Abstract

The training of sparse neural networks is becoming an increasingly important tool for reducing the computational footprint of models at training and evaluation, as well enabling the effective scaling up of models. Whereas much work over the years has been dedicated to specialised pruning techniques, little attention has been paid to the inherent effect of gradient based training on model sparsity. In this work, we introduce Powerpropagation, a new weight-parameterisation for neural networks that leads to inherently sparse models. Exploiting the behaviour of gradient descent, our method gives rise to weight updates exhibiting a “rich get richer” dynamic, leaving low-magnitude parameters largely unaffected by learning. Models trained in this manner exhibit similar performance, but have a distribution with markedly higher density at zero, allowing more parameters to be pruned safely. Powerpropagation is general, intuitive, cheap and straight-forward to implement and can readily be combined with various other techniques. To highlight its versatility, we explore it in two very different settings: Firstly, following a recent line of work, we investigate its effect on sparse training for resource-constrained settings. Here, we combine Powerpropagation with a traditional weight-pruning technique as well as recent state-of-the-art sparse-to-sparse algorithms, showing superior performance on the ImageNet benchmark. Secondly, we advocate the use of sparsity in overcoming catastrophic forgetting, where compressed representations allow accommodating a large number of tasks at fixed model capacity. In all cases our reparameterisation considerably increases the efficacy of the off-the-shelf methods.

1 Introduction

Deep learning models are emerging as the dominant choice across several domains, from language [e.g. 1, 2] to vision [e.g. 3, 4] to RL [e.g. 5, 6]. One particular characteristic of these architectures is that they perform optimally in the overparameterised regime. In fact, their size seems to be mostly limited by hardware constraints. While this is potentially counter-intuitive, given a classical view on overfitting, the current understanding is that model size tends to have a dual role: It leads to better behaved loss surfaces, making optimisation easy, but also acts as a regulariser. This gives rise to the double descent phenomena, where test error initially behaves as expected, growing with model size due to overfitting, but then decreases again as the model keeps growing, and asymptotes as the model size goes to infinity to a better performance than obtained in the classical regime [7, 8].
As a consequence, many of the state of the art models tend to be prohibitively large, making them inaccessible to large portions of the research community despite scale still being a driving force in obtaining better performance. To address the high computational cost of inference, a growing body of work has been exploring ways to compress these models. As highlighted by several works [e.g. 9, 10, 11], size is only used as a crutch during the optimisation process, while the final solution requires a fraction of the capacity of the model. A typical approach therefore is to sparsify or prune the neural network after training by eliminating parameters that do not play a vital role in the functional behaviour of the model. Furthermore, there is a growing interest in sparse training [e.g. 12, 13, 14], where the model is regularly pruned or sparsified during training in order to reduce the computational burden.

Compressed or sparse representations are not merely useful to reduce computation. Continual learning, for example, focuses on learning algorithms that operate on non-iid data [e.g. 15, 16, 17]. In this setup, training proceeds sequentially on a set of tasks. The system is expected to accelerate learning on subsequent tasks as well as using newly acquired knowledge to potentially improve on previous problems, all of this while maintaining low memory and computational footprint. This is difficult due to the well studied problem of catastrophic forgetting, where performance on previous tasks deteriorates rapidly when new knowledge is incorporated. Many approaches to this problem require identifying the underlying set of parameters needed to encode the solution to a task in order to freeze or protect them via some form of regularisation. In such a scenario, given constraints on model size, computation and memory, it is advantageous that each learned task occupies as little capacity as possible.

In both scenarios, typically, the share of capacity needed to encode the solution is determined by the learning process itself, with no explicit force to impose frugality. This is in contrast with earlier works on $L_0$ regularisation that explicitly restrict the learning process to result in sparse and compressible representations [18]. The focus of our work, similar to the $L_0$ literature, is on how to encourage the learning process to be frugal in terms of parameter usage. However instead of achieving this by adding an explicit penalty, we enhance the “rich get richer” nature of gradient descent. In particular we propose a new parameterisation that ensures steps taken by gradient descent are proportional to the magnitude of the parameters. In other words, parameters with larger magnitudes are allowed to adapt faster in order to represent the required features to solve the task, while smaller magnitude parameters are restricted, making it more likely that they will be irrelevant in representing the learned solution.

## 2 Powerpropagation

The desired proportionality of updates to weight magnitudes can be achieved in a surprisingly simple fashion: In the forward pass of a neural networks, raise the parameters of your model (element-wise) to the $\alpha$-th power (where $\alpha > 1$) while preserving the sign. It is easy to see that due to the chain rule of calculus the magnitude of the parameters (raised to $\alpha - 1$) will appear in the gradient computation, scaling the usual update. Therefore, small magnitude parameters receive smaller gradient updates, while larger magnitude parameters receive larger updates, leading to the aforementioned “rich get richer” phenomenon. This simple intuition leads to the name of our method.

More formally, we enforce sparsity through an implicit regulariser that results from reparameterising the model similar to [19, 20]. This line of research builds on previous work on matrix factorisation [e.g. 21, 22]. In [19] a parameterisation of the form $w = v \odot v - u \odot u$ is used to induce sparsity, where $\odot$ stands for element-wise multiplication and we need both $v$ and $u$ to be able to represent negative values, since the parameters are effectively squared. In our work we rely on a simpler formulation where $w = v|u|^{\alpha-1}$, for any arbitrary power $\alpha \geq 1$ (as compared to fixing $\alpha$ to 2), which, since we preserved the sign of $v$, can represent both negative and positive values. For $\alpha = 1$ this recovers the standard setting.

If we denote by $\Theta = \mathcal{R}^M$ the original parameter space or manifold embedded in $\mathcal{R}^M$, our reparameterisation can be understood through an invertible map $\Psi$, where its inverse $\Psi^{-1}$ projects $\theta \in \Theta$ into $\phi \in \Phi$, where $\Phi$ is a new parameter space also embedded in $\mathcal{R}^M$, i.e. $\Phi = \mathcal{R}^M$. The map is defined by applying the function $\Psi : \mathcal{R} \rightarrow \mathcal{R}, \Psi(x) = x|x|^{\alpha-1}$ element-wise, where by abuse of notation we refer to both the vector and element level function by $\Psi$. This new parameter space or manifold $\Phi$ has a curvature (or metric) that depends on the Jacobian of $\Psi$. Similar constructions have
been previously used in optimisation, as for example in the case of the widely known Mirror Descent algorithm \cite{23}, where the invertible map $\Psi$ is the link function. For deep learning, Natural Neural Networks \cite{24} rely on a reparameterisation of the model such that in the new parameter space, at least initially, the curvature is close to the identity matrix, making a gradient descent step similar to a second order step. Warp Gradient Descent \cite{25} relies on a meta-learning framework to learn a nonlinear projection of the parameters with a similar focus of improving efficiency of learning. In contrast, our focus is to construct a parameterisation that leads to an implicit regularisation towards sparse representation, following \cite{19}, rather then improving convergence.

Given the form of our mapping $\Psi$, in the new parameterisation the original weight $\theta_i$ will be replaced by $\phi_i$, where $\Psi(\phi_i) = \phi_i |\phi_i|^{\alpha-1} = \theta_i$ and $i$ indexes over the dimensionality of the parameters. Note that we apply this transformation only to the weights of a neural network, leaving other parameters untouched. Given the reparameterised loss $L(\cdot; \Psi(\phi))$, the gradient wrt. to $\phi$ becomes

$$
\frac{\partial L(\cdot; \Psi(\phi))}{\partial \phi} = \frac{\partial L}{\partial \Psi(\phi)} \frac{\partial \Psi(\phi)}{\partial \phi} = \frac{\partial L}{\partial \Psi(\phi)} \text{diag}(\alpha |\phi|^{\alpha-1}).
$$

Note that $\text{diag}$ indicates a diagonal matrix, and $|\phi|^{\alpha-1}$ indicates raising element-wise the entries of vector $|\phi|$ to the power $\alpha - 1$. $\frac{\partial L}{\partial \Psi(\phi)}$ is the derivative wrt. to the original weight $\theta = \phi |\phi|^{\alpha-1}$ which is the gradient in the original parameterisation of the model. This is additionally multiplied (element-wise) by the factor $\alpha |\phi_i|^{\alpha-1}$, which will scale the step taken proportionally to the magnitude of each entry. Finally, for clarity, this update is different from simply scaling the gradients in the original parameterisation by the magnitude of the parameter (raised at $\alpha - 1$), since the update is applied to $\phi$ not $\theta$, and is scaled by $\phi$. The update rule (1) has the following properties:

(i) 0 is a critical point for the dynamics of any weight $\phi_i$, if $\alpha > 1$. This is easy to see as $\frac{\partial L}{\partial \phi_i} = 0$ whenever $\phi_i = 0$ due to the $\alpha |\phi_i|^{\alpha-1}$ factor.

(ii) In addition, 0 is surrounded by a plateau and hence weights are less likely to change sign (gradients become vanishingly small in the neighbourhood of 0 due to the scaling). This should negatively affect initialisations that allow for both negative and positive values, but it might have bigger implications for biases.

(iii) This update is naturally obtained by the Backpropagation algorithm. This comes from the fact that Backpropagation implies applying the chain-rule from the output towards the variable of interest, and our reparameterisation simply adds another composition (step) in the chain before the variable of interest.

At this point the perceptive reader might be concerned about the effect of equation (1) on established practises in the training of deep neural networks. Firstly, an important aspect of reliable training is the initialisation (e.g. \cite{26, 27, 28}) or even normalisation layers such as batch-norm \cite{29} or layer-norm \cite{30}. We argue that our reparameterisation preserves all properties of typical initialisation schemes as it does not change the function. Specifically, let $\theta_i \sim p(\theta)$ where $p(\theta)$ is any distribution of choice. Then our reparameterisation involves initialising $\phi_i \leftarrow \text{sign}(\theta_i) \cdot \sqrt{|\theta_i|}$, ensuring the neural network and all intermediary layers are functionally the same. This implies that hidden activations will have similar variance and mean as in the original parameterisation, which is what initialisation and normalisation focus on.

Secondly, one valid question is the impact of modern optimisation algorithms (e.g. \cite{31, 32, 33}) on our reparameterisation. These approaches correct the gradient step by some approximation of the curvature, typically given by the square root of a running average of squared gradients. This quantity will be proportional (at least approximately) to $\text{diag}(\alpha |\phi|^{\alpha-1})$. This reflects the fact that our projection relies on making the space more curved and implicitly optimisation harder, which is what these optimisation algorithms aim to fix. Therefore, a naive use with Powerprop. would result in a reduction of the “rich get richer” effect. On the other hand, avoiding such optimisers completely can considerably harm convergence and performance. The reason for this is that they do not only

\footnote{To see this assume the weights do not change from iteration to iteration. Then each gradient is scaled by the same value $\text{diag}(\alpha |\phi|^{2(\alpha-1)})$ which factors out in the summation of gradients squared, hence the correction from the optimiser will undo this scaling. In practice $\phi$ changes over time, though slowly, hence approximately this will still hold.}
We provide code to reproduce the MNIST results (a) in the accompanying notebook.

correct for the curvature induced by our reparameterisation, but also the intrinsic curvature of the problem being solved. Removing the second effect can make optimisation very difficult. We provide empirical evidence of this effect in the Appendix.

To mitigate this issue and make Powerprop straightforward to use in any setting, we take inspiration from the target propagation literature [34, 35, 36, 37] which proposes an alternative way of thinking about the Backpropagation algorithm.

In our case, we pretend that the exponentiated parameters are the de-facto parameters of the model and compute an update wrt. to them using our optimiser of choice. The updated exponentiated parameters are then seen as a virtual target, and we take a gradient descent step on \( \phi \) towards these virtual targets. This will result in a descent step, which, while it relies on modern optimisers to correct for the curvature of the problem, does not correct for the curvature introduced by our parameterisation. Namely if we denote \( \text{optim} : \mathbb{R}^M \rightarrow \mathbb{R}^M \) as the function that implements the correction to the gradient done by some modern optimiser, our update becomes \( \Delta \phi = \text{optim}(\frac{\partial {\mathcal{L}}}{\partial \Psi(\phi)}) \text{diag}(\alpha |\phi|^{|\alpha|-1}) \).

The proof that our update is indeed correct follows the typical steps taken in the target propagation literature. From a first order Taylor expansion of \( \mathcal{L}(\phi - \eta \Delta \phi) \), we have that in order for \( \Delta \phi \) to reduce the loss, the following needs to hold: \( \langle \Delta \phi, \frac{\partial {\mathcal{L}}}{\partial \phi} \rangle > 0 \). But we know that \( \langle \text{optim}(\frac{\partial {\mathcal{L}}}{\partial \Psi(\phi)}), \frac{\partial {\mathcal{L}}}{\partial \Psi(\phi)} \rangle > 0 \) as this was a valid step on \( \Psi(\phi) \). Because \( \text{diag}(\alpha |\phi|^{|\alpha|-1}) \) is positive definite (diagonal matrix with all entries positive), we can multiply it on both sides, proving that \( \langle \Delta \phi, \frac{\partial {\mathcal{L}}}{\partial \phi} \rangle > 0 \). We provide more details in the Appendix. We will rely on this formulation in our empirical evaluation.

### 3 Effect on weight distribution and sparsification

At this point an empirical demonstration might illuminate the effect of equation (1) on model parameters and our ability to sparsify them. Throughout this work, we will present results from neural networks after the removal of low-magnitude weights. We prune such parameters by magnitude (i.e. \( \min |\theta| \)), following current best practice [e.g. 38, 39, 13]. This is based on a Taylor expansion argument [14] of a sparsely-parameterised function \( f(x, \theta_s) \) which we would like to approximate its dense counterpart \( f(x, \theta) : f(x, \theta_s) \approx f(\theta, x) + g^T(\theta_s - \theta) + \frac{1}{2}(\theta_s - \theta)^TH(\theta_s - \theta) + ... \) where \( g \) is the gradient vector and \( H \) the Hessian. As higher order derivatives are impractical to compute for modern networks, minimising the norm of \( (\theta_s - \theta) \) is a practical choice instead.

Following the experimental setup in [10] we study the effect of Powerpropagation at different powers \( \alpha \) relative to standard Backpropagation with otherwise identical settings. Figure 1a shows the effect of increasing sparsity on the layerwise magnitude-pruning setting for LeNet [40] on MNIST [41]. In both cases we notice a significant improvement over an otherwise equivalent baseline. While the choice of \( \alpha \) does influence results, all choices lead to an improvement\(^2\) in the MNIST setting. Where does this improvement come from? Figures 1c & 1d compare weights before and after training

\(^2\)For numerical stability reasons we typically suggest a choice of \( \alpha \in (1, 3) \) depending on the problem.
on MNIST at identical initialisation. We prune the network to 90% and compare the respective weight distributions. Three distinct differences are particularly noticeable: (i) Most importantly, weights initialised close to zero are significantly less likely to survive the pruning process when Powerpropagation is applied (see green Kernel density estimate). (ii) Powerpropagation leads to a heavy-tailed distribution of trained weights, as (1) amplifies the magnitude of such values. These observations are what we refer to as the “rich-get-richer” dynamic of Powerpropagation. (iii) Weights are less likely to change sign (see Figure 1d), as mentioned in Section 2.

One possible concern of (i) and (ii) are that Powerpropagation leads to training procedures where small weights cannot escape pruning, i.e. masks computed at initialisation and convergence are identical. This is undesirable as it is well established that pruning at initialisation is inferior [e.g. 11, 42, 43, 39]. To investigate this we plot the overlap between these masks at different pruning thresholds in Figure 1b. While overlap does increase with $\alpha$, at no point do we observe an inability of small weights to escape pruning, alleviating this concern. Code for this motivational example on MNIST is provided.  

4 Powerpropagation for Continual Learning

While algorithms for neural network sparsity are well established as a means to reduce training and inference time, we now formalise our argument that such advances can also lead to significant improvements in the continual learning setting: the sequential learning of tasks without forgetting. As this is an inherently resource constraint problem, we argue that many existing algorithms in the literature can be understood as implementing explicit or implicit forms of sparsity. One class of examples are based on weight-space regularisation [e.g. 17, 44] which can be understood as compressing the knowledge of a specific task to a small set of parameters that are forced to remain close to their optimal values. Experience Replay and Coreset approaches [e.g. 45, 46, 47] on the other hand compress data from previous tasks to optimal sparse subsets. The class of methods on which we will base the use of Powerpropagation for Continual Learning on implement gradient sparsity [e.g. 48, 49], i.e. they overcome catastrophic forgetting by explicitly masking gradients to parameters found to constitute the solution to previous tasks.

In particular, let us examine PackNet [48] as a representative of such algorithms. Its underlying principle is simple yet effective: Identify the subnetwork for each task through (iterative) pruning to a fixed budget, then fix the solution for each task by explicitly storing a mask at task switches and protect each such subnetwork by masking gradients from future tasks (using a backward Mask $M^b$). Given a pre-defined number of tasks $T$, PackNet reserves $1/T$ of the weights. Self-evidently, this procedure benefits from networks that maintain high performance at increased sparsity, which becomes particularly important for large $T$. Thus, the application of improved sparsity algorithms such as Powerpropagation are a natural choice. Moreover, PackNet has the attractive property of merely requiring the storage of a binary mask per task, which comes at a cost of 1 bit per parameter, in stark contrast to methods involving the expensive storage (or generative modelling) of data for each past task.

Nevertheless, the approach suffers from its assumption of a known number of maximum tasks $T$ and its possibly inefficient resource allocation: By reserving a fixed fraction of weights for each task, no distinction is made in terms of difficulty or relatedness to previous data. We overcome both issues through simple yet effective modifications resulting in a method we term EfficientPacknet (EPN), shown in Algorithm 1. The key steps common to both methods are (i) Task switch (Line 3), (ii) Training through gradient masking (Line 4), (iii) Pruning (Line 8), (iv) Updates to the backward mask needed to implement gradient sparsity.

Improvements of EPN upon PackNet are: (a) Lines 7-11: We propose a simple search over a range of sparsity rates $S = [s_1, \ldots, s_n]$, terminating the search once the sparse model’s performance falls short of a minimum accepted target performance $\gamma P$ (computed on a held-out validation set) or once a minimal acceptable sparsity is reached. While the choice of $\gamma$ may appear difficult at first, we argue that an maximal acceptable loss in performance is often a natural requirement of a practical engineering application. In addition, in cases where the sparsity rates are difficult to set, a computationally efficient binary search (for $\gamma P$) up to a fixed number of steps can be performed instead. Finally, in smaller models the cost of this step may be reduced further by instead using

https://github.com/deepmind/deepmind-research/tree/master/powerpropagation
an error estimate based on a Taylor expansion [50]. This helps overcome the inefficient resource allocation of PackNet. (b) **Line 8**: More subtly, we choose the mask for a certain task among all network parameters, including ones used by previous tasks, thus encouraging reuse to existing parameters. PackNet instead forces the use of a fixed number of new parameters, thus possibly requiring more parameters than needed. Thus, the fraction of newly masked weights per task is adaptive to task complexity. (c) **Line 13**: We re-initialise the weights as opposed to leaving them at zero, due to the critical point property (Section 2). While we could leave the weight at their previous value, we found this to lead to slightly worse performance.

Together, these changes make the algorithm more suitable for long sequences of tasks, as we will show in our experiments. In addition, they overcome the assumption of an a-priori known number of tasks $T$. Another beneficial property of the algorithm is that as the backward pass becomes increasingly sparse, the method becomes more computationally efficient with larger $T$. A possible concern for the method presented thus far is the requirement of known task ids at inference time, which are needed to select the correct mask for inputs. We present an algorithm to address this concern in the supplementary material.

### 5 Related work

Sparsity in Deep Learning has been an active research since at least the first wave of interest in neural networks following Backpropagation [51]. Early work was based on the Hessian [e.g. 52, 50, 53], an attractive but impractical approach in modern architectures. The popularity of magnitude-based pruning dates back to at least [54], whereas iterative pruning was first noted to be effective in [55]. Modern approaches are further categorised as Dense→Sparse or Sparse→Sparse methods, where the first category refers to the instantiation of a dense network that is sparsified throughout the training. Sparse→Sparse algorithms on the other hand maintain constant sparsity throughout, giving them a clear computational advantage.

Among Dense→Sparse methods, $L_0$ regularisation [18] uses non-negative stochastic gates to penalise non-zero weights during training. Variational Dropout [56] allows for unbounded dropout rates, leading to sparsity. Both are alternatives to magnitude-based pruning. Soft weight threshold reparameterisation [57] is based on learning layerwise pruning thresholds allowing non-uniform budgets across layers. Finally, the authors of [10] observe that re-training weights that previously survived pruning from their initial value can lead to sparsity at superior performance. Examples of Sparse→Sparse methods are Single Shot Network Pruning [58] where an initial mask is chosen according to the salience and kept fixed throughout. In addition, a key insight is to iteratively drop & grow weights while maintaining a fixed sparsity budget. Deep Rewiring for instance [59] augments SGD with a random walk. In addition, weights that are about to change sign are set to zero, activating other weights at random instead. Rigging the Lottery (RigL) [13] instead activates new weights by

---

**Algorithm 1: EfficientPackNet (EPN) + Powerpropagation.**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A^0_t \leftarrow 1 \lor \delta_t$ // Backward mask</td>
</tr>
<tr>
<td>2</td>
<td>$\phi_t \leftarrow \text{sign}(\theta_t) / \sqrt{</td>
</tr>
<tr>
<td>3</td>
<td>for $t \in [1, \ldots, T]$ do</td>
</tr>
<tr>
<td>4</td>
<td>$\phi_t \leftarrow \text{arg min}<em>\phi \mathcal{L}(X_t, y_t, \phi, M</em>{b})$ // Train on task $t$ with explicit gradient masking through $M_b$</td>
</tr>
<tr>
<td>5</td>
<td>$P_t \leftarrow \mathcal{L}(X_t, y_t, \phi) // Validation performance of dense model on task $t$</td>
</tr>
<tr>
<td>6</td>
<td>$t \leftarrow n$</td>
</tr>
<tr>
<td>7</td>
<td>do</td>
</tr>
<tr>
<td>8</td>
<td>$M^b_t = \begin{cases} 1 &amp; \text{if } i \in \text{TopK}(\phi_t, [s_t \cdot \text{dim}(\phi)]) \ 0 &amp; \text{otherwise} \end{cases}$ // Find new forward mask at sparsity $s_t$</td>
</tr>
<tr>
<td>9</td>
<td>$P_t \leftarrow \mathcal{L}(X_t, y_t, \phi \odot M^b_t) // Validation performance of sparse model</td>
</tr>
<tr>
<td>10</td>
<td>$t \leftarrow t - 1$</td>
</tr>
<tr>
<td>11</td>
<td>while $P_t &gt; \gamma p \land l \geq 1$;</td>
</tr>
<tr>
<td>12</td>
<td>$M_t^b \leftarrow \bigvee_{i=1}^l M^b_i // Update backward mask to protect all tasks 1, \ldots, t$</td>
</tr>
<tr>
<td>13</td>
<td>Re-initialise pruned weights</td>
</tr>
<tr>
<td>14</td>
<td>end</td>
</tr>
</tbody>
</table>
Figure 2: One-shot pruning results on CIFAR-10 (a) and ImageNet (b and c). Performance vs sparsity curves are shown in (a) and (b), highlighting significant improvements wrt. the baseline. This is due to Powerprop.’s effect on the weight distribution, which we show for the smallest 100k weights on ImageNet in (c). \( \alpha > 1 \) pushes the weight distribution towards zero, ensuring that more weights can be safely pruned. For CIFAR-10 we show mean and standard deviation over five runs. Repeated runs lead to almost identical results for ImageNet and are thus omitted.

highest magnitude gradients, whereas Top-KAST [14] maintains two set of masks for (sparse) forward and backward-passes. This also allows the authors to cut the computational budget for gradient computation. In addition, a weight-decay based exploration scheme is used to allow magnitude-based pruning to be used for both masks. Other notable works in this category are Sparse Evolutionary Training [12] and Dynamic sparse reparameterisation [39].

Powerpropagation is also related to Exponentiated Gradient Unnormalized [60] (Example 2) which is a reparametrisation equivalent to Powerpropagation and should be understood as a concurrent development. Indeed, the authors provide a preliminary experiment of the last layer of a convolutional network on MNIST, showing similar inherent sparsity results. We can view Powerprop. as a generalisation of this and related ideas [e.g. 19, 20, 21] with an explicit focus on modern network architectures, introducing necessary practical strategies and demonstrating the competitiveness of reparameterisation approaches at scale. In addition, the technique has not yet been used for Continual Learning.

While continual learning has been studied as early as [15, 16], the work on Elastic Weight Consolidation (EWC) [17] has recently started a resurgence of interest in the field. Contemporary approaches are subdivided into methods relying on regularisation [e.g. 44, 46, 47], data replay [e.g. 61, 62, 45, 63] or architectural approaches [e.g. 64, 65]. Works such as PackNet and SpaceNet [48, 49] have also introduced explicit gradient sparsity as a tool for more efficient continual learning. Other approaches include PathNets which [66] uses evolutionary training to find subnetworks within a larger model for individual tasks.

6 Experimental Evaluation

We now provide an experimental comparison of Powerpropagation to a variety of other techniques, both in the sparsity and continual learning settings. Throughout this section we will be guided by three key questions: (i) Can we provide experimental evidence for inherent sparsity? (ii) If so, can Powerprop. be successfully combined with existing sparsity techniques? (iii) Do improvements brought by Powerprop. translate to measurable advances in Continual Learning?

This section covers varying experimental protocols ranging from supervised image classification to generative modelling and reinforcement learning. Due to space constraints we focus primarily on the interpretation of the results and refer the interested reader to details and best hyperparameters in the Appendix.

6.1 Inherent Sparsity

Turning to question (i) first, let us start our investigation by comparing the performance of a pruned method without re-training. This is commonly known as the one-shot pruning setting and is thus a Dense \( \rightarrow \) Sparse method. If Powerprop. does indeed lead to inherently sparse networks its improvement should show most clearly in this setting. Figure 2 shows this comparison for image classification on the popular CIFAR-10 [67] and ImageNet [68] datasets using a smaller version of
We distinguish between three sub-settings a) Table 1 shows ResNet 50@ \{80\%, 90\%, 95\%\} sparsity, for which the largest number of baseline results are available. We also provide results using the Erdos-Renyi Kernel [13], a redistribution of layerwise sparsity subject to the same fixed overall budget. While this tends to lead to better results the computational training cost increases. c) Figure 3: Extreme sparsity at 95\% − 99\% thus testing methods at the limit of what is currently possible. All experiments are repeated 3 times to report mean and standard deviation. As Powerprop. introduces a negligible amount of extra FLOPs (over baseline methods) we only show such values in the extended training setting to provide a fair comparison to the setup in [13]. A longer discussion of a FLOP trade-off can be found in the work on TopKAST [14].

Examining the results, we note that Powerprop. improves both Iterative Pruning & TopKAST in all settings without any modification to the operation of those algorithms. In all settings a large array

<table>
<thead>
<tr>
<th>Method</th>
<th>80% Sparsity</th>
<th>90% Sparsity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter. Pruning</td>
<td>76.5 [0.84x]</td>
<td>75.2 [0.76x]</td>
</tr>
<tr>
<td>RigL (5x)</td>
<td>76.6±0.06 [1.14x]</td>
<td>75.7±0.06 [0.52x]</td>
</tr>
<tr>
<td>ERK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RigL (5x)</td>
<td>77.1±0.06 [2.09x]</td>
<td>76.4±0.05 [1.23x]</td>
</tr>
<tr>
<td>Powerprop. + TopKAST† (2x)</td>
<td>77.51±0.03 [1.21x]</td>
<td>76.94±0.10 [0.97x]</td>
</tr>
<tr>
<td>Powerprop. + TopKAST† (3x)</td>
<td>77.04±0.05 [1.81x]</td>
<td>77.16±0.19 [1.46x]</td>
</tr>
</tbody>
</table>

Table 2: Extended training. Numbers in square Brackets show FLOPs relative to training of a dense model for a single cycle.

We now address question (ii) by comparing Powerprop. when used in conjunction with and compared to exiting state-of-the-art algorithms on ImageNet. We orient ourselves primarily on the experimental setup in [13] & [14], both of which present techniques among the strongest in the literature. To obtain a Dense→Sparse method we combine Powerprop. with Iterative Pruning and combine TopKAST [14] with Powerprop. as our method of choice from the Sparse→Sparse literature.

AlexNet [3] and ResNet50 [4] respectively. In both situations we notice a marked improvement, up to 33\% for CIFAR and 18\% for ImageNet at specific sparsity rates. In Fig. 2c we show the density of the smallest 100K weights, in which we observe a notable peak around zero for \( \alpha > 1.0 \), providing strong evidence in favour of the inherent sparsity argument. Finally is worth noting that the choice of \( \alpha \) does influence the optimal learning rate schedule and best results were obtained after changes to the default schedule.

6.2 Advanced Pruning

We distinguish between three sub-settings a) Table 1 shows ResNet 50@ \{80\%, 90\%, 95\%\} sparsity, for which the largest number of baseline results are available. We also provide results using the Erdos-Renyi Kernel [13], a redistribution of layerwise sparsity subject to the same fixed overall budget. While this improves results, it increases the floating point operations (FLOPs) at test time. b) Table 2: Extended training, where we scale training steps and learning rate schedule by a factor of the usual 32K steps. While this tends to lead to better results the computational training cost increases. c) Figure 3: Extreme sparsity at 95\% − 99\% thus testing methods at the limit of what is currently possible. All experiments are repeated 3 times to report mean and standard deviation. As Powerprop. introduces a negligible amount of extra FLOPs (over baseline methods) we only show such values in the extended training setting to provide a fair comparison to the setup in [13]. A longer discussion of a FLOP trade-off can be found in the work on TopKAST [14].

Examining the results, we note that Powerprop. improves both Iterative Pruning & TopKAST in all settings without any modification to the operation of those algorithms. In all settings a large array
of other techniques are outperformed. To the best of our knowledge those results constitute a new state-of-the-art performance in all three settings. We emphasise that many of the methods we compare against are based on pruning weights by their magnitude and would thus also be likely to benefit from our method.

6.3 Continual Learning

6.3.1 Overcoming Catastrophic forgetting

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MNIST</th>
<th>notMNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>−258.40</td>
<td>−797.88</td>
</tr>
<tr>
<td>Laplace [69, 65]</td>
<td>−108.06</td>
<td>−225.54</td>
</tr>
<tr>
<td>EWC [17]</td>
<td>−105.78</td>
<td>−212.62</td>
</tr>
<tr>
<td>SI[44]</td>
<td>−111.47</td>
<td>−190.48</td>
</tr>
<tr>
<td>VCL [46]</td>
<td>−94.27</td>
<td>−187.34</td>
</tr>
<tr>
<td>Eff. PackNet (EPN)</td>
<td>−94.50</td>
<td>−177.08</td>
</tr>
<tr>
<td>Powerprop. + Laplace</td>
<td>−96.72</td>
<td>−196.87</td>
</tr>
<tr>
<td>Powerprop. + EWC</td>
<td>−95.79</td>
<td>−188.56</td>
</tr>
<tr>
<td>Powerprop. + EPN</td>
<td>−92.15</td>
<td>−174.54</td>
</tr>
</tbody>
</table>

Table 3: Results on the Continual generative modelling experiments. Shown is an importance-sampling estimate of the test log-likelihood.

We now turn to question (iii), studying the effectiveness of sparsity methods for Continual Learning. We first consider long task sequences on the Permuted MNIST benchmark, first popularised by [70].

While usually regarded to be an idealised and simple setting, we push its boundaries by drastically increasing the number of tasks, which was previously limited to 100 in [71] and 78 in [72], whereas we learn up to 300 in a single model. In all cases we use a network with a single output layer across all tasks, following the protocol by and comparing to [72]. Figure 4 shows the average accuracy and fraction of used weights. We significantly outperform baselines with an average acc. of ≈ 86% after 300 tasks, whereas the closest baseline (Laplace [65]) falls to 85.61% after merely 51 tasks. This also highlight the effect of α more noticeably, showing increased performance for higher α. Note also that the gradient of the used weights curve flattens, showcasing the dynamic allocation of larger number of parameters to tasks early in the sequence, whereas later tasks reuse a larger fraction of previous weights. We also attempted a comparison with standard PackNet, the model failed, as its default allocation of <0.4% of weights per task is insufficient for even a single MNIST task (see Figure 1a). The perspective reader might notice higher performance for considered baselines at small number of tasks. This is due to our choice of γ = 0.9 (see Section 4) which we optimise for long sequences. It is worth mentioning that α can be increased up to a certain limit at which point training for high number of tasks can become unstable. We will investigate this effect in future work.

Moving onto more complex settings, we use Powerprop. on the continual generative modelling experiment in [46] where 10 tasks are created by modelling a single character (using the MNIST and notMNIST datasets) with a variational autoencoder [73]. We report quantitative results in Table 3. In both cases we observe Powerprop. + EfficientPackNet outperforming the baselines. Interestingly, its effectiveness for Catastrophic Forgetting is not limited to PackNet. In particular, we provide results for the Elastic Weight Consolidation (EWC) method and its online version (Laplace) [17, 65] both of which are based on a Laplace approximation (using the diagonal Fisher information matrix). As Powerprop. results in inherently sparse networks, the entries of the diagonal Fisher corresponding to low magnitude weights are vanishingly small (i.e. the posterior distribution has high variance), resulting in virtually no regularisation of those parameters. We observe significant improvements with no other change to the algorithms.

Finally, we move onto Catastrophic Forgetting in Reinforcement Learning using six tasks from the recently published Continual World benchmark [74], a diverse set of realistic robotic manipulation tasks. This is arguably the most complex setting in our Continual Learning evaluation. Following the authors, we use Soft actor-critic [75], training for 1M steps with Adam [33] (relying on the formulation in Section 2) on each task while allowing 100k retrain steps for PackNet (acquiring no

---

Table 4: Reinforcement Learning Results on Continual World [74]. Error bars provide 90% confidence intervals.

<table>
<thead>
<tr>
<th>Method</th>
<th>Avg. success</th>
<th>Forgetting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine-tuning</td>
<td>0.00 [0.00, 0.00]</td>
<td>0.87 [0.84, 0.89]</td>
</tr>
<tr>
<td>A-GEM [62]</td>
<td>0.02 [0.01, 0.04]</td>
<td>0.89 [0.86, 0.91]</td>
</tr>
<tr>
<td>EWC [17]</td>
<td>0.66 [0.61, 0.71]</td>
<td>-0.01 [-0.04, 0.01]</td>
</tr>
<tr>
<td>MAS [78]</td>
<td>0.61 [0.56, 0.66]</td>
<td>0.07 [0.05, 0.10]</td>
</tr>
<tr>
<td>Perfect Memory</td>
<td>0.36 [0.32, 0.40]</td>
<td>-0.01 [-0.03, 0.01]</td>
</tr>
<tr>
<td>VCL [46]</td>
<td>0.52 [0.47, 0.57]</td>
<td>0.02 [0.04, 0.06]</td>
</tr>
<tr>
<td>PackNet [48]</td>
<td>0.80 [0.76, 0.84]</td>
<td>0.02 [0.04, 0.06]</td>
</tr>
<tr>
<td>Power. + EfficientPackNet $(\alpha=1.375)$</td>
<td>0.82 [0.77, 0.87]</td>
<td>0.01 [0.03, 0.01]</td>
</tr>
<tr>
<td>Power. + EfficientPackNet $(\alpha=1.5)$</td>
<td><strong>0.86 [0.82, 0.90]</strong></td>
<td><strong>0.00 [0.02, 0.02]</strong></td>
</tr>
</tbody>
</table>

additional data during retraining). We report the average success rate (a binary measure based on the distance to the goal) and a forgetting score measuring the difference of current performance to that obtained at the end of training (but importantly before pruning) on a certain task. Thus, if Powerprop. does improve results, this will show in the Forgetting metric. Results are shown in Table 4. Indeed, we notice that a higher $\alpha$ leads to a reduction of forgetting and hence an overall improvement over PackNet, resulting in superior performance. Finally, it is worth mentioning that (Efficient)PackNet still assume hard task boundaries that are known during training. However, it has previously been shown that it is possible to overcome these limitations using a changepoint detection algorithm [e.g. 76, 77].

7 Discussion

In this work we introduced Powerpropagation, demonstrating how its effect on gradients leads to inherently sparse networks. As we show in our empirical analyses, Powerprop. is easily combined with existing techniques, often merely requiring a few characters of code to implement. While we show its effect on only a few established algorithms, we hypothesise that a combination with other techniques could lead to further improvements. Bridging gaps between the fields of sparse networks and Continual Learning, we hope that our demonstration of its effect on Catastrophic Forgetting will inspire further interest in such ideas. In terms of future work, we believe that devising automated schemes for learning $\alpha$ may be worthwhile, making the use of our method even simpler. Particular attention ought to be paid to its interaction with the learning rate schedule, which we found worthwhile optimising for particular choices of $\alpha$. Finally, we also hope further work will bring additional theoretical insights into our method.

References


