Auto-encoders and K-means clustering

14-Nov-21

1. Overview

This project aims at building an Auto-encoder and using its intermediate output for K-means clustering. The Auto-encoder consists of an encoder and a decoder developed using neural networks. The objective is to find an encoding-decoding scheme by comparing the initial data and back-propagating the error to update the weights of the network.

2. Dataset

Dataset used is same as for part 1, which is Cifar-10 dataset.

3. Programming Languages and tools used

Apart from the libraries used in Part 1, some additional libraries/tools used for Part 2 include:
Keras – used to take input; build layers like 2D convolution and 2D convolution transpose; build, compile, fit and predict the model
Matplotlib – to display images
Sklearn.cluster – to implement K-means clustering
Sklearn.metrics – to calculate silhouette score

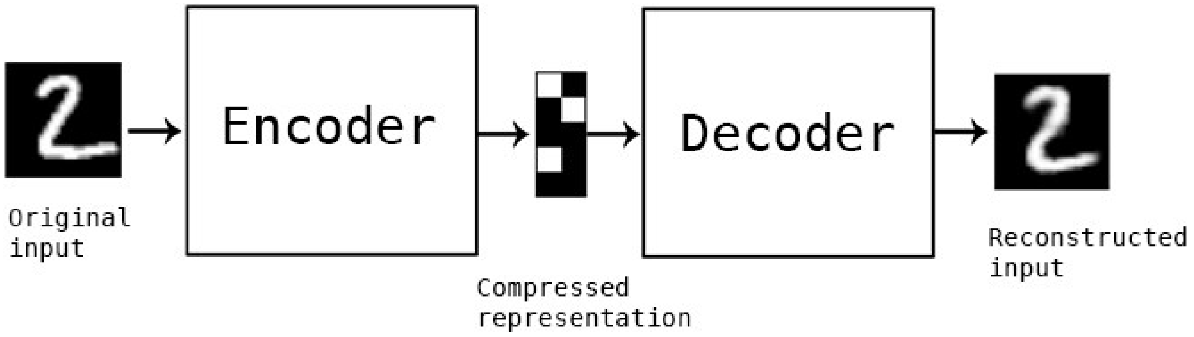
4. Data pre-processing

The dataset was loaded into x_train, x_test, y_train, y_test variables using keras function. The x_train dataset is then normalized (dividing by 255) and reshaped to get better results. This dataset (50000 images) was used to train the encoder and decoder.

5. Auto-encoders

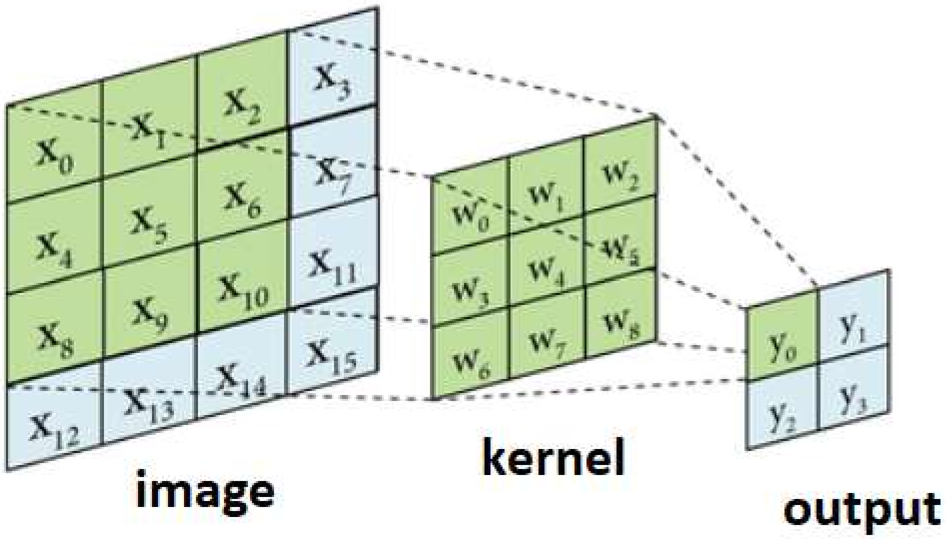
Auto-encoders are special type of neural networks where we produce an output which is same as the input. A typical auto-encoder consists of an encoder and a decoder. The encoder compresses the data to its lower dimensional latent representation which is also known as bottleneck. Because the input is compressed to its maximum at this moment, the bottleneck is also known as the ‘maximum point of compression’. The decoder then reconstructs the data from this latent representation to higher dimensional output in order to get the original data from fewer dimensions. The auto-encoders are designed to perform compression on highly specialized data meaning that auto-encoders used on animal images might not give efficient output on plant image dataset. This is different as compared to widely used compression techniques like JPEG, etc. which are independent of any features.

Convolutional autoencoders, denoising autoencoders, variational autoencoders, and sparse autoencoders are examples of autoencoders.
In this project, we are working on image dataset. Hence, we are using convolutional neural network (CNN) to build the encoder and decoder since CNNs work better on image datasets.



6. **Related concepts**

1. Convolution:
   2D convolution is starting with a kernel, which is simply a small matrix of weights. This kernel "slides" over the 2D input data, doing elementwise multiplication with the portion of the input it is currently on, and summing the results into a single output pixel. The kernel repeats this process for every location it slides over, which finally converts a 2D matrix of features into another 2D matrix of features.



   Mathematical formulation of 2-D convolution is given by

$$y[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n]$$

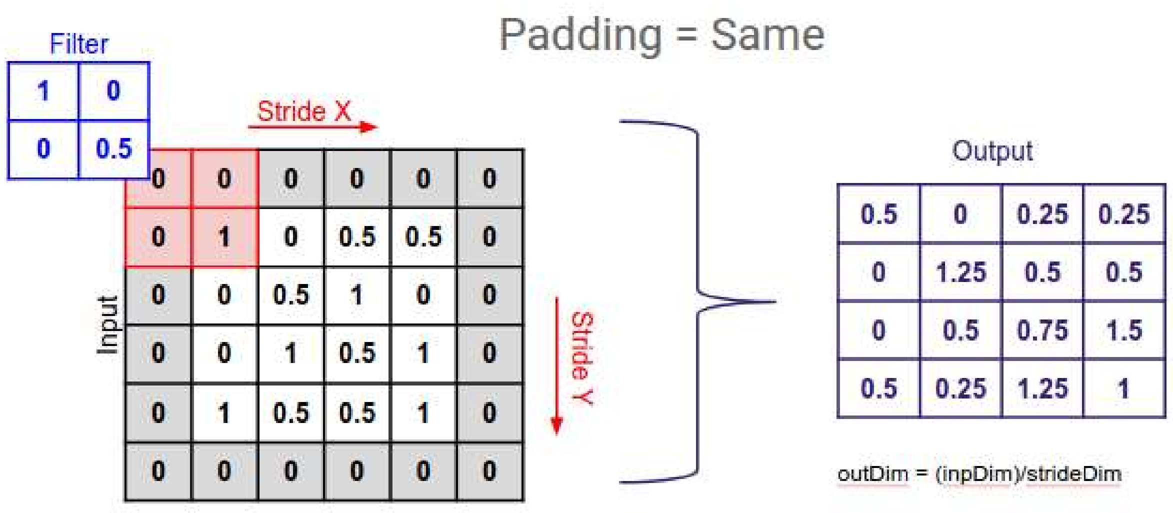
Where, x represents the input image matrix to be convolved with the kernel matrix h to result in a new matrix y, representing the output image. Here, the indices i and j are concerned with the image matrices while those of m and n deal with that of the kernel. If the size of the kernel involved in convolution is 3 × 3 (which I have used during the project), then the indices m and n range from -1 to 1.

2. Padding:

   Padding essentially means adding non-existent elements to an image template in order to match it with kernel size so that we are able to compute the convolution. It is mostly required for pixels at border of the images which do not have row/column of pixels above/below/besides them. Padding generally means appending the edges with extra, "fake" pixels (usually of value 0, hence the oft-used term "zero padding"). This way, the kernel when sliding can allow the original edge pixels to be at its center.

3. Strides:

   When we need to reduce the spatial dimensions of the input, we skip some of the slide locations of the kernel. A stride of 1 means the filter is moving by 1 row or column after every iteration which is the normal way. However, if we use stride of 2 or more, then the kernel skips over a pixel and slides over every 2nd pixel thus downsizing the image/data by a factor of 2. Similarly for stride =3 or even more. In this project, I have used stride =2. The Keras layer functions include this as an argument.



4. Optimizer:

   I have used Adam optimizer in this project. It stands for Adaptive Moment Estimation. It is basically a combination of 'Gradient descent with momentum' and the 'RMSP' algorithm. It requires less memory and more efficient. It is widely used for large datasets and deep CNN.

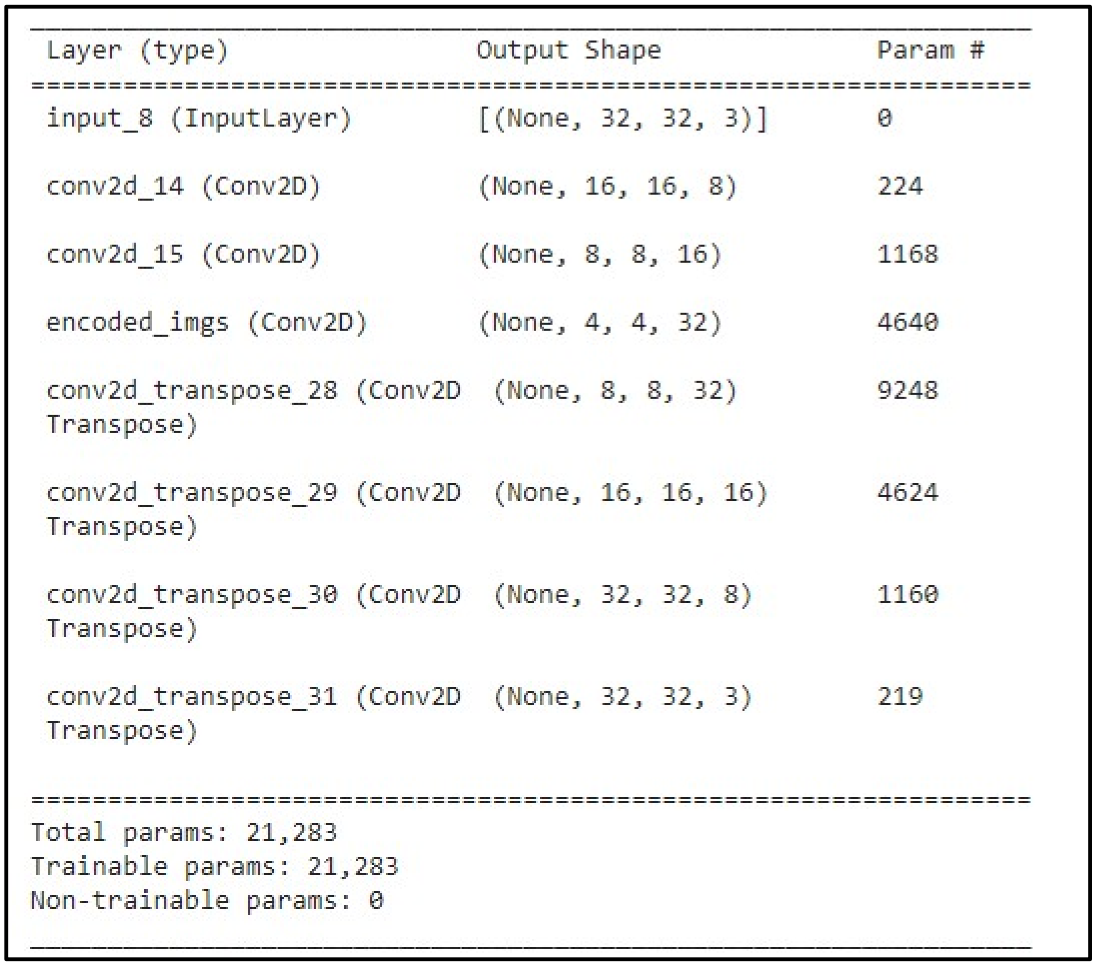
5. Activation functions:

   I have used ReLU activation function. It stands for rectified linear activation unit. The ReLU function is simple and it consists of no heavy computation .The model can, therefore, take less time to train or run. One more important property that we consider the advantage of using ReLU activation function is sparsity, which helps us in implementing Autoencoders.

   The sigmoid function is a special form of the logistic function and is usually denoted by σ(x) or sig(x). It is given by:
   $\sigma(x) = 1/(1+\exp(-x))$
   It is also called a squashing function as its domain is the set of all real numbers, and its range is (0, 1). Hence, if the input to the function is either a very large negative number or a very large positive number, the output is always between 0 and 1.

7. Methodology

After preprocessing the data, an input structure was defined using keras.Input( ) function. Since I used CNN, encoder layers were built using layers.Conv2D( ). I used 3 layers with 8, 16 and 32 nodes respectively. Each layer used Rectified Linear Unit 'relu' activation function with strides =2 in order to down-sample the image. The decoder was designed using 4 CNN layers with 32, 16, 8 and 3 nodes respectively in each layer. Here too, 'ReLU' activation function was used with strides =2. However, in the decoder layer, in order to up-sample the image I used layers.Conv2DTranspose( ) method. The last layer (output layer with 3 nodes) used sigmoid activation function in order to restrict the output between [0, 1]. The padding was kept as 'same' throughout the structure in order to maintain uniformity. The model snapshot looked as below:

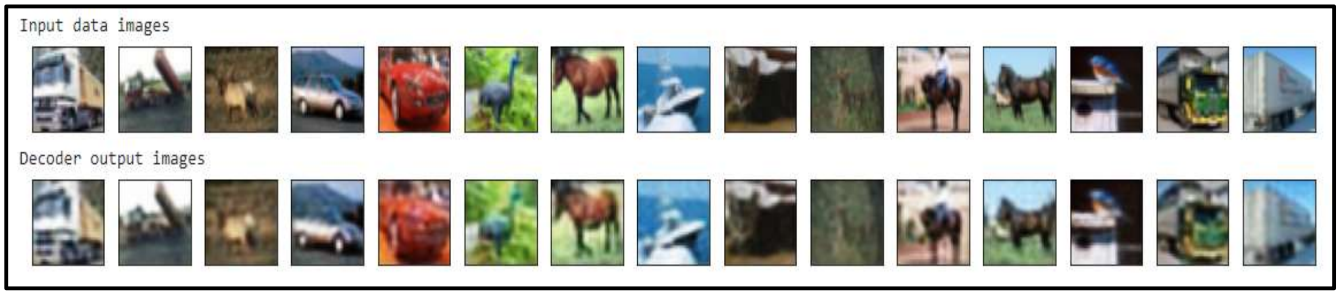


```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_8 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_14 (Conv2D)          (None, 16, 16, 8)         224

 conv2d_15 (Conv2D)          (None, 8, 8, 16)          1168

 encoded_imgs (Conv2D)       (None, 4, 4, 32)          4640

 conv2d_transpose_28 (Conv2D  (None, 8, 8, 32)         9248
 Transpose)

 conv2d_transpose_29 (Conv2D  (None, 16, 16, 16)       4624
 Transpose)

 conv2d_transpose_30 (Conv2D  (None, 32, 32, 8)        1160
 Transpose)

 conv2d_transpose_31 (Conv2D  (None, 32, 32, 3)        219
 Transpose)

=================================================================
Total params: 21,283
Trainable params: 21,283
Non-trainable params: 0
_________________________________________________________________
```

After this, I have compiled the model using optimizers like adam and calculated binary cross-entropy loss. The model was fit using training dataset to calculate the error in every iteration and backpropagate it to update the weights. Prediction of the model was calculated over the input dataset. All this was achieved using different functions provided by Keras library. The reconstructed output from decoded was displayed along with the input images (shown in results below) with the help of Matplotlib.
Now in order to implement the K-means part, I created another model where the input was training dataset images and the output was the encoded images. This model was then reshaped and fed to the KMeans( ) function to perform clustering. Lastly, Silhouette score was calculated using sklearn function (shared in results below).

8. Results

I tried to compare the training image dataset with the output produced by the auto-encoder. For this, I printed out the images from the original Cifar-10 dataset and also the images which were produced by the decoder i.e. the reconstructed images. The below results were obtained:



The encoder output (which was the input for the decoder part) was fed to a K-means algorithm to cluster the images. I could successfully reach an Average Silhouette Coefficient (ASC) value of 0.053.

```
Silhouette score:  0.052613564
```

9. Credits

1. https://www.datacamp.com/community/tutorials/autoencoder-keras-tutorial
2. https://blog.keras.io/building-autoencoders-in-keras.html
3. https://www.tensorflow.org/tutorials/generative/autoencoder
4. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D
5. https://keras.io/api/models/model_training_apis/
6. https://keras.rstudio.com/reference/fit.html
7. https://keras.io/api/metrics/
8. https://keras.io/api/models/model/
9. https://keras.io/api/layers/core_layers/input/
10. https://towardsdatascience.com/silhouette-coefficient-validating-clustering-techniques-e976bb81d10c
11. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html
12. https://stackoverflow.com/questions/51138686/how-to-use-silhouette-score-in-k-means-clustering-from-sklearn-library
13. https://towardsdatascience.com/k-means-clustering-with-scikit-learn-6b47a369a83c
14. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
15. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose