Powerpropagation

Aditya Jain, Vaibhav Raj and Vedang Asgaonkar

Spring 2022

Contents

1 Problem Statement

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. In this project, we have implemented a weight-reparameterisation for neural networks called power propagation that leads to inherently sparse models through sparsifying effects of gradients achieved after reparameterisation. 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.
In the project, we explore two different settings:

- Enforcing model sparsity to speed up training in resource constrained settings. Power propagation leads to a higher density of weights near zero, and is combined with traditional weight pruning techniques
- Compressing task representation in models of fixed size in order to accomodate larger number of tasks at a fixed model capacity and overcoming catastrophic forgetting.

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. We also propose a modification to the PowerPropogation algorithm allowing for adaptive tuning of its hyperparameter.

2 Powerpropagation: The basic idea

In order to achieve the desired proportionality of weights, in the forward pass of a neural networks, we raise the parameters of the model (element-wise) to the α-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.

If we denote by $\Theta = \mathbb{R}^M$ the original parameter space or manifold embedded in $\mathbb{R}^M$, our reparameterisation can be understood through an invertible map Ψ, where its inverse Ψ^{-1} projects $\theta \in \Theta$ into $\phi \in \Phi$, where Φ is a new parameter space also embedded in $\mathbb{R}^M$, i.e. $\Phi = \mathbb{R}^M$. The map is defined by applying the function $\Phi : \mathbb{R} \to \mathbb{R}, \Psi(x) = x|x|^{\alpha-1}$ element-wise, where by abuse of notation we refer to both the vector and element level function by Ψ. This new parameter space or manifold Φ has a curvature (or metric) that depends on the Jacobian of Ψ.

Given the form of our mapping Ψ, in the new parameterisation the original weight θ_i will be replaced by ϕ_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 $\mathcal{L}(\cdot, \Psi(\phi))$, the gradient with respect to ϕ becomes

$$\frac{\partial \mathcal{L}(\cdot, \Psi(\phi))}{\partial \phi} = \frac{\partial \mathcal{L}}{\partial \Psi(\phi)} \frac{\partial \Psi(\phi)}{\partial \phi} = \frac{\partial \mathcal{L}}{\partial \Psi(\phi)} \cdot diag(\alpha|\phi|^{\circ\alpha-1})$$

$\frac{\partial \mathcal{L}}{\partial \Psi(\phi)}$ is the derivative with respect 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 ϕ not θ, and is scaled by ϕ.

We consider that the exponentiated parameters are the de-facto parameters of the model and compute an update with respect 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 ϕ 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 $optim : \mathbb{R}^M \to \mathbb{R}^M$ as the function that implements the correction to the gradient done by some modern optimiser, our update becomes $\Delta\phi = optim(\frac{\partial\mathcal{L}}{\partial\Psi(\phi)})diag(\alpha|\phi|^{\alpha-1})$.

3 Weight distribution and Sparsification

As noted in the previous section, powerpropagation leads to increased density of weights around zero. Combining powerpropagation with pruning algorithms leads to better performance on standard benchmarks.

We tested powerpropogation on dense and convolutional neural networks. Custom layers were created using PyTorch modifying existing Linear and Conv2D classes to provide powerpropogation. The following experiments were conducted:

3.1 ANN for MNIST classification

A simple one hidden layer fully connected neural network was created for multiclass MNIST classification. Powerprogogation was done with $\alpha = 1.25$ and $\alpha = 1.5$. Greater sparsity, measured by fraction of weights close to 0 was observed with increasing values of α

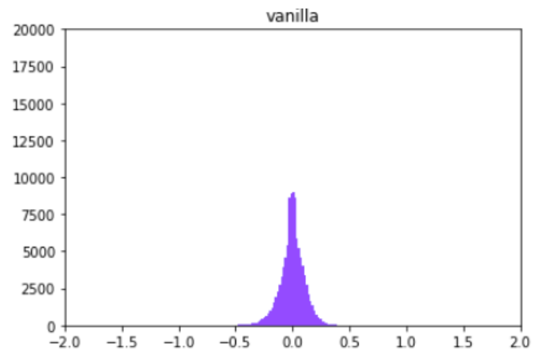


Figure 1: No powerprop. Accuracy: 0.9764

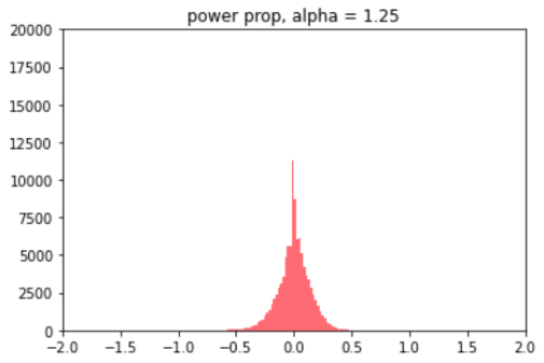


Figure 2: PowerPropogation. $\alpha = 1.25$. Accuracy: 0.9766

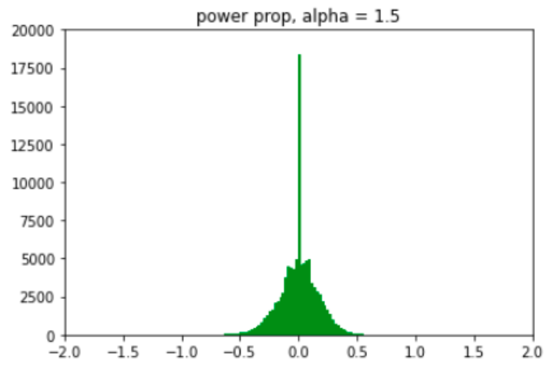


Figure 3: PowerPropogation. $\alpha = 1.5$. Accuracy: 0.9754

3.2 CNN for MNIST classification

We trained two neural networks with similar architectures, one with Powerpropagation (taking and the other without, and compared the results and weight distributions to obtain the following results:

The weight distributions for the three layers are as follows (blue bins represent the weight distribution with powerpropagation) :

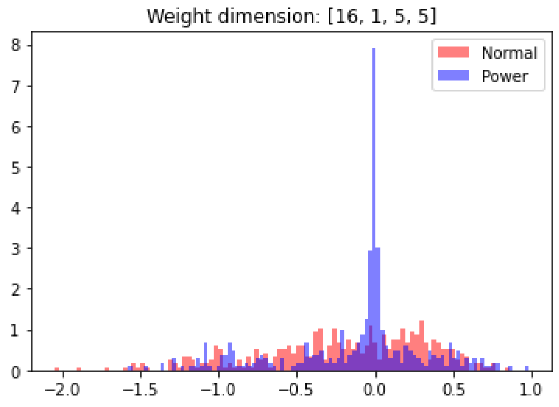


Figure 4: Layer 1

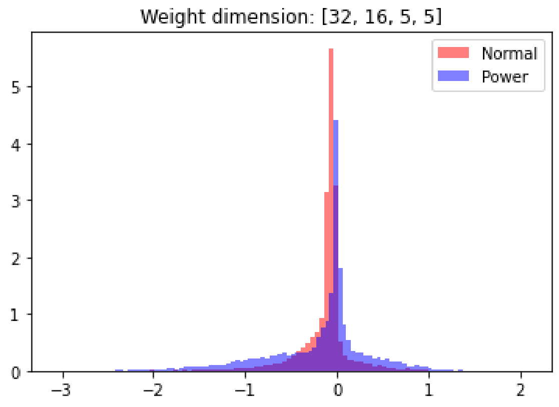


Figure 5: Layer 2

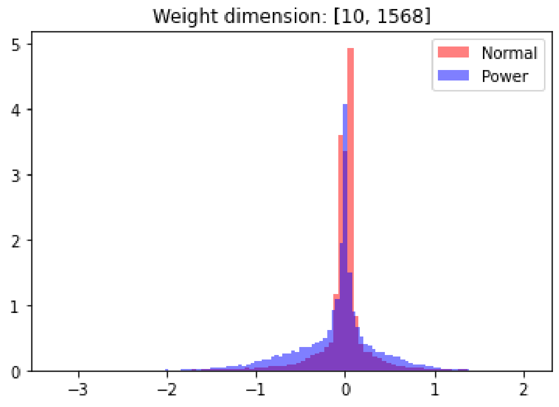


Figure 6: Layer 3

The test accuracy of the model which used powerpropagation was found to be 0.99 on the 10,000 image validation set, compared to an accuracy of 0.98 without powerpropagation.

3.3 CNN for CIFAR-10 classification

We trained two five-layer neural networks (again with and without powerpropagation), and recieved the following results:

The weight distributions for the five layers are as follows (blue bins represent the weight distribution with powerpropagation) :

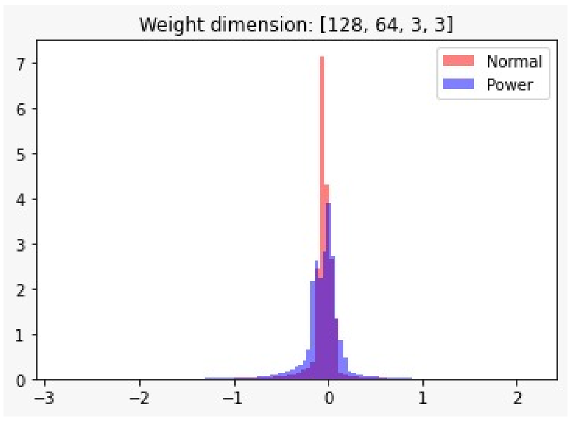


Figure 7: Layer 1

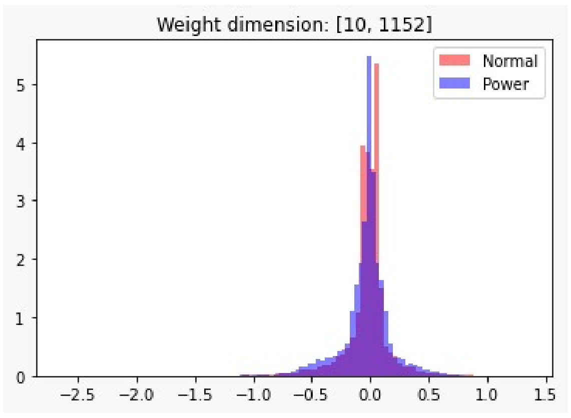


Figure 8: Layer 2

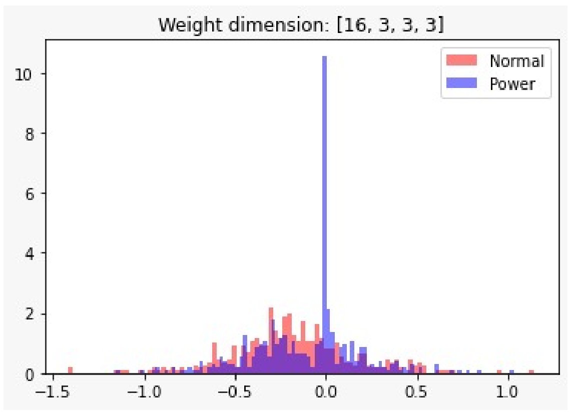


Figure 9: Layer 3

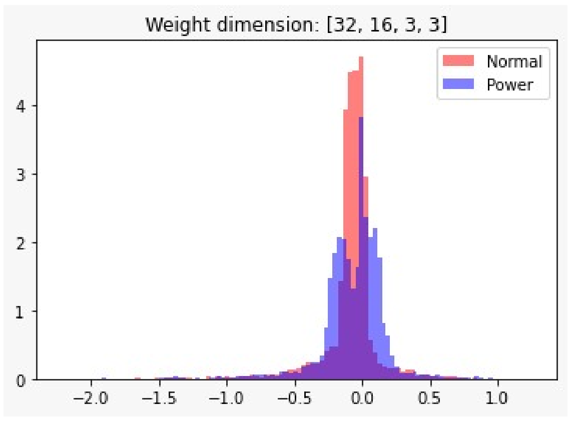


Figure 10: Layer 4

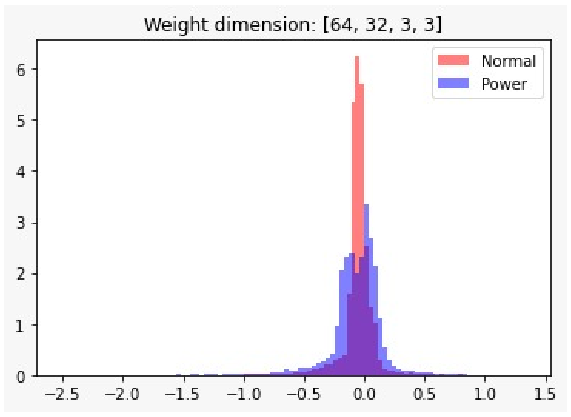


Figure 11: Layer 5

The test accuracy with powerpropagation was found to be 0.58, compared to 0.50 without powerpropagation.

3.4 Observations

A crucial observation from these experiments is that PowerPropogation induces sparsity without compromising on the accuracy of the model. Sparsity as measured by the number of weights close to zero increases with increase in value of the parameter α.

Caveat: For very large values of $\alpha > 1$, PowerPropogation is observed to cause weights to split away from 0 and reduce sparsity. This is primarily because large values of α tend to give large updates to weights pushing them away from 0. This is demonstrated by the following run of MNIST ANN with $\alpha = 4$

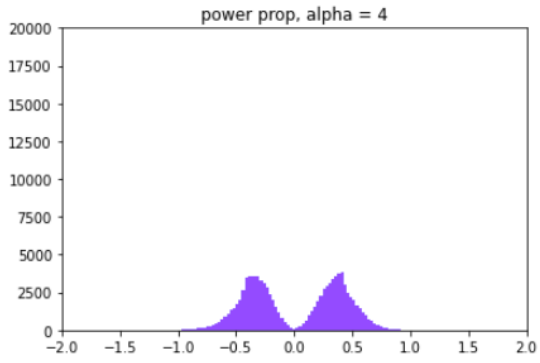
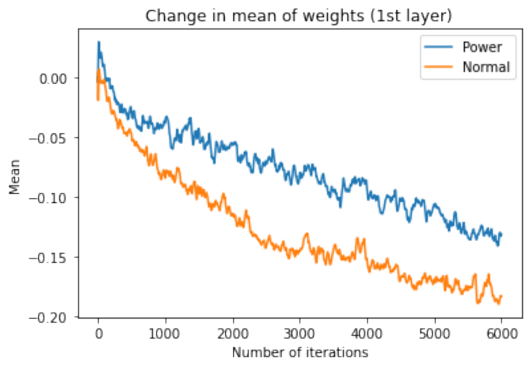
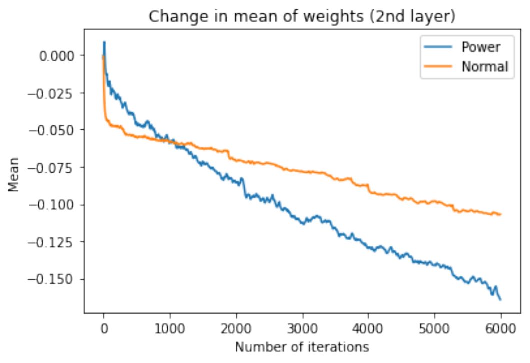


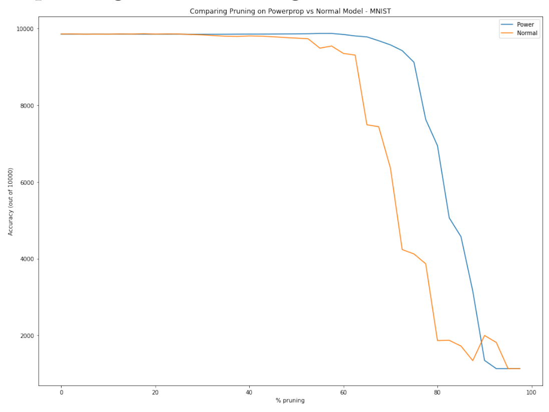
Figure 12: PowerPropogation $\alpha = 4$. Accuracy: 0.9746

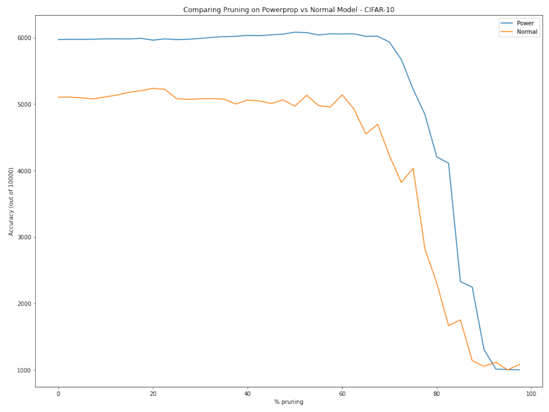
Another observation is that the effect of Powerpropragation is seen to dip as we go deeper in the network (closer to the output layer). This is demonstrated by the following shift in mean of weights - For layer 1, the mean of weights stayed close to 0 compared to the normal ConvNet while for layer 2, this pattern wasn't followed.





The performance of PowerPropogation is expected to be better than normal neural networks under pruning. The following observations on MNIST & CIFAR-10 bolster this intuition.





4 Overcoming catastrophic forgetting; EfficientPackNet

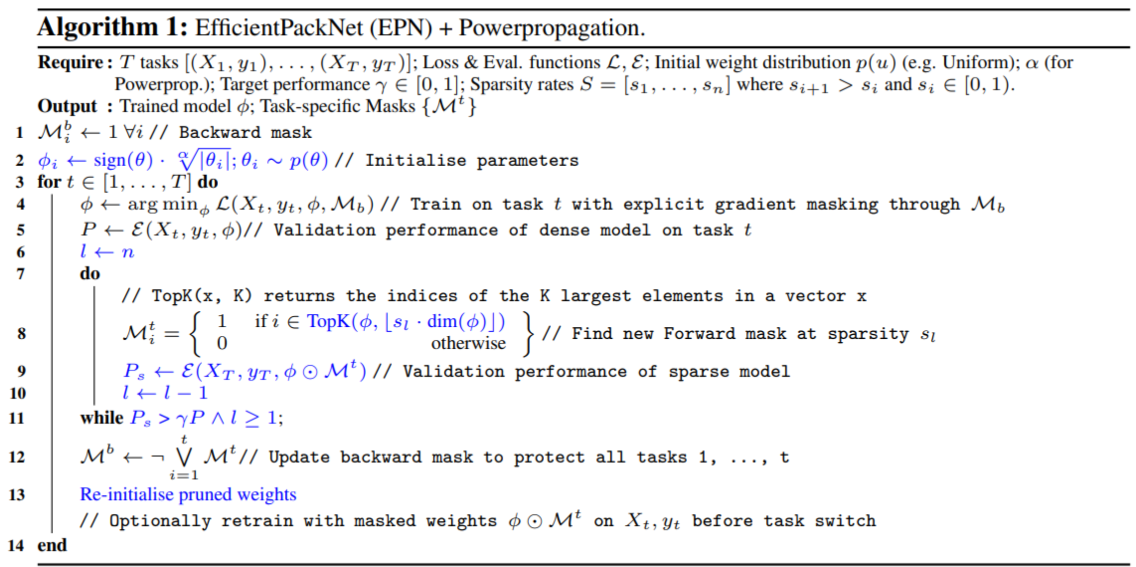
Continual learning is the sequential learning of tasks without forgetting. While training a single neural network on multiple tasks, the performance of the network on older tasks is found to drop sharply as it is trained on newer tasks. This is known as catastrophic forgetting, and is inherently a resource constraint problem. A way to overcome this problem is weight-space regularisation, 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.

PackNet is a method to overcome catastrophic forgetting by explicitly masking gradients to parameters found to constitute solution to previous tasks. 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 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.

Note that this approach has two issues:

- It assumes that the number of maximum tasks T is known, and
- the resource allocation may possibly be inefficient because a fixed number of weights is assigned for each task, irrespective of the complexity of the task or it's relatedness with previous data.

In the project we implement a modified version of this method called **EfficientPacknet** which overcomes both these issues.



Algorithm 1: EfficientPackNet (EPN) + Powerpropagation.

```
Require : $T$ tasks $[(X_1, y_1), \dots, (X_T, y_T)]$; Loss & Eval. functions $\mathcal{L}, \mathcal{E}$; Initial weight distribution $p(u)$ (e.g. Uniform); $\alpha$ (for
          Powerprop.); Target performance $\gamma \in [0, 1]$; Sparsity rates $S = [s_1, \dots, s_n]$ where $s_{i+1} > s_i$ and $s_i \in [0, 1)$.
Output  : Trained model $\phi$; Task-specific Masks $\{\mathcal{M}^t\}$
1  $\mathcal{M}_i^b \leftarrow 1 \; \forall i$ // Backward mask
2  $\phi_i \leftarrow \text{sign}(\theta) \cdot \sqrt[\alpha]{|\theta_i|}; \theta_i \sim p(\theta)$ // Initialise parameters
3  for $t \in [1, \dots, T]$ do
4      $\phi \leftarrow \arg\min_\phi \mathcal{L}(X_t, y_t, \phi, \mathcal{M}_b)$ // Train on task $t$ with explicit gradient masking through $\mathcal{M}_b$
5      $P \leftarrow \mathcal{E}(X_t, y_t, \phi)$// Validation performance of dense model on task $t$
6      $l \leftarrow n$
7      do
           // TopK(x, K) returns the indices of the K largest elements in a vector x
8          $\mathcal{M}_i^t = \left\{ \begin{array}{ll} 1 & \text{if } i \in \text{TopK}(\phi, \lfloor s_l \cdot \text{dim}(\phi) \rfloor) \\ 0 & \text{otherwise} \end{array} \right\}$ // Find new Forward mask at sparsity $s_l$
9          $P_s \leftarrow \mathcal{E}(X_T, y_T, \phi \odot \mathcal{M}^t)$ // Validation performance of sparse model
10         $l \leftarrow l - 1$
11     while $P_s > \gamma P \wedge l \geq 1$;
12     $\mathcal{M}^b \leftarrow \neg \bigvee_{i=1}^{t} \mathcal{M}^t$// Update backward mask to protect all tasks 1, ..., t
13     Re-initialise pruned weights
       // Optionally retrain with masked weights $\phi \odot \mathcal{M}^t$ on $X_t, y_t$ before task switch
14 end
```

The key steps common to PackNet and EfficientPackNet are: (i) Task switch (Line 3), (ii) Training through gradient masking (Line 4), (iii) Pruning (Line 8), (iv) Updates to backward mask needed to implement gradient sparsity.

Improvements of EPN upon PackNet are: (a) Lines 7-11: There is a simple search over a range of sparsity rates $= [s_1, ..., s_n]$, terminating the search once the sparse model's performance falls short of a minimum accepted target performance γP (computed on a held-out validation set) or once a minimal acceptable sparsity is reached. The choice of γ depends upon maximal acceptable loss in performance which 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 γP) up to a fixed number of steps can be performed instead. This helps overcome the inefficient resource allocation of PackNet. (b) Line 8: In EPN, 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. Leaving the weights at their previous values could lead to slighty worse performance.

We implemented the above algorithm with a single sparsity rate $S = 0.2$, and trained two identical neural networks taking value of α as 1 and 1.5 respectively. The number of tasks taken were 3, with the datasets being permuted MNIST.

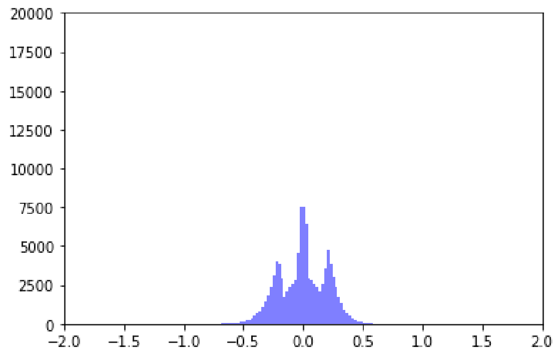


Figure 13: Weight distribution without power-propagation

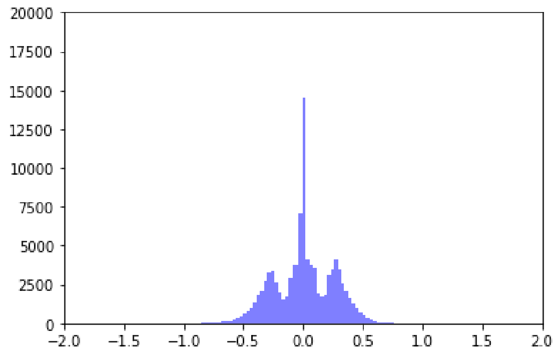


Figure 14: Weight distribution with powerpropagation ($\alpha = 1.5$)

Applying powerpropagation gave marginally better accuracy.

5 Adaptive Power Propogation

The paper suggests scope for improvement by adaptive tuning the value of α. This is necessitated by the fact that sparsity first increases with rise in α, but then tends to cause weights to spread away from 0 in two heaps. We propose a novel adaptive tuning algorithm which bears roots in control theory and the transmission control protocol. The salient features of the algorithm are as follows:

- The algorithm measures the sparsity by the fraction of weights within $(-\epsilon, \epsilon)$, where ϵ is a hyperparameter

- It is seeded by an initial value of α. The algorithm modifies the value of α after every 3 epochs of training.
- The values of α are modified based on a feedback mechanism mentioned below

Algorithm 1 Adaptive Tuning of α

```
if $sparsity(\alpha) \geq sparsity(\alpha_{old})$ then
    $\alpha_{old} \leftarrow \alpha$
    if $init$ then
        $\alpha \leftarrow 1 + cr * (\alpha - 1)$
    else
        $\alpha \leftarrow \alpha + cd$
    end if
else
    $\alpha_{old} \leftarrow \alpha$
    $\alpha \leftarrow 1 + (\alpha - 1)/cr$
    $init \leftarrow 0$
end if
```

- Here, the boolean variable *init* is set to 1 initially, indicating we are in the initial part of the loop. $cr = 1.5$ and $cd = 0.1$ are hyperparameters. The function *sparsity* measures sparsity of the weights for that particular α
- The algorithm begins by a geometric increase in α as long as the sparsity is increasing. When the sparsity reduces, it undergoes a geometric decrease till the sparsity increases again. Henceforth α undergoes a linear increase as long as sparsity is increasing, and a geometric decrease when the sparsity decreases.

This algorithm provides much better sparsity as compared to manually fixing the value of α. This was tested using a one hidden layer ANN on the MNIST data set. The final weights and the changes in α are shown below

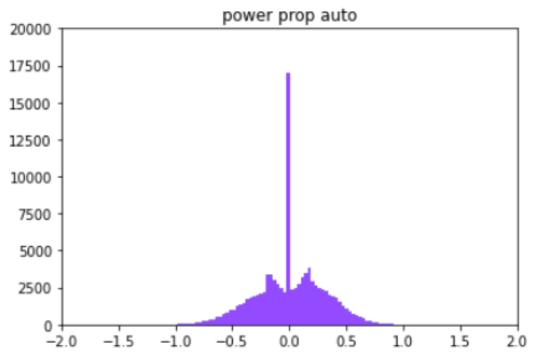


Figure 15: Adaptive powerprop.

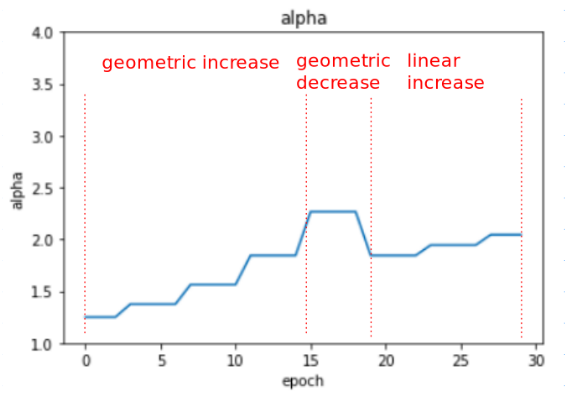


Figure 16: Variation of α with epochs

6 Conclusion

We implemented the PowerPropogation algorithm to induce sparsity in fully-connected and convolutional neural networks with marginal loss in accuracy. This can then be combined with pruning to reduce model size. Another application of sparsity is the packing of multiple tasks into a single neural network as demonstrated by our implementation of EfficientPackNet. We also propose modifications to the PowerPropogation algorithm by adaptively tuning the parameter α using a control-theoretic approach.

7 References

1. Powerpropagation: A sparsity inducing weight reparameterisation. *Jonathan Schwarz, Siddhant M. Jayakumar, Razvan Pascanu, Peter E. Latham, Yee Whye Teh*

2. Packnet: Adding multiple tasks to a single network by iterative pruning. *Arun Mallya, Svetlana Lazebnik*

3. RFC 5681: TCP Congestion Control. *M. Allman, V. Paxson, E. Blanton*