InvertibleNetworks.jl – Memory efficient deep learning in Julia

Philipp A. Witte\textsuperscript{1} Mathias Louboutin\textsuperscript{2} Ali Siahkoohi\textsuperscript{2} Bas Peters\textsuperscript{3} Gabrio Rizzuti and Felix J. Herrmann\textsuperscript{2}

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

Generative Modeling via Normalizing Flows

Pointwise standard deviation

Inverse problems & Uncertainty quantification (UQ)

(Kingma & Dhariwal, 2018)

(Siahkoohi et al., 2021)
Invertible Neural Networks
Invertible Neural Networks

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

INN & NF frameworks

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

• 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.

No frameworks that optimally take advantage of invertibility

Training INNs & NFs

Forward pass

\[ x \rightarrow f_1(x)^* \rightarrow h^* \rightarrow f_2(h)^* \rightarrow y^* \]

Backprop with AD

\[ \Delta x \leftarrow \partial f_1(\Delta h) \leftarrow \Delta h \leftarrow \partial f_2(\Delta y) \rightarrow \Delta y \]

* tracked
Training INNs & NFs

Forward pass

\[ x \xrightarrow{f_1(x)} f(x) \xrightarrow{h} f_2(h) \xrightarrow{y} \]

(no tracking)

Backprop layer-wise AD

\[ \bar{x}, \Delta x \xrightarrow{\bar{h}} f_1^{-1}(\bar{h}) \xrightarrow{f_1(x)^*} f_1(\bar{x}) \xrightarrow{\partial f_1(\Delta h)} f_2^{-1}(y) \xrightarrow{f_2(\bar{h})^*} f_2(\bar{h}) \xrightarrow{\partial f_2(\Delta y)} f^{-1}(y), \partial f(\Delta y) \]

* tracked
Training INNs & NFs

Forward pass

\[ x \xrightarrow{f_1(x)} f(x) \xrightarrow{h} f_2(h) \xrightarrow{y} \]

(no tracking)

Backprop no AD

\[ \bar{x}, \Delta x \xleftarrow{f_1^{-1}(\bar{h})} \partial f_1(\bar{x}, \Delta h) \xleftarrow{\bar{h}} \Delta h \xleftarrow{f_2^{-1}(y)} \partial f_2(\bar{h}, \Delta y) \xleftarrow{y, \Delta y} \]

(no tracking, no extra forward pass)
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)

InvertibleNetworks.jl

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

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

Installation

```julia
] 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

Github repository (MIT license)

https://github.com/slimgroup/InvertibleNetworks.jl
## Package overview

<table>
<thead>
<tr>
<th>Invertible layers</th>
<th>Invertible networks</th>
<th>Utilities</th>
<th>Examples</th>
</tr>
</thead>
</table>
| • 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 |

### Code Examples

```python
# 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 = vlog_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

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

Code structure of invertible layers

```haskell
# 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.

```python
# 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]

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

\[
\begin{align*}
\bar{s}, t &= \Phi(x_b) \\
\bar{s} &= \exp(\bar{s}) \\
y_a &= s \odot x_a + t \\
y &= y_b \\
\Phi(x): \text{Shallow CNN/Res-Net}
\end{align*}
\]

# 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)
```

Integration with Flux.jl

```julia
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

Integration with ChainRules.jl

• Combine INN & Flux layers via ChainRules.jl \(^{[1-2]}\)

ChainRule definition for reverse differentiation

```plaintext
# 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
```

Define multi-layer INN

```plaintext
# 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)
```
Unit tests

• Adjoint tests for linear operators:

\[ \epsilon \leq \langle Ax, y \rangle - \langle A^T y, x \rangle \]

• Gradient tests for (non-) linear layers:

\[ \Phi(x + h \cdot \Delta x) - \Phi(x) = O(h) \]
\[ \Phi(x + h \cdot \Delta x) - \Phi(x) - h \cdot \nabla \Phi(x)^T \Delta x = O(h^2) \]

\[ \Phi(w + h \cdot \Delta w) - \Phi(w) = O(h) \]
\[ \Phi(w + h \cdot \Delta w) - \Phi(w) - h \cdot \nabla \Phi(w)^T \Delta w = O(h^2) \]
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
Scenario 1: Loop-unrolled inverse problems

Image-to-image mapping
- Learned denoiser/all-at-once
- Fully data-driven

Data-to-image mapping
- Data $d = Jx$
- Fully data-driven or
- Physics-augmented/iterative (use operator $J$)

---

Scenario 1: Loop-unrolled inverse problems

Objective function for supervised learning

$$\text{minimize} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2} \| G_\theta(J_i, d_i) - \bar{x}_i \|^2_2$$

with

1. function $G(J, d)$
2. $x = 0$
3. for $j = 1, ..., n$
4. $x = Qx$
5. $x' = x_1$
6. $g = J^T(Jx'_{[1]} - d)$
7. $s', t = \text{NN}([g, x'_{[2:end]}])$
8. $s = \sigma(s')$
9. $x_2' = x_2 \odot s + t$
10. $x = Q^T x'$
11. end
12. end
13. return $x$
14. end

Results after 4 training epochs

Invertible recurrent inference machine (i-RIM)\(^{[1-2]}\)

---

[2] Putzky & Welling, Invert to learn to invert, NIPS Proceedings, 2019
Scenario 1: Loop-unrolled inverse problems

Objective function for supervised learning

\[
\text{minimize}_{\theta} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2} \| G_{\theta}(J_i, d_i) - \bar{x}_i \|^2_2
\]

with

1. function \( G(J, d) \)
2. \( x = 0 \)
3. for \( j = 1, ..., n \)
4. \( x = Q x \)
5. \( x'_i = x_1 \)
6. \( g = J\top (Jx'_i[1] - d) \)
7. \( s', t = \text{NN}([g, x'_i[2:end]]) \)
8. \( s = \sigma(s') \)
9. \( x'_2 = x_2 \odot s + t \)
10. \( x = Q\top x' \)
11. end
12. return \( x \)
13. end

Results after 4 training epochs

Invertible recurrent inference machine (i-RIM)\(^{[1-2]}\)
Scenario 1: Loop-unrolled inverse problems

Objective function for supervised learning

\[
\min_{\theta} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2} \|G_{\theta}(J_i, d_i) - \bar{x}_i\|^2
\]

with

```
1. function G(J, d)
2. x = 0
3. for j = 1, ..., n
4.   x = Qx
5.   x' = x1
6.   g = J^T(Jx'[:1] - d)
7.   s', t = NN([g, x'[:2:end]])
8.   s = \sigma(s')
9.   x0 = x2 \odot s + t
10. return x
```

Implementation

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

# Forward pass
\eta_, s_ = L.forward(\eta0, s0, d, J)

# Residual and function value
\Delta \eta = \eta_ - \eta
f = .5f0*\text{norm}(\Delta \eta)^2

# Backward pass (set gradients)
L.backward(\Delta \eta, 0f0.*s0, \eta_, 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

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*

Scenario 3: Weakly supervised segmentation

- Translate partial labels + prior information 
  → convex constraints

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

- Training: non-convex feasibility problem
  \[
  \text{find } g(K, d) \in D \iff \min_K \nu_D (g(K, d))
  \]

- Solve via projection-based point-to-set distance functions
  \[
  d_D^2(y) = \frac{1}{2} \| P_D(y) - y \|^2_2 \quad \nabla_y d_D^2(y) = y - P_D(y)
  \]

\( K \): Network weights \\
\( d \): Input data \\
\( y \): Output label
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

\[
\min_{\{\mathbf{K}\}} \frac{1}{2} \| P_D(\mathbf{y}_n) - \mathbf{y}_n \|_2^2 \text{ s.t.}
\]

\[
y_n = y_{n-1} - \sigma(\mathbf{K}_n y_{n-1})
\]

\[
\vdots
\]

\[
y_j = y_{j-1} - \sigma(\mathbf{K}_j y_{j-1})
\]

\[
\vdots
\]

\[
y_1 = \mathbf{d},
\]

• 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 \( G_\theta : \mathcal{Y} \times \mathcal{X} \to \mathcal{Z}_y \times \mathcal{Z}_x \)

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

\[
G_\theta(y, x) = \begin{bmatrix} G_{\theta_y}(y) \\ G_{\theta_x}(y, x) \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_y \\ \theta_x \end{bmatrix}
\]

- Perform conditional sampling via

\[
G_{\theta_x}^{-1}(G_{\theta_y}(y), z) \sim p(x \mid y), \quad z \sim \mathcal{N}(0, I)
\]


[2] Siahkoohi et al., Preconditioned training of normalizing flows for variational inference for inverse problems. 3rd Symposium on Advances in Approximate Bayesian Inference. 2021

Scenario 4: Normalizing flows & inference

Seismic wavefield reconstruction

Scenario 4: Normalizing flows & inference

Seismic image reconstruction

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.

https://github.com/slimgroup/InvertibleNetworks.jl
Thank you for your attention
&
Thanks to the Georgia Research Alliance and partners of the ML4Seismic Consortium