Applying Variants of Minimal Effort Backpropagation (meProp) on Feedforward Neural Networks

De Alba, Enrique edealba@stanford.edu
Kannan, Nithin nkannan@stanford.edu
Kim, Young Han yhkim99@stanford.edu

I. Introduction

Neural network training can often be slow, with the majority of training time spent on backpropagation. In July of this year, Wang *et al.* (2017) devised a technique called minimal effort backpropagation (meProp), which reduces the computational cost of backpropagation through neural networks by computing only the k most influential rows of the gradient for any hidden layer weight matrix. In their original work, they implemented and tested meProp on fully connected layers, and found that meProp not only offers a linear reduction in computational cost (depending on the value of k), but also marginally improves accuracy, likely as a result of reducing overfitting. Wang *et al.* produced another paper this September that extended meProp to convolutional neural networks. Besides these two papers, there is currently no other literature on meProp that we know of.

In our paper, we hope to experiment with a variety of different modifications of meProp. First, we hope to look at the effect of varying k during our backpropagation step, *e.g.* decreasing k over time as a further effort to reduce overfitting. Furthermore, we explore the possibility of different ways of selecting the '*most significant*' gradient vectors. Lastly, we try varying values of k across different layers, if our network happens to have several hidden layers. These variants are an effort to expand upon Wang *et al.*'s work, where k was constant across all epochs, examples, and layers. With our changes, we hope to maintain the speed of our neural network's training process, while also increasing the model's accuracy.

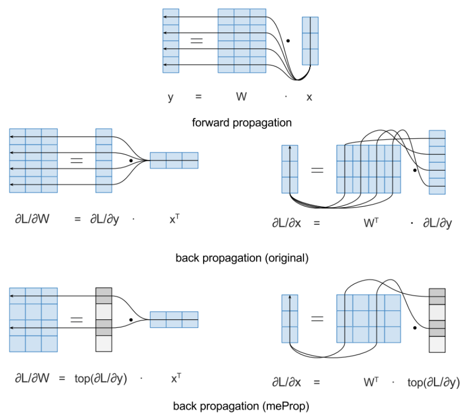


Fig. 1: The reduced meProp calculation

II. Introduction to Minimal Effort Backpropagation

For the sake of explanation, let's look at the effect of meProp between two arbitrary layers of the neural network:

- $W \in \mathbb{R}^{n \times m}$, the weight matrix
- $x \in \mathbb{R}^{m}$, our 'input' layer
- $y \in \mathbb{R}^{n}$, our 'output' layer

In backpropagation, one would normally calculate the gradient $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} \cdot x^T$. However, under meProp, we fix an integer $k < n$ and look at the k entries in $\frac{\partial L}{\partial y}$ of greatest magnitude, and only compute $\frac{\partial L}{\partial W_{ij}}$ for the corresponding values of i. That is, we assign

$$\frac{\partial L}{\partial W} \leftarrow \text{top}_k\left(\frac{\partial L}{\partial y}\right) \cdot x^T,$$

where $\text{top}_k(v)$ returns a vector v where all but the k entries of greatest magnitude are set to 0. One can interpret this as computing the gradient of the weights only for the k nodes in the 'output' layer that can contribute most to changing the loss, as indicated by Figure 1. It is easy to see why this would lead to a linear reduction in the runtime for backpropagation.

III. Our Motivations

As mentioned previously, meProp is a new technique, developed just this July. The original paper suggested multiple areas of further study and improvement. Below are some of the questions posed and answered by the paper:

- Q: Since dropout is so widely used in practice, it is important that meProp and dropout can be used simultaneously. How do their effects stack up?
  A: The paper empirically shows that meProp and dropout work together with no adversarial effect.
- Q: How would top-k meProp compare to randomized meProp?
  A: It turns out that while randomized meProp works decently well, reaching around 96 percent accuracy, top-k meProp works better.
- Q: How would having multiple layers (*i.e.* having a very deep net) affect meProp performance?
  A: While intuitively, it may seem that having multiple layers, each with meProp applied, would lead to a loss of information, the performance of meProp has been empirically shown to be independent of network depth.

While Wang *et al.* did address potential concerns with meProp, it seems as though there are even more areas requiring further exploration. Even with the simpler case of the fully connected neural network, we stumbled upon the following questions.

- What would the effect be of decreasing k as the neural network trains? Intuitively, since the neural network over several epochs learns which weights matter more than others, there should be fewer '*significant connections*' to pay attention to. Hence, gradually decreasing k could be even more effective at reducing overfitting.
- What would happen if we had different k values depending on the given training example? For different training examples, $\frac{\partial L}{\partial y}$ will have a different landscape of magnitudes, for which we shouldn't necessarily assign the same values of k. For example, a gradient vector $\frac{\partial L}{\partial y} = [2.0, 0.1, 0.1, 0.1, 2.0, 0.1]$ should be probably be prescribed a k value of 2, whereas a value of 4 may be more appropriate for $\frac{\partial L}{\partial y} = [3.0, 3.0, 0.1, 3.0, 0.1, 3.0]$.
- How can we optimize k between layers? Currently, if our neural network has several hidden layers, meProp still passes the same k value throughout the whole network. It deserves some attention as to whether we should vary k depending on the layer. For example, we could define k to be a fixed fraction of the layer dimension.

These questions correspond to the three experiments we ended up performing, which we delve into further in the following sections.

IV. Data/Model

For data, we use the MNIST handwritten digit dataset consisting of 60,000 examples, which we divide into a training set of 50,000 and a dev set of 10,000. This is one of the datasets tested by Wang *et al.*, so performing our variants on the same dataset allows for more straightforward and reliable comparisons. Our evaluation metric for accuracy is the percentage of correctly classified images in the dev set, and we evaluate this after each epoch.

Our neural network had an input layer of dimension 784 (the number of pixels in the image), one hidden layer of dimension 500 and a sigmoid activation function, an output layer of dimension 10 (the number of digits) and softmax activation, and our cost was the cross-entropy loss. We performed mini-batch gradient descent over 15 epochs with a mini-batch size of 500 and randomly shuffled the training set between epochs (for the first experiment, the number of epochs was not fixed). We also include L_2 regularization.

V. Annealing k with Learning Rate

Intuitively, the neural network over several epochs learns the significance of weights, so we look at the possibility of decreasing k as it trains. A common technique used in deep learning training is to anneal the learning rate after an increase in dev loss, which is often effective at optimizing the rate of gradient descent. Mirroring k to do the same could garner similar results and would also likely be more effective than prescribing a predefined schedule for k, for the latter would not be listening to the model as it trains. Therefore, we compared two runs of annealed training with meProp: one where we anneal the learning rate (exponentially) after a spike in dev loss, and one where where we anneal the value of k (linearly) after a spike in dev loss. Each run stopped after annealing three times. The reason for the linear decrease in k is because k is already a relatively small integer. We tried several trials, all of which shared similar results; one such run in displayed below in Figure 2.

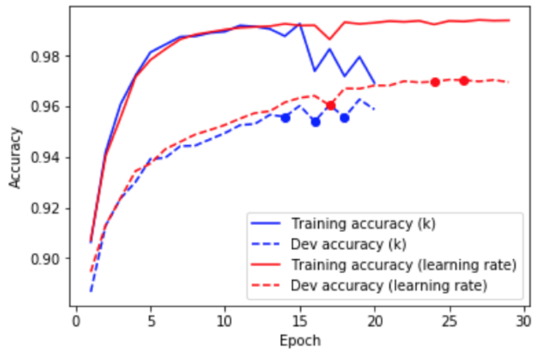


Fig. 2: Test accuracies for different variations of annealed k and learning rate

It seems that annealing k is not as effective as annealing the learning rate, as the training accuracy performs worse after annealment and the epochs of annealment are clustered closer together, suggesting that the reductions in the value of k are not leading to room for more descent.

VI. Norm-Limited k Selection

It may be optimal to assign different values of k for different training examples. So, we instead fix a threshold $0 < \tau < 1$ and assign for the i^{th} training example

$$k^{(i)} = \min\left\{k : \left\|\text{top}_k\left(\frac{dL}{dy^{(i)}}\right)\right\|_1 > \tau\left\|\frac{dL}{dy^{(i)}}\right\|_1\right\}.$$

That is, $k^{(i)}$ is however many elements of $\frac{dL}{dy^{(i)}}$ we need to include to capture at least τ of its L1 norm. Our hope with norm-limited k selection was to pass components of the gradients that were numerically significant, rather than using a fixed k throughout a layer. Pictured below in Figure 3 are accuracies of norm-limited meProp backpropagation with different values of τ.

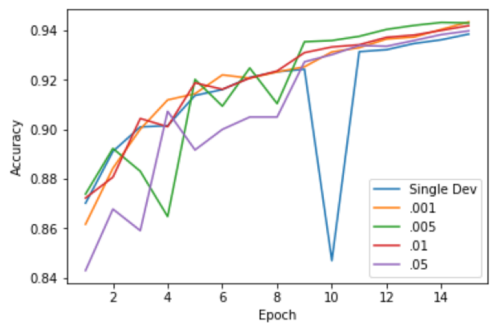


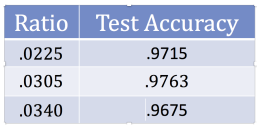
Fig. 3: Convergence of norm-limited meProp accuracies

Norm-limited k selection did not seem to improve the standard meProp algorithm's accuracy. Regardless of the value of τ, all models converged within error to the test accuracy of the neural network without meProp implemented. This may indicate that trying to fine-tune the efficacy of our gradients to the example level is too insignificant in the larger process of training, in contrast to more macroscopic characteristics of backpropagation, such as the dimension of a hidden layer. Furthermore, the failure of the technique may be a convenient finding for practice anyways, as having different values of $k^{(i)}$ may hinder the vectorization process and slow down training.

It's possible that we didn't test sufficiently high values of τ. In our future research section, we explore an alternate way of implementing norm-limited k selection to disregard noisy training data and select an optimal k.

VII. Changing k for Several Hidden Layers

For the following experiments, we simply added in between our existing hidden layer and output layer, a second layer of dimension 300 with a sigmoid activation function. Our first approach was setting k for both layers to be a certain fixed fraction of our hidden layer size. Our intuition for this approach was to maximize the effect of meProp at each layer by preventing meProp from removing too few or too many of the gradients at any particular layer.



Ratio	Test Accuracy
.0225	.9715
.0305	.9763
.0340	.9675

Fig. 4: Some different local optima for k ratios

The original paper describing meProp explained how they would get better results if k were some value between 1% to 4% of the gradient vector dimension, so we tested 30 ratio values between these two values. After repeated trials, we saw that there are discrete values for which k yields better testing accuracies that don't appear to follow any discernible pattern, *i.e.* k is highly stochastic; several of these values are showcased in Figure 4. Treating this ratio as a hyper-parameter may prove to be a useful modification to meProp.

Our second idea was to modify values of k depending on how deep into our neural network we were. We considered varying k between the two hidden layers, H_1 and H_2. For the first hidden layer we had k span between 4 and 20 ($\sim 0.8\% - 4\%$ of H_1's dimension) while we had k span between 2 and

15 ($\sim 0.67\% - 5\%$ of H_2's dimension) for the second hidden layer. Looking at the heat map (Figure 5) of this experiment we have brighter yellow spots corresponding to relatively higher average testing accuracies, while darker spots correspond to lower testing accuracies. The brightest spots in the map occur when k is 2 for H_2 and k is 10, 12, and 14 respectively for H_1. This highlights a trend that brighter spots corresponding to higher testing accuracies tend to occur when k is low $(0.6\% - 1.3\%)$ for the second hidden layer and k is between 6 and 13 $(1.2\% - 2.6\%)$ for the first hidden layer. This seems to suggest that meProp is more effective or more crucial on earlier layers of our network.

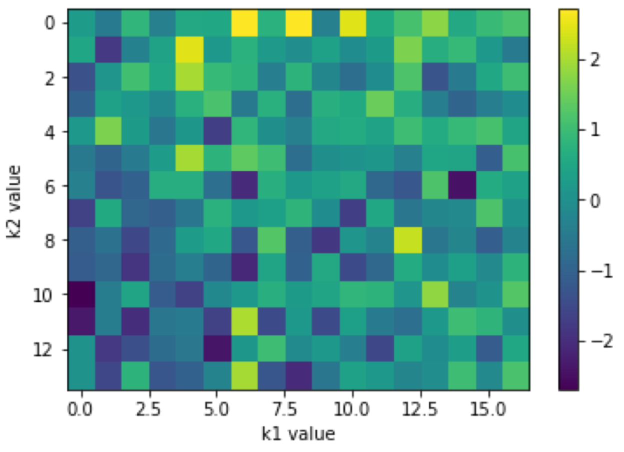


Fig. 5: Heat map of testing accuracies for varying k across layers

VIII. Future Research

In our norm-limited meProp experiments, it appears that insignificant training examples were being mishandled in our modified meProp algorithm. Since we had τ a fixed constant for all training examples, $\frac{dL}{dy^{(i)}}$ would be more likely to pick up on insignificant components. For example, suppose that $\frac{dL}{dy^{(1)}} = (0, 5, -.3, 0, 5)$ and $\frac{dL}{dy^{(2)}} = (0.1, .05, -.3, .1, .05)$. Our algorithm would take into account a larger fraction of the second gradient vectors at different values of τ. Intuitively, more values in $\frac{dL}{dy^{(2)}}$ should be ignored, since the data is probably noise. To account for this, a different and likely more effective way of choosing different significant components would be having the number of such components per training example depend not on the norm of $\frac{dL}{dy^{(i)}}$ but the distribution of magnitudes of entries in $\frac{dL}{dy^{(i)}}$ for all examples in a mini-batch. For example, if B denotes mini-batch size, then choosing the top kB entries of the vectors $\frac{dL}{dy^{(1)}}, \dots, \frac{dL}{dy^{(B)}}$ will likely choose more effective components than our initial norm-limited algorithm.

As mentioned earlier, all backpropagation updates, including those with meProp, included L2 regularization. However, it is unclear whether meProp interacts positively with regularization, or whether it is influenced by regularizaton at all. For example, if we lower the value of k, should we then also lower the value of λ associated with regularization? This is a potential avenue of exploration, and if regularization happens to not be as effective, it is worthy to explore other forms of optimizing gradient descent, such as adding momentum, or using Adam/Adagrad optimization methods.

Another potential future experiment to perform would be to more carefully evaluate the runtime of the norm-limited algorithm, for it may increase the computational cost, though we would anticipate the difference to be slight. Comparing the speedup/accuracy tradeoff would be another interesting route to study for these modified meProp algorithms. Furthermore, as all of these experiment were performed on a feedforward neural network, it is worthwhile to consider generalizing meProp and its modifications to sparsify backpropagation for other models, such as an RNN.

IX. Contributions

Each of the three members contributed similar amounts of work. For preliminary experiments and poster/paper writing, all three members worked together on a single device. The three experiments were split among the three members (Annealing k: Young Han Kim, Norm-Limited: Nithin Kannan, Changing k for Several Layers: Enrique De Alba), but we all helped each other out with the three experiments.

References

[1] Wang et al. "meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting" arXiv:1706.06197 (2017)

[2] Wei et al. "Minimal Effort Back Propagation for Convolutional Neural Networks" arXiv:1709.05804 (2017)

[3] LeCun, Yann and Cortes, Corinna. "MNIST handwritten digit database." (2010):