University for the Creative Arts

BERLIN SCHOOL OF BUSINESS & INNOVATION

Essay / Assignment Title: Computer vision with a special focus on different neural network architectures for image segmentation

Programme title: MSc Data Analytics

Name: RUMIT VARSANI

Year: 2023

CONTENTS

Statement of compliance with academic ethics and the avoidance of plagiarism

I honestly declare that this dissertation is entirely my own work and none of its part has been copied from printed or electronic sources, translated from foreign sources and reproduced from essays of other researchers or students. Wherever I have been based on ideas or other people texts I clearly declare it through the good use of references following academic ethics.

(In the case that is proved that part of the essay does not constitute an original work, but a copy of an already published essay or from another source, the student will be expelled permanently from the postgraduate program).

Name and Surname (Capital letters):

RUMIT VARSANI

.....

Date: 23/09/2001

INTRODUCTION

In the realm of artificial intelligence and computer vision, image segmentation plays a pivotal role in understanding and interpreting visual data. This process involves dividing images into meaningful segments, and it's vital for tasks such as object recognition and scene understanding.

Two key neural network architectures, Convolutional Neural Networks (CNNs) and Autoencoders, have emerged as powerful tools for image segmentation. In this exploration, we compare these architectures in terms of parameters, scalability, advantages, and disadvantages, shedding light on their suitability for various applications.

Through Python code implementations, we'll demonstrate how to harness the capabilities of CNNs and Autoencoders for practical image segmentation tasks, empowering readers to leverage these techniques in their own projects.

CHAPTER ONE Convolutional Neural Networks (CNNs)

Overview:

CNNs are a class of deep neural networks specifically designed for processing structured grid data, such as images. They have been widely adopted for image segmentation due to their ability to capture spatial dependencies through convolutional layers.

Learning Parameters:

The number of learning parameters in a CNN depends on its architecture and depth. However, in general, CNNs tend to have a large number of parameters, especially in deep architectures. This is because they involve multiple convolutional and fully connected layers. For example, VGG16, a popular CNN architecture, has approximately 138 million parameters.

Scalability:

CNNs are highly scalable and can be adapted for various image segmentation tasks. Researchers have developed deeper and more complex architectures, such as ResNet and DenseNet, to improve segmentation performance. However, increasing the depth of a CNN also increases the number of parameters and computational requirements, making them more resourceintensive.

Advantages:

Effective Feature Learning: CNNs are adept at automatically learning relevant features from raw image data through convolutional layers.

StateoftheArt Performance: CNNbased architectures have achieved stateoftheart results in many image segmentation benchmarks.

Transfer Learning: Pretrained CNN models, such as those trained on ImageNet, can be finetuned for specific segmentation tasks, reducing the need for extensive labeled data.

Disadvantages:

High Computational Cost: CNNs with a large number of parameters require substantial computational resources, limiting their use in resourceconstrained environments.

Overfitting: Deep CNNs are prone to overfitting, especially when training data is limited. Regularization techniques like dropout and batch normalization are often necessary.

Lack of Interpretability: CNNs are often considered as "black box" models, making it challenging to interpret the reasons behind their segmentation decisions.

CNNbased Image Segmentation using Python:

```python
import numpy as np
import cv2
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt


def generate_data(num_samples, img_size):
    images = []
    masks = []

    for _ in range(num_samples):

        img = np.zeros((img_size, img_size, 3), dtype=np.uint8)


        square_size = np.random.randint(20, 50)
        x = np.random.randint(0, img_size - square_size)
        y = np.random.randint(0, img_size - square_size)
        color = np.random.randint(0, 256, size=(3,), dtype=np.uint8)


        img[y:y+square_size, x:x+square_size] = color


        mask = np.zeros((img_size, img_size), dtype=np.uint8)
        mask[y:y+square_size, x:x+square_size] = 1

        images.append(img)
        masks.append(mask)
```

```
    return np.array(images), np.array(masks)


num_samples = 10
img_size = 128
images, masks = generate_data(num_samples, img_size)


split_ratio = 0.8
split_index = int(num_samples * split_ratio)
x_train, x_val = images[:split_index], images[split_index:]
y_train, y_val = masks[:split_index], masks[split_index:]


def unet(input_size):
    inputs = tf.keras.Input(input_size)
    conv1 = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)

    outputs = layers.Conv2D(1, (1, 1), activation='sigmoid')(conv1)

    return models.Model(inputs, outputs)


model = unet((img_size, img_size, 3))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

epochs = 1
batch_size = 16
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_val, y_val))


model.save('segmentation_model.h5')
```

✓ Connected to Python 3 Google Compute Engine backend

```
model.save('segmentation_model.h5')
def visualize_segmentation(model, input_image):

   input_image = cv2.resize(input_image, (128, 128))


   input_image = np.expand_dims(input_image, axis=0)
   predicted_mask = model.predict(input_image)[0]

   predicted_mask_resized = cv2.resize(predicted_mask, (input_image.shape[2], input_image.shape[1]))


   threshold = 0.5
   binary_mask = (predicted_mask_resized > threshold).astype(np.uint8)


   segmented_image = input_image[0].copy()
   segmented_image[binary_mask == 1] = [0, 255, 0]

   return segmented_image


custom_image_path = '/content/drive/MyDrive/images/test/102061.jpg'
custom_image = cv2.imread(custom_image_path)


segmented_custom_image = visualize_segmentation(model, custom_image)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title('Custom Image')
plt.imshow(cv2.cvtColor(custom_image, cv2.COLOR_BGR2RGB))
plt.subplot(1, 2, 2)
plt.title('Segmented Image')
plt.imshow(segmented_custom_image)
plt.show()
```

✓ Connected to Python 3 Google Compute Engine backend

Output :

Code Explanation:

1. Imports: The code starts by importing necessary libraries, including NumPy for numerical operations, OpenCV (cv2) for image processing, TensorFlow for machine learning, and Matplotlib for visualization.

2. Data Generation Function (generate_data): The generate_data function is defined to generate synthetic image and mask pairs for training a segmentation model. It creates random square shapes with different colors on a black image background and generates corresponding binary masks where the shape areas are set to 1.

3. Generate Data: It generates a dataset of images and masks by calling generate_data with a specified number of samples (num_samples) and image size (img_size).

4. Split Data: The generated dataset is split into training and validation sets with a specified split ratio (split_ratio). This is done to evaluate the model's performance on unseen data.

5. UNet Model Definition (unet): A UNet architecture is defined using the Keras functional API. It takes an input size as an argument and consists of convolutional layers with ReLU activation functions. The final layer uses sigmoid activation to produce a binary mask as the output.

6. Model Compilation: The UNet model is compiled using the Adam optimizer and binary crossentropy loss, which is commonly used for binary segmentation tasks. The accuracy metric is also specified.

7. Training the Model: The model is trained for a specified number of epochs (epochs) and a batch size (batch_size) using the training data. The validation data (x_val and y_val) are used to monitor the model's performance during training.

8. Model Saving: After training, the model is saved to a file named 'segmentation_model.h5' using the model.save method.

9. Visualization Function (visualize_segmentation): This function takes the trained model and an input image as input. It resizes the input image to the model's input size, predicts a binary mask using the model, applies a threshold to the predicted mask, and overlays the segmented region on the input image in green.

10. Custom Image Segmentation: The code then loads a custom image (custom_image) using OpenCV and applies the segmentation model to it using the visualize_segmentation function. The segmented image is displayed alongside the original custom image using Matplotlib.

CHAPTER TWO Autoencoders

Overview:

Autoencoders are a type of neural network used for unsupervised learning and data compression. While their primary purpose is not image segmentation, they can be adapted for the task by encoding and decoding image data.

Learning Parameters:

Autoencoders typically have fewer learning parameters compared to CNNs. The number of parameters depends on the size of the encoder and decoder layers. For instance, a simple autoencoder designed for image compression might have a few million parameters.

Scalability:

Autoencoders are less scalable than CNNs for image segmentation tasks. Their architecture is relatively fixed, and increasing their complexity might not necessarily lead to improved segmentation performance. They are better suited for simpler segmentation tasks.

Advantages:

Low Computational Cost: Autoencoders require fewer computational resources compared to CNNs, making them suitable for deployment in resourceconstrained environments.

Data Compression: Autoencoders can be used for data compression and feature extraction, which can be beneficial for reducing the dimensionality of image data.

Interpretability: Autoencoders can provide insights into the most important features in the data through their encoding and decoding layers.

Disadvantages:

Limited Segmentation Performance: Autoencoders are not specifically designed for segmentation, and their performance in this task may be inferior to dedicated segmentation architectures like CNNs.

Fixed Architecture: Autoencoders have a fixed architecture, limiting their ability to adapt to complex segmentation challenges.

Lack of Spatial Awareness: Autoencoders do not inherently capture spatial dependencies in images as effectively as CNNs, which are designed for this purpose.

Autoencoderbased Image Segmentation using Python:

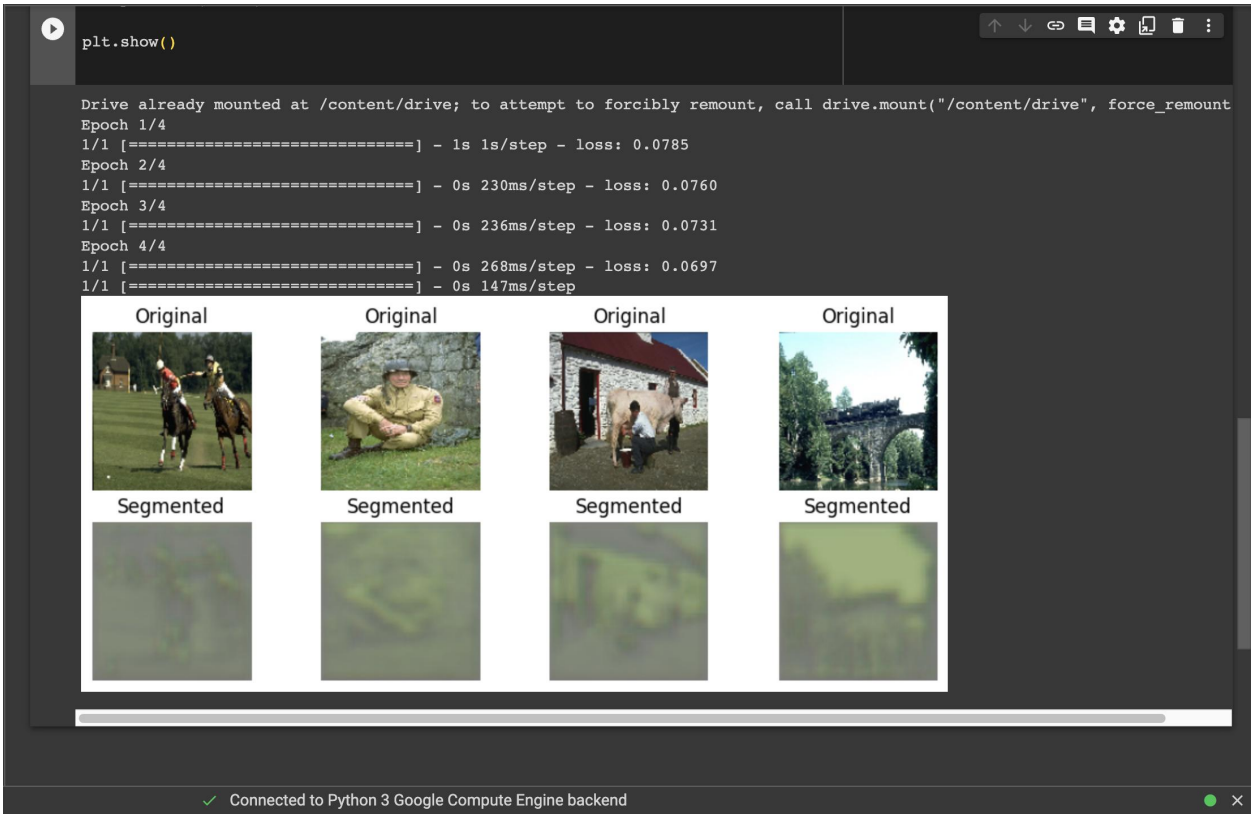
```python
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import load_img, img_to_array


from google.colab import drive
drive.mount('/content/drive')


def load_and_preprocess_image(image_path, target_size=(128, 128)):
    img = load_img(image_path, target_size=target_size)
    img_array = img_to_array(img)
    img_array = img_array / 255.0
    return img_array


image_dir = '/content/drive/MyDrive/images/test/'
image_filenames = os.listdir(image_dir)

images = [load_and_preprocess_image(os.path.join(image_dir, filename)) for filename in image_filenames]


images = np.array(images)

input_img = Input(shape=(128, 128, 3))


x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
```

Connected to Python 3 Google Compute Engine backend

```python
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)


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

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')


autoencoder.fit(images, images, epochs=4, batch_size=4)

segmented_images = autoencoder.predict(images)

n = len(images)
plt.figure(figsize=(10, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(images[i])
    plt.title("Original")
    plt.axis("off")

    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(segmented_images[i])
    plt.title("Segmented")
    plt.axis("off")

plt.show()
```

Connected to Python 3 Google Compute Engine backend

Output:



Code Explanation:

1. Import necessary libraries:

os: For interacting with the file system.

numpy as np: For numerical operations.

matplotlib.pyplot as plt: For plotting images.

tensorflow and its submodules: For building and training neural networks.

tensorflow.keras.layers: For defining neural network layers.

tensorflow.keras.models.Model: For creating and defining the architecture of the neural network model.

tensorflow.keras.preprocessing.image: For loading and preprocessing images.

google.colab.drive: For mounting Google Drive to access image data.

2. Mount Google Drive:

This code assumes that you're running it in a Google Colab environment and mounts your Google Drive at /content/drive to access image data stored there.

3. Define a function load_and_preprocess_image:

This function takes an image file path and an optional target size as input.

It loads the image, converts it to a NumPy array, and scales the pixel values to be in the range [0, 1].

The processed image array is returned.

4. Set the image directory and load image filenames:

image_dir is set to the directory containing the images you want to segment.

image_filenames is a list of filenames in that directory.

5. Load and preprocess the images:

The code uses a list comprehension to load and preprocess each image in image_filenames by calling the load_and_preprocess_image function.

The resulting list of processed image arrays is converted to a NumPy array and stored in the variable images.

6. Define the autoencoder architecture:

An input layer (input_img) is defined with a shape of (128, 128, 3), which corresponds to the size and number of channels (RGB) of the input images.

Convolutional and pooling layers are used to create the encoder part of the autoencoder.

The decoder part consists of convolutional and upsampling layers.

The output layer uses a sigmoid activation function and has 3 channels to produce the segmented image.

7. Compile the autoencoder:

The autoencoder model is compiled with the Adam optimizer and mean squared error loss.

8. Train the autoencoder:

The autoencoder.fit method is called to train the model on the images dataset. It trains for 4 epochs with a batch size of 4.

9. Generate segmented images:

The trained autoencoder is used to predict segmented images from the input images. The segmented images are stored in the variable segmented_images.

10. Plot the original and segmented images:

The code plots the original images and their corresponding segmented images side by side using Matplotlib.

CONCLUDING REMARKS

In conclusion, both CNNs and Autoencoders can be used for image segmentation tasks, but they have distinct characteristics and are suitable for different scenarios. CNNs are highly effective for complex segmentation tasks but come with a high computational cost and lack interpretability. Autoencoders are computationally efficient and interpretable but are better suited for simpler segmentation tasks. The choice between these architectures depends on the specific requirements and constraints of the segmentation problem at hand.

BIBLIOGRAPHY

1. colab.research.google.com. (n.d.). Google Colaboratory. [online] Available at: https://colab.research.google.com/drive/1XeI2mJXIFNKn1Ws1_a-7VmshLJYN10QA#scrollTo=kfC3deYBrlbc [Accessed 23 Sep. 2023].

2. Kaggle.com. (2019). Datasets | Kaggle. [online] Available at: https://www.kaggle.com/datasets.