Stem Cell Classification

Alice Karnsund and Elin Samuelsson

Abstract — **Machine learning and neural networks have recently become hot topics in many research areas. They have already proved to be useful in the fields of medicine and biotechnology. In these areas, they can be used to facilitate complicated and time consuming analysis processes. An important application is image recognition of cells, tumours etc., which also is the focus of this paper.**

Our project was to construct both Fully Connected Neural Networks and Convolutional Neural Networks with the ability to recognize pictures of muscular stem cells (MuSCs). We wanted to investigate if the intensity values in each pixel of the images were sufficient to use as indata for classification.

By optimizing the structure of our networks, we obtained good results. Using only the pixel values as input, the pictures were correctly classified with up to 95.1% accuracy. If the image size was added to the indata, the accuracy was as best 97.9 %.

The conclusion was that it is sensible and practical to use pixel intensity values as indata to classification programs. Important relationships exist and by adding some other easily accessible characteristics, the success rate can be compared to a human's ability to classify these cells.

I. INTRODUCTION

THE goal of this project was to construct and train two types of neural networks with the purpose of distinguishing muscular stem cells, MuSC, in pictures.

Worldwide, one out of 3500 boys are born with the disease Duchenne Muscular Dystrophy (DMD). They lack a certain protein, dystrophin, which causes an ongoing deterioration of heart and skeletal muscles [1]. Researchers are investigating if stem cell therapy may have a positive inhibitive effect on the disease course. A few patients have already received transplantation of healthy MuSCs with pleasing results such as increased muscle mass and better lung capacity [2]. This treatment is a lifelong procedure and far from fully developed. This paper presents a tool for speeding up the MuSC research, with the hope of finding a cure for DMD patients.

The dataset we used was a large collection of pictures taken with time-lapse microscopy. This means a microscope camera had captured samples containing MuSC with a low frame rate, in this case every three minutes. Studying such picture sequences plays a crucial role in biomedical research. It helps scientists observe dynamic phenomena such as proliferation, migration, mitosis and cell death. Before, the cells' behaviours were manually examined in hour-long videos and small slow movements could easily be missed. Thanks to the method of time-lapse microscopy, it is now possible to follow infinitesimal changes from picture to picture [3].

The next step in improving the research process is to replace humans with computers for lengthy observations of the cell sample pictures. Manual investigations are time consuming and can turn out to be subjective to some degree. In other words, people are not entirely consistent. With computers, we can increase both efficiency, precision and consistency.

The programs presented in this paper aims to contribute to such automated observation of MuSC cell. We examined an existing MuSC classification program, described in [3]. The program performed an advanced pre-analysis to identify several features of the cell, like area and elongation, which was then used for classification. We wanted to eliminate this pre-analysis demand by investigating if the images themselves, or more exactly the intensity value in each pixel of the images, were enough to use as indata for the classification process.

II. THEORY

A. Machine Learning

Machine Learning has already begun to change our society and will most certainly continue to do so. Due to new technology, it is possible to collect tremendous amounts of information about almost anything. By implementing Machine Learning, computers can analyse those huge data sets more thoroughly and efficiently than a human ever could. So far, the method has given rise to established applications like the spam filter in our email inbox, web page ranking, face recognition, automatic translation and much more [4].

The term was introduced by Arthur Samuel in 1959 and he defined it as following: "Machine Learning is a field of study that gives computers the ability to learn without being explicitly programmed" [5]. The idea is to provide a computer with a huge set of samples, from which it is supposed to find and learn patterns.

A more rigorous definition was stated by the American computer scientist Tom Mitchell in 1998: "Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" [5]. This means that the more examples the computer may study and learn from (i.e. the more experience it has), the better it will find some desired connections between them and the better it will perform a certain task.

B. Supervised Learning

One main group of Machine Learning algorithms is Supervised Learning. The idea is to teach the program what the outputs should be, given some specific inputs, by letting it examine a training data set on the form $\{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \dots, (\boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)})\}$. This set contains m separate examples, each with some input $\boldsymbol{x}$ and "correct" output $\boldsymbol{y}$. Note that both $\boldsymbol{x}$ and $\boldsymbol{y}$ can be multi-dimensional structures – vectors, matrices etc. – and contain many different values for each example [6].

The computer will implement some learning algorithm to fit a model to the data set. This means finding calculations to perform on the values in $\boldsymbol{x}$ resulting in output values as close to the desired $\boldsymbol{y}$ as possible. We say that the program is being trained. The found model may then be used on an arbitrarily chosen $\boldsymbol{x}$, to estimate what $\boldsymbol{y}$ it should correspond to [6]. In other words, the computer finds crucial links between $\boldsymbol{x}$ and $\boldsymbol{y}$ for the examples in the data set.

To clarify, we study a house pricing example described by Andrew Ng in [7]. The training data set is a batch of sold houses. For each house i we have an input vector $\boldsymbol{x}^{(i)}$ and an output scalar $y^{(i)}$. $\boldsymbol{x}^{(i)}$ contains a few characteristics of the house, such as living area, holding of sea view, number of bedrooms etc. $y^{(i)}$ is the price the house was sold for. The learning algorithm's mission is to find a function $f(\boldsymbol{x})$ from $R^{n\times 1} \rightarrow R^{1\times 1}$, for instance a polynomial, by minimizing the summed square error between the $f(\boldsymbol{x}^{(i)})$-values and the corresponding $y^{(i)}$-values for all examples $i = 1, 2, \dots, m$. If $\boldsymbol{x}$ is one dimensional ($n = 1$), this is a simple curve fitting, illustrated in Fig. 1.

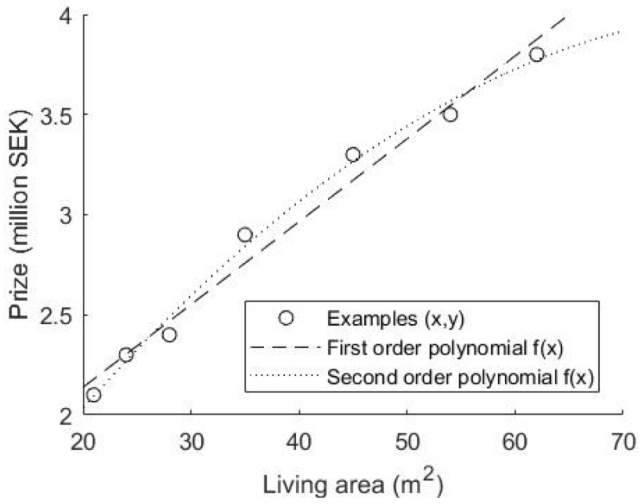

Fig. 1. The house pricing example when x holds only one characteristic - the living area.

The found function f is the desired link between the input $\boldsymbol{x}$ and the price y. It may be applied on the $\boldsymbol{x}$-vector of any house, to estimate what price y one can expect to sell it for.

$\boldsymbol{y}$ in the house pricing example takes continuous values, which makes it a regression problem. This is in contradiction to a classification problem, where $\boldsymbol{y}$ only may take discrete values. E.g. for a set of tumours, the $\boldsymbol{x}$-vector can consist of size, shape and other characteristics, while $\boldsymbol{y}$ only takes two values - 1 for being malignant or 0 for being benign. After training, we will have a model that gives us a 2-dimensional output $[p_1, p_0]$ with the probability of an arbitrary tumour being 1 respectively 0, based on its unique $\boldsymbol{x}$-vector [7].

The opposite group of Machine Learning algorithms, Unsupervised Learning, is used when we have training sets $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(m)}\}$ without any "correct answers" $\boldsymbol{y}$. The program's mission is to cluster them appropriately, without having any specific answers to aim for [8].

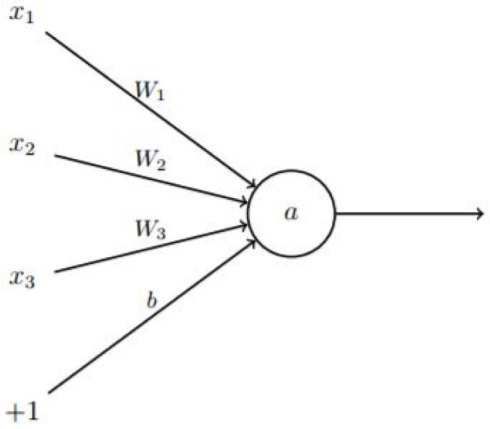

Fig. 2. Model of a single neuron.

C. Fully Connected Neural Network

For large data sets with multi-dimensional inputs and/or outputs, fitting a model may be challenging. Advanced learning algorithms are required. This leads us to the Neural Networks, which were developed with inspiration taken from the biological neural structures of the human brain. Their unique structure has turned out to be very efficient [9].

Fig. 2 shows a model of a single neuron, which is the smallest element of a Neural Network. Here the neuron is given the inputs of an $\boldsymbol{x}$-vector $[x_1, x_2, x_3]$ together with an additional $+1$ term called a bias unit. Each input is multiplied with a weight and then summed with the others. In Fig. 2 the weights are W_i for x_i and b for $+1$, and the sum becomes $z = \sum_{i=1}^{3} W_i x_i + b$, [10].

Lastly, an activation function f is applied on z, to receive an appropriate output, which we call the activation $a = f(z)$. Frequently used activation functions are the sigmoid function $f(z) = \frac{1}{1+e^{-z}}$, the hyperbolic tangent $f(z) = tanh(z)$ and the rectified linear function $f(z) = max(0, z)$. They all have the property that they "squeeze" the output value into some range: $[0, 1]$, $[-1, 1]$ or $[0, \infty]$, as visualized in Fig. 3 [10]. The function that fits best depends on the problem. For instance, the sigmoid function is applied in classification problems between two classes 1 and 0, since its output may be interpreted as p_1 = the probability of the example being in class 1 [11].

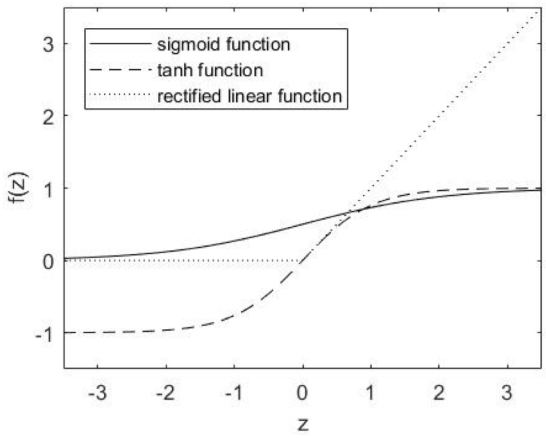

Fig. 3. Activation functions.

Neurons can be organized in layers. The input and output layers are the only layers that can be observed from outside the network. The layers in between are therefore called hidden layers [10].

By connecting certain neurons, that is letting the output of one neuron be the input to another neuron in the next layer, a Neural Network can be built. Fig. 4 is a model of a tiny Fully Connected Neural Network (FCNN), which requires the property that all neurons in one layer is connected to all neurons in the next layer (except for the +1-bias unit). The activation (i.e. the output) from a gathering unit i in layer $l+1$ is denoted $a_i^{(l+1)} = f(\sum_j W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)})$, where f is the activation function. With this notation, the initial inputs fulfil $x_i = a_i^{(1)}$. The final output of the whole network is called a hypothesis $h_{W,b}(\boldsymbol{x}) = \boldsymbol{a}^{(L)}$, where L denotes the index of the last layer [10].

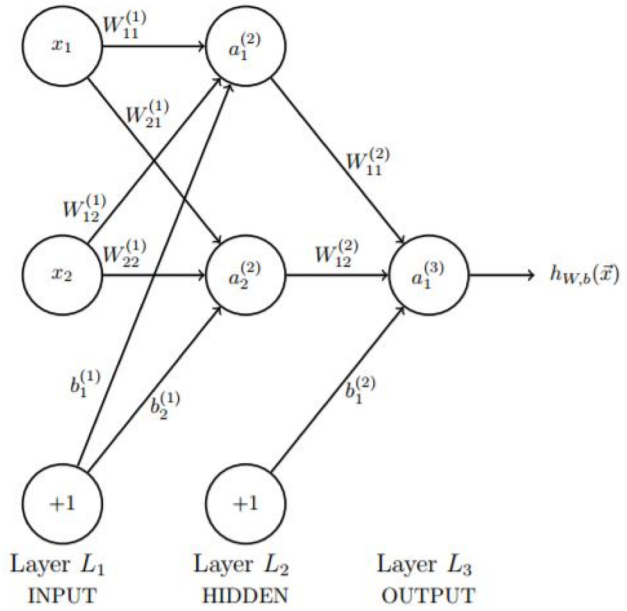

Fig. 4. Model of a small Fully Connected Neural Network.

This leads to the crucial question: How are the weights $W_{ij}^{(l)}$ and $b_i^{(l)}$ chosen? The answer is: by training the network. We start out with randomly selected weights, stored in matrices W_{start} and b_{start}. They are then iteratively amended under the condition: "the hypothesis $h_{W,b}(\boldsymbol{x})$ should be as close to the correct $\boldsymbol{y}$ as possible for all examples in the training set $\{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \dots, (\boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)})\}$ [10].

Each iteration of weight update is called an **Epoch** and is divided into partial steps. First, the $\boldsymbol{x}$-values of the training set are taken as inputs and led through the network to provide us with one hypothesis $h_{W,b}(\boldsymbol{x}^{(i)})$ for every example $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ $i = 1, 2, \dots, m$. This is called **Feedforward Propagation.** A cost function $J(W, b)$ then calculates the total error between all pair of hypotheses $h_{W,b}(\boldsymbol{x}^{(i)})$ and correct values $\boldsymbol{y}^{(i)}$, $i = 1, 2, \dots, m$ [10]. Finally, the weights are updated to decrease the cost function $J(W, b)$ and thereby the error. This method is called **Gradient Descent** and is visually explained in Fig. 5, taken from [12]. The new weights become:

$$W_{ij\,new}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij\,old}^{(l)}} J(W, b)$$

$$b_{i\,new}^{(l)} = b_{i\,old}^{(l)} - \alpha \frac{\partial}{\partial b_{i\,old}^{(l)}} J(W, b)$$

Calculating the derivatives is complicated and we will not describe the method here, but just mention that it is numerically implemented using an algorithm called **Backpropagation** [10].

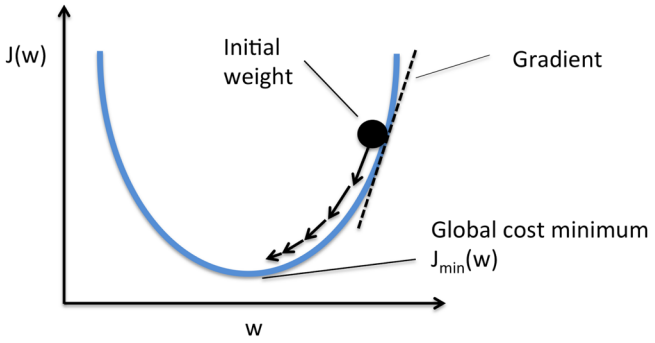

Fig. 5. Gradient Descent in one dimension, figure taken from [12].

The parameter α is called the learning rate. It controls the step size in every weight update. If α is too small, the network will be slow and might never terminate. If α is too large, we risk divergence of the cost function $J(W, b)$ since one step could accidently pass by the minimum and miss it.

The more executed epochs, the closer $J(W, b)$ will get to its minimum, i.e. the smaller error, the better the Neural Network. On the other hand, every epoch is time consuming and eventually a limit is reached, where weights are only marginally updated and does not make a significant difference to the output value anymore.

The choice of α and number of epochs are two important parameters to consider when constructing a Neural Network.

For a clear overview of all specific calculations and a more

detailed explanation of Backpropagation, we would like to recommend Andrew Ng et al.'s online guide to Multi-Layer Neural Network [10].

One might think that using the more neurons in a network, the better results. This is not necessarily the case. If too many, the network may learn to recognize the specific examples in the training set, instead of finding the patterns and connections that unite a class. Such Neural Network has no generality and does not work well on any other examples than those in the training set. This is called **overfitting** [13].

D. Convolutional Neural Network

Convolutional Neural Networks (CNNs) are especially convenient for image classification, since they take advantage of the two-dimensional structure of a picture. Instead of just connecting all neurons in all layers, like the Fully Connected Neural Network does, a Convolutional Neural Network concentrates on finding relationships to the neighbouring area. In other words, small parts of the picture are studied separately.

The basic building blocks of a CNN are:

- Convolutional Layers (Fig. 8)
- Max-pooling Layers (Fig. 10)
- Fully Connected Layers (Fig. 11)

They can be combined in many ways to create CNNs with different qualities. In between the layers, appropriate activation functions are applied [14].

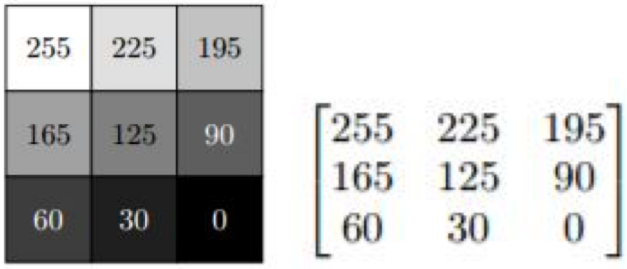

Fig. 6. A 3x3 picture in grayscale with corresponding pixel intensity matrix.

As always, we start out with our input $\boldsymbol{x}$. But instead of being a vector, $\boldsymbol{x}$ is now a matrix. Each element in the matrix represents a pixel of the picture by its intensity value. In this project, grey scale images are used, so one intensity value between 0 (black) and 255 (white) is enough, Fig. 6. For coloured pictures, three RGB-values would be needed [15].

The purpose of the Convolutional Layer is to identify features in the picture by letting filters stride across it. One filter represents one characteristic, such as a stripe, a curve etc. At each location, the filter is convolved with the picture, simply meaning that it performs element-wise multiplication and then summation. The result indicates how well this partial area holds this filter's feature. One common way is to use filters that outputs a large value if it matches and a small/negative value if it does not [15] [16], but there are other methods as well. A numerical example is displayed in Fig. 7.

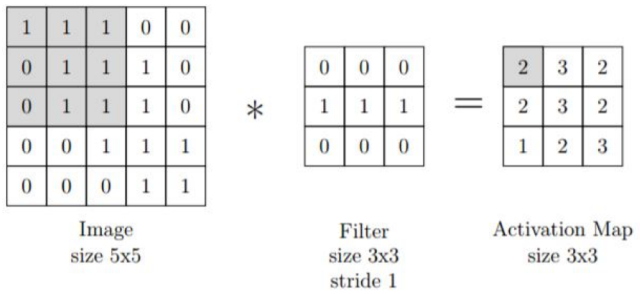

Fig. 7. Convolution example.

Various filters are used to identify different features. Each filter has its own output, called an activation map, shown in Fig. 8. The total output from a convolutional layer therefore adds one dimension: a depth equal to the number of filters [15]. In Fig. 8, three filters give three activation maps, i.e. the depth is three. Height and width are decreased according to this formula:

$$size_{activation\ map} = \frac{size_{input} - size_{filter}}{stride_{filter}} + 1$$

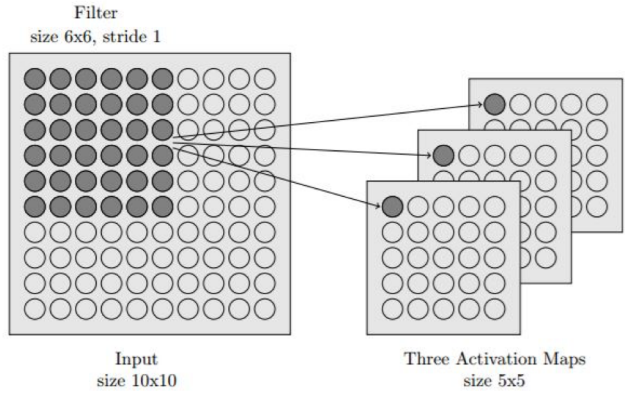

Fig. 8. Convolutional Layer example.

A useful tool when applying filters is padding. Padding can be viewed as adding a frame of zeros around the input image, as in Fig. 9. In this way, the filters can reach potentially crucial features along the edges. Using padding 1 and stride (step length) 1, the dimensions are kept after the Convolutional layer. This improves the CNN's performance and leaves all spatial down-sampling to the pool-layers [17].

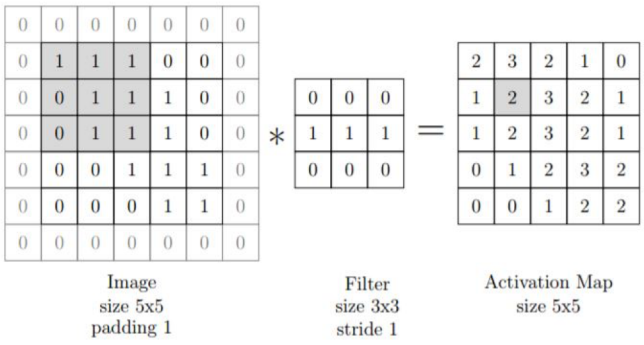

Fig. 9. Convolution with padding example.

It is important to choose sizes, strides and paddings appropriately. With strides larger than one, we risk losing information if the dimensions do not work out perfectly (i.e. the filters might not reach the whole picture). Then the leftover parts will just be ignored and that information will be lost [17].

A Convolutional Layer is in many cases, including this project, followed by a rectified linear activation function. The function replaces all negative values with zero [18], as we can see in Fig. 3.

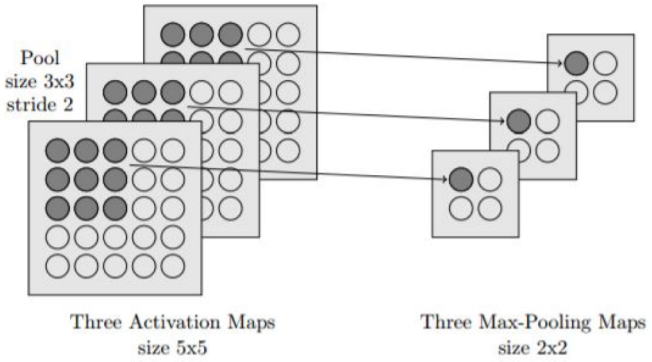

Fig. 10. Max-pooling Layer example.

To avoid overfitting (that is, the network starts to recognize each picture separately, instead of finding general patterns), we want to downsize the activation maps using a max-pooling layer. Fig. 10 shows an example. A pool strides across the activation maps just like a filter, picks out the maximum value of each area and saves it in a max-pooling map. In this way, the spatial dimension decreases but the depth (i.e. number of filters) stays the same [19].

In large pictures, small features may build up more complex features, like lines build up a grid. Therefore, it might be appropriate to loop these first layer combinations (Convolutional Layer + Rectified Linear Function + Max-pooling Layer) to perform more than one convolution [15].

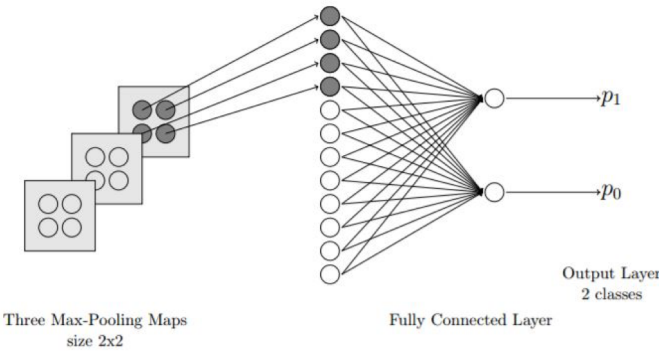

Fig. 11. Fully Connected Layer example.

The CNN always ends with at least one Fully Connected Layer, Fig. 11. All nodes in the last max-pooling maps are connected to the Fully Connected Layer, which has the same dimension as the number of classes. A Softmax activation function is then applied, resulting in probability values for each class as output. The earlier shown sigmoid function in Fig. 3 is the special case of a Softmax function, for when there are only two classes. Lastly, a Classification Layer concludes what the picture represents from the probabilities [14].

The many layers make the CNN training procedure very complicated. The program does not only need to find appropriate filter features, but also find connections between these features and full motives. Many weights need optimization and one training session is often very time consuming.

III. METHOD

A. Muscular Stem Cell Data

The data set for this project was two batches of pictures of MuSC samples. One example of a picture is shown in Fig. 12. A total of 170 MuSC samples had been captured through a microscope every three minutes. The first batch, from 2009, consisted of 25627 such pictures. The other batch, from 2010, had 59764 pictures.

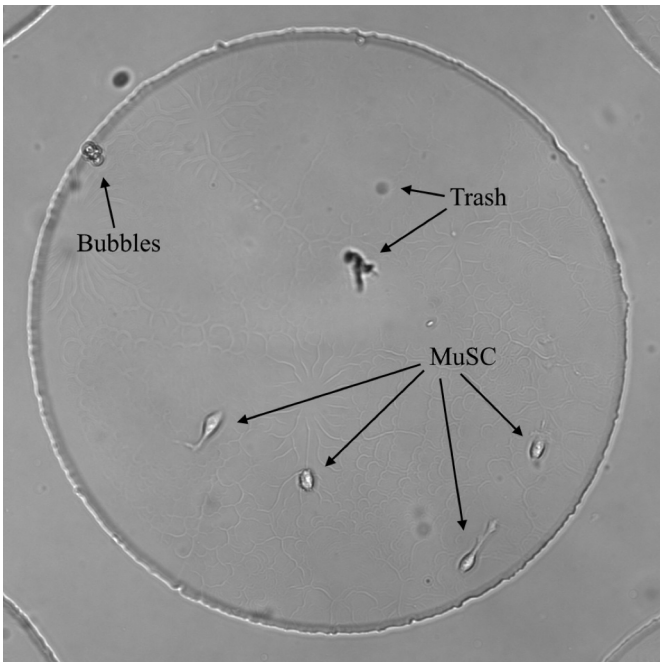

Fig. 12. A time-lapse picture of a sample, including labels on blobs.

Our project was an extension of the doctoral thesis "Segmentation and tracking of cells and particles in time-lapse microscopy" [3], in which an algorithm for cell tracking was developed, using the exact same data set. As part of that project, a contrast sensitive program was constructed to identify irregularities in the pictures. That program picked out all small rectangular areas where a cell or cell-like object appeared to be. In this paper, these cut out objects will be referred to as "blobs". Each blob had its unique index and the coordinates (location, width and height) of each blob were stored in a matrix on the row of their corresponding index. The pictures from 2009 contained a total of 103237 blobs and those from 2010 contained 148701 blobs.

As we can see in Fig. 12, not all blobs were cells. The cell tracking algorithm in [3] performed a pre-analysis to determine 73 advanced characteristics, e.g. area and elongation, as inputs. These were then used to classify the blobs as cells or trash/bubbles.

Our project aimed to ease that classification process by using only the pixel intensity values themselves as input. No calculations on the blobs were required. We did all necessary programming in Matlab.

B. Generating the training data set

We had access to the following information when we created our training data set.

1. The blob coordinate matrix. Each blob had its own row corresponding to its unique index. The four columns were x-coordinate, y-coordinate, width and height.
2. All pictures
3. A column vector with correct information about number of cells in each blob-picture. Each blob had its own row corresponding to the index.

The batches (2009 and 2010) were treated separately. We limited our task to identifying if there were one or more cells ($y = 1$) or none ($y = 0$) in each blob-picture. By replacing all values > 1 with 1 in the "number of cells"-vector (point three in the list), we created our y-vector, containing all correct answers. The dimension was 103237x1 for 2009 data and 148701x1 for 2010 data.

$\boldsymbol{x}$ was a two-dimensional matrix containing intensity values for the pixels in each blob-picture, which is why all $\boldsymbol{x}$:es were appropriate to store in a three-dimensional matrix. Using the coordinate information, each blob-picture was cut out from its main picture. The original size (height and width) of course varied between all blob-pictures. To use as input to the same neural network, all $\boldsymbol{x}$ needed to be the same dimension, which is why each blob-picture was rescaled with the Matlab command **imresize**, before saved to the 3D matrix.

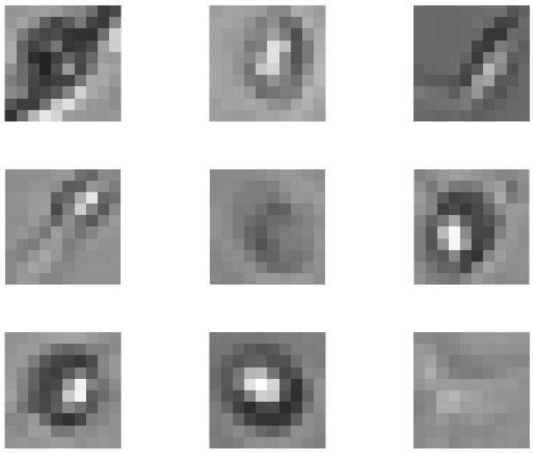

Fig. 13. Nine randomly chosen 10x10 blobs.

For each year, two training data sets were created. One with pictures rescaled to 10x10 pixels. This is because we wanted our program to be as fast and uncomplicated as possible. We studied some rescaled images manually, like the ones in Fig. 13, and decided that 10x10 was the smallest dimension where there was still significant difference between the blobs. The matrix dimensions became 10x10x103237 and 10x10x148701 respectively.

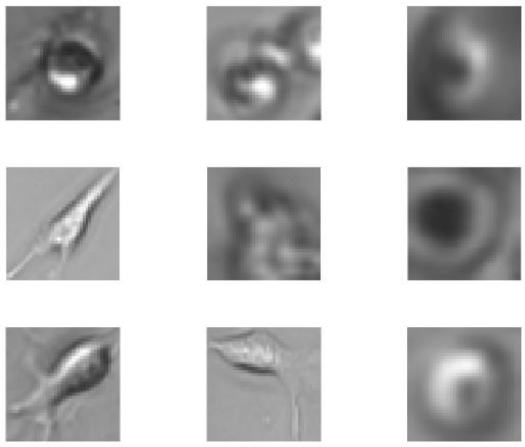

Fig. 14. Nine randomly chosen 40x40 blobs.

For the other training data set, the pictures were rescaled to 40x40 pixels, which was the average size of all blobs, Fig 14. The matrix dimensions became 40x40x103237 and 40x40x148701 respectively.

C. Implementation of Fully Connected Neural Network

We decided to take advantage of Matlab's embedded Neural Network toolbox. For Fully Connected Neural Networks, Matlab has designed apps together with wizards for several different problems. They guide the user step by step through constructing and training a neural network. They are reached with the command **nnstart**. One of them is the Pattern Recognition app, command **nprtool**, which suited our project perfectly [20].

The input to the wizard had to be two-dimensional, so we rewrote our $\boldsymbol{x}$ to have each blob's pixel intensity values stored on a row instead of in a matrix. The dimension was then changed from 10x10x103237 to 103237x100 for the 2009 data and likewise for the 2010 data. Lastly, we shuffled the rows, naturally in the same way for input data $\boldsymbol{x}$ and target data $\boldsymbol{y}$.

We choose the 2009 data for training, and used Matlab's wizard's standard settings to divide it: 70% was taken as training data, 15% as validation data and 15% as testing data. Validation data are used to measure network generalization and to halt training when generalization stops improving. Testing data are only used for evaluation [20].

The structure was a Fully Connected Neural Network (FCNN) with one hidden layer. The number of neurons in this hidden layer was the only parameter we could vary when constructing different networks, everything else (number of

epochs, learning rate etc.) was optimized by the wizard.

Beyond the 15% 2009 testing data, we also evaluated our networks with 2010 data to determine the generality. If we have very high accuracy when evaluating with 2009 data, but low accuracy with 2010, the network is probably overfitted.

Since we wanted our FCNN to be as efficient as possible, we considered possible improvements. We concluded that the original size of the blob-picture, i.e. the width and height before rescaling to 10x10 or 40x40, should be a relevant feature to add to the training set. We simply added the width and height from the coordinate matrix to the input rows.

To evaluate the importance of this addition, we also trained FCNNs using only the original sizes of the blob-picture as inputs.

D. Implementation of Convolutional Neural Network

There is no Matlab wizard or app for constructing Convolutional Neural Networks (CNNs), but all relevant commands are compiled in [14]. The input to a CNN in Matlab must be images, stored in maps of their correct classification, which in our case was one 0-map and one 1-map for each dataset. Using the **imwrite** command, we wrote a program that turned all our pixel matrices into png-files and saved them in their corresponding 0- or 1-map.

By following the steps in [14], we wrote our CNN script. The sample data was loaded as an **imageDataStore** object and divided into training data and test data. Just like with the Fully Connected Neural Networks, we trained our CNNs with only 2009 data, and then evaluated it with both 2009 and 2010 data to investigate its generality.

Out of the 103237 blobs from 2009, 53416 was trash (class 0) and 49821 cells (class 1), according to our $\boldsymbol{y}$-vector. We trained the network with 42000 of each class and evaluated it with the remaining 19237 blobs, which was 18.6% of the batch.

We built our CNNs of the following layers:

1. Image Input Layer
2. Convolutional Layer
3. ReLU Layer (Rectified Linear activation function)
4. Max-pooling Layer
5. Fully Connected Layer
6. Softmax Layer
7. Classification Layer

We trained it and evaluated it with different choices of the parameters stated below. By varying them one at a time, we found our optimal CNN.

Conv. Layer:	Filter size
	Filter stride (step length)
	Filter quantity
	Padding
Max-pool Layer:	Pool size
	Pool stride (step length)
Training:	Learning rate
	Number of epochs

IV. RESULTS

A. Fully Connected Neural Network 10x10 pixels

TABLE 1
RESULTS FOR FULLY CONNECTED NEURAL NETWORK USING 10 X 10 PIXELS

Input data (always from 2009)	# Neurons in the hidden layer	Accuracy for evaluation with 2009 data	Accuracy for evaluation with 2010 data
Only pixel values 100 inputs for each blob	1	0.686	0.580
	10	0.875	0.688
	20	0.904	0.703
	50	0.927	0.707
	100	0.920	0.711
Pixel values and sizes 102 inputs for each blob	1	0.897	0.808
	10	0.959	0.809
	20	0.964	0.815
	50	0.974	0.815
	100	0.978	0.830
	1000	0.979	0.811
Only sizes 2 inputs for each blob	1	0.855	0.780
	4	0.858	0.792
	10	0.863	0.792
	20	0.865	0.780
	50	0.871	0.788
	100	0.867	0.761

The maximum values are grey-marked.

B. Fully Connected Neural Network 40x40 pixels

TABLE 2
RESULTS FOR FULLY CONNECTED NEURAL NETWORK USING 40 X 40 PIXELS

Input data (always from 2009)	# Neurons in the hidden layer	Accuracy for evaluation with 2009 data	Accuracy for evaluation with 2010 data
Only pixel values 1600 inputs for each blob	1	0.693	0.596
	10	0.874	0.712
	50	0.928	0.736
	100	0.931	0.712
	200	0.951	0.744
	1000	0.947	0.741
Pixel values and sizes 1602 inputs for each blob	1	0.751	0.658
	10	0.930	0.779
	100	0.944	0.740
	200	0.941	0.727
	1000	0.960	0.760

The maximum values are grey-marked.

C. Convolutional Neural Network 10x10 pixels

Parameter choices for TABLE 3:

Conv. Layer:	Filter size:	**VARIED**
	Filter stride:	1
	Filter quantity:	10
	Padding:	**VARIED**
Max-pool Layer:	Pool size:	**VARIED**
	Pool stride:	1
Training:	Learning rate:	0.0001
	Epochs:	10

TABLE 3
RESULTS FOR CONVOLUTIONAL NEURAL NETWORK
USING 10 X 10 PIXELS

Filter size	Padding	Pool size	Accuracy for evaluation with 2009 data	Accuracy for evaluation with 2010 data
2	0	2	0.8247	0.7215
4	0	2	0.8521	0.6898
6	0	2	0.8608	0.7355
7	0	2	0.8795	0.7512
8	0	2	0.8735	0.7483
9	0	2	0.8759	0.7398
7	1	2	0.8757	0.7459
7	2	2	0.8821	0.7445
8	1	2	0.8936	0.7503
8	2	2	0.8980	0.7648
8	3	2	0.8842	0.7385
8	2	1	0.8904	0.7463
8	2	3	0.8974	0.7562
8	2	4	0.8885	0.7402

The currently varied optimization parameter and the maximum values are grey-marked.

Optimal parameter choices:
Filter size: 8
Padding: 2
Pool size: 2

Parameter choices for TABLE 4:
Conv. Layer: Filter size: 8
Filter stride: 1
Filter quantity: **VARIED**
Padding: 2
Max-pool Layer: Pool size: 2
Pool stride: 1
Training: Learning rate: 0.0001
Epochs: **VARIED**

TABLE 4
RESULTS FOR CONVOLUTIONAL NEURAL NETWORK
USING 10 X 10 PIXELS

Filter quantity	# Epochs	Training time (sec)	Accuracy for evaluation with 2009 data	Accuracy for evaluation with 2010 data
10	10	641	0.8980	0.7648
20	10	2885	0.9153	0.7665
10	20	1293	0.9020	0.7606

The maximum values are grey-marked.

There was only little significant difference when we trained more thoroughly, see TABLE 4, so we concluded that we had achieved results close to the maximum accuracy.

D. Convolutional Neural Network 40x40 pixels

Parameter choices for TABLE 5:
Conv. Layer: Filter size: 32
Filter stride: **VARIED**
Filter quantity: 10
Padding: 8
Max-pool Layer: Pool size: **VARIED**
Pool stride: **VARIED**
Training: Learning rate: 0.0001
Epochs: 10

TABLE 5
RESULTS FOR CONVOLUTIONAL NEURAL NETWORK
USING 10 X 10 PIXELS

Filter stride	Pool size	Pool stride	Accuracy for evaluation with 2009 data	Accuracy for evaluation with 2010 data
4	2	1	0.9007	0.7908
4	3	1	0.9060	0.7991
4	4	1	0.9166	0.7962
2	5	2	0.9226	0.7885
2	7	2	0.9199	0.7978
2	9	2	0.8703	0.7706
1	8	4	0.9162	0.7985
1	9	4	0.9192	0.7753

The currently varied optimization parameter and the maximum values are grey-marked.

Optimal parameter choices:
Filter stride: 4
Pool size: 3
Pool stride: 1

E. Variance analysis

No variance for Convolutional Neural Networks.

TABLE 6
AN EXAMPLE DISPLAYING VARIATIONS IN
RESULTS FOR FULLY CONNECTED NEURAL NETWORK
USING 10 X 10 PIXELS

Input data (always from 2009)	# Neurons in the hidden layer	Accuracy for evaluation with 2009 data	Accuracy for evaluation with 2010 data
Pixel values and sizes 102 inputs for each blob	20	0.964	0.815
		0.968	0.819
		0.961	0.813
		0.973	0.811
		0.971	0.818
		0.974	0.818
	MAXIMUM DIFFERENCE	0.013	0.008

The maximum difference values are grey-marked.

V. DISCUSSION

We could identify several patterns in our result tables. They gave us indications on which neural network parameters were most important for good accuracy and which possibilities there were for further development of the neural networks.

Since we aimed for a network with good generality, the 2010 accuracy was more important in our analysis than the 2009 accuracy.

A. Fully Connected Neural Network

The very best results were obtained with Fully Connected Neural Networks with a combined input of both 10x10 pixel intensity values and the original sizes, i.e. the width and height of the blob-picture before rescaling. As we can see in TABLE 1, that input gave 0.830 accuracy for 2010 evaluation. Most credit should be given to the size data though, since only sizes as indata gave better results than only pixels as indata – 0.792 compared to 0.711.

Another interesting observation from comparing TABLE 1 and TABLE 2 was that more pixels (40x40) was beneficial in the FCNNs when the indata was only pixel values, but disadvantageous when it was both pixels and sizes. This made sense. 40x40 pictures naturally contained more information than 10x10 pictures, which is why 40x40 gave better accuracy when analysing only the pixels. The original sizes were more important though. When adding them to the 40x40 pixel data they had to share the attention with 1600 other nodes, instead of just 100 as for 10x10 pictures, and the accuracy deteriorated with the number of pixels. But remember that this happened at some dimension limit nxn, up until which it was beneficial to combine both pixels and original sizes.

This concludes that the sizes were the most important parameters, though this is somehow unfair. In accordance with the theory, the two-dimensional structure of a picture suits a CNN better than a FCNN. The best results with pixel input to a CNN was 0.7991 (TABLE 5), which beat a FCNN with only the original sizes as input, of 0.792. It would have been interesting to somehow add the sizes to the CNN data as well. We have considered maybe analysing the pictures as usual with convolutional and max-pooling layer and then, when we return to one dimension, add original height and weight to the fully connected layer. This requires some complex coding though.

B. Convolutional Neural Network

There were many parameters to optimize for the CNNs and we suspect that there should be a lot of development potential left for the 40x40 data.

Pictures of 10x10 pixels are very small, so more than one convolution seemed overkill. Since the cells were closely cut out, it was reasonable that padding was beneficial. Otherwise, important features could hide along the edges. Besides the risk of overfitting, theory told us that stride 1 is always desirable since it minimizes the loss of information. In other words, we cannot think of many more possible improvements for our 10x10 data. Therefore, the value 0.7665 from TABLE 4 is probably quite close to the maximum general accuracy for such small and blurred pictures.

The 40x40 indata leaves more space for variations and we did not have time to try all combinations we wanted to. Therefore, we simply used our conclusions from the 10x10 CNNs and kept the ratio between picture and filter size and between picture and padding.

Pixels	10x10	→	40x40
Filter size	8x8	→	32x32
Padding	2	→	8

We varied the strides (step lengths) and pool size in TABLE 5. The main reason we tried larger stride values was to reduce the training time. It is worth mentioning that stride 1 for 10x10 pixels corresponds to stride 4 for 40x40 pixels. Stride 4 was never exceeded, which means we always kept at least the same level of details as we did for the smaller pictures.

However, the whole analysis for 40x40 CNN was completely built on the conclusions from the 10x10 CNN. We think a more fundamental analysis, combined with adding more layers, both convolutional and fully connected, should be able to improve the accuracy even further.

C. Overfitting

Another interesting observation was when overfitting occurred and generality deteriorated for different input data sets. It was clear in TABLE 1 for the FCNNs with only size as input. The input was only two parameters, width and height, for each blob. The 2009 accuracy improved with increasingly number of neurons, but the 2010 accuracy performed as best with fewer neurons. 4 to 10 neurons were just enough to find patterns between the two input parameters, without starting to recognize the specific training examples too well.

Such overfitting connections were never found in our CNNs, not in TABLE 3 or in TABLE 5. Improvement in the 2009 accuracy almost always meant improvement in the 2010 accuracy too. Maybe we would have been able to detect overfitting if we had added more convolutional or fully connected layers.

D. Training time

One last resulting parameter we would like to discuss is the training time, where our many networks differed enormously. A training session with 10x10 pictures as indata to a FCNN with few neurons took approximately one minute. This can be compared to one training session of a CNN, with 40x40 pictures as input, which took exactly 8442 seconds, that is 2 hours and 21 minutes. As we can see in TABLE 4, increasing number of epochs did give better accuracy, just like theory told us. The improvements were very small though, with respect to the extra training time they required. Training time was a crucial factor throughout this whole project, especially since it was the reason we could not optimize our 40x40 CNNs as much as we wanted to.

E. Variance

It is also worth mentioning that no FCNN was the other one alike, since the Matlab wizard started out with randomly

chosen weights and did a random division into training/test data. Even if we used the exact same network structure and input data to the wizard, we had different accuracies, as displayed in an example in TABLE 6. The order of magnitude of the maximum variations were 0.001, or 0.1 percentage unit. This does affect our specific choices of parameters. However, the overall patterns and conclusions we have identified stay within the error margin.

For the CNNs on the other hand, there were no variance at all. We wrote code instead of using a wizard and were able to make an equal division into training/test data for all CNNs. Also, these Matlab algorithms ended up with the same weights every time and the accuracies never varied between two CNNs that were trained with the same settings.

VI. SUMMARY

In this paper, we explored two different types of neural networks, the Fully Connected Neural Networks and the Convolutional Neural Networks.

From the project, we concluded that it is relevant to use pixel intensity values as inputs to a classification program. Our results show that there exists clear links between the pixel intensity values and the classification of a cell picture. CNN was a better neural network structure than FCNN for finding those links.

Unfortunately, our program is not applicable in its current shape. We require high generality and our 2010 accuracy of 0.7991 would not have been acceptable in real research processes. Though, if the pixel inputs to our CNN could be combined with some easily accessible features (e.g. the original blob-picture size) the performance should improve significantly.

Using the pictures themselves as input, instead of calculating certain features as in [3], has a clear advantage in its simplicity. Image classification is an established method. The best methods of image classifiers use multiple convolutions and huge databases. Our focus was to create a small and simple program just to examine the potential of the problem. There are room for many more improvements.

As mentioned in the beginning, machine learning has contributed to great progress in many research areas during the last few years. For example, the company IBM has launched something called IBM Watson health. Watson's vision is to improve the healthcare by examining a huge amount of collected personal and academic health data. This is done together with partners within the medical, pharmaceutical and hospital fields [21]. The Manipal Hospital in Florida have recently started to use Watson for Oncology (i.e. the study of tumours) in their treatment of patients. In a study at the hospital, Watson proved to be able to classify breast cancer tumours at 90% accuracy. Watson also suggests a treatment plan, based on its enormous data bank, containing much more information than a human doctor could ever access under his/her lifetime. Therefore, the doctors at Manipal Hospital now always use Watson as a check before they agree on a treatment plan. They are convinced that this is the future of medicine [22].

VII. ACKNOWLEDGEMENTS

First, we would like to turn to our supervisor Joakim Jaldén. We want to thank him for being such a great coach and for his pedagogical support when in times of confusion. Due to the great material he provided us with, it was possible for us to work and investigate the different areas mainly on our own.

Further we would like to thank MathWorks for developing such useful toolboxes in Matlab. Without those, these good results would probably not have been possible to reach in this short period of time.

REFERENCES

[1] Eurostemcell. (2017, April). Muscular dystrophy: how could stem cells help?. [Online]. Available: http://www.eurostemcell.org/muscular-dystrophy-how-could-stem-cells-help

[2] JLENNER. (2017, April). First Duchenne's Muscular Dystrophy Patient To Receive Umbilical Cord Stem Cell Therapy In US Turns 30. [Online]. Available: https://www.cellmedicine.com/first-duchennes-muscular-dystrophy-patient-to-receive-umbilical-cord-stem-cell-therapy-in-us-turns-30/

[3] K. E. G. Magnusson, "Segmentation and tracking of cells and particles in time-lapse microscopy," School of Electrical Engineering, KTH Royal Institute of Technology, Stockholm, Sweden, 2016.

[4] A. Smola, and S. V. N. Vishwanathan, "Applications," in *Introduction to Machine Learning*, Cambridge, United Kingdom: The Press Syndicate of the University of Cambridge, 2008, pp. 3-7.

[5] A. Ng. (2017, April). Introduction: What is Machine Learning?. [Online]. Available: http://openclassroom.stanford.edu/MainFolder/VideoPage.php?course=MachineLearning&video=01.2-Introduction-WhatIsMachineLearning&speed=100

[6] A. Ng. (2017, April). Linear Regression I: Model Representation. [Online]. Available: http://openclassroom.stanford.edu/MainFolder/VideoPage.php?course=MachineLearning&video=02.2-LinearRegressionI-ModelRepresentation&speed=100

[7] A. Ng. (2017, April). Linear Regression I: Supervised Learning Intro. [Online]. Available: http://openclassroom.stanford.edu/MainFolder/VideoPage.php?course=MachineLearning&video=02.1-LinearRegressionI-SupervisedLearningIntro&speed=100

[8] A. Ng. (2017, April). Introduction: Unsupervised Learning Introduction. [Online]. Available: http://openclassroom.stanford.edu/MainFolder/VideoPage.php?course=MachineLearning&video=01.4-Introduction-UnsupervisedLearning&speed=100

[9] Neuralyst. (2017, April). Basic Concepts for Neural Networks. [Online]. Available: https://www.cheshireeng.com/Neuralyst/nnbg.htm

[10] A. Ng, J. Ngiam, C. Yu Foo *et al.* (2017, April). Multi-Layer Neural Network. [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

[11] A. Ng, J. Ngiam, C. Yu Foo *et al.* (2017, April). Logistic Regression. [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/LogisticRegression/

[12] S. Raschka. (2017, April). Fitting a model via closed-form equations vs. Gradient Descent vs Stochastic Gradient Descent vs Mini-Batch Learning. What is the difference?. [Online]. Available: https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html

[13] MathWorks. (2017, April). Improve Neural Network Generalization and Avoid Overfitting. [Online]. Available: https://se.mathworks.com/help/nnet/ug/improve-neural-network-generalization-and-avoid-overfitting.html

[14] MathWorks. (2017, April). Create Simple Deep Learning Network for Classification. [Online]. Available: https://se.mathworks.com/help/nnet/examples/create-simple-deep-learning-network-for-classification.html

[15] A. Deshpande. (2017, April). A Beginner's Guide To Understanding Convolutional Neural Networks. [Online]. Available:

https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

[16] RoboRealm. (2017, April). Convolution Filter. [Online]. Available: http://www.roborealm.com/help/Convolution.php

[17] A. Karpathy. (2017, April). Convolutional Neural Networks (CNNs / ConvNets). [Online]. Available: http://cs231n.github.io/convolutional-networks/

[18] MathWorks. (2017, April). Create a Rectified Linear Unit (ReLU) layer. [Online]. Available: https://se.mathworks.com/help/nnet/ref/relulayer.html

[19] MathWorks. (2017, April). Create max pooling layer. [Online]. Available: https://se.mathworks.com/help/nnet/ref/maxpooling2dlayer.html

[20] MathWorks. (2017, April). Pattern Recognition and Classification. [Online]. Available: https://se.mathworks.com/help/nnet/pattern-recognition-and-classification.html

[21] L. Lorenzetti. (2017, April). From Cancer to Consumer Tech: A Look Inside IBM's Watson Health Strategy. [Online]. Available: http://fortune.com/ibm-watson-health-business-strategy/

[22] I. W. Health. (2017, April). Jupiter Medical Center Adopts Watson for Oncology - Watson Health Perspectives. [Online]. Available: https://www.ibm.com/blogs/watson-health/jupiter-wfo/