Training Glow with Constant Memory Cost

Xuechen Li, Will Grathwohl
University of Toronto, Vector Institute
lxuechen, wgrathwohl@cs.toronto.edu

Abstract

Flow-based models have recently received growing attention in the machine learning community. These models can fit to complex distributions, while also offering exact density computation and efficient sampling. The recently proposed Glow model successfully scaled up this approach to high-dimensional image data. Complementing previous work, we show that a class of flow-based models, known as real non-volume-preserving transformations (real NVP) can be trained efficiently with constant memory cost. We build on ideas from the reversible residual network (RevNet) to compute gradients for affine coupling layers using memory constant with respect to model depth, at the cost of only one extra inverse computation.

1 Introduction

Normalizing flows (NFs) have been developed as a means of modeling complex probability distributions [13, 17, 19, 6, 12, 10, 22, 14]. A related development, RevNets [7] resemble a type of volume-preserving flow known as non-linear independent component estimation (NICE) [5]. RevNets exploit the reversibility of its layers to allow gradients to be computed with memory usage constant with respect to model depth.

In this paper, we give a constant-memory-cost algorithm to compute gradients in real NVPs that use affine coupling layers. Empirically, our gradient computation adds merely a 40% computational overhead in most scenarios. We apply our method to the recently proposed Glow model [12]. On HQ-CelebA images downsampled to 256x256, this enabled us to fit on a *single* GPU a batch size that previously would require 40 GPUs even with the support of gradient checkpointing [20], providing a solution to further scale up these models with fewer computational resources and realizing the claimed memory saving potential [12].

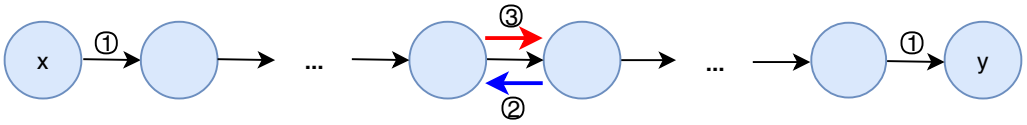

Figure 1: A general way to compute gradients for invertible transformations with constant memory cost (Sec. 2.1) ① first computes output y from input x, and then on the reverse pass ② computes both the inverse and ③ the forward pass a *second* time. Our proposed Algorithm 2 (Sec. 2.2) eliminates the extra forward pass ③.

2 Training Flow-Based Models with Constant Memory Cost

We first review a simple trick for computing gradients of general invertible transformations with constant memory cost in Sec. 2.1. We then describe in Sec. 2.2 our algorithm tailored to affine coupling layers, which has much less computational overhead.

Third workshop on Bayesian Deep Learning (NeurIPS 2018), Montréal, Canada.

2.1 Training Invertible Transformations with Constant Memory Cost

Suppose the transformation we intend to learn can be written as a composition:

$$F(x) = f_n \circ f_{n-1} \circ f_{n-2} \circ \cdots \circ f_1(x),$$

where $f_1, f_2, \ldots, f_n$ are differentiable with parameters $w_1, w_2, \ldots, w_n$, respectively. By the chain rule, the total derivative of $F(x)$ with respect to parameters w_i is:

$$\frac{dF(x)}{dw_i} = \frac{dF(x)}{df_i(x)} \frac{df_i(x)}{dw_i}.$$

In standard backpropagation, also known as reverse-mode automatic differentiation, we must store the intermediate quantities in order to efficiently compute the gradient with respect to each function's parameters. However, when each of the constituent transformations are invertible, we can avoid storing the intermediates, and instead reconstruct them by computing the inverse of each function.

In practice, however, this requires both an extra inverse as well as repeating the forward computation, if reverse-mode automatic differentiation is used naïvely. Specifically, to compute the gradient with respect to the input x given the output y and its gradient dF/dy, we first reconstruct x by inverting y, then compute the forward to reconstruct a new y'. In the end, we call an API provided by most autodiff frameworks to compute dy'/dx and dy'/dw. Algorithm 1 summarizes the idea in pseudocode.

Algorithm 1: Training General Invertible Transformations with Constant Memory
"compute_gradient" is the abstraction of an API supported in most automatic differentiation libraries.

Input: $x, w_1, \ldots, w_n$
Output: $\frac{dF(x)}{dw_1}, \ldots, \frac{dF(x)}{dw_n}$
$x_n = F(x)$
for $i = n$ **to** 1 **do**
  $x_{i-1} = f_i^{-1}(x_i)$
  $y = f_i(x_{i-1})$
  $\frac{dF(x)}{dx_{i-1}} = \text{compute\_gradient}(y, x_{i-1}, \frac{dF(x)}{dx_i})$
  $\frac{dF(x)}{dw_i} = \text{compute\_gradient}(y, w_i, \frac{dF(x)}{dx_i})$
end
return $\frac{dF(x)}{dw_1}, \ldots, \frac{dF(x)}{dw_n}$

2.2 Training Affine Coupling Layers with Constant Memory

Although Algorithm 1 is in principle applicable to learning any type of invertible transformation, the computational overhead of a backward plus extra forward pass may limit its usage in practice. For instance, either the forward or inverse computation for autoregressive flows [17, 13] takes $O(D)$ time, where D is the number of variables. Nevertheless, for real NVP and Glow, the overhead can be reduced by exploiting its architecture as in RevNet. One key ingredient of real NVP and Glow is the affine coupling layer:

$$y_1 = x_1, \quad y_2 = x_2 \circ \exp(\mathcal{F}(x_1)) + \mathcal{G}(x_1). \tag{1}$$

$\mathcal{F}$ and $\mathcal{G}$ can be arbitrary differentiable functions which themselves might not be invertible. The specific structure of this set of forward updates gives us a way to prune the extra forward computation in the method from Sec. 2.1 by reusing $\mathcal{F}(x_1)$ and $\mathcal{G}(x_1)$ computed in the extra backward pass to compute gradients of the loss with respect to parameters and input.

Additionally, the maximum likelihood objective in density estimation contains an extra log-determinant term:

$$\log p(x) = \log p(y) + \log|\det J| = \log p(y) + \sum_i \mathcal{F}(x_1)_i. \tag{2}$$

Here, the term is a sum over $\mathcal{F}(x_1)$'s vector components, whose derivative with respect to itself is a vector of ones. We simply add this to the partial derivate of $\log p(y)$ with respect to $\mathcal{F}(x_1)$ when

relaying the gradients backward. In practice, we compute the log-determinant term as the mean across a batch of inputs, in which case we add to $d \log p(y)/d\mathcal{F}(x_1)$ a vector whose every component is the reciprocal of the batch size.

Algorithm 2 summarizes the overall idea in pseudocode. A TensorFlow implementation of the backward pass can be found in App. 7.1.

Algorithm 2: Training Real NVPs with Constant Memory.
To simplify notation, let $\bar{m} = d \log p(y)/dm$ and $\tilde{m} = d \log p(x)/dm$, where $\log p(x)$ and $\log p(y)$ are in Eq. 2. $\mathcal{W}_\mathcal{F}$ and $\mathcal{W}_\mathcal{G}$ are parameters of $\mathcal{F}$ and $\mathcal{G}$ respectively.

Input: y_1, y_2, $\bar{y_1}$, $\bar{y_2}$
Output: x_1, x_2, $\tilde{x_1}$, $\tilde{x_1}$, $\tilde{\mathcal{W}_\mathcal{F}}$, $\tilde{\mathcal{W}_\mathcal{G}}$
$G = \mathcal{G}(y_1)$
$F = \mathcal{F}(y_1)$
$x_1 = y_1$
$x_2 = (y_2 - G) \circ \exp(-F)$
$\tilde{G} = \bar{y_2}$
$\tilde{F} = (x_2 \circ \exp(F))^\top \bar{y_2} + \mathbb{1}$ (In practice add batch size reciprocal, instead of $\mathbb{1}$)
$\tilde{x_1} = \bar{y_1} + (dG/dx_1)^\top \tilde{G} + (dF/dx_1)^\top \tilde{F}$
$\tilde{x_2} = \bar{y_2}$
$\tilde{\mathcal{W}_\mathcal{G}} = (dG/d\mathcal{W}_\mathcal{G})^\top \tilde{G}$
$\tilde{\mathcal{W}_\mathcal{F}} = (dF/d\mathcal{W}_\mathcal{F})^\top \tilde{F}$
return x_1, x_2, $\bar{x_1}$, $\bar{x_2}$, $\bar{\mathcal{W}_\mathcal{F}}$,$\bar{\mathcal{W}_\mathcal{G}}$

3 Related Work

Reversible Architectures The core architecture of RevNet can be interpreted as composing two sets of NICE additive coupling layers. Computing gradients with only one extra inverse in that setting was also derived [7]. Subsequent work improved on RevNet in terms of theoretical properties and practical performance [11, 2]. More recent work applied the idea to build reversible recurrent nets (revRNN) [15]. Our work complements these works in adapting the idea to non-volume-preserving transformations and may be applied to revRNN to reduce the computational overhead, which according to their experiments is 1-2x, hence a drawback of the approach.

Flow-based Models Flow-based models were first introduced as a method to learn flexible posterior approximations for variational inference [19, 13]. Concurrently, similar models were proposed for density estimation [5, 6, 12, 17]. Notably, flow-based models without function approximators occurred early on in the mathematics literature [21].

The authors of the Glow paper suggest that flow-based density estimators offer great potentials in memory saving at training time [12] without stating the specifics on how this would be realized efficiently. In fact, their released implementation of the model relies on a variant of gradient checkpointing [20, 16, 4, 9] which has similar computational overhead as our method. Yet, the memory usage there is proportional to the square root of model depth, which made their larger models trained on HQ-CelebA downsampled to 256×256 to require 40 GPUs merely to fit mini-batches of size 40.

Recent work has combined Glow with WaveNet [23], producing a simple model for fast and high-quality audio synthesis [18]. The prevalent usage of affine coupling layers suggests that Algorithm 2 may be applied to these models potentially for large-batch training of deeper models.

Orthogonal to designing flow-based models ordered by discrete steps, an alternative line of work focuses on flows that alter variables continuously and compute gradients with respect to variables that define the gradient field of ordinary differential equations using the adjoint method [3, 8].

4 Experiments

4.1 Memory Usage and Computational Overhead

We validate the memory saving and computational overhead of Algorithm 2. On a single Nvidia TITAN Xp, we train a 3 level Glow model [12] varying the number of flows per level on a batch of eight synthetic $256 \times 256 \times 3$ examples. We see that Algorithm 2 has an overhead of <0.4x, whereas the overhead for Algorithm 1 is roughly 1.3x. Profiling details can be found in App. 7.2.

Being able to train flow-based models with constant memory allows us to fit larger mini-batches during training. However, the more direct benefit is that we can fit larger and deeper models with fewer GPUs. For instance, the 6 level, 32 flows/level Glow model that produced state-of-the-art qualitative results on HQ-CelebA downsampled to 256×256 required 40 GPUs with per-PU batch size 1. Using our approach, we are able to fit the same batch on a single Nvidia TITAN Xp, despite being prohibitively slow.

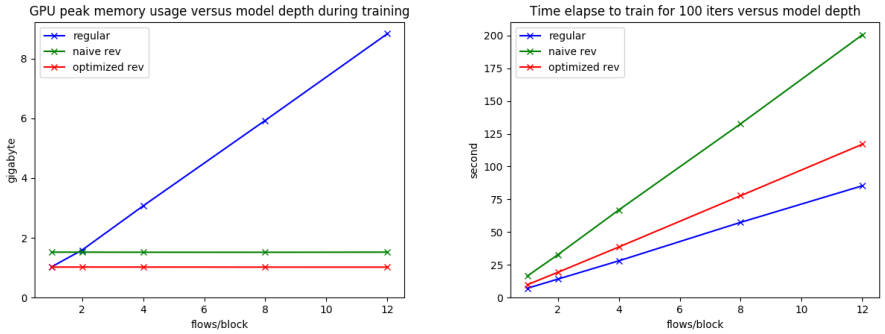

Figure 2: Comparison between usual backpropagation, Algorithm 1 (naive rev), and Algorithm 2 (optimized rev). **Left:** Memory. **Right:** Computational overhead.

4.2 Density Estimation and the Effect of Numerical Loss

As described in [7, 15], we expect gradients computed by Algorithm 2 to be numerically different than that computed by usual reverse-mode automatic differentiation due to limited floating point precision. To verify that performance does not degrade, we train Glow models both with the usual gradients and those computed by Algorithm 2 using single precision floats. From Tab. 1, we do not observe noticeable differences in final log-likelihood results. Training details are in App. 7.3.

	POWER	GAS	HEPMASS	MINIBOONE	BSDS300	MNIST	CIFAR
Usual	-0.22	-8.27	18.13	12.76	-154.74	1.05	3.35
Reversible	-0.23	-8.25	18.03	12.62	-154.83	1.05	3.34

Table 1: Comparison using negative log-likehood on test data between usual backprop (Usual) and Algorithm 2 (Reversible). Numbers are in nats for tabular data and bits/dim for MNIST and CIFAR. The datasets POWER, GAS, HEPMASS, MINIBOONE, and BSDS300 are from [17].

5 Discussions

By extending the gradient computation of RevNet to affine coupling layers, we completed the picture of how one would train real NVPs with constant memory and low overhead. Memory usually becomes the bottleneck with deeper flow-based models trained on higher dimensional data, e.g. the larger Glow model with 6×32 flows trained on HQ-CelebA originally required 40 GPUs' mainly due to the memory cost. We believe our work could benefit future work on further scaling up flow-based models. We also plan to adapt Algorithm 2 to recent works such as the reversible recurrent network [15] to reduce the computational overhead.

6 Acknowledgements

We thank Ricky Tian Qi Chen for helpful discussions. We thank Alexandre Passos, Chaoqi Wang, Shengyang Sun, and David Duvenaud for comments on earlier drafts.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. Reversible architectures for arbitrarily deep residual neural networks. *arXiv preprint arXiv:1709.03698*, 2017.

[3] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[5] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[6] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

[7] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in Neural Information Processing Systems*, pages 2214–2224, 2017.

[8] Will Grathwohl, Ricky TQ Chen, Jesse Betterncourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.

[9] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

[10] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. *arXiv preprint arXiv:1804.00779*, 2018.

[11] Jörn-Henrik Jacobsen, Arnold Smeulders, and Edouard Oyallon. i-revnet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

[12] Diederik P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *arXiv preprint arXiv:1807.03039*, 2018.

[13] Diederik P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in Neural Information Processing Systems*, pages 4743–4751, 2016.

[14] Christos Louizos and Max Welling. Multiplicative normalizing flows for variational bayesian neural networks. *arXiv preprint arXiv:1703.01961*, 2017.

[15] Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger Grosse. Reversible recurrent neural networks. *Advances in Neural Information Processing Systems*, 2018.

[16] James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural networks: Tricks of the trade*, pages 479–535. Springer, 2012.

[17] George Papamakarios, Iain Murray, and Theo Pavlakou. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347, 2017.

[18] Ryan Prenger, Rafael Valle, and Bryan Catanzaro. Waveglow: A flow-based generative network for speech synthesis. *arXiv preprint arXiv:1811.00002*, 2018.

[19] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

[20] Tim Salimans and Yaroslav Bulatov. Gradient checkpointing. *https://github.com/openai/gradient-checkpointing.*, 2018.

[21] Esteban G Tabak, Eric Vanden-Eijnden, et al. Density estimation by dual ascent of the log-likelihood. *Communications in Mathematical Sciences*, 8(1):217–233, 2010.

[22] Rianne van den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. *arXiv preprint arXiv:1803.05649*, 2018.

[23] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *SSW*, page 125, 2016.

7 Appendix

7.1 Sample Implementation in TensorFlow

```
def backward_grads(y1, y2, dy1, dy2):
  x1, dx1 = y1, dy1
  gx1 = g(x1)
  ggrads = tf.gradients(gx1, [x1] + g.variables, grad_ys=dy2)
  dx1 += ggrads[0]
  dg = ggrads[1:]
  fx1 = f(x1)
  expfx1 = tf.exp(fx1)
  x2 = (y2 - gx1) / expfx1
  fgrads = tf.gradients(
      fx1, [x1] + f.variables,
      grad_ys=dy2 * x2 * expfx1 + 1. / float(y1.shape.as_list()[0]))
  dx1 += fgrads[0]
  dx2 = dy2 * expfx1
  df = fgrads[1:]
  dw = df + dg

  with tf.control_dependencies(dw): # Enforce sequential computation
    x1 = tf.identity(x1)

  return x1, x2, dx1, dx2, dw
```

TensorFlow implementation of the gradient computation in Algorithm 2

7.2 Profiling Details

Since all of our models were implemented with TensorFlow [1], memory usage statistics were collected based on the metadata gathered from "session.run". In particular, we also used a custom wrapper [1] that facilitated organizing and interpreting the metadata. Profiling results for the computational overhead was performed with 3 warmup iterations before the 100 iterations of gradient update to reduce the affect of TF's slow initial setup.

7.3 Training Details

Architectural Decisions For experiments on CIFAR in Tab. 1, we follow the exact architecture use in [12]. For experiments on MNIST, we follow the architecture adopted in [8], and use a 2 level

[1] https://github.com/yaroslavvb/memory_util

model with 32 flows per block. For experiments on UCI datasets, we follow the exact architecture in [8], removing the ActNorm layers and replacing invertible convolutions with invertible dense layers. For experiments using invertible convolutions, we follow [12] and initialize the filter of the convolution as an orthogonal matrix. For experiments using invertible dense layers, we follow [8] and initialize the matrix to be the identity. In both cases, we did not explicitly enforce the matrix to be invertible throughout training, following [12, 8]. The matrix inverse and determinant are computed by "tf.matrix_inverse" and "tf.matrix_determinant", respectively.

Training and Testing Hyperparameters For experiments on CIFAR in Tab. 1, we follow the exact set of hyperparameters selected in [12], with the exception that we train on 4 GPUs instead of 8. For experiments on MNSIT and UCI datasets in Tab. 1, we follow the exact set of hyperparameters selected in [8].

Backward pass with constant memory Recall that the Glow model consists of a sequence of flows in a multi-scale architecture, where each flow is composed of an affine coupling layer, an invertible convolution, and an Activation Normalization (ActNorm) layer. The results for "optimized rev" in Fig. 2 used a Glow model, whose invertible convolution and ActNorm layer are implemented according to Algorithm 1, but whose affine coupling layer being implemented according to Algorithm 2. We also implemented a version of invertible convolution and ActNorm layer that avoided the repeated forward computation in the backward pass, but did not observe any significant improvements in speed. For experiments in Tab. 1, we also use the described implementation for constant memory backward pass.