DATA CLASSIFICATION USING CONVOLUTIONAL NEURAL NETWORK

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Degree Programme in Automation Engineering

Valkeakoski, Winter 2019

Aleksandr Bukatoi

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Name of degree programme
Campus

Author	Aleksandr Bukatoi	**Year** 2019
Subject	Data classification using Convolutional Neural Network	
Supervisor(s)	Raine Lehto	

ABSTRACT

The author's aim in this paper was to understand how deep learning can be connected to automation engineering and which solution would be best suited for data classification.

Based on the research project concluded here, Convolutional Neural Network was chosen as the most suitable solution for the aforementioned task. The goal was to create a network, which would be able to classify images based on features extracted and memorized by the program. The Python programming language and the Anaconda environment were chosen as the most user-friendly environment to demonstrate how the neural network operates.

In order to write this thesis, the author studied a new programming language, examined deep learning and related topics, and was supervised by Raine Lehto throughout all the steps in the project.

At the end of this work the author has achieved the desired results in the form of program classifying images in two categories with a possibility of improving the system for future projects.

Keywords Convolutional Neural Network, deep learning, Python.

Pages 34 pages including appendices 3 pages

CONTENTS

LIST OF ABBREVIATIONS

AI	Artificial intelligence
ANN	Artificial neural network
ANSI	American national standards institute
CNN	Convolutional neural network
CNTK	Microsoft cognitive toolkit
CPU	Central processing unit
DL	Deep learning
GUI	Graphical user interface
GPU	Graphics processing unit
ISO	International organization for standardization
JIT	Just-in-time compilation
JVM	Java virtual machine
LLVM	The LLVM compiler infrastructure project
Matplotlib	A plotting library for the Python programming language
MIT	Massachusetts institute of technology
ML	Machine Learning
MSIL	Microsoft intermediate language
NumPy	The fundamental package for scientific computing with Python
PyPI	The Python package index
PyPy	An alternative implementation of the Python programming language to CPython
PyQt	One of the most popular Python bindings for the Qt cross-platform C++ framework
QtPy	A small abstraction layer that lets user write applications using a single api call to either PyQt or PySide
SciPy	A free and open-source Python library used for scientific computing and technical computing

1 INTRODUCTION

Since ancient times the humankind has been striving to lessen the everyday burden of people. The progress has made a great path from the invention of wheel to self-driving cars. As automation is rising, less tasks require human labour while opening new possibilities to engineers and developers.

Artificial intelligence has been a fascinating topic for science fiction writers and directors for decades, we are yet to observe its birth. The results of various experiments with AI are just simple imitations of how science perceives the work of the human brain. Some of the solutions are called Neural Networks. The field responsible for building Neural Networks is called deep learning.

The aim of this thesis work was to study the Convolutional Neural Network as a part of deep learning, to build a sample program to represent the possibilities of this branch of artificial intelligence.

2 DEEP LEARNING

2.1 What is deep learning?

Before diving into deep learning, it is important to understand its relationship with machine learning and artificial intelligence. Figure below describes how deep learning is connected to other AI research fields.

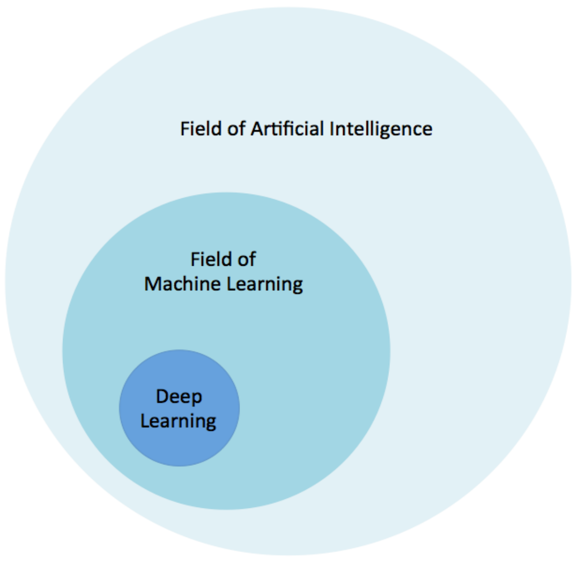


Figure 1. Relationship between deep learning, machine learning and artificial intelligence

Deep learning is a machine learning technique that teaches computers to do what comes naturally to us, humans: learn by example. Deep learning is a key technology behind driverless cars, enabling them to recognize a stop sign, or to distinguish a pedestrian from a lamppost. It is the key to voice control in consumer devices like phones, tablets, TVs, and hands-free speakers. Deep learning is getting lots of attention lately and for good reason. It's achieving results that were not possible before (What Is Deep Learning? October 15).

In Deep Learning, a computer model learns to perform classification tasks directly from images, text, or sound. Deep learning models can achieve state-of-the-art accuracy, sometimes exceeding human-level speed and efficiency. Models are trained by using multiple sets of labeled data and neural network structures that contain many hidden layers (What Is Deep Learning? October 15).

2.2 Difference between machine Learning and deep learning

Deep Learning is a form of Machine Learning. ML workflow starts with extracting manually relevant features from images. These features are then used to create a model that categorizes the objects in the image. Deep Learning, on the contrary, extracts these features automatically. In addition, Deep Learning performs "end-to-end learning". A neural network is given raw data and a task to perform, e.g. classification, and the system learns to do it automatically (What Is Deep Learning? October 15).

Another key difference is Deep Learning algorithms scale with data, whereas Shallow Learning converges. Shallow Learning is a method of Machine Learning that stops at a certain level of performance when user adds more examples and training data to the system (What Is Deep Learning? October 15).

A key advantage of deep learning networks is that they often continue to improve as the size of your data increases (What Is Deep Learning? October 15).

2.3 Brief history of deep learning

The history of Deep Learning starts in 1943, when a computer model based on the neural network was created by Walter Pitts and Warren McCulloch. The combination of algorithms and mathematics called "threshold logic" was used to mimic the thought process. Since that time, Deep Learning has been evolving steadily with two significant events in its development called Artificial Intelligence winters (Foote 2017).

Henry J. Kelley is famous for developing the basics of a continuous Back Propagation Model in 1960. A simpler version based only on the chain rule was developed by Stuart Dreyfus in 1962. However, the concept of back propagation existed in the early 1960s, it was inefficient and would not become beneficial until 1985 (Foote 2017).

The earliest efforts in developing Deep Learning algorithms provided Alexey Grigoryevich Ivakhnenko, who developed the Group Method of Data Handling, and Valentin Grigoryevich Lapa, author of Cybernetics and Forecasting Techniques, in 1965 (Foote 2017).

The first AI winter started during the 1970s, the result of promises that could not be kept. This resulted in lack of funding in both Artificial Intelligence and Deep Learning research. Fortunately, there were individuals who continued the research without funding (Foote 2017).

The first Convolutional Neural Networks were developed by Kunihiko Fukushima. Fukushima designed neural networks with multiple pooling and convolutional layers. He developed an Artificial Neural Network, called Neocognitron, which used a hierarchical, multilayered design in 1979. This design allowed the computer the “learn” to recognize visual patterns. The networks resembled modern versions, however, were trained with a reinforcement strategy of recurring activation in multiple layers, which gained strength over time. Additionally, Fukushima’s design allowed important features to be adjusted manually by increasing the “weight” of certain connections (Foote 2017).

The concepts of Neocognitron continued to be used. The use of top-down connections and new learning methods allowed a variety of neural networks to be realized. When more than one pattern is presented at the same time, the Selective Attention Model can separate and recognize individual patterns by shifting its attention from one to the other. A modern Neocognitron can identify patterns with missing information as well as complete the image by adding the missing information. This could be described as “inference” (Foote 2017).

The use of errors in training Deep Learning models called Back propagation evolved drastically in 1970. Seppo Linnainmaa wrote his master’s thesis, including a FORTRAN code for back propagation. Unfortunately, the concept was not applied to neural networks until 1985, when Rumelhart, Williams, and Hinton demonstrated back propagation in a neural network could provide “interesting” distribution representations. This discovery brought to light the question within cognitive psychology of whether human understanding relies on symbolic logic (computationalism) or distributed representations (connectionism). Yann LeCun provided the first practical demonstration of backpropagation at Bell Labs in 1989. He combined convolutional neural networks with back propagation, so the system could read “handwritten” digits. This system was eventually used to read the numbers of handwritten checks (Foote 2017).

The second Artificial Intelligence winter occurred in the second half of 1980s. Various overly-optimistic individuals had exaggerated the “immediate” potential of Artificial Intelligence, breaking expectations of investors. The impact of broken promises was so strong, the phrase Artificial Intelligence obtained pseudoscience status. The research still continued and significant advances were made. In 1995, Dana Cortes and Vladimir Vapnik developed the support vector machine (a system for mapping and recognizing similar data). LSTM (long short-term memory) for recurrent neural networks was developed in 1997, by Sepp Hochreiter and Juergen Schmidhuber (Foote 2017).

The next significant step for Deep Learning occurred in 1999, when computers started becoming faster at processing data and GPU (graphics processing units) were developed. Faster processing, with GPUs processing pictures, increased computational speeds by 1000 times over a decade. During this time, neural networks began to compete with support vector machines. While a neural network could be slow compared to a support vector machine, neural networks offered better results using the same data. Neural networks also have the advantage of continuing to improve as more training data is added (Foote 2017).

The Vanishing Gradient problem appeared in 2000. It was discovered that “features’ which formed in lower layers were not being learned by upper layers, because no learning signal reached these layers. This was not a fundamental problem for all neural networks, just the ones with gradient-based learning methods. The source of the problem turned out to be certain activation functions. A number of activation functions condensed their input, in turn reducing the output range in a somewhat chaotic fashion. This produced large areas of input mapped over an extremely small range. In these areas of input, a large change will be reduced to a small change in the output, resulting in a vanishing gradient. Two solutions used to solve this problem were layer-by-layer pre-training and the development of long short-term memory (Foote 2017).

In 2001, a research report by META Group (nowadays called Gartner) described the challenges and opportunities of data growth as three-dimensional. The report described the increasing volume of data and the increasing speed of data as increasing the range of data sources and types. This was a call to prepare for the onslaught of Big Data, which was just starting (Foote 2017).

The speed of GPUs had increased rapidly by 2011 making it possible to train Convolutional Neural Networks without the layer-by-layer pre-training. With increased computing speed, it became obvious that Deep

Learning had significant advantages in terms of speed and efficiency (Foote 2017).

Currently the processing of Big Data and the evolution of Artificial Intelligence are both dependent on Deep Learning, which is still developing and in need of new ideas (Foote 2017).

2.4 Methods of deep learning

First of all, Deep Learning consists of the following methods and their variations:

- Unsupervised learning systems such as Boltzman Machines for preliminary training, Auto-Encoders, Generative Adversarial Network.
- Supervised learning such as Convolution Neural Networks which brought technoogy of pattern recognition to a new level.
- Recurrent Neural Networks, allowing to train on processes in time.
- Recursive neural networks, allowing to include feedback between circuit elements and chains.

By combining these methods, complex systems are created that correspond to different tasks of Artificial Intelligence.

Deep training is an approved sample from a wide range of Machine Learning methods for data representations that are most appropriate to the nature of the task. The image, for example, can be represented in many ways, such as the intensity vector of the values per pixel, or (in a more abstract form) as a set of primitives, regions of a certain shape, etc. Successful data representations facilitate the solution of specific tasks - for example, face recognition and facial expressions. In systems of Deep Learning, the process of selecting and adjusting attributes automates itself, by training attributes without a teacher or with a partial involvement of the teacher, using effective algorithms and hierarchical extraction of characteristics

Deep Learning systems have found application in such areas as computer vision, speech recognition, natural language processing, audio recognition, bioinformatics, where for a number of tasks, much better results were demonstrated than previously.

3 CONVOLUTIONAL NEAURAL NETWORK

3.1 **Neuron**

An artificial neuron is a representation of how the functions of the human brain are recreated in computing. A neuron is essentially a unit which receives signals through multiple junctions named dendrites, also referred as synapses which can be seen on figure 2. These synapses serve as the inputs of the neuron. Every neuron has a certain threshold value, which if exceeded by the sum of the inputs the neuron shall send forward to another neuron and so on.

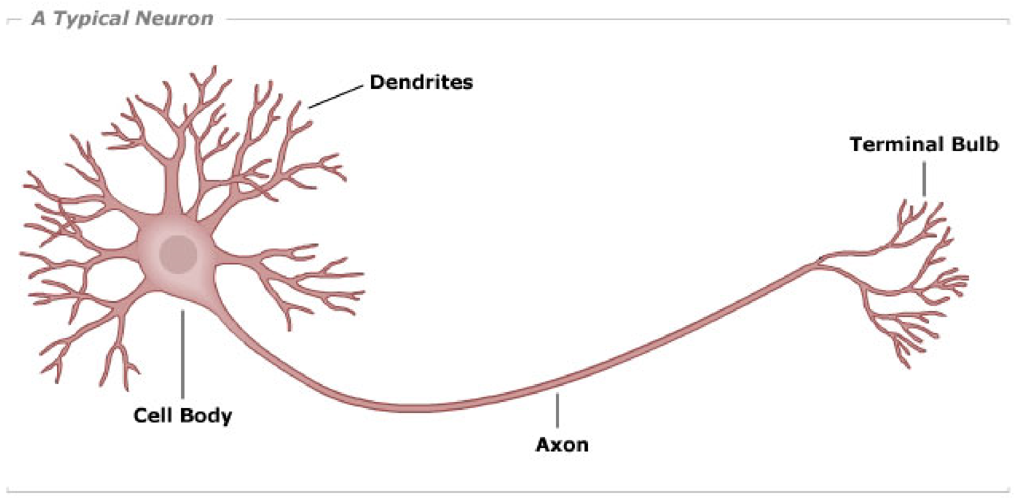


Figure 2. Neuron (Neurofantstic 2017)

3.1.1 An artificial neuron

An artificial neuron is a structural unit of an artificial neural network and is an analog of a biological neuron. Figure 3 demonstrates graphical representation of an artificial neuron.

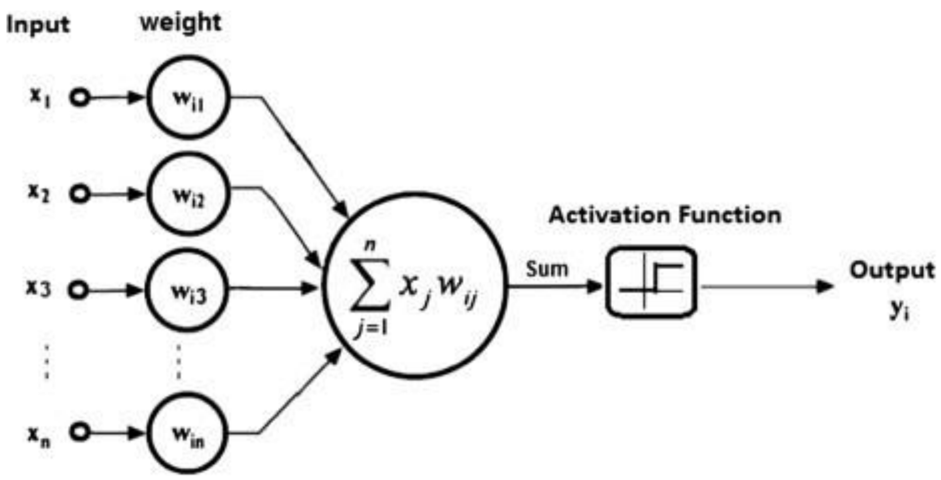


Figure 3. An artificial neuron (Saxena 2017)

From the mathematical point of view, the artificial neuron is the adder of all incoming signals, applying to the resulting weighted sum some simple, in general, nonlinear function that is continuous throughout the domain of definition. Usually, this function increases steadily. The result is sent to a single output (Saxena 2017).

Artificial neurons are combined in a certain way, forming an artificial neural network. Each neuron is characterized by its current state by analogy with the nerve cells of the brain, which can be excited or hindered. It has a group of synapses - unidirectional input links connected to the outputs of other neurons, and also has an axon - output connection of a given neuron, with which the signal enters the synapses of the following neurons (Saxena 2017).

Each synapse is characterized by the magnitude of the synaptic connection or its weight *w_i*, which is the equivalent of the electrical conductivity of biological neurons (Saxena 2017).

The current state of the neuron (1) is defined as the weighted sum of its inputs:

$$s = \sum_{i=1}^{n} x_i * w_i + w_0 \tag{1}$$

where w_0 is the displacement coefficient of the neuron (the weight of a single input (Saxena 2017).

The output of a neuron is a function of its state (2):

$$y = f(s) \tag{2}$$

Non-linear function f is called activation function and can be represented in different formulas (Saxena 2017).

3.2 **Activation function**

Although, the range of inputs may be different, the standard model of neuron is capable of giving only binary output values depending on whether the sum of its inputs exceeded or did not exceed the threshold value of the neuron. The activation function is used to is used to introduce non-linearity into the output of a neuron. A neural network without an activation function is simply just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks (Tiwari November 2).

3.3 Operating principles of neural networks

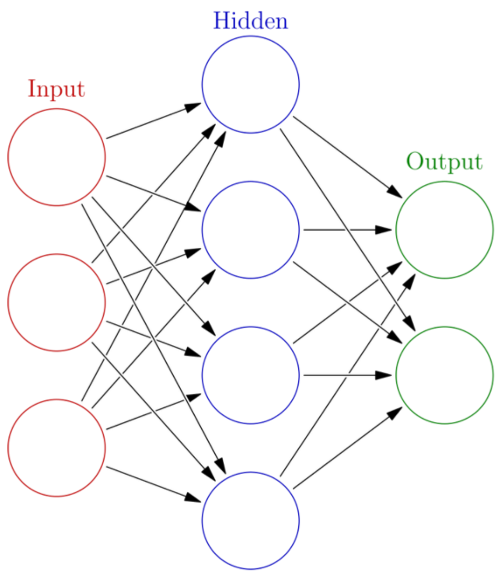


Figure 4. A neural network (Albright 2016)

The figure above shows a basic structure of an artificial neural network. Each of the circles is called a “node” and it represents a single neuron. On the left are input nodes, hidden nodes are located in the middle and output nodes are on the right (Albright 2016).

Input nodes accept information from input values, which, for example, could be a binary 1 or 0. The purpose of input nodes is to serve the data flowing into the network (Albright 2016).

Each input node is connected to a number of nodes in hidden layer (sometimes to every node, sometimes to a certain ones). Input nodes take the information they have received and pass it along to the hidden layer (Albright 2016).

Every connection, the equivalent of a neurons synapse, is given a certain weight, which allows the network to place a stronger emphasis on the action of specific node (Albright 2016).

Output layer nodes role is the same as hidden layer ones: output nodes sum the input from the hidden layer, and if they reach a required value, the output nodes fire and send specific signals. At the end of the process, the output layer will be transmitting a set of signals that represents the result of the input (Albright 2016).

The network shown above is simple, deep neural networks can have many hidden layers and hundreds and even thousands of nodes (Albright 2016).

3.4 Learning with neural networks

Comparison to a real-life analogy may be efficient in understanding the mechanisms of a neural network. Learning in a neural network is closely related to how people learn in their regular lives and activities – they perform an action and are either accepted or corrected by a teacher or coach to understand how to get better at a specific task. Similarly, neural networks require a trainer in order to describe what should have been produced as a response to the input. Based on the difference between the actual value and the value that was provided by the network, an error value is calculated and sent back through the system. The error value is analysed for each layer of the network and used to adjust the threshold and weights for the next input. This way, the error keeps becoming indirectly lesser each run as the network learns how to analyse values (Futurism 2015).

The procedure described above is known as backpropagation and is applied continuously through a network until the error value reaches a satisfactory value. At this point, the neural network no longer requires such training process and is allowed to run without adjustments. The network may then finally be applied, using the adjusted weights and thresholds as guidelines (Futurism 2015).

3.5 Usage of a neural network while running

When a neural network is active, no backpropagation takes place as there is no way to directly verify the expected response. Instead, the validity of output statements is corrected during a new training session or are left as is for the network to run. Many adjustments may need to be made as the network consists of a great number of variables that must be precise for the artificial neural network to function correctly (Futurism 2015).

A simple example of such a process can be researched by teaching a neural network to convert text to speech. User could pick multiple different articles and paragraphs and use them as inputs for the network and specify a desired input before executing the test. The training phase would then consist of input data going through the multiple layers of the network and using backpropagation to adjust the parameters and threshold value of the network in order to minimize the error value for all input examples. The network may then be tested on new articles to determine if it could correctly convert text to proper speech (Futurism 2015).

Networks like these may be used as models for a great number of mathematical and statistical problems, including but not limited to speech synthesis and recognition, face recognition and prediction, nonlinear system modelling and pattern classification (Futurism 2015).

3.6 Convolutional operation

Convolution is one of the most important operations in signal and image processing. It could operate in 1D, 2D or 3D depending on what kind of information is processed. This thesis is concentrated on convolution in 2D spatial which is mostly used in image processing for feature extraction and is also the core block of Convolutional Neural Networks. Generally, an image can be considered as a matrix whose elements are numbers between 0 and 255. The size of image matrix (3) is:

$$image\ height * image\ width * number\ of\ image\ channels \tag{3}$$

A grayscale image has 1 channel, where a colour image has 3 channels.

Each convolution operation has a kernel which could be any matrix smaller than the original image in height and width as illustrated in figure 5. Each kernel is useful for a specific task, such as sharpening, blurring, edge detection, and etc.

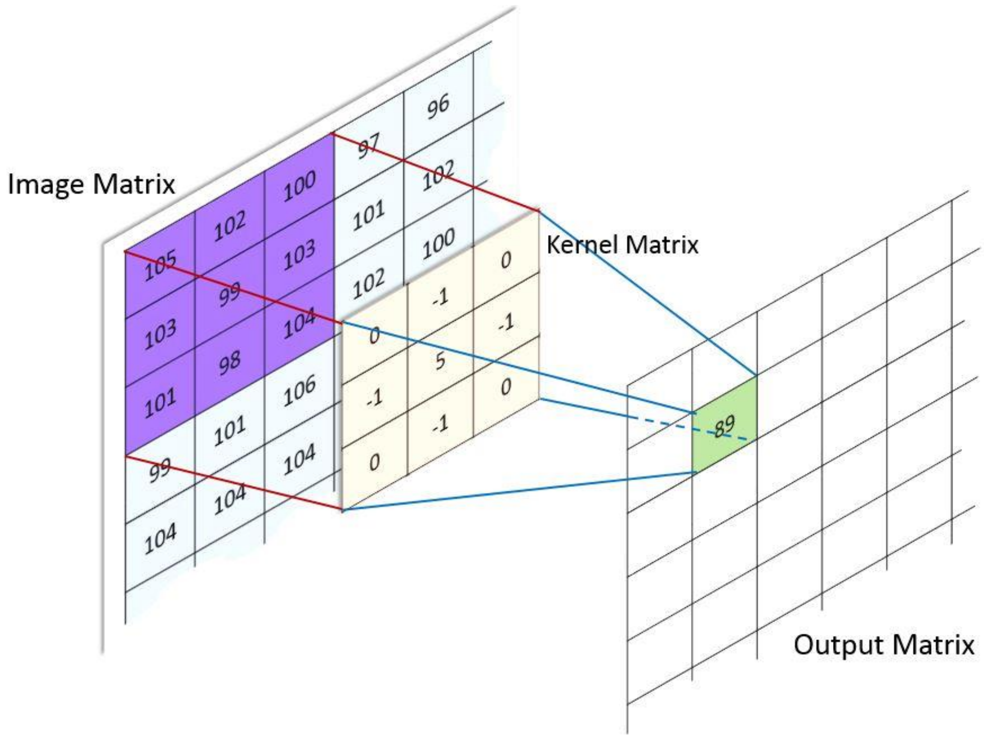


Figure 5. Image filtering

To calculate the convolution, the kernel is swept on the image and at every single location the output is calculated. The following equations are used to calculate the exact size of convolution output for an input with the size of (height = H, width = W) and a filter with the size of (height = F_h, width = F_w):

$$Output\ height = \frac{H - F_h + 2P}{S_h} + 1 \tag{4}$$

$$Output\ width = \frac{W - F_w + 2P}{S_w} + 1 \tag{5}$$

where S_h and S_w are vertical and horizontal stride of the convolution and P is the amount of zero-padding added to the border of the image (Karpathy 2016).

3.6.1 Convolutional layer

A convolution layer is used to perform a convolution operation in neural networks. Its parameters consist of a set of learnable filters. Each filter is small spatially but extends through the full depth of the input volume. For example, one of the filters applied on convolutional layer might have size $5 * 5 * 3$ (5 pixels width and height, 3 colour channels). During the process, each filter is convolved across width and height of the input volume and dot products are computed between the entries of the filter and input at any position. Generally, a 2-dimensional activation map is created allowing to give responses to the network when similar features are detected. Each filter produces a separate 2D activation map. These maps are stacked along the depth dimension and produce the output volume (Karpathy 2016).

When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, each neuron is connected to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume (Karpathy 2016).

3.6.2 Spatial arrangement in convolutional layer

There are three hyperparameters controlling the size of the output volume: the depth, stride and zero-padding (Karpathy 2016).

Depth of the output corresponds to the number of filters the user would like to implement, each learning to search for something different in the input (Karpathy 2016).

Stride is a parameter that specifies how filter slides on the image. When the stride is 1 the filters move one pixel at a time. Increasing stride allows to produce smaller output volumes spatially (Karpathy 2016).

Zero-padding is a parameter that adds zeros around the border of input. This allows to control the spatial size of the output (Karpathy 2016).

3.6.3 Parameter sharing

Parameter sharing in Convolutional neural networks is used to adjust number of parameters (Karpathy 2016).

The user can reduce the number of parameters by making an assumption that if one feature can compute at some spatial position (x, y), then it is useful to compute a different place (x_2, y_2). In other words, denoting a single 2D slice of depth as a depth slice. For example, during backpropagation, every neuron in the network will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice (Karpathy 2016).

If all neurons in a single depth slice are using the same weight vector, then the forward pass of the convolutional layer can be computed in each depth slice as a convolution of the neuron's weights with the input volume (Therefore the name: Convolutional layer). This is the reason why it is common to refer to the sets of weights as a filter (or a kernel), that is convolved with the input (Karpathy 2016).

3.7 **ReLU layer**

The Rectified Linear Unit (6) (also Rectifier) is used to increase non-linearity in images. It computes the function:

$$f(x) = \max(0, x) \tag{6}$$

In other words, the activation is thresholded at zero point as seen on figure 6.

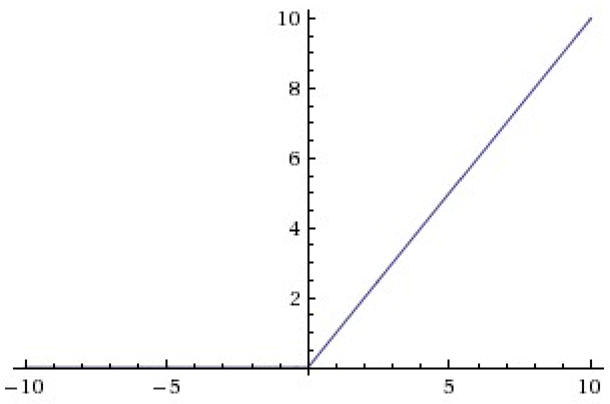


Figure 6. ReLU function (Anukarsh Singh 2017)

There are several pros and cons to using ReLUs:

- Greatly increases the convergence of stochastic gradient descent compared to sigmoid/tanh functions.

- Compared to sigmoid/tanh neurons that execute expensive operations, the ReLU can be used by thresholding a matrix of activation at zero point.
- Unfortunately, ReLU units are fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on (Karpathy 2016).

3.7.1 Leaky ReLU

Leaky ReLU (7) is one attempt to solve the “dying ReLU” problem. Instead of the function being zero when $x < 0$, a leaky ReLU has a negative slope.

$$f(x) = 1(x < 0)(ax) + 1(x \geq 0)(x) \tag{7}$$

where a is a small constant (Karpathy 2016).

3.7.2 Maxout

Maxout is another solution to the “dying ReLU”. The Maxout neuron (8) computes the function:

$$\max(w_1^T x + b_1 w_2^T x + b_2) \tag{8}$$

Both ReLU and leaky ReLU are a special case of this form. The Maxout neuron has all the advantages of a ReLU and has no drawbacks. However, the Maxout doubles the number of parameters for every single neuron (Karpathy, 2016).

3.8 **Pooling**

It is common to periodically insert a Pooling layer in-between successive convolutional layers in a neural network architecture. Its function is to progressively reduce the spatial size of the representation, to reduce the number of parameters and computation in the network, and therefore to also control overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most used form is a pooling layer with filters of size 2*2 applied with a stride of 2. It down-samples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. The depth dimension in pooling operation remains unchanged (Karpathy 2016).

3.8.1 General pooling

In addition to max pooling, the pooling layers can also perform other functions. Average pooling was often used historically but has recently fallen out of favour compared to the max pooling operation, which has been shown to be more efficient (Karpathy, 2016).

3.9 **Flattering**

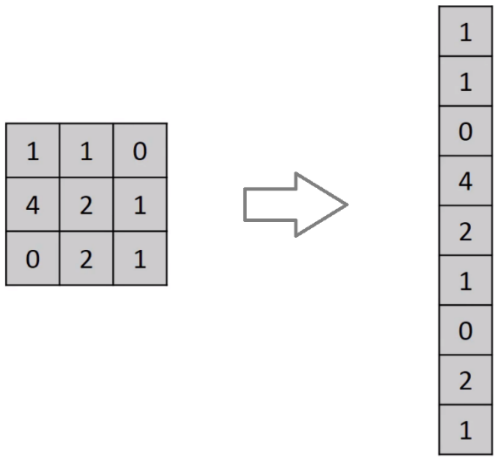


Figure 7. Flattening

Flattening layer is a simple layer that is used to prepare data to be the input of the final and most important layer – Fully-Connected Layer. Generally, neural networks receive data in one dimension in a form of an array of values, this layer uses data received from pooling layer or convolutional layer and squashed the matrixes into arrays as illustrated in figure 7. Obtained values are used as an input to the neural network (Karpathy, 2016).

3.10 **Full-connection**

After numerous convolution and pooling layers are used in neural network Fully connected layer is used in order to access all activation functions in previous layer (Karpathy 2016).

3.10.1 Converting fully connected layer into a convolutional layer

The only difference between fully connected and convolutional layer is that the neurons in latter are connected only to a local area of the input, and that neurons in convolutional layer share parameters. However, the neurons in both volumes calculate dot products, so their function is exactly alike. Thus, conversion between layers is possible:

- For every convolutional layer there is a fully connected one, that uses the same forward function. The weight is a large matrix which

is usually zero except for a specific blocks due to a local connectivity, where the weights in the blocks are equal because of parameter sharing.
- Fully connected layer can be converted to a convolutional simply using a filter, which size is set to be the same as the input (Karpathy, 2016).

3.11 **Softmax & Cross-Entropy**

To convert the output of convolutional neural network into probability – softmax function is used. Cross-Entropy serves the purpose of measuring loss and optimization.

3.11.1 Softmax

Softmax function (9) is used to transform an N-dimensional vector of real numbers into a vector of real number in range (0,1) which adds up to 1 (Dahal November 15).

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a} \tag{9}$$

As can be seen from the name, softmax function is another interpretation of max function. Instead of selecting one maximum value, it breaks the whole (1) with maximal element getting the largest portion of the distribution with other smaller elements getting some of it as well (Dahal November 15).

This property of softmax function that it outputs a probability distribution makes it suitable for probabilistic interpretation in classification tasks (Dahal November 15).

In python is necessary to remember that the numerical range of floating-point numbers in numpy is limited. For float64 the upper bound is10^{308}. For exponential function it is not difficult to overcome that limit (Dahal November 15).

To make softmax function numerically stable, the values in the vector are simply normalized by multiplying the numerator and denominator with a constant C (Dahal November 15).

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} = \frac{C * e^{a_i}}{C * \sum_{k=1}^{N} e^{a_k}} = \frac{e^{a_i + \log(C)}}{\sum_{k=1}^{N} e^{a_k + \log(C)}} \tag{10}$$

User can choose an arbitrary value for $\log(C)$ term, but usually $\log(C) = -\max(\alpha)$ is chosen, as it shifts all elements in the vector from negative to zero, and negatives with large exponents saturate to zero rather than the infinity, avoiding overshooting (Dahal November 15).

3.11.2 Derivative of softmax

Due to the specific property of softmax function to output a probability distribution, it used as a final layer in neural networks. For this derivative or gradient is calculated and passed back to previous layer during backpropagation (Dahal November 15).

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} \tag{11}$$

From quotient rule we know that for $f(x) = \frac{g(x)}{h(x)}$ we have $f(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$.

In this situation $g(x) = e^{a}$and $h(x) = \sum_{k=1}^{N} e^{a_k}$.In $h(x)$, $\frac{\partial}{\partial e^{a_j}}$ will always be e^{a_j}., But it is important to note that in $g(x)$ $\frac{\partial}{\partial e^{a}{}_{j}}$will be e^{a_j} only if $i = j$ otherwise is 0 (Dahal November 15).

If $i = j$

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \\ \frac{e^{a_i} \sum_{k=1}^{N} e^{a_k} - e^{a_j} e^{a_i}}{(\sum_{k=1}^{N} e^{a_k})^2} = \\ \frac{e^{a_i} (\sum_{k=1}^{N} e^{a_k} - e^{a_j})}{(\sum_{k=1}^{N} e^{a_k})^2} = \\ \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} * \frac{(\sum_{k=1}^{N} e^{a_k} - e^{a_j})}{\sum_{k=1}^{N} e^{a_k}} = \\ p_i(1 - p_j) \tag{12}$$

For $i \neq j$

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \\ \frac{0 - e^{a_j} e^{a_i}}{(\sum_{k=1}^{N} e^{a_k})^2} = \\ \frac{-e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} * \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} = \tag{13}$$

$$-p_j p_i$$

So the derivative of the softmax (14) function is given as,

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1-p_j)\ if\ i \neq j \\ -p_j p_i\ if\ i \neq j \end{cases} \tag{14}$$

Or using Kronecker delta $\partial ij = \begin{cases} 1\ if\ i = 1 \\ 0\ if\ i \neq j \end{cases}$

$$\frac{\partial p_i}{\partial a_j} = p_i(\partial_{ij} - p_j) \tag{15}$$

3.11.3 Cross-entropy loss

Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is. It is defined as, $H(y,p) = -\sum_i y_u \log(p_i)$. Cross-entropy measure is a widely used alternative of squared error. It is used when node activations can be understood as representing the probability that each hypothesis might be true, i.e. when the output is a probability distribution. Thus, it is used as a loss function in neural networks which have softmax activations in the output layer (Dahal November 15).

3.11.4 Derivative of cross-entropy loss with softmax

Cross-entropy loss with softmax function are used as the output layer extensively. Now derivative of softmax (14) that was derived earlier is used to derive the derivative of the cross-entropy loss function (16)(17)(18) (Dahal November 15).

$$L = \sum_i y_i \log(p_i) \tag{16}$$

$$\frac{\partial L}{\partial o_i} = \\ \sum_k y_k \frac{\partial \log(p_i)}{\partial o_i} = \\ -\sum_k y_k \frac{\partial \log(p_i)}{\partial p_k} * \frac{\partial p_k}{o_i} = \\ -\sum y_k \frac{1}{p_k} * \frac{\partial p_k}{\partial o_i} \tag{17}$$

From derivative of softmax (14) we derived earlier,

$$\frac{\partial L}{\partial o_i} = \tag{18}$$
$$-y_i(1-p_i) - \sum_{k\neq i} y_k \frac{1}{p_k}(-p_k p_i) =$$
$$-y_i(1-p_i) + \sum_{k\neq 1} y_k\, p_i =$$
$$-y_i + y_i p_i + \sum_{k\neq 1} y_k\, p_i =$$
$$p_i\left(y_i + \sum_{k\neq 1} y_k\right) - y_i$$

y is one but encoded vector for the labels, so $\sum_k y_k$=1 and $y_i + \sum_{k\neq 1} y_k$=1. So the formula is,

$$\frac{\partial L}{\partial o_i} = p_i - y_i \tag{19}$$

which is a very simple expression (Dahal November 15).

4 CONSTRUCTING A CONVOLUTIONAL NEURAL NETWORK IN PYTHON

4.1 Overview of the language

Python is a high-level general-purpose programming language designed to improve developer productivity and code readability. The syntax of the Python kernel is minimal. At the same time, the standard library includes a large amount of useful functions (Python November 22).

Python supports several programming paradigms, including structural, object-oriented, functional, imperative, and aspect-oriented. The main architectural features are dynamic typing, automatic memory management, complete introspection, exception handling mechanism, support for multi-threaded computing and convenient high-level data structures. The code in Python is organized into functions and classes that can be combined into modules (they in turn can be combined into packages) (Python November 22).

The reference implementation of Python is the CPython interpreter, which supports the most actively used platforms. It is distributed under the free license of the Python Software Foundation License, which allows to use it without restrictions in any applications, including proprietary ones. There are implementations of interpreters for JVM (with the ability to compile), MSIL (with the ability to compile), LLVM and others. The PyPy project offers a Python implementation using JIT compilation, which greatly increases the speed of execution of Python programs (Python November 22).

Python is an actively developing programming language, new versions (with the addition / modification of language properties) are published approximately every two and a half years. Because of this and some other reasons, Python does not have ANSI, ISO or other official standards, and CPython does (Python November 22).

4.2 Anaconda environment

Anaconda distribution comes with more than 1,000 data packages as well as the conda package and virtual environment manager, called Anaconda Navigator, so it eliminates the need to learn to install each library independently (Conda November 22).

The open source data packages can be individually installed from the Anaconda repository with the conda install command or using the pip install command that is installed with Anaconda. Pip packages provide many of the features of conda packages and in most cases they can work together (Conda November 22).

You can also make your own custom packages using the conda build command, and you can share them with others by uploading them to Anaconda Cloud, PyPI or other repositories (Conda November 22).

The default installation of Anaconda2 includes Python 2.7 and Anaconda3 includes Python 3.6. However, you can create new environments that include any version of Python packaged with conda (Conda November 22).

4.3 Spyder

Spyder is an open source cross-platform integrated development environment (IDE) for scientific programming in the Python language. Spyder integrates with a number of prominent packages in the scientific Python stack, including NumPy, SciPy, Matplotlib, pandas, IPython, SymPy and Cython, as well as other open source software. It is released under the MIT license (Spyder November 22).

Initially created and developed by Pierre Raybaut in 2009, since 2012 Spyder has been maintained and continuously improved by a team of scientific Python developers and the community (Spyder November 22).

Spyder is extensible with first- and third-party plugins, includes support for interactive tools for data inspection and embeds Python-specific code quality assurance and introspection instruments, such as Pyflakes, Pylint and Rope. It is available cross-platform through Anaconda, on Windows with WinPython and Python (x,y), on macOS through MacPorts, and on major Linux distributions such as Arch Linux, Debian, Fedora, Gentoo Linux, openSUSE and Ubuntu (Spyder November 22).

Spyder uses Qt for its GUI and is designed to use either of the PyQt or PySide Python bindings. QtPy, a thin abstraction layer developed by the Spyder project and later adopted by multiple other packages, provides the flexibility to use either backend (Spyder November 22).

4.4 Keras

Keras is an open neural network library written in Python. It is an add-on for the Deeplearning4j, TensorFlow and Theano frameworks. It is aimed at operational work with deep learning networks, while being designed to be compact, modular and expandable. It was created as part of the research effort of the ONEIROS project, and its main author and sponsor is François Chollet, a Google engineer (Keras November 22).

It was planned that Google will support Keras in the main TensorFlow library, however Scholl selected Keras as a separate add-on, as according to the concept Keras is more of an interface than a through machine learning system. Keras provides a higher-level, more intuitive set of abstractions that makes it simple to build neural networks, regardless of the library of scientific computing used at the bottom level. Microsoft is working on adding to the Keras and lower-level CNTK libraries (Keras November 22).

This library contains numerous implementations of widely used building blocks of neural networks, such as layers, target and transfer functions, optimizers, and many tools for simplifying the work with images and text. Her code is hosted on GitHub, and the support forums include the GitHub FAQ page, the Gitter channel and the Slack channel (Keras November 22).

4.5 Creating working environment

Creating a custom environment is an important step to make convolutional neural network work. It is required since Keras library is not officially supported by Anaconda. But the latter allows to manage custom environments with different settings.

Anaconda prompt or Windows Terminal is used for the following steps:

1. To create custom environment with a specific version of Python:

```
conda create --name myenv python=3.5
```

myenv is replaced with the environment name. Author uses tensorflow.

2. Activate environment virtual environment:

```
activate tensorflow
```

The prompt will be flanked by the name of the environment.

Earlier, the Anaconda installer automatically created a conda environment called root that houses the core libraries for data science. Since a different environment is used, those libraries cannot be accessed unless they are re-installed in the new environment. Fortunately, conda allows users to install packages that cover everything users need.

3. To install Spyder:

```
conda install spyder
```

Spyder can be used now.

Finally, Tensorflow and Keras libraries can be installed. Neither library is officially available via a conda package so they need to be installed with pip.

4. To install Tensorflow and Keras:

```
pip install --upgrade tensorflow
pip install --upgrade keras
```

Now all required libraries are installed.

5. To open Spyder:

```
spyder
```

4.6 Removing conda environment

In case user missed a step or made a mistake, conda environment can be removed:

```
conda remove --name tensorflow --all
```

4.7 Tensorflow and Keras libraries with GPU support

By deefault Tensorflow and Keras use CPU to work, but GPU can be used instead increasing neural network speed and performance significantly.

4.7.1 GPU+ Machine

TensorFlow relies on a technology called CUDA which is developed by NVIDIA. The GPU+ machine includes a CUDA enabled GPU and is a great fit for TensorFlow and Machine and Deep Learning in general.

4.7.2 CUDA

CUDA (Compute Unified Device Architecture) is a software-hardware architecture of parallel computing that allows you to significantly increase computational performance through the use of Nvidia graphics processors (CUDA November 22).

CUDA SDK allows programmers to implement on a special simplified dialect of the C programming language algorithms that can be executed on Nvidia graphics processors and include special functions in the text of a C program. The CUDA architecture allows the developer, at his discretion, to organize access to the set of instructions of the graphics accelerator and manage its memory (CUDA November 22).

Recommended version: Cuda Toolkit 8.0 (CUDA November 22).

4.7.3 cuDDN

The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. cuDNN is part of the NVIDIA Deep Learning SDK (cuDNN November 22).

Recommended version: cuDDN 5.1

On Windows, cuDNN is distributed as a zip archive. Extract it and add the Windows path. For example `C:\tools\cuda\bin` and run:
`set PATH=%PATH%;C:\tools\cuda\bin` (cuDNN November 22).

4.7.4 Instal Tensorflow with GPU support

1. To create custom environment with a specific version of Python:
```
conda create --name myenv-gpu python=3.5
```
„myenv“ is replaced with the environment name. Author uses tensorflow.

2. Activate environment virtual environment:
```
activate tensorflow
```
The prompt will be flanked by the name of the environment.

3. Install Tensorflow and Keras:
```
pip install -- tensorflow-gpu
pip install -- keras
```

4.8 **Creating code**

4.8.1 Prerequisites

To train convolutional neural network user has to prepare labelled images in advance. For example, author is using 5000 images of dogs and 5000 images of cats in total for binary classification. 8000 images are used for training the CNN while other 2000 are need for testing it. Training sets can be found and downloaded on GitHub and similar portals.

4.8.2 Setting path to training set folder

After initializing Spyder software it is important to remember to set a console working directory so that future neural network could access necessary data to train itself. To set directory in Spyder user has to find the required folder in file explorer. For example:
C:\Users\Aleksandr\Desktop\Thesis\CNN\dataset
After the working directory is found:
Options -> Set console working directory.

4.8.3 Importing the Keras libraries and packages

The first step in building code is to import all required packages beforehand. Following packages were used to create convolutional neural network:

```
from keras.models import Sequential

from keras.layers import Conv2D

from keras.layers import MaxPooling2D

from keras.layers import Flatten

from keras.layers import Dense
```

A sequential library is used to specify what input shape model should expect. This model is based on layers that go in sequence.

The Conv2D or 2D convolution library is needed since this specific program is doing image classification. 1D convolution package is used for sound signal whereas 3D is required for video.

A maxPooling2D is necessary for pooling operation for convolved images.

A flatten library is responsible for preparing data to be the input for fully connected layer.

A dense library allows to create additional layers such as fully-connected layer in neural network.

4.8.4 Core of Convolutional Neural Network

```
# Initialising the CNN

classifier = Sequential()


# Step 1 - Convolution

classifier.add(Conv2D(32,  (3, 3), input_shape = (64, 64, 3), activation = 'relu'))


# Step 2 - Pooling

classifier.add(MaxPooling2D(pool_size = (2, 2)))



# Step 4 - Full connection

classifier.add(Dense(units = 128, activation = 'relu'))

classifier.add(Dense(units = 1, activation = 'sigmoid'))


# Compiling the CNN

classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy',  metrics = ['accuracy'])
```

The code above is the neural network itself. First objective is to create an object of the sequential class. Since neural network is going to classify images, the name of the object is classifier.

The first layer in CNN is a convolutional layer. “classifier.add” is a method to implement a specific layer into network. 2D convolutional layer requires certain arguments to be specified: number of filters or feature maps, number of rows and columns of kernel. Number “32” is the most common practice for creating first layer in most CNN projects, but it doesn’t necessarily stop there. The higher number of filters allows to get higher accuracy in predictions. Common numbers of filters are: 32, 64, 128 and etc. Input shape is an argument related to how the system receives the image format. First arguments in input shape are width and height dimensions, the third one is number of color channels. For colored pictures it is 3. The last parameter for convolutional layer is activation function. To prevent negative pixel values ReLU function is used. With this convolutional layer is ready to be used.

The next layer in the sequence is maxpooling layer. It reduces the size of feature maps created after convolution operation leaving only important features for system to detect. Pool size is 2 by 2 in this model.

The third layer is flattening. It converts pooled feature maps from 2 dimensional arrays into 1 dimensional preparing it to be used by fully-connected layer.

As was mentioned above dense functions are used to add hidden layers. In this case fully-connected layer is added. Parameter “unit” is representing the number of nodes in the layer. There are no rules on what number of nodes should be used, but in general practice number of 128 is implemented. For this layer to be activated ReLU function is required.

The last layer in CNN is output layer. Number of output nodes is 1 which is the predicted probability of one class. Sigmoid activation function is required since the outcome in predictions is binary. For multiple outcomes Softmax activation is needed.

When the core of CNN is created the next is to compile it. Compile method uses following parameters: optimizer for stochastic descent algorithm, loss function and metrics parameter to choose the performance metric. In this model ‘adam” algorithm is used. “Binary cross-entropy” function is implemented for binary outcome classifier. The last argument is metric. Accuracy is what required for this CNN.

4.8.5 Fitting CNN into images

After the code is compiled the next task is to fit CNN into images. The following code can be found on Keras documentation website in the preprocessing section.

```
# Part 2 - Fitting the CNN to the images


from keras.preprocessing.image  import ImageDataGenerator


train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
```

```
test_datagen = ImageDataGenerator(rescale = 1./255)


training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                  target_size = (64, 64),
                                                  batch_size = 32,
                                                  class_mode = 'binary')


test_set = test_datagen.flow_from_directory('dataset/test_set',
                                             target_size = (64, 64),
                                             batch_size = 32,
                                             class_mode = 'binary')


classifier.fit_generator(training_set,


                          steps_per_epoch = 8000,
                          epochs = 25,
                          validation_data = test_set,
                          validation_steps = 2000)
```

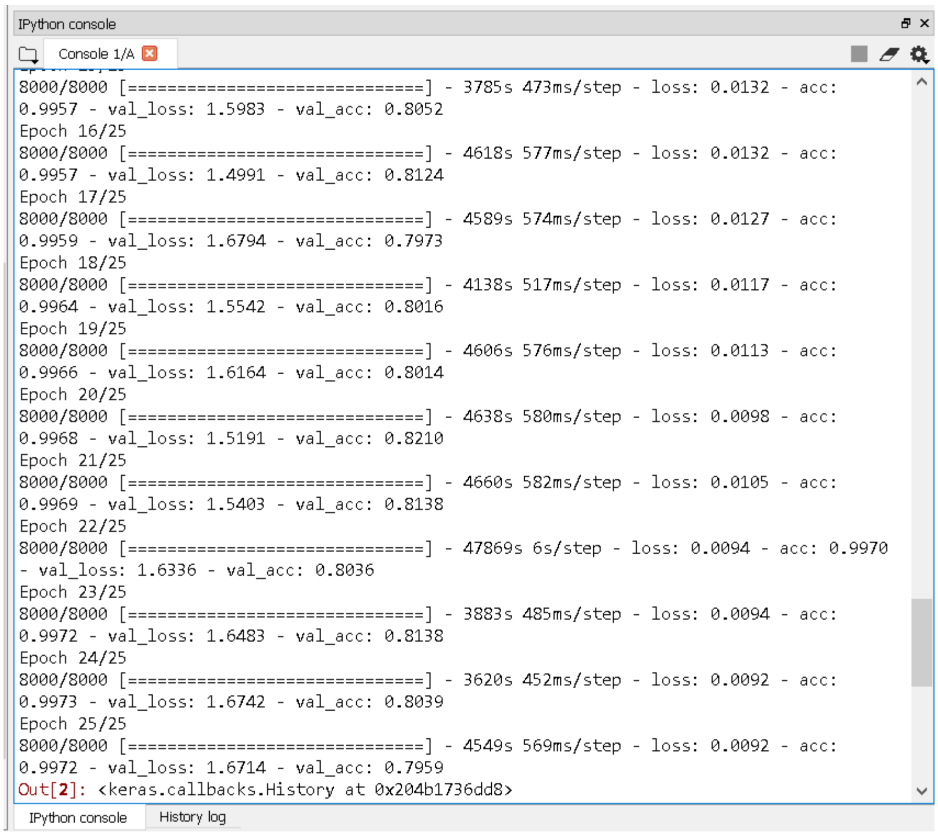
Deep networks need a large amount of training data to achieve a good performance. To build an efficient image classifier using very little training data, image augmentation is required to raise the performance of deep networks. Image augmentation artificially creates training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc. Thus, the first section of code is adding image generator in future CNN. Since pixels scale between values of 1 and 255, rescaling is required. Using 1/255 scale allows pixels to achieve values between 0 and 1. Same scale is used for test data generator. Default arguments for shear and zoom range as well as horizontal flip are used.

After importing an image generator and setting its arguments data directories are needed to be specified. As was mentioned before images are divided in training and test sets. Both “training_set” and “test_set” require the same scaling as has been specified in core section so that correct images are seen by the system.

The last code section is responsible for fitting the images into CNN as well as testing its performance on the test set. First arguments is the number of images that are expected to be used and how many times the system has to run through these images while "learning". The last arguments are for testing the efficiency of training. Number of images for test is to be specified.

4.8.6 Performance results

At this point the user can receive the first results. He needs to run the code and wait. The training process can take from several minutes to several hours depending on PC's capabilities and whether neural network is supported by CPU or GPU. The author's first results came with a 75% accuracy in 15 hours of training.



```
8000/8000 [==============================] - 3785s 473ms/step - loss: 0.0132 - acc:
0.9957 - val_loss: 1.5983 - val_acc: 0.8052
Epoch 16/25
8000/8000 [==============================] - 4618s 577ms/step - loss: 0.0132 - acc:
0.9957 - val_loss: 1.4991 - val_acc: 0.8124
Epoch 17/25
8000/8000 [==============================] - 4589s 574ms/step - loss: 0.0127 - acc:
0.9959 - val_loss: 1.6794 - val_acc: 0.7973
Epoch 18/25
8000/8000 [==============================] - 4138s 517ms/step - loss: 0.0117 - acc:
0.9964 - val_loss: 1.5542 - val_acc: 0.8016
Epoch 19/25
8000/8000 [==============================] - 4606s 576ms/step - loss: 0.0113 - acc:
0.9966 - val_loss: 1.6164 - val_acc: 0.8014
Epoch 20/25
8000/8000 [==============================] - 4638s 580ms/step - loss: 0.0098 - acc:
0.9968 - val_loss: 1.5191 - val_acc: 0.8210
Epoch 21/25
8000/8000 [==============================] - 4660s 582ms/step - loss: 0.0105 - acc:
0.9969 - val_loss: 1.5403 - val_acc: 0.8138
Epoch 22/25
8000/8000 [==============================] - 47869s 6s/step - loss: 0.0094 - acc: 0.9970
- val_loss: 1.6336 - val_acc: 0.8036
Epoch 23/25
8000/8000 [==============================] - 3883s 485ms/step - loss: 0.0094 - acc:
0.9972 - val_loss: 1.6483 - val_acc: 0.8138
Epoch 24/25
8000/8000 [==============================] - 3620s 452ms/step - loss: 0.0092 - acc:
0.9973 - val_loss: 1.6742 - val_acc: 0.8039
Epoch 25/25
8000/8000 [==============================] - 4549s 569ms/step - loss: 0.0092 - acc:
0.9972 - val_loss: 1.6714 - val_acc: 0.7959
Out[2]: <keras.callbacks.History at 0x204b1736dd8>
```

Figure 8. Training results

When the program starts working, the user will be able to see information on the training process such as the estimated time to finish an epoch, the loss that represents errors during training. The most important argument is validation accuracy. It presents how efficiently the program recognizes images that were not presented in the training set.

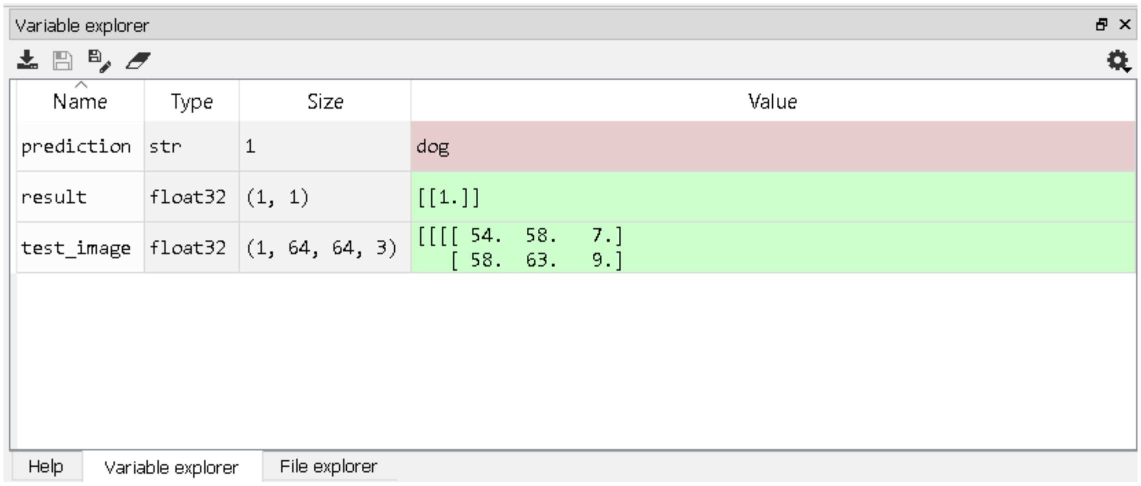
4.8.7 New predictions

```
import numpy as np

from keras.preprocessing import image

test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg', target_size = (64, 64))

test_image = image.img_to_array(test_image)

test_image = np.expand_dims(test_image, axis = 0)

result = classifier.predict(test_image)


training_set.class_indices

if result[0][0] == 1:

   prediction = 'dog'

else:

   prediction = 'cat'
```

Finally, the last part of the code is responsible for providing results based on images that are not used during training process. First part is used to set a path to an image that is going to be tested by CNN. Following next are commands to transform test image into 3D array and adding to that array another dimension which represents result for test image, which can be either 0 or 1 in binary classification.

To specify which value belongs to which class additional lines of code are added. User himself can determine the names of result variables. For example, cats equal 0 and dogs equal 1.

When the code is finished, and neural network is trained, user has to specify a path to image he wants to test and run the last part of code again. The result will be shown in variable explorer.



Name	Type	Size	Value
prediction	str	1	dog
result	float32	(1, 1)	[[1.]]
test_image	float32	(1, 64, 64, 3)	[[[[54. 58. 7.] [58. 63. 9.]

Figure 9. Variable explorer

4.9 Tuning CNN

Performance of neural network can be improved by several means:

- Adding new convolutional layers in sequence. This way the CNN will be creating more feature maps and thus receiving more data.
- Adding another fully-connected layer. The CNN's analysis will become more efficient.

First option is used in general practice. Also, it is important to mention that arguments values in layers influence the CNN efficiency as well.

5 CONCLUSIONS

The goal of this thesis project was to create a classifier based on a neural network using knowledge on deep learning and Python programming. A Convolutional Neural Network based on 2-dimensional input data was a suitable example of such a system. It is important to note that other types of neural networks as well as programming languages could be used to achieve the same results.

During the working process the author learnt new programming language and understood structure of CNN archetypes.

The author noticed that a vast amount of information connected to the thesis topic is discussed in blogs and forums of dedicated websites across the internet by developers, programmers and people interested in programming.

Convolutional Neural Networks might prove useful in areas, where accuracy and attention to minute details are important, such as medical field. Determining the slightest changes on medical scans at an early stage can help to decide on the best treatment.

In the end, set targets were reached and a suitable Convolutional Neural Network program was created to classify data by a binary outcome. Further development of this project is possible in the future, with the possibilities of creating more complex program for face or voice recognition as well as for creating a multi-purpose application.

REFERENCES

Albright D. (2016, December 13), *What Are Neural Networks and How Do They Work?* Retrieved from Make use of: https://www.makeuseof.com/tag/what-are-neural-networks/

Conda (November 22), *Concepts.* Retrieved from: https://conda.io/docs/user-guide/concepts.html

Dahal P. (November 15), *Classification and Loss Evaluation - Softmax and Cross Entropy Loss.* Retrieved from Deep notes: https://deepnotes.io/softmax-crossentropy

Foote K. D. (2017, February 7) *A Brief History of Deep Learning.* Retrieved from Dataversity: http://www.dataversity.net/brief-history-deep-learning/

Futurism (2015, August 18), *How Do Neural Networks Learn?* Retrieved from Futurism: https://futurism.com/how-do-artificial-neural-networks-learn

Karpathy A. (2016), *CS231n Convolutional neural networks for visual recognition.* Retrieved from: http://cs231n.github.io/convolutional-networks/#fc

Keras documentation (November 22). Retrieved from: https://keras.io/

MathWorks (October 15). *What Is Deep Learning?* Retrieved from: https://se.mathworks.com/discovery/deep-learning.html

Neurofantstic (2017, May 1), *Brain computation is a lot more analog than we thought.* Retrieved from: https://neurofantastic.com/brain/2017/4/13/brain-computation-is-a-lot-more-analog-than-we-thought

Nvidia (November 22), *CUDA toolkit.* Retrieved from: https://developer.nvidia.com/cuda-toolkit

Nvidia (November 22), *Nvidia cuDNN.* Retrieved from: https://developer.nvidia.com/cudnn

Python (November 22), *What is Python? Executive Summary.* Retrieved from: https://www.python.org/doc/essays/blurb/

Saxena S. (2017, October 26), *Artificial Neuron Networks (Basics) | Introduction to Neural Networks.* Retrieved from:

https://becominghuman.ai/artificial-neuron-networks-basics-introduction-to-neural-networks-3082f1dcca8c

Singh A. (2017, October 18), *How does the ReLU function work for z < 0?* Retrieved from: https://stats.stackexchange.com/questions/308689/how-does-the-relu-function-work-for-z-0

Spyder (November 22). Retrieved from: https://www.spyder-ide.org/

Tiwari S (November 2). *Activation functions in Neural Networks.* Retrieved from Geeks for geeks: https://www.geeksforgeeks.org/activation-functions-neural-networks/

Appendix 1

Contents of the file “CNN.py”

```
# Convolutional Neural Network

# Part 1 - Building the CNN

# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense

# Initialising the CNN
classifier = Sequential()

# Step 1 - Convolution
classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))

# Step 2 - Pooling
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Adding a second convolutional layer
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Step 3 - Flattening
classifier.add(Flatten())
```

```
# Step 4 - Full connection
classifier.add(Dense(units  = 128, activation = 'relu'))
classifier.add(Dense(units  = 1, activation = 'sigmoid'))

# Compiling the CNN
classifier.compile(optimizer  = 'adam', loss = 'binary_crossentropy',  metrics = ['accuracy'])

# Part 2 - Fitting the CNN to the images

from keras.preprocessing.image  import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale  = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)

test_datagen = ImageDataGenerator(rescale  = 1./255)

training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')

test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')

classifier.fit_generator(training_set,
                         steps_per_epoch  = 8000,
                         epochs = 25,
                         validation_data = test_set,
            validation_steps = 2000)

# Part 3 - Making new predictions


import numpy as np

from keras.preprocessing import image

test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg', target_size = (64, 64))

test_image = image.img_to_array(test_image)

test_image = np.expand_dims(test_image, axis = 0)

result = classifier.predict(test_image)


training_set.class_indices

if result[0][0] == 1:

    prediction = 'dog'

else:

    prediction = 'cat'
```