MOSAIC'23 PS-2

IMAGE-INPAINTING MODEL

@442|PURAV HARAN|PRASOON SAHAY|PRIYANSHU

OVERVIEW

-> Setup, installations and imports
-> Preparing dataset
-> Partial convolution based auto-encoder-decoder model
-> Training and testing

SETUP, INSTALLATIONS & IMPORTS

Setups, Installations and Imports

```
[ ] !pip install tensorflow-gpu==2.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
ERROR: Could not find a version that satisfies the requirement tensorflow-gpu==2.0 (from versions: 2.5.0, 2.5.1, 2.5.2, 2.5.3, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.6.5, 2.7.0rc0, 2.7.0rc1, 2.7.0, 2.7.1, 2.7.2, 2
ERROR: No matching distribution found for tensorflow-gpu==2.0
```

```
[ ] !pip install wandb -q
```

```
import tensorflow as tf
print(tf.__version__)
from tensorflow import keras

print('[INFO]', tf.config.experimental.list_physical_devices('GPU')[0])
```

```
2.12.0
[INFO] PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')
```

```
[ ] import wandb
from wandb.keras import WandbCallback

wandb.login()
```

```
wandb: Currently logged in as: purav-haran1305. Use `wandb login --relogin` to force relogin
True
```

```
[ ] import os
import cv2
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

from mpl_toolkits.axes_grid1 import ImageGrid
```

PREPARING DATASET

We've taken a CIFAR-10 dataset to train the model

▾ Visualization of Cifar 10

```
## Get first 32 images as samples
sample_images = x_train[:32]
sample_labels = y_train[:32]

fig = plt.figure(figsize=(16., 8.))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(4, 8),  # creates 2x2 grid of axes
                 axes_pad=0.3,  # pad between axes in inch.
                 )

for ax, image, label in zip(grid, sample_images, sample_labels):
  ax.imshow(image)
  ax.set_title(label[0])
```

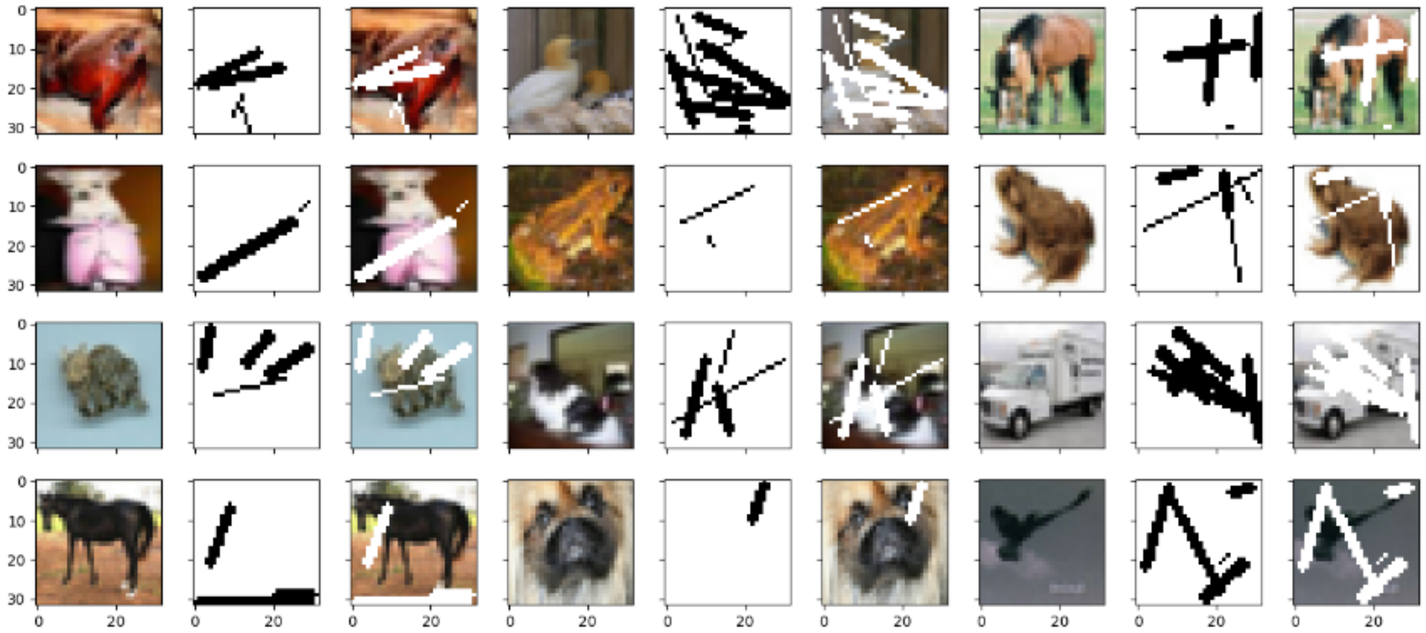
We've generated the data with Patch Augmentation

```
## Ref: https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly.
class createAugment(keras.utils.Sequence):
  'Generates data for Keras'
  def __init__(self, X, y, batch_size=32, dim=(32, 32), n_channels=3, shuffle=True):
      'Initialization'
      self.batch_size = batch_size
      self.X = X
      self.y = y
      self.dim = dim
      self.n_channels = n_channels
      self.shuffle = shuffle

      self.on_epoch_end()

  def __len__(self):
      'Denotes the number of batches per epoch'
      return int(np.floor(len(self.X) / self.batch_size))

  def __getitem__(self, index):
      'Generate one batch of data'
      # Generate indexes of the batch
      indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]

      # Generate data
      X_inputs, y_output = self.__data_generation(indexes)
      return X_inputs, y_output

  def on_epoch_end(self):
      'Updates indexes after each epoch'
      self.indexes = np.arange(len(self.X))
      if self.shuffle:
          np.random.shuffle(self.indexes)

  def __data_generation(self, idxs):
    # Masked_images is a matrix of masked images used as input
    Masked_images = np.empty((self.batch_size, self.dim[0], self.dim[1], self.n_channels)) # Masked image
    # Mask_batch is a matrix of binary masks used as input
    Mask_batch = np.empty((self.batch_size, self.dim[0], self.dim[1], self.n_channels)) # Binary Masks
    # y_batch is a matrix of original images used for computing error from reconstructed image
    y_batch = np.empty((self.batch_size, self.dim[0], self.dim[1], self.n_channels)) # Original image


    ## Iterate through random indexes
    for i, idx in enumerate(idxs):
      image_copy = self.X[idx].copy()

      ## Get mask associated to that image
      masked_image, mask = self.__createMask(image_copy)

      Masked_images[i,] = masked_image/255
      Mask_batch[i,] = mask/255
      y_batch[i] = self.y[idx]/255

    ## Return mask as well because partial convolution require the same.
    return [Masked_images, Mask_batch], y_batch

  def __createMask(self, img):
    ## Prepare masking matrix
    mask = np.full((32,32,3), 255, np.uint8) ## White background
    for _ in range(np.random.randint(1, 10)):
      # Get random x locations to start line
      x1, x2 = np.random.randint(1, 32), np.random.randint(1, 32)
      # Get random y locations to start line
```

```
        masked_image[mask==0] = 255

        return masked_image, mask
```

```
## Prepare training and testing mask-image pair generator
traingen = createAugment(x_train, x_train)
testgen = createAugment(x_test, x_test, shuffle=False)
```

```
# Legend: Original Image | Mask generated | Masked Image

## Examples
sample_idx = 90 ## Change this to see different batches

[masked_images, masks], sample_labels = traingen[sample_idx]
sample_images = [None]*(len(masked_images)+len(masks)+len(sample_labels))
sample_images[::3] = sample_labels
# masks[]
sample_images[1::3] = masks
sample_images[2::3] = masked_images

fig = plt.figure(figsize=(17., 8.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(4, 9),  # creates 2x2 grid of axes
                 axes_pad=0.3,  # pad between axes in inch.
                 )

for ax, image in zip(grid, sample_images):
  ax.imshow(image)

plt.show()
```



Partial convolution based auto-encoder-decoder model

```
## utils is present in the cloned repo. Visit repo for the implementation of PConv2D.
from utils.pconv_layer import PConv2D
```

```
[14] ## For more information into formulation: https://www.youtube.com/watch?v=AZr64OxshLo
     ## Metric
     def dice_coef(y_true, y_pred):
         y_true_f = keras.backend.flatten(y_true)
         y_pred_f = keras.backend.flatten(y_pred)
         intersection = keras.backend.sum(y_true_f * y_pred_f)
         return (2. * intersection) / (keras.backend.sum(y_true_f + y_pred_f))
```

```
[15] class InpaintingModel:
       """
       Build UNET like model for image inpaining task.
       """
       def prepare_model(self, input_size=(32,32,3)):
         input_image = keras.layers.Input(input_size)
         input_mask = keras.layers.Input(input_size, name='encoder_input')

         conv1, mask1, conv2, mask2 = self.__encoder_layer(32, input_image, input_mask, ['conv1', 'conv2'])
         conv3, mask3, conv4, mask4 = self.__encoder_layer(64, conv2, mask2, ['conv3', 'conv4'])
         conv5, mask5, conv6, mask6 = self.__encoder_layer(128, conv4, mask4, ['conv5', 'conv6'])
         conv7, mask7, conv8, mask8 = self.__encoder_layer(256, conv6, mask6, ['conv7', 'encoder_output'])

         conv9, mask9, conv10, mask10 = self.__decoder_layer(256, 128, conv8, mask8, conv7, mask7, ['conv9', 'conv10'])
         conv11, mask11, conv12, mask12 = self.__decoder_layer(128, 64, conv10, mask10, conv5, mask5, ['conv11', 'conv12'])
         conv13, mask13, conv14, mask14 = self.__decoder_layer(64, 32, conv12, mask12, conv3, mask3, ['conv13', 'conv14'])
         conv15, mask15, conv16, mask16 = self.__decoder_layer(32, 3, conv14, mask14, conv1, mask1, ['conv15', 'decoder_output'])

         outputs = keras.layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(conv16)

         return keras.models.Model(inputs=[input_image, input_mask], outputs=[outputs])

       def __encoder_layer(self, filters, in_layer, in_mask, names):
         conv1, mask1 = PConv2D(32, (3,3), strides=1, padding='same', name=names[0])([in_layer, in_mask])
         conv1 = keras.activations.relu(conv1)

         conv2, mask2 = PConv2D(32, (3,3), strides=2, padding='same', name=names[1])([conv1, mask1])
         # conv2 = keras.layers.BatchNormalization()(conv2, training=True)
         conv2 = keras.activations.relu(conv2)

         return conv1, mask1, conv2, mask2

       def __decoder_layer(self, filter1, filter2, in_img, in_mask, share_img, share_mask, names):
         up_img = keras.layers.UpSampling2D(size=(2,2))(in_img)
         up_mask = keras.layers.UpSampling2D(size=(2,2))(in_mask)
         concat_img = keras.layers.Concatenate(axis=3)([share_img, up_img])
         concat_mask = keras.layers.Concatenate(axis=3)([share_mask, up_mask])

         conv1, mask1 = PConv2D(filter1, (3,3), padding='same', name=names[0])([concat_img, concat_mask])
         conv1 = keras.activations.relu(conv1)

         conv2, mask2 = PConv2D(filter2, (3,3), padding='same', name=names[1])([conv1, mask1])
         # conv2 = keras.layers.BatchNormalization()(conv2)
         conv2 = keras.activations.relu(conv2)

         return conv1, mask1, conv2, mask2
```

Partial Convolution performs the normalization of the output to adjust for the fraction of missing data. A partial convolution layer comprises masked and re-normalized convolution operation followed by a mask-update setup.

TRAINING

Train

```
[ ] wandb.init(
        # set the wandb project where this run will be logged
        project="my-awesome-project"


    )
```

Tracking run with wandb version 0.14.0
Run data is saved locally in /content/deepimageinpainting/wandb/run-20230404_220934-7xto16n2
Syncing run different-pine-6 to Weights & Biases (docs)
View project at https://wandb.ai/purav-haran1305/my-awesome-project
View run at https://wandb.ai/purav-haran1305/my-awesome-project/runs/7xto16n2
Display W&B run

```
class PredictionLogger(tf.keras.callbacks.Callback):
    def __init__(self):
        super(PredictionLogger, self).__init__()

    def on_epoch_end(self, logs, epoch):
        sample_idx = 54
        [masked_images, masks], sample_labels = testgen[sample_idx]

        m_images = []
        binary_masks = []
        predictions = []
        labels = []

        for i in range(32):
          inputs = [masked_images[i].reshape((1,)+masked_images[i].shape), masks[i].reshape((1,)+masks[i].shape)]
          impainted_image = model.predict(inputs)

          m_images.append(masked_images[i])
          binary_masks.append(masks[i])
          predictions.append(impainted_image.reshape(impainted_image.shape[1:]))
          labels.append(sample_labels[i])

        wandb.log({"masked_images": [wandb.Image(m_image)
                                     for m_image in m_images]})
        wandb.log({"masks": [wandb.Image(mask)
                             for mask in binary_masks]})
        wandb.log({"predictions": [wandb.Image(inpainted_image)
                                   for inpainted_image in predictions]})
        wandb.log({"labels": [wandb.Image(label)
                              for label in labels]})
```

TESTING

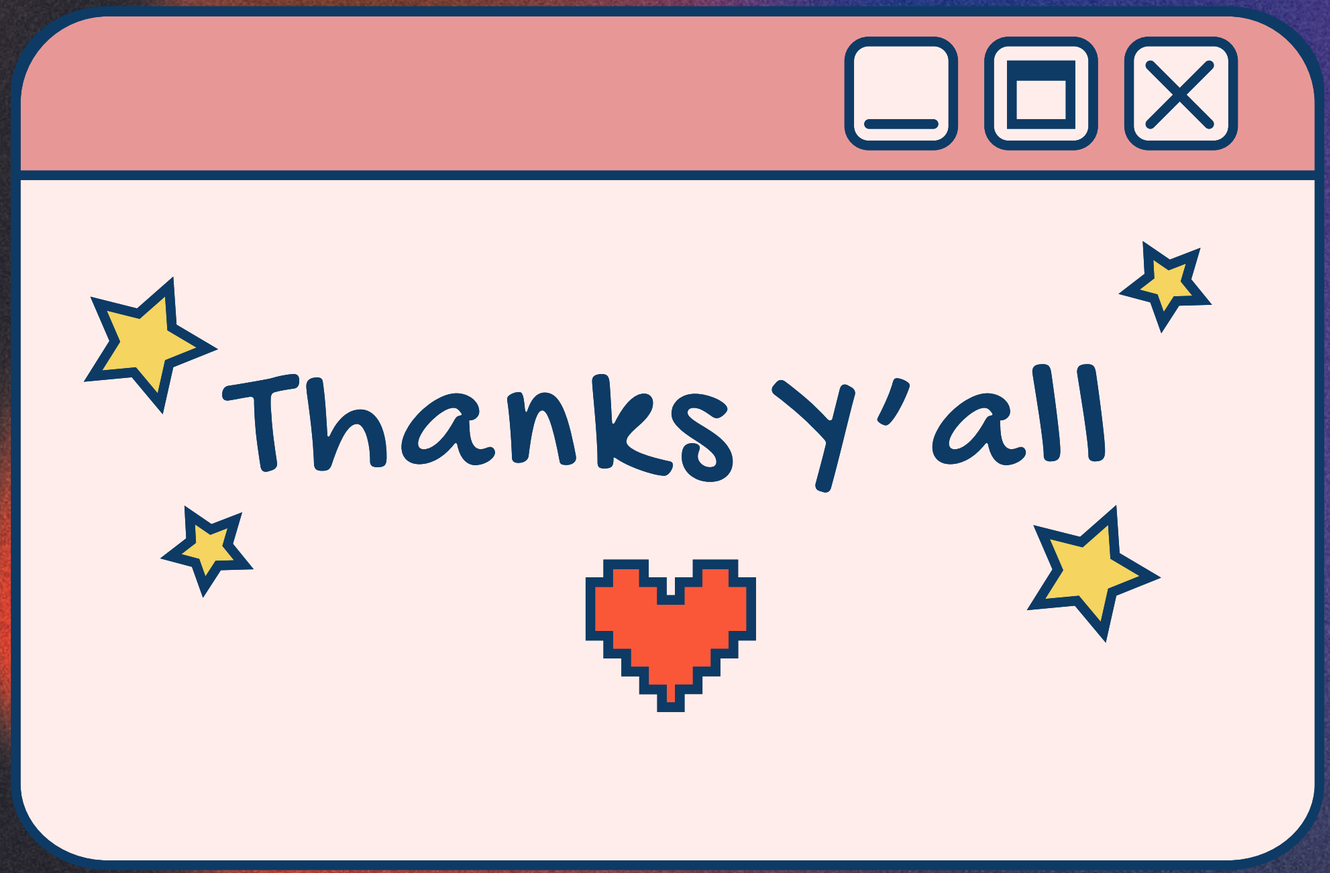
Testing on images

```
## Legend: Original Image | Mask generated | Inpainted Image | Ground Truth

## Examples
rows = 32
sample_idx = 54
[masked_images, masks], sample_labels = testgen[sample_idx]

fig, axs = plt.subplots(nrows=rows, ncols=4, figsize=(8, 2*rows))

for i in range(32):
  inputs = [masked_images[i].reshape((1,)+masked_images[i].shape), masks[i].reshape((1,)+masks[i].shape)]
  impainted_image = model.predict(inputs)
  axs[i][0].imshow(masked_images[i])
  axs[i][1].imshow(masks[i])
  axs[i][2].imshow(impainted_image.reshape(impainted_image.shape[1:]))
  axs[i][3].imshow(sample_labels[i])

plt.show()
```

```
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
```

SAVING THE MODEL BY:

MODEL.SAVE('/CONTENT/SAMPLE_DATA/MODEL1.H5')


Thanks Y'all