ENPM673: Project 6 - Report "Cat and Dog Classifier using CNN".

Smriti Gupta sgupta23@umd.edu
Varun Asthana vasthana@umd.edu
Saumil Shah sshah293@terpmail.umd.edu

***Abstract*— We used VGG-16 architecture to build deep convolutional network for classifying a large data set consisting 25000 images of cats and dogs into their respective classes.**

I. Data Set and Image Preprossessing

The most common image data input parameters are the number of images, image height, image width, and number of channel. Typically we have 3 channels of data corresponding to the colors Red, Green, Blue (RGB) Pixel levels are usually [0,255]. We resized our images into [224,224,3]. We have used Pytorch's *torchvision.transforms.Compose* module to compose a series of transforms on the images. Functional transforms give fine-grained control over the transformations.

$$torchvision.transforms.Compose(transforms) \quad (1)$$

a) Rescaling: - We rescaled the images to have the lower dimension of image height or width to a size of 250, while maintaining the aspect ratio. The Fig 2 shows a rescaled image (Fig 1 sows the original image from the data set).

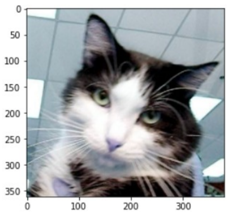


Fig. 1: Original Image

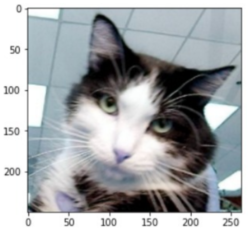


Fig. 2: Rescaling Image

b) Random Crop: - After rescaling, we randomly cropped the images to a size of (224,224). Fig 3 shows the image after random crop.

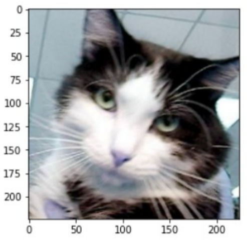


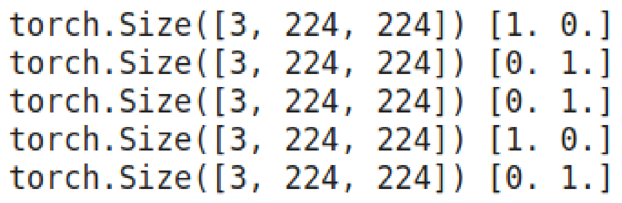
Fig. 3: Cropping Image

c) Normalize: - We converted the images into grayscale and normalized them form [0,255] to [0,1]. Data normalization is an important step which ensures that each input has a similar data distribution. This makes convergence faster while training the network. Data normalization is done by subtracting the mean from each pixel and then dividing the result by the standard deviation. The distribution of such data would resemble a Gaussian curve centered at zero.

d) Converting Images into Tensors: All the transformed images were in numpy arrays, we converted them into tensors to feed them into neural network.

$$\begin{aligned} datatransform = transforms.Compose([&Rescale(250),\\ &RandomCrop(224),\\ &Normalize(), ToTensor()]) \end{aligned} \quad (2)$$

e) Data Label Generation: Images are stored in a folder with a csv file containing the labels for every image, such that all cat images are labelled '0' and all dog images are labelled '1'. We have used PyTorch's data loading utility- *torch.utils.data.DataLoader* to load the images. It represents a Python iterable over a dataset, with support for map-style and iterable-style datasets, customizing data loading order, automatic batching, single- and multi-process data loading,automatic memory pinning to create our dataset of images and labels in a dictionary.Also, We split the data into training and validation set in the ratio of 80% and 20% respectively of the complete labelled image data.



```
torch.Size([3, 224, 224]) [1. 0.]
torch.Size([3, 224, 224]) [0. 1.]
torch.Size([3, 224, 224]) [0. 1.]
torch.Size([3, 224, 224]) [1. 0.]
torch.Size([3, 224, 224]) [0. 1.]
```

Fig. 4: Input

II. Architecture

We used VGG-16 Architecture to build a deep convolutional neural network with 13 convolutional layers (abbreviated as conv layers), 5 Maxpool layers, 2 dropout layers and 3 fully connected layers.The input to conv1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional layers. The convolution stride is fixed to 1 pixel. To introduce the non-linearity in the model we have used 'ReLu' activation. Spatial pooling is carried out by five max-pooling layers. Max-pooling is performed over a 22 pixel window, with stride 2.

After this we have 3 Fully-Connected (FC) layers with the first two layers having 4096 output channels each, while the third layer has 2 output channels, and thus contain 2 channels (one for each class). The final layer is the soft-max layer. Fig 5 shows the architecture of VGG-16.

A. Specifications

a) Convolutional Layers: -A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image. It is the first step to extract features from an image. We have used 13 convolutional Layers in our neural network.

$$conv1 = nn.Conv2d(in-channels, \\ out-channels, kernel-size) \tag{3}$$

b) Activation Function: - We have used ReLu activation function because it converges faster. Non-linearity means that the slope doesn't plateau, or "saturate," when x gets large. It doesn't have the vanishing gradient problem suffered by other activation functions like sigmoid or tanh.We have used activation function after every convolutional step.

$$A = torch.nn.ReLU(inplace = False) \tag{4}$$

c) Loss Function: For binary classification problem like this, CrossEntropy Loss Function is proved to give best results.

$$L = torch.nn.CrossEntropyLoss(weight = None, \\ size-average = None, ignore-index = -100, \\ reduce = None, reduction =' mean') \tag{5}$$

d) Maxpooling: : It is a sample-based discretization process to down-sample the image so as to reduce its dimensionality. We have used 5 Maxpool Layers.

$$P = torch.nn.MaxPool1d(kernel-size, \\ stride = None, padding) \tag{6}$$

e) Drop Out Layers: - We have added two drop out layers because during training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.This has proven to be an effective technique for regularization. We have used 2 dropout layers.

$$D = torch.nn.Dropout(p = 0.5, inplace = False) \tag{7}$$

f) Linear Layers: Applies a linear transformation to the incoming data. When the input features are received by a linear layer, they are received in the form of a flattened 1-dimensional tensor and are then multiplied by the weight matrix. This matrix multiplication produces the output features. We have used 3 linear layers.

$$fc = nn.Linear(in-features = 4, \\ out-features = 3, bias = False) \tag{8}$$

g) Batch Normalization: Batch normalization provides an elegant way of reparameterizing almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers. We have used 2 batch norm layers.

$$bn = nn.BatchNorm1d(param)$$

(9)

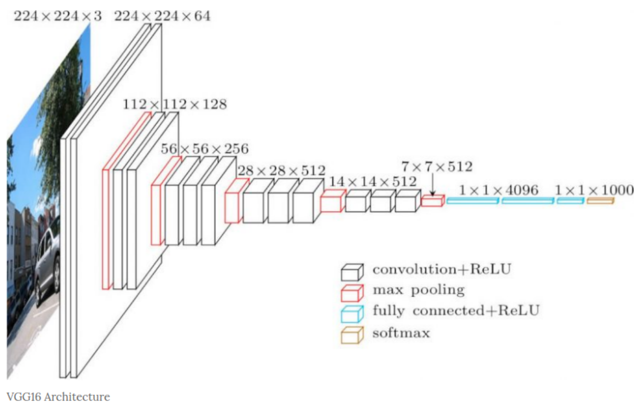


Fig. 5: VGG-16 Architecture

III. Results

Training data was divided in the 4:1 ratio, where 80% of the data was used for training and 20% was used for the validation. It gives the intuition about the over-fitting of the model on train data. VGG16 model is very good for the classification. We achieved 89.92 % accuracy on train data and 84.85% accuracy on validation data after just 15 epochs. Model was trained with the batch size of 15 and learning rate of 0.0001.Fig 6 shows the statistics of the model on training set consisting 20000 images. The red curve shows loss over epochs and blue curve show the accuracy over epochs. Fig 7 shows the same over validation data.

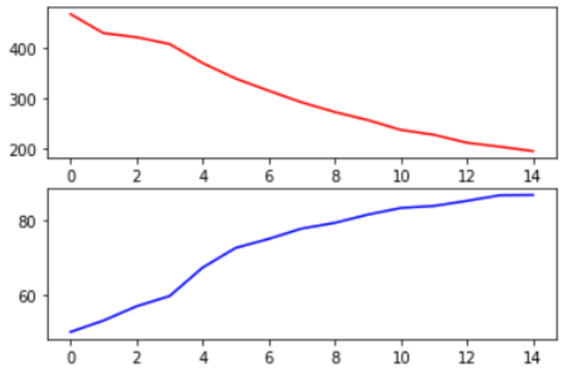


Fig. 6: Loss and accuracy of training data over epochs. Red line shows the loss and and blue line indicates the accuracy over epochs.

The figure 7 shows the statistics of the model on validation set.

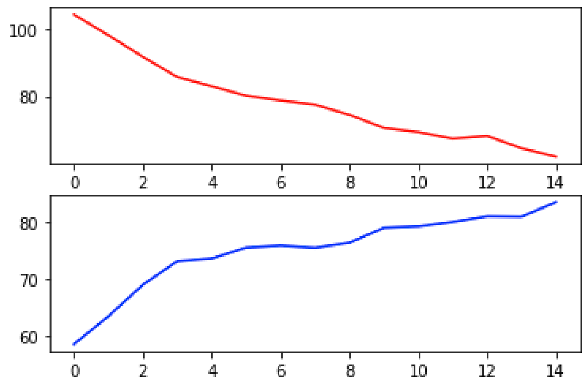


Fig. 7: Loss and accuracy of validation data over epochs. Red line shows the loss and and blue line indicates the accuracy over epochs.

IV. References

https://pytorch.org/docs/stable/index.html
https://neurohive.io/en/popular-networks/vgg16/
https://arxiv.org/pdf/1409.1556.pdf