# 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 13th of March 2017.

**Comment [SS1]:** Please add your imatriculation number here.

# 1 STRUCTURE

1 Structure........2
2 Introduction........3
2.1 Problem........3
2.2 Neural Networks........3
2.3 Convolutional Neural Networks........3
3 TensorFlow........6
4 Description of programming code........7
4.1 System........7
4.2 Graph........7
4.3 UML-Diagram........7
4.4 Description of modules........7
4.4.1 Cifar10.py........7
4.4.2 Cifar10_eval.py........8
4.4.3 Cifar10_eval_single_directory.py........8
4.4.4 Cifar10_train.py........8
4.4.5 CreateTestBatch.py........9
5 Evaluation........10
5.1 Environment........10
5.2 Accuracy........10
6 Summary........11
7 References........12
8 Appendix........13

# 2 Introduction

- Define Deep Learning
  - Training data
  - Test data
- Define Neural Network
  - Define Neuron
    - Each neuron performs a dot product and optionally follows it with a non-linearity
  - Consist of different layers
    - Neurons connected via edges
    - Each edge has a weight
    - Weights are initialized with small random values
  - 
- Define Convolutional Neural Networks
  - Neural Networks
    - Convolutional Neural Networks are special part of that
    - CNN are feed forward networks
  - 
- Define Classification (unequal to clustering)

## 2.1 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[1], a host for competitions in machine learning. The aim is to predict if an image shows one of the two classes cat and dog. The classes are mutual exclusive. Because of the known output value, it is a supervised learning problem.

The training dataset consists of 25,000 images, 12,500 for each class.

- Number of images which are doubled from hand checking should be inserted here (can be found within the presentation)
- Varying size
- Not all images show only one cat nor dog
- Some images are only black

## 2.2 Neural Networks

## 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 1: *Structure of the implemented CNN* has multiple appearances of convolutional, pooling, and normalization layer. Within neural networks several techniques are used two perform the

**Comment [SS2]:** Fields should look like this.

[1] kaggle.com

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[2] version 3 and TensorFlow[3] 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.

**Comment [SS3]:** Update link

**Comment [SS4]:** Line break is bad

**Comment [SS5]:** Add link

**Comment [SS6]:** 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 uni-directional 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[4] (TensorFlow, n.d.).

**Comment [SS7]:** 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.

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

# 4 DESCRIPTION OF PROGRAMMING CODE

Code is available at gitlab

## 4.1 SYSTEM

- TensorFlow
  - Only CPU computation is used due to the lack of
  - During this project version 1.0 together with the Python version 3 language was used.

## 4.2 GRAPH

The design of the CNN used within this work is shown in Figure 1: *Structure of the implemented CNN*.

- Images are represented as 24x24x3
  - 3 for RGB
- Running of TensorFlow
  - 1. Create the graph
  - 2. Place the graph in a session and run it (it runs multiple times doing the same operation on different data)
- Tensorflow
  - Only CPU computation is used due to the lack of
  - During this project version 1.0 together with the Python version 3 language was used.
-

## 4.3 UML-DIAGRAM

## 4.4 DESCRIPTION OF MODULES

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 Kaggle dataset consisting of 25000 images into cifar10 compatible batch files.

The cifar10 modified Tensorflow code along with additional python files to convert the data set into cifar10 batch files is explained below:

### 4.4.1 Cifar10.py

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 Appendix.

#### 4.4.2 Cifar10_eval.py

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.

#### 4.4.3 Cifar10_eval_single_directory.py

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.

#### 4.4.4 Cifar10_train.py

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).

### 4.4.5 CreateTestBatch.py

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 20000 images for training and 5000 images for testing. Now to create our first binary batch file, we read the file name, write it’s 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 4000 images to create one batch of cifar10 input image. We do the same iteration for remaining batches.

  - CreateTestBatch.py
  - Helper.py
  - ImageInformation.py
  - PredictImageInDirectory.py
  - Resizer.py
  - Settings.py
- This module stores all global variables to paths, especially to the data passes

**Questions**

  - Do we use dropout? (Reducing the number of parameters)
  - We somehow should mention hyperparameters

# 5 Evaluation

## 5.1 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.

## 5.2 Accuracy

- If the learning rate is too high due to the ReLU activation function parts of the used network can be dead (never activated during training)

- The training data is not perfect

## 6 SUMMARY

From the 25000 images of cats vs dogs classification problem, we trained and tested our model in 4:1 ratio respectively. After 100,000 training steps we were able to achieve around 84% accuracy on the test batch.

Also, as a part of our project completion requirement, we were successfully able to predict 12500 test images provided on the Kaggle competition. Unfortunately, we were not able to verify these results online because of submission deadline mismatch. However, we cross checked many of the images manually and we will show some of the prediction values on our presentation slides.

# 7 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
- Stanford course for neural networks
    - Quelle: http://cs231n.github.io/convolutional-networks/
- 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

# 8 APPENDIX

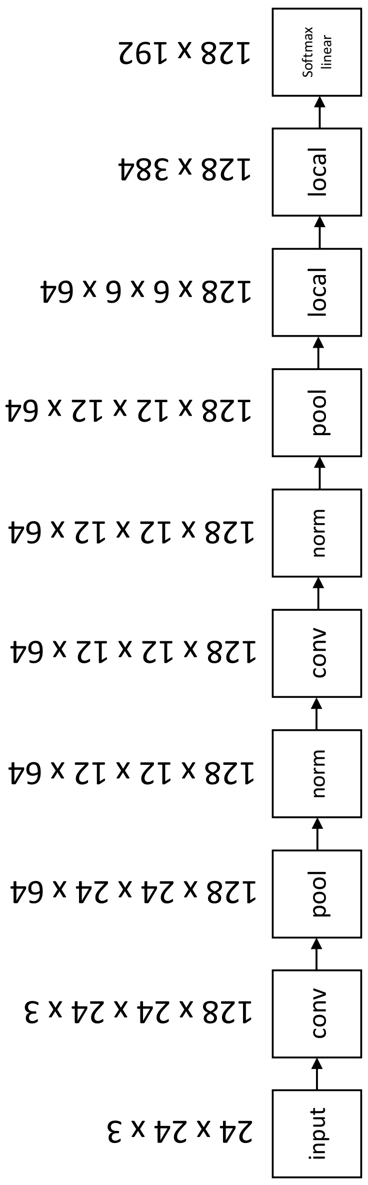

*Figure 1:* Structure of the implemented CNN