OPT2023: 15th Annual Workshop on Optimization for Machine Learning

How to Guess a Gradient

Utkarsh Singhal* S.UTKARSH@BERKELEY.EDU
UC Berkeley

Brian Cheung*
Kartik Chandra*
Jonathan Ragan-Kelley
Joshua B. Tenenbaum
Tomaso A. Poggio
MIT

Stella X. Yu
University of Michigan; UC Berkeley

Abstract

What can you say about the gradient of a neural network without *computing a loss* or *knowing the label?* This may sound like a strange question: surely the answer is "very little." However, in this paper, we show that gradients are more structured than previously thought. They lie in a predictable low-dimensional subspace that depends on the network architecture and incoming features.

Exploiting this structure can significantly improve gradient-free optimization schemes based on directional derivatives, which until now have struggled to scale beyond small networks trained on MNIST. We study how to narrow the gap in optimization performance between methods that calculate exact gradients and those that use directional derivatives, demonstrate new phenomena that occur when using these methods, and highlight new challenges in scaling these methods.

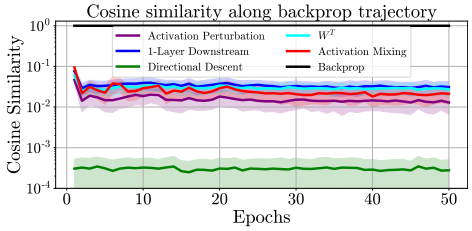
1. Introduction

Researchers have wondered for decades if neural networks can be optimized without using backpropagation to differentiate the loss function. One tantalizing possibility, first articulated by Polyak [15] and Spall [19], is to choose a *random direction* y, compute the *directional derivative* of the objective in the direction y, and take a step in the direction y scaled by the directional derivative, $(y \cdot \nabla L)$. For a neural network with weights w, this step is an elegant unbiased estimator of the true gradient: $w_{t+1} = w_t - \alpha(y \cdot \nabla L)y$. This method has recently seen renewed interest because directional derivatives can be computed very efficiently using forward-mode differentiation, which does not incur the immense memory cost of backpropagation [1, 3, 18].

Unfortunately, as the dimensionality of the optimization problem increases, so does the variance in the estimator, which in turn inhibits convergence. If the network has N parameters, the cosine similarity between the guess and true gradient falls to $O(\frac{1}{\sqrt{N}})$. For neural networks with billions of parameters, this guess is nearly orthogonal to the true gradient and thus impractical.

For an N-dimensional problem, Nesterov and Spokoiny [13] show that this method incurs an $O(N)$ slowdown over ordinary gradient descent. Belouze [2] provides further theoretical and em-

* These authors contributed equally to this work.



Method	Cosine Similarity	1-step effectiveness
Backprop (Oracle)	1	1
Directional Descent	0.0003 ± 0.00023	$1 \times 10^{-6} \pm 1 \times 10^{-6}$
Activation Perturbation	0.016 ± 0.0083	$6.9 \times 10^{-4} \pm 8.4 \times 10^{-4}$
Activation Mixing	0.025 ± 0.014	$3.4 \times 10^{-3} \pm 5.5 \times 10^{-3}$
W^T	0.030 ± 0.010	$1.7 \times 10^{-3} \pm 1.8 \times 10^{-3}$
1 Layer Downstream	0.034 ± 0.013	$2.7 \times 10^{-3} \pm 2.8 \times 10^{-3}$

Figure 1: **(left)** Guessed gradient cosine similarity for a 6-layer, 1024-wide MLP being trained on CIFAR10 using backpropagation. We track each method's cosine similarity along the backprop trajectory, and tabulate the average in the table on the right. Compared to directional descent, our proposed methods like W^T achieve approximately $100\times$ larger average cosine similarity. **(right)** We also tabulate the average cosine similarity as well as the loss reduction for 1 step (relative to backprop). Our methods reduce the loss more than $1000\times$ more for a single batch.

pirical evidence that the "curse of dimensionality" prevents this method from scaling. Similarly, Chandra [3] is able to train a small MLP on MNIST but not a ResNet on the CIFAR-10 dataset.

Can we do any better by guessing more intelligently? At first glance, isotropically random guessing appears to be the best we can do. But the intrinsic dimension of gradients is often much lower than N [9], which suggests that it might be possible to do better. In this paper, we observe that the topology and activations of the neural network heavily constrain the gradient even before a loss is computed—that is, we know information about the gradient before we even see the label. Thus, we ask: *How can we use knowledge about the network architecture and incoming features to make better gradient guesses?*

By carefully analyzing the structure of neural network gradients, we show it is possible to produce guesses with dramatically higher cosine similarity to the true gradient, even for networks with millions of parameters. We then compare these cosine similarities to those of stochastic gradient descent (SGD) to understand what cosine similarities are required for neural network optimization. Next, we analyze the variance and optimization properties of these guesses to highlight their improved convergence, and study limitations such as bias. Finally, we demonstrate a "*self-sharpening*" phenomenon, where the training dynamics induced by these guesses make it easier to guess the gradient over time. This phenomenon leads to $> 95\%$ training accuracy on CIFAR10 *without backpropagation*. Nonetheless, these advances come with some important limitations, which we also discuss — while the methods outlined in our work provide theoretical advances in our understanding of gradient structure, they are not yet ready for practical use.

2. Methods

In this section, we describe the proposed methods for narrowing the guess space. We begin by describing architecture-based constraints and then describe constraints based on knowledge about the relationship between gradients and activations.

2.1. Architecture-aware gradient guessing

Suppose we optimize an MLP with weights $W_1, W_2, \ldots, W_k$ using ReLU activation functions. At some layer i, we take as input some incoming activations x_i, compute the "pre-activations" $s_i = W_i x_i$, and then compute the "post-activations" $x_{i+1} = \text{ReLU}(s_i)$. We then pass x_{i+1} on to the

next layer, ultimately computing some loss $L = \ell(s_k)$. Finally, we wish to compute or approximate $\partial L/\partial W_i$ to train that layer. The current state-of-the-art method is to "guess" the unknown $\partial L/\partial W_i$ uniformly at random via a spherically-symmetric Gaussian. Can we do better?

Let us locally "unfold" the computation to see if there is any additional information we can exploit to narrow the space of our guessing. Recall that the first step is to multiply the incoming x_i by W_i to get the pre-activations $s_i = W_i x_i$. If we apply the chain rule at that point, we learn an important fact: $\partial L/\partial W_i$ is the outer product of the (unknown) future gradient and the (known) incoming activations x_i.

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial s_i} \cdot \frac{\partial s_i}{\partial W_i} = \frac{\partial L}{\partial s_i} \cdot \frac{\partial W_i x_i}{\partial W_i} = \frac{\partial L}{\partial s_i} \cdot x_i^\top \tag{1}$$

Notice in particular that $\partial L/\partial s_i$ is of significantly lower dimension than $\partial L/\partial W_i$. This leads us to our **first insight**: we can "guess" the low-dimensional $\partial L/\partial s_i$ and use the known $x_i^\top$ to compute a much lower-variance guess for the high-dimensional $\partial L/\partial W_i$. Note that for a neural network with width K, each weight has $K \times K = K^2$ parameters, and we have reduced the guessing space from $O(K^2)$ to $O(K)$. Practically, for neural networks with millions of parameters, this means guessing in a few thousand dimensions. This guess consists of perturbations of pre-activations (s_i), similar to the work of [16].

Let us keep unfolding. The next step is to take the ReLU of s_i to obtain x_{i+1}:

$$\frac{\partial L}{\partial s_i} = \frac{\partial L}{\partial x_{i+1}} \cdot \frac{\partial x_{i+1}}{\partial s_i} = \frac{\partial L}{\partial x_{i+1}} \cdot \frac{\partial \text{ReLU}(s_i)}{\partial s_i} \tag{2}$$

Our **second insight** is that by the very nature of ReLU activations, $\partial \text{ReLU}(s_i)/\partial s_i$ will typically be a sparse diagonal matrix, which will "zero out" many entries of the incoming gradient. This suggests that we should actually "guess" only the surviving entries of $\partial L/\partial x_{i+1}$, as determined by that sparse matrix (known at guess-time). This further decreases the dimensionality of our guessing space and, consequently, the variance of our guesses.

Let us unfold one last time, looking into the *next* weight matrix W_{i+1}. Again, we apply the chain rule, now at s_{i+1}:

$$\frac{\partial L}{\partial x_{i+1}} = \frac{\partial L}{\partial s_{i+1}} \cdot \frac{\partial s_{i+1}}{\partial x_{i+1}} = \frac{\partial L}{\partial s_{i+1}} \cdot \frac{\partial W_{i+1} x_{i+1}}{\partial x_{i+1}} = \frac{\partial L}{\partial s_{i+1}} \cdot W_{i+1} \tag{3}$$

As before, the future gradient $\partial L/\partial s_{i+1}$ is unknown and must be guessed. But we know that it will immediately be multiplied by $W_{i+1}^\top$. While this does not necessarily give a "hard" constraint on our guess, our **third insight** is that $W_{i+1}^\top$ often effectively has low rank. We can constrain our guesses to lie in the image of $W_{i+1}^\top$ by multiplying our guess with it to further lower the dimensionality of our guessing space. To summarize, we know that

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial s_{i+1}} W_{(i+1)} \cdot \frac{\partial \text{ReLU}(s_i)}{\partial s_i} \cdot x_i^\top$$

At "guess time" all of these quantities are known except for $\partial L/\partial s_{i+1}$, which we guess as random normal with zero mean and unit variance. We then apply a series of constraints to mould it into a more effective guess for $\partial L/\partial W_i$. We refer to the combination of these methods as "W^T".

Partial backpropagation: The previous approach incorporates local architecture information into the gradient guess. As a more general approach, we can consider guessing the gradient for some

neurons x_{i+l} which are l layers downstream of the current layer, and backpropagating through the intermediate portion of the graph: $\frac{\partial L}{\partial W_i} = \underbrace{\frac{\partial L}{\partial x_{i+l}}}_{\text{guess here}} \cdot \frac{\partial x_{i+l}}{\partial s_i} \cdot x_i^\top$. This approach requires storing the intermediate activations for the l layers, and in the full limit, is equivalent to regular backpropagation but with a random error vector. In our experiments, we find that using more than $l = 1$ has diminishing returns, so we stick to $l = 1$.

2.2. Feature-aware gradient guessing

We unroll SGD update steps and show that activations and gradients approximately lie in the same subspace. We visualize this phenomenon in Figure 6. This allows us to use a random mixture of activations x_{k+1} to produce good gradient guesses.

Consider the downstream weight matrix W_{k+1} being updated iteratively with SGD with a learning rate η. Then at timestep t:

$$W_{k+1}[t] = W_{k+1}[0] + \sum_{i=1}^{t-1} \Delta W_{k+1}[i] = W_{k+1}[0] + \eta \sum_{i=1}^{t-1} \frac{\partial L}{\partial s_{k+1}}[i] x_{k+1}^T[i]$$

. The term $\frac{\partial L}{\partial x_{k+1}}[t]$ can be expanded to:

$$\frac{\partial L}{\partial x_{k+1}}[t] = \frac{\partial L}{\partial s_{k+1}}[t] W_{k+1}[t] = \frac{\partial L}{\partial s_{k+1}}[t] W_{k+1}[0] + \eta \sum_{i=1}^{t-1} \beta_{k+1}[t,i] x_{k+1}^T[i] \approx \boxed{\eta \sum_{i=1}^{t-1} \beta_k[t,i] x_{k+1}^T[i]}$$

where we ignore the first term (weight at initialization), and where $\beta_{k+1}[t,i] = \frac{\partial L}{\partial s_k}[t]^T \frac{\partial L}{\partial s_k}[i]$ measures the similarity of s_k gradients at timesteps t and i. We thus see that the desired gradient approximately lies in the subspace of previously generated activations $x_{k+1}[i]$. Our experiments confirm that the activation subspace is often well-aligned with the gradient subspace across depths, widths, and training epochs (Figure 6).

We generate a guess for $\frac{\partial L}{\partial x_{k+1}}$ by taking current activations x_{k+1} and computing random linear combinations of all the activations in the batch. We call this method "activation mixing".

3. Results

We evaluate the directional descent baseline, activation perturbation baseline, activation mixing, W^T, and "1-Layer Downstream". Please see supplementary material for implementation details.

Cosine similarity along backprop: We compare each method's gradient guess on an MLP with depth 6, width 1024, trained using backprop and show the results in Figure 1. On average, our proposed methods such as W^T produce cosine similarities that are hundreds of times higher than directional descent. **One step effectiveness**: Cosine similarity ignores the curvature of the loss landscape and thus can be a misleading measure of a method's effectiveness. We directly compare the loss reduction relative to backprop and further search over multiple step sizes $[10^{-5}, \ldots, 10^{-1}]$ for each method (and backprop) to make the comparison as fair as possible. Our methods are thousands of times more effective than directional descent in the 1-step regime.

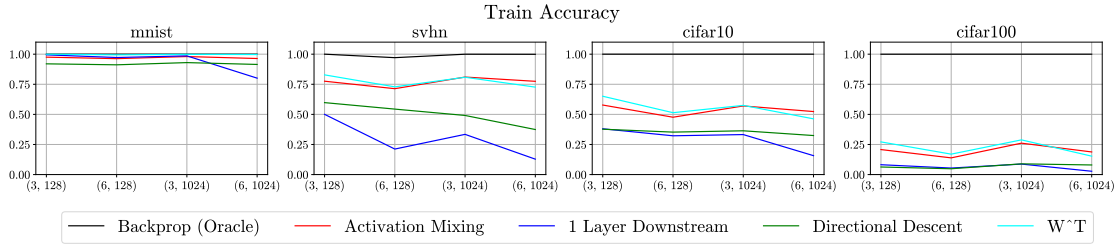


Figure 2: Our proposed methods outperform directional descent, but struggle against backprop. We plot MLP train and test accuracies for various methods and datasets. The columns refer to MNIST, SVHN, CIFAR10, and CIFAR100 respectively. The x-axis in each plot is labelled as (depth, width) for each MLP configuration, and sorted by the number of parameters. Refer to Table 1 for details.

Training MLPs on MNIST, SVHN, CIFAR10, CIFAR100: We next conducted experiments to train MLPs using our proposed methods on four commonly used datasets: MNIST, SVHN, CIFAR10, and CIFAR100. The plots are reported in Figure 2. While our proposed methods outperform directional descent, there is a large gap between these methods and backprop, and the gap grows larger with more complex datasets.

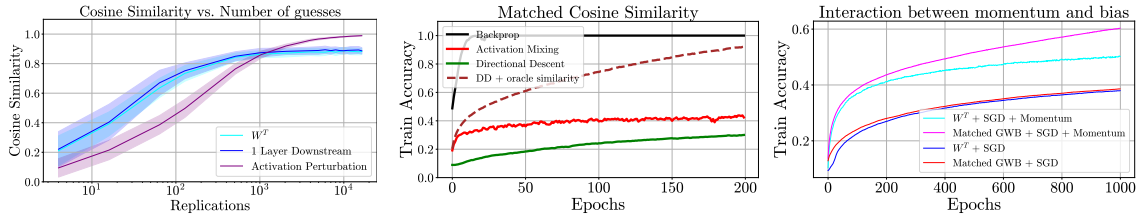


Figure 3: **(left)** Our methods (Mixing, Downstream, W^T) are biased estimators of the gradient. For a single example input, we average the multiple guesses and plot cosine similarity as a function of the number of guesses. In contrast to an unbiased random baseline where averaging over more guesses leads to better cosine similarities, the cosine similarity quickly saturates for the biased methods. **(middle)** The cosine similarity achieved by our methods, without the bias, is sufficient to achieve high training accuracy on tasks like CIFAR10. **(Right)** Momentum and bias interact. We replicate the matched cosine similarity, but using SGD with/without momentum. We find that when momentum is used, bias causes a large slowdown for our method related to the unbiased version (as observed in the paper). However, without momentum, the difference between the two conditions is negligible. This may indicate a crucial role of momentum in this gradient guessing problem.

Bias: Unbiased methods like directional descent display better cosine similarity when averaged over many guesses. Experiments employing a Multi-Layer Perceptron (MLP) on CIFAR10 show biased methods like W^T and activation mixing plateau in cosine similarity after around 1000 guesses, while unbiased methods continue to improve. We further show that our cosine similarities (0.02) are sufficient for high accuracy in the absence of bias (Figure 3). Finally, we also show that momentum contributes to the observed slowdown from bias.

Self-sharpening effect: We report a peculiar "self-sharpening" behavior seen in some methods, where the training dynamics lead to the guesses becoming better over time. While we do not know the precise cause of this effect, we hypothesize that the biased guesses dominated by a few singular values lead to lower rank weight matrices, which further lead to higher cosine similarity over the course of training. We can replicate this effect by modifying W^T guessing method: We compute the

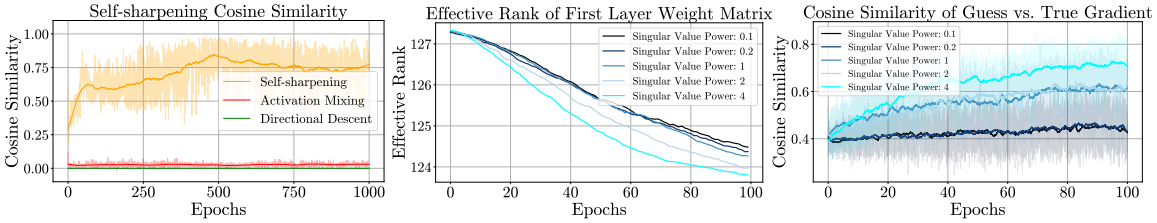


Figure 4: Self-sharpening effect leads to the cosine similarity increasing over the course of training and a higher training accuracy as a result. We re-create this effect by making some singular values dominate the guess. This leads to the weight matrices becoming lower rank over time, which leads to increased cosine similarity.

singular value decomposition of W and raise its singular values to various powers $[0.1, 0.2, \ldots, 4]$. Higher powers lead to a more imbalanced distribution, with a few singular values dominating the rest. We plot the resulting cosine similarity and effective rank [17] in Figure 4.

4. Conclusion

We show it is possible to produce gradient guesses with dramatically higher cosine similarities than directional descent. While these methods help close the gap between backprop and forward gradients, bias is a major limiting factor for the scalability of our methods.

References

[1] Atılım Güneş Baydin, Barak A Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without backpropagation. *arXiv preprint arXiv:2202.08587*, 2022.

[2] Gabriel Belouze. Optimization without backpropagation. *arXiv preprint arXiv:2209.06302*, 2022.

[3] Kartik Chandra. An unexpected challenge in using forward-mode automatic differentiation for low-memory deep learning, 2021. URL https://searchworks.stanford.edu/view/wk389rs3026.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[5] Menglin Jia, Luming Tang, Bor-Chun Chen, Claire Cardie, Serge Belongie, Bharath Hariharan, and Ser-Nam Lim. Visual prompt tuning. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXIII*, pages 709–727. Springer, 2022.

[6] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. Novel dataset for fine-grained image categorization: Stanford dogs. In *Proc. CVPR workshop on fine-grained visual categorization (FGVC)*, volume 2. Citeseer, 2011.

[7] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 554–561, 2013.

[8] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[9] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018.

[10] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning, 2022. URL https://arxiv.org/abs/2205.05638.

[11] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[12] Kevin Lu, Aditya Grover, Pieter Abbeel, and Igor Mordatch. Pretrained transformers as universal computation engines. *arXiv preprint arXiv:2103.05247*, 2021.

[13] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[15] Boris T Polyak. Introduction to optimization. optimization software. *Inc., Publications Division, New York*, 1:32, 1987.

[16] Mengye Ren, Simon Kornblith, Renjie Liao, and Geoffrey Hinton. Scaling forward gradient with local losses. *arXiv preprint arXiv:2210.03310*, 2022.

[17] Olivier Roy and Martin Vetterli. The effective rank: A measure of effective dimensionality. In *2007 15th European signal processing conference*, pages 606–610. IEEE, 2007.

[18] David Silver, Anirudh Goyal, Ivo Danihelka, Matteo Hessel, and Hado van Hasselt. Learning by directional gradient descent. In *International Conference on Learning Representations*, 2021.

[19] J. C. Spall. A stochastic approximation technique for generating maximum likelihood parameter estimates. In *1987 American Control Conference*, pages 1161–1167, 1987. doi: 10.23919/ACC.1987.4789489.

[20] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. The caltech-ucsd birds-200-2011 dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[21] Mitchell Wortsman, Tim Dettmers, Luke Zettlemoyer, Ari Morcos, Ali Farhadi, and Ludwig Schmidt. Stable and low-precision training for large-scale vision-language models. *arXiv preprint arXiv:2304.13013*, 2023.

Appendix A. Supplementary Materials

In this section, we describe experimental details for the experiments mentioned in the paper. We also run a hyperparameter sweep over learning rates and optimizers to check for the convergence speed of each method. All our code is implemented in PyTorch [14].

A.1. Bias Analysis

In this section, we discuss one possible source of the bias present in our proposed methods. We start with the unbiased estimator which uses JVP:

$$\hat{g} = (\nabla L.y)y \tag{4}$$

and we compute the expectation of this estimator:

$$\mathbb{E}\big[\hat{g}\big] = \mathbb{E}\big[(\nabla L.y)y\big] = \mathbb{E}\big[yy^T\big]\nabla L = \mathrm{Cov}(y)\nabla L \tag{5}$$

thus, in expectation, the gradient guess is equal to the original guess scaled by the covariance matrix of the guess. Thus the bias is:

$$\mathbb{E}\big[\hat{g}\big] - \nabla L = (\mathrm{Cov}(y) - I)\nabla L \tag{6}$$

Therefore, the guess can only be unbiased if the covariance matrix is equal to the identity matrix in the subspace that the gradients lie in.

For our proposed estimators, this is easily shown to be false. Activation mixing uses random mixtures of activations as the gradient guesses, and thus its covariance matrix is the same as the covariance matrix of the activations (and thus non-identity). Our methods rely on these covariance matrices being low rank and well-aligned with the gradient subspace to produce high cosine similarity guesses. Still, as a result, our expected guess is also scaled by these covariance matrices, thus biased. In future work, we hope to use this information to undo the bias caused by these non-identity covariance matrices.

Bias for W^T: The W^T method involves sampling a random normal noise vector and transforming it with W^T to confine it to the range space. Thus, the guess y for any given layer can be written as:

$$y = W^T\epsilon$$

where ϵ is a random normal vector, i.e., $\epsilon_i \sim \mathcal{N}(0, 1)$. Thus the guess vector y is also a multivariate normal vector with the covariance matrix:

$$\mathrm{Cov}(y) = W^TW$$

and so the bias for each layer's activation is given by:

$$\implies \mathrm{Bias}[W^T] = (W^TW - I)\nabla L$$

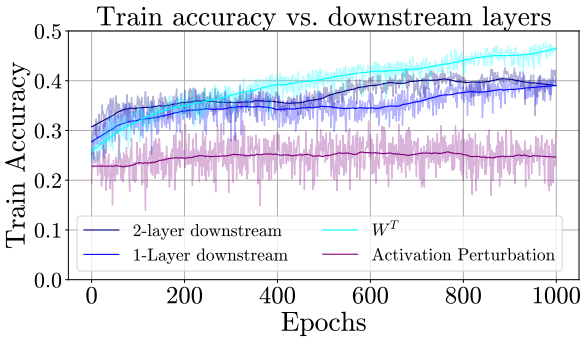


Figure 5: Additional downstream layers don't help. We train a 6-layer, 1024 wide MLP on CIFAR10 using various methods and plot their training curves above. We see that 2-layer downstream has no advantage on 1-layer downstream, indicating that the increase in bias counteracts any variance benefits.

Why more layers are not always better: Why not use as many steps as possible in partial backpropagation (e.g., using the next 2 or 3 downstream layers)? In practice, the bias can increase with each additional layer (see Figure 5). Here, we show how with a simple toy example.

Let the activation vector at Layer i be represented by a vector $x_i \in \mathbb{R}^n$, and let the jacobians of the next few layers (i.e. layers $i+1, i+2, \ldots, i+k$) be represented by $J_{i+1}, J_{i+2}, \ldots J_{i+k} \in \mathbb{R}^{n \times n}$ (here we assume that all layers are the same width without loss of generality). We denote their product, the accumulated jacobian for layers $i+1$ to $i+k$, as J for notational simplicity. Also, let $g_k \in \mathbb{R}$ be the true corresponding gradient.

We begin by noting that $g_i = J\frac{\partial L}{\partial x_{i+k}}$ by chain rule. Thus, g_i lies in the range space of J, i.e. $g_i \in \mathcal{R}(J)$. Using this knowledge can significantly reduce the guessing space.

Let's look at how our methods use this knowledge: They sample a random normal vector $n_i \in \mathbb{R}^{\times}$ multiply it by the jacobian to generate the guess direction $y = Jn$ for some normal vector $n \in \mathbb{R}^n$. This guess y is used in a forward JVP to generate an estimate $\hat{g} = JVP(y)y = (g.y)y$

Using equation 6, the bias of $\hat{g}$ is:

$$\mathbb{E}[\hat{g}] - g = (Cov(y) - I)g = (Cov(Jn) - I)g = (JJ^T - I)g \tag{7}$$

The method predicts $\hat{g} = JJ^T g$ instead of the true g, resulting in the bias $(JJ^T - I)g$. To show this bias can increase with more layers, we consider a simple case where each $J_i = 2 * I$. Then $J = J_{i+1} \ldots J_i + k = 2^k I$, and the bias is $(4^k - 1)||g||$, which increases with k.

A.2. Gradients and activations approximately lie in the same subspace

We compute the cosine similarity between the true gradient and its projection onto the subspace spanned by activations. If the activation and gradient subspaces are approximately aligned, the cosine similarity between the gradient and its projection should be high. We pick the basis for the activation subspace by running PCA and using the principal components as the basis vectors. We contrast this to a random subspace created by randomly sampling a set of vectors and orthonormalizing them. We plot the resulting curves for each layer in MLPs of depth 3,4, or 6, widths 1024, and during all 20 training epochs. We see that the activation subspace consistently requires much fewer basis vectors for a significantly better approximation than a random subspace, getting cosine

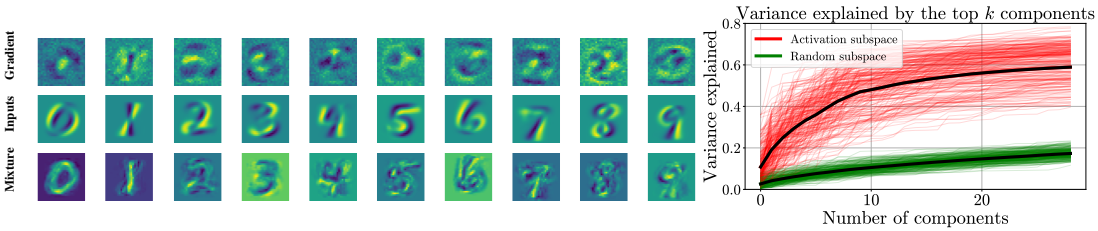


Figure 6: Activations and gradients approximately lie in the same subspace. For an MLP trained on MNIST digit classification, we plot (as images) for each class **(a)** the first principal component of gradients with respect to input images (top row), **(b)** the first principal components of the inputs (middle) and **(c)** random combinations of inputs (bottom row). Even though the MLP is initialized with random weights, the principal components of gradients look similar to inputs. Our "activation mixture" method uses random mixtures of activations to generate guesses in the same subspace as the gradients. **(Right)** Activation subspace is a better match for gradients than a random subspace.

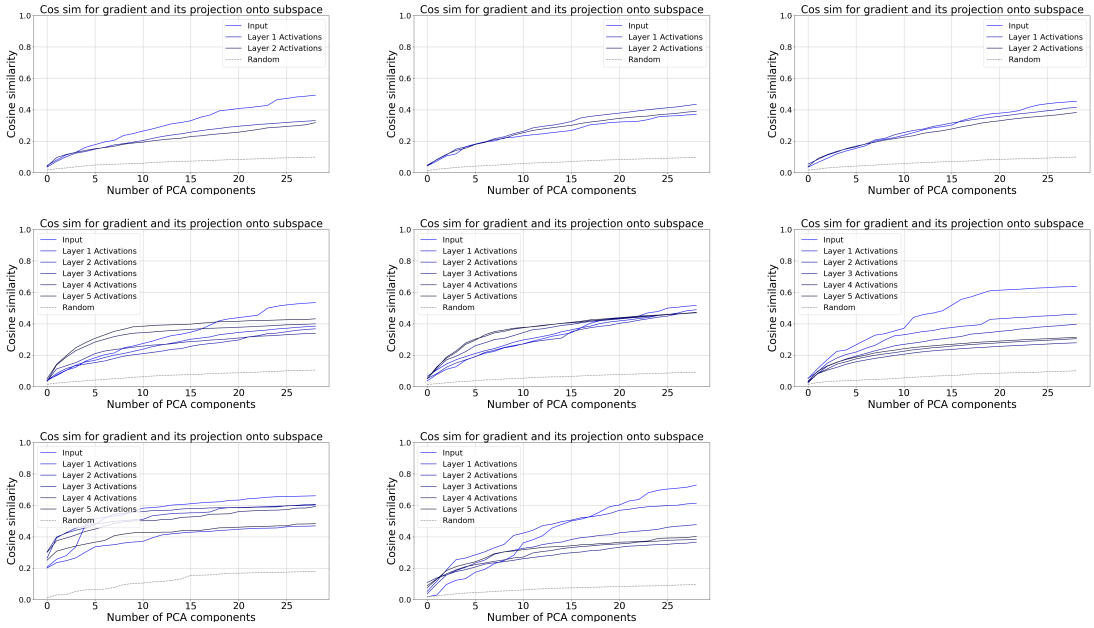


Figure 7: Activation and gradient subspace similarity

similarity as high as 0.5 with less than 10 principal components (in contrast, random subspace gets 0.1 cosine similarity). Figure 6 superimposes curves from many different networks and epochs, and Figure 7 shows some of them separately for various network widths, layers, and depths.

A.3. Learning Rate and Optimizer Sweep

To test each method's convergence speed, we run a sweep over learning rates [10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}] and optimizers [AdamW [11], SGD, StableAdamW [21]]. We fix the same MLP architecture (3 layers, width 128) and dataset (CIFAR10). We plot the training accuracy for each method and each optimizer, choosing the best learning rate (defined as the highest train accuracy at the end of 1000 epochs) (Figure 8). We see that backprop predictably converges much faster than other

methods, and gradient guessing methods such as directional descent and ours benefit greatly from Adam.

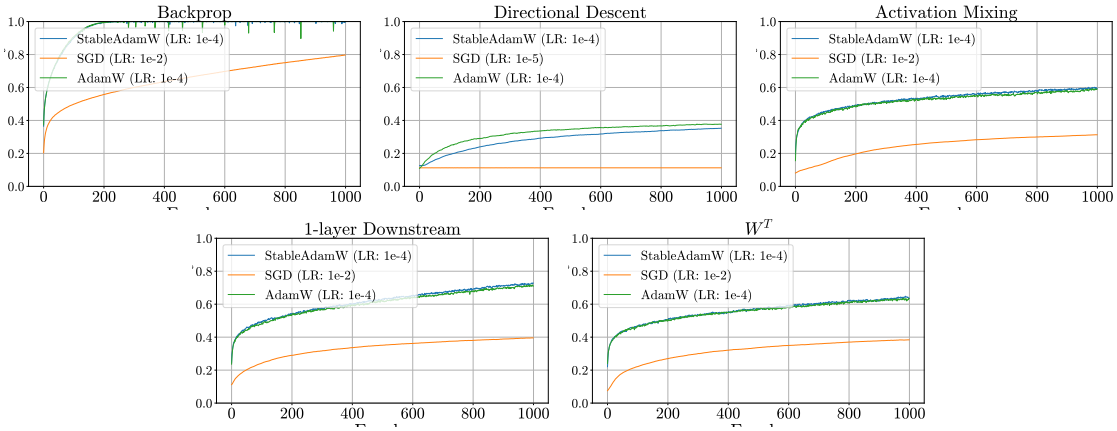


Figure 8: We plot each method's training curves for different optimizers on CIFAR10 for an MLP with 3 layers and width 128. We note that backprop converges significantly faster than directional descent and our proposed method. We also note that directional descent and our methods benefit greatly from variants of the Adam optimizer.

A.4. Experimental Details

Jacobian-vector product (JVP) and forward gradients: JVP takes as input a function $f(I)$, inputs i (called primals), and perturbations p (called tangents), and computes $J_f(I)|_{I=i} \cdot p$, where J_f refers to the jacobian of f and $J_f \cdot p$ refers to multiplying the jacobian matrix J_f with the perturbations p. It measures the effect of infinitesimal perturbations p around the original inputs i. This can be used for gradient estimation in two ways: (a) weight perturbations and (b) pre-activation perturbations.

For weight perturbations, the process is the same as used in [1]. A gradient estimate $\hat{g}$ of the true gradient $g = \nabla_W L$ is generated (for directional descent, this is a random normal vector). Given such a guess, the JVP $(g.\hat{g})$ is computed. Note that this process does not require knowledge of the true gradient or any backward pass. The final update is just the guess scaled by the JVP, $\hat{g} \cdot \text{JVP} = \hat{g} \cdot (\hat{g} \cdot g)$

For pre-activation perturbations, the guesses and gradients are for pre-activations (i.e. the neuron values after a linear layer and before an activation function like ReLU). The new guess $\hat{g} = \nabla_{\boldsymbol{x}} L$ perturbs each pre-activation, and the JVP of this perturbation is measured. After scaling thiw pre-activation guess with the JVP, it can be converted into a weight update by computing the outer product as normally done in backpropagation $\Delta W_k = x_k g_k^T$.

Note that unlike weight perturbations, pre-activation perturbations apply separately for each batch element. Thus the JVP is measured separately for each measurement as well (i.e., we get as many JVP values as the batch size instead of one JVP value per batch). This allows pre-activation perturbations to take advantage of batch parallelism and get more gradient information for the same computation for each batch.

Directional descent baseline: As described above, our guess is a random normal vector. Each entry is given by $\hat{g}_i \sim \mathcal{N}(0, \frac{1}{\sqrt{N}})$. The division by $\sqrt{N}$ ensures that the norm of the guess doesn't grow with the parameter count.

Activation perturbation baseline: This baseline uses random normal perturbations for activations instead of weights. As described above, after scaling the guess with the JVP, we convert the scaled pre-activation guess to a weight update using the outer product.

In practice, this conversion is implemented using the ".backward()" function available for linear layers in PyTorch. We note that although we use ".backward()" to update individual layers one at a time, we are not using backpropagation to generate the guesses or to evaluate the JVP. This process can be done in a second forward pass once the JVP has been computed and does not require storing activations for all layers, unlike backpropagation.

Activation mixing: For each layer k, we take the batch of activations $\left[x_k[1], \ldots, x_k[B]\right]$ where B is the number of batch elements. We compute a random linear combination $\alpha_1 x_k[1] + \ldots + \alpha_B x_k[B]$, where each weight $\alpha_i \sim \mathcal{N}(0, 1)$. To incorporate sparsity, we multiply each resulting guess by the ReLU activation mask $\frac{\partial \text{ReLU}(x_k[i])}{\partial x_k[i]}$. We also normalize the overall guess to ensure stability during training. In practice, this is implemented using backprop, as mentioned above. Since activation mixing does not apply to the last layer, we use a random normal guess for the last layer.

$\mathbf{W}^T$: For each batch element, we start with a random normal vector and multiply it by the transpose of the next layer's weight matrix W_{k+1}^T. We incorporate sparsity by multiplying with the ReLU mask as mentioned above and create a weight update in a similar fashion to previous methods. Similar to activation mixing, the last layer receives a random normal guess.

1-layer downstream: For each batch element, we trace the intermediate activations until the next layer and backpropagate a random normal vector through this graph. This method backpropagates signals from one layer to another but not across multiple layers as in regular backpropagation. We also note that the backpropagated vector is random and thus contains no information about the true gradient other than being in the same subspace. The update method and last layer treatment are the same as previous methods.

Self-sharpening experiments: For the self-sharpening experiments, we use the same framework as 1-layer downstream, but we use random uniform guessing instead of random normal, and the last layer gradients are replaced with the true error vector instead of a random guess. The last layer gradients are not required to see this effect, but they drastically speed up convergence. Since this method can be unstable, we also use the StableAdamW optimizer. **Self-sharpening through singular value manipulation**: This experiment was conducted on an MLP with 6 layers and 128 units per layer. We trained the MLP on CIFAR10 with batch size 128, LR 10^{-4} 0 weight decay, and 64 replicates for 50000 iterations.

Artificially modifying directional descent's cosine similarity: For our bias experiments, we modify directional descent to see how it would perform with a cosine similarity like our methods. To achieve this modified directional descent algorithm, we blend the random guess with the true gradient computed using full backpropagation. Specifically, we compute the two components of the guess: one along the gradient ($g_{\|}$) and one orthogonal to it ($g_{\perp}$). These two components are normalized to create an orthogonal basis. Then, a guess that has an angle θ with the gradient can be created as: $\hat{g}_\theta = \cos\theta g_{\|} + \sin\theta g_{\perp}$

Train Accuracy

	Method	CIFAR100	CIFAR10	SVHN	MNIST
D:3, W:128	Directional gradients	6.7 ± 0.3	37.4 ± 0.2	59.8 ± 0.4	92.3 ± 0.2
	Activation Perturbation	13.3 ± 0.1	45.9 ± 0.3	69.3 ± 1.3	98.7 ± 0.1
	Mixing (ours)	20.1 ± 0.6	57.2 ± 0.8	76.34 ± 0.6	96.9 ± 0.5
	W^T (ours)	25.6 ± 0.2	62.4 ± 0.8	81.3 ± 0.9	100.0 ± 0
	Downstream (ours)	30.8 ± 0.7	65.6 ± 0.5	83.2 ± 0.8	99.9 ± 0.1
	Self-sharpening (ours)	11.4 ± 1.7	51.5 ± 5.5	71.6 ± 1.9	99.5 ± 0.3
	Backprop (oracle)	100.0 ± 0	100.0 ± 0	99.5 ± 0.9	100.0 ± 0
D:6, W:128	Directional gradients	5.2 ± 0.4	35.9 ± 0.2	54.8 ± 1.2	91.4 ± 0.2
	Activation Perturbation	9.4 ± 0.5	38.9 ± 0.4	56.1 ± 1.8	97.3 ± 0.3
	Mixing (ours)	13.1 ± 0.7	48.5 ± 1.3	72.1 ± 1.3	96.6 ± 0.3
	W^T (ours)	16.5 ± 0.4	51.5 ± 0.2	73.2 ± 1.1	99.2 ± 0.1
	Downstream (ours)	16.1 ± 0.4	52.8 ± 0.8	77.0 ± 1.2	99.5 ± 0.1
	Self-sharpening (ours)	21.1 ± 8.5	42.5 ± 15.4	78.9 ± 11.2	100.0 ± 0.0
	Backprop (oracle)	100.0 ± 0	100.0 ± 0	100.0 ± 0	100.0 ± 0
D:3, W:1024	Directional gradients	9.0 ± 0.1	35.9 ± 0.2	54.8 ± 1.2	91.4 ± 0.2
	Activation Perturbation	9.9 ± 0.6	36.7 ± 2.7	50.1 ± 5.6	96.2 ± 1.2
	Mixing (ours)	26.9 ± 0.4	59.4 ± 1.2	80.8 ± 1.8	98.4 ± 1.2
	W^T (ours)	28.2 ± 0.5	56.9 ± 1.4	79.8 ± 1.3	99.6 ± 0.3
	Downstream (ours)	28.4 ± 0.7	58.5 ± 1.0	80.6 ± 0.5	99.9 ± 0.1
	Self-sharpening (ours)	23.1 ± 1.1	70.8 ± 15.0	69.8 ± 5.2	100.0 ± 0.0
	Backprop (oracle)	100.0 ± 0	100.0 ± 0	100.0 ± 0	100.0 ± 0
D:6, W:1024	Directional gradients	7.8 ± 0.1	32.2 ± 0.1	37.4 ± 0.2	91.4 ± 0.3
	Activation Perturbation	3.6 ± 0.4	26.9 ± 1.6	24.3 ± 4.5	95.5 ± 0.8
	Mixing (ours)	18.1 ± 0.4	52.2 ± 1.3	78.4 ± 2.2	97.2 ± 0.2
	W^T (ours)	14.6 ± 0.3	46.7 ± 0.9	73.6 ± 0.8	99.4 ± 0.2
	Downstream (ours)	9.7 ± 0.3	37.6 ± 2.6	71.3 ± 0.8	99.4 ± 0.2
	Self-sharpening (ours)	18.1 ± 9.1	97.5 ± 3.9	97.9 ± 0.5	99.9 ± 0.1
	Backprop (oracle)	99.9 ± 0.1	100.0 ± 0	100.0 ± 0	100.0 ± 0

Test Accuracy

	Method	CIFAR100	CIFAR10	SVHN	MNIST
D:3, W:128	Directional gradients	6.1 ± 0.4	35.8 ± 0.4	57.4 ± 0.4	92.2 ± 0.2
	Activation Perturbation	12.2 ± 0.2	42.3 ± 0.5	65.4 ± 1.3	96.9 ± 0.2
	Mixing (ours)	11.6 ± 0.3	44.2 ± 0.5	69.6 ± 0.4	95.7 ± 0.2
	W^T (ours)	17.1 ± 0.3	46.9 ± 0.8	76.2 ± 0.4	97.5 ± 0.1
	Downstream (ours)	18.1 ± 0.3	47.8 ± 0.7	76.9 ± 0.4	97.5 ± 0.1
	Self-sharpening (ours)	10.6 ± 0.9	35.4 ± 2.1	64.8 ± 1.6	94.2 ± 0.5
	Backprop (oracle)	19.7 ± 0.1	49.0 ± 0.3	81.8 ± 0.2	97.8 ± 0.1
D:6, W:128	Directional gradients	4.6 ± 0.4	34.4 ± 0.5	53.2 ± 1.3	91.4 ± 0.1
	Activation Perturbation	9.9 ± 0.5	38.9 ± 0.3	53.4 ± 1.1	96.2 ± 0.1
	Mixing (ours)	10.6 ± 0.5	42.5 ± 1.0	66.4 ± 0.9	95.3 ± 0.2
	W^T (ours)	14.1 ± 0.3	46.5 ± 0.4	69.4 ± 1.6	96.9 ± 0.2
	Downstream (ours)	14.1 ± 0.3	46.5 ± 0.5	72.8 ± 1.0	97.0 ± 0.1
	Self-sharpening (ours)	14.0 ± 2.7	43.2 ± 1.1	70.6 ± 0.8	96.2 ± 0.2
	Backprop (oracle)	17.9 ± 0.4	47.5 ± 0.3	80.2 ± 0.3	97.4 ± 0.1
D:3, W:1024	Directional gradients	8.0 ± 0.2	34.8 ± 0.3	47.2 ± 1.0	92.8 ± 0.1
	Activation Perturbation	9.4 ± 0.2	38.3 ± 0.4	55.6 ± 1.1	96.6 ± 0.2
	Mixing (ours)	15.0 ± 0.3	44.6 ± 0.7	73.3 ± 0.9	97.0 ± 0.2
	W^T (ours)	17.6 ± 0.5	48.0 ± 0.4	75.3 ± 0.6	97.7 ± 0.1
	Downstream (ours)	18.3 ± 0.3	48.4 ± 0.5	76.4 ± 0.5	97.8 ± 0.0
	Self-sharpening (ours)	11.9 ± 1.4	38.7 ± 1.1	65.1 ± 4.0	96.1 ± 0.1
	Backprop (oracle)	25.0 ± 0.5	53.8 ± 0.1	83.2 ± 0.1	98.3 ± 0.1
D:6, W:1024	Directional gradients	6.9 ± 0.5	31.7 ± 0.3	37.8 ± 0.4	91.3 ± 0.3
	Activation Perturbation	4.7 ± 0.2	32.4 ± 0.6	31.8 ± 2.0	96.0 ± 0.1
	Mixing (ours)	14.0 ± 0.2	44.9 ± 0.5	72.5 ± 0.8	95.7 ± 0.2
	W^T (ours)	14.0 ± 0.5	45.2 ± 0.5	70.7 ± 0.6	97.4 ± 0.1
	Downstream (ours)	10.2 ± 0.4	40.6 ± 0.4	69.6 ± 1.3	97.5 ± 0.1
	Self-sharpening (ours)	10.1 ± 0.5	33.5 ± 2.9	68.1 ± 0.8	93.7 ± 1.9
	Backprop (oracle)	25.2 ± 0.3	52.4 ± 0.3	82.9 ± 0.1	97.9 ± 0.2

Table 1: Train and test accuracies for all our proposed methods as well as the self-sharpening effect. We note that while the self-sharpening effect can deliver high train accuracy, it prevents the model from generalizing and thus achieving high test accuracy.

A.5. MLP results

Here we tabulate the train and test accuracy for all methods on various MLP sizes and datasets.

A.6. Visual prompt-tuning experiments

As models become ever larger, the burden of even instantiating the computational graph of a model for a single minibatch becomes untenable for modest computational resources. Backpropagation exacerbates this issue for tuning methods like [8, 10, 12], which require intermediate states of the graph to be saved for the backward pass. We apply our gradient guessing methods to the space of parameter-efficient fine-tuning (PEFT), where memory-efficient training methods are relevant. One popular approach is prompt-tuning [5, 8]. For pre-trained vision transformers (VIT-B/16 [4]), we tune the prompt tokens by guessing the gradient of the prompt. Figure Figure 9 shows that our W^T method consistently outperforms directional descent for all fine-tuning datasets (CUB-200 [20], Stanford Cars [7], Stanford Dogs [6]).

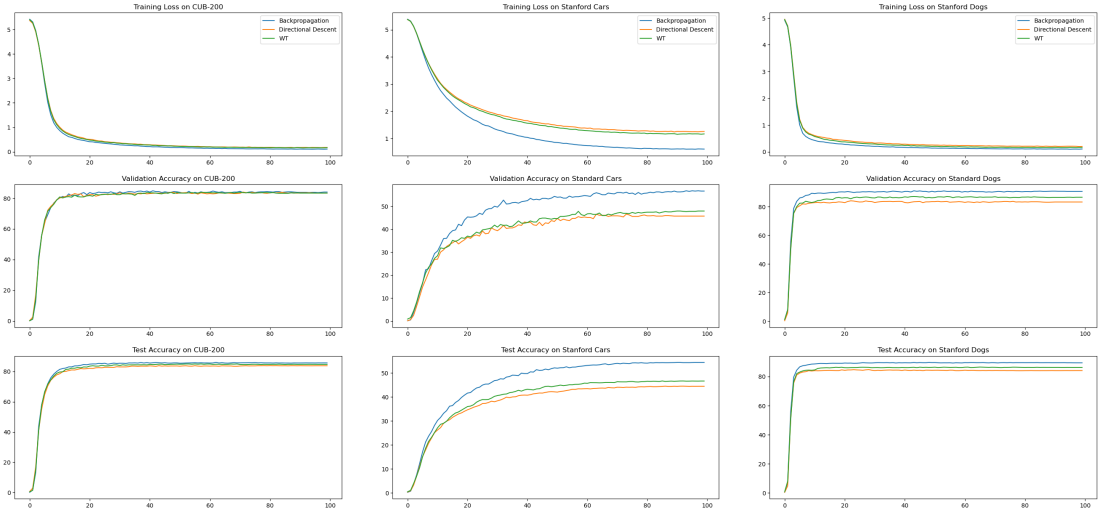


Figure 9: Gradient guessing strategies applied to Visual Prompt Tuning [5] where prompt tokens are trained with backpropagation, directional descent, and W^T.