Forward Gradients Are Not All You Need

Andre Daprato **Kevin Jain** **Rikki Hung**

Abstract

Backpropagation, which propagates gradients through layers backwards, forms the backbone for training neural networks. Recent work by Baydin et al. introduces an alternative method to backpropagation that estimates gradients by scaling a random perturbation vector by the directional derivative of that vector. We investigate the improvements claimed by the author, along with extending their work by investigating the performance of forward gradient on typical machine learning tasks. We find that the forward gradient lacks the stability of backpropagation and is not currently viable as a 'drop-in alternative'.

1 Introduction

Neural Networks (NN) have dominated the fields of image recognition and natural language processing in recent years. Training NNs is typically done with the use of gradient-based methods, which use backpropagation to determine the gradient of a neural network's function with respect to its parameters. Recent work (Baydin et al. 2022) has introduced an alternative to the gradient, the forward gradient, which is an estimate of the gradient given by a perturbation of a random vector that is independently sampled in each dimension from a distribution with mean 0 and unit variance scaled by the directional derivative of f in the direction of this vector.

Definition 1. Given a vector $\mathbf{v} \sim (\mathcal{N}(0,1))^n$, input parameters $\theta \in \mathbb{R}^n$, and function $f : \mathbb{R}^n \to \mathbb{R}$, the forward gradient, $g(\theta)$, is defined as:

$$g(\theta) = (\nabla f(\theta) \cdot \mathbf{v})\,\mathbf{v} \tag{1}$$

Note that $\nabla f(\theta) \cdot \mathbf{v}$ is the directional derivative of f in the direction of $\mathbf{v}$. Intuitively, the forward gradient is derived by taking a sample perturbation of the weights, and then updating the weights in the direction of this perturbation with step size proportional to how well this perturbation approximates the true gradient. The forward gradient is an unbiased estimate (Baydin et al. 2022) of the gradient and can be used as a direct substitute for the gradient in optimization methods. Unlike the gradient, which must by calculated with a forward and backward pass of the model, the directional derivative can be calculated in parallel with the forward pass–this eliminates the need to conduct the backward pass. Gradient descent is at the heart of deep learning: in this paper we investigate the viability of the forward gradient as an alternative to the gradient in various deep learning tasks.

2 Related Works

We attempt to replicate and extend the work of Baydin et al. 2022 to encompass different model architectures and gradient update methods. Preliminary experimentation has shown the naive implementation of the forward gradient trains to the same performance threshold nearly twice as fast as backpropagation using SGD on the MNIST dataset (LeCun and Cortes 2010) at the same learning rate. Baydin et al. 2022 provides an overview of optimization methods using random perturbations, as well as other alternatives to gradient descent. Silver et al. 2022 also presents an overlook on the gradient estimate $(\frac{\partial L}{\partial w} \cdot u)u$ using directional derivatives in the direction u and presents different ways of sampling the perturbation vector. Their investigation is focused on the improved performance of directional derivatives particularly in the context of recurrent neural networks where it is infeasible to get an exact gradient computed over long time steps. They also derive unbiasedness of the directional derivative and investigate the variance of these estimates with respect to model size (Silver et al. 2022).

Recent work on automatic differentiation libraries has enabled efficient computing of Jacobian-vector products along with the forward pass of the function (Bradbury et al. 2018, Paszke et al. 2019). Existing research on training neural networks with backpropagation has found a number of techniques to augment neural networks and improve performance (He et al. 2015, Ioffe and Szegedy 2015, Labach, Salehinejad, and Valaee 2019) without increasing the number of parameters; we explore some of these techniques in the setting of forward gradient descent.

3 Method

Given a fixed NN architecture and training data, we may define a loss function f based on this network from the parameter space, denoted by $\mathbb{R}^n$, to $\mathbb{R}$. Since f is the composition of linear transformations and almost everywhere smooth activation functions, it is itself smooth almost everywhere. For each value of the NN model weights, $p \in \mathbb{R}^n$, the loss f has an associated linear map $f_* : T_p\mathbb{R}^n \to T_{f(p)}\mathbb{R}$, called the push-forward (Lee 2000). Here $T_p\mathbb{R}^n$ is the tangent space at p, which may be thought of as a copy of $\mathbb{R}^n$ with origin at p.

Our implementation of the forward gradient is based on JAX (Bradbury et al. 2018) and its jvp[1] (Jacobian-vector product) operation, which computes the push-forward of our loss function: $f_* : (p, v) \mapsto (f(p), \nabla f(p) \cdot v)$, with respect to the current model weights p and our sampled perturbation of them, v. We obtain v by sampling a random vector in the same shape as the parameters with each dimension sampled independently from a Normal distribution with mean 0 and variance 1. We implement our experimental models in Haiku (Hennigan et al. 2020), which is based on the JAX library.

```
def fgd(model, params, images, targets, key):
  f = lambda W: loss_fn(W, model, images, targets)
  v = sample_perturb(params, key)
  L, u = jax.jvp(f, (params,), (v,))
  return L, v, u #loss, perturbation, forward gradient
```

Figure 1: A sample implementation of the forward gradient in JAX

3.1 Derivation of Variance

We extend the derivation of the variance of DODGE (Silver et al. 2022) to derive the variance of the forward gradient. Here the component-wise variance of the forward gradient. Suppose $\theta, v \in \mathbb{R}^n$, and v_k is the k-th component of the vector k.

$$\begin{aligned}
\text{Var}[g_k(\theta)] = \mathbb{E}[g_k^2(\theta)] - \mathbb{E}^2[g_k(\theta)] &= \mathbb{E}[g_k^2(\theta)] - \nabla f(\theta)_k^2 \text{ as the forward gradient is an unbiased estimate} \\
&= \mathbb{E}[(v_k \sum_{j=1}^{n} \nabla f(\theta)_j v_j)^2] - \nabla f(\theta)_k^2 \\
&= \mathbb{E}[v_k^2 (\sum_{j=1}^{n} (\nabla f(\theta)_j v_j)^2 + \sum_{j=1}^{n}\sum_{i=1}^{n} 2\nabla f(\theta)_j v_j \nabla f(\theta)_i v_i)] - \nabla f(\theta)_k^2 \\
&= \mathbb{E}[v_k^2 \sum_{j=1}^{n} (\nabla f(\theta)_j v_j))^2] - \nabla f(\theta)_k^2 \\
&= \sum_{j=1}^{n} \nabla f(\theta)_j^2 \mathbb{E}[(v_k v_j)^2] - \nabla f(\theta)_k^2 \\
&= \sum_{j=1}^{n} \nabla f(\theta)_j^2 - \nabla f(\theta)_k^2
\end{aligned}$$

[1] `https://jax.readthedocs.io/en/latest/_autosummary/jax.jvp.html#jax.jvp`

As the number of parameters, n, increases, we see that the variance of each component of the forward gradient also increases. Silver et al. 2022 describe improvements to this estimate which reduce the variance at a runtime cost.

4 Experiments

4.1 Sanity check: Forward Gradient vs Backpropagation

We use an MLP architecture of two linear layers each with `size` 512 followed by reLU activation, and one linear classification layer of size 10. We achieve similar results to Baydin et al. 2022 with small learning rates 2×10^{-4} and 2×10^{-5} (fig. 2). However, backpropagation is able to train with a higher learning rate in general than the forward gradient: at a learning rate of 2×10^{-3}, backpropagation achieves the best results, while the forward gradient collapses.

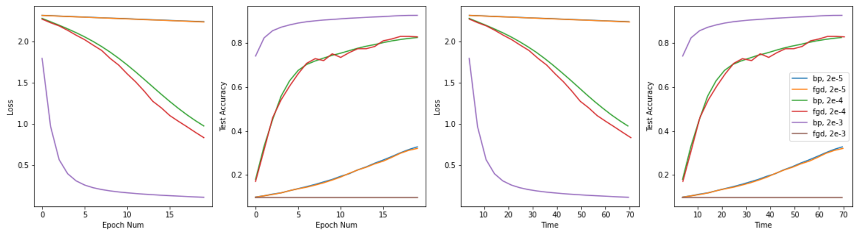

Figure 2: Forward gradient vs backprop with different learning rates

4.2 Model Size and Adam

On the same task, at a learning rate of 2×10^{-4}, we compare the forward gradient and backpropagation both with and without the Adam optimizer (fig. 3) on hidden layers of `size` 512, 1024, and 2048. We find that the addition of the Adam optimizer greatly improves training stability as model size increases. While backpropagation performance improves with model size, the forward gradient's performance did not improve with higher hidden layer size.

4.2.1 Batchnorm

We added two batchnorm layers, one after each of the first two linear layers in the MLP architecture from section 4.1. While batchnorm improved the performance of the regular backpropagation implementation, it worsened the performance of the forward gradient. Notably, the variance during the SGD forward gradient training substantially increased, but this variance was smoothed out in the Adam implementations.

4.3 CIFAR

We trained the MLP architecture from section 4.1 with a varied number of parameters (1024,) using both forward gradients and backpropagation. We found that even the smallest models trained with forward gradients and SGD experienced exploding gradients. Forward gradients paired with the Adam optimizer allowed for stable training, but loss performance was worse than backpropagation training with Adam and the same hyperparameters.

4.4 Convolutional / Residual Networks

We implement the same convolutional architecture as from (Baydin et al. 2022), with 4 convolutional layers (64 channels, 3 kernels) followed by two linear layers with dimension 1024 and 10. We perform max pooling after two of the convolutional layers. We also implement a small residual network (He et al. 2015) with 4 blocks containing 1 layer each and channel sizes of 32.

We find that the residual network is much more stable in training than the plain convolutional network, which collapsed when learning rate exceeded roughly 2×10^{-5}. The residual network allowed for the highest stable learning rate (2×10^{-3}) of any model. Residual networks augment

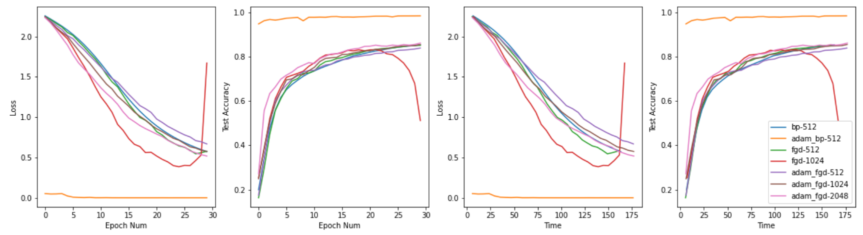

Figure 3: Training MLP with different hidden layer sizes, and optimizers, learning rate = 2×10^{-4}

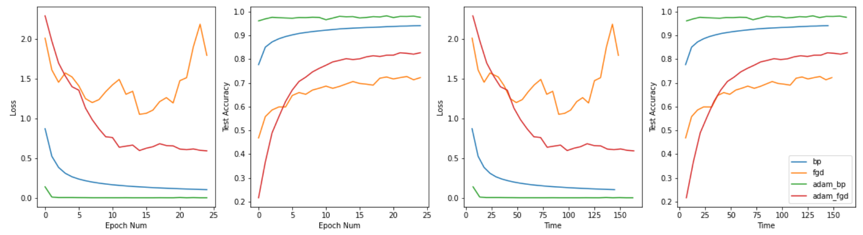

Figure 4: Training MLP with batch normalization, hidden size 512, learning rate = 2×10^{-4}

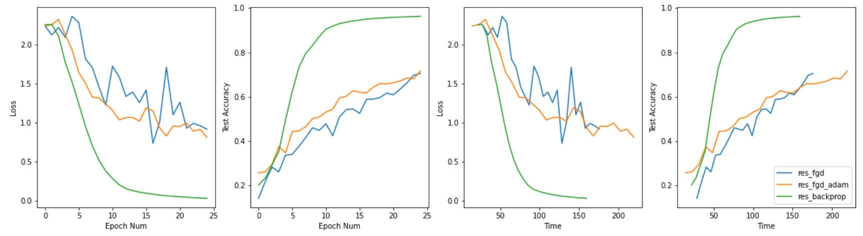

Figure 5: Training a residual network with forward gradient, using SGD and Adam, and a model trained with backpropagation and the Adam optimizer for reference. Learning rate = 2×10^{-3}

convolutional networks with residual connections and batch normalization, but while we found that batch normalization sped up initial training with MLPs, it did not improve learning rate sensitivity for resnet.

5 Conclusion

Under low learning rates for simple tasks, forward gradients achieves the same performance as backpropagation. However, training with backpropagation is much more stable than with the forward gradient, especially as model size and complexity increases. Thus, on the same task backpropagation will generally outperform the forward gradient overall due to the ability to select better hyperparameters. The increased stability also allows backpropagation to train more complex models.

The primary advantage of the forward gradient over backpropagation is the reduction in the overall compute time required per pass of the model, as the backward pass is eliminated. Although forward gradients are computationally cheaper to compute, models converge much slower with the best hyper-parameters due to the need for a lower learning rate in comparison to backpropagation. Additionally, backpropagation seems to converge at a higher training accuracy overall than the forward gradient in most scenarios, further weakening its case.

We found that with the forward gradient common deep learning optimization techniques do little to improve the performance of the forward gradient, limiting its ability to match the performance of backpropagation with these augmentations. Further work may be done on developing optimization methods that are specifically developed for use with the forward gradient. With the introduction of increasingly large and complex models the forward gradient does not currently appear to be a viable alternative to backpropagation.

Contributions

All group members contributed to experiments, code, and the report equally.

References

Baydin, Atılım Güneş et al. (2022). “Gradients without Backpropagation”. In: DOI: 10.48550/ARXIV.2202.08587. URL: https://arxiv.org/abs/2202.08587.

Bradbury, James et al. (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. URL: http://github.com/google/jax.

He, Kaiming et al. (2015). “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

Hennigan, Tom et al. (2020). *Haiku: Sonnet for JAX*. Version 0.0.3. URL: http://github.com/deepmind/dm-haiku.

Ioffe, Sergey and Christian Szegedy (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167. arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

Labach, Alex, Hojjat Salehinejad, and Shahrokh Valaee (2019). “Survey of Dropout Methods for Deep Neural Networks”. In: *CoRR* abs/1904.13310. arXiv: 1904.13310. URL: http://arxiv.org/abs/1904.13310.

LeCun, Yann and Corinna Cortes (2010). “MNIST handwritten digit database”. In: URL: http://yann.lecun.com/exdb/mnist/.

Lee, John M. (2000). *Introduction to Smooth Manifolds*.

Paszke, Adam et al. (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Silver, David et al. (2022). “Learning by Directional Gradient Descent”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=5i7lJLuhTm.