UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS no. 1861

Classification of 1D-Signal Types Using Deep Learning

Filip Floreani

Zagreb, June 2019.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 7 March 2019

MASTER THESIS ASSIGNMENT No. 1861

Student: **Filip Floreani (0036485656)**
Study: Computing
Profile: Computer Science

Title: **Classification of 1D-Signal Types Using Deep Learning**

Description:

During the de novo genome assembly process, an important step is calculating overlaps of sequenced reads. Accuracy is crucial because false overlaps greatly impede successful sequence reconstruction. It is possible to construct a pile-o-gram graph for each read. From this graph, we can obtain a 1D-signal that might be further analysed. Different signal types represent different types of reads that might produce false overlaps (i.e., false overlaps due to chimeric reads or reads from interspersed or tandem repeats).
The task is to develop a method for classification of these 1D-signals. Signals can be converted to images or kept in its existing form. Semi-supervised or supervised learning methods based on deep neural network should be used. Manually labelled signals should be used for training and evaluation of achieved results.
The solution should be implemented in Python with the TensorFlow or similar library. The source code has to be documented using comments and should follow the Google Python Style Guide when possible. The complete application should be hosted on GitHub under an OSI approved license.

Issue date: 15 March 2019
Submission date: 28 June 2019

Mentor:
Full Professor Mile Šikić, PhD

Committee Chair:
Assistant Professor Marko Čupić, PhD

Committee Secretary:
Associate Professor Tomislav Hrkać, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 7. ožujka 2019.

DIPLOMSKI ZADATAK br. 1861

Pristupnik: **Filip Floreani (0036485656)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Identifikacija tipova 1D-signala dubokog učenja**

Opis zadatka:

Tijekom de novo sastavljanja genoma, važan korak je pronalaženje preklapanja među očitanjima. Pogrešno detektirana preklapanja onemogućuju ispravno sastavljanje originalne sekvence. Za svako očitanje moguće je izgraditi graf pokrivenosti svake baze. Takav graf daje nam 1D-signal koji se može dodatno analizirati. Različiti tipovi signala predstavljaju različite tipove očitanja koja mogu rezultirati pogrešnim preklapanjem (npr. pogrešna preklapanja uzrokovana kimernim očitanjima ili očitanjima iz razasutih ili uzastopnih ponavljajućih regija).
Zadatak ovog rada je razviti metodu za klasifikaciju navedenih 1D-signala. Signali se mogu pretvoriti u slike ili zadržati u postojećoj formi. Potrebno je koristiti polu-nadzirane ili nadzirane metode učenja temeljene na dubokim neuronskim mrežama. Za učenje i evaluaciju postignutih rezultata koristiti ručno označene signale.
Rješenje treba biti napisano u jeziku Python pomoću biblioteke TensorFlow (ili slične). Programski kod je potrebno komentirati i pri pisanju pratiti stil opisan u Googleovom Python vodiču. Kompletnu aplikaciju postaviti na GitHub pod jednom od OSI odobrenih licenci.

Zadatak uručen pristupniku: 15. ožujka 2019.
Rok za predaju rada: 28. lipnja 2019.

Mentor:
Prof. dr. sc. Mile Šikić

Djelovođa:
Izv. prof. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:
Doc. dr. sc. Marko Čupić

I would like to give special thanks to my advisor prof. Mile Šikić, Ph.D. for the support, guidance and engagement from the very beginning.

Furthermore, I would also like to give great thanks to Lovro Vrček, mag. phys. for all his timely assistance and patience during the writing of this thesis.

Contents

1. Introduction **1**

2. DNA sequencing and assembly **3**

2.1. Overlap-Layout-Consensus 3

2.2. Read types 4

3. Deep learning methods **8**

3.1. Algorithm components 9

3.1.1. Loss function 9

3.1.2. Optimization 10

3.2. Common operations 11

3.2.1. Pooling 11

3.2.2. Activations 12

3.3. Convolutional Neural Networks 12

3.3.1. One-dimensional CNNs 15

3.4. Recurrent Neural Networks 16

3.4.1. Backpropagation through time 18

3.4.2. Long Short-Term Memory 19

3.5. Autoencoders 21

3.5.1. Variational autoencoders 23

4. Dataset **28**

4.1. Pairwise Mapping Format 28

4.2. Preprocessing 28

5. Results and evaluation **32**

6. Conclusion **38**

Bibliography **40**

List of Figures **43**

List of Tables **44**

1. Introduction

Bioinformatics is defined as an interdisciplinary scientific field, encompassing in it computer science, statistics, biology and information processing. It is concerned with developing methods and software tools for a better understanding of biological data. The first mentions of it were back in 1970 when it was defined as the study of information processes in biotic systems. With the advent of new technologies, the area of research has been ever growing, with some of the primary disciplines today being sequence analysis, gene and protein expression, analysis of cellular organisms, etc.

Thanks to the dramatic reduction in operating costs and improvements in data processing, sequence assembly has been a greatly popularized field within bioinformatics over the last twenty years or so. Sequence assembly can be described as a process of combining small DNA fragments received from a sequencing device into a full-fledged DNA sequence. These fragments are ordered by overlapping them with one another and seeing which ones fit together best, much in the same way as one goes when assembling a puzzle. Of course, there are specific issues with this process, and even though today's sequencing devices have made great leaps from their first generation, some of these fragments will still end up wrongly detected or malformed. These fragments tend to wreak havoc on the assembly process. Knowing which fragments to include in the assembly and which to skip is crucial for a successful result. More on the types of malformed fragments and the problems they cause can be found in chapter 2.

It is because of these issues that much research has been directed into discovering methods and techniques for the detection and removal of these fragments. Some of these methods are based on the use of deep learning algorithms. Through the use of long chains of complex nonlinear functions, these algorithms try and detect hidden characteristics and patterns in the underlying data in order to apply them to various learning tasks, such as classification, clustering or dimensionality

reduction. The specifics of the deep learning algorithms used in this thesis are described in chapter 3.

Chapter 4 concentrates on the data used, how it is initially formatted, what it represents and what transformations are applied to convert it into a form suitable for feeding into a deep network. Finally, the results and final discussion can be found in chapter 5.

2. DNA sequencing and assembly

The process of determining the order of nucleotides in DNA is called DNA sequencing. Knowing the inner workings of an organisms DNA is the key to understanding its genetic function and evolution. Data obtained by this process is now widely used in even the most basic biological research, while also being applied in various other fields such as medical diagnosis, forensic biology, or virology. For example, by comparing a healthy to a mutated DNA sequence, doctors and scientists can diagnose a host of various cancers and genetic disorder [7].

To be able to reconstruct a DNA sequence fully, one must complete two steps. Firstly, through the usage of a sequencer[1], generate fragments of DNA called reads. The main problem with this is the length of the sequence that is being examined, with some of the longer ones spanning into the tens of billions. The most advanced sequencers today can only read some 100.000 base pairs at a time with a satisfying degree of accuracy. This characteristic is why their output is a collection of reads and not a single sequence. For this reason, a second step is required called genome assembly, which takes the created reads and uses them to reconstruct the original sequence.

2.1. Overlap-Layout-Consensus

To assemble the reads received from the sequencer into a single DNA sequence again, a three-step approach is used (Figure 2.1). These steps are:

1. Overlap
2. Layout
3. Consensus

[1] *Sequencer* - Scientific instrument used to determine the order of nucleotides in a DNA sample

In the first step, all the reads are taken and an overlap graph is built to indicate the number of overlapping consecutive base pairs between reads. This is also the most resource-consuming step of the assembly process. The layout step is concerned with simplifying the previously created overlap graph by removing all edges that can be transitively inferred from other edges, followed by bundling stretches of the graph into contiguous blocks called contigs. The final step takes the contigs and picks the most likely nucleotide sequence for each of them, while also handling errors created by the sequencer when creating reads. Contigs that are identified to be connected and order between them is defined are grouped in blocks called scaffolds. The end result of the Overlap-Layout-Consensus method is a collection of one or more scaffolds, depending on the successfulness of all the previous steps.

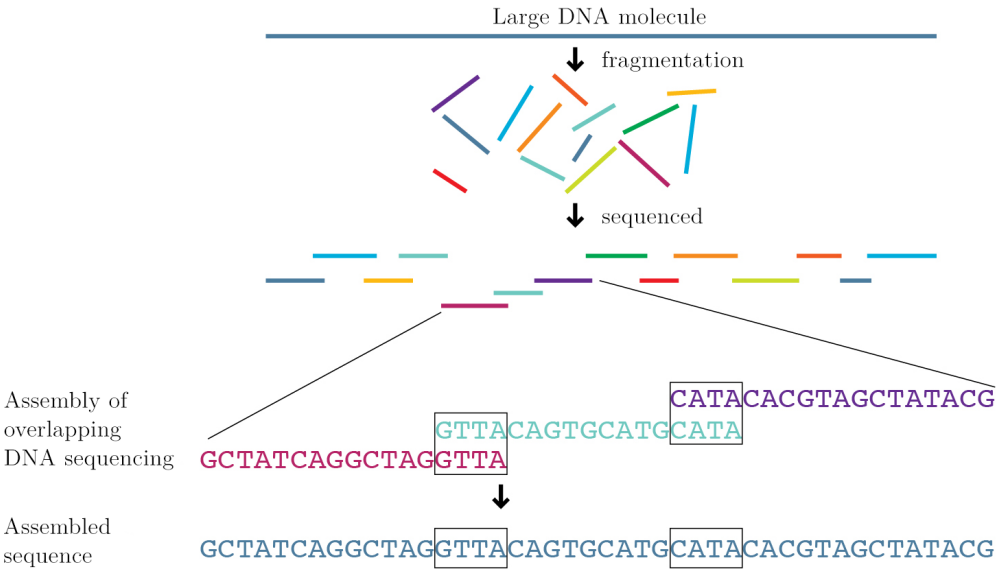


Figure 2.1: High level overview of the OLC approach to *de novo* sequencing. Source: [9]

2.2. Read types

The most significant issues in the genome assembly process described in the previous section have to do with the overlapping step. The number of overlaps of

each base pair in a single read with base pairs from other reads can be visualized by creating a coverage chart. The general assumption here is that the discriminative features of a read translate to its chart, which by extension means that classifying a chart also assigns the same type class to its associated read. There are four mayor read types that should be examined.

Regular reads are ones that have no outstanding features visible in their charts, with the majority of overlaps being in a relatively narrow range of values (the order of this range should not be of importance when ordering the read type). This means that the shape of the signal is more or less flat throughout all base pairs (see Figure 2.2).

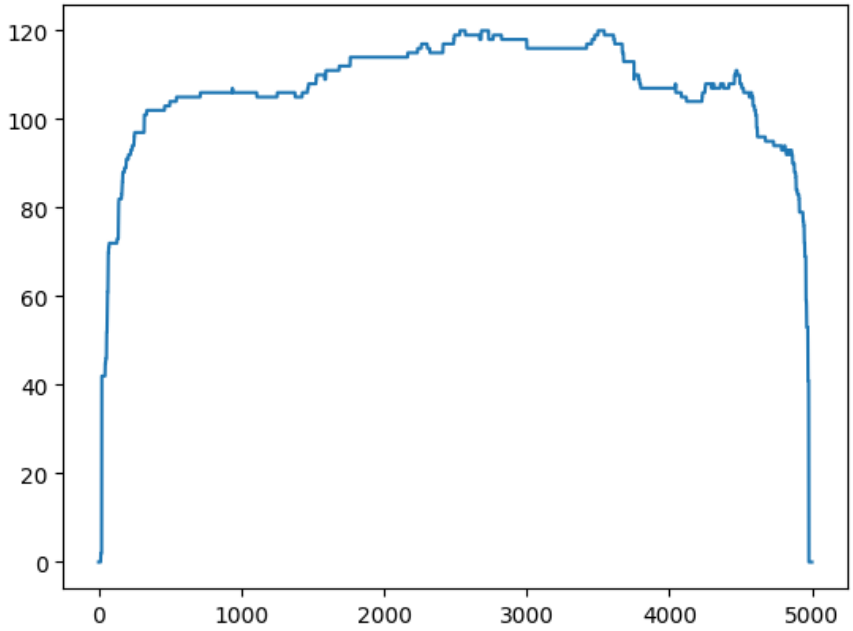


Figure 2.2: Regular coverage chart

Repeat reads have one or more characteristic sections on their coverage charts. These sections indicate to the presence of a set of consecutive base pairs that repeat themselves at various positions throughout the referenced genome. Even though they represent valid data (meaning that they can be found in the original sequence), they vastly complicate the assembly process. From a computational perspective, repeats create ambiguities in alignment, which, in turn, can produce biases and errors when interpreting results. Simply ignoring repeats is also not an option, as this creates problems of its own and may mean those important biological phenomena are missed [20]. These ambiguities are most

prominent when very short reads are received from the sequencer. If there are no reads long enough to reach over a repeating region into regular regions before and after it, the system will not be able to tell which read should go where in the final sequence (see Figure 2.3).

Chimeric reads are ones that give out the most concern, as, unlike the previous types, they represent invalid data. These reads are the result of malfunctions in the sequencer which make it skip a part of the sequence and, by doing so, create a read that contains base pairs that are in reality set far apart from each other. It is for this reason that reads like these should be discovered and removed before the assembly process so that they do not corrupt the resulting sequence. Making their discovery somewhat tricky is the fact that chimeric reads tend to be few and far between in most cases, which is why researchers are heavily focused on discovering new ways for their prediction and removal. The upside is that these reads have very characteristic coverage charts containing a noticeable spike in the signal at the location where the two sections make contact with each other. The sections on both sides of the spike have a significant amount of overlap compared to other reads, but almost no overlap exists along their connecting region, thus manifesting itself as a negative spike in the coverage signal, as shown in Figure 2.4b.

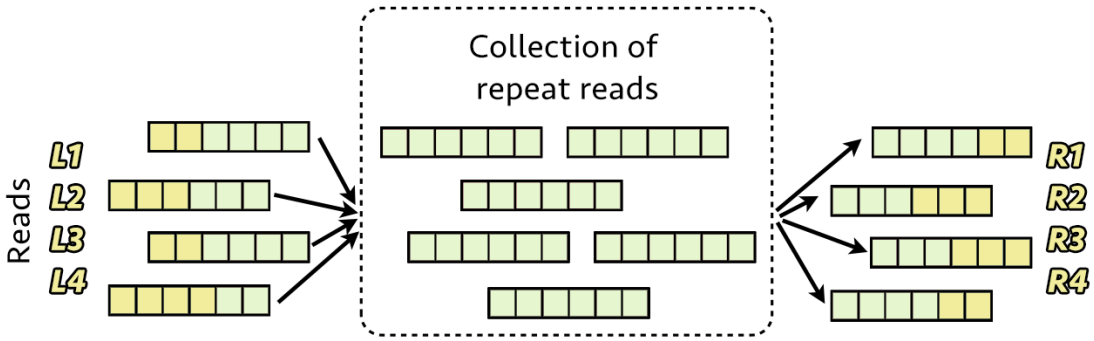
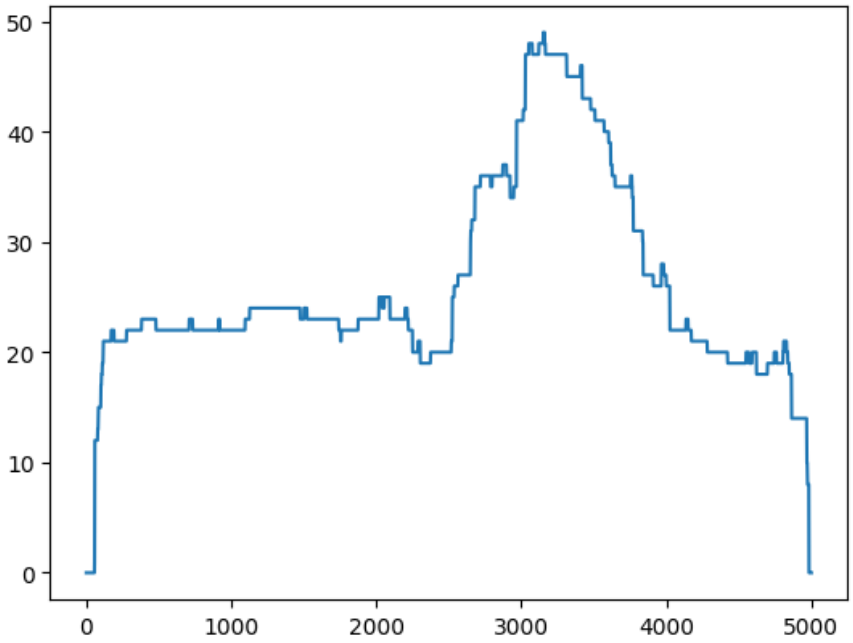
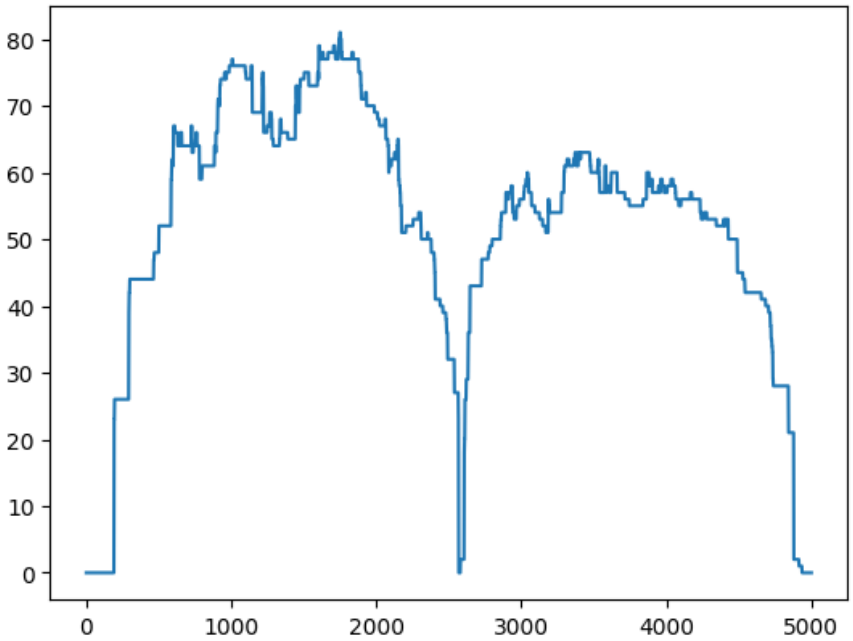


Figure 2.3: Repeat reads assembly ambiguities



(a) Repeat read coverage chart



(b) Chimeric read coverage chart

Figure 2.4: Irregular coverage charts

3. Deep learning methods

Deep structured learning, or more commonly called deep learning or hierarchical learning, has emerged as a new area of machine learning research during the past decade or so. It is the intersection of neural networks, artificial intelligence, optimization and pattern recognition. Due to this interleaved structure, there isn't one single definition of deep learning to quote on, though it is usually defined as a sub-field of machine learning that exploits many layers of non-linear information processing in order to model complex relationships among data. Common to most high-level definitions are two key aspects. First, that deep models consist of multiple layers of nonlinear operations, and second, that the learning methods of feature representation are based on successively higher abstraction levels [3].

Within the field of machine learning in general, there are two main types of tasks - supervised and unsupervised ones, with the main difference between the two being the existence of prior knowledge of the output values. Therefore, the goal of supervised learning is to create the best approximation of the relationship between the input and output values, while unsupervised learning aims to infer the natural structure present in the input data.

A separate categorization can also be done in aspect to model types, with the ones considered here being classification, regression, clustering and dimensionality reduction. With regard to the previously mentioned learning types, classification and regression are considered to be supervised learning models. By doing classification, the model learns to assign classes (labels from a predefined set) to samples of input data. Regression models, on the other hand, return a continuous variable instead of a label, which makes them useful for the detection of relationships between variables. The last two model types, clustering and dimensionality reduction, are regarded as unsupervised models. With clustering, the objective is to find different groups (clusters) within the given data by structuring off samples that are the most similar to each other. Lastly, dimensionality reduction

is defined as a process of reducing the number of variables under consideration by obtaining a set of principal variables through the use of feature selection and extraction [16].

3.1. Algorithm components

3.1.1. Loss function

The cost or loss function has an important job in that it must faithfully distill all aspects of a model down into a single number in such a way that improvements in that number are a sign of a better model. It is essential, therefore, that the function represents our design goals to a high degree. If a poor error function is chosen which obtains unsatisfactory results, the fault comes down to the researcher for badly specifying the goal of the search.

Unfortunately, there is no *one-size-fits-all* loss function. There are various factors involved in choosing one, such as the type of the algorithm or the ease of calculating derivatives. Broadly speaking, they can be divided into two categories depending on the type of learning task, resulting in either regression or categorization losses.

Cross-entropy loss

The loss function used with all models in this thesis is the cross-entropy loss or the negative log-likelihood. Before the actual function is fully defined, a couple of key definitions and assumptions must be made.

A function that maps an observation to a label is called a classifier and is usually defined as a model f_w with parameters $\mathbf{w}$. The process of training a model (finding the optimal parameters) is then obviously done by maximizing the likelihood of the model's parameters. If it is assumed that all samples are independent and identically distributed, then the likelihood over all examples can be decomposed as a product of likelihoods of individual samples (with $y^{(n)}$ representing the n-th ground-truth label, and $\hat{y}^{(n)}$ representing the n-th predicted label) [23].

$$L\left(\left\{y^{(n)}\right\}, \left\{\hat{y}^{(n)}\right\}\right) = \prod_n L\left(y^{(n)}, \hat{y}^{(n)}\right) \tag{3.1}$$

The second assumption states that all class labels are one-hot encoded, represented as vectors of N elements where only the corresponding index element is equal to 1, while all others are set to 0. If this is true, the likelihood of the n-th example is its exact predicted value:

$$L\left(y^{(n)}, \hat{y}^{(n)}\right) = y_n^{(n)} \hat{y_n}^{(n)}, n = 1, ..., N, y_n^{(n)} = 1 \tag{3.2}$$

Returning to the likelihood defined in 3.1, applying a monotonic function, such as a logarithm, to it does not change the original maximization goal. This, in turn, is equivalent to minimizing the negative log likelihood, and when taking into account 3.2, the loss functions final form is:

$$-\log L\left(\left\{y^{(n)}\right\}, \left\{\hat{y}^{(n)}\right\}\right) = \sum_n H\left(y^{(n)}, \hat{y}^{(n)}\right) \tag{3.3}$$

3.1.2. Optimization

As stated in the previous section, the priority goal of a deep learning model is to minimize a loss function, whose value then determines the final performance of the entire network on a given dataset. Thus, one can think of deep learning as just a large optimization problem, essentially a complex mathematical function with thousands or even millions of parameters. Accounting for the complexity and non-linearity of the entire neural network architecture, and it becomes clear that the solution will not be available in closed form[1]. It is for this reason that a different approach is needed, namely by using iterative optimization methods like gradient descent and its descendants.

All optimization methods share one common trait, requiring the calculation of the derivatives of the loss function L with respect to layer parameters. The coupling of parameters between layers can make this process quite complex (primarily as a result of using the product rule), and if not implemented cleverly, can make the final gradient descent calculations painfully slow. It is for this reason that a particular algorithm is introduced to calculate these derivatives, called backpropagation.

When using a feedforward neural network to accept an input $\boldsymbol{x}$ and produce an output $\boldsymbol{y}$, information flows forward through the network. The input provides the initial information that then propagates up to the hidden units at each layer and

[1] A mathematical expression that can be evaluated in a finite number of operations

finally produces the output result. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar loss. The back-propagation algorithm allows the information from the loss to then flow backward through the network in order to compute the gradient. The term backprop is often misunderstood as meaning the entire learning algorithm, but it actually refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient [6].

These learning algorithms, also called optimization algorithms, use the gradient of the loss function to tweak the network parameters in order to minimize the final loss value. A gradient is a vector which shows the direction of the functions rise, requiring the algorithm to move in the opposite direction while searching for the minima. This search does not come without a few challenges, mainly concerning local minimums, saddle points and vanishing gradients. As it was the case with the loss function, no optimization algorithm is perfect, but it has become common practice today to use some of the more advanced adaptive algorithms to try and overcome these issues, with one of them being Adam [11]. According to the original authors, the bias-correction introduced in Adam helps it to slightly outperform RMSprop, another advanced adaptive algorithm, towards the end of optimization as gradients become sparser, making it the currently the best overall choice [17].

3.2. Common operations

3.2.1. Pooling

The role of a pooling layer is to aggressively downsample input feature maps by applying a predefined mathematical operation. Ordinarily, these operations either take the mean or the maximum value of data inside a moving window. Pooling is commonly applied after a convolution layer in order to progressively reduce the number of parameters and computation in the network, which allows for control over overfitting. In all cases, pooling helps to make the representation become invariant to minor translations of the input, becoming more adept at detecting a specific feature, rather than binding a feature to a specific location inside a sequence.

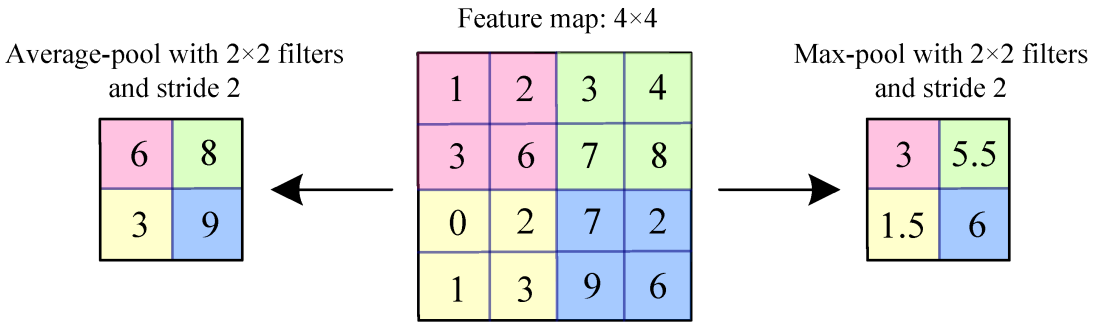


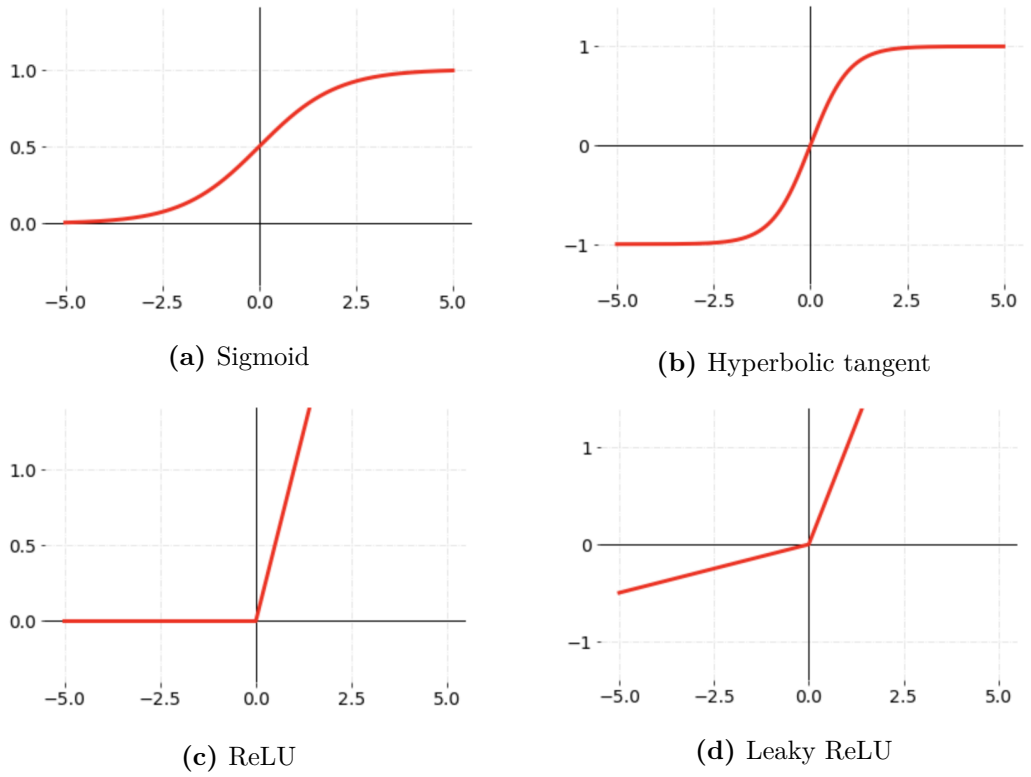
Figure 3.1: Average and max pooling on a 4x4 feature map. Source: [5]

3.2.2. Activations

The activation function can be seen as a mathematical transition between the input and output of a given neuron. A deep network without an activation function boils down to a linear regression model, which has limited performance, especially when it comes to complex tasks. Since activation functions are at their core simply nonlinear mathematical functions, introducing them into the network architecture allows for the computation of nontrivial problems. A good activation function should then be nonlinear, preferably continuously differentiable (for successful backpropagation) and monotonic. Some of the most used nonlinear activations include the sigmoid, hyperbolic tangent, ReLU (Rectified Linear Unit) and its derivatives (Figure 3.2).

3.3. Convolutional Neural Networks

The fundamental difference between a fully connected layer and a convolution layer is that the former learn global patterns in their input feature space (e.g., taking in a whole image at once), whereas the latter learn local patterns. They have been of great use in image recognition, thanks to their ability to detect and recognize various complex hierarchies and patterns. Lower layers learn small local patterns such as edges, while higher layers learn larger patterns made up of the features of the previous layers, and so on [2]. Another advantage of convolutions is the number of parameters used by each layer. A fully connected layer, as the name suggests, connects every input to every output, with each connection having its own weight parameter. Comparing this to local connections in a convolution layer, in which only a handful of inputs control a single output, it is clear that they relieve much of the heft that comes with using fully connected layers, hence



(a) Sigmoid

(b) Hyperbolic tangent

(c) ReLU

(d) Leaky ReLU

Figure 3.2: Nonlinear activation functions. Source: [19]

speeding up the learning process (see Figure 3.3).

A convolution layer is defined by four hyperparameters - filter size, output depth, stride and padding. A filter is essentially a group of weights with which the input features are convolved. The filter size defines the number of neighboring elements for a single position, while the output depth simply states how many filters have been defined. Even though the filter size and number are hyperparameters, the weights inside them are not, starting off with random values, with the learning algorithm modifying them during training.

To process a single input map, the filter slides around it and computes the sum of the elements it covers, with each element multiplied by its corresponding weight from the filter. An additional bias term is also added, and a single number, the weighted sum, is produced for each position that the filter slides over (see Figure 3.4 for a visual example).

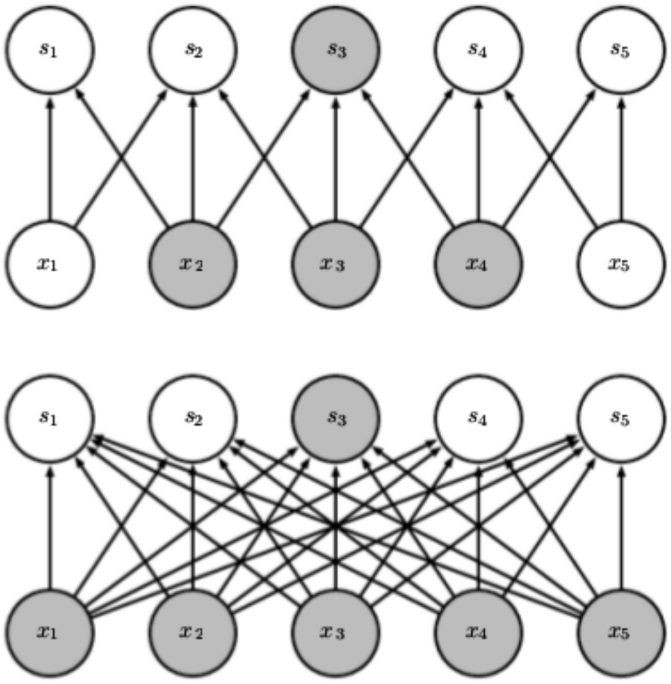


Figure 3.3: Convolutional and fully connected layer comparison. Source: [6]

Padding and strides

Using an out-of-the-box convolution results in an output feature map of smaller spatial dimensions compared to the input. This can be avoided, if required, by applying padding. Padding, another hyperparameter, consists of adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit the center convolution windows around every input tile. It is common to fill the padded area with zeros so to not affect the output values. In addition to the aforementioned benefit of keeping the spatial sizes constant, applying zero-padding actually improves performance and prevents the information at the borders being ignored too quickly [10].

The other factor that can influence the output size is strides. A stride is the distance between two successive windows and is also a hyperparameter of a convolution layer. By using a stride greater than one, the width and height of the feature map end up downsampled by this factor (in addition to any changes induced by border effects). Strided convolutions are rarely used in practice though, despite that they can come in handy for some types of models. In the end, the most reasonable subsampling strategy is to first produce dense maps of features and then look at the maximum activation of the features over small patches,

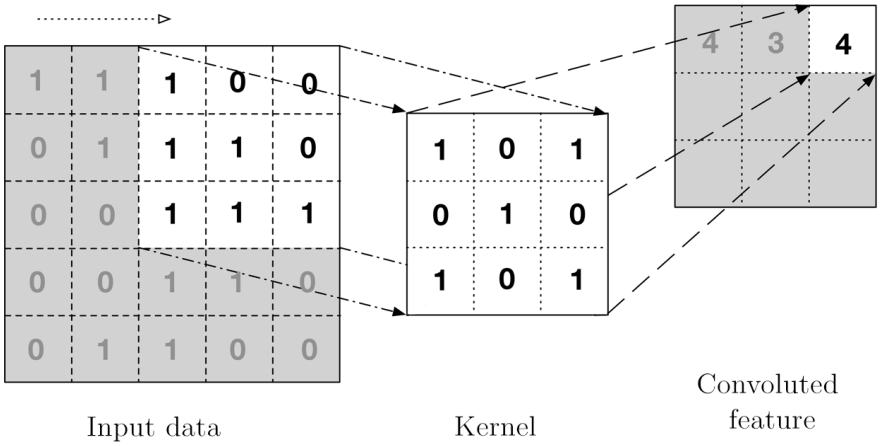


Figure 3.4: Example of a convolution operation with a 3x3 filter. Source: [15]

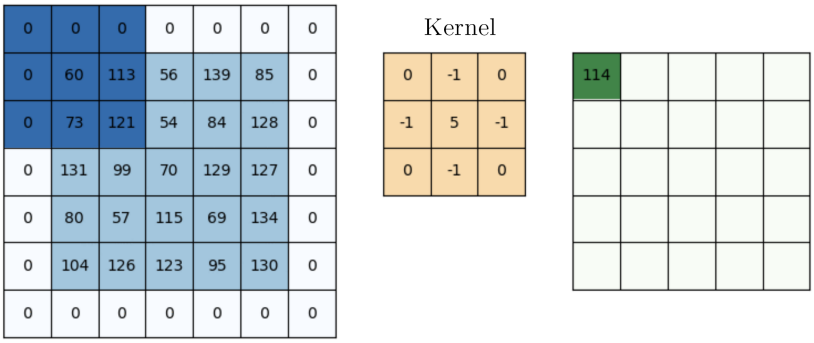


Figure 3.5: Usage of padding in preservation of output dimensions

rather than looking at averaged or global input patches, which could result in missed or diluted feature information.

3.3.1. One-dimensional CNNs

CNNs share the same characteristics and follow the same approach, no matter if they are 1D, 2D or 3D. The only differences are, clearly, in the input dimensionality and the way that filters slide over the data. The same properties that make convnets excel at computer vision also make them highly relevant to sequence processing. Time can be treated as a spatial dimension, like the height or width of a 2D image. This means, that even though the name suggests that the input data should have a single dimension, the network is set up to requires a 2D tensor. The first dimension defines the number of time steps in a single example, which in layman's terms stands for the total length of a 1D input vector. The other

parameter defines the number of features for each step. Taking again a 1D input vector, this parameter is set to 1, as there is only a single numeric value for every step.

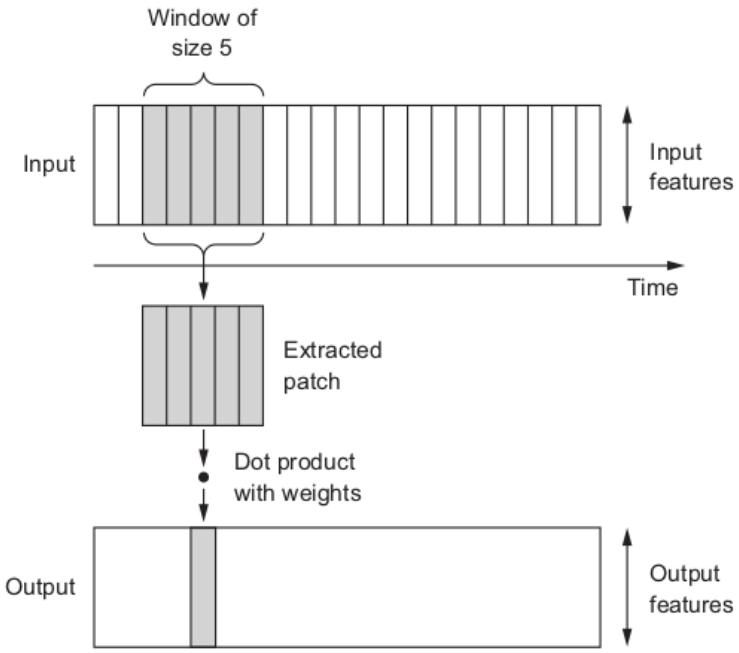


Figure 3.6: 1D convolutional layer. Source: [2]

One difference, though, is the fact that you can afford to use larger convolution windows with 1D CNNs. With a 2D convolution layer, a standard 3x3 convolution window contains nine feature vectors; but with a 1D convolution, a window of size 3 contains only 3 feature vectors, allowing for the use of larger values in 1D networks.

3.4. Recurrent Neural Networks

Fully connected and convolutional networks that have been described so far have a common characteristic, in that they have no memory. What this means is that each input is processed independently, with no state kept in between. Recurrent Neural Networks are similar to the human brain, which is a large feedback network of connected neurons that somehow can learn to translate a lifelong sensory input stream into a sequence of useful outputs.

An RNN takes in a three-dimensional tensor as its input, with two of those

three being quite similar to the 1D CNN input defined in the section above. The first dimension again defines the number of time steps (sequence length), and the second defines the number of features in a single step. The third dimension is new and shows the number of examples in a mini-batch, basically the number of input records (collections of points for a single source entity) to model per batch.

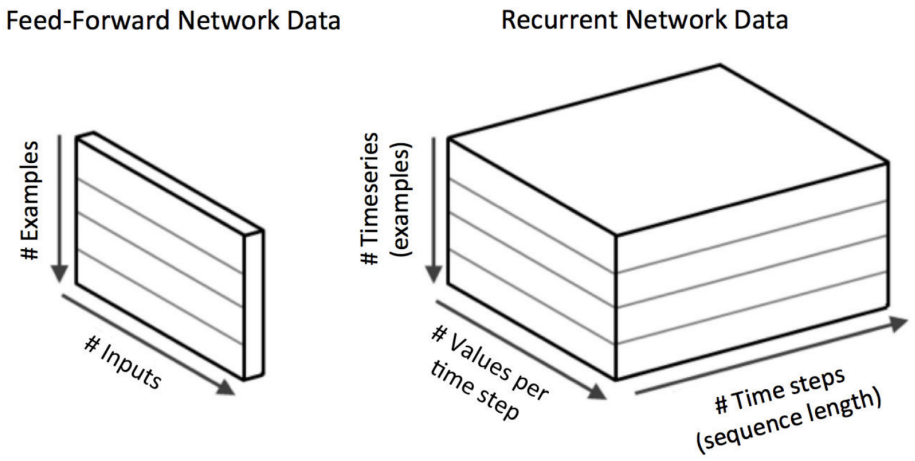


Figure 3.7: Comparison of normal an RNN input vectors. Source: [15]

Recurrent networks take as their input, not just the current example, but also what they have perceived previously in time. The decision a network made at time step $t-1$ affects the decision it will reach later at time step t. This means that every RNN cell has two input sources, the present, and the recent past, which combine into a unified hidden state that serves as "memory". A single cells hidden state at time t is then defined as:

$$h^{(t)} = \theta_h \left(W h^{(t-1)} + U x^{(t)} + b \right) \tag{3.4}$$

The $\boldsymbol{U}$ and $\boldsymbol{W}$ weight matrices behave similarly to CNN filters, determining how much importance to accord to both the present input and the past hidden state. The final loss will return via backpropagation and be used to adjust their weights during the learning process. A bias vector $\boldsymbol{b}$ can also be defined and added to the hidden state. Unlike the previously considered networks, RNNs share their weight matrices between cells, meaning that for an entire layer with multiple connected cells, only a single instance of both the $\boldsymbol{U}$ and $\boldsymbol{W}$ matrices exist at one time [22].
For the sake of simplicity, the architecture of the first cell in the network is kept

the same, with only the initial hidden state being set to zero:

$$h^{(0)} = 0 \tag{3.5}$$

The projection of each cell into the output layer is then calculated as:

$$o^{(t)} = \theta_o \left(V h^{(t)} + c \right) \tag{3.6}$$

As was the case with the hidden state equation, a third weight matrix $\boldsymbol{V}$ (shared throughout the cells) is used to filter the hidden state before passing it through to the output layer. The θ_o activation function can be applied here if necessary; otherwise, it simply defaults to the identity function. Lastly, if the network is tasked with classification, the output probability is simply a softmax function applied over the output layer:

$$\hat{y}^{(t)} = \text{softmax} \left(o^{(t)} \right) \tag{3.7}$$

Theoretically, any of the activation functions could be used for the θ_h and θ_o functions applied to the hidden state and the output projection, but the hyperbolic tangent is usually the preferred option. The main reason for this is to prevent a combinatorial explosion by squashing the values into a $[-1, 1]$ range. A hidden state activated by a ReLU function is defined as:

$$h^{(t)} = \text{ReLU} \left(W h^{(t-1)} + U x^{(t)} + b \right)$$

$$t \to T \gg 1 \qquad x^{(t)} = 0 \qquad b = 0 \qquad h_0 = 1$$

After T steps, disregarding the current step input and bias, the hidden state enters the combinatorial explosion phase:

$$h^{(T)} = w^T, \qquad w > 0$$

3.4.1. Backpropagation through time

The problem with applying regular backpropagation to an RNN is its cyclic nature. Unlike regular feed-forward networks which are acyclic and unidirectional and for which the loss derivatives could be calculated from the layer above, RNNs do not have such layering. The equation for calculating the hidden state loss derivative for a single RNN cell is defined as:

$$\frac{\partial \mathbb{L}}{\partial h^{(t)}} = \frac{\partial \mathbb{L}^{(t)}}{\partial h^{(t)}} + \frac{\partial \mathbb{L}^{(t+)}}{\partial h^{(t)}} \tag{3.8}$$

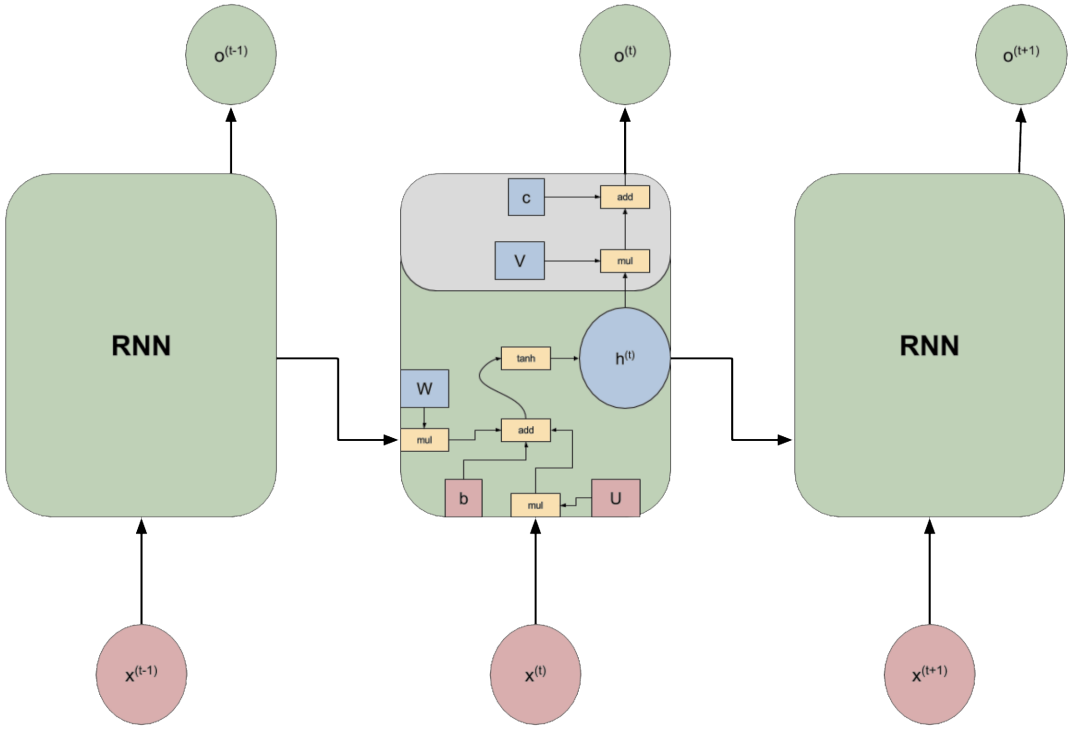


Figure 3.8: Structural diagram of an RNN cell. Source: [22]

By examining this equation (3.8), it becomes clear that every hidden state loss derivative depends on all upstream gradients from its successor all the way to the last cell in the layer. As this dependency is unique to RNNs, a modified back-propagation algorithm, called backpropagation through time (BPTT), is used for training.

At its core, BPTT is just a regular backpropagation, with a subtle twist at the first step. It begins by unfolding the RNN in time. Then the plain backpropagation algorithm is used to find the gradient of the cost with respect to all the network parameters, with the algorithm starting out from the last cell and working its way up to the first one [22].

3.4.2. Long Short-Term Memory

One of the appeals of RNNs is the idea that they should be able to connect previous information to a present task. For most tasks, it is usually enough to look at recent information to successfully perform them, but there are some tasks that require more context and longer-term dependencies, without which they can

not perform as it would be expected. In theory, standard RNNs are more than capable of handling such dependencies, albeit with carefully picked parameters. Unfortunately, in practice, it has been shown that RNNs struggle with grasping these dependencies [1].

To tackle the long-term dependencies problem, a new type of RNN was created, called the Long Short-Term Memory network (LSTM). Introduced by Hochreiter and Schmidhuber in 1997, they are explicitly designed to remember information for long periods of time [8]. Owing to this is the modified structure of a single cell, which now has four layers inside it, instead of the regular single layer in a plain RNN (Figure 3.9).

The core component of an LSTM cell is the so-called cell state. One can imagine it as a conveyor belt of sorts, running parallel to the sequence being processed, with information jumping onto the conveyor to be stored for later use, and jump off again when needed. The jumping of information onto the cell state is controlled by a set of three activation gates that define how much information is being added or removed from the state. Mathematically, the cell state is defined as:

$$s^{(t)} = f^{(t)} \odot s^{(t-1)} + g^{(t)} \odot \hat{s}^{(t)} \tag{3.9}$$

The first of the three gates, called the forget gate layer, controls what information should be removed from the cell state. Its task is to examine the previous step h_{t-1} and the current input x_t and apply a simple sigmoid to decide which information from $s^{(t-1)}$ should be canceled out.

$$f^{(t)} = \sigma\left(W_f h^{(t-1)} + U_f x^{(t)} + b_f\right) \tag{3.10}$$

Deciding what information to add to the cell state is the job of the next layer, aptly named the input gate layer. The equation is defined in the same way as the forget gate layer, just using different weight parameters (3.11). An additional candidate state is calculated by applying the hyperbolic tangent to the inputs (3.12). This new state is then combined with the input gate layer and used to create the new cell state as stated in equation 3.9.

$$g^{(t)} = \sigma\left(W_g h^{(t-1)} + U_g x^{(t)} + b_g\right) \tag{3.11}$$

$$\hat{s}^{(t)} = \tanh\left(W_s h^{(t-1)} + U_s x^{(t)} + b_s\right) \tag{3.12}$$

Finally, it is time to define the output of an LSTM cell. Firstly, the cell inputs are run through yet another sigmoid gate to decide what part of them should be output (3.13). The cell state passes through a hyperbolic tangent to push its values into a $[-1, 1]$ range. Results of these two steps are then multiplied to create the final output of the cell (3.14) [14].

$$q^{(t)} = \sigma\left(W_q h^{(t-1)} + U_q x^{(t)} + b_q\right) \tag{3.13}$$

$$h^{(t)} = q^{(t)} \odot \tanh\left(s^{(t)}\right) \tag{3.14}$$

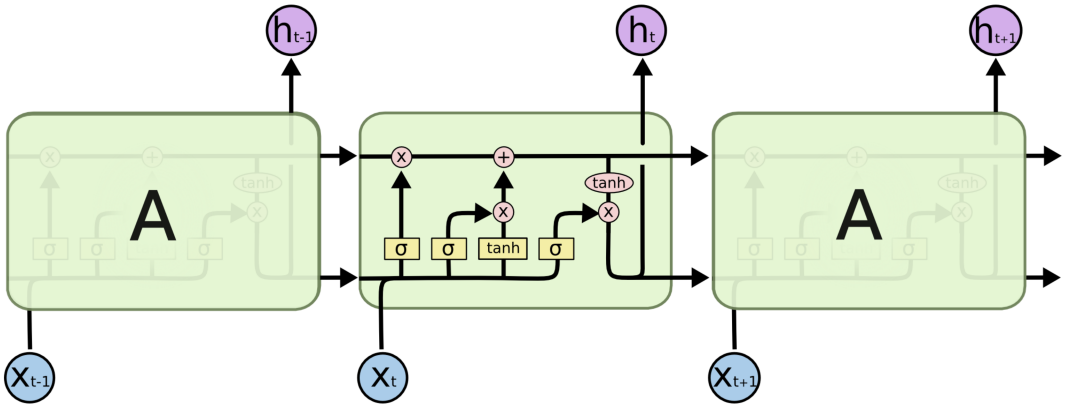


Figure 3.9: Structural diagram of an LSTM cell. Source: [14]

3.5. Autoencoders

Up until now, all described networks have fallen under the category of supervised learning. The thing is, clean, perfectly labeled datasets are not that easy to come by. The number of unlabeled or partially labeled samples is often larger than the number of labeled samples since the former are less expensive and easier to obtain. A never-ending problem of supervised learning algorithms is not having enough labeled data. One way of addressing this problem is by adding cheap and abundant unlabeled data, hoping that it will help build a better model than just by using supervised learning. The approach of using both a smaller set of labeled and a larger set of unlabeled data is called semi-supervised learning.

Autoencoders are defined as neural networks that are trained to attempt to copy their input to their output. Each autoencoder may be viewed as consisting

of two parts: an encoder function $h = f(x)$ and a decoder that produces the reconstruction $r = g(h)$. Even though they are tasked with learning to copy input to the output, If they somehow manage to set $r = g(f(x)) = x$ everywhere, then they stop being especially useful. Instead, they are designed to be unable to learn a perfect copy. Because the model is now forced to prioritize which aspects of the input it should copy, it tends to learn useful patterns and properties in the data along the way. One way to obtain useful features in the latent layer is to constrain it to have a smaller dimension than the input/output layers. An autoencoder whose latent dimension is less than the input dimension is called an under-complete autoencoder. Learning an under-complete representation forces the autoencoder to capture the most salient features of the training data.

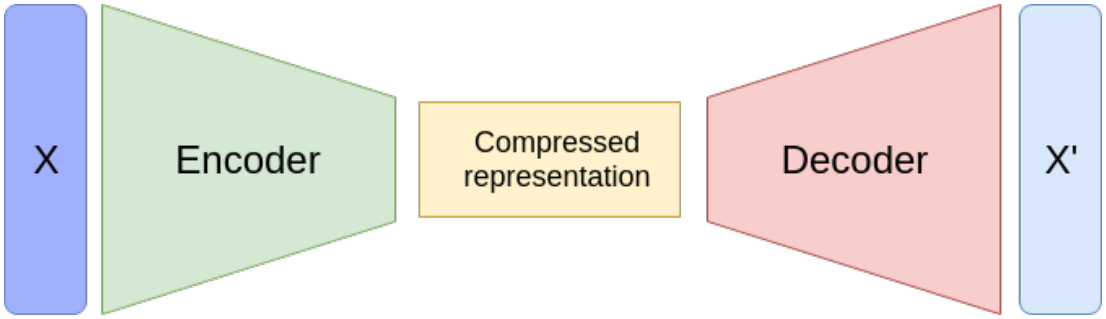


Figure 3.10: Representation of a basic autoencoder. Source: [13]

Because neural networks are capable of learning nonlinear relationships, autoencoders can be thought of as a more powerful nonlinear generalization of PCA[2]. Whereas PCA attempts to discover a lower dimensional hyperplane which describes the original data, autoencoders are capable of learning nonlinear manifolds (continuous, non-intersecting surfaces).

Training autoencoders is done the same way as with all other deep networks, by minimizing a predefined loss function, which usually correlates with the difference between the input and its reconstruction. For this correlation to be true, the encoder and decoder need to cooperate, so that the encoder extracts as many useful features from the input data, which the decoder then uses to provide a reconstruction as similar as possible to the input.

[2]Principal Component Analysis — Mathematical procedure that transforms a number of correlated variables into a smaller number of uncorrelated variables (principal components)

Even though many implementations use a simple shallow autoencoder with only a single layer encoder/decoder, this does not have to be the case. Recent studies have shown that increasing the depth of the hidden layers can result in reduced computational cost and the amount of training data required [6]. There are also many other types of autoencoders out there, with the most popular ones being denoising, sparse, convolutional and variational autoencoders.

3.5.1. Variational autoencoders

One of the more advanced types of autoencoders is the variational autoencoder (VAE). They are called "autoencoders" only because the final training objective that derives from this setup does have an encoder and a decoder, and resembles a traditional autoencoder [4].

Suppose that every example x can be generated through a random variable z. More formally, if z is conforms to a distribution $p(z)$, then x can be sampled from the conditional distribution $p(x|z)$. The problem with this is that only x is known, with the characteristics of z attempting to be inferred. Mathematically speaking, $p_\theta(z|x)$ needs to be computed, with it being equal to:

$$p(z|x) = \frac{p(x|z)\ p(z)}{p(x)} \tag{3.15}$$

As $p(z)$ is an independent distribution, it can be safely assumed and is chosen to match the normal distribution $\mathcal{N}(0, 1)$. A normal distribution is used for its simple formula and the ability to generate any other distribution from it [4]. Regarding $p(x)$, the goal during training is to maximize the probability of each example in the dataset using the following equation:

$$p(x) = \int p(z)\ p_\theta(x|z)\ dz \tag{3.16}$$

Looking at this equation, a problem arises. The integral used proves to be analytically intractable, meaning that the resulting distribution is also intractable. Propagating this issue back to 3.15, it turns out that sampling z from the conditional distribution $p(z|x)$ cannot be done in the proposed manner. The solution to this is to try and approximate it by using another distribution $q_\phi(z|x)$. If the parameters can be defined in such way that this distribution is very similar to $p_\theta(z|x)$, then it can be used to approximately infer the intractable distribution

$p(z|x)$. For the representation of this distribution, a multinomial normal distribution can be used, with its parameters usually determined by a deep network. Finally, this replacement distribution can then be written as:

$$q_\varphi(z|x) = \mathcal{N}\left(g_\mu(x;\ \varphi),\ g_\sigma(x;\ \varphi)\right) \tag{3.17}$$

With the distributions defined, the encoder part of the VAE will be handling the learning of $q_\phi(z|x)$, while the decoder will be assigned with learning $p_\theta(x|z)$.

The next task is to find the optimal values for ϕ and θ. The way the model is currently set up, it is exposed to a certain degree of variation when encoding a single sample, resulting in smooth latent spaces on a local scale, at least for similar samples. Ideally, there should also be some overlap between samples that are not very similar, in order to be able to interpolate between classes later on efficiently. However, since no limits have been set on the sampled μ and σ, the encoder could technically learn to generate very distant means for different classes, clustering them apart and with minimal deviation, effectively cordoning off classes from each other. In a perfect world, encodings should be as close as possible to each other while still being distinct, allowing for smooth interpolation and construction of new samples. Training VAEs with regular techniques will not honor the described behavior, meaning that it must be somehow added manually.

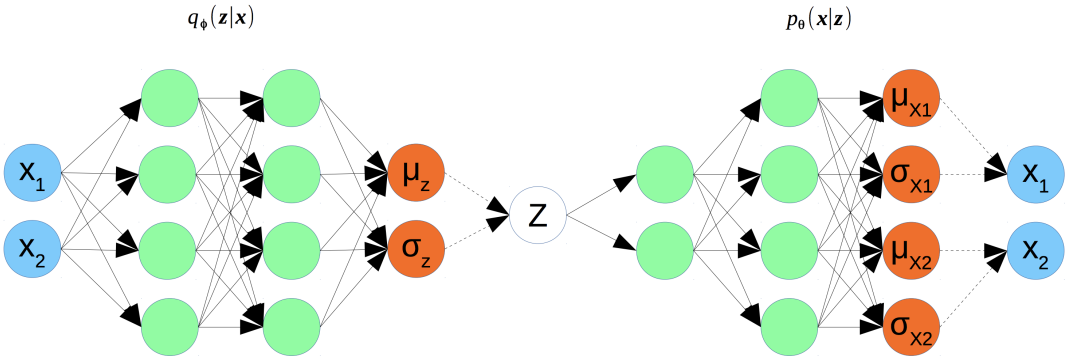


Figure 3.11: Variational autoencoder architecture (dashed lines represent sampling from a distribution). Source: [22]

Optimizing the objective

The original loss is a straightforward maximum log-likelihood (MLE), which has already been explained in detail in section 3.1.1.

$$\log(p(x)) = \sum_z q(z|x)\ \ \log(p(x))$$

$$= \sum_z q(z|x) \ \log\left(\frac{p(z,x)}{p(z|x)}\right)$$

$$= \sum_z q(z|x) \ \log\left(\frac{p(z,x)}{q(z|x)} \cdot \frac{q(z|x)}{p(z|x)}\right)$$

$$= \sum_z q(z|x) \ \log\left(\frac{p(z,x)}{q(z|x)}\right) + \sum_z q(z|x) \ \log\left(\frac{q(z|x)}{p(z|x)}\right)$$

$$L = L_v + D_{KL}(q(z|x) \parallel p(z|x)) \tag{3.18}$$

Interestingly enough, the log probability can, through some mathematical transformations, be decomposed into two separate components, the lower variational bound L_v and the Kullback-Leibler divergence D_{KL}. The Kullback-Leibler (KL) divergence is defined as the measure of how one probability distribution is different from a second. In the context of Bayesian inference, it is the amount of information lost when one distribution is used to approximate the other. Assuming a high enough precision, the two distributions will be identical and the KL divergence will equal zero [22].

To maximize the overall likelihood, the lower variational bound L_v must also be maximized. Therefore, further decomposition is necessary:

$$L_v = \sum_z q(z|x) \ \log\left(\frac{p(z,x)}{q(z|x)}\right)$$

$$= \sum_z q(z|x) \ \log\left(\frac{p(x|z) \ p(z)}{q(z|x)}\right)$$

$$= \sum_z q(z|x) \ \log\left(\frac{p(z)}{q(z|x)}\right) + \sum_z q(z|x) \ \log(p(x|z))$$

$$= -D_{KL}(q(z|x) \parallel p(z)) + \mathrm{E}_{q(z|x)}(\log(p(x|z))) \tag{3.19}$$

The latter component represents the quality of reconstruction and is essentially the probability that x is sampled from $p(x|z)$ with z being from $q(z|x)$, while the former shows how much the conditional distribution $q(z|x)$ differs from the prior $p(z)$. This KL divergence can be considered as a regularization factor in the final loss function [6].

Maximizing the lower variational bound as defined in 3.19 using backpropagation leads to yet another obstacle. It is impossible to calculate the gradient

of the loss function through a nondeterministic sampling function because the algorithm has no clue how to calculate a gradient at this point. To overcome this problem, the nondeterministic sampling function can be approximated by firstly randomly sampling ε from a standard Gaussian $\mathcal{N}(0,1)$, and then adding it to the latent distribution's mean and variance in the following manner:

$$z = \mu + \sigma \odot \varepsilon \tag{3.20}$$

This maneuver is called the reparameterization trick (Figure 3.12), as z is left with essentially the same distribution it had before, but is now differentiable and the backpropagation can now pass on through to the encoder without issue.

At last, it is time to express the definite formula for the loss function. As the distribution $p(x|\mu,\sigma)$ is a Gaussian and the multivariate normal distribution is assumed to have a diagonal covariance matrix (all its variables are independent), the probability can then be written as:

$$p(x|\mu,\sigma) = \prod_{i=1}^{N} p\left(x_i|\mu_i,\sigma_i\right) \tag{3.21}$$

To combine this equation with the quality of reconstruction defined in 3.19, a logarithm is applied:

$$\begin{aligned}
\log p(x|\mu,\sigma) &= \log \prod_{i=1}^{M} p\left(x_i|\mu_i,\sigma_i\right) \\
&= \log \frac{1}{\sqrt{2\pi}^{M} \cdot \prod_{i=1}^{M} \sigma_i} \cdot e^{-\sum_{i=1}^{M} \frac{\left(x_i-\mu_i\right)^2}{2\sigma_i^2}} \\
&= \log \frac{1}{\sqrt{2\pi}^{M}} - \log \prod_{i=1}^{M} \sigma_i - \sum_{i=1}^{M} \frac{\left(x_i-\mu_i\right)^2}{2\sigma_i^2}
\end{aligned} \tag{3.22}$$

Assuming that only a single sample from z is required for the training to work well, only the log probability remains:

$$\log p(x|\mu,\sigma) = -\sum_{i=1}^{M} \left(\log \sigma_i + \frac{\left(x_i-\mu_i\right)^2}{2\sigma_i^2} \right) \tag{3.23}$$

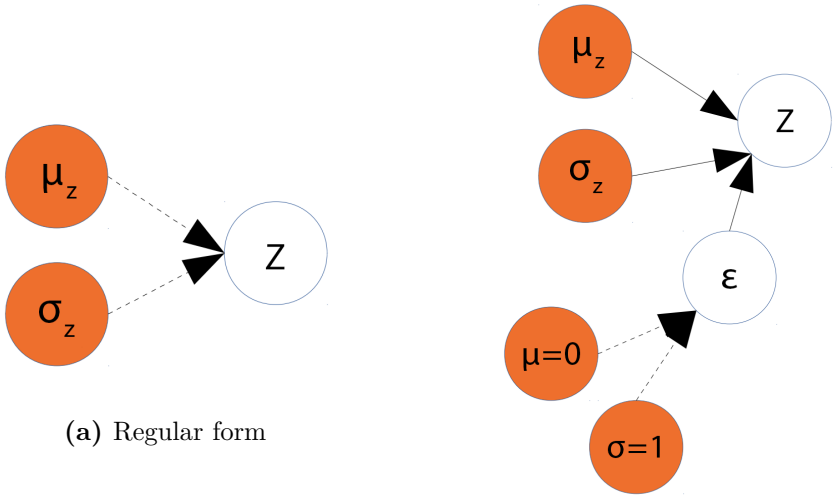
Applying the same methods to the KL regularization factor from 3.19, the equation is transformed to:

$$D_{KL}(q(z|x) \parallel p(z)) = -\frac{1}{2} \sum_{i=1}^{N} \left(1 + \log \sigma_{z_i}^2 - \mu_{z_i}^2 - \sigma_{z_i}^2\right) \tag{3.24}$$

Combining 3.23 and 3.24, the final definitive loss function is:

$$\mathrm{L}^{(i)} = \sum_{i=1}^{M} \left(\log \sigma_i + \frac{(x_i - \mu_i)^2}{2\sigma_i^2} \right) - \frac{1}{2} \sum_{i=1}^{N} \left(1 + \log \sigma_{z_i}^2 - \mu_{z_i}^2 - \sigma_{z_i}^2 \right) \tag{3.25}$$

Optimizing this loss results in the generation of a latent space which maintains the similarity of nearby encodings on the local scale via clustering, yet globally is very densely packed near the latent space origin. this is the equilibrium reached by the cluster-forming nature of the reconstruction loss, and the dense packing nature of the KL divergence loss, forming distinct clusters the decoder can decode. Thus, when randomly sampling a vector from the same prior distribution of the encoded vectors, the decoder will successfully decode it. And if interpolating, there are no sudden gaps between clusters, rather a smooth mix of features a decoder can understand [18].



(a) Regular form

(b) Reparameterized form

Figure 3.12: Latent state form comparison before and after reparameterization trick application. Source: [22]

4. Dataset

The data used for the training and testing of all models is a representation of the whole genome sequence of the *Escherichia Coli* bacteria (NCTC 86). This sequence has been converted from the original FASTA/FASTQ format into a more suitable format for analyzing overlaps by using the *minimap* tool. The result is the Pairwise mapping format (PAF) that is then forwarded into a preprocessor for transformation and conversion.

4.1. Pairwise Mapping Format

The Pairwise mApping Format (PAF) is a lightweight mapping format that describes the overlapping regions of pairs of reads, with the information on the starting and ending locations, length, quality, etc. There are twelve `TAB` delimited columns, whose short description can be viewed in table 4.1. All fields are obligatory, but each row may optionally have additional fields in the SAM-like[1] typed key-value format at the end of each mapping.

For the purposes of this thesis, only a handful of values from each mapping is used, with these being the query sequence name, length, start and end coordinate. All other information may be ignored and will not be used in the following preprocessing steps [12].

4.2. Preprocessing

Before the data can be used for model training, the overlaps must be calculated and stored in a separate file. For each new query in a PAF file, a new data model is created, containing four parameters - query ID, query length, overlap type and

[1]Sequence Alignment/MAP format — Generic text format for storing read alignments against reference sequences

Col	Type	Description
1	string	Query sequence name
2	int	Query sequence length
3	int	Query start coordinate (BED-like)
4	int	Query end coordinate (BED-like)
5	char	'+' if query and target on the same strand; '–' if opposite
6	string	Target sequence name
7	int	Target sequence length
8	int	Target start coordinate on the original strand
9	int	Target end coordinate on the original strand
10	int	Number of matching bases in the mapping
11	int	Number bases, including gaps, in the mapping
12	int	Mapping quality (0–255 with 255 for missing)

Table 4.1: PAF columns description

the overlap sequence. As described in section 2.2, an overlap can fall into one of several categories: regular, repeat or chimeric. An additional category, low-quality overlaps, has also been added here to represent overlaps whose average overlap count is found below a predefined threshold.

Once a data model is created, all mappings tied to its query ID are analyzed and the overlap count is updated for each query index accordingly. Upon parsing all associated mappings, the parser normalizes the overlap sequence so that they all match in total length. This is an important step as the initial length varies greatly, with the shortest overlaps having only a few hundred indexes, while the longest ones reaching all the way up to thirty thousand plus indexes (see Figure 5.1). By analyzing the initial lengths of all overlaps for the NCTC 86 PAF file, it was decided that all entries will be scaled to 5000 indexes. Downscaling is done simply by calculating the number of indexes that should be removed and then removing them at random positions so to not affect the overlap characteristics in the data model. Upscaling, on the other hand, is a bit more complex, as new data needs to be generated in order to match the defined length. Firstly, the number of indexes to be interpolated is calculated, the same way as it is done for downscaling. Then, random positions are chosen from the current length. New indexes are inserted at these positions, with the values being a mean of the values

to their immediate left and right. These steps are repeated until the total length does not equalize with the predefined length.

Upon normalization, each entry is written into a shared TSV[2] file for later use. The file contains four columns matching the attributes of the previously described data model (query ID, original length, overlap type and overlap sequence). The overlap sequence is a comma-delimited list of values, one for each index. This data is what is used for training of the models. Additionally, in order to help the optimization process, all values are normalized into the [0, 1] range. This transformation does not have an effect on the shape or characteristics of the data, only the scale of its values. As the actual value is not directly used, it can be safely scaled down.

For the purpose of generating labeled data, around four hundred samples have been manually categorized by referencing the overlap charts. The charts are generated by simply parsing the TSV file and plotting the overlap sequence. The rest of the generated samples have been left uncategorized.

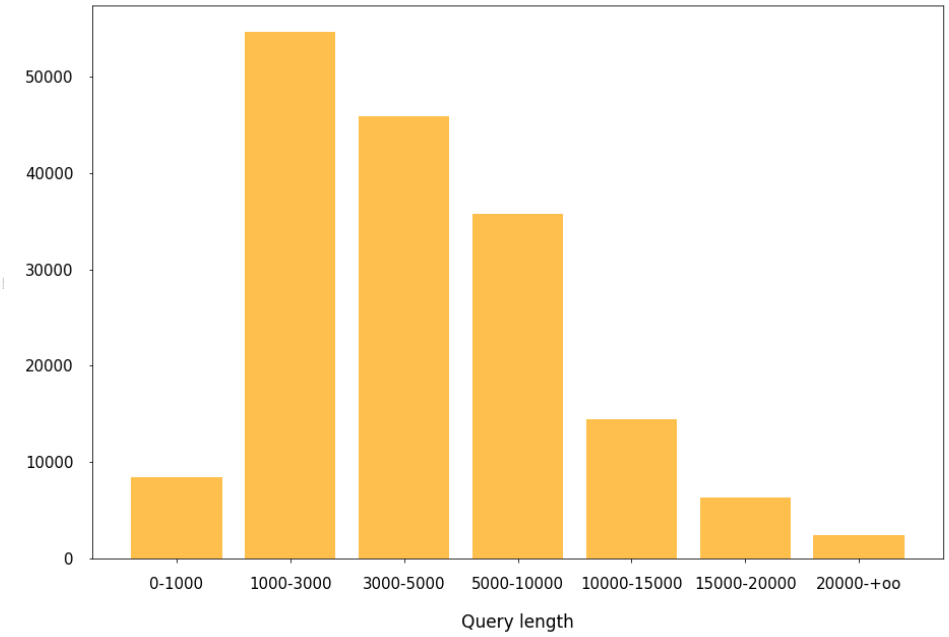


Figure 4.1: Bar plot of query sequence length in the original NCTC86 dataset

[2]Tab-separated value — Simple text format for storing data in a tabular structure

Dataset	No. samples
Labeled data (training)	231
Labeled data (test)	99
Unlabeled data	7461

Table 4.2: Data distribution

An additional problem is the unbalanced nature of the classes. Even though not many samples have been manually classified, out of the classified ones, repeat and regular sequences are most prominent and account for more than 85 percent of the total set. The most problematic ones are chimeric reads, with only 15 instances in a dataset of 330 sequences, accounting for only 0,05 percent of the total labeled data.

5. Results and evaluation

Three different models were created and evaluated, each of them theoretically described in chapter 3. These are:

1. One-dimensional convolutional neural network
2. Long short-term memory recurrent neural network
3. Variational autoencoder with an external classifier

Common settings include the use of the Adam optimizer with a learning rate of 10^{-4} and a batch size of 64. If not stated otherwise, the activation function defaults to ReLU. To prevent overfitting and cut down on unneeded execution time, an early stop was implemented into each training process, in such way that if the validation loss has a positive trend in five continuous epochs, the learning is automatically cut off. The architectures of the implemented models are defined in tables 5.1, 5.2 and 5.3 below.

Regarding the 1D-CNN, several different architectures have been designed, most differing in the number of convolutional layers and the use of pooling layers. The best performing one consists of twelve layers, including three pairs of convolutional layers. Hyperparameters for this model includes the kernel size and amount of filters for the convolutions and the pool size for the pooling layers. A grid search for these hyperparameters was executed, resulting in the values visible in table 5.1.

The LSTM model proved to be an issue with long static sequences like these. Even though the final model does not have that many parameters in total, the learning process turned out to be extremely slow, comparing to all other tested models. This has also affected the overall architecture, keeping it rather simple

Layer	Hyperparameters
1D Convolution	Filters = 128; Kernel size = 23
1D Convolution	Filters = 64; Kernel size = 23
Max. pooling	Pool size = 10
Batch normalization	—
1D Convolution	Filters = 64; Kernel size = 23
1D Convolution	Filters = 32; Kernel size = 23
Max. pooling	Pool size = 10
Batch normalization	—
1D Convolution	Filters = 32; Kernel size = 23
1D Convolution	Filters = 16; Kernel size = 23
Global average pooling	—
Fully connected	Units = 3

Table 5.1: Layer architecture for a 1D-convolutional neural network

Layer	Hyperparameters
LSTM	Units = 32
LSTM	Units = 32
Flatten	—
Fully connected	Units = 3

Table 5.2: Layer architecture for an LSTM recurrent neural network

(see table 5.2). All in all, there are two long short-term memory layers, each with 32 units, with other hyperparameters left default. As was the case with the 1D-CNN, the unit number was determined by a simple grid search.

Lastly, several implementations of a variational autoencoder have been constructed, with the main one being a relatively simple triple layer fully-connected architecture. Each of these components is described in table 5.3 below. As the autoencoder does not require labels for training, and with labeled data being scarce, the entire unlabeled subset was used. Once the autoencoder training is complete, the encoder is extracted as a separate component and used to transform labeled sequences into latent representations. These representations, along with the accompanying labels, are then fed into a classifier for training. As the

Deep variational autoencoder

Encoder		**Decoder**	
Layer	*Hyperparameters*	*Layer*	*Hyperparameters*
Fully-connected	Units = 1536	Fully-connected	Units = 512
Fully-connected	Units = 1024	Fully-connected	Units = 1024
Fully-connected	Units = 512	Fully-connected	Units = 1536
		Fully-connected	Units = 5000

Table 5.3: Layer architecture for a deep variational autoencoder

training process comes to a finish, a smaller subset of the labeled data is used for classifier testing. This way, only a small subset of data needs to be labeled so that classification metrics can be calculated, while all other samples are forwarded to the autoencoder. Since the latent representation is of much lower dimensionality, regular machine learning classifiers can also be used side by side. Considered classifiers are a simple fully-connected feed-forward neural network, a non-linear Gaussian support vector machine, and a k-nearest neighbors algorithm. Just like the 1D-CNN, these models also have the early stopping condition built to try and prevent overfitting.

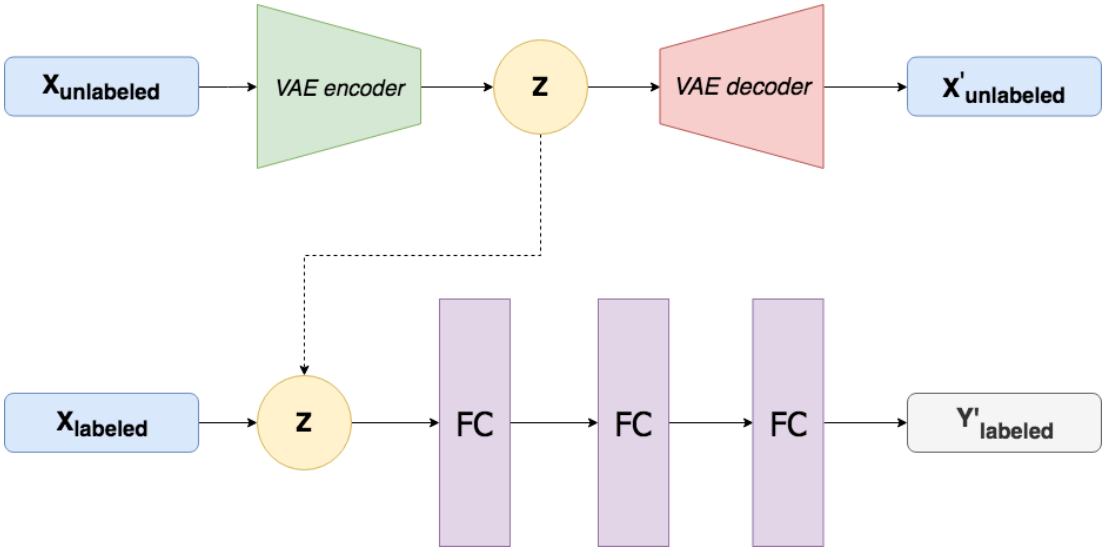


Figure 5.1: Structure of the VAE + FC model. Unlabeled data is fed into the variational autoencoder for training. The trained encoder then encodes labeled data into the latent state and forwards it to the classifier which outputs the classes.

All classifiers have been trained on 70 percent of the original labeled dataset,

with the remaining 30 percent left for testing and validation. As stated in the paragraph above, since the autoencoder does not require a labeled set for training, it can use the full potential of the unlabeled data set which include more than 7000 samples. As stated in table 4.2, this train-test split results in 231 samples for training and 99 for testing and validation. The models were trained inside Jupyter notebooks hosted on a remote server, with the hardware including Nvidia Tesla K80 graphic cards with 12 GB of VRAM, Intel Xeon CPUs and an additional 61 GB of standard RAM, hosted on Floydhub[1]. All developed code can be found on Github[2], publicly available under an MIT license.

An overview of the performance of all models is given in table 5.4. The main point of reference for determining the success or failure of the implemented model is a thesis by Šebrek, T. (2017), which was tasked with the same problem, the only difference being that the input data were images of overlap charts. A classification report, together with a confusion matrix, has been calculated for each implementation, with the F1 score being listed for easy model comparison.

	1D-CNN	LSTM	VAE+SVM	VAE+FC
F1 score	0.5466	0.5800	0.7300	0.6733

Table 5.4: F1 score for each tested model

	FF	SEMI-AE	M1+M2	SEMI-GAN
F1 score	0.7408	0.9000	0.9341	0.9708

Table 5.5: Macro F1 score for each model from Šebrek (2017). FF = Simple feed-forward neural network; SEMI-AE = Semi-supervised regular autoencoder; M1+M2 = Semi-supervised variational autoencoder model; SEMI-GAN = Semi-supervised generative adversarial network

Unfortunately, even a quick skim through the confusion matrices of the tested models (tables 5.6, 5.7, 5.8, 5.9) leads to the conclusion that there is no model that could faithfully recognize chimeric reads with a satisfying degree of certainty, with most of them tending to classify these reads as repeats. This was something that could have been expected, mostly due to the small number of these reads in the testing subset. On a positive note, there are almost no cases where some

[1] www.floydhub.com

[2] www.github.com/ffloreani/masters-thesis

	Regular	Repeat	Chimeric
Regular	11	11	0
Repeat	0	70	0
Chimeric	0	6	1

Table 5.6: Confusion matrix for the 1D-CNN model

	Regular	Repeat	Chimeric
Regular	14	9	0
Repeat	3	67	0
Chimeric	0	5	1

Table 5.7: Confusion matrix for the LSTM RNN model

other read type ends up as a chimeric one, which is a good indicator of the possible area of interest for future work. Comparing the F1 scores to the ones from Šebrek's work, even though some of the models differ in architecture, it is clear that the models implemented here still need some work, with his best score being an almost perfect 0.9708, a far shot from the best ones here [21].

One important thing that can be inferred from the total scores in table 5.4 is that there is a statistically significant difference between the models that use autoencoders and those that do not. This is a step in the right direction when it comes to the question of using semi-supervised models. Even though the autoencoder does not classify the samples directly, it plays a major part in the process. Dimensionality reduction is especially important in cases like these when large samples with many features are used. This type of data tends not to play nice with any but the most simple models, like LSTMs. Additionally, by using variational autoencoders, one can greatly benefit from its generative structure. This means that not only are the input samples being represented in a compressed state, but that new samples can also be easily generated by sampling the latent dimension and decoding new input samples from them through a decoder.

	Regular	Repeat	Chimeric
Regular	23	7	0
Repeat	5	58	0
Chimeric	0	3	3

Table 5.8: Confusion matrix for the deep VAE + SVM model

	Regular	Repeat	Chimeric
Regular	14	9	0
Repeat	2	68	0
Chimeric	0	4	2

Table 5.9: Confusion matrix for the deep VAE + FF model

6. Conclusion

The main purpose of this thesis was to create a deep learning network for classification of one-dimensional overlap signals depending on their type, defined to be either regular, repeat or chimeric. Several ideas have been tried out, concentrating on using the signals in their raw vectored form instead of converting them to charts. The compared methods are a 1D-convolutional neural network, a long short-term memory recurrent neural network and a combined model based on a variational autoencoder connected to a 1D-CNN.

The original PAF dataset has been converted into a makeshift database, with each entry defining one overlap query. To obtain some labeled data, around four hundred samples were manually classified. Several problems have been identified while analyzing the dataset contents, with the main issue being a gross distance between class representations, with chimeric reads being the most underrepresented.

Regarding the results, the best overall model is considered to be the variational autoencoder with an additional Support Vector Machine classifier. The autoencoder does marvels when it comes to dimensionality reduction, although it tends to struggle with consistently representing some read types. Out of the tested classifiers, SVMs tend to do best when it comes to drawing boundaries between latent samples, which may have to do with its Gaussian kernel being similar to distributions used by the autoencoder. All that said, comparing these models with similar works shows that there is still a long way to go before any of the implemented models can come close to the currently leading ones.

There is still a lot that can be done in order to obtain better results. Firstly, a more balanced dataset would have a substantial impact on the end result, as in the current situation, all models seem to struggle with classifying chimeric reads. Another approach to this issue is to consider chimeric reads as anomalies

and treat the problem as anomaly detection, a heavily researched field with a lot of good references to start from. Pivoting the problem again, the 1D-signal can be imagined as a waveform, which would then make it applicable for the use of classification methods like Dynamic Time Wrapping or Echo state networks. There is also hope for the future application of autoencoders and other generative methods, as there will always be a shortage of labeled data.

Bibliography

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, March 1994. ISSN 1045-9227. doi: 10.1109/72.279181. URL http://dx.doi.org/10.1109/72.279181.

[2] François Chollet. *Deep Learning with Python*. Manning, November 2017. ISBN 9781617294433.

[3] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014.

[4] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[5] Wenfeng Gong, Hui Chen, Zehui Zhang, Meiling Zhang, Ruihan Wang, Cong Guan, and Qin Wang. A novel deep learning method for intelligent fault diagnosis of rotating machinery based on improved cnn-svm and multichannel data fusion. *Sensors*, 19(7), 2019. ISSN 1424-8220. doi: 10.3390/s19071693. URL https://www.mdpi.com/1424-8220/19/7/1693.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Anthony J.F. Griffiths. Dna sequencing, 2012. URL https://www.britannica.com/science/DNA-sequencing.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] National Human Genome Research Institute. Shotgun sequencing. URL https://www.genome.gov/genetics-glossary/Shotgun-Sequencing.

[10] Andrej Karpathy. Lecture notes from cs231n convolutional neural networks for visual recognition, 2018. URL http://cs231n.github.io/convolutional-networks/.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014. URL https://arxiv.org/abs/1412.6980.

[12] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 03 2016. URL https://doi.org/10.1093/bioinformatics/btw152.

[13] Gerard Martinez. Autoencoders for the compression of stock market time series, 2019. URL https://towardsdatascience.com/autoencoders-for-the-compression-of-stock-market-data-28e8c1a2da3e.

[14] Christopher Olah. Understanding lstm networks, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs.

[15] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly, 2017. ISBN 978-1-4919-1425-0.

[16] S. T. Roweis and L. K. Saul. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290:2323–2326, December 2000. doi: 10.1126/science.290.5500.2323.

[17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. URL https://arxiv.org/abs/1609.04747.

[18] Irhum Shafkat. Intuitively understanding variational autoencoders, 2018. URL https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf.

[19] Kai Siebenrock. Activation functions. URL https://www.kaggle.com/siebenrock/activation-functions.

[20] Todd J. Treangen and Steven L. Salzberg. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, 13, Nov 2011. URL https://doi.org/10.1038/nrg3117. Review Article.

[21] Tomislav Šebrek. Classification of 1d-signal types using semi- supervised deep learning. Master's thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2017.

[22] S. Šegvić, M. Tutek, et al. Lecture notes from deep learning, 2019. URL `http://www.zemris.fer.hr/~ssegvic/du`.

[23] Jan Šnajder and Bojana Dalbelo Bašić. Machine learning. Unpublished book, 2014.

List of Figures

2.1. High level overview of the OLC approach to *de novo* sequencing . 4
2.2. Regular coverage chart 5
2.3. Repeat reads assembly ambiguities 6
2.4. Irregular coverage charts 7

3.1. Average and max pooling on a 4x4 feature map 12
3.2. Nonlinear activation functions 13
3.3. Convolutional and fully connected layer comparison 14
3.4. Example of a convolution operation with a 3x3 filter 15
3.5. Usage of padding in preservation of output dimensions 15
3.6. 1D convolutional layer 16
3.7. Comparison of normal an RNN input vectors 17
3.8. Structural diagram of an RNN cell 19
3.9. Structural diagram of an LSTM cell 21
3.10. Representation of a basic autoencoder 22
3.11. Variational autoencoder architecture 24
3.12. Latent state form comparison before and after reparameterization trick application 27

4.1. Bar plot of query sequence length in the original NCTC86 dataset 30

5.1. Structure of the VAE + FC model 34

List of Tables

4.1. PAF columns description 29
4.2. Data distribution 31

5.1. Layer architecture for a 1D-convolutional neural network 33
5.2. Layer architecture for an LSTM recurrent neural network 33
5.3. Layer architecture for a deep variational autoencoder 34
5.4. F1 score for each tested model 35
5.5. Macro F1 score for each model from Šebrek (2017) 35
5.6. Confusion matrix for the 1D-CNN model 36
5.7. Confusion matrix for the LSTM RNN model 36
5.8. Confusion matrix for the deep VAE + SVM model 37
5.9. Confusion matrix for the deep VAE + FF model 37

Identifikacija tipova 1D-signala metodama dubokog učenja

Sažetak

De-novo sastavljanje genoma proces je temeljen na preklapanju i analizi kratkih genetskih očitanja. Uslijed raznolikih tehničkih izazova, određene vrste lažnih očitanja i preklapanja mogu također biti sintetizirane, što uvelike otežava ispravno sastavljanje originalne sekvence. Jedna od metoda za otkrivanje takvih preklapanja je i pretvorba istih u 1D-signal, pomoću kojih se onda može jednostavnije zaključiti koji tip preklapanja predstavlja. U okviru ovog rada, predstavljeno je nekoliko metoda duboko učenja za klasifikaciju ovih signala, točnije 1D-konvolucijske mreže, povratne mreže, kao i primjena autoenkodera. Dodatno, prikazana je i njihova primjena na stvarnim podacima, kao i usporedba uspješnosti.

Ključne riječi: bioinformatika, sastavljanje genoma, lažna preklapanja, duboko učenje

Classification of 1D-Signal Types Using Deep Learning

Abstract

The *de novo* genome assembly process is based on overlapping and analyzing short reads of genetic information. Due to various technical challenges, certain types of false overlaps can also be generated, which greatly impedes successful reconstruction. One of the methods for detecting such overlaps is by generating a 1D-signal for each read, which can then be used to determine its exact overlap type. This thesis proposes several deep learning methods for classifying these signals, including 1D-convolutional and recurrent networks, as well as autoencoders. A detailed comparison of their application on real-world data is also included.

Keywords: bioinformatics, sequence assembly, false overlaps, deep learning