Hyper-Parameter Optimization for Remote Homology Detection with Protein Language Models

Master's Thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
Artificial Intelligence Group

submitted by:

Laura Maria Engist
Matriculation No.: 22-067-730
Email: l.engist@unibas.ch

supervised by:

Prof. Dr. Malte Helmert[1]
Dr. Janani Durairaj[2]
Dr. Florian Pommerening[1]

Submission date, place:

20.08.2025, DE-Vogtsburg

[1] Department of Mathematics and Computer Science, University of Basel
[2] Biozentrum, University of Basel

Acknowledgments

First, I would like to express my most profound appreciation to Prof. Dr. Malte Helmert and Prof. Dr. Torsten Schwede for allowing me to conduct this Master's thesis. Special thanks to Dr. Florian Pommerening, Dr. Janani Durairaj, Dr. Lorenzo Pantolini, and Dr. Gabriel Studer for their support and guidance throughout this project. I am grateful to the sciCORE center for scientific computing at the University of Basel for providing computational resources and system administration support. I would also like to thank Emmanuel Xavier, Thorsten Faber, and Miroslava Macejková for proofreading this thesis. Lastly, I would like to thank all members of the research group of Prof. Torsten Schwede at the Biozentrum of the University of Basel for the great time and pleasant atmosphere, which provided many valuable insights into bioinformatics research.

Abstract

Proteins are fundamental building blocks of life, composed of sequences of amino acids. Remote homology detection seeks to identify distant evolutionary relationships between proteins. Protein Language Models, adapted from natural language processing, have become increasingly prominent in tasks such as remote homology detection. Deep learning methods that use Protein Language Model representations as input often rely on parameters that remain unexplored. Systematic parameter tuning helps to search this space and identify configurations that improve performance.

This thesis addresses the gap of unexplored parameter spaces by applying parameter optimization to a neural network discretization method incorporating Protein Language Model representations for remote homology detection. Our objective is to discover parameter configurations that enhance detection quality.

We present a targeted Bayesian optimization method that systematically explores the parameter spaces of said neural networks. With repeated parallelized execution, we achieve an improvement of up to 7% in detection performance compared to configurations selected without exploration. These findings highlight the importance of thorough parameter selection and provide practical insights for improving protein analysis using Protein Language Models.

Contents

Chapter 1

Introduction

Proteins are made up of chains of building blocks, called amino acids, arranged in a specific sequence. These sequences carry important information about the structure and function of a protein. A key topic in computational biology is the search for sequences that are similar to another sequence to uncover evolutionary relationships, a so-called protein sequence homology search [1]. The main goal is to transfer biological knowledge, such as protein annotations and functions, based on sequence similarity. Since similar sequences often imply similar structure and function, these searches help to identify proteins with shared characteristics [2]. The core of this process is the alignment of sequences, a method that matches parts of protein sequences, called residues, to highlight similarities. A sequence alignment can contain gaps that help to align homologous regions of sequences correctly. The costs of inserting or extending a gap in such an alignment are called gap penalties.

Recently, Protein Language Models were introduced to translate protein sequences into rich, detailed continuous representations called embeddings [3]. These embeddings describe each part of a protein in the context of the surrounding sequence, helping to detect evolutionary relationships that traditional sequence alignment might miss [3]. However, well-known algorithms for detecting similarities cannot be applied to these continuous representations because they require a discrete input [4, 5, 6, 7, 8]. Thus, current research is looking for ways to discretize their complex representations [9]. Pantolini et al. [9] developed an approach using neural networks to create a discrete set of symbols, a new alphabet, that captures key information such as context in a sequence. With this approach, they were able to translate the continuous representations into discrete sequences and apply algorithms to align them effectively. The hyper-parameters of an

applied neural network can be varied. In previous work, these parameters were set to the best of current knowledge, but were not optimized.

To optimize the parameters for a specific approach, so-called parameter tuners can be used. Parameter tuners find a set of parameters to get the best result for a defined problem. They have improved the performance of various applications, such as the SAT Solvers [10] and the Fast Downward planning system [11]. These tools explore a predefined configuration space of parameter values by testing different settings and using performance feedback to guide the search. A particular set of parameter values is called a configuration. While occasionally sampling random configurations, the tuners adjust their search direction based on the Bayesian optimization idea to balance exploration and exploitation, a probabilistic approach for efficiently exploring and sampling a hyper-parameter space [12].

In this thesis, we present a method for tuning the hyper-parameters of the procedure developed by Pantolini et al. [9]. We are using Bayesian optimization and random forests, a learning method that builds decision trees using random subsets of data and features, and combines their predictions [13]. Specifically, we employ the SMAC hyper-parameter optimization algorithm [14] to guide this process. Two central questions drive our work.

1. Are sequence alignments significantly affected by the choice of gap penalties? Is it necessary to tune them?

2. What parameter values work well for configuring the procedure to compute discrete embedded sequences?

We developed optimization pipelines to address these questions. For each pipeline, we define a performance metric that serves as the objective of SMAC to optimize. With this metric, a structured and quantifiable evaluation of different parameter settings is possible. The primary goal of our approach is to identify configurations that improve the detection of distant protein relationships compared to the manually chosen parameter values in the previous work. In addition to optimizing the performance, our method also provides a systematic analysis of how individual hyper-parameters affect results. These insights can be used to refine and improve the existing procedure.

Compared to the parameters used in the previous work, our approach finds parameters that can detect distant protein relationships with an improvement in accuracy of up to 7%.

The remainder of this thesis is organized as follows. Chapter 2 introduces the necessary biological background, provides an overview of related research, and reviews the existing relevant methods. In the course of this work, we developed two new methods to support

our implementation. Chapter 3 describes a method for mapping existing alignments to an alternative alphabet and the construction of a custom substitution matrix based on these mapped alignments. To evaluate results and support parameter tuning, Chapter 4 introduces two scoring metrics that quantify alignment and identification quality.

All methods, both existing and newly developed, are integrated into processing pipelines, which are detailed in Chapter 5. Chapter 6 presents the experimental setup and results, including the configuration of the optimization process and the parameter tuning experiments. The thesis concludes with Chapter 7, which summarizes the main findings and their implications, followed by Chapter 8, which outlines the potential directions for future work.

Chapter 2

Background

In this chapter, we provide an overview of the background knowledge that is necessary to follow the process of this thesis. First, we explain important terminology in Chapter 2.1. Afterwards, in Chapter 2.2, we review previous work related to this thesis. In Chapter 2.3, we explain the existing methods that we use.

2.1 Biological Terminology

In this part of the chapter, we describe biological terminology and introduce notations that will be used throughout the thesis.

Proteins

A *protein* is a polypeptide structure. It consists of one or more chains of connected amino acids that fold into a three-dimensional structure. Amino acids are the building blocks of proteins, determining their *sequence*, with a specific character representing each amino acid out of the 20 proteinogenic amino acids [15, 16, 17].

A *residue* is an amino acid, or character, that is connected to a neighboring amino acid. *Mutation* of a protein denotes the alteration of its sequence, spawning evolution [16]. Proteins derived from their ancestors by mutation are related. This origin can be inferred by homology detection. A *protein family* consists of closely related proteins with clear evidence of their common evolutionary origin due to a high structural and sequence similarity. The *protein superfamily* groups more distantly related proteins with high structural, but potentially low sequence similarity. The *protein fold* groups structurally highly similar proteins that may not be evolutionarily related at all [18].

Homology

Protein homology is the similarity between two (or more) protein sequences of a common evolutionary origin. A *homologous region* is therefore a conserved part of the sequences that are being compared.

Consequently, when we talk about the *similarity* of two proteins, we talk about similar parts in their sequences. The term *sensitivity* represents the probability of a correct diagnosis of positive cases [19]. In protein homology detection, the *sensitivity* is the probability of detecting the actual remote homologues. Generally, homology detection tries to maximize sensitivity to increase the chances of finding very remote homologues.

Alignment

The *alignment* of protein sequences establishes a character-by-character relationship between them. We distinguish between *sequence-based* and *structure-based* alignment approach. A *sequence-based* alignment solely employs sequence information, while *structure-based* alignment approaches take contextual and more detailed information by operating on the 3D shape of the protein. A *sequence-based* alignment is created using a so-called *substitution matrix*, such as one of the BLOSUM family matrices [20], assigning scores for observing a specific pair of aligned characters. The derivation of an actual alignment of sequences using such a matrix becomes an optimization problem.

Pairwise Sequence Alignment

When aligning two protein sequences, we talk about a *pairwise sequence alignment*. We introduce a notation to refer to the process of pairwise sequence alignment throughout this thesis.

Let A be the alphabet. A protein sequence is a finite string over this alphabet: $S = s_1 s_2 ... s_n \in A^n$ where each $s_i \in A$, and the length of the sequence is denoted as $n = |S|$. We extend this alphabet A to include a gap character $A' = A \cup \{-\}$. S' is the aligned sequence of S over this extended alphabet A', of length $m \geq n$, such that some positions may be filled with gaps represented as "$-$": $S' = s'_1 s'_2 ... s'_m \in (A^{'})^m$. Removing all gaps from S' yields the original sequence S consisting of letters only.

Let $S \in A^n$ and $R \in A^n$ be two protein sequences and let $S' \in (A^{'})^m$ and $R' \in (A^{'})^m$ be their aligned sequences respectively. We denote their pairwise alignment as (S', R'). The character $x' \in S'$ is aligned with $y' \in R'$, if there exists an i such that $s'_i = x$ and $r'_i = y$.

A pairwise sequence alignment containing gaps is referred to as *gapped alignment*. On the contrary, a pairwise sequence alignment without gaps is considered an *ungapped alignment*.

We distinguish between *local* and *global* alignments. A *local alignment* finds and aligns the most similar parts of two sequences. An approach that computes a *local alignment* returns the aligned local parts, together with the indices from the original sequences at which the local alignment starts as output. However, a *global alignment* aligns entire sequences.

An *optimal pairwise alignment* is defined as the alignment of two sequences with the highest alignment score. This score can be computed using a substitution matrix, where the scores for each aligned pair of characters are summed up.

Protein Language Model

Protein Language Models are similar in principle to Natural Language Models, with the difference that they are trained on protein sequences of amino acids instead of text. They are trained to represent the semantic meaning of protein sequences of continuous representations, called embeddings.

The input, a large set of sequences, is broken down into manageable units. This process is called tokenization. In Natural Language Models, either each word, each character, or subwords can be such tokens. In Protein Language Models, amino acid tokenization is used, which means that each amino acid (each character) is a token.

Natural Language Models are trained by predicting the next token, the next character in a text. Protein Language Models are trained similarly, but to predict an arbitrary amino acid in a sequence. The goal is to train the Protein Language Model such that it can represent the functional meaning of the input protein space [21].

2.2 Related Work

In this chapter, we review methods and studies relevant to our work. First, we discuss the application of parameter tuning in general. Second, we take a look at homology detection. Third, we narrow our perspective to Protein Language Model methods for homology detection. Finally, we summarize all methods.

2.2.1 Hyper-Parameter Tuning

Parameter tuning is used in various use cases. Two studies where the optimization tool SMAC3 [14] enabled improvements in parameter settings and runtime are its use for SAT Solvers [10] and the Fast Downward planning system [11].

Hyper-Parameter Tuning to configure SAT Solvers

The application of the model-based Bayesian Optimization tool SMAC [22] achieved the most considerable speedups in a challenge to configure SAT Solvers conducted by Hutter et al. [23]. The configuration phase consisted of multiple SMAC runs that ran independently. This way, they could parallelize the runs on compute clusters. A toolbox to provide insights into the configuration process of SAT solvers enabled Falkner et al. [10] to find a configuration that lowers the average runtime while reducing the number of timeouts.

Hyper-Parameter Tuning to configure Fast Downward

Given a planning domain, Seipp et al. [11] used SMAC to define a single configuration for Fast Downward, a classical planner based on heuristic search [24]. SMAC was run independently five times in parallel. The configuration chosen for the International Planning Competition was the one that performed best out of these five, defined by a test set.

2.2.2 Homology Detection

Homology detection is the identification of evolutionary relationships between proteins with protein sequence or structural similarity. The detection of homologous regions enables us to search against databases and to find other related protein sequences [9].

Alignments to Measure Similarity

One classical way to measure similarity between proteins is by comparing amino acid sequences based on their alignment. Two approaches measuring similarity by aligning pairs

are BLAST [5] and Smith-Waterman [25, 26].

BLAST Since the early 1990's BLAST has been the driving force of homology searches [4, 5, 27]. BLAST, Basic Local Alignment Search Tool, is used for rapid sequence comparison to search protein sequence databases and analyze regions of similarity in DNA sequences, among others [5]. The method uses a similarity score matrix to align sequences. The pair of residues from two sequences with identical lengths and the highest score is defined as the maximal segment pair.

Altschul et al. [5] introduced two versions of BLAST to search for such maximal segment pairs. One version uses dynamic programming and the other tables of k-mers. The first version allows gaps in the resulting alignments, which leads to a slowdown of the extension process. Furthermore, in some test cases, they observed a trade-off between the sensitivity of amino acid searches and selectivity. The second version requires random access to the database tables, which, as a consequence, slows down the whole approach [5].

MMseqs2 To resolve the speed-sensitivity trade-off, Steinegger et al. developed MMseqs2, another search tool [8]. Using a pre-filter with k-mer counting, the approach limits the amount of computationally expensive alignments [3]. This filtering is the reason why, compared to numerous established methods like BLAST, MMseqs2 achieves better sensitivity and speed. This leads to it closing the gap between cost and performance and facilitates the analysis of extensive data sets [8].

Smith-Waterman Another factor that influences the speed of MMseqs2 is the alignment algorithm used, for example, Smith-Waterman. The Smith-Waterman algorithm finds the pair of segments of two sequences with the most significant similarity. The lengths, deletions, and insertions can be arbitrary [25]. This approach's problem concerning creating alignments is the speed and completeness trade-off [28]. Either it creates global alignments which align the entire sequences but are time inefficient, or it creates local alignments fast but with missing alignments at the ends of the sequences [28].

Hidden Markov Models to Measure Homology

Another way to measure homology is by using hidden Markov Model (HMM) techniques, such as HMMER and HHblits.

HMMER HMMER is a software package using probabilistic inference methods for protein sequence similarity searches [27, 29]. HMMER compares a sequence to a statistical model. This model, for example, describes a family of protein sequences. The result is a score of the probability of the sequence being related to the model, the family in this example. Distantly related proteins are only related to a small extent. Compared to methods using standard substitution matrices, probabilistic methods are more likely to identify such distant relationships between sequences [27].

HHblits The HHblits tool presented by Remmert et al. does iterative protein sequence searching [6]. In contrast to HMMER, the query sequence is also represented by a profile hidden Markov Model. Profile hidden Markov Models represent multiple sequence alignments specifying the probability of observing each of the amino acids in evolutionarily related proteins for each position in a sequence. Profile-profile and HMM-HMM alignment sequence-search methods are considered very sensitive [6]. Thus, it has a higher sensitivity and more accurate alignments than BLAST.

2.2.3 Homology Detection using Language Models

Although some methods manage to be fast while remaining sensitive, those methods struggle when sets of proteins have sequences with few identical residues, also called low sequence identity [30]. Now we will dive into methods that use language models for homology detection. More precisely, Protein Language Models manage to detect low identity. Their ability to regard more information is due to high-dimensional contextual embeddings and the extraction of evolutionary information from large databases of sequences [30, 31].

pLM-BLAST Inspired by BLAST [5], Kaminski et al. developed a tool detecting distant homology [32]. The tool compares single-sequence representations from a Protein Language Model with each other, the embeddings. They have shown that their approach achieves a similar level of accuracy to hidden Markov Model searches while being faster. Furthermore, pLM-BLAST was able to uncover the local evolutionary relation of highly divergent proteins that globally do not have similarities [32]. Because of its speed, pLM-BLAST is an efficient tool for searching databases. However, it is limited to smaller databases, as searching large ones with millions of sequences would be computationally costly [32].

EBA Embedding-based alignment by Pantolini et al. [3], that will be described in detail in Chapter 2.3, shows excellent accuracy compared to classical methods and other approaches based on Protein Language Models such as pLM-BLAST [3]. However, the computation times of this approach are higher than those of tools like MMseqs2, which becomes limiting when conducting large-scale analyses [3].

2.2.4 Summary of Related Approaches

Hyper-parameter tuning has been successfully applied to tools, including but not limited to SAT Solvers and Fast Downward. It was shown that the optimization can reduce the runtime.

Homology detection plays a crucial role in bioinformatics, and various approaches have been developed for this purpose. On one hand, there are alignment approaches like the search tools BLAST and MMseqs2, and the alignment algorithm Smith-Waterman, using sequence-based comparisons. On the other hand, there are probabilistic models such as hidden Markov Models. Here, not pairs of sequences are compared, but probabilistic models representing protein families [27] to increase sensitivity in detecting distant homologues. Everything started with BLAST, and then MMseqs2 appeared as the fast approximate version of BLAST. With hidden Markov Models, the identification could be expanded to distant homologues while maintaining speed and sensitivity.

Protein Language Models provide further improvement. EBA performs best, compared to approaches such as HHblits and pLM-BLAST, regarding homology detection [32]. Considering speed, approaches using continuous embeddings, such as EBA, perform much slower than discrete sequence-based approaches like BLAST [32]. Parameter tuning is not very common when using Protein Language Models.

We have seen that parameter tuning can improve a process. Furthermore, we are combining the strengths of EBA and MMseqs2 to achieve good speed, sensitivity, and accuracy. However, the risk of overfitting keeps us cautious in applying hyper-parameter optimization.

2.3 Existing Methods

In this part of the chapter, we provide an overview of the existing methods applied throughout this thesis. The original papers are referred to for more detailed descriptions.

2.3.1 MMseqs2

MMseqs2 is a protein database search tool that can detect homologues and create pairwise sequence alignments for a given protein sequence, also known as the query sequence. For a given query sequence, MMseqs2 searches a database of target protein sequences in three stages of increasing sensitivity, progressively filtering unlikely matches to minimize computation time. The output of one stage is the input of the next. The final output consists of local alignments of the query sequence with every target in the database, if an alignment can be found. The resulting alignments are local because the primary goal for MMseqs2 is to detect regions of similarity between sequences efficiently. When searching large sequence databases, MMseqs2 is not interested in aligning entire sequences, but the most relevant local matches. This type of alignment improves the scalability and efficiency of the method.

First, the database is filtered for k-mer matches, similar, not necessarily exact, parts of a target sequence to the query sequence with k connected characters [8]. The similarity is determined by a similarity score [8]. For that, MMseqs2 uses per default one of the well-established evidence-based standard matrices for protein alignments, the BLOSUM62 matrix [20]. A BLOSUM62 similarity score of a k-mer in the query sequence to a k-mer in a target sequence is the sum of the scores present in the BLOSUM62 matrix for the paired characters in the respective k-mer parts.

Second, the set of sequences resulting from the previous step is filtered for double k-mer matches, two consecutive matches of k-mers between the query sequence and a target [8]. The set of targets that includes double k-mer matches is used to create so-called ungapped alignments with the query sequence. An ungapped alignment results in aligned parts of sequences that can be aligned without inserting a gap. For these alignments, the scores are calculated.

In the third filter stage, the input target database consists of the target sequences resulting from the previous step that achieved a score higher than a specified threshold. Then, for each of the resulting target sequences, gapped alignments with the query sequence are created. These gapped alignments do contain gaps. Nevertheless, the

resulting set of alignments returned from MMseqs2 can contain both gapped and ungapped alignments.

Using these three consecutive filters, MMseqs2 facilitates the analysis of extensive protein sequence data sets [8].

We are using MMseqs2 to compute the alignments used in our pipelines, explained in detail in Chapter 4 and Chapter 5. The input for MMseqs2 is a set of protein sequences and a substitution matrix that shall be used. For each query sequence and each target sequence aligned with it, the output contains an alignment quality measure and the additive score for the alignment computed with the substitution matrix. This quality measure is the so-called *E-value*. In addition, MMseqs2 returns an identification of the family, superfamily, and fold of the target sequence.

2.3.2 k-Means Clustering

The k-Means clustering algorithm clusters data with similar characteristics by exploiting the structure of their distribution [33, 34]. The number k defines the number of clusters.

The input consists of data points in $\mathbb{R}^n$ for some n and the number of clusters k. The initialization strategy is defined by the parameter *init*. The algorithm first chooses k starting centers randomly (*init*='random') or to be (generally) distant from each other (*init*='k-means++') [35]. Using the Euclidean distance metric, each data point is then assigned to its nearest center, creating clusters. The centers are then updated to the average of all data points assigned to this center. This assignment and update are repeated until the centers do not change anymore [33]. The output of the algorithm is the assignment to the center of the cluster for each data point.

Outliers are vectors with significantly larger variance in their entries, compared to other vectors. To remove the impact of such outliers, a normalization step can be added. This normalization step divides each entry of such a vector by the average of all its entries and can be controlled via the boolean parameter *normalize*.

The way we use this algorithm is illustrated in the following section 2.3.3.

2.3.3 Embeddings from a Protein Language Model

High-dimensional embedding representations from Protein Language Models have been shown to capture rich information about protein similarity, including structural similarity, despite using only sequences as input [3]. This richness of information is achieved by computing high-dimensional embedding vectors to represent the semantic meaning of each character in the context of the whole protein sequence.

The embedding layer of a Protein Language Model conducts a 1-to-1 translation from a token from the input space to one in the latent space, which captures the semantic meaning of tokens in context [21]. This captured meaning is the semantic meaning we are interested in. The difficulty with this is that these vectors are continuous and not direct translations from one character to another. That means we now have sequences that represent the functional content, but they are continuous vectors and not a fixed alphabet. As a consequence, this continuous representation needs to be discretized in order to be used in established sequence search algorithms such as MMseqs2.

We are using three discretization approaches. The first and simplest is the k-Means clustering algorithm, briefly described in section 2.3.2. This algorithm uses embedding vectors as input and treats the centers of the computed clusters as discrete characters in a so-called codebook [36, 37]. The drawback of this approach is that there is no specific way to cluster for the remote homology task.

The second approach is *VQ-VAE*, established in Deep Learning research [36]. *VQ-VAE* requires training and tuning, and will be explained in more detail in the next section. In our case, this approach is expected to obtain discretizations that are better suited to the purpose than k-Means.

The third approach is to use the language modeling head *LM-head* of a Protein Language Model. The *LM-head* is the final part of a Protein Language Model that converts the embedding vectors into a vector of probabilities at the amino acid level with the number of amino acids as its dimension [38]. It is already trained for discretization, but needs to be fine-tuned to detect remote homology. The fine-tuning is outlined in the next section. This approach is also expected to be better than k-Means.

In this thesis, we use the embedding layer and the language modeling head of the pre-trained Protein Language Model *ESM-1* by Lin et al. [31].

2.3.4 Training and Usage of a Neural Network

A neural network is a machine learning algorithm inspired by the human brain. The network consists of knots, so-called neurons. Each neuron receives input, processes it, and forwards the output. The connections between these neurons are weighted. Those weights are adapted during the training of the neural network, resulting from the application of the so-called loss functions. Loss functions quantify the difference between the predictions of a model and the actual target values. Network training works by adjusting its parameters to minimize loss functions. This approach is called gradient descent, an optimization algorithm commonly used to train neural networks.

There are neural networks that we can use to generate meaningful representations for embeddings from Protein Language Models at the level of characters (see Chapter 2.3.3) [3]. Pantolini et al. [9] designed a neural network to pool embeddings from a Protein Language Model more effectively than clustering methods like k-Means clustering. We are going to use this network, and one using a *LM-head* for discretizations [38], in our neural network pipelines.

To understand the usage of neural networks, we first look at their architecture and training mechanism.

The Architecture of the VQ-VAE Neural Network

Figure 2.1 illustrates the architecture of the first neural network, which is called a *VQ-VAE Neural Network* [36]. The architecture consists of an encoder (E), a vector quantization module (VQ), and a decoder (D). In the following, we work through the process step by step.

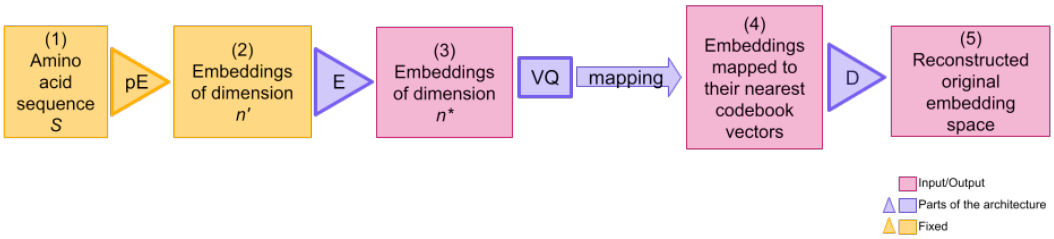


Figure 2.1: *Schema of the architecture of the VQ-VAE Neural Network. The yellow parts are fixed and identical for each training. The red boxes represent the input and output of respective steps in the process given in purple shapes. The numbers in the red boxes enumerate the input and output for referring.*

A fixed Protein Language Model encoder (pE) provides the input, high-dimensional representation of one sequence of amino acids, for this network.

The encoder (E) converts the high-dimensional embeddings from the pE to lower-dimensional contextual embeddings. These contextual embeddings carry characteristics that are important for sequence alignment. The reduction in the dimensions also improves the overall performance of the model.

Next, vector quantization (VQ) finds optimal positions for the codebook vectors of these embeddings with reduced dimensions, each representing one letter in the alphabet.

The contextual embeddings are then mapped to their nearest codebook vectors using the Euclidean distance metric.

This is followed by a decoder (D) taking the quantized representations as input and returning the reconstructed original embedding space, which should essentially be the space in (2) after the Protein Language Model encoder (pE) was used.

The Training of the VQ-VAE Neural Network

To train the network, we define a training set of protein sequences combining two protein sequence databases: SCOPe [39] and Pfam [40]. All parameters, including those we are going to optimize, can be defined in a configuration file with all the variables of the neural network.

Training is divided into so-called epochs. One epoch corresponds to one pass through this training set. Training data is grouped into batches. One batch corresponds to one pair of sequences from the training set, because in the training, we process pairs of sequences.

We are now having a look at the loss functions we want to minimize by training the network, and thus, we explain how the training works. The affected parts of the architecture are mentioned with the number in brackets, visualized in Figure 2.1. The formulas are applied per batch, for each pair of sequences.

For each of the two sequences in a batch, the *Contrastive Loss* comes into play. The *Contrastive Loss* (l_{cn}) penalizes if similar objects, in our case aligned residues, are not positioned close together in the learned embedding space.

The input is a pair of sequences S_i and S_j and a pair of embedded sequences S'_i and S'_j. For all aligned residues in the original sequences, (r_{ik}, r_{jk}) with $r_{ik} \in S_i$ and $r_{jk} \in S_j$, we want to minimize the *Contrastive Loss* function taking another non-aligned residue r_{jl} as third parameter. This residue r_{jl} is in a window of one to five residues before and after the aligning position with $(k-5) \leq l \leq (k+5)$. The loss function is therefore defined as:

$$l_{cn} = l(a, p, n) = l(r_{ik}, r_{jk}, r_{jl}) = d(r_{ik}, r_{jl}) - d(r_{ik}, r_{jk}) + margin \tag{2.3.1}$$

with the Euclidean distance metric d, defining the shortest distance between two points, a as the anchor, p as a positive example of an aligned residue, and n as a negative example [41]. The *margin* is a non-negative value representing the minimum difference between the positive $d(r_{ik}, r_{jk})$ and negative $d(r_{ik}, r_{jl})$ distances that is required for the loss to be 0. The parameter *margin* is also a part of the parameter set that is to be optimized.

After decreasing the dimension of the embeddings, the vector quantization mechanism (VQ) is applied to map continuous latent representations, the embedding vectors, into discrete embeddings, influenced by multiple parameters. The *Codebook Management Loss* (l_{cbm}) is used to update the dictionary and optimize the embeddings. Here, we want to minimize the function:

$$l_{cbm} = l(z_e(x), e) = \sum_{i}^{N} ||sg[z_e(x)] - e_i||_2^2 \tag{2.3.2}$$

with the input sequence x, the length of x given as N, $z_e(x)$ being the output of the encoder (3) for a sequence x, e_i the codebook vectors for each character of the embedding (4), and sg the so-called stop gradient operator [36]. This operator constrains its operand to be a non-updated constant. Minimizing this function means minimizing the distance between e_i and $z_e(x)$. To minimize this, the codebook is adapted.

The output of the vector quantization consists of discrete representations of the embeddings. Now, a decoder is used to reconstruct the original embeddings, prior to the vector quantization (3). The *Reconstruction Loss* (l_{rc}) and *Foldseek-like Loss* (l_{fs}) penalize if the reconstruction was not successful. *Reconstruction Loss* focuses on the reconstruction of a sequence, while the *Foldseek-like Loss* focuses on the reconstruction of aligned residues in the two input sequences. The formula we want to minimize for the *Reconstruction Loss* is defined as the following log-likelihood:

$$l_{rc} = l(x, z_q(x)) = \log p(x|z_q(x)) \tag{2.3.3}$$

with $z_q(x)$ being the output of the decoder(5), and x the embedding (3) [36]. The log-likelihood is used because that way the product of the probabilities is converted into the sum of logarithm of the probabilities, which simplifies the computation. Equation 2.3.3 is

minimized if there is no difference between the embedding prior to the vector quantization x (3) and the discrete decoded one $z_q(x)$ (5).

The formula we want to minimize for the *Foldseek-like Loss* is identical to the one for the *Reconstruction Loss*, with the difference that now we penalize if the decoded embedding of the residue $r_1 j$ (5) is not close to its original embedding $r_2 j$ (3):

$$l_{fs} = \sum_{j}^{M} l(z_q(r_{1j}), z_q(r_{2j})) = \sum_{j}^{M} \log\, p(z_q(r_{1j})|z_q(r_{2j})) \tag{2.3.4}$$

with M being the length of the aligned sequences from a batch [37].

The parameters defining the configuration of this network are represented as θ_{VQ-VAE}, adapting the weights applied to the loss functions explained above. The set θ_{VQ-VAE} consists of 21 hyper-parameters: θ_{VQ-VAE} ={cnE, cnQ, cnD, cb, em, en, us, dv, rcP, fs, $margin$, use_cnE, use_cnQ, use_cnD, use_cb, use_em, use_en, use_us, use_dv, use_rcP, use_fs}. The parameters cnE, cnQ, cnD, cb, em, en, us, dv, rcP, fs, and $margin$ are the weights visualized in Figure 2.2, which will be explained in more detail in the following section. Whereas the parameters use_cnE, use_cnQ, use_cnD, use_cb, use_em, use_en, use_us, use_dv, use_rcP, and use_fs decide whether the parameter corresponding by name is applied to add weight to the loss function or not. This will be useful to determine the importance of each hyper-parameter.

Each of the parameters can add weight to a corresponding loss function as a result of being multiplied with it. Which parameter influences which of the described loss functions will be enumerated in the following.

The parameters *cnE*, *cnQ*, and *cnD* add weight to the *Contrastive Loss*. Recall that the *Contrastive Loss* penalizes if the aligned residues in two input sequences (1) are not positioned close together in the embedding space. The three parameters differ in the embedding space they relate to. The parameter *cnE* adds weight to the loss function where the pair of embedded sequences is taken from the output (3). For the parameter *cnQ*, the pair is from (4), and for the parameter *cnD* from (5).

The parameters *em*, *en*, *us*, and *dv* influence the *Codebook Management Loss*. They are applied to this loss function to minimize the distance between e_i and $z_e(x)$. With the parameters *em* and *cb*, the function minimizes the difference between the embedding (3) and the codebook vector (4). As we want the codebook vectors to be heterogeneous, the parameter *dv* is added to minimize the similarity of the codebook vectors to each other. The parameter *en* computes the entropy of the distribution of the codebook vectors, illustrating the certainty that the mapping assigns an embedding vector to the correct codebook vector.

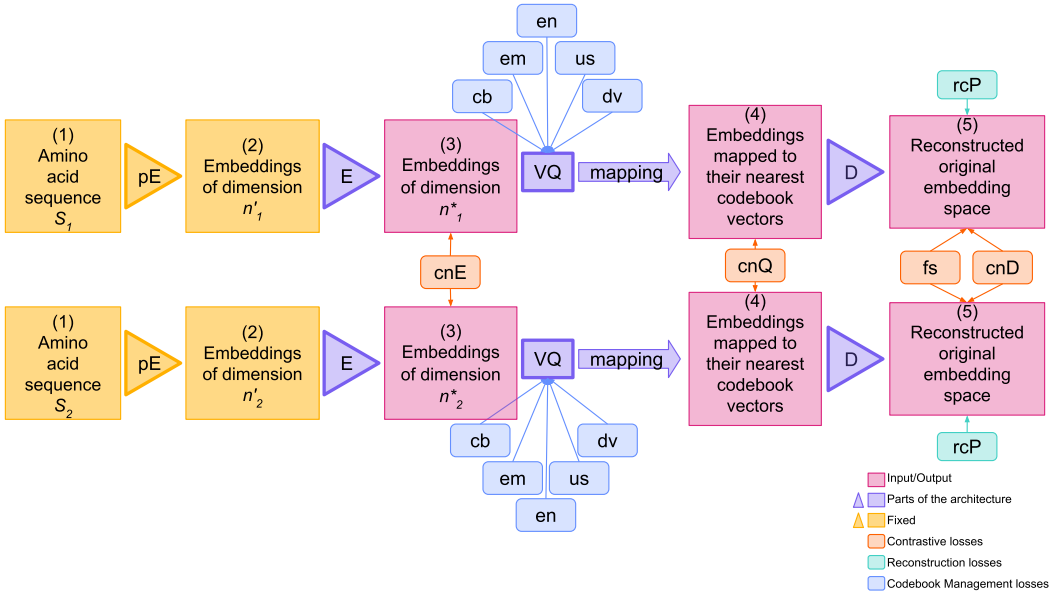


Figure 2.2: *Schema of the training of the VQ-VAE Neural Network. Again, the yellow parts are fixed and identical for each training. The red boxes represent the input and output of respective steps in the process given in purple shapes. The numbers in the red boxes enumerate the input and output for referring. Additionally, the smaller boxes represent the hyper-parameters and their sphere of action.*

With the parameter *us*, it is penalized if the distribution of used codebook vectors is far from uniform. We want all codebook vectors to be used and thus a distribution that is close to uniform.

The *Reconstruction Loss* and *Foldseek-like Loss* functions are weighted by the parameters rcP and fs.

The total loss function for the *VQ-VAE Neural Network* is the weighted sum of all loss functions weighted by the corresponding parameter given as:

$$\begin{aligned} l_{VQ-VAE_{total}} =& l_{cn_3} * cnE + l_{cn_4} * cnQ + l_{cn_5} * cnD + \\ & l_{cb} * cb + l_{cb} * em + l_{cb} * en + l_{cb} * us + l_{cb} * dv + \\ & l_{rc} * rcP + l_{fs} * fs \end{aligned}$$

The parameter *margin* has a configuration space of 0.001 to 10 because it cannot be 0. It is not a weight and can take any value greater than 0. For the scope of our experiments, we restricted it to a maximum of 10. All loss weight parameters have a configuration space of 0.0 to 1.0 because this facilitates normalization and interpretation, ensures numerical

stability, and makes training easier to tune.

The Architecture of the LM-head Neural Network

The other network we are using is a *LM-head Neural Network* [38], which significantly simplifies the quantization aspect by using the pre-trained ESM *LM-head*, which already outputs logits and probabilities corresponding to a 20 letter alphabet. Here we fine-tune this *LM-head* to exchange the amino acid alphabet to a new alphabet of length 20 better suited for remote homology detection. Instead of mapping codebook vectors to characters, we now have vectors of 20 probabilities, one probability per character of our alphabet. The mapping step chooses the character with the highest probability in such a vector.

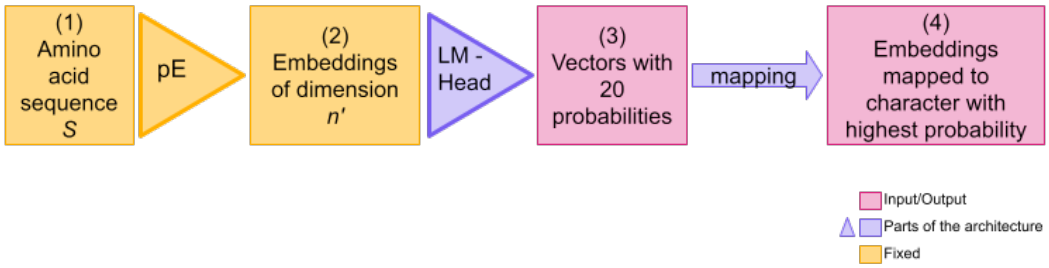


Figure 2.3: *Schema of the architecture of the LM-head Neural Network. The yellow parts are fixed and identical for each training. The red boxes represent the input and output of respective steps in the process given in purple shapes. The numbers in the red boxes enumerate the input and output for referring.*

The Training of the LM-head Neural Network

The training of the *LM-head* network is also similar to the training of the *VQ-VAE* network.

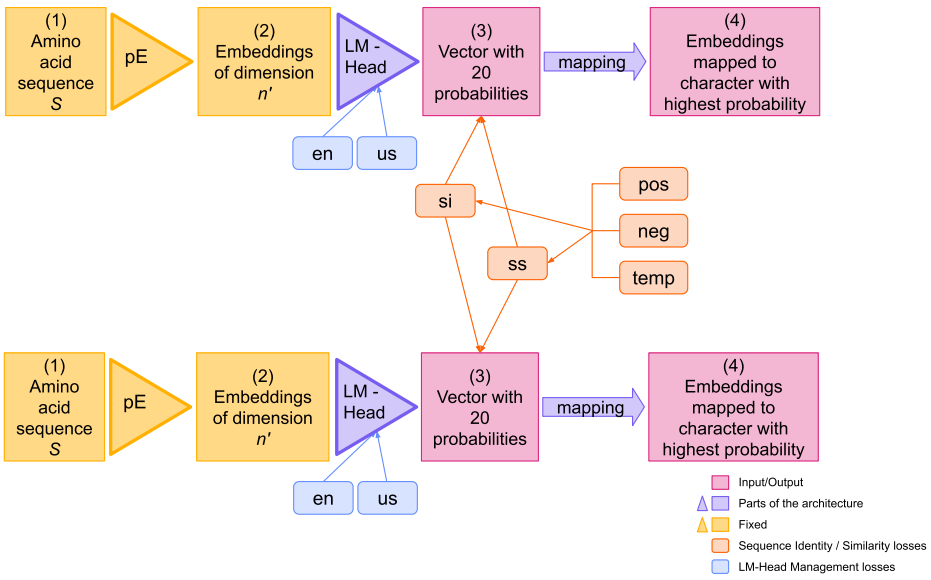


Figure 2.4: *Schema of the training of the LM-head Neural Network. Again, the yellow parts are fixed and identical for each training. The red boxes represent the input and output of respective steps in the process given in purple shapes. The numbers in the red boxes enumerate the input and output for referring. Additionally, the smaller boxes represent the hyper-parameters and their sphere of action.*

There are 14 hyper-parameters setting the weights for the losses of the network represented as $\theta_{LM-head} = \{en,\ us,\ si,\ ss,\ pos,\ neg,\ temp,\ use\_en,\ use\_us,\ use\_si,\ use\_ss,\ use\_pos,\ use\_neg,\ use\_temp\}$. The parameters en, us, si, ss, pos, neg, and $temp$ are the weights added to the loss functions. The parameters use_en, use_us, use_si, use_ss, use_pos, use_neg, use_temp decide on whether the parameter with the corresponding name is used to add weight to the models loss function or not.

The parameters en and us are similar to their namesakes in the previous model, with the difference that we are not looking at codebook vectors, but instead vectors with 20 probabilities for each character of a sequence (3). The parameter en computes the entropy of the distribution of the probability vectors, illustrating the certainty that the character with the highest probability is the correct one. With the parameter us, it is penalized if the distribution of probabilities in all vectors is far from uniform. We want all characters to be used and thus a distribution that is close to uniform. The parameters si and ss are similar to the ones for the *Contrastive Loss* in the *VQ-VAE* network, but with a difference.

Recall that for each of the two sequences in a batch, the *Contrastive Loss*, which penalizes if similar objects are not positioned close together, comes into play. The input is now a pair of sequences S_1 and S_2 and the vectors with 20 probabilities for each character in those

sequences. For all aligned residues in the original sequences, (r_{1j}, r_{2j}) with $r_{1j} \in S_1$ and $r_{2j} \in S_2$, we want to minimize the sequence identity loss (si) function taking another non-aligned residue in a window of one to five residues before and after the aligning position r_{2k} with $j - 5 \leq k \leq j + 5$ as third parameter. The loss function to be minimized is defined as:

$$l_{cn_{si}} = l(r_{1j}, r_{2j}, r_{2k}) = d(\frac{r_{1j}}{temp}, \frac{r_{2k}}{temp}) * neg - d(\frac{r_{1j}}{temp}, \frac{r_{2j}}{temp}) * pos \tag{2.3.5}$$

with the cosine distance metric d, r_{1j} as the anchor, r_{2j} as a positive example of an aligned residue, r_{2k} as a negative example [41]. The vectors r_{1j}, r_{2j}, and r_{2k} are divided by *temp* to increase the differences in probabilities in the vectors. This reinforces the choice of the highest probability and therefore controls the certainty of the model. The weights *pos* and *neg* define the emphasis on the positive and negative values.

The sequence similarity loss *ss* is a contrastive-style loss as well, by penalizing high similarity to negative samples and rewarding high similarity to positives. Additionally, the probabilities represented in the vectors are weighted by the BLOSUM60 matrix using matrix multiplication. The sequence similarity loss function to be minimized [41] is defined as:

$$l_{cn_{ss}} = l(r_{1j}, r_{2j}, r_{2k}) = d(\frac{r_{1j}}{temp} \times B60, r_{2k}) * neg - d(\frac{r_{1j}}{temp} \times B60, r_{2j}) * pos \tag{2.3.6}$$

The vector of the anchor (of dimension $(1, 20)$) is divided by *temp* and multiplied by the BLOSUM60 matrix (of dimension $(20, 20)$). The resulting $(1, 20)$ matrix is once more multiplied with the positive example vector and once with the vector of the negative example. This results in two $(1, 20)$ vectors. The resulting vector for the positive example is weighted by the parameter *pos*. The resulting vector for the negative example is weighted by the parameter *neg*. This way, the comparison becomes weighted by an actual BLOSUM matrix.

The total loss function for the *LM-head Neural Network* is the weighted sum of all losses and the corresponding parameters are given as:

$$l_{LM-head_{total}} = l_{cb} * en + l_{cb} * us + l_{cn_{si}} * si + l_{cn_{ss}} * ss$$

All parameters have a configuration space of 0.0 to 1.0 for the same reason as in the *VQ-VAE Neural Network*.

The Usage of the Neural Networks

As illustrated by the architectures, we are essentially training an encoder (E), a decoder (D), and a vector quantizer (VQ) for the *VQ-VAE Neural Network*, and the language modeling head (LM-head) for the *LM-head Neural Network*. During training, their weights get adapted. We use subsets of the SCOPe and Pfam databases as training data and another subset of each database for validation. The training and validation data are identical for both networks. The training data contains 24627 pairs of sequences, and the validation data contains 11549 pairs of sequences. Due to the embedding process, we are limited to sequences of length up to 1024 characters. One epoch is one pass through the entire training set. After every epoch, the discretized embeddings of all sequences from the validation data are created. The corresponding substitution matrix is computed, MMseqs2 is run, and the identification quality score is calculated, what will be explained in Chapter 4. This score is logged. As soon as this score stops decreasing for a set number of epochs, the training stops, and the run with the lowest cost is saved.

Chapter 3

Custom Substitution Matrix

Protein alignment approaches make use of so-called substitution matrices. For the alphabet A, different from the one of amino acids, we are missing a suitable substitution matrix. The substitution matrices for the amino acid alphabet are 20 by 20 matrices. Based on empirically observed amino acid exchanges in evolutionarily related protein sequences, each entry represents the approximate probability at which one amino acid substitutes another one of the 20 amino acid residues in proteins over time. There is no matrix that can be used on all occasions. Instead, different situations require different matrices to create biologically significant alignments [42]. To use protein alignment approaches on sequences in another alphabet A, we create a substitution matrix custom-made for that alphabet. Inspired by the approach of Van Kempen et al. [37] to create a substitution matrix for a protein structure search, we create this matrix as follows:

For the SCOPe database [39] that we use in our experiments, there are pairwise structural alignments of proteins in the amino acid alphabet. We call these alignments our ground-truth alignments as they were computed using an established structural alignment method and substitution matrix.

We have an alphabet A with m letters and the ground-truth alignments of the SCOPe sequences. We want these alignments for the sequences in this alphabet A. Therefore, we apply the sequences in this alphabet A to ground-truth alignments by inserting gaps in predefined positions.

As an example, we have the pairwise structural alignment of the two proteins *d1s69a_* and *d1dlwa_* . The alignment consists of the two aligned lines (1). The sequences of *d1s69a_* and *d1dlwa_* in the new alphabet are given in (2). With the conversion explained above,

the lines converted to the new alphabet are given in (3).

```
(1) The aligned form of d1s69a_ and d1dlwa_:
d1s69a_ = S T L ... K E L V E N H G L N ... A G A P A H K R ...
d1dlwa_ = - S L ... A N M - - - - G V S ... A E T - - V R G ...

(2) d1s69a_ and d1dlwa_ with the new alphabet:
d1s69a_ = C T O ... K P M M M Q L P C C ... A K P P K I P H ...
d1dlwa_ = T O A ... M P N P C C C I O A ... N K Q O A I P N ...

(3) The aligned form of d1s69a_ and d1dlwa_ with the new alphabet:
d1s69a_ = C T O ... K P M M M Q L P C C ... A K P P K I P H ...
d1dlwa_ = - T O ... M P N - - - - P C C ... N K Q - - O A I ...
```

Now that we have the alignments in our new alphabet, we want to compute an $m \times m$ substitution matrix.

This matrix must describe how likely it is that another character substitutes a character. To estimate this, we use the Bayesian approximation approach by transforming this into a log odds score (Equation 3.0.1), which compares the probability of observing two characters aligned against the probability of observing these characters randomly. In essence, it measures how much more likely two characters are to align than to occur together by chance. These probabilities are extracted from the ground-truth alignments.

The input consists of pairwise protein sequence alignments in the alphabet A. We do not consider all alignments when calculating the scores for the matrix. The benchmarking process to check correctness and reliability resulted in thresholds to pre-filter alignments, ensuring diversity in the sequences considered. The first step is therefore to filter the alignments according to these thresholds [43]. The remaining set of alignments is processed sequentially as follows.

Let L be the set of alignments in the alphabet A after filtering. For each alignment in L, $l \in L$, the occurrences of pairs (s_r, r_j) and single characters $s_k \in S$ and $r_k \in R$ are counted. This results in frequencies per letter of the alphabet and frequencies per aligned pair of letters. Now, the entries can be calculated. The calculation of the entry for the substitution of X by Y (equation 3.0.1) works as follows.

We calculate the substitution frequency $p(X, Y)$ by dividing the number of occurrences of the aligned pair (X, Y) by the number of aligned pairs in total. Then we calculate the probability for the letter X to occur independently ($p(X)$) by dividing the number of occurrences of the letter X in all alignments by the number of characters in total. The

estimated probabilities $p(X)$ and $p(Y)$ are then multiplied to get the estimated probability that the two letters X and Y occur independently. For protein sequence alignment, we want an additive scoring system, which means that the overall score of an alignment is the sum of the individual scores per substitution [44]. This, for example, enables the use of penalty scores added to the scores of the matrix [44].

Furthermore, the score for aligning two sequences consists of the conditional probability for each substitution. This means that the total probability is the product of each probability, which is identical to the sum of the logarithms of individual probabilities. From this, we can work with the logarithms of the individual probabilities instead of with the probabilities directly. That simplifies the calculation and is the reason we take the logarithm of the ratio of $p(X, Y)$ and $p(X)p(Y)$. This is known as the log-odds ratio [44].

$$S(X, Y) = log_2 \frac{p(X, Y)}{p(X)p(Y)} \tag{3.0.1}$$

We can use this approach to compute the entries of our substitution matrix because it is an established process, applied to compute well-known amino acid substitution matrices [44].

Substitution matrices calculated using this approach are further used to obtain sequence alignments for the methods trained in this thesis.

Chapter 4

Scoring Metric

We want to evaluate the alignment result created by MMseqs2. We introduce two scoring functions. One to evaluate the quality of computed alignments, and the other one to evaluate the ability to identify the correct family, superfamily, and fold of a protein.

4.1 Alignment Quality

The quality of an alignment is evaluated using a scoring metric based on the benchmark used to train Foldseek, a protein structure search method [37]. They compare the alignment between two sequences and a reference alignment from a predefined ground-truth set. The comparison is based on two values that are computed, called *sensitivity* and *precision*.

The *sensitivity* evaluates how many of these aligned residues from the reference alignment are actually aligned in the created alignment. This means that the *sensitivity* is lower in the case of missing pairs of residues in the created alignment. We compute the *sensitivity* by dividing the number of pairs of aligned residues that are identical in both alignments by the total number of aligned pairs in the ground-truth.

The *precision* evaluates how many of the created pairs are actually correct. This means that *precision* is low in the case of created aligned characters that are not aligned in the reference alignment. The *precision* is computed by dividing the number of pairs of aligned residues that are identical in both alignments by the total number of aligned pairs in the created alignment.

As a conclusion, the *sensitivity* states the coverage of the ground truth and the *precision* the correctness of the created alignment.

In the following section, we are going to provide a formal definition of the scoring metric, followed by a concrete visual example.

Formal Definition For one local alignment ab of two sequences a and b, denoted as $ab = (a', b')$ (see the definition of local alignment for more details in Chapter 2), the *sensitivity* and *precision* are calculated as follows. First, we create a set $pairs$ that includes the pairs of aligned letters, defined in equation 4.1.1.

We iterate over each character (character means letters and gaps) in the aligned sequences a' and b' beginning at the start indices $s_{a'} \in \mathbb{N}$ and $s_{b'} \in \mathbb{N}$ respectively. In addition, we maintain two continuous indices $i_{a'} \in \mathbb{N}$ and $i_{b'} \in \mathbb{N}$ beginning at $i_{a'} = s_{a'}$ and $i_{b'} = s_{b'}$. When iterating, we check at each step if both characters are letters and not gaps. If they are both letters, the current pair of continuous indices $(i_{a'}, i_{b'})$ is added to the set $pairs$. The index $i_{a'}$ increases by one every time there is a letter in a' and not a gap, identically for $i_{b'}$.

$$
\begin{aligned}
&\textbf{Given:} \quad ab = (a', b'), \quad s'_a, s'_b \in \mathbb{N}, \quad \text{len}(a') = \text{len}(b') = n \\
&\textbf{Initialize:} \quad pairs = \emptyset, \quad i_{a'} = s_{a'}, \quad i_{b'} = s_{b'} \\
&\textbf{For all } k \in \{s_{a'}, \ldots, n\},\ l \in \{s_{b'}, \ldots, n\} : \\
&\begin{cases}
\text{if } a'[k] \notin \{-\} \text{ and } b'[l] \notin \{-\}, & pairs := pairs \cup \{(i_{a'}, i_{b'})\} \\
\text{if } a'[k] \notin \{-\}, & i_{a'} := i_{a'} + 1 \\
\text{if } b'[l] \notin \{-\}, & i_{b'} := i_{b'} + 1
\end{cases}
\end{aligned}
\tag{4.1.1}
$$

This iterative process is done for both the computed alignment ($pairs_c$) and the reference alignment ($pairs_r$). Now that we have obtained the set $pairs$ for both alignments, we can start to count. The number of pairs of aligned residues that are identical in both alignments is the number of pairs of indices that are in both sets $pairs$, we define it as $num_{id} = |pairs_c \cap pairs_r|$. Furthermore, the total number of aligned pairs in the computed alignment is denoted as $total_c = |pairs_c|$. The total number of aligned pairs in the reference alignment is denoted as $total_r = |pairs_r|$. The *sensitivity* is computed as:

$$sensitivity = \frac{num_{id}}{total_r} \tag{4.1.2}$$

and the *precision* as:

$$precision = \frac{num_{id}}{total_c} \tag{4.1.3}$$

The *sensitivity* and *precision* are computed for each alignment created by MMseqs2. The total *sensitivity* for all computed alignments is then the arithmetic mean of all *sensitivity* scores. The total *precision* for all computed alignments is the arithmetic mean of all *precision* scores.

The resulting alignment quality score, which we refer to as *qualityAln* (equation 4.1.4), is furthermore the arithmetic mean of the total *sensitivity* and the total *precision*.

$$qualityAln = \frac{sensitivity_{total} + precision_{total}}{2} \tag{4.1.4}$$

From given structural alignments as ground-truth we are using a random sample of 10% alignments as reference alignments for the scope of this thesis. The reason for not using the full set is that it would take more time for the experiments to get to a comparable outcome.

Example In the following, we are going to describe a complete example on how to obtain the alignment quality for one concrete alignment of two sequences. We want to compute the alignment quality for the MMseqs2 alignment c of the two sequences a and b: $c = (a', b')$. The structural alignment of the reference set is indicated by $r = (a'', b'')$. The sequence a is given as:

M I C R F I D T H C G M L E K

and sequence b as:

H H H H X V D T H A H D A K E

In this example, the structural reference alignment r looks as follows:

a''	M	I	C	R	F	I	D	-	T	H	C	G	M	L	E	K
b''	H	H	H	H	X	V	D	T	H	A	H	D	A	K	E	-

And the computed local alignment c is given as:

a'	F	I	D	T	H	C	-	G	M	L	E	K
b'	X	V	D	T	H	A	H	D	A	K	E	-

With the help of these visualizations we can see that the computed alignment c does not start at the first position of the sequences a and b. This is because MMseqs2 is computing a local alignment and restricts the output to the locally aligned part of the alignment (see Chapter 2.3). Therefore, the start indices for the reference alignment sequences are given

as $s_{a'} = 5$ and $s_{b'} = 5$ because both start with the fifth character of the original sequences a and b.

We want to create the sets $pairs_c$ and $pairs_r$. For that, we iterate through both alignments separately. Exemplary for the computed alignment:

	5	6	7	8	9	10		11	12	13	14	15
a'	F	I	D	T	H	C	-	G	M	L	E	K
	5	6	7	8	9	10	11	12	13	14	15	
b'	X	V	D	T	H	A	H	D	A	K	E	-

We start at index five. The characters at this index are both letters, so we can add this pair to $pairs_c$: $pairs_c = \{(5,5)\}$. Identically for the next five characters which brings us to: $pairs_c = \{(5,5),(6,6),(7,7),(8,8),(9,9),(10,10)\}$. The next pair includes a gap in sequence a', therefore only the continuous index of sequence b' is increased by one. This brings us to the next pair, which consists of both letters, being at indices $(11,12)$. The following three pairs are again ony letters, so we increase both continuous indices, which brings us to: $pairs_c = \{(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,12),(12,13),(13,14),(14,15)\}$. Next, there is a gap in sequence b'. We therefore increase the index only for sequence a' but do not add another pair.

The resulting set $pairs_c$ is thus:
$pairs_c = \{(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,12),(12,13),(13,14),(14,15)\}$ with a length of: $total_c = |pairs_c| = 10$.

The resulting set $pairs_r$ is the following:
$pairs_r = \{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,9),(9,10),(10,11),(11,12),$
$(12,13),(13,14),(14,15)\}$ with a length of: $total_r = |pairs_r| = 14$.

Now that we have the two sets, we are going to count the pairs that are present in both, using the following visualization.

The intersection is therefore: $num_{id} = |pairs_c \cap pairs_r| = |\{5,5),(6,6),(7,7),(11,12),$
$(12,13),(13,14),(14,15)\}| = 7$, marked in orange.

Now we can compute the *sensitivity* and *precision* for the computed alignment c. The *sensitivity* is computed as $sensitivity = \frac{num_{id}}{total_r} = \frac{7}{14} = 0.5$ and the *precision* as $precision = \frac{num_{id}}{total_c} = \frac{7}{10} = 0.7$.

	1	2	3	4	5	6	7		8	9	10	11	12	13	14	15
a''	M	I	C	R	F	I	D	-	T	H	C	G	M	L	E	K

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
b''	H	H	H	H	X	V	D	T	H	A	H	D	A	K	E	-

	1	2	3	4	5	6	7	8	9	10		11	12	13	14	15
a'					F	I	D	T	H	C	-	G	M	L	E	K

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
b'					X	V	D	T	H	A	H	D	A	K	E	-

4.2 Identification Quality

The scoring function to evaluate the identification of the correct family, superfamily, and fold of a protein is based on the SCOPe benchmarking process presented by Van Kempen et al. [37]. This corresponds to the area under the curve of the sensitivity up to the first false positive detection [37].

MMseqs2 computes a quality measure *E-value* for every alignment of a query with a target [8]. We use this *E-value* to compute the score for this metric. For a query Q, let targets $= T_1, ..., T_n$ be the sequences ordered by decreasing *E-value*. The score $scoreFAM(Q)$ is the largest number i such that $FAM(Q) = FAM(T_j)$ for all $j \leq i$ ($scoreSFAM(Q)$ and $scoreFold(Q)$ for the superfamily and the fold, respectively). We compute the arithmetic mean over all queries as $scoreFAM$ ($scoreSFAM$, $scoreFOLD$). The resulting overall score for the identification is defined as the arithmetic mean of the mean scores for all queries and family, superfamily, and fold presented in equation 4.2.1.

$$qualityIdent = \frac{scoreFAM + scoreSFAM + scoreFOLD}{3} \tag{4.2.1}$$

Chapter 5

Optimization Pipelines

We implement optimization processes, called pipelines. In the following sections, we formally introduce the pipelines, their inputs, computational steps, and outputs.

The *Amino Acid Pipeline* (AAP) and the *k-Means Pipeline* (KMP) are developed to answer the question regarding the influence of gap penalties. The *VQ-VAE Neural Network Pipeline* and the *LM-head Neural Network Pipeline* are used to optimize the training hyperparameters for the two neural networks.

So far, we have computed the quality metrics, *qualityAln* and *qualityIdent*, which are best when maximized. We need to reformulate it to be a minimization problem for the hyperparameter optimization package SMAC3 [14]. SMAC3 wants to minimize costs; therefore, we introduce the *costs*. High quality corresponds to low costs and vice versa. We calculate the costs by subtracting the quality from 1: $Cost = 1 - Quality$. Depending on whether we are interested in alignment or identification quality, the respective scoring metric is applied.

All methods mentioned in the following sections are explained in detail in Chapter 2, 3, and 4.

5.1 Influence of Gap Penalties

We want to find out the influence of the gap penalties on the alignment process. To do so, we performed a grid search of all possible penalty values to observe the impact of changing penalties on alignment in the amino acid (AAP) and k-Means (KMP) cases.

To compute alignments, the AAP and KMP find good parameter values for the gap penalties. There are two gap penalties to define, one for opening (go) and one for extending (ge) a gap with $go, ge \in [0, 1]$. The respective value penalizes when a new gap is opened

and when an existing one is extended while aligning two sequences [45]. We transform these penalty values so that the penalties used with MMseqs2 depend on the applied substitution matrix M as follows. The interval $[min(M), max(M)]$ is defined by the minimum $min(M)$ and maximum $max(M)$ score present in M. We call the gap-open penalty transformed to be used with MMseqs2 $go_{mmseqs2}$. To get $go_{mmseqs2}$, we multiply go with the range of maximum and minimum values present in M: $go_{mmseqs2} = go * (max(M) - min(M)) + min(M)$. Equally we get $ge_{mmseqs2}$ utilizing the maximum and minimum values of the matrix M as follows: $ge_{mmseqs2} = ge * (max(M) - min(M)) + min(M)$. This way, we can reach the $min(M)$ and $max(M)$ values as penalty values. This discretization enables us to not depend on a specific matrix, so that we can get penalty values for the actual substitution matrix used.

The AAP and KMP differ with respect to the representation of the protein sequences and alignments. One approach uses the original amino acid alphabet. The other approach uses an alphabet generated by the k-Means clustering of the embeddings of the SCOPe protein sequences [39] computed by the Protein Language Model ProtT5 [46]. In the following subsections, each pipeline is visualized and explained in sequential order.

5.1.1 Precomputations

Some data are independent of the variable parameters for the pipelines and can be precomputed. In the upcoming figures, the precomputed parts are marked in pink.

Furthermore, we are using the BLOSUM62 substitution matrix, which is one of the BLOSUM standard matrices for protein alignments [20].

We computed structural pairwise sequence alignments of all protein sequences S_{AA} from the SCOPe database [39]. This set of alignments is reused in all pipelines. We refer to it as the ground-truth alignment $GTA(S)$ with $S = S_{AA}$.

We use the embedding layer of the ProtT5 Protein Language Model to obtain the high-dimensional embedding vectors for all protein sequences from the SCOPe database [39]. We refer to these vectors as $E = embed(S_{AA})$.

5.1.2 The Amino Acid Pipeline

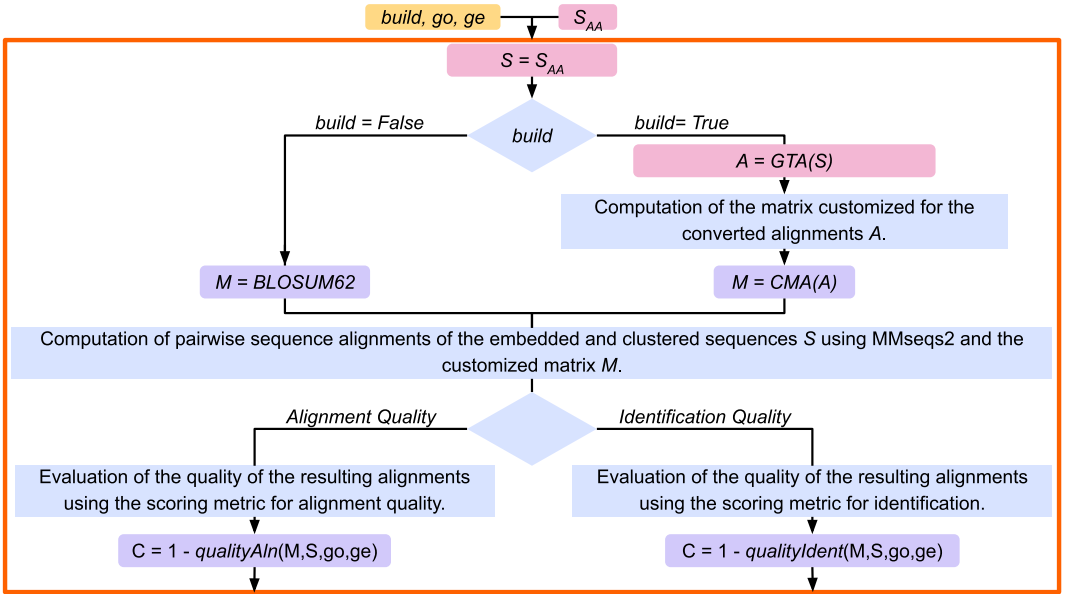


Figure 5.1: *Schema of the Amino Acid Pipeline. The input consists of the parameters to be optimized in the yellow box, and the fixed and precomputed input in the red boxes. Visualized is a flow diagram of the process for the Amino Acid Pipeline with all steps and intermediate outputs.*

The parameter space of this pipeline is ($build$, go, ge). Depending on the value of $build$, the BLOSUM62 matrix or a custom matrix is chosen.

In case of not building a matrix ($build = False$), MMseqs2 is applied to compute alignments of the protein sequences S_{AA} from the SCOPe database using the BLOSUM62 matrix and the provided parameter values for the gap penalties go and ge. The quality of the resulting alignments is then evaluated using one of our scoring metrics. What metric to use depends on the exact experiment and will be elaborated in Chapter 6. The pipeline output is the cost C obtained from this quality.

In the contrary case of building a customized matrix ($build = True$), the matrix must be built first. To compute the scores of this matrix M, the existing ground-truth alignments of the original sequences S_{AA} are used ($GTA(S_{AA})$). MMseqs2 is applied to compute alignments of S using the new matrix M and the gap penalties go and ge. The quality of the resulting alignments is then calculated using one of the scoring metrics and subsequently converted to the cost C.

5.1.3 The k-Means Pipeline

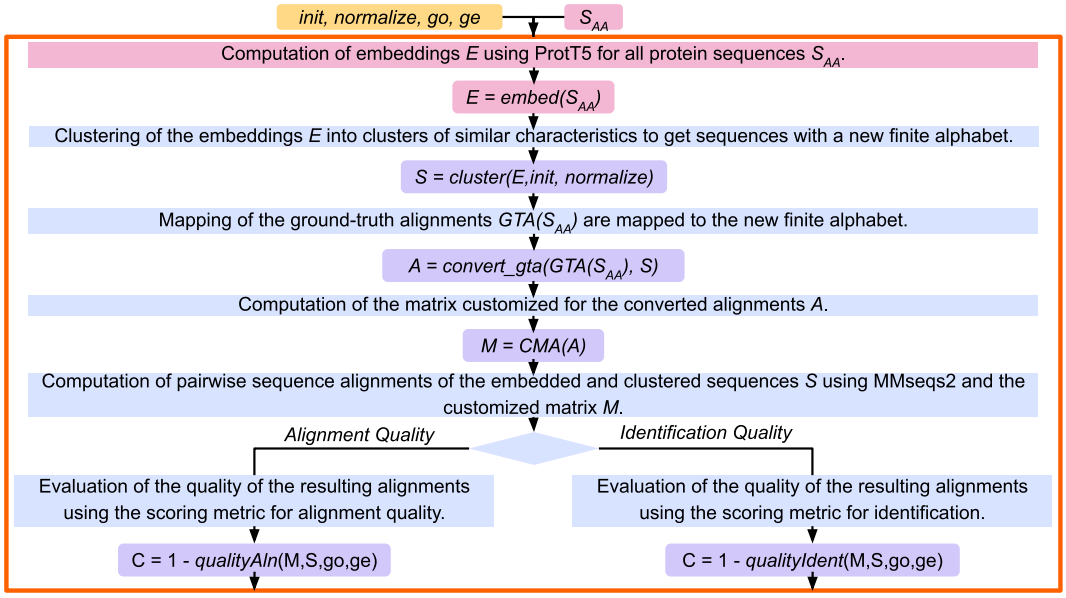


Figure 5.2: *Schema of the k-Means Pipeline. The input consists of the parameters to be optimized in the yellow box, and the fixed and precomputed input in the red boxes. Visualized is a flow diagram of the process for the k-Means Pipeline with all steps and intermediate outputs.*

The parameter space of this pipeline is ($init$, $normalize$, go, ge). This pipeline uses the precomputed high-dimensional embedding vectors E from the language model ProtT5. The input consists of all sequences S_{AA} of the SCOPe database. The resulting embeddings E are sequences of vectors from $\mathbb{R}^n$. The k-Means clustering algorithm clusters vectors in a sequence into clusters of similar characteristics and maps their centers to a finite alphabet (see Chapter 2.3.2). The parameters $init$ and $normalize$, define the clustering process. This results in a set of sequences S in this finite alphabet.

After clustering the embeddings, we map the ground-truth alignments $GTA(S_{AA})$ to the set of sequences S. For these converted alignments A we compute a custom matrix M. With this customized matrix, MMseqs2 subsequently computes pairwise sequence alignments of the embedded and clustered sequences S. Again, the quality of these alignments is evaluated using one of the scoring metrics, depending on the experimental interest, and then converted to a cost C.

5.2 Parameters to Compute Discrete Embedded Sequences

To answer the questions regarding the influence of gap penalties and regarding the parameters for the neural networks, we implemented two pipelines. One pipeline for each of the two neural networks explained in Chapter 2.3.

5.2.1 The VQ-VAE Neural Network Pipeline

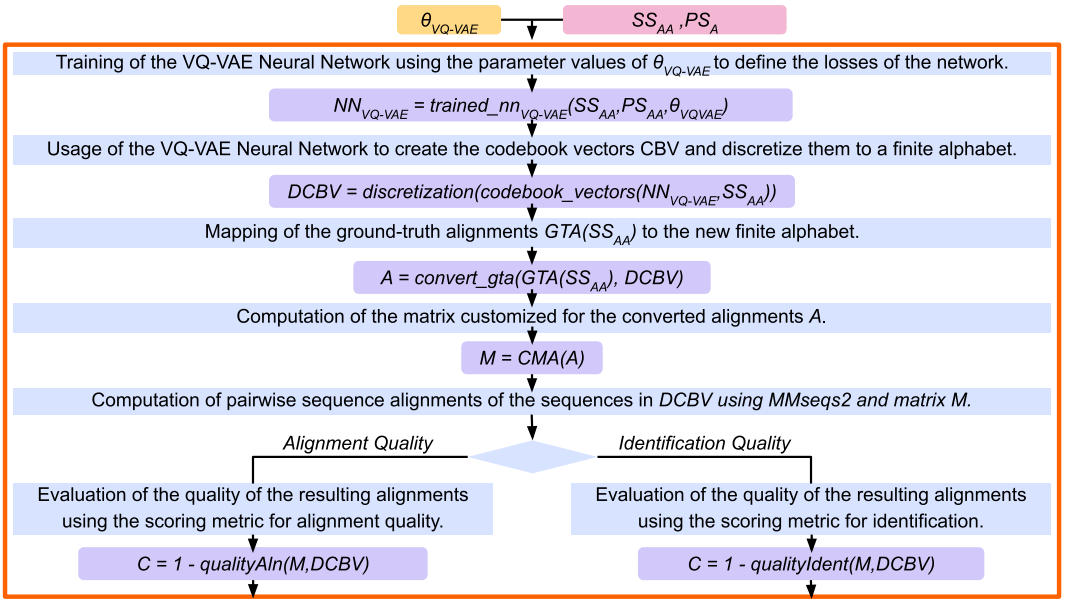


Figure 5.3: *Schema of the VQ-VAE Neural Network Pipeline. The input consists of the parameters to be optimized in the yellow box, and the fixed input sequences in the red box. Visualized is a flow diagram of the process for the VQ-VAE Neural Network Pipeline with all steps and intermediate outputs.*

With this pipeline, we want to optimize multiple parameters, represented as θ_{VQ-VAE}. A detailed explanation of these parameters is provided in the *VQ-VAE Neural Network* training part of Chapter 2.3. The set of SCOPe protein sequences SS_{AA} in the amino acid alphabet and the Pfam sequences PS_{AA}, another database of protein sequences [40] are given as fixed input.

This pipeline starts with the training of a neural network. The trained network is then used to create codebook vectors for protein sequences from a subset of the SCOPe database SS_{AA} and the Pfam database PS_{AA}. Next, we discretize these vectors to obtain a finite alphabet. The remaining part is identical to the previous pipelines. We convert the

ground-truth alignments to the sequences in this alphabet, compute a customized substitution matrix, and create alignments using MMseqs2. The quality of the output is calculated using one of the scoring metrics and converted to a cost.

5.2.2 The LM-head Neural Network Pipeline

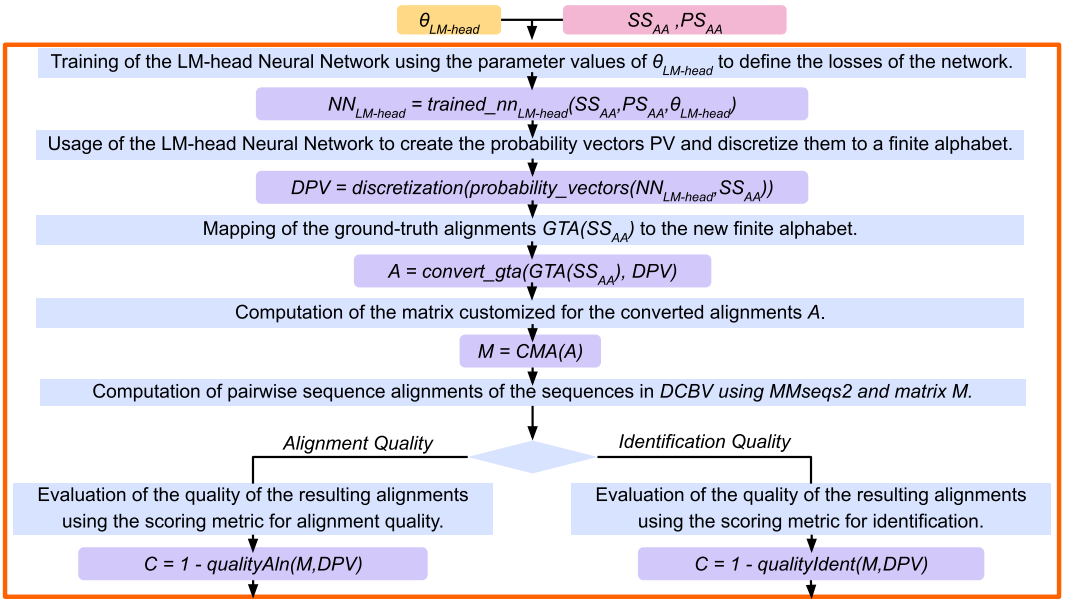


Figure 5.4: *Schema of the LM-head Neural Network Pipeline. The input consists of the parameters to be optimized in the yellow box, and the fixed input sequences in the red box. Visualized is a flow diagram of the process for the LM-head Neural Network Pipeline with all steps and intermediate outputs.*

This pipeline is designed to optimize the parameters denoted by $\theta_{LM-head}$. A detailed explanation of these parameters is provided in the *LM-head Neural Network* training section of Chapter 2.3. The pipeline shares the same fixed input and structure as the *VQ-VAE Neural Network*, with the primary difference being that it uses a pre-trained language modeling head to generate probability vectors, rather than relying on codebook vectors. This way, the pipeline is identical to the previous one from the point where we have obtained the new finite alphabet. The mapping of the ground-truth alignments, the matrix computation, the application of MMseqs2, as well as the computation of the scoring metric is the same as for the *VQ-VAE* network. The output again is the quality measure obtained by one of the scoring metrics.

Chapter 6

Experiments and Results

In this chapter, we provide an overview of the experiments we conduct.

Each section dedicated to an experiment has the following structure. First, we indicate what knowledge we want to gain ('Why'), then describe how we set up the experiment ('How'), followed by a definition of what we want to measure ('What') and the result ('Result').

The resulting cost values of both scoring metrics are values between 0 and 1: $0 \leq cost \leq 1$, where 0 is the best.

6.1 Implementation

The programming language we used is Python. The experiments run on the sciCORE cluster of the University of Basel.

For each pipeline, we created a separate SMAC3 script. We evaluate the results by comparing the cost.

6.2 Number Of Epochs

Why The parameter *patience* decides on the number of epochs after which the training concludes in case of no further reduction of cost. If the *patience* is set to, for example, 6, the training will conclude if there is no further improvement in the last 6 epochs. We want to find the optimal value of the parameter *patience*, to not miss good results, but also to avoid overtraining and excessive use of computational resources.

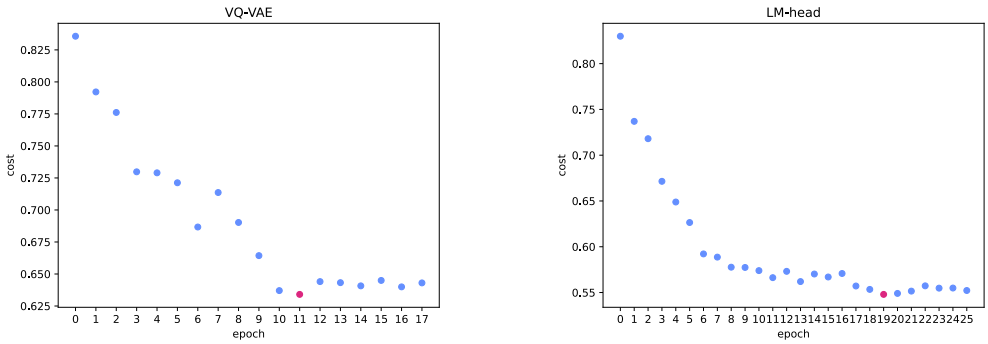
How We run the optimization process for both neural network pipelines for ten trials with patience set to six and a parameter configuration from previous work. The number six comes from the fact that it was manually set in the configurations of previous work, and we are using it as a starting point.

What We want to create plots that show the change in cost across the epochs of training. For each training, one per trial, we evaluate the minimum number of epochs after which the training could have stopped after finding a reasonable cost while still returning the same best value.

Result We chose two exemplary trials, one out of ten per model, to visualize the answer to our question regarding the minimum number of epochs we need to set the patience to in Figure 6.1. Since the behavior across trials was consistent, selecting a single representative run per model yields the same conclusion as considering all ten. Figure 6.1(a) shows that if we find a new minimum cost and result in a higher cost in the following epoch, a maximum of 3 epochs later, a lower cost is found. This, for example, is the case for epoch 6. Here we find the lowest cost found so far, and in the next epoch, the cost increases. In the following epoch number 8, the cost decreases again, but is still higher than the cost in epoch 6. Only in epoch 9 we find a cost that is again lower, so essentially three epochs later. With a patience set to 3, we therefore would not stop here, as with the patience set to 6. In this case, a patience of 3 would have resulted in the same output, as epoch 11 with a cost of 0.67883 being the best found solution and the criterion to stop the training.

Figure 6.1(b) shows that whenever we arrive at a new local optimum regarding the costs followed by an increase, for example in the epochs 8, 11, 13, 15, and 19, the costs are decreasing again a maximum of 4 epochs later (this is the case for the loss in epoch 13).

In summary, for these examples, patience of 4 has the same outcome as the previously used patience of 6 while preserving computational resources. This value is therefore used for further experiments.



(a) *One application of the VQ-VAE Neural Network: best cost found in epoch 11*

(b) *One application of the LM-head Neural Network: best cost found in epoch 19*

Figure 6.1: *Visualizations of costs per epochs during exemplary chosen trainings of each model. The dot of the epoch with the best cost is marked in red.*

6.3 Baseline - Reference Cost

Why To evaluate how much we can improve the neural network training with our hyperparameter optimization, we need to obtain a baseline result for comparison.

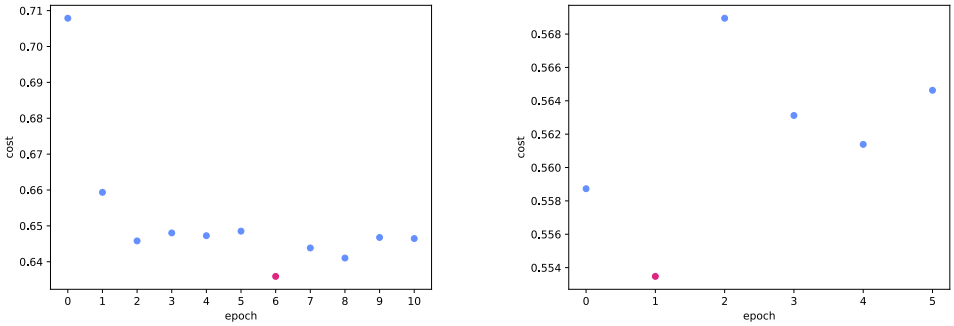
How We run the neural network pipelines for the same patience as our optimization process will have allocated to ensure comparability. Essentially, we set it to 4, as the previous experiment showed that this is enough. The configuration of the parameter values we use is based on intuitively chosen values that have been used in previous work [3] and were set within our research group.

What We want to compute the cost for the parameter values used in previous work. This cost will be our baseline for evaluating whether and how much the optimization process improves training and the ability to detect remote homology.

Result For the *VQ-VAE Neural Network*, the training stopped after **11** epochs because there was no improvement in the previous 4 epochs. The lowest cost found with the baseline configuration of the parameters θ_{VQ-VAE} was **0.63592**.

For the *LM-head Neural Network*, the training stopped after **6** epochs because there was no improvement in the previous 4 epochs. The lowest cost found with the baseline configuration of the parameters $\theta_{LM-head}$ was **0.55347**.

These cost values represent our reference values to assess whether optimization improves the procedure. If we find a configuration with a lower cost, we have found an improvement.



(a) *VQ-VAE Neural Network: baseline cost per epoch. Best cost of* 0.63592 *at epoch* 6.

(b) *LM-head Neural Network: baseline cost per epoch. Best cost of* 0.55347 *at epoch* 1.

Figure 6.2: *Baseline cost per epoch visualization. The lowest cost is marked in red.*

6.4 Experiment 1: Influence of Gap Penalties

What influence do gap penalties have? We want to know if and how much the choice of gap penalties influences the result. The outcome of this experiment determines whether we will include the gap penalties as parameters to be optimized in the following experiment.

We are trying all combinations of possible parameter values for the amino acid alphabet, an alphabet introduced by Van Kempen et al. [37], capturing the fundamental structure of a protein's fold, and the alphabets of the baseline configurations for the two neural networks.

The possible parameter values for the gap penalties depend on the minimum and maximum scores present in the substitution matrix used.

To assess the result, we conduct the experiment twice. First, using the alignment quality scoring metric (Chapter 6.4.1) and then using the identification quality scoring metric (Chapter 6.4.2).

We want to measure the difference in cost for varying combinations of penalties. We computed the cost for each possible parameter combination and created color-coded plots. The gray area represents the combinations where MMseqs2 did not find alignments. One reason is that MMseqs2 does not allow the gap-extension (ge) penalty to be larger than the gap-open (go) penalty. To answer our question about this experiment, we need to examine the different colors and costs plotted in the corresponding cells.

6.4.1 Influence on the Alignment Quality

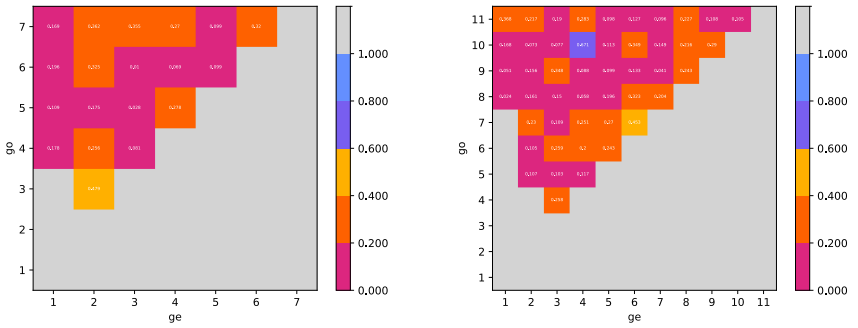
Why We want to know what influence the gap penalties have on the quality of an alignment.

How As described above, we try all combinations of possible parameter values for *go* and *ge* for the different alphabets. To assess the results, we are using the scoring metric for the alignment quality described in Chapter 4.

What We visualize the range of cost values for all possible pairs of *go* and *ge* values. The color gray indicates that no alignments could be created by MMseqs2.

Result We evaluate the results per alphabet and draw a conclusion for the question afterwards.

Amino Acid Alphabet The Amino Acid Alphabet seems to hinder good alignments visible due to the gray blocks in Figure 6.3 also for pairs of gap penalties that are not prohibited by MMseqs2. The discrepancy between the different costs is what is of interest to us. For the customized matrix, see Figure 6.3 (a), the cost values are in a range of 0.469. For the BLOSUM62 matrix, see Figure 6.3 (b), there is a range of 0.63.

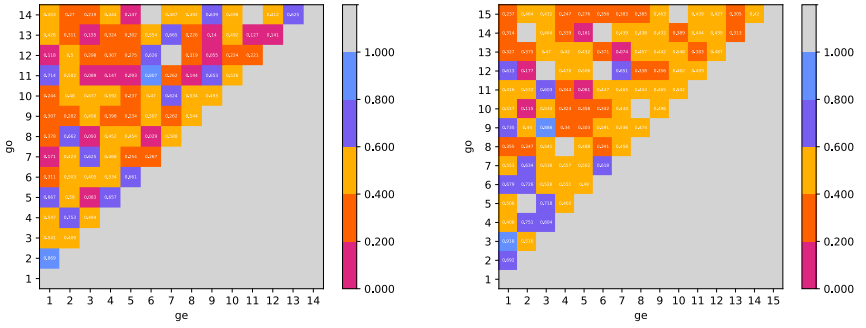


(a) *Amino Acid Alphabet and Custom Substitution Matrix*

(b) *Amino Acid Alphabet and BLOSUM62 Matrix*

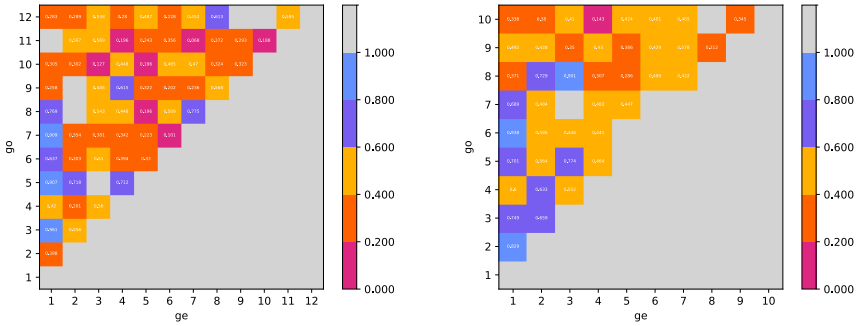
Figure 6.3: *Alignment quality per pair of gap penalties for the Amino Acid alphabet with a custom built substitution matrix (a) and the BLOSUM62 matrix (b). The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

For the k-Means pipeline, we distinguish two different initialization strategies, and whether or not we are using a normalization step (as described in Chapter 2). For the *k-means++* initialization without normalization (Figure 6.4(a)), the cost is in a range of 0.84. The *k-means++* initialization with normalization (Figure 6.4(b)) shows a range of 0.857. The range for *random* initialization without normalization (Figure 6.4(c)) conducts 0.893. The *random* initialization with normalization resulted in a range of 0.795.



(a) *k-Means Alphabet, Custom Substitution Matrix with k-means++ initialization, and without normalization.*

(b) *k-Means Alphabet, Custom Substitution Matrix with k-means++ initialization, and with normalization.*



(c) *k-Means Alphabet, Custom Substitution Matrix with random initialization, and without normalization.*

(d) *k-Means Alphabet, Custom Substitution Matrix with random initialization, and with normalization.*

Figure 6.4: *Alignment quality per pair of gap penalties for the Amino Acid alphabet with a custom built substitution matrix for the k-Means clustering approach. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

3Di Alphabet The grid for the *3Di* alphabet in Figure 6.5 has a range of different cost values of 0.723.

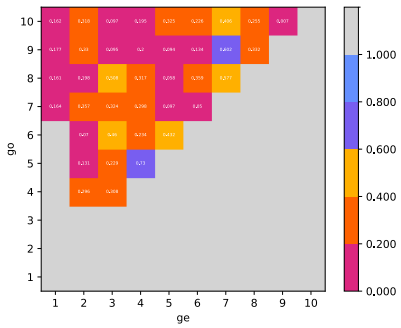


Figure 6.5: *Alignment quality per pair of gap penalties for the 3Di alphabet with a custom built substitution matrix. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

VQ-VAE Baseline Alphabet The grid for the alphabet computed with the *VQ-VAE* network in Figure 6.6 has a range of different cost values to vary the gap penalties of 0.941.

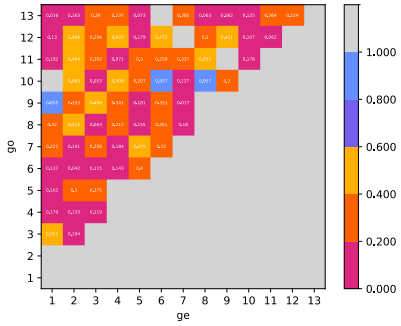


Figure 6.6: *Alignment quality per pair of gap penalties for the alphabet computed with the baseline configuration of the VQ-VAE network. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

LM-head Baseline Alphabet The grid for the alphabet computed with the *LM-head* network in Figure 6.7 has a range of different cost values to vary the gap penalties of 0.602.

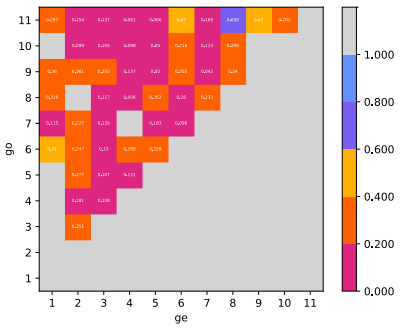


Figure 6.7: *Alignment quality per pair of gap penalties for the alphabet computed with the baseline configuration of the LM-head network. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

We could show that the gap penalties do have an impact on the alignment quality due to largely differing cost values in the ranges from 0.469 up to 0.941. This brings us to the question whether we should add these two parameters to the set of hyper-parameters optimized by SMAC for the neural network pipelines. We will answer this question with the next experiment.

6.4.2 Influence on the Identification

Why For the question regarding good parameter values to compute discrete embedded sequences with the neural network pipelines, we are going to use the scoring metric for the identification because we want to optimize the networks to improve their search ability. As search is more important, the fact that the identification is correct is more important than the quality of an alignment.

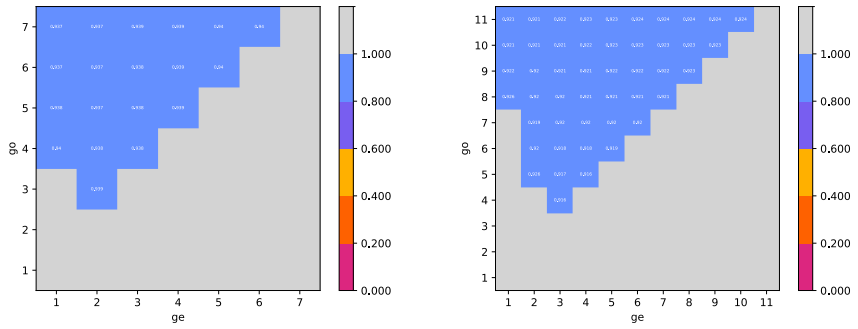
Because the penalties have a proven impact on the alignment quality, as shown in the previous experiment, we want to find out if they influence the identification as well or if we can exclude these two parameters from our neural network optimization process to save resources.

How We try all combinations of possible parameter values for the different alphabets, as in the previous experiment. The difference now is that we are using the identification scoring metric to assess the results.

What We want to measure the variety in cost values between all possible penalty value pairs. We created color-coded plots to indicate the costs per pair. Again, the gray area corresponds with pairs where MMseqs2 failed to find alignments.

Result As previously, we now evaluate the results per alphabet and subsequently answer our question.

Amino Acid Alphabet When using a custom substitution matrix (Figure 6.8(a)), varying the gap penalties results in a range of cost values of 0.003. With the default BLOSUM62 matrix (Figure 6.8(b)) this range conducts 0.01.

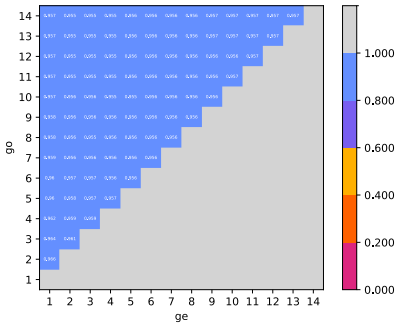


(a) *Amino Acid Alphabet and Custom Substitution Matrix*

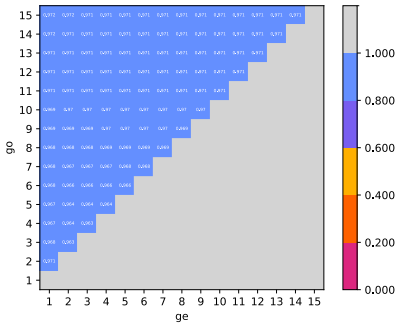
(b) *Amino Acid Alphabet and BLOSUM62 Matrix*

Figure 6.8: *Identification quality per pair of gap penalties for the Amino Acid alphabet with a custom built substitution matrix (a) and the BLOSUM62 matrix (b). The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

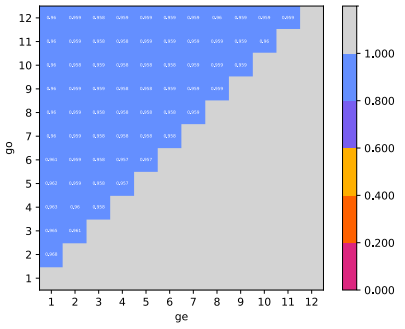
The k-Means approach results in ranges of 0.006 (Figure 6.9(a)), 0.009 (Figure 6.9(b)), and 0.11 (Figure 6.9(c) and (d)).



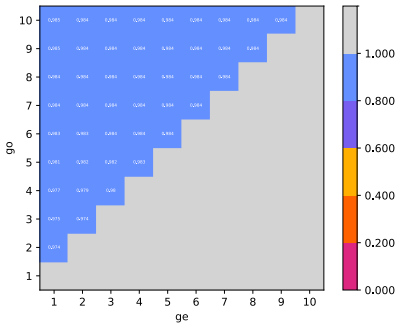
(a) *k-Means Alphabet, Custom Substitution Matrix with k-means++ initialization, and without normalization.*



(b) *k-Means Alphabet, Custom Substitution Matrix with k-means++ initialization, and with normalization.*



(c) *k-Means Alphabet, Custom Substitution Matrix with random initialization, and without normalization.*



(d) *k-Means Alphabet, Custom Substitution Matrix with random initialization, and with normalization.*

Figure 6.9: *Identification quality per pair of gap penalties for the Amino Acid alphabet with a custom built substitution matrix for the k-Means clustering approach. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

3Di Alphabet For the 3Di alphabet the range of cost values comprises 0.03.

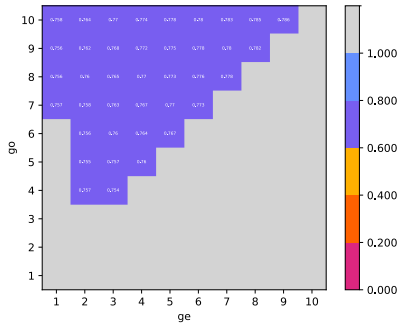


Figure 6.10: *Identification quality per pair of gap penalties for the 3Di alphabet with a custom built substitution matrix. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

VQ-VAE Baseline Alphabet The cost values for the alphabet computed with the *VQ-VAE* network are in a range of 0.024.

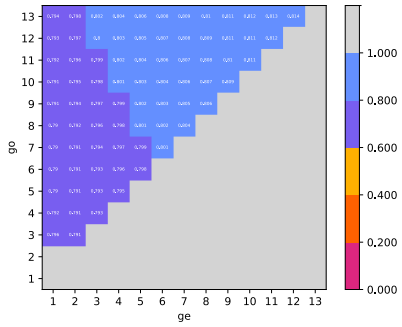


Figure 6.11: Identification quality per pair of gap penalties for the alphabet computed with the baseline configuration of the VQ-VAE network. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.

LM-head Baseline Alphabet The cost values for the alphabet computed with the *LM-head* network are in a range of 0.013.

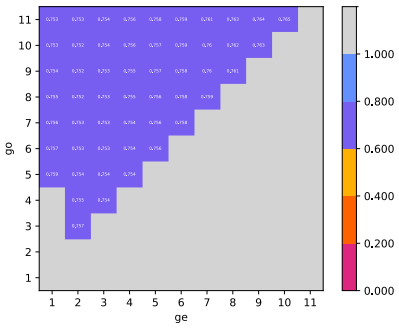


Figure 6.12: *Identification quality per pair of gap penalties for the alphabet computed with the baseline configuration of the LM-head network. The x-Axis contains all possible gap-extension penalty values and the y-Axis the values for the gap-open penalty. The color indicates the cost as represented on the legend on the right side. The lower the cost, the better the result.*

This experiment shows that although the gap penalties influence the alignment quality, they have no significant impact on the identification quality. The cost values vary in a maximum range of 0.11, which is considered not significant, as the cost value can take on values between 0.0 and 1.0. Therefore, it is not necessary to optimize the penalties together with the neural network hyper-parameters regarding the quality of identification. Instead, it is sufficient to optimize the gap penalties for the best found configuration for each of the network models.

6.5 Experiment 2: Parameters to Compute Discrete Embedded Sequences

For the question regarding good hyper-parameter values to compute discrete embedded sequences with the neural network pipelines, we are going to use the scoring metric for the identification because we want to improve the networks in detecting rather than creating alignments.

There are five options for conducting experiments to answer this question. Option one is to add the gap penalties as parameters to optimize. Option two is to run a manual grid search to find the best penalties within each epoch in the training. The problem with these two options is the penalties, and therefore, one specific alphabet influences the training of our neural network. As we want the network to perform well for any alphabet, these options are rejected.

The third option is to do a manual search after each training. However, the slight influence of the penalties on the costs regarding the quality of the identification does not justify this significant computational effort.

A fourth option is to optimize the gap penalties using the Amino Acid and the k-Means Pipeline and using these values as fixed constants in the neural network pipelines. The issue here is that the penalties are dependent on the substitution matrix used. In particular, the possible values depend on the minimum and maximum values present in the matrix. Therefore, these fixed values might be suitable for the matrices computed for the AAP and KMP, but not necessarily for the neural network pipelines.

The fifth, and finally chosen option, is first to optimize the hyper-parameters for the network and then find the best penalties for the configuration with the lowest cost found. With the last experiment, we have shown that the gap penalties do not affect the detection but the alignment quality. As we want to optimize the network's search ability, we first optimize its hyper-parameters based on the identification scoring metric. For the best configuration, we then want to find the best gap penalties based on the alignment quality scoring metric.

With this approach, we can ensure that the alphabet we create with the best configuration can be used with other tools where adaptations of the gap penalties depending on the use case are necessary.

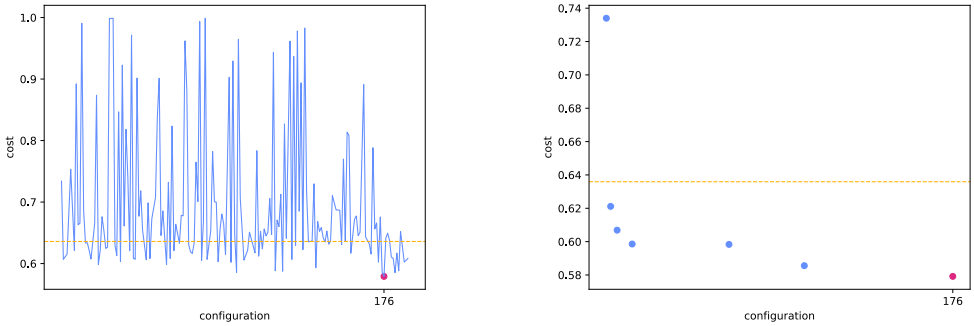
6.5.1 VQ-VAE Neural Network Pipeline

Why We want to find a set of parameter values for the hyper-parameters θ_{VQ-VAE} with a *cost* lower than the baseline *cost* of **0.63592**, which we obtained in Chapter 6.3.

How We start the *VQ-VAE Neural Network* pipeline for 1000 runs. For each run, SMAC samples a different configuration and returns a value for the corresponding *cost*.

What When we reach the time limit or conclude that the *cost* converges, we stop the run and choose the configuration with the lowest *cost* found until this point in time.

Result We stopped the pipeline after 190 trials for time reasons. The configuration with the best *cost* is the one of trial number 176 with a *cost* of **0.57011**. Figure 6.13(a) is a visual representation of the found *cost* values for all explored configurations during the 190 trials, with a red mark for the best one found. The orange line indicates the baseline *cost* we want to underbid. On the right in Figure 6.13(b), we plotted all previous incumbents to illustrate the improvement of our process over time. One trial took on average 7 hours and 52 minutes on one $a100$ GPU node with 40 GB RAM and 60 GB allocated memory. This trial with the lowest *cost* found was done after 11 epochs, as visualized in Figure 6.14. Figure 6.15 illustrates the importance of the hyper-parameters for all logged incumbents (shown in Figure 6.13(b)). The higher the bar, the more important it is to set the hyper-parameter to a value greater than 0. The importance is the ratio of a hyper-parameter being used to the total number of incumbents. For the *VQ-VAE Neural Network*, this means that the hyper-parameters cnQ, us, and dv are the most important and should always be used. Followed by cnD, cb, en, cnE, em, fs, and rcP, in this order.



(a) *Varying cost values for all* 190 *explored configurations.*

(b) *Varying cost values for all incumbent configurations.*

Figure 6.13: *Visual representation of selected hyper-parameter configurations and their corresponding cost of the VQ-VAE Neural Network pipeline run. The cost of the baseline is displayed as a orange horizontal line. The configuration with the lowest cost found is marked with a red dot.*

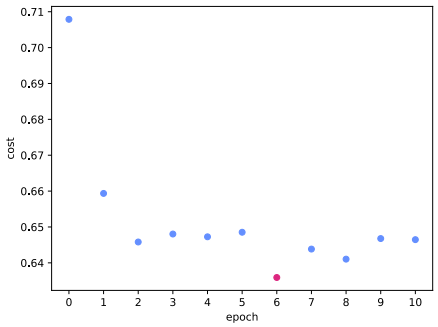


Figure 6.14: *Visual representation of the run with the configuration resulting in the best cost found. This trial stopped after* 11 *epochs where the best cost of* ***0.57011*** *was found in epoch* 6.

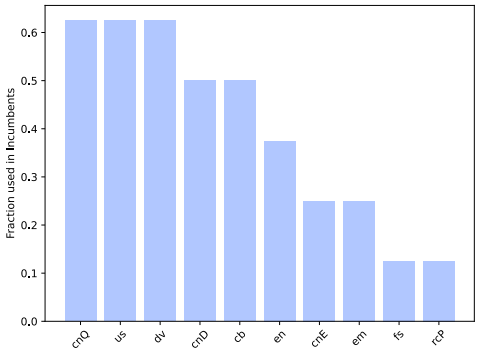


Figure 6.15: *Bar plot of the parameter importance for the VQ-VAE Neural Network regarding the logged incumbents. The higher the bar, the greater the importance.*

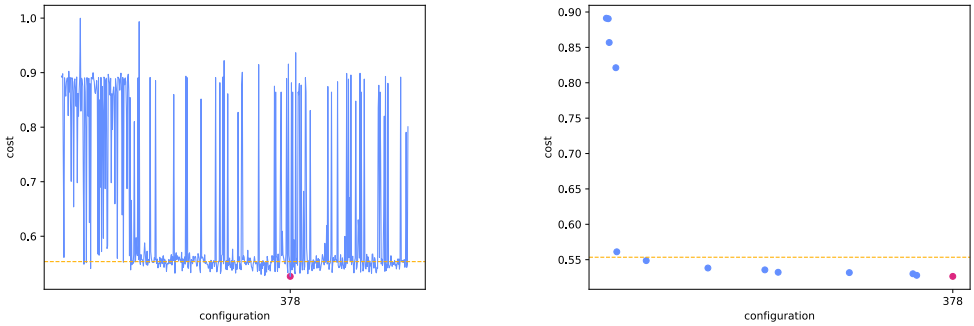
6.5.2 LM-head Neural Network Pipeline

Why We want to find a set of parameter values for the hyper-parameters $\theta_{LM-head}$ with a *cost* lower than the baseline *cost* of **0.55347**.

How We start the *LM-head Neural Network* pipeline for 1000 runs. Each run tries a different configuration and returns the corresponding *cost* value.

What We stop the run and choose the configuration with the lowest *cost* found until this point in time when we reach the time limit or conclude that the *cost* converges.

Result We stopped the pipeline after 570 trials for time reasons. The configuration with the best *cost* is the one of trial number 378 with a *cost* of **0.5264**. The *costs* per explored configuration are visualized in Figure 6.16(a) for all 570 configurations. The configuration with the best *cost* found is marked in red. The baseline *cost* is visualized as an orange horizontal line. Figure 6.16(b) displays the previous incumbents illustrating the improvement of the value *cost* over time. The average time for one trial is 2 hours and 48 minutes on one $a100$ GPU node with 40 GB RAM and 60 GB allocated memory. As indicated in Figure 6.17, the configuration with the lowest *cost* found stopped after 5 epochs. Figure 6.18 illustrates the importance of each hyper-parameter in all incumbents. The importance is calculated for all incumbents as the number of times a hyper-parameter is greater than 0 divided by the total number of incumbents. For the *LM-head Neural Network*, this means that the hyper-parameters *si*, *pos*, and *ss* are the three hyper-parameters influencing the *cost* in a positive way the most. Followed by *us*, *temp*, *neg*, and *en*, in this order.



(a) *Varying cost values for all* 570 *explored configurations.*

(b) *Varying cost values for all incumbent configurations.*

Figure 6.16: *Visual representation of selected hyper-parameter configurations and their corresponding cost of the LM-head Neural Network pipeline run. The cost of the baseline is displayed as a orange horizontal line. The configuration with the lowest cost of* ***0.5264*** *found is marked with a red dot.*

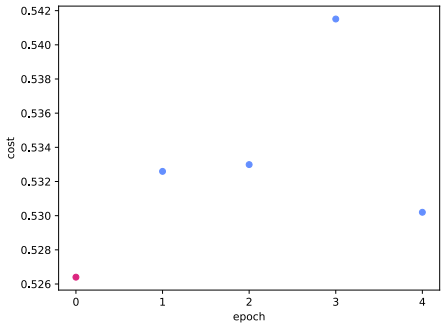


Figure 6.17: *Visual representation of the run with the configuration resulting in the best cost found of* ***0.5264****. This trial stopped after* 5 *epochs where the best cost was found in epoch* 0*.*

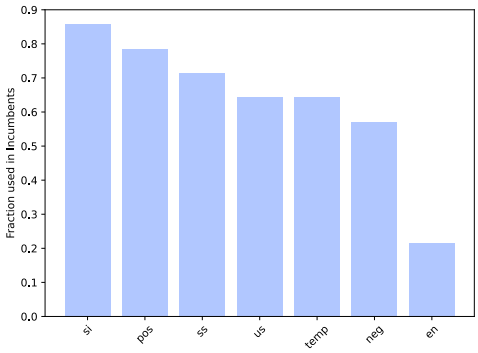


Figure 6.18: *Bar plot of the parameter importance for the LM-head Neural Network regarding the logged incumbents. The higher the bar, the greater the importance.*

6.6 Experiment 3: Influence of Gap Penalties after Optimization

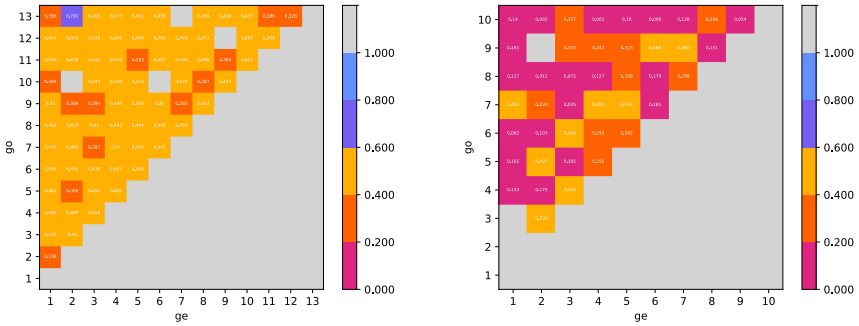
Why We have shown that the selection of gap penalties influences the alignment quality. With Experiment 2 (Chapter 6.5), we found a configuration for each of the neural networks with a *cost* lower than the baseline *cost*. For the alphabets resulting from the usage of these configurations, we want to identify the values for *go* and *ge* optimized for alignment quality. In addition, we want to check whether our hypothesis that gap penalties have a negligible impact on the identification task holds with the new alphabets.

How For both, the *LM-head* and *VQ-VAE* neural network models with optimal configuration as identified in Experiment 2, we repeated the steps of Experiment 1 (Chapter 6.4) to identify the respective optimal gap penalties.

What As in Experiment 1, we visualize the *cost* values for each possible pair of penalties. Additionally, we plot the identification quality of family, superfamily, and fold for both models using optimized penalty settings, and compare these results with the MMseqs2 default settings. This illustration represents the quality, which is the area under the curve (defined in Chapter 4.2). A large area under the curve corresponds to a good quality and therefore a low cost.

Result The gap penalty pair of $go = 9$ and $ge = 7$ (and $go = 11$ and $ge = 5$) results in the lowest alignment *cost* of **0.283** for the *VQ-VAE Neural Network* alphabet (see Figure 6.19(a)). For the *LM-head Neural Network* alphabet, the gap penalty pair of $go = 10$ and $ge = 6$ achieves the best result with an alignment *cost* of **0.006** (see Figure 6.19(b)).

The search costs for the all-vs-all alignment using the best configuration found in Experiment 2 with the default penalties of MMseqs2 are **0.61175** for *VQ-VAE* and **0.55059** for the *LM-head*. Using the optimized penalties, the search costs conduct **0.63721** for *VQ-VAE* and **0.56579** for *LM-head*. The slight differences again confirm that gap penalties do not have a significant impact on the cost of identification. This difference is visualized in Figure 6.20 and Figure 6.21.



(a) *VQ-VAE Neural Network alphabet: pair of penalty values with the lowest cost is go = 9 and ge = 7 (and go = 11 and ge = 5) with cost =* ***0.283****.*

(b) *LM-head Neural Network alphabet: pair of penalty values with the lowest cost is go = 10 and ge = 6 with cost =* ***0.006****.*

Figure 6.19: *Color-coded grids visualizing the range of cost values per pair of go and ge values for the alphabets computed with the best configuration found per neural network.*

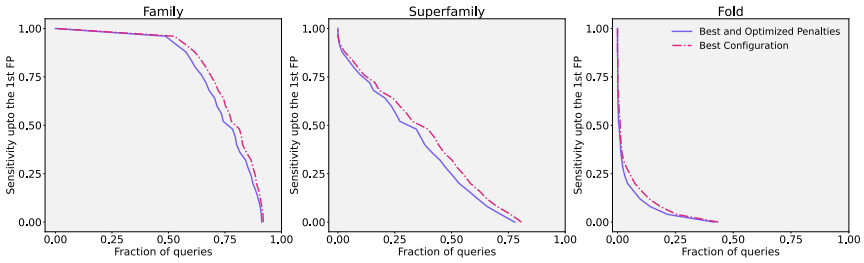


Figure 6.20: *Visualization of the difference in identification quality for the VQ-VAE Neural Network with optimized configuration and optimized penalties in purple, in comparison to using default penalties from MMseqs2 in red.*

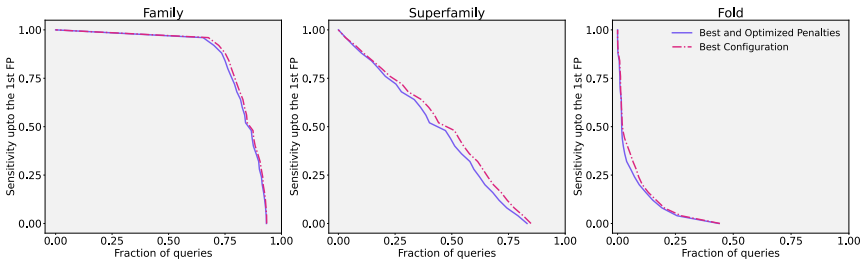


Figure 6.21: *Visualization of the difference in identification quality for the LM-head Neural Network with optimized configuration and optimized penalties in purple, in comparison to using default penalties from MMseqs2 in red.*

Chapter 7

Conclusions

In this thesis, we present a method to tune the hyper-parameters for two neural networks that are used to compute discrete representations of continuous Protein Language Model embeddings of protein sequences. To achieve this, we develop optimization pipelines and scoring metrics to assess their results systematically. We incorporate resource awareness by setting the patience of the neural networks to the minimum value of four epochs and using only a representative 10% of the available data for the evaluation process. In addition, we considered the role of gap penalties in the optimization process.

The first experiment in Chapter 6.4 demonstrated that the choice of gap penalties does influence the quality of alignments, but has no significant effect on the identification quality. This observation shaped the selection of hyper-parameters to be optimized within our optimization pipelines.

With the second experiment in Chapter 6.5, we identified good hyper-parameter configurations for both neural networks. As visualized in Figure 7.1 and Figure 7.2, these configurations improved the identification quality compared to the baseline setup. The plots in the two figures illustrate the identification quality metric for family, superfamily, and fold separately. A rectangle corresponds to an area under the curve of 1 and a *cost* of 0 respectively.

For the *LM-head Neural Network* we achieved a cost reduction of 0.02707 (3%). This improvement is also visible due to the slightly larger area under the red curve compared to the yellow curve of the *LM-head* baseline.

For the *VQ-VAE Neural Network* we achieved an even greater improvement of 0.06581 (7%). Likewise, this improvement is observable due to the larger area under the red curve compared to the yellow curve of the *VQ-VAE* baseline.

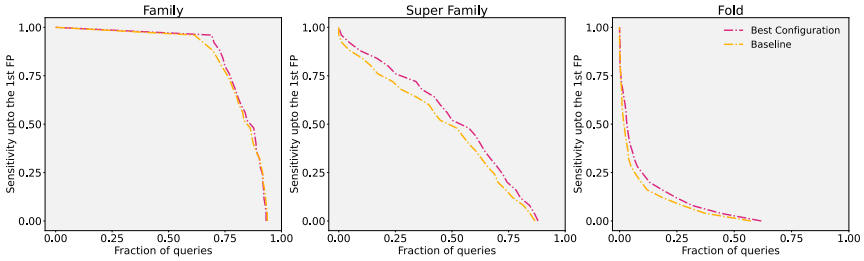


Figure 7.1: *Identification quality of the best configuration found for the LM-head Neural Network in red compared to the LM-head baseline in yellow. A larger area under the curve corresponds to superior identification.*

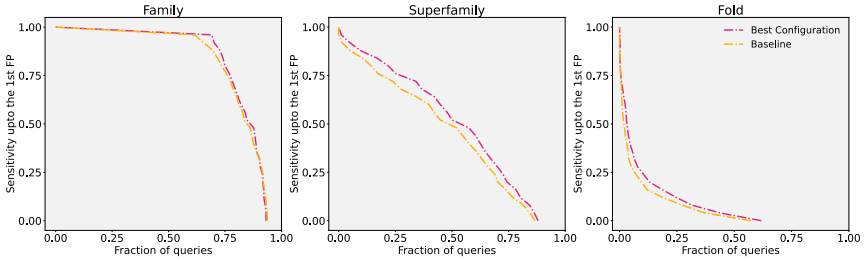


Figure 7.2: *Identification quality of the best configuration found for the VQ-VAE Neural Network in red compared to the VQ-VAE baseline in yellow. A larger area under the curve corresponds to superior identification.*

The minor difference for the *LM-head Neural Network* allows us to conclude that this network seems more robust to hyper-parameter values than the *VQ-VAE Neural Network*, where more improvement with respect to the baseline could be obtained. For the *VQ-VAE Neural Network*, this highlights the value of optimization for larger parameter spaces.

These results can be reproduced with our setup. Furthermore, our results highlight which hyper-parameters do have the most significant influence on performance. Interestingly, fewer experiments would have been sufficient for the *LM-head Neural Network*, since no further cost reduction was observed for the last 192 trials. For the *VQ-VAE Neural Network*, on the other hand, we cannot make the same conclusion, since the number of trials for which there was no further improvement is only 14.

While our method demonstrates promising results, we are facing practical constraints. The optimization process relies on the performance and availability of the compute cluster, influencing experimental throughput. Moreover, we used fixed databases, which provided consistency and comparison, but may limit the generalizability of our results. We computed the scoring metrics on a reduced subset of available data to ensure resource efficiency at the expense of a potential bias. Finally, we employed the hyper-parameter optimization

framework SMAC3, which yielded good results. However, exploring alternative frameworks could further improve the performance.

Chapter 8

Outlook

Building on our findings, future work could focus on utilizing the finite alphabets obtained with the best configurations to translate entire databases of protein sequences, thereby enabling more effective large-scale search. In addition, the parameter importance analysis offers the opportunity to restrict the optimization process to a smaller subset of hyper-parameters, excluding those with negligible impact. Using a smaller set of hyper-parameters would potentially shorten the number of runs needed to find further performance improvements.

We declare that we used Grammarly [47] to detect grammatical mistakes. ChatGPT version 5.0 [48] was used to identify missing clarity in the spelling style. We revised only existing text; no new content was generated.

Bibliography

[1] J. Chen, M. Guo, X. Wang, and B. Liu, "A comprehensive review and comparison of different computational methods for protein remote homology detection," *Briefings in Bioinformatics*, vol. 19, no. 2, pp. 231–244, 2018.

[2] J. Söding and M. Remmert, "Protein sequence comparison and fold recognition: progress and good-practice benchmarking," *Current Opinion in Structural Biology*, vol. 21, no. 3, pp. 404–411, 2011.

[3] L. Pantolini, G. Studer, J. Pereira, J. Durairaj, G. Tauriello, and T. Schwede, "Embedding-based alignment: combining protein language models with dynamic programming alignment to detect structural similarities in the twilight-zone," *Bioinformatics*, vol. 40, 2024.

[4] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D. J. Lipman, "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, pp. 403-410, 1990.

[6] M. Remmert, A. Biegert, A. Hauser, and J. Söding, "HHblits: lightning-fast iterative protein sequence searching by HMM-HMM alignment," *Nature Methods*, vol. 9, no. 2, pp. 173–175, 2012.

[7] J. Söding, "Protein homology detection by HMM–HMM comparison," *Bioinformatics*, vol. 21, no. 7, pp. 951–960, 2005.

[8] M. Steinegger and J. Söding, "MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets," *Nature Biotechnology*, vol. 35, no. 11, pp. 1026–1028, 2017.

[9] L. Pantolini, “Unmasking hidden protein relations through representation learning,” *Ph.D. dissertation, Philosophisch-Naturwissenschaftlichen Fakultät der Universität Basel*, Basel, 2024.

[10] S. Falkner, M. Lindauer, and F. Hutter, "SpySMAC: Automated Configuration and Performance Analysis of SAT Solvers," in *Proceedings of the International Conference on Satisfiability Solving (SAT'15)*, pp. 1-8, 2015.

[11] J. Seipp, S. Sievers, and F. Hutter, “Fast Downward SMAC,” *Planner Abstract, IPC*, 2014.

[12] E. Gomede, F. Silva, M. Mendonça, and R. H. C. Palácios, “Bayesian Optimization for Sampling the Hyper Parameter Space,” preprint, 2023.

[13] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5-32, 2001.

[14] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, and D. Deng, “SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization,” *Journal of Machine Learning Research*, vol. 23, pp. 1–9, 2022.

[15] T. Sanvictores and F. Farci, "Biochemistry, Primary Protein Structure," *StatPearls Publishing*, 2020.

[16] N. Ferruz and B. Höcker, "Controllable protein design with language models," *Nature Machine Intelligence*, vol. 4, pp. 521-532, 2022.

[17] D. L. Nelson and M. M. Cox, *Lehninger Principles of Biochemistry*, 6th ed. New York: W. H. Freeman and Company, 2013.

[18] A. Andreeva, E. Kulesha, J. Gough, and A. G. Murzin, “The SCOP database in 2020: expanded classification of representative family and superfamily domains of known protein structures,” *Nucleic Acids Research*, vol. 48, no. D1, pp. D376–D382, 2020.

[19] J. Yerushalmy J, "Statistical problems in assessing methods of medical diagnosis with special reference to x-ray techniques," *Public Health Reports*, vol. 62, no. 2, pp. 1432–1449, 1947.

[20] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 89, pp. 10915-10919, 1992.

[21] M. Celik, X. Xie, "Efficient Inference, Training, and Fine-tuning of Protein Language Models," bioRxiv preprint, 2024.

[22] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential Model-Based Optimization for General Algorithm Configuration," in *Proceedings of the conference on Learning and Intelligent OptimizatioN (LION 5)*, pp. 507-523, 2011.

[23] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown, "The configurable SAT solver challenge (CSSC)," *Artificial Intelligence*, vol. 243. pp. 1–25, 2017.

[24] M. Helmert, "The Fast Downward Planning System," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[25] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[26] A. Ben-Hur and D. Brutlag, "Sequence Motifs: Highly Predictive Features of Protein Function," in *Feature Extraction*, vol. 207, I. Guyon, M. Nikravesh, S. Gunn, and L. A. Zadeh, Eds., in *Studies in Fuzziness and Soft Computing*, vol. 207. Springer, 2006, pp. 625–645.

[27] S. R. Eddy, "A new generation of homology search tools based on probabilistic inference," *Genome Informatics 2009*, vol. 23, pp. 205–211, 2009.

[28] T. Hankeln, "Alignment von DNA- und Proteinsequenzen," in *Einführung in Methoden der Bioinformatik*, Universität Mainz, Mainz, 2018.

[29] R. D. Finn, J. Clements, and S. R. Eddy, "HMMER web server: interactive sequence similarity searching," *Nucleic Acids Research*, vol. 39, pp. W29–W37, 2011.

[30] C. D. McWhite, I. Armour-Garb, and M. Singh, "Leveraging protein language models for accurate multiple sequence alignments," *Genome Research*, vol. 33, pp. 1145-1153, 2023.

[31] Z. Lin, H. Akin, R. Rao, B. Hie, Z. Zhu, W. Lu, N. Smetanin, R. Verkuil, O. Kabeli, Y. Shmueli, A. d. S. Costa, M. Fazel-Zarandi, T. Sercu, S. Candido, A. Rives, "Evolutionary-scale prediction of atomic-level protein structure with a language model," *Science*, vol. 379, pp. 1123-1130, Mar. 2023.

[32] K. Kaminski, J. Ludwiczak, K. Pawlicki, V. Alva, and S. Dunin-Horkawicz, "pLM-BLAST: distant homology detection based on direct comparison of sequence representations from protein language models," *Bioinformatics*, vol. 39, no. 10, 2023.

[33] A. K. Jain, M. N. Murty, P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[34] M. Ahmed, R. Seraj, and S. M. S. Islam, "The k-means Algorithm: A Comprehensive Survey and Performance Evaluation," *Electronics*, vol. 9, no. 8, pp. 1295, 2020.

[35] 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, E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825-2830, 2011.

[36] A. v. d. Oord, O. Vinyals, and K. Kavukcuoglu, "Neural Discrete Representation Learning," *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[37] M. Van Kempen, S. S. Kim, C. Tumescheit, M. Mirdita, J. Lee, C. L. M. Gilchrist, J. Söding, and M. Steinegger, "Fast and accurate protein structure search with Foldseek," *Nature Biotechnology*, vol. 42, pp. 243–246, 2024.

[38] A. Rives, J. Meier, T. Sercu, S. Goyal, Z. Lin, J. Liu, D. Guo, M. Ott, C. L. Zitnick, J. Ma, and R. Fergus, "Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 118, no. 15, 2021.

[39] N. K. Fox, S. E. Brenner, J. M. Chandonia, "SCOPe: Structural Classification of Proteins — extended, integrating SCOP and ASTRAL data and classification of new structures," *Nucleic Acids Research*, vol. 42, pp. D304 - D309, 2014.

[40] J. Mistry, S. Chuguransky, L. Williams, M. Qureshi, G.A. Salazar, "Pfam: The protein families database in 2021," *Nucleic Acids Research*, vol. 49, no. D1, pp. D412–D419, 2021.

[41] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," *29th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), 2024.

[42] R. Trivedi, HA. Nagarajaram, "Substitution scoring matrices for proteins - An overview," *Protein Science*, vol. 29, no. 11, pp. 2150-2163, 2020.

[43] L. M. Engist, "CMA - Custom Substitution Matrices for sensitive sequence alignments with large language models," Project Report, Biozentrum Universität Basel, Basel. [Online]. Available: https://wiki.biozentrum.unibas.ch/x/MhlNGw. Accessed 03 Feb. 2025.

[44] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, "Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids," *Cambridge University Press*, 1998.

[45] M. Vingron and M. S. Waterman, "Sequence Alignment and Penalty Choice Review of Concepts, Case Studies and Implications," *Journal of Molecular Biology*, vol. 235, pp. 1–12, 1994.

[46] A. Elnaggar, M. Heinzinger, C. Dallago, G. Rehawi, Y. Wang, "ProtTrans: Toward Understanding the Language of Life Through Self-Supervised Learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 10, pp. 7112–7127, 2022.

[47] Grammarly, Inc, Grammarly, https://www.grammarly.com, 2025.

[48] OpenAI, ChatGPT (Version 5.0 August 2025). https://chat.openai.com/chat, 2023.