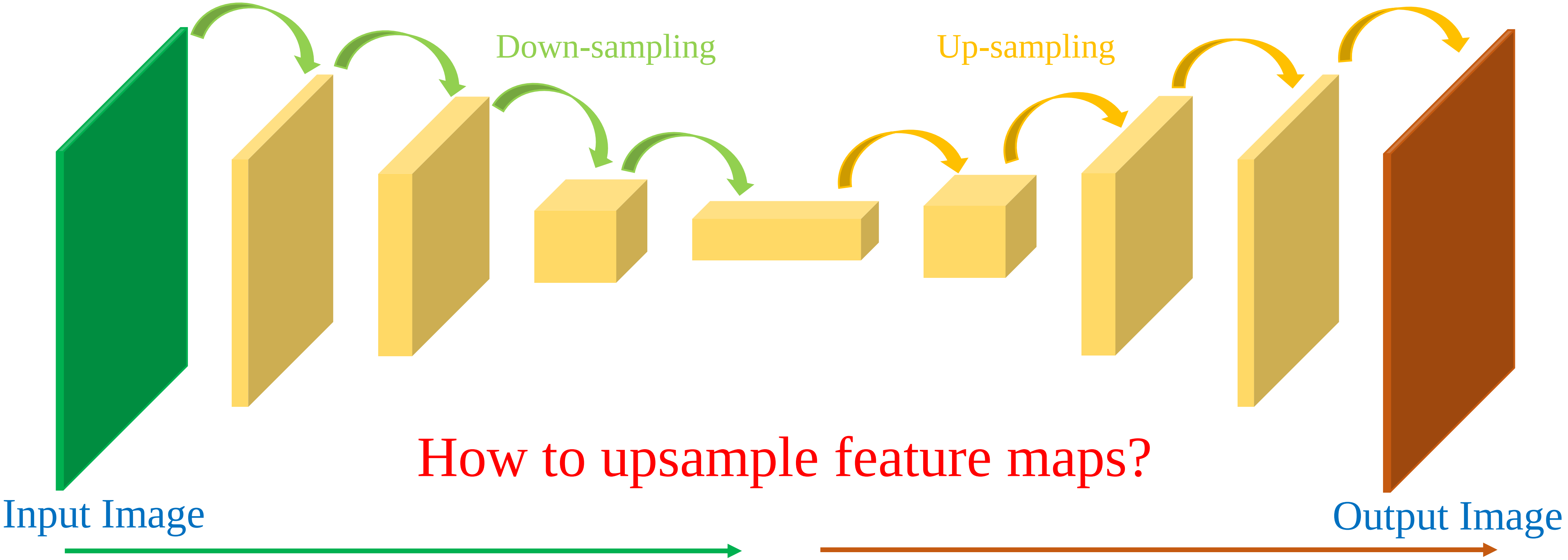
AI VIETNAM
AI Insight Course

Image Domain Conversion

Quang-Vinh Dinh
Ph.D. in Computer Science

Year 2021

Motivation

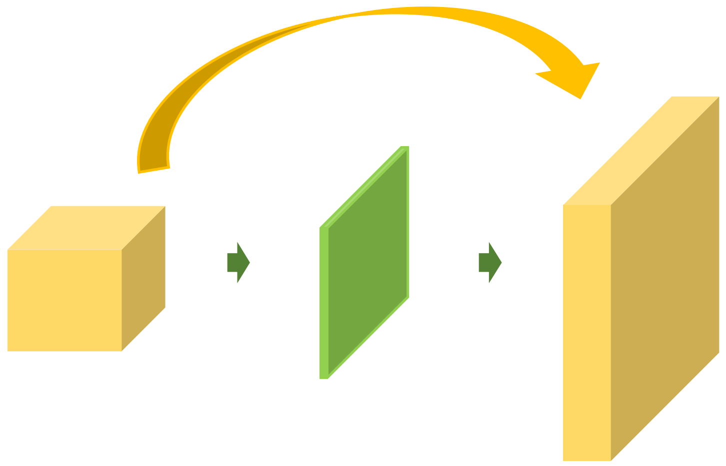


How to upsample feature maps?

Encoding phase	Decoding phase
#channels increases	#channels decreases
resolution decreases	resolution increases

How to Upsample Feature Maps

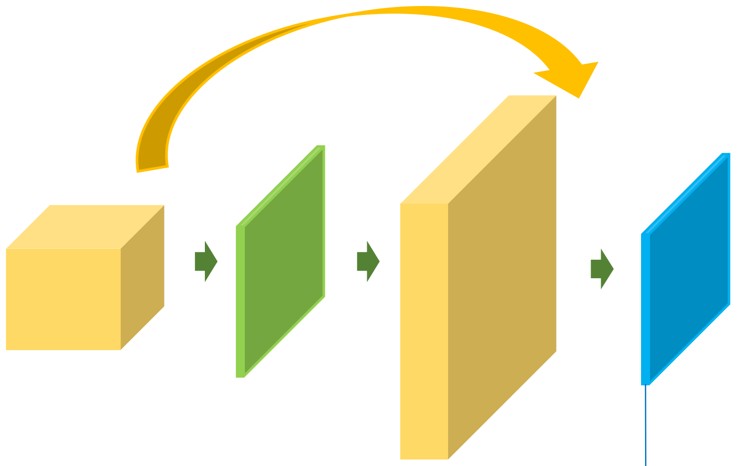
Naïve approach: Only use 'image upsampling'



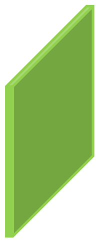
Output feature maps are lack of details

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.UpSampling2D(interpolation='bilinear'))
model.add(tf.keras.layers.Conv2D(num_filters,
                                 kernel_size,
                                 padding='same',
                                 kernel_initializer=initializer))
```

Use 'image upsampling'+Conv



Reduce the weakness from upsampling

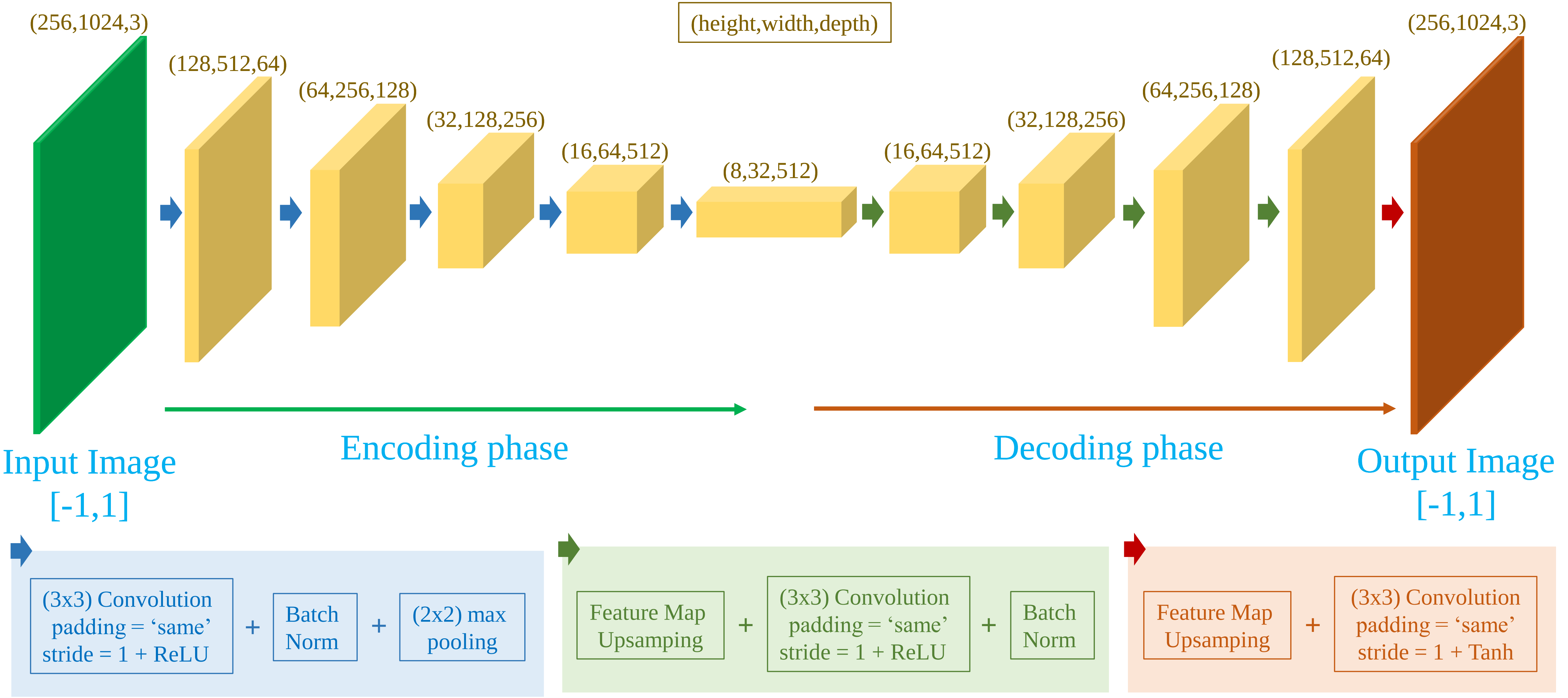


upsampling

conv
padding='same'
stride=1

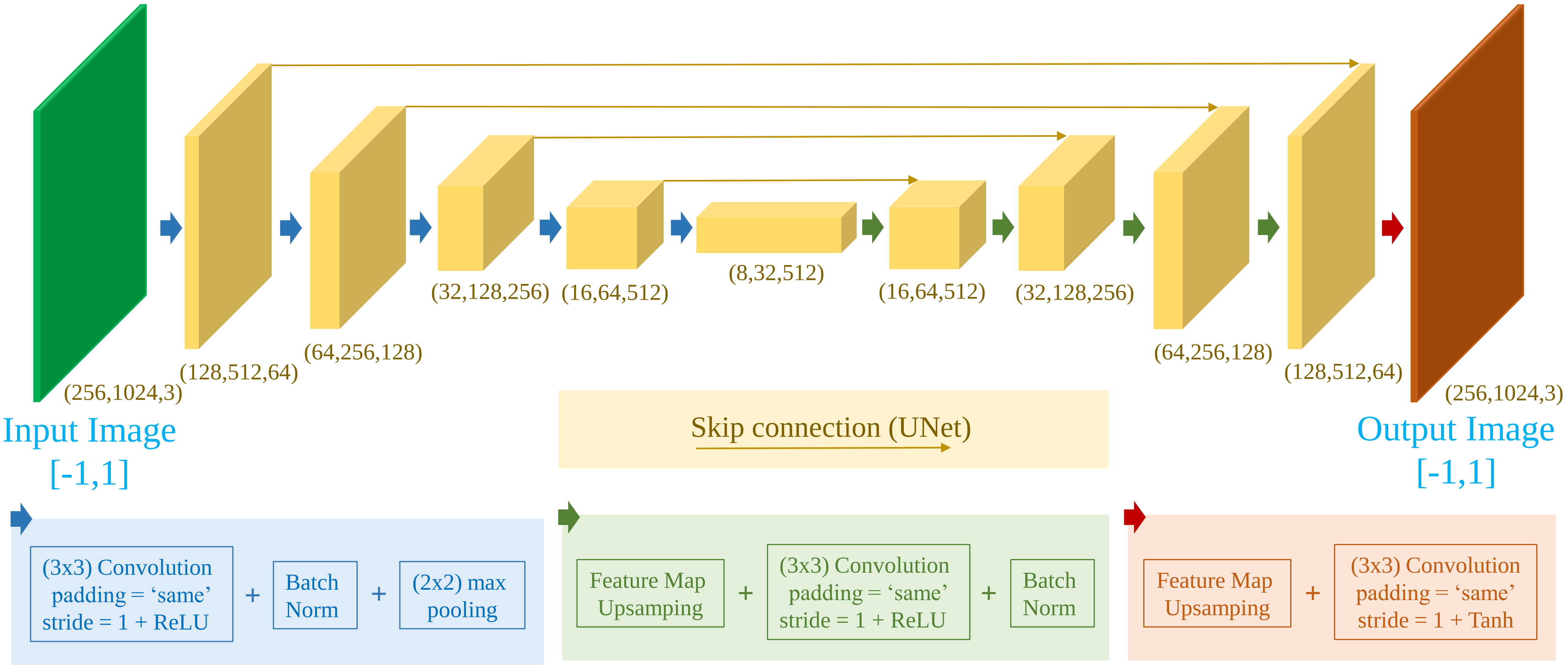
AI VIETNAM
AI Insight Course

How to Upsample Feature Maps



AI VIETNAM
AI Insight Course

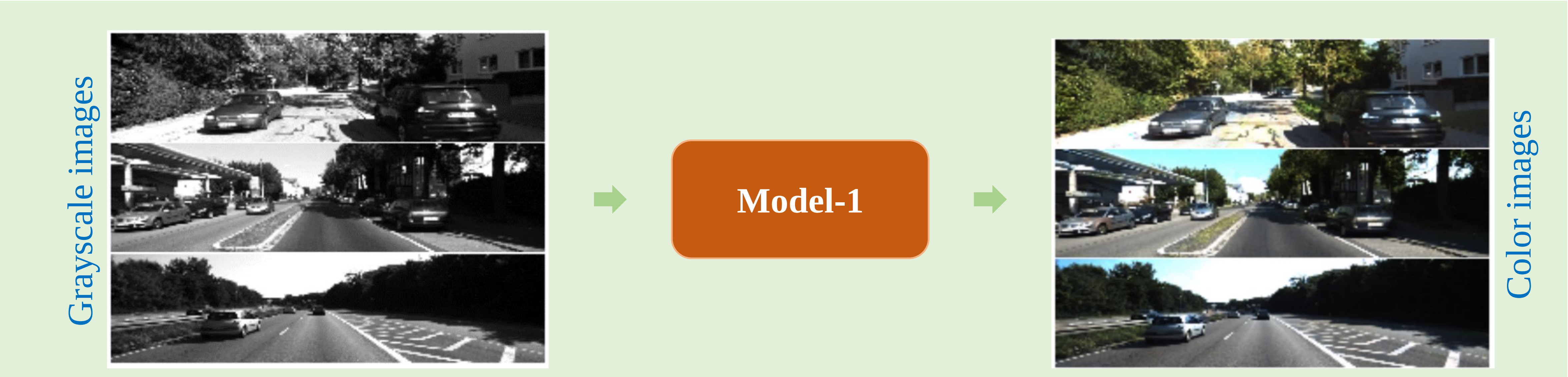
How to Upsample Feature Maps



Applications

❖ **Colorization**

1	Data preparation
2	Network construction
3	Loss and optimizer
4	Training



AI VIETNAM
AI Insight Course

Applications

1 Data preparation

❖ Colorization

```
PATH = '/content/gdrive/My Drive/data/'
IMG_WIDTH    = 1024
IMG_HEIGHT   = 256

def load(image_file):
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)

    # resize
    image = tf.image.resize(image, (IMG_HEIGHT, IMG_WIDTH))

    # copy
    gray = tf.identity(image)
    color= tf.identity(image)

    # convert to gray
    gray = tf.image.rgb_to_grayscale(gray)

    gray  = tf.cast(gray, tf.float32)
    color = tf.cast(color, tf.float32)

    return gray, color
```

```
gray, color = load(PATH+'unet/kitti_train/000008_10.png')

# Show the image
plt.figure()
plt.axis('off')
plt.imshow(gray[:,:,0]/255.0, cmap='gray')

plt.figure()
plt.axis('off')
plt.imshow(color/255.0)
```

```
<matplotlib.image.AxesImage at 0x7f7f0324e550>
```

Applications

❖ Colorization

```
BUFFER_SIZE = 50
BATCH_SIZE  = 1

# train_dataset
train_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_train/0000*_10.png')
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)

# test_dataset
test_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_test/0000*_10.png')
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(1)
```

```
# normalizing the images to [-1, 1]
def normalize(gray, color):
    gray  = (gray / 127.5) - 1
    color = (color / 127.5) - 1

    return gray, color

def random_jitter(gray, color):
    if tf.random.uniform(()) > 0.5:
        # random mirroring
        gray  = tf.image.flip_left_right(gray)
        color = tf.image.flip_left_right(color)

    return gray, color

def load_image_train(image_file):
    gray, color = load(image_file)
    gray, color = random_jitter(gray, color)
    gray, color = normalize(gray, color)

    return gray, color

def load_image_test(image_file):
    gray, color = load(image_file)
    gray, color = normalize(gray, color)

    return gray, color
```

Applications

❖ Network construction

```python
def UNet_process(x): # (1, 256, 1024, 3)
  # encoding
  down_stack = [
    downsample(64, 4),  # (bs, 128, 512, 64)
    downsample(256, 4), # (bs, 64, 256, 256)
    downsample(512, 4), # (bs, 32, 128, 512)
    downsample(512, 4), # (bs, 16, 64, 512)
    downsample(512, 4), # (bs, 8, 32, 512)
  ]

  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])
```

```python
# decoding
up_stack = [
  upsample(512, 4), # (bs, 16, 64, 512)
  upsample(512, 4), # (bs, 32, 128, 512)
  upsample(256, 4), # (bs, 64, 256, 256)
  upsample(64, 4),  # (bs, 128, 512, 64)
]

concat = tf.keras.layers.Concatenate()
for up, skip in zip(up_stack, skips):
  x = up(x)
  x = concat([x, skip])


# last layer
OUTPUT_CHANNELS = 3
last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS,
                                       4, strides=2,
                                       padding='same',
                                       activation='tanh')

x = last(x)

return x
```

Applications

❖ Colorization

3	Loss and optimizer

```
# loss function
def compute_loss(img1, img2):
  return tf.reduce_mean(tf.abs(img1-img2))

#optimizer
optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
```

Applications

❖ Colorization

4 Training

```
@tf.function
def train_step(gray, color):
    with tf.GradientTape() as gen_tape:
        # output
        fake_color = generator([gray], training=True)
        loss = compute_loss(fake_color, color)

    gradients = gen_tape.gradient(loss, generator.trainable_variables)
    optimizer.apply_gradients(zip(gradients, generator.trainable_variables))

    return loss

def fit(train_ds, epochs, test_ds):
    for epoch in range(epochs):
        # Train
        for gray, color in train_ds:
            loss = train_step(gray, color)
```

Applications

❖ Colorization

5	For debugging

```
def generate_images(model, gray, real):
    fake = model([gray], training=True)
    plt.figure(figsize=(15,20))

    display_list = [gray[0,:,:,0], real[0], fake[0]]
    title = ['Input Left', 'Real Left', 'Predicted Left']

    for i in range(3):
        plt.subplot(1, 3, i+1)
        plt.title(title[i])
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.show()
```

```
def evaluate(model, epoch):
    psnr_mean = 0.0
    count = 0
    for gray, real in test_dataset:
        fake = model([gray], training=True)

        psnr = tf.image.psnr(fake*0.5 + 0.5,
                             real*0.5 + 0.5,
                             max_val=1.0)
        psnr = tf.math.reduce_mean(psnr)

        psnr_mean += psnr
        count =count + 1

    psnr_mean = psnr_mean/count
    return psnr_mean
```

Applications

❖ Colorization

4	Training

22.3.UNet-Colorization_v1_showGrayImage.ipynb

```python
@tf.function
def train_step(gray, color):
    with tf.GradientTape() as gen_tape:
        # output
        fake_color = generator([gray], training=True)
        loss = compute_loss(fake_color, color)

    gradients = gen_tape.gradient(loss, generator.trainable_variables)
    optimizer.apply_gradients(zip(gradients, generator.trainable_variables))

    return loss

def fit(train_ds, epochs, test_ds):
    best_pnsr = 0.0
    for epoch in range(epochs):
        # Train
        for gray, color in train_ds:
            loss = train_step(gray, color)

        # for debug
        pnsr = evaluate(generator, epoch)
        if best_pnsr < pnsr:
            best_pnsr = pnsr
            print(best_pnsr)

            for gray, color in test_ds.take(1):
                generate_images(generator, gray, color)
```

Applications

❖ **Colorization**

5	For debugging

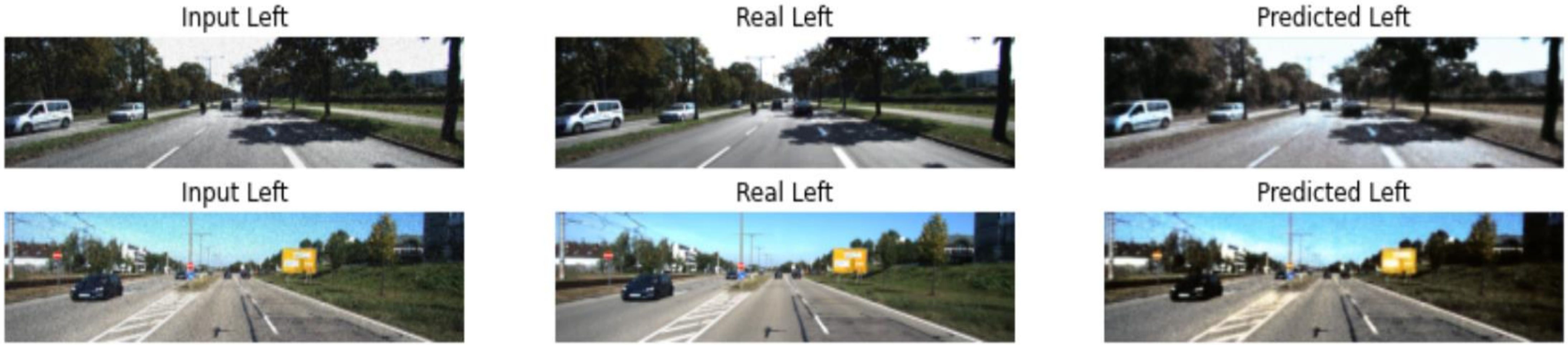
22.3.UNet-Colorization_v3.ipynb

```
def generate_images(model, gray, real):
    fake = model([gray], training=True)
    plt.figure(figsize=(15,20))

    display_list = [gray[0,:,:,0], real[0], fake[0]]
    title = ['Input Left', 'Real Left', 'Predicted Left']

    i = 0
    plt.subplot(1, 3, i+1)
    plt.title(title[i])
    plt.imshow(display_list[i]*0.5 + 0.5, cmap='gray')
    plt.axis('off')

    for i in range(1,3):
        plt.subplot(1, 3, i+1)
        plt.title(title[i])
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.show()
```

Applications

❖ Denoising

1	Data preparation
2	Network construction
3	Loss and optimizer
4	Training



Applications

❖ Denoising

```
def load(image_file):
    target = tf.io.read_file(image_file)
    target = tf.image.decode_jpeg(target)

    target = tf.image.resize(target, (IMG_HEIGHT, IMG_WIDTH))

    # make input
    noise = tf.identity(target)

    noise_level = 30.0
    n = tf.random.normal((IMG_HEIGHT, IMG_WIDTH, 1))*noise_level

    noise = noise + n
    noise = tf.clip_by_value(noise, clip_value_min=0, clip_value_max=255)

    return noise, target
```

Applications

❖ **Denoising**

1	Data preparation

```
noise, target = load(PATH+'unet/kitti_train/000008_10.png')

# casting to int for matplotlib to show the image
plt.figure()
plt.axis('off')
plt.imshow(noise/255.0)

plt.figure()
plt.axis('off')
plt.imshow(target/255.0)
```

```
<matplotlib.image.AxesImage at 0x7f70c0b5d910>
```

Applications

❖ Denoising

```
BUFFER_SIZE = 50
BATCH_SIZE  = 1

# train_dataset
train_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_train/0000*_10.png')
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)

# test_dataset
test_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_test/0000*_10.png')
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(1)
```

Applications

❖ **Denoising**

```python
# normalizing the images to [-1, 1]
def normalize(input_img, target_img):
    input_img  = (input_img / 127.5) - 1
    target_img = (target_img / 127.5) - 1

    return input_img, target_img

def random_jitter(input_img, target_img):
    if tf.random.uniform(()) > 0.5:
        # random mirroring
        input_img  = tf.image.flip_left_right(input_img)
        target_img = tf.image.flip_left_right(target_img)

    return input_img, target_img

def load_image_train(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = random_jitter(input_img, target_img)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img

def load_image_test(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img
```

Applications

❖ Denoising

```python
def UNet_process(x): # (1, 256, 1024, 3)
  # encoding
  down_stack = [
    downsample(64, 4),  # (bs, 128, 512, 64)
    downsample(256, 4), # (bs, 64, 256, 256)
    downsample(512, 4), # (bs, 32, 128, 512)
    downsample(512, 4), # (bs, 16, 64, 512)
    downsample(512, 4), # (bs, 8, 32, 512)
  ]

  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])
```

```python
# decoding
up_stack = [
  upsample(512, 4), # (bs, 16, 64, 512)
  upsample(512, 4), # (bs, 32, 128, 512)
  upsample(256, 4), # (bs, 64, 256, 256)
  upsample(64, 4),  # (bs, 128, 512, 64)
]

concat = tf.keras.layers.Concatenate()
for up, skip in zip(up_stack, skips):
  x = up(x)
  x = concat([x, skip])


# last layer
OUTPUT_CHANNELS = 3
last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS,
                                       4, strides=2,
                                       padding='same',
                                       activation='tanh')

x = last(x)

return x
```

Applications

❖ Denoising

3	Loss and optimizer

```
# loss function
def compute_loss(img1, img2):
  return tf.reduce_mean(tf.abs(img1-img2))

#optimizer
optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
```

Applications

❖ **Denoising**

4	Training

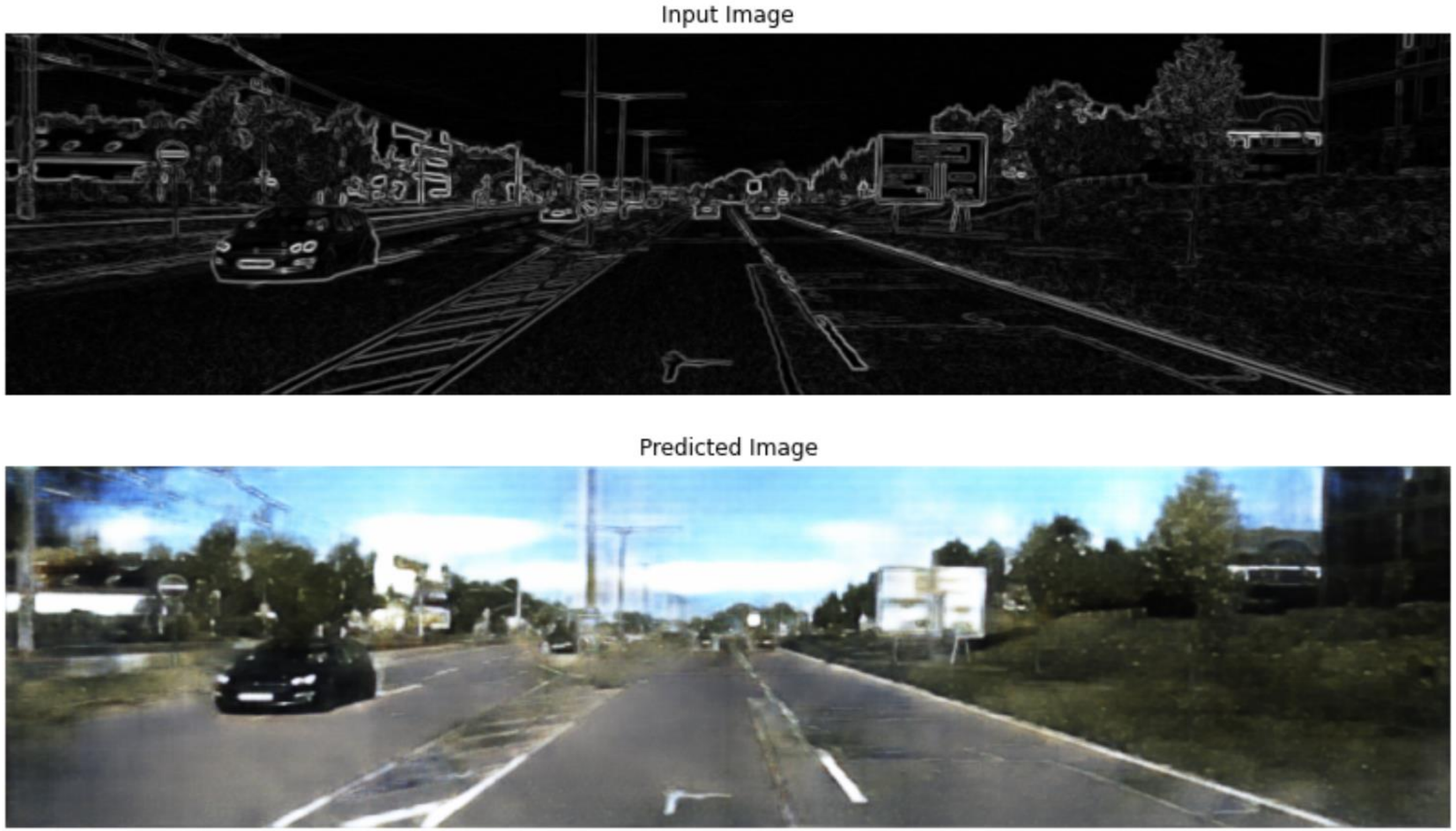
```
@tf.function
def train_step(gray, color):
    with tf.GradientTape() as gen_tape:
        fake_color = generator([gray], training=True)
        loss = compute_loss(fake_color, color)

    gradients = gen_tape.gradient(loss, generator.trainable_variables)
    optimizer.apply_gradients(zip(gradients, generator.trainable_variables))

    return loss

def fit(train_ds, epochs, test_ds):
    for epoch in range(epochs):
        for noise_img, clean_img in train_ds:
            loss = train_step(noise_img, clean_img)
```

23.2.UNet-Denoising-L1.ipynb

Applications

❖ Edge2Scene

1	Data preparation
2	Network construction
3	Loss and optimizer
4	Training



Applications

- Edge2Scene

1	Data preparation

```python
def load(image_file):
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)

    # resize
    image = tf.image.resize(image, (IMG_HEIGHT, IMG_WIDTH))

    # copy
    gray = tf.identity(image)
    color= tf.identity(image)

    # convert to gray
    gray = tf.image.rgb_to_grayscale(gray)
    gray = tf.reshape(gray, (1, 256, 1024, 1))

    edges = tf.image.sobel_edges(gray)
    edges = edges**2

    edges = tf.math.reduce_sum(edges,axis=-1)
    edges = tf.sqrt(edges)

    edges = tf.cast(edges, tf.float32)
    color = tf.cast(color, tf.float32)

    return edges[0], color
```

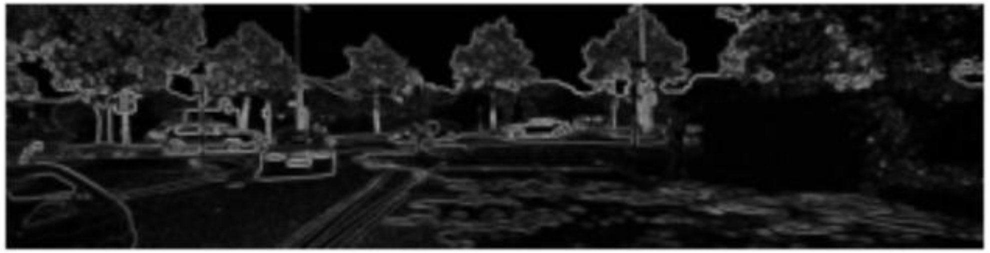
Applications

❖ **Edge2Scene**

1	Data preparation

```
edges, color = load(PATH+'unet/kitti_train/000008_10.png')

# Show the image
plt.figure()
plt.axis('off')
plt.imshow(edges[:,:,0]/255.0, cmap='gray')
plt.figure()
plt.axis('off')
plt.imshow(color/255.0)
```

<matplotlib.image.AxesImage at 0x7f58f019d590>



Applications

❖ Edge2Scene

```
BUFFER_SIZE = 50
BATCH_SIZE  = 1

# train_dataset
train_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_train/0000*_10.png')
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)

# test_dataset
test_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_test/0000*_10.png')
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(1)
```

Applications

❖ **Edge2Scene**

1 Data preparation

```
# normalizing the images to [-1, 1]
def normalize(input_img, target_img):
    input_img  = (input_img / 127.5) - 1
    target_img = (target_img / 127.5) - 1

    return input_img, target_img

def random_jitter(input_img, target_img):
    if tf.random.uniform(()) > 0.5:
        # random mirroring
        input_img  = tf.image.flip_left_right(input_img)
        target_img = tf.image.flip_left_right(target_img)

    return input_img, target_img

def load_image_train(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = random_jitter(input_img, target_img)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img

def load_image_test(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img
```

Applications

❖ Edge2Scene

```
def UNet_process(x): # (1, 256, 1024, 3)
  # encoding
  down_stack = [
    downsample(64, 4),  # (bs, 128, 512, 64)
    downsample(256, 4), # (bs, 64, 256, 256)
    downsample(512, 4), # (bs, 32, 128, 512)
    downsample(512, 4), # (bs, 16, 64, 512)
    downsample(512, 4), # (bs, 8, 32, 512)
  ]

  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])
```

```
# decoding
up_stack = [
  upsample(512, 4), # (bs, 16, 64, 512)
  upsample(512, 4), # (bs, 32, 128, 512)
  upsample(256, 4), # (bs, 64, 256, 256)
  upsample(64, 4),  # (bs, 128, 512, 64)
]

concat = tf.keras.layers.Concatenate()
for up, skip in zip(up_stack, skips):
  x = up(x)
  x = concat([x, skip])


# last layer
OUTPUT_CHANNELS = 3
last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS,
                                       4, strides=2,
                                       padding='same',
                                       activation='tanh')

x = last(x)

return x
```

Applications

❖ Edge2Scene

3	Loss and optimizer

```
# loss function
def compute_loss(img1, img2):
  return tf.reduce_mean(tf.abs(img1-img2))

#optimizer
optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
```

Applications

❖ **Edge2Scene**

4 | Training

```
@tf.function
def train_step(edge_img, scene_img):
    with tf.GradientTape() as gen_tape:
        fake_scene = generator([edge_img], training=True)
        loss = compute_loss(fake_scene, scene_img)

    gradients = gen_tape.gradient(loss, generator.trainable_variables)
    optimizer.apply_gradients(zip(gradients, generator.trainable_variables))

    return loss

def fit(train_ds, epochs, test_ds):
    best_pnsr = 0.0
    for epoch in range(epochs):
        for edge_img, scene_img in train_ds:
            loss = train_step(edge_img, scene_img)
```

23.3.UNet-Edge2Scene-L1.ipynb

Applications

❖ **Super-resolution**

1	Data preparation
2	Network construction
3	Loss and optimizer
4	Training

Applications

❖ Super-resolution

1 Data preparation

```
def load(image_file):
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)

    image = tf.image.resize(image, (IMG_HEIGHT, IMG_WIDTH))

    # make input
    lr = tf.identity(image)
    lr = tf.image.resize(lr, (IMG_HEIGHT//4, IMG_WIDTH//4))

    return lr, image
```

Applications

❖ **Super-resolution**

1	Data preparation

```python
lr, image = load(PATH+'unet/kitti_train/000008_10.png')
print(lr.shape)
print(image.shape)

# casting to int for matplotlib to show the image
plt.figure()
plt.axis('off')
plt.imshow(lr/255.0)
plt.figure()
plt.axis('off')
plt.imshow(image/255.0)
```

```
(64, 256, 3)
(256, 1024, 3)
<matplotlib.image.AxesImage at 0x7fd2202f77d0>
```

Applications

❖ Super-resolution

```
BUFFER_SIZE = 50
BATCH_SIZE  = 1

# train_dataset
train_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_train/0000*_10.png')
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)

# test_dataset
test_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_test/0000*_10.png')
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(1)
```

Applications

❖ **Super-resolution**

1	Data preparation

```python
# normalizing the images to [-1, 1]
def normalize(input_img, target_img):
    input_img  = (input_img / 127.5) - 1
    target_img = (target_img / 127.5) - 1

    return input_img, target_img

def random_jitter(input_img, target_img):
    if tf.random.uniform(()) > 0.5:
        # random mirroring
        input_img  = tf.image.flip_left_right(input_img)
        target_img = tf.image.flip_left_right(target_img)

    return input_img, target_img

def load_image_train(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = random_jitter(input_img, target_img)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img

def load_image_test(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img
```

Applications

❖ Super-resolution

```python
def UNet_process(x): # (1, 256, 1024, 3)
  # encoding
  down_stack = [
    downsample(64, 4),  # (bs, 128, 512, 64)
    downsample(256, 4), # (bs, 64, 256, 256)
    downsample(512, 4), # (bs, 32, 128, 512)
    downsample(512, 4), # (bs, 16, 64, 512)
    downsample(512, 4), # (bs, 8, 32, 512)
  ]

  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])
```

```python
# decoding
up_stack = [
  upsample(512, 4), # (bs, 16, 64, 512)
  upsample(512, 4), # (bs, 32, 128, 512)
  upsample(256, 4), # (bs, 64, 256, 256)
  upsample(64, 4),  # (bs, 128, 512, 64)
]

concat = tf.keras.layers.Concatenate()
for up, skip in zip(up_stack, skips):
  x = up(x)
  x = concat([x, skip])


# last layer
OUTPUT_CHANNELS = 3
last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS,
                                       4, strides=2,
                                       padding='same',
                                       activation='tanh')

x = last(x)

return x
```

Applications

❖ Super-resolution

3	Loss and optimizer

```
# loss function
def compute_loss(img1, img2):
  return tf.reduce_mean(tf.abs(img1-img2))

#optimizer
optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
```

Applications

❖ **Super-resolution**

4 Training

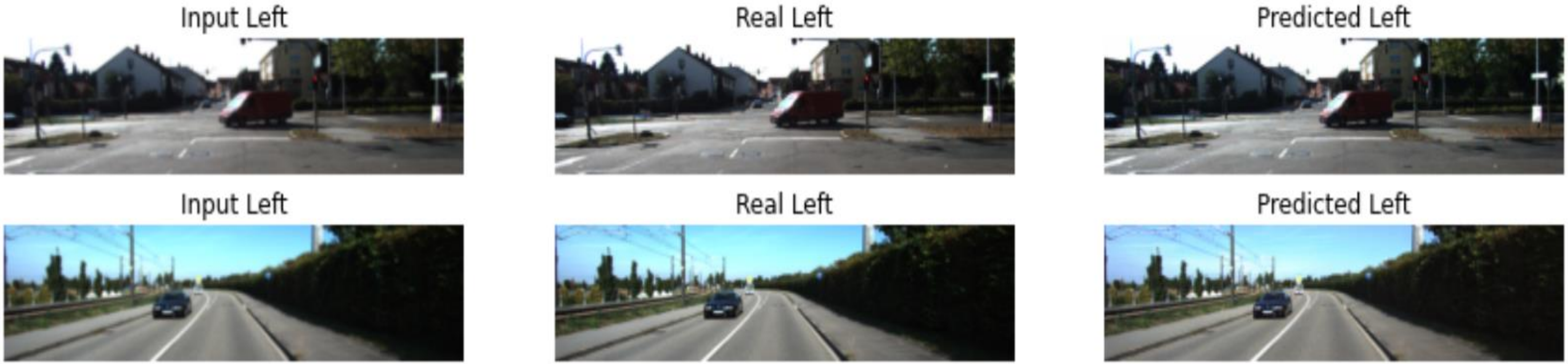
```
@tf.function
def train_step(lr_img, hr_img):
    with tf.GradientTape() as gen_tape:
        fake_hr = generator([lr_img], training=True)
        loss = compute_loss(fake_hr, hr_img)

    gradients = gen_tape.gradient(loss, generator.trainable_variables)
    optimizer.apply_gradients(zip(gradients, generator.trainable_variables))

    return loss

def fit(train_ds, epochs, test_ds):
    for epoch in range(epochs):
        for lr_img, hr_img in train_ds:
            loss = train_step(lr_img, hr_img)
```

23.4.UNet-Super-resolution-L1.ipynb

Applications

1	Data preparation
2	Network construction
3	Loss and optimizer
4	Training

❖ **Deblur**



Applications

❖ **Deblur**

1 Data preparation

```
def load(image_file):
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)

    image = tf.image.resize(image, (IMG_HEIGHT, IMG_WIDTH))

    # make input
    blur = tf.identity(image)

    blur = tf.reshape(blur, (1, 256, 1024, 3))

    blur = tfa.image.mean_filter2d(blur)
    blur = tfa.image.mean_filter2d(blur)
    blur = tfa.image.mean_filter2d(blur)

    return blur[0], image
```

Applications

❖ Deblur

1	Data preparation

```
blur, target = load(PATH+'unet/kitti_train/000008_10.png')

# Show the image
plt.figure()
plt.axis('off')
plt.imshow(blur/255.0)

plt.figure()
plt.axis('off')
plt.imshow(target/255.0)
```

```
<matplotlib.image.AxesImage at 0x7f85b437f2d0>
```

Applications

❖ Deblur

```
BUFFER_SIZE = 50
BATCH_SIZE  = 1

# train_dataset
train_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_train/0000*_10.png')
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)

# test_dataset
test_dataset = tf.data.Dataset.list_files(PATH+'unet/kitti_test/0000*_10.png')
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(1)
```

Applications

❖ **Deblur**

1	Data preparation

```python
# normalizing the images to [-1, 1]
def normalize(input_img, target_img):
    input_img  = (input_img / 127.5) - 1
    target_img = (target_img / 127.5) - 1

    return input_img, target_img

def random_jitter(input_img, target_img):
    if tf.random.uniform(()) > 0.5:
        # random mirroring
        input_img  = tf.image.flip_left_right(input_img)
        target_img = tf.image.flip_left_right(target_img)

    return input_img, target_img

def load_image_train(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = random_jitter(input_img, target_img)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img

def load_image_test(image_file):
    input_img, target_img = load(image_file)
    input_img, target_img = normalize(input_img, target_img)

    return input_img, target_img
```

Applications

❖ Deblur

```
def UNet_process(x): # (1, 256, 1024, 3)
  # encoding
  down_stack = [
    downsample(64, 4),  # (bs, 128, 512, 64)
    downsample(256, 4), # (bs, 64, 256, 256)
    downsample(512, 4), # (bs, 32, 128, 512)
    downsample(512, 4), # (bs, 16, 64, 512)
    downsample(512, 4), # (bs, 8, 32, 512)
  ]

  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])
```

```
# decoding
up_stack = [
  upsample(512, 4), # (bs, 16, 64, 512)
  upsample(512, 4), # (bs, 32, 128, 512)
  upsample(256, 4), # (bs, 64, 256, 256)
  upsample(64, 4),  # (bs, 128, 512, 64)
]

concat = tf.keras.layers.Concatenate()
for up, skip in zip(up_stack, skips):
  x = up(x)
  x = concat([x, skip])


# last layer
OUTPUT_CHANNELS = 3
last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS,
                                       4, strides=2,
                                       padding='same',
                                       activation='tanh')

x = last(x)

return x
```

Applications

❖ Deblur

3	Loss and optimizer

```
# loss function
def compute_loss(img1, img2):
  return tf.reduce_mean(tf.abs(img1-img2))

#optimizer
optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
```

Applications

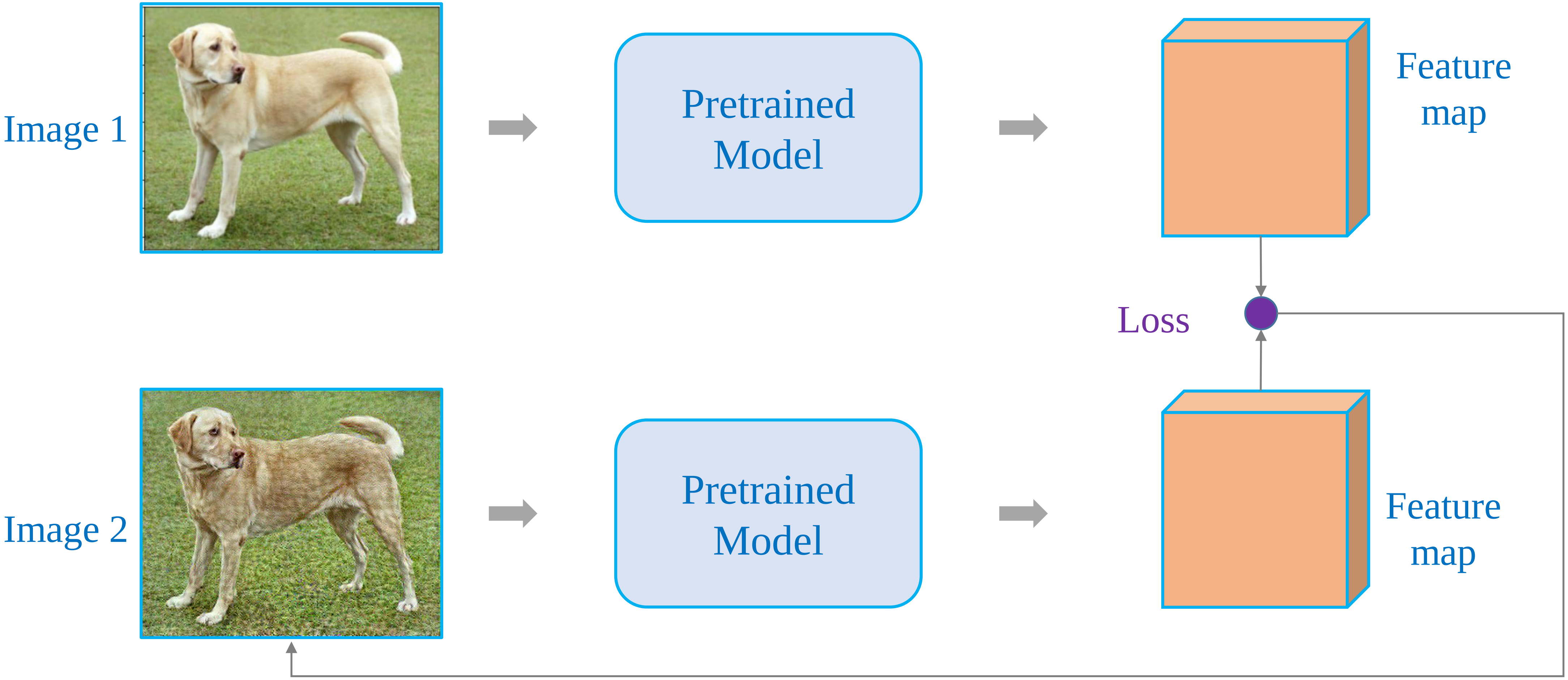
❖ **Deblur**

4 Training

```
@tf.function
def train_step(blur_img, delur_img):
    with tf.GradientTape() as gen_tape:
        fake_delur = generator([blur_img], training=True)
        loss = compute_loss(fake_delur, delur_img)

    gradients = gen_tape.gradient(loss, generator.trainable_variables)
    optimizer.apply_gradients(zip(gradients, generator.trainable_variables))

    return loss

def fit(train_ds, epochs, test_ds):
    best_pnsr = 0.0
    for epoch in range(epochs):
        for blur_img, delur_img in train_ds:
           loss = train_step(blur_img, delur_img)
```

23.5.UNet-Deblur_v1.ipynb

Perceptual Loss



Perceptual Loss

❖ Code

```
from tensorflow.python.keras.applications.vgg19 import VGG19

def get_vgg(output_layer):
    vgg = VGG19(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3),
                include_top=False, weights='imagenet')
    return tf.keras.Model(vgg.input, vgg.layers[20].output)

vgg = get_vgg()
```

23.6.UNet-Colorization-PerceptualLoss.ipynb

```
img1 = tf.random.normal((1, IMG_HEIGHT, IMG_WIDTH, 3))
img2 = tf.random.normal((1, IMG_HEIGHT, IMG_WIDTH, 3))

img1_fea = vgg(img1)
img2_fea = vgg(img2)

loss = mean_squared_error(img1_fea, img2_fea)
```

Perceptual Loss

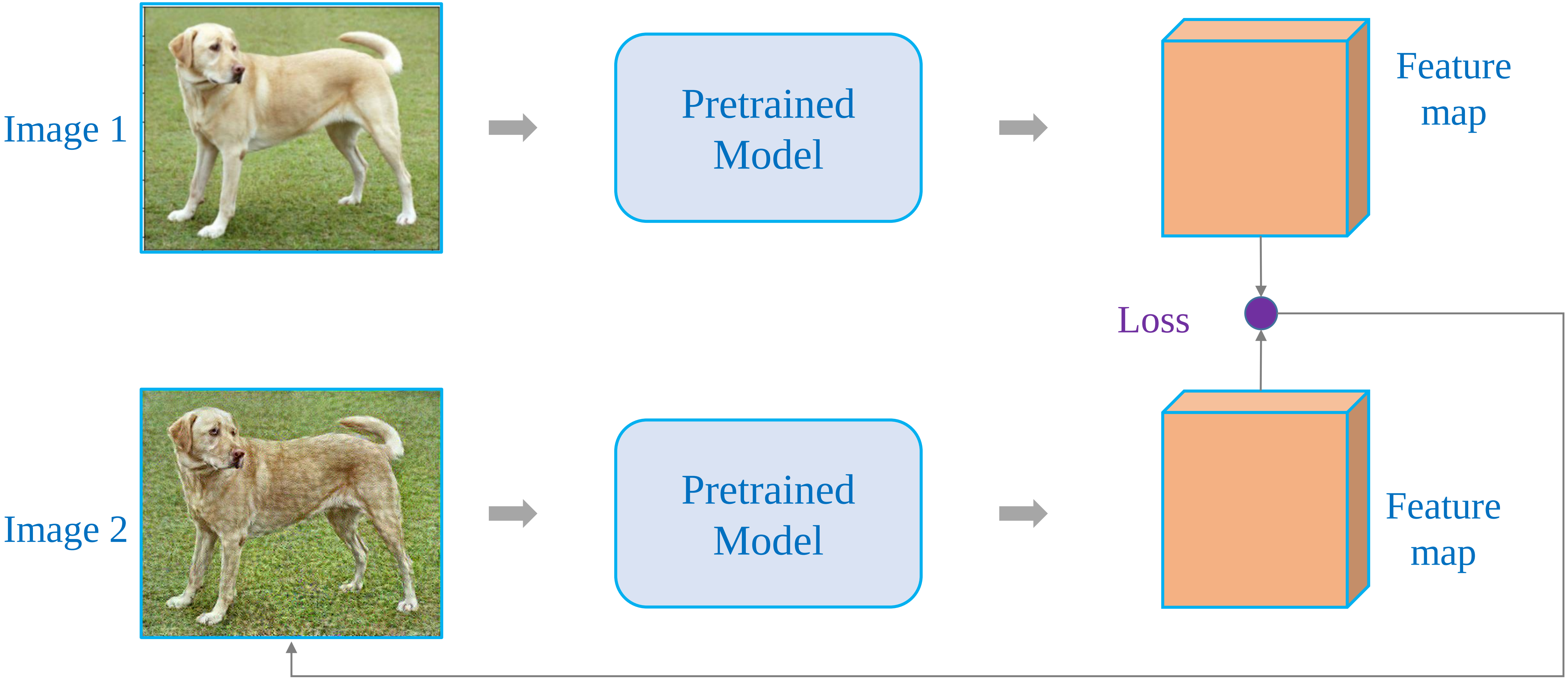
❖ Code

```
from tensorflow.python.keras.applications.vgg19 import VGG19

def get_vgg(output_layer):
    vgg = VGG19(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3),
                include_top=False, weights='imagenet')
    return tf.keras.Model(vgg.input, vgg.layers[20].output)

vgg = get_vgg()
```

```
mean_squared_error = tf.keras.losses.MeanSquaredError()
def compute_perceptual_loss(img1, img2):
  if img1.shape[3]==1:
      img1 = tf.concat([img1, img1, img1], 3)
  if img2.shape[3]==1:
      img2 = tf.concat([img2, img2, img2], 3)
  img1_fea = vgg(img1)
  img2_fea = vgg(img2)

  loss = mean_squared_error(img1_fea, img2_fea)
  return loss
```

23.6.UNet-Colorization-PerceptualLoss.ipynb

Perceptual Loss + L1



Perceptual Loss + L1

❖ Code

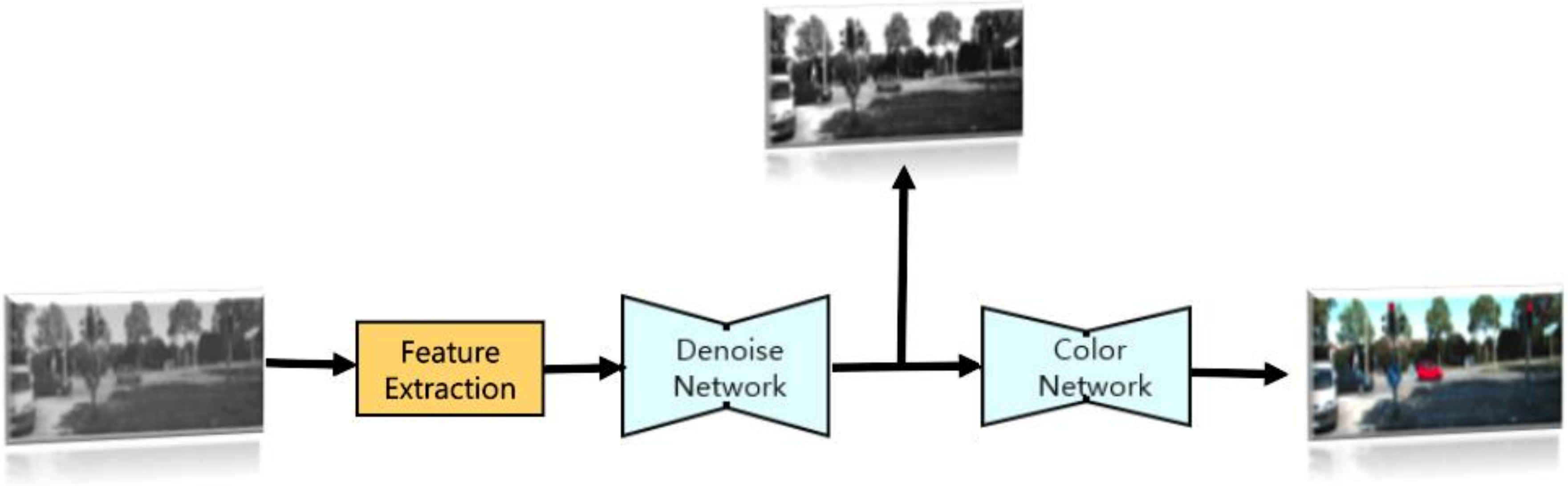
```
mean_squared_error = tf.keras.losses.MeanSquaredError()
def compute_perceptual_loss(img1, img2):
  if img1.shape[3]==1:
      img1 = tf.concat([img1, img1, img1], 3)
  if img2.shape[3]==1:
      img2 = tf.concat([img2, img2, img2], 3)
  img1_fea = vgg(img1)
  img2_fea = vgg(img2)

  loss = mean_squared_error(img1_fea, img2_fea)
  return loss
```

```
def compute_loss_l1(img1, img2):
  return tf.reduce_mean(tf.abs(img1-img2))
```

```
def compute_loss(img1, img2):
  return compute_loss_l1(img1, img2) + compute_perceptual_loss(img1, img2)*100.0
```

Denoising+Colorization

❖ **Complex domain conversion**

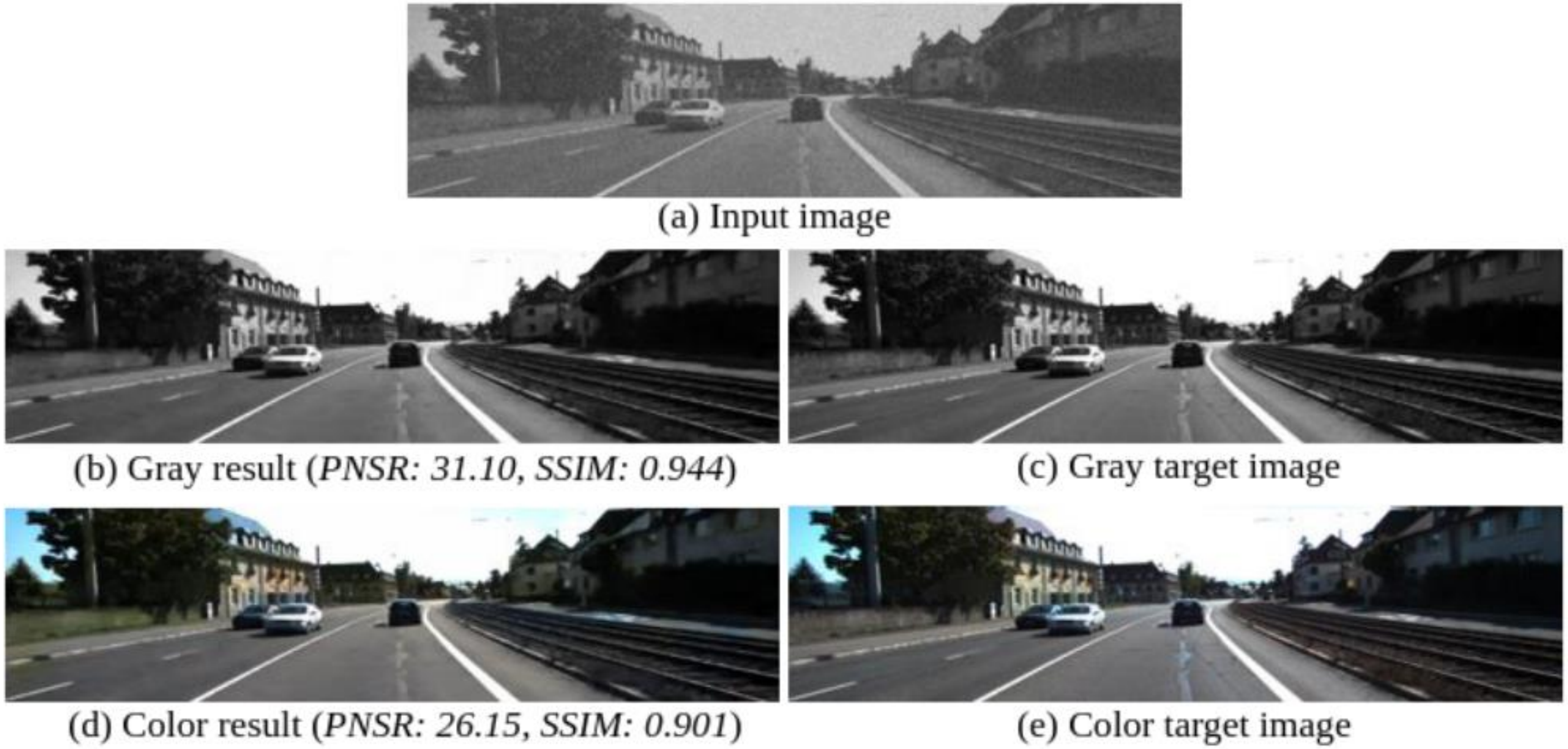


Denoising+Colorization

❖ Loss function

Experimental Results

❖ Qualitative result



(a) Input image

(b) Gray result (*PNSR: 31.10, SSIM: 0.944*)

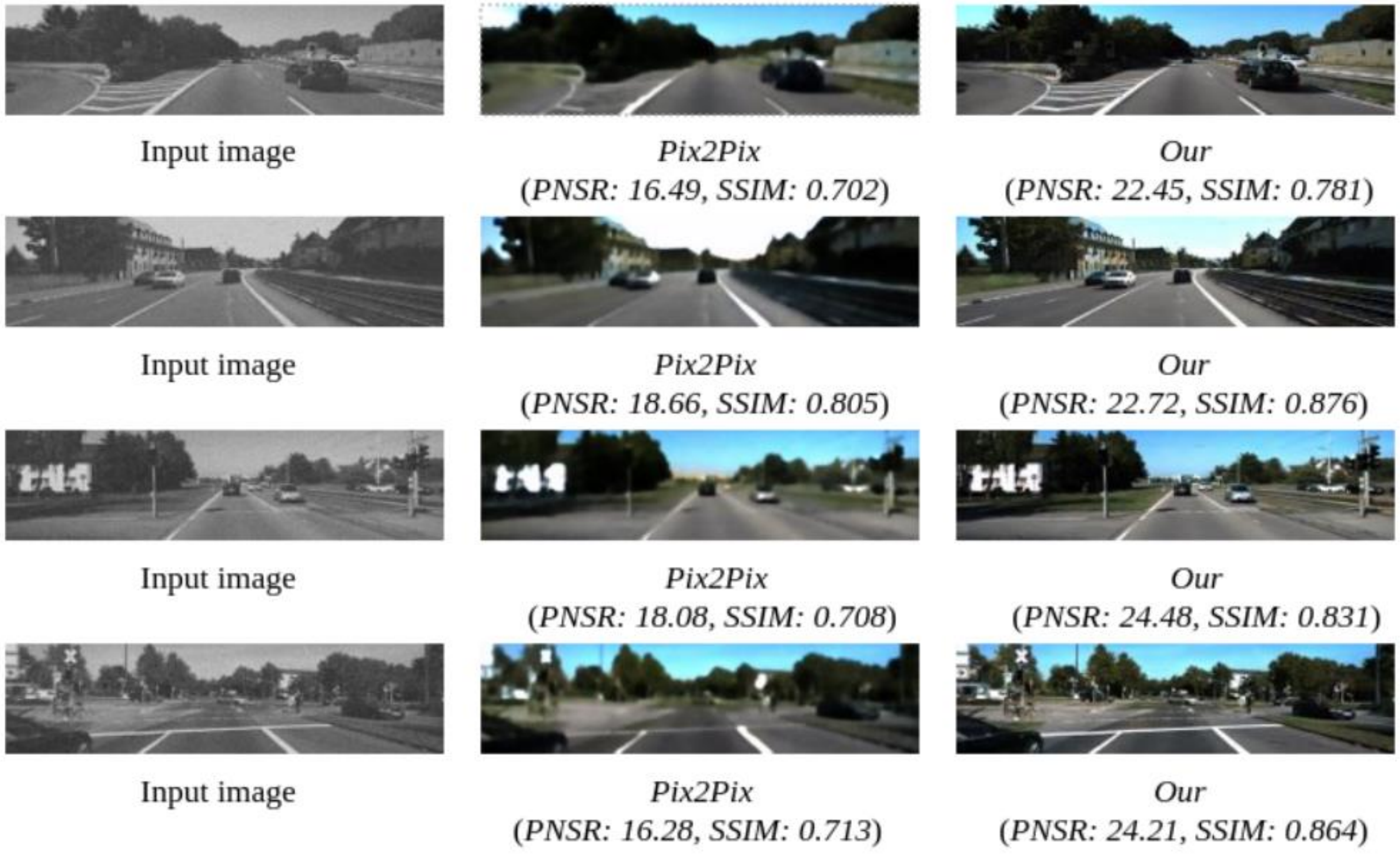
(c) Gray target image

(d) Color result (*PNSR: 26.15, SSIM: 0.901*)

(e) Color target image

Experimental Results

❖ Compare with a baseline method



Ablation Studies

❖ **Different losses**

(a) Target image

(b) Loss1 (*PNSR: 26.51*, *SSIM: 0.789*)

(c) Loss2 (*PNSR: 25.53*, *SSIM: 0.747*)

(d) Loss3 (*PNSR: 24.91*, *SSIM: 0.706*)

(e) Loss4 (*PNSR: 23.11*, *SSIM: 0.716*)

(f) Loss5 (*PNSR: 22.31*, *SSIM: 0.706*)


Thank you!