University of South Bohemia in České Budějovice

and

Johannes Kepler University

Faculty of Science

Bachelor's Thesis

Auto-Encoding Amino Acid Sequences with LSTM

Author:
Markus Promberger

Supervisor:
Univ.-Prof. Dr. Sepp Hochreiter (JKU)
Co-Supervisor:
Bernhard Schäfl, Msc (JKU)

Linz, March 2022

Bibliographical Detail

Promberger, M., 2022: Auto-Encoding Amino Acid Sequences with LSTM. Bachelor Thesis, in English. - 56 p., Institute for Machine Learning, Faculty of Engineering & Natural Sciences, Johannes Kepler University, Linz, Austria

Annotation

In this thesis a sequence to sequence autoencoder for amino acid sequences is constructed. The latent representation of the autoencoder is then used to classify the amino acid sequences according to their animal kingdom. The data consists of sequences from three different kingdoms, mammals, fish and birds. The thesis includes the preprocessing necessary for the data, the construction of the sequence to sequence autoencoder and the process of classification in the latent space.

Declaration

I declare that I am the author of this qualification thesis and that in writing it I have used the sources and literature displayed in the list of used sources only.

Linz, 8th March 2022
....
Place, date

Markus Promberger
....
Markus PROMBERGER

Contents

Abstract

Clustering biological sequences into phylogenetic trees and assigning newly found sequences to previously known groups is an important task in biological research. When dealing with a large amount of sequences, tools from the field of bioinformatics become essential for the success of the research. There are well established methods for clustering sequences like agglomerative clustering or maximum parsimony methods. These techniques however are not well suited for integrating new sequences into them, because the whole structure needs to be adjusted. This process is time consuming, so in order to improve it, this thesis aims at implementing a technique that utilizes machine learning methods for this task. The Long Short-Term Memory (LSTM) introduced by Hochreiter und Schmidhuber is the main component of the machine learning architecture proposed in this work.

The proposed approach aims at creating a compressed representation, a so called code, of known amino acid sequences. These sequences are from three different kingdoms of animals and the code is used to assign new sequences to the kingdoms. The technique utilizes sequence alignment algorithms and agglomerative clustering to preprocess the data for the machine learning architecture. That architecture is a sequence to sequence autoencoder that uses LSTMs to process the sequences. An autoencoder architecture is composed of an encoder that creates the before mentioned code from the input and a decoder that learns to recreate the input from that code. K-means and k-nearest neighbors classifiers are then fit to this code produced from the sequence to sequence autoencoder in order to classify the new sequences. An independent test set is used to estimate the accuracy of the classifiers.

With a classification accuracy of over 85% on the test set this approach could potentially be a way of predicting the origin of amino acid sequences or other characteristics.

Chapter 1

Introduction

1.1 Motivation

Bioinformatics is about the extraction and storage of information from biological data. Protein sequences offer a wide variety of possible research, one of them is the construction of phylogenetic trees. Phylogenetic trees show the distance between organisms often based on sequential data. These trees can be used for analyzing and visualizing evolutionary relationships between sequences or organisms. There are well established methods for generating phylogenetic trees like maximum parsimony methods or distance-based methods. Recent works from Sinai et. al [27] or Schäfl [25] show that machine learning could be a valuable tool for analyzing protein sequences.

Because of the potential of machine learning, this thesis focuses on developing a sequence to sequence autoencoder [29] for protein sequences of the hemoglobin family. For that, sequences from three different animal kingdoms, birds, fish and mammals, are first pairwise aligned and then clustered based on the distance between them. These clusters are then used to perform clustered cross-validation to learn a sequence to sequence autoencoder. This machine learning architecture computes a fixed size representation, called latent space or code, for each sequence, which may have different lengths. Having a fixed size representation of the sequences can be useful when trying to compare the sequences to each other. Both the encoder and decoder of the autoencoder are LSTMs and a dense feed-forward neural network is used for dimensionality reduction. The latent space between the encoder and decoder is analyzed using k-means [16] and k-nearest neighbors clustering with the goal that three clusters corresponding to the three kingdoms can be observed in the latent space. For a visual inspection the dimensionality reduction techniques t-SNE [31] and UMAP [18] are used.

1.2 Related Work

The work from Sinai et al. [27] shows that variational autoencoders (VAE) can be used for estimating protein functions from the sequence. In their work a VAE with three dense layers of dimensionality 250 with exponential linear units (ELU) [2] as activation function was used to learn the distribution $p(x, z) = p(z)p(x|z)$. Here $z \in Z$ are the latent variables and $x \in X$ are the observed sequences. Once a good distribution $p(x, z)$ is learned, new sequences $\hat{x}$ similar to the ones from X can be generated.

Sinai et al. [27] made three main observations about their results: (i) The learned probability distributions correlate well with experimental measured protein functions. (ii) When using a two dimensional latent variable the sequences build sensible clusters that correspond to the distance between the sequences. (iii) The VAE learned higher level interactions within the protein sequences.

As potential future work Sinai et al. [27] suggest to use recurrent or convolutional architectures which might outperform the used VAE. This thesis uses a traditional sequence to sequence autoencoder with LSTMs as recurrent neural nets for the encoder and decoder. The traditional autoencoder doesn't learn a distribution $p(x, z)$ but rather computes a fixed, both in size and values, representation from the sequences.

Chapter 2

Data and Methods

2.1 Structure of Data

The data consists of hemoglobin related protein sequences of three different kingdoms of animals. The first group consists of mammals, the second of fish and the third group are birds. In total there are 736 sequences: 384 of mammals, 92 of fish and 260 of birds. Protein sequences consist of 20 different amino acids that can be further divided into 4 groups according to their chemical properties. There are 10 non-polar amino acids, 5 polar, 3 positively charged and 2 with a negative charge. Amino acid sequences are usually stored in the *FASTA* [1] format, which is also the case for this thesis. In this format, unknown positions in the protein sequence are represented with an *X* as the 21^{st} symbol. The codes for the amino acids are described in Table 2.1 according to the *IUPAC-IUB* rules [3]. The chemical characteristic of each amino acid is also described in this table. There are in general four groups of amino acids, these characteristics are determined by the side chain of the AA. The AA are divided into the four groups hydrophobic, polar uncharged, positively charged and negatively charged.

For a first glance at the data, the sequence lengths as well as the k-mer distribution have been analyzed.

2.1.1 Sequence Length

The lengths of the sequences are mostly in the range between 141 and 143 with 682 of the 736 sequences being in this range. These make up almost 93% of the data. 47 of the sequences are shorter than 141 amino acids (AAs) with the shortest being 9 AAs long. 7 of the sequences

Amino acid	Abbreviation	Chemical characteristic
Alanine	A	hydrophobic
Arginine	R	positively charged
Asparagine	N	polar uncharged
Aspartic acid	D	negatively charged
Cysteine	C	polar uncharged
Glutamine	Q	polar uncharged
Glutamic acid	E	negatively charged
Glycine	G	hydrophobic
Histidine	H	positively charged
Isoleucine	I	hydrophobic
Leucine	L	hydrophobic
Lysine	K	positively charged
Methionine	M	hydrophobic
Phenylalanine	F	hydrophobic
Proline	P	hydrophobic
Serine	S	polar uncharged
Threonine	T	polar uncharged
Tryptophan	W	hydrophobic
Tyrosine	Y	hydrophobic
Valine	V	hydrophobic
Unknown or ’other’	X	no characteristic

Table 2.1: One-letter codes of the 21 present symbols in the dataset. These include 20 amino acids as well as the *X* for unknown positions. The third column describes the chemical characteristic for each amino acid.

are longer than 143 and the longest is 250 AAs long. The means and standard deviations of the sequence lengths for the different kingdoms are shown in Table 2.2. The mammalian sequences have the lowest mean length with 138.3 and the highest standard deviation with 17.4. This is due to the fact that 26 of the 47 sequences shorter than 141 are from this category. Overall the mean sequence length is 139.3 with a standard deviation of 14.9. A naive approach of clustering the sequences would be to cluster them according to the sequence length. If the means are far apart and the standard deviation is low, a statistical error bound could be calculated. However in this case, the means are very similar and even within one standard deviation the means are overlapping. For that reason a classification based on the sequence length is not reliable.

Mean sequence lengths		
kingdom	mean	std
mammals	138.3	±17.4
fish	142.9	±11.7
birds	139.4	±11.3
all sequences	139.3	±14.9

Table 2.2: Mean and standard deviation for sequence lengths for every kingdom individually and for all sequences.

2.1.2 k-mer Plot

The term k-mer in bioinformatics refers to the sub-sequences of length k of a biological sequence like an amino acid sequence. These k-mers can be overlapping or non-overlapping but the order of the elements cannot be changed. For example all possible 3-mers of the amino acid sequence *MTHCG* are *MTH*, *THC* and *HCG*. In general for sequences of length L there are $(L - k + 1)$ overlapping and $\lfloor \frac{L}{k} \rfloor$ non-overlapping ones. Using these k-mers, a rough estimate of how similar sequences are to each other can be made by comparing the frequencies of the sub-sequences. In a k-mer plot this is done by plotting the frequency with which a k-mer appears on the x-axis. The values on the y-axis show how many different k-mers appear x-times. For example in Fig. 2.1 for the k-mer size of 100, there are 29 k-mers that appear 22 times within the birds data (big red peak at $x = 22$).

From the graphs it can be observed that the birds sequences are the most similar, followed by the mammals. The fish are the most distinct within their group. This conclusion is based on the high y-values in the right region of the x-axis for the bird sequences in the high k-mer size plots. A high concentration around the left x-axis values shows that a lot of k-mers appear rarely or only once in the data. If the data would be similar, then the same k-mer would show up more often, leading to high y-values more to the right of the x-axis. This can be seen in the plot with k-mer size 100 for the bird sequences with high peaks upwards of $x = 20$.

2.2 Pairwise Sequence Alignment

Pairwise sequence alignment is needed to find similarities between two sequences by aligning them to each other. This can be achieved by matching similar elements in the two sequences and inserting gaps. An alignment has an associated alignment score that consists of a score

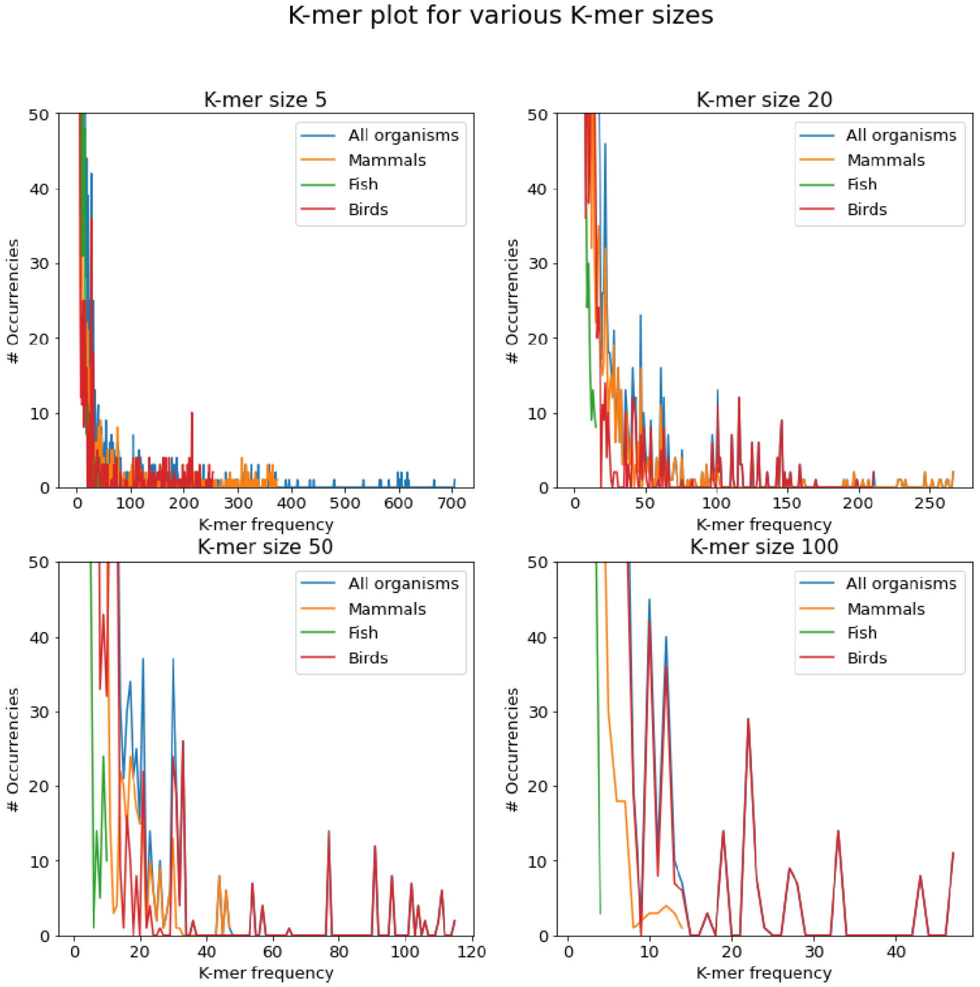


Figure 2.1: k-mer plot where all organisms are compared with each other with respect to a specific k-mer size.

achieved by the matches between the sequences and the number of gaps inserted to align the elements. The scoring metric used for calculating the score of the matches can either be simple, e.g. $+1$ for match and -1 for mismatch, or more complex, e.g. point accepted mutation (PAM) [4] or block substitution matrices ($BLOSUM$) [9]. Both PAM and $BLOSUM$ have different versions that are better in certain situations. $PAM100$, similarly $BLOSUM90$, is suited for similar sequences, for example sequences from closely related organisms, while $PAM250$, and $BLOSUM45$ are rather used for sequences from distantly related organisms.

The insertion of gaps can be used to align matches within two sequences, but the insertion of gaps is usually penalized. This is done because a gap represents a difference between the two sequences. If for example the only difference between two sequences is the addition of a single amino acid in the middle of one sequence, then without the gap penalty, the two sequences would have a perfect alignment score. This is not an accurate alignment score, because there is a difference, namely the added amino acid. This distance can be incorporated into the alignment by using a gap penalty. There are two different approaches for penalizing gaps, affine and linear gap penalty. For linear gap penalty every gap is seen as an independent event and there is always the same penalty for inserting a gap into the alignment. In affine gap penalty scenarios the opening of a gap has a different penalty than the elongation of an already opened one. Usually the opening is more expensive than the elongation, leading to longer continuous gaps instead of many short ones. The score achieved by a pairwise sequence alignment, taking into account the matches and mismatches as well as the amount of gaps, is proportional to the similarity between the two sequences.

Pairwise sequence alignment algorithms are used for finding such alignments that maximize the previously mentioned alignment score. The original sequential order of the elements must be preserved. The algorithms can be divided into two classes, global and local alignment algorithms. While global algorithms optimize the complete alignment, local algorithms find matching sub-sequences between the two sequences. In the following, the most prominent algorithm of each class will be introduced.

2.2.1 Global Sequence Alignment

In global sequence alignment the whole sequence a is aligned to the whole sequence b. The most known algorithm for this is the Needleman-Wunsch algorithm [20]. This algorithm uses dynamic programming in order to solve the complex problem of finding the best pairwise alignment. It achieves this by solving a series of smaller problems by aligning smaller sub-sequences and combining the solutions of the sub-problems. Every possible alignment gets assigned a score and for the purpose of finding the best alignment, only the ones with the highest score are returned. In the following the Needleman-Wunsch algorithm with linear gap penalty d and a scoring matrix S for sequences a and b is explained.

The Needleman-Wunsch algorithm starts by creating a matrix S with dimensions $[(l_a + 1) \times (l_b + 1)]$ where l_a and l_b are the lengths of the two sequences. For the initialization $S(0,0)$

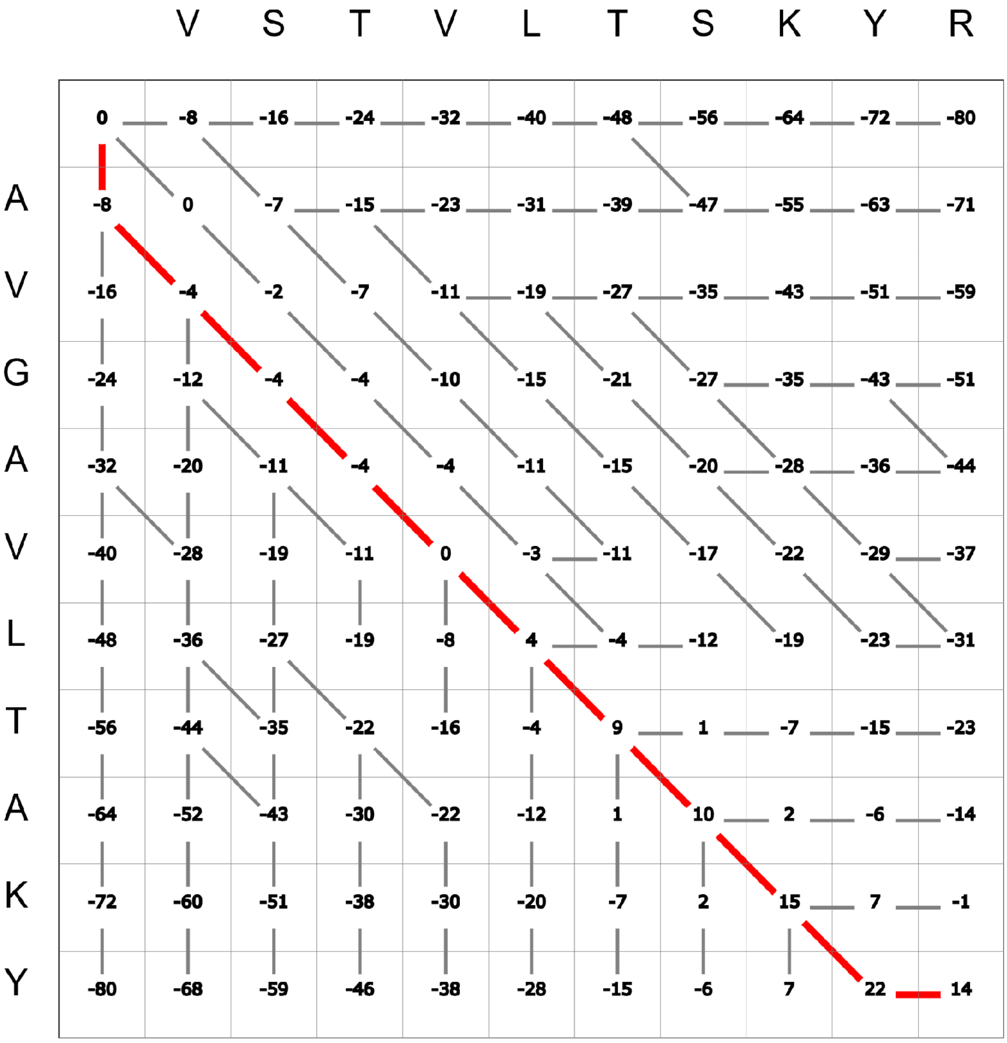
is set to zero and the first row $S(0,j)$ with $j \in [1,...,l_b]$ and column $S(i,0)$ with $i \in [1,...,l_a]$ are filled with $d \cdot j$ and $d \cdot i$ respectively. Following the initialization, all matrix entries $S(i,j)$ with $i \in [1,...,l_a]$ and $j \in [1,...,l_b]$ are calculated in the following way

$$S(i,j) = max \begin{cases} S(i-1,j-1) + s(a_i,b_j) \\ S(i-1,j) - d \\ S(i,j-1) - d \end{cases} \tag{2.1}$$

The matrix S is filled by following the instructions given by equation 2.1. The diagonal move $S(i-1,j-1) + s(a_i,b_j)$ through the matrix represents a match or mismatch by adding the score of the scoring matrix $S(a_i,b_j)$ to the previous score $S(i-1,j-1)$. The value $S(i-1,j)-d$ is a vertical move and corresponds to a gap in the sequence b and $S(i,j-1)-d$ is a horizontal move that shows up in the sequence alignment as a gap in sequence a. The maximum of these 3 values is used to fill the matrix S at the corresponding indices.

In the example in Fig. 2.2 a gap penalty d of -8 is used, matches and mismatches are scored according to $BLOSUM62$. Solving for example the value of $S(2,1) = -4$ is done by looking at the diagonal move $S(1,0) + s(a_2,b_1)$ which results in $-8 + 4 = -4$. The horizontal move $S(2,0) - d$ leads to $-16 - 8 = -24$ and the vertical move with $S(1,1) - d$ with $0 - 8 = -8$. The maximum of these values is therefore the diagonal move with $S(1,0) + s(a_2,b_1) = -4$, which is marked with the diagonal line. Keeping track of which of these three values is the maximum (visually in the form of lines) allows a construction of the optimal global sequence alignment by backtracking from the bottom right to the top left corner of the matrix S (as in Fig. 2.2). The alignment has a gap at the beginning of the vertical sequence a and ends with a gap at the end of the horizontal sequence b. The final alignment has a length of 11 and looks like the following:

```
- V S T V L T S K Y R
A V G A V L T A K Y -
```



		V	S	T	V	L	T	S	K	Y	R
	0	-8	-16	-24	-32	-40	-48	-56	-64	-72	-80
A	-8	0	-7	-15	-23	-31	-39	-47	-55	-63	-71
V	-16	-4	-2	-7	-11	-19	-27	-35	-43	-51	-59
G	-24	-12	-4	-4	-10	-15	-21	-27	-35	-43	-51
A	-32	-20	-11	-4	-4	-11	-15	-20	-28	-36	-44
V	-40	-28	-19	-11	0	-3	-11	-17	-22	-29	-37
L	-48	-36	-27	-19	-8	4	-4	-12	-19	-23	-31
T	-56	-44	-35	-22	-16	-4	9	1	-7	-15	-23
A	-64	-52	-43	-30	-22	-12	1	10	2	-6	-14
K	-72	-60	-51	-38	-30	-20	-7	2	15	7	-1
Y	-80	-68	-59	-46	-38	-28	-15	-6	7	22	14

Figure 2.2: Needleman-Wunsch algorithm showing the global alignment of two sequences. The red line represent the best alignment with an alignment score of 14. [30]

2.2.2 Local Sequence Alignment

Local sequence alignment algorithms find the best matching sub-sequences between two sequences. The Smith-Waterman algorithm [28] is the most prominent example for such an algorithm. The Smith-Waterman algorithm works similar to the Needleman-Wunsch algorithm (see 2.2.1) with the difference that the values of the matrix S cannot become smaller than zero. So whenever the matrix entry $S(i, j)$ would become negative, it is set to 0. Following this rule the matrix is filled according to Equation 2.2.

$$S(i,j) = max \begin{cases} S(i-1,j-1) + s(a_i,b_j) \\ S(i-1,j) - d \\ S(i,j-1) - d \\ 0 \end{cases} \tag{2.2}$$

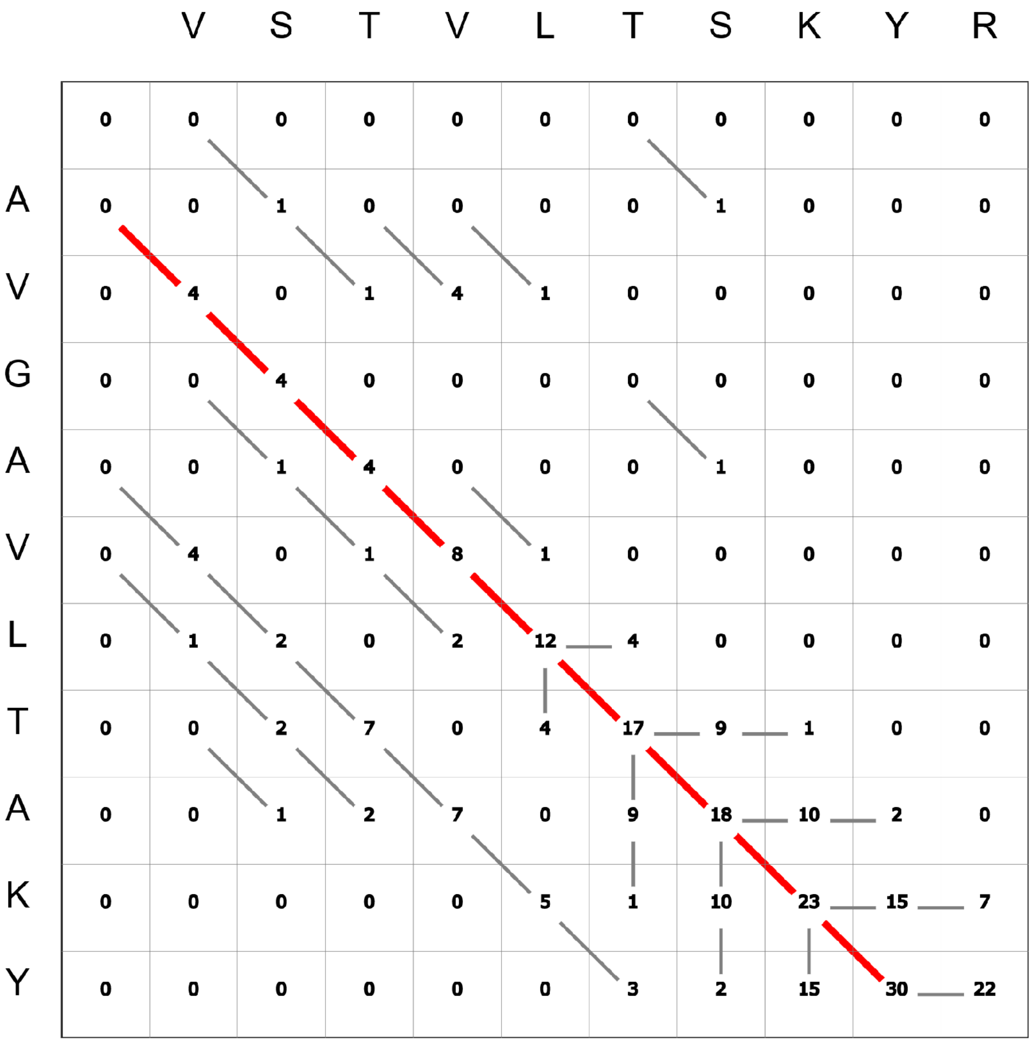
Finding a sub-sequence is then done by backtracking from the highest value in S until a zero entry is reached. Because the highest value in S is not necessarily the bottom right corner and backtracking can end before hitting the top left corner, it is possible that the algorithm returns sub-sequences. In the example in Figure 2.3 the alignment has a length of 9, which is similar to the global alignment from Chapter 2.2.1 with the start and end gap not included in the local alignment. Having a gap at the end leads to a lower final score than not having one and a gap at the start is only possible if the starting position in the matrix is negative, which is not possible for the Smith-Waterman algorithm. The following is the ideal local alignment found by the algorithm:

```
V S T V L T S K Y
V G A V L T A K Y
```

2.3 Clustering

Clustering is a popular method for analyzing data by grouping data points based on similarity or distance. Data points with a short distance between them are grouped together, they form clusters. One class of clustering algorithms is hierarchical clustering, which can be further divided into agglomerative and divisive algorithms. Agglomerative represents bottom up algorithms, starting with all data points being in a separate cluster and merging them until all observations are in one combined cluster. Divisive algorithms on the other hand are top down, so they start with all data in the same cluster and splitting the groups until every observation is in its own cluster.

For agglomerative algorithms there are three common approaches on how to use a distance metric between data points to merge clusters. In the following $D(x_i, x_j)$ is the distance



		V	S	T	V	L	T	S	K	Y	R
	0	0	0	0	0	0	0	0	0	0	0
A	0	0	1	0	0	0	0	1	0	0	0
V	0	4	0	1	4	1	0	0	0	0	0
G	0	0	4	0	0	0	0	0	0	0	0
A	0	0	1	4	0	0	0	1	0	0	0
V	0	4	0	1	8	1	0	0	0	0	0
L	0	1	2	0	2	12	4	0	0	0	0
T	0	0	2	7	0	4	17	9	1	0	0
A	0	0	1	2	7	0	9	18	10	2	0
K	0	0	0	0	0	5	1	10	23	15	7
Y	0	0	0	0	0	0	3	2	15	30	22

Figure 2.3: Example of Smith-Waterman algorithm. Backtracking the red line gives the optimal local sequence alignment with a score of 30. [30]

between two observations x_i and x_j from the clusters X_i and X_j respectively. $\Delta(X_i, X_j)$ is called linkage or sub-set distance.

1. Single Linkage:

$$\Delta(X_i, X_j) = \min_{x_i \in X_i, x_j \in X_j} D(x_i, x_j) \tag{2.3}$$

2. Complete Linkage:

$$\Delta(X_i, X_j) = \max_{x_i \in X_i, x_j \in X_j} D(x_i, x_j) \tag{2.4}$$

3. Group Average Linkage:

$$\Delta(X_i, X_j) = \frac{1}{|X_i||X_j|} \sum_{x_i \in X_i} \sum_{x_j \in X_j} D(x_i, x_j) \tag{2.5}$$

Single linkage is used to ensure a minimum distance between any two clusters in the clustering by using the minimum distance between two observations, as described in equation 2.3. This can be used in leave-one-cluster-out cross-validation, because the clusters are sure to be dissimilar up to a certain threshold (see Chapter 3.2.2). One drawback of single linkage is the so called chaining phenomenon, where on average very distinct clusters are combined because of single elements being close. This phenomenon can be avoided by using the complete linkage (see equation 2.4), but this linkage distance is vulnerable to outliers and cannot guarantee a minimal distance between clusters. The group average linkage (see equation 2.5) also does not produce the chaining phenomenon and is more robust to outliers. [21]

The selection of the distance metric $D(x_i, x_j)$ is crucial for the overall clustering algorithm. For this elaboration the distance associated with the pairwise identity (PID, see Equation 2.6) as defined in Equation 2.7 [25] was used.

$$PID(a, b, \lambda) = 100 \cdot \frac{n_{identical}(\lambda)}{min(l_a, l_b)} \tag{2.6}$$

$$d(a, b, \lambda) = 100 - PID(a, b, \lambda) \tag{2.7}$$

The PID function takes three arguments a, b, λ where a and b are the two sequences for which to compute the similarity and λ is an alignment between the sequences. l_a and l_b in equation 2.6 are the lengths of the two sequences a and b respectively. $n_{identical}(\lambda)$ is the amount of identical elements in the alignment. If the number of identical elements in the alignment is equal to the length of the shorter sequence, then the fraction in the equation is equal to one and the overall PID is 100. If the alignment of the two sequences has no identical elements, the PID value is zero. This distance metric is useful for protein sequences because a pairwise sequence alignment (see Chapter 2.2) can be taken into account with λ.

Chapter 3

Machine Learning

3.1 Data Preparation

The data consists of amino acid sequences, which cannot be fed directly into an LSTM model. The data has to be transformed into a mathematical representation. For this thesis the approach of one-hot encoding was taken (see 3.1.1), because it is a broadly used way of encoding categorical data. Another problem is the variable sequence lengths of the protein sequences. This problem is tackled by zero padding and masking the sequences (see 3.1.2).

3.1.1 One-hot Encoding

One-hot encoding is a commonly used method for converting categorical data, like amino acids, into numerical data vectors and protein sequences into matrices. It is achieved by creating a zero vector with the length of the possible values the data can take. In the case of amino acid sequences this would be 21, including the *X* character for unknown AAs. In this zero vector one position is "switched on" by setting it to 1. For example in the case of Alanine (A) only the position referring to this amino acid is set to 1, the remaining 20 positions in the vector stay at 0. This procedure is repeated for the whole protein sequence and leads to a matrix of shape $[n \times 21]$ where n represents the length of the sequence (see Fig. 3.1). The rows of this matrix correspond to the positions of the AAs in the sequence, the columns to the AAs themselves.

$$\begin{aligned}
\mathtt{A} &= [1, 0, 0]\\
\mathtt{C} &= [0, 1, 0]\\
\mathtt{T} &= [0, 0, 1]\\
\mathtt{ACT} &= [1, 0, 0]\,[0, 1, 0]\,[0, 0, 1]
\end{aligned}$$

Figure 3.1: One-hot encoding with three different amino acids.

3.1.2 Padding

Models like LSTMs can handle different sequence lengths if the sequences are processed one at a time. Once the sequences are combined into batches, the models expect the sequences in the batch to be of the same length. Because this is usually not the case for biological sequences like proteins, the data has to be adapted. In order to deal with this problem, a common approach is to use zero padding. For this the shorter sequences are padded by concatenating zeros until they reach the length of the longest sequence in the batch (see Fig. 3.2). When using zero padding, it is important to mask the padded part so that it is not used for the calculations and to keep track of the original sequence lengths. This masking is needed because otherwise the model might change its prediction because of the artificially added padding. The original sequence lengths are needed for what is called *unpadding*. This process is the removal of the previously concatenated zeros to get the prediction for the original sequence lengths. To handle one-hot encoding in combination with padding, an additional position can be added to the possible positions described in Chapter 3.1.1. Then the corresponding position is set to 1 for the padded part of the sequence. The padding token for the one-hot encoded part should then be ignored during the learning procedure of the model, because the artificially added characters should not contribute to the models prediction.

Start-Of-Sequence (SOS) and *End-Of-Sequence* (EOS) tokens can be added to the start and end of the sequences respectively. In a sequence to sequence autoencoder (see Chapter 3.4.1) EOS tokens are needed for the encoder to know the end of the input and gives the decoder the possibility to terminate the prediction for the sequence. SOS tokens are used to initiate the prediction of the decoder together with the latent representation. The two tokens are appended to the sequences prior to the zero-padding described above.

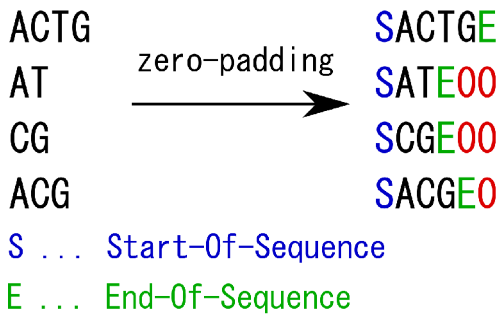


Figure 3.2: Example for zero-padding where the first sequence is the longest with a length of four. The shorter three sequences are padded with zero until they reach the maximal length of four. End-Of-Sequence and Start-Of-Sequence tokens are inserted at their respective position.

3.2 Training Procedure

The data for a supervised machine learning project is usually split into three subsets, the training $\{X_{train}, Y_{train}\}$, validation $\{X_{val}, Y_{val}\}$ and test sets $\{X_{test}, Y_{test}\}$. In supervised machine learning tasks the datasets consist of both the data itself X and some corresponding output labels Y. The goal is to find parameters W for a model $g(x; W)$ such that it is able to map any datapoint x to its respective label y. The first step of the training procedure in supervised machine learning is to fit a model $g(.; W)$ with the weights W to the training data $\{X_{train}, Y_{train}\}$ such that the loss $L(Y_{train}, g(X_{train}; W))$ becomes minimal. A loss function is a measurement of difference between the prediction of the model $g(x; W)$ and the true label y. This loss function depends on the nature of the data. In order to minimize the loss function, a minimization technique called gradient descent (see Chapter 3.2.1) is often applied. The model $g(.; W)$ should be able to map any datapoint to its respective label, including data that was not used for the training process. The validation error, or validation loss, $L_{val}(Y_{val}, g(X_{val}; W))$ is used to evaluate the model on previously unseen data. In Figure 3.3 the Bias-Variance tradeoff is described. Optimizing the model weights W so far that they fit the training data perfectly might lead to the model not being able to generalize to never seen data. This is known as overfitting, this means the model has a high variance. High complexity of the model class can lead to the model overfitting the data. In the training process this can be observed when the training error is decreasing but the validation error is increasing. If the model is not complex enough to fit a mapping from an input x to a label y, it is underfitting the data which corresponds to a high bias. An underfitted model has both a high training and validation error. This problem occurs when the complexity of the model class is not high enough to fin an appropriate mapping for the data. An ideal model class complexity would be in between these two regions.

After the training procedure, a trained model is selected by comparing the trained models based on their performance on the validation set. For this the loss L_{val} or another metric like the F1-score (see Chapter 4.1.2) is calculated for the model's predictions on the validation set. The model with the best performance on the validation set is then taken to perform a final prediction on the test set $\{X_{test}, Y_{test}\}$. It is important that the test set is not touched before this final evaluation to ensure an unbiased estimation of the generalization capability of the model. The performance of the model on the test set is a good estimate to see if it can generalize to never seen data.

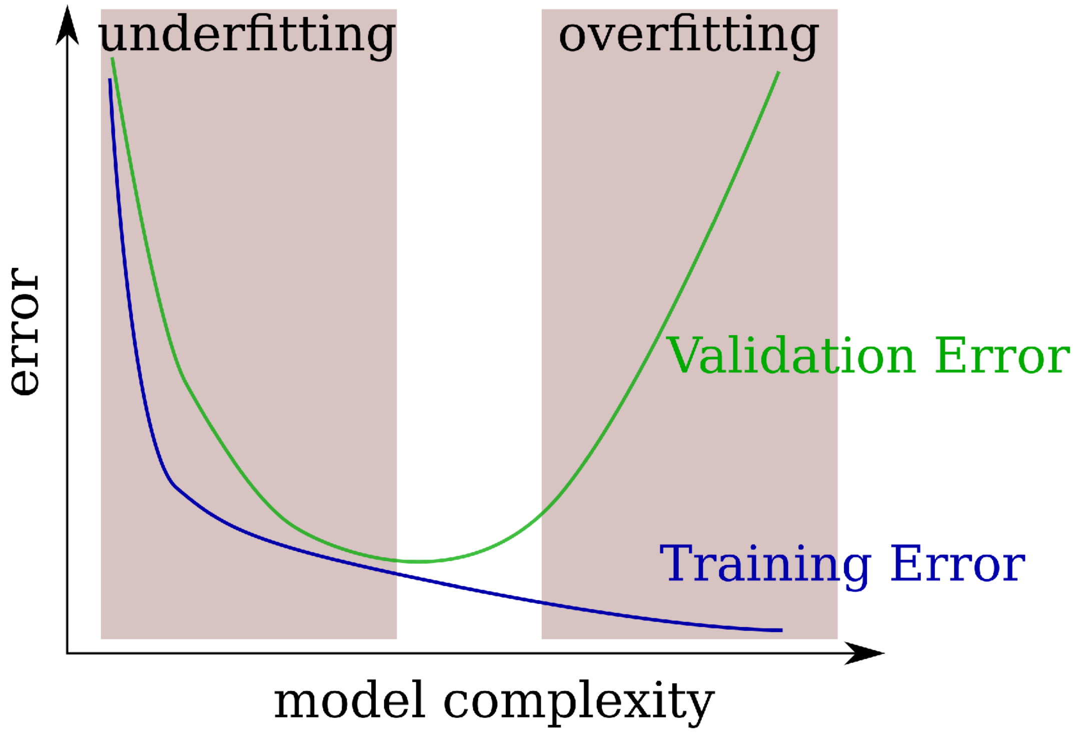


Figure 3.3: Bias-variance tradeoff, on the left side the model is underfitting while on the right side it is overfitting to the data.

3.2.1 Gradient Descent

A loss function $L(y, g(x; W)$ is used to calculate the error between the prediction of the network $g(x; W)$ and the true value y. With the weights W, the network $g(x; W)$ predicts the label y by mapping the input x to the labels. What mathematical form this mapping takes depends on the nature of the data x and the labels y. The gradient $\frac{\delta L(y,g(x;W))}{\delta W}$ combined with a learning rate λ is used to change the weights of the network in a way that reduces the training error. This is achieved by doing multiple steps in the opposite direction of the gradient, described in Equation 3.1. These steps lead to a local minimum of the loss function.

$$w_{new} = w_{old} - \lambda \frac{\delta L(y, g(x; W))}{\delta W} \tag{3.1}$$

The loss function used in a machine learning task depends on the type of data and the goals. For categorical data like protein sequences the cross entropy loss as described in Equation 3.2 may be used for classification. The final step for the network in case of categorical data is usually to map the predictions of the network $\hat{y}$ into the value range $[0, 1]$ with the softmax function. This is achieved by dividing the $e^{\hat{y}_i}$ with the sum of the same value over all possible classes C in the data. This function ensures that the values are in the range $[0, 1]$ and the sum of the predictions is 1. Because of that the values can then be interpreted as probabilities for the individual classes c and the loss $L(y, g(x; W))$ is then calculated using these values.

$$\begin{aligned} L(y, g(x; W)) &= -\sum_{i \in X} y_i \log(g(x_i; W)) \\ g(x_i; W) &= \frac{e^{\hat{y}_i}}{\sum\limits_{c \in C} e^{\hat{y}_i^{(c)}}} \end{aligned} \tag{3.2}$$

Using the whole dataset for one update step can be computationally infeasible if the dataset is big. For that reason stochastic gradient descent is usually used in machine learning where the whole dataset is split into multiple smaller mini-batches and each mini-batch is used to calculate an approximation of the gradient and a step is done using this approximation. More advanced optimization techniques have been introduced. One of them is the Adam optimizer[14] which has been used in this work. Adam uses different effective learning rates for different parameters and uses estimations for the first (see Equation 3.3) and second (see Equation 3.4) momentum of the gradients g. The division by $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$ respectively ensure an unbiased estimator of the momentum values. Both the first and second momentum contain exponentially averaged gradients with decay rates β_1 and β_2 close to 1. The update step of the weights w_t is then performed according to Equation 3.5. ϵ is set to 10^{-8} and is used to prevent a division by zero in the case the second momentum is zero.

$$m_t = \frac{\beta_1 m_{t-1} + (1 - \beta_1) g_t}{(1 - \beta_1^t)} \tag{3.3}$$

$$v_t = \frac{\beta_2 v_{t-1} + (1 - \beta_2) g_t^2}{(1 - \beta_2^t)} \tag{3.4}$$

$$w_t = w_{t-1} - \lambda \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{3.5}$$

3.2.2 Cross-Validation

Splitting into three subsets can be problematic if the original dataset is small. If the validation set is small, then the validation error may not be a good representation of the generalization ability. A small validation set might not cover enough of the possible range of values from the overall data potentially leading to a biased estimator of the performance of the model. The evaluated model might perform worse on samples that are not in the validation set but from another set from the same origin that is not included in the validation set. In k-fold-cross-validation (CV) the dataset is first split into test and training sets in order to increase as much of the available data into training as possible. The test set is then put aside and the training set is randomly subdivided into k folds. Every fold is once used as the validation set, so there are a total of k evaluations. The mean of these k evaluations is then used as a less biased estimator of the models performance. If the data in the k folds are not independent of each other the estimate of the performance is biased. If for example the dataset consists of protein sequences of similar organisms, like in this elaboration, the folds might not be independent if randomly split. For this reason clustered cross-validation is used for this thesis.

For clustered CV the dataset is clustered using an appropriate clustering technique for the given data and task. For this thesis agglomerative hierarchical clustering with single linkage was used (see Chapter 2.3). Single linkage is important to ensure a minimal distance between the clusters to get a less biased estimate of the models performance compared to random splitting. One of the clusters is put aside as the test set and the remaining clusters are similarly used as the folds in k-fold CV.

3.3 Neural Networks

A popular technique in machine learning are the so called neural networks (NNs). This term was coined in 1943 by McCulloch and Pits [17]. Decades of research by various researchers lead to the many different kinds of architectures available today. This journey has been summarized by Schmidhuber in 2015 [26]. For this thesis dense feed-forward neural networks (FNN) have been used for dimensionality reduction and recurrent neural networks (RNN), specifically LSTMs, for extracting data from sequences (see chapters 3.3.1 and 3.3.2). An FNN represents a linear combination of an input vector x and a weight matrix W combined with a non-linear activation function a as described in Equation 3.6. Popular activation functions are the Sigmoid function in Equation 3.7 and the hyperbolic tangent in Equation 3.8. These functions are bounded between 0 and 1 and -1 and 1 respectively. These non-linear activation functions are needed because otherwise multiple layers could be collapsed and expressed by a single calculation, this is described in Equation 3.9. A three layer network consisting of layers g_1, g_2, g_3 with the weights W_1, W_2, $W_3 \in \mathbb{R}^{I \times D1}, \mathbb{R}^{D1 \times D2}, \mathbb{R}^{D2 \times O}$ respectively can be collapsed into a single layer with the weights W_4. Fitting this matrix W_4 to the data can solve the same problems as the three individual matrices combined.

$$g(x; W) = a(W^T x) \tag{3.6}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.7}$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.8}$$

$$g_3(g_2(g_1(x; W_1); W_2); W_3) = W_3^T(W_2^T(W_1^T x)) = (W_3^T W_2 W_1^T)x \tag{3.9}$$

The input into a feed-forward network with the dimensions $x \in \mathbb{R}^D$ combined with the parameters $W \in \mathbb{R}^{D \times I}$ lead to an output $g(x; W) \in \mathbb{R}^I$. The dimensions D and I can be either the same or different. If they are different from each other, the linear layer can be used for dimensionality reduction or even an increase in dimensionality, depending on the use case. For example in convolutional neural networks the final classification from the output of the convolutional layers usually consists of feed-forward networks. If the number of features from the convolutional layer is higher than the number of classes in the dataset, a decrease of dimensionality with the FNN is needed and vice versa [8].

3.3.1 Recurrent Neural Networks

A feed-forward neural network takes a data vector x and produces a prediction vector $\hat{y}$ by using a function $g(x; W)$ with the parameters W and activation function a. Theoretically, a sequence could also be fed into a feed-forward network if all sequence elements x_t are combined into one vector x. This approach however is limited by the need of constant input size for FNNs. Therefore different lengths of sequences cannot be processed this way, because a new weight matrix would be needed for every different length. One approach for learning from sequential data with FNNs was developed in the 1980s for speech recognition. This architecture is called *time-delay neural network* [32]. For this approach an overlapping sliding window of fixed size is used to ensure that the input is of the same size. The idea to take a fixed size window restricts the network from learning dependencies that are outside of the window range. Because of this problem, more specialized architectures, so called recurrent neural networks (RNN), were developed.

RNNs take a sequence x_t, $t \in [1, ..., T]$, as input and produce an output sequence $\hat{y}_t$, $t \in [1, ..., T]$, $\hat{y}_T$ is the last element in this output sequence. For every timestep t the prediction is based on the current input x_t and the context, containing information from the previous timesteps.

An early example of a recurrent neural network is the *Jordan Network* [13] described in

Equation 3.10. In the *Jordan Network* the context, or hidden state, h_t is calculated by first adding up the previous prediction $\hat{y}_{t-1}$ and the input x_t, both projected by the weight matrices U_h and W_h respectively. The activation function σ_h is then applied on the result of this summation. The hidden state h_t is multiplied by a weight matrix W_y and the results are then mapped with the activation function σ_y which results in the prediction $\hat{y}_t$. $\hat{y}_t$ is the prediction of the model for the timestep t based on the input at timestep t and the previous prediction $\hat{y}_{t-1}$.

$$\begin{aligned} h_t &= \sigma_h(W_h x_t + U_h \hat{y}_{t-1}) \\ \hat{y}_t &= \sigma_y(W_y h_t) \end{aligned} \tag{3.10}$$

In the *Elman Network* [5] described in Equation 3.11 the weighted previous hidden state h_{t-1} is summed up together with the weighted input x_t to calculate the hidden state h_t. Similar to the *Jordan Network* an activation function σ_h is applied on this sum. For the prediction $\hat{y}$ the hidden state is weighted by a weight matrix and finally another activation function is applied.

$$\begin{aligned} h_t &= \sigma_h(W_h x_t + U_h h_{t-1}) \\ \hat{y}_t &= \sigma_y(W_y h_t) \end{aligned} \tag{3.11}$$

The most common approach of learning for these RNNs is a method called *backpropagation through time* (BPTT). This technique was developed by various researchers independently [33][24][19]. In BPTT the network is unfolded along the time axis, meaning that for every timestep t an FNN with x_t and the context h_{t-1} as input and $\hat{y}_t$ as output is constructed. These FNNs have the same architecture and share their weights. For the first timestep usually a zero vector is used as the context input.

3.3.2 Long Short-Term Memory

In his diploma thesis Hochreiter showed mathematically that BPTT with unfolding the network has a serious problem - the *vanishing gradient problem* [11]. The problem is that because of the chain rule at every timestep t the gradient of the activation function is multiplied with the gradient of timestep $(t + 1)$. This becomes a problem, if the activation functions have gradients that are smaller or larger than one, because repeatedly multiplying values smaller than 1 leads to an exponential decay, a *vanishing*, of the gradients. Gradient values larger than 1 lead to the *exploding gradient problem*, where, as the name describes, the gradients get very big. Both *exploding* and *vanishing* gradients have a detrimental effect on the learning process. The *vanishing* gradient might slow down the process or even make it impossible while the *expoloding* gradient might cause the network to oscillate around a minimum and stop the convergence to it. This problem is not only the case with RNNs, but for every network using backpropagation if there are many layers. Because for RNNs mostly the logistic function and the hyperbolic tangent were used, with both having gradients smaller or equal to 1, no long term dependencies could be learned because of the described problem.

Hochreiter and Schmidhuber came up with a solution to this problem with the *long short-term memory* (LSTM) in 1997 [12]. This architecture (Equations 3.12 to 3.17) solves the vanishing gradient problem by introducing a constant error carousel, where the derivative of the memory cell with respect to the previous timestep is 1. This leads to stable derivatives and for that reason LSTMs can learn long term dependencies. The original architecture by Hochreiter and Schmidhuber did not contain the so called forget gate f_t in Equations 3.14 and 3.16.

For this elaboration the PyTorch (version $1.8.1$) [22] version of the LSTM was used, which includes the forget gate as described by Gers et. al in 1999 [7]. The architecture of this specific LSTM model is described in equations 3.12 3.13 3.14 3.15 3.16 3.17.

An LSTM-cell has for every timestep the input x_t from the sequence and two inputs from the previous timestep, the hidden state h_{t-1} and the memory cell c_{t-1}. The input gate 3.12 and output gate 3.13 are used to control how much information is stored in the cell state and how much is propagated to the next timestep. The additional forget gate 3.14 is a mechanism that enables the cell to forget past cell memory. These gates all have the logistic function σ which shuts the gate if the value is 0 and opens it if the value is 1. Equation 3.15 describes the cell input which is a combination of the current input x_t and the previous output h_{t-1}, the hyperbolic tangent is used as non-linearity. 3.16 defines the equation for the cell memory, where the forget gate defines the amount that is remembered and the input gate the amount that is newly learned. Equation 3.17 shows how the output gate limits the information flow of the cell memory to the output.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{3.12}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{3.13}$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{3.14}$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{3.15}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{3.16}$$

$$h_t = o_t \odot \tanh(c_t) \tag{3.17}$$

3.4 Autoencoder

Autoencoders (AE) are machine learning architectures that can be used for information compression and feature extraction. An AE consists of an encoder $f(x)$ that takes the input x and converts it into a latent code h like in Equation 3.18. This code h is then used by a decoder $g(h)$ to try and create a reconstruction r of the original input (see Equation 3.19). Using an identity function for both f and g would lead to a perfect reconstruction, but this wouldn't be of any use. For this reason AEs have certain restrictions on the encoder, for example a reduction in dimensionality. With this the AE has to prioritize certain aspects of the input and learn a new representation of the input data. This reduction in dimensionality can be seen as a compression of the original data x. AEs that reduce the dimensions like that are called *undercomplete autoencoders*, such an architecture is depicted in Figure 3.4[8]. When the encoder and decoder consist of linear functions the compression is comparable to principal component analysis (PCA). The difference is that PCA is a linear transformation but AEs are not limited to linear functions. The axis in PCA are ordered according to their contribution to the variance in the data, however for autoencoders there is no ordering of the axis in the latent space [15].

For learning how to compress the input in an AE, gradient descent is performed on a loss function $L(x, g(f(x)))$ between the input x and the reconstruction $g(f(x))$. This loss function depends on the type of input x, for numerical data the mean squared error (see Equation 3.20) is used. The mean squared error is squared difference between the actual value x_i and the prediction of the autoencoder $g(f(x))$ averaged with the number of datapoints n in x. This average is computed with the arithmetic mean. For categorical data like amino acid sequences a viable loss function is the cross entropy loss (see Equation 3.2).

$$h = f(x) \tag{3.18}$$

$$r = g(h) \tag{3.19}$$

$$L(x, g(f(x))) = \frac{1}{n}\sum_{i=1}^{n}(x_i - g(f(x_i)))^2 \tag{3.20}$$

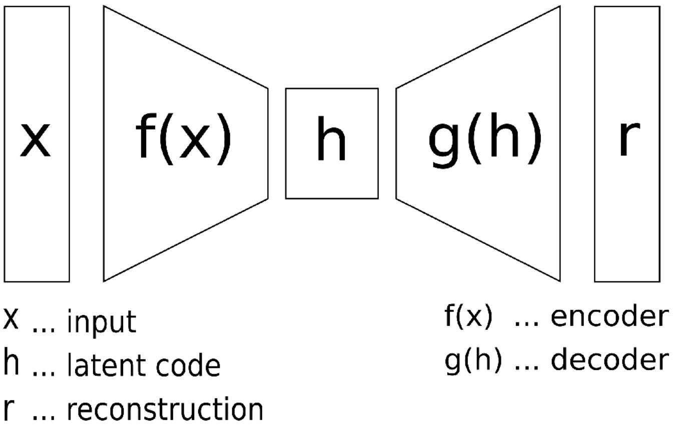


Figure 3.4: Undercomplete autoencoder structure with x being the input, $f(x)$ the encoder, h the latent code, $g(h)$ the decoder and r the final reconstruction of the input.

3.4.1 Sequence to Sequence Autoencoder

In the case of regular AEs the encoder and decoder are FNNs, which as mentioned in Chapter 3.3.1 are not well suited for sequences. For sequential data such as protein sequences, sequence to sequence AEs are used [29]. Both the encoder and decoder are RNNs for these AEs, in this thesis LSTMs were used. The encoder LSTM processes the padded and one-hot encoded sequences until the last timestep where the hidden state and the cell state combined form the latent code. For the latent representation of an encoded sequence the hidden state of the last timestep is used. This latent code is then used to initialize the decoder LSTM, which together with the initialization and a *Start-Of-Sequence* token predicts the first element of the reconstruction. The decoder then uses its own prediction of the previous timestep as input to predict the next element until the length of the input sequence is reached. The end of the sequence is marked with an *End-Of-Sequence* token and if the sequence to sequence AE is used for generating new sequences, the generation process ends with this token. The decoder LSTMs outputs have the same dimensionality as the latent code, which is most of the time different from the input dimensions. Because of that an FNN is used to reduce the dimensionality to the original input dimensions. A sequence to sequence autoencoder architecture is visualized in Figure 3.5.

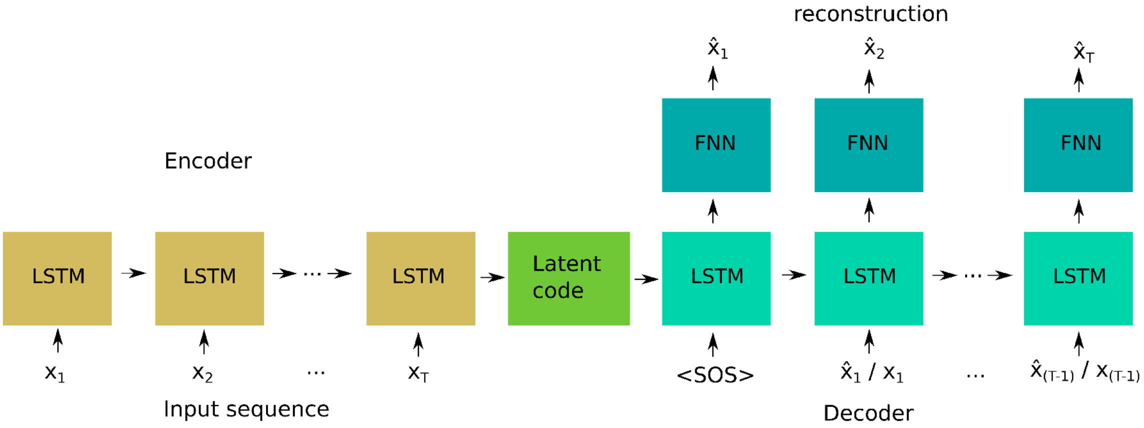


Figure 3.5: Scheme of a sequence to sequence autoencoder with an LSTM for encoder and decoder. The encoder takes the original sequence $[x_1, ..., x_T]$ and the latent code is both the hidden state and the cell state at the last timestep. The hidden state is used as the representation of the encoded sequence. The decoder takes in a *Start-Of-Sequence* token (<*SOS*>) and then uses its own predictions $[\hat{x}_1, ..., \hat{x}_{(T-1)}]$ or in case of teacher forcing the original sequence $[x_1, ..., x_{(T-1)}]$ to create the reconstruction $[\hat{x}_1, ..., \hat{x}_T]$.

3.4.2 Teacher Forcing

The decoder of the sequence to sequence autoencoder utilizes its own previous prediction to predict the next element in the sequence. This means that it is depending on its own ability to reconstruct the input. If the decoder is not able to produce an accurate prediction at the beginning of the sequence the following inputs for the decoder will also be inaccurate. This could lead to the decoder not being able to learn how to predict the correct sequence. This problem can be tackled with an approach called *teacher forcing*, which was already discussed in the context of RNNs by Williams and Zisper [34]. The idea is to use the ground truth to keep the predictions of the network close to the truth speeding up the process of learning how to reconstruct the input. For this thesis teacher forcing is implemented by using the original sequence element x_t as input for the decoder instead of the prediction $\hat{x}_t$. The choice whether or not to perform teacher forcing is done with a fixed probability p_{TF}, described in Equation 3.21.

$$\hat{y}_t, (h_t, c_t) = \begin{cases} LSTM_{decoder}(\hat{x}_{t-1}; (h_{t-1}, c_{t-1})) & if\ TF = False \\ LSTM_{decoder}(x_{t-1}; (h_{t-1}, c_{t-1})) & if\ TF = True \end{cases} \tag{3.21}$$
$$P(TF = True) = p_{TF}$$

3.5 Clustering of Latent Space

The latent space, the code between the encoder and decoder, is the autoencoder's internal representation of the input. Because of the dimensionality reduction of undercomplete autoencoders, certain features from the input might be lost in the process of the compression. The dimensionality reduction can however also compress the information and create a more information dense representation of the input. This compression is comparable to principal component analysis (PCA) when the encoder and decoder are linear functions [15]. However the encoder and decoder in a sequence to sequence autoencoder are not linear and therefor the representation is also more complex. In the case of categorical data, like protein sequences, the input needs to be converted into a numerical representation before it can be analyzed by algorithms like k-means or k-nearest neighbors. A sequence to sequence autoencoder is used in this work to convert the amino acid sequences into numerical representations. Specifically the last hidden state of the encoder LSTM clustered with both the *k-means* algorithm [16] as well as the *k-nearest neighbors* algorithm.

3.5.1 K-means Clustering

The k-means algorithm is an iterative algorithm that forms k clusters from the data in the following three steps.

I. Initialization of k random clusters $[C_1, \ldots, C_k]$ with means $m_1^0, \ldots, m_k^0$

II. Assigning all datapoints $[x_0, \ldots, x_i]$ to the cluster C_i with the nearest mean m_i^t

$$x_n \in C_i^t \quad if \quad ||x_n - m_i^t||^2 < ||x_n - m_j^t||^2 \quad \forall \; j \; \in \{[1, \ldots, k] \setminus i\}$$

III. Calculating new means $[m_1^{t+1}, \ldots, m_k^{t+1}]$ of the clusters by dividing the sum of the values in the clusters by the number of members $|C_i^t|$

$$m_i^{t+1} = \frac{1}{|C_i^t|} \sum_{x_i \in C_i^t} x_i$$

Steps two and three are repeated until the clusters no longer change or another stop condition is reached. The above steps describe the algorithm with one dimensional data. The algorithm can be used for higher dimensional data as well, the distances as well as the calculation of the means would need to be adapted to the higher dimensionality.

One difference between k-means clustering and the hierarchical clustering algorithms described in Chapter 2.3 is the need of a fixed number of cluster centers. This needs prior knowledge or assumptions about the data. Because there are three different groups of organisms in the data, k-means with three centers was used for the analysis of the latent space. The k-means implementation of *scikit-learn* (version $0.24.1$) [23] was used in this thesis.

3.5.2 K-nearest Neighbors Clustering

The idea of the k-nearest neighbors algorithm dates back all the way to 1951 when Fix E. and Hodges J. L. introduced a non-parametric classification approach that is now known as the k-nearest neighbor rule [6]. K-nearest neighbors clustering is one of the simplest supervised classifiers to implement and works in a straightforward way. In the first step the datapoints from the training set are stored together with their labels. The unknown datapoints from a test set are then classified individually. For this a distance measure is needed, usually the Euclidean distance (see Equation 3.22) is taken for continuous variables. The unknown datapoint is then assigned to the most common class in the k points that are closest to it, its k-nearest neighbors.

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} \tag{3.22}$$

3.6 Visualization of Latent Space

Because k-means clustering and k-nearest neighbors operate without dimensionality reduction, they are not well suited for visual analysis. For visual inspection of high dimensional spaces a wide variety of techniques have been developed that aim at decreasing dimensionality. There are certain problems connected with the reduction of the dimensions. In an n-dimensional space, it is possible to have $n + 1$ points that are mutually equidistant. This cannot be represented in a two dimensional space, this problem is known as the crowding problem [31]. There are two different categories of dimensionality reduction techniques, one that aims towards maintaining pairwise distance structure in between all data points and another that prioritize local distances over global distances [18].

3.6.1 t-SNE

t-distributed stochastic neighbor embedding (t-SNE) by van der Maaten and Hinton [31] is considered the state-of-the-art algorithm for dimensionality reduction for visualization. It is based on stochastic neighbor embedding (SNE) by Hinton and Roweis [10]. The reduction in SNE is achieved by calculating the conditional probability $p_{j|i}$ of datapoint x_i to choose x_j as its neighbor in the high dimensional space, see Equation 3.23. The low dimensional counterpart to $p_{j|i}$ is the conditional probability $q_{j|i}$ with the low dimensional points y_i and y_j, see Equation 3.24. If the low dimensional mapping from x_i and x_j to y_i and y_j is correctly modeled, then the probabilities $p_{j|i}$ and $q_{j|i}$ will be equal. Using the Kullback-Leibler divergence as an objective (Equation 3.25), SNE aims to find a low dimensional representation that minimizes this divergence. In the case of t-SNE, for the high dimensional space a Gaussian density function and for the low dimensional space a Student's t-distribution with 1 degree of freedom centered around x_i, is used to calculate the low dimensional representation. Via gradient descent the objective is minimized, which leads to the distribution in the low dimensional space being close to the one in the high dimensional space, preserving similarities between points.

$$p_{j|i} = \frac{exp(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2})}{\sum_{k \neq i} exp(\frac{\|x_i - x_k\|^2}{2\sigma_i^2})} \tag{3.23}$$

$$q_{j|i} = \frac{exp(-\|y_i - y_j\|^2)}{\sum_{k \neq i} exp(\|y_i - y_k\|^2)} \tag{3.24}$$

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} log \frac{p_{j|i}}{q_{j|i}} \tag{3.25}$$

3.6.2 UMAP

Uniform manifold approximation and projection (UMAP) by McInnes et. al [18] is another suitable technique for dimensionality reduction for the sake of visualization. This algorithm is based on Riemannian geometry and algebraic topology. The first step for the UMAP algorithm is to construct a fuzzy topological representation of the high dimensional data based on simplicial complexes and Riemannian geometry. The next step is to create a low dimensional representation and to optimize it such that it is as close to the high level representation as possible. This is achieved with stochastic gradient descent. In their paper McInnes et. al [18] demonstrate that the UMAP algorithm's benefit over t-SNE is that it is scalable to large datasets and it is better at conserving global differences.

Chapter 4

Methods and Results

4.1 Methods

4.1.1 Preprocessing of Data

Clustered cross-validation with agglomerative hierarchical clustering using the distance associated with the PID (see Chapter 2.3) was used as the training procedure of the autoencoder for this thesis. For calculating the distance with PID, a sequence alignment is needed. For that purpose both local and global sequence alignments (see Chapter 2.2) on all sequences combined as well as the three kingdoms separated were compared.

Although the sequences for this thesis tend to be of a similar size, a few outliers in sequence length caused problems when using global sequence alignment. For that reason local sequence alignment using the Smith-Waterman algorithm was used. Because the sequences are from the hemoglobin family, and therefore closely related, *BLOSUM90* was used as the scoring matrix. To avoid having many gaps in the alignment, an affine gap penalty with a penalty of 5 for opening and 2 for extending a gap was applied.

The aligned sequences were used to calculate the PID and then the corresponding distance. This distance was then taken to perform agglomerative hierarchical clustering with single linkage. Aligning and clustering all sequences together lead to clusters containing only one of the three kingdoms. This is to be expected because the evolutionary distance between kingdoms is bigger than the distance within the kingdoms. The fact that each cluster only contains one kingdom is not optimal for clustered cross-validation because each cluster is not a good representation of the whole population. For that reason a stratified sampling approach was taken where the data was first divided into the three subgroups given by the kingdoms. These three subgroups were then separately aligned and 6 clusters were formed using agglomerative hierarchical clustering with single linkage. The PID values were chosen in a way that two conditions are met. (i) The number of clusters should not be big, because the bigger this number, the more clusters with a single member are formed. (ii) The mean size of the clusters should be small because with only a few big clusters, no separation can be achieved. A good estimate for a PID value, that satisfies these two conditions is the point of intersection visualized in Figuer 4.1. For both the mammals and birds a PID of 90% and for the fish 75% was used to create 6 clusters. This ensures that the distance between the clusters is at least 10% and 25% respectively. One cluster of each kingdom was put aside as test set which lead to 139 of the 736 total sequences. The remaining 5 clusters were used as the subset for the clustered CV.

The sequences from the 6 clusters were further processed by converting them to $[n \times 24]$ one-hot encoded matrices, where n is the length of the sequences. The 24 possible values consist of the 20 amino acids and the *unknown token X*, as well as *Start-Of-Sequence* and *End-Of-Sequence* tokens and a padding index for ignoring the predictions of the network in the padded region of the input sequences. For zero-padding and later *unpadding* the sequences, the module *torch.nn.utils.rnn* from PyTorch (version $1.8.1$) [22] was used. This library, in combination with the module *torch.nn.rnn*, provides an efficient implementation for dealing with different sequence lengths.

For loading the data and the creation of mini-batches, the $DataLoader$ class of the PyTorch (version $1.8.1$) [22] library was utilized. The number of sequences in these mini-batches, the so called batch size, is a hyperparameter for the learning process of the AE model and part of the hyperparameter search described in Chapter 4.1.2.

4.1.2 Autoencoder

The sequence to sequence autoencoder in this thesis consists of an LSTM as encoder and an LSTM combined with a dense feed-forward neural network as decoder. To build the autoencoder architecture, the implementations of LSTMs as well as the FNN from the PyTorch (version $1.8.1$) [22] library are utilized.

The encoder $LSTM_e$ maps the one-hot encoded and padded amino acid sequences S into the latent code H (see Equation 4.1). S is a three dimensional tensor with the length of the padded sequences l, the batch size b and the 24 possible values for the one-hot encoding

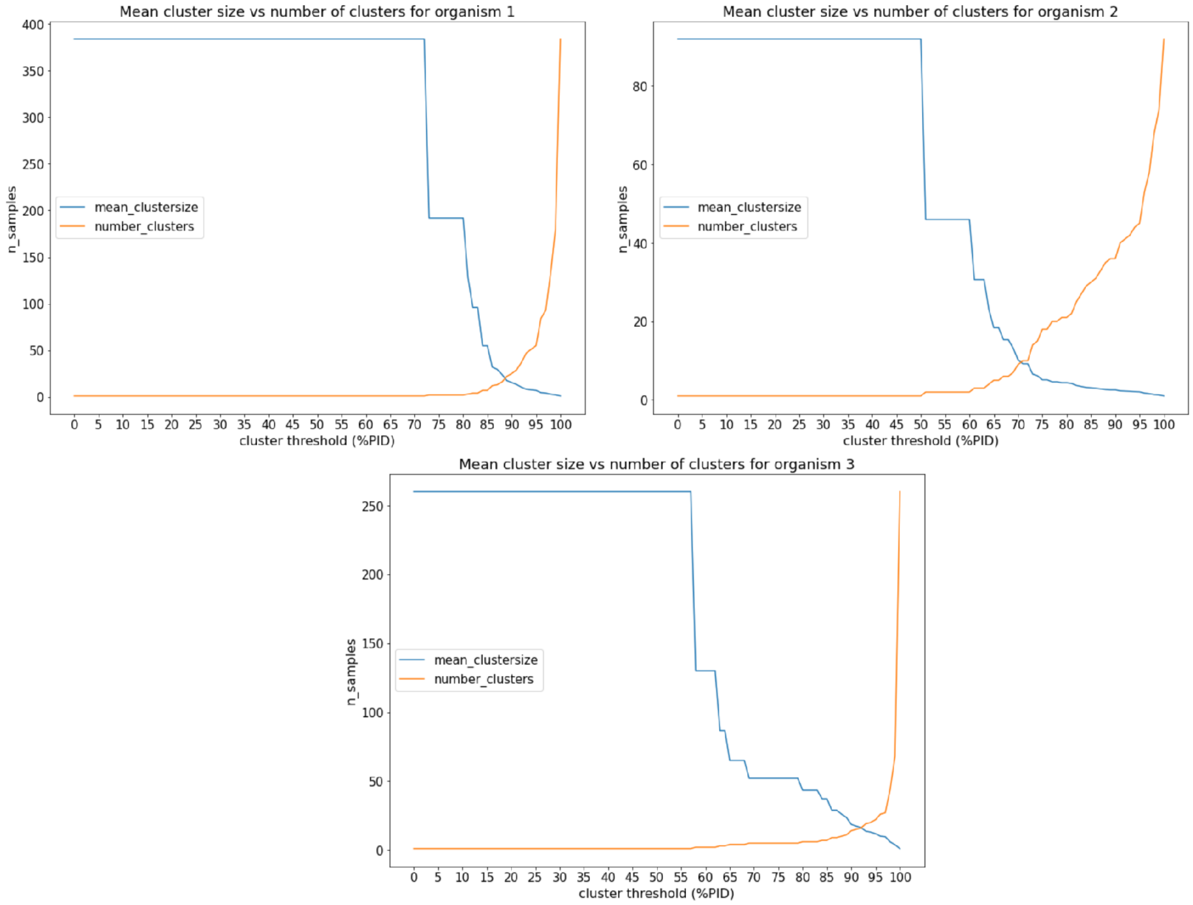


Figure 4.1: Comparison between mean cluster size and number of sequences per cluster. The clusters are formed with the PID specified on the x-axis, the y-axis show the number of samples.

described in Chapter 4.1.1 as dimensions. H contains both the hidden state h as well as the cell state c, both of them are matrices with the batch size b and the latent dimension d as the dimensions. The values of h and c are from the last timestep of the input sequence, for padded sequences this is the last non-padded timestep. For timestep 1 in the encoding process c_0 and h_0 (see Equations 3.12 to 3.16) are zero matrices.

$$\begin{aligned} & H = LSTM_e(S;(h_0,c_0)) \\ & S \in 0,1^{l\times b\times 24}, \quad H \in \mathbb{R}^{2\times b\times d} \\ & h_0, c_0 \in 0^{b\times d} \end{aligned} \tag{4.1}$$

The code H is used as the initial hidden state and cell state combined with an SOS token for the decoder $LSTM_d$ to start the decoding process (see Equation 4.2). Because the output of the decoder LSTM $\hat{y_t}$ has the dimensionality of the latent code, an FNN is used to change the dimensionality to *24* to match the original input. The output of the FNN $\hat{s}_t$ is then used as the prediction of the network at the current timestep t. For predicting the next amino acid, the output as well as the hidden and cell state from the previous timestep are used (see Equation 4.3). With a certain fixed probability p_{TF} teacher forcing was done where the amino acid from the original sequence S is used instead of the prediction of the decoder (see Equation 3.21). Both the latent dimension d as well as the teacher forcing probability p_{TF} are important hyperparameters and part of the hyperparameter search.

$$\begin{aligned} &\hat{y_1}, (h_1, c_1) = LSTM_d(SOS; H) \\ &\hat{s}_1 = FNN(\hat{y_1}) \\ &\hat{y_1} \in \mathbb{R}^{b \times d}, \quad \hat{s}_1 \in \mathbb{R}^{b \times 24} \end{aligned} \tag{4.2}$$

$$\begin{aligned} &\hat{y_t}, (h_t, c_t) = LSTM_d(\hat{s}_{t-1}; (h_{t-1}, c_{t-1})) \\ &\hat{s}_t = FNN(\hat{y_t}) \\ &\hat{y_t} \in \mathbb{R}^{b \times d}, \quad \hat{s}_t \in \mathbb{R}^{b \times 24} \end{aligned} \tag{4.3}$$

The PyTorch (version $1.8.1$) [22] implementation of the Adam optimizer [14] combined with the multiplicative learning rate scheduler $StepLR$ (see Figure 4.2) was used for the training procedure. This learning rate scheduler multiplies the learning rate λ of the optimizer with the number γ after a fixed number of epochs. The learning rate λ, γ and the number of epochs for each learning rate update were part of the hyperparameter search.

Finding good parameters for the mentioned hyperparameters was done, as previously described, with clustered cross-validation (see Chapter 3.2.2). As an approach for hyperparameter search a mixture of grid search and manual search was chosen. Grid search is a generally applicable and exhaustive search with predefined parameter value combinations. The model is trained with all of these combinations and compared given some evaluation metric. Manual search, on the other hand, is more tuned towards a specific problem and non-exhaustive. The parameter values are chosen with experience or prior knowledge about the task and the

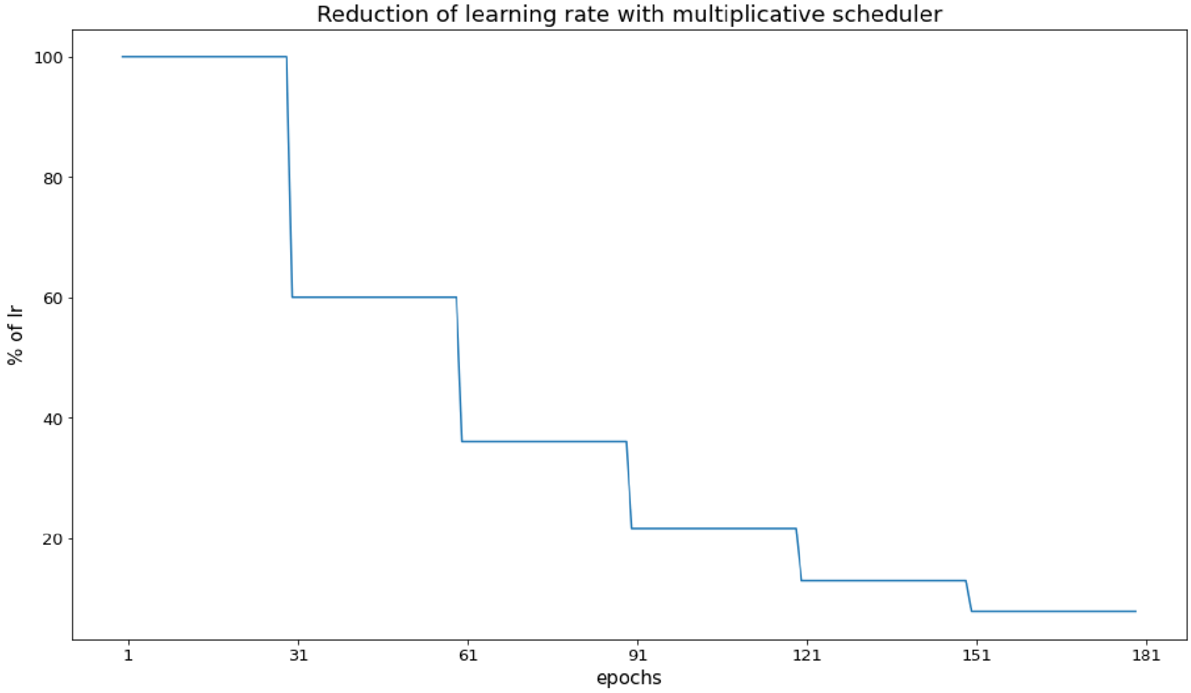


Figure 4.2: Percent reduction of the learning rate with an multiplicative learning rate scheduler. The step size is set to 30 and the γ to 0.6

model is trained with these values and the performance is compared. In this elaboration, the grid search was used to get an estimate for good hyperparameter values for the batch size, the latent dimension and the learning rate λ. The manual search was applied to find good values for λ, the step size and γ of the learning rate scheduler and the probability for teacher forcing p_{TF} (see Equation 3.21).

To compare the performances of the hyperparameter settings the F1-score (*F1*) was used, which is the harmonic mean between the precision and the recall (see Equation 4.4). The recall, or true positive rate (TPR), is the ratio between the number of predicted amino acids that match the original AA, the true positives (*TP*), and the total number of this AA in the original sequence (*P*). Precision, or positive predictive value (*PPV*), is the number of true positives divided by the sum of TP and the number of times when the predicted AA does not match the original AA. In order to determine the values *TP*, *P* and *FP* an approach called one-versus-rest was taken. With this approach, a prediction for an AA either matches the original AA (*TP*), or the prediction does not match and is counted as a false positive (*FP*). A higher F1-score means a better performance of the hyperparameter setting.

$$
\begin{aligned}
TPR &= \frac{TP}{P} \\
PPV &= \frac{TP}{TP + FP} \\
F_1 &= \frac{2}{TPR^{-1} + PPV^{-1}}
\end{aligned}
\tag{4.4}
$$

The value ranges for the grid search are shown in Table 4.1.2. The results of the grid search can be summarized like the following. A run with batch size 32 is faster to finish than with batch size 8 but has a worse performance after the same amount of epochs. Also the performance for batch size 8 was slightly better than 32. For the latent dimension the highest one with 512 had the best performance out of tested values. Increasing this value further would likely increase the F1-score but would decrease the amount of reduction in space. For the learning rate λ the values 0.001 and 0.0001 outperformed the higher learning rate of 0.01 but between the two values the performance was comparable.

hyperparameters	values
batch size	8, 16, 32
latent dimension	8, 16, 32, 62, 128, 256, 512
learning rate λ	0.01, 0.001, 0.0001

Table 4.1: Values for hyperparameter grid search.

After getting an estimate for good values for the batch size, the latent dimension and the learning rate a manual search to further optimize the learning rate, the hyperparameters of the learning rate scheduler and the probability for teacher forcing was performed (see Table 4.1.2. The batch size and latent dimension were fixed to 8 and 512 respectively for this procedure. It was first discovered, that a teacher forcing probability of 0%, so no teacher forcing, was performing the best. This is probably due to the output of the autoencoder not being one-hot like the original sequence. For the learning rate scheduler two different value combinations were taken, a high step size combined with a low γ and vice versa. A low step size with small γ leads to a too fast decay of the learning rate which hinders the learning process and too large values only have a minor impact on the learning process. In the end the best values for the learning rate scheduler were a step size of 30 and a γ of 0.6. The best performing model during the clustered CV had a learning rate of 0.0005. This model with the previously described parameters had an average F1-score over the 5 folds of 0.684 with a standard deviation of 0.02 and was the best after 200 epochs of training.

hyperparameters	values
learning rate	0.0001, 0.0003, 0.00045, 0.0005, 0.00055, 0.00075, 0.001
teacher forcing	0, 0.1, 0.2, 0.5, 0.75
step size	20, 25, 30, 35, 50, 75
γ	0.1, 0.3, 0.5, 0.6, 0.7, 0.75, 0.8, 0.9

Table 4.2: Values for manual hyperparameter search.

4.1.3 Clustering of Latent Space

For the clustering of the latent space, both algorithms k-nearest neighbors and k-means (see Chapter 3.5) were used to cluster the latent space, both on the cell state and on the hidden state. The implementations of the scikit-learn library (version 0.24.1) [23] were taken for both approaches. For the k-means algorithm 3 cluster centers were assumed, corresponding to the different kingdoms of animals. This algorithm performed poorly on the data at hand, presumably because of some outliers in the data and the vulnerability of the k-means algorithm against these outliers. The k-nearest neighbors algorithm with is more robust towards outliers and for that reason it was also applied for the task with $k = 5$.

For the visualization UMAP [18] and t-SNE [31] were used. Both the hidden state h as well as the cell state c were visualized.

4.2 Results

4.2.1 Autoencoder

The best hyperparameter values found through the search were a learning rate of 0.0005, a batch size of 8, a latent dimension of 512, teacher forcing probability of 0. The values for the learning rate scheduler were a γ of 0.6 and a stepsize of 30. On the 5 training sets the models with these parameter settings had an average F1-score of 0.690 with a standard deviation of 0.203. On the validation sets the average F1-score was 0.684 with a standard deviation of 0.209. In Table 4.3 the performance of the 5 models on their respective validation set is described. The numbers for *Labels*, *Predictions* and *True positives* are summed up, the value in *F1-score* is averaged over the 5 folds. The amino acid with the most predictions and the most appearances in the data is alanine A, the models reach an average F1-score of 0.655 on Alanine. The lowest F1-score on an amino acid is isoleucine with 0.369 and the highest is arginine with 0.868. A potential interpretation of the different F1-scores is that the chemical property of the AA has an impact on the model's ability to reconstruct the input. The AA with the lowest F1-score is isoleucine, which is part of the hydrophobic group see Table 2.1. Arginine, the most common amino acid in the data, is also part of this group. The low difference in chemical properties could mean that positions with arginine and isoleucine are more variable between the sequences of the different animals, for example an arginine could be replaced with an isoleucine in the evolutionary process. The amino acids with a high F1-score, like arginine and histidine are in the positively charged group. The properties of this group are highly different than the hydrophobic group, because of that they could be essential for the protein function and might not be easily substituted by another AA. This could lead to the positions with arginine being very similar in the different sequences from the different animals. The *End-Of-Sequence* token has a high F1-score of 0.903 with 556 correctly predicted out of 597.

Class	Labels	Predictions	True positives	F1-score
A	11222	14026	8285	0.655
R	1928	1745	1596	0.868
N	1861	1597	1205	0.700
D	4990	4455	3771	0.798
C	945	575	509	0.672
Q	1461	814	608	0.529
E	2679	2282	1483	0.598
G	5099	3739	2969	0.672
H	5356	6526	4929	0.828
I	1994	1128	592	0.369
L	9430	10292	7662	0.776
K	6741	6847	5872	0.864
M	1518	1094	1014	0.776
F	4275	3540	2918	0.746
P	3513	3546	2917	0.826
S	5907	6371	4205	0.684
T	4613	3858	2902	0.684
W	420	479	261	0.512
Y	2049	2303	1843	0.847
V	7172	7980	5615	0.742
X	69	0	0	0
SOS	0	0	0	0
EOS	597	642	556	0.903
Padding	0	0	0	0

Table 4.3: Predictions of the model with the best performing hyperparameter settings on the validation sets in cross-validation. First column represents the one letter code of the amino acids plus the *Start-Of-Sequence* and *End-Of-Sequence* as well as the Padding token. The second column are the true labels in the data summed up over the 5 folds, followed by the predictions of the 5 models and the true positives. The F1-score is averaged over the 5 folds.

The model with the best hyperparameters was trained on the whole training data, which are the 5 CV-folds combined, and then tested on the test set. On the training data this model achieved an average F1-score of 0.694 with a standard deviation of 0.189. This is slightly higher than the F1-score average for the training sets during CV. Table 4.4 shows the labels of the data, the total number of predictions, the true positives and the F1-score per class for the model on the test data. The F1-score is in general lower than the values for the validation sets in CV. The mean F1-score is 0.426 with a standard deviation of 0.145. The highest F1-score is Lysine with 0.551, the lowest one is Isoleucine with 0.161. These results support the previously stated interpretation, that the chemical group the AA belongs to has an impact on the ability of the network's ability of reconstruction. The *End-Of-Sequence* token is again the highest F1-score for any token with 0.662.

Class	Labels	Predictions	True positives	F1-score
A	2383	3117	1310	0.476
R	509	422	235	0.505
N	578	416	146	0.294
D	1191	1003	536	0.489
C	203	109	50	0.321
Q	504	241	96	0.258
E	675	485	184	0.317
G	1069	933	441	0.441
H	1154	1492	700	0.529
I	422	250	54	0.161
L	2298	2477	1205	0.505
K	1472	1625	853	0.551
M	355	230	154	0.526
F	1031	796	413	0.452
P	802	842	431	0.524
S	1451	1506	671	0.454
T	1051	797	384	0.416
W	157	110	69	0.517
Y	496	516	252	0.498
V	1472	1922	824	0.486
X	4	0	0	0
SOS	0	0	0	0
EOS	139	127	88	0.662
Padding	0	0	0	0

Table 4.4: Predictions of the model with the best performing hyperparameter settings on the test set. First column represents the one letter code of the amino acids plus the *Start-Of-Sequence* and *End-Of-Sequence* as well as the Padding token. The second column are the true labels in the test set, the third are the number of predictions and the fourth the true positives. The fifth column is the F1-score of the model for each class.

4.2.2 Clustering of Latent Space

The latent space was clustered using both k-means and k-nearest neighbors classifiers. For this the classifier was fit to the training data and evaluated with the validation set for each fold in the cross-validation. For the final model the same process was applied to the training data and the test data. This was done for the latent variable h from Equation 3.17 as well as the cell state c from Equation 3.16. The complete data set is unbalanced with more than 4 times as many sequences of mammals than of fish. Because of that, the balanced accuracy metric from the scikit-learn library (version 0.24.1) [23] was taken to assess the performance of the clustering. This metric ACC_b is calculated by averaging the recall TPR_c for each class c over the number of total classes $|C|$ (see Equation 4.5. For the cross-validation the k-means algorithm with 3 centers achieved a balanced accuracy of 0.423 on the training data and 0.380 on the validation sets on the latent variable h, averaged over the 5 folds. On the cell state c the balanced accuracy is lower with 0.352 on the training data and 0.308 on the validation sets. This balanced accuracy score is low and an issue is that there are folds where the algorithm only predicts one class, which yields a balanced accuracy of $\frac{1}{3}$. A possible explanation of this low balanced accuracy and the dominant prediction of one class is that the k-means algorithm is not robust towards outliers. Although the sequences are very similar to each other, some of them are very different. For example there is one sequence in the data that has a length of 9 and this sequence is sometimes isolated in one of the three clusters and or in a cluster with very few members. To overcome the problem with outliers, the more robust k-nearest neighbor classifier was applied on h and c. This classifier achieved a mean balanced accuracy score of 0.975 and 0.977 on the training sets and 0.942 and 0.949 on the validation data for variable h and c respectively. Tables 4.5 and 4.6 show the values described above.

$$ACC_b = \frac{\sum\limits_{c \in C} TPR_c}{|C|} \tag{4.5}$$

For the final model the latent variables c and h were classified for both the training and the test set. On the test data the balanced accuracy for the k-means algorithm on the latent variable h was 0.255 and on the training data it was 0.371. For the clustering of the cell state c the balanced accuracy on the training set was 0.363 and 0.317 on the test set. Again this balanced accuracy score is low and near the $\frac{1}{3}$ accuracy for predicting only 1 class. The balanced accuracy on the training set for k-nearest neighbors is 0.986. The predictions for the classifier

Algorithm	Training fold 1	Training fold 2	Training fold 3	Training fold 4	Training fold 5	Folds average
k-NN h	0.987	0.945	0.980	0.982	0.983	0.975
k-NN c	0.995	0.974	0.987	0.977	0.977	0.982
k-means h	0.620	0.388	0.395	0.368	0.346	0.423
k-means c	0.276	0.373	0.398	0.372	0.341	0.351

Table 4.5: Balanced accuracy scores for the k-nearest neighbor (k-NN) and the k-means algorithms on the latent variable h and the cell state c. The values depicted are from the 5 training sets from the 5 CV-folds individually and the average of these 5 sets.

Algorithm	Validation fold 1	Validation fold 2	Validation fold 3	Validation fold 4	Validation fold 5	Folds average
k-NN h	0.831	0.971	0.956	0.977	0.977	0.942
k-NN c	0.871	0.955	0.966	0.977	0.977	0.949
k-means h	0.476	0.320	0.301	0.333	0.470	0.380
k-means c	0.191	0.294	0.270	0.333	0.455	0.308

Table 4.6: Balanced accuracy scores for the k-nearest neighbor (k-NN) and the k-means algorithms on the latent variable h and the cell state c. The values depicted are from the 5 validation sets from the 5 CV-folds individually and the average of these 5 sets.

on the variable h can be viewed in the confusion matrix in Table 4.2.2. Of the 597 training sequences only 10 of them were wrongly classified. On the test set the classifier predicted 101 of the 139 correct. 30 of the bird sequences were wrongly classified as fish. These wrong predictions lead to the relatively low balanced accuracy of 0.672. The classifications on the cell state c are shown in Table 4.2.2. Again the classification of the training set has a high balanced accuracy of 0.981 with 11 wrong classifications. However on the test set the balanced accuracy is higher with 0.864 and 126 correct predictions.

Figure 4.3 shows a visualization of the cell state c from Equation 3.16 for the final model on both the training and test set. Both UMAP [18] and t-SNE [31] are represented in the graphs. Every point in the scatter plot corresponds to a sequence of the dataset. There is a clear separation visible between the kingdoms in all four plots. This clear separation could explain the high balanced accuracy for the k-nearest neighbor classifier. Figure 4.4 shows the visualization of the latent variable h of the final model for both the training data and the test data. The points are not as well separated as for the cell state plots. This might be an indication as to why the classifier on the test set had a lower balanced accuracy.

		Predictions		
		M	F	B
Actual Classes	M	304	0	3
	F	0	70	0
	B	6	1	213

Training set

		Predictions		
		M	F	B
Actual Classes	M	73	3	1
	F	0	18	4
	B	0	30	10

Test set

Table 4.7: Confusion matrix for the k-nearest neighbor classifier on the latent variable h.

		Predictions		
		M	F	B
Actual Classes	M	305	0	2
	F	0	69	1
	B	5	3	212

Training set

		Predictions		
		M	F	B
Actual Classes	M	73	4	0
	F	0	18	4
	B	7	0	33

Test set

Table 4.8: Confusion matrix for the k-nearest neighbor classifier on the cell state c for the final model.

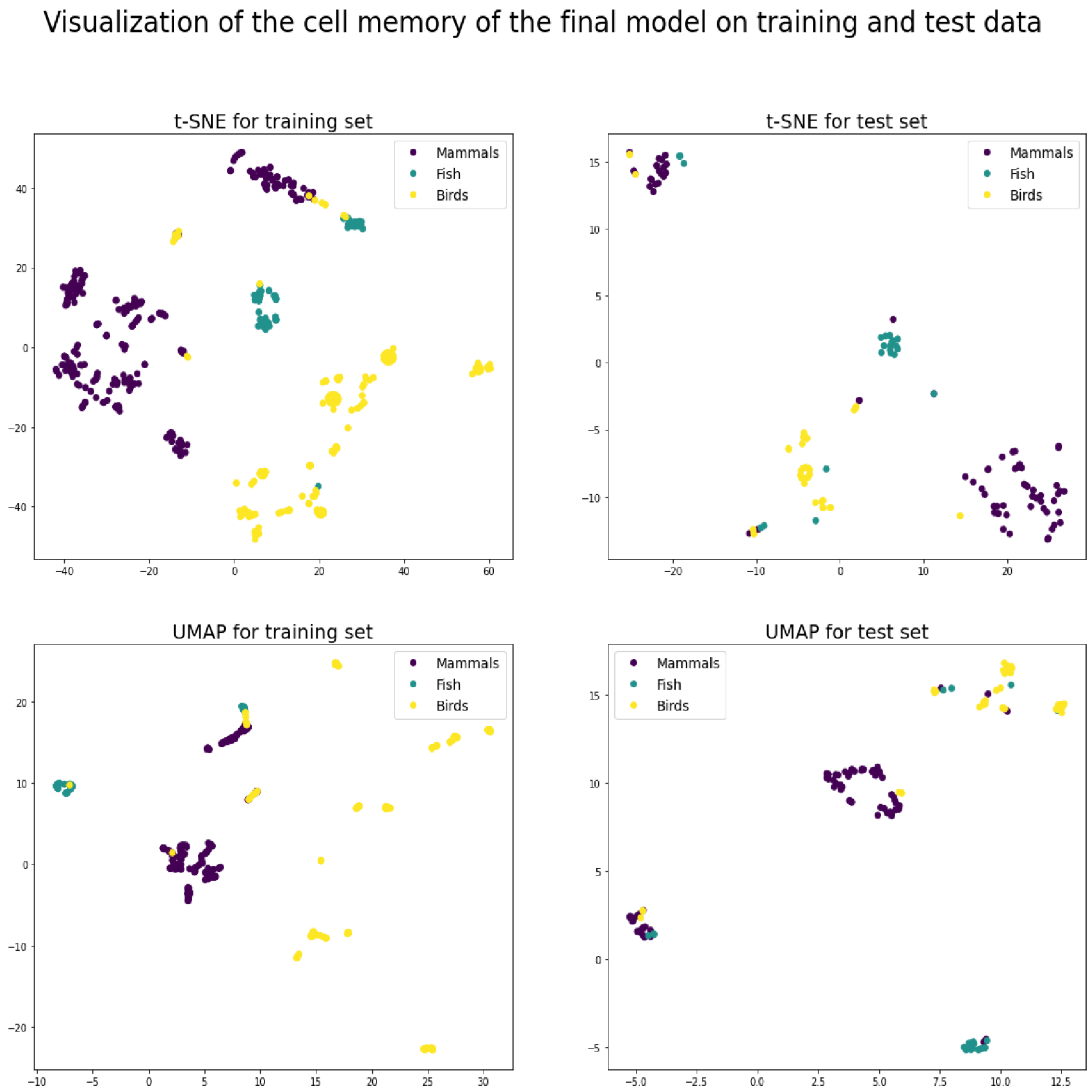


Figure 4.3: t-SNE and UMAP visualizations for the cell state c of the final model. Visualized are the cell states of both the training and test set.

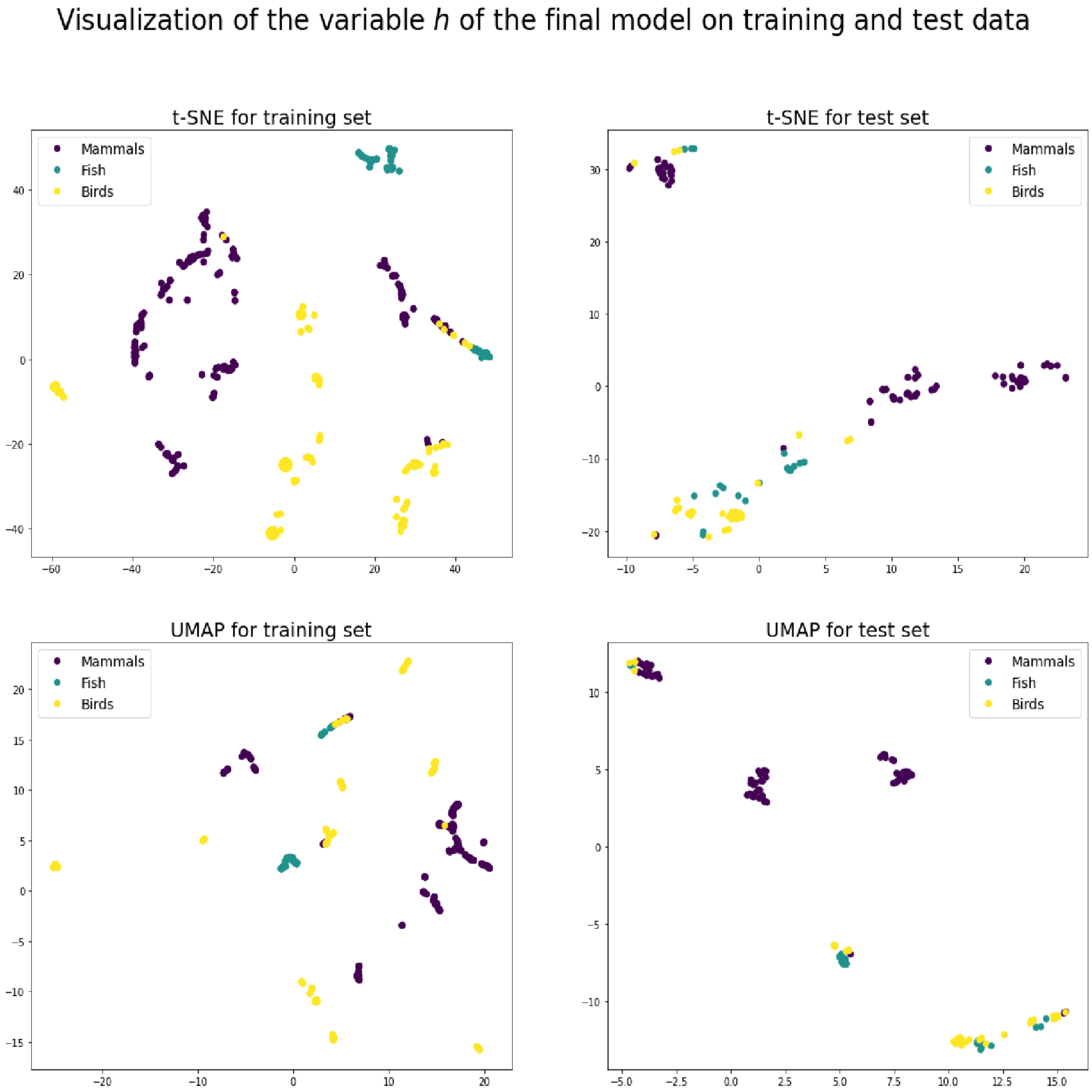


Figure 4.4: t-SNE and UMAP visualizations for the latent space variable h of the final model. Visualized are the latent spaces of both the training and test set.

Chapter 5

Conclusion

5.1 Further Possible Work

The first part of this elaboration is the data preprocessing and the training procedure for the sequence to sequence auto encoder. A variety of approaches are possible to extend this part in further research. For the data preprocessing an encoding technique other than one-hot encoding could be beneficial. One-hot encoding assumes an equal distance between the classes in the sequence. This does not take into consideration the nature of amino acids, as they are not equidistant to each other. The use of a trained embedding layer could increase the overall performance of the model. This could potentially also decrease the dimension of the latent code because the embedding layer can decrease the dimensionality of the input. Lower dimensional representations need less space for storage, which could be interesting for bigger datasets. Another potentially beneficial approach for encoding might be the use of distance matrices like PAM or $BLOSUM$, as they take into account the natural distances between the amino acids. A comparison between the one-hot encoding used in this elaboration, an embedding layer and a distance matrix could prove to be fruitful for the performance of the model.

The number of sequences for the task in this thesis is 736, which is small for a machine learning task. This problem is partially tackled by the usage of the clustered CV approach. Adding more sequences of all three kingdoms would most certainly improve the results. Especially adding sequences of the least prominent kingdom, the fish kingdom with 93 sequences, could increase the overall performance. Also the balanced accuracy for the clustering of the latent space might improve with a more balanced dataset.

Multiple approaches are thinkable to potentially improve the training procedure used in this elaboration. For example increasing the number and the range of hyperparameters in the grid search described in Chapter 4.1.2 would be one point to start. A potential addition to this search could be different learning rate schedulers and optimizers.

An interesting finding during the hyperparameter search is that the best performing teacher forcing probability is 0%, so to not use teacher forcing at all. One hypothesis why that is the case is the output of the decoder, that is again used for prediction, is not one-hot encoded. For teacher forcing the original sequence is used as input and in this case, the used token is one-hot encoded. This difference between the input appears to have a disruptive effect on the training of the decoder. One possible way to overcome this is to convert the output of the decoder into one-hot vectors by e.g. setting the maximum value of the output to 1 and the remaining positions to 0. When using an embedding layer or a distance matrix, like described above, this would need to be taken into considerations for the teacher forcing as well. Most likely the output of the decoder would need to first be converted into a single prediction and then converted by the embedding layer or the distance matrix.

The second part is the clustering of the latent space. During the clustered CV the k-nearest neighbor classifier have a high balanced accuracy on both the validation and the training sets with over 95%. For the final model the classifier of the latent space had a worse balanced accuracy on the test set with 67% and 86% on the latent variable h and the cell state c respectively. The visualization in Figure 4.4 shows that the points in the training set seem to be quite similar as there is no clear separation between them. Different classification algorithms might improve this accuracy.

5.2 Conclusion

An approach of using a sequence to sequence autoencoder on amino acid sequences and clustering the latent representation of the AE was described with this elaboration. The thesis contains not only the machine learning model but also the data preprocessing necessary to handle the available hemoglobin sequences and the clustering of the latent space. The reconstruction of the sequences through the autoencoder was presented and evaluated using the F1-score, achieving a decent performance. The classification of the latent space was rather successful with a balanced accuracy of more than 85%. This demonstration could lead to the autoencoder being used on a wider field of applications. For example predicting the functionality or other characteristics of unknown proteins by clustering them in the latent code could be a potential usage of the technique. In conclusion, the approach proved to be fruitful and could open the door for more machine learning applications in the field of biological research and bioinformatics.

Bibliography

[1] Ncbi blast website. `https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=BlastHelp`. Accessed: 2022-03-07.

[2] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[3] IUPAC-IUB Comm. A one-letter notation for amino acid sequences. tentative rules. *Biochemistry*, 7(8):2703–2705, 1968.

[4] M Dayhoff, R Schwartz, and B Orcutt. 22 a model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5:345–352, 1978.

[5] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[6] Hodges J.L. Fix E. Discriminatory analysis, nonparametric discrimination: Consistency properties. *Technical Report 4*, 1951.

[7] F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2. IEEE Press, 1999.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] Steven Henikoff and Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

[10] Geoffrey Hinton and Sam T Roweis. Stochastic neighbor embedding. In *NIPS*, volume 15, pages 833–840. Citeseer, 2002.

[11] Sepp Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, Technische Universität München, 1991.

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

[13] Michael I Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Artificial neural networks: concept learning*, pages 112–127. IEEE Press, 1990.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Saïd Ladjal, Alasdair Newson, and Chi-Hieu Pham. A pca-like autoencoder. *CoRR*, abs/1904.01277, 2019.

[16] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[17] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[18] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Großberger. Umap: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29):861, 2018.

[19] Michael Mozer. A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, 3, 1995.

[20] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[21] Frank Nielsen. Hierarchical clustering. In *Introduction to HPC with MPI for Data Science*, pages 221–239. Springer, 2016.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering Cambridge, MA, 1987.

[25] Bernhard Franz Schäfl. An lstm-based approach for coiled-coil domain prediction. Master's thesis, Universität Linz, 2018.

[26] JÃ¼rgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[27] Sam Sinai, Eric Kelsic, George M Church, and Martin A Nowak. Variational autoencoding of protein sequences. *arXiv preprint arXiv:1712.03346*, 2017.

[28] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[29] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[30] Greg Tucker-Kellogg. Optimal pairwise sequence alignment, dynamic programming matrix visualisation, 2016. [Online, accessed 07-July4-2021].

[31] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[32] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[33] Paul J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. In *Neural Networks*, pages 339–356. Elsevier, 1988.

[34] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

List of Figures

List of Tables