# The Reversible Residual Network

Aidan Gomez, Mengye Ren, Raquel Urtasun and Roger Grosse

Computer Science
UNIVERSITY OF TORONTO

UBER

# What are Deep Neural Networks (DNNs)?

- A ‘deep’ composition of functions.

$$f_1 \circ f_2 \circ \ldots \circ f_n$$

- Generally, each function is parametrized by some weights

**Single layer of an MLP**

$$f_k(x, W_{f_k}) = \varphi(W_{f_k}^\top x)$$

- These weights can then be augmented to manipulate the computation being performed.

**Convolution**

$$f_k(x, W_{f_k}) = \varphi(x * W_{f_k})$$

# Training DNNs

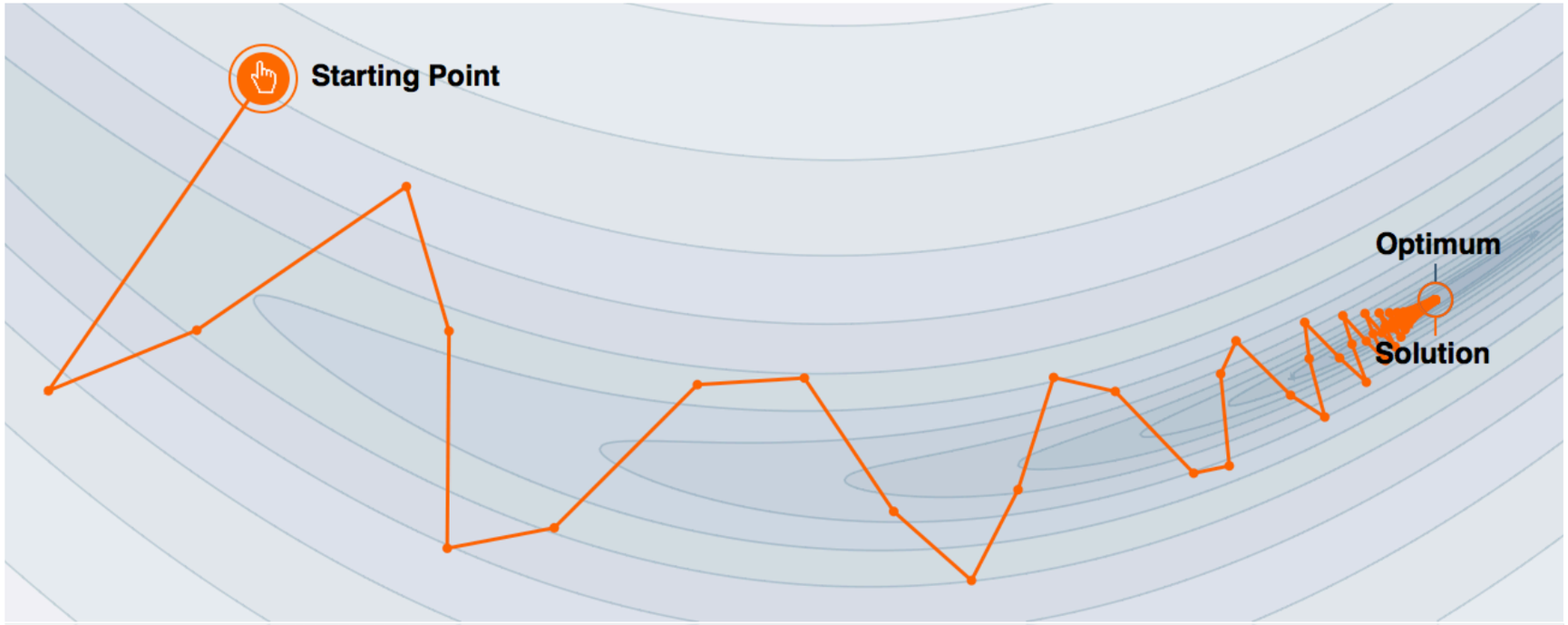



Image: (Goh, 2017)

- Network weights are updated using ‘gradient descent’

$$f_1 \circ f_2 \circ \ldots \circ f_n$$

- Take steps in local direction of steepest descent in the loss

$$L = \text{Loss}(f_1 \circ \cdots \circ f_n, y)$$

# Training DNNs

- Network weights are updated using ‘gradient descent’
- Take steps in local direction of steepest descent in the loss
- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

$$f_1 \circ f_2 \circ \ldots \circ f_n$$

$$L = \text{Loss}(f_1 \circ \cdots \circ f_n, y)$$

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

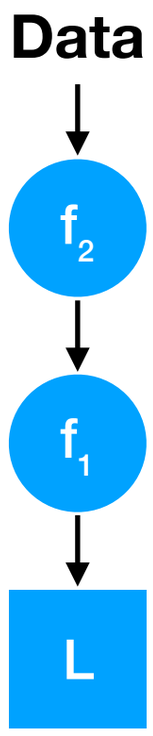
# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.




- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

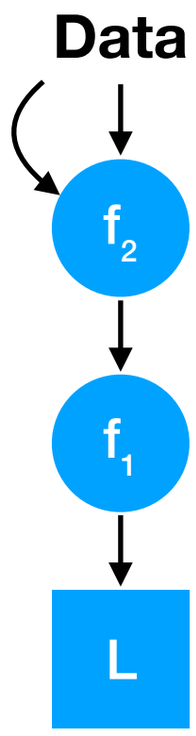
# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.




- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

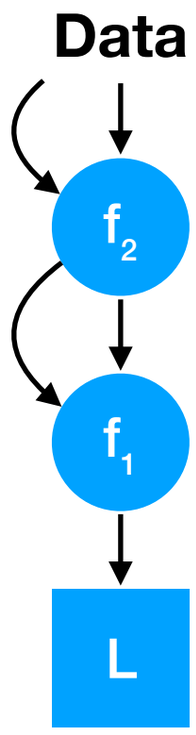
# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.




- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

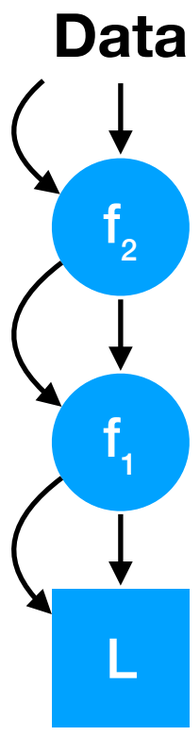
# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.




- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

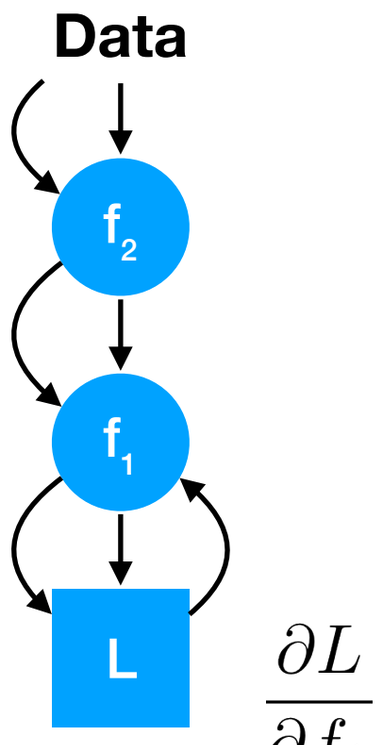
# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.




$$\frac{\partial L}{\partial f_1}$$

- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

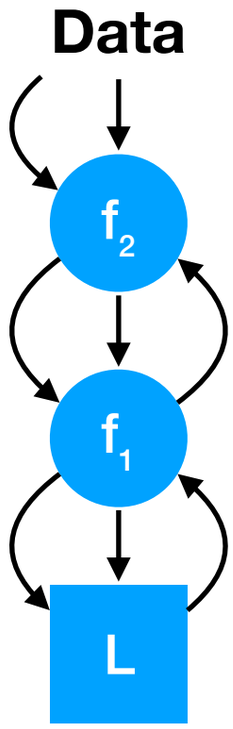
- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.

- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$




$$\frac{\partial L}{\partial f_2} = \frac{\partial L}{\partial f_1} \cdot \frac{\partial f_1}{\partial f_2}$$

$$\frac{\partial L}{\partial f_1}$$

# Training DNNs

- Chain-rule expansion can be recursively computed using algorithm known as backpropagation

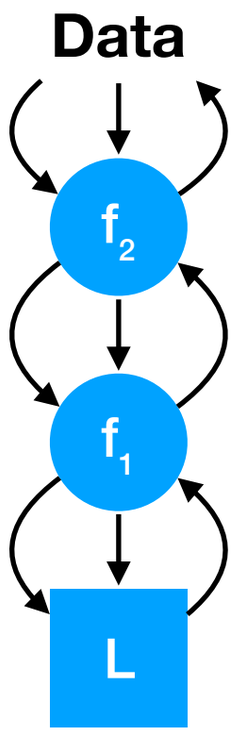
- ‘Backprop’ occurs in two steps:
  - A forward pass that computes the output of each layer.
  - A backward pass the competes the gradients of each layer.

- The partial derivates w.r.t. the weights of each layer rely on the gradients of layer outputs, and the inputs to the layer.

- The consequence is the need for storing all layer activations at each

$$\nabla_{f_k} L = \frac{\partial L}{\partial f_1} \cdots \frac{\partial f_{k-2}}{\partial f_{k-1}} \cdot \frac{\partial f_{k-1}}{\partial f_k}$$

$$\nabla_{f_k} L = \nabla_{f_{k-1}} L \cdot \frac{\partial f_{k-1}}{\partial f_k}$$




$$\frac{\partial L}{\partial \text{Data}} = \frac{\partial L}{\partial f_2} \cdot \frac{\partial f_2}{\partial \text{Data}}$$

$$\frac{\partial L}{\partial f_2} = \frac{\partial L}{\partial f_1} \cdot \frac{\partial f_1}{\partial f_2}$$

$$\frac{\partial L}{\partial f_1}$$

# Training Gets Expensive

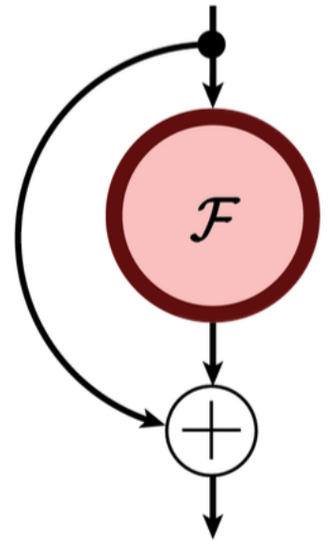
- The observation in deep networks is: `deep = better`
- Modern networks can have hundreds of layers each with large activation maps
- GPUs tend to have 8-16GB in memory and are easily exhausted by high dimensional data.

# Residual Networks (ResNets)

- Class of layer originally developed for vision tasks

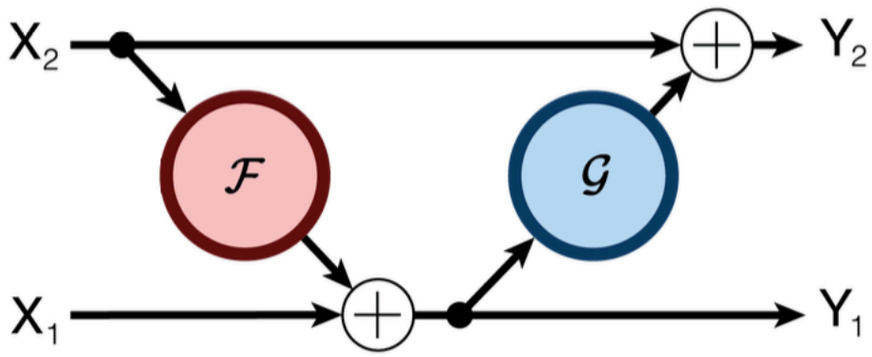
- Learns a perturbation of its input.

- Extremely successful in multiple domains: vision, text and audio.



# Reversible Residual Networks

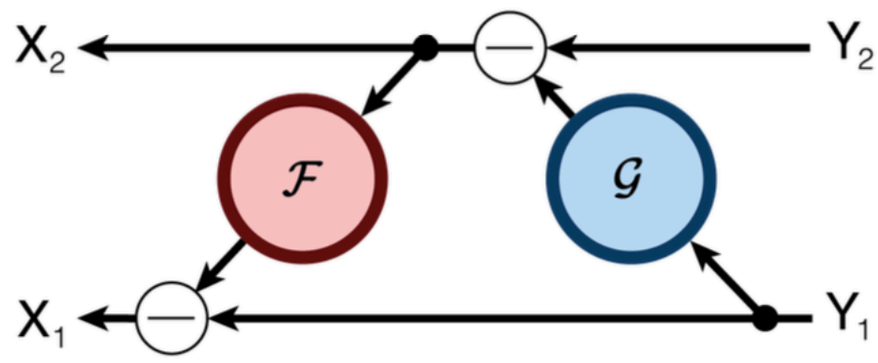
- By partitioning the graph and learning a pair of residuals we can enable exact reconstruction of inputs.




$$y_1 = x_1 + \mathcal{F}(x_2)$$
$$y_2 = x_2 + \mathcal{G}(y_1)$$

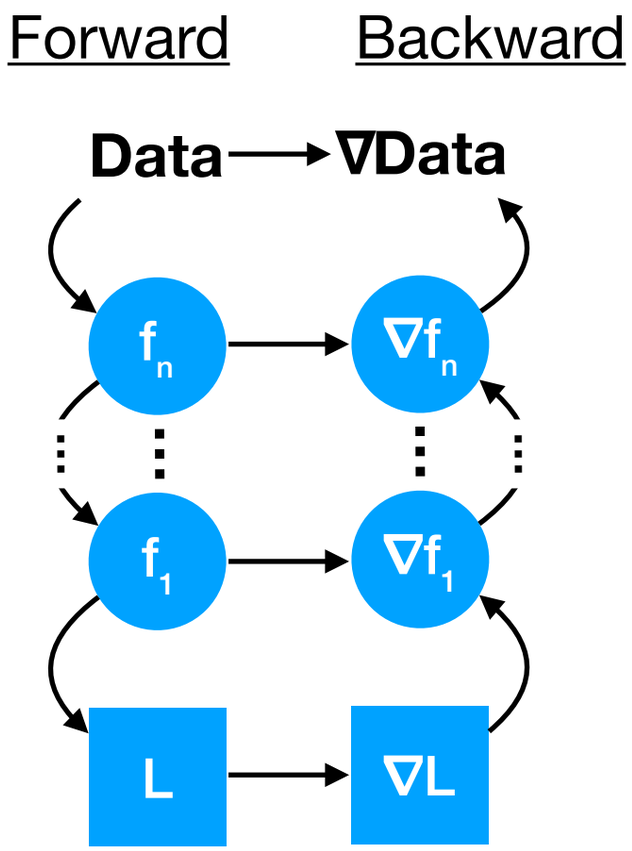
- Given a large stack of these layers, preserving activations is no longer necessary.

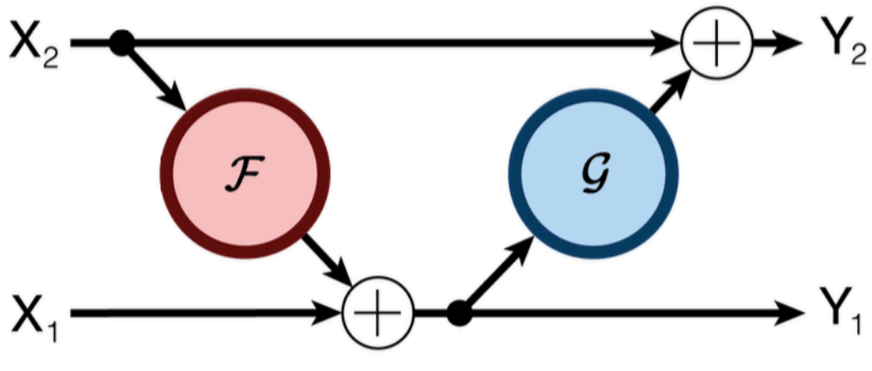
- Simply reconstruct activations from outputs **during** backprop.



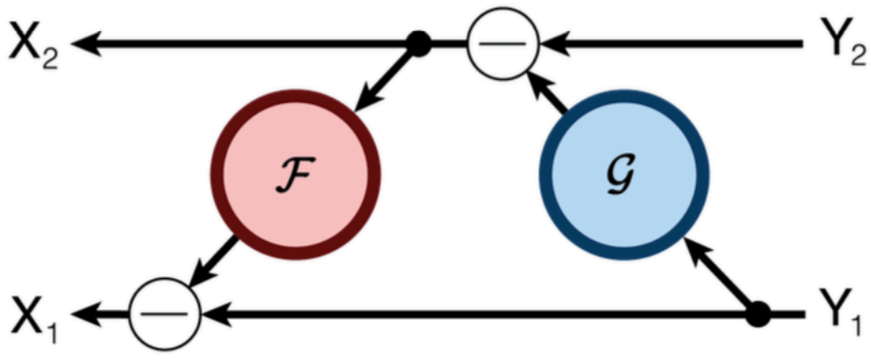

$$x_2 = y_2 - \mathcal{G}(y_1)$$
$$x_1 = y_1 - \mathcal{F}(x_2)$$

# Reversible Residual Networks (RevNet)



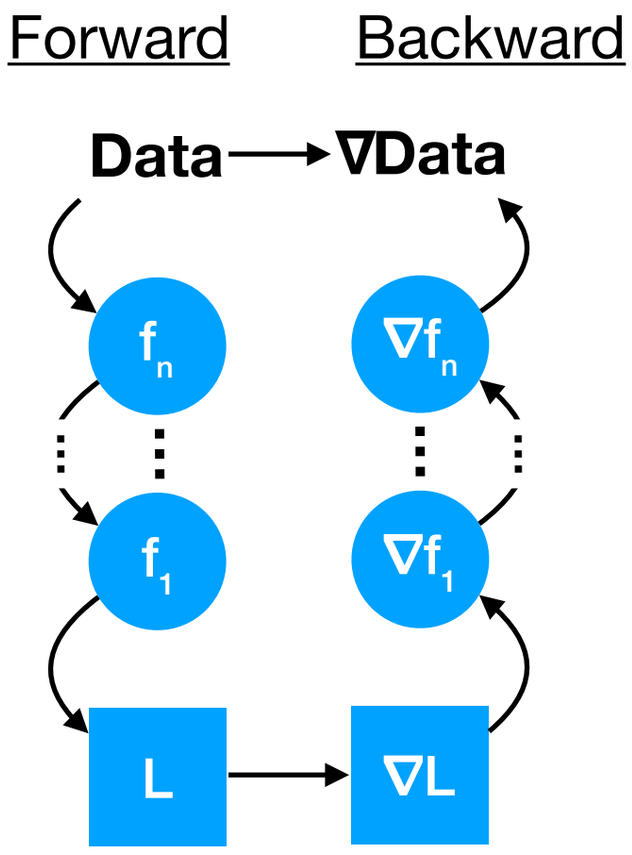



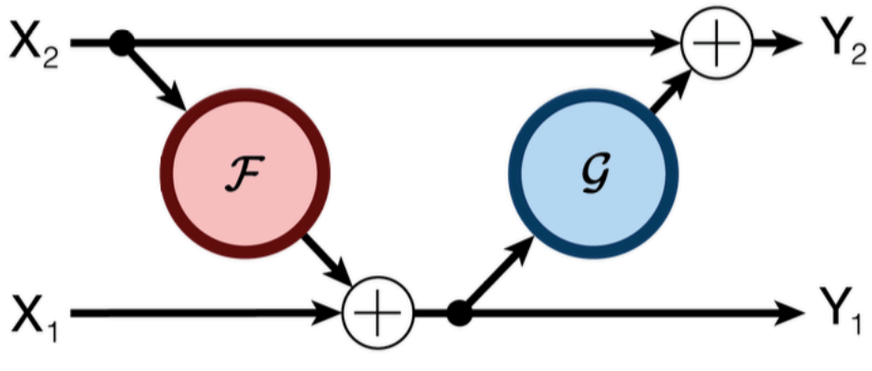

$$y_1 = x_1 + \mathcal{F}(x_2)$$
$$y_2 = x_2 + \mathcal{G}(y_1)$$



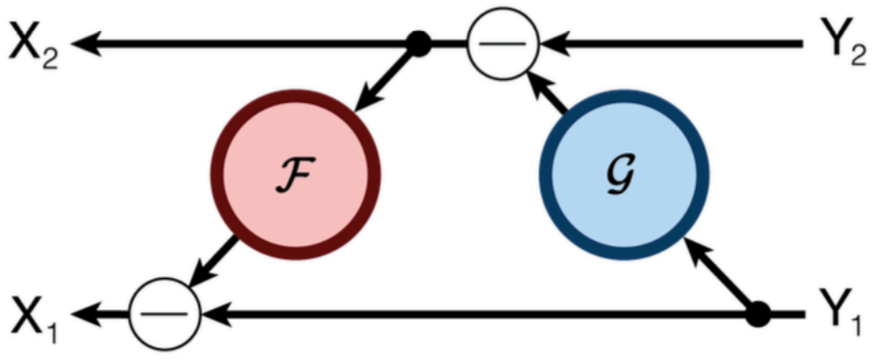

$$x_2 = y_2 - \mathcal{G}(y_1)$$
$$x_1 = y_1 - \mathcal{F}(x_2)$$

# Reversible Residual Networks (RevNet)



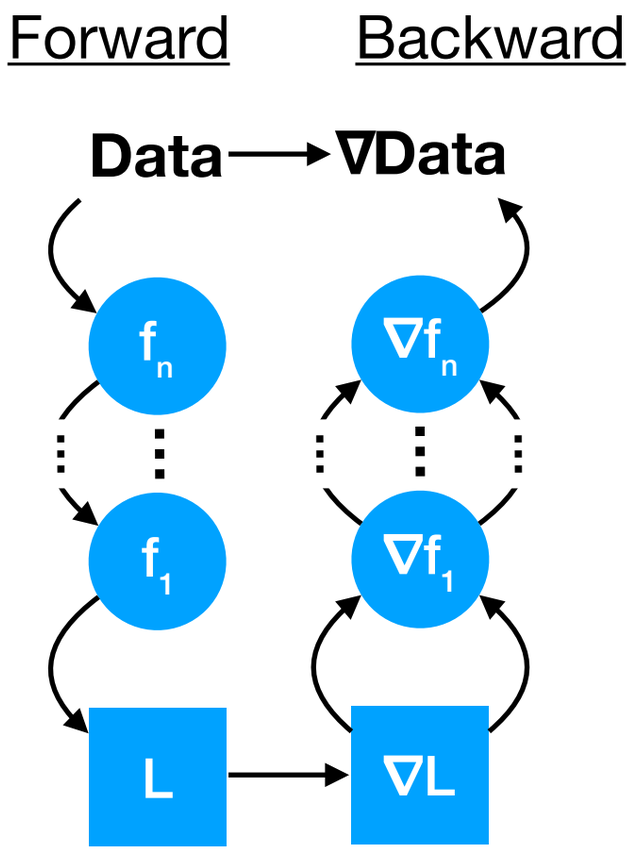



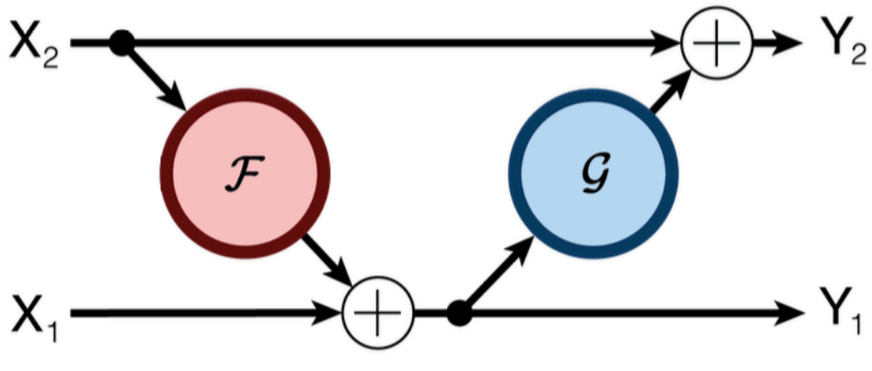

$$y_1 = x_1 + \mathcal{F}(x_2)$$
$$y_2 = x_2 + \mathcal{G}(y_1)$$



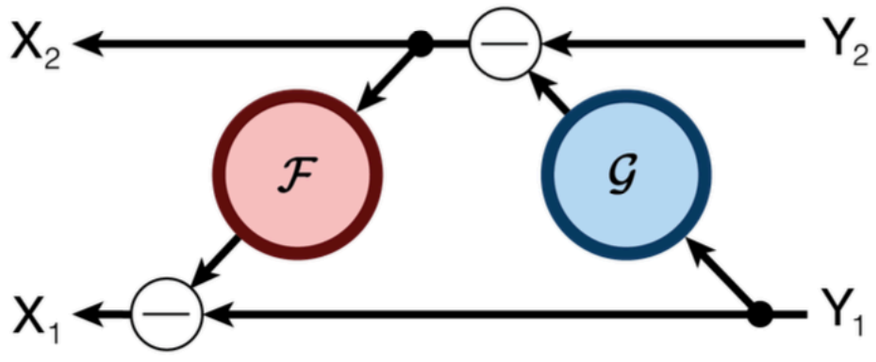

$$x_2 = y_2 - \mathcal{G}(y_1)$$
$$x_1 = y_1 - \mathcal{F}(x_2)$$

# Reversible Residual Networks (RevNet)




$$y_1 = x_1 + \mathcal{F}(x_2)$$
$$y_2 = x_2 + \mathcal{G}(y_1)$$




$$x_2 = y_2 - \mathcal{G}(y_1)$$
$$x_1 = y_1 - \mathcal{F}(x_2)$$

# Results

| Architecture | CIFAR-10 [15] | | CIFAR-100 [15] | |
|---|---|---|---|---|
| | ResNet | RevNet | ResNet | RevNet |
| 32 (38) | **7.14%** | 7.24% | 29.95% | **28.96%** |
| 110 | **5.74%** | 5.76% | 26.44% | **25.40%** |
| 164 | 5.24% | **5.17%** | **23.37%** | 23.69% |

| ResNet-101 | RevNet-104 |
|---|---|
| **23.01%** | 23.10% |

- No noticeable loss in performance
- Massive gains in network depths (~600 vs ~100 layers on ImageNet, using a single GPU)
- 4x increase in batch size (128 vs 32)
- only 1.5x the runtime cost of normal training (in practice, sometimes even less)

# Future Work

- Recurrent Neural Networks
- Application to tasks requiring large activation maps (image segmentation, video, etc.)
- Implement your network within our codebase!
  github.com/renmengye/revnet-public