FLIPOUT: EFFICIENT PSEUDO-INDEPENDENT WEIGHT PERTURBATIONS ON MINI-BATCHES

Anonymous authors
Paper under double-blind review

ABSTRACT

Stochastic neural net weights are used in a variety of contexts, including regularization, Bayesian neural nets, exploration in reinforcement learning, and evolution strategies. Unfortunately, due to the large number of weights, all the examples in a mini-batch typically share the same weight perturbation, thereby limiting the variance reduction effect of large mini-batches. We introduce flipout, an efficient method for decorrelating the gradients within a mini-batch by implicitly sampling pseudo-independent weight perturbations for each example. Empirically, flipout achieves the ideal linear variance reduction for fully connected networks, convolutional networks, and RNNs. We find significant speedups in training neural networks with multiplicative Gaussian perturbations. Flipout allows us to vectorize evolution strategies, so that a single GPU with flipout can handle the same throughput at least as 40 CPU cores under existing methods, equivalent to a factor-of-4 cost reduction on Amazon Web Services.

1 INTRODUCTION

Stochasticity is a key component of many modern neural net architectures and training algorithms. The most widely used regularization methods are based on randomly perturbing a network's computations (Srivastava et al., 2014; Ioffe & Szegedy, 2015). Bayesian neural nets can be trained with variational inference by perturbing the weights (Graves, 2011; Blundell et al., 2015). Weight noise was found to aid exploration in reinforcement learning (Plappert et al., 2017; Fortunato et al., 2017). Evolution strategies minimizes a black-box objective by evaluating many weight perturbations in parallel, and have recently seen impressive performance on robotic control tasks (Salimans et al., 2017).

Some methods perturb a network's activations (Srivastava et al., 2014; Ioffe & Szegedy, 2015), while others perturb its weights (Graves, 2011; Blundell et al., 2015; Plappert et al., 2017; Fortunato et al., 2017; Salimans et al., 2017). Stochastic weights are appealing in the context of regularization or exploration because they can be viewed as a form of posterior uncertainty about the parameters. However, compared with stochastic activations, they have a serious drawback: because a network typically has many more weights than units, it is very expensive to compute and store separate weight perturbations for every example in a mini-batch. Therefore, stochastic weight methods are typically done with a single sample per mini-batch. In contrast, activations are easy to sample independently for different training examples within a mini-batch. This allows the training algorithm to see orders of magnitude more perturbations in a given amount of time, and the variance of the stochastic gradients decays as $1/N$, where N is the mini-batch size. We believe this is the main reason stochastic activations are far more prevalent than stochastic weights for neural net regularization. In other settings, such as Bayesian neural nets and evolution strategies, one is forced to use weight perturbations and live with the resulting inefficiency.

In order to achieve the ideal $1/N$ variance reduction, the gradients within a mini-batch need not be independent, but merely uncorrelated. In this paper, we present flipout, an efficient method for decorrelating the gradients between different examples while still giving unbiased gradient estimates. It applies to any perturbation distribution which factorizes by weight and is symmetric around 0 — including DropConnect, multiplicative Gaussian perturbations, evolution strategies, and variational Bayesian neural nets — and to a variety of architectures, including fully connected nets, convolutional nets, and RNNs.

In Section 3, we introduce flipout, show that it gives unbiased stochastic gradients, and discuss its efficient vectorized implementation which incurs only a factor-of-2 computational overhead compared with shared perturbations. We then analyze the asymptotics of gradient variance with and without flipout, demonstrating strictly reduced variance. In Section 4, we measure the variance reduction effects on a variety of architectures. Empirically, flipout gives the ideal $1/N$ variance reduction in all architectures we have investigated, just as if the perturbations were done fully independently for each training example. We demonstrate that Flipout achieves competitive performance in regularizing LSTMs, compared to several methods that use dropout. We demonstrate significant speedups in training time in a large batch regime. Finally, we apply flipout to vectorize evolution strategies (Salimans et al., 2017), allowing a single GPU to handle the same throughput under 40 CPU cores using existing approaches; this corresponds to a factor-of-4 cost reduction on Amazon Web Services.

2 BACKGROUND

2.1 WEIGHT PERTURBATIONS

We use the term "weight perturbation" to refer to a class of methods which sample the weights of a neural network stochastically at training time. More precisely, let $f(x, W)$ denote the output of a network with weights W on input x. The weights are sampled from some distribution $q_{\boldsymbol{\theta}}$ parameterized by $\boldsymbol{\theta}$. We aim to minimize the expected loss $\mathbb{E}_{(x,y)\sim\mathcal{D}, W\sim q_{\boldsymbol{\theta}}}[\mathcal{L}(f(x, W), y)]$, where $\mathcal{L}$ is a loss function, and $\mathcal{D}$ denotes the data distribution. The distribution $q_{\boldsymbol{\theta}}$ can often be described in terms of perturbations: $W = \overline{W} + \Delta W$, where $\overline{W}$ are the mean weights (typically represented explicitly as part of $\boldsymbol{\theta}$) and ΔW is a stochastic perturbation. We now give some specific examples of weight perturbations.

Gaussian perturbations. If the entries ΔW_{ij} are sampled independently from Gaussian distributions with variance σ_{ij}^2, this gives the distribution $q(W_{ij}) = \mathcal{N}(W_{ij}; \overline{W}_{ij}, \sigma_{ij}^2)$. Using the reparameterization trick (Kingma et al., 2015), this can be rewritten as $W_{ij} = \overline{W}_{ij} + \sigma_{ij}\epsilon_{ij}$, where $\epsilon_{ij} \sim \mathcal{N}(0, 1)$; this representation allows the gradients to be computed using backprop. A variant on this is *multiplicative* Gaussian perturbation, where the perturbations are scaled according to the weights: $W_{ij} \sim \mathcal{N}(\overline{W}_{ij}, \sigma_{ij}^2 \overline{W}_{ij}^2)$, or $W_{ij} = \overline{W}_{ij}(1 + \sigma_{ij}\epsilon_{ij})$, where again $\epsilon_{ij} \sim \mathcal{N}(0, 1)$. Multiplicative perturbations can be more effective than additive ones because the information content of the weights is the same, regardless of their scale.

DropConnect. DropConnect (Wan et al., 2013) is a regularization method inspired by dropout (Srivastava et al., 2014) which randomly zeros out a random subset of the weights. In the case of a 50% drop rate, this can be thought of as weight perturbation where $\overline{W} = W/2$ and each entry ΔW_{ij} is sampled uniformly from $\pm\overline{W}_{ij}$.

Variational Bayesian neural nets. Rather than fitting a point estimate of a neural net's weights, one can adopt a Bayesian approach of putting a prior distribution $p(W)$ over the weights and approximating the posterior distribution $p(W|\mathcal{D}) \propto p(W)p(\mathcal{D}|W)$, where $\mathcal{D}$ denotes the observed data. Graves (2011) observed that one could fit an approximation $q_{\boldsymbol{\theta}}(W) \approx p(W|\mathcal{D})$ using variational inference; in particular, one could maximize the evidence lower bound (ELBO) with respect to $\boldsymbol{\theta}$:

$$\mathcal{F}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{W\sim q_{\boldsymbol{\theta}}}[\log p(\mathcal{D} \mid W)] - \mathrm{D}_{\mathrm{KL}}(q_{\boldsymbol{\theta}} \,\|\, p).$$

The negation of the second term can be viewed as the description length of the data, and the negation of the first term can be viewed as the description length of the weights (Hinton & Van Camp, 1993). Graves (2011) observed that if q is chosen to be a factorial Gaussian, it can be thought of as Gaussian weight perturbation, except where the variance is adapted to maximize $\mathcal{F}$. (Note that the first term is proportional to the negative expected loss if we define $\mathcal{L} = -\log p(y|x)$.) Blundell et al. (2015) later combined this insight with the reparameterization trick (Kingma & Welling, 2014) to derive unbiased stochastic estimates of the gradient of $\mathcal{F}$.

Evolution strategies. Evolution strategies (ES) (Eigen, 1973) is black box optimization algorithms which involves a heavy use of weight perturbation. It doesn't scale well to large dimensional space so it has limited use in neural networks until recently proposed as an alternative reinforcement learning algorithm (Schmidhuber et al., 2007; Salimans et al., 2017). ES generates a collection of weight perturbations as candidates and evaluates them by some objective functions. The gradient of

the parameters can be estimated given the evaluation result. ES is a highly parallelizable algorithm, since generating and evaluating perturbations can be done independently. Suppose N is the number of workers, σ is the perturbation standard deviation and $\overline{W}$ is the model parameter. The algorithm is trying to maximize $\mathbb{E}_{\Delta W}\left[F\left(\overline{W}+\sigma\Delta W\right)\right]$, where $F(\cdot)$ is the objective function. The gradient of the objective function and the update rule can be given as:

$$\begin{aligned} \nabla_{\overline{W}} \mathop{\mathbb{E}}_{\Delta W}\left[F(\overline{W}+\Delta W)\right] &= \frac{1}{\sigma^2}\mathop{\mathbb{E}}_{\Delta W}\left[\Delta W F(\overline{W}+\Delta W)\right], \quad \text{where } \Delta W \sim \mathcal{N}(0,\sigma I) \\ \implies \overline{W}_{t+1} &= \overline{W}_t + \alpha\frac{1}{N\sigma^2}\sum_{i=1}^{N} F(\overline{W}_t+\Delta W_i)\Delta W_i \end{aligned} \tag{1}$$

where α is the learning rate, $F(\cdot)$ is the objective function, and ΔW_i is the Gaussian noise generated at worker i.

2.2 Reparameterization gradient

As mentioned in Section 2.1, the reparameterization trick allows the gradients to be computed using backprop (Kingma & Welling, 2014). It is important to contrast this with regularizers such as dropout (Srivastava et al., 2014), which perturb *activations* rather than weights. Because activations can be perturbed independently for every training example within a mini-batch, the variance of the stochastic gradients decreases linearly with the size of the mini-batch. Unfortunately, it would be extremely expensive to explicitly sample an independent ΔW for every training example — this would require storing tens or hundreds of copies of the parameters in memory. Therefore, weight perturbation gradients are typically computed using only a single perturbation per mini-batch.

The variance reduction effect of mini-batches is one of the key advantages of activation perturbations over weight perturbations. It is possible to derive efficient independent weight perturbation methods in particular cases; for instance, independent DropConnect masks could be stored since they require only one bit per weight Wan et al. (2013). Kingma et al. (2015) showed that in the case of fully connected networks with no weight sharing, unbiased stochastic gradients could be computed without explicit weight perturbations using the local reparameterization trick (LRT). Specifically, suppose X is the input mini-batch, W is the weight matrix and $B = XW$ is the activation. The LRT samples activation B rather than the weights W. In the case of a Gaussian posterior, the local reparameterization trick is given as:

$$\begin{aligned} q_\theta(W_{i,j}) = \mathcal{N}(\mu_{i,j},\sigma^2_{i,j}) \quad \forall W_{i,j}\in W \implies q_\theta(b_{m,j}|X) = \mathcal{N}(\gamma_{m,j},\delta_{m,j}) \\ \gamma_{m,j} = \sum_{i=1} x_{m,i}\mu_{i,j}, \quad \text{and} \quad \delta_{m,j} = \sum_{i=1} x^2_{m,i}\sigma^2_{i,j} \end{aligned} \tag{2}$$

Notice that the local reparameterization trick gives an unbiased gradient estimate when the activations are independent. However, when weight sharing is involved, such as in convolutional neural networks and recurrent neural networks, the LRT does not give an unbiased gradient estimate.

2.3 Other related work

Control variates are another general class of strategies for variance reduction, both for black-box optimization (Williams, 1992; Ranganath et al., 2014; Mnih & Gregor, 2014) and for gradient-based optimization (Roeder et al., 2016; Miller et al., 2017; Louizos et al., 2017). Control variates are complementary to flipout, so one could potentially combine these techniques to achieve a larger variance reduction. We also note that the fastfood transform (Le et al., 2013) is based on similar mathematical techniques. However, whereas fastfood is used to approximately multiply by a large Gaussian matrix, flipout preserves the random matrix's distribution and instead decorrelates the gradients between different samples.

3 Methods

As described above, weight perturbation algorithms suffer from a high variance of the gradient estimates because all training examples in a mini-batch share the same perturbation. More precisely,

sharing the perturbation induces correlations between the gradients, implying that the variance can't be eliminated by averaging. (This argument is detailed in Section 3.2.) In this section, we propose flipout, an efficient way to perturb the weights quasi-independently within a mini-batch. We show that this almost fully decorrelates the gradients within a mini-batch, allowing us to achieve the ideal variance reduction as if the gradients were independent.

3.1 FLIPOUT

We make two assumptions about the weight distribution $q_{\boldsymbol{\theta}}$: (1) the perturbations of different weights are independent; and (2) the perturbation distribution is symmetric around zero. These are nontrivial constraints, but they encompass important applications: independent Gaussian perturbations (e.g. as used in variational BNNs and evolution strategies) and DropConnect with drop probability 0.5. We observe that, under these assumptions, the perturbation distribution is invariant to elementwise multiplication by a random sign matrix (i.e. matrix whose entries are ± 1). In the following, we denote elementwise multiplication by $\circ$.

Proposition 1. *Let $q_{\boldsymbol{\theta}}$ be a perturbation distribution that satisfies the above assumptions, and let $\widehat{\Delta W} \sim q_{\boldsymbol{\theta}}$. Let E be a random sign matrix that is independent of $\widehat{\Delta W}$. Then $\Delta W = \widehat{\Delta W} \circ E$ is identically distributed to $\widehat{\Delta W}$. Furthermore, the loss derivatives computed using ΔW are identically distributed to those computed using $\widehat{\Delta W}$.*

Flipout exploits this fact by using a base perturbation $\widehat{\Delta W}$ shared by all examples in the mini-batch, and multiplies it by a different rank-one sign matrix for each example:

$$\Delta W^{(i)} = \widehat{\Delta W} \circ e_1^{(i)} e_2^{(i)\top}, \tag{3}$$

where the superscript denotes the index within the mini-batch, and $e_1^{(i)}$ and $e_2^{(i)}$ are random vectors whose entries are sampled uniformly from ± 1. According to Theorem 1, the marginal distribution over gradients computed for individual training examples will be identical to the distributions computed using shared weight perturbations. In particular, flipout yields an unbiased gradient estimator for the original loss function. However, by decorrelating the gradients between different training examples, we can achieve much lower variance updates when averaging over a mini-batch.

Vectorization. The advantage of flipout over explicit perturbations is that computations on a mini-batch can be written in terms of matrix multiplications. This enables efficient implementations on GPUs and modern accelerators such as the Tensor Processing Unit (TPU) (Jouppi et al., 2017). Let x denote the activations in one layer of a neural net. The next layer's activations are given by:

$$\begin{aligned} y^{(i)} &= \phi\left(W^\top x^{(i)}\right) \\ &= \phi\left(\left(\overline{W} + \widehat{\Delta W} \circ e_1^{(i)} e_2^{(i)\top}\right)^\top x^{(i)}\right) \\ &= \phi\left(\overline{W}^\top x^{(i)} + \left(\widehat{\Delta W}^\top (x^{(i)} \circ e_2^{(i)})\right) \circ e_1^{(i)}\right), \end{aligned}$$

where ϕ denotes the activation function. To vectorize these computations, we define matrices E_1 and E_2 whose rows correspond to the random sign vectors $e_1^{(i)}$ and $e_2^{(i)}$ for all examples in the batch. The above equation is vectorized as:

$$Y = \phi\left(X\overline{W} + \left((X \circ E_2)\widehat{\Delta W}\right) \circ E_1\right). \tag{4}$$

This defines the forward pass. Because E_1 and E_2 are sampled independently of $\overline{W}$ and $\widehat{\Delta W}$, we can also backpropagate through Eqn. 4 to obtain derivatives with respect to $\overline{W}$, $\widehat{\Delta W}$, and X.

Computational cost. In general, the most expensive operation in the forward pass is matrix multiplication. Flipout's forward pass requires two matrix multiplications instead of one, and therefore should be roughly twice as expensive as a forward pass with a single shared perturbation when done in sequence.[1] However, note the two matrix multiplications are independent and can be parallelized.

[1] Depending on the efficiency of the underlying libraries, the overhead of sampling E_1 and E_2 may be non-negligible. If this is an issue, these matrices may be reused between all mini-batches. In our experiments, this does not cause any drop in performance.

As for the backward pass, a general rule of thumb for neural nets is that it requires roughly twice as many FLOPs as the forward pass. This suggests that each update using flipout ought to be about twice as expensive as an update with a single shared perturbation (again, if not parallelized); this is consistent with our experience. (Note this is the same overhead for the local reparameterization trick (Kingma et al., 2015).)

Evolution strategies. Evolution Strategies (ES) is a highly parallelizable algorithm. Most implementations run on a multi-core CPU machine. Flipout enables the ES to run more efficiently on a GPU because it allows each worker to evaluate a batch of quasi-independent perturbations rather than only one. For example, in the reinforcement learning task, we need to replicate the starting state M times. It follows that we can evaluate M perturbations with flipout at each worker. Instead of Eqn. 1, the update rule becomes:

$$\overline{W}_{t+1} = \overline{W}_t + \alpha \frac{1}{MN\sigma^2} \sum_{i=1}^{M} \sum_{j=1}^{N} F_{ij} \left\{ \widehat{\Delta W}_j \circ \left(e_{1,j}^{(i)} e_{2,j}^{(i)\,T} \right) \right\} \tag{5}$$

where $e_{1,j}^{(i)}$ and $e_{2,j}^{(i)}$ are the ith random flipout sign vectors sampled at the worker j, and F_{ij} is the reward evaluated with i^{th} perturbation at the worker j. The same flipout algorithm can also be applied to supervised learning tasks. As a result, it can accelerate the training process of the current ES algorithm in terms of the wall-clock time with the same amount of resources.

3.2 Variance analysis

In this section, we analyze the variance of stochastic gradients under the flipout perturbation. We show that flipout is guaranteed to reduce the variance of the gradient estimates compared to the naïve shared perturbation.

Without loss of generality, let $\mathcal{G}(x, \Delta W)$ denote the stochastic gradient under the perturbation ΔW for a single training example x w.r.t. the weight W, i.e. $\mathcal{G}(x, \Delta W) = \nabla_{\overline{W}} \mathcal{L}(y, f(x, \overline{W}, \Delta W))$. We define the mini-batch gradient $\mathcal{G}_{\mathcal{B}}(\Delta W)$ as the averaged gradient under perturbation over a mini-batch $\mathcal{B}$ of size N, $\mathcal{G}_{\mathcal{B}}(\Delta W) = \frac{1}{N} \sum_{x \in \mathcal{B}} \mathcal{G}(x, \Delta W)$, where the subscript is used to denote the random mini-batch sampled from the whole training set.

Assume that the training examples in the mini-batch are sampled independently from the data distribution. Using the Law of Total Variance, we can expand the variance of the mini-batch gradient $\mathrm{Var}(\mathcal{G}_{\mathcal{B}}(\Delta W))$ into the gradient variance between mini-batches (data term) and the estimation variance (estimation term) under perturbation noise:

$$\mathrm{Var}\left(\mathcal{G}_{\mathcal{B}}(\Delta W)\right) = \underbrace{\operatorname*{Var}_{\mathcal{B}} \left(\operatorname*{E}_{\Delta W \mid \mathcal{B}} \left[\mathcal{G}_{\mathcal{B}}(\Delta W) \right] \right)}_{\text{data}} + \underbrace{\operatorname*{E}_{\mathcal{B}} \left[\operatorname*{Var}_{\Delta W \mid \mathcal{B}} \left(\mathcal{G}_{\mathcal{B}}(\Delta W) \right) \right]}_{\text{estimation}}. \tag{6}$$

Notice that the data term in the above equation is not related to the perturbation noise ΔW and decays with the mini-batch size N. The estimation term can be further divided and manipulated so that we can reach the following conclusion.

Theorem 1 (Variance Decomposition Theorem)**.** *Let ΔW be the weight perturbations sampled independently given the base perturbations $\widehat{\Delta W}$ from Eqn. 3. Define α, β and γ to be*

$$\alpha = \operatorname*{E}_{x} \left[\operatorname*{E}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\mathcal{G}(x, \Delta W) \right) \right] \right], \qquad \gamma = \operatorname*{Var}_{x} \left(\operatorname*{E}_{\Delta W \mid x} \left[\mathcal{G}(x, \Delta W) \right] \right),$$

$$\beta = \operatorname*{E}_{x} \left[\operatorname*{Var}_{\widehat{\Delta W} \mid x} \left(\operatorname*{E}_{\Delta W \mid \widehat{\Delta W}, x} \left[\mathcal{G}(x, \Delta W) \right] \right) \right] \tag{7}$$

Under the assumption of Proposition 1, the variance of the gradients under the shared and the flipout perturbation can be written in terms of α, β and γ as following:

$$\textit{Shared perturbation:}\ \mathrm{Var}(\mathcal{G}_{\mathcal{B}}(\Delta W)) = \alpha + \beta + \frac{1}{N}\gamma \tag{8}$$

$$\textit{Flipout:}\ \mathrm{Var}(\mathcal{G}_{\mathcal{B}}(\Delta W)) = \frac{1}{N}\alpha + \beta + \frac{1}{N}\gamma \tag{9}$$

Proof. Details of the proof are provided in Appendix A. □

We can interpret α, β and γ in the above theorem. The γ term is the variance from sampling different mini-batch and perturbation, $\alpha + \beta$ is the expected covariance of the gradients among the training examples in the mini-batch. When a single perturbation is used for whole mini-batch, it introduces additional covariance among the gradients contributing to the total variance of the mini-batch gradient. The quasi-independent perturbation from flipout is guaranteed to reduce the gradient covariance (the α term) linearly in the number of training examples. Therefore, flipout achieves a lower-variance gradient estimate than methods based on a shared perturbation.

We further examine the contribution of the gradient covariance to the variance of the mini-batch gradient. In the shared weight perturbation case, if the mini-batch size N is large enough, Eqn. 9 is dominated by the first two covariance terms, i.e. $\alpha + \beta$, which cannot be reduced by increasing the size of the mini-batch N. This suggest the gradient variance should level off after a certain mini-batch size for the shared perturbation. In contrast, the gradient variance under flipout should be considerably lower given that the α term decays with the mini-batch size. Moreover, we argue that if the β term is insignificant in practice, then the total flipout variance should decrease linearly in terms of N. Empirically, we found that the β term is indeed close to 0 for a various neural network architectures as discussed in Section 4.1. Thus, in practice, flipout obtains the ideal variance reduction in the weight perturbation.

4 EXPERIMENTS

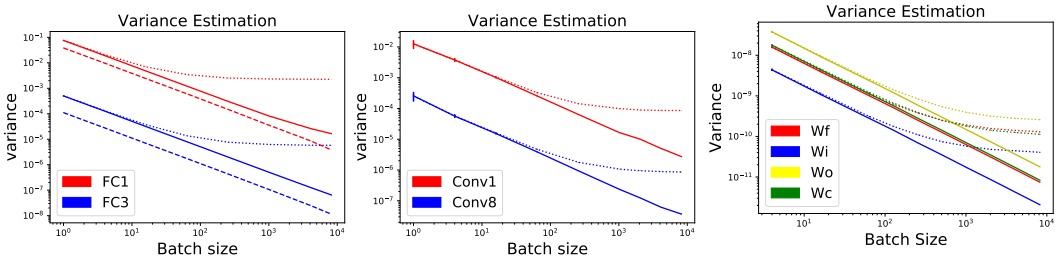
We first empirically verify the variance reduction effect using flipout predicted by Theorem 1 in Section 4.1. We study the variance of the gradients under different perturbations for a wide variety of neural network architectures and batch sizes. In Section 4.2, we show that flipout is effective at regularizing LSTM networks. Furthermore, we demonstrate that the flipout perturbation outperforms the shared perturbations method, which we call "non-flipout", when training with large mini-batches in Section 4.3. Lastly, we present the experimental results of Evolution Strategies with flipout in both supervised learning and reinforcement learning tasks in Section 4.4.

4.1 VARIANCE REDUCTION

Since the main effect of flipout is intended to be variance reduction of the gradients, we first estimated the gradient variances of several architectures with mini-batch sizes ranging from 1 to 8196. We experimented with three perturbation methods: a single shared perturbation per mini-batch, the local reparameterization trick (LRT) of Kingma et al. (2015), and flipout. We computed Monte Carlo estimates of the gradient variance, including both the data and estimation terms in Eqn. 6. (The estimation variance dominated data variance in all of our results except for LRT.) Confidence intervals are based on 50 independent runs of the estimator.

We considered four architectures: a fully connected MNIST classifier, a LeNet-like MNIST conv net (LeCun et al., 1998), a VGG-like CIFAR conv net (Simonyan & Zisserman, 2014), and an LSTM language model trained on Penn Treebank (Marcus et al., 1993). These models are henceforth referred to as **FC**, **Conv1**, **Conv2**, and **LSTM**, respectively. For each of these networks, we froze a partially trained network to use for all variance estimations, and we used multiplicative Gaussian perturbations with $\sigma^2 = 1$. Details of architectures and training methods are given in Appendix B.

Estimation variance plots for all of these models are given in Figure 1. We note that the analysis of Section 3.2 makes strong predictions about the shapes of these curves. In particular, by Theorem 1, the variance curves for flipout and shared each have the form $a + b/N$, where N is the mini-batch size. On a log-log plot, this functional form appears as a linear regime with slope -1, a constant regime, and a smooth phase transition in between. Also, because the distribution of individual gradients is identical with and without flipout, the curves must agree for $N = 1$. Our plots are consistent with both of these predictions. We observe that for shared perturbations, the phase transition consistently occurs for batch sizes somewhere between 100 and 1000. In contrast, flipout gives the ideal linear variance reduction throughout the range of mini-batch sizes we investigated, i.e., its behavior is indistinguishable from fully independent perturbations. In the following experiments, we measure the variance of the stochastic gradients in two settings: (1) **non-flipout**, with per-batch



(a) Fully-connected Net (FC) (b) Convolutional Net (Conv2) (c) LSTM (LSTM)

Figure 1: Empirical variance of gradients with respect to mini-batch size for several architectures. FC1 stands for the first layer in the fully connected network. Similarly, Conv1 stands for the first convolutional layer. **Dotted:** shared perturbations. **Dashed:** LRT (Kingma et al., 2015). **Solid:** flipout. (a) FC on MNIST (b) Deep CNN (vgg-like) on CIFAR-10 (c) LSTM network on Penn Treebank

Gaussian weight perturbations (shared for all examples in a mini-batch), and (2) with **flipout** applied to yield quasi-independent perturbations for each example.

As analyzed by Kingma et al. (2015), the LRT gradients are fully independent within a mini-batch, therefore are guaranteed to achieve the ideal $1/N$ variance reduction. Furthermore, they reduce the variance below that of explicit weight perturbations, so we would expect them to achieve smaller variance than flipout. This is borne out by our plots. However, flipout is applicable to a wider variety of architectures, including convolutional nets and RNNs.

4.2 Regularization for Language Modeling

We evaluate the regularization performance of flipout on both the character-level and word-level language modeling tasks with the Penn Treebank corpus (PTB) (Marcus et al., 1993). We compare flipout to several other methods for regularizing RNNs: naïve dropout (Zaremba et al., 2014), variational dropout (Gal & Ghahramani, 2016b), recurrent dropout (Semeniuta et al., 2016), and zoneout (Krueger et al., 2016). Zaremba et al. (2014) apply dropout only to the feed-forward connections (to the input, output, and connections between layers). Several methods have been proposed to regularize the recurrent connections as well: Semeniuta et al. (2016) apply dropout to the cell update vector, with masks sampled either per step or per sequence; Gal & Ghahramani (2016a) propose to apply dropout to the forward and recurrent connections, with all dropout masks sampled per sequence. Here, we apply flipout to all input-to-hidden and hidden-to-hidden weight matrices, as well as to the hidden-to-output layer, providing feed-forward and recurrent regularization.

Character-Level. For our character-level experiments, we use a single-layer LSTM with 1000 hidden units, sequences of 100 characters in batches of size 32, and train for 55 epochs using Adam with a learning rate of 0.002. Model hyperparameters are given in Appendix D. The results, measured in bits-per-character (BPC) for the validation and test sequences of PTB are shown in Table 1. In the table, non-flipout and flipout are denoted by Mult. Gauss and Mult. Gauss + Flipout, respectively. We see that flipout achieves comparable results with the other methods.

Word-Level. For word-level language-modeling, we follow the experimental setup of Gal & Ghahramani (2016a). We use a 2-layer LSTM with 650 hidden units per layer and a 650-dimensional embedding layer for words. We train on sequences of length 35 in batches of size 20. We use SGD with initial learning rate 20, and train for 40 epochs, decaying the learning rate by a factor of 1.2 each epoch, starting after epoch 6. For the naïve dropout LSTM (Zaremba et al., 2014) and variational LSTM (Gal & Ghahramani, 2016a), we use the hyperparameters described in the original papers. The hyperparameters used for recurrent dropout (Semeniuta et al., 2016) and zoneout (Krueger et al., 2016) are given in Appendix D. Gal's full model (denoted by Gal w/ EmbDrop) uses embedding dropout (setting rows of the embedding matrix to 0). Our flipout model does not use embedding dropout; thus, we also train Gal's model without embedding dropout, where we find optimal input, hidden, and output dropouts of 0.5, 0.3, and 0.5, respectively. The validation and test perplexities of these models are shown in Table 2. We see that flipout outperforms recurrent dropout and zoneout, and achieves competitive performance with naïve dropout and the variational dropout LSTM. In addition, flipout achieves better performance than non-flipout.

	Char-PTB	
Model	**Valid**	**Test**
Unregularized LSTM	1.47	1.43
Semeniuta (2016)	1.35	1.32
Zoneout (2016)	**1.33**	**1.30**
Gal (2016a)	**1.33**	**1.30**
Mult. Gauss ($\sigma = 0.3$)	1.34	**1.30**
Mult. Gauss + Flipout	1.34	**1.30**

Table 1: Bits-per-character (BPC) for the character-level PTB Task.

	Word-PTB	
Model	**Valid**	**Test**
Unregularized LSTM	125.19	123.5
Zaremba (2014)	86.32	82.68
Gal w/ EmbDrop (2016a)	**82.81**	**79.92**
Gal w/o EmbDrop, Opt.	83.83	80.18
Semeniuta (2016)	90.83	86.97
Zoneout (2016)	91.84	88.44
Mult. Gauss ($\sigma = 1$)	89.16	85.40
Mult. Gauss + Flipout	87.31	83.15

Table 2: Perplexity on the PTB word-level validation and test sets.

4.3 LARGE BATCH TRAINING WITH FLIPOUT

Theorem 1 and the Fig.1 suggest that the variance reduction is more effective when the model is trained with a large mini-batch size. In this section, we train a Bayesian neural network with mini-batch size 8192 and show that flipout can speed up the training process in terms of the number of iterations, due to its variance reduction effect. We ran the large mini-batch size training experiment with the Bayes by Backprop algorithm (Blundell et al., 2015), under a 3-layer fully connected network and a LeNet-like convolutional neural network (FC and Conv1 respectively). The purpose of this section is to show the speed up of the training process due to the variance reduction. Thus, we focus on the training loss in this section and present other training details in Appendix E.

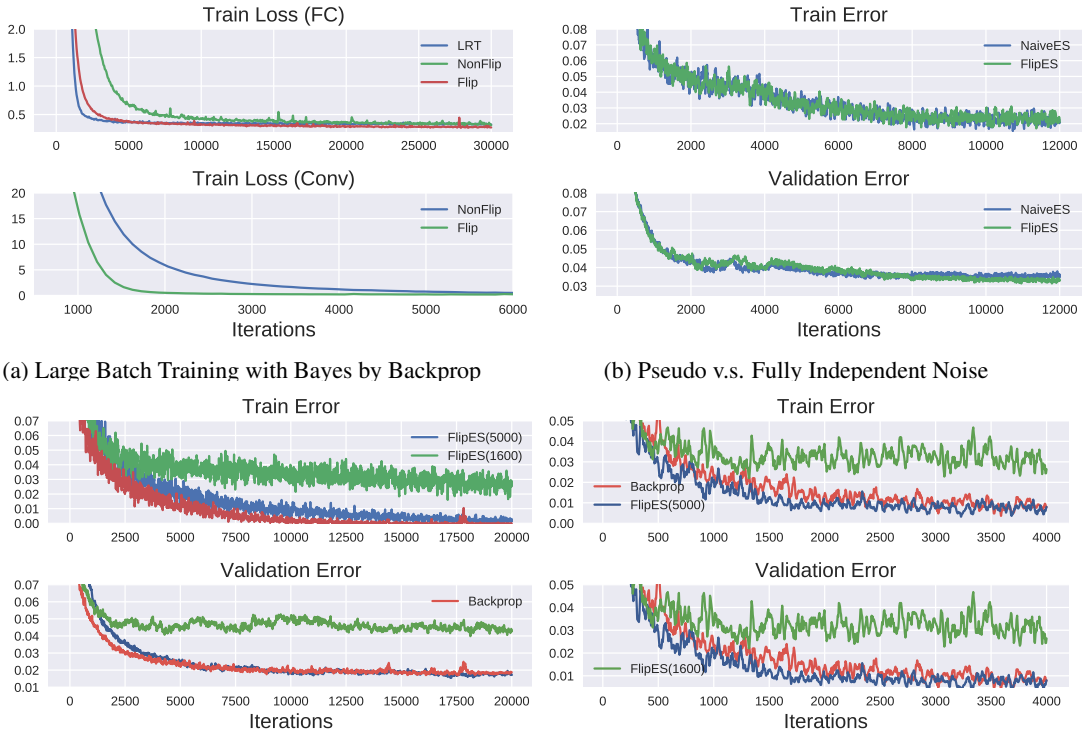
Fig.2a shows the training loss of the large batch experiment. In the fully connected network (FC), we compare the local reparameterization trick, flipout, and non-flipout. Only flipout and non-flipout are compared in the convolutional network (Conv1) experiment since the local reparameterization trick does not provide an unbiased estimator. We found that flipout trained in about 3 times fewer iterations than non-flipout in both fully connected and convolutional networks, while achieving comparable performance with the local reparameterization trick.

In current experiments, flipout is roughly two times slower than non-flipout in FLOPs, which leads to a 1.5 times faster convergence overall. This slowdown is due to an additional matrix multiplication (second half of Eqn. 4). However, this computation is independent from the first half and can be parallelized. In preliminary experiments, we have found this is possible using modern accelerators via the second-generation Tensor Processing Unit (TPU) (Jouppi et al., 2017). A key challenge in utilizing the TPU is to prevent a bottleneck in the input pipeline. In general, large batch sizes, as well as deeper and wider nets, improve utilization where an initial enqueing phase is most expensive. In this regime, weight variance is the biggest concern and flipout provides a promising solution.

4.4 EVOLUTION STRATEGIES

Evolution Strategies (ES) is a highly parallelizable algorithm that typically runs on multiple CPU cores. The limitation that only one perturbation can be evaluated locally restricts its ability to run efficiently on GPU. In Section 3.1, we showed that ES with flipout allows each local worker to evaluate a batch of perturbations, which enables the algorithm to run on a single GPU more efficiently. In this section, we show that flipout accelerates the ES algorithm and narrows the computational gap between ES and back-propagation.

The gradient approximated by Eqn. 1 has high variance. Therefore, a large number of samples are generally needed before applying the update Eqn. 1. We empirically found that $5,000$ samples are needed to achieve a stable performance in the supervised learning tasks. The naïve way to run ES on GPU would be iterating the forward pass $5,000$ times to perform the update. FlipES is designed to reduce the number of iterations, since it can evaluate a batch of perturbations in one single forward pass. Throughout the experiments, we set the flip size to be 40 (i.e. $M = 40$ in Eqn. 5). Naïve ES uses fully independent noise during each forward pass while flipout uses pseudo-independent noise. We compared these two methods with a fully connected network (FC) on the MNIST dataset. We only sampled $1,600$ times per update during training, because the purpose of this experiment is to show no performance loss with pseudo-independent noise, as shown in Fig.2b. Next, we compared FlipES and cpuES (40 cores) in terms of the per update time with respect to the model size. The result (Appendix E.3) shows cpuES has an increasing computational cost as the model size increases, while FlipES scales better because it runs on GPU. Finally, we compare FlipES and the back propagation algorithm in both a fully connected network (FC) and a LeNet-like neural network (Conv1). Fig.2c and Fig.2d show that FlipES achieves a data efficiency comparable with the back propagation algorithm. The naïve ES has a much higher computational cost than



(a) Large Batch Training with Bayes by Backprop

(b) Pseudo v.s. Fully Independent Noise

(c) Bp v.s. FlipES (FC)

(d) Bp v.s. FlipES (Conv1)

Figure 2: Experiment results on large batch training and evolution strategies. Upper left: plot of the training loss over number of iterations with batch size 8192 in the FC network and Conv1 network. Upper right: error rate over iterations; no performance lost to replace independent noise with pseudo-independent one (1,600 samples per update). Lower left: error over iterations in the fully connected network; we compared FlipES (5,000 samples and 1,600 samples per update) with back propagation in terms of the data efficiency. Notice that FlipES is around 60 times more expensive than backprop in each update. Lower right: the same as the lower left plot except the experiment ran on a LeNet-like network.

the back propagation algorithm, due to the large number of forward passes. FlipES can narrow the computational gap between them. Additionally, FlipES is still a highly parallelizable algorithm since there is no dependency in each iteration. Although evolution strategies is more expensive than back propagation, it applies to models which are not entirely differentiable, such as a model with discrete loss (accuracy or BLEU score) or with stochastic units. Moreover, evolution strategies naturally induces the variance of gradient, which makes it easier to perform second-order optimization. We leave these as future work.

5 Conclusions

We have introduced flipout, an efficient method for decorrelating the weight gradients between different examples in a mini-batch. We showed that flipout is guaranteed to reduce the variance compared with shared perturbations. Empirically, we demonstrated significant variance reduction in the large-batch setting for a variety of network architectures, as well as significant speedups in training time. Flipout also makes it practical to apply GPUs to evolution strategies, resulting in substantially increased throughput for a given computational cost. We believe flipout will make weight perturbations practical in the large batch setting favored by modern computational architectures such as Tensor Processing Units (Jouppi et al., 2017).

REFERENCES

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp. 1613–1622, 2015.

Manfred Eigen. Ingo rechenberg evolutionsstrategie optimierung technischer systeme nach prinzipien der biologishen evolution. *mit einem Nachwort von Manfred Eigen, Friedrich Frommann Verlag, Struttgart-Bad Cannstatt*, 45:46–47, 1973.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1019–1027, 2016a.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1019–1027, 2016b.

Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2348–2356, 2011.

Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the 6th Annual Conference on Computational Learning Theory*, pp. 5–13. ACM, 1993.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pp. 448–456, 2015.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. In-datacenter performance analysis of a tensor processing unit. 2017. URL `https://arxiv.org/pdf/1704.04760.pdf`.

Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, 2014.

Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. In *Technical Report*. University of Toronto, 2009.

David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Hugo Larochelle, Aaron C. Courville, and Chris Pal. Zoneout: Regularizing RNNs by randomly preserving hidden activations. *CoRR*, abs/1606.01305, 2016.

Quoc Le, Tamás Sarlós, and Alex Smola. Fastfood-approximating kernel expansions in loglinear time. In *Proceedings of the international conference on machine learning*, volume 85, 2013.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. *arXiv preprint arXiv:1705.08665*, 2017.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

Andrew C Miller, Nicholas J Foti, Alexander D'Amour, and Ryan P Adams. Reducing reparameterization gradient variance. *arXiv preprint arXiv:1705.07880*, 2017.

Andriy Mnih and Karol Gregor. Neural variational inference and learning in belief networks. *arXiv preprint arXiv:1402.0030*, 2014.

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

Rajesh Ranganath, Sean Gerrish, and David Blei. Black box variational inference. In *Artificial Intelligence and Statistics*, pp. 814–822, 2014.

Geoffrey Roeder, Yuhuai Wu, and David Duvenaud. Sticking the landing: A simple, reduced-variance gradient estimator for variational inference. In *Advances in Approximate Bayesian Inference Workshop (NIPS)*, 2016.

Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. In *Proceedings of the 26th International Conference on Computational Linguistics (COLING)*, pp. 1757–1766, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pp. 1058–1066, 2013.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

A PROOF OF THEOREM 1(VARIANCE DECOMPOSITION THEOREM)

In this section, we provide the proof to Theorem 1 (Variance Decomposition Theorem).

Proof. We use the notations in 3.2. Specifically, let $\mathcal{G}(x, \Delta W)$ denote the stochastic gradient under the perturbation ΔW for a single training example x w.r.t. the weight W, i.e. $\mathcal{G}(x, \Delta W) = \nabla_{\overline{W}} \mathcal{L}(y, f(x, \overline{W}, \Delta W))$. We define the mini-batch gradient $\mathcal{G}_{\mathcal{B}}(\Delta W)$ as the averaged gradient under perturbation over a mini-batch $\mathcal{B}$ of size N, $\mathcal{G}_{\mathcal{B}}(\Delta W) = \frac{1}{N} \sum_{x \in \mathcal{B}} \mathcal{G}(x, \Delta W)$, where the subscript is used to denote the random mini-batch sampled from the whole training set. Let x, x' be two arbitrarily chosen training examples from the mini-batch, $\Delta W, \Delta W'$ be the two weight perturbations they received, and $\mathbb{1}_{\text{Flip}}$ be the indicator function for using flipout or not. Following Eqn. 6, the variance can be split into a data term and an estimation term. We simplify the estimation term by replacing the averaged mini-batch gradient with the single training example gradient :

$$\operatorname*{Var}_{\mathcal{B}} \left(\mathop{\mathbb{E}}_{\Delta W \mid \mathcal{B}} \left[\mathcal{G}_{\mathcal{B}}(\Delta W) \right] \right) = \frac{1}{N} \operatorname*{Var}_{x} \left(\mathop{\mathbb{E}}_{\Delta W \mid x} \left[\mathcal{G}_{\mathcal{B}}(\Delta W) \right] \right) \tag{10}$$

$$\begin{aligned}
\mathop{\mathbb{E}}_{\mathcal{B}} \left[\operatorname*{Var}_{\Delta W \mid \mathcal{B}} \left(\mathcal{G}_{\mathcal{B}}(\Delta W) \right) \right] &= \mathop{\mathbb{E}}_{\mathcal{B}} \left[\operatorname*{Var}_{\widehat{\Delta W} \mid \mathcal{B}} \left(\mathop{\mathbb{E}}_{\Delta W \mid \widehat{\Delta W}, \mathcal{B}} \left[\mathcal{G}_{\mathcal{B}}(\Delta W) \right] \right) + \mathop{\mathbb{E}}_{\widehat{\Delta W} \mid \mathcal{B}} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, \mathcal{B}} \left(\mathcal{G}_{\mathcal{B}}(\Delta W) \right) \right] \right] \\
&= \mathop{\mathbb{E}}_{\mathcal{B}} \left[\operatorname*{Var}_{\widehat{\Delta W} \mid \mathcal{B}} \left(\mathop{\mathbb{E}}_{\Delta W \mid \widehat{\Delta W}, \mathcal{B}} \left[\frac{1}{N} \sum_{x \in \mathcal{B}} \mathcal{G}(x, \Delta W) \right] \right) + \mathop{\mathbb{E}}_{\widehat{\Delta W} \mid \mathcal{B}} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, \mathcal{B}} \left(\mathcal{G}_{\mathcal{B}}(\Delta W) \right) \right] \right] \\
&= \mathop{\mathbb{E}}_{x} \left[\operatorname*{Var}_{\widehat{\Delta W} \mid x} \left(\mathop{\mathbb{E}}_{\Delta W \mid \widehat{\Delta W}, x} \left[\mathcal{G}(x, \Delta W) \right] \right) \right] + \frac{1}{N^2} \mathop{\mathbb{E}}_{x} \left[\mathop{\mathbb{E}}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\sum_{x \in \mathcal{B}} \mathcal{G}(x, \Delta W) \right) \right] \right]
\end{aligned} \tag{11}$$

We can further simplify the last term in Eqn. 11 for flipout and shared perturbation cases. In the shared perturbation case, it leads to quadratic N^2 number of the variance terms:

$$\text{Shared:} \quad \frac{1}{N^2} \mathop{\mathbb{E}}_{x} \left[\mathop{\mathbb{E}}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\sum_{x \in \mathcal{B}} \mathcal{G}(x, \Delta W) \right) \right] \right] = \mathop{\mathbb{E}}_{x} \left[\mathop{\mathbb{E}}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\mathcal{G}(x, \Delta W) \right) \right] \right] \tag{12}$$

With flipout, ΔW are conditionally independent given $\widehat{\Delta W}$, which leads to linear number of variance terms.

$$\text{Flipout:} \quad \frac{1}{N^2} \mathop{\mathbb{E}}_{x} \left[\mathop{\mathbb{E}}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\sum_{x \in \mathcal{B}} \mathcal{G}(x, \Delta W) \right) \right] \right] = \frac{1}{N} \mathop{\mathbb{E}}_{x} \left[\mathop{\mathbb{E}}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\mathcal{G}(x, \Delta W) \right) \right] \right] \tag{13}$$

Substituting the above expansions back into Eqn. 6, the total variance of the gradient estimates can be expressed as:

$$\begin{aligned}
\operatorname{Var}\left(\mathcal{G}_{\mathcal{B}}(\Delta W) \right) = & \underbrace{\frac{1}{N} \mathop{\mathbb{E}}_{x} \left[\mathop{\mathbb{E}}_{\widehat{\Delta W} \mid x} \left[\operatorname*{Var}_{\Delta W \mid \widehat{\Delta W}, x} \left(\mathcal{G}(x, \Delta W) \right) \right] \right] + \mathop{\mathbb{E}}_{x} \left[\operatorname*{Var}_{\widehat{\Delta W} \mid x} \left(\mathop{\mathbb{E}}_{\Delta W \mid \widehat{\Delta W}, x} \left[\mathcal{G}(x, \Delta W) \right] \right) \right]}_{\text{estimation}} . \\
& + \underbrace{\frac{1}{N} \operatorname*{Var}_{x} \left(\mathop{\mathbb{E}}_{\Delta W \mid x} \left[\mathcal{G}_{\mathcal{B}}(\Delta W) \right] \right)}_{\text{data}}
\end{aligned} \tag{14}$$

Therefore, the conditionally independence perturbation help reduce the first term in the gradient variance in Eqn. 14.

□

B NETWORK CONFIGURATION IN THE EXPERIMENTS

In this section, we provide the detailed network configurations to reproduce the results in the experiments Section 4.

Table 3: Network Configurations

	Network Type	Data Set
Conv1	(Shallow) Convolutional	MNIST (LeCun et al., 1998)
Conv2	(Deep) Convolutional	CIFAR-10 (Krizhevsky & Hinton, 2009)
FC	Fully Connected	MNIST
LSTM	LSTM Network	Penn Treebank (Marcus et al., 1993)

The (Shallow) convolutional neural networks (Conv1) is a LeNet-like network (LeCun et al., 1998). In specific, the first two layers are convolutional layers with 32 and 64 filters and relu non-linearity. Both filters are of size $[5, 5]$. A 2×2 max pooling layer is followed after each convolutional layer. Dimension reduction only takes place in the pooling layer, which means the stride of the pooling is two and padding is used in the convolutional layers to keep the dimension. Two fully connected layer with 1024 and 10 are attached to produce the classification result.

The (Deep) convolutional neural networks (Conv2) is a VGG16-like network (Simonyan & Zisserman, 2014). We only modified the last fully connected layer in order to run experiments on CIFAR-10 dataset. We didn't use the batch normalization in the variance reduction experiment since it introduces extra stochasticity. Also, multiple GPUs are needed when running the experiment with large mini-batch size (such as 4096 and 8192). In such case, the algorithm naturally induces multiple (depends on the number of GPUs) independent noises due to the distributed computation. In Section 4.1, we fixed the noise among all GPUs to produce the plot. We show in the Appendix E that flipout still results in smaller variance even in the case of distributed computation.

Type FC network is a 3-layer fully connected network. The number of hidden units are 512, 512 and 10 without extra mentioning.

C VARIANCE REDUCTION EXPERIMENT DETAILS

Given a network architecture and a specific algorithm, we compute the empirical stochastic gradient update variance as follows. Starting from a moderately pre-trained model, such as a network with 85% training accuracy on MNIST. Without updating the parameters, we obtain the gradient of each weight in each layer by performing the feed-forward pass with sampling ϵ, E_1 and E_2 and back propagation. The gradient variance of each weight is computed by repeating the above procedure 200 times in the experiments. Denote the variance estimation of layer l, the weight j as $\widetilde{\mathrm{Var}_{lj}}$, the above can be written in the following form,

$$\widetilde{\mathrm{Var}_{lj}} = \frac{1}{200}\sum_{i=1}^{200}(g_{lj}^i - \overline{g_{lj}})^2 \quad \text{where} \quad \overline{g_{lj}} = \frac{1}{200}\sum_{i=1}^{200} g_{lj}^i$$

where g_{lj}^i is the gradient received by the weight j in the layer l. Then we can use the average variance over weights $\frac{1}{|J|}\sum_j \widetilde{\mathrm{Var}_{lj}}$ as the variance estimation of the gradients in the layer l. In order to compute the confidence interval of the gradient variance estimation, we repeat the above procedure 50 times. Suppose we obtain the average variance estimation sequence $\widetilde{V_1}, \ldots, \widetilde{V_{50}}$. Assuming the sequence follows an i.i.d. normal distribution with unknown mean and unknown variance, it follows that $\frac{\overline{V}-\mu}{S/\sqrt{50}} \sim t_{49}$, where $\overline{V}$ is the mean of the sequence and S is the sample standard deviation. From this, we can compute the 90% confidence interval of the mean of the variance estimation and it leads to the error plot in Fig.1.

To measure the variance reduction effect of flipout in LSTMs, we train a flipout LSTM with the two-layer architecture described in Section 4.2, on the word-level Penn Treebank dataset (Marcus et al., 1993). We train on sequences of length 35 for 3 epochs using plain SGD with learning rate 20.

We then measure the variance for each of the hidden-to-hidden weight matrices in the first layer (W_f, W_i, W_c, and W_o). We split large mini-batches (size 128 and higher) into multiple sub-batches of size 64; we sample the noise ϵ once for the set of sub-batches (ϵ is shared among sub-batches), and we sample new E_1 and E_2 matrices for each sub-batch. We see that as the mini-batch size increases, the variance computed with the flipout model continues to decrease, while the non-flipout model converges to a constant variance.

D LSTM EXPERIMENT DETAILS

Long Short-Term Memory networks (LSTMs) are defined by the following equations:

$$\mathbf{i}_t, \mathbf{f}_t, \mathbf{o}_t = \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}) \tag{15}$$

$$\mathbf{g}_t = \tanh(\mathbf{W}_g \mathbf{h}_{t-1} + \mathbf{U}_g \mathbf{x}_t + \mathbf{b}_g) \tag{16}$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t \tag{17}$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t) \tag{18}$$

where $\mathbf{i}_t$, $\mathbf{f}_t$, and $\mathbf{o}_t$ are the input, forget, and output gates, respectively, $\mathbf{g}_t$ is the candidate update gate, and $\circ$ denotes elementwise multiplication. Naïve application of dropout on the hidden state of an LSTM is not effective, because it leads to significant memory loss over long sequences. Several approaches have been proposed to regularize the recurrent connections, based on applying dropout to specific terms in the LSTM equations. Semeniuta et al. (2016) propose to drop the cell update vector, with a dropout mask $\mathbf{d_t}$ sampled either per-step or per-sequence: $\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ (\mathbf{d_t} \circ \mathbf{g}_t)$. Gal & Ghahramani (2016a) apply dropout to the input and hidden state at each time step, $\mathbf{x}_t \circ \mathbf{d_x}$ and $\mathbf{h_{t-1}} \circ \mathbf{d_h}$, with dropout masks $\mathbf{d_x}$ and $\mathbf{d_h}$ sampled once per sequence (and repeated in each time step). Krueger et al. (2016) proposes to *zone out* units rather than dropping them; in the LSTM, the hidden state and cell values are either stochastically updated or maintain their previous value: $\mathbf{c}_t = \mathbf{d_t^c} \circ \mathbf{c_{t-1}} + (1 - \mathbf{d_t^c}) \circ (\mathbf{f_t} \circ \mathbf{c_{t-1}} + \mathbf{i_t} \circ \mathbf{g_t})$ and $\mathbf{h}_t = \mathbf{d_t^h} \circ \mathbf{h_{t-1}} + (1 - \mathbf{d_t^h}) \circ (\mathbf{o_t} \circ \tanh(\mathbf{f_t} \circ \mathbf{c_{t-1}} + \mathbf{i_t} \circ \mathbf{g_t}))$, with zoneout masks $\mathbf{d_t^h}$ and $\mathbf{d_t^c}$ sampled per step.

D.1 HYPERPARAMETER DETAILS

For the word-level models, we use the following hyperparameters. We do not use input or output dropout for any of these models.

- For the full model of (Gal & Ghahramani, 2016a), which includes embedding dropout, we reproduce the parameters given in the paper: we use 0.35 dropout probability on inputs and outputs, 0.2 hidden state dropout, and 0.2 embedding dropout.
- For the variant of (Gal & Ghahramani, 2016a) without embedding dropout, we tune the hyperparameters to achieve the best validation performance, and find that the optimal dropout probabilities for inputs, outputs, and hidden states are 0.5, 0.5, and 0.3.
- For (Semeniuta et al., 2016), we use 0.5 dropout probability on inputs and outputs, and 0.3 dropout probability on cell updates, with per-step mask sampling.
- For (Krueger et al., 2016) we use 0.5 dropout on inputs and outputs, and cell and hidden state zoneout probabilities of 0.25 and 0.025, respectively.
- For the flipout and non-flipout LSTMs, we use Gaussian noise with $\sigma = 1$.

For the character-level models, we use the following hyperparameters:

- For (Semeniuta et al., 2016), we use 0.25 dropout probability on the cell state, and per-step mask sampling
- For Zoneout (Krueger et al., 2016), we use 0.5 and 0.05 for the cell and hidden state zoneout probabilities, respectively.
- For the variaional LSTM (Gal & Ghahramani, 2016a) we use 0.25 recurrent dropout.
- For the flipout and non-flipout LSTMs, we use Gaussian noise with $\sigma = 0.3$.

E SUPPLEMENTARY PLOTS

E.1 VARIANCE REDUCTION

As discussed in Appendix B, training on multiple GPUs naturally induces independent noise for every sub-batch. Fig.3 below shows that flipout still has lower variance in such cases.

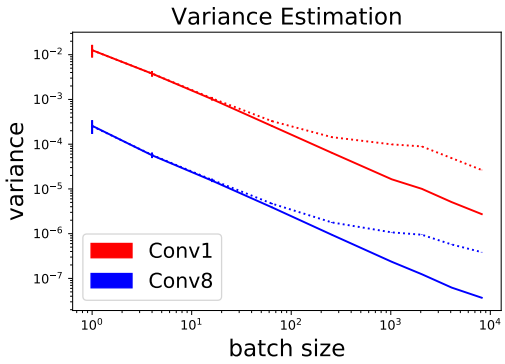


Figure 3: When estimating the variance with mini-batch size 8192, running on 4 GPUs naturally induce 4 different independent noises for 4 sub-batches with size 2048, which also leads to a smaller variance. The same happens to mini-batch size 4096 where 2 GPUs are needed (2 independent noises are generated in this case). The plot shows that flipout still has a lower variance.

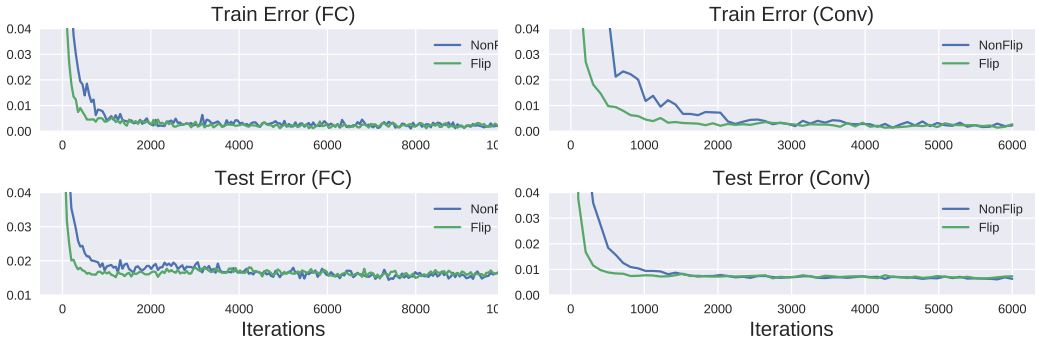


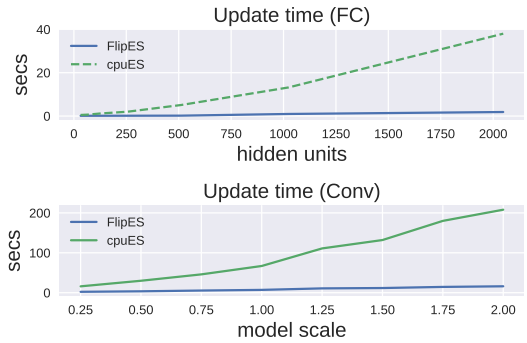
Figure 4: Left: The training and test error plot of large batch (8192) training on a fully connected network (FC) with Bayes by Backprop algorithm. Fig.2a shows flipout has a faster training loss convergence. This plot shows that flipout has the same generalization property as non-flipout (the faster convergence doesn't result in overfitting). Right: The same plot as on the left, except the experiment ran on a LeNet-like network (Conv1).

E.2 LARGE BATCH TRAINING WITH FLIPOUT

Fig.4 is the plot of the training and test accuracy in the large batch training experiments in Section 4.3. The fully connected network and convolutional network we used below have exactly the same architecture as explained in Appendix B. We used an Adam optimizer with a 0.003 learning rate in both networks. We downscaled the KL term by a factor of 10 to achieve a higher accuracy.

E.3 FLIPES V.S. CPUES

Fig.5 shows that cpuES has an increasing computational cost as the model size increases, while FlipES scales better because it runs on GPU.



# Hidden Units	FlipES	cpuES
32	0.12s	0.51s
128	0.13s	1.22s
512	0.18s	5.10s
2048	1.86s	38.0s

# Filters	FlipES	cpuES
0.25	2.3s	16s
0.75	5.48s	46s
1.0	7.12s	67s
1.5	11.77s	132s

Figure 5: Per update time comparison between FlipES and 40-core cpuES (5,000 samples) w.r.t. the model size. In the fully connected network the model is scaled by the number of hidden units, and we scale the number of filters in the convolutional network (1.0 stands for 32 filters in the first convolutional layer and 64 filters for the second one).