

# DeepInverse: Deep Learning for imaging inverse problems

# Introduction to DeepInverse

- DeepInverse is a PyTorch-based library for solving imaging inverse problems with DL.
- Driving principle: we don't need to reinvent the wheel!
- Avoid fragmentation in computational imaging community.
- Goal: accelerate the development of DL based methods for imaging inverse problems.

## Why Use DeepInverse?

- Combination of learning-based reconstruction approaches in a single framework.
- Standardization of forward imaging models.
- Simple creation of imaging datasets.

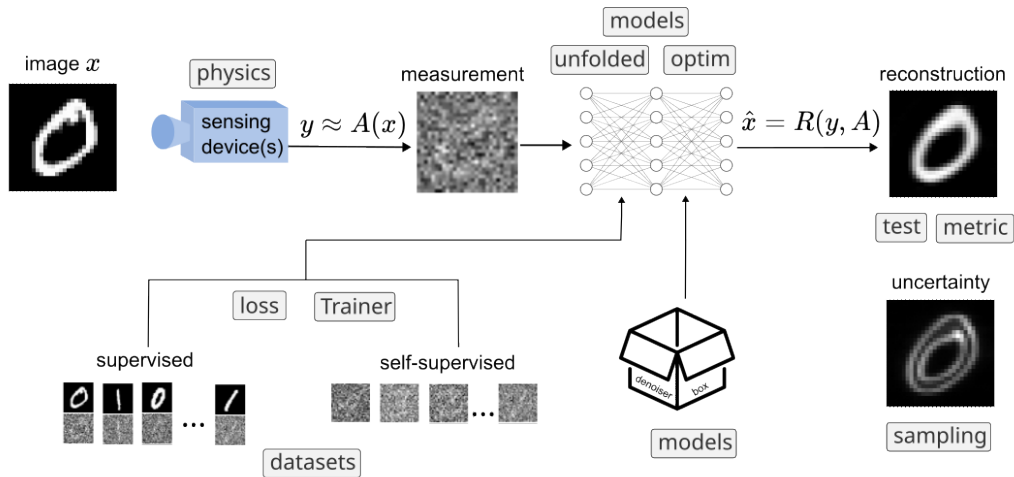


## Lead Developers

- Julian Tachella, CNRS research scientist at ENS de Lyon
- Dongdong Chen, Assistant Professor at Heriot-Watt University Edinburgh.
- Samuel Hurault, Postdoctoral researcher at ENS Paris.
- Matthieu Terris, Postdoctoral researcher Université Paris-Saclay, Inria
- Andrew Wang, PhD Student at University of Edinburgh



# Introduction to DeepInverse



# Importing images

## Import from file in Python

```
import skimage  
image = skimage.io.imread(filepath)
```

- The image is imported as a Numpy array of size (H,W,C).
- In PyTorch the images are treated as tensors of size (B,C,H,W).

## Importing images in DeepInverse

- `get_image_url` get URL for image from DeepInverse HuggingFace repository.
- `load_url_image` load an image from a URL and return a `torch.Tensor`.

## Linear acquisition model

$$b = \mathcal{N}(Ax)$$

- $\mathcal{N}$  noise model described by a noise distribution.
- $A$  forward operator.
- $b$  acquired measurement.
- $x$  real image.

In DeepInverse the acquisition model is represented and implemented within the `Physics` class.

# Denoising Physics

The forward operator for a denoising problem is  $A = I$ .

```
physics = dinv.physics.Denoising(noise_distribution)
```

where `noise_distribution` can be defined as:

```
dinv.physics.SaltPepperNoise(p=0.1, s=0.1)
```

where the pixelwise distribution is  $y = \begin{cases} 0 & \text{if } z < p \\ x & \text{if } z \in [p, 1 - s] \text{ with } z \sim \mathcal{U}(0, 1) \\ 1 & \text{if } z > 1 - s \end{cases}$

# Denoising Physics

The forward operator for a denoising problem is  $A = I$ .

```
physics = dinv.physics.Denoising(noise_distribution)
```

where `noise_distribution` can be defined as:

```
dinv.physics.GaussianNoise(sigma=0.1)
```

where the pixelwise distribution is  $y \sim \mathcal{N}(z, I\sigma^2)$

```
dinv.physics.PoissonNoise(gain=0.1)
```

where the pixelwise distribution is  $y \sim \mathcal{P}(z/\gamma)$



The `LinearPhysics` class has many attributes:

- `physics.A` is a function that applies the forward operator  $A$ .
- `physics.A_adjoint` is a function that applies the adjoint operator  $A^T$ .
- `physics.A_adjoint_A` is a function that applies the normal operator  $A^T A$ .
- `physics.A_dagger` is a function that applies the pseudoinverse  $A^\dagger$  (executing CG).
- `physics.compute_norm` computes the  $\ell_2$  spectral norm of the operator.

# Available operators

- `deepinv.physics.Denoising` Forward operator for denoising problems.
- `deepinv.physics.Inpainting` Inpainting forward operator, keeps a subset of entries.
- `deepinv.physics.Blur` Blur operator.
- `deepinv.physics.BlurFFT` FFT-based blur operator.
- `deepinv.physics.Downsampling` Downsampling operator for super-resolution problems.
- `deepinv.physics.Tomography` (Computed) Tomography operator.
- `deepinv.physics.Decolorize` Converts n-channel images to grayscale.
- `deepinv.physics.MRI` Single-coil accelerated 2D or 3D magnetic resonance imaging.

We need to quantify how close the reconstructed image  $\bar{x}$  is to the ground truth  $x$  to evaluate image reconstruction algorithms.

- MSE - Mean Squared Error
- SNR - Signal-to-Noise Ratio
- ISNR - Improved Signal-to-Noise Ratio
- PSNR – Peak Signal-to-Noise Ratio
- SSIM – Structural Similarity Index Measure

# Peak Signal-to-Noise Ratio (PSNR)

$$\text{PSNR}(x, \bar{x}) = 10 \cdot \log_{10} \left( \frac{L^2}{\text{MSE}(x, \bar{x})} \right)$$

- $L$  is maximum possible pixel value (e.g., 255 for uint8 images and 1 for float images).
- $\text{MSE}(x, \bar{x}) = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x}_i)^2$
- Higher PSNR = better quality (equivalently less error).
- Sensitive to pixel-wise differences, but not structure.

# Structural Similarity Index Measure (SSIM)

$$\text{SSIM}(x, \bar{x}) = \frac{(2\mu_x\mu_{\bar{x}} + C_1)(2\sigma_{x\bar{x}} + C_2)}{(\mu_x^2 + \mu_{\bar{x}}^2 + C_1)(\sigma_x^2 + \sigma_{\bar{x}}^2 + C_2)}$$

- $\mu_x, \mu_{\bar{x}}$ : local means.
  - $\sigma_x^2, \sigma_{\bar{x}}^2$ : local variances.
  - $\sigma_{x\bar{x}}$ : local cross-covariance.
  - $C_1, C_2$ : constants to stabilize division.
- 
- $\text{SSIM} \in [0, 1]$ ; closer to 1 is better.
  - Models luminance, contrast, and structure similarity.
  - More perceptually relevant than PSNR.

# Visual Comparison Example



Original



Increased brightness

In this example PSNR=24.21 dB, SSIM=0.975.

Even when PSNR is moderate, SSIM highlight structural fidelity.

$$x^* \in \operatorname{argmin}_x \{F(x) = f(x, A, b) + \lambda g(x)\}$$

- $f(x, A, b)$  data fidelity  $\rightarrow$  `DataFidelity` class inside `deepinv.optim`.
- $g(x)$  prior  $\rightarrow$  `Prior` class inside `deepinv.optim`.

- `dinv.optim.data_fidelity.L1()`: operator which sets  $\|Ax - b\|_1$
- `dinv.optim.data_fidelity.L2()`: operator which sets  $\frac{1}{2}\|Ax - b\|_2^2$

$$x^* \in \operatorname{argmin}_x \{F(x) = f(x, A, b) + \lambda g(x)\}$$

- $f(x, A, b)$  data fidelity  $\rightarrow$  `DataFidelity` class inside `deepinv.optim`.
  - $g(x)$  prior  $\rightarrow$  `Prior` class inside `deepinv.optim`.
- 
- `dinv.optim.Zero()`: operator which sets  $g(x) = 0 \ \forall x$
  - `dinv.optim.L1Prior()`: operator which sets  $g(x) = \|x\|_1$
  - `dinv.optim.Tikhonov()`: operator which sets  $g(x) = \frac{1}{2}\|x\|_2^2$
  - `dinv.optim.TVPrior()`: operator which sets  $g(x) = \|Dx\|_{1,2}$



# Defining a reconstruction model

```
rec_model = dinv.optim.optim_builder(  
    iteration="PGD", prior=prior, data_fidelity=data_fidelity,  
    max_iter=100, crit_conv='residual', thres_conv=1e-4,  
    early_stop=True, params_algo={"stepsize": 1.0, "lambda": 1.})  
x_hat = rec_model(y, physics)
```

- iteration the name of the algorithm to be used:
  - GD gradient descent
  - PGD proximal gradient descent
  - ADMM ADMM
  - HQS Half-Quadratic Splitting
- data\_fidelity data-fidelity term of type `deepinv.optim.DataFidelity`.
- prior regularization prior of type `deepinv.optim.Prior`.

# Defining a reconstruction model

```
rec_model = dinv.optim.optim_builder(  
    iteration="PGD", prior=prior, data_fidelity=data_fidelity,  
    max_iter=100, crit_conv='residual', thres_conv=1e-4,  
    early_stop=True, params_algo={"stepsize": 1.0, "lambda": 1.})  
x_hat = rec_model(y, physics)
```

- `max_iter` maximum number of iterations of the optimization algorithm (default 100).
- `early_stop` whether to stop the algorithm once the convergence criterion is reached.
- `thres_conv` value of the threshold  $\tau$  for claiming convergence.
- `crit_conv` convergence criterion:

- `residual`  $\frac{\|x_{k+1} - x_k\|}{\|x_k\|} < \tau$
- `cost`  $\frac{\|F(x_{k+1}) - F(x_k)\|}{\|F(x_k)\|} < \tau$

# Defining a reconstruction model

```
rec_model = dinv.optim.optim_builder(  
    iteration="PGD", prior=prior, data_fidelity=data_fidelity,  
    max_iter=100, crit_conv='residual', thres_conv=1e-4,  
    early_stop=True, params_algo={"stepsize": 1.0, "lambda": 1.})  
x_hat = rec_model(y, physics)
```

- `params_algo` a dictionary containing all the relevant parameters for running the algorithm, e.g. the stepsize, regularization parameter, denoising standard deviation.
  - `stepsize`: gradient steplength.
  - `lambda`: regularization weight.