# Machine Learning Engineer Nanodegree

## Capstone Project

Mohammad Al-Fetyani

May 27, 2020

## 1 Definition

### 1.1 Project Overview

Image retrieval has always been an active area of research. In the past few years, the field of computer vision has grown exponentially due to the impressive progress in deep learning and recent development in technology, there is an increase in the usage of digital cameras, smartphone, and Internet. The shared and stored multimedia data are growing, and to search or to retrieve a relevant image from an archive is a challenging research problem [1]

The fundamental need of any image retrieval model is to search and arrange the images that are in a visual semantic relationship with the query given by the user. Most of the search engines on the Internet retrieve the images on the basis of text-based approaches that require captions as input [2]. The user submits a query by entering some text or keywords that are matched with the keywords that are placed in the archive. The output is generated on the basis of matching in keywords, and this process can retrieve the images that are not relevant.

Content-based image retrieval (CBIR) is a framework based on the visual analysis of contents that are part of the query image. To provide a query image as an input is the main requirement of CBIR and it matches the visual contents of query image with the images that are placed in the archive, and closeness in the visual similarity in terms of image feature vector provides a base to find images with similar contents. In CBIR, low-level visual features (e.g.,color, shape, texture, and spatial layout) are computed from the query and matching of these features is performed to sort the output [3].

After the successful implementation of the abovementioned models, CBIR and feature extraction approaches are applied in various applications such as medical image analysis, remote sensing, crime detection, video analysis, military surveillance, and textile industry. Figure 5 provides an overview of the basic concepts and mechanism of image retrieval [4].

### 1.2 Problem Statement

The problem stems from the *AI Meets Beauty Challenge 2020* [5] where the problem is stated as follows:

> *"Given a real-world image containing one beauty or personal care item, the task is to match the real-world example of this item to the same item in the Perfect-500K data set. This is a practical but extremely challenging task, given the limitation that only images from e-commerce sites are available in Perfect-500K and no real-world examples will be provided in advance."*

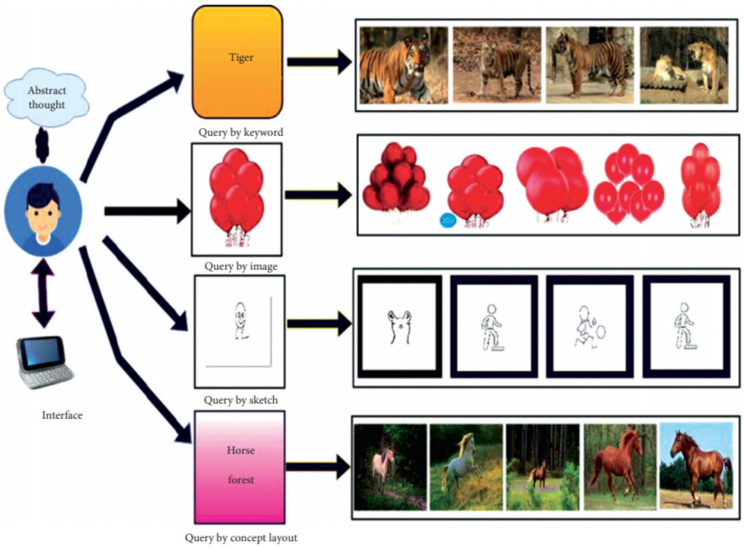

Figure 1: Pictorial representation of different concepts of image retrieval. [2]

This is an unsupervised problem and in this project, we will handle simpler problem since the dataset of the previous challenge is huge and hard to manage.

The problem we are tackling in this project can be simplified to a small manageable dataset. MNIST dataset is used in this problem because it is small and can be used to train models fast. The problem states that you are required to retrieve the most similar images to a given input image. In the perfect scenario, the retrieved images should belong to the same class of the query image. That is, if the end user inputs an image of digit 7, the model should return images of digit 7.

Autoencoders can be a great choice to tackle this problem since they don't require labels during training and can provide a model to extract features trained on the problem in hand. This is considered a much better choice than using an already trained models like VGG or ResNet to extract features for a certain domain that they were not trained on. For example, using ImageNet trained models to extract features from images in another field, such as medical images, can be a difficult task and may give inaccurate results. Since Autoencoders do not require labels during training, they can easily provide a model for feature extraction in any field if they are trained on images from that field.

Once the Autoencoder is trained on given images, it can be used to accurately extract features specified to the domain that was trained on, which is a big advantage of using Autoencoders. These features can be then compared with the features of query images and return the best matching images.

## 1.3 Metrics

This is an unsupervised problem where human evaluation is considered the best choice. However, since we have the labels of the images, we can use them to calculate the accuracy of a given model. That is, for the top $n$ images, calculate the accuracy as follows:

$$\text{accuracy} = \frac{\sum_{\text{query images}} \text{number of correctly retrieved images}}{\sum_{\text{query images}} \text{number of all retrieved images}} \tag{1}$$

where we calculate the correctly retrieved images over all query images and then divide it by all retrieved images over all query images.

This problem can be interpreted as a classification problem for evaluation purposes only, labels should not be used during training as in real-life problems labels are usually not provided. Since the dataset is balanced, the most simple evaluation metric to use is accuracy. Accuracy evaluates how well the model is able to correctly retrieve images. The correctly retrieve images are considered corrected if they belong to the same query image class.

One can argue about other metrics like recall, precision and F1-score, however, such metrics are considered for imbalanced datasets. As mentioned earlier, the dataset in hand is balanced and accuracy can be used to evaluate the model because it gives an indication of how well the model performs and it is easy to interpret.

It is to be noted that human evaluation is essential in a real-world problem and labels are provided in some real-world problems [5]. Also, the current widely adopted metric to evaluate image retrieval tasks is mAP that requires labels to be calculated even for unsupervised tasks. I used accuracy because the end user usually can understand what does it refer to and can easily interpret the results.

# 2 Analysis

## 2.1 Data Exploration

The MNIST database consists of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

The images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field. A sample of the dataset is provided in Figure 2.

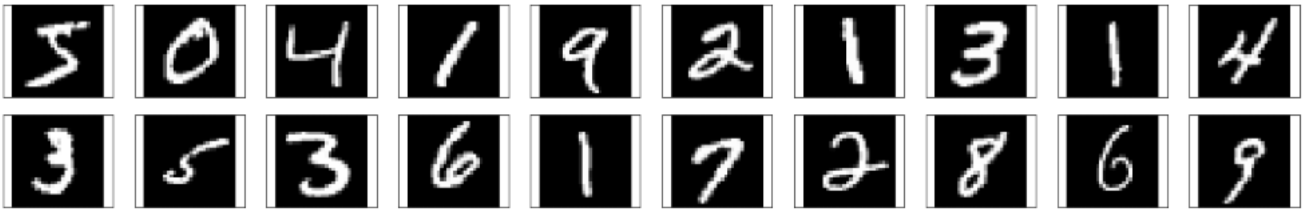

Figure 2: Sample of MNIST dataset.

## 2.2 Exploratory Visualization

The dataset has ten target classes that represent the numbers from 0-9 where the distribution of the digits in the training set is provided in Figure 3. It is clear that the dataset is balanced with no biases toward any digit.

For illustration purposes, number 1 has the highest frequency among others with a value of 6742 times followed by the number 7 which appeared 6265 times. Number 5 is the least-frequent with a value of 5421 occurrences.

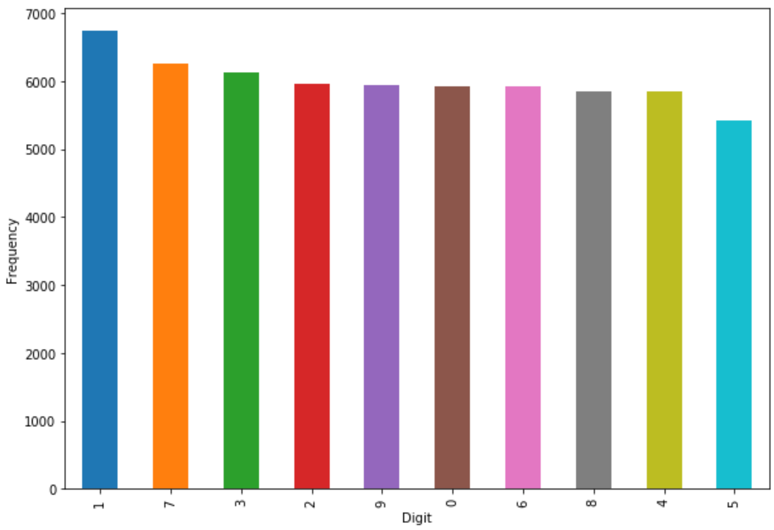

Figure 3: Distribution of the digits in the dataset.

### 2.3 Algorithms and Techniques

The technique adopted in this project starts by training an Autoencoder, which is an unsupervised model, to reconstruct input images. Once the Autoencoder is trained, we can extract the encoder of it and calculate the features of all images in the training set. Then, we compare the features of a given query image to the features of the training images and finally return the closest images.

Autoencoders as presented in Figure 4 are unsupervised models that work as follows:

1. Accept an input set of data (i.e., the input)
2. Internally compress the input data into a **compressed representation**
3. Reconstruct the input data from this compressed representation (i.e., the output)

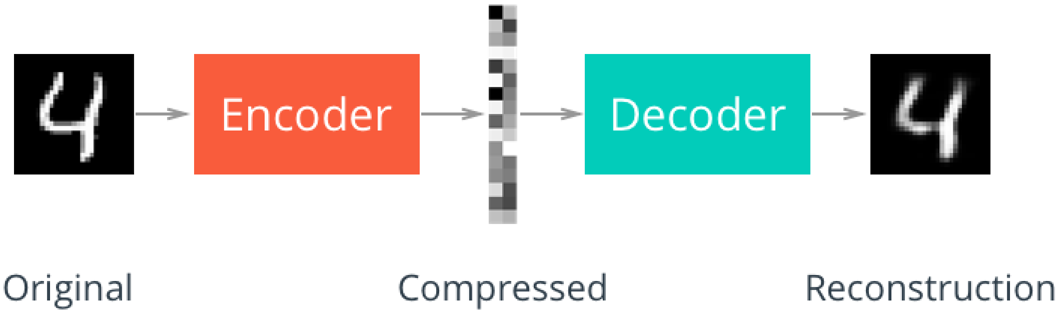

Figure 4: Autoencoder structure.

The encoder part of the autoencoder can be a typical convolutional pyramid, which is what we will use in this project. Each convolutional layer can be followed by a max-pooling layer to reduce the dimensions of the layers. It can also be fully connected neural network with multiple hidden layers.

The decoder needs to convert from a narrow representation to a wide, reconstructed image. For example, the representation could be a 7x7x4 max-pool layer. This is the output of the encoder, but also the input to the decoder. We want to get a 28x28x1 image out from the decoder so we need to work our way back up from the compressed representation.

To build an image retrieval system with an autoencoder, what we really care about is that compressed representation vector. Once an autoencoder has been trained to encode images, we can:

1. Use the encoder portion of the network to compute the compressed representation of each image in our dataset — **this representation serves as our feature vector that quantifies the contents of an image**
2. Compare the feature vector from our query image to all feature vectors in our dataset (typically we would use either the Euclidean or cosine distance)

Feature vectors that have a smaller distance will be considered more similar, while images with a larger distance will be deemed less similar.

We can then sort our results based on the distance (from smallest to largest) and finally display the image retrieval results to the end user. This procedure is illustrated in Figure 5.

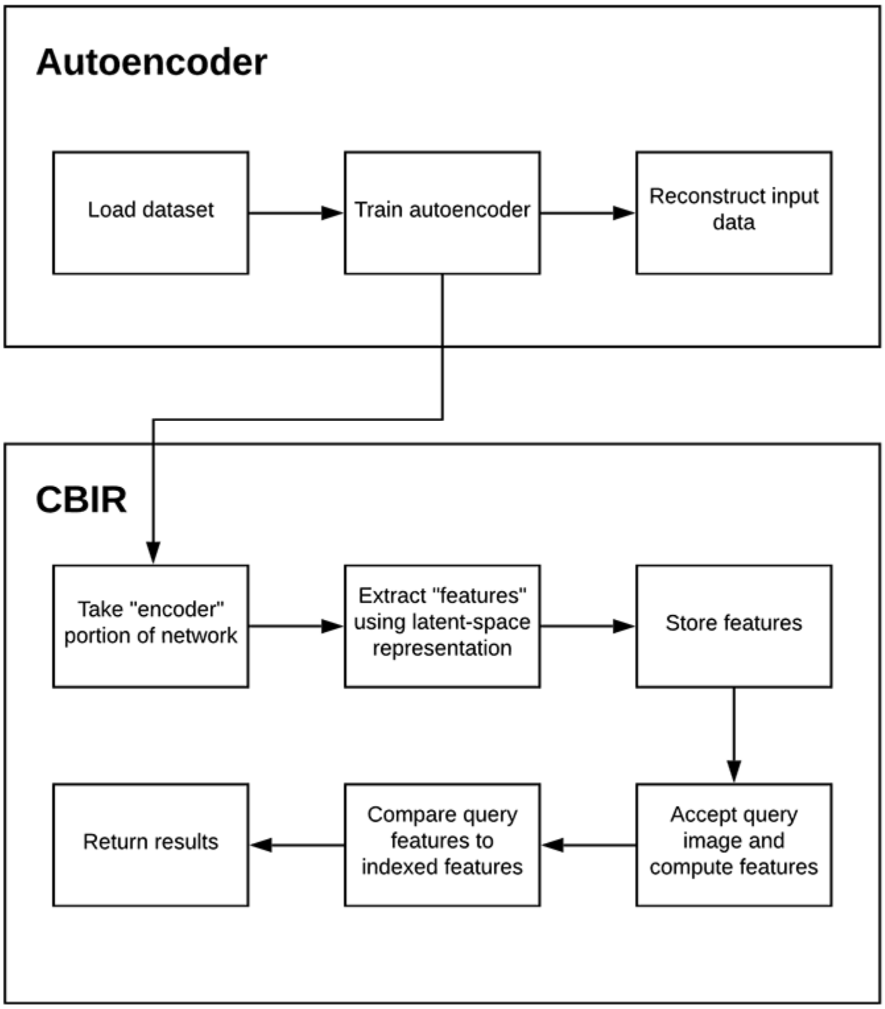
Autoencoder
Load dataset
Train autoencoder
Reconstruct input data
CBIR
Take "encoder" portion of network
Extract "features" using latent-space representation
Store features
Accept query image and compute features
Compare query features to indexed features
Return results

Figure 5: Procedures of image retrieval using autoencoders. [6]

### 2.4 Benchmark

A simple encoder of fully connected layers can be used as the benchmark for this project because it is simple to implement and fast to train. It also gives better results than random guessing for this problem.

Because we can calculate the accuracy of the model, we will also aim for an accuracy over 90%. This is a secondary benchmark to evaluate the results of this project.

## 3 Methodology

### 3.1 Data Preprocessing

The dataset is already scaled to a fixed size of 25x25 and normalized to values between 0-1. Therefore, there are no need to scale or normalize the dataset. The dataset only needs to be transformed to tensors, which is what we do using the *torchvision.transforms.ToTensor()* function.

### 3.2 Implementation

This project is programmed completely in Python using PyTorch library. the first thing to do is to import the libraries we will use through out the project. Then, we need to load the train and test datasets and transform them to tensors.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torchvision import datasets
import torchvision.transforms as transforms

# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# load the training and test datasets
train_data = datasets.MNIST(root=’data’, train=True,
                                   download=True, transform=transform)
test_data = datasets.MNIST(root=’data’, train=False,
                                  download=True, transform=transform)
```

Now, we need to prepare data loaders for both train and test sets.

```python
# how many samples per batch to load
batch_size = 20

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)
```

For the simple benchmark model, we'll train an autoencoder with the images by flattening them into 784 length vectors. The images from this dataset are already normalized such that the values are between 0 and 1. The encoder and decoder are made of one linear layer as presented in Figure 6. The units that connect the encoder and decoder will be the compressed representation.

Since the images are normalized between 0 and 1, we need to use a sigmoid activation on the output layer to get values that match this input value range.
The compressed representation is chosen with a dimension of 32 neurons.

```python
class Autoencoder(nn.Module):
    def __init__(self, encoding_dim):
        super(Autoencoder, self).__init__()
```

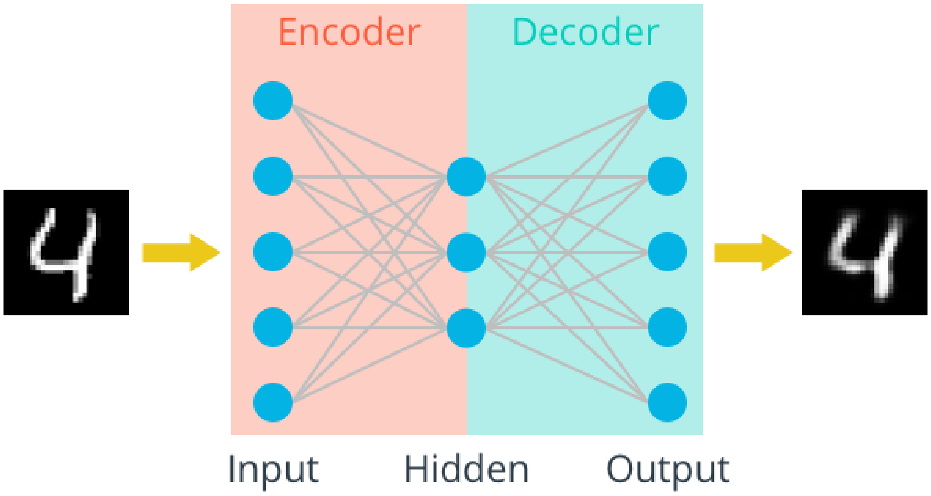

Figure 6: Simple autoencoder.

```
        ## encoder ##
        # linear layer (784 -> encoding_dim)
        self.fc1 = nn.Linear(28 * 28, encoding_dim)

        ## decoder ##
        # linear layer (encoding_dim -> input size)
        self.fc2 = nn.Linear(encoding_dim, 28*28)


    def forward(self, x):
        # add layer, with relu activation function
        x = F.relu(self.fc1(x))
        # output layer (sigmoid for scaling from 0 to 1)
        x = F.sigmoid(self.fc2(x))
        return x

# initialize the simple Autoencoder
encoding_dim = 32
simple_model = Autoencoder(encoding_dim)
```

For the proposed model, we'll build a convolutional Autoencoder as presented in Figure 7. Each convolutional layer will be followed by a max-pooling layer to reduce the dimensions of the layers.
Here the final encoder layer has size 7x7x4 = 196. The original images have size 28x28 = 784, so the encoded vector is 25% the size of the original image. These are just suggested sizes for each of the layers.

```
class ConvAutoencoder(nn.Module):
    def __init__(self):
        super(ConvAutoencoder, self).__init__()
        ## encoder layers ##
        self.conv1 = nn.Conv2d(1,16,3,padding = 1)
        self.conv2 = nn.Conv2d(16,4,3,padding = 1)
        self.pool = nn.MaxPool2d(2, 2)
        ## decoder layers ##
        ## a kernel of 2 and a stride of 2 will increase the spatial dims by 2
        self.t_conv1 = nn.ConvTranspose2d(4, 16, 2, stride=2)
        self.t_conv2 = nn.ConvTranspose2d(16, 1, 2, stride=2)


    def forward(self, x):
        ## encode ##
        x = F.relu(self.conv1(x))
        x = self.pool(x)
```

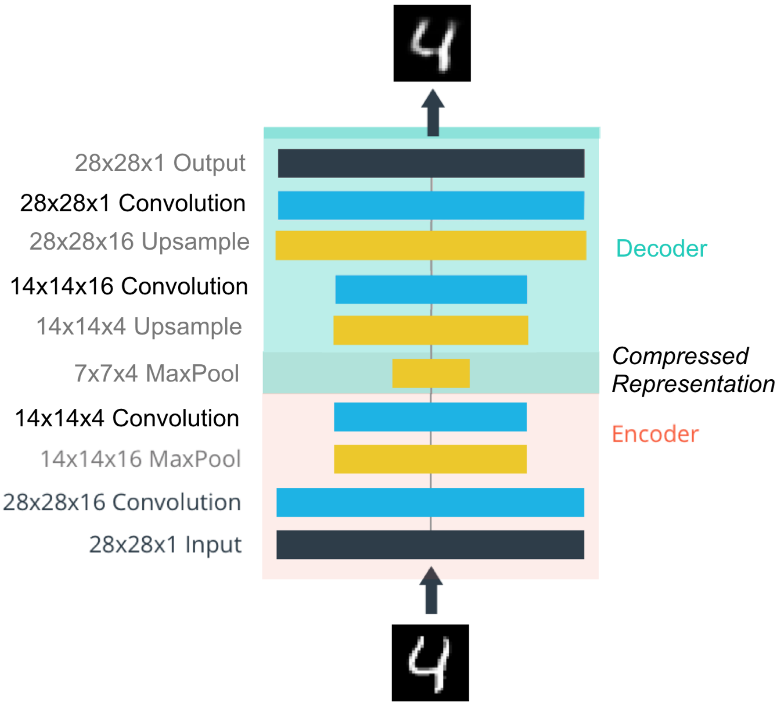

Figure 7: Convolutional autoencoder.

```
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        ## decode ##
        ## apply ReLu to all hidden layers *except for the output layer
        ## apply a sigmoid to the output layer
        x = F.relu(self.t_conv1(x))
        x = F.sigmoid(self.t_conv2(x))

        return x

# initialize the convolutional Autoencoder
model = ConvAutoencoder()
```

We are not concerned with labels in this case, just images, which we can get from the ‘train_loader‘. Because we’re comparing pixel values in input and output images, it will be best to use a loss that is meant for a regression task. Regression is all about comparing quantities rather than probabilistic values. So, in this case, I’ll use ‘MSELoss‘. I’ll also use Adam optimizer for this problem.

```
# specify loss function
criterion = nn.MSELoss()

# specify optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

I have implemented a function to train models as follows:

```
def train_model(model, optimizer, criterion, device, n_epochs = 100, simple = False):
    model.to(device)
    for epoch in range(1, n_epochs+1):
    # monitor training loss
        train_loss = 0.0
        ###################
        # train the model #
        ###################
        for data in train_loader:
            # _ stands in for labels, here
            # no need to flatten images
            images, _ = data
            if(simple):
                images = images.view(images.size(0), -1)
            images = images.to(device)
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            outputs = model(images)
            # calculate the loss
            loss = criterion(outputs, images)
            # backward pass: compute gradient of the loss with respect to model
                                                        parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update running training loss
            train_loss += loss.item()*images.size(0)

        # print avg training statistics
        train_loss = train_loss/len(train_loader)
        print('Epoch: {} \tTraining Loss: {:.6f}'.format(
            epoch,
            train_loss
            ))
```

To retrieve the top $n$ closest images the results, I implemented the following function.

```
# Function to retrieve the closest images
def retrieve_closest_images(test_feature, trained_features, n=7):

    # initialize the distance list
    distances = []

    for features in trained_features:
        distance = np.linalg.norm(features - test_feature)
        distances.append(distance) # append to the distance list

    n_elements = trained_features.shape[0] # total number of images in the training set
    distances = np.array(distances) # convert the distance list to a numpy array
    trained_features_index = np.arange(n_elements) # creae an index list from 0 -
                                                n_elements

    # create a numpy stack with the distances, index_list
    distances_with_index = np.stack((distances, trained_features_index), axis=-1)
    sorted_distance_with_index = distances_with_index[distances_with_index[:,0].argsort
                                                ()] # sort the stack

    sorted_distances = sorted_distance_with_index[:, 0].astype('float32') # change the
                                                datatype
    sorted_indexes = sorted_distance_with_index[:, 1]

    kept_indexes = sorted_indexes[:n] # Get the first n indexes of the sorted_indexes
                                                list

    return kept_indexes.astype('int')
```

### 3.3 Refinement

The initial accuracy for the simple Autoencoder benchmark model on the first 100 images from the test set was approximately 10%.

```
The accuracy of the simple autoencoder: 0.10428571428571429
```

To check whether the Autoencoder are trained well or not, Figure 8 shows the original images in the first row and the reconstructed images using the simple Autoencoder in the second row. For the convolutional Autoencoder, Figure 9 shows the reconstructed images in the second row. It is clear that both models are able to somehow reconstruct the original images. However, the convolutional Autoencoder has a slight advantage since the reconstructed images are much cleaner.

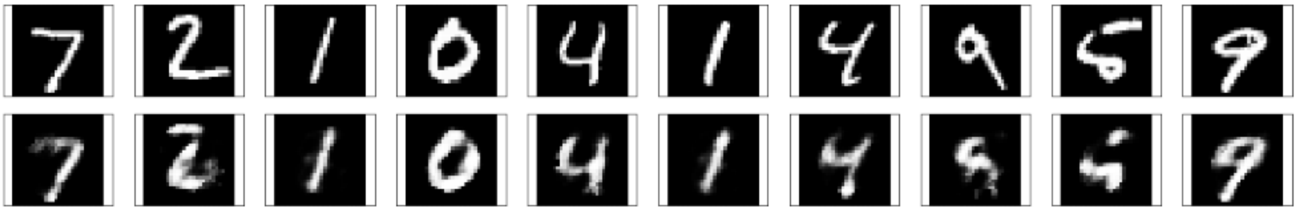

Figure 8: Reconstructed images using the simple Autoencoder.

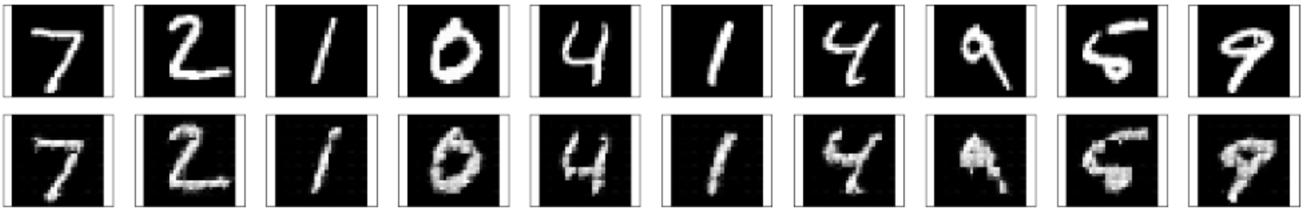

Figure 9: Reconstructed images using the convolutional Autoencoder.

The retrieved images from the simple Autoencoder and the convolutional Autoencoder are presented in Figure 10 and Figure 11 respectively. The query image to the left is followed by 7 similar images retrieved from the training data set. As mentioned earlier, the simple autoencoder performs poorly in this problem with an accuracy of about 11%.

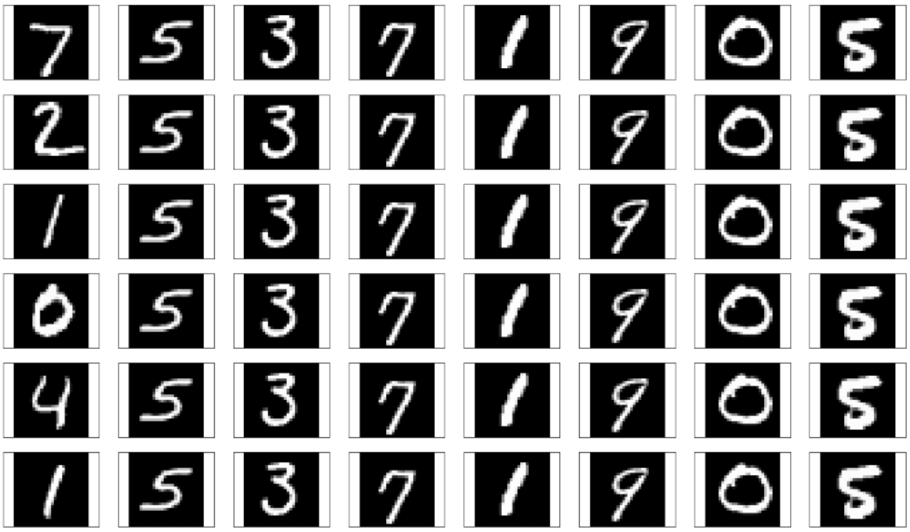

Figure 10: Retrieved images from the simple Autoencoder.

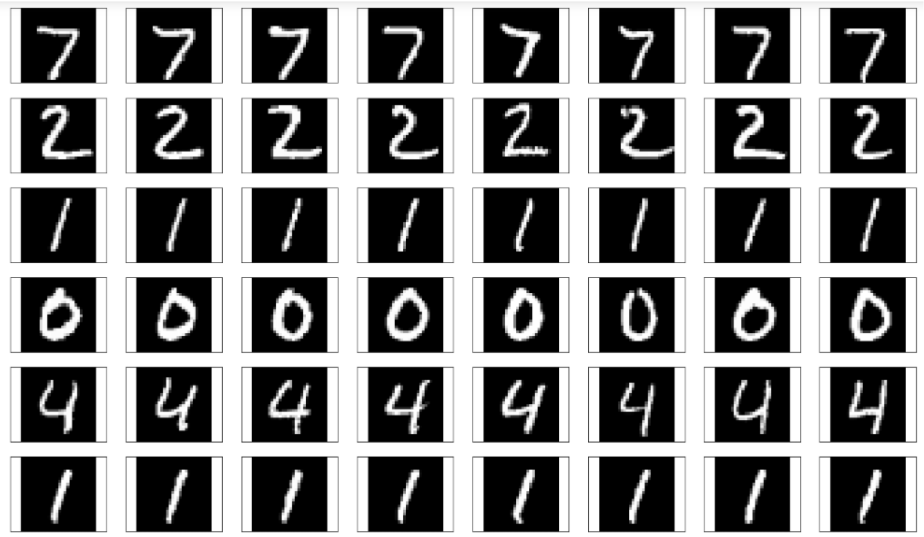

Figure 11: Retrieved images from the convolutional Autoencoder.

The refined second model, the convolutional auto encoder managed to achieve much better accuracy of about 91%. We were aiming for an accuracy above 90% and we got there.

```
The accuracy of the convolutional autoencoder: 0.9057142857142857
```

I have tried many ways to try increasing the accuracy and found that adding a batch normalization layer after each convolution layer of the encoder helps a lot. Therefore, I trained the same previous model with a batch normalization layer added after each convolution layer. The accuracy of this model increased to exactly 93%.

```
The accuracy of the refined convolutional autoencoder: 0.93
```

# 4 Results

## 4.1 Model Evaluation and Validation

The final model is evaluated on the test set, which was left aside during training. The parameters of the final model are presented in Figure 12 with two additional layers. One additional batch normalization layer is added after each convolution layer in the encoder.
The final model is found manually by trying different possible parameters. The best training loss was recorded to be 0.159577.

## 4.2 Justification

On the test set the benchmark simple Autoencoder model achieved an accuracy of about 11%. The intermediate convolutional Autoencoder managed to achieved an accuracy of about 91% on the test set. The final model achieved the highest accuracy of 93% and the best training loss. Since the test-set is unseen dataset, this means that the final model can achieve well on unseen datasets with high accuracy.

# 5 Conclusion

## 5.1 Free-Form Visualization

More retrieved images using the final model are presented in Figure 13, where the query image is located to the left followed by the top seven similar images.

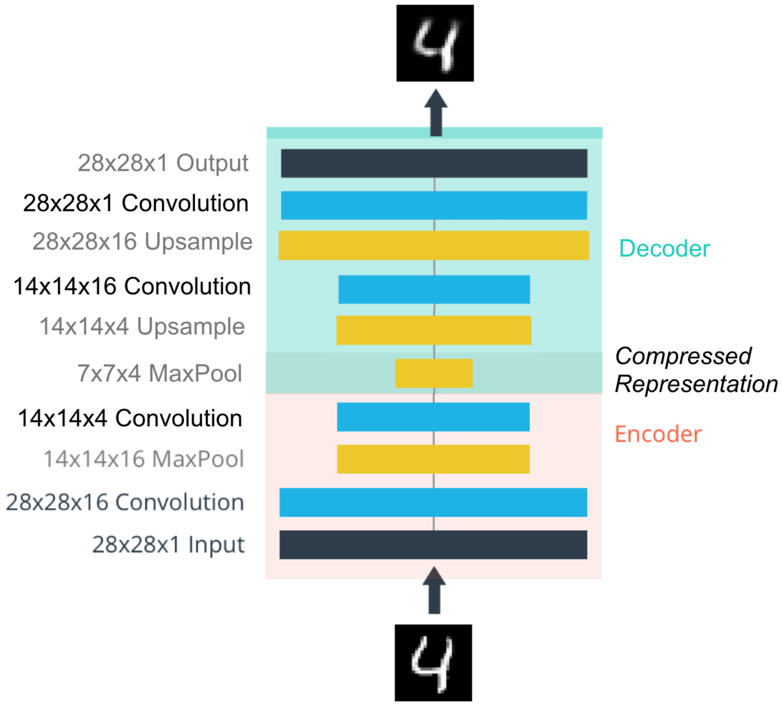

Figure 12: Final convolutional autoencoder.

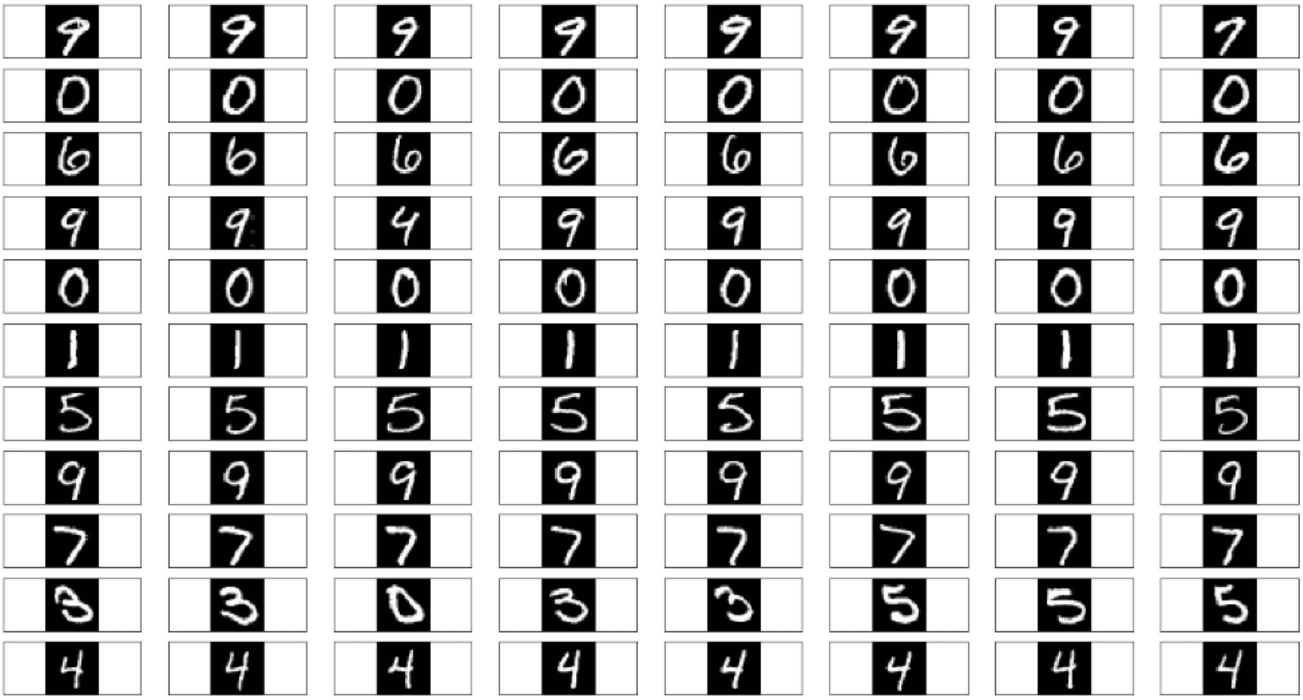

Figure 13: Retrieved similar images from a given input image.

## 5.2 Reflection

The overall process can summarized as follows:

1. Train Autoencoder.

2. Compute the compressed representation of the training set.

3. Compute the compressed representation of a given query image.

4. Compare the compressed representation of the given query image to all compressed representation of the training set.

5. Return the top $n$ similar images to the end user.

One of the most interesting aspects of this project was the fact that images from the same class have the roughly the same compressed representation. This idea can be utilized in many interesting project. One idea is that you completely automate the process of labeling images using Autoencoders.

### 5.3 Improvement

With more computation power, it would be possible to try deep state of the art architectures on real datasets. For example the Perfect Half Million Beauty Product Image Recognition Challenge [5] offers Perfect-500K dataset which is a collection of beauty and personal care items from 14 popular e-commerce sites, including Amazon (USA, India), Cult Beauty, Flipkart and Target. This requires better computers with really high processing units. It would actually be interesting to find how this approach performs on such challenge.

## References

[1] Y. Liu, D. Zhang, G. Lu, and W.-Y. Ma, "A survey of content-based image retrieval with high-level semantics," *Pattern recognition*, vol. 40, no. 1, pp. 262–282, 2007.

[2] W. Zhou, H. Li, and Q. Tian, "Recent advance in content-based image retrieval: A literature survey," *arXiv preprint arXiv:1706.06064*, 2017.

[3] D. Zhang, M. M. Islam, and G. Lu, "A review on automatic image annotation techniques," *Pattern Recognition*, vol. 45, no. 1, pp. 346–362, 2012.

[4] D. ping Tian *et al.*, "A review on image feature extraction and representation techniques," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 8, no. 4, pp. 385–396, 2013.

[5] S. L. J. F. J. L. S. C. H. J. T. Wen-Huang Cheng, Jia Jia and J. Huang, "Perfect corp. challenge 2020: Half million beauty product image recognition." `https://challenge2020.perfectcorp.com/`, 2020.

[6] "Autoencoders for content-based image retrieval with keras and tensorflow." `https://www.pyimagesearch.com/2020/03/30/autoencoders-for-content-based-image-retrieval-with-keras-and-t` 2020.