods fair we keep the growing schedule (**where** and **when**) fixed. Our approach can also be used to guide when and where to grow new neurons.

How to grow new neurons? Suppose that during training we would like to grow new neurons at layer ℓ as depicted in fig. 1. After the growth, new neurons are appended to the existing weight matrices as follows:

$$\boldsymbol{W}_{\ell}^{+} = \begin{bmatrix} \boldsymbol{W}_{\ell} \\ \boldsymbol{W}_{\ell}^{\text{new}} \end{bmatrix} \tag{2}$$

$$\boldsymbol{W}_{\ell+1}^{+} = \begin{bmatrix} \boldsymbol{W}_{\ell+1} & \boldsymbol{W}_{\ell+1}^{\text{new}} \end{bmatrix} . \tag{3}$$

A desirable property of growing algorithms is to preserve the information that the network has learned. Therefore the initialization of new neurons should ensure our neural network has the same outputs before and after growth.

We describe two main approaches to growing: splitting and adding. **Splitting** (Chen et al., 2016; Liu et al., 2019) duplicates existing neurons and adjusts outgoing weights so that the output in the next layer is unchanged. However, splitting has some limitations: (1) It creates redundant neurons and small changes required to break symmetry cause changes in the output distribution; (2) It can't be used for growing new layers as it requires existing neurons to begin with.

Another line of work focuses on **adding** new neurons while ensuring that the output does not change by setting either $\boldsymbol{W}_{\ell}^{\text{new}}$ or $\boldsymbol{W}_{\ell+1}^{\text{new}}$ to zero. This approach doesn't have the limitations mentioned above and often results in better performance (Wu et al., 2020a). Below we summarize two methods that use this approach:

Random Berner et al. (2019) initialize new neurons randomly when growing layers from existing checkpoints.

Firefly Wu et al. (2020a) combine splitting and adding random neurons. Random neurons are trained for a few steps to reduce loss directly and the most promising neurons are selected for growing.

Previous work focused either on solely keeping the output of the neural network the same (Random) or on immediately reducing the training objective (Firefly). In this work we take a different approach and focus on improving the training dynamics. We initialize the weights so that the network's output remains unchanged while trying to improve the training dynamics by maximizing the gradient norm. The hypothesis is that this accelerates training and leads to a better model in the long run.

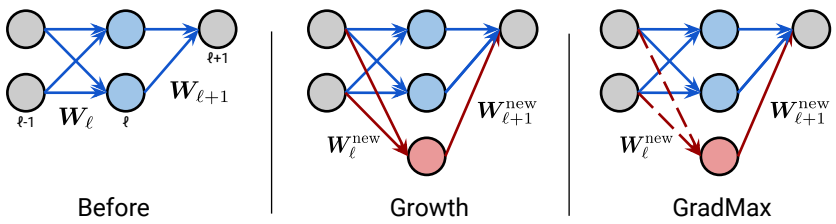


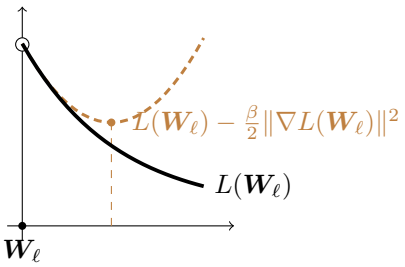
Figure 1: Schematic view of the GradMax algorithm. Growing new neurons requires initializing incoming ($\boldsymbol{W}_\ell^{\text{new}}$) and outgoing $\boldsymbol{W}_{\ell+1}^{\text{new}}$ weights for the new neuron. GradMax sets incoming weights to zero (dashed lines) in order to keep the output unchanged, and initializes outgoing weights using SVD (equation 11). This maximizes the gradients on the incoming weights with the aim of accelerating training.

3 GradMax

This section describes our main contribution: GradMax, a method that maximizes the gradient norm of the new weights. We constrain the norm of the new weights to avoid the trivial solution where the weight norm tends to infinity. We also require that the network output before and after the growth remains unchanged.

$$\underset{\boldsymbol{W}_\ell^{\text{new}},\boldsymbol{W}_{\ell+1}^{\text{new}}}{\arg\max} \left\|\mathbb{E}_D\left[\frac{\partial L}{\partial \boldsymbol{W}_\ell^{\text{new}}}\right]\right\|_F^2 + \left\|\mathbb{E}_D\left[\frac{\partial L}{\partial \boldsymbol{W}_{\ell+1}^{\text{new}}}\right]\right\|_F^2 \text{ s.t. } \begin{cases} \|\boldsymbol{W}_\ell^{\text{new}}\|_F, \|\boldsymbol{W}_{\ell+1}^{\text{new}}\|_F \leq c \\ \boldsymbol{W}_{\ell+1}^{\text{new}}\boldsymbol{h}_\ell^{\text{new}} = 0 \end{cases} \tag{4}$$

Motivation: Large gradients lead to large objective decrease Consider running gradient descent on a function that is differentiable with a β-Lipschitz gradient. A classical result for this class of functions is that the decrease in objective function after one step of gradient descent with step-size $1/\beta$ is upper bounded by $L(\boldsymbol{W}_\ell) - \frac{\beta}{2}\|\nabla L(\boldsymbol{W}_\ell)\|^2$ (Nesterov, 2003). This is an expression that decreases as the norm of the gradient increases.



This observation implies that the larger the gradient, the larger the expected decrease (assuming a constant Lipschitz constant). Hence, interleaving gradient maximizing steps within training will lead to an improved decrease in later iterations. We use this observation in the context of growing neural networks and propose a method that interleaves the normal training process with steps that maximize the gradient norm by adding new units, which we now describe in detail.

GradMax The general maximization problem in eq. (4) is non-trivial to solve. However, we show that with some simplifying assumptions we can find an approximate solution using a singular value decomposition (SVD).

We begin by introducing the notation. Consider 3 fully connected layers denoted with indices $\ell - 1$, ℓ and $\ell + 1$ and the following recursive definition:

$$\boldsymbol{z}_\ell = \boldsymbol{W}_\ell \boldsymbol{h}_{\ell-1} \tag{5}$$
$$\boldsymbol{h}_\ell = f(\boldsymbol{z}_\ell)\,, \tag{6}$$

where subscripts denote layer-indices. Let M_ℓ denote the number of units in layer ℓ, and f the activation function. The vectors $\boldsymbol{z}_\ell, \boldsymbol{h}_\ell \in \mathbb{R}^{M_\ell}$ are pre-activations and activations, with $\boldsymbol{h}_0 = \boldsymbol{x}$ an input sampled from the dataset D. The entries of $\boldsymbol{W}_\ell \in \mathbb{R}^{M_{\ell-1} \times M_\ell}$ are the parameters of the layer.

When growing k neurons the weight matrices $\boldsymbol{W}_\ell$ and $\boldsymbol{W}_{\ell+1}$ are replaced by $\boldsymbol{W}_\ell^+$ and $\boldsymbol{W}_{\ell+1}^+$ as defined in eqs. (2) and (3). We denote the pre-activations and activations of the new neurons with

$\boldsymbol{z}_\ell^{\text{new}}$ and $\boldsymbol{h}_\ell^{\text{new}}$. The gradients of the new weights can be derived using the chain rule:

$$\frac{\partial L}{\partial \boldsymbol{W}_\ell^{\text{new}}} = \left(f'(\boldsymbol{z}_\ell^{\text{new}}) \odot \boldsymbol{W}_{\ell+1}^{\text{new},\top} \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \right) \boldsymbol{h}_{\ell-1}^\top \tag{7}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_{\ell+1}^{\text{new}}} = \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_\ell^{\text{new},\top} . \tag{8}$$

The simplifying assumptions that we will now make are that $\boldsymbol{W}_\ell^{\text{new}} = 0$ and that $f(0) = 0$ with gradient $f'(0) = 1$. Note that this guarantees that $\boldsymbol{W}_{\ell+1}^{\text{new}} \boldsymbol{h}_\ell^{\text{new}} = 0$, independent of the training data. Moreover, it simplifies the gradients to

$$\frac{\partial L}{\partial \boldsymbol{W}_\ell^{\text{new}}} = \boldsymbol{W}_{\ell+1}^{\text{new},\top} \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top \tag{9}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_{\ell+1}^{\text{new}}} = 0 , \tag{10}$$

which reduces our problem to

$$\underset{\boldsymbol{W}_{\ell+1}^{\text{new}}}{\arg\max} \left\| \boldsymbol{W}_{\ell+1}^{\text{new},\top} \mathbb{E}_D \left[\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top \right] \right\|_F^2 , \quad \text{s.t.} \ \left\| \boldsymbol{W}_{\ell+1}^{\text{new}} \right\|_F \leq c . \tag{11}$$

The solution to this maximization problem is letting the columns of $\boldsymbol{W}_{\ell+1}^{\text{new}}$ be the top k left-singular vectors of the matrix $\mathbb{E}_D \left[\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top \right]$ and scaling them by $\frac{c}{\|(\sigma_1, \ldots, \sigma_k)\|}$ (where σ_i is the i-th largest singular value). In order to make a fair comparison between different methods we scale each initialization such that their norm is equal to the mean norm of the existing neurons.

Note that it is feasible to also use the singular values to guide **where** and **when** to grow. For example, neurons could be added when the singular values meet a certain threshold, and layers to grow could be chosen depending on which have the largest singular values. We leave this for future work.

Non-linearities and normalization Note that our assumptions that $f(0) = 0$ and $f'(0) = 1$ apply to common activation functions such as rectified linear units (if zero is included in the linear part) and the hyperbolic tangent. Other functions could be adapted to fit this definition as well (e.g., by shifting the output of the logistic function by $-\frac{1}{2}$).

In fact our derivation holds for $f'(0) = a, a \neq 0$. This means that batch normalization can also be used, since it meets these assumptions. However, care must be taken with activation functions (or their derivatives) that are unstable around zero. For example, batch normalization scales the gradients by $\frac{1}{\varepsilon}$ when all inputs are zero, which can lead to large jumps in parameter space (Lewkowycz et al., 2020).

GradMax-Optimized (GradMaxOpt) One can also directly optimize eq. (11) using an iterative method such as projected gradient descent. Our experiments show that this approach generally does not work as well as using the SVD, highlighting the benefit of having a closed-form solution.

However, if the outgoing weights are set to zero ($\boldsymbol{W}_{\ell+1}^{\text{new}} = 0$) instead of the incoming weights then the solution can no longer be found using SVD and direct optimization of eq. (4) could provide a solution. This could be preferable in some situations since it removes the constraints on the activation function. Moreover, it can avoid the unstable behaviour of functions such as batch normalization.

Full gradient estimation Solving eq. (11) requires calculating the gradient $\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top$ over the full dataset. This can be expensive so in practical applications we propose using a large minibatch instead. If the minibatch gradient is sufficiently close to the full gradient then the top singular vectors are likely to be close as well (Stewart, 1998).

To validate that we can approximate the SVD of the full gradient with the SVD of a large minibatch we calculate both and plot the alignment of the left-singular vectors. We define alignment as the absolute value of the Frobenius trace of the two matrices of singular vectors, normalized with $\frac{1}{k}$.

The plots in fig. 2 show that at the beginning of training the singular vectors strongly align. At the end of training the gradients are less correlated and hence the alignment decreases. However, it is still a reasonable estimate.

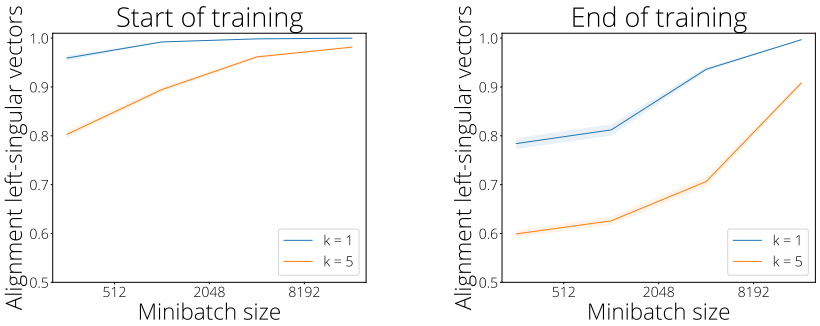


Figure 2: The alignment of the top-k left-singular vectors for WRN-28-1. The alignment is calculated between the full gradient and minibatches of varying sizes. We do not apply random cropping or flips to the inputs. A total of 10 experiments are run and confidence intervals of 95% are plotted.

4 Experiments

We implement GradMax using Tensorflow (Abadi et al., 2015). We modify the implementation of the standard ReLU activation to have gradient 1 at 0. See appendix A for further implementation details. We compare GradMax against random growing.

We have 3 main goals in this section. First, we empirically verify that the gradient norm is significantly increased using our method. Then, we verify our hypothesis that increasing the gradient norm at a given step improves the training dynamics beyond the growing step. Finally, we gather experimental evidence that networks grown with GradMax can achieve better training loss and generalization compared to baselines and other methods in a teacher-student setting.

4.1 Teacher-Student experiments with MLPs

Teacher-Student task We use a teacher-student task (Lawrence et al., 1997) to compare different growing algorithms. In this setting a student network must learn the function of a given teacher network. Our teacher network f_t consists of m_i input nodes, m_h hidden nodes and m_o output nodes (denoted with $m_i : m_h : m_o$). We initialize weights of f_t randomly from the range $[-1, 1]$ and then sample $N = 1000$ training points D_x from a Gaussian distribution with zero mean and unit variance and pass it through f_t to obtain target values D_y. With this training data we train various student networks (f_s) minimizing the squared loss between $f_s(D_x)$ and D_y.

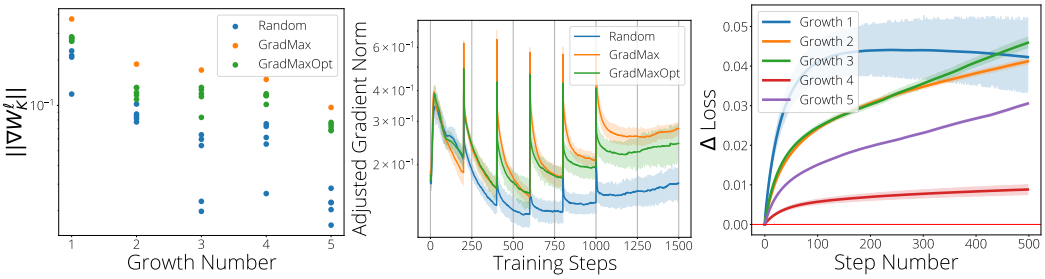
A key property of this teacher-student settings is that when the student network has the same architecture as the teacher network, the optimization problem has multiple global minima with 0 training loss. Here we highlight results using fully connected layers alone. In Appendix C we show additional validation experiments as well as repeat experiments from this section with convolutions and batch normalization.

Verifying GradMax In our initial experiments we use a teacher network of size $20 : 10 : 10$. All growing student networks begin with a smaller hidden layer of $20 : 5 : 10$. We grow the hidden layer to match the teacher architecture in 5 growth steps performed every 200 training steps. The baselines *Baseline-Small* and *Baseline-Big* are trained using the fixed architectures $20 : 5 : 10$ and $20 : 10 : 10$. After the final growth we perform 500 more steps resulting in 1500 training steps in total. We also run the version of GradMax in which we optimize eq. (11) directly using gradient descent (GradMaxOpt). We start with a random initialization and use the Adam optimizer. The weight matrix is scaled to the target norm c after each gradient step.

In fig. 3a we show that in this setup GradMax is able to initialize the new neurons with a significantly higher gradient norm compared to random growing. The results for GradMaxOpt show that naive direct optimization of the gradient norm does not recover the solution found by GradMax using the SVD. This highlights the benefit of having formulated a problem that can be directly solved. In fig. 3b we plot the total adjusted gradient norm[1] and observe that the larger gradient norm af-

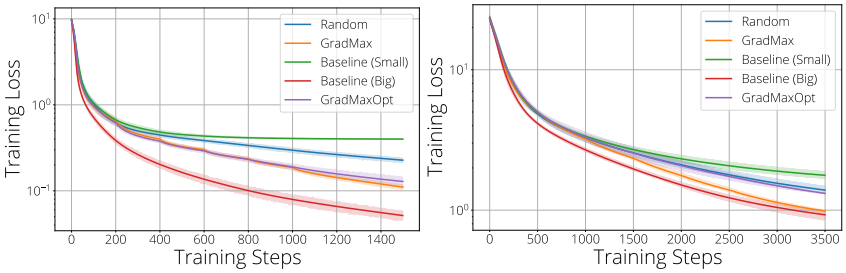
[1]We divide the gradients by the training loss to factor out the linear scaling effect of the training loss on the gradient norms.

ter growing persists for future training steps. Finally, in fig. 3c we plot the difference in training loss when GradMax is used for growing compared to randomly growing. We observe consistent improvements in the training dynamics which last for over 500 training steps. This supports our hypothesis that increasing the gradient norm accelerates training and leads to a greater decrease in the training loss for subsequent steps.



(a) After Growth (b) During Training (c) After N Steps

Figure 3: **(a)** In this plot we load checkpoints from each growth step generated during the Random (Random) experiments. We grow a single neuron and measure the norm of the gradients with respect to $\boldsymbol{W}_\ell^{\text{new}}$. Since Random and GradMaxOpt are stochastic we repeat those experiments 10 times. **(b)** We measure the norm of the flattened parameters throughout training. **(c)** Similar to (a), we load checkpoints from each growth step and grow a new neuron using GradMax (f_g) and Random (f_r). Then we continue training for 500 steps and plot the difference in training loss (i.e., $L(f_r) - L(f_g)$). This experiments is repeated 5 times.



(a) Small ($K = 1$, $f_t = 20 : 10 : 10$) (b) Large ($K = 5$, $f_t = 100 : 50 : 10$)

Figure 4: Training curves averaged over 5 different runs and provided with 80% confidence intervals. In both settings GradMax significantly improves optimization over Random.

Training Curves We plot the training curves for two different teacher networks in fig. 4. For subfigure 4a we trained a teacher network of $20 : 10 : 10$ whereas subfigure 4b uses a larger teacher network of size $100 : 50 : 10$. For the large teacher network setting we begin with student networks of size $100 : 25 : 10$ that are grown every 500 steps, adding 5 neurons at a time resulting for 5 growth steps. We train the grown networks for an additional 1000 iterations resulting in 3500 iterations in total. Further experimental details are shared in appendix C.

In these experiments the initial student models require about half the number of FLOPS required by the teacher models. Therefore training and growing student models with a linear growing schedule costs only 75% of the FLOPS required to train *Baseline-Big*[2]. In all cases GradMax achieves lower training loss compared to the random and GradMax-Optimized (GradMaxOpt) methods. For the larger network, GradMax even matches the performance of a network trained from scratch (*Big-Baseline*). However for small teacher networks, all growing methods fall short of matching the *Baseline-Big* performance. This highlights an important gap to fill in future research. These results agree with the results in Berner et al. (2019) in which the authors show that training the final agent from scratch works better than growing an existing agent with random growth.

[2]The extra operations required by GradMax run faster than a single training step and therefore their cost is negligible.

Dataset	Architecture	Baseline-S	Baseline-B	Random	Firefly	Gradmax
CIFAR-10	WRN-28-1	89.9±0.3	92.9±0.2	90.6±0.2	**90.8±0.3**	**91.1±0.1**
	VGG11	84.1±0.1	86.6±0.3	83.8±0.6	**84.0±0.2**	**84.4±0.4**
CIFAR-100	WRN-28-1	63.7±0.0	69.3±0.1	**66.7±0.4**	**66.5±0.1**	**66.8±0.2**
ImageNet-2012	Mobilenet-V1	55.0±0.0	70.8±0.0	66.9±0.3	66.4±0.1	**68.6±0.2**

Table 1: Test accuracy of different baselines and growing methods on different tasks. All results are averaged over 3 random seeds (used for training).

4.2 Image Classification Experiments

In this section we benchmark GradMax using various growing schedules and architectures on CIFAR-10. Note that GradMax can be applied to convolutional networks (see Appendix B for implementation details). Similar to our results in the previous section, we share baselines where we train the seed architecture (*Small-Baseline*) and the target architecture (*Big-Baseline*) from scratch without growing. As an additional baseline we implement a simpler version of Firefly which grows neurons by minimizing the training loss directly (without the extra candidates used by the original method). We refer to this baseline as Firefly-Optimized (Firefly-Opt). Firefly requires non-zero outgoing weights in order to optimize the loss. Therefore we initialize the outgoing weights for all methods to small random values ($\|\boldsymbol{W}_{\ell+1}^{\text{new}}\| = \varepsilon$ for $\varepsilon = 10^{-4}$). We use a batch size of 512 for ImageNet (128 for CIFAR) experiments and calculate the SVD using the same batch size.

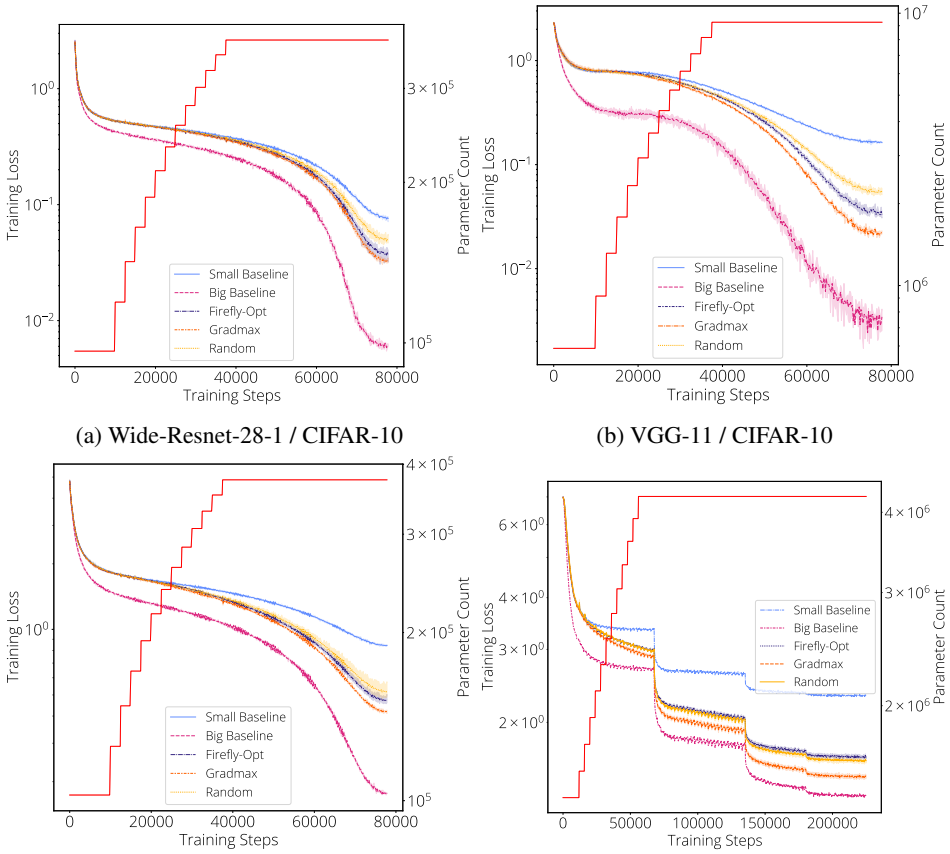
We add new neurons during the first half of training. Here our goal is not to obtain state of art results, but to assess how the initialization of new neurons affects the final performance. All networks are trained for the same total number of steps, and thus *Baseline-Small* and the grown models require less FLOPS than *Baseline-Big* to train. We share our results in table 1 and observe consistent improvements when GradMax is used.

CIFAR-10/100 We perform experiments on CIFAR-10 and CIFAR-100 using two different architectures: VGG-11 (Simonyan & Zisserman, 2015) and WRN-28-1 (Zagoruyko & Komodakis, 2016). For both architectures we start using a width multiplier of 0.25 for our seed architecture. For the ResNet architecture we only scale down the first convolutional layer to prevent mismatch when adding skip-connections. We train the networks for 200 epochs using SGD with a momentum of 0.9 and decay the learning rate using a cosine schedule. In both experiments we use a batch size of 128 and perform a small learning rate sweep to find learning rates that give best test accuracy for baseline training. We find 0.1 for Wide-ResNet and 0.05 for VGG to perform best and use the same learning rates for all different methods. In figs. 5a to 5c we plot the training loss for different growing strategies and share the test accuracy in table 1.

ImageNet-2012 We grow MobileNet-v1 architectures on ImageNet-2012 using a seed architecture with width multiplier 0.25. The MobileNet-v1 architecture contains depth-wise convolutions between each convolutional layers. When growing convolutional layers, we initialize the depth-wise convolution in between to identity. Training loss for this settting is shared in fig. 5d.

Batch Normalization Results and Inverse Formulation We perform our main set of experiments using networks without batch norm and setting the incoming weights to zero. However, in some cases using batch norm or setting outgoing weights to zero might be useful. table 2 shows the results for such alternatives. For this particular setting we observe that batch norm has limited effect on results, however we observe consistent improvements when outgoing weights are set to zero. We use GradmaxOpt in this setting and observe improvements over both random and Firefly.

Hyperparameters In fig. 6 we show the effect of various hyperparameters on the performance of GradMax when growing MobileNet-v1 on ImageNet-2012. First on fig. 6a we compare different growing schedules. Growing neurons early in the training and growing every 5000 step achieves the best results. Subfigure fig. 6b shows the effect of ε used in our experiments. As expected, larger values of ε (and therefore not preserving the output of the network during the growing) harm the final test accuracy. Finally, fig. 6c shows the effect of the scale used when initializing the new neurons. We compare normalizing the scales using the mean norm of existing neurons (*mean*) to using no normalization (*fixed*) and observe no visible different in performance. We observe relatively robust for values larger than 0.5 and therefore simply set the scale to 0.5 in all of our experiments.



(a) Wide-Resnet-28-1 / CIFAR-10

(b) VGG-11 / CIFAR-10

(c) Wide-Resnet-28-1 / CIFAR-100

(d) MobileNet-v1 / ImageNet-2012

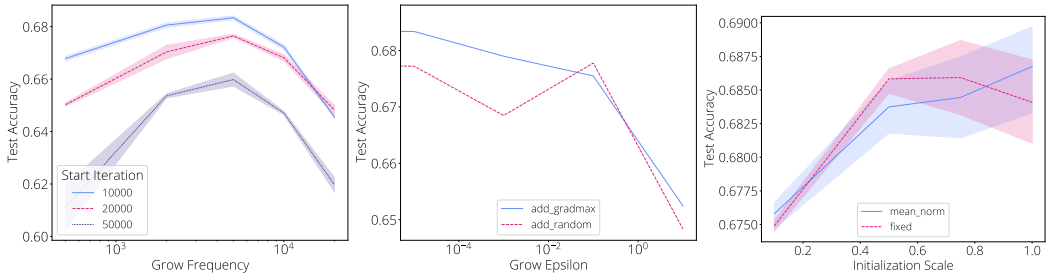
Figure 5: We plot training loss over time, each experiment is averaged over 3 experiments. Number of parameters of the network trained increases over time. Red lines indicate number of parameters over training.

BN	Inverse	Baseline-S	Baseline-B	Random	Firefly	Gradmax(-Opt)
✗	✗	89.9±0.3	92.9±0.2	90.6±0.2	90.8±0.3	91.1±0.1
✗	✓			92.1±0.2	92.2±0.2	92.4±0.1
✓	✗	90.2±0.3	93.4±0.1	91.0±0.2	90.9±0.3	91.1±0.1
✓	✓			92.1±0.2	92.2±0.2	92.4±0.1

Table 2: Average test accuracy when growing networks with batch normalization and outgoing weights set to zero. *BN* refers to batch normalization and *inverse* indicates that we set the outgoing weights to zero. When the outgoing weights are set to zero, we use GradmaxOpt.

5 Related Work

Pruning This method is commonly used to reduce the size of a neural network after training. Liu et al. (2018) showed structured pruning algorithms that remove entire neurons often can't improve over training the final architecture from scratch. Therefore they argued that the utility of structured pruning algorithms lie in finding optimal widths for each layer. Since then various works have focused on finding the optimal widths for a given architecture using group sparsity (Alvarez & Salzmann, 2016), penalizing batch-norm parameters (Ye et al., 2018; Liu et al., 2017), and RL (He et al., 2018b). Gao et al. (2019) focused on dynamically pruning neurons conditioned on input in order to save run-time. He et al. (2018a); Gao et al. (2020) allowed pruned neurons to be updated to find more efficient architectures. Ye et al. (2020) selects neurons from a pretrained model to minimize the loss in a round robin fashion.



(a) (b) (c)

Figure 6: We grow MobileNet-v1 networks during the ImageNet training to investigate the effect of (a) growing schedule, (b) ε, and (c) scale used in initialization on the test accuracy.

Growing neural networks Fukumizu & Amari (2000) show that local minima can turned into saddle points by adding neurons. Chen et al. (2016) proposed splitting neurons and Wei et al. (2016) presented a more general study of performance-preserving architecture transformations. Elsken et al. (2017) apply these transformations to NAS. Wen et al. (2019) proposed growing neurons and discovered that it is best to grow layers early on. Lu et al. (2018) also add new layers, but they sparsify the resulting network. *Convex neural networks* frames the learning of one-hidden layer neural networks as an infinite-dimensional convex problem (Bengio et al., 2006; Rosset et al., 2007; Bach, 2017) and then approximates the solution by an incremental algorithm that adds a neuron at each step.

Kilcher et al. (2019) show that the scaling of the outbound weights can be optimized with respect to gradient norm when the original network is at local minima. The solution is given by the closed form using a matrix of outbound partial derivatives. Liu et al. (2019) propose an algorithm for splitting neurons. For each neuron it computes an approximate reduction in loss if this weight were cloned and adjusted by a given offset. The solution to this problem is given by the eigendecomposition of a "splitting matrix". The splitting is most effective when the network is very close to convergence. Calculating the splitting matrix is expensive so in Wang et al. (2019) the authors propose a faster approximation. Wu et al. (2020b) expande the set of splitting directions which helps avoid local minima in architecture space. In the context of manifold learning, Vladymyrov (2019) also proposed to grow the number of dimensions of the embedding manifold in order to escape local minima. Similar to our approach, new parameters are found in a closed form.

NAS usually involves optimizing the weights and architecture separately using RL (Baker et al., 2016; Zoph & Le, 2016; Cai et al., 2017) or evolutionary (Real et al., 2017) algorithms to optimize the architecture. This can require training thousands of networks from scratch. More recent work tries to address by using "network morphisms", i.e., transformations that leave the function output unchanged. GradMax is an example of such a network morphism.

6 Discussion

Limitations GradMax requires activation functions to map zero to zero and have non-zero derivative at the origin. Some activation functions, such as real-basis functions, don't satisfy these requirements. We studied fully connected layers and convolutional layers, but did not consider the combination of the two or other architectures such as transformers.

Future work We are looking to address these limitations in future work. Moreover, in this work we didn't address the questions as to when and where networks should be grown, although we outlined ways in which our method could be adapted to do so. Our method could also be a useful network morphism to be used as part of a more complex NAS method. These are further avenues for exploration.

7 Conclusion

In this work we presented GradMax, a new approach for growing neural networks that aims to maximize the gradient norm when growing. We verified our hypothesis and demonstrated that it provides faster training in a variety of settings.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/.

Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, 2016. URL https://proceedings.neurips.cc/paper/2016/file/6e7d2da6d3953058db75714ac400b584-Paper.pdf.

Francis Bach. Breaking the curse of dimensionality with convex neural networks. *The Journal of Machine Learning Research*, 2017.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

Yoshua Bengio, Nicolas Roux, Pascal Vincent, Olivier Delalleau, and Patrice Marcotte. Convex neural networks. In *Advances in Neural Information Processing Systems*, volume 18, 2006. URL https://proceedings.neurips.cc/paper/2005/file/0fc170ecbb8ff1afb2c6de48ea5343e7-Paper.pdf.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, 2019. URL http://arxiv.org/abs/1912.06680.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Reinforcement learning for architecture search by network transformation. *arXiv preprint arXiv:1707.04873*, 2017.

Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. In *4th International Conference on Learning Representations*, 2016.

Stéphane dÁscoli, Levent Sagun, Giulio Biroli, and Joan Bruna. Finding the needle in the haystack with convolutions: on the benefits of architectural bias. In *Advances in Neural Information Processing Systems 32*, 2019.

Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 2019.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pp. 1189–1232, 2001.

Kenji Fukumizu and Shun-ichi Amari. Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural networks*, 2000.

Shangqian Gao, F. Huang, J. Pei, and H. Huang. Discrete model compression with resource constraint for deep neural networks. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Xitong Gao, Y. Zhao, L. Dudziak, R. Mullins, and C. Xu. Dynamic channel pruning: Feature boosting and suppression. *ArXiv*, 2019.

Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Y. Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *IJCAI*, 2018a.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, L. Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, 2018b.

Yannic Kilcher, Gary Bécigneul, and Thomas Hofmann. Escaping flat areas via function-preserving structural network modifications, 2019. URL https://openreview.net/forum?id=H1eadi0cFQ.

Steve Lawrence, C. Giles, and Ah Tsoi. Lessons in neural network training: Overfitting may be harder than expected. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

Aitor Lewkowycz, Yasaman Bahri, Ethan Dyer, Jascha Sohl-Dickstein, and Guy Gur-Ari. The large learning rate phase of deep learning: the catapult mechanism. *ArXiv*, abs/2003.02218, 2020.

Qiang Liu, Lemeng Wu, and Dilin Wang. Splitting steepest descent for growing neural architectures. *ArXiv*, 2019. URL https://arxiv.org/abs/1910.02366.

Zhuang Liu, J. Li, Zhiqiang Shen, Gao Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the Value of Network Pruning. *ArXiv*, 2018.

Jun Lu, Wei Ma, and Boi Faltings. Compnet: Neural networks growing via the compact network morphism. *arXiv preprint arXiv:1804.10316*, 2018.

Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2003.

Behnam Neyshabur. Towards learning convolutions from scratch. *ArXiv*, 2020. URL https://arxiv.org/abs/2007.13657.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*. PMLR, 2017.

Saharon Rosset, Grzegorz Swirszcz, Nathan Srebro, and Ji Zhu. ℓ_1 regularization in infinite dimensional feature spaces. In *International Conference on Computational Learning Theory*. Springer, 2007.

Andrei A. Rusu, Neil C. Rabinowitz, G. Desjardins, Hubert Soyer, J. Kirkpatrick, K. Kavukcuoglu, Razvan Pascanu, and R. Hadsell. Progressive neural networks. *ArXiv*, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2015.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 2002. URL https://doi.org/10.1162/106365602320169811.

Gilbert W Stewart. Perturbation theory for the singular value decomposition. Technical report, 1998.

Neil Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel Manso. Deep learning's diminishing returns. *IEEE Spectrum*, 58(10), 2021.

Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pp. 19–24. IEEE, 2017.

Max Vladymyrov. No pressure! addressing the problem of local minima in manifold learning algorithms. *arXiv preprint arXiv:1906.11389*, 2019.

Dilin Wang, Meng Li, Lemeng Wu, Vikas Chandra, and Qiang Liu. Energy-aware neural architecture optimization with fast splitting steepest descent. *arXiv preprint arXiv:1910.03103*, 2019.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International Conference on Machine Learning*. PMLR, 2016.

Wei Wen, Feng Yan, and Hai Helen Li. Autogrow: Automatic layer growing in deep convolutional networks. *Arxiv*, 2019. URL http://arxiv.org/abs/1906.02909.

Lemeng Wu, Bo Liu, Peter Stone, and Qiang Liu. Firefly neural architecture descent: a general approach for growing neural networks. In *Advances in Neural Information Processing Systems*, 2020a. URL https://proceedings.neurips.cc/paper/2020/file/fdbe012e2e11314b96402b32c0df26b7-Paper.pdf.

Lemeng Wu, Mao Ye, Qi Lei, Jason D Lee, and Qiang Liu. Steepest descent neural architecture optimization: Escaping local optimum with signed neural splitting. *arXiv preprint arXiv:2003.10392*, 2020b.

Jianbo Ye, Xin Lu, Zhe Lin, and James Z. Wang. Rethinking the Smaller-Norm-Less-Informative Assumption in Channel Pruning of Convolution Layers. *arXiv*, 2018. URL http://arxiv.org/abs/1802.00124.

Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam R. Klivans, and Qiang Liu. Good subnetworks provably exist: Pruning via greedy forward selection. In *ICML*, 2020.

Xin Yuan, Pedro Henrique Pamplona Savarese, and Michael Maire. Growing efficient deep networks by structured continuous sparsification. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=wb3wxCObbRT.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *ArXiv*, abs/1605.07146, 2016.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A Implementation details

GradMax requires the calculation of the quantity $\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top$ (eq. (9)). The user could perform this outer product manually, but in most machine learning frameworks the following approach is easier to implement.

We note that when $\boldsymbol{W}_\ell^{\text{new}} = 0$ and $f(0) = 0$ we can write

$$\boldsymbol{W}_{\ell+1}^{\text{new}} f(\boldsymbol{W}_\ell^{\text{new}} \boldsymbol{h}_{\ell-1}) = \boldsymbol{W}_\ell^{\text{aux}} \boldsymbol{h}_{\ell-1}, \quad \boldsymbol{W}_\ell^{\text{aux}} \in \mathbb{R}^{M_{\ell-1} \times M_{\ell+1}} \tag{12}$$

and

$$\boldsymbol{z}_{\ell+1} = \boldsymbol{W}_{\ell+1} \boldsymbol{h}_\ell + \boldsymbol{W}_\ell^{\text{aux}} \boldsymbol{h}_{\ell-1} \,. \tag{13}$$

Note that $\boldsymbol{W}_\ell^{\text{aux}} = 0$. The advantage of this formulation is that

$$\frac{\partial L}{\partial \boldsymbol{W}_\ell^{\text{aux}}} = \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top \,. \tag{14}$$

Hence GradMax can easily be implemented in any framework with automatic differentiation by temporarily introducing this auxiliary matrix $\boldsymbol{W}_\ell^{\text{aux}}$ in order to calculate $\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \boldsymbol{h}_{\ell-1}^\top$. After the columns of $\boldsymbol{W}_{\ell+1}^{\text{new}}$ are calculated using the SVD, the matrix $\boldsymbol{W}_\ell^{\text{aux}}$ is replaced with the new layer ($\boldsymbol{z}_{\ell+1} = \boldsymbol{W}_{\ell+1} \boldsymbol{h}_\ell + \boldsymbol{W}_{\ell+1}^{\text{new}} f(\boldsymbol{W}_\ell^{\text{new}} \boldsymbol{h}_{\ell-1})$).

B CONVOLUTIONS

The derivation of GradMax for fully connected layers can readily be extended to convolutional layers:

$$\frac{\partial L}{\partial \boldsymbol{W}_\ell^{\text{new}}} = \left(f'(\boldsymbol{z}_\ell^{\text{new}}) \odot \boldsymbol{W}_{\ell+1}^{\text{new}} *^\top \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} \right) * \boldsymbol{h}_{\ell-1} \tag{15}$$

$$= \boldsymbol{W}_{\ell+1}^{\text{new}} * \left(\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} * \boldsymbol{h}_{\ell-1} \right) \tag{16}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_{\ell+1}^{\text{new}}} = \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} * \boldsymbol{h}_\ell^{\text{new}} \tag{17}$$

$$= 0 \tag{18}$$

where $*$ is a 2D convolution and $*^\top$ is the transposed 2D convolution (i.e., the gradient of the convolution).

As before, the required gradient is in practice most easily calculated by introducing an auxiliary convolutional layer such that $\frac{\partial L}{\partial \boldsymbol{W}_\ell^{\text{aux}}} = \frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} * \boldsymbol{h}_{\ell-1}$. We let

$$\boldsymbol{z}_{\ell+1} = \boldsymbol{W}_{\ell+1} * \boldsymbol{h}_\ell + \boldsymbol{W}_\ell^{\text{aux}} * \boldsymbol{h}_{\ell-1} \,. \tag{19}$$

where $\boldsymbol{W}_\ell^{\text{aux}} = 0$ are filters of the appropriate size. Namely, if $\boldsymbol{W}_\ell$ are filters of size (h_ℓ, w_ℓ) with i_ℓ input channels and $\boldsymbol{W}_{\ell+1}$ contains filters of size $(h_{\ell+1}, w_{\ell+1})$ with $o_{\ell+1}$ output channels, then $\boldsymbol{W}_\ell^{\text{aux}}$ will have filters of size $(h_\ell + h_{\ell+1} - 1, w_\ell + w_{\ell+1} - 1)$ with i_ℓ input and $o_{\ell+1}$ output channels.

We now have the equivalent of equation 11 to solve for convolutional layers:

$$\underset{\boldsymbol{W}_{\ell+1}^{\text{new}}}{\arg\max} \left\| \boldsymbol{W}_{\ell+1}^{\text{new}} * \mathbb{E}_D \left[\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} * \boldsymbol{h}_{\ell-1}^\top \right] \right\|_F^2 , \quad \text{s.t.} \ \left\| \boldsymbol{W}_{\ell+1}^{\text{new}} \right\|_F \leq c \,. \tag{20}$$

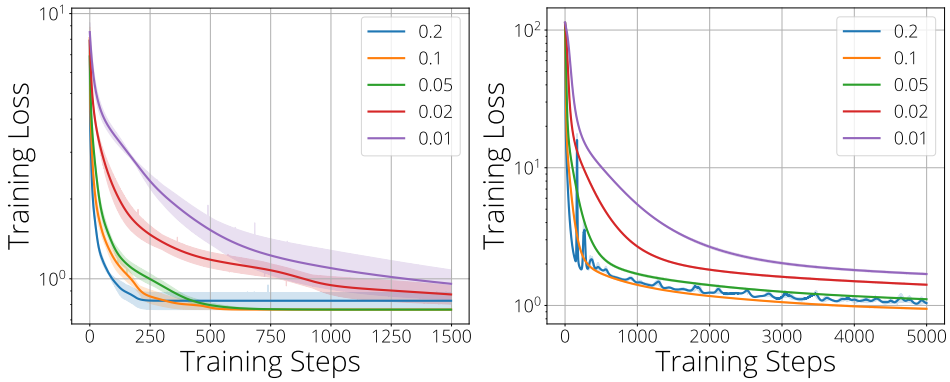
We express the 2D convolution as a matrix multiplication (Vasudevan et al., 2017) using the *im2col* method. This means that $\boldsymbol{W}_{\ell+1}^{\text{new}}$ is flattened to a matrix of size $\mathbb{R}^{o_{\ell+1} \cdot h_{\ell+1} \cdot w_{\ell+1} \times k}$. The matrix $\mathbb{E}_D \left[\frac{\partial L}{\partial \boldsymbol{z}_{\ell+1}} * \boldsymbol{h}_{\ell-1}^\top \right]$ is of size $\mathbb{R}^{i_\ell \times o_{\ell+1} \times h_\ell + h_{\ell+1} - 1 \times w_\ell + w_{\ell+1} - 1}$ but is turned into a larger matrix of size $\mathbb{R}^{o_{\ell+1} \cdot h_{\ell+1} \cdot w_{\ell+1} \times i_\ell \cdot h_\ell \cdot w_\ell}$ by extracting the appropriate patches (depending on the filter sizes, padding, and strides) which are flattened and then concatenated. SVD can then be readily applied to this reformulated problem and the resulting top-k left-singular vectors can be reshaped to form the filters of the new layer.

C ADDITIONAL EXPERIMENTAL DETAILS FOR STUDENT-TEACHER EXPERIMENTS

Learning rate. For both experiments we perform a small hyper parameter search over the learning rate. In fig. 7 we share learning curves using different learning rates that range between 0.2 and 0.01. Higher learning rates bring faster convergence, however best results are obtained with the second highest learning rate equal to 0.1. To avoid confounds related to the optimizer, we restrict the experiments to gradient descent with constant learning rate.

Correlation between singular values and the final decrease. We know from the theory that the norm of the gradient is inversely proportional to the singular values given by the solution of equation 11. In other words, solution corresponding to the top singular values should give the largest norm of the gradient for new weights and therefore should have the best training dynamics in the long run. In fig.8 we demonstrate this on a simple Student/Teacher set up as we used in the main paper. We first train own network without any growing (blue curve on the top graph). We then grow the network at different stages of optimization using any of the five singular values and let the network converge to some value. As we can see, darker colors corresponding to the larger singular values generally converge faster and correspond to the lower values of the objective function.

At the lower portion of the figure, we quantify this result more precisely by repeating the experiment five times and computing the Pearson correlation coefficient between the vector of singular values



(a) Large ($f_t = 20 : 10 : 10$) (b) Large ($f_t = 400 : 200 : 10$)

Figure 7: Effect of learning rate on the optimization speed and quality. We repeat each experiment 3 times with a different seed and report the average values with 80% confidence intervals.

and the loss decrease some iterations after growing. There x-axis represents how many iterations have passed since the growth and the y-axis shows the correlation between the loss values at that iteration and the singular values. The lower the values the more (anti-)correlated the values are and, therefore, the more we can trust the GradMax algorithm. The iteration it always negative and gets worse as we iterate. However, if we use GradMax at the later stages of the algorithm close to convergence, the correlation holds for a while.

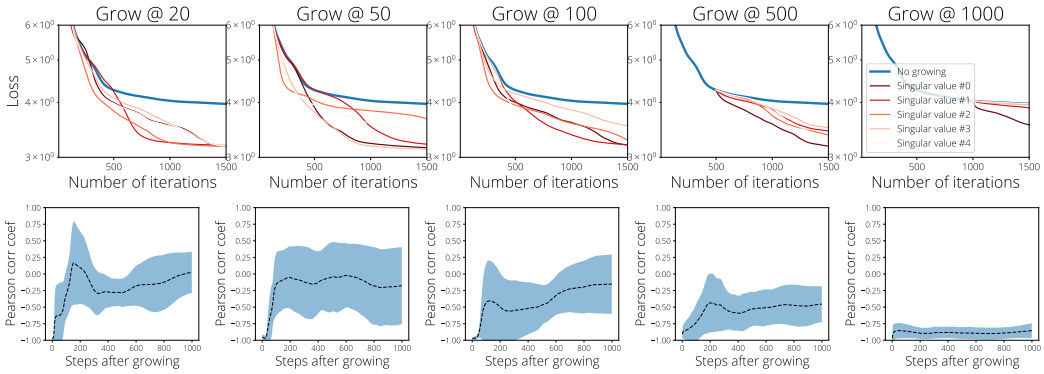
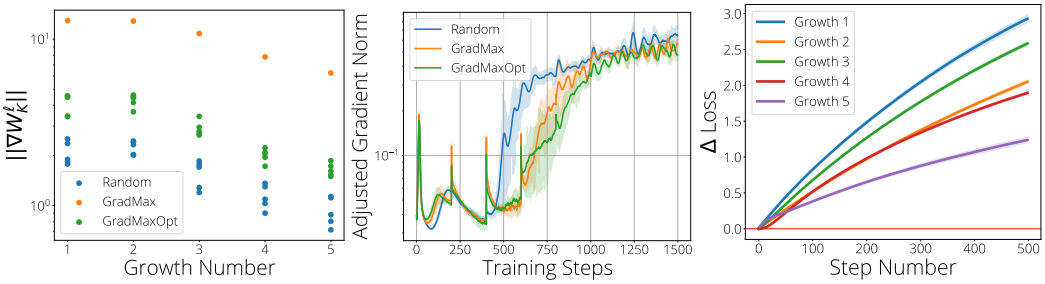


Figure 8: Long-term effect of growing. *Top:* Growing once with one of 5 singular values starting at different iteration (20th, 50th, 100th, 500th or 1000th). Blue curve represents no growing. *Bottom:* The correlation between the singular values and the loss function decrease after a certain growing iteration.

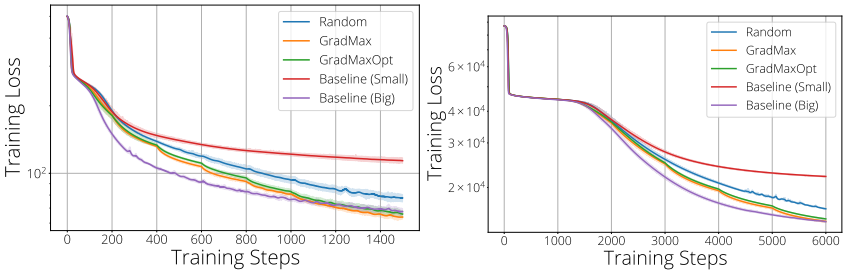
Student-Teacher Experiments with Convolutional Networks. Similar to the experiments in fig. 3a, here we show the results of the Student-Teacher task on a convolutional model. As visible on fig. 9, the same conclusions can be made for this case as well. GradMax achieves the best norm of the gradient after each growth. Interestingly, the gradient norm is somewhat larger for the Random growth, but in fig. 10 we see that GradMax achieves better results overall.

Student-Teacher Experiments with Batch Norm Similar to the MLP and convolution cases, figures 11c and 12 provide results for the GradMax performance for the model with BatchNorm.



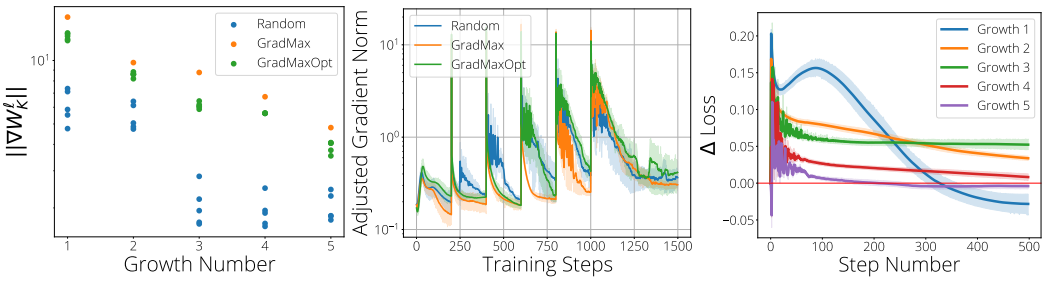
(a) After Growth (b) During Training (c) After N Steps

Figure 9: **(a)** In this plot we load checkpoints from each growth step generated during the Random experiments. We grow a single neuron and measure the norm of the gradients with respect to $\boldsymbol{W}_\ell^{new}$. Since Random and GradMaxOpt are stochastic we repeat those experiments 10 times. GradMax provides the maximum gradient norm **(b)** We measure the norm of flattened parameters throughout the training. GradMax improves gradient norm over Random. **(c)** Similar to (a), we load checkpoints from each growth step and grow a new neuron using GradMax (f_g) and Random (f_r). Then we continue training for 500 steps and plot the difference in training loss (i.e. $L(f_r) - L(f_g)$). All experiments are repeated 5 times.



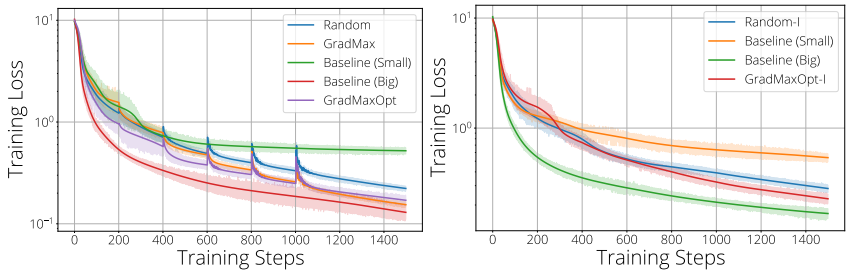
(a) Small ($K = 1$, $f_t = 20 : 10 : 10$) (b) Large ($K = 5$, $f_t = 100 : 50 : 10$)

Figure 10: Training curves averaged over 5 different runs and provided with 80% confidence intervals. In both settings GradMax significantly improves optimization over Random. Split-based methods seems to cause some instability in large network settings causing frequent jumps in the training objective.



(a) After Growth (b) During Training (c) After N Steps

Figure 11: **(a)** In this plot we load checkpoints from each growth step generated during the Random experiments. We grow a single neuron and measure the norm of the gradients with respect to $\boldsymbol{W}_\ell^{new}$. Since Random and GradMaxOpt is stochastic we repeat those experiments 10 times. GradMax provides the maximum gradient norm **(b)** We measure the norm of flattened parameters throughout the training. GradMax improves gradient norm over Random. **(c)** Similar to (a), we load checkpoints from each growth step and grow a new neuron using GradMax (f_g) and Random (f_r). Then we continue training for 500 steps and plot the difference in training loss (i.e. $L(f_r) - L(f_g)$). All experiments are repeated 5 times.



(a) Regular

(b) Outgoing zero$\boldsymbol{W}_{\ell+1}^{new} = 0$

Figure 12: Training curves averaged over 5 different runs and provided with 80% confidence intervals. In both settings GradMax significantly improves optimization over Random. Split-based methods seems to cause some instability in large network settings causing frequent jumps in the training objective.