Backprop Considered Harmful? The Hybrid-Evolution Strategy for Supervised Learning Training

Andrew Bartolo
bartolo@stanford.edu

Abstract— **Modern supervised learning training typically involves (mini-batch) gradient descent, using backpropagation to update parameters at each iteration. The backpropagation step is very computationally intensive, and parallelizing it across discrete worker nodes involves transferring potentially hundreds of megabytes of parameters for each training iteration. This work assesses the feasibility of replacing traditional backprop with evolution strategies, a stochastic optimization technique more commonly used in reinforcement learning. By using evolution strategies in lieu of traditional backprop, we might drastically reduce the amount of data transferred over the network at each training iteration (thus reducing overall training time).**

For a smaller test network (multilayer perceptron on MNIST[1]), we found that, in standard minibatch GD, the weights gradients did not change much in magnitude over the course of training. (They did change in angle from iteration to iteration.) We then implemented the full Hybrid-ES algorithm, and performed a hyperparameter search to optimize it. We also created a model to demonstrate the runtime, memory usage, and network bandwidth utilization of Hybrid-ES vs. standard batch GD. Finally, we evaluated Hybrid-ES on a larger ConvNet, but found that it required more hyperparameter tuning to achieve acceptable accuracy.

I. INTRODUCTION

Modern supervised learning training typically involves (mini-batch) gradient descent, using backpropagation to update parameters at each iteration. The backpropagation step is very computationally intensive, and requires 1.) computing many partial derivatives across the computation graph, and 2.) caching these partials in memory for subsequent parameter updates. Parallelized gradient descent involves worker nodes computing gradients on each mini-batch, but *then* each worker node must synchronize the updated parameters across the entire cluster. To do this, the weights and biases must be broadcast across the network, which can involve hundreds of megabytes transferred.

In this work, we propose a modification of *evolution strategies*, a technique from reinforcement learning, for use in supervised learning. For the first training iteration in a *cycle*, the full analytical backprop gradient is calculated. Then, for the rest of the cycle, the gradient is updated stochastically via a series of k small random perturbations. We evaluate the cost function on each of these, and choose the best perturbed gradient as the new gradient going forward. The primary benefit of this approach is that parallel workers can deterministically generate the updated gradients using only two scalars: 1.) the seed for the PRNG used to generate the best (lowest cost) randomly-perturbed matrix, and 2.) the cost for this matrix. Each worker node chooses the the lowest-cost seed broadcast to it to reconstruct the matrix.

II. EXPLORING THE GRADIENT

Before implementing the Hybrid-ES algorithm itself, we did some exploratory work to see how the gradients behaved across training iterations.

A. Base Network

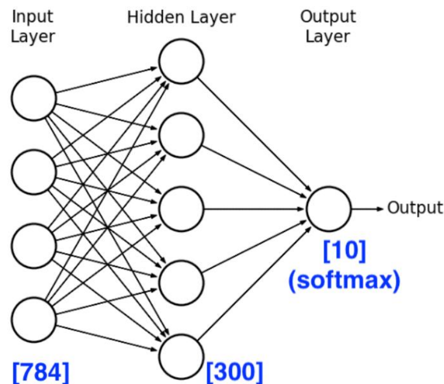

Fig. 1. The network used to perform most algorithm design exploration.

The base network is a 2-layer MLP with 300 hidden units, and a 10-class softmax output layer. MNIST uses 28x28 grayscale images, so the input layer is 784 pixels, each containing a value 0-255 [1]. This network trained over 30 epochs each of 50 iterations, so 1,500 iterations total. L2 regularization was used with $\lambda = 10^{-4}$, and a fixed learning rate of $\alpha = 5$. Minibatch size was set to 1000. Experiments were performed using 50,000 training set examples, and 10,000 dev set examples.

B. Gradient behavior: magnitude & angle

Initially, we wanted to see how the gradient changed over the course of training, both in magnitude and direction.

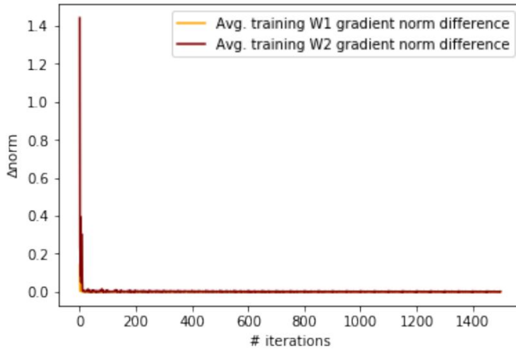

Fig. 2. Unmodified network gradient norm divergence.

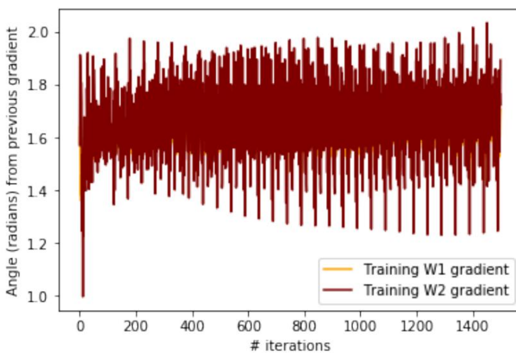

Fig. 3. Unmodified network gradient angle divergence.

From Figure 2, we can see that once training proceeds past the first few iterations, the gradients change very little in magnitude. (The initial spike at the beginning is most likely an artifact due to setting the "previous" gradient to all-ones for iteration 0.) This lack of change is likely due in large part to L2 regularization.

Figure 3 was generated by flattening the neurons' gradients into a big vector, caching that vector from iteration to iteration, and computing the cosine similarity between the vectors at iterations t and $t-1$. For comprehensibility, the angle itself (in radians) is plotted, and not the cosine. We can see that the gradients do change in direction from iteration to iteration. We surmise that the periodicity of the graph is due to cycling over minibatches repeatedly.

Together, these graphs indicate that, for hybrid-ES, 1.) Whatever random shift we apply to the initial gradient shouldn't be too large in magnitude, and 2.) since the gradients change direction frequently anyway, the randomization might not be a bad idea. However, there seems to be a lot of information in the periodicity/pattern of the angle change, and we unfortunately don't make heavy use of this information by just applying a shift sampled from the normal distribution. In the future, we'd like to try learning this pattern somehow and using it to inform the shift matrices.

C. Re-using the same gradient

As another exploratory exercise, we asked the following: What would happen if we fixed the gradient at its epoch-0 value, and used it to make updates for the entirety of training? The results from this experiment are shown below. Note that this gradient was computed across the entire training set (50 iterations), but only on the first epoch.

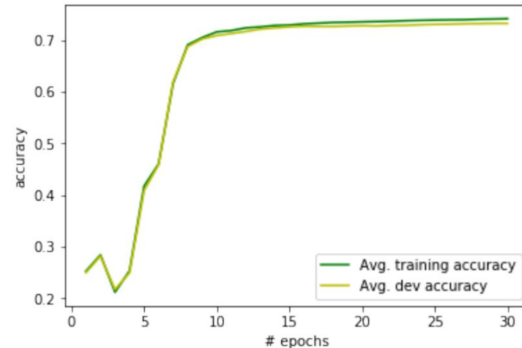

Fig. 4. Fixed-gradient accuracies.

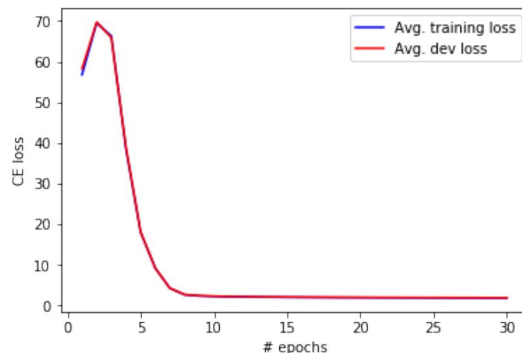

Fig. 5. Fixed-gradient losses.

We see that, compared to the unmodified reference network, both accuracies suffer a decent amount. However, this really isn't too bad, considering that the gradient is only calculated once. The network is still making forward progress in training. This is encouraging, as it suggests we can get away with computing the full gradient less frequently, and updating it slightly on each iteration.

III. HYBRID-ES

Below, we define a standard "off-the-shelf" parallel batch gradient descent algorithm, and below that, the Hybrid-ES algorithm. Note that in some cases, notation has been overloaded, as the `transmit`, `receive`, and `combine` functions work differently in standard parallel BGD and Hybrid-ES.

In parallel BGD, we first partition the training set up so that each node can train over a subset. Then, each node performs forward prop, backprop, and a tentative update over

Algorithm 1 Parallel Batch GD

```
1: Split training set T into N subsets, T_n.
2: for every iteration i, each worker node do
3:     forward_prop(T_n, θ)
4:     backprop(T_n, θ)
5:     θ_n := θ_n − α dθ_n
6:
7:     transmit(θ_n)
8:     receive(θ_{1...n−1,n+1...N})
9:     θ = combine(θ_{1...N})
```

Algorithm 2 Parallel Hybrid-ES

```
1: if iteration i % r == 0 then                      ▷ Full update
2:     forward_prop(T, θ)
3:     backprop(T, θ)
4:     θ := θ − α dθ
5:
6:     transmit(θ_n)
7:     receive(θ_{1...n−1,n+1...N})
8:     θ := combine(θ_{1...N})
9:
10: else                                             ▷ Stochastic update
11:     for K attempts, each worker node do
12:         dθ_{n,k} := dθ + N(0, σ^2)               ▷ Sample random shift
13:         θ_{n,k} := θ − α dθ_{n,k}
14:     θ_n := arg min_cost forward_prop(T, θ_{n,k})
15:
16:     transmit(rseed_n, best_cost_n)
17:     receive(rseed_{1...n−1,n+1...N}, best_cost_{1...n−1,n+1...N})
18:     θ := combine(rseed_{1...N}, best_cost_{1...N})
```

that subset. Now, the nodes must exchange parameters in order to arrive at the combined updated θ (from all nodes). This is done in lines 7-9, and here, the full parameters must be exchanged over the network. (This doesn't necessarily involve an all-to-all broadcast: some tree-like form of routing/replication might be used.) Regardless, a lot of data is being sent at each iteration. In this algorithm, the `combine` function could be a simple averaging of the parameters received from all nodes, or it could be something more elaborate.

Note that for Parallel Hybrid-ES, each worker node needn't necessarily train over a subset of the training set (they could, though the algorithm is agnostic to training set partitioning). The "Hybrid" in Hybrid-ES comes from the fact that we're still computing the full backprop gradient once every r iterations. (r is a key hyperparameter that we'll delve into more later.) For these "full" iterations, parameter broadcasting and updates work much like they do in the standard algorithm.

However, for non-r-th iterations, the stochastic update occurs. For K times, each node samples from a normal distribution centered at 0 and with variance σ^2 (another key hyperparameter) and adds this random shift to its previous gradient value. Each shifted gradient is used to update a "temp" θ, which we compute the cost function over. The best such randomly-shifted θ is chosen for broadcast. However, here, instead of broadcasting the entire weights and bias matrices, each node need only broadcast two scalars: 1.) the random seed to generate the best shift, and 2.) the cost of the best-shift parameters. From this information, each worker node can reconstruct the best stochastic update.

IV. Evaluation

We first assessed an initial "assisted" version of the hybrid evolution strategy algorithm on the MNIST MLP. This algorithm calculates the full gradient at the start of each iteration, and then perturbs it slightly K times. This incarnation of the algorithm doesn't save anything over traditional parallel BGD - in fact, it requires more computation as it's noising the gradients at each step. However, it served as a milestone for checking that the noised gradients weren't adversely affecting accuracies.

	Train	Dev
Reference	99.54%	96.57%
Assisted H-ES	99.85%	96.08%

Fig. 6. Assisted H-ES vs. reference.

From this, it seems the added randomization caused the algorithm to overfit the training set a bit. This makes some sense, as the threshold for deciding whether a random perturbation was better than the unperturbed gradient is the cost function computed over a *training set* minibatch (since the algorithm makes no assumptions about worker nodes' access to the *dev set*). However, since the dev set is much smaller relatively, it may make sense to just keep a copy of it on each node for this purpose.

V. Hyperparameters Definitions and Search

The Hybrid-ES algorithm adds the following hyperparameters to the model:

- K, the number of random perturbations tried by each worker node upon each iteration.
- r - the interval for performing a full backprop gradient calculation, as opposed to stochastic. Higher values of r mean the full gradient is computed less frequently.
- (μ, σ^2) for the random shift matrices. Initially, we set $\mu = 0$ and $\sigma^2 = 0.1$ (more on this later).

The number of worker nodes present, N, might also be considered a hyperparameter. However, for the remainder of the work, we let K be the sole parallelization factor in the algorithm.

r is particularly important, as it both influences the accuracy of the H-ES algorithm, and also defines the resource savings of H-ES over BGD. Consider the following results for varying values of r (each with $\sigma^2 = 1.25$):

	Train	Dev
Reference	99.54%	96.57%
Assisted H-ES (r = 1)	99.85%	96.08%
H-ES, r = 2	96.80%	94.21%
H-ES, r = 3	95.34%	93.06%
H-ES, r = 4	94.21%	92.23%

Fig. 7. Algorithm accuracies, varying r.

As we can see, increasing the full-backprop interval results in a fall-off of both the training set and dev set accuracies. This makes sense, as we don't expect the stochastic gradients to be as dialed-in as the analytical gradient. For r much greater than 3, Hybrid-ES becomes uncompetitive with the reference implementation, but of course the definition of acceptable accuracy can vary from task to task. For some tasks, the large resource savings may be worth it (more on this later).

Concerning the $N(0, \sigma^2)$ shift matrices, there is no real theoretical justification for sampling from a normal distribution. Indeed, the strong periodicity of Figure 3 means that a smarter means of sampling may produce better stochastic gradients. Assuming Gaussian noise, a mean of 0 is natural, and initially, $\sigma^2 \approx 0.1$ came from an empirical observation of individual gradient components early-on in training. (It turned out that a much higher value of $\sigma^2 = 1.25$ was the optimal setting.) In the future, it may be interesting to consider setting σ^2 with momentum or a similar scheme, as the gradients' individual components may also stabilize in magnitude as training proceeds.

Here are results from varying values of σ^2 (each with $r = 2$):

	Train	Dev
Reference	99.54%	96.57%
H-ES, σ^2 = 0.01	96.49%	93.58%
H-ES, σ^2 = 0.1	96.54%	93.73%
H-ES, σ^2 = 0.5	96.72%	94.01%
H-ES, σ^2 = 0.75	96.69%	93.98%
H-ES, σ^2 = 1.0	96.78%	94.16%
H-ES, σ^2 = 1.25	96.80%	94.21%
H-ES, σ^2 = 1.5	96.86%	94.09%

Fig. 8. Algorithm accuracies, varying σ^2.

As we can see, $\sigma^2 = 1.25$ seems to be the optimal fixed variance (per dev set accuracy), though we expect this specific number to depend strongly on the underlying network. Note that $\sigma^2 = 1.0$ is also competitive: this might be due to an effect of different σ^2s being better for earlier vs. later training stages. In the future, we hope to simultaneously sweep σ^2 while also sweeping r for a more complete analysis. Figures 9 and 10 are the loss and accuracy plots for $K = 10$, $r = 2$, and $\sigma^2 = 1.25$, demonstrating H-ES's training progress.

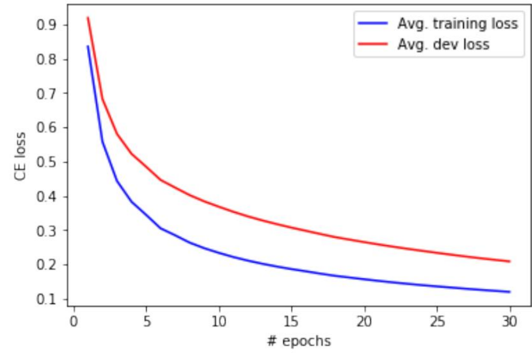

Fig. 9. H-ES training losses.

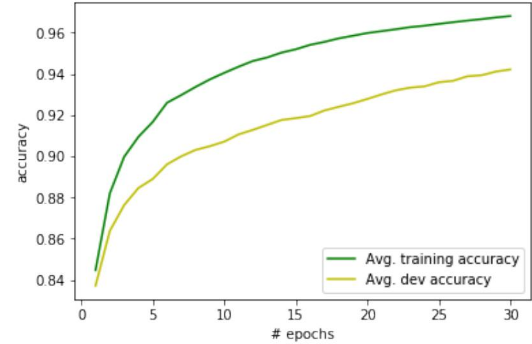

Fig. 10. H-ES training accuracies.

As a final exploratory exercise, we also considered normalizing the gradients to length one before shifting. We surmised that this might help, given that we had fixed σ^2 (so forcing the gradient to start out at a certain magnitude might homogenize the effect of adding the fixed-variance random shifts). As we see in Figure 11, normalizing the gradients adversely affected both accuracies, so we didn't continue with it. Interestingly, it also caused the training and dev set accuracies to converge, so it seemed to serve as a (very poor) regularizer. Normalizing the gradient is rarely performed in practice.

	Train	Dev
Reference	99.54%	96.57%
Normalized reference	93.69%	93.23%

Fig. 11. Reference vs. normalized reference.

Apparently, the gradients' magnitudes are important, even if they don't change much across iterations (see Figure 2).

VI. PERFORMANCE AND RESOURCE UTILIZATION MODEL

As the primary goal of H-ES is approximating BGD with lower overheads, we created a resource utilization model and analyzed it for varying values of r. We model wall-clock runtime, memory consumption, and network bandwidth (note that TX and RX bandwidth can be considered the same

across an entire cluster: every sent packet must be received somewhere). Here are the results in graphical and tabular form:

In general, the resource models follow the pseudocode given in the Algorithms section. See the provided source code for details. Runtime was measured empirically using Python's time.time() method. Memory consumption was calculated based on a liveness analysis of variables in the algorithms' methods, and the size of the matrices. Network bandwidth was similarly calculated using matrix sizes.

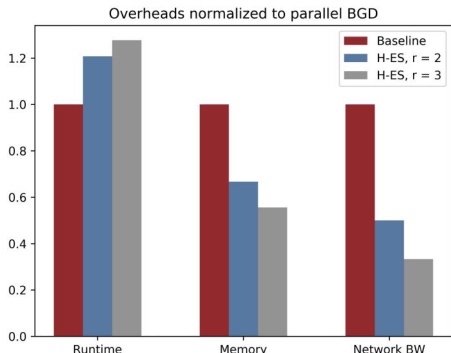

Fig. 12. H-ES resource utilization graph.

	Runtime	Memory	Net BW
H-ES, r = 2	120.8%	66.7%	50.0%
H-ES, r = 3	127.7%	55.6%	33.3%

Fig. 13. H-ES resource utilization table.

It is interesting to note that in the results above, H-ES actually loses out in runtime versus the baseline model! However, we believe this is due to artificial limitations in our testing setup. First, we assume that sending data over the network is instantaneous (takes 0 time), which likely leads to a conservative estimate of H-ES's runtime savings. Second, we measured the wall-clock time taken by forward prop and backprop, and found that, for our small net, forward prop actually takes slightly longer (which we don't expect to be the case for a larger net). This may be due to a cold-cache effect, since we perform forward prop first (for backprop, the data is already loaded into the cache). See the model source code for details.

For memory overheads, we indeed see H-ES winning out (as it doesn't need to store all the chain-rule gradient components). Likewise, H-ES saves significantly on network bandwidth, essentially by a factor of $\frac{1}{r}$ (since each node only needs to send two scalars for the stochastic updates).

Finally, note that the efficiency of H-ES does depend on the relative efficiency of generating pseudorandom numbers. However, both modern CPUs (via RDRAND [5]) and modern GPUs (via cuRAND [6]) include hardware-accelerated PRNGs, so we expect PRNG generation within a node to be faster than broadcasting huge parameter matrices over a network. In the future, we plan to measure this empirically.

VII. HYBRID-ES WITH A CONVNET

Though most of the design and evaluation of H-ES was performed with the MLP on MNIST, we also wanted to see how it behaved on a convolutional neural net. We evaluated Assisted H-ES on a small MNIST ConvNet, with 5x5x6 CONV $\rightarrow$ 2x2 MAXPOOL $\rightarrow$ 5x5x16 CONV $\rightarrow$ 2x2 MAXPOOL $\rightarrow$ tanh activation $\rightarrow$ 10-class softmax output layer. The code (in `convnet-h-es.py`) uses the Autograd [7] NumPy automatic differentiation framework. This grants us the easy ability to take gradients without the full API overhead of something like TensorFlow [8] or Keras [9].

Unfortunately, the increased size of the net made training more intensive. In the interest of time and space constraints for this paper, we omit a detailed hyperparameter search, and instead highlight the number of "improvements" (lower-cost stochastic updates, with cost computed over the entire dev set) that Assisted H-ES was able to make in its training batches. Here, each batch contains 256 examples, with $K = 10$ and $\sigma^2 = 1.0$:

Training batch index #	# improvements
1	5
2	10
3	6
4	2
5	4

Fig. 14. ConvNet Assisted H-ES, number of improved updates.

As we can see, even minus any hyperparameter search, Assisted H-ES can make useful updates.

VIII. RELATED WORK

Much of the inspiration for this work was taken from OpenAI's Evolution Strategies works [2], [3], as here we adapted ES to a supervised learning setting. OpenAI's method also uses Gaussian noise for the random shifts. [4] provides a nice summary of several black-box optimization techniques with examples. Yann LeCun's MNIST dataset [1] was useful in tuning the H-ES algorithm.

IX. FUTURE WORK

After completing this initial exploratory work, the most interesting avenue for further progress seems to be extracting information from the periodicity of the gradient. Gaussian-shifting the matrices is simple, but it potentially throws away a lot of information captured by that periodicity. Future investigations will center on the gradients' behaviors: specifically, how we can use that information to adjust the gradients in a more precise direction. Finally, recall that the overall goal of the work is not to produce an algorithm that is more accurate than traditional backprop; rather, its goal is to approximate backprop with something computationally cheaper and *more parallelizable*. To this end, we hope to demonstrate H-ES running across a real (and possibly heterogeneous) parallel compute cluster, and measure its resource savings entirely empirically.

We are excited by our initial findings, and will work to improve the H-ES algorithm in the near future.

X. CONTRIBUTIONS

Andrew (Andy) performed all experiments, coding, and report writing. Andy's roommate, Sabeek Pradhan, suggested investigating the OpenAI ES method [2] [3] for supervised learning.

Code for the experiments is available on GitHub, at `https://github.com/andrewbartolo/CS230-Project`. Here, you'll find code for the MNIST MLP base network, the runtime, memory, and network bandwidth model, and the Autograd ConvNet Hybrid-ES code.

REFERENCES

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998. Dataset from http://yann.lecun.com/exdb/mnist/.

[2] Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *OpenAI blog*. https://blog.openai.com/evolution-strategies/

[3] Evolution Strategies as a Scalable Alternative to Reinforcement Learning. (full paper) https://arxiv.org/abs/1703.03864

[4] Gradient-Free Optimization. *Stanford AA222*. http://adl.stanford.edu/aa222/lecture_notes_files/chapter6_gradfree.pdf.

[5] Intel Digital Random Number Generator (DRNG) Software Implementation Guide. *Intel Corporation*. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide.

[6] The NVIDIA CUDA Random Number Generation library (cuRAND). *Nvidia Corporation*. https://developer.nvidia.com/curand.

[7] Autograd. Dougal Maclaurin, David Duvenaud, and Matt Johnson. *HIPS*. https://github.com/HIPS/autograd.

[8] TensorFlow: A System for Large-Scale Machine Learning. Abadi et. al. https://tensorflow.org.

[9] Keras: The Python Deep-Learning library. https://keras.io.