Dog Breed Classification Using Convolutional Neural Networks

Gokul Anantha Narayanan, Raghavan Renganathan
{ananthanarayanan.g, renganathan.r}@husky.neu.edu
INFO 7390, Fall 2017, Northeastern University

1. INTRODUCTION

Machine Learning has been a trending research and experimentation topic over the past few decades. It has been implemented to solve most day-to-day problems. However, it is still challenging when it comes to computer vision. To address this challenge Neural Networks were introduced. Neural network is a system of computer software/hardware that is patterned after the working of neurons in the human brain. Deep learning refers to a subdivision of Machine Learning, wherein a system of neural networks is used for learning from data that is unstructured or unlabeled. One of the most popular deep learning technique is a Convolutional Neural Network which is a class of deep, feed-forward artificial neural networks [1]. CNNs are commonly used for solving problems related to Computer Vision. This project involves creating a Convolutional Neural Network to identify the breed of a dog from its image. The dataset used for this project is the Stanford Dogs dataset [2].

2. RELATED WORK

Classifying a breed of a dog is an example of a fine-grained image classification problem. Fine grained classification is challenging due to subtle differences among the images pertaining to each class. Among the various approaches for performing fine-grained image classification, one approach stood apart. This approach has been consumed in this project and has been discussed briefly below:

2.1 Part Localization:

For fine grained classification, the background information will contain more noise than useful information. Hence, the classification can be improved if the features used for classification is localized to these parts with meaningful information. [3]

This suggests that in our case, instead of using the whole image of the dog for training the model, using the cropped part of the image containing only the dog's face improves the performance of the model. A dog's body shape is not only difficult to identify and often not present in images, but also offers little additional information except in a more extreme case. [3]

The work done by Jiongxin Liu, Angjoo Kanazawa, David Jacobs and Peter Belhumeur in Fine Grained Classification using Part Localization [3] implements SVM regressor to detect a dog's face in each image and then feeds it to the Convolutional Neural Network.

Due to complexity in implementing SVM regressor to identify the dog's face in each image and time limitation of this project, we have used image annotations specifying the bounding box covering the dog's face. We then crop the image before feeding it to the convolution network.

3. DATASET

The Stanford Dogs dataset contains images of 120 breeds of dogs from around the world. This dataset has been built using images and annotation from ImageNet for the task of fine-grained image categorization. The dataset contains 20850 images of dogs in total. The dataset also contains annotations that include the breed name labels for each image and bounding boxes which indicate the region in the image that contains the face of the dog.

The dataset has the following distribution:

1. There are 120 breeds in total and each breed has approximately 150 images
2. Train set contains roughly 60% of the total images
3. Test set contains roughly 40% of the total images

Sample images from the dataset are shown below in figure 3.1

Figure 3.1 – Random Sample Images from the dataset

4. MODEL SELECTION

4.1 Pre-trained model:

The dataset is a subset of images from ImageNet. There are lot of models with pre-trained weights that are trained with all the images in ImageNet. Hence, these pretrained models can be used to obtain the bottleneck features from the images which are then used as input for the custom made fully-connected dense layer which classifies the image into respective classes.

The following are the pre-trained models that are available in KERAS library.

1. ResNet50
2. VGG16
3. Xception
4. Inception

In this project we have used all the above-mentioned models to classify the images.

The pre-trained models are used to obtain the bottle-neck features and then a densely connected layer is added to classify the images from the higher dimensional feature vector obtained from the pre-trained models. The model details are given below:

Pretrained Model	No. of Layers
ResNet50	50
VGG16	16
Xception	36
Inception	48

4.2 Our own ConvNet:

For this project, we constructed our own Convolution Neural Network to classify the images and compared the results obtained with those obtained from the pre-trained models.

The architecture of our Convolutional Neural Network is shown in figure 4.1.

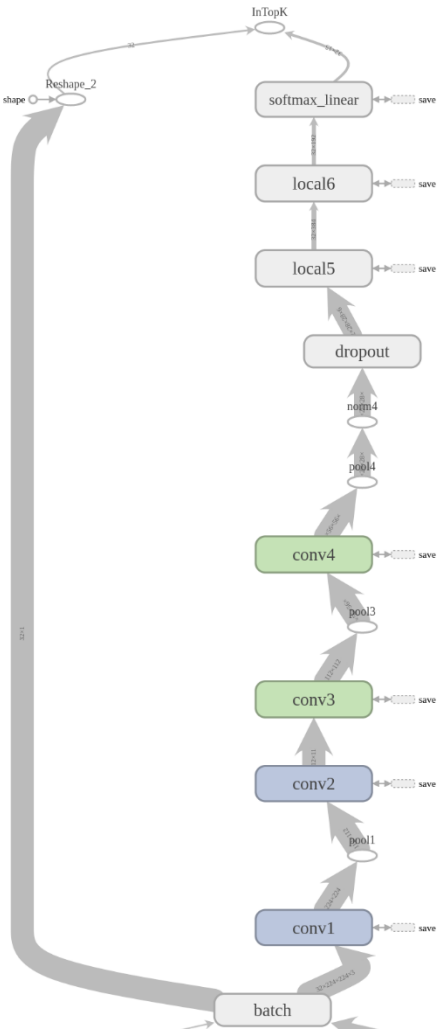

Figure 4.1 – CNN architecture

4.2.1 CNN Architecture:

Our CNN is stacked with the following layers. The functionality of each layer is briefly explained below:

- **Convolutional Layer:** Core building block of a Convolutional Neural Network that does most of the computational heavy lifting

- **Pooling Layer:** Progressively reduces the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting.
- **Dense Layer:** comprises of the traditional fully connected neural network which classifies the input to their respective classes

The CNN is designed with the following parameters:

Input layer: (224,224,3)
224 refers to no. of pixels in each dimension and 3 refers to the RGB values (channels) for each pixel.

Conv-1:	64 filters of size 5 x 5 **tanH activated**
Pool-1:	2 x2 filter of stride 2 **Max pool**
Conv-2:	64 filters of size 5 x 5 **tanH activated**
Conv-3:	64 filters of size 5 x 5 **tanH activated**
Pool-3:	2 x2 filter of stride 2 **Max pool**
Conv-4:	64 filters of size 5 x 5 **tanH activated**
Pool-4:	2 x2 filter of stride 2
Norm-4:	lrn layer
Local-5:	fully-connected **ReLU activated** 384 neurons
Local-6:	fully-connected **ReLU activated** 192 neurons
Dropout:	60% of the neurons
SoftMax-Linear:	fully-connected 120 neurons output

4.2.2 Salient features of the CNN model:

The layers are designed in the following way in our model:

1. The decaying weights are used for certain conv layers to prevent the weights from over shooting
2. The layers conv1 and conv2 use ordinary weights while the remaining conv layers use decaying weights
3. Less number of Pooling layers are used as too many pooling layers would result in feature loss
4. The layers except conv1 and conv2 uses decaying weights, due to which sparsity is introduced in these layers
5. Sparsity results in only few neurons in the layer being active, which effectively increases the speed and helps the CNN to consume less resources while training and evaluating the model
6. Biases are added to the convolution layers
7. tanH activation function is used for the conv layers and ReLU is used for the dense layers as tanH increased the accuracy of the model by 10% when trained with 6 classes.

5. METHODOLOGY

For this project, we have designed our classifiers using both pre-trained models and our own ConvNet.

Note: The pre-trained models are high-level and have as many as 20 – 30 layers. They are also trained with millions of training samples before they are used in this project. Hence, the model using these pre-trained models will have very high accuracy when compared with our custom-made CNN model.

The following steps discuss about how the inputs(images) are processed and the model is trained.

5.1 Processing images for the input:

The images of the dogs are placed in the folder containing its breed name. The whole list is populated in a data dictionary along with a field that specifies whether it is a training sample or test sample.

Step 1:
The images are cropped based on the bounding box specified in their corresponding annotations. This process replaces identifying the dog's face in each image to implement **Part Localization.**

Step 2:
These cropped images will have different dimensions which cannot be used for training the model. Hence,

all the images are reshaped to **224x224** pixels using crop-or-pad operation.

Step 3:

In order read the images quickly and to store them efficiently, we then encode the images and their labels into binary format. The images are converted into their (r, g, b) values and each value is converted into binary. The labels (breed names) are one-hot encoded and then converted into binary value. The data is then stored in the following format in a binary file:

<1 x label><150,528 x pixel>
…
<1 x label><150,528 x pixel>

In other words, the first byte will represent the label of the image and followed by 150,528 bytes which represent the pixel values of the image (224x224x3). The pixel values are placed in such a way that first 50,176 bytes represent the R values, followed by G values and then B values.

5.2 Preprocessing the images for the training:

Distortions are introduced into the training set to avoid the model from getting overfitted. The images are randomly flipped horizontally. Then the brightness and contrast of the images are randomly increased or

decreased to add more distortion.

There are various other approaches for adding distortions, but too much of distortions will result in the feature loss. Hence the above-mentioned distortions alone are used in the image pre-processing. Figure 5.2.1 shows some of the sample images after preprocessing.

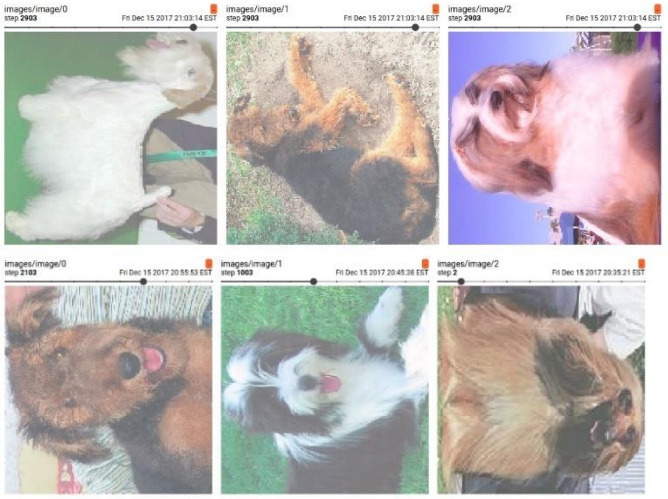

Figure 5.2.1 – Preprocessed images samples

5.3 Training the model:

The model is trained in the following environment:

Hardware Specs:

Processor: Intel i7 – 7700HQ
Physical Cores: 8
Clock Speed: 2.80 GHz
RAM: 8 GB
GPU: NVIDIA GeForce GTX 1050
Memory: 4 GB

Software Specs:

Tensorflow 1.3
CUDA 8.0
cuDNN 6.1
python 3.6

We used different approaches while training the pre-trained model and our own CNN model. This is because the pre-trained models will already have pre-trained weights along with the fixed architecture and hence can be trained with large dataset. On the other hand, our own CNN does not have optimized weights yet and hence the training process should be iterated by changing the hyper parameters.

5.3.1 Pre-trained models:

For implementing and training pre-trained models, we used keras library with tensorflow as backend. Keras provide the following pre-trained models:

1. RestNet50
2. VGG16
3. Inception v3
4. Xception

All these models are already trained with ImageNet dataset and hence can be used directly in our project to extract the bottleneck features.

Note: These are not completely trained CNN model that can be used to classify the breed of the dog directly. These models can only be used to extract the bottleneck features. We will still need the dense layers (fully-connected end layers) to get the complete model and then to train the whole model with our training dataset to be able to classify the breed of the dogs.

Once the bottleneck features are obtained from the model, they are then given as an input for the end layers which applies logistic regression to obtain the probabilities for all the classes. The model is initially trained for 15 Classes and after adjusting the parameters of the end layer, it is then trained and evaluated for all the 120 classes.

5.3.2 Custom made CNN:

The CNN model is initially constructed with most commonly used hyper-parameters. As we need to iteratively update the hyper parameters and if need the architecture of the model, initially 6 classes are only used for the training and evaluation of the model.

The top 6 class based on the amount of training samples present are chosen and the model is trained iteratively while the following hyper-parameters are changed. Then the number of classes are increased to 12, 15, 20 and so on.

5.3.2.1 Network Architecture:

Initially the model was constructed with only **3 conv layers, 2 pool layers and 2 dense layers**. Adding more conv layers did not improve the accuracy by significant amount when there were only 6 classes. As expected the accuracy dropped when more classes are included. Hence, few more layers are added to the model so that we can have maximum possible accuracy. In the end, having **4 conv layers, 2 pooling layers, 1 normalization layer, 1 dropout layer and 3 dense layers** provided the maximum performance. Hence this network architecture is used for the remaining process.

5.3.2.2 Activation Function:

Various activations functions are used for the convolution layers and the dense layers and the accuracy was calculated. When using activation functions like ReLU, sigmoid, Logistic, Gaussian, etc., the accuracy varied from 20% to 30%. But using **tanH** function provided the maximum accuracy of 44%, hence it is used for the convolution layers while using traditional ReLU function for the dense layers.

5.3.2.3 Cost Function:

Just like activation function, most of the cost functions resulted in decreased accuracy while **Cross Entropy** cost function maintained the accuracy at 44%. Hence, it is used as the cost function for our model.

5.3.2.4 Kernel Initializer:

Using various kernel initializers did not produce significant changes in the model accuracy or performance. Hence, we decided to use the default initializer provided by tensorflow module.

5.3.2.5 Optimizer Function:

The model was trained with Gradient Descent Optimizer, RMSProp and Momentum as optimizer function. The network plateaued fast enough when we used Gradient Descent Optimizer when compared to other optimizer functions. Hence, we went with Gradient Descent Optimizer for or model

5.3.2.6 Number of Epochs:

Initially we ran the model with only one epoch. But as the number of classes increased, the accuracy dropped. Hence, we specified a different value based on the following calculation:

Batch Size = 32 (i.e. 32 examples per step)
Max Steps = 100,000
Examples Per EPOCH = 10,000
(i.e. 32*100000/10000 = 320 EPOCH)

However, if the model plateaus for many consecutive steps, we stop the training before 320 epochs to prevent the CNN from over training itself.

5.3.2.7 Learning rate:

The model was trained with 0.001 as the training rate when we had 6 classes. But as the classes increased, the model took a lot of steps to plateau. Hence, we implemented decaying learning rate with the initial learning rate of **0.05** and the decay factor as **0.1**, which decays every **50 epochs**.

The model is then trained with all the hyper-parameters set to the optimal values.

5.4 Evaluating the model:

Like training, the approach for evaluating the model is different for pre-trained models and for our own CNN model.

This is because, unlike pre-trained models, the CNN will have to calculate weights which consumes lot of resources (CPU and GPU). Hence running both

training and evaluation in a single session without clearing the memory crashes the system. Whereas in the case pre-trained models, the weights are already calculated, hence will not consume as much resources as the untrained model.

5.4.1 Pre-trained models:

The train data is used for training the model and the fully trained model is provided with the test images and the predictions are made. The predictions are then compared with the actual values and the model is evaluated. The evaluation is done for the model with 15 classes and 120 classes.

5.4.2 Our own CNN model:

The model is trained for the training samples for the specified epochs or till the network plateaus. Then the model weights are stored on to the hard disk along with the tensorflow graph. Once the optimized weights are stored, the session is closed, and the memory is cleared.

Then evaluation is done. First, the model weights are obtained from the stored files and the tensorflow graph is restored. Then the test samples are given as the input to the model with optimized weights to obtain predictions. Then it is compared with the actual values to get the accuracy score of the model.

Note: The distorted images are only used for training the model. For evaluation purposes actual images are directly used.

6. RESULTS

We used keras as a wrapper for tensorflow for creating networks with pre-trained models. Hence, once the model is evaluated the output is printed which contains the results for that model.
In case of our CNN model, we used tensorflow directly to construct the model and train them. Hence, we created checkpoints frequently which contained all the data related to the model at that instant. Then we used **tensorboard** – a visualization tool for tensorflow to visualize various performance characteristics of our model along with the accuracy scores.

6.1 Pre-trained models:

The following table is populated with the accuracy scores for the pre-trained models which we used for creating the network.

Pre-trained model	Accuracy (Top 1)	
	15 classes	120 classes
ResNet50	75%	41%
VGG16	85.4%	59.6%
Inception V3	**98%**	71%
Xception	90%	66%

6.2 Our own CNN model:

The CNN is evaluated for 6 classes, then with 15 classes and then 120 classes. As the number of classes increases the model's accuracy decreased.

The accuracy score for Top 1, Top 3 and Top 10 are calculated whenever it made sense.

No. of classes	Accuracy		
	Top 1	Top 3	Top 10
6	44.5%	n/a	n/a
15	17.8%	32.6%	n/a
120	3.2%	7.5%	15.4%

Using tensorboard various characteristics of the model are shown below:
The value of the loss function and the cross-entropy at various steps are shown in the figure 6.1 and figure 6.2 respectively.

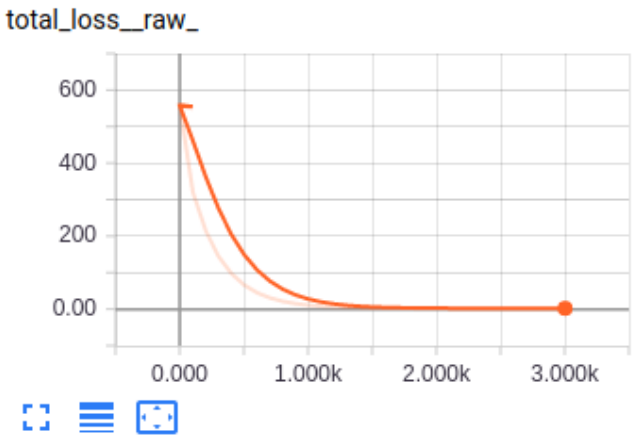

Figure 6.1 – Total Loss vs. Steps

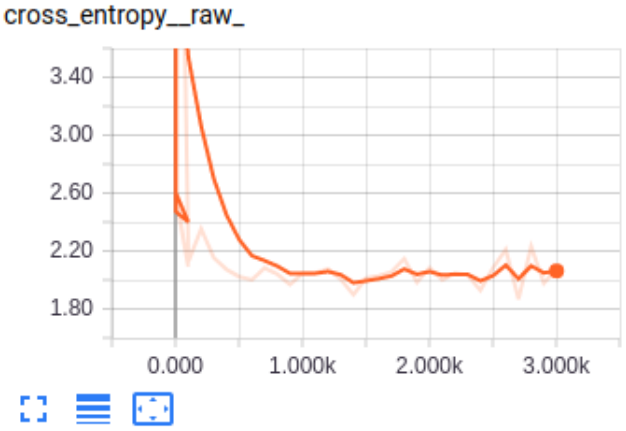

Figure 6.2 – Cross Entropy vs. Steps

The distributions of various parameters of the model like biases, activations, gradients and weights are shown in figures, figure 6.3 to figure 6.14.

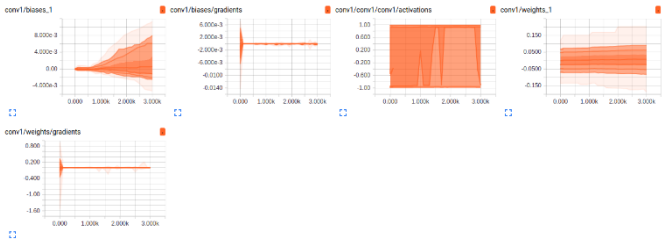

Figure 6.3 – Distribution of parameters for conv1 layer

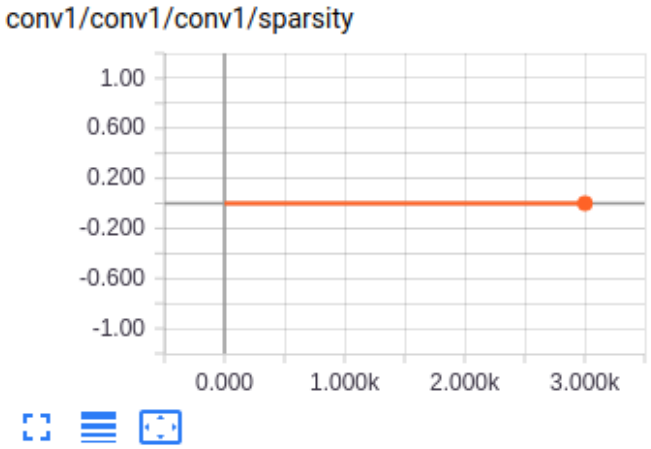

Figure 6.4 – Sparsity for conv1 layer

Figure 6.5 – Distribution of parameters for conv2 layer

Figure 6.6 – Distribution of parameters for conv3 layer

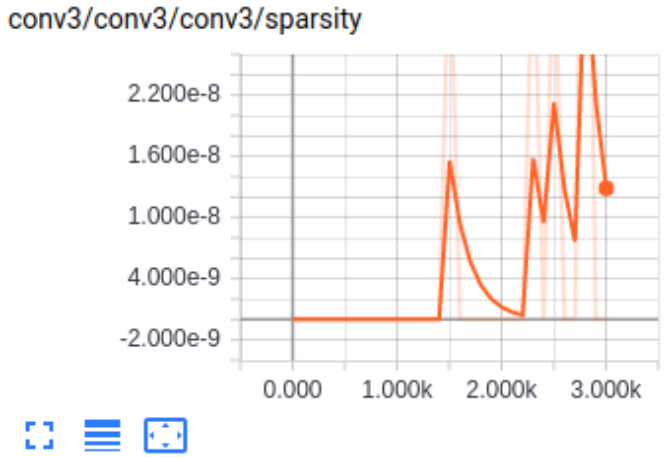

Figure 6.7 – Sparsity for conv2 layer

Figure 6.8 – Sparsity for conv3 layer

Figure 6.9 – Distribution of parameters for conv4 layer

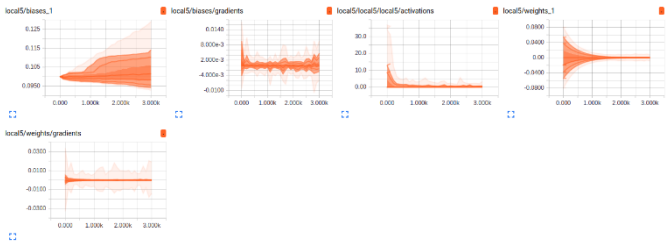

Figure 6.10 – Distribution of parameters for local5 layer

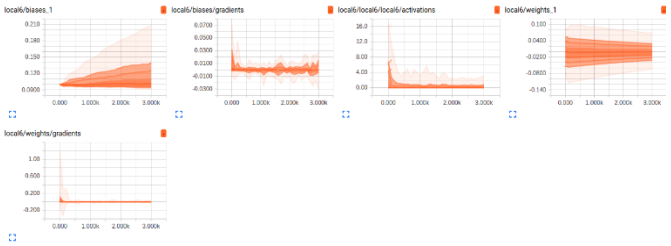

Figure 6.11 – Distribution of parameters for local6 layer

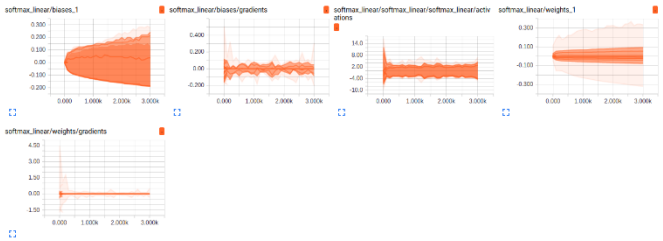

Figure 6.12 – Distribution of parameters for SoftMax layer

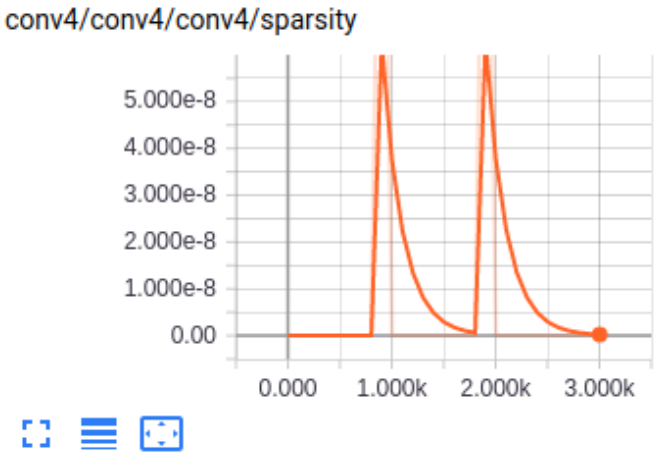

Figure 6.13 – Sparsity for conv4 layer

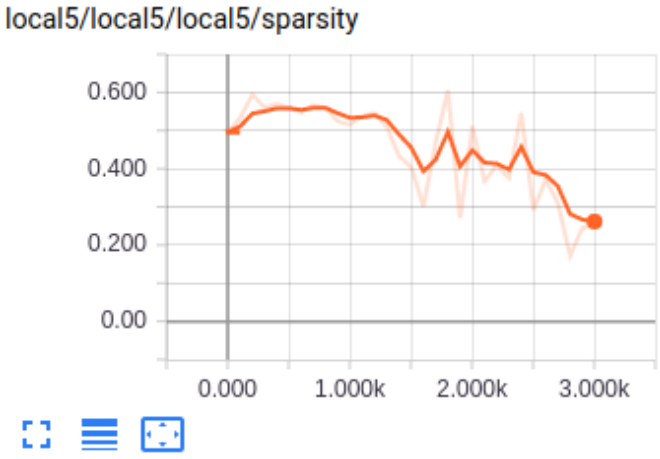

Figure 6.14 – Sparsity for local5 layer

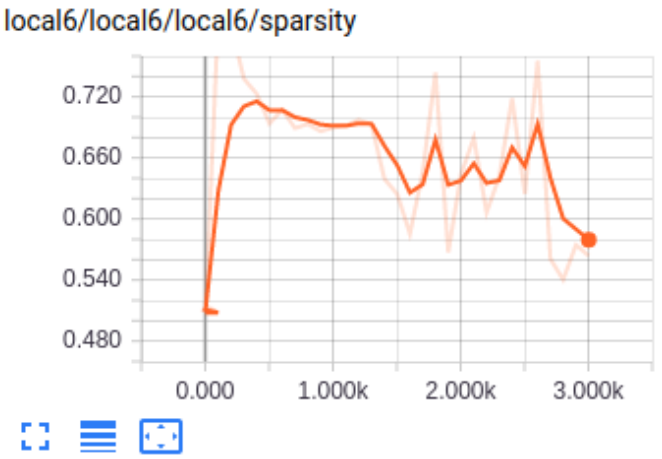

Figure 6.14 – Sparsity for local6 layer

From the distributions, the activation of the neurons in convolution layers 1 and 2 are distributed towards both the positive and negative ends uniformly. This suggests that the features are not extracted from the images completely. Hence using decaying weights in these layers as well might improve the activation and thus the performance of the model.

The sparsity graph of conv3 and conv4 indicates wrongly chosen decay value for the weights as the curve is not converging. The weights are very sparse during certain steps and gradually increasing in its values and the repeating the behavior.

7. CONCLUSION

We implemented a Convolution Neural Network to solve a fine-grained classification problem. We achieved 35% Top 3 accuracy among 15 classes and 15.4 % Top 10 accuracy among 120 classes. Using tensorboard, various hyper-parameters of the model and its effects on the performance of the model are studied. However, having just over 100 images per class for a fine-grained image classification problem is not enough for us to train our model completely to get a good accuracy. Even fine tuning the hyper-parameters is not helping beyond a certain extent as the model is getting over-fitted to the training dataset. Having more number of training samples will help the model to get more generalized and hence will improve its performance.

In future, we plan to implement multiple CNNs to extract features from multiple localized part like eyes, eyes and mouth and then the bottleneck features of each CNN are combined with a neural network to finally classify the image. This would improve the accuracy of the model by a greater amount for the fine-grained image classification problem. We also plan on collecting more training samples from ImageNet and other sources, such that the model can be trained properly.

REFERENCES

[1] https://en.wikipedia.org/wiki/Convolutional_neural_network

[2] Khosla, Aditya, et al. "Novel dataset for fine-grained image categorization: Stanford dogs." *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*. Vol. 2. 2011.

[3] Liu, Jiongxin, et al. "Dog breed classification using part localization." *Computer Vision–ECCV 2012* (2012): 172-185.

[4] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao and Li Fei-Fei. First Workshop on Fine-Grained Visual Categorization (FGVC), IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2011.

[5] Moghaddam, Baback, Tony Jebara, and Alex Pentland. "Bayesian face recognition." Pattern Recognition 33.11 (2000): 1771-1782.

[6] Viola, Paul, and Michael J. Jones. "Robust real-time face detection." International journal of computer vision 57.2 (2004): 137-154

[7] Angelova, Anelia, and Shenghuo Zhu. "Efficient object detection and segmentation for fine-grained recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2013

[8] Foody, Giles M., and Ajay Mathur. "A relative evaluation of multiclass image classification by support vector machines." IEEE Transactions on geoscience and remote sensing 42.6 (2004): 1335-1343

[9] Haralick, Robert M., and Karthikeyan Shanmugam. "Textural features for image classification." IEEE Transactions on systems, man, and cybernetics 6 (1973): 610-621

[10] Lu, Dengsheng, and Qihao Weng. "A survey of image classification methods and techniques for improving classification performance." International journal of Remote sensing 28.5 (2007): 823-870