Autoencoders for Image Retrieval

Adarsh Prakash
adarshpr@buffalo.edu — 5020 8760

December 8, 2017

Abstract

Autoencoders are effective unsupervised learning models that first encode an input into a lower dimensional representation. This representation, which constitutes the features in the input, can be particularly useful in the domain of image processing. Contents of a picture can be encoded to a compressed representation, which can be used to build effective image retrieval applications. This paper compares and evaluates several architectures, models and training approaches necessary to build such a retrieval system. Let us start with a brief introduction of Autoencoders and architectures that are effective for image inputs, and then move on to evaluate training approaches to generate quality encoded representations. Finally, let us look at an end-to-end system, that brings together these ideas, to perform image (reverse) search.

1 Introduction

An image retrieval system is typically an application where users can construct queries such as "Find me cars like this one" or "Show medical history for brain injury cases with CT scans similar to this one". The problem at the core of such applications is finding images that are *visually similar*. There are many ways to compute visual similarity but not all are efficient. Simple and classic measures such as *Euclidean distance* or *Manhattan distance* only compute the difference between pixel values while totally ignoring visual queues. There are other sophisticated approaches in computer vision such as *SIFT* [1], *Shock Graph* [2] and *Geometric Blur* [3] that promise scale and translation invariant results. However, they are not robust challenges such as - view point variation, illumination, occlusion and deformation. This is where *deep neural networks* are effective, particularly convolutional neural networks.

Deep neural networks can learn many levels of nonlinearity in the images to extract and represent features in an image. *Autoencoders*, a particular type of deep neural networks, learn to reconstruct images by first transforming an input into a hidden representation of its features. These representations can be used to compute semantic similarity in the images.

In the further sections, we will briefly discuss Autoencoders and a comparison of neural network architectures, each of which are progressively better suited for image retrieval than the previous ones. We will then discuss efficient training approaches to build an end-to-end system.

1.1 Related Work

Alex Krizhevsky and Geoffrey Hinton [4] were one of the first to propose the use of deep autoencoders for content-based image retrieval. They utilized of RBMs for encoder and decoder to reduce images to 256-bit codes for effective retrieval on CIFAR10 dataset. Babenko et al., [5] utilized PCA to create neural codes for efficient compression of representation while learning more features on Image-Net dataset. Wei Liu [6] from IBM introduced Supervised Deep Hashing technique for effective retrieval. Lin et al., suggested *Approximate Nearest Neighbor* (ANN) instead of linear search to speedup retrieval of compressed codes from Convolutional Neural Networks. The current work borrows the idea of ANN from this paper.

1.2 Datasets

MNIST This dataset composed of 28x28 greyscale images of handwritten digits will be used for comparing the effectiveness of various models. This will only be used for evaluation purposes and the previously proposed end-to-end system will make use of CIFAR4 dataset.

CIFAR4 This is a subset of the CIFAR10 dataset created to demonstrate the ideas presented in this paper. The original CIFAR10 dataset contains 32x32 RGB images of real worlds objects of 10 different categories. CIFAR4, however, includes train and test images of four categories, namely - Airplanes, Ships, Automobiles (cars) and Trucks. No other modifications have been made to the original dataset.

2 Autoencoders

An Autoencoder is a deep neural network that learns to reconstruct its input. It typically has an *encoder* part which learns to represent the features in an input as a vector in latent space. This is followed by a *decoder* part which learns to reconstruct the input image from this hidden representation. Figure 1 demonstrates this idea applied to an image input.

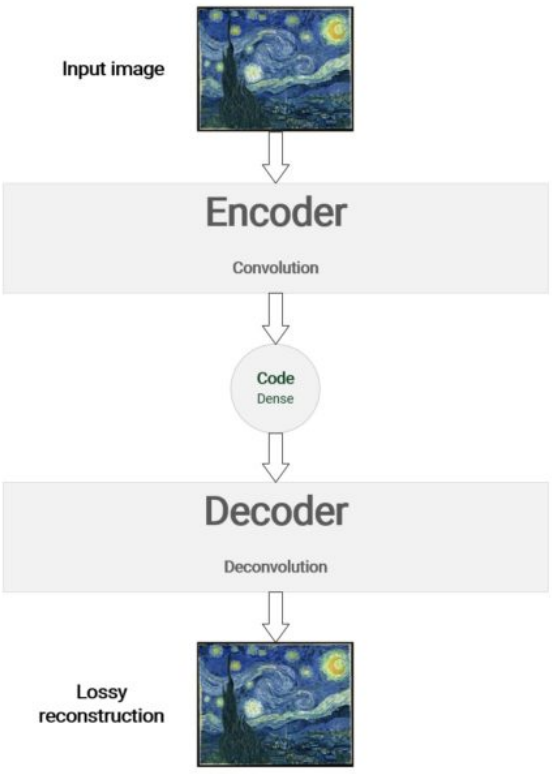

Figure 1: Encoder-Decoder architecture

Notice that the size of hidden representation can be smaller than the original input, in which case the network is called *undercomplete autoencoder*. Typical applications of such a network is image compression and image retrieval. It is also possible that the size hidden representation can be larger than input size, in which case it is called a *overcomplete autoencoder*. This is primarily used for *sparse encoding* an input.

2.1 Evaluation on Images

Now that we have a brief understanding of Autoencoders, let us choose an appropriate autoencoder of adequate complexity that is capable of transforming images to encoded vectors. Three models of varying complexity were built to fit MNIST dataset and evaluated to understand how they perform.

Model A (**Simple Autoencoder**), constitutes only a single hidden layer per encoder and decoder. This was trained with Adadelta optimizer for 50 epochs over 50k MNIST images. Model B (**Deep Autoencoder**), constitutes three hidden layers per encoder and decoder. This model was found to perform best when trained with Adadelta optimizer for 60 epochs over the same train set. Model C (**Convolutional Autoencoder**), consists of three convolutional layer blocks with pooling layers (maxpool) per block. A complementing network forms the decoder part of the model. This model was trained with Adam optimizer for 80 epochs over the same train set. The following plots and tables demonstrate the reconstruction quality and capacity for each of these models.

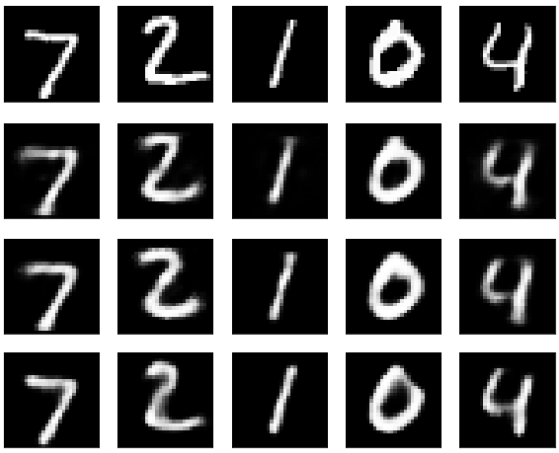

Figure 2: Row 1 - Original Image; Row 2 - reconstruction from Model A; Row 3 - reconstruction from Model B; Row 4 - reconstruction from Model C

Model (layers per encoder)	**MSE Error**
Model A (1 hidden layer)	0.1266
Model B (3 hidden layers)	0.1127
Model C (3 Conv-Pool blocks)	0.1015

Table 1: Comparison of reconstruction errors

It is clear from these results that Model C with convolutional layers performs best. Although, it is difficult to notice a significant difference in visual quality, the smaller differences in reconstruction error will be large offsets when applied to real world images. Also, please note that *these images are greyscale and real world images have RGB channels.*

We can conclude that an Autoencoder with convolutional layers would be a better candidate for our subsequent task of Image Retrieval. Given the relative complexity of CIFAR images when compared to MNIST training images, we would need additional convolutional layers.

3 Image Retrieval

Let us now see how Autoencoders can be utilized to build a Content Based Image Retrieval system.

3.1 Architecture

CIFAR images have objects with varying degree of scale, translation, rotation and occlusion. This demands a sophisticated model with the ability to extract features from images effectively. The proposed architecture consists of *four convolution blocks*. Each block has *two convolutional layers with 3x3 filters*, followed by a Max-Pooling layer with 2x2 filters. Dropout layers are placed after each convolution block to regularize the model. All the convolutional layers have a stride of 1 and padding to preserve initial size.

At the end of the fourth convolutional block, we flatten out 2048 features extracted from the input image and feed it to *three fully connected layers* which progressively reduce the size of hidden representation to 128 or 64. This encoded vector can be used to index and retrieve images upon query.

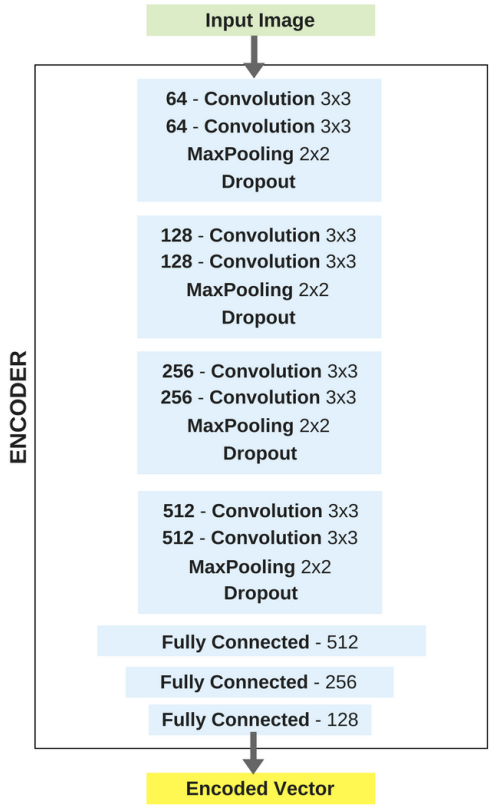

Figure 3: Illustration of how the encoder part of the network works

3.2 Training Approaches

Although it may seem straightforward, training this architecture is not trivial. One can take two approaches to train this model.

Approach #1 This involves stacking up the layers of both encoder (Fig 3) and decoder (a mirrored version of Fig 3), and training all the layers together. This is demonstrated in Fig 4 (left).

Approach #2 Pre-train the Convolutional layers for a different supervised task such as classification. Use these pre-trained Convolutional layers with fully connected layers from Fig 3, and fine tune this fully connected autoencoder to reconstruct extracted features instead of input image. This is demonstrated in Fig 4 (right).

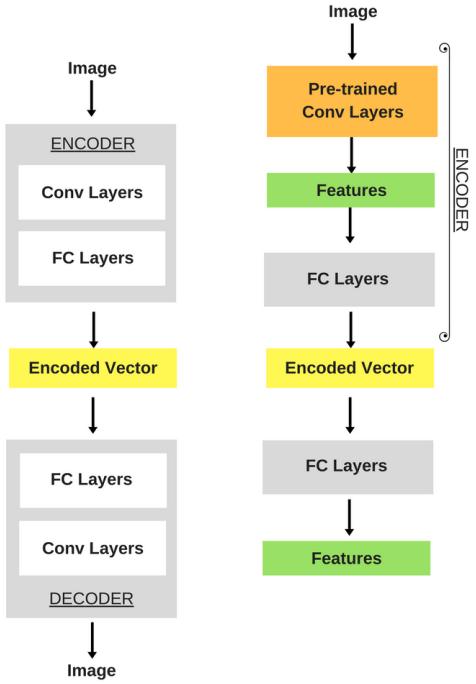

Figure 4: Two approaches to training - (left) training all layers end-to-end; (right) pre-training convolutional layers and post training the fully connected layers.

The first approach has several disadvantages that are not apparent at a glance:

- The mean-squared-error and binary-crossentropy loss functions are not constrained enough to train such a deep stack of convolutional layers.
- Gradient flow is limited at first layers. Decoder is better trained than encoder, which is not a desired outcome for Image Retrieval.
- Too many parameters and hence higher memory needs. Problematic for GPU training.

- Limited compression. Imagine reconstructing image back from a vector of size 64.
- Poor performance. Fails to overfit even a subset of dataset.
- Longer training times.

In contrast, there are several advantages to using pre-trained convolutional layers:

- Convolutional layers have already learnt how to extract features. They can be fine tuned only if necessary.
- Fully connected layers learn to compress the extracted features instead of input image. Learning objective is simpler.
- Allows for higher order of compression.
- Significantly faster.

3.3 Pre-training

The convolutional layers were pre-trained with a classification task on the CIFAR4 dataset. This classification model consisted of the four convolution blocks from Fig 3 and three fully connected layers with a softmax loss function. The model was trained to reach state-of-the art accuracy of **92.7%** on CIFAR dataset. The following plots and tables show the progress of learning steps.

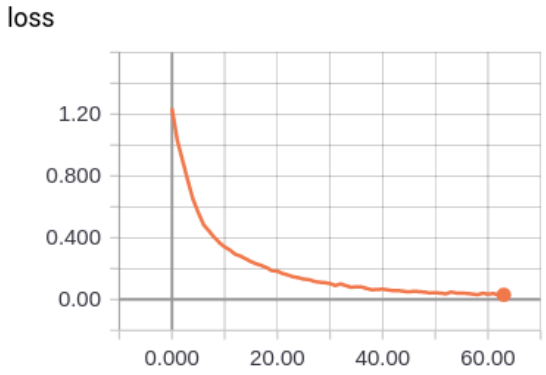

Figure 5: Loss vs Epochs - Pre-training classification model - learning progress

The fully connected layers had 4096.4096.4 hidden units. The weights were trained using Adam optimizer with a learning rate of 1e-4 and decay rate of 1e-6. Available GPU hardware permitted a batch size of 128 images per batch, with a training time of 80s per epoch (one pass through training set). The model was regularized with *Dropout* and Batch Normalization layers as described previously. Dropout for first two Convolution blocks was set to 0.25 while the last two blocks had a dropout of 0.5

It should be noted that the model was trained to its full capacity and compares with state-of-the art accuracies of models similar in architecture.

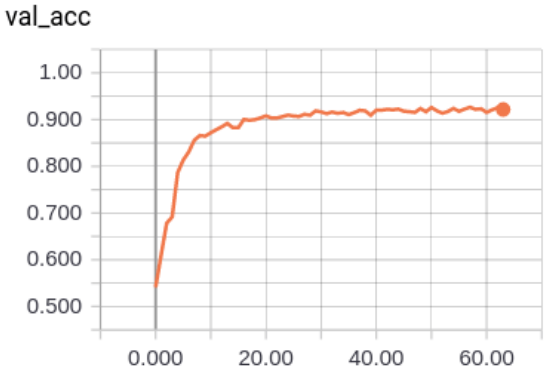

Figure 6: Validation Accuracy vs Epochs - Pre-training classification model

3.4 Post Training

The pre-trained convolutional layers were stacked back to their original architecture as show in Fig 3. The Conv Layers are now capable of precisely extracting features from the images. The fully connected layers and the complementary decoder network were trained to reconstruct image features as shown in Fig 4 (right). The following plots and tables show the progress of learning steps.

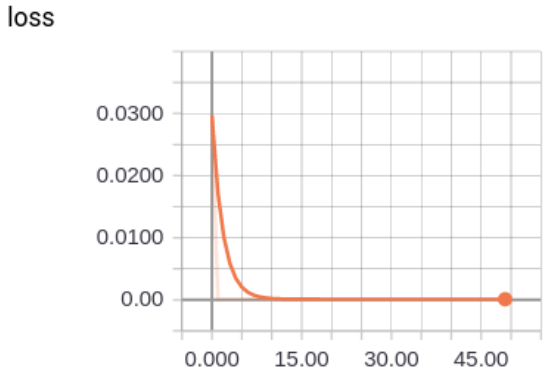

Figure 7: Loss vs Epochs - Post training fully connected autoencoder layers

The weights were learnt using *Adam optimizer* with a learning rate of 1e-4 and no decay. A batch size of 128 used and a time of 3s per epoch was observed.

We now have an efficient encoder network that can transform images to encoded vector representations of size 64 or 128 (as specified). In the subsequent sections we will discuss how we can index these compressed vectors and use this index to run an end-to-end application

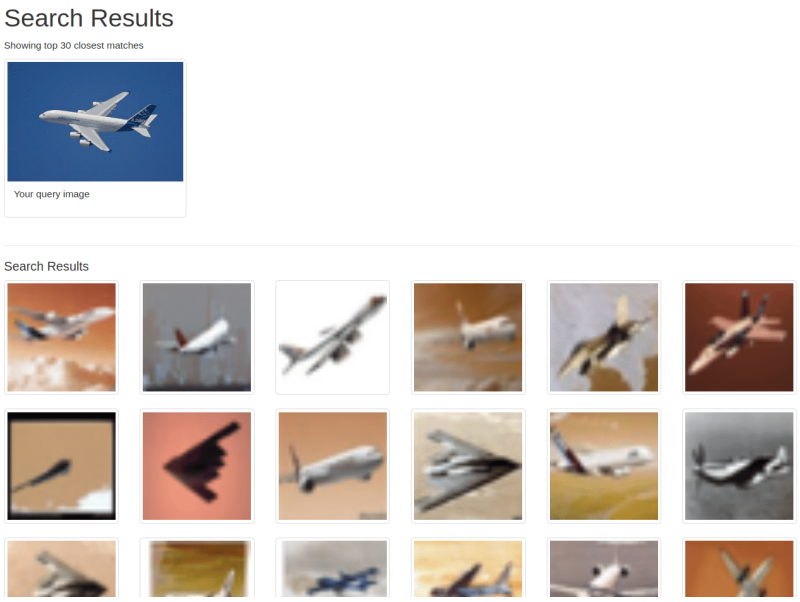

Figure 8: Image Retrieval - an example search usecase of the application

3.5 Index Construction

We have thus far taken efficient approaches to perform each task. Despite that, if we were to store the compressed image vectors in a flat file and perform linear search, that would prove to be a bottleneck. A linear search involves:

- Loading all the images onto memory.
- Computing vector distance against query vector.
- Ranking (sorting) all vector distances to get the closest results.

Clearly, this is would not work in a practical application with real-time user needs. An effective alternative approach would be **Approximate Nearest Neighbor** (ANN).

Approximate Nearest Neighbor indexes each image and divides the search vector space into sub-trees. Each sub-tree may be, optionally, divided further to reduce search space. At the base of the tree, ANN performs an approximation of Nearest Neighbor search to retrieve search results. ANN is thus a minor trade-off of precise results in return for improved speed and memory utilization. This trade-off is perfectly acceptable in case of image data.

4 An End-to-End Application

Bringing together all the pieces from previous sections, an end-to-end system has been implemented to search for CIFAR4 images. That is, a user is free to use any image to query the system for available images of Cars, Airplanes, Ships and Trucks. This application is attached with the source for this project.

A brief outline of how the system works:

- User visits the web application
- User uploads an image of interest, say a car, as search query
- The application presents the user with image matches from CIFAR4 dataset

A screenshot of the application works is illustrated in Fig 8.

4.1 System Evaluation

While it is not possible to evaluate the relevance of search results on a random image, we could leverage the classification labels from CIFAR4. We randomly choose an image from test set and search for the image results. We can compare the CIFAR labels of images in search results with the label of query image to obtain precision, recall and F1-score. For example, if we randomly pick an image from CIFAR test set, say *car*, if every search result has the same label *car* we can conclude the search results are accurate.

This evaluation was performed on the 4000 test images from CIFAR4 and a *F1 Score of 0.9829* was obtained. While it is not necessary, even when extensive index of 20 sub-trees was constructed with a vector size of 128, the size of the search index was 15.2 MB. The size of the original set of images was 83.4 MB. Ideally, we can index compressed image vectors of size 64 with 10 sub-trees for this dataset. This would create an index file of 3.8 MB (20x compression!)

5 Conclusion

In summary, the following were explored:

- Three different autoencder models were built and evaluated to quantify the effectiveness of Convoluational Autoencoders when compared to Simple and Deep Autoencoders for image data.
- Two types of approaches to training Convolutional Autoencoders were compared and evaluated.
- A classification model was built to pre-train Convolutional layers to extract features with an accuracy of 92.5% on CIFAR4.
- Pre-trained Convolution layers and fully connected layers were combined and an autoencoder was trained to transform images to compressed vector.
- A search index of compressed vectors was created using Approximate Nearest Neighbor.
- An end-to-end web application was built to query this index.

Note on Implementation

Languages Used: Python, Javascript, HTML, CSS
IDE: Jupyter Notebook
Libraries used:
Models - Tensorflow, Keras, Numpy adn Scikit-learn
Index Construction - Annoy
Web Application - Flask

These libraries need to be installed in order to run the web application attached with source.

Description of Files

pretrain-conv-layer.ipynb - This file contains the code to train a classification model on CIFAR4. It will save pre-trained convolution layers, that can be consumed later.
train-dense-layer.ipynb - This file trains the fully connected layers by utilizing features extracted from pre-trained layers.
build-search-index.ipynb - Creates a search index of CIFAR4 images using Approximate Nearest Neighbor technique. Index file is saved and then consumed by web application to perform image search.
other-models - This directory contains models used to compare and evaluate before arriving at the final architecture. This is discussed in Section 2 and 3 of this document.

References

[1] SIFT (Scale-Invariant Feature Transformation) - D. Lowe. Object recognition from local scale-invariant features. International Conference on Computer Vision, 1999.

[2] K. Siddiqi, A. Shokoufandeh, S. J. Dickinson, S. Zucker. Shock graphs and shape matching. International Journal of Computer Vision, 35(1), 1999.

[3] Geometric Blur - A. C. Berg, T.L. Berg, and J. Malik. Shape matching and object recognition using low distortion correspondence. IEEE Computer Vision and Patter Recognition (CVPR), 2005.

[4] Alex Krizhevsky, and Geoffrey E. Hinton. Using Very Deep Autoencoders for Content-Based Image Retrieval. ESANN 2011.

[5] A. Babenko, A. Slesarev, A. Chigorin and V. Lempitsky. Neural Codes for Image Retrieval, European Conference on Computer Vision (ECCV), Zurich, 2014

[6] Wei Liu. Hashing by Deep Learning. IBM T. J. Watson Research Center.

[7] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, Chu-Song Chen. Deep Learning of Binary Hash Codes for Fast Image Retrieval. CVPR Workshop 2015.