Universiteit Leiden
Opleiding Informatica

Gene Prediction Using Unsupervised Deep Networks

Name: Sevastakis Dimitrios

Date: 15/06/2017

1st supervisor: Dr. E. M. Bakker
2nd supervisor: Dr. M. Lew

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Abstract

Gene finding is a well studied topic in bioinformatics. Supervised methods have shown great success in accurately annotating DNA sequences, of which Hidden Markov Models have been the most widely used approach, both in supervised and unsupervised settings. In this thesis we explore the possibilities of using unsupervised neural networks for identifying signals in DNA sequences. First, it is shown that Convolutional Neural Networks are capable of classifying DNA sequences with high specificity and reasonably high sensitivity. Furthermore, we show that features extracted using Convolutional AutoEncoders do not separate the data enough for either unsupervised or supervised learning. Self-Orginizing Maps, though, are able to create some significant visual separation. To the extend of our knowledge this is the first proposal for an unsupervised signal sensor for gene prediction, and the first time Convolutional AutoEncoders have been used for features extraction on DNA sequences.

1 Introduction

Gene finding is the process of identifying gene coding regions in DNA sequences. In disciplines such as structural genomics, functional genomics, where the objectives are to predict the 3D structure of gene products, to identify the function and/or interaction of certain genes, knowing the exact location of gene coding regions in DNA/RNA sequences is required. In the early days this was a slow process that involved experimenting with live organisms in labs. Fortunately, due to our increasing understanding of DNA functions and with the advancements of bioinformatics tools, the use of statistical analysis and the, nowadays, available computational resources, much progress has been made.

Many approaches for gene finding already exist which rely on statistical analysis, pattern recognition and signal processing, as well as homology based methods using information from already well studied organisms.

In this thesis we investigate the possibilities of using Convolutional Neural Networks as signal sensors for gene prediction, as well as using Unsupervised Neural Networks for feature extraction and clustering. The goal is to identify the signals that prepend coding regions, that signify the start of the coding. Raw DNA sequence windows are encoded into one-hot encoding vectors, from which features are extracted using Convolutional AutoEncoders (CAE). Self-Organizing Maps (SOM) are then trained using the CAE feature representation of the sequences.

To the extend of our knowledge this is the first proposal for a completely unsupervised signal sensor method for gene prediction. For this reason we cannot compare our method with any other, and we used random classification as our baseline. Additionally this is the first work that uses Convolutional Neural Networks for features extraction of DNA sequences.

We show that even though the SOM visualizations show some separation between the windows that contain the aforementioned signal, and windows that don't contain such signal, the difference is not big enough for a clustering algorithm or a supervised method to achieve any significant results.

Additionally we investigate the possibility of existence of the seven clusters structure [38] using the CAE features. We show that no indication of such structure is present.

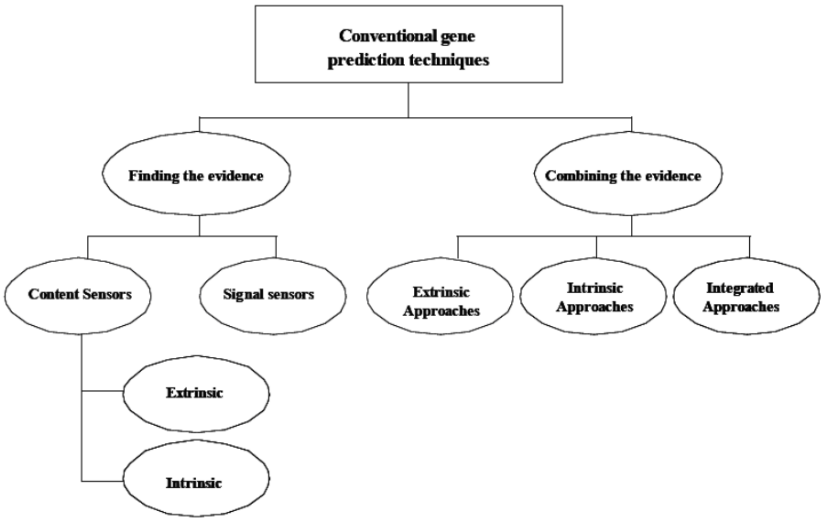

Figure 1: Categorization of gene finding approaches as given by Mathe et al. [3] [4]

2 Related Work

Gene finding is an easier task when working with prokaryotic sequences (see Section 3), due to the high density of genes in the genome, as opposed to the high sparsity of genes in eukaryotic genomes. The main challenge for prokaryotic gene finding is identifying the overlapping genes. In eukaryotic DNA sequences, coding regions can be very hard to identify due to the fact that some may be very short, followed by long introns (non-coding regions). Additionally, alternative splicing makes the process of identifying genes even more challenging.

Generally, the methods for gene finding are divided into two categories, *i) extrinsic methods*, also known as empirical, comparative, similarity or homology based methods and *ii)* intrinsic or *ab initio* methods. Empirical methods, using local alignment methods ([6],[7],[10]), compare sequences with known genes from a database and, given the degree of similarity, try to classify regions as coding or non-coding for prokaryotic DNA and introns, exons, splice sites, for eukaryotic etc. Comparing closely related species DNA for finding similar regions also falls under the empirical category. Most state of the art software for gene finding use a combination of all three methods to achieve the best possible result.

Ab initio methods, on the other hand, rely on predictive models, statistical features and/or signal properties to make predictions for the location of genes and even their structures. Because of the uncertainty of results that always comes with predictive models, gene finding using ab initio methods is more appropriately called gene prediction. Ab initio methods can be further divided into signal sensors or content sensors [8]. Signal sensors detect "signals" in the DNA sequence, such as splice sites, whereas content sensors use statistics, such as codon distributions or nucleotide distribution, to make predictions about sequences and/or to compare with other known genes.

Ab initio methods often yield lower specificity than comparative methods, while comparative methods suffer from weaker sensitivity. For this reason many methods combine the two in order to maximize predictive performance [9].

To the extend of our knowledge the work of Nguyen et al. [28], is the only example of Deep Neural Networks being applied to DNA sequences for classification and gene prediction.

2.1 Supervised methods

Of the ab initio methods, Hidden Markov Models (HMM) have been proven to be able to accurately predict gene position in prokaryotic organisms as well as whole gene structure in eukaryotes. GeneMark.hmm [11], HMMGene [12], Veil system [13], AUGUSTUS [14], SNAP [15] are among the many software that are available that are HMM based for gene prediction or/and gene structure prediction. Genie [16] uses a combination of a HMM model along with 2 neural network classifiers which use dinucleotide frequencies for splice site detection. Most of the HMM based approaches are considered to be content sensors, since they rely on the probability distribution of the bases in a sequence.

GRAIL-I [17], GRAIL-II [17] and GeneParser [19] were among the earliest attempts to use Neural Networks in order to find protein genes in DNA. The first iteration of grail used statistical features from a 99-base windows in order to predict whether that windows was coding or non-coding. Due to the small windows, it was prone to missing a large number of short exons. GRAIL-II was an improvement over the first iteration of the system, which solved the problem of short exons identification by introducing variable length window. Additionally it included additional features, such as transcription promoter recognition, PolyA site recognition, protein coding region prediction, gene model construction, translation to protein and DNA/protein database searching capabilities. For the gene prediction process, first a rule based method is applied to discard most of the improbable candidate sequences. Then three neural networks are used to evaluate the remaining candidates. The sequences that pass this screening, are clustered into three groups. From each group the top candidates are picked as being likely to be exons.

GeneParser uses a method similar to GRAIL, but combines it with a Dynamic Programming (DP) algorithm. Similar in spirit with Dynamic Programming for speech segmentation [21], DNA and RNA sequences can be considered as languages with different grammatical constraints and the benefit of having discrete values in the sequence. By having the likelihood of a segment of belonging to one or more classes, using DP the maximum likelihood solution while enforcing the grammatical constraints.

GlimmerM [37] is a HMM based system which uses Dynamic Programming techniques to efficiently search though all the possible combinations of exons for inclusion in the gene model. To select the best of all these combinations Interpolated Markov Models (IMM) are used, trained on complete coding regions. For the splice site prediction, Maximal Dependence Decomposition (MDD), and first order Markov Chains are used.

Li et al. [20] propose the usage of a small neural network for predicting protein coding genes in yeast genome. The neural network takes a 12-dimensional input vector, which contains the frequency of each amino acid in one of three position in each codon in the reading frame. Since we have 4 amino acids and 3 possible positions a codon (triplet of amino acids), we end up with a 12 dimensional feature vector. For the single hidden layer of the network they used 21 units, while for the output they used a single unit, with intended outputs 0 when a input vector does not represent a coding region and 1 when the input does represent a coding region.

Chicco et al. [22] proposed a system for gene annotation, where either an AutoEncoder (AE) Neural Network or Singular Value Decomposition is used to get a real value noisy reconstruction of binary feature vector. If the values of the reconstruction are above a predefined threshold, it is an indication that the future is present, of absent if the value is bellow the threshold. This method helps identify features that may be missed by other methods, or indicate false

positives in other cases.

Bayesian Networks, in combination with Hidden Markov Models, have been used for combining gene predictions from multiple systems, replacing the naive majority voting scheme [23] and showing significant improvement over individual systems or combination through majority voting.

Evolved Neural Networks [32], which are neural networks which use evolutionary computations for the training of the network, were used for identifying abstract-functional RNAs (fRNAs) genes in *H. Sapiens* and *C. Elegans* [33]. fRNA genes can be harder to identify as they may have short coding regions and do not use similar signals as common protein coding genes.

TigrScan and GlimmerHMM [34] are two more HMM based frameworks for gene prediction for eukaryotic organisms. They both have modular and extensible architectures and are open source. Their main differences are that while TigrScan uses weight matrices and Markov Chains, GlimmerHMM additionally utilizes splice site models adapted from GeneSplicer [35], a system for splice site detection for various eukaryotic organism, and decision trees adapted from GlimmerM.

BRAKER1 [36] is a pipeline framework which uses the semi-supervised version of GeneMark, GeneMark-ET, for making gene predictions. RNA-Seq information are used in the process, extracting spliced alignment information. GeneMark-ET uses the genome data along with the RNA-seq extracted data for unsupervised training. The predicted genes are then used by AUGUSTUS to further create refined gene models, again using RNA-Seq as extrinsic evidence during the training.

Though Convolutional Neural Networks have been used for DNA sequence related tasks, such as annotating gene expressions [25][26] and gene structure prediction [27], there is little research on the ability of CNNs for gene prediction and annotation.

Nguyen et al. [28] used Convolutional Neural Networks for binary classification of DNA sequences. They used as input a one hot encoding, which was inspired from text classification methods, of sequences 500 nucleotides long, and they labeled each window as 0 or 1 depending on the classification task. They trained networks for identifying splice cite junctions, promoter sequences, and parts of the DNA that wrap around histone proteins. Using this method they achieved increase of 1%-6% compared to the state of the art, reaching accuracy of 99.06% for promoter recognition and 95.87% for splice site recognition.

In this thesis we also experiment with Convolutional Neural Networks for DNA sequence classification. The main differences with Nguyen et al.'s work are in that we employ a different kind of one hot encoding scheme, and use a smaller window size of 200 nucleotides. We achieve sensitivity of 84.41%, specificity of 98.75%, f1 score of 0.8432, and accuracy of 91.58% on the genome of *E. Coli* on our best setup.

2.2 Unsupervised methods

The methods mentioned so far, all fall under the category of supervised or semi-supervised learning, since to create the model for the predictions a labeled dataset is needed. The methods that follow are considered unsupervised, since they only use unlabeled data to make predictions. Little work has been done in this area, and most of it has been focused on self training of HMM models.

Gorban et al. [38][39] explained and visualized why unsupervised methods are effective in gene prediction. They showed that the distribution of 64-dimensional vectors of triplet frequencies (codons) shows that existence of seven distinct clusters. The existence of such structure was

already implicitly known, but formally studied.

Mahony et al. [24] (RescueNet) used Self-Organizing Maps with relative synonymous codon usage (RSCU) encoding. This method, though unable to yield state of the art results, it is able to identify multiple gene models within a genome and identify some genes that other methods miss. Additionally it shows the potential of SOM networks for gene prediction.

GeneMarkS [40] is a HMM based approach, similar to the original GeneMark. It uses an iterative training process over unlabeled data to train the HMM on prokaryotic sequences. This method was later improved with GeneMark.hmm ES-3.0 [41] [42] (E-Eukaryotic, S-Self training, 3.0-version number) which was used on novel eukaryotic genomes, yielding state of the art results comparable or better than other supervised methods. GeneMark-ES is able to identify exons with 88.9% sensitivity and 89.2% specificity in the fungal genome of *S.Pombe*.

Chan et al. [29] proposed a pipeline framework, where multiple HMM models are self trained for any given organism. The framework includes data set preparation, parameter estimation for the specific genome, training of HMM based models using existing frameworks, namely GlimmerHMM, AUGUSTUS, SNAP and MAKER2 [30], and an additional step for result filtering, resulting in correct annotation of at least 95% of BUSCOs plantae dataset [31].

In our method we approach unsupervised learning from a signal sensor direction, where we try to identify signals in the genome that signify the start of a protein coding region. By taking sequences 200 nucleotides long, we encode them in an array of one-hot vectors (Section 5.1) and train a Convolutional AutoEncoder (CAE) to reconstruct those sequences. We convert the encoded representation of the CAE to the Bag of Surrogate Parts (BoSP, Section ??) which is used as a feature vector for each sequence. These feature vectors are then used to train a Self-Organizing Map. We show that, even though in visualizations there is separation between sequences that contain the the signal and sequences that do not, there is also a significant overlap between them. Results from clustering the feature vectors are inconsistent and yield an average sensitivity of 70% and specificity of 51%

3 Molecular Biology Background

DNA is a molecule that carries most of the information needed by organisms in order to grow, develop, function and reproduce. A single molecule consists of two biopolymer strands coiled together, forming a double helix. Each strand consists of a sequence of nucleotides, which each consist of a nucleobase, a sugar called deoxyribose and a phosphate group. In DNA the nucleobases can be one of the following: *i) Adenine (A)*, *ii) Thymine (T)*, *iii) Guanine (G)* and *iv) Cytosine (C)*. More in depth information on the chemical structure of DNA can be found in [1].

Similarly, RNA is also a molecule, which plays various roles in processes such as coding, decoding regulation and expression of genes. RNA has a very similar structure to DNA, with three main differences: *i)* RNA is typically a single stranded polynucleotide, *ii)* uses *Uracil (U)* instead of Thymine (A,U,G,C) and *iii)* it has ribose instead of deoxyribose. DNA and RNA both belong to the category of nucleic acids.

As mentioned above, DNA stores biological information, where the information is encoded using sequences of the 4 nucleotides. This information is transcribed into RNA and translated into amino acid sequences, i.e., proteins, which play an essential role in organisms. Regions where such information are encoded are called *genes*. During protein synthesis, information is read from the gene by reading the nucleotides in triplets called *codons*, which represent genetic

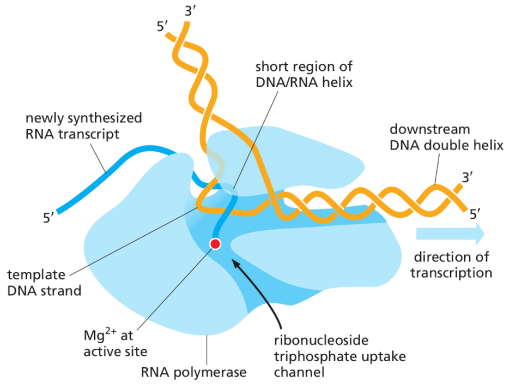

Figure 2: Image showing the transcription process. [1]

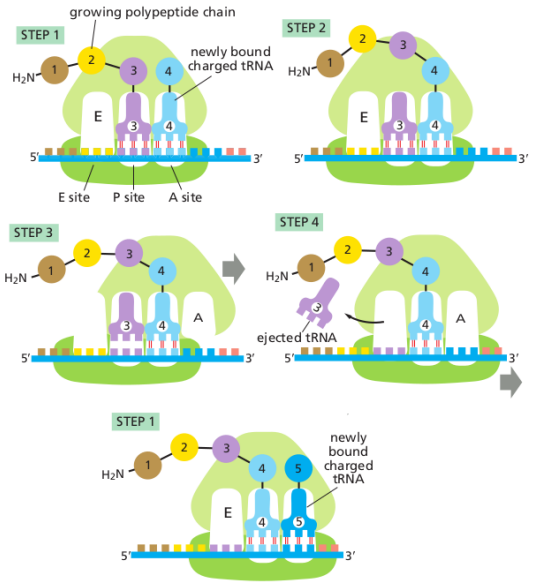

Figure 3: Image showing the translation process [1].

words which specify which *amino acid* should be added next to create the specific protein which this gene encodes. Codons are triplets of nucleobases, of which we have 4 different types. Hence, for each type of nuclei acid, we can have a total of $4^3 = 64$ combinations. This would mean that codons have the ability to code for 64 distinct amino acids, but only 20 amino acids are actually used in protein synthesis. As a result multiple codons can code for the same amino acid, creating redundancy, and thus making the protein synthesis process more tolerant to transcription errors, mutations, etc.

In prokaryotic DNA genes are densely populated and may take up to 90% of the whole sequence (*E. Coli*), while in eukaryotic DNA genes are usually sparse, and could make up as little as 3% of the whole DNA sequence (*Homo Sapiens*). Moreover, genes differ structurally between eukaryotic and prokaryotic organisms. Bellow, a short description will be given for the basic building blocks of genes for prokaryotes and eukaryotes separately, and the basic differences will be highlighted.

Protein synthesis can be viewed as two separate steps, *transcription* and *translation*. During transcription, in eukaryotes, RNA is produced inside the cell nucleus. The hydrogen bond, that holds the DNA double helix together, is broken around the gene to be transcribed, exposing the strands (Figure 2). The strand where the gene is located is then used as a template for the synthesis of the RNA molecule, where Uracil binds across Adenine, Adenine across Thymine, Guanine across Cytosine and vice versa. When the RNA synthesis has reached the end of the gene, it detaches from the DNA strand and leaves the cell nucleus and heads to the ribosome to undergo the translation process (Figure 3). It is worth noting at this point that the RNA product of transcription is different for prokaryotes and eukaryotes. For prokaryotes the product is messenger RNA (mRNA) which needs no further processing, while in eukaryotes the first product is called primary transcript. The primary transcript goes through post-transcript modification which produces heterogeneous nuclear RNA (hnRNA) which, in turn, undergoes splicing of introns, which strips away introns (non-coding regions inside the gene, which are not present in prokaryotes).

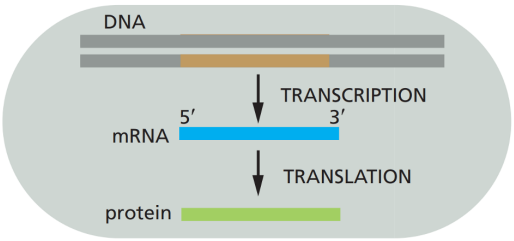

Figure 4: Image showing the transcription process in prokaryotic organisms [1]. The highlighted area in the DNA strand indicates the position of the gene.

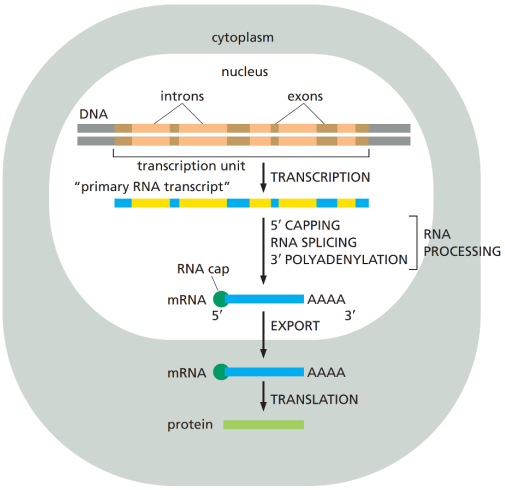

Figure 5: Image showing the translation process in eukaryotic organisms [1]. In this example the gene contains 5 introns, which are the non coding parts of the gene. They are included in the primary RNA transcript as shown below, and are stripped away during splicing in order to produce the final mRNA.

3.1 Prokaryotic DNA structure

In prokaryotes the DNA structure is fairly straight forward, and can be broken down to the following parts.

Regulatory sequence: All genes contain a regulatory sequence, which is necessary for the gene to be expressed (be used for protein synthesis). They usually are made of 2 parts, *Promoter* and *Enhancer/Silencer*. Promoters are needed by genes in order to be expressed. mRNA binds to the promoter site in order to start the transcription process. Enhancers and Silencers can be found kilobases (10^3 nucleobases) upstream or downstream of the *Open Reading Frame* (ORF) [1][2] and regulate the expression of the gene. Enhancers increase the likelihood of transcription of the gene by binding activator proteins which help the mRNA bind to the promoter. On the other hand Silencers reduce the likelihood of transcription of a gene by binding to repressor proteins making it harder for the mRNA to bind to the promoter. By increasing or decreasing the likelihood of transcriptions, cells regulate the quantity of proteins that are produced.

Start codon: Is the codon that marks the starting point for translation of the mRNA during the translation process. The most common start codon is *AUG*.

Coding sequence: The coding sequence is the portion of the gene that encodes the information for building a specific protein. In prokaryotes this is a continuous - uninterrupted sequence, until a stop codon is found.

Stop codon: Signals the end of translation of the mRNA (usually *UAA*, *UAG* or *UGA*).

[1]Reading Frame is a sequence of non-overlapping triplets (codons). Thus every strand of DNA can be interpreted in 3 different ways depending the starting point of the Reading Frame, and including the complementary strand we get a total of 6 ways to read the DNA.

[2]Open Reading Frame is part of the Reading Frame that has the potential to code for a protein

It is often the case that ORFs are grouped together into a single *operon* [1]. Operons can contain multiple codings for different proteins, which usually serve related functions, after a single promoter. All protein templates within one operon will be transcribed by a single mRNA.

3.2 Eukaryotic DNA structure

Eukaryotes share some similarities with prokaryotes, but have additional features that make up for a more complex structure. The similarities and differences are highlighted bellow. *Regulatory sequence*: Regulatory sequences in eukaryotes share similar structure as in prokaryotes, though promoter regions have more complex structure and therefore are more difficult to identify in eukaryotes.

Start codon: Equal to *AUG*, as for prokaryotes.

Coding sequence: As in prokaryotes, this is the part of the gene that encodes the information for building a protein. But, unlike prokaryotes, in many cases it is not a continues sequence, but rather, it is interrupted by non-coding, untranslated segments called intergenic regions or simply *introns*. The coding parts of of the ORF are called expressed regions or *exons*. Introns are transcribed along with exons, but are, as mentioned earlier, removed from the primary transcript during the process called splicing, and exons are joined together.

Stop codon: *UAA*, *UAG* or *UGA*, as for prokaryotes.

The structure of eukaryotic genes is more complex, as one gene may code for more than one protein using *alternative splicing*. When alternative splicing occurs, in the most usual case, the exons of a gene may simply be spliced out of the primary transcript and will not be part of the final mRNA that undergoes translation, while in another case the same exons may be included. This is the most common version of alternative splicing, called *exon skipping*, but there are other modes, namely *mutually exclusive exons*, *alternative donor site*, *alternative acceptor site*, and *intron retention*.

4 Deep Learning Background

Machine learning is a subfield of computer science where computers create models that describe sets of data, which can be used in order to make predictions on new data. This can be done by creating either statistical models or mathematical models. In the case of statistical models, a model tries to find statistical patterns between data and labels, whereas mathematical models try to find values for variables such that they minimize or maximize a cost or gain function.

Machine learning itself is divided into two main categories, *supervised learning* and *unsupervised learning*. In supervised learning tasks are mainly classification and regression tasks, where models are trained to classify items between N classes, or to predict a real value, based on datasets where labels are available. One of the most popular datasets is the MNIST dataset for handwritten digit classification [45], which is a database of 60.000 images of handwritten digits, each labeled with the value the image represents. This dataset is often used as a benchmark for researchers to test their algorithms.

Another example is the House Prices dataset [44] which contains data about houses (e.g. surface area, number of rooms, whether the house has central heating, air conditioning etc) and can be used to regression models which can predict prices for other houses.

The main problem of supervised training is that in order to create complex and accurate models, there is a need of big datasets which need to be annotated. In recent years, there has

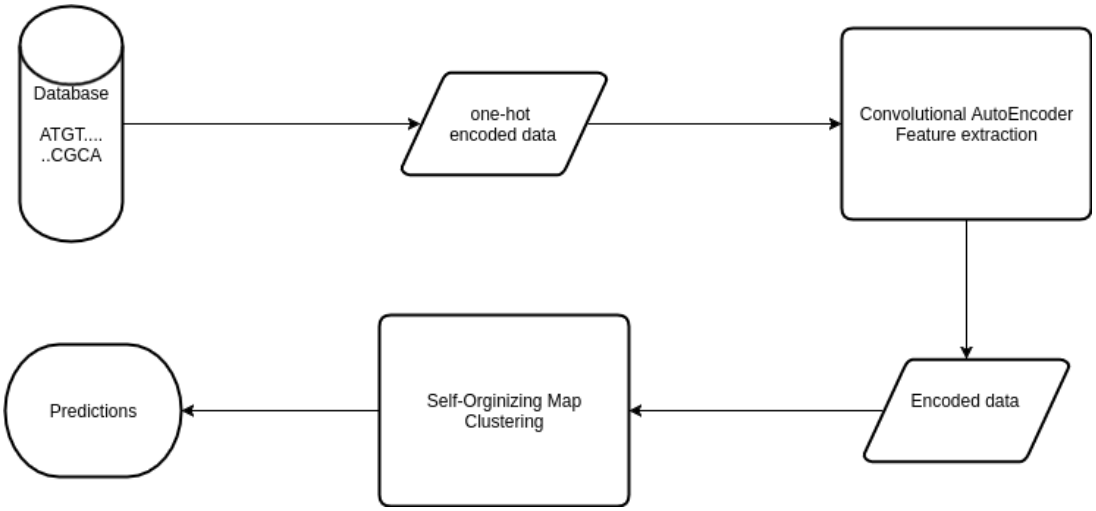

Figure 6: Data Flow Diagram of the end goal of this project.

been significant increase in the quantity and quality of available datasets for computer vision tasks, speech recognition and others.

Unsupervised training on the other hand is used for clustering data, or to estimate values for latent variables of an underling distribution of the data. Philosophical discussions can be had on whether a system is ever unsupervised, since there is always some kind of information/parameters given to the system (e.g. number of clusters).

In unsupervised learning one of the biggest challenges is feature extraction and feature selection. It is important to create discriminative features for the data, which will separate them in the desirable manner. In many cases many redundant features are created from a single method, from which only a few are selected in order to remove features that only add noise to the data, and also to make the training process more efficient.

Semi-supervised learning sits somewhere in between supervised and unsupervised learning, where the model tries to learn a specific task from sparse data.

The end goal of this thesis is to have a completely unsupervised method for identifying signals in DNA sequences.

Withing the machine learning field one family of models has gained a lot of popularity over the years and is called *Artificial Neural Networks* (ANN or simply NN).

In this project the goal is to make use of Unsupervised Neural Networks in order to cluster DNA sequences in a way such that signals that signify the start of a coding region are grouped separately from coding and non-coding regions. A data flow diagram of the system can be seen in Figure 6. Raw nucleotide sequences are encoded as vectors of one-hot vectors (Section 5.1) which are used to train a Convolutional AutoEncoder (Section 4.2.5). The dataset is then converted into an encoded feature representation. This representation is fed to a Self-Organizing Map (Section 4.2.6) in order to visualize and cluster the sequences.

4.1 Supervised Neural Networks

4.1.1 Perceptron

Artificial neural networks, inspired by the biological neural networks, are networks of interconnected *neurons* which try to estimate or approximate a function and are usually used in

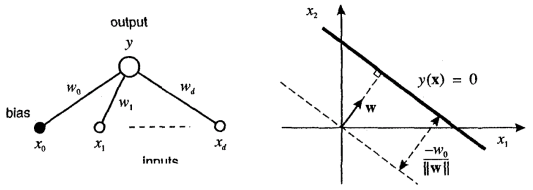

Figure 7: Left: A single perceptron with d inputs and the associated weights.Right: A 2-dimensional representation of a plane, and how the weights affect it.

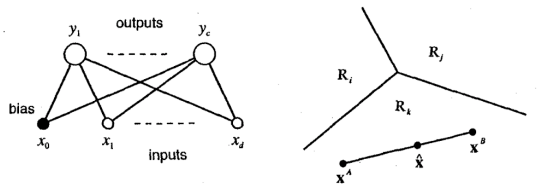

Figure 8: Left: A network of 2 perceptron sharing an input vector. Right: Example of decision boundaries produced by a multi-class linear discriminant.

a supervised learning setting. For the simplest form of NNs, the single layer perceptron [43], neurons are simple processing nodes that take input X of length N and produce 1 output, which is a linear combination of the inputs. The output of the neuron has the following form:

$$y = w_0 + x_1 * w_1 + \cdots + x_N * w_n = w_0 + \sum_{i=1}^{N} x_i * w_i = w_0 + \boldsymbol{w}^T * \boldsymbol{x} \tag{1}$$

Where x_i is the i^{th} input of the neuron, w_i is a weight that is associated with that input and w_0 is called the bias. For more convenience usually equation 1 takes the following form

$$y = \sum_{i=0}^{N} x_i * w_i = \boldsymbol{w}^T * \boldsymbol{x}, \quad x_0 = 1 \tag{2}$$

It is easy to see that this function represents a hyperplane

$$a * x + b * y + c * z + \cdots = 0 \tag{3}$$

in which case $x_i,\ w_i$ are the independent variable and the slope in dimension i respectively, and w_0 is the y intercept of the plane. The N-dimensional hyperplane is a linear discriminant function, which creates two regions.

We can "learn" the parameters of the hyperplane using a training set, in order to move the hyperplane in a position where it optimally separates two classes of our data.

The perceptron is just a single node NN with a non-linear activation function

$$a = \sum_{i=0}^{N} x_i * w_i, \quad g(a) = \begin{cases} g(a) = +1 & \text{if } a \geq 0 \\ g(a) = -1 & \text{if } b < 0 \end{cases} \tag{4}$$

A network of c perceptrons which share the same input vector, but each one has its own set of weights, creates c discriminant functions, or c convex regions, which can be used for multi-class problems.

The algorithm for learning the weights of the network is called $Gallant's\ pocket\ algorithm$ and is shown in algorithm 1.

The core of the whole algorithm is the weight update rule, shown in equation 5:

$$w_{new} = w_{old} + \eta dx \tag{5}$$

Algorithm 1 Perceptron training algorithm

```
1: weight_matrix := random()
2: pocket := weight_matrix
3: best_run := 0  current_run = 0
4: while current_run < num_examples do
5:    example := Select_random_example()
6:    if example is misclassified then
7:       apply_weight_update_rule()
8:    else
9:       current_run++
10:      if current_run > best_run then
11:         best_run = current_run
12:         pocket = weight_matrix
13:      end if
14:   end if
15: end while
```

where η is a predefined learning rate (usually around 0.1) d is the label of the example, and x is the input. The motivation behind this is that if x is misclassified and the target $d = 1$, then the wx should be bigger. Alternatively, if x is misclassified and $d = -1$, wx should be smaller.

Gallant's algorithm is proven to converge, as long as the data are linearly separable, meaning that there exists a hyperplane such that it perfectly separates the two classes. A simple example of a problem that cannot be solved by a linear discriminant, and thus is not linearly separable, is the XOR problem.

In order to solve more complex problems, layered network are required, which are able to create convex regions whose edges are used as the decision boundaries.

4.1.2 Multilayer Perceptron

The most common type of Neural Network, for a long time, was the *multilayer perceptron* (MLP). The structure of the MLP is very similar to the single layer perceptron, but instead on connecting the input directly to the output nodes, we can add an arbitrary number of "hidden" layers, each with its own set of neurons and weights. MLP is the most common example of *feedforward* NN, because the connections do not form circles, thus information only moves only in one direction.

We saw earlier the algorithm for training a single layer perceptron, but the question now would be how can we find the appropriate values for the weights in layered networks. The first step to the answer is to replace the simple sign function of the perceptron with a differentiable smooth sigmoid function. Popular for this task are the *logistic sigmoid* (Equation 6)and the *Tanh* (Equation 7) functions.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{7}$$

We define an error function for our network, which we will try to minimize. For classification tasks the most common error function used is the *mean square error*.

$$E = \frac{1}{2} \sum_{outputs} (y_i - d_i)^2 \tag{8}$$

By writing out the error function as a function of the network weights, we can then calculate the partial derivatives of the output nodes, which will give us the direction in which the error decreases the fastest. We can use this to adjust the weights in order to minimize the error.

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{9}$$

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{10}$$

where η is the predefined learning rate. This is the gradient descent algorithm.

In order to calculate the partial derivative $\frac{\partial E}{\partial w_{ij}}$ we can expand it using the chain rule (Equation 11)

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial x_j} \frac{\partial x_j}{\partial w_{ij}} \tag{11}$$

where x_j is the input of node j, which is the weighted sum of the outputs of the nodes in the previous layer, and y_j is the output of node j, thus $y_j = g(x_j)$, where $g()$ is an arbitrary activation function. This means that we need three things in order to complete the update rule for each weight, the partial derivative of the error function w.r.t. the output of each neuron (which is frequently called the error message or delta), the partial derivative of the output of each neuron w.r.t. its input, and the partial derivative of the input w.r.t. each corresponding weight.

The second term, the derivative of the output w.r.t. the input, is simply just the derivative of the activation function, which is show in Equation 12

$$\begin{gathered} \frac{\partial y_j}{\partial x_j} = \frac{\partial g(x_j)}{\partial x_j} \\ \text{When: } g(x) = \frac{1}{1 - e^{-x}}, \quad \frac{g(x_j)}{x_j} = g(x_j)(1 - g(x_j)) \end{gathered} \tag{12}$$

where, again, $g()$ denotes any activation function. In the example above the logistic activation function is used.

The last term, the partial derivative of the input w.r.t. a specific weight, is simply the input j of the neuron, as shown in equation 13.

$$\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial (\sum_{k=1}^{n} w_{kj} * y_k)}{\partial w_{ij}} = y_i \tag{13}$$

where y_i is the output of node i in the previous layer, because only one term in the sum, y_k when $k = i$, is depended on w_{ij}.

For the first term, the partial derivative of the error function w.r.t. the output of a specific neuron, it is more straight forward if the neuron is in the output layer, as seen in equation

$$\frac{\partial E}{\partial y_j} = \delta_j = \frac{\partial}{\partial y_j}\frac{1}{2}(y-t)^2 = y_j - t_j, \quad \text{when: } E = \frac{1}{2}(y-t)^2 \tag{14}$$

For neurons in an arbitrary hidden layer, things get a bit more complicated, as seen in equation 15.

$$\frac{\partial E}{\partial y_j} = \delta_j = \frac{\partial E(x_u, x_v, \ldots, x_w)}{\partial y_j} = \sum_{l \in L}\left(\frac{\partial E}{\partial x_l}\frac{\partial x_l}{\partial y_j}\right) = \sum_{l \in L} \delta_l w_{jl} \tag{15}$$

where L denotes the layer above, w_{jl} is the weight from neuron j to neuron l in the layer layer above. This is, thus, a recursive function which we can use to calculate the derivative for y_j is we know all the derivatives with respect to y_l on the layer above. So we are basically back-propagating the error, weighted by the weights of the connections between neurons.

The update rule can be summarized as

$$\begin{aligned} &\Delta w_{ji} = \eta \frac{\partial E}{\partial w_{ij}} y_i, \quad where: \\ &\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = \begin{cases} (y_j - t_j) y_j (1-y_j) y_i & \text{if j is output node} \\ (\sum_{l \in L} \delta_l w_{jl}) y_j (1-y_j) y_i & \text{if j is hidden node} \end{cases} \end{aligned} \tag{16}$$

where y_j is the output of the j^{th} node in the current layer, and y_i is the output of the i^{th} node in the previous layer. In this derivation is is assumed that the square error function is used and a logistic activation function for the neurons.

In short, the steps the steps are:

i. apply input vector x and forward propagate through the network to produce outputs for all nodes using Equation 2

ii. compute the deltas for output nodes using Equation 16

iii. back-propagate the deltas and compute the deltas and gradients of the hidden units, also shown in Equation 16

iv. update the weights by making a small step towards the steepest descent.

We repeat this process for all examples in the training set, for a pre-specified number of *epochs* (iterations over the whole train set) , or the error over all examples converges.

There are no guarantees that a neural network will converge, since there are many factors to consider when training a neural network, all of which affect the behavior of the network. Such factors are the number of layers, the number of nodes per layer, the learning rate, number of training examples, cost function, activation functions, etc. There are some rules of thumb for choosing these values, but in practice a lot of parameter tuning has to be made in order to get optimum results.

4.1.3 Convolutional Neural Networks

Convolutional NN (CNN) [45] are another type of feedforward NNs, which were originally designed to be used with image data, but recently they have been used very successfully in

Algorithm 2 Feed Forward Neural Network training algorithm

```
1: weight_matrix := random()
2: epoch := 0 , error_dif := max_float , epoch_error := 0 , previous_error := max_float
3: while epoch < max_epochs and error_dif > converge_threshold do
4:    previous_error := epoch_error
5:    epoch_error := 0
6:    for i:=0  i<dataset_length  i++ do
7:       for j:=0  j<num_layers  j++ do
8:          for k:=0  k<num_layers  k++ do
9:             activation[j][k] := compute node activation using Equation 4
10:         end for
11:      end for
12:      output_error := sum of errors of output nodes
13:      for j:=num_layers-1  j>-1  j- - do
14:         for k:=0  k<num_layers  k++ do
15:            gradient[j][k] := compute node gradient using Equation 16
16:            weight_matrix[j][k] := weight_matrix[j][k] + delta[j][k]
17:         end for
18:      end for
19:      epoch_error += output_error
20:      error_dif := abs(previous_error - epoch_error)
21:   end for
22: end while
```

other fields, such as recommender systems [46], speech recognition [48], and natural language processing [49].

Traditional MLPs suffer from the curse of dimensionality. Let us consider a simple netowrk that takes as input a small image of size 32x32 and has 3 color channels. For such network, a node in the first hidden layer would require $32\text{x}32\text{x}3 = 3072$ weights. By increasing the size of the image to 200x200, which by today's standards is also considered small, each neuron in the hidden layer would require $200\text{x}200\text{x}3 = 120000$, which is two orders of magnitude higher. Additionally, MLPs do not take into consideration spacial structure of the data, meaning that it treats data that are far away the same way as if they were near by.

CNNs, on the other hand, try to take advantage of the strong local correlation of pixels, that is usually found in natural pictures while requiring fewer weights. They achieve that by having two main differences from traditional MLPs, *sparse connectivity* and *weight sharing*.

Typical CNNs consist of 4 basic building blocks

i) Convolutional layers

ii) Pooling layers

iii) ReLU (Rectified Linear Unit) Activation Function

iv) Fully connected layers (classifier)

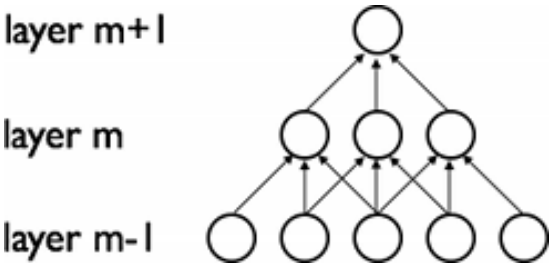

Figure 9: Example of connectivity of nodes in convolutional layer

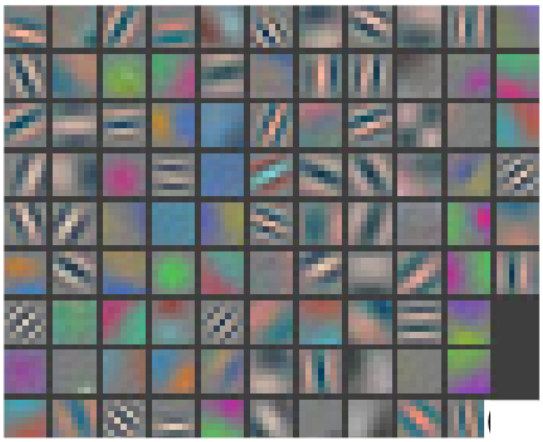

Figure 10: Example of features learned by the first convolutional layer in network trained on image data [47].

4.1.3.1 Convolutional Layers

In CNNs the neurons in a convolutional layer are connected only to a small subset of neurons of the previous layer, which means that a neuron in the convolutional layer encodes information only from a specific region of the input. On the layer above, a neuron will encode information from a group of neurons which encode information from a bigger region of the input, and so on (Figure 9).

It is easier to understand CNNs if we avoid to think in terms of individual neurons, but rather in terms of groups of weights, commonly referred to as $filters$. These filters are the features that the network learns during the training. For image data, the first layer would learn simple lines in different orientations, blobs or gradients (Figure 10). The second layer would then learn features that are combinations of those lower features that would usually be found nearby each other. Ultimately, the filters in the highest layers will be able to recognize features that look like wheels, eyes, legs, et cetera.

Sparse connectivity solves the problem of the curse of dimensionality that traditional feed-forward networks suffer from, but now local features will be learned for specific regions of the input. To solve this problem, the filters are shifted over the input in steps (usually of size 1), using the same weights. In each position the dot product of the weights of a filter and the input at that position is calculated. In a sense, the weights of a filter are shared between neurons at different positions in each step. This allows local features to be detected independently from their position in the input. For each filter in the layer, a pass is made over the whole input, which produces a so called feature map. Feature maps are transformations of the original input specified by the filters. For image data, if the filters detect edge-like features, the feature map will be an image similar the output of a simple edge detector.

It is important to note at this point that the dot product spans through the whole depth of the input. So lets take a closer look at the computations that take place. Assuming a filter of height H_f, width W_f and depth (channels) D_f, the value y_{ijk} of the feature map is the dot product of the input patch of size H_fxW_fxD_f, centered around input at x_{ijk}, and the weights in the filter. The equivalent mathematical expression of this is shown in equation 17

$$y_{i'j'k'} = b_{f'} + \sum_{i=1}^{H_f} \sum_{j=1}^{W_f} \sum_{k=1}^{D_f} x_{i'+i-1,j'+j-1,k'+k-1} w_{ijk} \tag{17}$$

where $b_{f'}$ is the bias term for the filter, $i'j'k'$ are the indexes of the position where the filter is applied on the input and also the indexes of the value in the feature map that is calculated, and w_{ijk} is the weight at position ijk in the filter matrix.

By shifting and applying the filter on the whole input step by step, we produce the complete feature map for a single filter, which is the convolution of the input with that specific filter as a convolution kernel. Convolutional layers usually include multiple filters, whose feature maps become the depth dimension for the convolutional layer above.

$$feature\_map_i = X * F_i \tag{18}$$

where X is the input matrix and F_i is a filter in the convolutional layer.

The weights in the filters are learned, as weights in any MLP, using back-propagation. The only difference here is the way we propagate the deltas in the convolutional layers, since the weights are shared.

First lets take a quick look at the convolution.

$$y = w * x = [y_n], y_n = (w * x)[n] = w^T x_{n:n+|w|-1} \tag{19}$$

Equation 19 tells us that the output y of a convolutional layer is the convolution of input x with the weight matrix w, and y_n is the output at position n.

To reduce the error in a convolutional layer, as in a simple MLP, we want to modify the weights in a way that it reduces the error function, thus once again we return to our trusty equation 11, but because of the convolution, there are some differences. When we introduce the input component x_{ij} we have to include all the expressions where it takes place, thus Equation 11 becomes:

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \frac{\partial x_{ij}^l}{\partial w_{ab}} \tag{20}$$

where N is the size of one dimension of the NxN input (assuming a square image is the input) and m is the the size of the one dimension of the mxm filter.

As with the MLP we have the same components which we have to calculate. Furthermore, similar to a fully connected layer, the second and third component respectively become Equation 21 and Equation 22:

$$\frac{\partial y_{ij}}{\partial x_{ij}^l} = g'(x_{ij}) \tag{21}$$

$$\frac{\partial x_{ij}^l}{\partial w_{ab}} = y_{(i-a)(j-b)}^{l-1} \tag{22}$$

And lastly, the first component:

$$\frac{\partial E}{\partial y_{ij}^l} = \delta_{y_{ij}}^l = \sum_{i'=0}^{m-1}\sum_{j'=0}^{m-1} \frac{\partial E}{\partial x_{(i-i')(j-j')}^{l+1}} \frac{\partial x_{(i-i')(j-j')}^{l+1}}{\partial y_{ij}^l} = \sum_{i'=0}^{m-1}\sum_{j'=0}^{m-1} \frac{\partial E}{\partial x_{(i-i')(j-j')}^{l+1}} w_{ab}$$
$$\frac{\partial E}{\partial y_{ij}^l} = \sum_{i'=0}^{m-1}\sum_{j'=0}^{m-1} \frac{\partial E}{\partial y_{(i-i')(j-j')}^{l+1}} \frac{\partial y_{(i-i')(j-j')}^{l+1}}{\partial x_{(i-i')(j-j')}^{l+1}} w_{ab} = g'(x_{ij}^l) \sum_{i'=0}^{m-1}\sum_{j'=0}^{m-1} \delta_{i'j'}^{l+1} w_{ab} \tag{23}$$

which, intuitively, is the weighted cumulative error message of the next layer w.r.t. each input from the current layer. From the term $\frac{\partial y_{(i-i')(j-j')}^{l+1}}{\partial x_{(i-i')(j-j')}^{l+1}}$, only the term w_{ab} survives when $a = i' - i$ and $b = j' - j$. So we can rewrite the function as

$$\frac{\partial E}{\partial y_{ij}^l} = g'(x_{ij}^l) \sum_{i'=0}^{m-1}\sum_{j'=0}^{m-1} \delta_{i'j'}^{l+1} w_{(i'-i)(j'-j)} = g'(x_{ij}^l)\delta_{ij}^{l+1} * flip180(w)_{ij}^{l+1} \tag{24}$$

By looking at it a little closer, it looks like a convolution of the input of layer $l+1$ and the filter matrix w, but the indexes of w are now $w_{(i-a)(j-b)}$ instead of $w_{(i+i')(j+j')}$. We can flip the weight matrix by 180 degrees and thus make it a convolution :

$$\frac{\partial E}{\partial y_{ij}^l} = \delta_{ij}^l = g'(x_{ij}^l) flip180(w) * \delta^{l+1} \tag{25}$$

Putting it all together again, equation 20 becomes:

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \delta_{ab}^l y_{(i-a)(j-b)}^{l-1} g'(x_{ij}) = \delta_{ab}^l * flip180(y)_{ab}^{l-1} g'(x_{ij}) \tag{26}$$

Finally, we can computed the new weights for by using the standard update rule used for MLP.

Now, if you are thinking "oh, finally we are done", you probably forgot that in the beginning of this section it was mentioned that typical CNNs consist of 4 different types of layers, and we just described one of them. Luckily for me the rest of the layers are easier to describe.

4.1.3.2 Pooling Layers

Pooling layers are just sampling layers that are used to reduce the dimensionality of the data, reducing computational cost. Additionally, by shrinking the input data, we are essentially bringing features that were further away closer together so the network can find relations between them.

There are different kind of pooling that we can do, such as max, min, average and more. We can also adjust the window of the pooling, and also the stride, which will directly affect the size of the output of the layer.

It should be clear that pooling layers do not do any learning themselves, so deriving the back propagation rule is quite simple. For max and min pooling the error is caused by just one of neurons in the previous layers, which would be the one with the highest or lowest activation respectively. In this case we just have to keep track of the neurons whose output is propagated forward through the pooling layer, and propagate backward the deltas only to them.

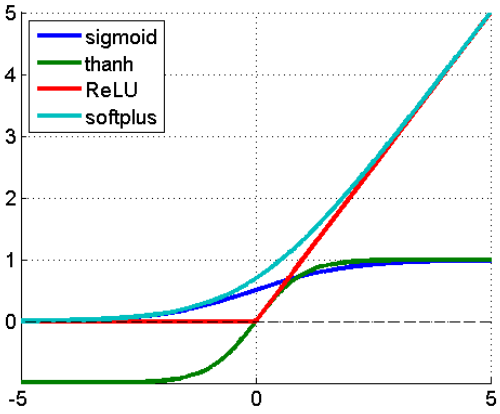

Figure 11: Visualization of the four most common activation functions used in Artificial NNs.

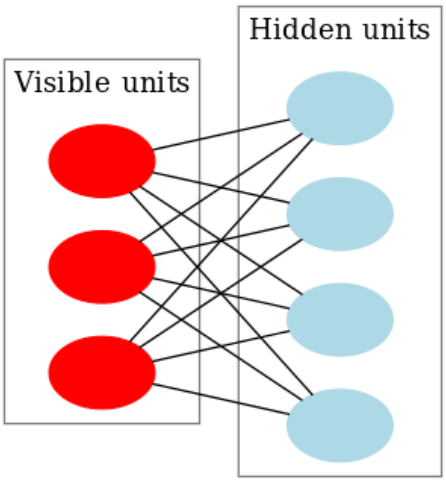

Figure 12: Example of a simple RBM showing the connections between layers, omitting the biases for simplification.

4.1.3.3 Fully Connected Layers

Finally, we use fully connected layers on the top of the network to train our models. In essence, we use the convolutional layers as feature extractors, pooling layers for dimensionality reduction and to make the feature detectors more scale invariant, and we feed the detected features to a MLP for model training and classification.

4.1.3.4 Rectifier

In more recent years, in larger network structures, the Rectifier activation function [50] has become increasingly popular and is preferred to the more traditional *logistic* and *tanh* functions. The Rectifier is defined as

$$g(x) = max(0, x) \tag{27}$$

This type of activation is considered to be more biologically plausible [50] than the probability based activation functions and have been shown to improve various network structures in various tasks [51][52][53].

Units that employ such activation function are commonly referred to as Rectified Linear Units (ReLU). ReLU offer some significant advantages over other activations functions. Firstly, because activations that are smaller than zero become nullified, in a randomly initialized layer, about half of the nodes are expected to have zero activations. This means that the layers learn sparse representations, which make high dimensional more separable [54]. Furthermore, it has been shown [54][53][51] that by using the Rectifier activation function instead of the more traditional ones, eliminates the need for unsupervised pre-training the network and the network converges more quickly.

4.1.4 Cost functions

So far we have seen most of the fundamental building blocks of Neural Networks and Convolutional Neural Networks, but thus far the assumption has been that the mean square error cost function has was used in order to derive the gradients in the final layer. There are a number of cost functions that are more suitable for some tasks than others.

4.1.4.1 Quadratic cost function

Quadratic is the commonly known mean squared error which has already been mentioned, and is also know as syn squared error and maximum likelihood. It is defined as the sum of the squared difference between the target and the prediction over each output (Equation 28), and is most suitable for regression tasks.

$$E_{mse} = \frac{1}{2}\sum_{i=0}^{n-1}(y_i - d_i)^2 \tag{28}$$

where n is the number of output nodes, y_i is the i^{th} output given by the network and d_i is the desired output for node i.

In oder to calculate the gradients for each node in the output layer, we also need to calculate the derivative of the error function with respect to the output of each node.

$$\frac{\partial E_{mse}}{\partial y_i} = y_i - d_i \tag{29}$$

4.1.4.2 Cross-Entropy cost function

Cross entropy is another very popular cost function that is most suitable for classification tasks.

$$E_{ce} = -\sum_{i=0}^{n-1}\left[d_i \ln y_i + (1 - d_i)\ln(1 - y_i)\right] \tag{30}$$

Respectively, the derivative of the cross entropy cost function is computed as follows

$$\frac{\partial E_{ce}}{\partial y_i} = \frac{y_i - d_i}{(1 - y_i)y_i} \tag{31}$$

4.1.4.3 Exponential cost function

The exponential cost function is a tunable cost function which allows you to scale the error experimentally, until the network achieves the desired behavior.

$$E_{exp} = \tau \exp\frac{1}{\tau}\sum_{i}(y_i - d_i)^2 \tag{32}$$

where τ is the tunable parameter.

The derivative of this cost function is shown in Equation 33

$$\frac{\partial E_{exp}}{\partial y_i} = \frac{2}{\tau}(y_i - d_i)E_{exp} \tag{33}$$

4.1.5 Improving the learning process

4.1.5.1 Batch Training

In section 4.1.2 we saw an overview of the stochastic back-propagation algorithm, where we took each example in the training set and applied it to the input of the neural network, did a forward pass, followed by a back propagation of the gradients. Doing so for each example independently is inefficient and suboptimal, as each example's gradient may not be representative of it's class, guiding the network's weights towards the wrong directions, taking longer to converge.

An alternative way of training is using the batch method. Here a predefined batch size k is selected, where a forward pass is done for k examples. Once the batch forward passes are completed the mean error is calculated for the final layer, and it is used to do a single back propagation.

This, obviously, is much more efficient as it requires just a single backwards pass every k examples, and it also helps converge faster, since each batch may contain examples from more than a single class.

4.1.5.2 Weight Decay

Krogh and Hertz [57] proposed Weight Decay as a method to improve model generalization for neural networks during training. They introduced an additional term in the cost function which punishes weights when they grow too large. This effectively forces the network to try and find better solutions instead of exploding some of it's terms, and allowing some weights to grow when it is only necessary.

The the additional term introduced has the following form

$$E(w) = E(w) + \frac{\lambda}{2} \sum_i w_i \tag{34}$$

where E denotes any cost function of our choosing, w is a vector containing all the weights of the network and λ is the term that dictates how much weights are punished, which usually takes values close to 0 (e.g. 0.0005).

When using back propagation to train the network, the update rule for the weights is adjusted accordingly as shown in Equation 35

$$\Delta w_i = \frac{\partial E}{\partial w_i} - \lambda w_i \tag{35}$$

4.1.5.3 Dropout

Dropout [58] is another method for improving model generalization and reduce overfitting. The idea behind this is that in an ideal scenario, multiple networks with different architectures would be trained on different subsets of the training set, and would finally vote for the classification. But given that in many cases there are not enough data to train different networks, finding optimal parameters for different architectures is a tedious task, and the time and resources to train and evaluate many networks is prohibitive in most cases, this approach is not feasible.

By introducing dropout, Srivastava et al. try to simulate this process by using a single network. By having nodes randomly "dropped out" of the network along with all their incoming and

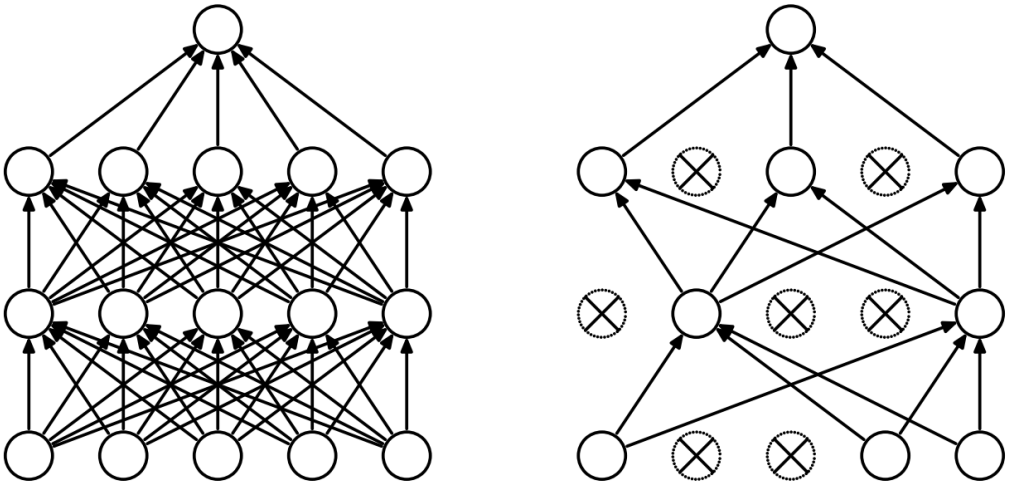

Figure 13: Left: Example of a fully connected network. Right: The same network when dropout is applied.

outgoing connections, with probability 0.5, for a single batch iteration, essentially a thinned network is trained for that specific batch of examples. By choosing different nodes for dropout for each batch in a network of n nodes, there are 2^n possible thinned networks with shared weights that can be trained.

Finally, while dropout simulates the effect of having multiple neural networks for classification, the goal is to have a single neural network during the evaluation process. This means that when making predictions, the weights have to be scaled proportionally to the probability of the dropout.

$$w_{eval} = pw_{dropout} \tag{36}$$

where w is the weight vector for the whole network, w_{eval} is the scaled weight vector to be used for evaluation, $w_{dropout}$ is the weight vector with the weight values that were produced during training, and p is the probability of dropout.

This method proves to be very effective for training both Neural Networks and Restricted Boltzmann Machines.

4.1.5.4 Bag of Surrogate Parts

Bag of Surrogate Parts [55] is a method for extracting features from feature maps of Convolutional Neural Networks. The feature vector is created as follows

$$BoSP = \sum_{i=1}^{n} [P_1^i, P_2^i, P_3^i, \ldots, P_M^i] \tag{37}$$

where n is the number of elements in each feature map, M is the number of feature maps and P is the normalized activations of each feature map

$$P_j^i = \frac{A_j^i}{max(A^i)} \tag{38}$$

This features give us an indication of how much each filter has been activated given a certain input. In order to reduce the noise from the activations and make the features more discriminative, an additional step is taken, reducing the activations of each map to 0 if they are bellow the mean of the activations of that map.

$$P_j^i = \begin{cases} 0 & \text{if } A_i^j < mean(A^i) \\ (\frac{A_j^i}{max(A^i)} & \text{if } A_i^j \geq mean(A^i) \end{cases} \tag{39}$$

4.2 Unsupervised Neural Networks

Unsupervised NNs can be used both for unsupervised learning of features and clustering. Additionally, with the recent need for deeper architectures, the problem of training such networks became more and more challenging. Unsupervised pre-training each layer, as opposed to randomly initializing the weights, has made it it possible [56]. Firstly we are going to take a closer look at two different types of networks which can learn features of the data in an unsupervised manner.

4.2.1 Restricted Bolzmann Machines

Restricted Bolzman Machines (RBM) [59] are types of Artificial Neural Networks that can learn probability distributions over sets of inputs. Traditional RBMs consist of a binary valued visible layer and a binary valued hidden layer. The layers are fully connected and a weight is associated with every edge. Additionally, every node (in the visible as well as in the hidden layer) has a bias weight associated with it. In Figure 11 an example of a simple RBM is shown. Note that there are no intra-layer connections in any layer. Given a network structure with boolean visible units v, boolean hidden units h, weights w between units v and h, bias terms of the visible units a, and bias terms of the hidden units b, we compute the energy function of the network as follows

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j \tag{40}$$

The probability distribution of visible and hidden units are given by

$$P(v,h) = \frac{1}{Z} e^{-E(v,h)} \tag{41}$$

where Z is simply simply the sum of $e^{-E(v,h)}$ for all possible v, h, which ensures that the probabilities will sum up to 1. Then we can calculate the probability of a given visible vector

$$P(v) = \frac{1}{Z} \sum_{h \in H} e^{-E(v,h)} \tag{42}$$

and similarly the probability of a hidden vector

$$P(h) = \frac{1}{Z} \sum_{v \in V} e^{-E(v,h)} \tag{43}$$

where V and H are all the possible visible and hidden vectors respectively.

We can compute the probability of a single hidden unit activating, given the visible unit states using equation 44

$$P(h_j = 1|v) = g\left(b_j + \sum_{i=1}^{m} w_{i,j} v_i\right) \tag{44}$$

and similarly, we can use equation 45 to calculate the probability of a visible unit activating, given the hidden unit states

$$P(v_i = 1|h) = g\left(a_i + \sum_{j=1}^{n} w_{i,j} h_j\right) \tag{45}$$

where $g()$ is the activation function, usually the logistic function, m and n are the number of units in the visible and hidden layers respectively.

The activations of the nodes in the hidden layer are independent of each other, given the activations of the visible layer after applying an input vector on the visible layer, and vice versa, the activations of the nodes in the visible layer are independent of each other, given the hidden unit activations. Thus we can simply calculate the probability of a hidden state vector given a visible unit vector using equation 46

$$P(v|h) = \prod_{i=1}^{m} P(v_i|h) \tag{46}$$

and similarly the probability of a visible vector given a hidden unit vector using equation 47

$$P(h|v) = \prod_{i=1}^{n} P(h_i|v) \tag{47}$$

An extension to the traditional RBMs allows for the visible units to be of multinomial distribution, while the hidden units remain of Bernoulli distribution, allowing the visible layers to take discrete values. In this case, for the activation of the visible units the softmax function is used.

$$g(x) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \text{ for j = 1,2..,K} \tag{48}$$

where K is the number of dimension of our data.

Now we have all the tools we need in order to train an RBM, which has some similarities with training a Feed Forward NN, since it also uses the gradient descent algorithm. The method proposed my Hinton [60][61] called contrastive divergence (CD) and goes as follows

- take random sample v from the set and apply it on the input. Given input v calculate the probabilities of the hidden units and sample an activation vector h from this distribution.
- compute the *positive gradient*, which is the outer product of v and h
- perform Gibbs sampling, where we sample a reconstruction v' based on the hidden activations, and the re-sample activations h'
- compute *negative gradient*, which is the outer product of v' and h'
- update the weight matrix of the network $\Delta W = \eta(vh^T - v'h'^T)$, where η is some predefined learning rate.

- update biases using $\Delta a = \eta(v - v'), \Delta b = \eta(h - h')$.

As in a FFNN we repeat the process a number of times for all the samples in the training set until the network converges.

4.2.2 Convolutional Restricted Bolzman Machines

The success of CNNs showed the potential of convolutions inside neural networks, which lead researchers to mold convolutions into existing architectures. Such examples are the Convolutional RBMs (CRBMs) and Convolutional Autoencoders (CAE), which will be discussed further on.

CRBMs [66] are basically simple RBMs which borrow the notion of shared weights from CNNs. The same analogies apply here as well, having different sets of weights make up the trainable filters used in the CRBMs.

Of course, the formulas used for CRBMs have to be adjusted to take the convolutions into account. The probability distribution of visible and hidden units remain the same

$$P(v, h) = \frac{1}{Z} e^{-E(v,h)} \tag{49}$$

but the energe function E is modified to

$$E(v, h) = \sum_{k=1}^{K} \sum_{j=1}^{N_H} \sum_{s=1}^{N_W} h_{ij}^k W_{rs}^k v_{i+r1,j+s1} - \sum_{k=1}^{K} b_k \sum_{i,j=1}^{N_H} h_{ij}^k - c \sum_{i,j=1}^{N_V} v_{ij} \tag{50}$$

We can simplify the above function as follows

$$E(v, h) = \sum_{k=1}^{K} h^k \bullet (\tilde{W}^k * v) - \sum_{k=1}^{K} b_k \sum_{i,j=1}^{N_H} h_{ij}^k - c \sum_{i,j=1}^{N_V} v_{ij} \tag{51}$$

where $\tilde{W}$ denotes the weight matrix flipped horizontally and vertically, $*$ denoted convolution and $\bullet$ denotes dot product. Now we can train a CRBM using the contrastive divergence algorithm, as in a simple RBM.

4.2.3 Convolutional Deep Belief Networks

Deep Belief Networks (DBNs) [66] is a type of deep neural networks which is trained to probabilistically reconstruct its inputs. The term "deep" refers to the architecture of the network which typically involves many layers. Each layer learns features of the previous layer in an unsupervised manner. The most popular approach is to stack RBMs on top of the other and train the in a layer-wise fashion. Alternatively, networks such as Autoencoders can also be used.

By stacking CRBMs create convolutional deep belief networks, which have enjoyed a lot of success in computer vision problems [65][66], as well as other applications [62] [63].

Additionally, CDBNs have been used as a method for pre-training deep convolutional neural networks. By learning general filters in an unsupervised manner, and then fine tuning the network in a supervised manner, the network converges much faster, and the vanishing gradient is not a problem anymore, since the weights have to be adjusted just by a small margin.

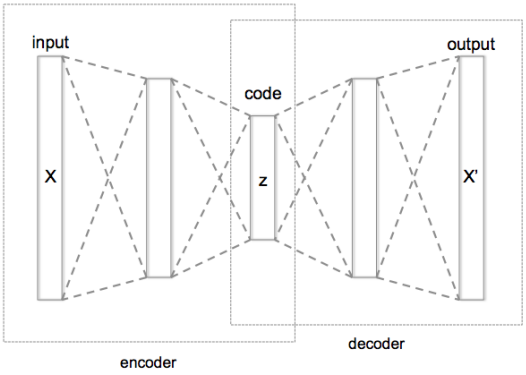

Figure 14: Example of a simple 2-layer Autoencoder. It has 2 layers in the coding part and 2 layers in the decoding part. The middle layer is shared between the coding and decoding part.

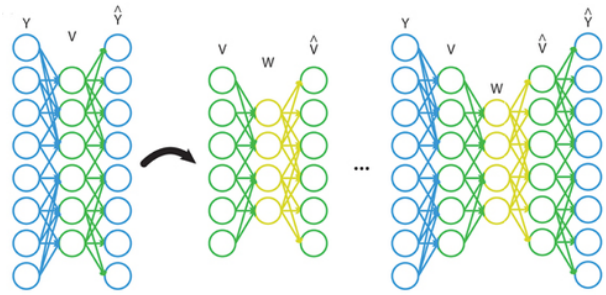

Figure 15: Example of a Stacked Autoencoder. Each Autoencoder is trained independently by using the input of the previous trained AE. When all AEs are trained we can stack the coding parts and the decoding parts together to form a Stacked Autoencoder, which has the same structure as a normal AE.

4.2.4 Autoencoders

Autoencoders (AE) [64] are types of NNs which are used to reduce the dimensionality of the input space. The simplest form of an Autoencoder is similar to the traditional multilayer perceptron with one hidden layer, with two constraints; the output layer has to have the exact same number of nodes as the input layer, and the hidden layer has to have less nodes than the input layer.

The labels used to calculate the error are the inputs themselves. Effectively this forces the network to reconstruct its input on the output nodes, after having reduced the dimensionality of the data in the hidden layer. Thus, the hidden layer learns lower dimensionality representations of the data, which make further supervised or unsupervised learning more efficient. Since finding a large unlabeled training set is much easier than finding a labeled one we can first train in unsupervised manner. Then we can take the coding part of the autoencoder, add another fully connected layer on top that will perform the classification, and using a smaller, labeled, training set we can fine tune the network. Since the network is already pre-trained, and already "knows" the data, it will converge much faster and more accurately.

For some applications, though, trying to learn the identity function might not be sufficient. For object recognition in images, for example, we need a representation of an object that is as generic as possible. Vincent et al. proposed the Denoising Autoencoders (DAE) [67], to solve this problem for learning image features, where the input x would be corrupted into $\hat{x}$, while the network would try to reconstruct the initial values of x. The noise introduced to the data would be of binomial distribution for black and white images, which would turn pixels on and of, and RGB images would be blurred with Gaussian noise.

We can, of course, extend the Autoencoder architecture and add more layers in between the input and output layers in order to learn more complicated data (figure 14). There are two approaches to training such network, as a traditional MLP or layer wise.

In the case of full network propagation we train the network as we would train any multilayer feed forward network; make a forward propagation to get the activations of all the nodes, calculate the error on the output nodes, backward propagate the error through the layers

and update the weights. The first part of the network is the called the coding part, where each successive layer has fewer nodes, and thus encodes or compresses the information, and the second part, called decoding part, has successively more nodes in each layer, which are symmetric to the coding layers. For very shallow networks this works fine, but for deeper architectures we start to encounter the problem of the vanishing gradient. Because traditional activation functions used have gradients in the range $(-1, 1)$ or $[0, 1)$ and the and the gradient of nodes in the lower levels of a n-layered deep network is calculated by multiplying n numbers smaller than 1, the numbers tend to get extremely small. Thus, the upper layers of the network get trained quite fast, but the lower layers learn very slowly, and only learn the average of the gradient over the dataset.

Stacked Autoencoders [65] are the second type of Autoencoders, which solve the problem of the vanishing gradient by training in a layer-wise fashion. We would start by training the first hidden layer of the network as we would in a single layer Autoencoder. Once we are confident that the first layer has learned to encode the input sufficiently well, we proceed to train the next layer of the network by taking the output of the first hidden layer and try to reconstruct it after the second hidden layer has encoded it. We repeat this process until every layer is trained (figure 15). Thus the network learns features hierarchically and because each layer is trained individually, the problem of the vanishing gradient, well.. vanishes.

Sparsity in feature representation has been shown [69][70][71] to increase the accuracy of various classification tasks [72]. Hinton et al. [70] and Lee et al. [71] propose methods for achieving sparsity in AE networks by using combination of different activation functions, sampling steps and the incorporation of penalties in the penalties in the error function which encourages nodes to activate less frequently. Makhzani et al. proposed the k-Sparse Autoencoder, which in each layer only the activations of the k highest activations are preserved, while the rest are set to zero. Cho [73] showed that introducing sparsity in Denoising AEs improves the network's ability to denoise highly noisy images.

4.2.5 Convolutional AutoEncoders

Sparse AE and DAE can learn features that very well represent the data, but the problem with these approaches is that the features learned are not global. In (Stacked) Convolutional Autoencoders (CAE) [68] the idea is to replace the fully connected layers with convolutional layers, and make the learned features global.

As with CNNs, we have two main types of layers, the convolutional layers and pooling layers. In the coding part of the network convolutional layers can be used in combination with pooling layers, as normal. On the decoding part, though, we have to reverse the operations. The reconstruction of the convolutional layer is obtained by:

$$y = g(\sum_{k \in H} h^k * \tilde{W}^k + b_k) \tag{52}$$

where, as before, $g()$ is the activation function, b_k is the bias term of the filter k, h^k is the feature map of the k-th filter, and $\tilde{W}^k$ is the weight matrix of filter k, flipped in both dimensions. The reconstruction of the pooling layer is done by replicating each point of each feature map n times, where n is the pooling kernel size.

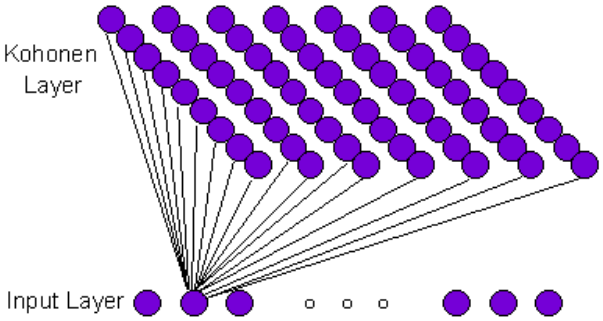

Figure 16: Example of a SOM topology.

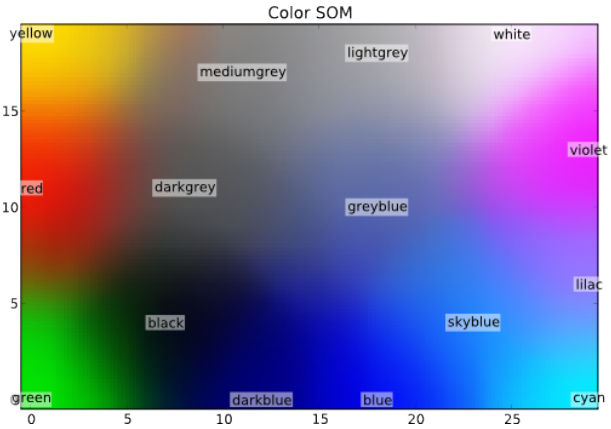

Figure 17: Example of a SOM projecting RGB colors into a 2d space.

4.2.6 Self Organizing Maps

Self Organizing Maps (SOM) [74], otherwise known as Kohonen Networks, are another type of unsupervised neural networks. SOMs can be used in order to project multidimensional data into lower dimensions, usually one or two, while preserving the topological properties of the input. They have been very useful in visualizing high dimensional data in lower dimensional views.

Structurally, SOMs are similar to traditional neural networks with one essential difference. We consider the nodes of a SOM to be organized in a lattice structure, which whose properties we take advantage of during the training of the network (figure 16). As with traditional NNs, each node has a connection to every input node and a weight associated with each connection. Each node forms a discriminant function and computes

$$y = \sum_{i=1}^{N} x_i w_i \tag{53}$$

where N is the number of inputs, x_i is the i^{th} element in the input vector and w_i is the weight of the node associated with that input. SOMs are winner-take-all networks, meaning that the node with the highest activation is the "winner", for one specific input. Once the winner is identified, the weights are updated the following mechanism

$$w = w + \eta(x - w) \tag{54}$$

where w is the weight vector of the node, η is a predefined learning rate and x is the input vector. Effectively, the winning node adjust its weights in a way to better model the specific input.

Now we have reach the point where we take advantage of the lattice structure of the network. The weight update rule is applied to the winning node, as well as the 8 neighbors surrounding it. This causes the neighborhood to model data similar to the input with some variations, since the weights are most likely different in their initial state.

As with traditional NNs, during the training phase we iterate over all our examples a number of times. In the end the network will have made a kind of topological "map" of the data in low dimensions, as showed in figure 17.

After the original publications, several improvements were suggested such as using Euclidean distance as a discriminant function, and the winner node is the one with the smallest distance. Several improvements have been suggested regarding the weight update rule. Namely Shah et al. [75] proposed having a time adjusted SOM, where each neuron has an individual learning rate and neighborhood size that changes over time.

5 UDN Gene Predictor

In this thesis we propose the Unsupervised Deep Network Gene Predictor (UDN Gene Predictor), that attempts to identify signals in DNA sequence that signify the start of a protein coding region in an unsupervised manner. To the extend of our knowledge this is the first work to propose an unsupervised signal sensor for gene prediction.

Features are extracted using a Convolutional AutoEncoder from the one-hot encoded representation of the raw sequences. BoSP features are then extracted from the feature maps of the encoder part of the CAE which are then used to train a Self Organizing Map, whose mappings are used as an additional feature for the final clustering.

5.1 Data representation

For this project, we would like to give input to the network sequences with as little preprocessing as possible. However, in order to take advantage of the properties of Convolutional Neural Networks, an encoding is proposed.

Nguyen et al. in [28] worked with windows of 500 bases for classification. We choose to work with sequences of 200 nucleotides long to reduce the ambiguity in the labeling, since we reduce the probability of having windows overlap coding, non-coding and coding in the compementary strand regions. Additionally the reduced window size helps keep the computational time of the experiments within feasible limits. To achieve the classification of genes, we train the networks to identify any kind of signal that signifies the start of a coding region. Given that we have 4 possible bases (A,T,G,C) we propose using a one hot encoding with 4 channels for each base position. Thus, for a sequence of 200 bases, we end up with a vector of dimensions (Height x Width x Channels) 1 x 200 x 4. Assuming that 'A' is mapped to the first channel, 'T' to the second, 'G' to the third and 'C' to the fourth, each channel gets a value 255 if the base at that position corresponds to that channel or 0 if it does not.

As an example, the sequence $ATCG$ would be encoded into:

$$[[255, 0, 0, 0][0, 255, 0, 0][0, 0, 0, 255][0, 0, 255, 0]] \tag{55}$$

We aim for our networks to identify signals that signify the start of a coding regions. To achieve that we ought to have enough bases before the start of any coding region within the input window. We choose a size of 50 bases. Assume a window of length 200. If a coding region starts at least 50 bases after the start of that window, or just before the end of the window, that window is labeled as '1', meaning it contains the signal that signifies the start of a coding region. Otherwise we label the window as '0'. This guarantees that the positively labeled windows contain at least 150 bases of the signal.

5.2 Convolutional AutoEncoder

For the CAE, we chose an architecture that is able to reconstruct the input sufficiently well, and has as few parameters as possible. The final result is a 6 layer CAE, 3 convolutional layers in the encoding part and 3 deconvolutional layers in the decoding part. The first convolutional layer shares weights with the last deconvolutional layer, the second convolutional layer shares weights with the second deconvolutional layer, and the last convolutional layer shares weights with the first deconvolutional layer. No pooling was used in this network.

The first convolutional layer consists of 60 filters of shape 7x1 (width, height), while the second and third layers consist of 30 filters of shape 5x1. Effectively this means that each node in the last convolutional layer encodes information from $7 + 5 + 5 - 2 = 15$ nucleotides in the input.

For the training process the quadratic function was used as a cost function, and stochastic gradient descent with momentum was used for training the network. The network was trained over 100000 batches of size 64, with initial learning rate 0.01, learning rate decay rate 0.1 with the decay being applied every 20000 batches.

Once the CAE is trained, the dataset is converted to the CAE encoded representation, by taking the activations of the last convolutional layer, for each individual example in our dataset.

5.3 BoSP

After the dataset has been converted to the CAE encoded representation, we apply the BoSP transformation to the dataset, as indicated in Section 4.1.5.4, to get our final feature representation.

5.4 Self-Organizing Map

A Self-Organizing Map is trained on the BoSP representation of the dataset. The grid used in the following experiments is of size 10x10.

Once the SOM is trained, we get the coordinates of each individual example in our dataset, and we append them to their feature vectors, which can be scaled in a way to play more or less significant role during the clustering.

5.5 K-Means

We use K-Means to cluster our data into two clusters. Ideally we would like the examples containing the signal to end up in one cluster and the rest to be in the other. As input, the BoSP features of the CAE encoded sequences are given, concatenated by their coordinates in the SOM.

6 Experiments

6.1 In search of Seven Clusters

In Section 2 the work of Gorban et al. was mentioned, where they visualized sequences by projecting them into the space of triplet frequencies. Additionally, they visualized the first and third principal components of features extracted from sequences using GLIMMER [37]. In both

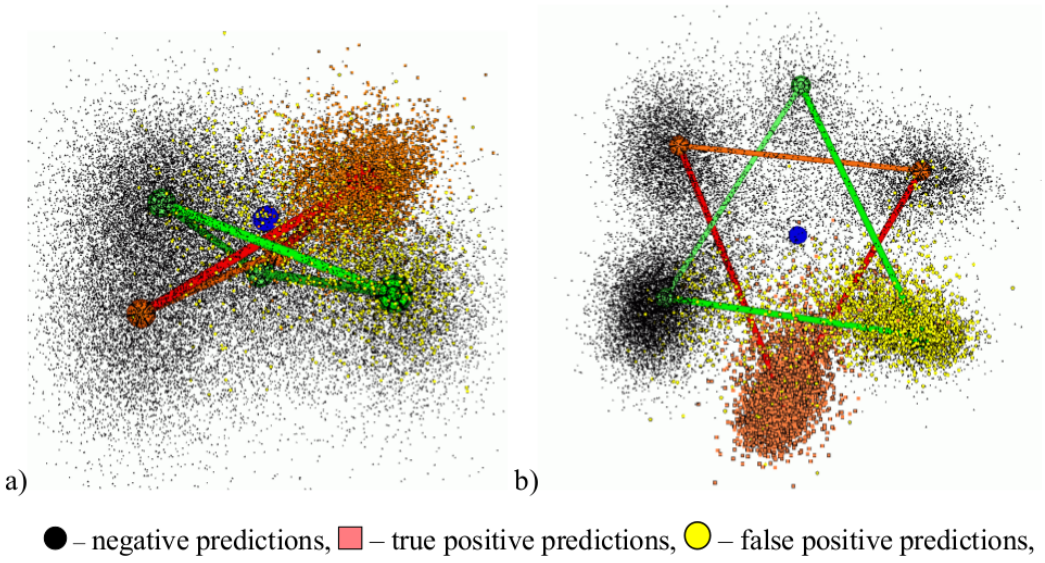

Figure 18: Visualization of the distribution of predictions of GLIMMER gene-finder in 64-dimensional space of codon frequencies. Every point corresponds to one ORF [38].
a) Escherichia coli. Projection on the 1st and 3d principal components.
b) Caulobacter crescentus. Projection on the 1st and 2d principal components.

cases they show that 7 visible clusters are formed, as shown in Figure 18. The labels they used for the sequences are *non-coding*, *coding in forward strand*, *coding in complementary strand*.

In the same spirit, we devised similar experiments. New datasets were created using the same labeling strategy as Gorban et al. for the sequences, with window sizes of 200 nucleotides. In our case the features used for the visualizations are the first and second principal components of the BoSP representation of sequences, extracted from the trained CAE. In Figures 19 20 21 it can be seen that the data are distributed mostly normally with the examples of the coding sequences in the positive strand (green) being slightly more dominant in the top right part of the graph, while the coding sequences in the complementary strand (blue) and non coding sequences (red) have a very similar distribution. No seven clusters are visible in this case.

Additionally, in Figure 22 the SOM mapping can be seen for the same dataset. It presents similar results as the scatter plot of Figure 19, where the green channel, which represents the positive examples in the positive strand, forms a small clear cluster. The blue and red channels, which represent the positive examples in the complementary strand and negative examples respectively, overlap for the most part, forming a purple blob in the image. Additionally there is a small cluster where all three classes overlap.

6.2 Preliminary Experiments

In order to reach the final point of our setup, we break down the experiments into 3 parts. The first part was designed in order to show that Convolutional Neural Networks are indeed capable of capturing and extracting relevant information from DNA sequences and distinguish where coding regions start. This is done by training a CNN network in a supervised manner.

In the second part we train Convolutional AutoEncoders in order to extract unsupervised

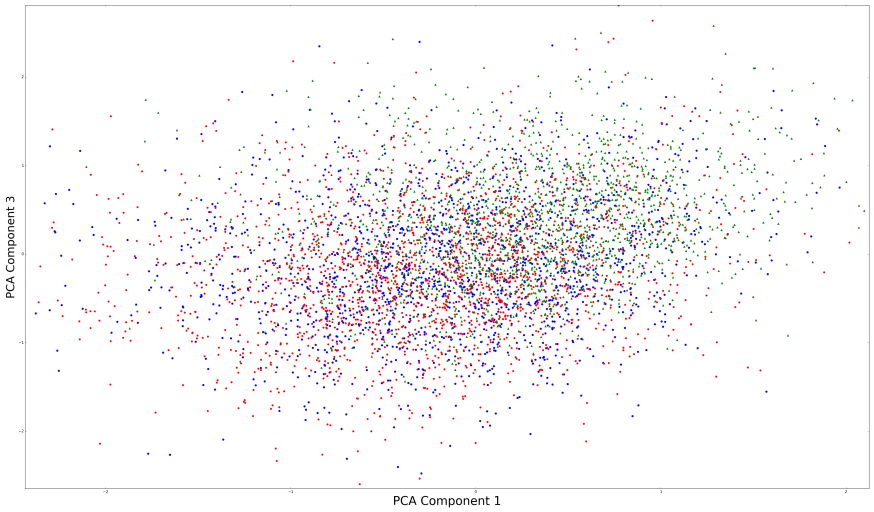

Figure 19: Visualization of the first and second Principal Components extracted from the CAE-BoSP features of E.Coli sequences. Green points are coding regions on the positive strand, blue points are coding regions in the negative strand and red points are non-coding regions.

Figure 20: Visualization of the first and second Principal Components extracted from the CAE-BoSP features of E.Coli sequences. Green points are coding regions on the positive strand and red points are non-coding regions.

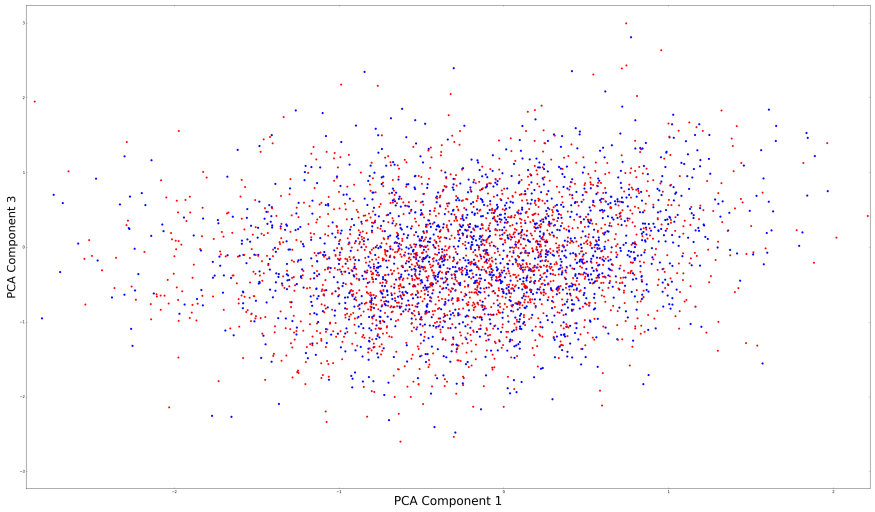

Figure 21: Visualization of the first and second Principal Components extracted from the CAE-BoSP features of E.Coli sequences. Blue points are coding regions in the negative strand and red points are non-coding regions.

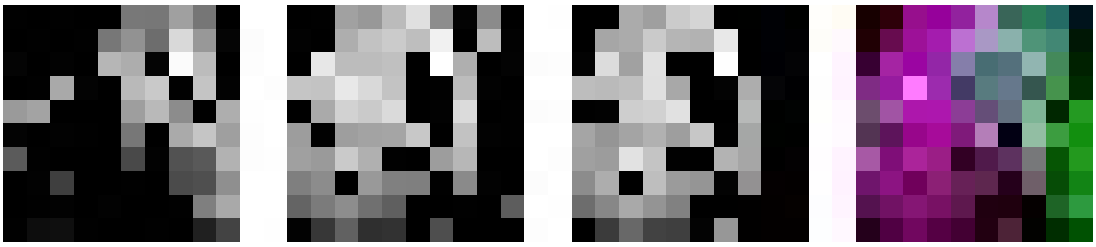

Figure 22: Visualization of a SOM mapping of the CAE-BoSP features of E.Coli sequences. In the left image only the mapping of the positive examples in the positive strand are shown. In the second image the positive examples in the complementary strand are shown. In the third image the negative examples are shown. In the last image all the examples are shown, with the green channel representing the positive examples in the positive strand, the blue channel the positive examples in the complementary strand and the red channel the negative examples.

features from the sequences. Then, we use the encoded representation of the sequences to train a supervised CNN as in the previous step. This part serves to show how discriminative those features are for this specific task. Ideally we would want features that yield the comparable results in classification as the original sequences.

As a final part of the experiments, we want to use the features learned in the second part, and train a Self-Organizing Map to see whether coding sequences form any visible and distinct clusters. The BoSP method is used in order to reduce the dimensionality of the encoded representation, while keeping the discriminative information.

6.2.1 Dataset

In order to get an intuition of whether the system is capable of such a task, we start off with a simple prokaryotic organism, *Escherichia Coli* (E. Coli), which has been studied extensively. E. Coli is used for our supervised experiments as well as for the supervised learning with unsupervised features experiments.

For the unsupervised learning 3 organisms are used, namely, *Escherichia Coli*, *Salmonella Enterica* and *Saccharomyces Cerevisiae*.

The datasets used are available on the NCBI site [76].

6.2.2 Supervised Training

In the first round of experiments the optimal architecture for a CNN has to be figured out. As a starting point a simple network topology is chosen, which consists of 3 convolutional layers, with the first two being followed by a pooling layer and a Local Response Normalization layer. Finally the network is completed by three fully connected layers. Similar networks are often used for small datasets such as MNIST for hand written digit recognition or cifar-10 for object recognition in images.

One of the problems we have when trying to identify signals just before the coding parts, is that the grand majority of windows, in a sliding window approach, will be negative examples and just very few will be positive examples. For this reason we experimented with different ratios of positive and negative examples in the training set. In Table 1 we show the results of those experiments. The sensitivity and specificity that are reported come from a separate test set, which consists of complete genomes. No preprocessing is done to the test set, and the ratio between positive and negative examples is set to default.

	def:def	1:1	2:1	3:1	4:1	5:1	6:1
sensitivity	0.8247	0.9911	0.8816	0.8671	0.8593	0.8484	0.8441
specificity	0.9933	0.9354	0.9662	0.9754	0.9829	0.9845	0.9875
f1 score	0.8640	0.6622	0.7630	0.7960	0.8278	0.8301	0.8432

Table 1: Sensitivity, specificity and f1 score of experiments using various ratios of positive and negative examples during training. The test set contains 4 complete genomes of E. Coli. The same network structure is used for all experiments.

Once the optimum ratio of examples has been established, similar network architectures are tested by removing the pooling layers one by one, starting with the one following the second convolutional layer. The sensitivity and specificity of each network can be seen in Table 2.

	c-p-c-p-c	c-p-c-c	c-c-c
sensitivity	0.8438	0.8381	0.8367
specificity	0.9836	0.9835	0.9831
f1 score	0.8225	81.52	0.81

Table 2: Sensitivity specificity and f1 score from 3 different networks. The first network consists of 3 convolutional layers where the first two are followed by pooling. The second network has 3 convolutional layers 'c' where only the first layer is followed by pooling 'p'. The third network consists only of convolutional layers. Note that the first two convolutional layers in all cases are followed by Local Response Normalization layers and all networks are completed with fully connected layers.

The parameters of those networks are shown in Table 3.

network \ layer	conv	pool	conv	pool	conv	fc	do	fc	do	fc
c-p-c-p-c	180x7x1	2x1	256x5x1	0.8671	2x1	1500	0.5	1500	0.5	2
c-p-c-c	180x7x1	2x1	256x5x1	0.9754	none	1500	0.5	1500	0.5	2
c-c-c	180x7x1	none	256x5x1	0.7960	none	1500	0.5	1500	0.5	2

Table 3: The network parameters used for the structure optimization experiments. For the convolutional layers (conv), the parameters shown are *number of filters x filter width x filter height*. For the pooling layers (pool) the parameters are *pooling width x pooling height*. For the fully connected (fc) layers the parameter shown is the number of nodes. Finally, for the dropout layer (do) the parameter is the probability of shutting down the output of a node in the previous layer.

From the results seen in Table 2 we can see that by removing the pooling layers the difference in performance is minimal.

Furthermore, we compare the performances of the model trained on *E. Coli* with the performance of a model trained on *S. Enterica*. Both models are evaluated using DNA sequences of *E. Coli*, *S. Enterica* and *S. Cerevisiae*. In Tables 4,5 and 6 the f1 score, sensitivity and specificity is shown for each model when evaluated for each organism.

organism \ model	E.Coli	S.Enterica
E.Coli	0.8127	0.7370
S.Enterica	0.7449	0.8273
S.Cerevisiae	0.1315	0.1329

Table 4: F1 score of each model (columns) when applied to each corresponding organism (rows)

organism \ model	E.Coli	S.Enterica
E.Coli	0.8306	0.7406
S.Enterica	0.7592	0.8538
S.Cerevisiae	0.2192	0.2330

Table 5: Sensitivity of each model (columns) when applied to each corresponding organism (rows)

organism \ model	E.Coli	S.Enterica
E.Coli	0.9831	0.9788
S.Enterica	0.9787	0.9840
S.Cerevisiae	0.9184	0.9123

Table 6: Specificity of each model (columns) when applied to each corresponding organism (rows)

Is is clear from the results that using a model trained on one organism to make predictions for another, significantly reduces its performance. Though the model maintains a high degree of specificity, the sensitivity drops substantially.

6.2.3 Unsupervised Features - Supervised Training

For the unsupervised feature extraction a 6-layer Convolutional AutoEncoder is trained, of similar architecture to the supervised training. Once the network is trained, the training set is converted to the BoSP representation of the CAE encoded features, which is used to train a 3 layer fully connected network in a supervised manner. The BoSP features are normalized to have 0 mean and standard deviation 1.

The neural network is unable reach the performance of the fully supervised approach, yielding sensitivity of 85.55% specificity of 49.90% and f1-score of 0.2082. This shows that the features extracted from the CAE are not discriminative enough to produce similar results as the fully supervised method.

Different network architectures where tested and different normalization techniques, but without any success.

6.3 UDN Gene Finder Experiments

For the UDN Gene Finder experiments we employ the pipeline introduced in Section 5. To get a sense of whether this method is applicable to a broader spectrum of organisms, this pipeline is applied to the genomes of the prokaryotes *Escherichia Coli*, *Salmonella Enterica* and *Saccharomyces Cerevisiae*.

6.3.1 Prediction results from K-Means

We use K-Means for clustering our dataset, in the BoSP representation concatenated by their SOM position, into two clusters.

Finally, we compare the clustering performance on each organism when using a model trained on its own genome, against the results when a model that trained on another organism (BoSP

representation of the CAE encoded sequence trained on the same organism as the K-Means model). The results are shown in Tables 7,8,9.

organism \ model	E.Coli	S.Enterica	S.Cerevisiae
E.Coli	0.070	0.1219	0.1232
S.Enterica	0.1050	0.1222	0.1163
S.Cerevisiae	0.0792	0.0749	0.0737

Table 7: F1 score of each model (columns) when applied to each corresponding organism (rows)

organism \ model	E.Coli	S.Enterica	S.Cerevisiae
E.Coli	0.4151	0.4763	0.4536
S.Enterica	0.4094	0.4845	0.4288
S.Cerevisiae	0.5228	0.5080	0.4873

Table 8: Sensitivity of each model (columns) when applied to each corresponding organism (rows)

organism \ model	E.Coli	S.Enterica	S.Cerevisiae
E.Coli	0.5129	0.5135	0.5462
S.Enterica	0.5129	0.5085	0.5466
S.Cerevisiae	0.5499	0.5348	0.5476

Table 9: Specificity of each model (columns) when applied to each corresponding organism (rows)

The results presented above are the average achieved over 5 runs.

It is hard to compare our method with any other, since, to the extend of our knowledge, this is the only unsupervised signal sensor approach for gene prediction. But it is clear from the results that the performance is comparable to random classification.

6.3.2 Visualization of Self-Organizing Maps

As mentioned in Section 4.2.6, Self Organizing Maps project multidimensional data to a 2-dimensional grid based on similarity. Here we use the BoSP representation of our dataset as input to a SOM of grid size 10x10. We compare the SOM visualizations of the BoSP representation of raw sequences in one hot encoding, with the data produced with the aforementioned approach (Figure 23).

We start off with the BoSP representation of raw nucleotide sequences in the one-hot representation described in 5.1, and used it as input to a SOM 24. Since the one-hot representation uses 4 channels, we can convert it to a BoSP representation, which effectively will give us a vector of length 4, containing a relative distribution of nucleotides within the current window.

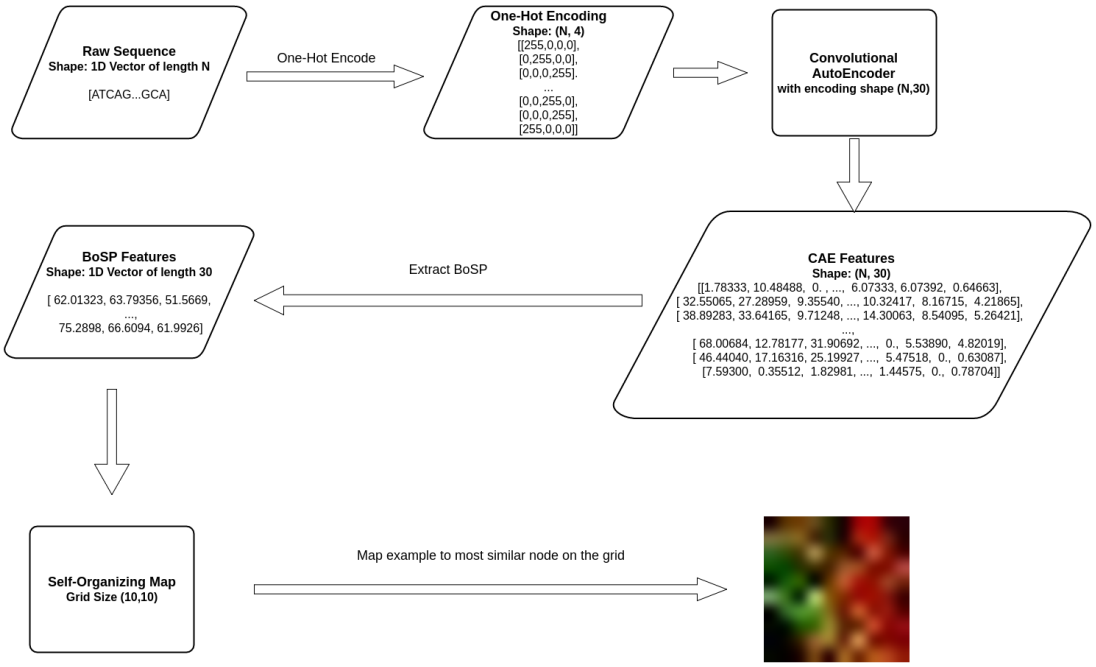

Figure 23: Data flow diagram of the visualization process using Self-Organizing Maps. In the SOM visualization the intensity of each pixel shows the amount of examples that mapped to that corresponding node. The positive examples are represented by the green channel, while the negative examples are represented by the red channel. Thus, a intense green pixel means that many positive examples mapped to that node, while intense red means many negative examples mapped to that node. Yellow tinted pixels are nodes that many positive and negative examples have mapped to that node.

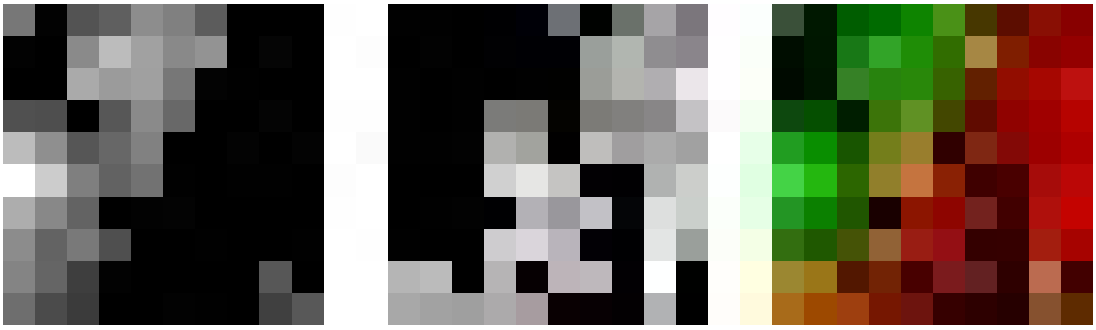

Figure 24: SOM visualization of BoSP representation of E.Coli nucleotide sequences, with 5:1 ratio of negative to positive examples.

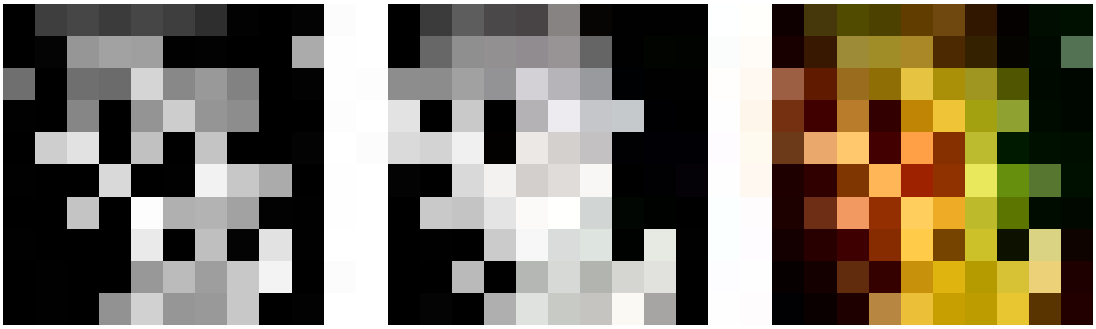

Figure 25: SOM visualization of the BoSP representation of E.Coli nucleotide sequences, with default ratio of negative to positive examples.

Using this as input to a SOM gives us an intuition of whether a simple feature as that can be used for clustering the sequences. it is clear from the image that there are visible clusters for positive and negative examples.

For all SOM visualizations the following structure is followed:

- Left image: Mapping of only the positive examples.
- Center image: Mapping of only the negative examples.
- Right image: Mapping of both positive and negative examples. Positives are represented by the green channel, and negatives by the red.

In the case of Figure 24, a ratio of 5:1 is used between negative and positive examples, as is used during the supervised learning. In Figure 25 we see the SOM visualization of BoSP representation of the one-hot encoding, of the same sequences using the default ratio of positive and negative examples. It can be seen that there are some distinct clusters formed, but the amount of overlap between the two classes increased significantly.

The difference in this visualization steps indicates that the main problem here is the ratio between our two classes. Both in the case of SOM and K-Means, the negative examples are too few in order to form their own distinct cluster, and are taken in by the much larger cluster of negative examples.

Since we use unsupervised NNs to solve this problem, the question becomes whether we can extract features in an unsupervised manner, that would highlight the essential differences

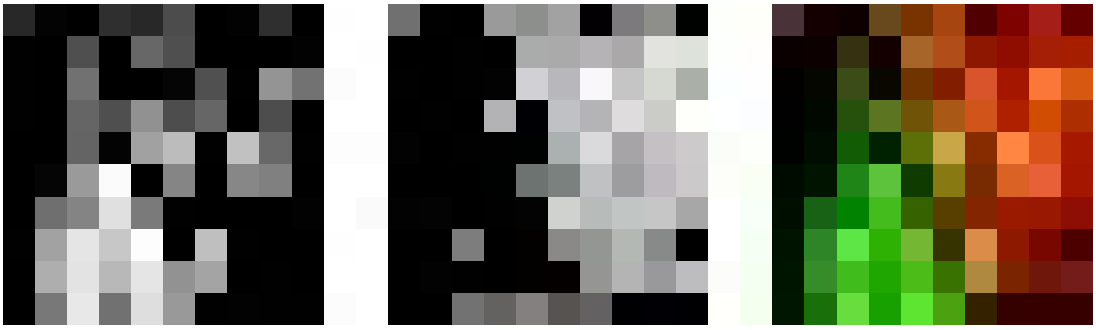

Figure 26: SOM visualization of the CAE-BoSP features of E.Coli, using ratio of 5:1.

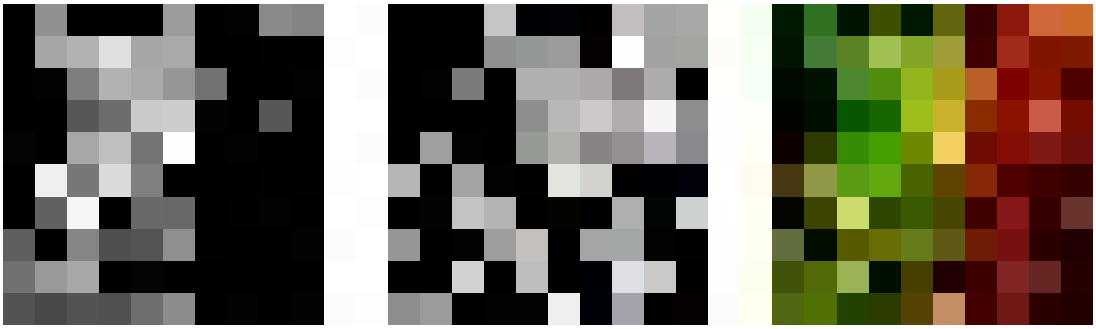

Figure 27: SOM visualization of the CAE-BoSP features of E.Coli, using the default ratio.

between two classes. Using the Convolutional Auto-Encoders described in Section 6.2.3, the dataset was converted to the encoded representation. In Figure 26 the results can be seen for the SOM mapping of the dataset using the 5:1 ratio, and in Figure 27 the results of the same process using the default ratio.

This time the visualization tells a somewhat different story. Using the BoSP representation of the CAE features, the data are somewhat separated even when using the default ratio. This hints that the CAE-BoSP features increase the seperability of the data.

Additionally, Figures 28 and 29 show the SOM mapping for the organisms *Salmonella Enterica* and *Saccharomyces Cerevisiae* respectively. Both these organisms have more sparse genome than *Escherichia Coli*, meaning that the default ratio for positive and negative examples is even smaller. Even so, the SOM is able to map the different classes with relatively small amount of overlap.

7 Conclusions and Future Work

In this thesis we studied the abilities of Convolutional Neural Networks to classify segments of DNA sequences. We showed that there is a signal leading to protein coding regions, which CNNs are able to capture with high sensitivity and specificity. Following that, we studied whether Convolutional AutoEncoders can extract features from DNA sequences which are descriptive enough to yield similar results under unsupervised setting, as the raw sequences. From the results we can conclude that the features produces by the CAE are not optimal for this classification task.

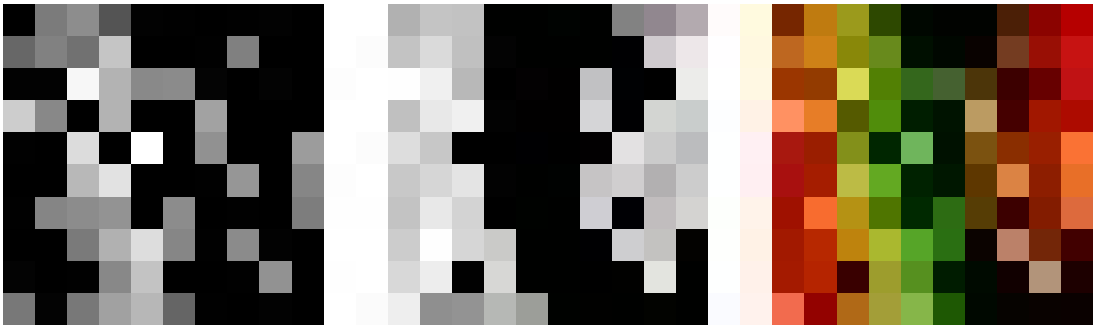

Figure 28: SOM visualization of the BoSP features of S. Enterica , using the default ratio.

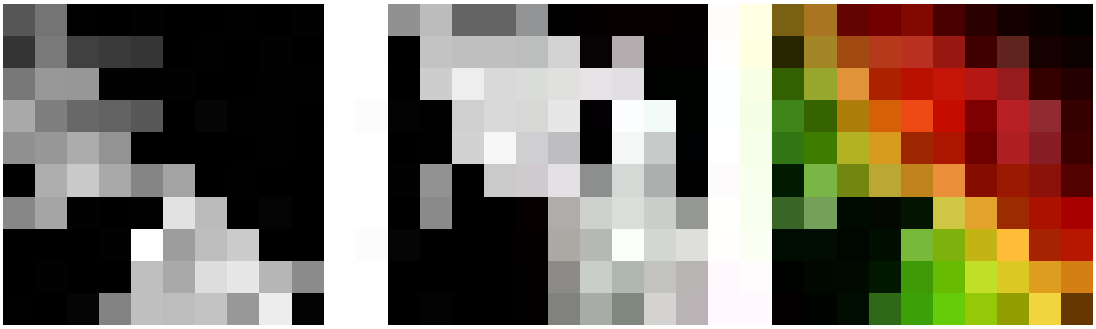

Figure 29: SOM visualization of the BoSP features of S. Cerevisiae , using the default ratio.

We introduced a novel approach for gene prediction, the UDN gene finder, an unsupervised signal sensor for gene prediction. We show that, even though clustering techniques are not able to separate the data in the desired way, when visualizing the data using Self-Organizing Maps, there is a significant degree of separation of the sequences that contain the signal we are searching for and the sequences that do not.

For feature work it would be interesting to investigate why SOMs are able to separate the two classes while K-Means is unable to. Additionally we can try and apply other unsupervised feature selection algorithms. There are plenty proposed in the literature, all of them with their advantages and disadvantages. Such a study will not provide us with better features, rather, it will help create more crisp clusters and thus more meaningful results.

Acknowledgments

I would like to thank my first supervisor, Dr. Erwin Bakker, for supporting and guiding me throughout the whole duration of my thesis. He was always eager to advise me, and every meeting was productive, interesting and, most of all, fun.

Secondly, I would like to thank my second supervisor, Dr. Michael Lew, for taking the time to assess and evaluate my work, as well as giving me many opportunities to work with him during my thesis.

I want to extend my gratitude to my friends with whom I had many constructive discussions over my thesis, and supported me during the whole period.

Most of all I want to thank my family and friends for supporting me and being patient with

me, allowing me to do everything at my own pace.

References

[1] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter. "Molecular Biology of the Cell (Fourth ed.)". ISBN 978-0-8153-3218-3, 2002.

[2] P. Russell. "iGenetics". New York: Benjamin Cummings. ISBN 0-8053-4553-1, 2001.

[3] C. Mathe, M. F. Sagot, T. Schiex, P. Rouze. "Current methods of gene prediction, their strengths and weaknesses". Oxford University Press. Nucleic Acids Research. Vol. 30 No. 19 pp. 4103-4117, 2002.

[4] S. Lakshmi, A. Shahin. "A Survey on Gene Prediction Using Neural Network". International Journal of Computer & Organization Trends. Volume 3, Issue 4, pp.88-93, 2013.

[5] N. Goel, S. Singh, T.C. Aseri. "A Review of Soft Computing Techniques for Gene Prediction". ISRN Genomics Volume 2013, article ID 191206, 2013.

[6] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman. "Basic local alignment search tool". Journal of Molecular Biology. Volume 215, Issue 3, pp. 403410, 1990.

[7] D. Lipman, W. R. Pearson. "Rapid and sensitive protein similarity searches". Science Volume 227 (4693) pp. 1435-41, 1985.

[8] G. D. Stormo. "Gene-Finding Approaches for Eukaryotes". Cold Spring Harbor Laboratory Press, Issue 10, pp. 394-397, 2016.

[9] N. Yu, Z. Yu, B. Li, F. Gu, Y. Pan. "A Comprehensive Review of Emerging Computational Methods for Gene Identification". In Neural Information Processing Systems (NIPS), Volume 12, pp. 1-34, 2016

[10] T. F. Smith, M. S. Waterman. "Identification of Common Molecular Subsequences". Journal of Molecular Biology Volume 147 pp. 195-197, 1981.

[11] A. V. Lukashin, M. Borodovsky."GeneMark.hmm: new solutions for gene finding". Oxford University Press. Nucleic Acids Research, Volume 26, No. 4 pp. 1107-1115, 1998.

[12] A. Krogh. "Using Database Matches with HMMGene for Automated Gene Detection in Drosophila". Genome Research. Volume 10, Issue 4, pp. 523-528, 2000

[13] J. Henderson , S. Salzberg, K.H. Fasman. "Finding Genes in DNA with a Hidden Markov Model". Journal of Computational Biology. Volume 4 Issue 2, pp. 127-141, 2009

[14] M. Stanke, B. Morgenstern. "AUGUSTUS: a web server for gene prediction in eukaryotes that allows user-defined constraints". Nucleic Acids Research, Volume 33, Issue suppl_2, pp. W465-467, 2005.

[15] I. Korf. "Gene finding in novel genomes". BMC Bioinformatics, Volume 5, Article 59, 2004.

[16] M.G. Reese, F.H. Eeckman, Kulp D., D. Haussler. "Improved Splice Site Detection in Genie". Journal of Computational Biology. Volume 4 Issue 3, pp. 311-323, 1997

[17] E. C. Uberbacher and R. J. Mural. "Locating protein-coding regions in human DNA sequences by a multiple sensor-neural network approach". Proceedings of the National Academy of Sciences of the United States of America, Volume 88, Issue 24, pp. 11261-11265, 1991.

[18] Y. Xu, J. R. Einstein, R. J. Mural, M. Shah, and E. C. Uberbacher. "An improved system for exon recognition and gene modeling in human DNA sequences". in Proceedings of the 16th Annual International Conference Intelligent Systems for Molecular Biology, pp. 376-383, 1994.

[19] E. E. Snyder and G. D. Stormo. "Identification of protein coding regions in genomic DNA". Journal of Molecular Biology, Volume 248, Issue 1, pp. 1-18, 1995.

[20] C. Li, P. He, J. Wang. "Artificial Neural Network Method for Predicting Protein Coding Genes in the Yeast Genome" Internet Electronic Journal of Molecular Design, Volume 2 pp. 527-538, http://www.biochempress.com, 2003.

[21] J.R. Cohen. "Segmenting speech using dynamic programming". The Journal of the acoustical society of America, Volume 69, Issue 5, pp. 1430-1438, 1981.

[22] D. Chicco, P. Sadowski, P. Baldi. "Deep Autoencoder Neural Networks for Gene Ontology Annotation Predictions". Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, pp. 533-540, 2014.

[23] V. Pavlivic, A. Garg, S. Kasif. " A Bayesian Framework for combining gene predictions". Bioinformatics, Volume 18, No. 1, pp 19-27, 2002.

[24] S. Mahony, J.O. McInerney, T.J. Smith, A. Golden. "Gene prediction using the Self-Organizing Map: automatic generation of multiple gene models". BMC Bioinformatics, Volume 5, article 23, pp. 1-9, 2004.

[25] T.Zeng, R.Li, R.Mukkamala, J.Ye, S.Ji. "Deep convolutional neural networks for annotating gene expression patterns in the mouse brain". BMC Bioinformatics, Volume 16, article 147, 2015.

[26] R. Singh, J. Lanchantin, G. Robins, Y. Qi. "DeepChrome: Deep-learning for predicting gene expression from histone modifications". Bioinformatics Volume 32, no. 17, pp. i639-i648, 2016.

[27] S. Wang, J. Peng, J. Ma, J. Xu. "Protein Secondary Structure Prediction Using Deep Convolutional Neural Fields". Scientific Reports 6, Articlenumber18962, 2016.

[28] N.G. Nguyen, V.A. Tran, D.L. Ngo, D. Phan, F.R Lumbanraja, M.R. Faisal, B. Abapihi, M. Kubo, K. Satou. "DNA Sequence Classification by Convolutional Neural Network". Journal of Biomedical Science and Engineering Volume 9, No. 5, pp. 280-286, 2016.

[29] K.L. Chan, R. Rosli, T.V. Tatarinova, M. Hogan, M. Firdaus-Raih, E.T.L. Low. "Seqping: gene prediction pipeline for plant genomes using self-training gene models and transcriptomic data". International Conference on Bioinformatics of Genome Regulation and Structure \Systems Biology, 2016

[30] C. Holt, M. Yandell. "MAKER2: an annotation pipeline and genome-database management tool for second-generation genome projects". BMC Bioinformatics, Volume 12, article 491, 2011.

[31] Benchmarking Universal Single-Copy Orthologs (BUSCO). http://busco.ezlab.org/

[32] D. Weekes, G.B. Fogel. "Evolutionary optimization, backpropagation, and data preparation issues in QSAR modeling of HIV inhibition by HEPT derivatives". BioSystems, Issue 72, pp. 149-158, 2003.

[33] M. Cheung, G.B. Fogel. "Identification of Functional RNA Genes Using Evolved Neural Networks". Computational Intelligence in Bioinformatics and Computational Biology, 2005.

[34] W.H. Majoros, M. Pertea, S.L. Salzberg. "TigrScan and GlimmerHMM: two open source ab initio eukaryotic gene-finders". Bioinformatics, Volume 20, Issue 16, pp. 2878-2879, 2004.

[35] M. Pertea, X. Lin, S.L. Salzberg. "GeneSplicer: a new computational method for splice site prediction". Nucleic Acids Research, Volume 29, Issue 5, pp. 1185-1190, 2001.

[36] K.J. Hoff, S. Lange, A. Lomsadze, M. Borodovsky, M. Stanke. "BRAKER1: Unsupervised RNA-Seq-Based Genome Annotation with GeneMark-ET and AUGUSTUS." Bioinformatics, Volume 32, Issue 5, pp. 767-769, 2015.

[37] W.H. Majoros, M. Pertea, C. Antonescu, S.L. Salzberg. "GlimmerM, Exonomy and Unveil: three ab initio eukaryotic genefinders". Nucleic Acids Research, Volume 31, Issue 13, pp. 3601-3604, 2003.

[38] A.N. Gorban, T.G. Popova, A.Y. Zinovyev. "SEVEN CLUSTERS AND UNSUPERVISED GENE PREDICTION". In proceedings of the fourth international conference on bioinformatics and genome regulation and structure, Volume 2, 2004

[39] A.N. Gorban, A.Y. Zinovyev, T.G. Popova. "SEVEN CLUSTERS AND UNSUPERVISED GENE PREDICTION". In silico biology, Volume 3, Issue 4, pp. 471-82, 2003.

[40] J. Besemer, A. Lomsadze, M. Borodovsky. "GeneMarkS: a self-training method for prediction of gene starts in microbial genomes. Implications for finding sequence motifs in regulatory regions". Nucleic Acids Research, Volume 29, Issue 12, pp. 2607-2618, 2001.

[41] A. Lomsadze, V. Ter-Hovhannisyan, Y.O. Chernoff, M. Borodovsky. "Gene identification in novel eukaryotic genomes by self-training algorithm". Nucleic Acids Research, Volume 33, Issue 20, pp. 6494-6506, 2005.

[42] V. Ter-Hovhannisyan, A. Lomsadze, Y.O. Chernoff, M. Borodovsky. "Gene prediction in novel fungal genomes using an ab initio algorithm with unsupervised training". Genome Research, Volume 18, Issue 12, pp. 1979-1990, 2008.

[43] C. M. Bishop. "Neural Networks for Pattern Recognition". ISBN-13: 978-0198538646, 1995.

[44] https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

[45] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. "Gradient-based learning applied to document recognition". Proceedings of the IEEE, Volume 86, Issue 11, pp. 22782324, 1998.

[46] A. van den Oord; S. Dieleman; B. Schrauwen. "Deep content-based music recommendation". Curran Associates, Inc.: 2643-2651, 2013.

[47] A. Krizhevsky, I. Sutskever, G. Hinton. "Imagenet classification with deep convolutional neural networks". In Advances in Neural Information Processing Systems NIPS, 2012.

[48] O. Abdel-Hamid, A. R. Mohamed, H. Jiang, L. Deng, G. Penn, D. Yu. "Convolutional Neural Networks for Speech Recognition". IEEE/ACM Transactions on audio, speech and language processing, VOL. 22, 2014.

[49] R. Collobert, J. Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning". Proceedings of the 25th International Conference on Machine Learning. ICML '08 (New York, NY, USA: ACM): 160167, 2008.

[50] X. Glorot, A. Bordes; B. Yoshua. "Deep Sparse Rectifier Neural Networks". Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11): 315-323, 2011

[51] V. Nair, G.E. Hinton."Rectified Linear Units Improve Restricted Boltzmann Machines". In Proceedings of the 27 th International Conference on Machine Learning, Haifa, Israel, 2010.

[52] A.L. Maas, A.Y. Hannun, A.Y. Ng."Rectifier Nonlinearities Improve Neural Network Acoustic Models".In Proceedings of the 30 th International Conference on Machine Learning, Atlanta, Georgia, USA. JMLR: W&CP volume 28, 2013.

[53] M.D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q.V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, G.E. Hinton. "On Rectified Linear Units For Speech Processing". Conference PaperinAcoustics, Speech, and Signal Processing, 1988. ICASSP-88, pp. 3517-3521. International Conference, 2013.

[54] X. Glorot, A. Bordes, Y. Bengio. "Deep Sparse Rectifier Neural Networks". In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. JMLR W&CP (AISTATS 2011), 2011.

[55] Y. Guo, M.S. Lew. "Bag of Surrogate Parts: one inherent feature of deep CNNs". 27th British Machine Vision Conference, 2016.

[56] D. Erhan, P.A. Manzagol, Y. Bengio, S. Bengio, P. Vincent. "The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training". In Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 153160, 2009.

[57] A. Krogh, J.A. Hertz "A simple weight decay can improve generalization". In Proceedings of the 4th International Conference on Neural Information Processing Systems, 1991.

[58] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". Journal of Machine Learning Research, Volume 15, Issue 1, pp. 1929-1958, 2014.

[59] P. Smolensky. "Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory". In Rumelhart, David E.; McLelland, James L. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations. MIT Press. pp. 194281. ISBN 0-262-68053-X, 1986.

[60] G.E. Hinton. "Products of Experts". ICANN, 1999.

[61] G.E. Hinton. "Training Products of Experts by Minimizing Contrastive Divergence". Neural Computation 14, pp. 17711800, 2002.

[62] H. Lee, Y. Largman, P. Pham, A.Y. Ng. "Unsupervised feature learning for audio classification using convolutional deep belief networks". In Advances in Neural Information Processing Systems 22 (NIPS), 2009.

[63] A. Mohamed, D. Yu, L. Deng. "Investigation of Full-Sequence Training of Deep Belief Networks for Speech Recognition". In International Symposium of Computer Architecture (ISCA), 2010.

[64] D.E. Rumelhart, G.E. Hinton, R.J. Williams. "Learning internal representations by error propagation". In Parallel Distributed Processing. Vol 1: Foundations. MIT Press, 1986.

[65] Y. Bengio. "Learning deep architectures for AI". Foundations and Trends in Machine Learning.

[66] H. Lee, R. Grosse, R. Ranganath, A.Y. Ng. "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations". In Proceedings of the 26 th International Conference on Machine Learning, 2009.

[67] P. Vincent, H. Larochelle, Y. Bengio, P.A. Manzagol. "Robust Features with Denoising Autoencoders". Neural Information Processing Systems (NIPS), 2008.

[68] J. Masci, U. Meier, S. Ciresan, J. Schmidhuber "Stacked convolutional auto-encoders for hierarchical feature extraction". In Proceedings of the 21th international conference on Artificial neural networks - ICANN 2011, pp. 52-59, 2011.

[69] B.A. Olshausen, D.J. Field. "Sparse coding with an overcomplete basis set: A strategy employed by v1?". Vision research, Volume 37, Issue 23, pp. 3311-3325, 1997.

[70] V. Nair, G.E. Hinton. "3d object recognition with deep belief nets". In Advances in Neural Information Processing Systems, pp. 1339-1347, 2009.

[71] H. Lee, C. Ekanadham, A. Ng. "Sparse deep belief net model for visual area v2". In Advances in neural information processing systems, pp. 873-880, 2007.

[72] A. Makhzani, B. Frey. "k-Sparse Autoencoders". International Conference on Learning Representations, ICLR, 2014.

[73] K. Cho. "Simple Sparsification Improves Sparse Denoising Autoencoders in Denoising Highly Noisy Images". Proceedings of the 30 th International Conference on Machine Learning, Atlanta, Georgia, USA. JMLR: W&CP Volume 28, 2013.

[74] T. Kohonen. "Self-Organized Formation of Topologically Correct Feature Maps". In Biological Cybernetics, Springer-Verlag. Volume 43, pp. 59-69, 1982.

[75] H. Shah-Hosseini, R. Safabakhsh. "TASOM: A new time adaptive self-organizing map". IEEE Transactions on Cybernetics. Volume 33, Issue 2 pp. 271-82, 2003.

[76] US National Center for Biotechnology Information, E.Coli repository. https://www.ncbi.nlm.nih.gov/genome/

[77] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". arXiv preprint arXiv:1408.5093, 2014.