University for the Creative Arts

BERLIN SCHOOL OF BUSINESS & INNOVATION

Essay / Assignment Title: Computer Vision with a Special Focus on Noise Removal Modelling

Programme title: MSc Data Analytics

Name: KEVADIYA SMIT KAMLESHBHAI

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):

SMIT KEVADIYA

.....

Date: 08/10/2023

INTRODUCTION

The realm of image processing often grapples with the challenge of noise interference, a persistent issue affecting the clarity and quality of images. Noise can be introduced to an image through various means: during the capture due to environmental factors or inherent camera sensor limitations, or post-capture, such as during transmission or storage. This perturbation not only degrades image quality but can also significantly hinder subsequent image analysis tasks like object recognition, image segmentation, and more. Traditional denoising methods, although effective to some extent, often struggle to preserve the intricate details of an image while removing noise. Enter deep learning: a powerful paradigm that has dramatically revolutionized image processing tasks. Among the deep learning architectures, Convolutional Neural Networks (CNNs) have exhibited outstanding performance in handling image data. In this assignment, our journey navigates through using a Convolutional Autoencoder, a specialized type of CNN, to tackle the image denoising problem. Our choice of dataset, the CIFAR-10, is renowned for its diverse set of images across ten categories, providing a robust platform for experimentation. Our approach is twofold: first, to artificially introduce noise, mimicking real-world scenarios, and second, to train the autoencoder to effectively extract underlying image patterns and reconstruct the clean image from the noisy counterpart.

CHAPTER ONE (LOADING THE DATASET AND ADDING NOISE TO THE IMAGES)

CODE 1:

```
import tensorflow as tf

# Load CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.cifar10.load_data()

# Print the shape of the datasets
print("Training images shape:", train_images.shape)
print("Training labels shape:", train_labels.shape)
print("Test images shape:", test_images.shape)
print("Test labels shape:", test_labels.shape)
```

The TensorFlow library and gives it the alias tf. TensorFlow is an open-source deep learning framework developed by Google.

(train_images, train_labels), (test_images, test_labels): The load_data() function returns the dataset as a tuple of two tuples. The first tuple contains the training images and their associated labels, and the second tuple contains the test images and their labels. By using this syntax, the code is unpacking the returned data into four separate arrays.

train_images.shape: This will print the shape of the training images dataset. For CIFAR-10, it will return (50000, 32, 32, 3), indicating there are 50,000 images, each of size 32x32 pixels and with 3 color channels (Red, Green, Blue).

train_labels.shape: This will print the shape of the labels corresponding to the training images. It will return (50000, 1), indicating there are 50,000 labels, one for each training image.

test_images.shape: Similarly, this will print the shape of the test images dataset. It will return (10000, 32, 32, 3), meaning there are 10,000 test images of size 32x32 pixels with 3 color channels.

test_labels.shape: This prints the shape of the labels for the test images. It will return (10000, 1), meaning there are 10,000 labels for the test images.

OUTPUT of Code 1:

```
Training images shape: (50000, 32, 32, 3)
Training labels shape: (50000, 1)
Test images shape: (10000, 32, 32, 3)
Test labels shape: (10000, 1)
```

CODE:2

```
import numpy as np

def add_gaussian_noise(images, mean=0, sigma=0.1):
    """Add Gaussian noise to a list of images."""
    noise = np.random.normal(mean, sigma, images.shape)
    noisy_images = images + noise
    noisy_images = np.clip(noisy_images, 0, 255).astype(np.uint8)
    return noisy_images

def add_salt_and_pepper_noise(images, salt_prob=0.005, pepper_prob=0.005):
    """Add salt and pepper noise to a list of images."""
    noisy_images = np.copy(images)
    total_pixels = images.shape[1] * images.shape[2]

    # Salt noise
    num_salt = np.ceil(salt_prob * total_pixels)
    coords = [np.random.randint(0, i-1, int(num_salt)) for i in
images.shape]
    noisy_images[coords[0], coords[1], coords[2]] = 255

    # Pepper noise
    num_pepper = np.ceil(pepper_prob * total_pixels)
    coords = [np.random.randint(0, i-1, int(num_pepper)) for i in
images.shape]
    noisy_images[coords[0], coords[1], coords[2]] = 0

    return noisy_images
```

Gaussian Noise:

Here, the code uses numpy's random.normal() function to generate noise values. This function creates a sample (or samples) from the “standard normal” distribution.

mean: Central tendency of the distribution.

sigma: Standard deviation, determining the spread or width of the bell curve.

images.shape: Ensures the noise generated has the same shape as the images.

By adding the generated Gaussian noise to the original images, the images are perturbed.

np.clip() is used here to ensure all pixel values stay within the 0-255 range (valid range for images). Any value below 0 becomes 0, and any value above 255 becomes 255. The resultant noisy image matrix might not be of integer type; hence .astype(np.uint8) is used to convert all values to unsigned 8-bit integers.

Salt-and-Pepper Noise:

Here, salt_prob is the probability for a pixel to turn white (salt), and pepper_prob is the probability for a pixel to turn black (pepper).

The function first makes a copy of the original images, ensuring not to modify the original array.

This line computes the total number of pixels in one image by multiplying the dimensions (width x height).

For Salt noise:

Calculate the number of pixels that will be turned to salt based on the given probability. np.ceil is used to round up to ensure at least one pixel is changed when the product of salt_prob and total_pixels is not an integer.

For each dimension of the image (height, width, channels), the code is generating random integer coordinates. The purpose is to randomly select which pixels will be turned to salt.

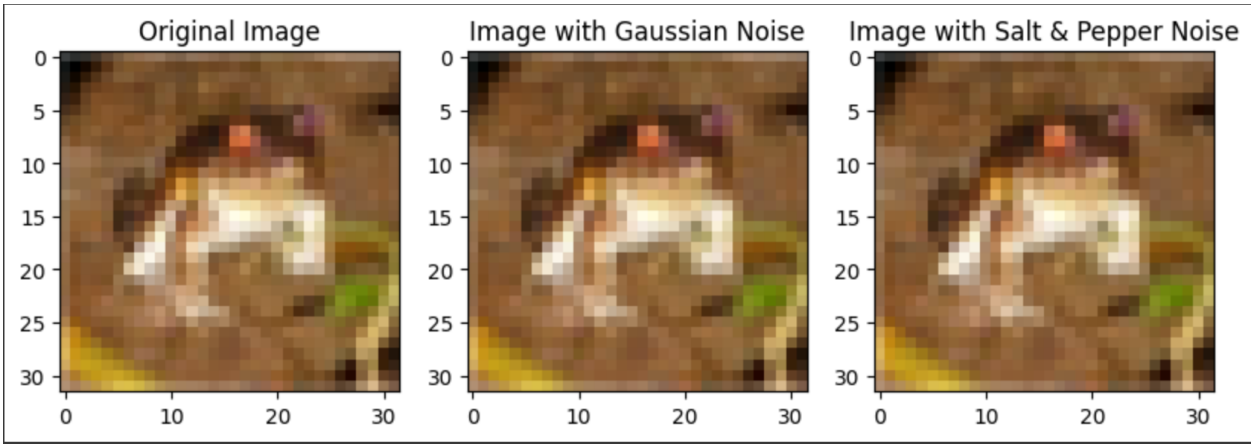
For the randomly selected coordinates, the pixel values are set to 255 (white), thus introducing salt noise.

The procedure for Pepper noise is similar, but the pixel values are set to 0 (black) instead.

CODE:3

```
# Add Gaussian noise
train_images_gaussian_noisy = add_gaussian_noise(train_images)
test_images_gaussian_noisy = add_gaussian_noise(test_images)

# Add salt and pepper noise
train_images_sp_noisy = add_salt_and_pepper_noise(train_images)
test_images_sp_noisy = add_salt_and_pepper_noise(test_images)

# Check out one of the noisy images
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 4))
plt.subplot(1, 3, 1)
plt.imshow(train_images[0])
plt.title('Original Image')
plt.subplot(1, 3, 2)
plt.imshow(train_images_gaussian_noisy[0])
plt.title('Image with Gaussian Noise')
plt.subplot(1, 3, 3)
plt.imshow(train_images_sp_noisy[0])
plt.title('Image with Salt & Pepper Noise')
plt.show()
```

OUTPUT of CODE 3:



Similarly, these lines use the add_salt_and_pepper_noise() function to introduce Salt-and-Pepper noise to the training and testing images. After applying the function, you get two more sets of images: train_images_sp_noisy and test_images_sp_noisy.

Here, the matplotlib library is imported, which is a commonly used library in Python for plotting and visualizing data.

This line creates a new figure with specified dimensions (width=10, height=4) where the images will be displayed.

This creates a subplot grid of dimensions 1 row x 3 columns. The last argument (1) specifies that the subsequent plot/image will be in the 1st cell of this grid.

Here, the first image from the original training dataset (without noise) is displayed using imshow(). The title() function sets a title above this image as 'Original Image'.

This sets the subsequent plot/image to be in the 2nd cell of the grid.

Displays the first image from the train_images_gaussian_noisy set (i.e., the original image with added Gaussian noise) and sets its title.

This sets the subsequent plot/image to be in the 3rd cell of the grid.

Displays the first image from the train_images_sp_noisy set (i.e., the original image with added Salt-and-Pepper noise) and sets its title.

Finally, the show() function is called to display the figure with all the subplots.

The code uses two functions to add Gaussian and Salt-and-Pepper noise to the CIFAR-10 training and testing images. It then visualizes an example image from the training dataset in three versions: original, with Gaussian noise, and with Salt-and-Pepper noise, using matplotlib.

CHAPTER TWO (DESIGNING THE CONVOLUTIONAL AUTOENCODER)

CODE 4:

```
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
UpSampling2D
from tensorflow.keras.models import Model

input_img = Input(shape=(32, 32, 3))  # CIFAR-10 images shape

# Encoder
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)
x = MaxPooling2D((2, 2), padding='same')(x)

# Bottleneck
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)

# Decoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
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)

# Compile the model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

print(autoencoder.summary())
```

1. Importing necessary modules: This imports various layers and the model class from Keras (which is a part of TensorFlow).

2. Setting the input layer: This line creates an input layer for the neural network. The shape (32, 32, 3) corresponds to the shape of CIFAR-10 images, which are 32x32 pixels and have 3 channels (RGB).

3. Building the Encoder: The encoder compresses the input into a compact representation.

Conv2D: This adds 2D convolutional layers. The first argument specifies the number of filters, the second is the kernel size.

activation='relu': Uses ReLU (Rectified Linear Unit) activation function.

padding='same': Ensures the output has the same spatial dimensions as the input by padding the edges of the input.

OUTPUT of CODE 4:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 32)        896

 max_pooling2d (MaxPooling2  (None, 16, 16, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 16, 16, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 8, 8, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 8, 8, 128)         73856

 conv2d_3 (Conv2D)           (None, 8, 8, 64)          73792

 up_sampling2d (UpSampling2  (None, 16, 16, 64)        0
 D)

 conv2d_4 (Conv2D)           (None, 16, 16, 32)        18464

 up_sampling2d_1 (UpSamplin  (None, 32, 32, 32)        0
 g2D)

 conv2d_5 (Conv2D)           (None, 32, 32, 3)         867

=================================================================
Total params: 186371 (728.01 KB)
Trainable params: 186371 (728.01 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

MaxPooling2D: Pooling layers are used to down-sample the spatial dimensions.

By the end of this encoder, the spatial dimensions of the image have been reduced by a factor of 4 (two max pooling layers with size 2x2).

4. Bottleneck: This is the compressed representation of the input data.

5. Building the Decoder: The decoder reconstructs the data from the compact representation back to its original form. UpSampling2D: Upsamples the spatial dimensions, essentially the opposite of MaxPooling2D.

The final layer has 3 filters, corresponding to the RGB channels of the CIFAR-10 images. The activation function here is sigmoid since the model will be trained with a binary_crossentropy loss, making the output range between 0 and 1.

6. Compiling the Model: The Model class creates the autoencoder model, taking the input and output layers.

The compile method configures the model for training. It uses the Adam optimizer and binary cross-entropy as the loss function.

7. Print Model Summary: The summary () function prints a detailed summary of the model's architecture, showing each layer's type, shape, and the number of parameters.

CHAPTER THREE (TRAINING AND TESTING THE MODEL ON THE NOISY-CLEAN IMAGE PAIRS AND A NEW SET OF NOISY IMAGES)

CODE 5:

```
# Normalize the images to [0, 1]
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# For the sake of demonstration, let's use Gaussian noise
train_images_noisy = add_gaussian_noise(train_images)
test_images_noisy = add_gaussian_noise(test_images)
# Training parameters
epochs = 1
batch_size = 128

# Train the autoencoder
history = autoencoder.fit(
    train_images_noisy, train_images,
    epochs=epochs,
    batch_size=batch_size,
    shuffle=True,
    validation_data=(test_images_noisy, test_images)
)
```

OUTPUT of CODE 5:

```
391/391 [==============================] - 272s 696ms/step - loss: 0.6195 - val_loss: 0.6187
```

1. Image Normalization:

Images typically have pixel values in the range of [0, 255] for each channel (R, G, B). Normalizing these pixel values to the range [0, 1] helps in speeding up the convergence of training and can sometimes lead to better performance.

- `.astype('float32')`: Converts the data type of the images to `float32`. This is necessary to allow decimal points after division.

- `/ 255.0`: Divides every pixel value by 255, transforming the range from [0, 255] to [0, 1].

2. Adding Gaussian Noise:

For this demonstration, Gaussian noise is added to the images using the previously defined `add_gaussian_noise()` function. This results in two new sets of images (`train_images_noisy` and `test_images_noisy`), where the original images have noise added to them.

3. Training the Autoencoder:

Here, we're setting up training parameters:

- `epochs`: Number of times the training algorithm will cycle through the entire training dataset.

- `batch_size`: Number of samples used in each iteration to update the model's weights.

This is the main training loop for the autoencoder:

- `train_images_noisy`: This is the set of images with noise that the autoencoder receives as input.

- `train_images`: This is the set of clean images the autoencoder tries to reproduce (i.e., the target output).

- `epochs` and `batch_size`: These parameters were defined earlier.

- `shuffle=True`: This ensures that the training data is randomly shuffled before each epoch, which can lead to better generalization.

- `validation_data=(test_images_noisy, test_images)`: The noisy test images are used for validation during training, and the model's output is compared to the clean test images to measure its performance on unseen data.

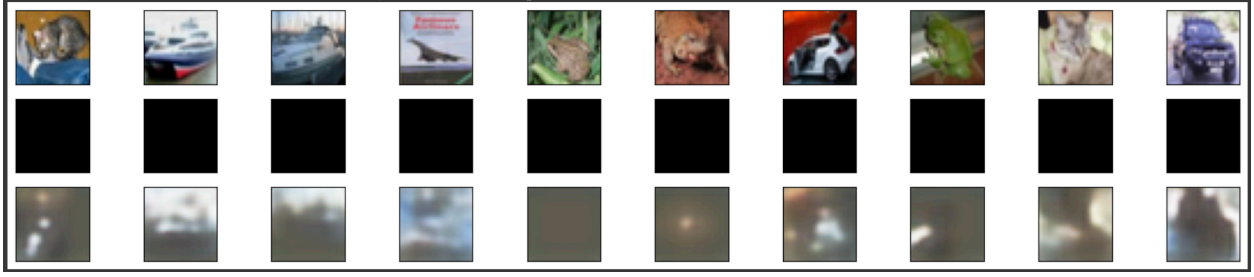
The `**fit()**` function returns a `history` object that contains details about the training process, such as the training and validation loss values after each epoch.

CODE 6:

```
# Predict the Autoencoder output from corrupted test images
test_images_decoded = autoencoder.predict(test_images_noisy)

# Display the original, noisy, and decoded images
n = 10  # number of images to be displayed
plt.figure(figsize=(20, 4))

for i in range(n):
    # Original images
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(test_images[i].reshape(32, 32, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Noisy images
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(test_images_noisy[i].reshape(32, 32, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Decoded (denoised) images
    ax = plt.subplot(3, n, i + 1 + n*2)
    plt.imshow(test_images_decoded[i].reshape(32, 32, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(test_images.reshape(-1),
test_images_decoded.reshape(-1))
print(f"Mean Squared Error (MSE) for the denoised images: {mse}")
mse_per_image = [mean_squared_error(original.reshape(-1),
denoised.reshape(-1))
                 for original, denoised in zip(test_images,
test_images_decoded)]

# Print MSE for the first 10 images
for i, mse_img in enumerate(mse_per_image[:10]):
    print(f"MSE for test image {i+1}: {mse_img}")
```

OUTPUT of CODE 6:

```
313/313 [==============================] - 16s 51ms/step
```



```
Mean Squared Error (MSE) for the denoised images: 0.03025410696864128
MSE for test image 1: 0.029519854113459587
MSE for test image 2: 0.026354799047112465
MSE for test image 3: 0.029265666380524635
MSE for test image 4: 0.027996337041258812
MSE for test image 5: 0.02923966385424137
MSE for test image 6: 0.024497078731656075
MSE for test image 7: 0.06544875353574753
MSE for test image 8: 0.014314289204776287
MSE for test image 9: 0.033988986164331436
MSE for test image 10: 0.029081866145133972
```

Prediction and Visualization:

1. Prediction:

This line predicts the denoised output for the noisy test images using the trained autoencoder. The result (`test_images_decoded`) contains the cleaned-up versions of `test_images_noisy`.

2. Visualization Setup:

The code sets up to display 10 images side-by-side with a figure size of (20, 4) inches.

3. Visualization Loop:

The for loop that follows constructs three rows of images:

Each image in the loop is reshaped to its original size of (32, 32, 3) before being displayed. The axes for each subplot are hidden using `ax.get_xaxis().set_visible(False)` and `ax.get_yaxis().set_visible(False)`.

Model Evaluation using Mean Squared Error (MSE):

1. Overall MSE Calculation:

Here, the Mean Squared Error between the actual clean images (`test_images`) and the denoised images from the autoencoder (`test_images_decoded`) is calculated. The `.reshape(-1)` method is used to flatten the 3D image data into 1D arrays, which makes it easier to compute the MSE over all pixels.

2. MSE Calculation Per Image:

This line computes the MSE for each individual image in the test set. A list comprehension loops through pairs of original and denoised images, flattens them, and calculates the MSE for each pair.

3. Print MSE for Specific Images:

The code then prints the MSE values for the first 10 images in the test set.

In essence:

The code achieves three main objectives:

- Uses the trained autoencoder to predict denoised versions of the noisy test images.

- Visually compares the original, noisy, and denoised images.

- Evaluates the denoising performance of the autoencoder quantitatively using the MSE between the original and denoised images, both overall and for individual test samples.

CONCLUDING REMARKS

Concluding our explorative journey into the domain of image denoising using Convolutional Autoencoders, several observations and insights have emerged. First and foremost, neural network-based denoising techniques, as showcased in our experiment using the CIFAR-10 dataset, offer a potent blend of effectiveness and adaptability. By training the model on pairs of clean and noisy images, it was conditioned to discern and learn inherent image structures, all while discarding the superficial noise patterns. Our use of the Mean Squared Error (MSE) metric to gauge the model's performance provided a quantitative perspective on its denoising capabilities, shedding light on areas of success and potential improvement.

The potential of deep learning in the arena of image denoising is vast, and our experiment serves as a testament to this claim. However, the road ahead is long, with numerous avenues for exploration and refinement. Different neural architectures, innovative loss functions, and advanced training methodologies await experimentation. Moreover, diversifying the experiment to encompass various datasets, noise types, and even exploring real-world noisy images can broaden our understanding of the model's versatility. As the world of deep learning continues to expand, its application to image denoising promises to push the boundaries of what's possible, ensuring clearer, more accurate images for a myriad of applications.

BIBLIOGRAPHY

1. https://colab.research.google.com/drive/1oISwduW9JbxqPmLIZsZedVCB0LzJGJ_l?usp=sharing
2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). "ImageNet classification with deep convolutional neural networks." Advances in neural information processing systems, 25. https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
3. CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html
4. Goodfellow, I., Bengio, Y., & Courville, A. (2016). "Deep Learning." MIT press. https://www.deeplearningbook.org/
5. He, K., Zhang, X., Ren, S., & Sun, J. (2016). "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf
6. Goodfellow, I., Bengio, Y., & Courville, A. (2016). "Deep Learning." MIT Press.
7. Chollet, F. (2017). "Deep Learning with Python." Manning Publications Co.
8. Bishop, C. M. (2006). "Pattern Recognition and Machine Learning." Springer.
9. Hastie, T., Tibshirani, R., & Friedman, J. (2009). "The Elements of Statistical Learning: Data Mining, Inference, and Prediction." Springer Series in Statistics.

APPENDIX (if necessary)