Project: Removing noise from the images

In this module, we will start with the CIFAR-10 data set (Present in Google drive)but this time we will introduce some random noise in each of the images. To initiate, let us read the images in the environment:

1. *# Importing basic libraries*
2. import pandas as pd
3. import numpy as np
4. import matplotlib.pyplot as plt
5. from PIL import Image
6. import os
7. *# Reading all the images in a python list*
8. img\_arr = []
9. for i in range(1, 151):
10. img\_path = os.path.join('cifar10/'+str(i) +'.png')
11. img = np.array(Image.open(img\_path))/255. *# Scaling*
12. img\_arr.append(img)
13. *# Converting back to numpy array*
14. img\_arr = np.array(img\_arr)
15. img\_arr.shape

(150, 32, 32, 3)

So, as you can observe in the above code, we have used only 150 CIFAR-10 dataset images and stored all of these 32x32x3 dimensional images to a numpy array. Now, we can add noise to each of these images:

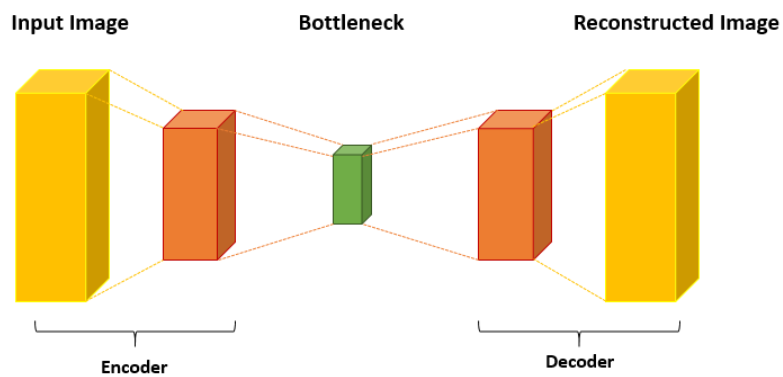
1. *# Original image*
2. plt.imshow(img\_arr[4])
3. plt.show()
4. *# Adding random noise to the images*
5. noise\_factor = 0.05
6. noisy\_imgs = img\_arr + noise\_factor \* np.random.normal(size=img.shape)
7. *# Image with noise*
8. plt.imshow(noisy\_imgs[4])
9. plt.show()

Introducing Auto-Encoders

An auto-encoder forms a part of unsupervised learning in the neural network domain. They are non-linear and helps in applications like data compression, image denoising, watermark removal from images and much more. Their applications are similar to that of PCA (Principal Component Analysis) except the fact that PCA works with linear transformation and auto-encoders are based on non-linear transformation.

An auto-encoder as the name suggests is made up of two modules as described below:

**Encoder**: This module helps in breaking down the input into a smaller size (often compresses it) into something named as latent-space or bottleneck which illustrates the point of maximum compression. The compression which takes place isn't the lossless rather lossy which means that the reconstructed image in the output will not be exactly the same bits by bits as the provided input.

**Decoder**: This module helps to decode the encoded input from the latent-space. If the decoder is able to reconstruct almost a similar image as the input then you can consider the model as robust. However, auto-encoders are data specific, therefore, if you train it on the data set of flowers and assume it to work on the vehicles data set then you'd receive wrong results.

The basic architecture of an auto-encoder is shown in the image below:There are various auto-encoders based on their applications in use like convolutional, sparse, contractive, and as usual deep auto-encoders.

Implementation: Image denoising using auto-encoders

We will continue the codes which are used to add the noise to the dataset. Let us first add the Keras modules:

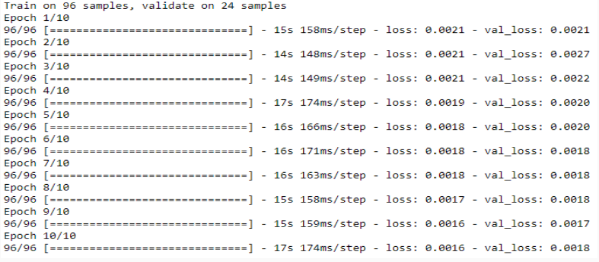
1. from keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
2. from keras.models import Model

Now, let us design the auto-encoder having two downsampling layers in encoding with filters 32 and 64, followed by a final layer with 128 filters. In the decoding module, we have 128 and 64 filters in the first two upsampling layers followed by a final softmax layer with 3 nodes (representing each color channel).

1. def auto\_encoder(img):
2. *# Encoder module*
3. f\_size = 3 *# filter size*
4. p\_size = 1 *# pool size*
5. conv\_1 = Conv2D(32, (f\_size, f\_size), activation='relu', padding='same')(img)
6. pool\_1 = MaxPooling2D(pool\_size=(p\_size, p\_size))(conv\_1)
7. conv\_2 = Conv2D(64, (f\_size, f\_size), activation='relu', padding='same')(pool\_1)
8. pool\_2 = MaxPooling2D(pool\_size=(p\_size, p\_size))(conv\_2)
9. conv\_3 = Conv2D(128, (f\_size, f\_size), activation='relu', padding='same')(pool\_2)
10. *# Decoder module*
11. conv\_4 = Conv2D(128, (f\_size, f\_size), activation='relu', padding='same')(conv\_3)
12. up\_1 = UpSampling2D((p\_size, p\_size))(conv\_4)
13. conv\_5 = Conv2D(64, (f\_size, f\_size), activation='relu', padding='same')(up\_1)
14. up\_2 = UpSampling2D((p\_size, p\_size))(conv\_5)
15. decoded = Conv2D(3, (f\_size, f\_size), activation='sigmoid', padding='same')(up\_2)
16. return decoded

The above function holds the complete architecture of the auto-encoder to be used in this scenario. Now, let us compile and train the model on 120 images out of which 20% are kept for validation for 10 epochs and remaining 30 images are left for test purposes.

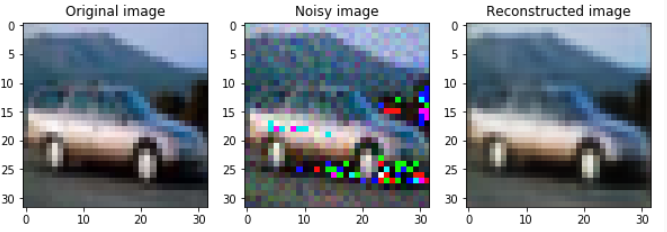
1. *# Calling the network design function and compiling the model*
2. img = Input(shape=(32, 32, 3))
3. model = Model(img, auto\_encoder(img))
4. model.compile(loss='mean\_squared\_error', optimizer='adam')
5. *# Training the model*
6. model.fit(noisy\_imgs[:120], img\_arr[:120], epochs=10, validation\_split=0.2)

Let us perform the prediction on training and test data:

1. pred = model.predict(img\_arr)

**Training data prediction**

1. plt.figure(figsize=(10, 5))
2. ax1 = plt.subplot2grid((1, 3), (0,0))
3. ax1.set\_title('Original image', fontsize='large')
4. ax1.imshow(img\_arr[4])
5. ax2 = plt.subplot2grid((1, 3), (0,1))
6. ax2.set\_title('Noisy image', fontsize='large')
7. ax2.imshow(noisy\_imgs[4])
8. ax3 = plt.subplot2grid((1, 3), (0,2))
9. ax3.set\_title('Reconstructed image', fontsize='large')
10. ax3.imshow(pred[4])
11. plt.show()



**Test data prediction**

1. plt.figure(figsize=(10, 5))
2. ax1 = plt.subplot2grid((1, 3), (0,0))
3. ax1.set\_title('Original image', fontsize='large')
4. ax1.imshow(img\_arr[131])
5. ax2 = plt.subplot2grid((1, 3), (0,1))
6. ax2.set\_title('Noisy image', fontsize='large')
7. ax2.imshow(noisy\_imgs[131])
8. ax3 = plt.subplot2grid((1, 3), (0,2))
9. ax3.set\_title('Reconstructed image', fontsize='large')
10. ax3.imshow(pred[131])
11. plt.show()

As we can observe the given auto-encoder design has provided great results. However, it is recommended to always set the optimal hyperparameters before you finalize the model.