InvertibleNetworks.jl – Memory efficient deep learning in Julia

Philipp A. Witte[1], Mathias Louboutin[2], Ali Siahkoohi[2], Bas Peters[3], Gabrio[4] Rizzuti and Felix J. Herrmann[2]

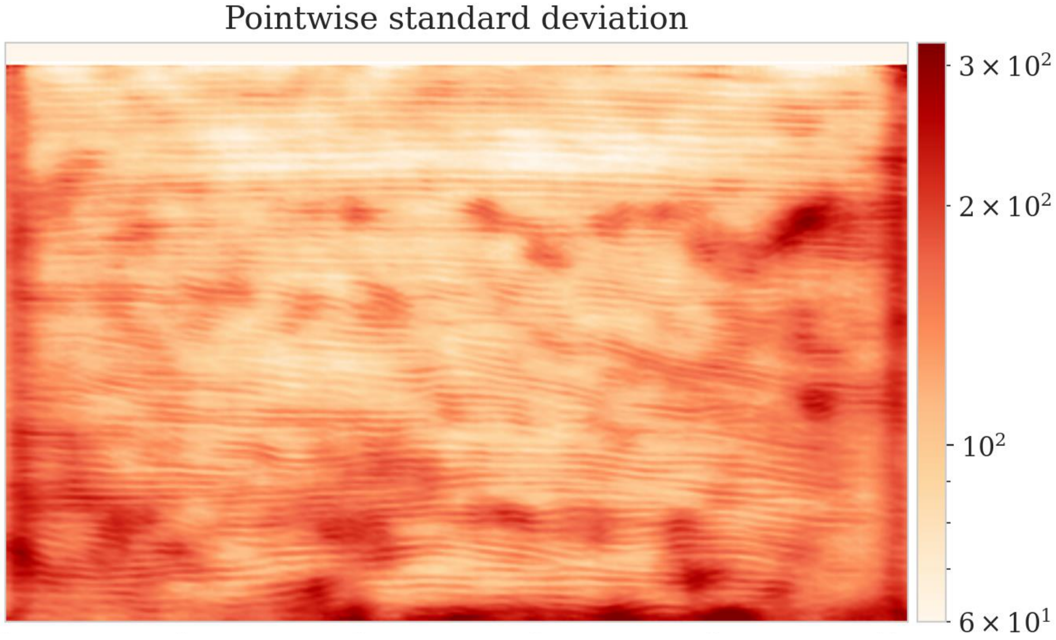
(1) Georgia Institute of Technology, now Microsoft
(2) Georgia Institute of Technology
(3) Emory University
(4) Georgia Institute of Technology, now Utrecht University

Seismic Laboratory for Imaging and Modeling

Invertible Neural Networks

(Kingma & Dhariwal, 2018)

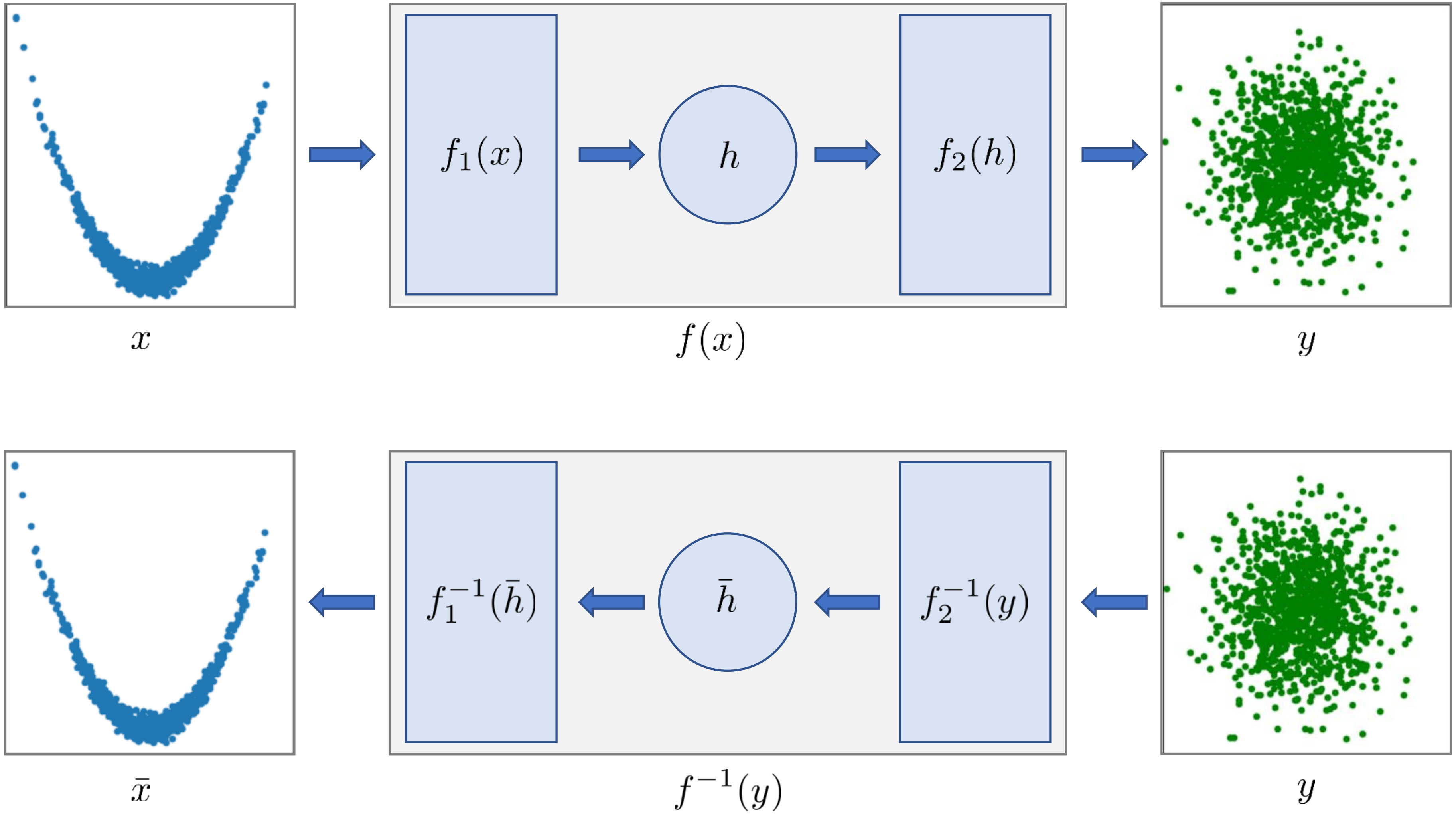
Generative Modeling via Normalizing Flows



(Siahkoohi et al., 2021)

Inverse problems & Uncertainty quantification (UQ)

Invertible Neural Networks



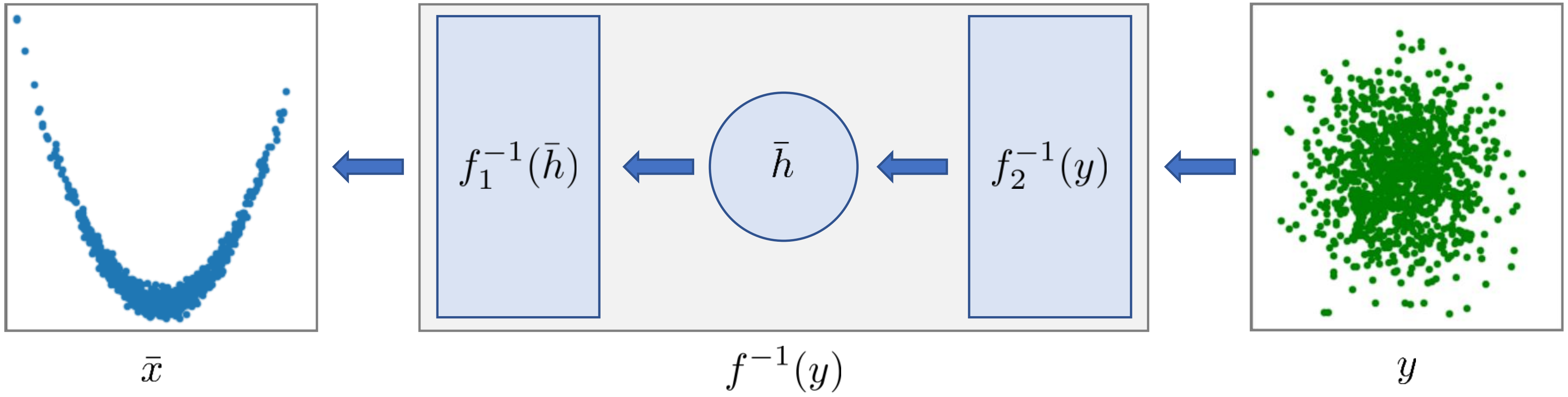
$f_1(x)$ → h → $f_2(h)$

x | $f(x)$ | y

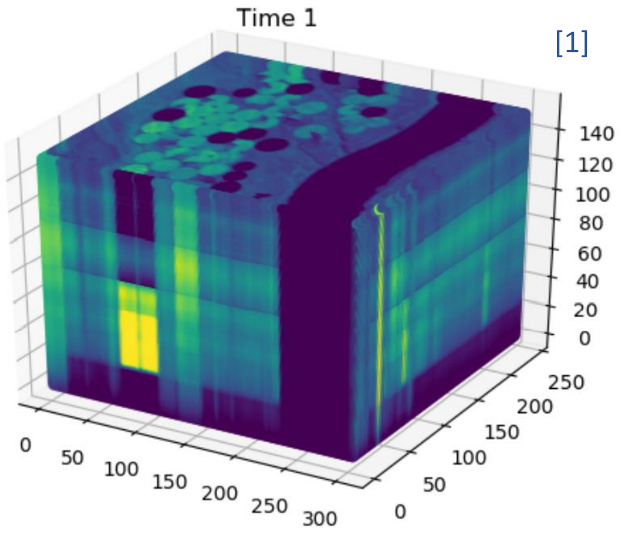
$f_1^{-1}(\bar{h})$ ← $\bar{h}$ ← $f_2^{-1}(y)$

$\bar{x}$ | $f^{-1}(y)$ | y

Invertible Neural Networks



- Train deep 3D neural networks
  - Take advantage of invertibility
  - No need to store hidden states



[1]

[1] Peters et al., Fully reversible neural networks for large-scale surface and sub-surface characterization via remote sensing. arXiv preprints, 2020.

INN & NF frameworks

- Relevant Julia packages
  - Flux.jl [1]
  - Knet.jl [2]
  - Bijections.jl [3]
  - No specific INN & NF frameworks

→ **No frameworks that optimally take advantage of invertibility**

- Python packages
  - Frameworks for Easily Invertible Architectures (FrEIA) [4]
  - MemCNN [5]
  - PyTorch Normalizing Flows [6]

- Papers with code
  - Glow [7]
  - Invertible RIM + Fast MRI [8]
  - Invertible Residual Networks [9]
  - Etc.

[1] Innes, Mike. *Flux: Elegant Machine Learning with Julia*, Journal of Open Source Software, 2018

[2] Yuret, Deniz. *Knet: Beginning deep learning with 100 lines of Julia.* Machine Learning Systems Workshop, NeurIPS, 2016

[3] Scheinerman et al., *Bijections.jl.* https://github.com/scheinerman/Bijections.jl, 2021

[4] Kruse et al., *FrEIA*. https://github.com/VLL-HD/FrEIA, 2021

[5] Van de Leemput et al. *MemCNN: A Python/PyTorch package for creating memory-efficient INNs*, Journal of Open Source Software, 2019

[6] Karpathy, Andrew. *PyTorch Normalizing Flows*. https://github.com/karpathy/pytorch-normalizing-flows, 2019.

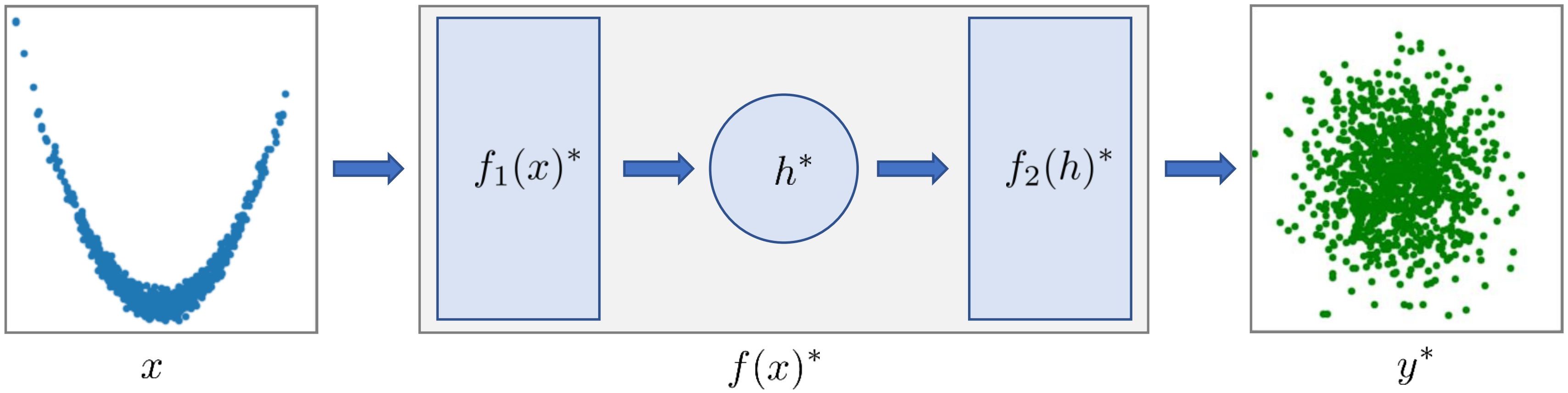
[7] Kingma & Dhariwal. *Glow: Generative Flow with Invertible 1x1 Convolutions*. arXiv preprints, 2018.

[8] Putzky et al. *i-RIM applied to the fastMRI challenge*. ariXiv preprints, 2019.

[9] Behrmann et al. *Invertible Residual Networks*. ICML, 2019.

Training INNs & NFs

Forward pass

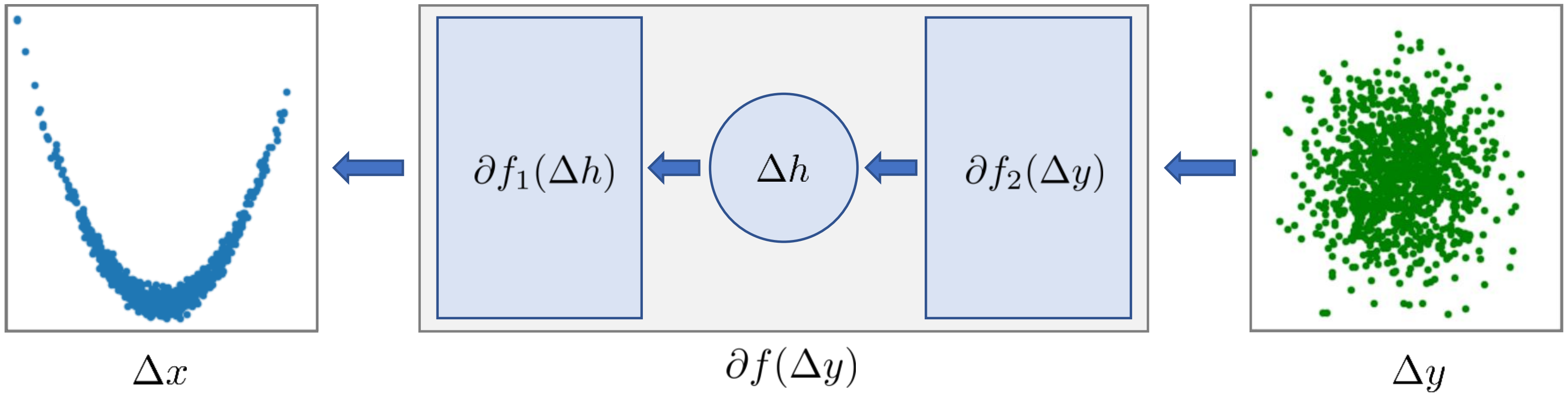


$f_1(x)^*$ → h^* → $f_2(h)^*$

x → $f(x)^*$ → y^*

* tracked

Backprop with AD

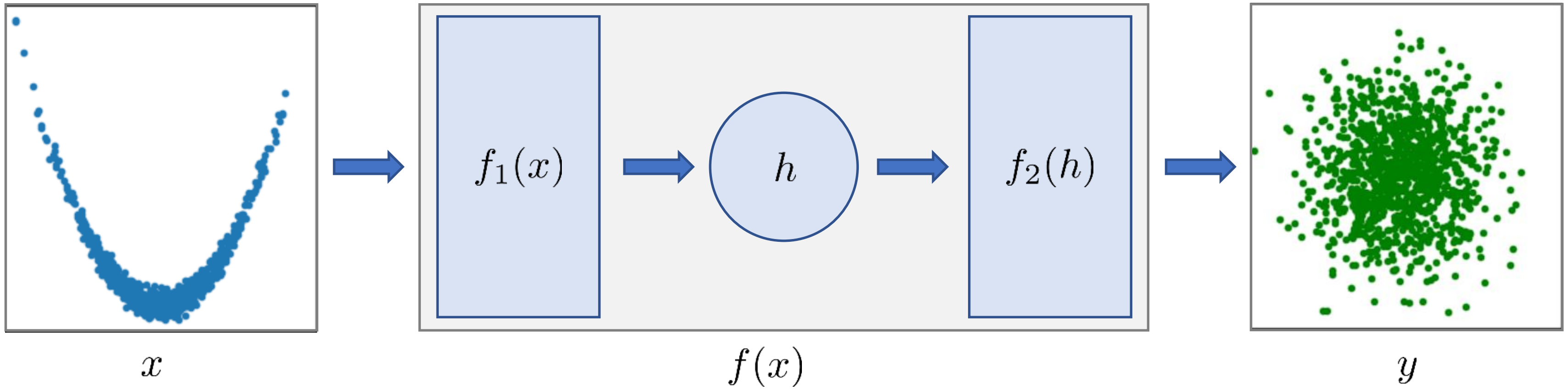


$\partial f_1(\Delta h)$ ← Δh ← $\partial f_2(\Delta y)$

Δx ← $\partial f(\Delta y)$ ← Δy

Training INNs & NFs

Forward pass

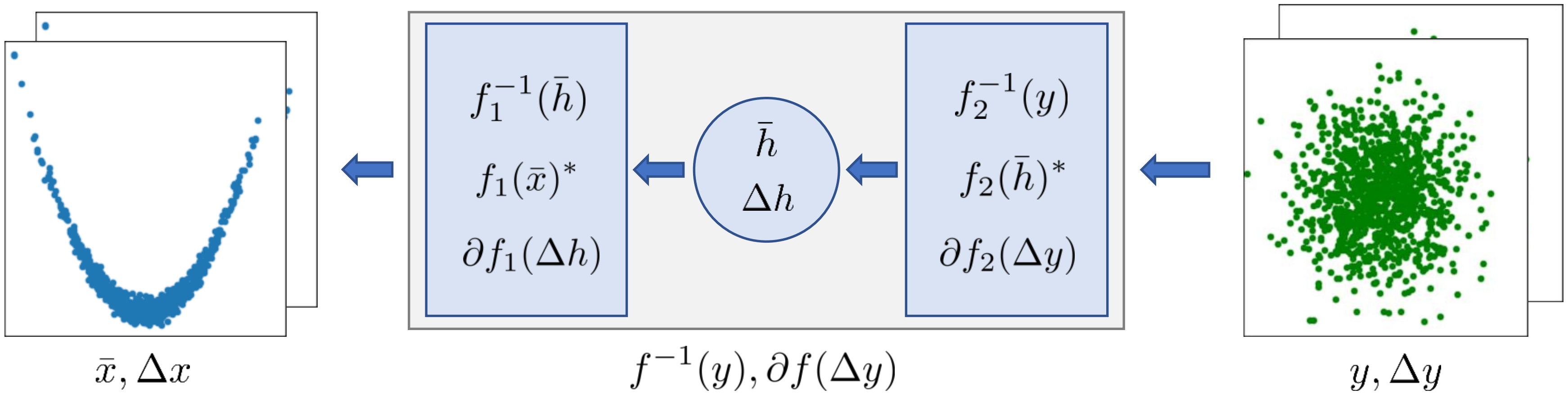


$f_1(x)$ → h → $f_2(h)$

x $f(x)$ y

(no tracking)

Backprop layer-wise AD



$f_1^{-1}(\bar{h})$

$f_1(\bar{x})^*$

$\partial f_1(\Delta h)$

$\bar{h}$

Δh

$f_2^{-1}(y)$

$f_2(\bar{h})^*$

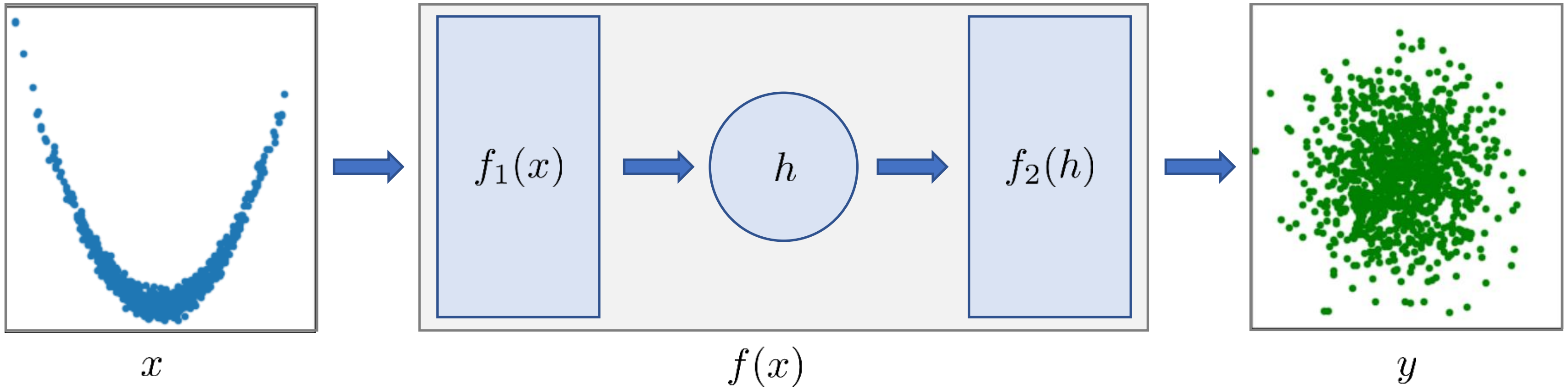
$\partial f_2(\Delta y)$

$\bar{x}, \Delta x$ $f^{-1}(y), \partial f(\Delta y)$ $y, \Delta y$

* tracked

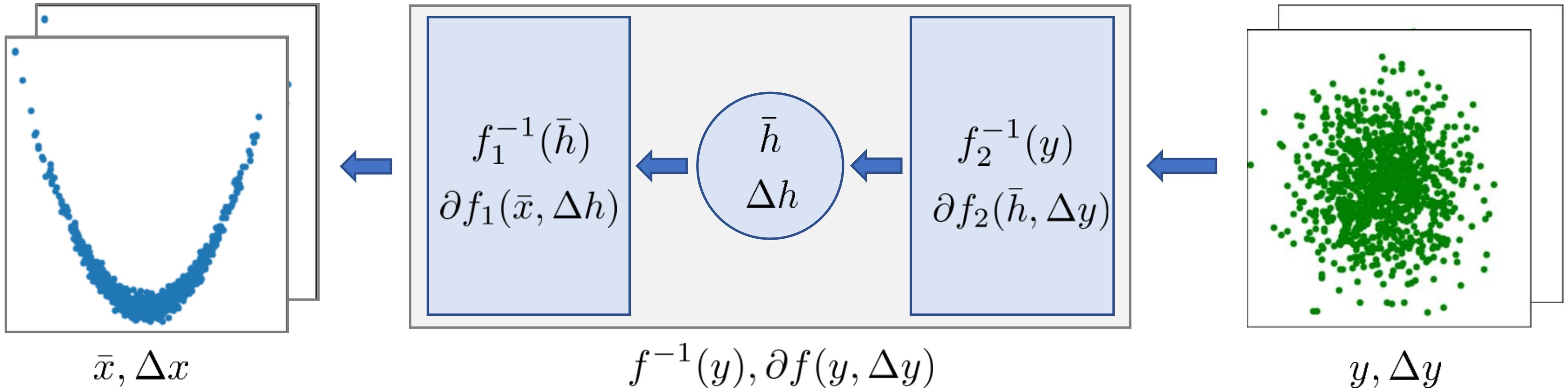
Training INNs & NFs

Forward pass



(no tracking)

Backprop no AD



(no tracking,
no extra forward pass)

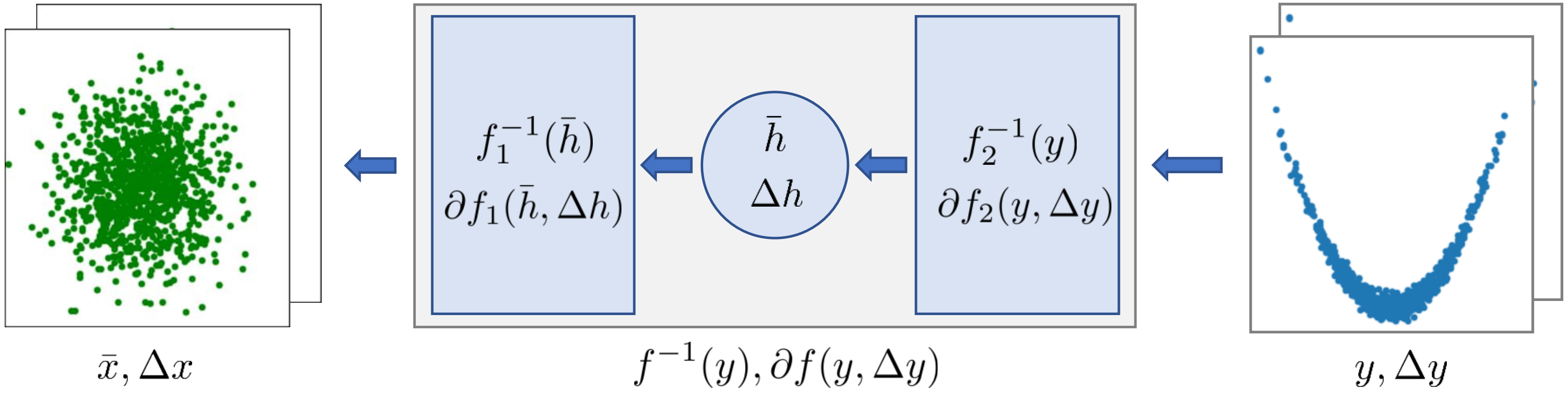
[1] Kruse et al., *HINT: Hierarchical Invertible Neural Transport for Density Estimation and Bayesian Inference*. Proceedings of AAAI, 2021

[2] Putzky et al. *i-RIM applied to the fastMRI challenge*. ariXiv preprints, 2019.

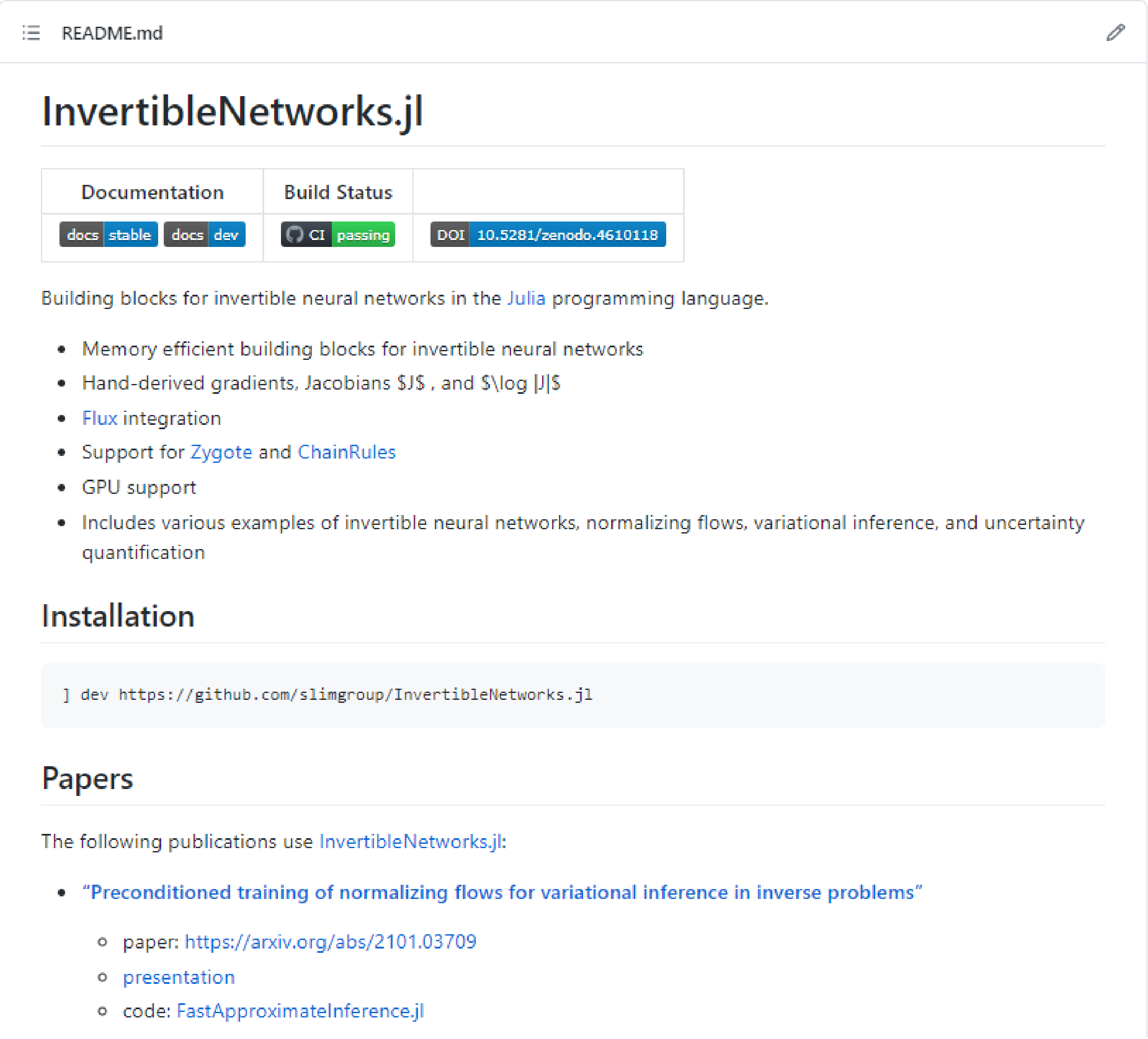
[3] Kingma & Dhariwal. *Glow: Generative Flow with Invertible 1x1 Convolutions*. arXiv preprints, 2018.

InvertibleNetworks.jl

- Memory efficient training for INNs & NFs (MIT license)



- Common building blocks from literature
  - Coupling layers, hyperbolic layers, i-RIM, HINT, 1 x 1 convolutions, etc. [1-3]
  - Log-dets for training via change of variables
  - Forward + adjoint Jacobians (forward + backward differentiation)



README.md

InvertibleNetworks.jl

Documentation	Build Status	
docs stable docs dev	CI passing	DOI 10.5281/zenodo.4610118

Building blocks for invertible neural networks in the Julia programming language.

- Memory efficient building blocks for invertible neural networks
- Hand-derived gradients, Jacobians J , and $\log |J|$
- Flux integration
- Support for Zygote and ChainRules
- GPU support
- Includes various examples of invertible neural networks, normalizing flows, variational inference, and uncertainty quantification

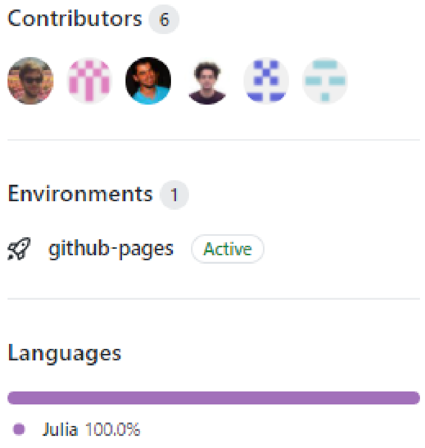
Installation

```
] dev https://github.com/slimgroup/InvertibleNetworks.jl
```

Papers

The following publications use InvertibleNetworks.jl:

- “Preconditioned training of normalizing flows for variational inference in inverse problems”
  - paper: https://arxiv.org/abs/2101.03709
  - presentation
  - code: FastApproximateInference.jl



Contributors 6

Environments 1

github-pages Active

Languages

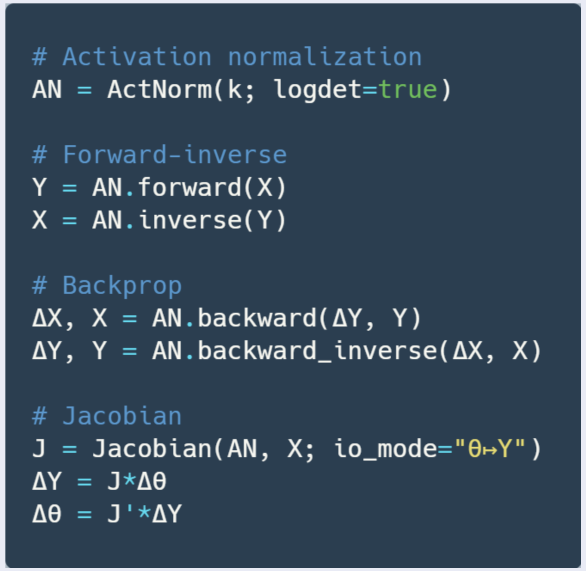
Julia 100.0%

Github repository (MIT license)

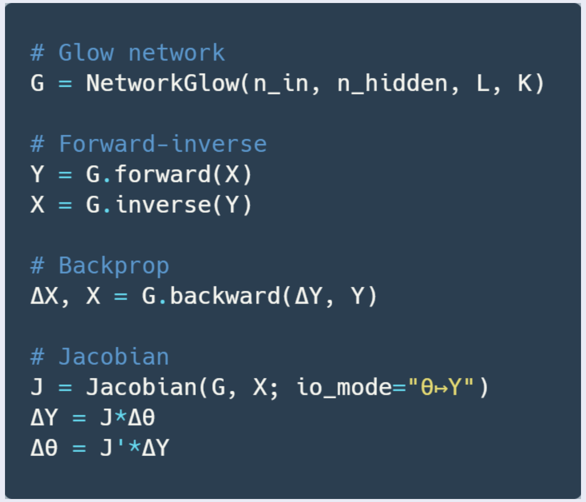
https://github.com/slimgroup/InvertibleNetworks.jl

Package overview

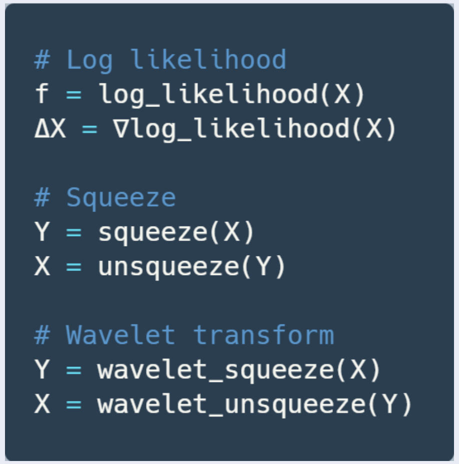
Invertible layers	Invertible networks	Utilities	Examples
• Coupling layers (affine, additive, Glow, HINT) • Hyperbolic layers • Activation normalization • 1 x 1 convolutions w/ Householder matrices • Log-dets for NFs	• Glow • Hyperbolic networks • HINT • i-RIM	• Activations • Dimensionality operations (squeeze, checkerboard, wavelet transform) • Objective functions • Log-dets	• Generative models • Seismic imaging/inversion • Image segmentation • Loop unrolling for inverse problems



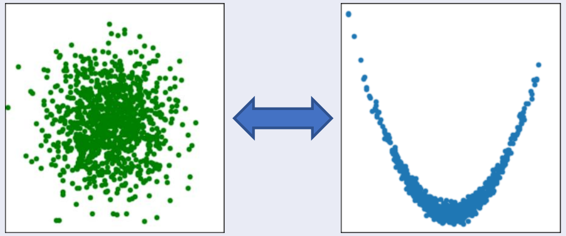
```
# Activation normalization
AN = ActNorm(k; logdet=true)

# Forward-inverse
Y = AN.forward(X)
X = AN.inverse(Y)

# Backprop
ΔX, X = AN.backward(ΔY, Y)
ΔY, Y = AN.backward_inverse(ΔX, X)

# Jacobian
J = Jacobian(AN, X; io_mode="θ↔Y")
ΔY = J*Δθ
Δθ = J'*ΔY
```



```
# Glow network
G = NetworkGlow(n_in, n_hidden, L, K)

# Forward-inverse
Y = G.forward(X)
X = G.inverse(Y)

# Backprop
ΔX, X = G.backward(ΔY, Y)

# Jacobian
J = Jacobian(G, X; io_mode="θ↔Y")
ΔY = J*Δθ
Δθ = J'*ΔY
```

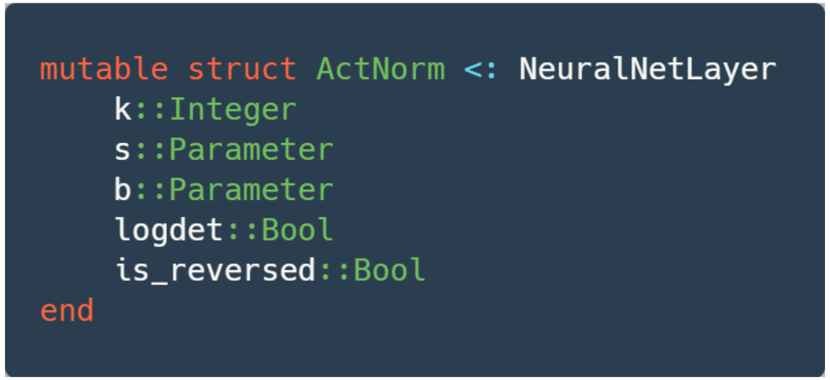


```
# Log likelihood
f = log_likelihood(X)
ΔX = ∇log_likelihood(X)

# Squeeze
Y = squeeze(X)
X = unsqueeze(Y)

# Wavelet transform
Y = wavelet_squeeze(X)
X = wavelet_unsqueeze(Y)
```



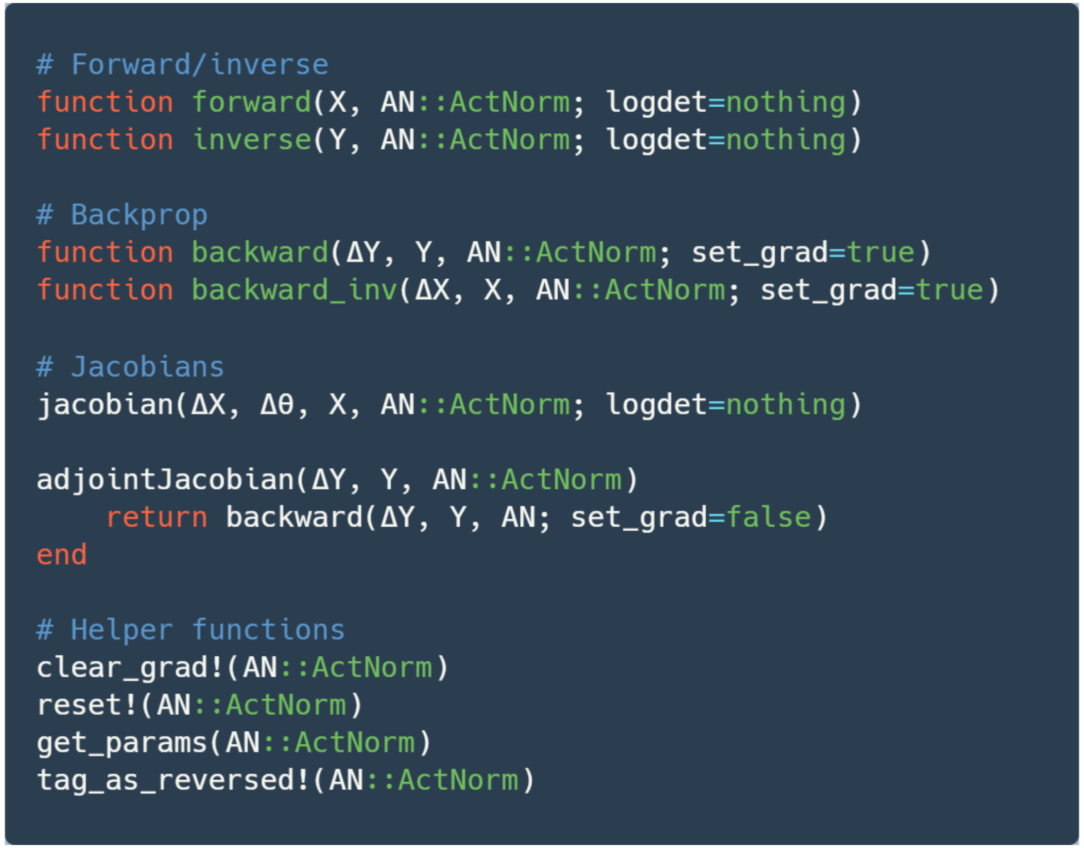
Architecture

- Each layer is mutable structure with associated methods



```
mutable struct ActNorm <: NeuralNetLayer
    k::Integer
    s::Parameter
    b::Parameter
    logdet::Bool
    is_reversed::Bool
end
```

Code structure of invertible layers

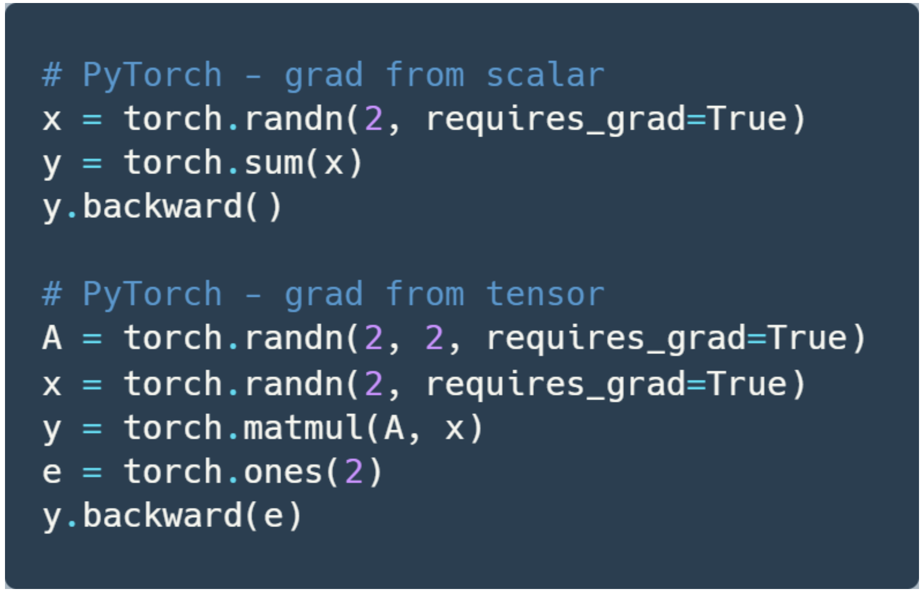


```
# Forward/inverse
function forward(X, AN::ActNorm; logdet=nothing)
function inverse(Y, AN::ActNorm; logdet=nothing)

# Backprop
function backward(ΔY, Y, AN::ActNorm; set_grad=true)
function backward_inv(ΔX, X, AN::ActNorm; set_grad=true)

# Jacobians
jacobian(ΔX, Δθ, X, AN::ActNorm; logdet=nothing)

adjointJacobian(ΔY, Y, AN::ActNorm)
    return backward(ΔY, Y, AN; set_grad=false)
end

# Helper functions
clear_grad!(AN::ActNorm)
reset!(AN::ActNorm)
get_params(AN::ActNorm)
tag_as_reversed!(AN::ActNorm)
```

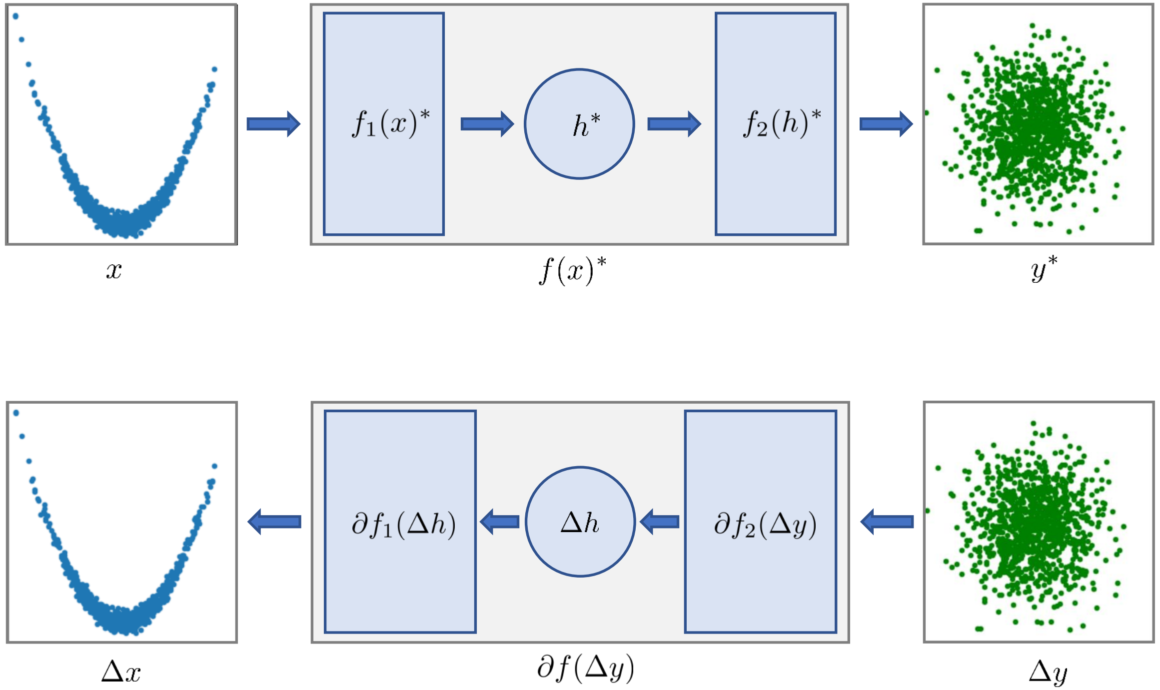
Gradients & Jacobians

PyTorch Autograd

- Does not take original input as argument
- Input tracked during forward pass
- Same for TensorFlow, Flux, etc.



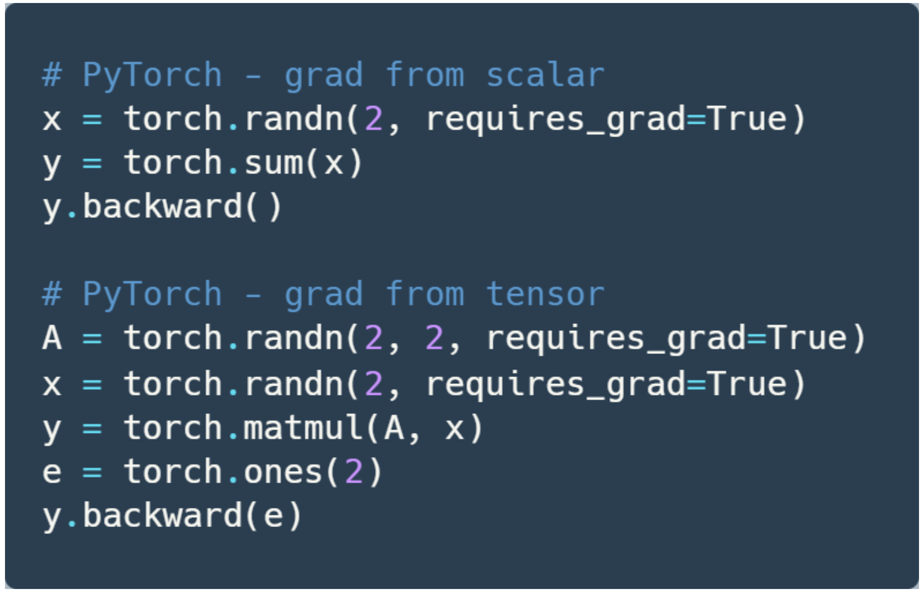
```
# PyTorch - grad from scalar
x = torch.randn(2, requires_grad=True)
y = torch.sum(x)
y.backward()

# PyTorch - grad from tensor
A = torch.randn(2, 2, requires_grad=True)
x = torch.randn(2, requires_grad=True)
y = torch.matmul(A, x)
e = torch.ones(2)
y.backward(e)
```



Gradients & Jacobians

PyTorch Autograd

- Does not take original input as argument
- Input tracked during forward pass
- Same for TensorFlow, Flux, etc.



```
# PyTorch - grad from scalar
x = torch.randn(2, requires_grad=True)
y = torch.sum(x)
y.backward()

# PyTorch - grad from tensor
A = torch.randn(2, 2, requires_grad=True)
x = torch.randn(2, requires_grad=True)
y = torch.matmul(A, x)
e = torch.ones(2)
y.backward(e)
```

Backprop w/ layer-wise AD

1. Recompute input (inverse layer)
2. Forward pass w/ tracking enabled
3. Call torch autograd for layer
4. Extract + set gradients
5. Return original input + grads

(**MemCNN**, **i-RIM**)[1-2]

---

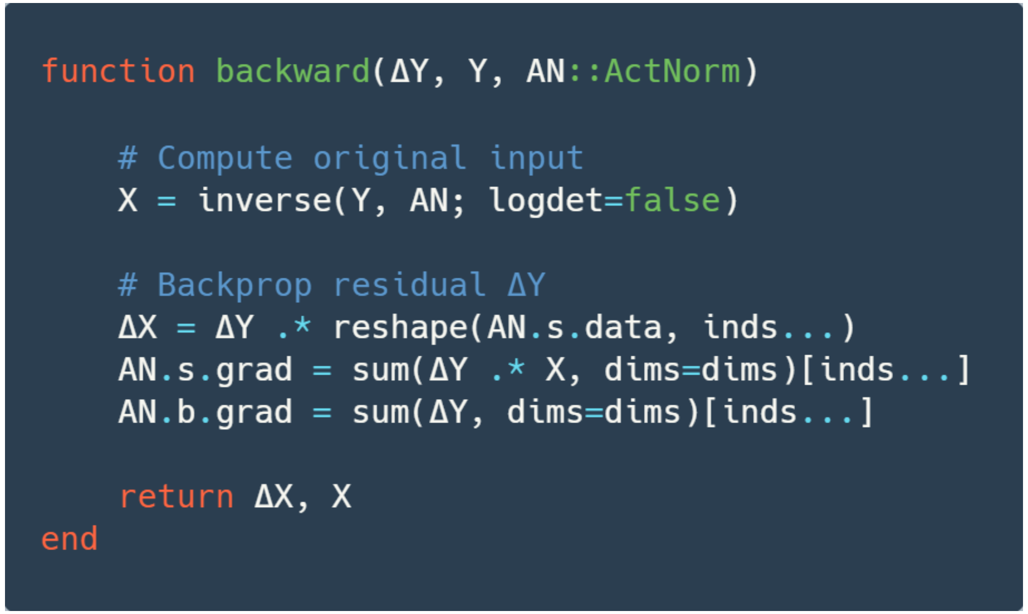
[1] Van de Leemput et al. *MemCNN: A Python/PyTorch package for creating memory-efficient INNs*, Journal of Open Source Software, 2019

[2] Putzky et al. *i-RIM applied to the fastMRI challenge*. ariXiv preprints, 2019.

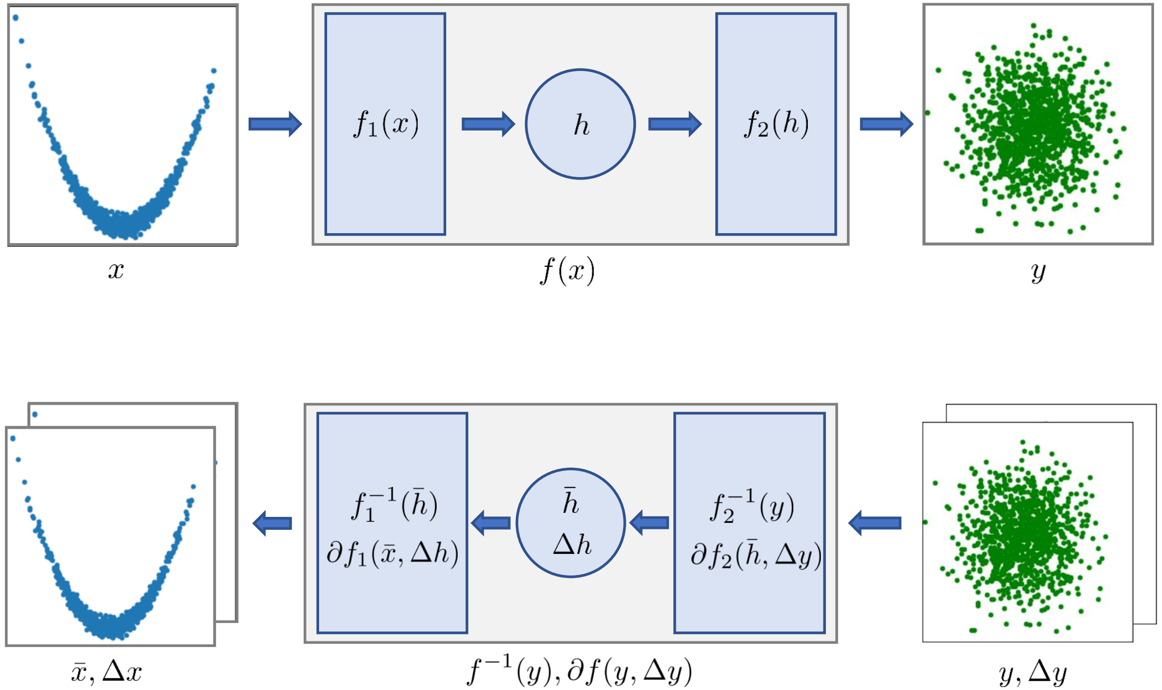
Gradients & Jacobians

InvertibleNetworks

- All-at-once layer for inverse + backward pass
- No tracking of variables

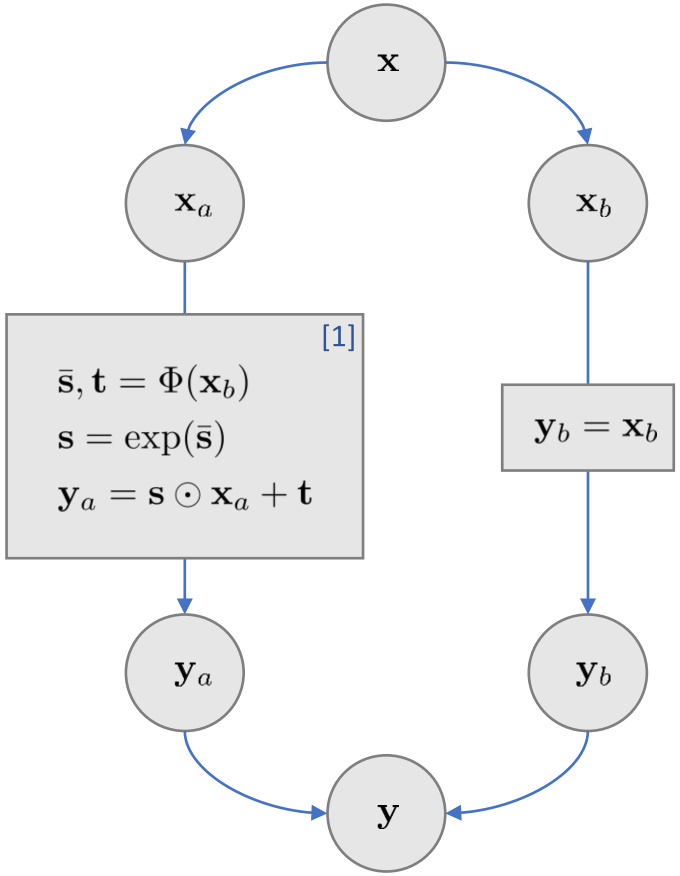


```
function backward(ΔY, Y, AN::ActNorm)

    # Compute original input
    X = inverse(Y, AN; logdet=false)

    # Backprop residual ΔY
    ΔX = ΔY .* reshape(AN.s.data, inds...)
    AN.s.grad = sum(ΔY .* X, dims=dims)[inds...]
    AN.b.grad = sum(ΔY, dims=dims)[inds...]

    return ΔX, X
end
```

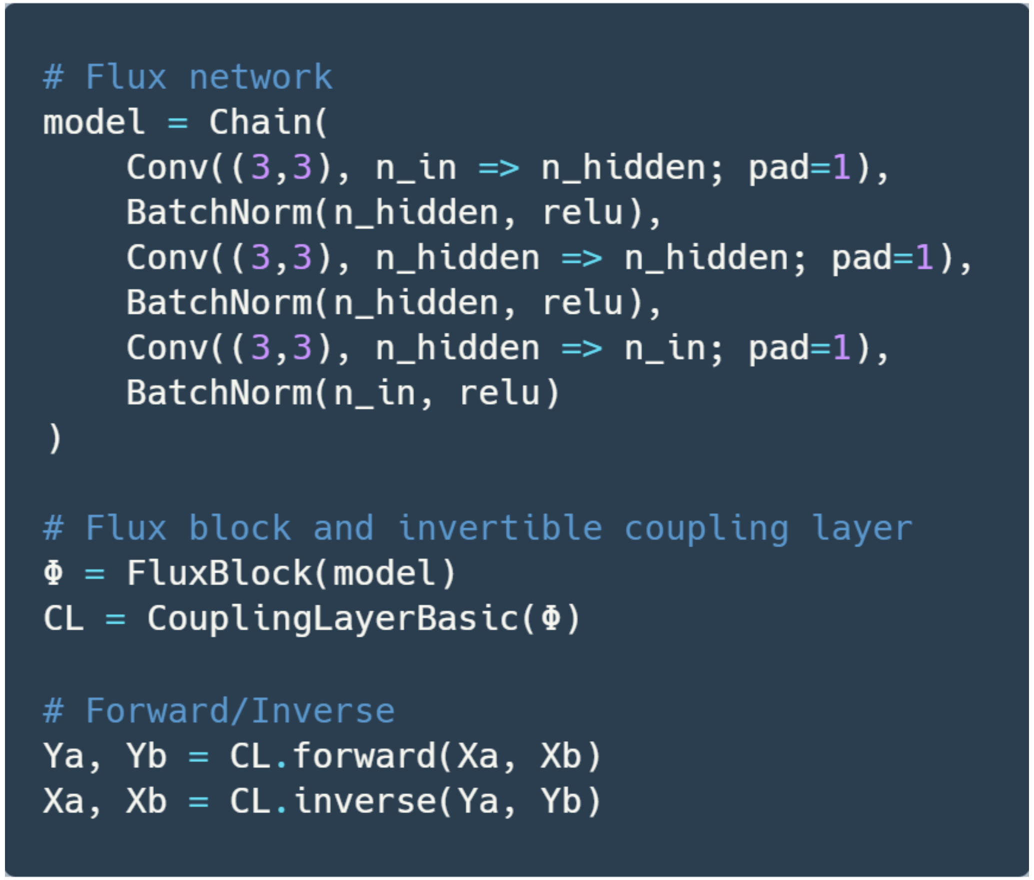


Integration with Flux.jl

Invertible coupling layers with Flux.jl



$\Phi(\mathbf{x})$: Shallow CNN/Res-Net

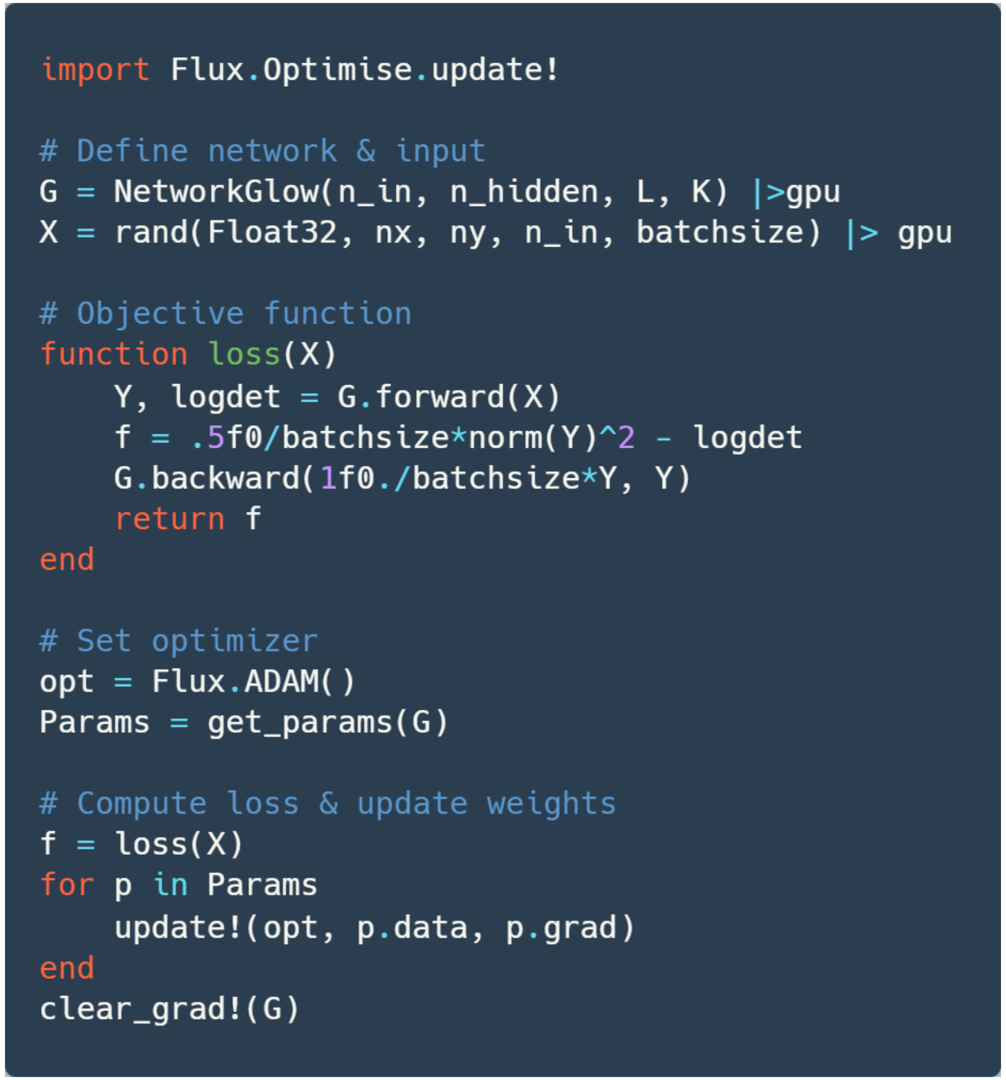


```
# Flux network
model = Chain(
    Conv((3,3), n_in => n_hidden; pad=1),
    BatchNorm(n_hidden, relu),
    Conv((3,3), n_hidden => n_hidden; pad=1),
    BatchNorm(n_hidden, relu),
    Conv((3,3), n_hidden => n_in; pad=1),
    BatchNorm(n_in, relu)
)

# Flux block and invertible coupling layer
Φ = FluxBlock(model)
CL = CouplingLayerBasic(Φ)

# Forward/Inverse
Ya, Yb = CL.forward(Xa, Xb)
Xa, Xb = CL.inverse(Ya, Yb)
```

[1] Dinh et al., *Nice: Non-linear independent component estimation*. arXiv preprints, 2014.

[1] Innes et al., Flux.jl, https://github.com/FluxML/Flux.jl, 2021

Integration with Flux.jl



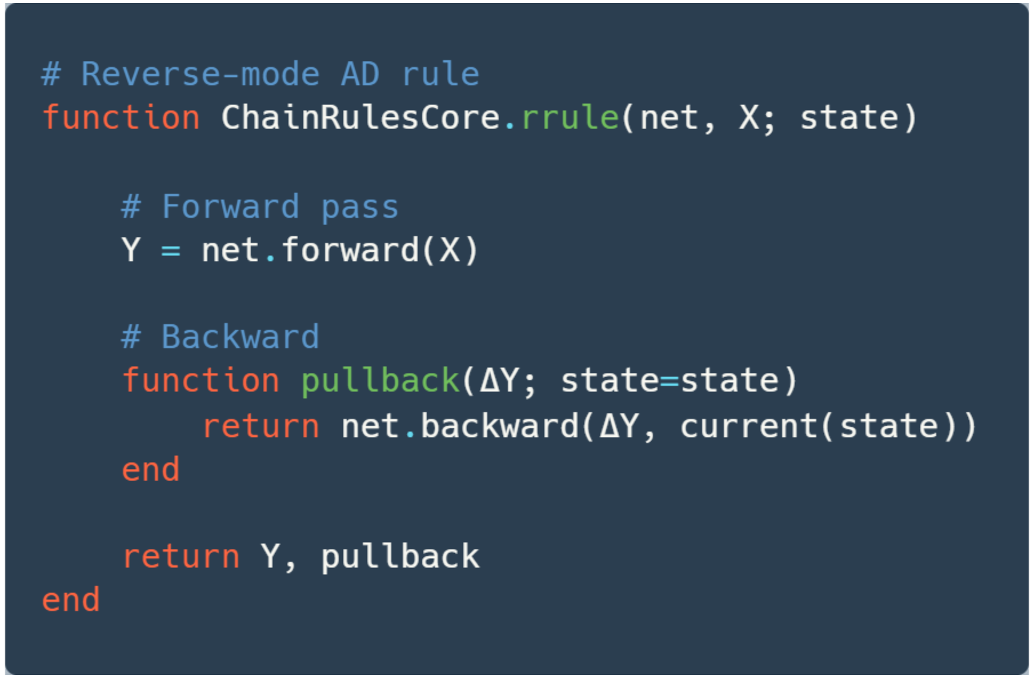
```
import Flux.Optimise.update!

# Define network & input
G = NetworkGlow(n_in, n_hidden, L, K) |>gpu
X = rand(Float32, nx, ny, n_in, batchsize) |> gpu

# Objective function
function loss(X)
    Y, logdet = G.forward(X)
    f = .5f0/batchsize*norm(Y)^2 - logdet
    G.backward(1f0./batchsize*Y, Y)
    return f
end

# Set optimizer
opt = Flux.ADAM()
Params = get_params(G)

# Compute loss & update weights
f = loss(X)
for p in Params
    update!(opt, p.data, p.grad)
end
clear_grad!(G)
```

- Training INNs with Flux [1]
    - Flux optimizers (ADAM, etc.)
    - Update weights of INNs
    - Same as Flux networks

[1] Innes et al., *Flux.jl*, https://github.com/FluxML/Flux.jl, 2021
[2] White et al., *JuliaDiff/ChainRules.jl*, https://github.com/JuliaDiff/ChainRules.jl, 2021

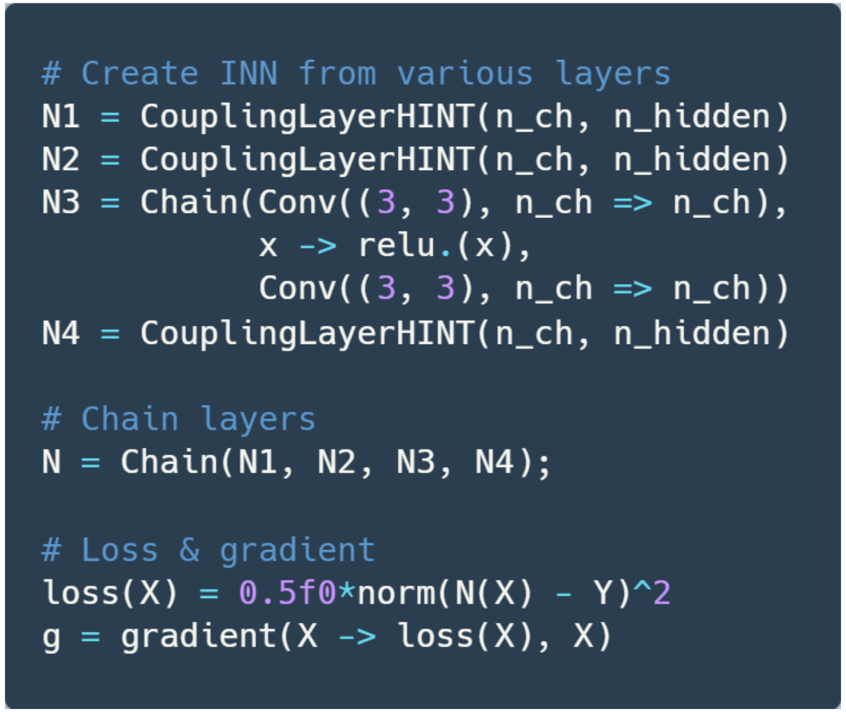
Integration with ChainRules.jl

- Combine INN & Flux layers via ChainRules.jl [1-2]



```
# Reverse-mode AD rule
function ChainRulesCore.rrule(net, X; state)

    # Forward pass
    Y = net.forward(X)

    # Backward
    function pullback(ΔY; state=state)
        return net.backward(ΔY, current(state))
    end

    return Y, pullback
end
```

ChainRule definition for reverse differentiation



```
# Create INN from various layers
N1 = CouplingLayerHINT(n_ch, n_hidden)
N2 = CouplingLayerHINT(n_ch, n_hidden)
N3 = Chain(Conv((3, 3), n_ch => n_ch),
           x -> relu.(x),
           Conv((3, 3), n_ch => n_ch))
N4 = CouplingLayerHINT(n_ch, n_hidden)

# Chain layers
N = Chain(N1, N2, N3, N4);

# Loss & gradient
loss(X) = 0.5f0*norm(N(X) - Y)^2
g = gradient(X -> loss(X), X)
```

Define multi-layer INN

Unit tests

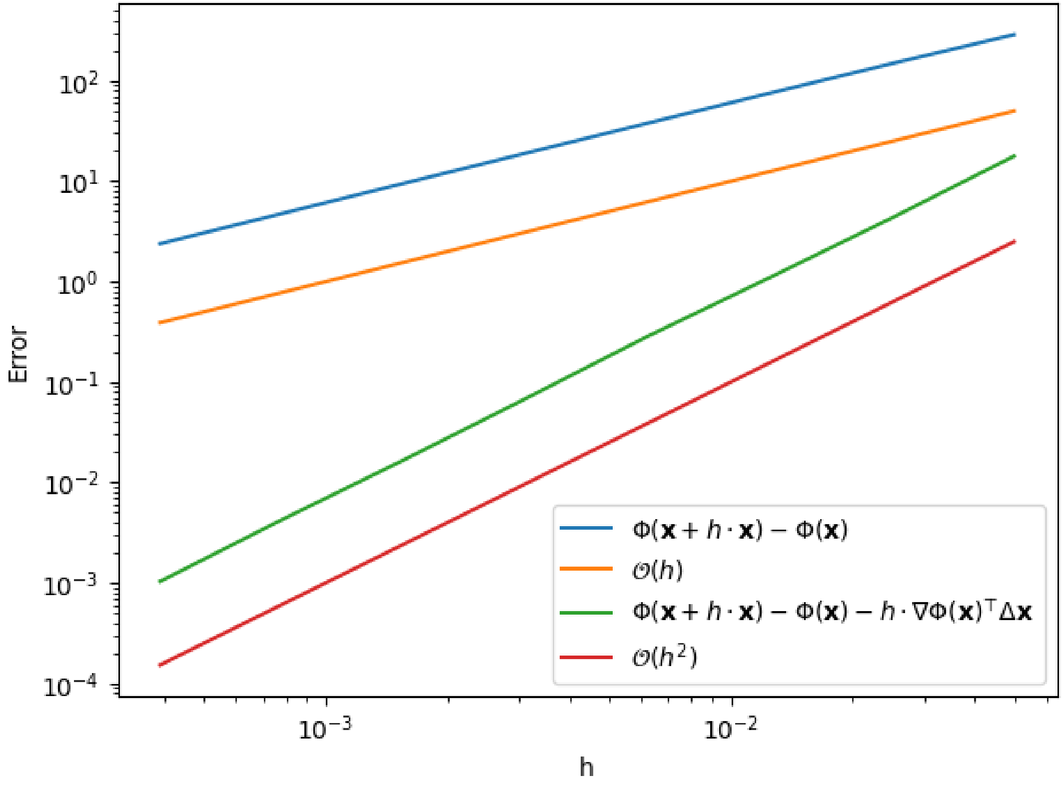
- Adjoint tests for linear operators:

$$\epsilon \leq \langle \mathbf{A}\mathbf{x}, \mathbf{y} \rangle - \langle \mathbf{A}^\top \mathbf{y}, \mathbf{x} \rangle$$

- Gradient tests for (non-) linear layers:

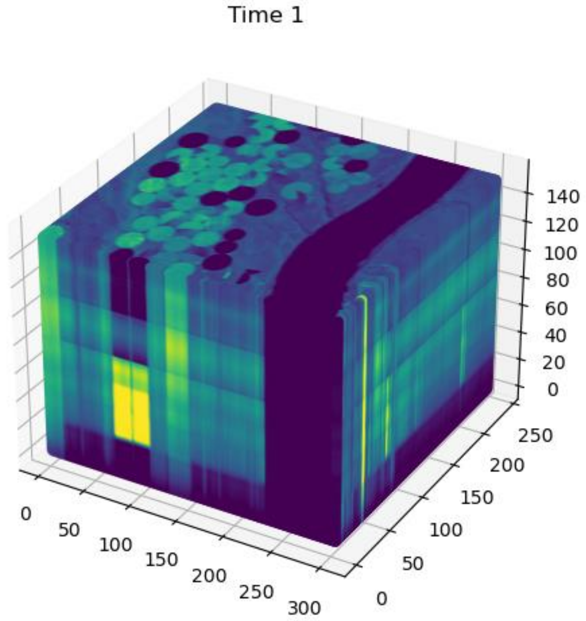
$$\Phi(\mathbf{x} + h \cdot \Delta\mathbf{x}) - \Phi(\mathbf{x}) = \mathcal{O}(h)$$
$$\Phi(\mathbf{x} + h \cdot \Delta\mathbf{x}) - \Phi(\mathbf{x}) - h \cdot \nabla\Phi(\mathbf{x})^\top \Delta\mathbf{x} = \mathcal{O}(h^2)$$

$$\Phi(\mathbf{w} + h \cdot \Delta\mathbf{w}) - \Phi(\mathbf{w}) = \mathcal{O}(h)$$
$$\Phi(\mathbf{w} + h \cdot \Delta\mathbf{w}) - \Phi(\mathbf{w}) - h \cdot \nabla\Phi(\mathbf{w})^\top \Delta\mathbf{w} = \mathcal{O}(h^2)$$



Gradient test for Glow network

Examples & applications



- Example applications
  - Inverse problems & loop unrolling
  - Image segmentation with partial labels and/or weak supervision
  - Normalizing flows & Bayesian inference

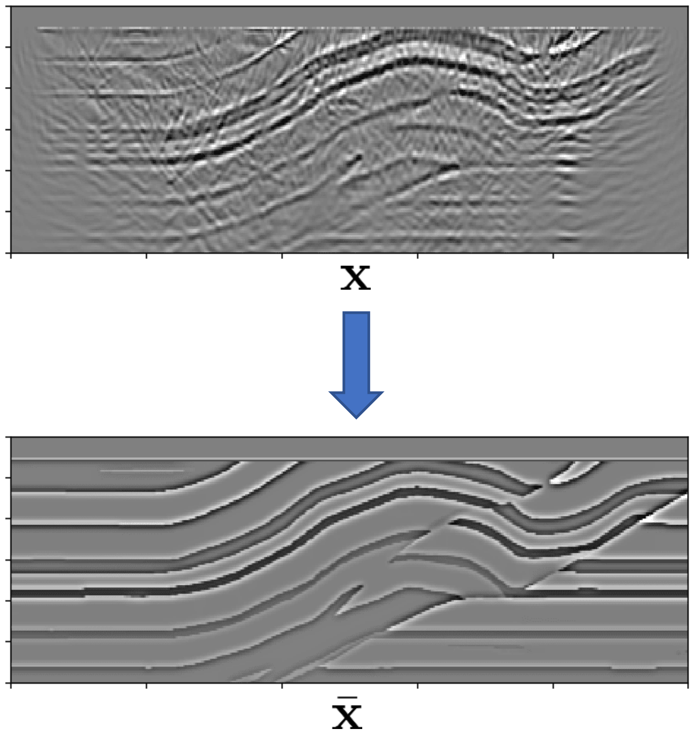
- All examples implemented with InvertibleNetworks.jl
  - Reproducible examples at https://github.com/slimgroup/InvertibleNetworks.jl/tree/master/examples

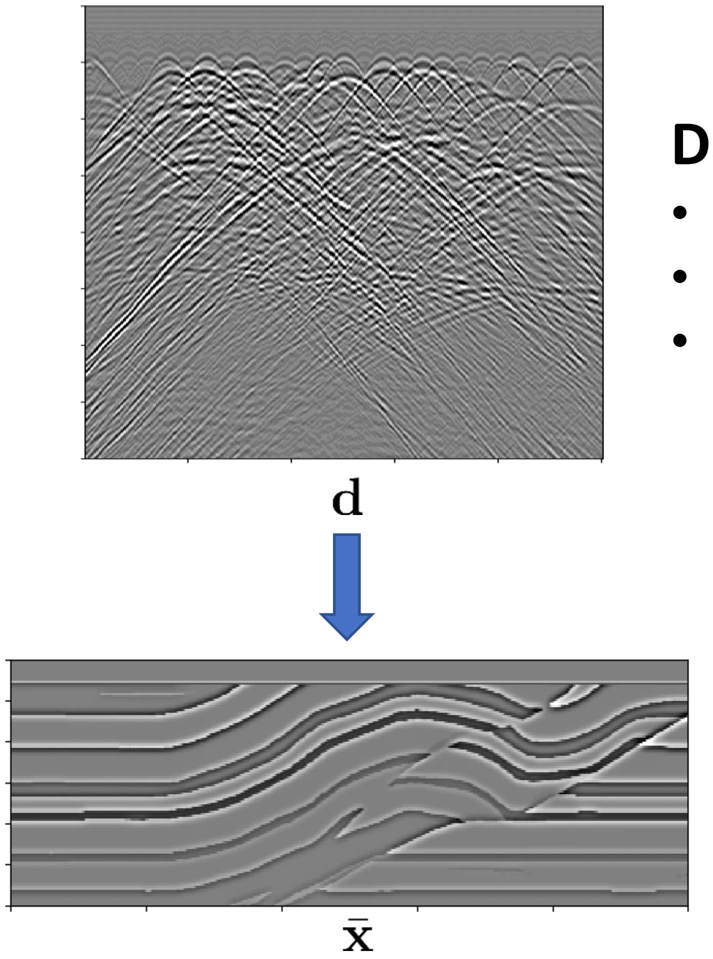
e.g. [1] Adler & Öktem., *Solving ill-posed inverse problems using iterative deep neural networks.jl,* Inverse Problems, 33, 2017
[2] Jin et al. *Deep convolutional neural network for inverse problems in imaging*. IEEE Transactions on Image Processing, 26 (9), 2017

Scenario 1: Loop-unrolled inverse problems [1-2]

Image-to-image mapping

- Learned denoiser/all-at-once
- Fully data-driven





Data-to-image mapping

- Data $\mathbf{d} = \mathbf{J}\mathbf{x}$
- Fully data-driven *or*
- Physics-augmented/iterative (use operator $\mathbf{J}$)

[1] Putzky & Welling, *Recurrent inference machines for solving inverse problems*, arXiv preprints, 2017
[2] Putzky & Welling, *Invert to learn to invert*, NIPS Proceedings, 2019

Scenario 1: Loop-unrolled inverse problems

Objective function for supervised learning

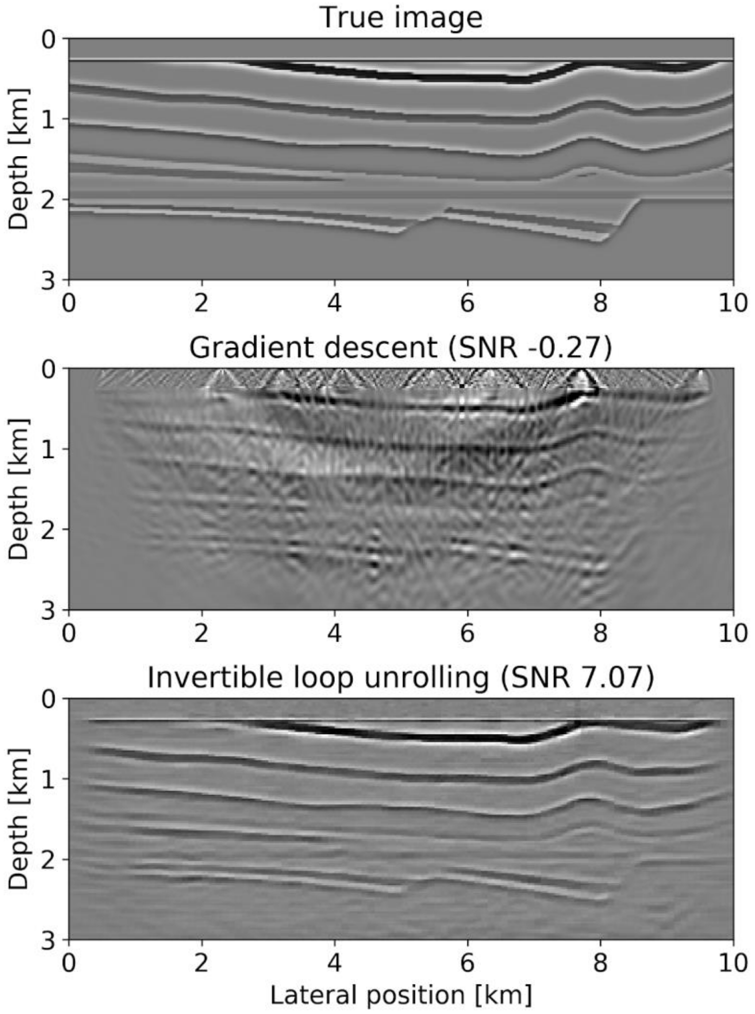
$$\underset{\theta}{\text{minimize}} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2}\|\mathcal{G}_\theta(\mathbf{J}_i, \mathbf{d}_i) - \bar{\mathbf{x}}_i\|_2^2$$

with

```
function $\mathcal{G}(\mathbf{J}, \mathbf{d})$
    $\mathbf{x} = 0$
    for $j = 1, ..., n$
        $\mathbf{x} = \mathbf{Q}\mathbf{x}$
        $\mathbf{x}'_1 = \mathbf{x}_1$
        $\mathbf{g} = \mathbf{J}^\top(\mathbf{J}\mathbf{x}'_1[1] - \mathbf{d})$
        $\mathbf{s}', \mathbf{t} = \mathrm{NN}([\mathbf{g}, \mathbf{x}'_1[2{:}\mathrm{end}]])$
        $\mathbf{s} = \sigma(\mathbf{s}')$
        $\mathbf{x}'_2 = \mathbf{x}_2 \odot \mathbf{s} + \mathbf{t}$
        $\mathbf{x} = \mathbf{Q}^\top\mathbf{x}'$
    end
    return $\mathbf{x}$
end
```

Invertible recurrent inference machine (i-RIM)[1-2]

Results after
4 training epochs



[1] Putzky & Welling, *Recurrent inference machines for solving inverse problems*, arXiv preprints, 2017
[2] Putzky & Welling, *Invert to learn to invert*, NIPS Proceedings, 2019

Scenario 1: Loop-unrolled inverse problems

Objective function for supervised learning

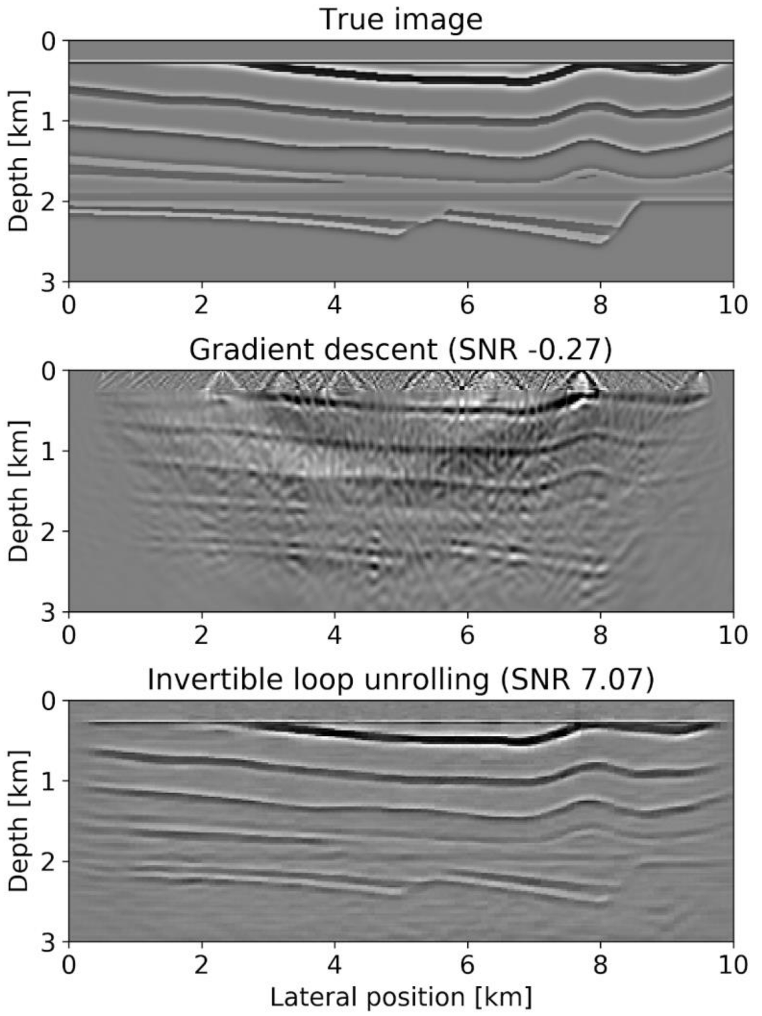
$$\underset{\theta}{\text{minimize}} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2}\|\mathcal{G}_\theta(\mathbf{J}_i,\mathbf{d}_i)-\bar{\mathbf{x}}_i\|_2^2$$

with

```
function $\mathcal{G}(\mathbf{J},\mathbf{d})$
    $\mathbf{x}=0$
    for $j=1,...,n$
        $\mathbf{x}=\mathbf{Q}\mathbf{x}$
        $\mathbf{x}_1'=\mathbf{x}_1$
        $\mathbf{g}=\mathbf{J}^\top(\mathbf{J}\mathbf{x}_1'[1]-\mathbf{d})$
        $\mathbf{s}',\mathbf{t}=\mathrm{NN}([\mathbf{g},\mathbf{x}_1'[2\text{:end}]])$
        $\mathbf{s}=\sigma(\mathbf{s}')$
        $\mathbf{x}_2'=\mathbf{x}_2\odot\mathbf{s}+\mathbf{t}$
        $\mathbf{x}=\mathbf{Q}^\top\mathbf{x}'$
    end
    return $\mathbf{x}$
end
```

Invertible recurrent inference machine (i-RIM)[1-2]

Results after 4 training epochs


True image

Gradient descent (SNR -0.27)

Invertible loop unrolling (SNR 7.07)

Scenario 1: Loop-unrolled inverse problems

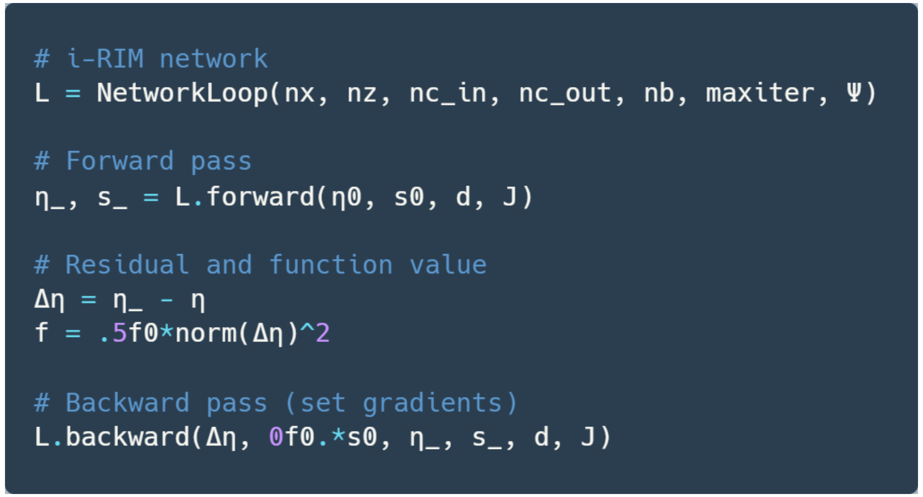
Objective function for supervised learning

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2} \|\mathcal{G}_\theta(\mathbf{J}_i, \mathbf{d}_i) - \bar{\mathbf{x}}_i\|_2^2$$

with

```
function $\mathcal{G}(\mathbf{J}, \mathbf{d})$
    $\mathbf{x} = 0$
    for $j = 1, ..., n$
        $\mathbf{x} = \mathbf{Q}\mathbf{x}$
        $\mathbf{x}_1' = \mathbf{x}_1$
        $\mathbf{g} = \mathbf{J}^\top(\mathbf{J}\mathbf{x}_1'[1] - \mathbf{d})$
        $\mathbf{s}', \mathbf{t} = \mathrm{NN}([\mathbf{g}, \mathbf{x}_1'[2{:}\mathrm{end}]])$
        $\mathbf{s} = \sigma(\mathbf{s}')$
        $\mathbf{x}_2' = \mathbf{x}_2 \odot \mathbf{s} + \mathbf{t}$
        $\mathbf{x} = \mathbf{Q}^\top \mathbf{x}'$
    end
    return $\mathbf{x}$
end
```

Implementation

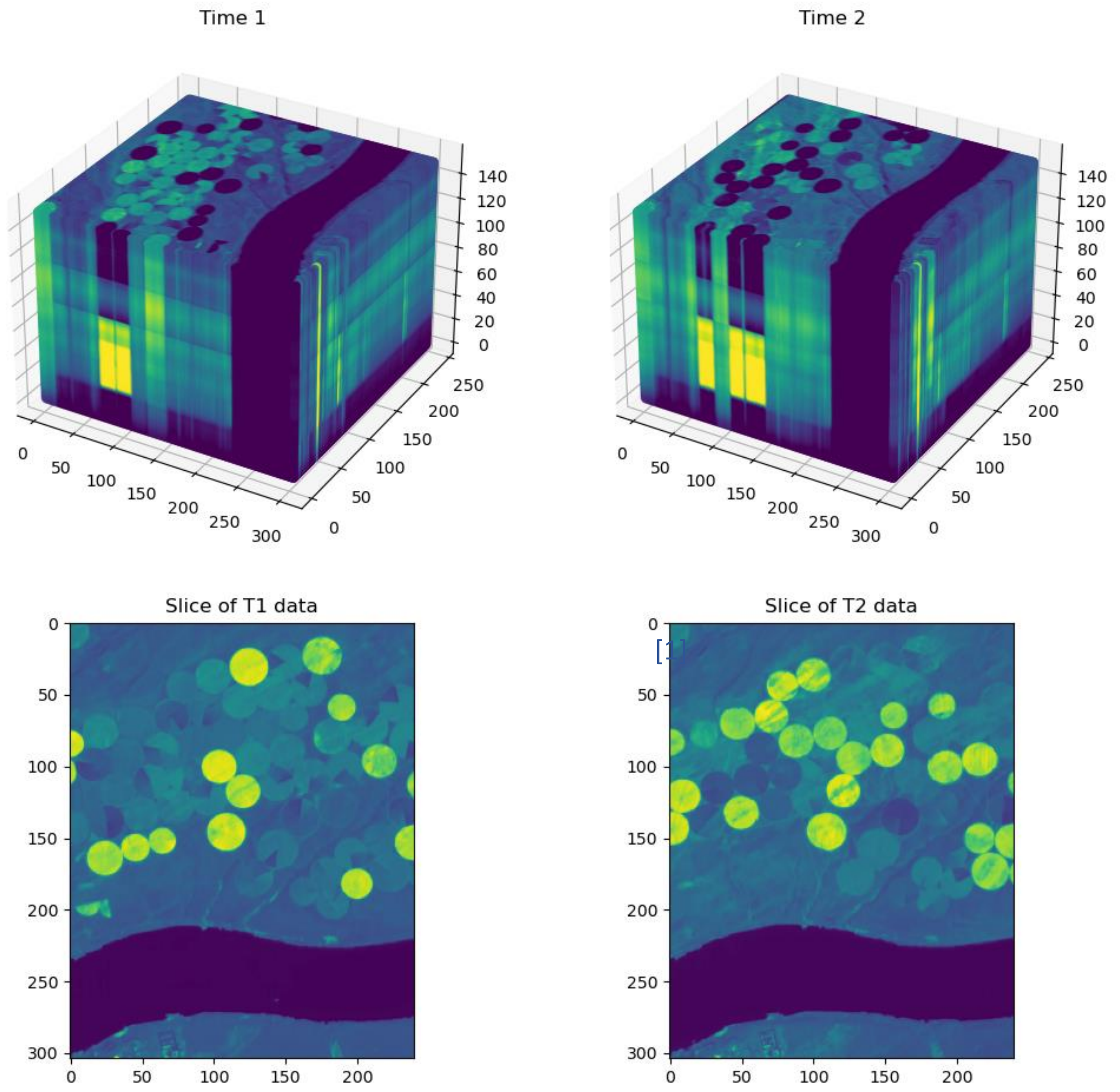


```
# i-RIM network
L = NetworkLoop(nx, nz, nc_in, nc_out, nb, maxiter, Ψ)

# Forward pass
η_, s_ = L.forward(η0, s0, d, J)

# Residual and function value
Δη = η_ - η
f = .5f0*norm(Δη)^2

# Backward pass (set gradients)
L.backward(Δη, 0f0.*s0, η_, s_, d, J)
```

Compatible w/ matrix-free linear operators

Scenario 2: 4D image segmentation

- Time-lapse hyper-spectral land use change [1]
  - Single large-scale 4D input volume (307 x 241 x 154 x 2)
  - 18 layer invertible hyperbolic net with 3D convolutions
  - Coupled space-frequency approach
  - 128 channels



Time 1

Time 2

Slice of T1 data

Slice of T2 data

---

[1] Peters et al., Fully reversible neural networks for large-scale surface and sub-surface characterization via remote sensing. arXiv preprints, 2020.

Scenario 2: 4D image segmentation

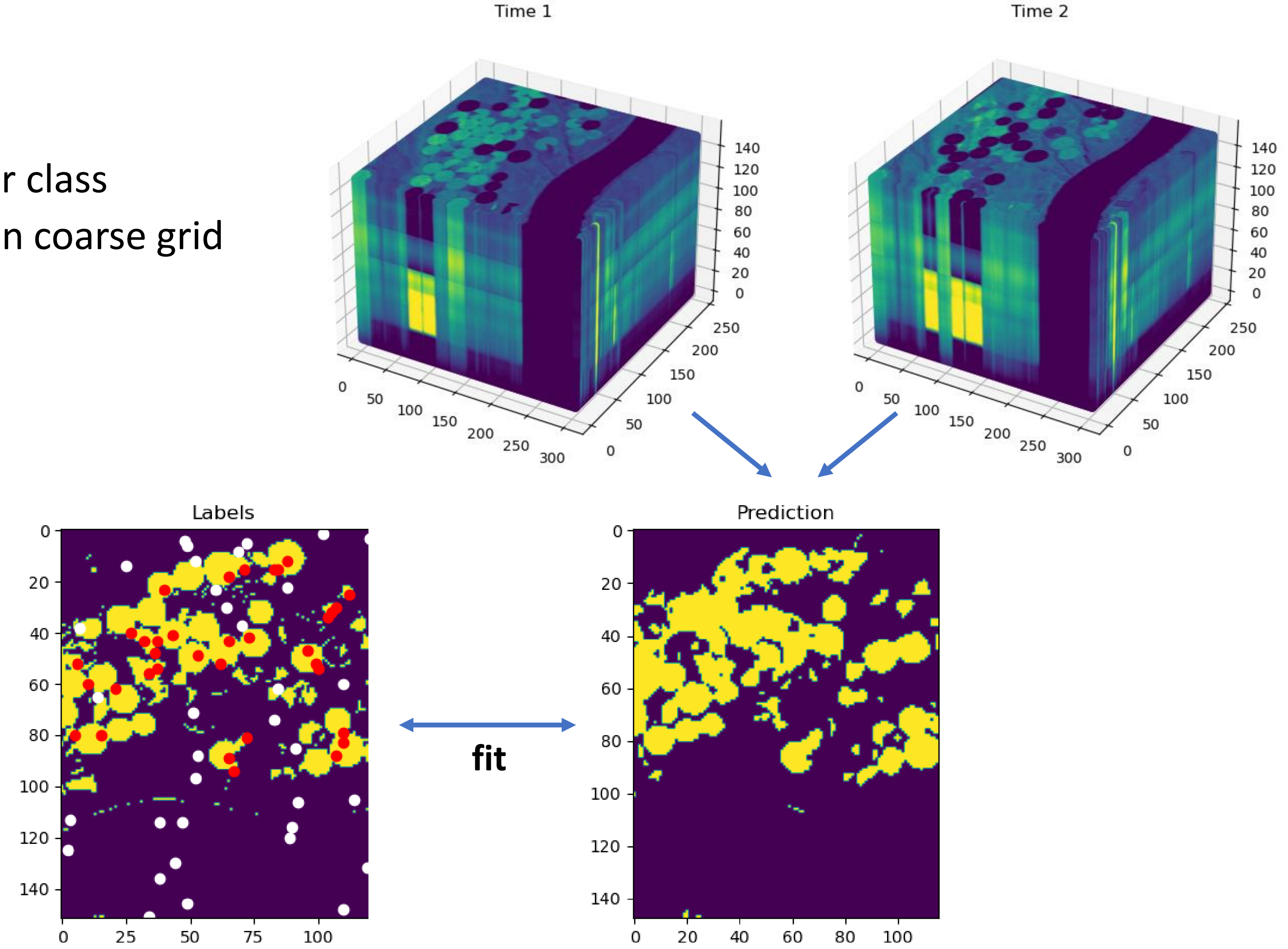
Goal: predict land use change

- Only 35 point annotations per class
- Predict change everywhere on coarse grid

Memory requirements

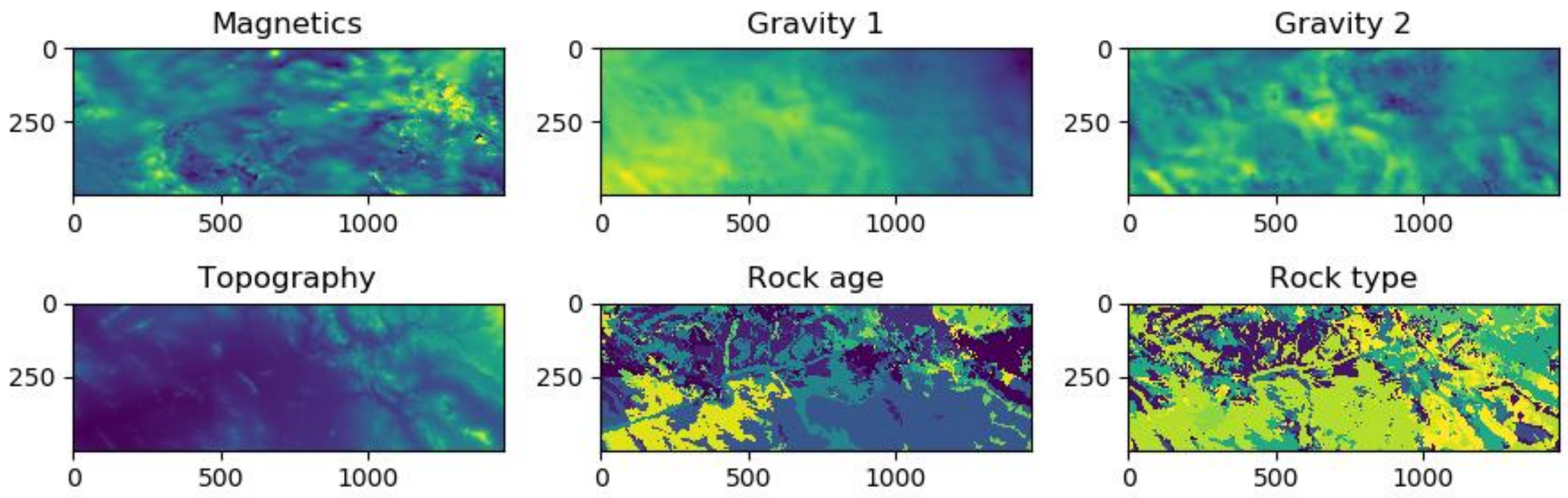
- 18 layer INN, 128 channels
- Image: 307 x 241 x 154 x 2
- Invertible hyperbolic net: **17 GB**
- Non-invertible equivalent: **307 GB**

Enable deeper networks and/or larger data sets

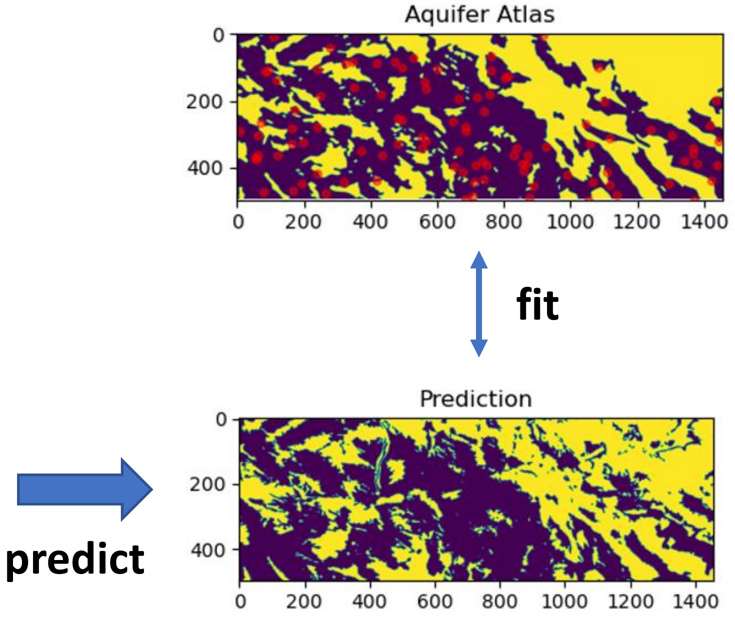


Scenario 3: Weakly supervised segmentation

- Goal: Map out geological aquifers from multi-modal geophysical data [1]
  - Class 1: partial point annotations
  - Class 2: No labels, occupies ~50 to 65 % per domain
  - Learn from partial label + priors using *constrained optimization*



Input data: 1,450 x 450 x 56

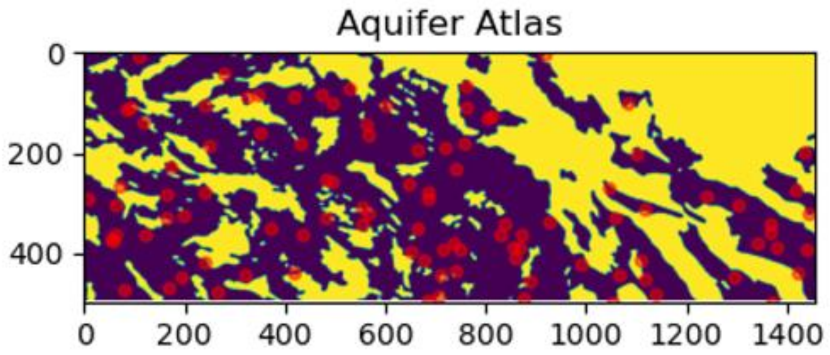


[1] Peters, Bas. Point-to-set distance functions for weakly supervised segmentation. arXiv preprints, 2021.

[1] Peters, Bas. Point-to-set distance functions for weakly supervised segmentation. arXiv preprints, 2021.

Scenario 3: Weakly supervised segmentation

- Translate partial labels + prior information
  → convex constraints



- Constraints on network output $\mathbf{y} = g(\mathbf{K}, \mathbf{d})$ (not on weights)

$\mathbf{K}$: Network weights
$\mathbf{d}$: Input data
$\mathbf{y}$: Output label

- Training: non-convex feasibility problem

$$\text{find}\, g(\mathbf{K}, \mathbf{d}) \in D \Leftrightarrow \min_{\mathbf{K}} \iota_D(g(\mathbf{K}, \mathbf{d}))$$

- Solve via projection-based point-to-set distance functions

$$d_D^2(\mathbf{y}) = \frac{1}{2}\|P_D(\mathbf{y}) - \mathbf{y}\|_2^2 \qquad \nabla_{\mathbf{y}} d_D^2(\mathbf{y}) = \mathbf{y} - P_D(\mathbf{y})$$

[1] Peters, Bas. Point-to-set distance functions for weakly supervised segmentation. arXiv preprints, 2021.

Scenario 3: Weakly supervised segmentation

- Train neural network as:

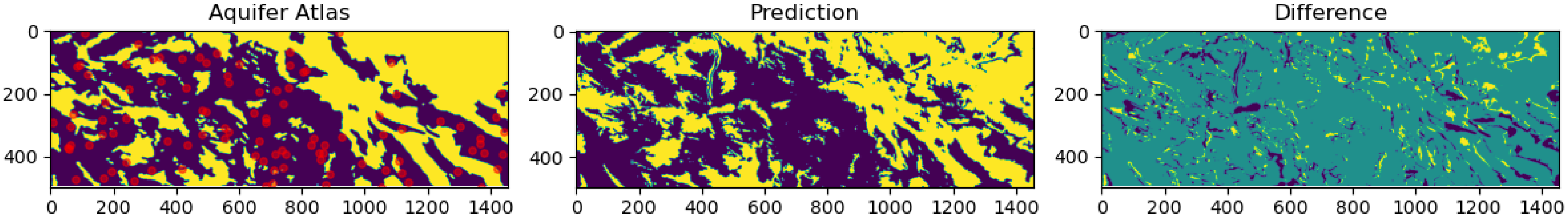
$$\min_{\mathbf{K}} \frac{1}{2}\|P_D(g(\mathbf{K},\mathbf{d})) - g(\mathbf{K},\mathbf{d})\|_2^2$$

Add INN as constraint

$$\xrightarrow{\hspace{4em}}$$

$$\begin{aligned}
&\min_{\{\mathbf{K}\}} \frac{1}{2}\|P_D(\mathbf{y}_n) - \mathbf{y}_n\|_2^2 \text{ s.t.}\\
&\mathbf{y}_n = \mathbf{y}_{n-1} - \sigma(\mathbf{K}_n\mathbf{y}_{n-1})\\
&\vdots\\
&\mathbf{y}_j = \mathbf{y}_{j-1} - \sigma(\mathbf{K}_j\mathbf{y}_{j-1})\\
&\vdots\\
&\mathbf{y}_1 = \mathbf{d},
\end{aligned}$$

- Form Lagrangian + conventional backpropagation
- Difference to previous examples using labels
  - Gradient of loss ⇨ gradient of distance function



Scenario 4: Normalizing flows & inference

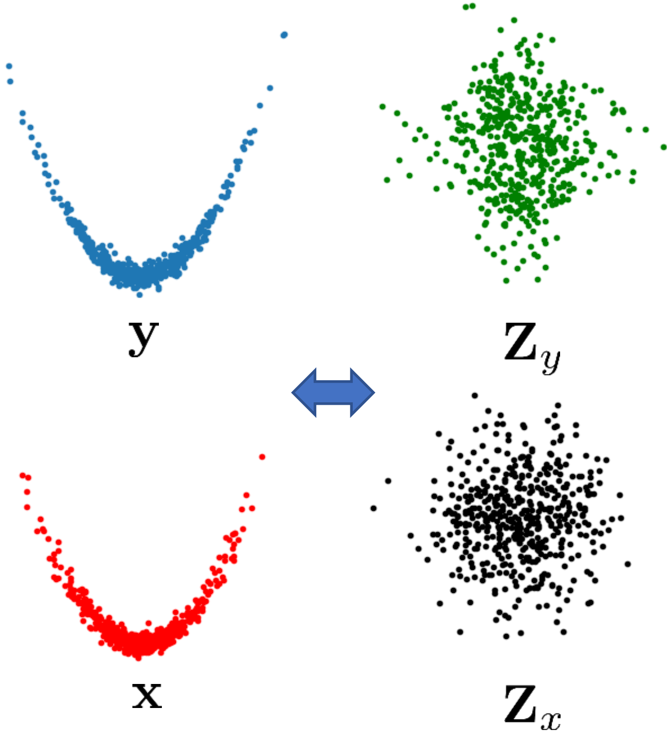
- Goal: perform Bayesian inference for data & image reconstruction [1-3]
- Train conditional INN $G_\theta : \mathcal{Y} \times \mathcal{X} \to \mathcal{Z}_y \times \mathcal{Z}_x$

$$\min_\theta \mathbb{E}_{\mathbf{y},\mathbf{x}\sim p(\mathbf{y},\mathbf{x})}\left[\frac{1}{2}\left\|G_\theta(\mathbf{y},\mathbf{x})\right\|^2 - \log\left|\det \nabla_{y,x}\, G_\theta(\mathbf{y},\mathbf{x})\right|\right]$$

$$G_\theta(\mathbf{y},\mathbf{x}) = \begin{bmatrix} G_{\theta_y}(\mathbf{y}) \\ G_{\theta_x}(\mathbf{y},\mathbf{x}) \end{bmatrix}, \quad \boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_y \\ \boldsymbol{\theta}_x \end{bmatrix}$$

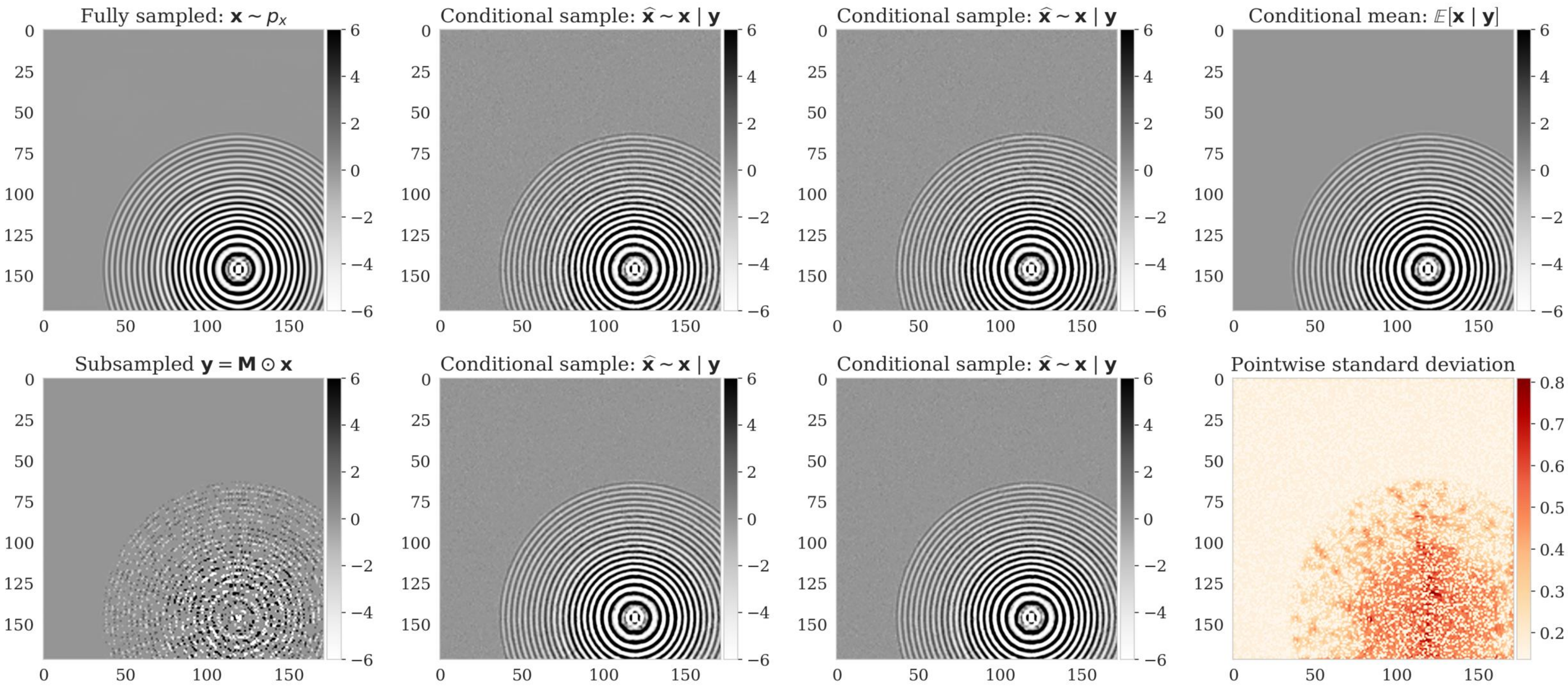
- Perform conditional sampling via

$$G_{\theta_x}^{-1}(G_{\theta_y}(\mathbf{y}),\mathbf{z}) \sim p(\mathbf{x} \mid \mathbf{y}), \quad \mathbf{z} \sim \mathrm{N}(\mathbf{0},\mathbf{I})$$



[1] Kruse et al., *HINT: Hierarchical Invertible Neural Transport for Density Estimation and Bayesian Inference*. Proceedings of AAAI, 2021
[2] Siahkoohi et al., *Preconditioned training of normalizing flows for variational inference for inverse problems*. 3rd Symposium on Advances in Approximate Bayesian Inference. 2021
[3] Rizzuti et al., Parameterizing uncertainty by deep invertible networks, an application to reservoir characterization. SEG, 2020.

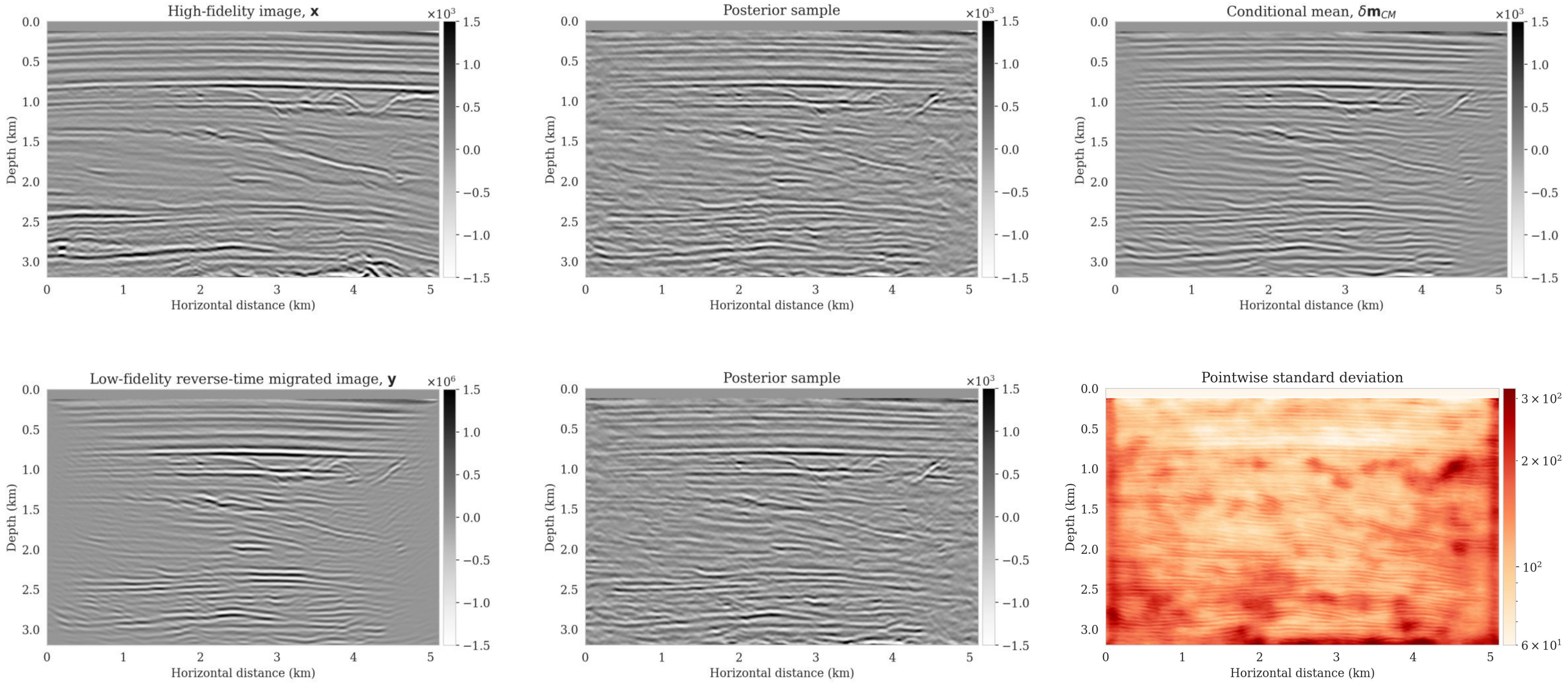
Scenario 4: Normalizing flows & inference



Seismic wavefield reconstruction

[1] Kumar et al., "Enabling uncertainty quantification for seismic data pre-processing using normalizing flows (NF) – an interpolation example. SEG, 2021

Scenario 4: Normalizing flows & inference

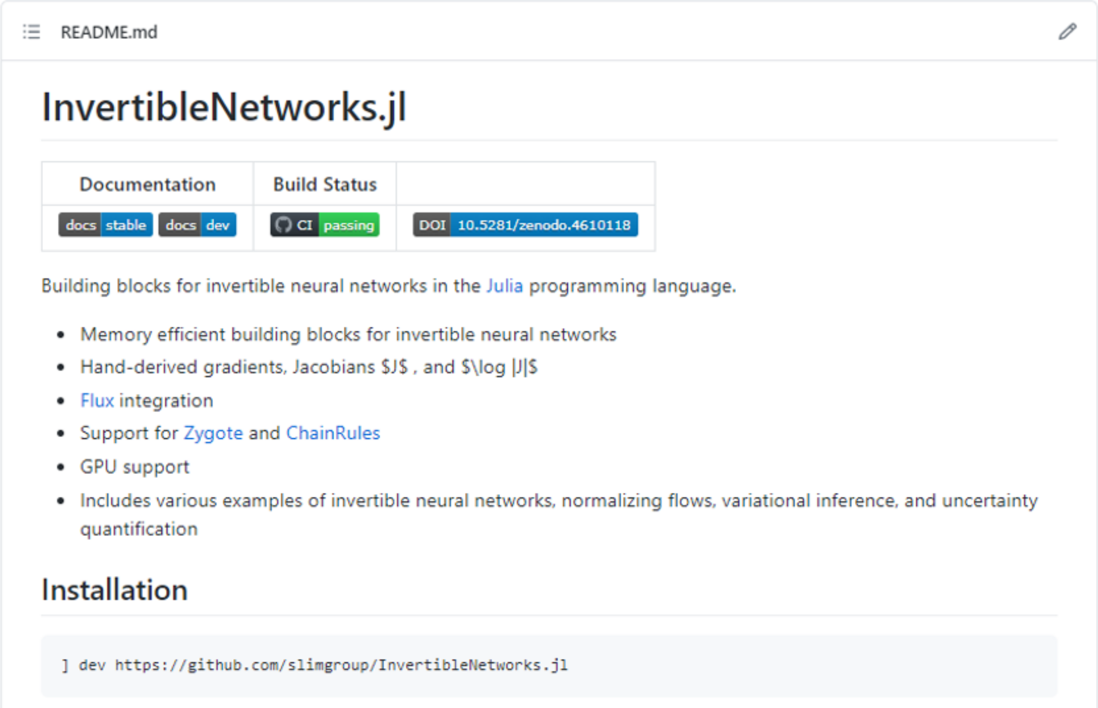


Seismic image reconstruction

[1] Siakhooi et al., Learning by example: fast reliability-aware seismic imaging with normalizing flows. SEG, 2021.

Summary & conclusions

- Invertible CNNs & Normalizing Flows in Julia language
- Memory efficient/optimal training
  - *Stateless* training
  - No extra forward evaluations
- Integration with Julia ecosystem
  - Flux.jl, ChainRules.jl, Zygote.jl
- Julia enables ML innovation
  - No comparable frameworks with PyTorch, TensorFlow, etc.



InvertibleNetworks.jl

Documentation	Build Status	
docs stable docs dev	CI passing	DOI 10.5281/zenodo.4610118

Building blocks for invertible neural networks in the Julia programming language.

- Memory efficient building blocks for invertible neural networks
- Hand-derived gradients, Jacobians J , and $\log |J|$
- Flux integration
- Support for Zygote and ChainRules
- GPU support
- Includes various examples of invertible neural networks, normalizing flows, variational inference, and uncertainty quantification

Installation

```
] dev https://github.com/slimgroup/InvertibleNetworks.jl
```

https://github.com/slimgroup/InvertibleNetworks.jl

Thank you for your attention

&

Thanks to the Georgia Research Alliance
and partners of the ML4Seismic Consortium

Georgia Tech

Seismic Laboratory for Imaging and Modeling