**Computer Vision [H02A5a]**

Final Project Report  
**‘IN THE NAME OF DEEP LEARNING’**

Master of Artificial Intelligence

2018-19

Michael Bouchirab (rXXXXXX)

Shreya Saxena (r0689164)

**TABLE OF CONTENTS**

**1. Introduction 3**

**1. Dataset Description 3**

**3. PCA vs. Auto-encoder 4**

**4. Non- Linear Auto-encoder 4**

4.1 Architecture 4.2 Loss Function

4.3 Training the network

4.4 Reconstruction Results

**5. Classification 5**

5.1 Training Network 1

5.2 Training Network 2

5.3 Results

**6. Segmentation**

6.1 Results

1. **Introduction:**

This report gives an overview of our approach and results of implementing deep learning in computer vision applications. During the course of project implementation, we got to try hands on various aspects of deep learning. The project can be viewed as consisting of 3 major tasks:

The first goal, discussed in Section 4, is to transition from linear PCA to non-linear auto-encoders. Second goal, discussed in Section 5, is to use the learned, abstract latent space for classification. Further, our final goal, discussed in Section 6, is to translate image classification into a pixel-wise classification task for segmentation.

1. **Dataset Description:**

We are using PASCAL VOC-2009 dataset for this project. This dataset originally comprises of colored images and annotations of 20 different object classes.   
In our project, we are using 5 of these classes, namely ‘airplane’, ‘car’, ‘chair’, ‘dog’ and ‘bird’. These yield us a total of 1489 training images and 1470 validation images. The dataset is divided equally for training and validation purposes, i.e. 50% of images each for training and validation. We have re-sized all the images in training and validation set to dimensions 28x28 for better computation efficiency.

Fig: *Sample images of classes ‘airplanes’, ‘car’, ‘chair’, ‘dog’ and ‘bird’ from the dataset*

1. **PCA vs Auto-encoder:**

Principal Component Analysis is a technique that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information of the large set. So, the underlying idea is to reduce the dimensionality of data sets with minimal information loss. Whereas, an auto-encoder is a special type of feed forward neural network which encodes input *x* into hidden layer *h* and then decodes it back from its hidden representation. The model is then trained to minimize the loss between the input and output layer.

Our first task is to simulate a linear PCA using an auto- encoder. We begin with creating a fully connected neural network containing an input layer, hidden layer and an output layer.

First of all, an input layer is created with dimensions 28 x 28 to which a flattened layer is added to make it easier to work with and also to store in memory easily. We build the network with a code layer of size 32. Finally, we used ‘ADAMAX’ as optimizer and loss function ‘MSE’ and trained our network for 500 epochs.

We can say our encoder mimics a PCA, as it reduces the dimensions of the input image from its original form similar to what a PCA does.

Though, auto-encoder bears a significant resemblance to PCA, there are two major differences which are cited below:

* Firstly, the auto-encoder is a non-linear transformation, contrary to PCA, which makes the auto-encoder more flexible and powerful.
* Secondly, the axes found by a PCA are orthogonal, and are ordered in terms of the amount of variability which the data presents along these axes. This makes the interpretability of the PCA much greater than that of the auto-encoder, which does not have these attributes.

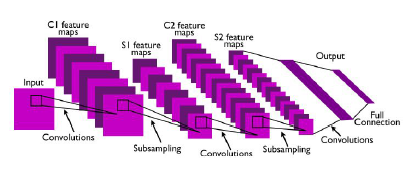
1. **Non-linear auto encoder:**
   1. **Architecture:**

Next, we move a step ahead of linear auto- encoder and create a convolutional neural network which has added features over the auto- encoder we built in previous section. CNNs are deeper neural networks and have greater ability to successfully capture the spatial and temporal dependencies in an image through the application of relevant filters.

a full CNN architecture is obtained

by stacking several of these layers

A full CNN architecture is obtained by stacking several types of layers, namely, convolutional layer, pooling layers and a fully connected layer. The figure below shows a typical CNN architecture with two feature stages:



* ***Input Layer:*** This layer loads inputs and produces output used to feed convolutional layers. In our case, input images have dimensions 28 x 28 and number of channels (3 for RGB).
* ***Convolutional Layer:*** This layer convolves the input images with a set of learnable filters, each producing one feature map in the output image.

We are using 6 convolutional layers in our model each using ‘ReLU’ as an activation function. The first and last convolutional layer is set to learn 16 filters while all others learn 8 filters each.

* ***Pooling Layers:***  These layers are responsible for down-sampling the spatial dimension of the input. There is one pooling layer after each convolutional layer. We have set a stride of (2 x 2) for each of these layers.

The deepest layer representation is 4 x 4x 8 i.e. 128 dimensional.

* ***Fully connected Layers:*** These layers treat the input as a simple vector and produce an output in the form of a single vector. There is one such layer in our model. The last one uses soft-max activation function.
  1. **Loss Function:**

Since our dataset contains images which belong to multiple classes because of multiple objects present in them, we are using Binary cross- entropy loss function for the CNN. Since, this particular loss function works well for multi-label classification, it seems to be a perfect fit for our model.

* 1. **Training the network:**

The model is trained to 10 epochs using the training and validation set provided in the dataset.

* 1. **Reconstruction results:**

1. **Classification**

The model obtained in previous section transforms our input image into a number of coding variables. Now, the next task is to use these variables, train them appropriately and convert them into a classifier. The following subsections discuss how we train them into two different networks.

* 1. **Training Network 1:**

For building our network 1, we take the encoder part of our model and add a fully connected layer at the end while making sure that the current weights of encoding layers are frozen and don’t get updated when we train our new network.

We freeze the layers by setting the flag ‘*layer.trainable*’ to ‘False’ and use activation function ‘*softmax*’ for our fully connected layer added at the end of the network. Finally, we will train our model using optimizer function ‘*adam*’ and loss function *‘binary\_crossentropy’*. We are using ‘*binary\_crossentropy*’ as the loss function here, because of multi-label classification as discussed in previous section.

* 1. **Training Network 2:**

For building our network 2, we have trained all the parameters from scratch. We train the below layers of our model:

* ***Convolutional Layer:*** We re-use the 3 layers of encoding part and randomly initialize their weights by setting the flag ‘*kernel\_initializer*’ to ‘*random\_normal*’. The first layer learns 16 filters and the remaining two layers learn 8 filters, similar to what we had trained before.
* ***Pooling Layers:***  There is one pooling layer after each convolutional layer. We have set a stride of (2 x 2) for each of these layers. The deepest layer representation is 4 x 4x 8 i.e. 128 dimensional.
* ***Fully connected Layers:*** We have added a fully connected layer with activation function ‘softmax’ at the end of the network.

Finally, we train the model using loss function ‘*binary\_crossentropy*’ as we did earlier.

* 1. **Results:**