Classification of images with Convolutional Neural Networks

This scripture documents the work done by Aditya Raj (471387) and Sören Schleibaum (474562) for the course Neural Networks with Statistical Learning at the Technical University of Clausthal. The date of committing is the 14th of March 2017.

Kommentiert [SS1]: Please add your imatriculation number here.

TABLE OF CONTENTS

1 INTRODUCTION

This work documents our work to classify images of cats and dogs with a Convolutional Neural Net (CNN). The overall accuracy is around 84 percent for the test badge after 100,000 steps of training with the implemented structure of a CNN.
We start by describing the background in 2 and focus on neural nets (NN) in general and motivate CNN from there. Afterwards we explain the basic concept of the machine learning framework we have used for solving the deep learning problem in 3. After a brief description of the actual problem in 4 we describe the code for solving the problem in 5 by shortly describing each module. Within in 6 we verify that our solution works and in 7 we summarize the results.

2 BACKGROUND

2.1 DEEP LEARNING

Deep Learning is a class of machine learning algorithms which aim to learn representations of data using multiple stacked layers of non-linear processing units for feature extraction and transformation. In such an arrangement, the output from the current layer becomes an input to next layer. There has been significant ongoing research in this field since past decade to make better representations of data and learn the labels from a large scaled and unlabeled data. Under this learning paradigm, we need to acquire labeled training data and test data.

2.2 NEURAL NETS

A Neural Net (NN) is one example for a deep learning structure. It can consist of different Neurons which are structured in layers. The layers and its neurons are connected via edges. One simple example for a NN with two hidden layers is shown in the following graph.

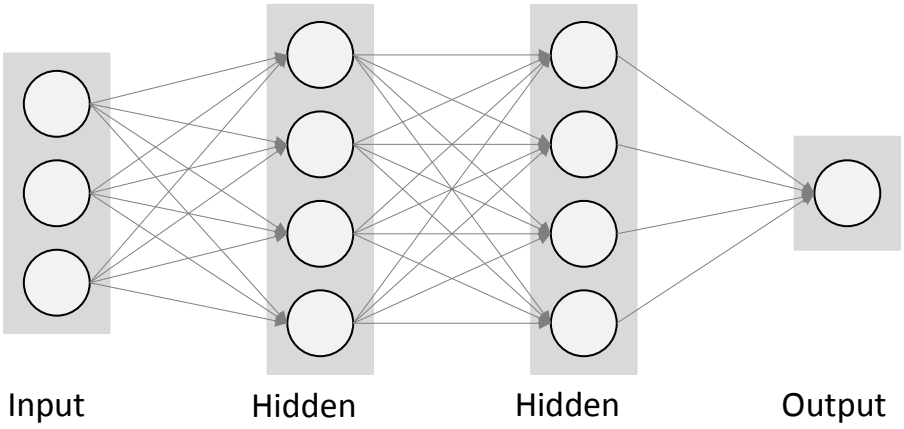


Figure 1: Sample structure for a Neural Net *(Karpathy)*

The idea to train a deep learning systems such as a NN to learn patterns in the input has originated from the computational model of a human brain. The brain consists of neurons approximately 86 billion neurons which are connected over edges. Neurons are the individual cells which decide if an input value is sufficient to provide an output (which is 1 if a neuron should fire and 0 for vice-versa). Similar to a neuron response to certain patterns or activities as an input, an artificial neural network is also based on training of a mathematical function based on certain parameters to provide a response when provided with certain stimuli also known as function input in mathematical notation. From the mathematical perspective, the idea is to: train a function on parameters called weights and biases, which are randomly initialized in the beginning. The output is then predicted based on these weights and biases. Further, we compute how much the correct output value (which comes from the given label in case of supervised learning) differs from the computed function value. This step is termed as Loss computation. Finally, based on the mathematical algorithms (like Gradient Descent, Adam Optimizer, etc.) the weights and biases are adjusted to expect lower loss value in the next iteration. This concept is depicted with the help of following diagram:

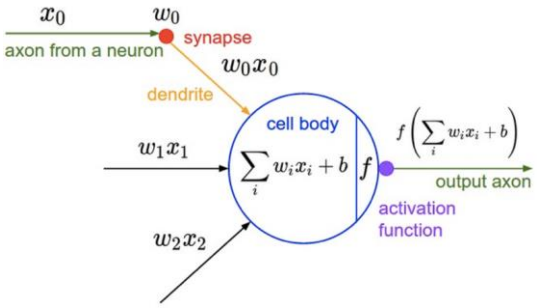


Figure 2: Mathematical model of a neuron *(Karpathy)*.

Here an input X_0 to a neuron is multiplied with the weight and added to the bias to compute the output value. This step is done for all individual inputs (i.e. from $X_1, X_2, \ldots X_n$) and summed up together. Further, the output value is passed to a function which decides if the function output value exceeds the threshold value to fire a neuron. This operation is carried out in multiple layers and finally we get a label 0 or 1 at the end (Karpathy).

For an image classification problem, we cannot afford to have so many parameters (weights and biases). Therefore, researchers decided to modify the architecture of a general neural network by providing a mechanism to reduce the total number of parameters involved in training the image data set. This also removes the possibility of overfitting the function to predict images by learning large collection of parameters. Further, it also ensures that we make our predictions based on important matching features of an image data. This process involves a multilayer feature learning backpropagation algorithm to update weights and biases in the background to train the data sets. At the end when the model is trained on the parameters, we save these variables and use them to predict a new input image (Karpathy).

2.3 Convolutional Neural Networks

Due to the lower number of parameters within the network CNNs are preferred above NN in case of image classification. This reduces the time used for training. The reason for the decreased number of parameters is the insertion of a convolutional layer. Beside this one, the following text describes briefly input, pooling, normalization, fully-connected, SoftMax linear, and output layer. Neither has every CNN all the listed ones nor are these all possible layers. We have chosen to describe these ones shortly because of their use within the network used to classify images of cats and dogs. The structure of the CNN shown in Figure 8 has multiple appearances of convolutional, pooling, and normalization layer. Within neural networks several techniques are used two perform the task of classification. The concept of an activation function and of backpropagation are described within this work.

Activation functions are used after the dot product of a neuron is calculated and the bias added. They are used within convolutional and fully-connected layers. A common choice for constructors of NN was the sigmoid function. It maps a real number to the range between [0,1] and has the form

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and fall out of favor because of different reasons. In case of an output close to 0 or 1 the gradient will become very small. This means only a small update of the parameters during backpropagation. Another activation function is the ReLU function. Its formula has the following form:

$$f(x) = \max(0, x)$$

This means that all outgoing values are at minimum 0. The function is continuous but not differentiable everywhere. In comparison to the sigmoid function it speeds up the gradient descent and is faster to compute due to its simple form.

Backpropagation is used to update the weights within the network. Based on the loss between prediction and actual value which the training data has, the local gradients are calculated. With an application of the chain rule by going recursively through the network all weights can be updated.

The **Loss-function** calculates the difference between the calculated prediction and the known real label of the given data. The total loss is an average of the loss over all training data. In case of a single correct class for each label the following formula for SoftMax classifier is one example:

$$f(x) = \log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_i}}\right)$$

The corresponding formula for f is $f(x) = f(x_i; W)$ where x_i is the data and W represents the weights.

The **input** layer gets the input data and passes it forward to the next layer. Within this layer no computation is performed.

In traditional NN fully-connected layers are used over the whole range of the network. Within CNN the **convolutional** layer replaces fully-connected layer especially in the early layers. The main reason for is the reduced number of weights. Because of this reduction, the training via backpropagation is simplified and the overall training time is decreased (Karpathy, kein Datum).

The **pooling** layer reduces the size of the data by applying a function like max- or average-pooling is computed on the given data. Beside the function to hyperparameters can be chosen: the spatial extend and the stride. When stride and spatial extend are two and the given input is a 4x4, the output data is of dimension 2x2. This means a reduction by factor two for the outgoing data.

Within the **normalization** layer the values are normalized or inhibited. The layer helps to reduce the size of parameters and the excited neurons can fire relatively more. Within the current development of CNN, the normalization layer fall out of favor, because there is little to no documented use.

Fully-connected or **local** layers are used to combine the results of all filters especially in the end. Each neuron in this layer is connected to all outgoing edges of the previous node. It is possible to convert a fully-connected layer into a convolutional layer by increasing the local region of the convolutional layer.

- The **softmax-linear** layer

The **output** layer computes the predicted value for the input data for each class. No activation function is applied here.

With the described layers a CNN can be constructed. Passing through all layers the input data is reduced through several computations until only the prediction is left. In the example of cat and dog classification for each image a possibility for both classes are the output. The initially random weights

are updated during the training process. Within each training step calculations on the current input are performed including dot products, activation functions and for instance max-pooling. After the loss is computed the weights are updated through backpropagation or the learned concepts are applied to the network. The trained model can afterwards be applied to new and never seen data to predict.

3 TensorFlow

To solve the image classification problem described above we have chosen to use the programming language Python[1] version 3 and TensorFlow[2] version 1.0. Within the area of Machine Learning multiple software libraries are available. We have chosen TensorFlow because of its multi-language support. Moreover, TensorFlow provides detailed documentation and tutorials and huge API library for creating a neural network. This prevents us from re-inventing the wheels of neural network and instead focus on the important concepts of modifying a neural network for our image classification problem. We also believe that our project can easily be migrated to multiple GPUs for faster processing of neural network. But we were not able to test this feature because of time frame restrictions.

Kommentiert [SS2]: Update link

Kommentiert [SS3]: Line break is bad

Kommentiert [SS4]: Add link

Kommentiert [SS5]: Source is missing

The TensorFlow API library supports general operations used in machine learning. It was released by Google Brain Team in November 2015, to provide specific support for Deep Learning Neural Networks. It provides support for widely used programming languages like Java and Python. The TensorFlow computation engine is designed to run smoothly, both on Central Processing Unit (CPU) and GPU (TensorFlow, n.d.). (Krizhevsky, 2009)

The concept behind TensorFlow is to perform graph based computations. In this graph based architecture, nodes are considered as computational functions whereas edges represent the data flow pipeline. Nodes are connected to each other via unidirectional link via edges and data flow occurs through Tensors.

Tensor is a class provided by the TensorFlow library, which can store multi-dimensional arrays. For instance, an RGB image of height and width of 60 pixels is represented as a tensor of shape 60x60x3. In the color representation, the last dimension stands for the three-color components namely: red, green, and blue. The tensors are stored in a format provided by the Python scientific library NumPy[3] (TensorFlow, n.d.).

Kommentiert [SS6]: Set up the right link within the footnote

Nodes represent computations within the graph model. It can have multiple inputs and outputs. The incoming data from an edge is processed within the node and passed to all its outgoing edges. One example for a node is a pooling layer. When data comes through an incoming edge, the pooling function of the corresponding layer is applied and the modified tensors are passed to the outgoing edges (TensorFlow, n.d.).

The training or later usage of a model works in two phases. Firstly, the graph, consisting of nodes and operations, is constructed and the input data is inserted into the graph. Secondly the graph is placed on a device, for example the CPU. Now to run the computation on the input data using the pre-defined graph architecture, a session needs to be initialized and run in the next step. Further, the defined neural network learns the parameters and updates the inputs on the same graph iteratively.

[1] https://www.python.org/
[2] https://www.tensorflow.org/
[3] http://www.numpy.org/

4 PROBLEM

A common problem for CNN is the classification of images. We chose to classify between cat and dog images. The dataset used for this course was provided by Kaggle[4], a host for competitions in the machine learning environment. The aim is to predict the real class of given images. The classes are mutual exclusive. Because of the known output value for the training data this classification problem is a supervised learning one. The training dataset consists of 25,000 images, 12,500 for each class. Moreover, we have got 12,500 images which are not labeled and which we can use for manual evaluation.

In general, some of the images are in the dataset multiple times and the shape and content of the images vary. For instance, an image could also contain two cats or a human and a cat. Moreover, some of the images are only black images or text. From 100 randomly chosen images three images where including one or more of the previously described errors.

[4] kaggle.com

5 DESCRIPTION OF PROGRAMMING CODE

Cifar-10 is a popular image database for image recognition. It consists of 60,000 color images of one the 10 object classes namely airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. To predict the images provided by Kaggle, out of the two possible classes, i.e. cat and dog, we modified the cifar10 code and also converted the dataset consisting of 25,000 images into cifar10 compatible batch files.

We can visualize our entire work as a black box which takes the input image in form of cifar10 batch format and produces an image label at the end. The entire system model processes a sequential mathematical computational model on every image. This architecture follows the following sequence: Firstly, we need an input image which is 32 * 32 image in a cifar10 batch format. Then, we build an inference of this image using inference function provided by cifar10.py module. In Inference method, Cifar10 batches go through all the layers of CNN as mentioned in the Figure 3 with the stated dimensions. Further we compute the SoftMax output of the output information from previous layer. Finally, we measure the SoftMax cross entropy to get the difference between actual and predicted value. TensorFlow provides tf.nn.in_top_k method to predict the top k predictions of a given image for the given labels. We take the maximum value of these predictions to know the top most class. These functions are elaborated in more detail in the following subsections.

The code used and created within this project is accessible under a private repository on GitLab[5]. The cifar10 modified Tensorflow code along with additional python files to convert the data set into cifar10 batch files is explained below.

5.1.1 Cifar10

This module is the building block for the entire Tensorflow model. The functions defined are summarized as follows: **_activation_summary(x)** method helps creating summaries for activations of a tensor 'x' and provides a histogram for the activation. These results are then plotted in the TensorBoard. **_variable_on_cpu**(name, shape, initializer) method creates and returns a Variable stored on the CPU memory with inputs arguments: name of the variable, shape (list of integers), and initializer for Variable. If the variable does not exist on the memory, it creates a new variable and then returns it. The function used to get this information is *tf.get_variable(name, shape, initializer=initializer, dtype=dtype).*The function **distorted_inputs** constructs distorted input for CIFAR training. This returns images (4D tensor: batch_size, height, width, depth), and labels (1d tensor: batch_size). The distortion of the image is done by taking randomized brightness, contrast, and cropping. This is quite useful for repeatedly training the model not only on the given image but also using the multiple copies of the same image, and further doing randomized modifications to these images. Finally, we build the Tensorflow cifar10-model from distorted_inputs function using **inference** function. This function returns a softmax_linear output. The overall structure of the inference function follows the diagram shown in Figure 8: *Structure of the implemented CNN*Figure 8.

5.1.2 Cifar10_eval

This module is to evaluate the test batch with correct prediction labels. It produces the precision of accuracy by comparing the actual output with the predicted output. It provides two methods namely eval_once and evaluate. The first method checks for the existing checkpoints (created after training a CNN model) and restores the parameters learnt during the training phase. It then calls evaluate function to get label of the images, builds the model using inference method, predicts the output using and builds the summary operation useful for visualization in TensorBoard.

[5] git@gitlab.com:Cranefly/ConvNet.git

5.1.3 Cifar10_eval_single_directory

It is a modified version of cifar10_eval.py. We add an input argument called 'logits' to **evaluate** function to write the labels predicted in a text file or display this result in the python console. The logits argument passed here is computed by calling the inference function on the image batch. To print this result, we display the logits[0] result and compute the maximum value of the tensor shaped [1, 2] within a TensorFlow session. The index pointing the maximum value of this array contains the maximum probability for the occurrence of a certain class for a given image and therefore we take this index and print the image as "cat" for index 0 and display it as "dog" for index 1.

5.1.4 Cifar10_train

This module trains a batch of images inputted in cifar10 format. The global constants determine the number of training steps, training directory for input batch and maximum steps to run the training set. It follows sequentially the following order to train a batch of images: Get images and labels for CIFAR-10, Build a Graph that computes the logits predictions from the inference model, calculate loss, build a graph that trains the model with one batch of examples and updates the model parameters, create a saver, initialize all TensorFlow variables and run the TensorFlow session. On completion of every 10 steps it displays the loss value and at every 1000 steps it saves the variables in a checkpoint (so that we need not start training from scratch when we run this train file again).

5.1.5 CreateTestBatch

This module helps converting a directory full of images into batches of images in cifar10 format, so that we may use it further for training and evaluation purposes. To understand in detail about creating the cifar10 batches, we need to first mention how a cifar10 batch looks like. As mentioned on cifar10 website (Krizhevsky, 2009) the structure of this batch is defined as "the binary version of cifar10 should contains the files 'data_batch_1.bin, data_batch_2.bin, ..., data_batch_5.bin', as well as 'test_batch.bin'. Each of these files is formatted as follows:

<1 x label><3072 x pixel>

<1 x label><3072 x pixel>

where the first byte is the label of the first image, which is a number in the range 0-9. The next 3072 bytes are the values of the pixels of the image. The first 1024 bytes are the red channel values, the next 1024 the green, and the final 1024 the blue. The values are stored in row-major order, so the first 32 bytes are the red channel values of the first row of the image."

We also stick to the same binary definition of the images in the training directory. First, we create a pickled data structure for the images in this directory with the key values as [data, labels, filenames, height, width, ratio, size]. Pickling is basically the process to convert a Python object hierarchy into a byte stream. We chose pickle data structure because of the efficient storing and retrieval of the key-value pairs on-demand. We chose to store other parameters (like height, width and so on) to do image analysis and obtain patterns like size distribution, image ratio distribution and more on given image data.

First, we divide our given image data set in 80:20. We take the 20,000 images for training and 5,000 images for testing. Now to create our first binary batch file, we read the file name, write its label based on filename (0 for cat and 1 for dog) in binary file, read the image, and finally append the image array in the binary file. We loop this process for the 4,000 images to create one batch of cifar10 input image. We do the same iteration for remaining batches.

5.1.6 Evaluate_single_image

This file defines classes and methods to predict the correct label of a single image file or a list of image files in a directory. The idea behind predicting a list of images is to iterate over all the images in a directory and apply the same logic as of predicting a single image. The results of this prediction are stored in *Resources->EvaluateImageResults* directory.

To predict a single image, we create a test batch of binary file (as explained in 'CreateTestBatch.py') from single input image. Further, we fake the same image 128 times so that each batch has the dimension of 128 x 24 x 24 x 3. This is the requirement of building an inference on the image, and finally getting a *Softmax Linear* output from the CNN layer. Although this may not seem a good approach, but we could not find any better solution to predict results in more organized fashion. We tried finding solutions on forums like Stackoverflow [6], google group discussions [7] and github [8] community groups but on the contrary, we found that many developers have faced the same problem and none of the posts on these groups point to a working/better solution to this problem.

This file provides two classes **evaluate_single_image**, **predict_files.** The former class is responsible for creating cifar10 batch of 128 images from single image using method **create_binary_file**. Method **get_item_names** a list of image names that are used to testing the cifar10 model. These images are the 20 percent images from training data set and the image names are retrieved from a pickle file stored in *Resources -> OtherResults -> test*.pickle.* predict_files class defines methods to predict single image or list of images stored in a directory or evaluate the 20 percent image files from training data set.

5.1.7 Helper

It provides a list of utilities to retrieve/add information like get current date, get path of the file, get parent directory, pickle a given file, unpickle a given file, create a directory and check if a directory exists. These methods are used at several places in the cifar10 modules to meet the training and prediction requirements.

5.1.8 ImageInformation

This file provides methods to store data, labels, filenames, height, width, ratio and size information of an image in a python dictionary data structure and further pickle this information. This is done to store the list of image attributes in an efficient way and further retrieve it on-demand. This also allows faster access to the image related information. It is also being used in evaluate_single_image class to get the filenames of 20 percent training datasets for evaluation purposes.

5.1.9 Resizer

This class provides **resize_image** method which takes an input directory (containing a list of images) as a function argument, resizes them and finally saves them to a given output directory. Cifar10 expects 32 x 32 x 3 image for training the CNN. Therefore, resizing method is run on Kaggle training data sets to resize it to 32 x 32 and thus make the images compatible with cifar10 input pipeline.

5.1.10 Settings

This file defines global path variables and creates missing directories required for training and evaluation purposes.

---

[6] https://stackoverflow.com/
[7] https://groups.google.com/forum/#!overview
[8] https://github.com/open-source

6 EVALUATION

6.1 ENVIRONMENT

The machine used to train and evaluate the network is 64-bit Linux system with 7.7 Gigabyte of memory. The computations are performed on one core of an Intel Core i7-4510U CPU with 2.00 GHz x 4. Due to the high time used to train the network the test run through the night without performing other heavy computations on the machine.

6.2 AUTOMATED VERIFICATION

Within this paragraph, we describe four cases that we have evaluated. This first case is **Standard 100 k** which runs with the CNN structure shown in Figure 8. It ran for 99,900 steps on a Windows machine and took over 6 hours to complete. Table 1*: The training results of the different CNNs running on different computers are listed below. Total loss and time measured at the last step. The machine refers to the two previously described machines.*The **standard 40k** runs with the same CNN structure but on the Linux machine. It took nearly 8.5 hours to compute 40,600 steps. Afterwards we have added a convolutional layer directly after the first convolutional layer. This case is referenced to as **Additional ConvLayer**. It took over 9 hours to perform 13,500 steps and reach a total loss of 0.3128. Within the last case, I**ncreased size**, we increased the size of images used in the network from 24 to 28 pixels for width and height. It took 8 hours and 40 minutes to reach 0.3346 of total loss. This result appeared after 30,100 steps. Table 1 refers to the results and the development of total loss is described and visualized in the following.

We aim to plot the loss function used in our CNN and see how the loss progresses as we proceed with training of our network. It is generally expected to decrease with an increase in number of steps. In a SoftMax Linear output function the derivative of the cost function with respect to an individual input Z_i is $\frac{\delta C}{\delta Z_i} = Y_i - T_i$ (where Y_i refers to the predicted output and T_i refers to the actual target output). We get negative slope when the actual value and predicted values are opposite. So, we expect a very steep slope at the beginning and further as the time progresses, the network tries to optimize itself at a lower learning rate.

	Number of steps	Total loss	Time	Machine
Standard 100k	99,900	0.1132	6h 15m 50s	Windows
Standard 40k	40,600	0.3316	8h 26m 58s	Linux
Additional ConvLayer	13,500	0.3128	9h 19m 34s	Linux
Increased size	30,100	0.3446	8h 37m 30s	Linux

Table 1: The training results of the different CNNs running on different computers are listed below. Total loss and time measured at the last step. The machine refers to the two previously described machines.

Kommentiert [SS7]: Maybe we should add the overall precision as well

On the X axis, we plot Number of training steps in thousands and on the Y axis we plot the SoftMax loss output. We also take smoothened visualization of the graph shown in dark black and plot the actual graph in light grey because of knowing how the loss function trends over increase in number of steps. Because of big oscillations in the graph, we prefer representing the values by smoothened curve. These tests were done on different operating systems with different system configurations. As of result of such an arrangement we can see different run times in the table. Also, we see a lesser average loss value when we have trained more number of steps.

Standard 100k: Figure 3 shows a decrease in loss value as the training steps increase over time. This represents the total loss of the standard CNN until 99,900 steps on a Windows machine. After a steep decline in loss function output the function oscillates to find the global minimum to fit perfectly in the

given input data. We can also notice a decrease of learning rate at 0.08 at step 65,000 to 0.01 at step 70,000. Because of training the network for 100,000 steps the learning steeps from 0.08 to 0.01.

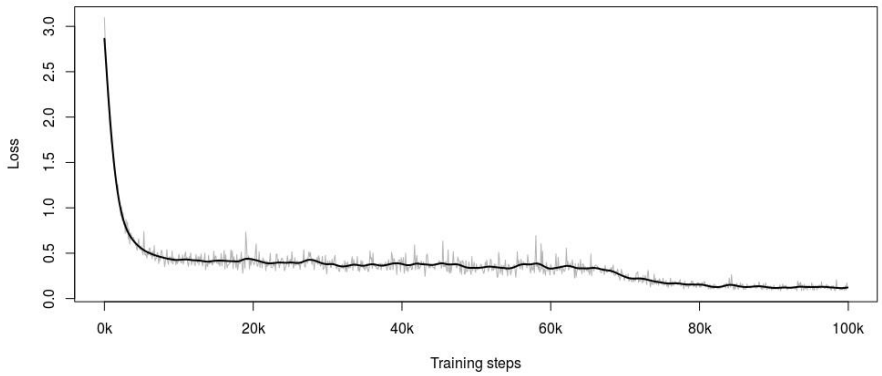


Figure 3: Total loss of the standard CNN until 99,900 steps on a Windows machine.

Standard 40k: Figure 4 shows a decrease in loss value as the training steps increase over time. This represents the total loss of the standard CNN until 40,600 steps on a Linux machine. After a steep decline in loss function output the function oscillates to find the global minimum to fit perfectly in the given input data. This also takes longer time to run because of running only on CPU in Linux machine. Here we also notice no decrease in learning rate because we not trained the network for many steps. But in general, the graph looks similar to standard100K. Only we can see different initial average loss values because of different randomized weight initializations.

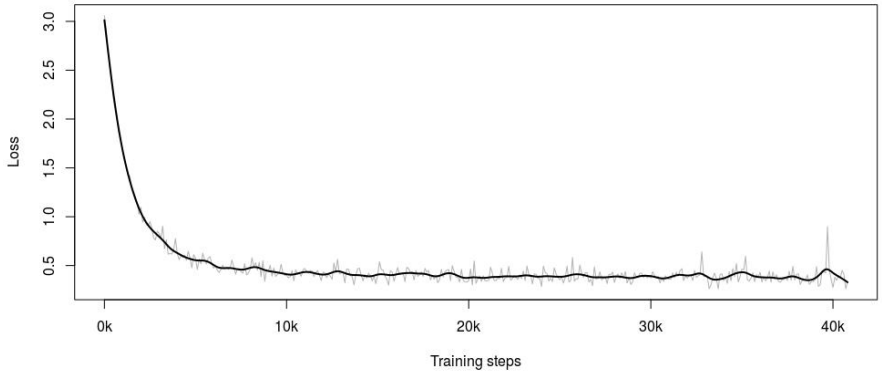


Figure 4: Total loss of the standard CNN until 40,600 steps on a Linux machine.

Additional Conv Layer: Figure 5 shows a decrease in loss value as the training steps increase over time. This represents the total loss of the standard CNN until 30,100 steps on a Linux machine. This testing is done with an increased input image size that is from 24 x 24 x 3 we have increased the neural

network size to 28 x 28 x 3. It can be concluded from the graph that with an increase in CNN input image size, there is less oscillatory behavior in the graph because of the narrowing down of the computational elements in the network. Here we also note that the time to train to network gets longer because of the increased image size.

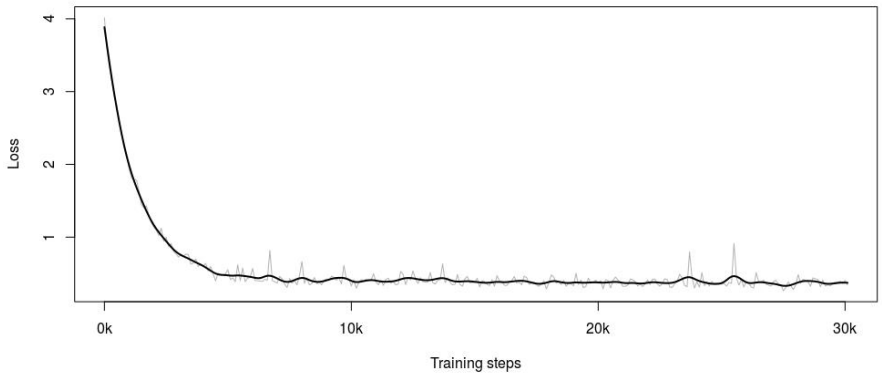


Figure 5: Total loss of the standard CNN with increased input image size until 30,100 steps on a Linux machine.

Increased Size: Figure 4 shows a decrease in loss value as the training steps increase over time. This represents the total loss of the standard CNN until 13,500 steps on a Linux machine. Keeping the input image size to 24 x 24 x 3, we have added an extra 'conv' layer. It can be inferred from the visual inspection of the graph that the oscillatory behavior of the function has reduced because of an added conv layer. This happens because we provide more number of features to learn for the same layer and thereby increasing the number of parameters to accurately get the prediction for the image. To summarize this, we can say that increasing the granularity of the image, is leading to more oscillations. Certain areas of the image may trigger a neuron and force it to adapt to the actual output. It leads to not much improvement of the neural network but may lead to overfitting of the curve. However, it can also be seen that the training time triples to that of training around 40,000 images without an extra conv layer.

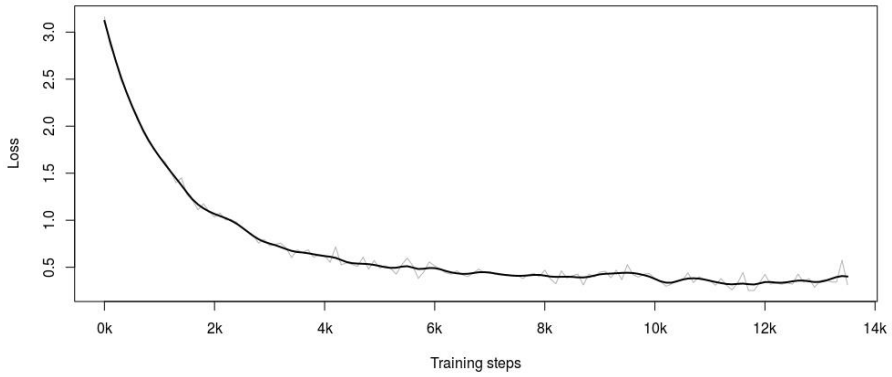


Figure 6: Total loss of the CNN with an added convolutional layer until 13,500 steps on a Linux machine.

The overall loss computation in each of the four scenarios can be summarized together in Figure 7. This is done to for a comparison perspective. We can see that there are less number of oscillations in Additional Conv layer graph as compared to other three because of more narrowing down of the network and thus reducing the number of computational elements in the network which makes it much easier for the network to predict correct value. The increased image size graph takes the longest time to train among other three scenarios. Moreover, we also see a steep learning rate decay after training for longer time steps.

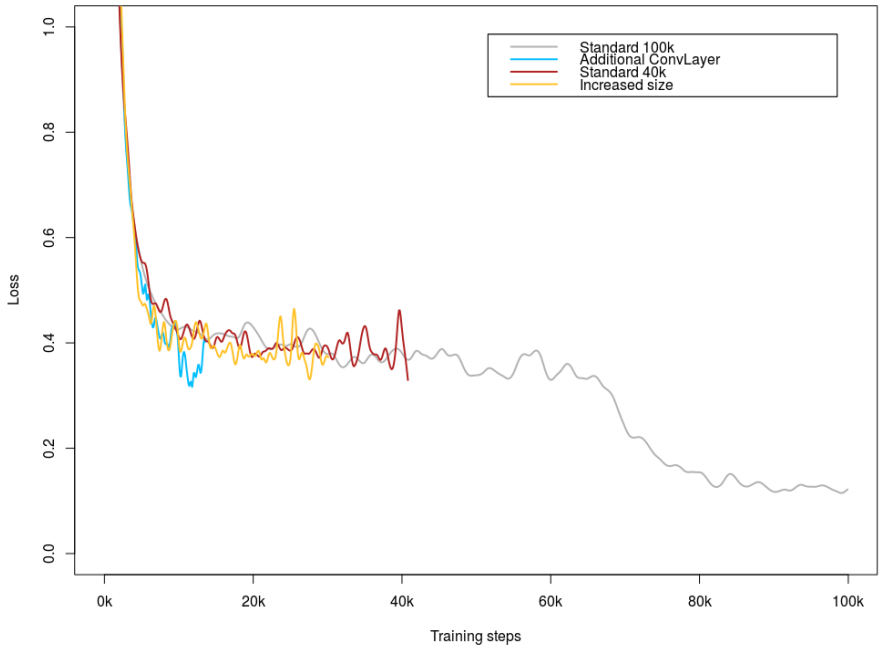


Figure 7: Total loss of the different options compared.

6.3 Manual Verification

From the 12,500 test images, we did manual verification of first 100 images and cross referenced our visual inspection with the predicted output. The prediction was done from the CNN trained with 100K training steps. Overall, seven predictions were wrong out of these first 100 images.

Images numbered 2, 3, 4, 5 and many others shown in Table 2 were correct predictions. It becomes quite clear from the predictions that the network predicts the image accurately when it could easily recognize important features of these classes of animals being trained. In this case, it seems quite reasonable to easily distinguish facial and other body features even by visual inspection.

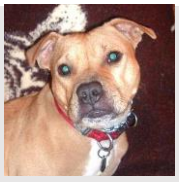
			
2.jpg	3.jpg	4.jpg	5.jpg

Table 2: Correctly predicted images.

Images numbered 24, 29, 33, 62, 63, 64, 76 shown in Table 3 were incorrect predictions. It becomes quite clear from the predictions that the network fails to predict these images accurately when it could not easily recognize important features of these classes of animals being trained. In this case, it seems that because of background interference, dark background, multiple animal images or improper rotation angle of images, the animals were hard to recognize.

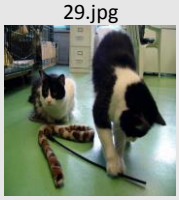
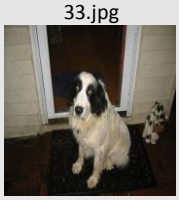
			
24.jpg	29.jpg	33.jpg	62.jpg
			
63.jpg	64.jpg	76.jpg	

Table 3: Wrongly predicted images.

Images numbered 32, 59, 90 shown in Table 4 were confusing even after visual inspections, However, some of them were classified correctly by the CNN. The reason for these images appearing confusing is because of lying postures of animals, not facing directly towards the viewers and other reasons argued in Wrong images.

		
32.jpg	59.jpg	90.jpg

Table 4: Confusing images.

7 SUMMARY

From the 25000 images of the cat versus dog classification problem, we trained and tested our model in 4:1 ratio respectively. After training for 100,000 steps we could achieve around 84 percent accuracy on the test batch consisting of 5,000 images. The results within the manual verification section show even better results. The accuracy was 93 percent for the tested images.

Also, as a part of our project completion requirement, we were successfully able to predict 12,500 test images provided on the Kaggle competition and checked the results partly. After the results of automated and manual verification from the previous section, the implemented solutions could be verified. Some options for future work are to initially clean the dataset from duplicates so that the data is more clear. Moreover, the structure of the CNN and its parameters could be tested in more detail to increase the accuracy further.

8 REFERENCES

TensorFlow. (n.d.). *Basic Usage*. Retrieved 03 07, 2017, from https://www.tensorflow.org/versions/r0.10/get_started/basic_usage

Karpathy, A. (n.d.). *CS231n Concolutional Neural Networks for Visual Recognition*. Retrieved 03 07, 2017, from http://cs231n.github.io/convolutional-networks/

- TensorFlow documentation
- Python 3 documentation
- Alex Krizhevsky and his description of the net used
- ImageNet paper should be used
- The literature Feng send us should be used
- http://neuralnetworksanddeeplearning.com/chap3.html#the_cross-entropy_cost_function
- https://www.cs.toronto.edu/~kriz/cifar.html

9 APPENDIX

Figure 8: Structure of the implemented CNN