Reversible Nets

Sayantan Saha Lakshmi Priyanka

EECE 7398
Deep Learning Embedded Systems
Northeastern University

Introduction

- Reversible Networks is a variation of Neural networks where we do not need to store the activations of the different layers during forward propagation but can recalculate it during the backward propagation as long as the layers are homemorphic.
- Make our deep learning model more memory efficient

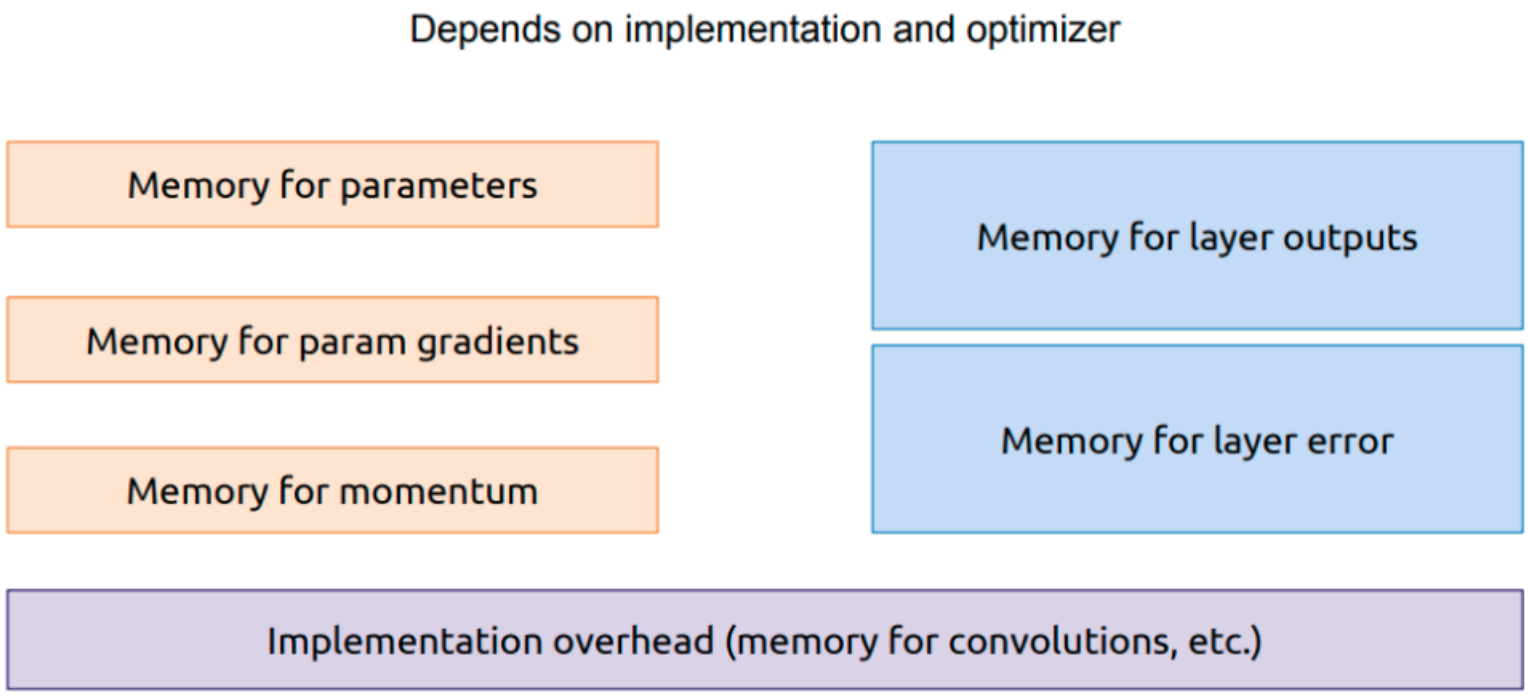
Motivation

- Memory[1] happens to be one of the biggest challenges in deep learning as modern networks get deeper and wider.
- The main Objective of our project was to explore a methodology using which memory overheads during training for a deep neural network can be reduced.
- In our project we look into the software implementation to reduce overheads.
- Our project explores the aspect of reversibility to reduce activation storage memory .

[1] K. Siu, D. M. Stuart, M. Mahmoud and A. Moshovos, "Memory Requirements for Convolutional Neural Network Hardware Accelerators," 2018 IEEE International Symposium on Workload Characterization (IISWC), Raleigh, NC, 2018, pp. 111-121.

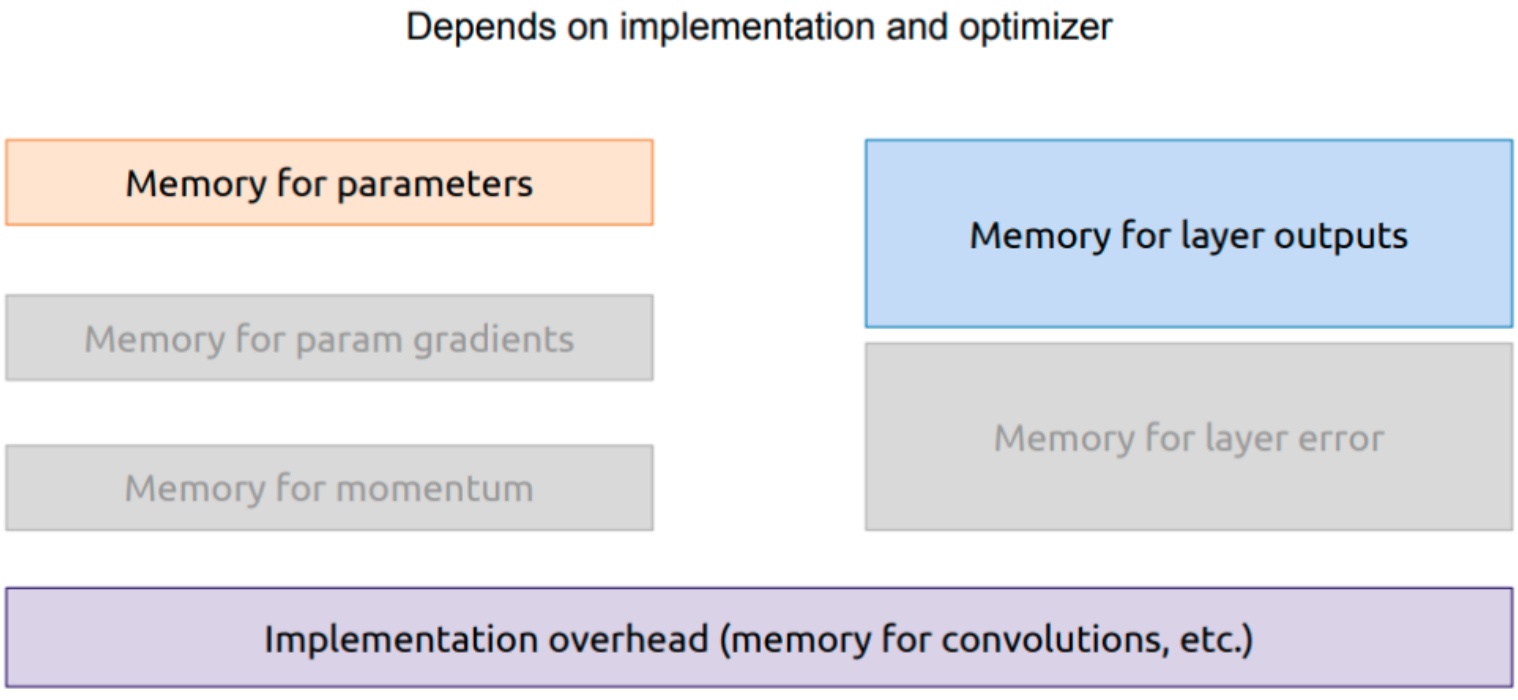
Motivation

Figure: Memory requirements during Training

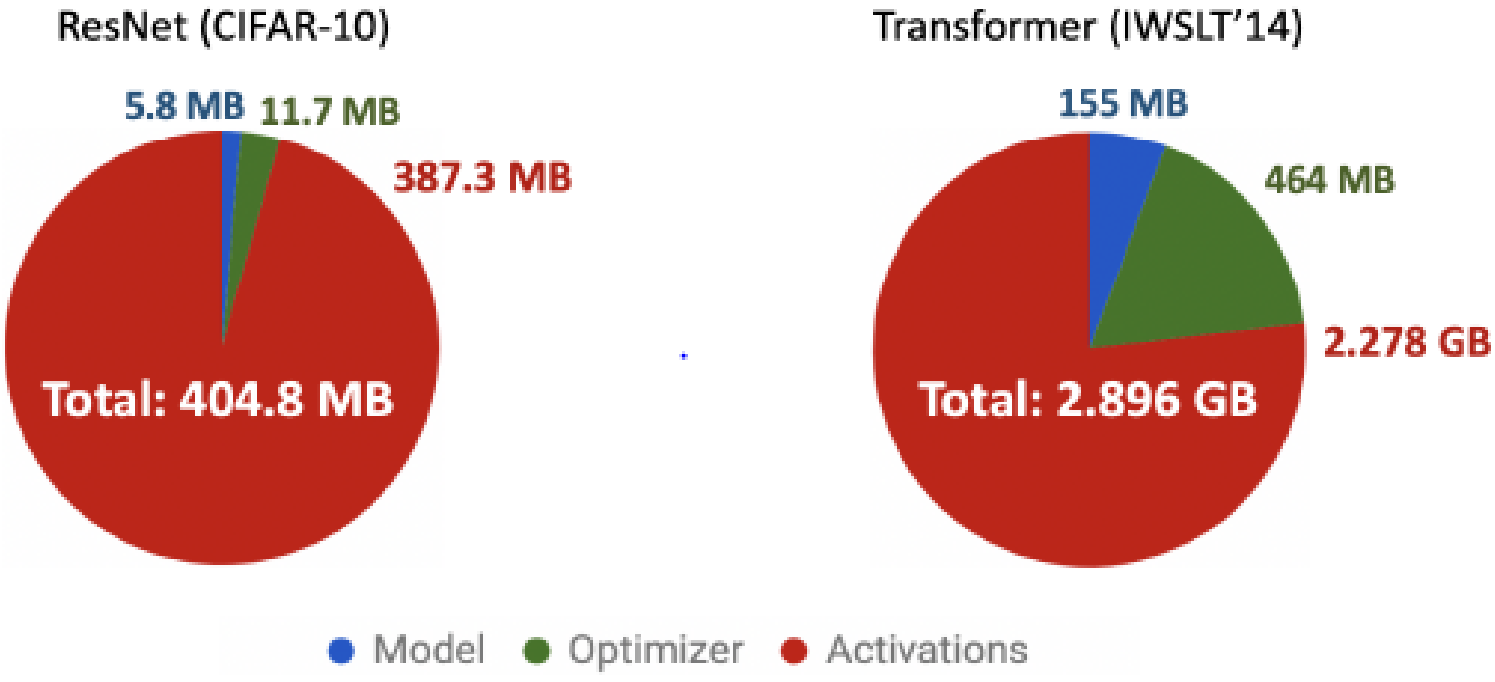


Motivation

Figure: Memory requirements during Testing



Why activation's are so important?



How do we Make a Network Reversible ?

- The concept of reversibility comes from firstly the layers being homeomorphic.
- Homemorphic refers to a function with the same size in the output and input. e.g. Input size N = output size N.
- The subsequent condition being that the weight matrix of the transformation should be non-singular(None zero determinant).
- There needs to exist a bijective(one to one) relation between x and y where y=f(x).

How do we Make a Network Reversible ?

- The concept was derived from NICE[2] (Non Linear Independent Component Estimation) used for generative modelling.

  Consider the transformation $h = f(x)$

  Where $P_X(x)$ = Distribution prior to the transformation.

  and $P_H(f(x))$ = Distribution after the transformation.

  Change of variable: $P_X(x) = P_H(f(x))|det\frac{\partial f(x)}{\partial x}|$

[2] Dinh, Laurent, et al. "NICE: Non-Linear Independent Components Estimation." ArXiv:1410.8516 [Cs], Apr. 2015. arXiv.org, http://arxiv.org/abs/1410.8516.

How do we Make a Network Reversible ?

- For computation sake the function 'f' needs to be easily Invertible and determinant of the Jacobian should be easy to compute.
- Core Idea: Split x into x_1 and x_2

FORWARD:

$y_1 = x_1$

$y_2 = x2 + m(x_1)$

Backward:

$x_1 = y_1$

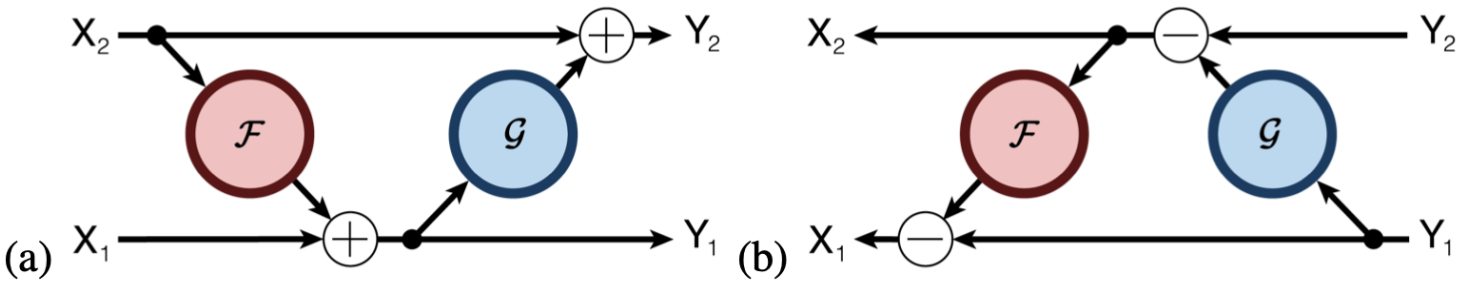
$x_2 = y2 - m(y_1)$

- m refers to the complex bijective transformation which our model will learn.

Methodology

- Units in each layer are partioned into groups denoted by x_1 and x_2

Figure: Forward and Reverse Computations



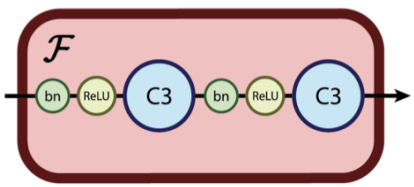


Figure: Resnet block

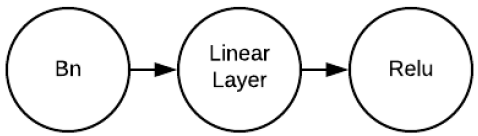


Figure: Linear Block

Methodology

Each reversible block takes inputs (x_1, x_2) and produces outputs (y_1, y_2) according to the following additive coupling rules – inspired by NICE's transformation and F and G are and residual functions F and G analogous to those in standard ResNets.

$$y_1 = x_1 + F(x_2)$$
$$y_2 = x_2 + G(y_1)$$

Each layer's activations can be reconstructed from the next layer's activations as follows:

$$x_2 = y_2 - G(y_1)$$
$$x_1 = y_1 - F(x_2)$$

Note that unlike residual blocks, reversible blocks must have a stride of 1 because otherwise the layer discards information, and therefore cannot be reversible.

Algorithm

Figure: Reversible Residual Block Backprop [3]

Algorithm 1 Reversible Residual Block Backprop

function BLOCKREVERSE($(y_1, y_2), (\overline{y}_1, \overline{y}_2)$)
  $z_1 \leftarrow y_1$
  $x_2 \leftarrow y_2 - \mathcal{G}(z_1)$
  $x_1 \leftarrow z_1 - \mathcal{F}(x_2)$
  $\overline{z}_1 \leftarrow \overline{y}_1 + \left(\frac{\partial \mathcal{G}}{\partial z_1}\right)^\top \overline{y}_2$ ▷ ordinary backprop
  $\overline{x}_2 \leftarrow \overline{y}_2 + \left(\frac{\partial \mathcal{F}}{\partial x_2}\right)^\top \overline{z}_1$ ▷ ordinary backprop
  $\overline{x}_1 \leftarrow \overline{z}_1$
  $\overline{w}_\mathcal{F} \leftarrow \left(\frac{\partial \mathcal{F}}{\partial w_\mathcal{F}}\right)^\top \overline{z}_1$ ▷ ordinary backprop
  $\overline{w}_\mathcal{G} \leftarrow \left(\frac{\partial \mathcal{G}}{\partial w_\mathcal{G}}\right)^\top \overline{y}_2$ ▷ ordinary backprop
  return (x_1, x_2) and $(\overline{x}_1, \overline{x}_2)$ and $(\overline{w}_\mathcal{F}, \overline{w}_\mathcal{G})$
end function

[3] Gomez, Aidan N., et al. "The Reversible Residual Network: Backpropagation Without Storing Activations." ArXiv:1707.04585 [Cs], July 2017. arXiv.org, http://arxiv.org/abs/1707.04585.

Implementation Details

- The earliest conception of the idea was to check if a reversible neural network could be created wrt an architecture that did not have residual structures.
- We have slightly modified the VGG-19[4] structure with an additional convolution layer to make it reversible.
- We use the CIFAR-10 dataset owing to its smaller size to test our concept.

[4] Simonyan, Karen, and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." ArXiv:1409.1556 [Cs], Apr. 2015. arXiv.org, http://arxiv.org/abs/1409.1556.

Implementation Details

- We have used SGD as our optimizer with a momentum value of 0.9.
- Learning rate begins at 0.1 and decreases by $\frac{1}{10}^{th}$ with 50 epochs but could be changed to every 30 or 40 as well.
- We ran the model for 300 epochs.
- Training time was around 4 hours on a Tesla P4.

Implementation Details

- Thankfully we didn't have to implement back-propagation from scratch and could bank on the pytorch.autograd.grad functionality.
- For our modified reversible structure we need to define separate functions for the forward propagation and backward propagation.
- These functions need to be instantiated with " @static methods".
- Owing to this we don't need to separate invoke functions for the forward and backward propagations.
- @static methods will call these new functions along with the usual forward and backward call of the base class that torch.nn.Module.
- Accordingly we only need to make sure that the particular weights, buffers of the appropriate sizes are initialized and stored.

VGG-19 Details

Input Image Size	Layer	Input_channel	Output_channel	Kernal_size	Stride	Output_Image	# parameters	# Activations
(3,32,32)	Conv2d	3	64	3	1	(64,32,32)	1792	65536
(64,32,32)	BatchNorm2d	64	64			(64,32,32)	128	65536
(64,32,32)	Conv2d	64	64	3	1	(64,32,32)	36928	65536
(64,32,32)	BatchNorm2d	64	64			(64,32,32)	128	65536
(64,32,32)	Maxpool2d			2	2	(64,16,16)	0	16384
(64,16,16)	Conv2d	64	128			(128,16,16)	73856	32768
(128,16,16)	BatchNorm2d	128	128			(128,16,16)	256	32768
(128,16,16)	Conv2d	128	128			(128,16,16)	147584	32768
(128,16,16)	BatchNorm2d	128	128			(128,16,16)	256	32768
(128,16,16)	Maxpool2d			2	2	(128,8,8)	0	8192
(128,8,8)	Conv2d	128	256			(256,8,8)	295168	16384
(256,8,8)	BatchNorm2d	256	256			(256,8,8)	512	16384
(256,8,8)	Conv2d	256	256			(256,8,8)	590080	16384
(256,8,8)	BatchNorm2d	256	256			(256,8,8)	512	16384
(256,8,8)	Conv2d	256	256			(256,8,8)	590080	16384
(256,8,8)	BatchNorm2d	256	256			(256,8,8)	512	16384
(256,8,8)	Conv2d	256	256			(256,8,8)	590080	16384
(256,8,8)	BatchNorm2d	256	256			(256,8,8)	512	16384
(256,8,8)	Maxpool2d			2	2	(256,4,4)	0	4096
(512,4,4)	Conv2d	256	512			(512,4,4)	1180160	8192
(512,4,4)	BatchNorm2d	512	512			(512,4,4)	1024	8192
(512,4,4)	Conv2d	512	512			(512,4,4)	2359808	8192
(512,4,4)	BatchNorm2d	512	512			(512,4,4)	1024	8192
(512,4,4)	Conv2d	512	512			(512,4,4)	2359808	8192
(512,4,4)	BatchNorm2d	512	512			(512,4,4)	1024	8192
(512,4,4)	Conv2d	512	512			(512,4,4)	2359808	8192
(512,4,4)	BatchNorm2d	512	512			(512,4,4)	1024	8192
(512,4,4)	Maxpool2d			2	2	(512,2,2)	0	2048
(512,2,2)	Conv2d	512	512			(512,2,2)	2359808	2048
(512,2,2)	BatchNorm2d	512	512			(512,2,2)	1024	2048
(512,2,2)	Conv2d	512	512			(512,2,2)	2359808	2048
(512,2,2)	BatchNorm2d	512	512			(512,2,2)	1024	2048
(512,2,2)	Conv2d	512	512			(512,2,2)	2359808	2048
(512,2,2)	BatchNorm2d	512	512			(512,2,2)	1024	2048
(512,2,2)	Conv2d	512	512			(512,2,2)	2359808	2048
(512,2,2)	BatchNorm2d	512	512			(512,2,2)	1024	2048
(512,2,2)	Maxpool2d			2	2	(512,1,1)	0	512
512	Linear	512	512			512	262656	512
512	Linear	512	512			512	262656	512
512	Linear	512	10			10	5130	10
			Total Parameters				20565834	638474
			Total Memory(Batch_sz=128)				78.45MB	311.75 MB

Rev-VGG Details

Input Image	Layer	Input_cha	Output_channel	Kernal_size	Stride	Output_Image	# parameters	# Activations
(3,32,32)	Conv2d	3	64	3	1	(64,32,32)	1792	65536
(64,32,32)	Maxpool2d			2	2	(64,16,16)	0	16384
(64,16,16)	RevBlock_1	64	128			(128,16,16)	517632	0
(128,16,16)	Maxpool2d			2	2	(128,8,8)	0	16384
(128,8,8)	RevBlock_2	128	256			(256,8,8)	2067456	0
(256,8,8)	Maxpool2d			2	2	(256,4,4)	0	4096
(256,4,4)	RevBlock_3	256	512			(512,4,4)	8263680	0
(512,4,4)	Maxpool2d			2	2	(512,2,2)	0	2048
(512,2,2)	RevBlock_4	512	512			(512,2,2)	9443328	0
(512,2,2)	Maxpool2d			2	2	(512,1,1)	0	512
512	Linear block	512	512			512	525312	512
512	Linear	512	10			10	5130	10
			Total Parameters				20824330	105482
			Total Memory(batch_szie =128)				79.43MB	51.50 MB

Results

- We have an overall accuracy of 92 %.

Figure: Precision, Recall, F1-score

	precision	recall	f1-score	support
0	0.94	0.93	0.94	1000
1	0.97	0.96	0.96	1000
2	0.93	0.88	0.91	1000
3	0.82	0.86	0.84	1000
4	0.92	0.94	0.93	1000
5	0.87	0.87	0.87	1000
6	0.94	0.96	0.95	1000
7	0.97	0.93	0.95	1000
8	0.95	0.97	0.96	1000
9	0.96	0.95	0.95	1000
accuracy			0.92	10000
macro avg	0.93	0.92	0.93	10000
weighted avg	0.93	0.92	0.93	10000

- Precision for top 5 classes = 95.8 %.

Results

- VGG-19 on CIFAR 10 has an accuracy between 92%-93%.
- Our take on the reversible VGG-19 structure has provided us with comparable results of 92% accuracy.
- With a little more fine tuning we can squeeze another 1% accuracy.

Results

- Our Objective here was different and it was to have a successful model with reduced activation memory.
- VGG 19 Activation Memory = 311.75 Mb
- VGG 19 Reversible = 51.50 Mb
- we have successfully reduced memory overhead by 83.4%
- For the sake of simplicity this project was implemented on a smaller dataset and a smaller model to check on the conception of the idea.
- This result will be even more pronounced when model sizes and data size increases.

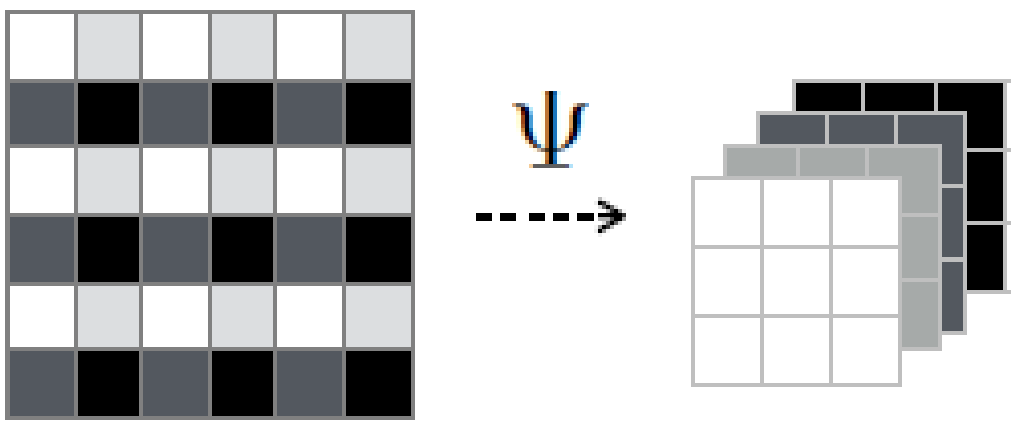
Improvements compared to previous Work

- Major improvements was in terms of overhead memory which was reduced by 83.4%
- Proof of concept in terms of reversibility not having to much in theory with a residual structure is also a novelty.

Future Work

- A layer such as max-pooling /average-pooling need to be replaced with a layer that does reduce the image size but not lose information to make them reversible.
- Eg:- Dilated convolution.Reduce image size by increase the channel depth (As implemented in i-Revnet[5]).

Figure: Dilated Convolution



[5] Jacobsen, Jörn-Henrik, et al. "I-RevNet: Deep Invertible Networks." ArXiv:1802.07088 [Cs, Stat], Feb. 2018. arXiv.org, http://arxiv.org/abs/1802.07088.

Contribution of each Team Member

- Reading of papers was divided equally
- Coding was divided between linear layers and reversible layers
- Ppt : each person included slides that covered topics studied by them.