Image Inpainting

148254 Aleksandra Krasicka

Problem

Requirements

Problem	Points
Semantic segmentation	1
Instance segmentation	3
Colourization	1
Depth estimation	1
Domain adaptation	3
Super-resolution	3
Image inpainting	3
Search engine	2
3D reconstruction	4

There can be no classification or regression as the primary problem, others are possible for discussion.

Selected Image inpainting.

Dataset

Minimal size of input image 200x200px.

Requirements

- At least 1000 photos
- not mnist

Additional points

- Evaluation on a set with at least 10000 photos +1pk
- Your own part of the dataset (> 500 photos) +1pk

Description

name: Animal Faces-HQ (AFHQ)

link: https://www.kaggle.com/datasets/andrewmvd/animal-faces

description:

This dataset, also known as Animal Faces-HQ (AFHQ), consists of 16,130 high-quality images at 512×512 resolution. There are three domains of classes, each providing about 5000 images. By having multiple (three) domains and diverse images of various breeds per each domain, AFHQ sets a challenging image-to-image translation problem. The classes are:

- Cat;
- Dog;
- Wildlife

For the model, only the **Cat** category was used.

Processing

Each image was scaled to size 256×256.

Then each image is processed to add a black square of size 100×100 and return a binary mask showing the square, the image with the square and the image with a blurred square.

```python
def apply_random_mask(img):
    mask_size = 100
    y1, x1 = np.random.randint(0, img.shape[0] - mask_size, 1)[0], np.random.randint(0, img.shape[1] - mask_size, 1)[0]
    y2, x2 = y1 + mask_size, x1 + mask_size

    binary_mask = np.zeros((img.shape[0], img.shape[1]), dtype=np.uint8)
    binary_mask[y1:y2, x1:x2] = 1

    masked_img = img.copy()
    masked_img[y1:y2, x1:x2] = 0

    blurred_img = cv2.GaussianBlur(img, (35, 35), 0)
    blurred_version = img.copy()
    blurred_version[y1:y2, x1:x2] = blurred_img[y1:y2, x1:x2]

    return binary_mask, masked_img, blurred_version
```

The images are later normalised.

Training

Requirements

- The correctly selected loss function
- Split data into train, validation and test set
- Metrics (at least 2)

Additional points

- Hyperparameter tuning or estimation +1pk
- Architecture tuning (at least 3 architecture) +1pk
- Overfitting some examples from the training set +1pk
- Data augmentation +1pk
- Cross-validation +1pk
- Distributed learning +1pk
- Federated learning +2pk
- Testing a few optimizers (at least 3) +1pk
- Testing various loss functions (at least 3) +1pk
- Calculating intrinsic dimension +1pk

```python
train_original, test_original, train_blurred, test_blurred, train_masks, test_masks = train_test_split(
    original_images, blurred_images, masked_images, test_size=0.2, random_state=42
)
```

Model

Requirements

1st model	Points
pre-trained model on the same problem	0
pre-trained model on the different problem (transfer-learning)	1
ready architecture trained from scratch	1
own architecture (over 50% of own layers)	2

Additional points:

- Each subsequent model with a different architecture +1pk, for next own architecture (over 50% of own layers) +2pk
- +1pk for a every non-trivial solution in own architecture (use of attention, GAN, RL, contrastive learning, metric learning)

GAN 1

Very bad, not in the final notebook on the GitHub.

```python
def build_generator(input_shape=(256, 256, 3)):

    inputs = layers.Input(shape=input_shape)

    d1 = layers.Conv2D(64, kernel_size=4, strides=2, padding="same", activation="relu")(inputs)
    d2 = layers.Conv2D(128, kernel_size=4, strides=2, padding="same", activation="relu")(d1)
    d3 = layers.Conv2D(256, kernel_size=4, strides=2, padding="same", activation="relu")(d2)
    d4 = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d3)
    d5 = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d4)
    d6 = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d5)
    d7 = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d6)

    bottleneck = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d7)

    u1 = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same", activation="relu")(bottleneck)
    u1 = layers.Dropout(0.5)(u1, training=True)
    u1 = layers.Concatenate()([u1, d7])
    u2 = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same", activation="relu")(u1)
    u2 = layers.Dropout(0.5)(u2, training=True)
    u2 = layers.Concatenate()([u2, d6])
    u3 = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same", activation="relu")(u2)
    u3 = layers.Dropout(0.5)(u3, training=True)
    u3 = layers.Concatenate()([u3, d5])
    u4 = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same", activation="relu")(u3)
    u4 = layers.Concatenate()([u4, d4])
    u5 = layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same", activation="relu")(u4)
    u5 = layers.Concatenate()([u5, d3])
    u6 = layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same", activation="relu")(u5)
    u6 = layers.Concatenate()([u6, d2])
    u7 = layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same", activation="relu")(u6)
    u7 = layers.Concatenate()([u7, d1])

    outputs = layers.Conv2DTranspose(3, kernel_size=4, strides=2, padding="same", activation="tanh")(u7)

    return Model(inputs, outputs)
```

```python
def build_discriminator(input_shape=(256, 256, 3)):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(64, kernel_size=4, strides=2, padding="same", activation="relu")(inputs)
    x = layers.Conv2D(128, kernel_size=4, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, kernel_size=4, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(x)

    x = layers.Flatten()(x)

    # outputs = layers.Conv2D(1, kernel_size=4, strides=1, padding="same", activation="sigmoid")(x)
    outputs = layers.Dense(1, activation='sigmoid')(x)
    model = Model(inputs, outputs)
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
                loss="binary_crossentropy",
                metrics=["accuracy"])
    return model
```

```python
def gan_model(g_model, d_model, input_shape=(256, 256, 3)):
    d_model.trainable = False
    inputs = layers.Input(shape=input_shape)
    gen_output = g_model(inputs)
    output = d_model(gen_output)
    model = Model(inputs, output)
    model.compile(optimizer=tf.keras.optimizers.Adam(0.0002,0.5), loss = "binary_crossentropy")
    return model
```

```python
def train_step(noise, real_images, real_labels, fake_labels):
    fake_images = g_model(noise, training=True)

    d_loss_real = d_model.train_on_batch(real_images, real_labels)
    d_loss_fake = d_model.train_on_batch(fake_images, fake_labels)

    d_loss = 0.5 * tf.add(d_loss_real[0], d_loss_fake[0])

    g_loss = gan_model.train_on_batch(noise, np.ones((half_batch, 1)))

    return d_loss, g_loss
```

```python
for epoch in tqdm(range(epochs), desc="Training Progress", dynamic_ncols=True):
    idx = np.random.randint(0, train_blurred.shape[0], half_batch)
    real_images = train_blurred[idx]

    idx = np.random.randint(0, train_masks.shape[0], half_batch)
    noise = train_masks[idx]

    real_labels = tf.ones((half_batch, 1))
    fake_labels = tf.zeros((half_batch, 1))

    d_loss, g_loss = train_step(noise, real_images, real_labels, fake_labels)

    if epoch % 100 == 0:
        print(f"{epoch} [D loss: {d_loss} | G loss: {g_loss}]")

    if epoch % 5 == 0:
        generated_images = g_model.predict(np.random.normal(0, 1, (5, 256, 256, 3)))
        fig, axs = plt.subplots(1, 5, figsize=(15, 3))
        for i in range(5):
            axs[i].imshow((generated_images[i] + 1) / 2)  # Rescale to [0, 1]
            axs[i].axis('off')
        plt.show()
```

GAN 2

```
def build_generator(input_shape=(256, 256, 3)):
    inputs = layers.Input(shape=input_shape)

    d1 = layers.Conv2D(64, kernel_size=4, strides=2, padding="same", activation="relu")(inputs)
    d2 = layers.Conv2D(128, kernel_size=4, strides=2, padding="same", activation="relu")(d1)
    d3 = layers.Conv2D(256, kernel_size=4, strides=2, padding="same", activation="relu")(d2)
    d4 = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d3)
    d5 = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d4)

    bottleneck = layers.Conv2D(512, kernel_size=4, strides=2, padding="same", activation="relu")(d5)

    u1 = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same", activation="relu")(bottleneck)
    u1 = layers.Dropout(0.5)(u1, training=True)
    u1 = layers.Concatenate()([u1, d5])

    u2 = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same", activation="relu")(u1)
    u2 = layers.Dropout(0.5)(u2, training=True)
    u2 = layers.Concatenate()([u2, d4])

    u3 = layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same", activation="relu")(u2)
    u3 = layers.Concatenate()([u3, d3])

    u4 = layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same", activation="relu")(u3)
    u4 = layers.Concatenate()([u4, d2])

    u5 = layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same", activation="relu")(u4)
    u5 = layers.Concatenate()([u5, d1])

    outputs = layers.Conv2DTranspose(3, kernel_size=4, strides=2, padding="same", activation="tanh")(u5)

    return Model(inputs, outputs)
```

```
def build_discriminator(input_shape=(256, 256, 3)):
    inputs = layers.Input(shape=input_shape)

    x = layers.Conv2D(64, kernel_size=4, strides=2, padding="same")(inputs)
    x = layers.LeakyReLU(alpha=0.2)(x)

    x = layers.Conv2D(128, kernel_size=4, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)

    x = layers.Conv2D(256, kernel_size=4, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)

    x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)

    x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)

    outputs = layers.Conv2D(1, kernel_size=4, strides=1, padding="same", activation="sigmoid")(x)  # 16x16 output

    model = Model(inputs, outputs)
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
                  loss="binary_crossentropy",
                  metrics=["accuracy"])

    model.summary()

    return model
```

```python
def gan_model(g_model, d_model, input_shape=(256, 256, 3)):
    inputs = layers.Input(shape=input_shape)
    gen_output = g_model(inputs)
    output = d_model(gen_output)

    model = Model(inputs, output)

    model.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5), loss="binary_crossentropy")

    return model
```

```python
def train_step(noise, real_images, generator, discriminator, combined_model, batch_size, g_optimizer, d_optimizer):
    real_labels = tf.ones((batch_size, 16, 16, 1))
    fake_labels = tf.zeros((batch_size, 16, 16, 1))

    with tf.GradientTape() as tape_d:
        fake_images = generator(noise, training=True)

        real_output = discriminator(real_images, training=True)
        fake_output = discriminator(fake_images, training=True)

        d_loss = discriminator_loss(real_output, fake_output)

    gradients_d = tape_d.gradient(d_loss, discriminator.trainable_variables)
    d_optimizer.apply_gradients(zip(gradients_d, discriminator.trainable_variables))

    with tf.GradientTape() as tape_g:
        fake_images = generator(noise, training=True)

        fake_output = discriminator(fake_images, training=True)

        g_loss = generator_loss(real_images, fake_images, fake_output)

    gradients_g = tape_g.gradient(g_loss, generator.trainable_variables)
    g_optimizer.apply_gradients(zip(gradients_g, generator.trainable_variables))

    return d_loss, g_loss
```

```python
g_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
d_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

for epoch in tqdm(range(epochs), desc="Training Progress", dynamic_ncols=True):
    for step in range(len(train_masks) // batch_size):
        idx = np.random.randint(0, train_masks.shape[0], batch_size)
        real_images = train_original[idx]
        noise = train_masks[idx]

        d_loss, g_loss = train_step(noise, real_images, g_model, d_model, gan_model, batch_size, g_optimizer, d_optimizer)

    if epoch % 2 == 0:
        print(f"{epoch} [D loss: {d_loss:.4f}] [G loss: {g_loss:.4f}]")

    if epoch % 5 == 0:
        idx = np.random.randint(0, val_blurred.shape[0], 5)
        val_images = val_masks[idx]
        generated_images = g_model.predict(val_images)

        fig, axs = plt.subplots(2, 5, figsize=(15, 6))
        for i in range(5):
            axs[0, i].imshow((val_masks[i] + 1) / 2)
            axs[0, i].axis('off')

            axs[1, i].imshow((generated_images[i] + 1) / 2)
            axs[1, i].axis('off')

        plt.show()
```

Custom loss

```
def discriminator_loss(real_output, fake_output):
    real_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(real_output), logits=real_output))
    fake_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.zeros_like(fake_output), logits=fake_output))

    return real_loss + fake_loss
```

```
def generator_loss(y_true, y_pred, discriminator_output, gamma_L1=100):
    L1_loss = tf.reduce_mean(tf.abs(y_true - y_pred))

    adv_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(discriminator_output), logits=discriminator_output))

    return gamma_L1 * L1_loss + adv_loss
```

CNN

```
[ ]  def autoencoder(input_shape=(256, 256, 3)):
        input_img = layers.Input(shape=input_shape)
        x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
        x = layers.MaxPooling2D((2, 2), padding='same')(x)
        x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
        x = layers.MaxPooling2D((2, 2), padding='same')(x)
        x = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)
        encoded = layers.MaxPooling2D((2, 2), padding='same')(x)


        x = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(encoded)
        x = layers.UpSampling2D((2, 2))(x)
        x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
        x = layers.UpSampling2D((2, 2))(x)
        x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
        x = layers.UpSampling2D((2, 2))(x)
        decoded = layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)

        return Model(input_img, decoded)
```

Keras-SRGAN

Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network implemented in Keras
by deepak112
https://github.com/deepak112/Keras-SRGAN

Tools

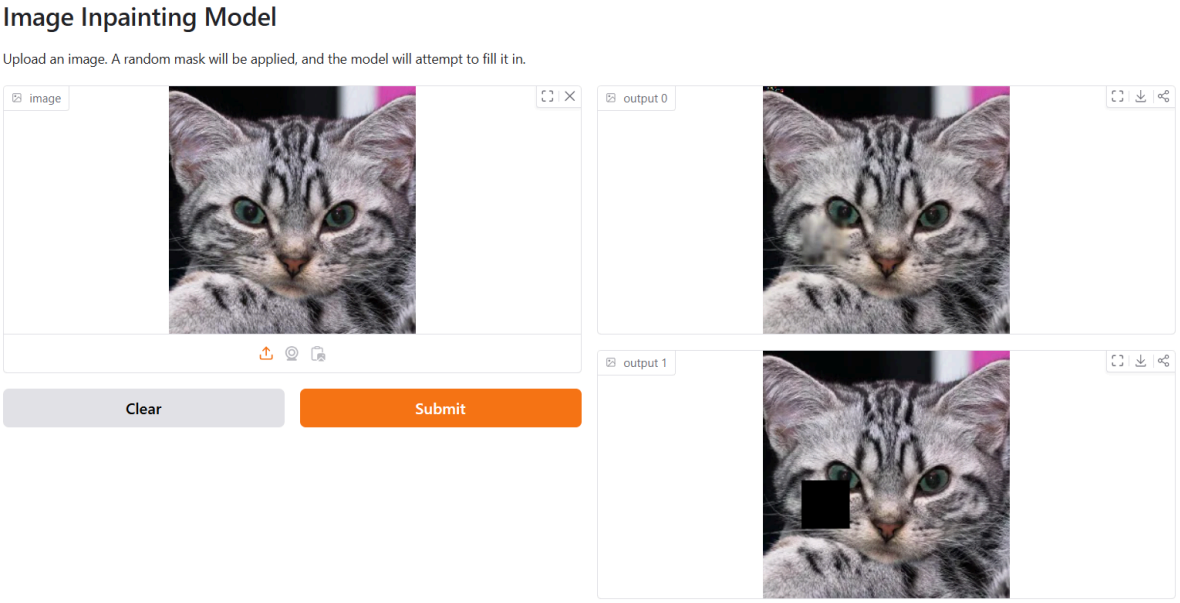
Requirements

Git with Readme

Additional points:

- MLflow,Tensorboard, Neptune, Weights & Biases (along with some analysis of experiments) +1pk
- Run as docker/ docker compose +1pk
- REST API or GUI for example Gradio, Streamlit +1pk
- DVC +2pk
- Every other MLOps tool +1pk
- Label Studio or other data labeling tools +1pk

https://huggingface.co/spaces/akrasi/CATastrophe



Grading

Part	Points
Problem	3 (image inpainting)
Dataset	0
Model	2 (GAN2) +1 (GAN1) +2 (CNN) +1 (Keras-SRGAN) =6
Training	1+1=2

Tools	1 (Gradio)
Total	**12**