The Journal of Open Source Software

MemCNN: A Python/PyTorch package for creating memory-efficient invertible neural networks

Sil C. van de Leemput[1], **Jonas Teuwen**[1], **Bram van Ginneken**[1], **and Rashindra Manniesing**[1]

1 Radboud University Medical Center, Department of Radiology and Nuclear Medicine, Nijmegen, The Netherlands

DOI: 10.21105/joss.01576

Software

- Review
- Repository
- Archive

Submitted: 10 July 2019
Published: 30 July 2019

Summary

Neural networks are computational models that were originally inspired by biological neural networks like animal brains. These networks are composed of many small computational units called neurons that perform elementary calculations. Instead of explicitly programming the behavior of neural networks, these models can be trained to perform tasks, like classifying images, by presenting them examples. Sufficiently complex neural networks can automatically extract task-relevant characteristics from the presented examples without having prior knowledge about the task domain, which makes them attractive for many complicated real-world applications.

Reversible operations have recently been successfully applied to classification problems to reduce memory requirements during neural network training. This feature is accomplished by removing the need to store the input activation for computing the gradients at the backward pass and instead reconstruct them on demand. However, current approaches rely on custom implementations of backpropagation, which limits applicability and extendibility. We present MemCNN, a novel PyTorch framework that simplifies the application of reversible functions by removing the need for a customized backpropagation. The framework contains a set of practical generalized tools, which can wrap common operations like convolutions and batch normalization and which take care of memory management. We validate the presented framework by reproducing state-of-the-art experiments using MemCNN and by comparing classification accuracy and training time on Cifar-10 and Cifar-100. Our MemCNN implementations achieved similar classification accuracy and faster training times while retaining compatibility with the default backpropagation facilities of PyTorch.

Background

Reversible functions, which allow exact retrieval of its input from its output, can reduce memory overhead when used within the context of training neural networks using backpropagation. That is since only the output requires to be stored, intermediate feature maps can be freed on the forward pass and recomputed from the output on the backward pass when required. Recently, reversible functions have been used with some success to extend the well established residual network (ResNet) for image classification from He, Zhang, Ren, & Sun (2016) to more memory efficient invertible convolutional neural networks (Chang et al., 2017; Gomez, Ren, Urtasun, & Grosse, 2017; Jacobsen, Smeulders, & Oyallon, 2018) showing competitive performance on datasets like Cifar-10, Cifar-100 (Krizhevsky, 2009) and ImageNet (Deng et al., 2009). However, practical applicability and extendibility of reversible functions for the reduction of memory overhead have been limited, since current implementations require customized

backpropagation, which does not work conveniently with modern deep learning frameworks and requires substantial manual design.

The reversible residual network (RevNet) of Gomez et al. (2017) is a variant on ResNet, which hooks into its sequential structure of residual blocks and replaces them with reversible blocks, that creates an explicit inverse for the residual blocks based on the equations from L. Dinh, Krueger, & Bengio (2014) on nonlinear independent components estimation. The reversible block takes arbitrary nonlinear functions $\mathcal{F}$ and $\mathcal{G}$ and renders them invertible. Their experiments show that RevNet scores similar classification performance on Cifar-10, Cifar-100, and ImageNet, with less memory overhead.

Reversible architectures like RevNet have subsequently been studied in the framework of ordinary differential equations (ODE) (Chang et al., 2017). Three reversible neural networks based on Hamiltonian systems are proposed, which are similar to the RevNet, but have a specific choice for the nonlinear functions $\mathcal{F}$ and $\mathcal{G}$ which are shown stable during training within the ODE framework on Cifar-10 and Cifar-100.

The i-RevNet architecture extends the RevNet architecture by also making the downscale operations invertible (Jacobsen et al., 2018), effectively creating a fully invertible architecture up until the last layer, while still showing good classification accuracy compared to ResNet on ImageNet. One particularly interesting finding shows that bottlenecks are not a necessary condition for training neural networks, which shows that the study of invertible networks can lead to a better understanding of neural network training in general.

The different reversible architectures proposed in the literature (Chang et al., 2017; Gomez et al., 2017; Jacobsen et al., 2018) have all been modifications of the ResNet architecture and all have been implemented in TensorFlow (Abadi et al., 2015). However, these implementations rely on custom backpropagation, which limits creating novel invertible networks and application of the concepts beyond the application architecture. Our proposed framework MemCNN overcomes this issue by being compatible with the default backpropagation facilities of PyTorch. Furthermore, PyTorch offers convenient features over other deep learning frameworks like a dynamic computation graph and simple inspection of gradients during backpropagation, which facilitates inspection of invertible operations in neural networks.

Methods

The reversible block

The core operator of MemCNN is the reversible block which is an operator which takes a function f and outputs a function $R : X \to Y$, and an inverse function $R^{-1} : Y \to X$ which resembles an invertible version of f. Here, $x \in X$ and $y \in Y$ can be arbitrary tensors with the same size and number of dimension, i.e.: $\text{shape}(x) = \text{shape}(y)$. Additionally, it must be possible to partition the input $x = (x_1, x_2)$ and output tensors $y = (y_1, y_2)$ in half, where each partition has the same shape, i.e.: $\text{shape}(x_1) = \text{shape}(x_2) = \text{shape}(y_1) = \text{shape}(y_2)$. Formally, the reversible block operation (1), its inverse (2), and its partition constraints (3) provide a sufficiently general framework for implementing reversible operations.

For example, if one wants to create a reversible block performing a convolution followed by a ReLu f, the input $x \in X$ is partitioned in (x_1, x_2) of equal sizes to which this convolution block f is applied twice (say $\mathcal{F}$ and $\mathcal{G}$). The Reversible Block takes these two operators ($\mathcal{F}$ and $\mathcal{G}$) and outputs a "resblock"-like version R of the operator and an explicit inverse R^{-1}. Effectively the learnable function f is replaced by a learnable approximation R with an explicit inverse R^{-1}.

$$R(x) = y \tag{1}$$

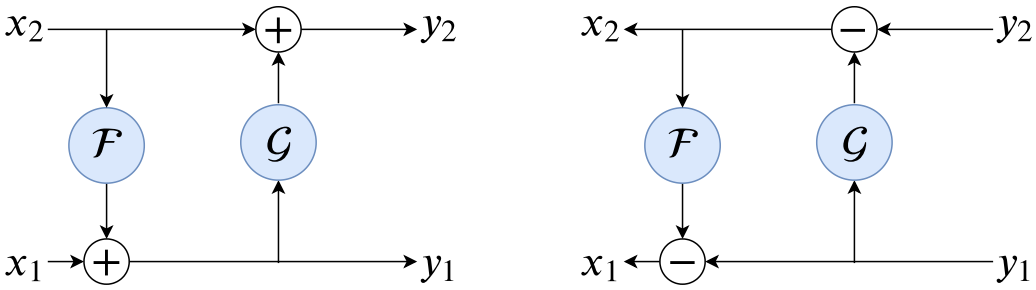

Figure 1: Graphical representation of additive coupling. The left graph shows the forward computations and the right graph shows its inverse. First, input x_1 and $\mathcal{F}(x_2)$ are added to form y_1, next x_2 and $\mathcal{G}(y_1)$ are added to form y_2. Going backwards, first, $\mathcal{G}(y_1)$ is subtracted from y_2 to obtain x_2; subsequently, $\mathcal{F}(x_2)$ is subtracted from y_1 to obtain x_1. Here, $+$ and $-$ stand for respectively element-wise summation and element-wise subtraction.

$$R^{-1}(y) = x \tag{2}$$

with

$$\text{shape}(x_i) = \text{shape}(x_2) = \text{shape}(y_1) = \text{shape}(y_2) \tag{3}$$

Couplings

Using the above definitions we provide two different implementations for the reversible block in MemCNN, which we will call 'couplings'. A coupling provides a reversible mapping from (x_1, x_2) to (y_1, y_2). MemCNN supports two couplings: the additive coupling and the affine coupling.

Additive coupling

Equation 4 represents the additive coupling, which follows the equations of L. Dinh et al. (2014) and Gomez et al. (2017). These support a reversible implementation through arbitrary (nonlinear) functions $\mathcal{F}$ and $\mathcal{G}$. These functions can be convolutions, ReLus, etc., as long as they have matching input and output shapes. The additive coupling is obtained by first computing y_1 from input partitions x_1, x_2 and function $\mathcal{F}$ and subsequently y_2 is computed from partitions y_1, x_2 and function $\mathcal{G}$. Next, (4) can be rewritten to obtain an exact inverse function as shown in (5). Figure 1 shows a graphical representation of the additive coupling and its inverse.

$$\begin{aligned} y_1 &= x_1 + \mathcal{F}(x_2), \\ y_2 &= x_2 + \mathcal{G}(y_1) \end{aligned} \tag{4}$$

$$\begin{aligned} x_2 &= y_2 - \mathcal{G}(y_1), \\ x_1 &= y_1 - \mathcal{F}(x_2) \end{aligned} \tag{5}$$

Affine coupling

Equation (6) gives the affine coupling, introduced by Laurent Dinh, Sohl-Dickstein, & Bengio (2016) and later used by Kingma & Dhariwal (2018), which is more expressive than the

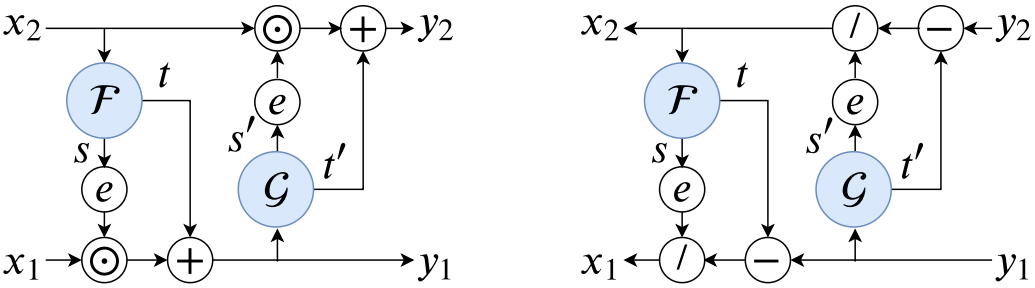

Figure 2: Graphical representation of the affine coupling. The left graph shows the forward computations and the right graph shows its inverse. Here, $\odot, /, +, -$, and e stand for element-wise multiplication, element-wise division, element-wise addition, element-wise subtraction, and element-wise exponentiation with base e respectively. First, s, t are computed for $\mathcal{F}(x_2)$, next input x_1 is element-wise multiplied with e^s and added to t to form y_1, subsequently s', t' are computed for $\mathcal{G}(y_1)$ and then x_2 is element-wise multiplied with $e^{s'}$ and added to t' to form y_2.

additive coupling. The affine coupling, similar to the additive coupling, supports a reversible implementations through arbitrary (nonlinear) functions $\mathcal{F}$ and $\mathcal{G}$. It also first computes y_1 from input partitions x_1, x_2 and function $\mathcal{F}$ and subsequently it computes y_2 from partitions y_1, x_2 and function $\mathcal{G}$. The difference with the additive coupling is that now the functions $\mathcal{F} = (s, t)$ and $\mathcal{G} = (s', t')$ each produce two equally sized partitions for scaling and translation, so $\text{shape}(x_1) = \text{shape}(s) = \text{shape}(t) = \text{shape}(s') = \text{shape}(t')$ holds. These components are then used to compute the output using element-wise product ($\odot$) and element-wise exponentiation with base e and element-wise addition ($+$). Equation (6) can be rewritten to obtain an exact inverse function as shown in (7), which uses element-wise division ($/$) and element-wise subtraction ($-$). Figure 2 shows a graphical representation of the affine coupling and its inverse.

$$\begin{aligned} y_1 &= x_1 \odot e^s + t \quad \text{with} \quad \mathcal{F}(x_2) = (s, t) \\ y_2 &= x_2 \odot e^{s'} + t' \quad \text{with} \quad \mathcal{G}(y_1) = (s', t') \end{aligned} \tag{6}$$

$$\begin{aligned} x_2 &= (y_2 - t')/e^{s'} \quad \text{with} \quad \mathcal{G}(y_1) = (s', t') \\ x_1 &= (y_1 - t)/e^{s} \quad \text{with} \quad \mathcal{F}(x_2) = (s, t) \end{aligned} \tag{7}$$

Implementation details

The reversible block has been implemented as a `torch.nn.Module` which wraps other PyTorch modules of arbitrary complexity for coupling functions $\mathcal{F}$ and $\mathcal{G}$. Each memory saving coupling is implemented using at least one `torch.autograd.Function`, which provides a custom forward and backward pass that works with the automatic differentiation system of PyTorch. Memory savings are implemented at the level of the reversible block and are achieved by setting the size of the underlying tensor storage to zero for inputs on the forward pass and restoring the storage size to the original size on the backward pass once it is required for computing gradients.

Building larger networks

The reversible block R can be chained by subsequent reversible blocks, e.g.: $R_3 \circ R_2 \circ R_1$ for reversible blocks R_1, R_2, R_3, which creates a fully reversible chain of operations (see Figure 3). Additionally, reversible blocks can be mixed with regular functions f, e.g. $f \circ R$ or $R \circ f$ for

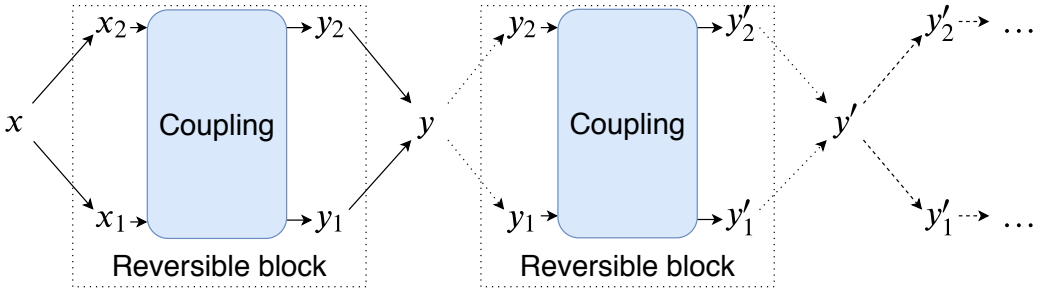

Figure 3: Graphical representation of chaining multiple reversible block layers.

reversible block R and regular function f. Note that mixing regular functions with reversible blocks often breaks the invertibility of reversible chains.

Memory savings

Table 1: Comparison of memory and computational complexity for training a residual network (ResNet) between various memory saving techniques (extended table from Gomez et al. (2017)). L depicts the number of residual layers in the ResNet.

Technique	Authors	Memory Complexity	Computational Complexity
Naive		$O(L)$	$O(L)$
Checkpointing	Martens et al. (2012)	$O(\sqrt{L})$	$O(L)$
Recursive	Chen et al. (2016)	$O(\log L)$	$O(L \log L)$
Additive coupling	Gomez et al. (2017)	$O(1)$	$O(L)$
Affine coupling	Dinh et al. (2016)	$O(1)$	$O(L)$

The reversible block model has an advantageous memory footprint when chained in a sequence when training neural networks. After computing each $R(x) = y$ by (1) on the forward pass, input x can be freed from memory and be recomputed on the backward pass, using the inverse function $R^{-1}(y) = x$ from (2). Once the input is restored, the gradients for the weights and the inputs can be recomputed as normal using the PyTorch 'autograd' solver. This effectively yields a memory complexity of $O(1)$ in the number of chained reversible blocks. Table 1 shows a comparison of memory versus computational complexity for different memory saving techniques.

Experiments and results

Table 2a: Accuracy comparison of the PyTorch implementation (MemCNN) versus the Tensorflow implementation from Gomez et al. (2017) on Cifar-10 and Cifar-100 (Krizhevsky, 2009). Accuracies were approximately similar between implementations.

	Cifar-10		Cifar-100	
Model	**Tensorflow**	**PyTorch**	**Tensorflow**	**PyTorch**
ResNet-32	92.74	92.86	69.10	69.81
ResNet-110	93.99	93.55	73.30	72.40
ResNet-164	94.57	94.80	76.79	76.47
RevNet-38	93.14	92.80	71.17	69.90
RevNet-110	94.02	94.10	74.00	73.30
RevNet-164	94.56	94.90	76.39	76.90

Table 2b: Training time (in hours:minutes) comparison of the PyTorch implementation (MemCNN) versus the Tensorflow implementation from Gomez et al. (2017) on Cifar-10 and Cifar-100 (Krizhevsky, 2009). Training times were significantly less for the PyTorch implementation than for the Tensorflow implementation.

	Cifar-10		Cifar-100	
Model	**Tensorflow**	**PyTorch**	**Tensorflow**	**PyTorch**
ResNet-32	2:04	1:51	1:58	1:51
ResNet-110	4:11	2:51	6:44	2:39
ResNet-164	11:05	4:59	10:59	3:45
RevNet-38	2:17	2:09	2:20	2:16
RevNet-110	6:59	3:42	7:03	3:50
RevNet-164	13:09	7:21	13:12	7:17

To validate MemCNN, we reproduced the experiments from Gomez et al. (2017) on Cifar-10 and Cifar-100 (Krizhevsky, 2009) using their Tensorflow (Abadi et al., 2015) implementation on GitHub[1], and made a direct comparison with our PyTorch implementation on accuracy and train time. We have tried to keep all the experimental settings, like data loading, loss function, train procedure, and training parameters, as similar as possible. All experiments were performed on a single NVIDIA GeForce GTX 1080 with 8GB of RAM. The accuracies and training time results are listed in respectively Table 2a and Table 2b. Model performance of our PyTorch implementation obtained similar accuracy to the TensorFlow implementation with less training time on Cifar-10 and Cifar-100. All models and experiments are included in MemCNN and can be rerun for reproducibility.

Table 3 shows memory usage statistics (parameters and activations) during training for all PyTorch models. Here, the ResNet model uses a conventional implementation and the RevNet model uses the reversible blocks from MemCNN. The results show that significant activation memory reduction was obtained using the reversible block implementation (RevNet) when the number of layers of the models increased.

[1] https://github.com/renmengye/revnet-public

Table 3: Model statistics for all PyTorch model implementations on memory usage (parameters and activations) in MB during training and the number of layers and parameters. The ResNet model was implemented using a conventional non-reversible implementation while the RevNet model uses MemCNN with memory saving reversible blocks. To facilitate comparison, each row lists the statistics of one ResNet and one RevNet model which have a comparable number of layers and number of parameters. Significant memory savings for the activations were observed when using reversible operations (RevNet) as the number of layers increased. Model parameter memory usage stayed roughly the same between implementations.

Layers		Parameters		Parameters (MB)		Activations (MB)	
ResNet	RevNet	ResNet	RevNet	ResNet	RevNet	ResNet	RevNet
32	38	466906	573994	1.9	2.3	238.6	85.6
110	110	1730714	1854890	6.8	7.3	810.7	85.7
164	164	1704154	1983786	6.8	7.9	2452.8	432.7

Works using MemCNN

MemCNN has recently been used to create reversible GANs for memory-efficient image-to-image translation by T. F. van der Ouderaa & Worrall (2019). Image-to-image translation considers the problem of mapping both $X \rightarrow Y$ and $Y \rightarrow X$ given two image domains X and Y using either paired or unpaired examples. In this work, the CycleGAN (Zhu, Park, Isola, & Efros, 2017) model has been enlarged and extended with an invertible core using the reversible block, which they call RevGAN. Since the invertible core is weight tied, training the model for the mapping $X \rightarrow Y$ automatically trains the model for mapping $Y \rightarrow X$. They show similar or increased performance of RevGAN with respect to similar non-invertible models like the CycleGAN with less memory overhead during training. The RevGAN model has also been applied to chest CT images (T. F. van der Ouderaa, Worrall, & Ginneken, 2019).

Conclusion

We have presented MemCNN, a novel PyTorch framework, for creating and applying reversible operations for neural networks. It shows similar accuracy on Cifar-10 and Cifar-100 datasets with the current state-of-the-art method for reversible operations in Tensorflow and provides overall faster training times. The main features of the framework are smooth integration of reversible functions with other non-reversible functions by removing the need for a custom backpropagation and simple wrapping of arbitrary complex non-invertible nonlinear functions. The presented framework is intended to facilitate the study and application of invertible functions in the context of neural networks.

Acknowledgements

This work was supported by research grants from the Netherlands Organization for Scientific Research (NWO), the Netherlands and Canon Medical Systems Corporation, Japan.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Retrieved from http://tensorflow.org/

Chang, B., Meng, L., Haber, E., Ruthotto, L., Begert, D., & Holtham, E. (2017). Reversible architectures for arbitrarily deep residual neural networks. arXiv:1709.03698 [cs.CV]. Retrieved from https://arxiv.org/abs/1709.03698

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. IEEE. doi:10.1109/cvprw.2009.5206848

Dinh, L., Krueger, D., & Bengio, Y. (2014). NICE: non-linear independent components estimation. arXiv:1410.8516 [cs.LG]. Retrieved from https://arxiv.org/abs/1410.8516

Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2016). Density estimation using Real NVP. arXiv:1605.08803 [cs.LG]. Retrieved from https://arxiv.org/abs/1605.08803

Gomez, A. N., Ren, M., Urtasun, R., & Grosse, R. B. (2017). The reversible residual network: Backpropagation without storing activations. arXiv:1707.04585 [cs.CV]. Retrieved from https://arxiv.org/abs/1707.04585

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. doi:10.1109/cvpr.2016.90

Jacobsen, J.-H., Smeulders, A., & Oyallon, E. (2018). i-RevNet: Deep invertible networks. In *ICLR*. arXiv:1802.07088 [cs.LG]. Retrieved from https://arxiv.org/abs/1802.07088

Kingma, D. P., & Dhariwal, P. (2018). Glow: Generative flow with invertible 1x1 convolutions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems 31* (pp. 10215–10224). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/8224-glow-generative-flow-with-invertible-1x1-convolutions.pdf

Krizhevsky, A. (2009, April). *Learning multiple layers of features from tiny images* (Master's thesis). University of Toronto, Toronto, Ontario, Canada.

Ouderaa, T. F. van der, & Worrall, D. E. (2019). Reversible GANs for memory-efficient image-to-image translation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. arXiv:1902.02729 [cs.CV]. Retrieved from https://arxiv.org/abs/1902.02729

Ouderaa, T. F. van der, Worrall, D. E., & Ginneken, B. van. (2019). Chest CT super-resolution and domain-adaptation using memory-efficient 3D reversible GANs. In *International conference on medical imaging with deep learning*. London, United Kingdom. Retrieved from https://openreview.net/forum?id=SkxueFsiFV

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2223–2232). doi:10.1109/iccv.2017.244