Computer Vision Project

Sarah Binta Alam Shoilee(r0729856)

10 June, 2019

In this era of computational advancement, image processing is taking a new shape with the help of deep learning. In this project, we tried to demonstrate the approach of neural network in image classification and segmentation. If we consider every image to be a data of a large dimension(every pixel considered to be a dimension), processing image data is very expensive in terms of memory and computation time. Using the power of deep networks, we will see how efficiently the algorithm can learn the pattern of images and extract semantic information from them. To process image data it is trivial to reduce the dimension in favor of computation complexity. In this project, we have shown that how can we use neural network to mimic linear Principal Component Analysis(PCA) and later how to improve this dimensional reduction using a deeper network. Next we tried to map image on its given classes using the learned feature from the previous step. At the end, we plotted semantic segmentation based on different classes within the same image. [5].

1 Data Set

For our project, we use the PASCAL VOC2009 dataset which is known to be standard image data set to measure performance over task like image classification and segmentation. To build my project, I use only two classes from the data set "dogs" and "chairs". Based on this two classes, we filtered data from both training set and validation set given within the dataset. All my finding are based on these two classes. Due to computation resource limitation I had to con-size the image to 128*128 pixels.

2 Auto-Encoder

2.1 PCA vs auto-encoder

Principal Component Analysis(PCA) is one of the well known method to shrink data over a lower dimensional space.We can describe PCA as a dimensional reduction technique by linear transformation. The idea behind this approach comes from the tendency of maximizing the variances between data points and discard the correlated axis values. This is calculated from the co-variance matrix of observation data set. The newly formed coordinate of data set is always the m-largest variance value sorted in decreasing order and each coordinate is independent of each other. The conventional way of calculating PCA is to projecting on eigenvalue of co-variance matrix. Given a data $X = (x_1, x_2, ..., x_n, PCAwillconvertitintodataZ = z_1, z_2, ..., z_m$ where m¡n by using formula

$$z_d = e_d^T x$$

Any data $X \in R^n$ will be the projected to $Z \in R^m$ with $m < n$. Here, e_d is an unknown vector and T symbolizes the transpose of the vector. To find this e_d, we calculate the co-variance matrix of our given data and take its d number of eigenvectors corresponding to d largest eigenvalues. To reconstruct the data from Z, we need to find a good value for m so that there is less reconstruction error. To ensure that we take d most largest eigenvalues those contributes the majority of the sum of all eigenvalue and ignore the rest. The construction is done using the following:

$$x_{new} = f_d z$$

or,

$$x_{new} = f_d(e_d^T x)$$

If we can well chosen f_d, we can reconstruct the data with least possible error and generate $x_{new} \approx x$.
The main idea of auto-encoder is also reconstructing the same data by compressing it into lower-dimensional which is known as the bottleneck. This a specific type of neural network where the input is equal to output to learn feature in an unsupervised manner. This is exactly like the reconstruction from PCA. In the encoder part of auto-encoder, we do the compression to data and in the decoder part we do the re-construction. To train the network, we focus on minimizing the construction error as our loss function. [3]

In figure 1, the mechanism for an auto-encoder that construct linear PCA is shown.
We tried implementing both approach and took measure for reconstruction error. Both of them has similar performance, though PCA performed higher. For the calculation of PCA, the square matrix computation has been occured which is very expensive in terms of memory consumption. For constructing the auto encoder, we have taken 512 weights at the first layer and decreased eventually at the bottleneck. The amount of approximation justifies the performance of auto encoder though it can be overcame with more training epochs.

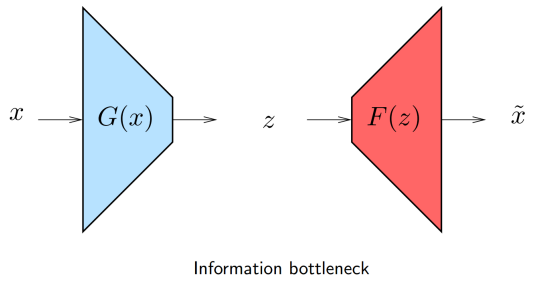

Figure 1: Linear PCA mimicking auto-encoder

2.2 Non-linear and convolutional

It is been proven that convolutional neural networks work better for spatial data. For image data set, this assumption is highly correct. In an image, closer pixel contains more co-relation than the pixel lies further. Convolutional neural network(CNN) works based on this assumption and compress data into lower dimension. For our data set, it is wise to use CNN to build our autoencoder.
To build this auto encoder, we have flexibility to use the hyper-parameters to get less reconstruction error. In encoder setting, when we use more layer it learns sparse feature(like blob and edges) in initial layer and starts building complex feature in deeper layers. In this project, we are interested to learn meaningful feature in our latent space, rather than minimizing reconstruction error, for that reason we tried to make a balance between them.
The first approach, we selected to build an autoencoder is consist of 2 layers in both encoder and decoder with one code layer in between and we also have

a output layer in between. For every architecture, this is common to have a code layer which is our lower dimensional representation of data and an output layer. The thing we have play with is the number of layer, number of nodes in each layer and operation over each layer. In this architecture, we selected more weights(number of kernels, n) in the initial layer than latter with the interest of keeping less number of nodes in the latent space. In each layer, we use maxpooling of kernel size= $2 * 2$ with stride 2 that reduces the size of nodes in half. In this way, we end up having $16 * 16$ node in code layer with n number of channels. In the next phase of tuning parameter, we keep the same architecture but put more kernel in each layer and measure the reconstruction error. Turns out, we get better result than previous approach.
To measure reconstruction error, we used both loss function "mean squared error" and "binary cross-entropy"(approach one and three based on same architecture of auto-encoder). In our context both are meaningful since we are trying to construct the same image through the network with error tolerance. In each training epoch, the algorithm is counting pixel to pixel mismatch and trying to adjust the loss in next epochs. "Mean square error" measures the error of each pixel corresponding to the newly formed image and produces mean squared value on the whole data. On the other hand, "binary cross-entropy" is interested in maximizing the log-likelihood of the probability of newly formed data giving the value of the previous data. The concept of binary cross-entropy still stands for real-valued data because it is force to maximize the likelihood of probability of $x \approx \mathrm{x}_{new}$ irrespective of what value we have in our data. [2] From our experiment between both the approach, we find "mean square error" produced more meaningful feature on our data set and help reconstructing our image better.
Next, we tried experimenting with a newer architecture which is deeper than the previous and also inspire by VGG-16 architecture. VGG-16 architecture has been quite successful on this PASCAL data set which has been proven to be a bench mark for operation like image classification and detection. [1] We applied the same architecture to build our encoder and did the reverse of it to build decoder where we have 5 pooling layers in each. As in VGG-16, we also applied normalization on each layer after convolution operation to get rid off extreme value. This architecture failed terribly in image reconstruction while we use the number of kernel in each lager in decreasing order in encoder[64, 32, 16, 8, 8].
To experiment more with this architecture, we tried using number of kernel in increasing order[8, 8, 16, 32, 32] in encoder and ends up having $32 * 8 * 8$

nodes in the code layer. Though this approach increases the space in code layer, it is able to find feature best to reconstruct images. While in the previous approach using same architecture but different number of nodes in each layer fails to reconstruct images. This result was obvious here because in the initial layer of convolutional neural network we learn feature which is sparse that justify us having less number of weights there. In the deeper layer, it tries to construct more complex feature which is why we need thick layer of weights. For example, is we want to construct a face from images, in CNN, we learn edges in the initial layers, as the layer goes deeper, we started learning body parts and end up constructing the picture of whole face.

In Figure-2, we can see original image vs reconstructed image using the above mentioned auto encoder architecture while we training the network over 200 epochs.

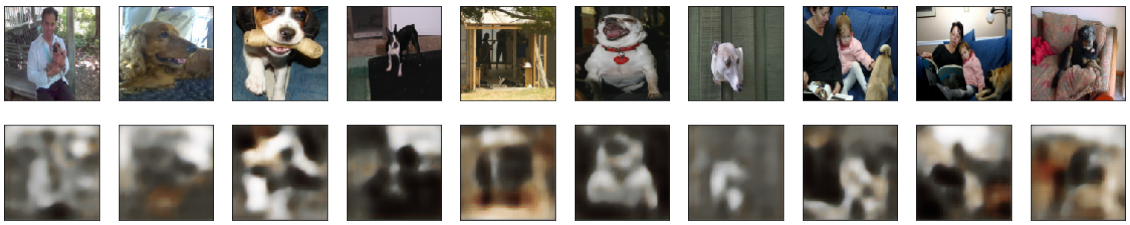

Figure 2: Image reconstruction using autoencoder. The top row shows the original images and the bottom rows shows the reconstructed images from validation set.

We tried to visualize our learning using T-SNE to see the feature learned can find two pure cluster or not. From Figure-3, we cannot say our feature is best learned, but we can see it generate different data representation than it was in the original data. Moreover, T-SNE may not be the best representing of our learned feature since it works best for at most 32 dimensional data and we have features of size $32 * 8 * 8$.
Here, we make this trade off between feature learning and image reconstruction(with Training accuracy: 63.01% and Validation accuracy: 61.22%). Given the complex data set along with overlapping of multiple class in same image and less number of training image, we will still accept our current architecture and proceed further with that.

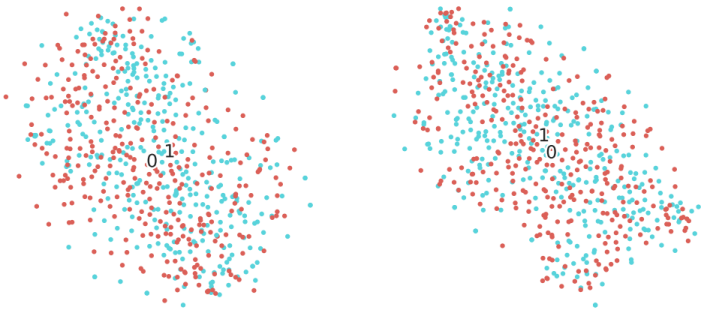

Figure 3: Cluster representation with T-SNE; Red dot represents the dogs image and blue dot represents the chair image. The first plot represents the original dataset representation and the second one is the plot using latent space features.

3 Classification

In the previous section, we trained an autoencoder and gain weights of this network. In code layer, we represents the feature learned from the autoencoder. Using this feature, in this current section, we will try to build a classifier.

At first, we build a classifier using the encoded part of the autoencoder by adding to more layer with it. At the very beginning, we flattened our output for CNN to convert it in one dimenisional data to train classifier. Then we add a fully connected layer with 4096 hidden units with 20% input dropout to avoid over fitting. Then we connect same layer again to help it be well converted to 4096 points. At the last fully connected layer, we compute the "softmax" activation function which generates the probability of being in class 0 or 1.

In figure-4 we can see the accuracy and loss we found from the network implementation. Here, we use the weights learns from the auto-encoder and only trained the fully connected layers to learn classification. The accuracy from this network is 48.27% from where we can conclude that our network did not learn enough feature to classify images in distinct classes.

To illustrate our findings further, we tried training the whole network again, but this time we also tuned the weights learned from the previously trained autoencoder. But, find out the performance is still the same. The loss func-

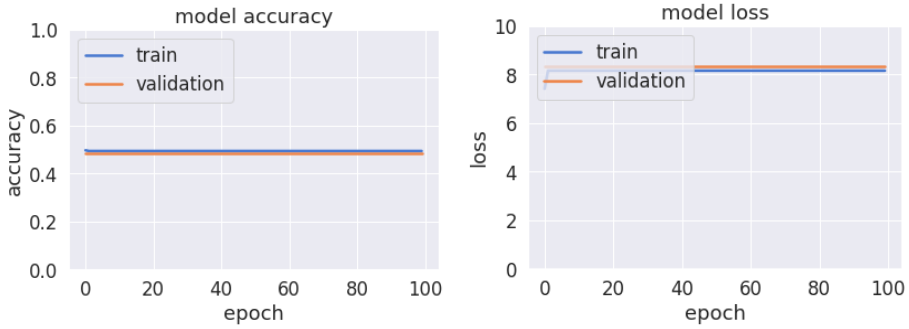

Figure 4: Classifier from trained autoencoder

tion reaches its minimum level and it cannot train any further. We can conclude that the feature learned from the latent space is not enough to find good classification performance. To validate our findings over the failure of feature extraction through auto encoder, we train another model using the same architecture. We tried to find out if the network could perform better given this architecture to learn classifier. To do that, this time we train every parameter from scratch without any prior assumption of weight values. From

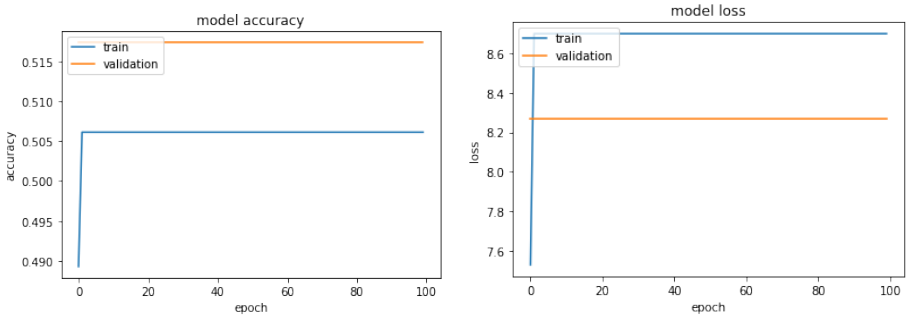

Figure 5: Classifier from scratch

figure-5, we can though the is a little improvement in the performance, it is not much. This little improvement justifies of not using dropout while training which means the network fails to fit even the training data properly.(with 50% accuracy over training data) Overall, we can say the hyper parameter we choose to build this network is not correct. Though the autoencoder manages to retain structer of the original images, it fails to exract feature for classification. Not surprisingly, encoder performs bad in classification be-

cause while building the autoencoder we tried to keep latent space as small as possible to find better image construction. On the other hand, to extract feature in deeper layer they need more kernel to make a representation. Due to this opposite interest of these two approach, my model fails to classify the dataset into two classes.

4 Segmentation

In this section, we are going to take our experiment further. Here, we are going to use convolutional neural network in image segmentation. We are not only interested to find the existence of image classes, but also we want to make pixel wise classification. Meaning that each pixel of the image will have a value based on their belongings to a particular class and thus they can make the image segmentation. To train our model, we use images corresponding to their ground truth images(mask) to learn the pattern of segmentation. The following images depicts the training input and output for segmentation.

Figure 6: Input and output for image segmentation. The first image is the original image from our data set and the next one is the ground truth image for that image.

Like before for this task, I stick with the images of "dogs" "chairs" from the PASCAL VOC2009 dataset, but now we'll just have to use a subset of it. This data set contains a lot of images which does not have a corresponding segmentation image. We only take those images from the given two classes who have images in the segmentation class and end up with having our 57 and 54 pairs of picture and segmentation respectively in our training and validation dataset.
To find the semantic segmentation within the image, we can train our network only by providing image and its corresponding ground truth. If we do so, two problem occurs; one the network only fit to the train data, do not learn the pattern of the segmentation and fails to perform well for the validation set. The other problem is it is also computationally expensive. For this task, we have to classify every pixel of the image like we did for the whole image. If we have a image of size N, then we have classify N^2 pixel.
To solve our issue, we use the convolutional neural network with deconvolutional layer where as a output we want n(# of classes) number of N*N matrix. In each N*N matrix, there is a probability score for each pixel position to belong the particular class.This ensures the class prediction along with localization. We can say, this mechanism is quite close to the approach of autoencoder, but we do the upsampling in a different way. For upsampling we use convolutional transpose here, unlike autoencode we do layerwise transpose. Since the deeper layers contain most information about overall picture constructions rather than local patterns, using transpose on this layer will result to a coarse structure of this image. To find smaller details we apply deconvolution in the shallow layer and add them with the previous one. Since they have some over lapping while upsizing, the two tranpose will produce object with thin and sharp edges and will put the detail that was missed from the previous transpose.
To implement our network, we also stick with VGG-16 architecture for our network model. This time, we are going to use all 5 blocks of VGG network for convolution. To apply all 5 blocks of VGG-16 and come up with 7* feature matrix for the next layer, we kept the image size at 224*224 this time. We also follow the number of kernel chosen for each block as the given network [64, 128, 256, 512 512].
As our loss function, we choose Dice coefficient which is a prevalent choices for our context. We will use it for our training because later I will transform segmentation images into binary masks that contain only 0 and 1 values so that they can be easily analyzed by the Dice coefficient. We will also use

it as our performance metrics because accuracy is not quite suitable in this context. In our mask image most of the pixel appears black, if we use accuracy as our performance matrix it will show us higher accuracy even if we fail to find right segmentation.

References

[1] Vgg16 – convolutional network for classification and detection, 2018.

[2] Why binary crossentropy can be used as the loss function in autoencoders?, 2018.

[3] A. Dertat. Applied deep learning-part 3: Autoencoders, 2017.

[4] Trevor Darrell Jonathan Long, Evan Shelhamer. Fully convolutional networks for semantic segmentation. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[5] Jean Marie Linhart. Teaching writing and communication in a mathematical modeling course. *PRIMUS*, 24(7):594–607, 2014.

[6] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, page 579–2605, 2008.