De-convolution Project Lab Book

Contents

Learning Pytorch

Pytorch is a machine learning library and will be used in this project to build and train neural networks. It is built on top of the tensor library `torch`, which is a library for high performance numerical computing.

Aims

- Practice using and troubleshooting pytorch
- Understand the advantages of using pytorch over numpy
- Build a basic machine learning algorithm

Tensors

- A specialized data structure very similar to numpy arrays
- Similar to numpy arrays but can also run on the GPU as well as the CPU, meaning thousands of operations can be handled simultaneously. Making them a more suitable data structure for training neural networks.
- **TLDR they are like numpy arrays but with the ability to run on high performance hardware**

Basic Tensor Examples

```
import torch
import numpy as np
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import torch
      2 import numpy as np

ModuleNotFoundError: No module named 'torch'
```

```
# From 2d list
data = [[2, 2],[3, 4]]
x_data = torch.tensor(data)

# From numpy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
# Overides initial x_data with the same shape but random values

x_rand = torch.rand_like(x_data, dtype=torch.float)
print(x_rand)
```

```
tensor([[0.0394, 0.2424],
        [0.1560, 0.1370]])
```

Loading a dataset

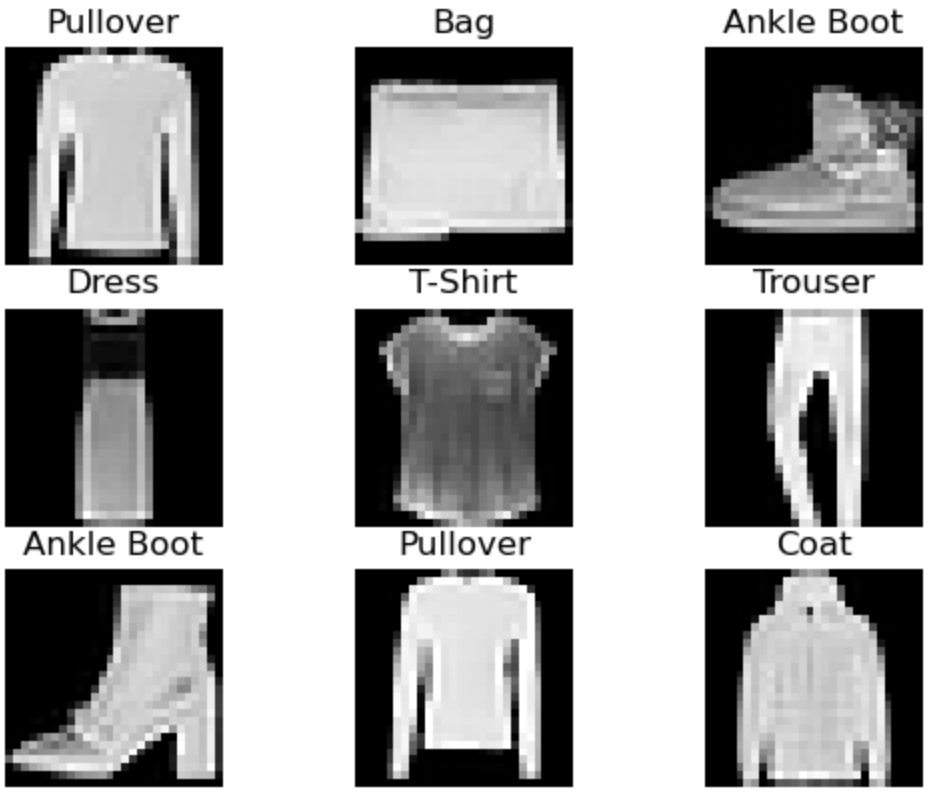
Below is an example of how to load the Fasion-MINST dataset from TorchVision. The dataset contains 60,000 training examples and 10,000 test examples. Each example comprises a 28x28 greyscale image and an associated label

```
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
```

```
training_data = datasets.FashionMNIST(
    root="data", # where the train test data is stored
    train=True, # specifies training or testing dataset
    download=True, # specifies if its local or on internet
    transform=ToTensor() # specifies the type it is transformed to
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Displaying random items from dataset

```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
} # Labels for each of the clothes in the dataset

fig, axs = plt.subplots(3, 3)
for i in range(3):
    for j in range(3):
        sample_idx = torch.randint(len(training_data), size=(1,)).item()
        img, label = training_data[sample_idx]
        axs[i][j].imshow(img.squeeze(), cmap="gray")
        axs[i][j].set_title(labels_map[label])
        axs[i][j].axis("off")
```



Example: A basic machine learning algorithm with pytorch

Firstly random input and output data is generated. Here x will represent the angle in radians and y will represent the sin of that angle, being predicted. y^- is the prediction of y. The loss function is given as: $$L = \sum_{i=1}^{n} (y_i^- - y_i)^2$$

$Minimizing the loss function means the predicted curve gets closer to the true curve. The loss with respect to each parameter. \frac{\partial L}{\partial a_t} = 2\sum_{i=1}^{n}(y_i^- - y_i)\frac{\partial L}{\partial b_t} = 2\sum_{i=1}^{n}(y_i^- - y_i)x_i$

$\frac{\partial L}{\partial c_t} = 2\sum_{i=1}^{n}(y_i^- - y_i)x_i^2 \frac{\partial L}{\partial d_t} = 2\sum_{i=1}^{n}(y_i^- - y_i)x_i^3$

$These gradients are then used to update the parameters using gradient descent.$

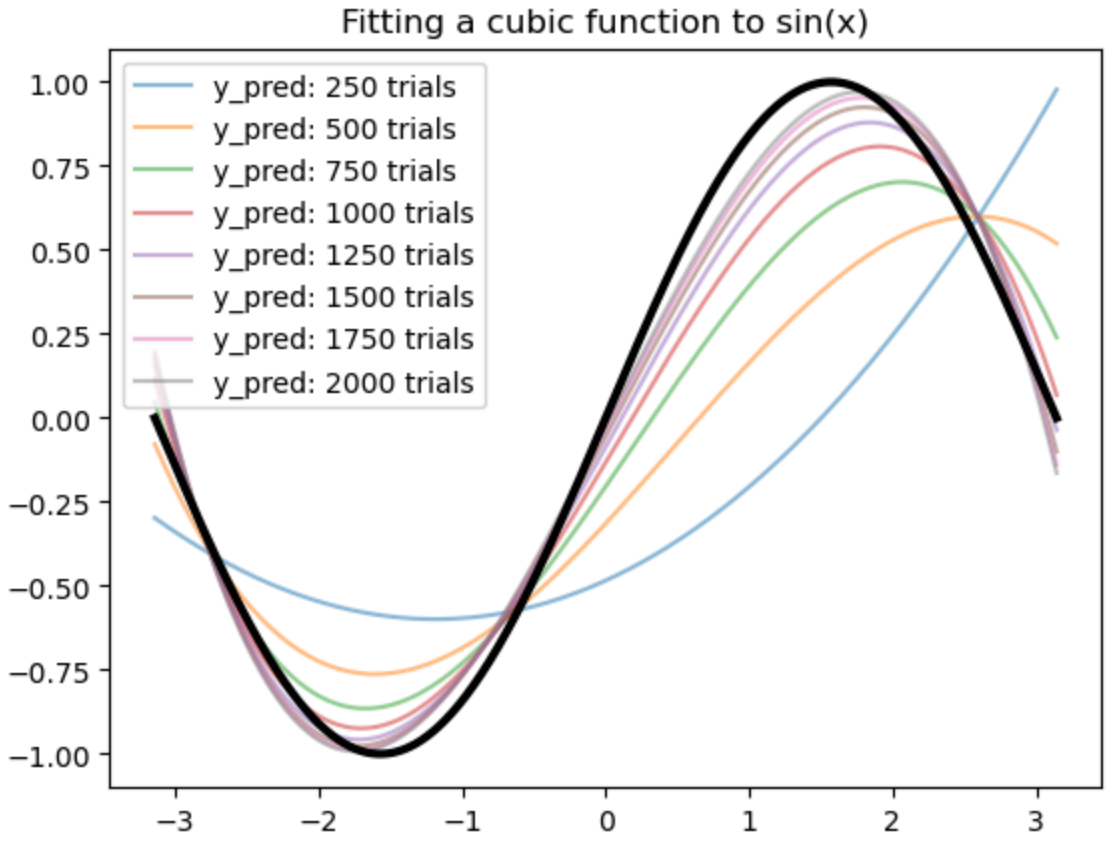
$$a_{t+1} = a_t - \eta\frac{\partial L}{\partial a_t} b_{t+1} = b_t - \eta\frac{\partial L}{\partial b_t} c_{t+1} = c_t - \eta\frac{\partial L}{\partial c_t} d_{t+1} = d_t - \eta\frac{\partial L}{\partial d_t}$$

Where η is the learning rate, a hyperparameter that controls how much to change the parameters in response to the computed gradients. *η is initially set to be low to visualize the effects of gradient descent*

```
dtype = torch.float
device = torch.device("cpu")

# Generate random input and output data
x = torch.linspace(-np.pi, np.pi, 2000, device=device, dtype=dtype)
y = torch.sin(x)

# Randomly initialise weights for polynomial
a = torch.randn((), device=device, dtype=dtype)
b = torch.randn((), device=device, dtype=dtype)
c = torch.randn((), device=device, dtype=dtype)
d = torch.randn((), device=device, dtype=dtype)

# colours for the graph

fig, ax = plt.subplots()

learning_rate = 1e-6
for t in range(2000):
    # predict y
    y_pred = a + b*x + c*x**2 + d*x**3

    # compute loss
    loss = (y_pred - y).pow(2).sum().item()

    # compute gradients with respect to a, b, c, d
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x**2).sum()
    grad_d = (grad_y_pred * x**3).sum()

    # update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

    if t % 250 == 249:
        print(t, loss)
        ax.set_title("Fitting a cubic function to sin(x)")
        ax.plot(x.cpu(), y_pred.detach().cpu(), label=f"y_pred: {t+1} trials", alpha
        ax.legend()

ax.plot(x.cpu(), y.cpu(), label="y=sin(x)", color="black", linewidth=3)
```

```
249 678.7615356445312
499 260.70501708984375
749 104.14237976074219
999 45.139129638671875
1249 22.75472640991211
1499 14.203689575195312
1749 10.913901329040527
1999 9.639151573181152
```

```
[<matplotlib.lines.Line2D at 0x74267d270690>]
```



Fourier Transforms in 2D

Fourier Analysis offers a way to analyse images in terms of their frequency components. This is important in image processing, as it can offer a better description of an image rather than to say sharp/blury or clear/noisy. Furthermore fourier transfomrs are an important tool to understand how different filters work in the frequency domain.

Aims

- To compute the 2D fourier transform of basic images
- To understand how different image features (edges, lines, gradients) are represented in the frequency domain

Theory

A 2D Fourier Transform or a discrtete Fourier Transform (DFT) transforms a 2D signal (like an image) from the spartial domain to the frequency domain. The 2D Fourier transform of an image does not have a an analytic result (the result is a table of values rather than an analytic expression as in the 1D case), it however possible to visualise the results of a 2D fourier transform. The DFT is similarly to the 1D case :

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-2\pi i \left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

and its inverse is given by:

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{2\pi i \left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

Where $f(x,y)$ is the input image, $F(u,v)$ is the fourier transform of the image, and M and N are the dimensions of the image. The variables x and y are the spatial coordinates, while u and v are the frequency coordinates.

Load image

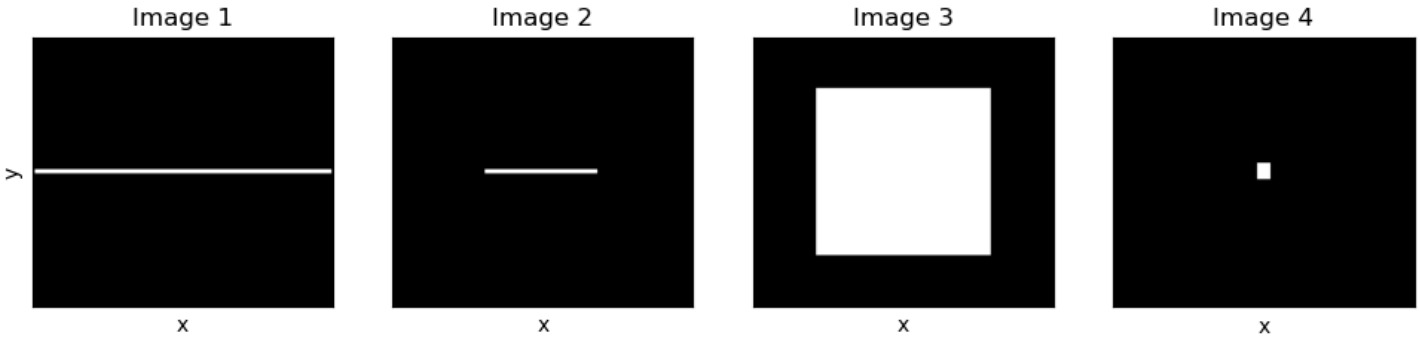
First 4 basic images were loaded and there fourier transforms were analyzed.

```
from torchvision import transforms
import torch
from PIL import Image
import matplotlib.pyplot as plt

# Load images
folder = 'Images/'
images = ['LongLine.png', 'ShortLine.png', 'BigSquare.png', 'SmallSquare.png']
img_tensors = []

for image in images:
    image_path = folder + image  # Replace with your image path
    image = Image.open(image_path)
    transform = transforms.Compose([
        transforms.Grayscale(),  # Convert to grayscale
        transforms.ToTensor()  # Resize to 64x64 for simplicity
    ])
    img_tensor = transform(image).squeeze()  # Remove channel dimension if grayscale
    img_tensors.append(img_tensor)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from torchvision import transforms
      2 import torch
      3 from PIL import Image

ModuleNotFoundError: No module named 'torchvision'
```

Displaying Images

```
# Display the images
fig, ax = plt.subplots(1, 4, figsize=(12, 3), sharex = True)
ax[0].set_ylabel('y')

for i in range(4):
    ax[i].imshow(img_tensors[i], cmap='gray')
    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].title.set_text(f'Image {i+1}')
    ax[i].set_xlabel('x')

plt.show()
```



Hypothesizing Fourier transform results

- **Image 1:** A thin horizontal line along the entire x-axis is made up from a single component with a frequency of 0. Much like a 1D fourier transform the fourier transform of this should be a bright central line along the u-axis (the x frequency axis).
- **Image 2:** A thin horizontal line along part of the x-axis is made up from a range of low frequency components. The fourier transform of this should be a bright central line along the u-axis (representing the zero frequency line) but with some width to it.
- **Image 3:** A large square in the center of the image is made up from a range of low frequency components, due to the sharp edges of the square.
- **Image 4:** A small square in the center of the image is going to be made up from higher frequency components than image 3, since the square is smaller.

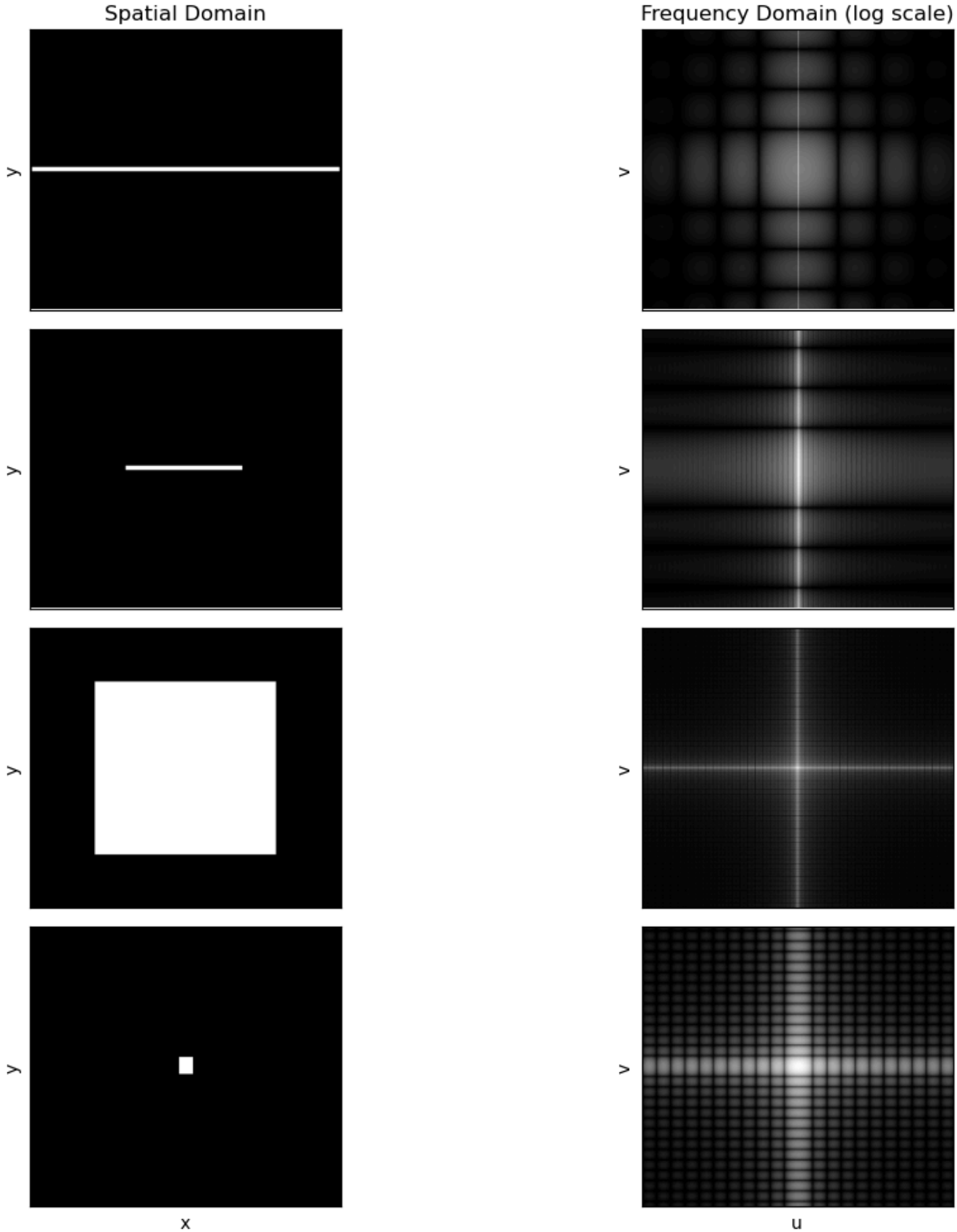
2D DFT using PyTorch

By applying the fast fourier transform (FFT) function in PyTorch, the 2D DFT of the images were calculated. The FFT function in PyTourch uses the Cooley-Tukey algorithm which has efficency $\mathbb{O}N \log N$, significantly faster than direct DFT. It is expected that the DFT of the first image…

```
img_tensor_fs = []
magnitude_spectrum_tensors = []

for img_tensor in img_tensors:
    img_tensor_f = torch.fft.fftshift(torch.fft.fft2(img_tensor))
    img_tensor_fs.append(img_tensor_f)
```

Display results

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(4, 2, sharex=True, sharey=True, figsize=(12, 10))


ylabels = ['y', 'y', 'y', 'y']
xlabels = ['', '', '', 'x']
freq_xlabels = ['', '', '', 'u']

ax[0, 0].set_title('Spatial Domain')
ax[0, 1].set_title('Frequency Domain (log scale)')

for i in range(4):
    # Spatial domain
    ax[i, 0].imshow(img_tensors[i], cmap='gray')
    ax[i, 0].set_ylabel(ylabels[i])
    ax[i, 0].set_xlabel(xlabels[i])
    ax[i, 0].set_xticks([])
    ax[i, 0].set_yticks([])

    # Frequency domain
    ax[i, 1].imshow(torch.log1p(torch.abs(img_tensor_fs[i])), cmap='gray')
    ax[i, 1].set_ylabel('v')
    ax[i, 1].set_xlabel(freq_xlabels[i])
    ax[i, 1].set_xticks([])
    ax[i, 1].set_yticks([])

fig.tight_layout()
plt.show()
```


Spatial Domain
Frequency Domain (log scale)
y
v
x
u

```
img = torch.zeros(256, 256)

center = 256//2
square_size = 20
half_square = square_size // 2

img[center-half_square:center+half_square+1,
    center-half_square:center+half_square+1] = 1.0
```

Fourier transform of real image

```
# Forier transformed pigeon
image_path = "Images/pigeon.jpeg"

pigeon = Image.open(image_path)
transform = transforms.Compose([
        transforms.Grayscale(),  # Convert to grayscale
        transforms.Resize((256, 256)), # Resize to 256x256 to speed up processing
        transforms.ToTensor()
    ])
pigeon_tensor = transform(pigeon).squeeze()  # Remove channel dimension if grayscale
box_tensor = transforms.ToTensor()(img).squeeze()
fft_img_tensor = torch.fft.fftshift(torch.fft.fft2(pigeon_tensor))

fig, ax = plt.subplots(2,1)
ax[0].axis('off')
ax[0].imshow(torch.log1p(torch.abs(fft_img_tensor)), cmap='plasma')
ax[1].axis('off')
ax[1].imshow(pigeon_tensor, cmap='gray')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 11
      5 transform = transforms.Compose([
      6         transforms.Grayscale(),  # Convert to grayscale
      7         transforms.Resize((256, 256)), # Resize to 256x256 to speed up proce
      8         transforms.ToTensor()
      9     ])
     10 pigeon_tensor = transform(pigeon).squeeze()  # Remove channel dimension if g
---> 11 box_tensor = transforms.ToTensor()(img).squeeze()
     12 fft_img_tensor = torch.fft.fftshift(torch.fft.fft2(pigeon_tensor))
     14 fig, ax = plt.subplots(2,1)

File ~/anaconda3/envs/BScProject/lib/python3.11/site-packages/torchvision/transforms
    129 def __call__(self, pic):
    130     """
    131     Args:
    132         pic (PIL Image or numpy.ndarray): Image to be converted to tensor.
   (...)    135         Tensor: Converted image.
    136     """
--> 137     return F.to_tensor(pic)

File ~/anaconda3/envs/BScProject/lib/python3.11/site-packages/torchvision/transforms
    140     _log_api_usage_once(to_tensor)
    141 if not (F_pil._is_pil_image(pic) or _is_numpy(pic)):
--> 142     raise TypeError(f"pic should be PIL Image or ndarray. Got {type(pic)}")
    144 if _is_numpy(pic) and not _is_numpy_image(pic):
    145     raise ValueError(f"pic should be 2/3 dimensional. Got {pic.ndim} dimensi

TypeError: pic should be PIL Image or ndarray. Got <class 'torch.Tensor'>
```

Results

- For the first two images there are additional low frequency components. This is because the line is not infinitely thin, and so there are some low frequency components to represent the thickness of the line in the v direction (the y frequency axis).
- DFTs mostly align with the predictions. Except for perhaps image 4 where there are more higher frequency components in the u direction than expected.

Conclusion

Basics of image de-blurring

Blur filters are low pass filters, they remove high frequency components which are usually associated with noise. Low pass filters applied to images makes them look bury. A pixel is blurred by taking the average over the surrounding pixels. Doing this for each pixel individually is slow but can be sped up using convolution.

Contents

- Image noise
- How convolution can be used to speed up image filtering?
- The process which noise and blur effect images
- Convolution and de-convolution to blur and de-blur images
- De-convolution in de-blur noisy images

```
from torchvision import transforms
import torch
from PIL import Image
import matplotlib.pyplot as plt
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from torchvision import transforms
      2 import torch
      3 from PIL import Image

ModuleNotFoundError: No module named 'torchvision'
```

Load images

```
# Load images
folder = 'Images/'
device = 'cpu'
image = 'pigeon.jpeg'

image_path = folder + image  # Replace with your image path
image = Image.open(image_path)
transform = transforms.Compose([
        transforms.Grayscale(),  # Convert to grayscale
        transforms.Resize((256, 256)), # Resize to 256x256 to speed up processing
        transforms.ToTensor()
    ])
img_tensor = transform(image).squeeze()  # Remove channel dimension

# Display the image
plt.imshow(img_tensor, cmap='gray')
plt.axis('off')
```

```
(np.float64(-0.5), np.float64(255.5), np.float64(255.5), np.float64(-0.5))
```

Shot noise (poisson noise)

Occurs when there is a finite number of particles that carry energy, such as photons in an image sensor or electrons in a circuit. Shot noise is one of the main types of noise that degrade image quality.

Shot noise follows a Poisson distribution because:

1. **Discrete countable events**: Each photon arrival is a discrete, countable event
2. **Constant Average Rate** λ: The average rate of photon arrivals depends on the light intensity (brighter = more photons per second)
3. **No simultaneous Events**: In any tiny time interval, the probability of a photon arriving is very small. In an infinitesimally small window the probability of more than one event occuring is ≈ 0
4. **Independent**: Previous photon arrivals don't affect future arrivals

These conditions exactly match the assumptions of a Poisson process. The number of photons detected in a fixed time period (Δt) follows:

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

Where:

- k = number of photons detected
- λ = average number of photons arriving in Δt (proportional to light intensity)
- Higher λ = brighter pixel = more photons = less relative noise
- Lower λ = darker pixel = fewer photons = more relative noise

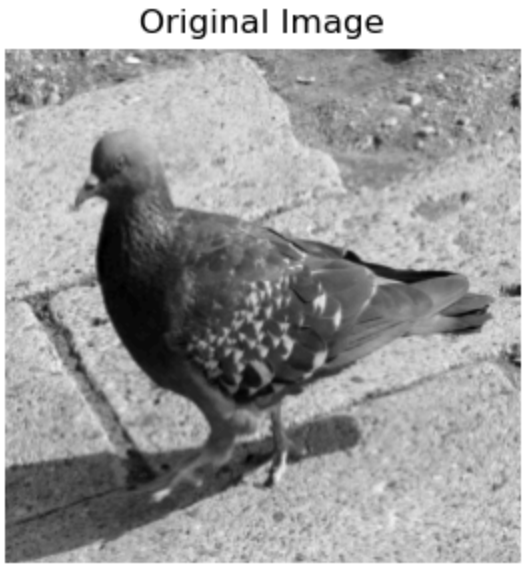
This is why images taken in low light (few photons) are much noisier than images in bright light (many photons).

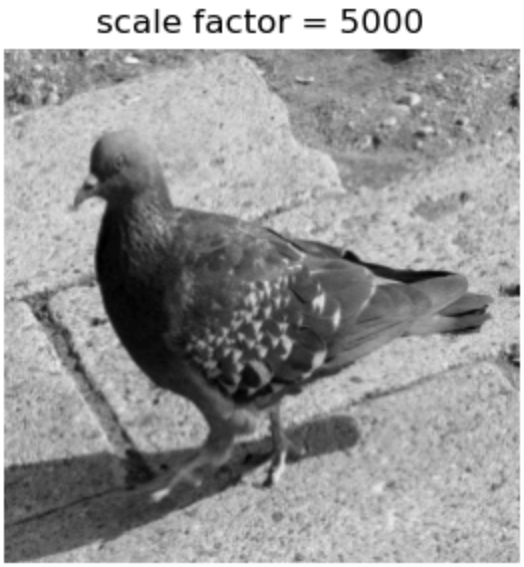
Method

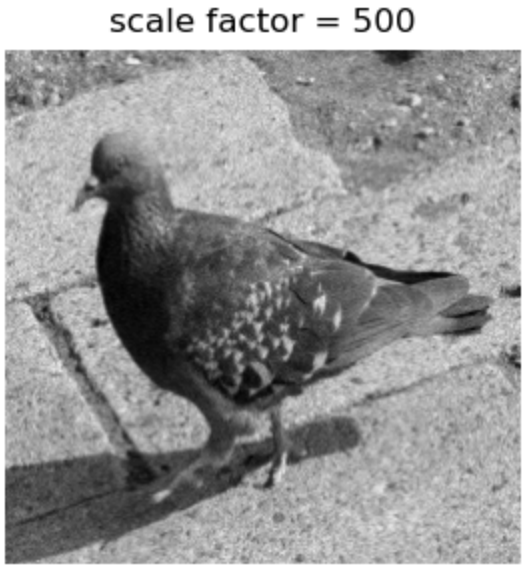
- The number of photons in each pixel was simulated depending on the brightness of the pixel and its scale factor (s).
- The photon count was set as the expected number of photons in the pixel $sP(x)$

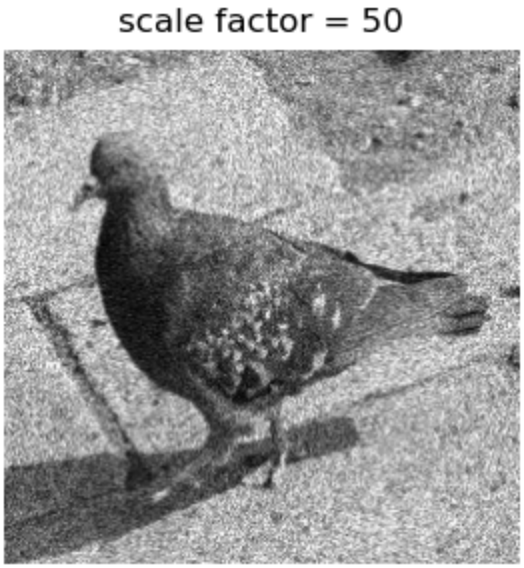
- `torch.poisson` randomly generates the number of photons present due to random variation

```python
def add_poisson_noise(image, scale_factor=1000):
    """
    Add realistic Poisson (shot) noise to an image

    Args:
        image: Input tensor in range [0,1]
        scale_factor: Higher values = less noise (more photons)
    """
    # Convert to photon counts (scale up to simulate photon detection)
    photon_counts = image * scale_factor

    # Apply Poisson noise
    noisy_photons = torch.poisson(photon_counts)

    # Convert back to [0,1] range
    noisy_image = noisy_photons / scale_factor

    return torch.clamp(noisy_image, 0, 1)

# Create noisy images with different noise levels
torch.manual_seed(42)

# Low noise (high photon count)
noisy_low = add_poisson_noise(img_tensor, scale_factor=5000)

# Medium noise
noisy_medium = add_poisson_noise(img_tensor, scale_factor=500)

# High noise (low photon count - like low light)
noisy_high = add_poisson_noise(img_tensor, scale_factor=50)

# Display results
fig, ax = plt.subplots(2, 2, figsize=(8, 6))
ax[0,0].imshow(img_tensor, cmap='gray')
ax[0,0].set_title('Original Image')
ax[0,0].axis('off')

ax[0,1].imshow(noisy_low, cmap='gray')
ax[0,1].set_title('scale factor = 5000')
ax[0,1].axis('off')

ax[1,0].imshow(noisy_medium, cmap='gray')
ax[1,0].set_title('scale factor = 500')
ax[1,0].axis('off')

ax[1,1].imshow(noisy_high, cmap='gray')
ax[1,1].set_title('scale factor = 50')
ax[1,1].axis('off')

plt.tight_layout()
plt.show()
```









Results

- The smaller the scale factor (number of photons) the noisier the image, which in reality is what happens.
- A key property of a poisson distribution is that the mean and variance are both λ. Increasing s increases λ and therefore should increase the absolute noise. The relative noise decreases, this is clear from looking at the signal to noise ratio R $$R = \frac{\text{noise}}{\text{signal}} = \frac{\sqrt{\lambda}}{\lambda} = \frac{1}{\sqrt{\lambda}}$$
- This explains why higher photon counts produce clearer images.

Filtering in 2D

Filtering in 2D works by acting a discrete function q (dimensions X, M) on an underlying image p: $ $p'(x,y) = \sum_{n=-X}^{X} \sum_{m=-Y}^{Y} p(x+n, y+m) \cdot q(n,m)$$

Filtering Example: The box filter

The box filter is a type of low pass filter which helps to remove noise and blurs images. It works by setting the pixel value $p_{x,y}$ to be the average of the N pixels above it and M pixels below it.

$$p'(x,y) = \frac{1}{(2N+1)(2M+1)} \sum_{n=-N}^{N} \sum_{m=-M}^{M} p(x+n, y+m) \tag{1}$$

This algorithm is incredibly slow when implemented with nested loops since computing each pixel value has an efficiency of $\mathbb{O}(N^2)$ which for the overall image will have efficiency $\mathbb{O}(N^4)$, since each pixel needs to be filtered in the x and y direction. This can be rapidly increased using convolution.

Convolution

The convolution of two functions is given by the multiplication of the fourier transforms. The convolution theorem states that the convolution of two functions $v(x)$ and $u(x)$ can be given in terms of their fourier transforms $V(k)$ and $U(k)$ respectively:

$$f(x) = v(x) * u(x) = \mathcal{F}^{-1}\{V(k) \cdot U(k)\} \tag{2}$$

Where:

- $*$ denotes convolution
- $\mathcal{F}^{-1}$ is the inverse Fourier transform
- $\cdot$ denotes pointwise multiplication Convolution in spatial domain is the same as multiplication in frequency domain. The convolution method can greatly improve the efficiency of applying filters especially when using the fast fourier transform. The nested loop approach has efficiency $\mathbb{O}(N^4)$ ($\mathbb{O}(N^2)$ per pixel) where as transfering to the frequency domain increases the efficency to $\mathbb{O}(N^2)\log(N)$ ($\mathbb{O}(\log N)$ per pixel). Furthermore there are advantages to using the better optimized pytorch tensors compared to using inefficient python loops. Convolution corresponds

to numerically represents multiplication of every value in the list $v(x)$ with every value in the list $u(x)$.

The box filter

The box function is the “convolution kernel” for this type of filter, which describes which surrounding pixel values will be used to form the filtered pixel value $p'_{x,y}$. It is described as:

$$\text{box}_{N,M}(n,m) = \begin{cases} \frac{1}{(2N+1)(2M+1)} & \text{if } -N \leq n \leq N \text{ and } -M \leq m \leq M \\ 0 & \text{else} \end{cases}$$

e.g.

$$\text{box}_{2,2}(n,m) =$$

To blur the image the the box function is applied to the image using convolution.

$$p'(x,y) = p(x,y) * \text{box}_{N,M}(x,y)$$

or generally $$p'(x,y) = p(x,y) * q(x,y)$$

Creating box functions

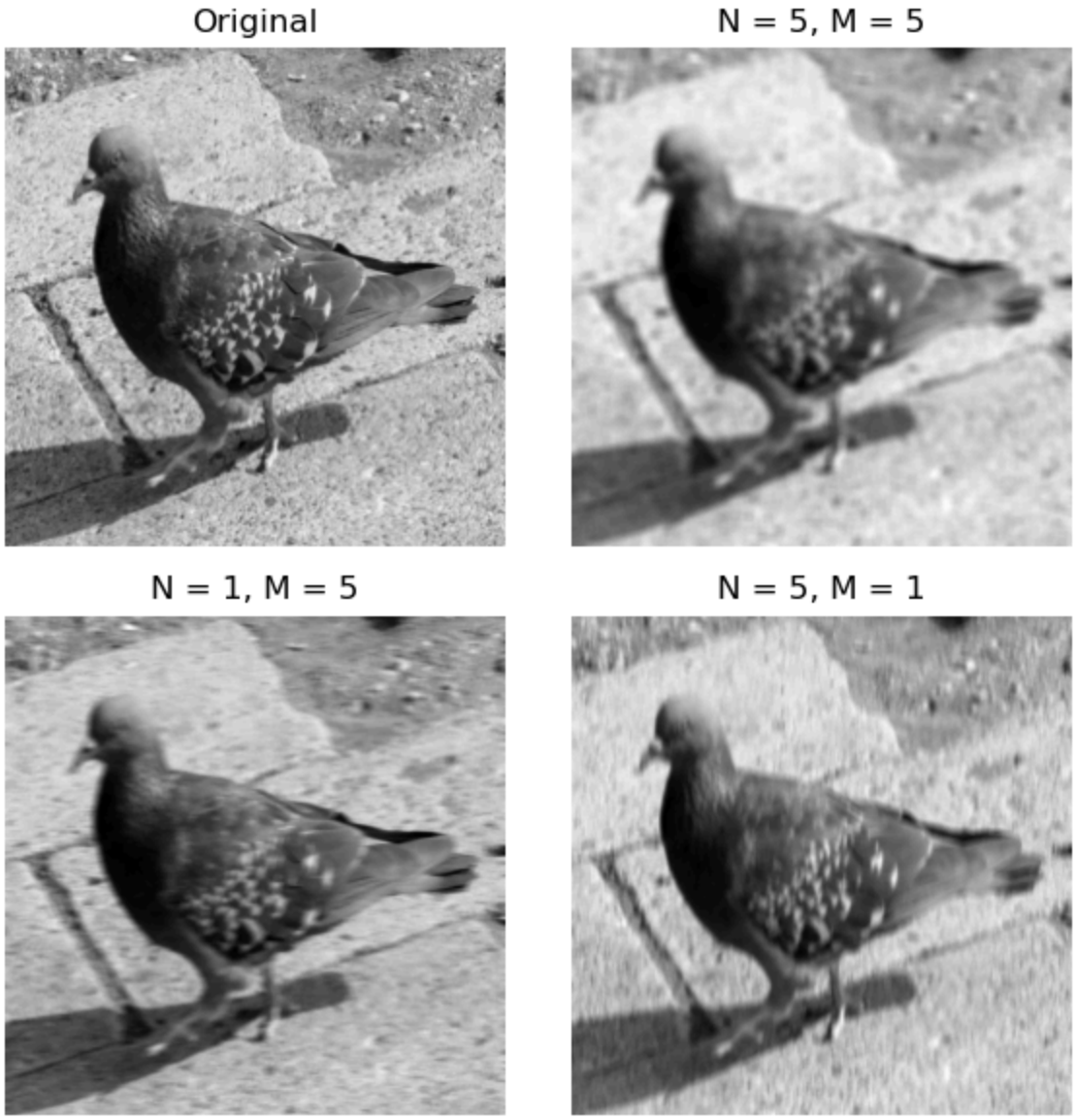
Different shapes of box functions were created and there bluing effects were analyzed.

```
def box_function(N, M):
    kernel = torch.ones(1, 1, N, M) / (N * M)
    return kernel

val = 5
square_kernel = box_function(val, val)
horizontal_kernel = box_function(1, val)
vertical_kernel = box_function(val, 1)
```

Carry out convolution

$$p'(x,y) = p(x,y) * \text{box}_{N,M}(x,y)$$

```
import torch.nn.functional as F

def convolution(image, kernel, padding_mode='zeros'):
    # Apply convolution with padding to maintain image size
    padding = (kernel.size(2) // 2, kernel.size(3) // 2)
    if padding_mode == 'zeros':
        filtered = F.conv2d(image.unsqueeze(0).unsqueeze(0), kernel, padding=paddin
    elif padding_mode == 'circular':
        image_padded = torch.cat([image[:, -padding[0]:], image, image[:, :padding[
        image_padded = torch.cat([image_padded[-padding[1]:, :], image_padded, imag
        filtered = F.conv2d(image_padded.unsqueeze(0).unsqueeze(0), kernel, padding=
    elif padding_mode == 'reflect':
        image_padded = F.pad(image.unsqueeze(0).unsqueeze(0), (padding[1], padding[
        filtered = F.conv2d(image_padded, kernel, padding=0)
    elif padding_mode == 'replicate':
        image_padded = F.pad(image.unsqueeze(0).unsqueeze(0), (padding[1], padding[
        filtered = F.conv2d(image_padded, kernel, padding=0)
    elif padding_mode == None:
        filtered = F.conv2d(image.unsqueeze(0).unsqueeze(0), kernel)
    else:
        raise ValueError(f"Unsupported padding mode: {padding_mode}")

    # Remove batch and channel dimensions: [1, 1, H, W] -> [H, W]
    return filtered.squeeze(0).squeeze(0)

fig, ax = plt.subplots(2, 2, figsize=(6, 6))
ax = ax.flatten()
ax[0].imshow(img_tensor, cmap='gray')
ax[0].set_title('Original')
ax[0].axis('off')

for i, kernel in enumerate([square_kernel, horizontal_kernel, vertical_kernel]):
    filtered_img = convolution(img_tensor, kernel, padding_mode='replicate')
    ax[i+1].imshow(filtered_img, cmap='gray')
    titles = [f'N = {val}, M = {val}', f'N = 1, M = {val}', f'N = {val}, M = 1']
    ax[i+1].set_title(titles[i])
    ax[i+1].axis('off')
fig.tight_layout()
```



Discussion

- The results show that the square box filter causes equal bluing in both directions.
- The horizontal box filter causes bluing in only the horizontal direction.
- Similarly for the vertical box filter.

De-convolution

The images can be de-blurred via de-convolution using the convolution kernel. De convolution is working out $v(x)$ knowing $f(x)$ and $u(x)$ given $f(x) = v(x) * u(x)$. The inverse (de-convolution) is: $$v(x) = \mathcal{F}^{-1} \left\{ \frac{F(k)}{H(k)} \right\}$$

$Since the convolution kernel for the blured image is known the original image can be recover p(x,y) = \mathcal{F}^{-1} \left\{ \frac{P'(u,v)}{B(u,v)} \right\}$$

Where:

- $B(k) = \mathcal{F}\{h(x)\}$ is the Fourier transform of the blur kernel in this case the fourier transform of the box function
- $P'(u,v) = \mathcal{F}\{p'(x,y)\}$ is the Fourier transform of the blurred image

Convolving and then de-convolving the image

- The image is first blurred using convolution with the box function
- The blurred image is then de-blurred using de-convolution with the box function

```python
def deconvolution(blurred_image, kernel, epsilon=1e-5):
    # Compute Fourier transforms
    P_blurred = torch.fft.fft2(blurred_image)
    B = torch.fft.fft2(kernel.squeeze(), s=blurred_image.shape)

    # Avoid division by zero by adding a small constant (epsilon)
    B_conj = torch.conj(B)
    B_magnitude_squared = torch.abs(B)**2
    B_inv = B_conj / (B_magnitude_squared + epsilon)

    # Perform deconvolution in the frequency domain
    P_deblurred = P_blurred * B_inv

    # Inverse Fourier transform to get the deblurred image
    deblurred_image = torch.fft.ifft2(P_deblurred).real

    return torch.clamp(deblurred_image, 0, 1)

square_kernel = box_function(5, 5)
square_blured = convolution(img_tensor, square_kernel, padding_mode='replicate')
deblurred_img = deconvolution(square_blured, square_kernel, epsilon=1e-3)
fig, ax = plt.subplots(3, 1, figsize=(3, 9))
ax[0].imshow(img_tensor, cmap='gray')
ax[0].set_title('Original Image')
ax[0].axis('off')

ax[1].imshow(square_blured, cmap='gray')
ax[1].set_title('Blurred Image')
ax[1].axis('off')

ax[2].imshow(deblurred_img, cmap='gray')
ax[2].set_title('Deblurred Image (Square Kernel)')
ax[2].axis('off')

fig.tight_layout()
```

Original Image

Blurred Image

Deblurred Image (Square Kernel)

Discussion

- The deconvolution process is not perfect. Artifacts are introduced in the image, especially around the edges potentially due to boundary effects and noise amplification.
- Furthermore, the box filter is not a perfect low-pass filter. While it should attenuate higher frequencies, this doesn't always work due to its frequency response characteristics.
- **Counter-example**: A high frequency alternating signal $p(x) = [\dots, 1, -1, 1, -1, 1, \dots]$ convolved with a 1×1 box filter: $$p'(x) = p(x) * \text{box}_1(x)$$ The high frequency signal passes through unchanged.
- **Example of lower frequency attenuation**: A lower frequency signal $p(x) = [\dots, 1, -0.5, -0.5, 1, -0.5, -0.5, \dots]$ when convolved with box_1 gives $\mathbb{0}$, showing selective frequency attenuation.
- Box filters give each pixel within the "box" equal weighting. In reality pixels closer to the middle will have larger weighting
- Convolving a box filter on a box filter does not give a box filter. This is a problem since applying a box filter twice isn't doubling the amount of "blur".
- Division by small values in $H(k)$ can amplify noise and introduce artifacts. Especially as the box filter may attenuate certain frequencies to near zero.

Padding

Padding is likely the cause of some of the artifacts seen in the de-blurred images. When convolving near the edges of an image, there are not enough surrounding pixels to apply the filter properly. Padding adds extra pixels around the border of the image to mitigate this issue. In this case

Gaussian filters

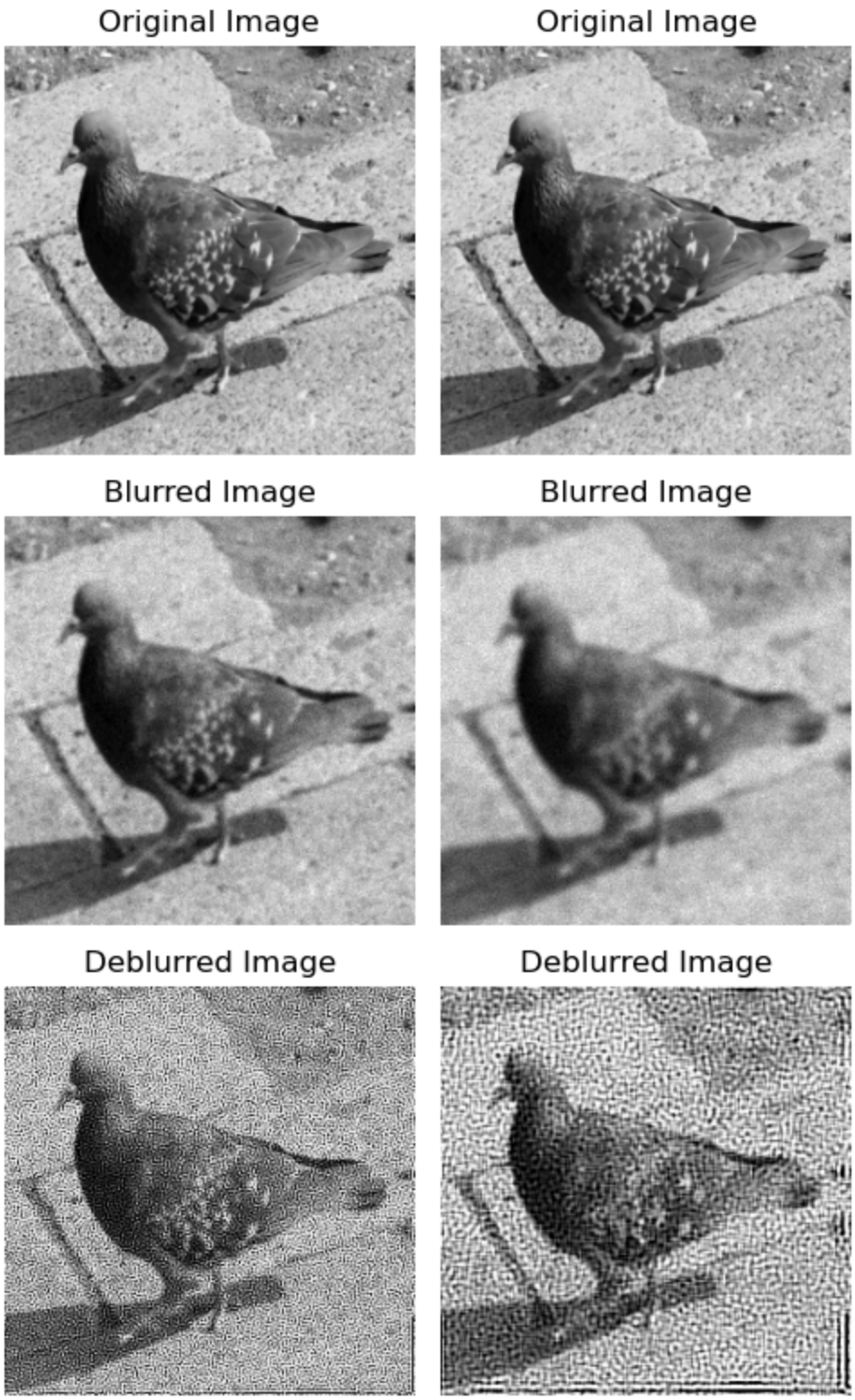
The gaussian filter is defined in 1 dimension as: $$g_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} or in 2 dimensions as: g_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} Where \sigma is the standard deviation which controls the width of the gaussian. The larger the \sigma$$ the wider the gaussian and therefore the more blurring that occurs. The gaussian filter is a better low-pass filter than the box filter since it gives more weighting to pixels closer to the center.

Image Degrading

images become degraded by convolving the image with a guassian kernel then adding noise. $
$d(x,y) = p(x,y) * q(x,y) + n(x,y)$
$then an attempt to recover the original image was made using$
$p(x,y) = d(x,y)/q(x,y) = \mathcal{F}^{-1}\left\{\frac{D(x,y)}{Q(x,y)}\right\}$$

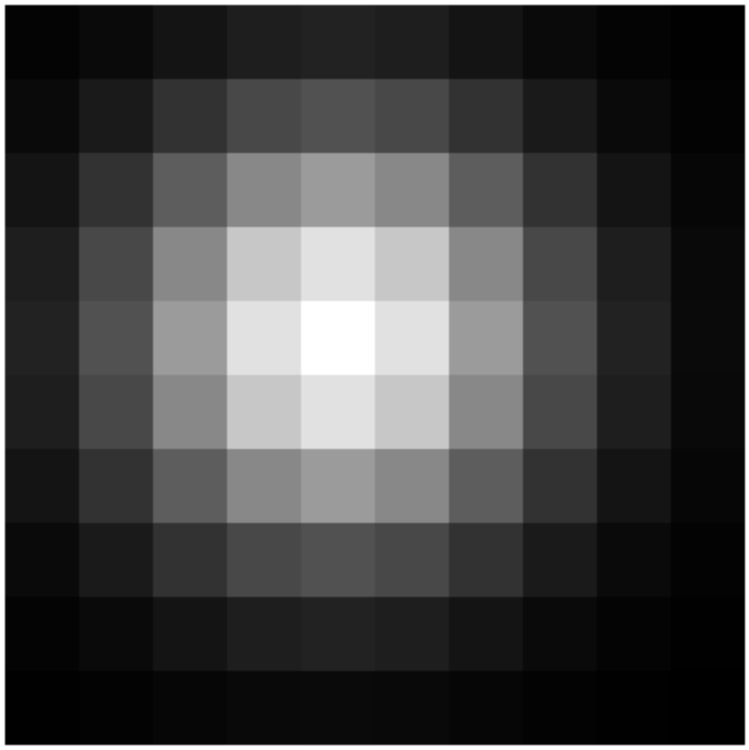
```
def gaussian_normalised_kernel(size, sigma):
    """Generate a 2D Gaussian kernel."""
    ax = torch.arange(-size // 2 + 1., size // 2 + 1.)
    xx, yy = torch.meshgrid(ax, ax, indexing='ij')
    kernel = torch.exp(-(xx**2 + yy**2) / (2. * sigma**2))
    kernel = kernel / torch.sum(kernel)
    return kernel.unsqueeze(0).unsqueeze(0)  # Shape: [1, 1, size, size]

gaussian_kernel1 = gaussian_normalised_kernel(size=10, sigma=2.0)
gaussian_blured1 = convolution(img_tensor, gaussian_kernel1, padding_mode = "replica
degraded_gaussian_blured1 = add_poisson_noise(gaussian_blured1, scale_factor=500)
deblurred_gaussian1 = deconvolution(degraded_gaussian_blured1, gaussian_kernel1, eps

gaussian_kernel2 = gaussian_normalised_kernel(size=5, sigma=1.0)
gaussian_blured2 = convolution(img_tensor, gaussian_kernel2, padding_mode = "replica
degraded_gaussian_blured2 = add_poisson_noise(gaussian_blured2, scale_factor=500)
deblurred_gaussian2 = deconvolution(degraded_gaussian_blured2, gaussian_kernel2, eps
```

```
fig, ax = plt.subplots(3, 2, figsize=(5, 8))
ax[0, 1].imshow(img_tensor, cmap='gray')
ax[0, 1].set_title('Original Image')
ax[0, 1].axis('off')

ax[1, 1].imshow(degraded_gaussian_blured1, cmap='gray')
ax[1, 1].set_title('Blurred Image')
ax[1, 1].axis('off')

ax[2, 1].imshow(deblurred_gaussian1, cmap='gray')
ax[2, 1].set_title('Deblurred Image')
ax[2, 1].axis('off')

ax[0, 0].imshow(img_tensor, cmap='gray')
ax[0, 0].set_title('Original Image')
ax[0, 0].axis('off')

ax[1, 0].imshow(degraded_gaussian_blured2, cmap='gray')
ax[1, 0].set_title('Blurred Image')
ax[1, 0].axis('off')

ax[2, 0].imshow(deblurred_gaussian2, cmap='gray')
ax[2, 0].set_title('Deblurred Image')
ax[2, 0].axis('off')
fig.tight_layout()
```




```
fig, ax = plt.subplots()
ax.imshow(gaussian_kernel1.squeeze(), cmap='gray')
ax.axis('off')
```

```
(np.float64(-0.5), np.float64(9.5), np.float64(9.5), np.float64(-0.5))
```



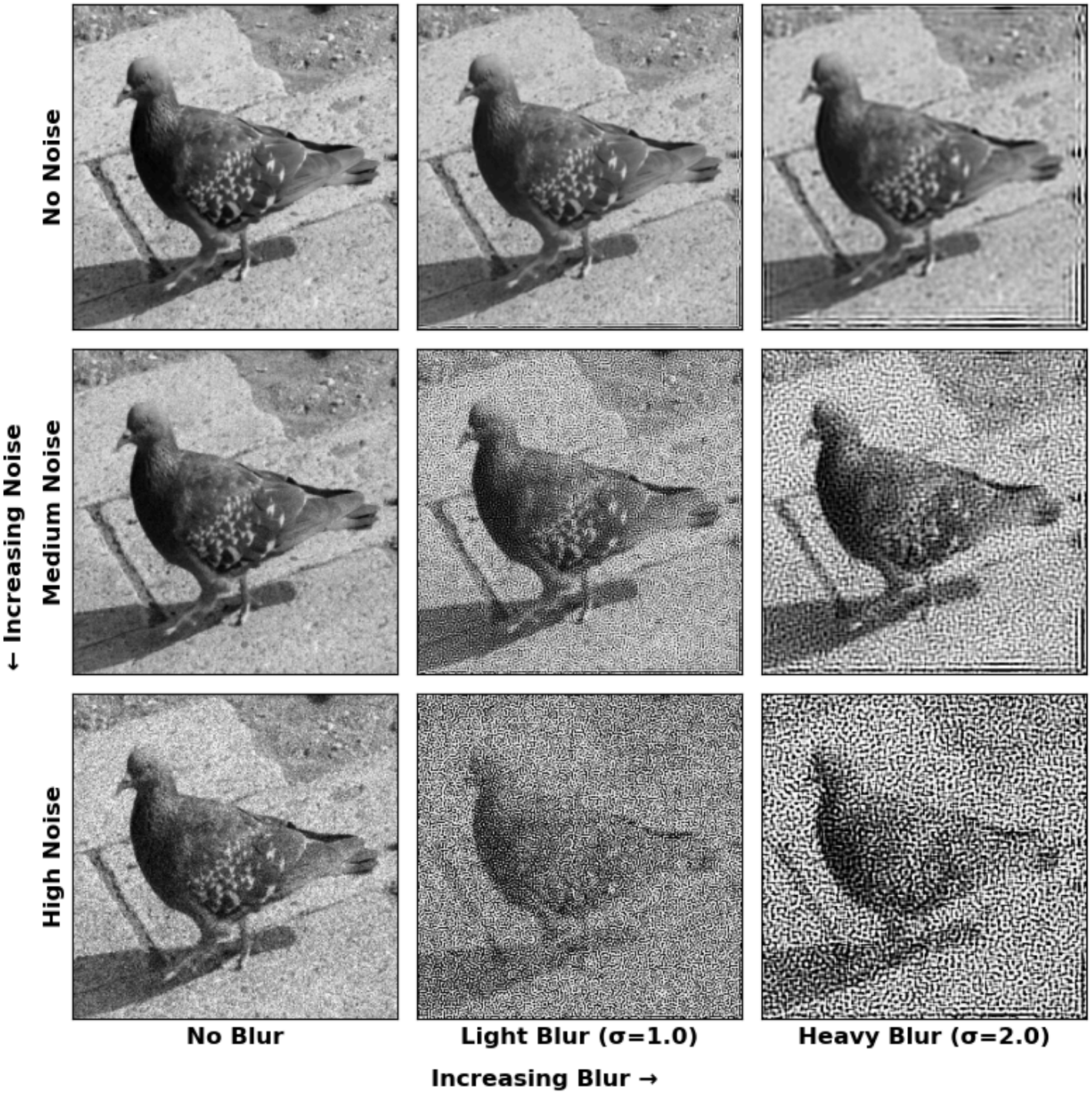
Discussion

- The de-bluring from the gaussian filters is significantly better than the box filter, especially for the heavily blured image.
- There are some edge artifacts present in the de-blured images, but not as bad as the box filter.
- This seems to be related to kernel size since larger kernels produce more edge artifacts
- This could be because the code assumes that the image is infinately periodic.

Realistic noise

Realistically noise and blur occur together. This time an image will be blured using the gaussian filter and then poisson noise will be added, to simulate realistic image acquisition conditions. The image will then be de-blured, using the convolution kernel.

```
# Using your existing variables

gaussian_kernel3 = gaussian_normalised_kernel(size=5, sigma=0.5)
no_blur_kernel = torch.tensor([[[[1.0]]]])  # Identity kernel for no blur

blur_levels =  [(no_blur_kernel, 'No Blur'),
                (gaussian_kernel2, 'Light Blur (σ=1.0)'),
                (gaussian_kernel1, 'Heavy Blur (σ=2.0)')]
noise_levels = [(50000000, 'No Noise'), (500, 'Medium Noise'), (50, 'High Noise')]

fig, axes = plt.subplots(len(noise_levels), len(blur_levels), figsize=(8, 8))

for i, (noise_scale, noise_label) in enumerate(noise_levels):
    for j, (kernel, blur_label) in enumerate(blur_levels):
        # Blur then add noise
        blurred = convolution(img_tensor, kernel, padding_mode='replicate')
        noisy_blurred = add_poisson_noise(blurred, scale_factor=noise_scale)
        if kernel.equal(no_blur_kernel):
            deblured = noisy_blurred
        else:
            deblured = deconvolution(noisy_blurred, kernel, epsilon=1e-3)
        axes[i, j].imshow(deblured, cmap='gray')
        axes[i, j].set_xticks([])
        axes[i, j].set_yticks([])

        if i == 2:
            axes[i, j].set_xlabel(blur_label, fontsize=12, fontweight='bold')
        if j == 0:
            axes[i, j].set_ylabel(noise_label, fontsize=12, fontweight='bold')


fig.text(0.5, 0.02, 'Increasing Blur →', ha='center', fontsize=12, fontweight='bold
fig.text(0.02, 0.5, '← Increasing Noise', va='center', rotation=90, fontsize=12, fo

fig.tight_layout()
plt.subplots_adjust(bottom=0.08, left=0.08)
fig.savefig('deblurring_results.png', dpi=500)
```



Discussion

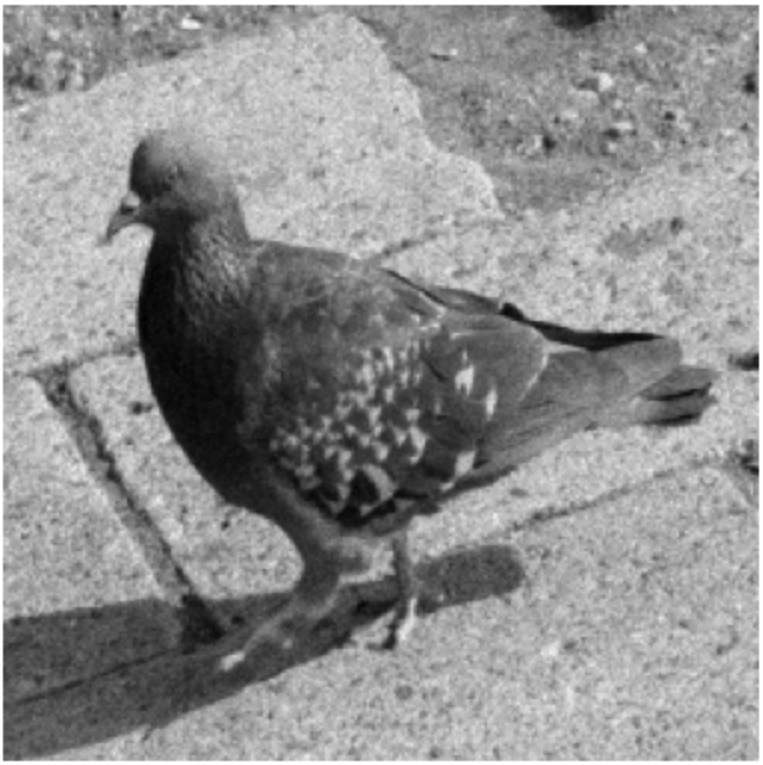
- Blur on its own can be successfully removed using de convolution.
- However when noise is added the de convolution process generates artifacts which makes it even more difficult to recover the original image.
- After noise present after the de convolution process is no longer random but seems to have more structure than the original noise since there are more distinct patches of black and white.

- This noise due to noise and blur is more visually disturbing than noise or blur.

Summary

De convolution was used to de blur images which had been convolved with a known kernel. This worked very well for images which had been blurred but not corrupted with noise. But realistically acquired images will be blurred and will have noise on top of this. De convoluting the noisy images did not work well and created artifacts in the image, which was visually more disturbing than gaussian noise.

```
noisy_img_tensor = add_poisson_noise(img_tensor, scale_factor=500)
blurred_noisy_img = convolution(noisy_img_tensor, box_function(5,5), padding_mode='
fig, ax = plt.subplots(2, 1, figsize=(5, 8))
ax[0].imshow(noisy_img_tensor, cmap='gray')

ax[0].axis('off')
ax[1].imshow(blurred_noisy_img, cmap='gray')

ax[1].axis('off')

fig.tight_layout()
fig.savefig('noisy_blurred_image.png', dpi=500)
```



Richardson Lucy Algorithm

Even knowing convolution kernel isn’t enough to de-blur an image. This is because any noise present in the image is amplified during de-convolution and adds significant visual disturbances. This section

looks at the Richardson Lucy algorithm which is an iterative method for recovering convoluted images which are also corrupted by noise.

Aims

- Understand the Richardson Lucy algorithm
- Implement the Richardson Lucy algorithm and deblur images
- Compare the Richardson Lucy algorithm with just using the Fourier transform to deconvolve a noisy image.

Theory

The matrices f, f' and k describe the proportion of light in: the true image, the observed image and the convolution kernel respectively, and are related by either $f'_i = \sum_j k_{i,j} f_j$ or $f' = f * k$. When the letters are unscripted (e.g. f) that means they are referring to the whole array. When they are scripted (e.g. f_i) then they are referring to the value in the i th location.

The Richardson Lucy algorithm is an iterative process which predicts predicts the true image accounting for noise: $$\hat{f}_j^{t+1} = \hat{f}_j^t \sum_i \frac{f_i'^t}{\hat{f'}_i^t} k_{i,j} \ [1]$$

- Where $\hat{f}_j^{t+1}$ is the $t+1$ th estimated value of f_j.
- $\hat{f'}_i$ is the predicted light intensity in the observed image given as $\hat{f'}_i = \sum_j k_{i,j} \hat{f}_j$.
- $\frac{f_i'^t}{\hat{f}_i^t}$ is the correcting factor. It determines the difference between the measured light intensity and the prediction of the light intensity from the previous estimate.
- If $p_{i,j}$ is normalized ($\sum_j p_{i,j} = 1$)
- The richardson lucy algorithm often converges as the process is self correcting [1].

Using convolution the algorithm can be written for the 2D case: $$\hat{p}^{t+1} = \hat{p}^t \cdot \left(\frac{d^t}{\hat{d}^t} * q^* \right)$$

- Where $\hat{p}^t$ is the t th estimated value of p.
- Where q^* is the mirrored convolution kernel.
- $\hat{d}$ is the predicted light intensity in the observed image given as: $\hat{d}^t = \hat{p}^t * q$.
- $\frac{d_i^t}{\hat{d}_i^t}$ is the correcting factor.

Basic 1D implementat

Python implementation

```
from torchvision import transforms
import torch
from PIL import Image
import matplotlib.pyplot as plt
from ImageDebluring import *
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from torchvision import transforms
      2 import torch
      3 from PIL import Image

ModuleNotFoundError: No module named 'torchvision'
```

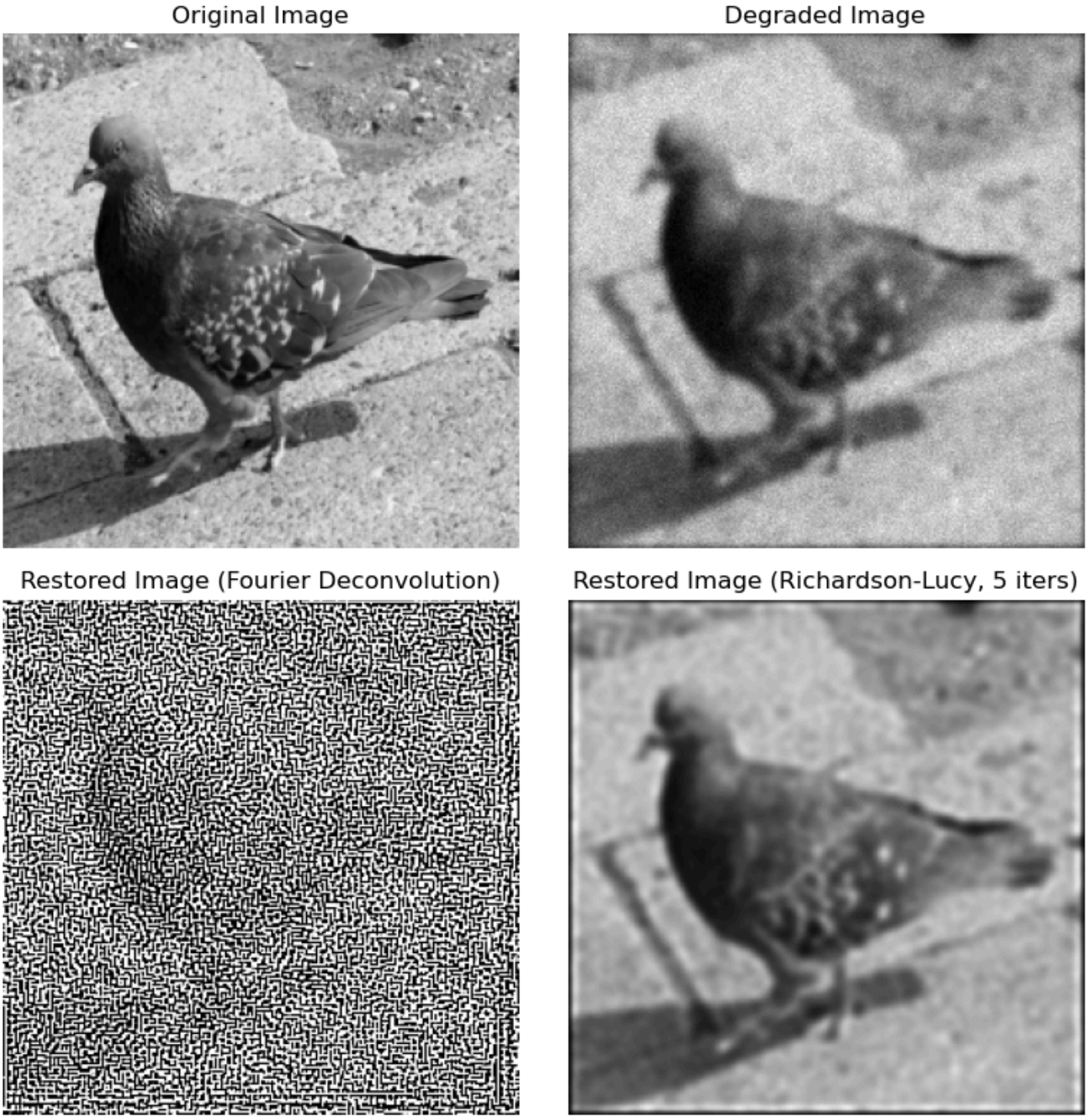
Degrading the image

```
# Load images
folder = 'Images/'
device = 'cpu'
image = 'pigeon.jpeg'

image_path = folder + image  # Replace with your image path
image = Image.open(image_path)
transform = transforms.Compose([
        transforms.Grayscale(),  # Convert to grayscale
        transforms.Resize((256, 256)), # Resize to 256x256 to speed up processing
        transforms.ToTensor()
    ])
img_tensor = transform(image).squeeze()  # Remove channel dimension

# Display the image
plt.imshow(img_tensor, cmap='gray')
plt.axis('off')
```

```
(np.float64(-0.5), np.float64(255.5), np.float64(255.5), np.float64(-0.5))
```

```
from scipy.signal import convolve2d as conv2

from skimage import color, data, restoration


iters_rl = 5
kernel = gaussian_normalised_kernel(size=11, sigma=2.0)
blury_image = convolution(img_tensor, kernel)
degraded_image = add_poisson_noise(blury_image, scale_factor=500)

restored_image_ft = deconvolution(degraded_image, kernel)
restored_image_rl30 = richardson_lucy(degraded_image, kernel, num_iters=iters_rl)
restored_image_rlski = restoration.richardson_lucy(degraded_image.numpy(), kernel.nu


# Display the results
fig, axs = plt.subplots(2, 2, figsize=(8,8))
axs[0,0].imshow(img_tensor, cmap='gray')
axs[0,0].set_title('Original Image')
axs[0,0].axis('off')
axs[0,1].imshow(degraded_image, cmap='gray')
axs[0,1].set_title('Degraded Image')
axs[0,1].axis('off')
axs[1,0].imshow(restored_image_ft, cmap='gray')
axs[1,0].set_title('Restored Image (Fourier Deconvolution)')
axs[1,0].axis('off')
axs[1,1].imshow(restored_image_rl30, cmap='gray')
axs[1,1].set_title(f'Restored Image (Richardson-Lucy, {iters_rl} iters)')
axs[1,1].axis('off')

fig.tight_layout()
```

```
kernel shape: torch.Size([11, 11])
```



Discussion

- De-convolution by fourier transform doesn’t work. The image is unrecognisable and the noise seems to have been amplified by the de-convolution.

- The Richardson Lucy algorithm removes most of the noise and also helps to de-blur the image. The contrast on the new image isn't as clear as the original image but is much better than the observed image.
- The skimage implementation of the RLA has the same effect as the one programmed above with no noticeable difference for 5 iterations. However for a larger number of iterations the above implementation causes the image to become very dark. Prehaps this is due to the normalisation factor $\frac{f_i'^t}{\hat{f}_i^t}$ always being less than 1. It is unclear why this is happening.
- Although the RLA seems to work for the complex image (pigeon) the darkening after a larger number of iterations is concerning. Each time the convolution kernel is applied over the image it makes the image smaller, therefore padding is used to keep the image the same size. The padding involves adding zeros to the edge of the image where the convolution cannot be applied. This means that the edge of the image is darker than the rest of the image. The RLA is an iterative process and therefore this darkening effect is amplified each time the convolution kernel is applied, therefore the image becomes darker after many iterations especially when the kernel size and standard deviation are large.

Tuning the Richardson-Lucy Algorithm with 1D functions

Its faster and simpler to experiment with 1D functions than 2D images. The same algorithm is used in 1D as it is in 2D. Furthermore there is more 1D data than 2D data.

Aims

- Understand how random noise is changed by convolution
- Test the Richardson-Lucy Algorithm (RLA) in 1D
- Tune the RLA parameters in order to maximize PSNR (peak signal to noise ratio) and SSIM (structural similarity index measure).

Deconvolving a 1D signal

A square wave was convolved using a gaussian kernel and then poisson noise was added. The square wave was normalized between 0 and 1 as this was the 2D signal (image) was prepared as the same algorithms were used. The RLA was then used to try and recover the original signal.

```
import torch
import matplotlib.pyplot as plt
from ImageDebluring import *
from PIL import Image
from torchvision import transforms


def step_function(period=2.0):
    xs = torch.linspace(-5, 5, steps=1000)
    ys = (((xs.round() % period) == 0).float() + 0.2)/2
    return xs, ys

def sin_function(period=2.0):
    xs = torch.linspace(-5, 5, steps=1000)
    ys = (torch.sin(xs * (2 * torch.pi / period)) + 1) / 2
    return xs, ys

def pigeon_function():
    image = Image.open('Images/pigeon.jpeg').convert('L')  # Convert to grayscale
    transform = transforms.Compose([
        transforms.Resize((1, 1000)),  # Resize to 1x1000
        transforms.ToTensor()
    ])
    img_tensor = transform(image).squeeze()  # Remove channel dimension
    return torch.linspace(0, 10, steps=1000), img_tensor


def plot_deblurring_example(func, padding_mode = 'replicate', title = None, num_ite
    if title is None:
        title = func.__name__.replace('_', ' ').title()
    x, y = func()
    signal = y.unsqueeze(0).unsqueeze(0)  # Add batch and channel dimensions
    kernel = gaussian_normalised_kernel_1D(size = 61, sigma = 10.0)

    blurred_signal = convolution_1D(signal, kernel, padding_mode=padding_mode)
    measured_signal = add_poisson_noise(blurred_signal, scale_factor=scale_factor)

    deblurred_signal = richardson_lucy(measured_signal, kernel, num_iters=num_iters,

    fig, axs = plt.subplots(2, 2, figsize=(12, 10))
    fig.suptitle(title, fontsize=16)
    axs[0, 0].plot(y)
    axs[0, 0].set_title('Step Function')
    axs[1, 0].plot(measured_signal.squeeze())
    axs[1, 0].set_title('Measured Signal (Blurred + Noisy)')
    axs[0, 1].plot(blurred_signal.squeeze())
    axs[0, 1].set_title('Blurred Signal')
    axs[1, 1].plot(deblurred_signal.squeeze())
    axs[1, 1].set_title(f'Estimated Signal (Richardson-Lucy, {num_iters} iters)')
    plt.tight_layout()
```

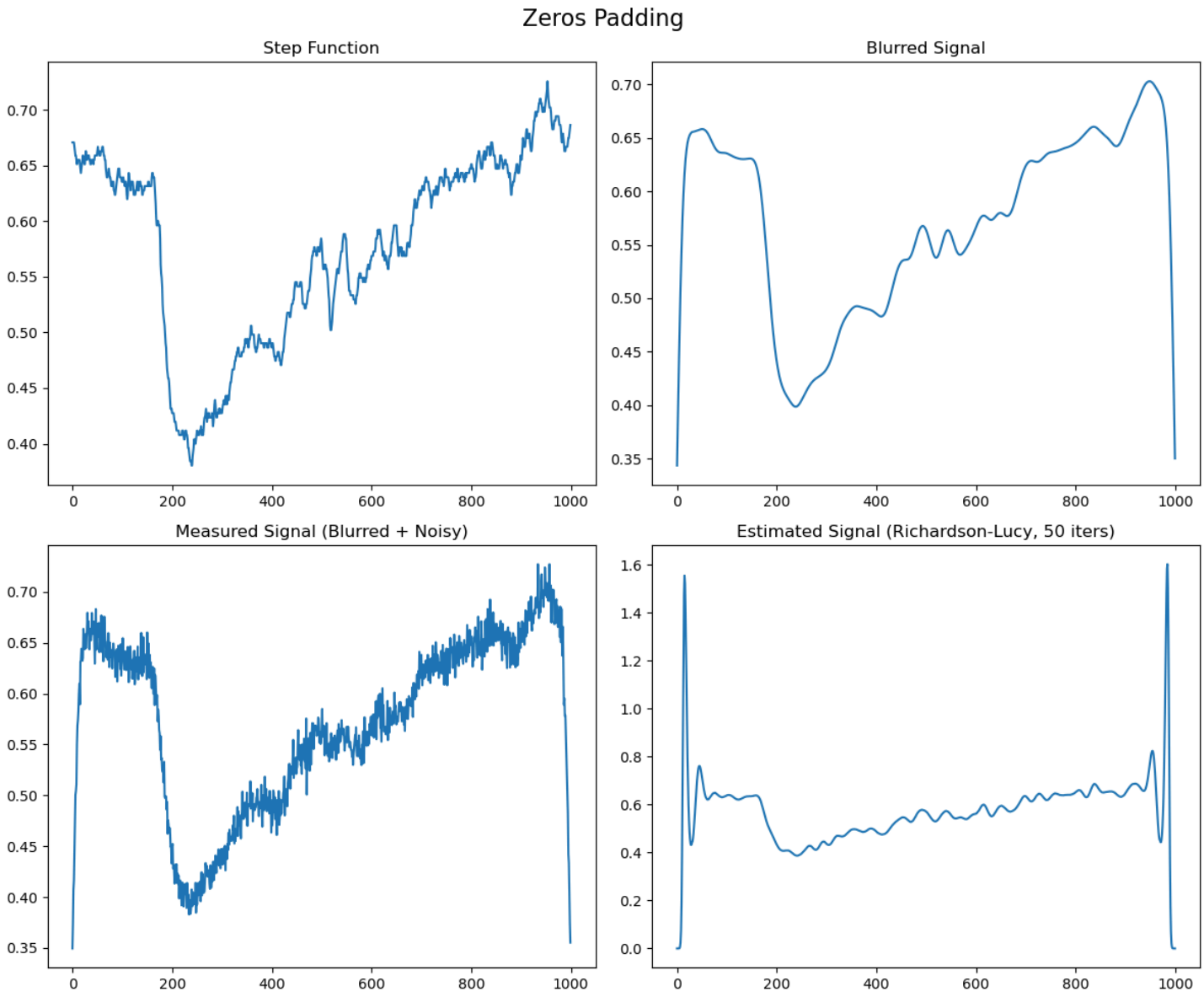
```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import torch
      2 import matplotlib.pyplot as plt
      3 from ImageDebluring import *

ModuleNotFoundError: No module named 'torch'
```

Padding

After a convolution the size of the image decreases since the kernel cannot be applied around edge pixels. For a kernel of size n and an image of size m the convoluted image will have size $m - n + 1$. This is problematic since the LRA requires the measured image and the kernel to be the same size. Padding increases the size of the image by adding pixels around the edge.

- Zeros padding: the easiest to implement form of padding. The edge pixels are set to zero. The problem with zeros padding is that overtime it will decrease the overall brightness of the image starting from the edges. This means that a more iterations will gradually decrease the brightness of the image.

```
plot_deblurring_example(pigeon_function, padding_mode='zeros', title = 'Zeros Paddi
```



The measured signal is far from the true signal since the edge pixels are much darker due to the zeros padding. Here there are also erroneous results since some pixel intensities are greater than 1.

- replicate padding repeats the value of the edge pixels. This means that the overall image is not darkened like when using zeros padding.

```
plot_deblurring_example(pigeon_function, padding_mode='replicate', title = 'Replicat
```

The replicate padding works much more successfully as there isn't issues around edge pixels. The image will not lose brightness over time.

Measures of image quality

- R^2 measures the correlation between the estimated signal and the true signal. However this has low human visual relevance. A higher R^2 indicates that the overall shape of the estimated signal is a close match to the true signal.
- MSE measures the average squared difference between the estimated signal and the true signal. Again this has low human visual relevance. This doesn't capture perceptual image quality well. A lower MSE indicates a higher quality image.

- The PSNR is the logarithmic ratio between signal power to error power and is given in db since its logarithmic. This has a higher visual relevance compared to MSE. A value between 30 and 50 db indicates that the image is a good quality. Below 25db indicates heavy distortion or noise.
- The SSIM compares the luminance, contrast and structure of the estimated signal to the true signal. This has a high visual relevance. A SSIM of 1 indicates a perfect match to the true signal a SSIM of above 0.97 indicates a good match a SSIM of below 0.95 indicates significant degregation.

Number of iterations

Its also easier to find training datasets for machine learning algorithms in 1D. The Richardson lucy algorithm has 3 “tuning parameters” the first is the convolution kernel which is assumed to be a gaussian with known standard deviation. The second is the number of iterations the algorithm runs for. The third is the initial guess. The following graphs show how the number of iterations of the RLA effects how close the estimated signal is to the true signal. The results from this section will be used to help train a machine learning algorithm to predict the optimal number of iterations for a given image.

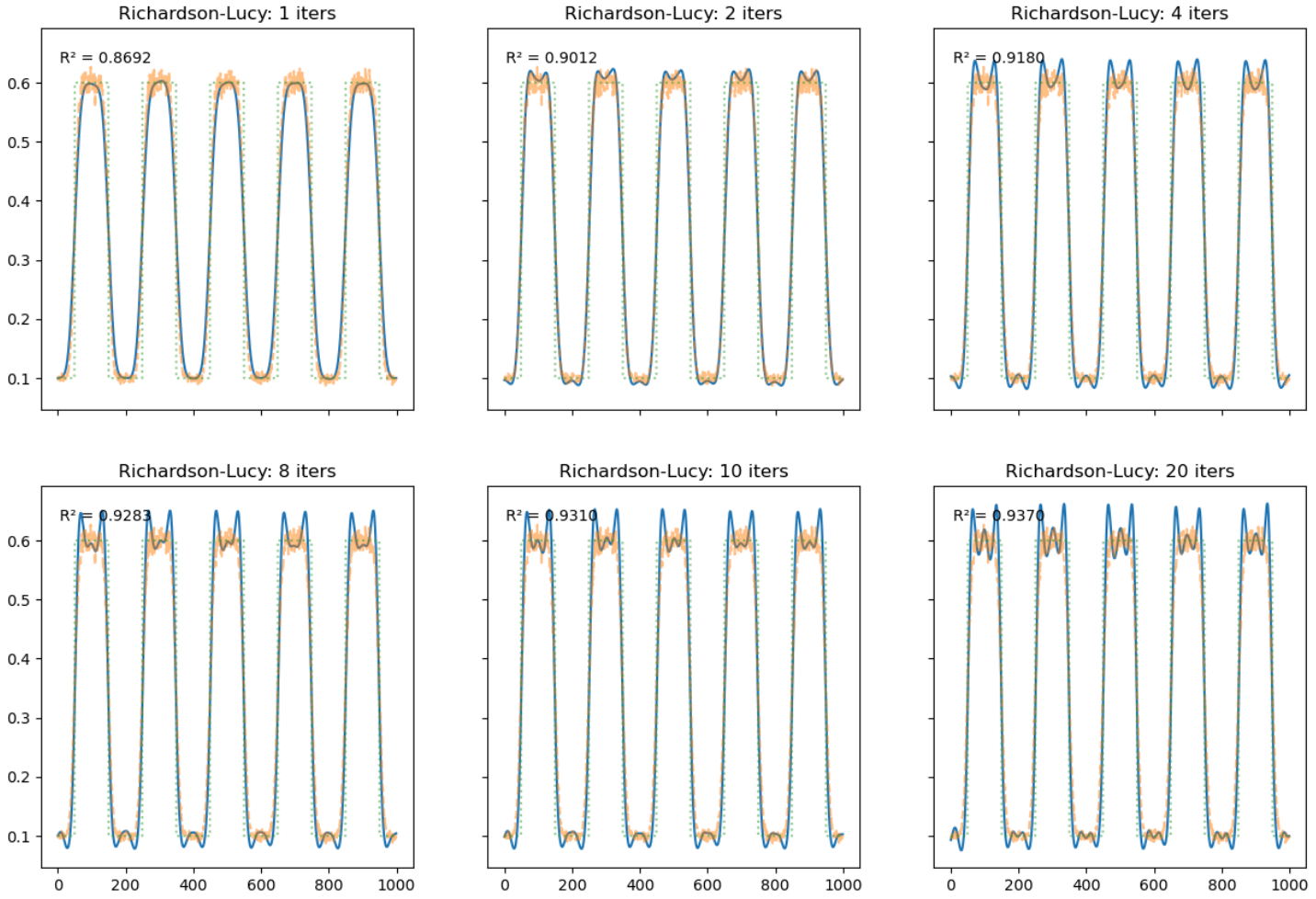
```
import torchmetrics
from skimage import metrics

def plot_iterations_example(func, padding_mode = 'replicate', title = None, scale_fa
                            , iters = [1,2,4,8,10,20,30,50,100], plot_all = True, fi
    x, y = func(**kwargs)
    signal = y.unsqueeze(0).unsqueeze(0)
    kernel = gaussian_normalised_kernel_1D(size = 61, sigma = 10.0)
    blurred_signal = convolution_1D(signal, kernel, padding_mode=padding_mode)
    measured_signal = add_poisson_noise(blurred_signal, scale_factor=scale_factor)
    r2s = []
    mses=[]
    PSNRs = []
    SSIMs = []

    if plot_all:
        fig1, axs1 = plt.subplots(2, 3, figsize=(15, 10), sharex=True, sharey=True)
        axs1 = axs1.flatten()
        j = 0

    for i in range(1,iters[-1]+1):
        deblurred_signal = richardson_lucy(measured_signal, kernel, num_iters=i, pad
        r2 = torchmetrics.functional.r2_score(deblurred_signal.squeeze(), y.squeeze(
        mse = torchmetrics.functional.mean_squared_error(deblurred_signal.squeeze(),
        r2s.append(r2.item())
        mses.append(mse.item())
        PSNR = 10 * torch.log10(1 / mse)
        SSIM = metrics.structural_similarity(y.squeeze().numpy(), deblurred_signal.s
        PSNRs.append(PSNR.item())
        SSIMs.append(SSIM.item())
        print(i/iters[-1]*100, '%', end='\r')

        if i in iters and plot_all:
            axs1[j].plot(deblurred_signal.squeeze())
            axs1[j].plot(measured_signal.squeeze(), linestyle='dashed', alpha=0.5)
            axs1[j].plot(y.squeeze(), linestyle='dotted', alpha=0.5)
            axs1[j].plot()
            #axs1[j].legend(['Estimated Signal', 'Measured Signal', 'True Signal'],
            axs1[j].set_title(f'Richardson-Lucy: {i} iters')
            axs1[j].text(0.05, 0.9, f'R² = {r2:.4f}', transform=axs1[j].transAxes, f
            j += 1


    if fig is None or axs is None:
        fig, axs = plt.subplots(1, 2, figsize=(12, 5))
    axs[0].plot(PSNRs, label = label)
    axs[0].set_xlabel('Iterations')
    axs[0].set_ylabel('PSNR')

    axs[1].plot(SSIMs, label = label)
    axs[1].set_xlabel('Iterations')
    axs[1].set_ylabel('SSIM')
```

```
    fig.tight_layout()

    return fig, axs
```

```
plot_iterations_example(step_function, padding_mode='replicate', title = 'Replicate
```

```
25.0 %999999999998 %
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 plot_iterations_example(step_function, padding_mode='replicate', title = 'Re

Cell In[4], line 34, in plot_iterations_example(func, padding_mode, title, scale_fac
     31 print(i/iters[-1]*100, '%', end='\r')
     33 if i in iters and plot_all:
---> 34     axs1[j].plot(deblurred_signal.squeeze())
     35     axs1[j].plot(measured_signal.squeeze(), linestyle='dashed', alpha=0.5)
     36     axs1[j].plot(y.squeeze(), linestyle='dotted', alpha=0.5)

IndexError: index 6 is out of bounds for axis 0 with size 6
```



When the number of iterations are low the estimated signal is smooth and this doesn't reflect the shape of the true signal, especially around the vertical lines. As the number of iterations increases the shape of the estimated signal improves. The steepness of the lines from where the square wave changes value value increases. However as the number of iterations increases the noise from the

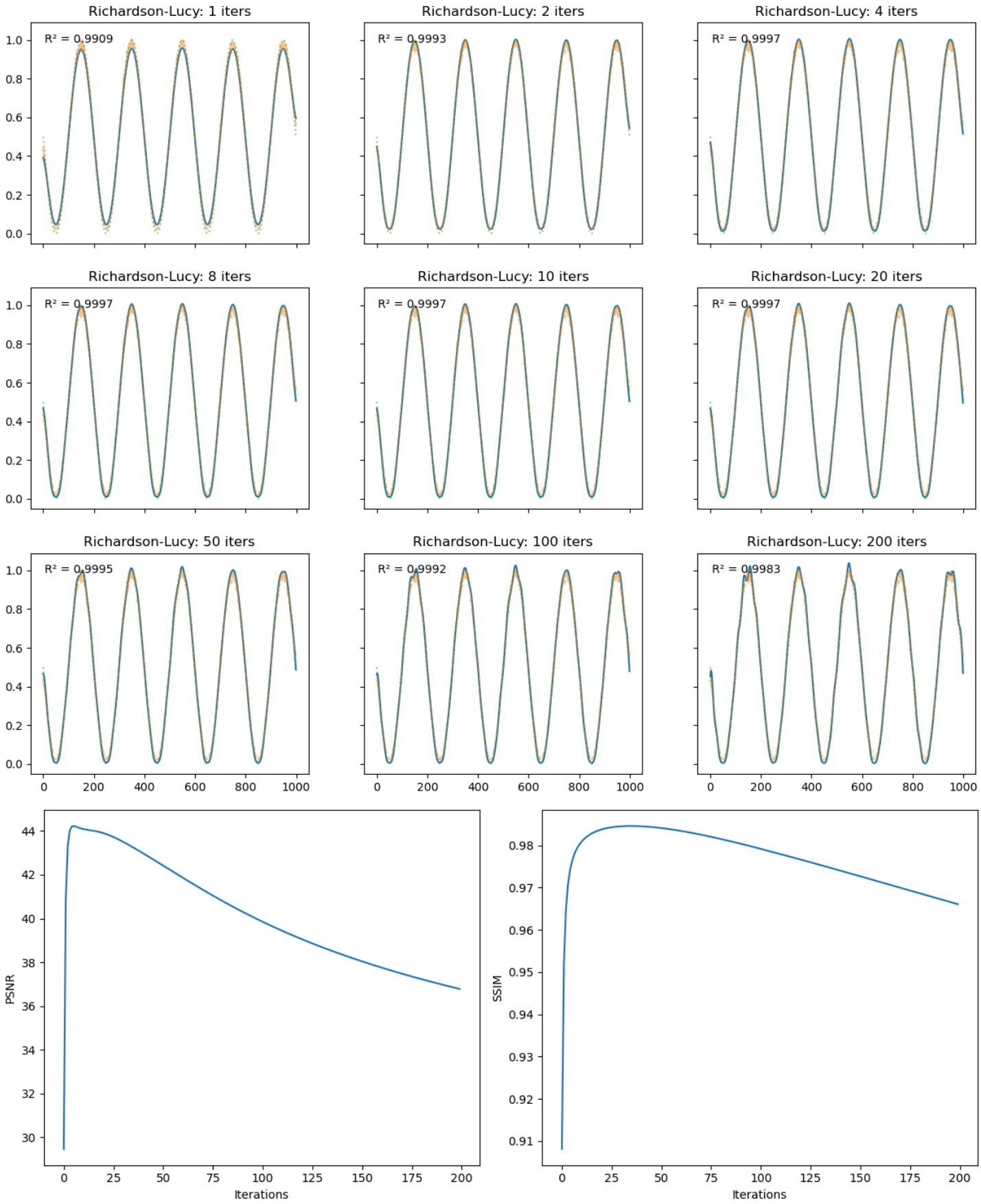
estimated signal also increases. Often the signal will become noisy even when there is no noise in the measured signal.

It is clear from both the graph of R^2 score against iteration and the graph of MSE against iteration that the RLA is converging towards a solution. This solution doesn't appear to be the optimal solution since after 200 iterations the R^2 value hadn't yet hit 0.95. Similarly the MSE seems to be converging to a value at about 0.003.

```
plot_iterations_example(sin_function, padding_mode='replicate', title = 'Replicate P
```

```
100.0 %9999999999 %%
```

```
(<Figure size 1200x500 with 2 Axes>,
 array([<Axes: xlabel='Iterations', ylabel='PSNR'>,
        <Axes: xlabel='Iterations', ylabel='SSIM'>], dtype=object))
```



In this example the convolution doesn't make so much difference to the true signal as its smoother. This means that there is less to be gained by carrying out more iterations of the RLA since the shape

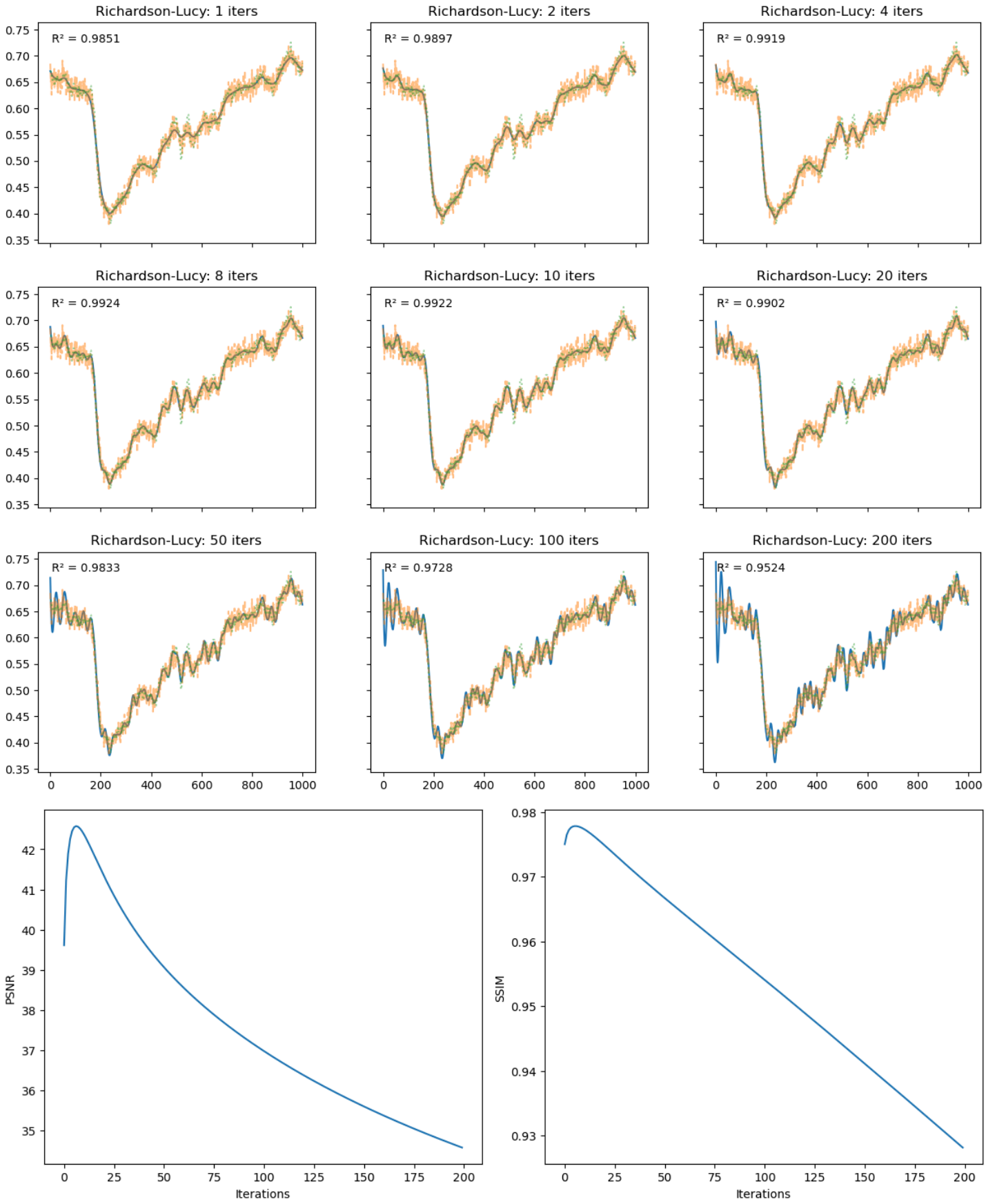
of the signal is already satisfactory. Carrying out more iterations just adds noise to the signal although this seems relativly insignificant as seen from the very large R^2 value of 0.998.

Here the RLA initially gets very close to the true signal but then starts to diverge after 10 iterations. The reasons for this are unclear but may be to do with specific frequencies which the RLA is amplifying. By extending the graph it is clear that it doesn't seem to a solution after 1000 iterations, perhaps the rate of convergence is very slow.

```
plot_iterations_example(pigeon_function, padding_mode='replicate', title = 'Replica
```

```
100.0 %9999999999 %%
```

```
(<Figure size 1200x500 with 2 Axes>,
 array([<Axes: xlabel='Iterations', ylabel='PSNR'>,
        <Axes: xlabel='Iterations', ylabel='SSIM'>], dtype=object))
```



After about 4 iterations the RLA has got as close as it gets to the true signal with a R^2 of about 0.993. After this the RLA diverges away from the true signal as the estimated signal becomes very noisy.

The divergence in this case is much worse than in the previous example. This suggests that the number of iterations is an important variable to tune using machine learning.

Conclusion

Padding is necessary to keep the signal size the same after convolution which is essential in the RLA. Replicate padding doesn't cause the images to darken over time and prevents artifacts being introduced into the solution due to boundary effects.

The number of iterations effects how close the estimated signal can get to the true signal. Selecting a number of iterations is a tradeoff between how well the shape of the estimated signal matches the true signal and how noisy the estimated signal is. The optimal number of iterations seems to be lower for real world signals perhaps due to the different frequency components of the signal.

Neural Networks for Image De-noising

- Understand how multi-layer neural networks function.
- Build a multi layer perceptron that can analyze and deconvolve step functions

Theory

Assume the noisy image can be written as a matrix P_i where each element represents the pixel intensity of the i th pixel. The goal is to train a neural network $f(P;K)$ where K is the known convolution kernel to output a denoised image $Q_i = f(P;K)$. The nodes between the inputs and the outputs are the 'hidden nodes'. All nodes have a value between 0 and 1.

The node $a_0^{(1)}$ will have its value computed from a sum of the weighted inputs from the first layer.

$$a_0^{(1)} = \sigma(b_0 + \sum_i w_i P_i)$$

Where σ is the normalisation function given by $\sigma(x) = \frac{1}{1+e^{-x}}$. w_i are the weights associated with each input pixel P_i, b_0 is the bias term.

The first next layer of hidden nodes can be calculated from:

$$\mathbf{a}^{(i+1)} = \sigma(W^{(i)}\mathbf{a}^{(i)} + \mathbf{b}^{(i)})$$

Where $W^{(i)}$ is the weight matrix and $\mathbf{b}^{(i)}$ is the bias vector which transforms the i th layer into the $i+1$ th layer. The values of $W^{(i)}$ and $\mathbf{b}^{(i)}$ are learned during training.

Gradient Descent

The cost function measures the success of the machine learning algorithm one way it could be defined is as follows:

$$C(\mathbf{w}) = \sum_i (\hat{p}_i - p_i)^2$$

Where $\hat{p}_i$ is the estimated state from the MLP and p_i is the known state. The vector $\mathbf{w}$ contains all the weights in the network. This sum is small when the network is close to being “correct”.

Using gradient decent can be used to find a local minima, an optimized solution. The vector $-\Delta C(\mathbf{w})$ points in the direction of steepest decent.

Generate test data

```
import torch
import matplotlib.pyplot as plt
from ImageDebluring import *
from sklearn.preprocessing import MinMaxScaler

def step_function(period=2.0, amplitude=0.8, y_intercept=0.0, steps=1000, x_offset=(
    s = steps
    xs = torch.linspace(-5, 5, steps=s)
    ys = amplitude * torch.heaviside(torch.sin(2 * torch.pi * (xs + x_offset) / peri
    return ys

def generate_random_function(vary_period = False, vary_amplitude = False, vary_y_int
    if vary_period:
        period = torch.rand(1, generator=rng).item()*3 + 5
    else:
        period = 5
    if vary_amplitude:
        amplitude = torch.rand(1, generator=rng).item() * 0.9 + 0.1
    else:
        amplitude = 0.3
    if vary_y_intercept:
        y_intercept = torch.rand(1, generator=rng).item() * 0.3
    else:
        y_intercept = 0.3
    if vary_x_offset:
        x_offset = torch.rand(1, generator=rng).item() * 2*torch.pi
    else:
        x_offset = 0.0
    return step_function(period, amplitude, y_intercept, x_offset=x_offset).unsqueez

def generate_test_data(amount = 10000, noise_scale = 5000, blur_std = 61, kernel_siz
    if rng is None and seed is not None:
        rng = torch.Generator().manual_seed(seed)
    y = []
    y_scalers = []
    X = []
    X_scalers = []
    for i in range (amount):
        data = generate_random_function(vary_period = vary_period, vary_amplitude =
        degraded_data = degrade_image_1D(data, noise_scale=noise_scale, kernel=gauss
        scaler_x = MinMaxScaler()
        scaler_y = MinMaxScaler()
        # reshape to 2D (n_samples, n_features) for sklearn, then back to 1D
        degraded_np = degraded_data.squeeze().numpy().reshape(-1, 1)
        data_np = data.squeeze().numpy().reshape(-1, 1)
        degraded_scaled = scaler_x.fit_transform(degraded_np).reshape(-1)
        data_scaled = scaler_y.fit_transform(data_np).reshape(-1)
        degraded_data = torch.tensor(degraded_scaled, dtype=torch.float32)
        data = torch.tensor(data_scaled, dtype=torch.float32)
        X_scalers.append(scaler_x)
        y_scalers.append(scaler_y)
        X.append(degraded_data.squeeze())
        y.append(data.squeeze())
```

```
    # Make tensors
    X = torch.stack(X)
    y = torch.stack(y)

    return X, y, X_scalers, y_scalers
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import torch
      2 import matplotlib.pyplot as plt
      3 from ImageDebluring import *

ModuleNotFoundError: No module named 'torch'
```

Defining the MLP model

- input size for this dataset is 1000 (number of points in the 1D signal)
- hidden size can be adjusted, start with 128
- output size is also 1000 (number of points in the 1D signal)
- requires_grad=True to enable backpropagation

Forward Pass

- Linear transformation of inputs $z^{(1)} = XW^{(1)} + b^{(1)}$
- Apply activation function $a^{(1)} = \sigma(z^{(1)})$ to node values
- Linear transformation of hidden layer $z^{(2)} = a^{(1)}W^{(2)} + b^{(2)}$
- Apply activation function $a^{(2)} = \sigma(z^{(2)})$ to get final output

Backpropagation

- `backward` updates the weights and the baises
- `epochs` the number of times the model sees the entire dataset
- `lr` hyperparameter that controls the step size for weighted updates
- `loss` the MSE

```
class MLP:
    def __init__(self, input_size, hidden_size, output_size, rng = None):
        self.rng = rng
        self.W1 = torch.randn(input_size, hidden_size, requires_grad=True, generato
        self.b1 = torch.randn(1, hidden_size, requires_grad=True, generator=self.rn
        self.W2 = torch.randn(hidden_size, output_size, requires_grad=True, generat
        self.b2 = torch.randn(1, output_size, requires_grad=True, generator=self.rn

    def forward(self, X):
        self.z1 = torch.matmul(X, self.W1) + self.b1
        self.a1 = torch.sigmoid(self.z1) # applies sigmoid activation function
        self.z2 = torch.matmul(self.a1, self.W2) + self.b2
        self.a2 = torch.sigmoid(self.z2)
        return self.a2

    def backward(self,X,y,output,lr=0.01):
        m = X.shape[0]
        dz2 = output - y
        dW2 = torch.matmul(self.a1.T, dz2)
        db2 = torch.sum(dz2, axis=0)/m

        da1 = torch.matmul(dz2, self.W2.T)
        dz1 = da1*(self.a1*(1-self.a1))
        dW1=torch.matmul(X.T,dz1)/m
        db1 = torch.sum(dz1, axis=0) / m

        with torch.no_grad():
            self.W1 -= lr * dW1
            self.b1 -= lr * db1
            self.W2 -= lr * dW2
            self.b2 -= lr * db2

    def train(self, X, y, epochs = 1000, lr = 0.01):
        losses = []
        print("\nlosses")
        for i in range(epochs):
            output = self.forward(X)
            # Compute loss using MSE
            loss = torch.mean((output - y)**2)
            losses.append(loss.item())
            # Update weights
            self.backward(X, y, output, lr)

            print(f'\rEpoch {i}, Loss: {loss.item()}', end='', flush=True)
        return losses
```

Initialize model

```
def plot_losses(losses):
    fig, ax = plt.subplots()
    ax.scatter(range(len(losses)),  losses)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')

def train_model(X_train_scaled, y_train_scaled, hidden_size = 128, epochs = 100, lr
    input_size = X_train_scaled.shape[1]
    model = MLP(input_size, hidden_size, input_size, rng = rng)
    losses = model.train(X_train_scaled, y_train_scaled, epochs=epochs, lr=lr)
    plot_losses(losses)
    return model
```

Testing

- `amount` is the number of waves that are generated for training and testing
- `noise_scale` is the scale factor for the signal before poisson noise is added
- `blur_std` is the standard devation of the guassian being convolved with the signal
- `kernel_size` is the size of the guassian being convolved with teh signal
- `epochs` is the number of iterations that the machine learning algorithm will train for
- `lr` is the hyperparameter that controls the step size for weighted updates
- `hidden_size` the number of hidden nodes in the MLP
- `vary_period/amplitude/y_intercept/x_offset` boolean describes if these should be variables when generating the waves.
- `X/y` an optional parameter which if not provided the programme will generate its own data
- `X/y-scalers` scalers so data can be unscaled after running through the machine learning algorithm

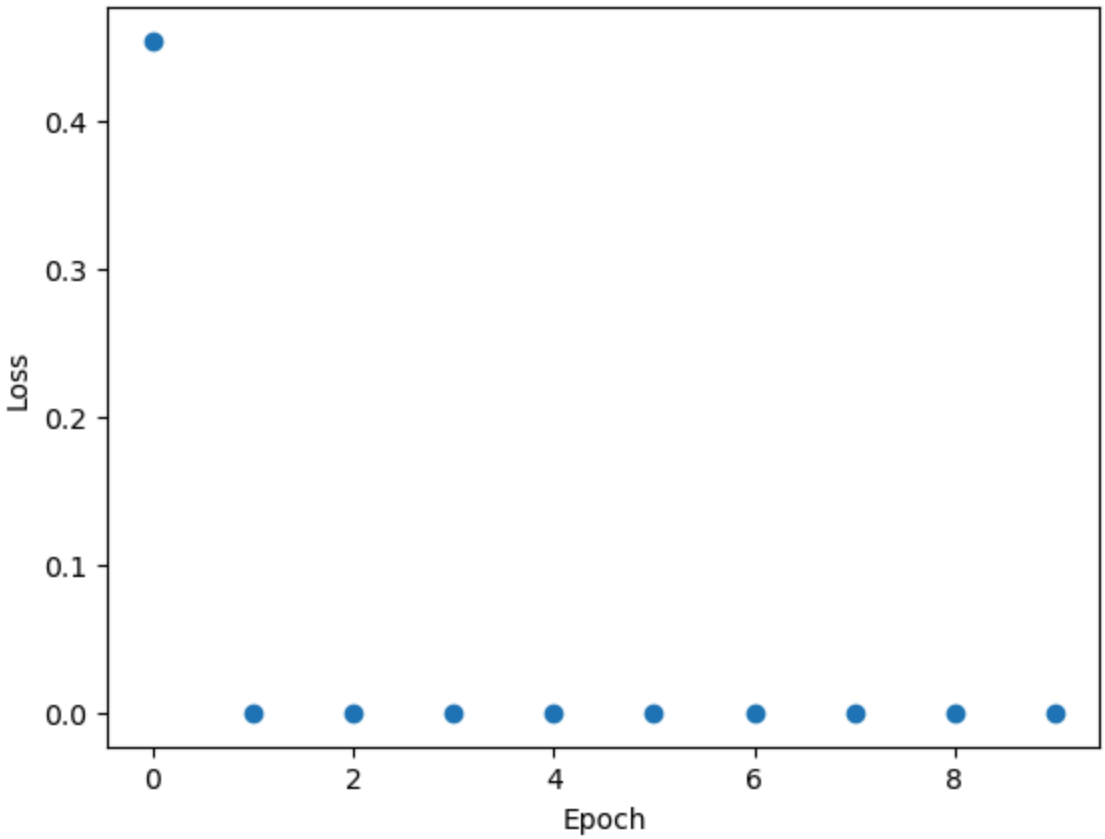
```
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split

def test_model(amount = 10000, noise_scale = 5000, blur_std = 61, kernel_size=20, ep
    if X is not None and y is not None:
        X_scaled = X
        y_scaled = y
    rng_data = torch.Generator().manual_seed(seed_data)
    rng_model = torch.Generator().manual_seed(seed_model)
    if X is None or y is None:
        print("Generating data...", end='', flush=True)
        X_scaled, y_scaled, X_scalers, y_scalers = generate_test_data(amount = amou
        print("\nData generated", end='', flush=True)
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled = train_test_split
    print("\nData preppared", end='', flush=True)
    model = train_model(X_train_scaled, y_train_scaled, epochs = epochs, lr = lr, hi
    print("\nModel trained", end='', flush=True)
    y_pred_scaled = model.forward(X_test_scaled)

    # Select 4 waves to vosialise
    fig, axs = plt.subplots(2,2, figsize=(8,8), sharex = True, sharey=True)
    axs = axs.flatten()

    for i in range (4):
        y_pred = y_scalers[i].inverse_transform(y_pred_scaled[i].detach().numpy().re
        X_test = X_scalers[i].inverse_transform(X_test_scaled[i].detach().numpy().re
        y_test = y_scalers[i].inverse_transform(y_test_scaled[i].detach().numpy().re
        sns.lineplot(x=np.arange(len(y_pred)), y=y_pred, ax=axs[i], label='predicte
        sns.lineplot(x=np.arange(len(X_test)), y=X_test, ax=axs[i], label='degraded
        sns.lineplot(x=np.arange(len(y_test)), y=y_test, ax=axs[i], label='original
        axs[i].legend()
    print("\nDone!")
```

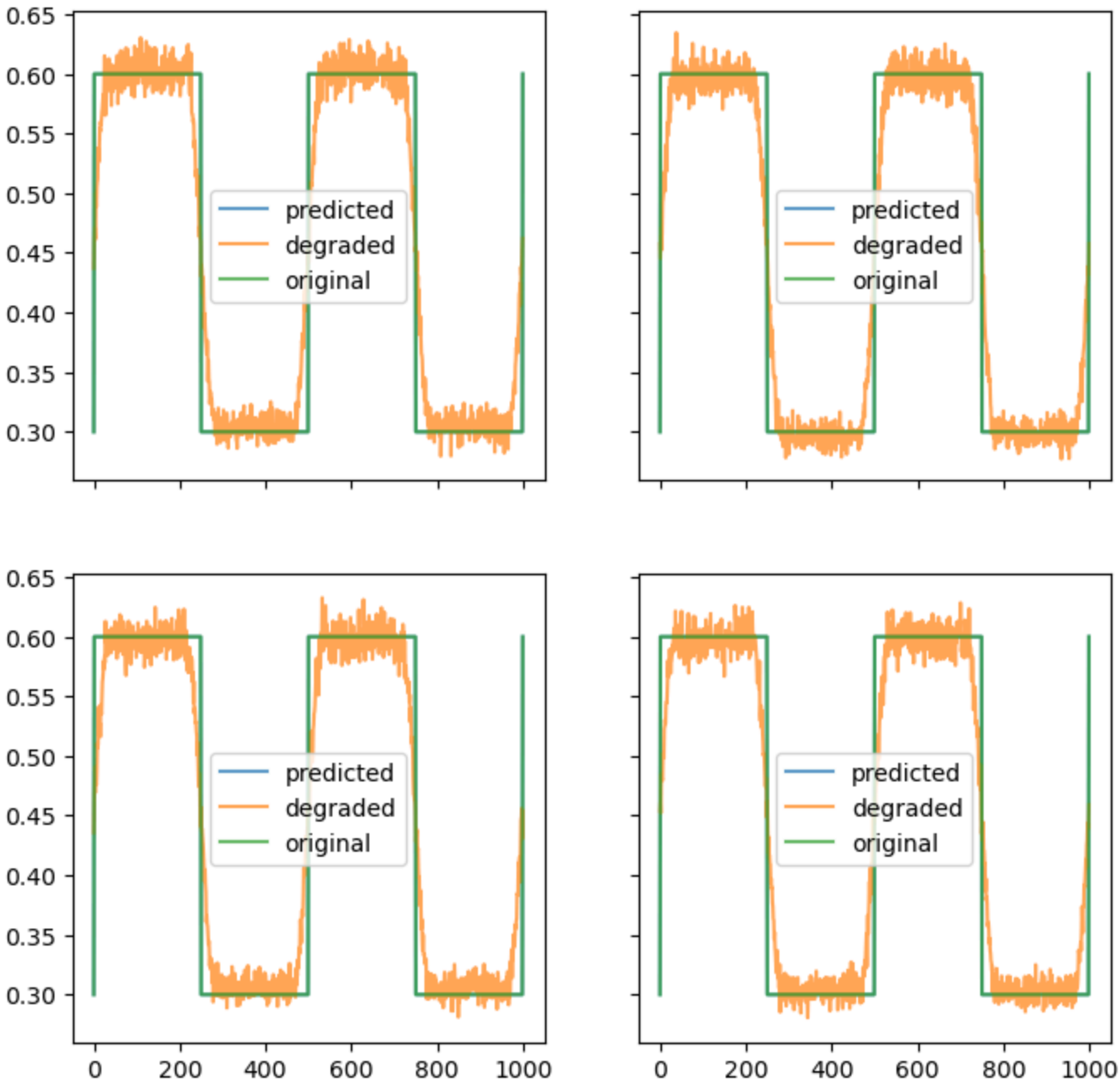
Simplest model

- Simple model where the parameters defining the square wave are the same for each iteration
- the only thing that changes is the noise that is added to the square waves each time

```
test_model(epochs=10)
```

```
Generating data...
Data generated
Data preppared
losses
Epoch 9, Loss: 3.0431750869253094e-11
Model trained
Done!
```


0.4
0.3
Loss
0.2
0.1
0.0
0
2
4
6
8
Epoch



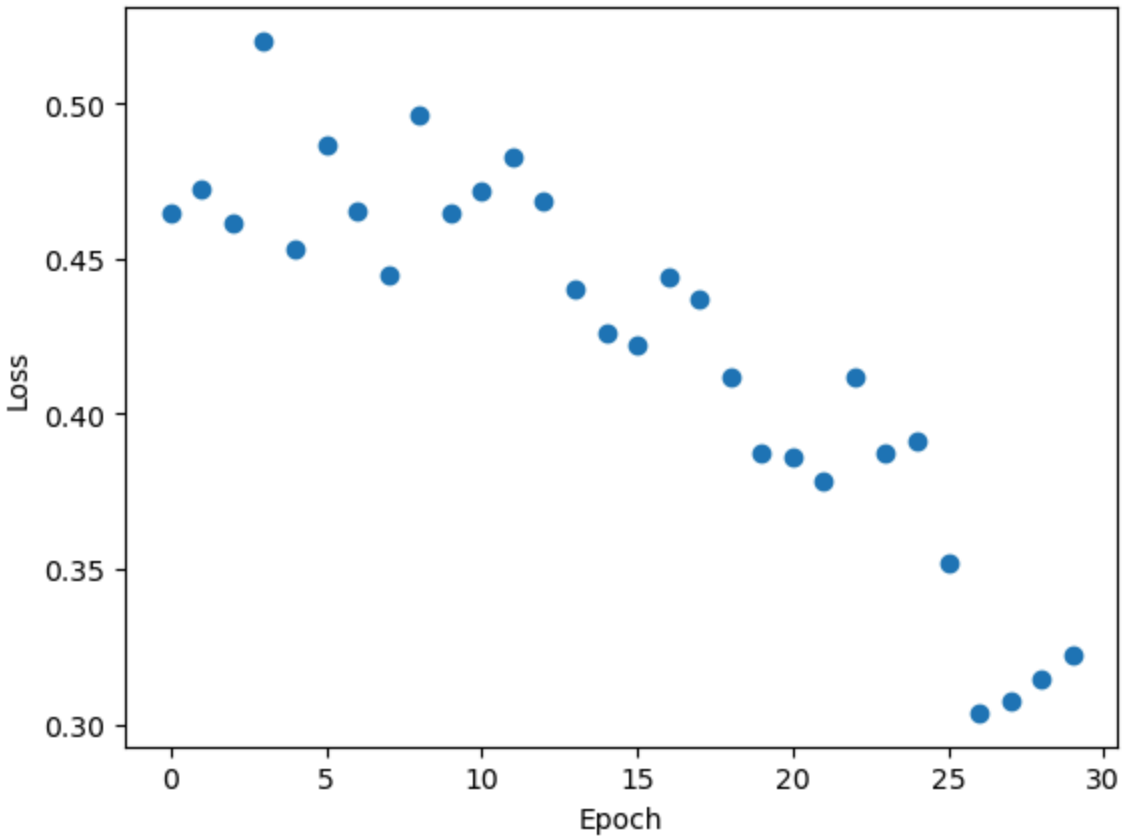
- Converges very quickly
- the predicted waves are identical to the square waves

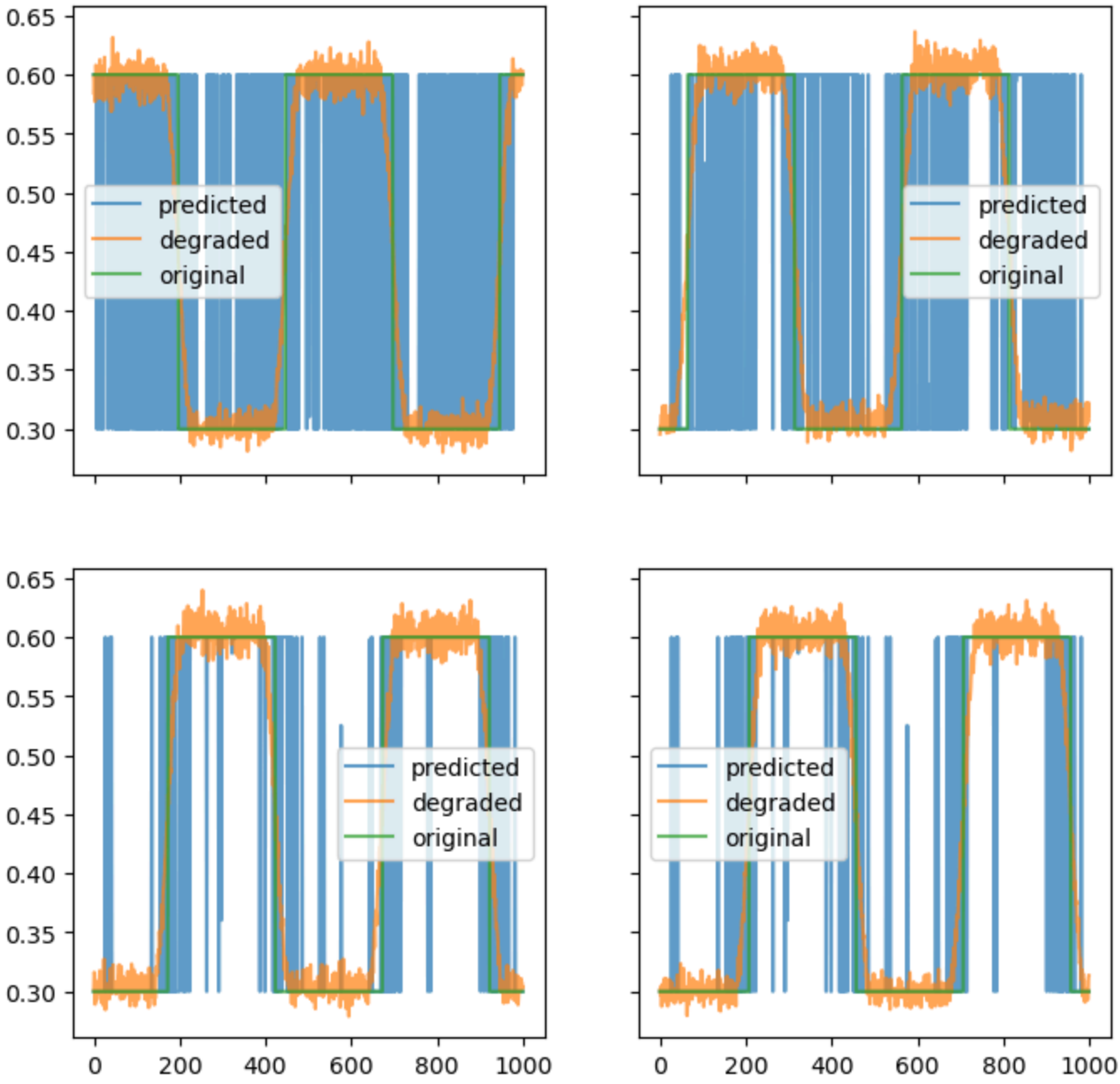
Varying the x offset

- now also varying the x offset which is generated randomly

```
X, y, X_scalers, y_scalers = generate_test_data(amount = 1000, noise_scale = 5000, l
test_model(X = X, y = y, hidden_size=256, epochs = 30, lr = 0.05, seed_data=42, seed
```

```
Data preppared
losses
Epoch 29, Loss: 0.32221731543540955
Model trained
Done!
```

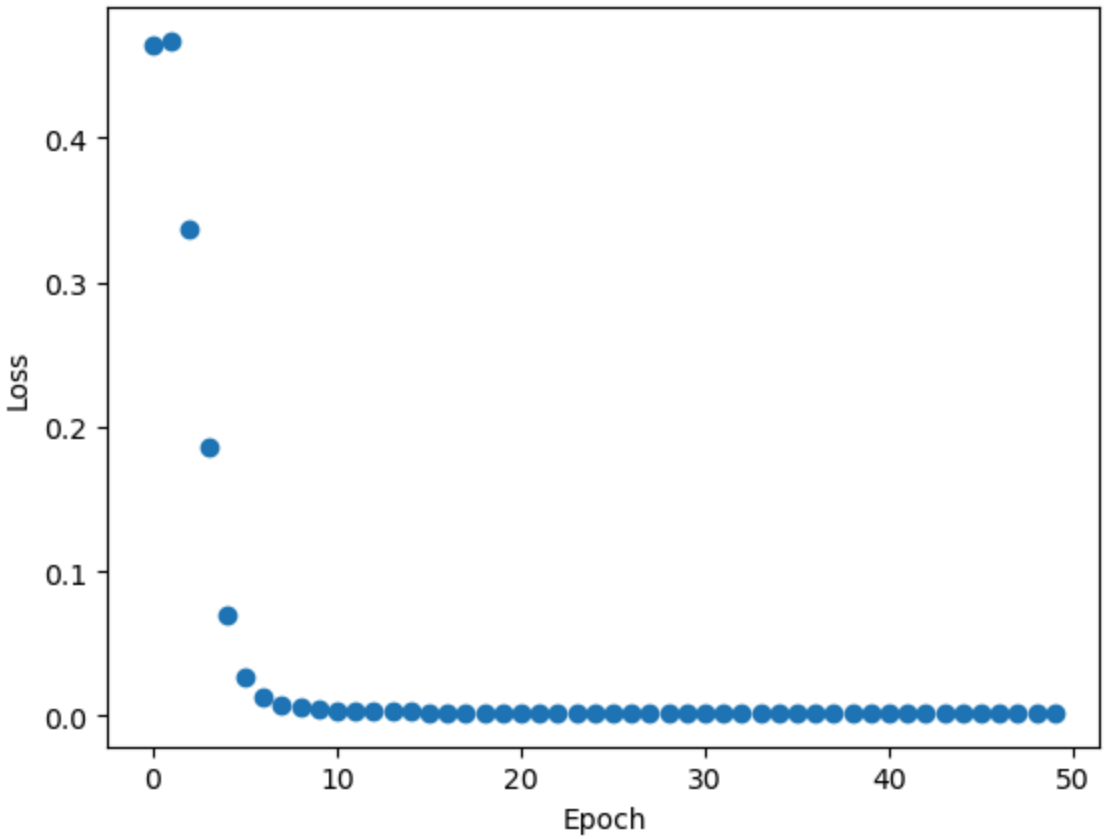

0.50
0.45
0.40
0.35
0.30
Loss
0
5
10
15
20
25
30
Epoch

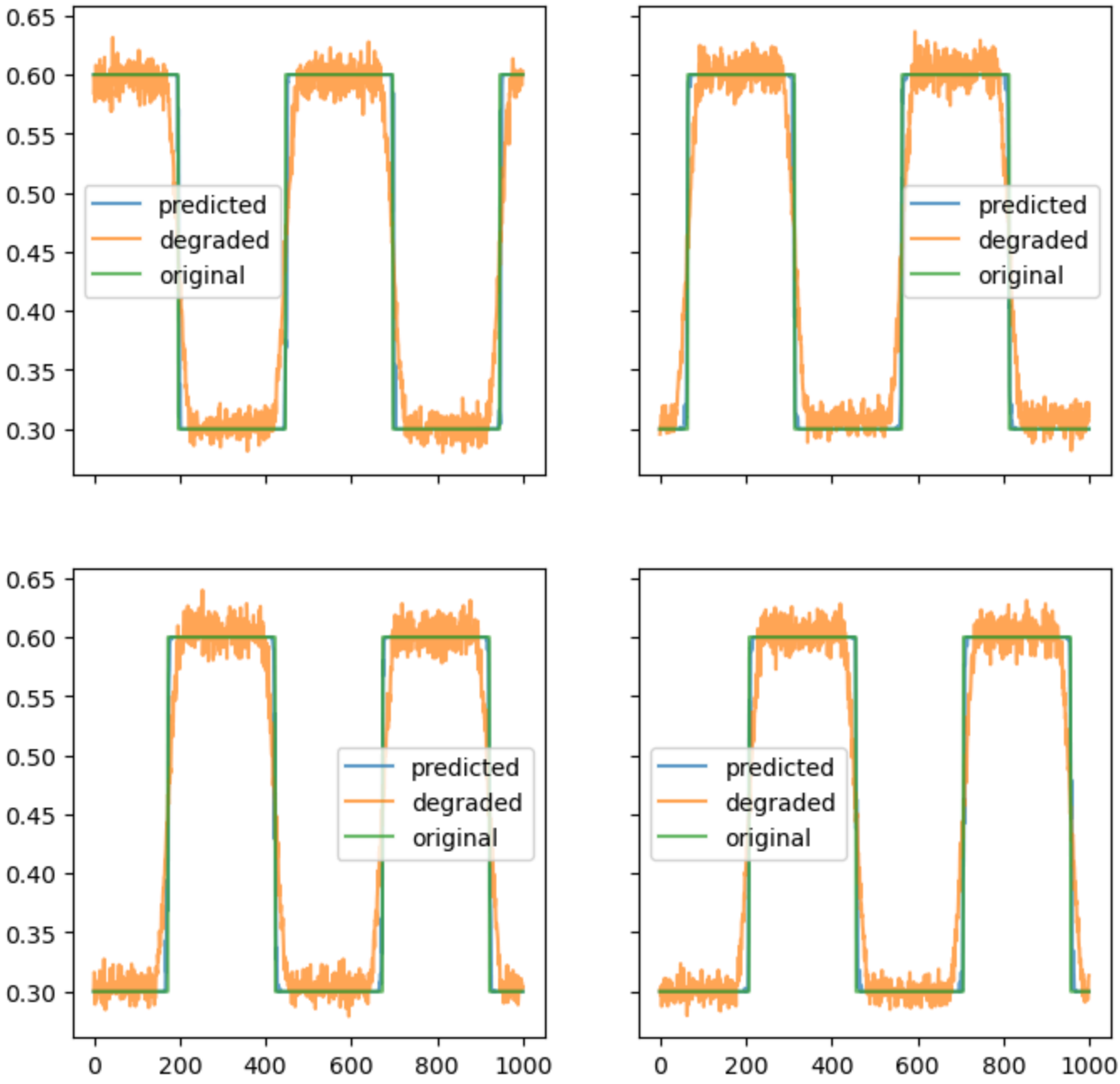


- This example doesn't work
- loss vs epoch is noisy
- considered decreasing `lr`

```
test_model(X = X, y = y, hidden_size=256, epochs = 50, lr = 0.005, X_scalers = X_sc
```

```
Data preppared
losses
Epoch 49, Loss: 0.0021088530775159597
Model trained
Done!
```


0.4
0.3
0.2
0.1
0.0
Loss
0
10
20
30
40
50
Epoch

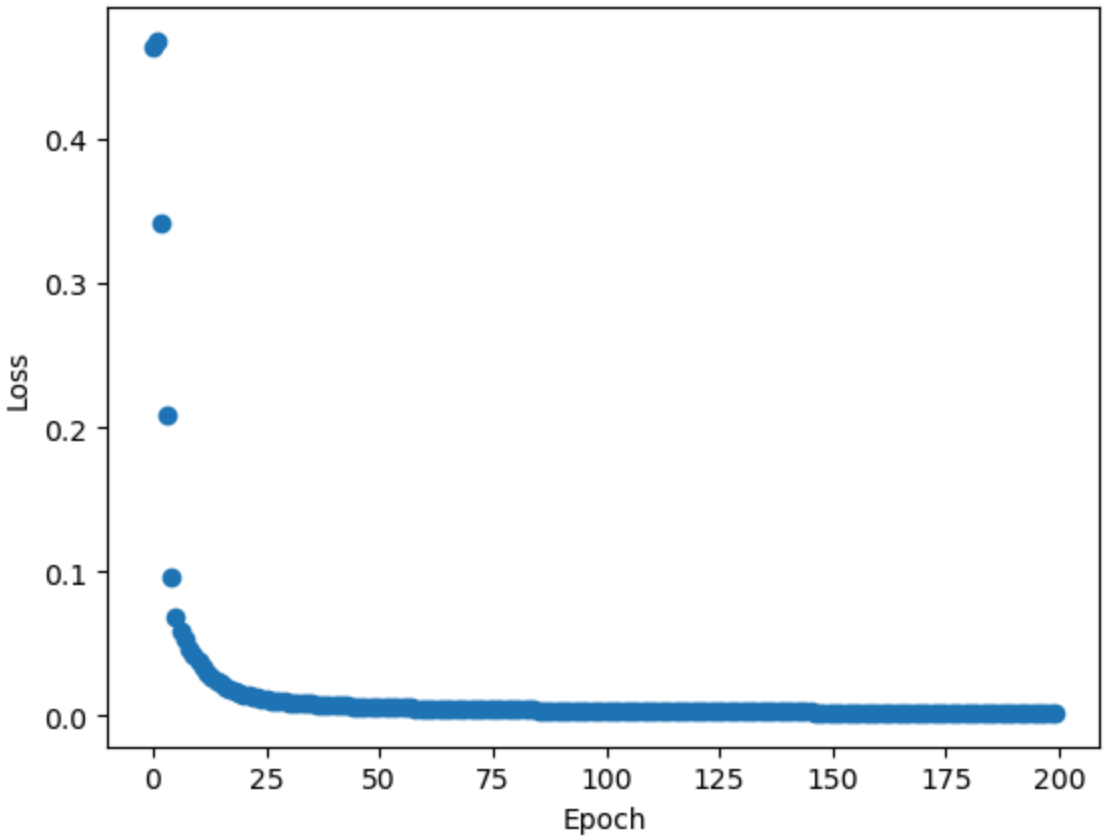


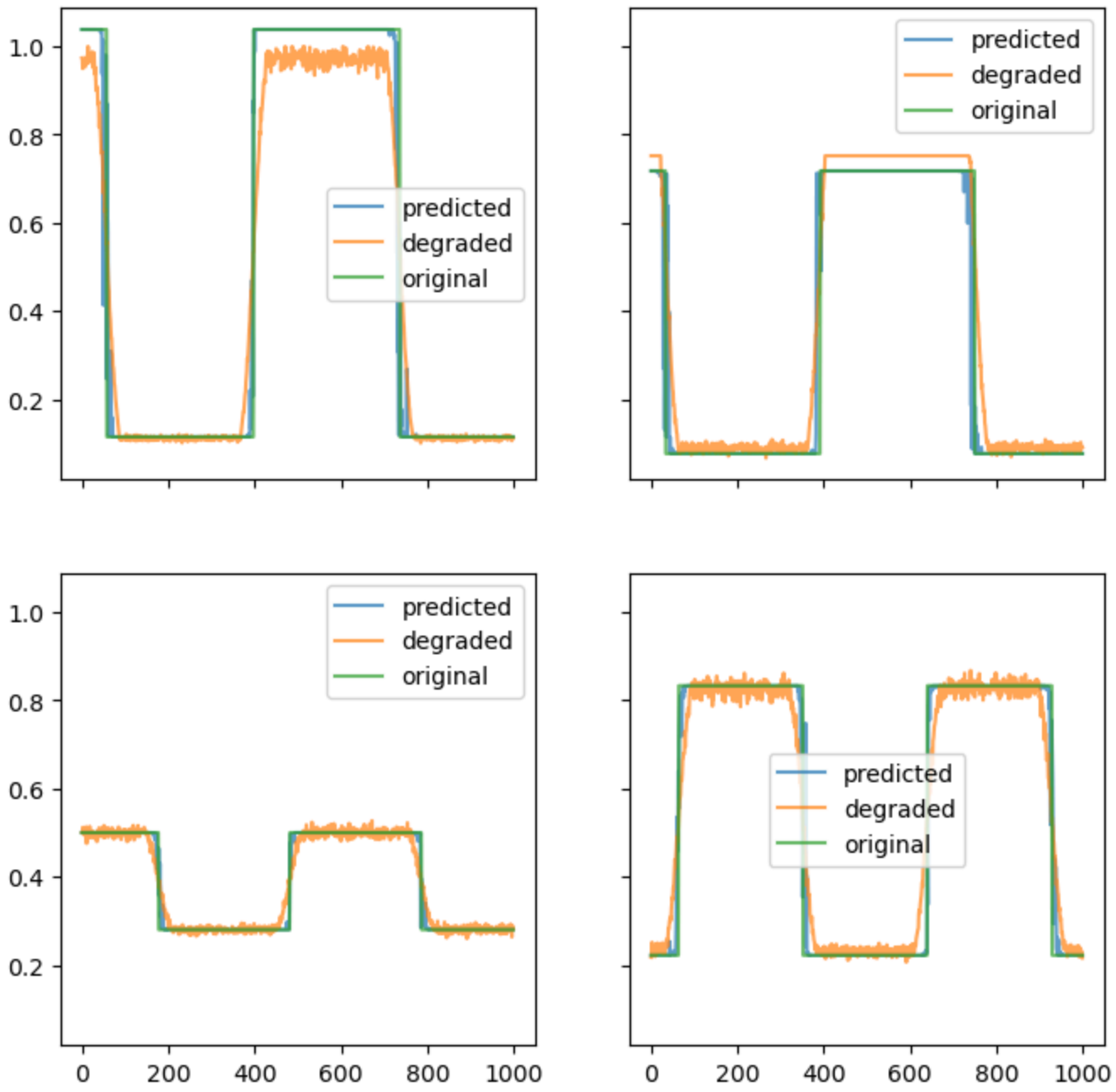
- fixes the problem model fits data well

modifying x offset, y intercept, period and amplitude

```
test_model(X=X, y=y, hidden_size=256, epochs = 200, lr = 0.005, seed_model = 42, X_
```

```
Data preppared
losses
Epoch 199, Loss: 0.0021223679650574923
Model trained
Done!
```


0.4
0.3
0.2
0.1
0.0
Loss
0
25
50
75
100
125
150
175
200
Epoch



Conclusion

A MLP was sucessfully built which can deconvolve step functions. Future steps include blending this with the richardson lucy algorithm and then testing this on a broader range of functions. Prehaps building this code into a class would be more managable.