An Efficient GPU Implementation of Natural Evolution Strategies for Deep Learning

Anonymous authors
Paper under double-blind review

Abstract

In Natural Evolution Strategies, the weights of a neural network are perturbed with random noise in order to evaluate the effect on the objective function. This poses a heavy memory bottleneck for training large batches, as each batch element requires sampling and storing noise tensors the size of the network itself. Consequently, previous implementations mostly rely on CPU clusters for parallelization, with only a few batch elements per CPU. We present several methods for reusing noise between batch elements. As a result, memory usage is greatly reduced, enabling efficient batched training on the GPU. We demonstrate 2-10x speedups over IID sampling. This puts NES within the reach of users without access to CPU clusters, and opens up new scaling possibilities for GPU or TPU clusters. We open source a reference PyTorch library implementing these methods for fully-connected and convolutional layers.

1 Introduction

Backpropagation is the dominant method for training neural networks. Despite having been introduced in Rumelhart et al. (1986), or possibly earlier, it faced many practical barriers that were not overcome until much later, when a number of supporting improvements and innovations were added. Though highly optimized GPU implementations of backpropagation are now taken for granted, Krizhevsky et al. (2012) went to great pains to create an efficient GPU implementation in order to train AlexNet. This played a large role in its success and helped to kick off the deep learning era.

Natural Evolution Strategies (Wierstra et al., 2014) has also emerged as a promising method for training neural networks. Though it is much less efficient than backpropagation in continuous domains, it can be highly competitive when the gradient is difficult or impossible to calculate, such as in reinforcement learning (Salimans et al., 2017). Unfortunately, not as much time has been spent optimizing NES to run efficiently on the GPU, despite the fact that many of the necessary algorithmic/software ingredients can be reused from existing deep learning libraries. This places NES at a significant and unnecessary disadvantage, greatly limiting its potential.

Like backpropagation, NES requires processing large batches in parallel to provide a good gradient estimate. Previous implementations mostly use CPU clusters in order to train their models, which is undesirable for a number of reasons. Communication and synchronization costs can be significant, especially at scale, and each node must run its own copy of the code/neural network. GPUs have much higher raw compute abilities than CPUs and are more efficient in FLOPs per watt and per hardware dollar.

We provide an efficient GPU implementation that reduces the number of distinct nodes needed for a desired level of compute. For small to medium size applications, users may be able to obtain satisfactory computational performance on a single node, avoiding the various difficulties and expenses of obtaining access to a CPU cluster and distributing tasks across it.

At the core of our implementation are several sampling methods, which we present here. These methods reuse noise between batch elements without reducing sample quality. These methods greatly reduce the memory requirements of NES, and also provide speed improvements of up to 10 times over a baseline implementation.

2 Related Work

Zhang et al. (2017) analyze NES on MNIST and find good scaling performance with many CPUs, but run into a bottleneck when calculating weight updates; each node only calculates the forward pass of a small fraction of the batch but must calculate weight updates for the entire batch. Both Zhang et al. (2017) and Salimans et al. (2017) resort to sparse perturbations to ease this bottleneck. Our implementation also offers sparsity, but is able to improve on the update operation in two additional ways. First, as noted before, performance per node is higher, meaning less redundant work is needed. Second, in our noise reuse methods, the update calculations themselves are less expensive.

Antithetic sampling is often used in NES as a variance reduction technique (Brockhoff et al., 2010; Mania et al., 2018; Salimans et al., 2017). However, as their implementations are CPU-based, they cannot effectively use the optimization for antithetic sampling that we use in our implementation.

Ailon & Rauhut (2013) and Dasgupta et al. (2011) use randomized Fourier or Hadamard transforms in order to approximate multiplication with a random matrix for various other problems; our sign flip sampling method closely resembles this approach. Their method could potentially be applied to our context, but is unlikely to offer significant practical improvements.

Such et al. (2018) use IID sampling for NES on the GPU, which we also implement as a baseline.

A number of related neuroevolutionary methods (Conti et al., 2018; Gajewski et al., 2019; Mania et al., 2018) also sample a population of neural networks whose weights are perturbed with Gaussian noise, and therefore also benefit from our implementation.

3 Perturbed Matrix Multiplication

Here we provide an abstract characterization of the operation of a fully connected layer during evolutionary training; this can easily be extended to other layers, such as convolution or multi-head attention. We define the base weight matrix as $\boldsymbol{W}$ with a rows and b columns; $\boldsymbol{X}$ is the input matrix, where the ith row corresponds to the set of incoming activations of the ith sample. $\boldsymbol{XW}$ is the standard matrix product calculated in an unperturbed forward pass.

1. The *sample step* samples random perturbations at the start of every batch. In the case of IID sampling, which is the default for NES, a batch of n noise matrices $\mathsf{E}_{i,:,:}$ are drawn independently from $\mathcal{N}(0, \sigma^2 \boldsymbol{I}_{a\times b})$. We also define $\mathsf{W}'_{i,:,:} := \boldsymbol{W} + \mathsf{E}_{i,:,:}$.
2. The *forward pass* computes the batch matrix-vector product $\boldsymbol{X}\mathsf{W}'$. Note that $\boldsymbol{X}\mathsf{W}' = \boldsymbol{XW} + \boldsymbol{X}\mathsf{E}$. We use this decomposition in our noise reuse methods to avoid explicitly constructing W'.
3. The *update step* receives a vector of coefficients $\boldsymbol{c}$ from the optimizer. The weights are updated by setting $\boldsymbol{W} = \boldsymbol{W} + \sum \boldsymbol{c}_i \mathsf{E}_{i,:,:}$.

In an iteration of vanilla NES, we run the sample step, then use the forward pass to calculate the objective function $\boldsymbol{r} = f(\boldsymbol{x}; \mathsf{W}')$. We then call the update function with $\boldsymbol{c}_i = \alpha(\boldsymbol{r}_i - \frac{1}{n}\sum_i \boldsymbol{r}_i)$, where α is the step size.

4 Sampling Methods

4.1 Requirements

Before explaining our noise reuse methods, we need to consider what properties the joint distribution of E should satisfy. In contexts such as Monte Carlo methods, sampling is done in order to estimate an expectation over a particular distribution. However, here we are merely interested in finding a gradient estimate, so our choice of sampling distribution is less restricted. To this end, we consider two principles for sample quality:

1. The marginal noise distribution of each sample should be identical, and the support should include all directions in parameter space, as the gradient estimate will be a linear combination of the samples.
2. The joint distribution (i.e. the correlation of samples in a batch) should be chosen to minimize the variance of the gradient estimate.

Although IID sampling fulfills these principles, sampling, storing, and batch-multiplying a full tensor of IID noise matrices leads to poor computational efficiency. This can be circumvented by utilizing sparse matrices and/or reusing noise in some way between samples.

Sparsity improves computational efficiency at the cost of compromising principle 1. Empirically, sample quality suffers with extreme sparsity (Salimans et al., 2017; Zhang et al., 2017), despite the fact that sparse noise still spans parameter space. It is likely that extreme sparsity has a high probability of not perturbing any important weights. Naively using the same noise matrix for many samples would compromise principle 2, so a more careful approach is needed.

4.2 Antithetic Sampling

In antithetic sampling, $E_i \sim \sigma N(0, I)$ for i odd and $E_i = -E_{i-1}$ for i even.

Besides variance reduction, antithetic sampling has an additional advantage: only one noise matrix needs to be stored for each antithetic pair. By using the decomposed forward pass $\boldsymbol{X}\mathbf{W}' = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{X}\mathbf{E}$, we are able to cut memory usage in half in our implementation of antithetic sampling. Despite the additional operations, our tests show that this forward pass is often faster than IID sampling, as both methods are mostly bottlenecked by the time spent loading weights into local memory.

4.3 Permutation and Sign flip Sampling

We extend the principle of noise reuse to permutation or sign flip sampling. We sample only a single noise matrix $\boldsymbol{E}$ for the entire batch. For each batch element, we also sample matrices $\boldsymbol{A}_i, \boldsymbol{B}_i$, drawn uniformly from the set of permutation matrices or the set of sign flip matrices (diagonal matrices with random $\{+1, -1\}$ elements on the diagonal). Then each $\boldsymbol{E}_i = \boldsymbol{A}_i\boldsymbol{E}\boldsymbol{B}_i$. Note that each $\boldsymbol{E}_i$ retains the original multivariate Gaussian marginal distribution. $\boldsymbol{X}\mathbf{E}$ can be calculated efficiently, using the associative property: First apply permutations/sign flips to each input, matrix multiply by $\boldsymbol{E}$, then apply permutations/sign flips to each output.

For large batch sizes, memory usage is dominated by the permutations/sign flips that correspond to the $\boldsymbol{A}_i, \boldsymbol{B}_i$. These are respectively a factor of b, a smaller than the noise matrices in IID sampling.

In the interests of speed, we can apply only $\boldsymbol{A}_i$ or only $\boldsymbol{B}_i$ instead of both, which we find to be unnecessary for maintaining sample quality. Sparsity is easily implemented by letting $\boldsymbol{E}$ have less than a rows and/or less than b columns, and using partial permutations. This effectively creates unique row/column sparsity masks for each $\boldsymbol{E}_i$.

We show in Appendix A that sample quality is not appreciably decreased by our noise reuse methods.

5 Performance Experiments

We demonstrate empirical speed tests using the implementation in the repository. Sign flip and permutation sampling, as well as our implementation of antithetic sampling, are novel. Testing was done with half precision on an NVIDIA 2070 GPU. We report the median of 100 runs.

We express forward pass times as multiples of the unperturbed forward pass time, which serves as an upper bound on performance.

We also analyze update step performance in Appendix B.

5.1 Fully Connected Layers

Batch size is set to 8192. Layer sizes are formatted as (input dimension, output dimension).

Table 1: Forward pass, fully connected

Method	Layer Size 256, 256	512, 512	512, 10
IID	28.52	42.96	13.04
Antithetic	14.82	22.4	6.26
Sign flip	**2.39**	2.46	**2.22**
Permutation	3.28	2.9	2.37
Permutation (50% sparsity)	3.02	2.23	2.37
Permutation (10% sparsity)	3.14	**2.08**	2.29
	x	x	x
Unperturbed	0.141 ms	0.354 ms	0.233 ms

Sign flip sampling consistently does well. Permutation sampling with sparsity is only faster for large layers.

5.2 Convolutional Layers

Batch size is set to 512. Layer sizes are formatted as (input channels, output channels, height x width). All layers use a 3x3 kernel.

Table 2: Forward pass, convolution

Method	Layer Size 3,64,32x32	64,64,32x32	128,128,16x16	256,256,8x8	512,512,4x4
IID	5.99	2.54	3.74	10.96	18.88
Antithetic	4.39	3.79	3.86	6.56	10.42
Sign flip	2.2	2.23	2.14	2.07	2.01
Permutation	1.78	2.27	2.15	2.09	2.05
Perm (50% sp.)	1.69	1.78	1.67	1.61	1.58
Perm (10% sp.)	**1.69**	**1.69**	**1.44**	**1.29**	**1.25**
	x	x	x	x	x
Unperturbed	1.161 ms	3.792 ms	2.918 ms	2.729 ms	2.871 ms

IID sampling does decently well as it is able to leverage instance-level spatial parallelism, though its memory usage makes it unusable in practice. Permutation sampling with sparsity is highly efficient.

6 Conclusion

In this paper, we identify and resolve the GPU memory bottleneck that until now has prevented high performance neural network training using Natural Evolution Strategies. We accomplish this by providing two noise reuse methods, sign flip and permutation sampling. These methods also ease the synchronization bottleneck, leading to a net gain in sample quality versus the highly sparse noise previously used to reduce synchronization costs. Further gains in speed and memory can be squeezed out using tricks like custom GPU kernels and bitpacking, but our basic implementations are already relatively close to the theoretical maximum in terms of speed. This allows NES to compete on a more fair hardware footing against gradient-based methods.

Practitioners often must balance performance and technical complexity when deciding whether or not to use an optimization. Our implementation provides a decrease in technical complexity, as we use only existing PyTorch functions, and we offer strong performance without any need to resort to distributed computing. We hope that this makes NES more accessible to the deep learning community.

References

Nir Ailon and Holger Rauhut. Fast and rip-optimal transforms, 2013.

Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V. Arnold, and Tim Hohm. Mirrored sampling and sequential selection for evolution strategies. In *Proceedings of the 11th International Conference on Parallel Problem Solving from Nature: Part I*, 2010.

Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents, 2018.

Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. Fast locality-sensitive hashing. In *KDD*, 2011.

Alexander Gajewski, Jeff Clune, Kenneth O. Stanley, and Joel Lehman. Evolvability es: Scalable and direct optimization of evolvability, 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning, 2018.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 1986.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2018.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 2014.

Xingwen Zhang, Jeff Clune, and Kenneth O. Stanley. On the relationship between the openai evolution strategy and stochastic gradient descent, 2017.

A Sample Diversity

We analyze several measures of sample diversity for sign flip sampling when only column signs are flipped (i.e. $\boldsymbol{B}_i$ are sampled from sign flips). This analysis is for a single layer; if column signs are flipped for all layers then b should be summed over all layers. There are 2^b possible flip vectors, so the probability of a collision is small. The column vectors of $\boldsymbol{E}$ span a subspace of dimension b almost surely, out of the entire space of dimension ab, so the probability of linear dependence among samples is $\leq 2^{n-b}$. The variance of the dot product of two samples is $(2 + b/4)$ times larger than for two IID samples. (The first is the sum of a independent chi-squared distributions with b degrees of freedom times a random sign, and the second is the sum of ab independent products of IID Gaussians).

Zhang et al. (2017) provides an analysis on MNIST that strongly suggests that NES works by approximating the SGD gradient. That is, the performance of NES is mostly a function of its correlation with the SGD gradient. Furthermore, the correlation remains fairly consistent over the course of training. This provides a straightforward way to test sample quality, as we can measure the average correlation of the NES gradient estimate under different sampling methods to the SGD gradient. We do not observe any lower correlation when using our noise reuse methods, even for permutations with 1/16th sparse weights.

B Update Step Performance

We provide timings for the update steps using the same settings as in Section 5. We are able to substantially speed up the update step for the sign flip and permutation sampling methods by using the associative property when calculating $\sum \boldsymbol{c}_i \boldsymbol{E}_i$. A fast update step is important for problems where the forward pass is not called many times per batch iteration or when many parallel nodes are used.

Table 3: Update, fully connected (milliseconds)

	Layer Size		
Method	256, 256	512, 512	512, 10
IID	4.44	14.314	0.41
Antithetic	1.887	7.188	**0.281**
Sign flip	**0.314**	**0.325**	0.329
Permutation	2.674	5.092	0.537
Permutation (50% sparsity)	1.501	2.595	0.531
Permutation (10% sparsity)	0.656	0.907	0.483

Table 4: Update, convolution (milliseconds)

	Layer Size				
Method	3,64,32x32	64,64,32x32	128,128,16x16	256,256,8x8	512,512,4x4
IID	0.359	0.521	1.611	6.243	29.101
Antithetic	**0.272**	**0.273**	0.33	1.113	4.397
Sign flip	0.329	0.324	**0.318**	**0.288**	**0.322**
Permutation	0.546	0.59	0.659	0.64	0.818
Perm (50% sp.)	0.482	0.598	0.633	0.587	0.686
Perm (10% sp.)	0.516	0.58	0.564	0.544	0.625

C PSEUDOCODE

We present pseudocode for the various sampling methods mentioned. Bias terms have been omitted for simplicity.

Algorithm 1: BATCH MATRIX MULTIPLICATION

```
Input :
    $A : [n \times a \times b]$
    $B : [n \times b \times c]$
Output:
    $C : [n \times a \times c]$
    $\forall i\, C[i] = matmul(A[i], B[i])$
```

Algorithm 2: IID SAMPLING

```
$W' : [n \times a \times b]$
Sample()
    for $i \in [n]$ do
        $W'[i] = \sigma * randnormal(a \times b) + W$
    end
Forward($X$:$[n \times a]$)
    return $batchmm(X.reshape(n \times a \times 1), W')$
Update($c$:$[n]$)
    $U = zeros(a \times b)$
    for $i \in [n]$ do
        $U = U + c[i] * (W'[i] - W)$
    end
    return $U$
```

Algorithm 3: ANTITHETIC SAMPLING

```
$E : [n/2 \times a \times b]$
Sample()
    for $i \in [n/2]$ do
        $E[i] = \sigma * randnormal(a \times b)$
    end
Forward($X$:$[n \times a]$)
    $B = matmul(X, W)$
    $R = batchmm(X.reshape(n/2 \times 2 \times a), E)$
    $R[:, 1, :] = -R[:, 1, :]$
    return $B + R.reshape(n \times b)$
Update($c$:$[n/2]$)
    $U = zeros(a \times b)$
    for $i \in [n/2]$ do
        $U = U + c[i] * E[i]$
    end
    return $U$
```

Algorithm 4: PERMUTATION SAMPLING

```
$E : [a \times b]$
$\pi_{in} : [n \times a]$
$\pi_{out} : [n \times b]$
Sample()
    $E = \sigma * randnormal(a \times b)$
    if permute inputs then
        for $i \in [n]$ do
            $\pi_{in}[i] = randperm(a)$
        end
    endif
    if permute outputs then
        for $i \in [n]$ do
            $\pi_{out}[i] = randperm(b)$
        end
    endif
Forward($X$:$[n \times a]$)
    $B = matmul(X, W)$
    if permute inputs then
        $X = \pi_{in} \circ X$
    endif
    $R = matmul(X, E)$
    if permute outputs then
        $R = \pi_{out} \circ R$
    endif
    return $B + R$
Update($c$:$[n]$)
    [See repository]
```

Algorithm 5: SIGN FLIP SAMPLING

```
$E : [a \times b]$
$\pi_{in} : [n \times a]$
$\pi_{out} : [n \times b]$
Sample()
    $E = \sigma * randnormal(a \times b)$
    **if** *flip inputs* **then**
        $\pi_{in} = randsign(n \times a)$
    **endif**
    **if** *flip outputs* **then**
        $\pi_{out} = randsign(n \times b)$
    **endif**
Forward(X:$[n \times a]$)
    $B = matmul(X, W)$
    **if** *flip inputs* **then**
        $X = \pi_{in} * X$
    **endif**
    $R = matmul(X, E)$
    **if** *flip outputs* **then**
        $R = \pi_{out} * R$
    **endif**
    **return** $B + R$
Update(c:$[n]$)
    [See repository]
```