Improving Reliability of Shotgun Proteomics with Machine Learning

Jakub Mosiński[1]

MSc Data Science and Machine Learning

Supervisors: Mark Herbster, Karim Malki

Submission date: 13 September 2021

Abstract

The aim of this thesis is to improve the reliability of shotgun proteomics. Robust quantification of proteins is vital for understanding diseases and drug discovery. The large scale study of proteins suffers from yet not fully understood problems in accurate measurement of proteins. The thesis addresses the peptide detectability prediction as well as investigates the reasons for the poor reproducibility of proteomics experiments. It includes the background on deep learning for sequential modelling and a literature review of successful machine learning methodologies for proteomics data. Here we present an end-to-end ensemble of recurrent neural networks that can predict peptide detectability with high accuracy based on amino acid sequences. Our model outperforms other state-of-the-art methods on homo sapiens and mus musculus benchmark data sets. We also show that the model generalizes well across different species. Moreover, we disprove the hypothesis that the peptide distance from the protein centre impacts the reproducibility of shotgun proteomics experiments. The thesis also includes the comparison of different peptide encoding methods and connects peptide retention time to the reproducibility of mass spectrometry results. The source code and models are available at `https://github.com/jmosinski/MSc-Thesis`.

Acknowledgements

I would like to thank Mark Herbster, Karim Malki, Nico Kist for their support, encouragement and guidance. I especially thank Nico Kist for being readily available to discuss my work. I would also like to thank Michael MacCoss for his proposed hypothesis.

Contents

Chapter 1

Introduction

1.1 What are Proteins and How to Study Them

Proteins are vital parts of living organisms. They are responsible for transporting nutrients (Hemoglobin), detecting and neutralizing harmful bacteria or viruses (Immunoglobin), building the structure of organisms (Collagen). We can define proteins as amino acid (residue) sequences of length ranging from hundreds to thousands. We can also distinguish shorter amino acid sequences called peptides built from several up to tens of amino acids. There are twenty amino acids found in human proteins comprised of Carbon, Oxygen, Nitrogen, Hydrogen or Sulfur atoms. They can interact well with water when hydrophobic or poorly when hydrophilic. Also, charged amino acids interact strongly with oppositely charged amino acids. These chemo-physical properties define how the chain of amino acids folds into a complex 3D structure. That unique shape determines the properties and function of a protein. We can distinguish four structural representations of proteins presented in Figure 1.1.

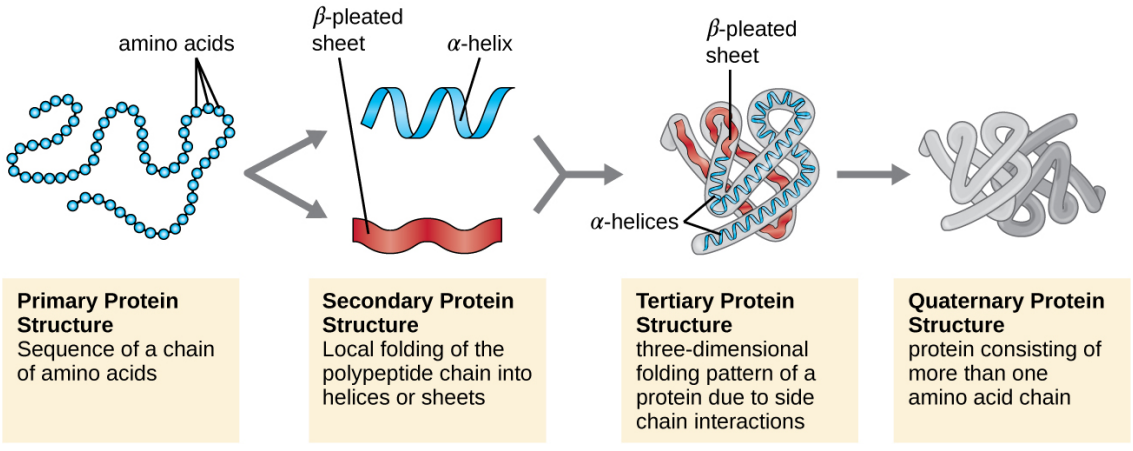


Figure 1.1: Protein structures [1].

To study proteins, we need to quantify them. Proteomics is a large-scale study of proteins. One of the main protein analysis methods is shotgun proteomics, see Figure 1.2. First, proteins are extracted from the cells or tissues. Then they are enzymatically digested to acquire peptides.

The peptide mixtures consist of different compounds. If we were to pass the mixture to the mass spectrometer, the resulting spectrum would be too complex to extrapolate compounds of interest. Therefore, we use high-performance liquid chromatography (HPLC) to separate peptides by physical properties, such as polarity or molecular size. HPLC dissolves the mixture in a liquid and passes through a column through which different compounds have varying travel times, also referred to as retention times. Then the unique peptides are processed by a mass spectrometry machine to acquire the peptide abundance against the mass to charge ratio (m/z). Finally, we can infer the peptides from the spectrum referencing known databases.

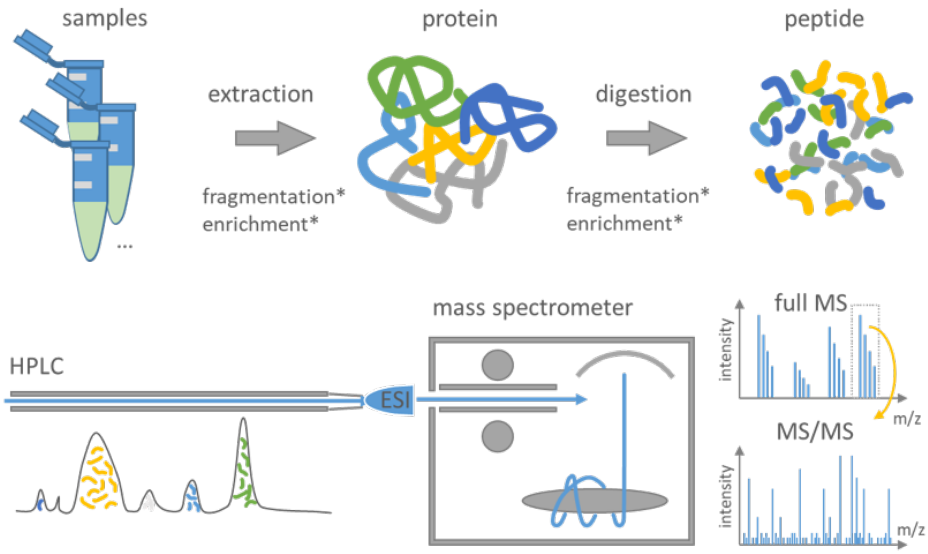


Figure 1.2: General workflow of shotgun proteomics [2].

1.2 Motivation

Robust quantification and detection of proteins is vital for understanding diseases and designing better drugs. Advanced mass spectrometry methods allow for the detection of protein subsequences called peptides. However, it is still challenging to acquire accurate measurements due to the stochastic nature of the peptide selection, presence of degenerated peptides or difficulties in the statistical analysis of the results. To increase the accuracy of the results, we could only consider peptides that we know are easily detected. For that reason, we need methods for the accurate classification of detectable peptides. We will try to improve the existing solutions for peptide detectability prediction employing deep learning techniques operating only on the peptide amino acid sequence. Furthermore, the study concerning the reproducibility of MS results conducted at UCB Pharma showed that the detections for some peptides correlate weakly between different mass spectrometers. It raises a concern about the trust in the MS results. The thesis will provide the first insights into the problem using statistical modelling.

1.3 Structure of the Thesis

This thesis is organized in the following chapters. Chapter 2 provides a background to deep learning, focusing on architectures for sequential modelling followed by the literature overview of machine learning in proteomics. Chapter 3 introduces the data sets of interest. Chapter 4 addresses the problem of peptide detectability prediction explaining the modelling scheme and results. Chapter 5 is devoted to the reproducibility of mass spectrometry experiments. It includes testing the hypothesis of the field expert, statistical modelling of the problem determining possible links between peptide properties and reproducibility of MS results. Finally, Chapter 6 provides conclusions and possible directions for further research.

Chapter 2

Background and Related Work

In this chapter we briefly introduce the main concepts behind artificial neural networks, specifically focusing on architectures suitable for modelling sequential data. We also discuss common regularization techniques as well as gradient descent and hyperparameter search algorithms. Following, we exhibit successful machine learning practicies for proteomics data.

2.1 Artificial Neural Networks

The origin of contemporary neural network architectures dates back to 1943 when Warren McCulloch and Walter Pitts created a computational model for neural networks based on threshold logic algorithms. Following, in 1958, Rosenblatt introduced the perceptron [3] and the first multi-layer perceptron occurred in 1965 published by Ivakhenko and Lapa [4]. However, the algorithms were slow to train as they were not fully differentiable. That problem was solved in 1986 when Rumelhart, Williams, and Hinton applied a backpropagation algorithm to neural networks and argued that they could provide "interesting" distribution representations [5]. Not long after, the interest in the field of artificial intelligence diminished due to over-confident claims about the potential of the neurally inspired algorithms losing investor's interest. Despite the bad reputation, some scientists continued to work in the field. In 1989, Yann LeCun provided the first practical application of convolutional neural networks. He used the backpropagation algorithm for efficient network training on the handwritten digits recognition problem. The system was later used for processing checks. In 1995 Dana Cortes and Vladimir Vapnik developed the support vector machine [6], and in 1997 Sepp Horchreiter and Juergen Schmidhuber introduce long short-term memory cells (LSTM) [7]. The rebirth of the field started in the early 20th century with the advancements in computational solutions such as developing graphical processing units (GPU), making the training of neural networks more practical. The neural networks were still slower than support vectors machines but could offer better results with the same data. A breakthrough for deep learning happened in 2012, when a convolutional neural network - AlexNet, won the ImageNet competition outperforming other solutions by over ten percentage points [8]. From that point, neural networks continued to set new standards in data-rich problems.

2.1.1 Feedforward Neural Networks

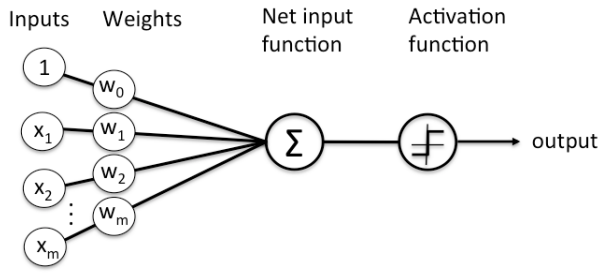


Figure 2.1: Single neuron [9].

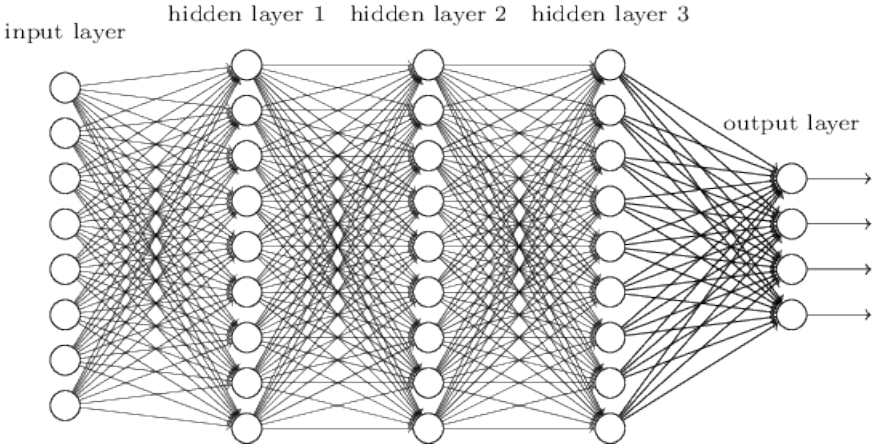


Figure 2.2: Multi-layer neural network [10].

The main building block of a feedforward neural network is a single neuron, see image 2.1. It dot multiplies inputs with weights and applies an activation function. In 1989, Cybenko argued that if we stack together enough neurons, we could approximate an arbitrary function [11]. For that to hold, we need to use non-linear activation functions. Otherwise, we could only represent linear hypothesis spaces. However promising that may sound, the dimensionality of a single layer neural network would often make it impractical to use. A more efficient solution would be to not only stack neurons but also layers to create a multi-layer neural network, see Figure 2.2. Intuitively, we could view that as later layers can find better data representations by leveraging outputs from prior layers. In the context of image face recognition, a deep neural network at first represents the input image in terms of edges and corners. Then it combines these edges to build face descriptions useful for final classification.

The feedforward algorithm is simply a matrix multiplication of input vector with layer weights followed by an activation function. Below is a mathematical expression of a forward pass for a

neural network in Figure 2.2.

$$h_1 = g_1(W_1 x + b_1) \tag{2.1}$$
$$h_2 = g_2(W_2 h_1 + b_2) \tag{2.2}$$
$$h_3 = g_3(W_3 h_2 + b_3) \tag{2.3}$$
$$out = g_4(W_4 h_3 + b_4) \tag{2.4}$$
$$\tag{2.5}$$

where

$x -$ input column vector,

$W_i \in \mathbb{R}^{\text{output}_i \times \text{input}_i} -$ weight matrix of layer i,

$b_i -$ bias of layer i,

$g_i() -$ activation function of layer i.

Some of the commonly used activation functions include:

$$\textbf{Sigmoid: } y = \frac{1}{1 - e^{-x}} \tag{2.6}$$
$$\textbf{SoftPlus: } y = \log(1 + e^x) \tag{2.7}$$
$$\textbf{Tanh: } y = \tanh(x) \tag{2.8}$$
$$\textbf{ReLU: } y = \max(0, x) \tag{2.9}$$
$$\textbf{LeakyReLU: } y = \begin{cases} x, \text{ if } x > 0 \\ 0.01z, \text{ otherwise} \end{cases} \tag{2.10}$$
$$\textbf{ELU: } y = \begin{cases} x, \text{ if } x > 0 \\ \alpha(e^x - 1), \text{ otherwise} \end{cases} \tag{2.11}$$

To train a neural network, we can use a backpropagation algorithm. However, first, we need to define an optimization objective. It may be a cross-entropy loss for classification or mean squared error for regression problems. The backpropagation algorithm computes partial derivatives of the loss function with respect to the network's weights. We can derive the gradients simply by applying the chain rule, starting at the loss function. Nowadays, we most often use automatic gradient differentiation software implemented in libraries such as PyTorch [12], which is the choice in this thesis. It creates a computation graph that can efficiently reuse previous calculations, also limiting implementation errors. Later, we can use a gradient descent algorithm to move toward a minimum updating the parameters accordingly. More details about the gradient descent algorithms are present in section 2.1.8 Gradient Descent Algorithms.

2.1.2 Convolutional Neural Networks

Convolutional neural networks (CNN) arose from the necessity of models with better sample complexity for image processing. It grows linearly with the number of pixels in an image for a

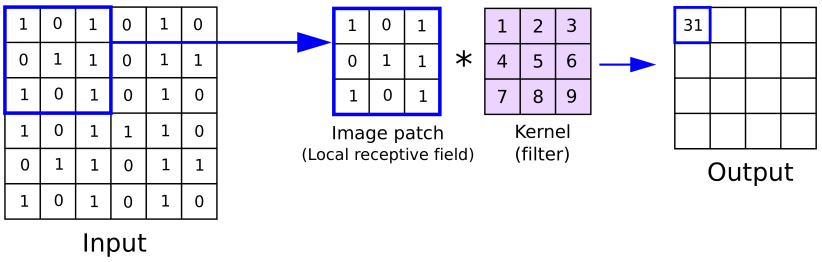


Figure 2.3: Cross-correlation of an image patch with a filter [13].

fully connected architecture resulting in poor data efficiency. CNNs leverage the assumption that the same features are present in different parts of an image (ex. edges). That way, we can introduce weight sharing and drastically improve the sample efficiency of the network. Convolutional neural networks learn appropriate filters to convolve with an input signal. However, in practice, instead of convolution, we apply correlation. The difference is that convolution first rotates the filter by 180 degrees and then applies correlation. The inversion operation has important mathematical properties but is meaningless for neural networks and would only slow down the algorithm. In Figure 2.3 we can see how the filter is correlated with an image patch to produce the output. The mathematical formulation of cross-correlation is as follows

$$O_{m,l} = \sum_{i=-M}^{M} \sum_{j=-L}^{L} K_{i,j} I_{m+i,l+j} \tag{2.12}$$

where O is output, K denotes a kernel and I is an input image of shape $M \times L$.

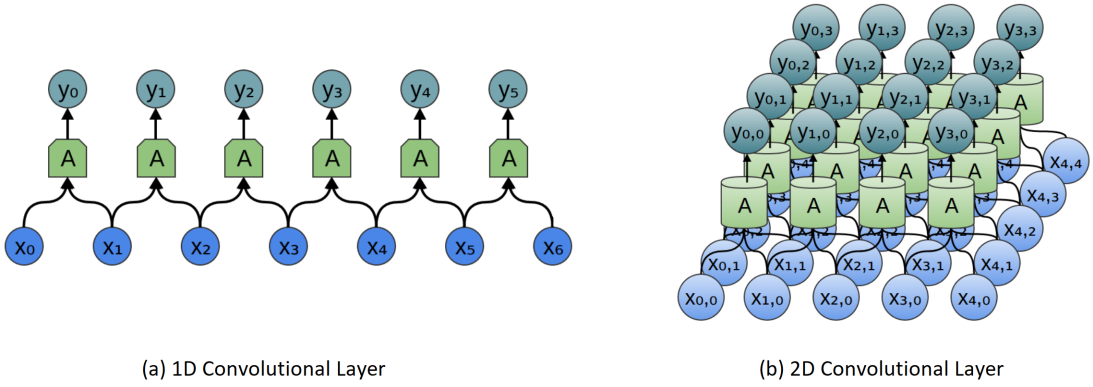


Figure 2.4: Convolutional layers [14].

Alternatively, we can view a convolutional layer as a network with weight sharing as depicted in Figure 2.4. Considering the 1D convolutional layer, module A is a feedforward neural network applied in a sliding window fashion to the input sequence.

$$y_i = g(x_0 w_1 + x_1 w_2 + b) \tag{2.13}$$

Where g is an activation function, w is a fully connected network's weight, and b is a bias term. Note that the sequence length is reduced by unity. We can easily counter that effect with padding. Meaning that we pad the sequence with zeros on both sides. We calculate the padding size as follows

$$p = \text{ceil}\Big(\frac{k-1}{2}\Big) \tag{2.14}$$

where p is padding size, and k is the kernel size. It also generalizes to the 2D correlation, but then we pad the image borders accordingly.

CNNs have significant implementation advantages. The correlation operation can be performed in $O(n\log(n))$. What is more, we can parallelize the computations and make use of modern GPUs.

2.1.3 Recurrent Neural Networks

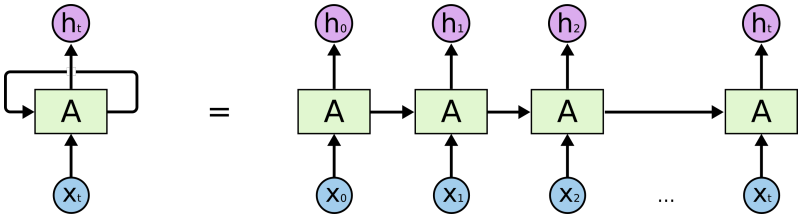


Figure 2.5: Unrolled recurrent neural network [15].

In proteomics, we often want to find sequential dependencies in amino acid chains. Dense neural networks are limited in the sense that they can not efficiently capture this kind of relationship. They can process one input at a time, unconditioned on the past. Therefore, recurrent neural networks are often the choice for modelling such data. RNNs have loops that recursively propagate information through the network. The basic RNN cell takes as input features x_t and previous hidden state h_{t-1}. Then outputs a new hidden state h_t and uses it for the next time step. If we unroll the recurrent neural network as in Figure 2.5, we can view it as multiple copies of the same network, each passing a message to a successor. The weight sharing results in an increased sample efficiency compared to feedforward networks when working with sequential data. Another advantage is that we can process sequences of varying lengths. However, due to the sequential nature of the network, we can no longer optimize the model with standard backpropagation. Instead Werbos proposed backpropagation through time (BBTM) [16]. The main idea of the algorithm is to unroll the Recurrent network and treat it as a sequence of feedforward networks with the same weights. Later calculate gradients for each time step to next sum them and update the weights accordingly. BBTM allows for efficient training of RNNs, but it has difficulty with local optima [17]. Recurrent feedback creates chaotic responses in the error surface, causing local optima to occur more frequently than in feedforward networks.

The simplest hidden state calculation looks as follows:

$$h_t = \tanh(W_i x_t + b_i + W_h h_{t-1} + b_h) \tag{2.15}$$

where

W_i – weight matrix of input layer

b_i – bias of input layer

W_h – weight matrix of hidden layer

b_h – bias of hidden layer

The recurrent neural networks can also suffer from vanishing and exploding gradients. It happens due to the gradient multiplication in the recurrent cells, causing them to either exponentially vanish or explode. Vanishing gradients cause information degradation from previous time steps, making it hard to capture long term dependencies. For that reason, other recurrent cells introduce mechanisms that create a notion of memory. We discuss them in detail in the subsequent sections. To deal with exploding gradients, we manually cap the gradient's value. It is called gradient clipping and works well in practice.

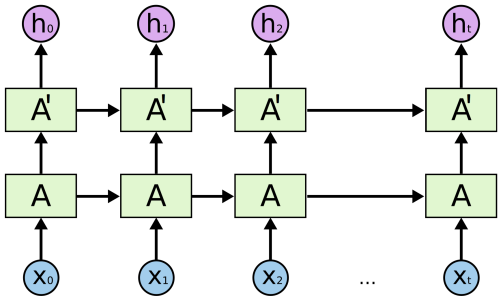


Figure 2.6: Multi-layer recurrent neural network [15].

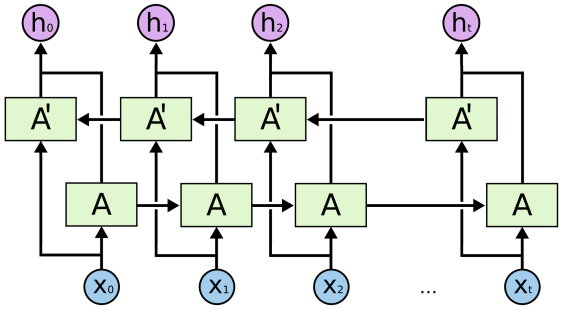


Figure 2.7: Bi-directional recurrent neural network [15].

We can also create multi-layer recurrent neural networks by treating hidden states as the sequence of inputs to another RNN layer, see Figure 2.6. Also, when working with a protein sequence, we would like to find amino acid representations conditioned not only on the prior amino acids but also the subsequent. For that purpose, we could use a bidirectional recurrent neural network (BiRNN), see Figure 2.7. A BiRNN consist of two recurrent networks, forward and backward. The forward network outputs hidden states conditioned on the prior sequence inputs. The backward network takes the inverted input sequence. Later, those two representations are

concatenated to output the final hidden states.

2.1.4 Long Short-Term Memory

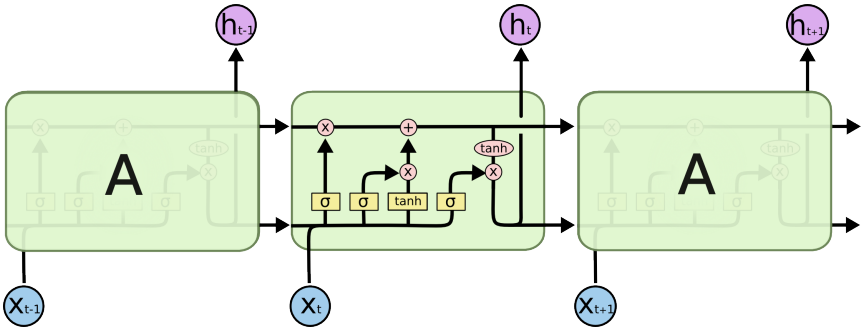


Figure 2.8: LSTM network [15].

The Long Short-Term Memory, or LSTM in short, was introduced by Horchreiter and Schmidhuber in 1997 [7]. It explicitly addresses the problem of vanishing gradients. Authors propose a cell state C_t that acts as a memory. Then, three gates regulate the flow of information to the cell state, see Figure 2.8. In the following part, we will go through each gate and mechanism of an LSTM cell.

Forget Gate

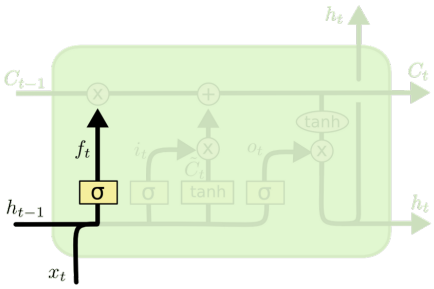


Figure 2.9: LSTM forget gate [15].

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.16}$$

First, the forget gate decides how much of the previous information we should forget, see Figure 2.9. It takes the previous hidden state h_{t-1} and current input x_t. Then applies sigmoid function σ that returns output f_t between 0 (forget) and 1 (keep).

Input Gate

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.17}$$

$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{2.18}$$

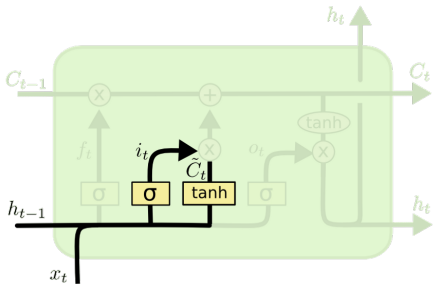


Figure 2.10: LSTM input gate [15].

The input gate shown in Figure 2.10 controls how to update the cell state with new information. The sigmoid layer σ determines which values of previous hidden state h_{t-1} and current input x_t will be updated by returning the value i_t between 0 (do not update) and 1 (update). Then the tanh gives weightage to the passed values and returns the level of importance $\tilde{C}_t$ ranging from -1 to 1.

Cell State Update

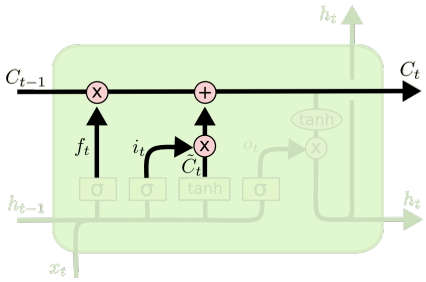


Figure 2.11: LSTM cell state update [15].

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{2.19}$$

At this stage, we multiply the old cell state with the output from the forget gate f_t to down-weight irrelevant information. We also impute new information by adding $\tilde{C}_t$ modulated with i_t. The procedure results in the new cell state C_t that we pass to the subsequent cell. Please see Figure 2.11 for the cell state update diagram.

Output Gate

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.20}$$

$$h_t = o_t \odot \tanh(C_t) \tag{2.21}$$

Finally, the output gate decides what the next hidden state should be, see Figure 2.12. It applies sigmoid function σ to resolve which states to let through o_t and tanh regularizes the current cell

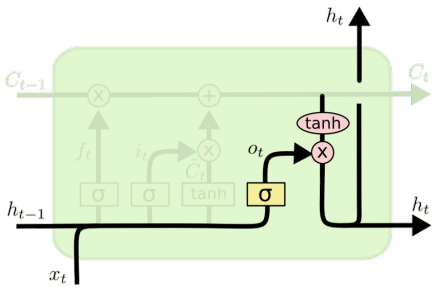


Figure 2.12: LSTM output gate [15].

state C_t. Multiplication of both outputs determines which information the hidden state should carry.

2.1.5 Gated Recurrent Unit

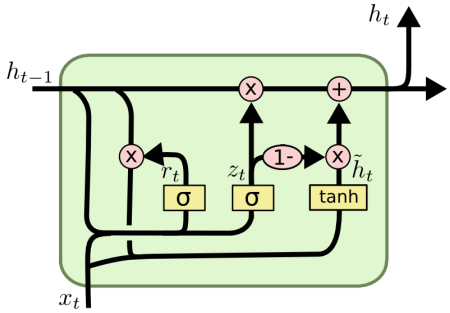


Figure 2.13: GRU cell [15].

A Gated Recurrent Unit (GRU) is a variation of LSTM introduced by Cho et al. in 2014 [18]. Please see Figure 2.13 for the GRU cell diagram. It drops the cell state C_t and operates only on the hidden state h_t to propagate information forward. GRU also has only two gates - reset and update. The update gate replaces the forget and input gates of LSTM. That makes GRU simpler and faster in practice compared to LSTM as there are fewer tensor operations. Authors also show that it performs comparably to the LSTM, and the choice between the two cells is highly dependent on the problem at hand. As previously, we will in detail go through the forward pass of the recurrent cell.

Reset Gate

$$r_t = \sigma(W_r[h_t - 1, x_t] + b_r) \tag{2.22}$$

The reset gate applies a sigmoid function to the linear combination of previous hidden state h_{t-1} and input x_t. The output r_t is later used to calculate a new candidate hidden state $\tilde{h_t}$ as follows

$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) \tag{2.23}$$

In the above equation, we can see how the reset gate controls the influence of the previous hidden state on the candidate state. When the entry in r_t is 1, it designates to consider all the information from the past hidden state. Likewise, when the value of r_t is 0, it ignores the previous hidden state.

Update Gate

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{2.24}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{2.25}$$

The update gate acts as both input and forget gates in LSTM. It controls how much of the historical information h_{t-1} should be updated with the new one coming from the candidate state $\tilde{h_t}$. When z_t is close to 0, it would indicate that past information is no longer relevant and should be replaced with the more recent one. On the other hand, when z_t is close to 1, the old hidden state still holds relevant information and should be kept unchanged.

2.1.6 Attention Mechanism and Transformers

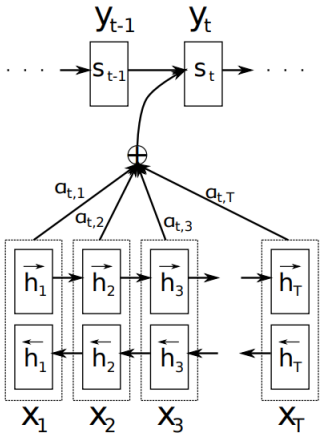


Figure 2.14: The encoder-decoder model with additive attention mechanism [19].

We have shown methods to partially tackle the vanishing gradients in RNNs with special gates introduced in LSTM or GRU cells. However, the issue persists, and it is hard for recurrent neural networks to capture very long dependencies. The problem was most apparent in neural machine translation, where the recurrent encoder compresses the sentence to a fixed-length representation. Bahdanau et al. introduce an attention mechanism to address the issue [19], see Figure 2.14. The main idea is to focus on parts of the sentence most relevant to the currently generated target word. To achieve that behaviour, Bahdanau et al. create a context vector c_i as a sum of encoder's hidden states h weighted by the input word importance for the current translation α.

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j \tag{2.26}$$

The importance α_{ij} is computed as

$$\alpha_{ij} = \text{softmax}(\text{align}(s_{i-1}, h_j)) \tag{2.27}$$

where

$$\text{softmax}(x_{ij}) = \frac{\exp(x_{ij})}{\sum_{k=1}^{T} \exp(x_{ik})} \tag{2.28}$$

The align is the alignment model providing the alignment score. It is a feedforward neural network and looks as follows

$$a(s_{i-1}, h_j) = v_a^\top \tanh(W_a[s_{i-1}, h_j]) \tag{2.29}$$

Both v_a and W_a are trainable parameters of the alignment model.

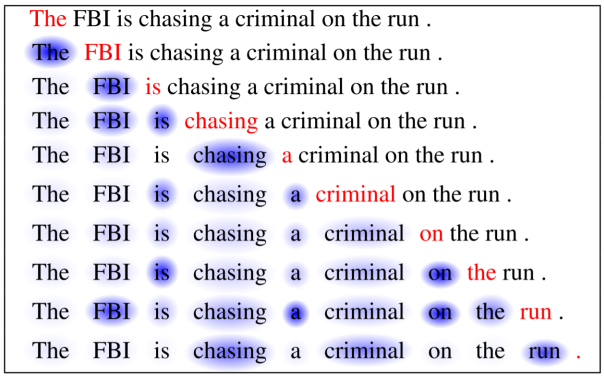


Figure 2.15: Self-attention activations. The current word is in red and the size of the blue shade indicates the activation level [20].

Self-attention is a variation of the attention mechanism. It finds relationships between embeddings at different time steps. Cheng et al. augmented LSTM with self-attention to improve the cell state proposing Long Short-Term Memory-Network [20]. Authors state that: "This enables adaptive memory usage during recurrence with neural attention, offering a way to weakly induce relations among tokens". Self-attention is more versatile than the previously described attention mechanism as it no longer requires the encoder-decoder architecture. It finds meaningful connections between input embeddings and is not affected by the sequence length. Please refer to Figure 2.15 to see how the mechanism attends to the past words.

In 2017 Vaswani, et al. released a paper "Attention is All You Need" [21]. As the title suggests, the proposed architecture allows for sequence processing relying primarily on the self-attention mechanism. Authors argue that it is possible to do seq2seq modelling without recurrent units. They call their proposed encoder-decoder model a transformer.

The transformer uses a multi-head self-attention mechanism. Here the attention maps a query (Q), keys (K) and values (V) vectors to an output. We use the query and the values to compute compatibility weights. Then they are used for the calculation of a weighted sum of the values. One problem with dot-product attention is that as the dimensionality of the input grows, so does the

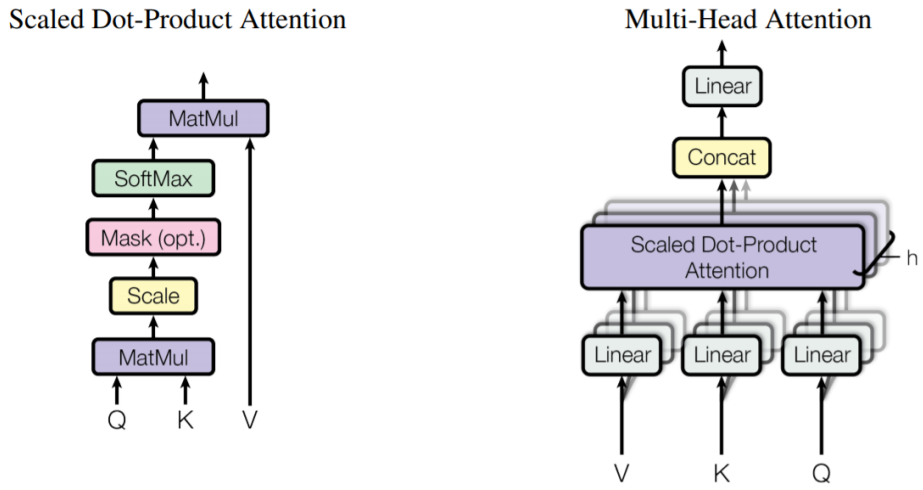


Figure 2.16: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. [21].

average size of the dot product, pushing the softmax function to the regions with small gradients. To address the issue, the authors introduce scaled dot-product attention. They divide it by $\sqrt{n}$, where n is the dimensionality of the Q and K vectors, see Figure 2.16.

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^\top}{\sqrt{n}}\Big)V \tag{2.30}$$

Vaswani et al., rather than compute the attention once, propose Multi-Head Attention that runs through the scaled dot-product attention multiple times in parallel, see Figure 2.16. The independent attention outputs are concatenated and linearly transformed to the desired dimensions.

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.31}$$

$$\text{MultiHead}(Q, K, V) = [\text{head}_1, ..., \text{head}_h]W^O \tag{2.32}$$

Here W^Q, W^K, W^V are parameter matrices to be learned. The necessity of this approach derives from the hypothesis that different words relate to each other in different ways. The full transformer architecture is present in Figure 2.17. In addition to Masked Multi-Head Attention blocks, we can see Add & Norm blocks. They sum all the incoming connections in the diagram and apply layer normalization [22] inspired by batch normalization [23]. The batch normalization proposed by Ioffe et al. standardises the activations of each input variable per batch, meaning reducing to zero mean and unit variance. The procedure proved to accelerate and stabilize the training of deep neural networks, making them less sensitive to the choice of learning rate. Originally, the authors claimed that increased performance is due to the reduction of the internal covariate shift. However, that hypothesis has been severely challenged [24]. New hypotheses argue that batch normalization mitigates the interdependency between layers during training or smoothens the optimization landscape. In contrast, the layer normalization proposed by Hinton et al. normalizes the activations along the feature direction instead of the batch direction. It removes the dependency on batches making it applicable to RNNs. In essence, it normalizes each feature of

the activations to zero mean and unit variance.

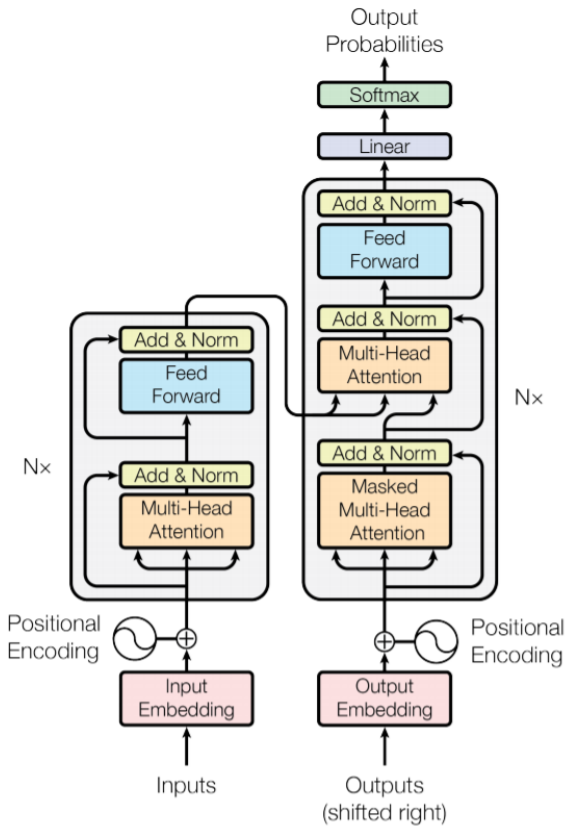


Figure 2.17: The transformer model architecture [21]. The left module is an encoder that passes output to the decoder module on the right.

The multi-head attention does not encode sequential information in itself. It operates on the input set as a whole. To impute positional information, the authors of "Attention is All You Need" solve the problem by adding a positional encoding to the input sequence embeddings, see Figure 2.17. Instead of learning the positional encoding vectors, the authors use a mixture of sine and cosine functions of different frequencies.

$$\mathrm{PE}_{pos,2i} = \sin\Big(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\Big) \tag{2.33}$$

$$\mathrm{PE}_{pos,2i+1} = \cos\Big(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\Big) \tag{2.34}$$

The Use of deterministic functions allows the model to extrapolate to sequence lengths longer than present in the training data set.

Layer Type	**Complexity**	**Maximum Path Length**
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurent	$O(n \cdot d^2)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(\log_k(n))$

Table 2.1: Comparison of layer complexities and maximum path lengths [21].

Another benefit of self-attention is the computation complexity. When the sequence length n is larger than the dimensionality of the representation d, self-attention is faster in comparison to recurrent and convolutional layers, see Table 2.1. What is more, self-attention is highly parallelizable and has the shortest maximum path length between sequence positions.

2.1.7 Regularizing Neural Networks

Flexible models such as neural networks can suffer from overfitting - fitting to the noise. Therefore, we need to constrain them to improve their generalization capabilities. Here we describe several methods to mitigate the problem: early stopping, weight decay, dropout.

Early Stopping

One solution to overfitting could be reducing the number of parameters in the network. That way, the model might be too simple to overfit. However, it may not be able to account for all dependencies in the data resulting in underfitting. Instead, we could stop the training early. To apply early stopping, we would require validation data set to monitor the model's performance. Then stop the neural network training when there was no improvement on the validation data set for a given number of epochs. Early stopping concurrently prevents overfitting and underfitting.

Weight Decay

Another solution is to penalize network complexity. That way, we can use numerous parameters to capture complex non-linearities while preventing the weights from being too variable, causing the network to fit the weak signals in the data, such as noise. We could add the L1 or L2 norm of weights as a penalty to a loss function multiplied by λ - weight decay parameter.

Loss with L1 penalty:

$$\text{Cost}(x, y, \theta) = \text{Loss}(x, y) + \lambda \sum_i |\theta_i| \tag{2.35}$$

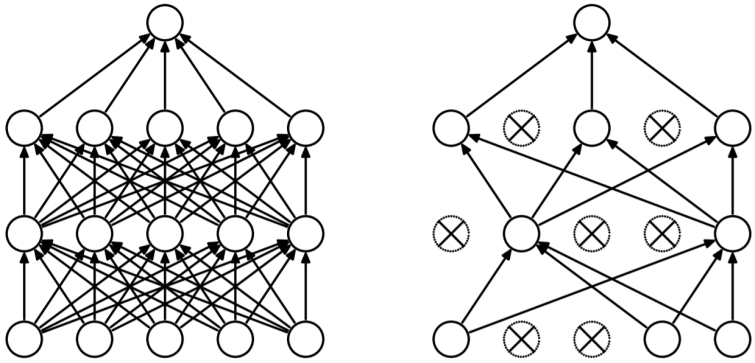
Loss with L2 penalty:

$$\text{Cost}(x, y, \theta) = \text{Loss}(x, y) + \lambda \sum_i \theta_i^2 \tag{2.36}$$

The L1 norm can reduce some weights to zero, which we might leverage to compress a neural network. Otherwise, L2 is a more common choice.

Dropout

Overfitting can also be caused by co-adaptation, meaning that the neurons are highly dependent on each other. That makes the network highly sensitive to the input and lingers generalization. We can also observe that the predictive capacity of neurons can vary significantly. Hinton et al. address this problem with Dropout [26] and later with Srivastava et al. applies it to a feedforward neural network [25]. Dropout randomly omits neurons in the training phase as present in Figure 2.18. The probability of omitting a neuron follows a Bernoulli distribution with probability p.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.18: Example of dropout in a feedforward neural network. (a) shows a network with all connections active. (b) presents the network after randomly deactivating some of the neurons [25].

Authors recommend to use $p = 0.5$ for hidden layers and $p = 0.2$ for the input layer. To apply dropout, we can element-wise multiply the layer output with a mask m.

Training phase:

$$y = g(Wx) \odot m,\ m_i \sim Bernoulli(p) \tag{2.37}$$

Testing phase:

$$y = (1 - p)g(Wx) \tag{2.38}$$

Please see that during the testing phase, we do not apply dropout. To compensate for the additional information, we need to weigh the layer output with the probability of a neuron being active. Another family of Dropout methods is Gaussian Dropout [27, 28, 29]. On a high level, the mask is a set of random variables following a $Normal(1, p(1-p))$ distribution. This method does not result in better regularization compared to standard dropout. However, it makes the training more efficient. The mask is no longer binary, so all neurons are considered in the update. Also, we do not need to weigh the layer output during testing as the element-wise product with the Gaussian mask is not biased. Alternatively, instead of dropping neurons, we could only consider dropping the connections between them. That is what Wan et al. propose in DropConnect [30]. They apply a Bernoulli mask to the matrix of weights rather than to the output vector from the layer.

The above method is well suited for dense networks. However, convolutional and recurrent neural networks require variations of these methods. If we consider an image as an input, we can easily see that adjacent pixels are heavily correlated. Therefore, randomly dropping pixels would not be a valid regularization technique. Alternatively, we could use Spacial Dropout [31], where Tompson et al. propose to exclude full feature maps from the forward pass. Then, using an RGB image as an example, we randomly drop one of the colours. The method also applies a Bernoulli mask during training and weights by the probability of presence when the dropout is unused. The Cutout [32] proposed by Devries and Taylor masks image regions with random squares to

overcome the problem of highly correlated pixels.

Recurrent neural networks propagate sequential information. If we would randomly block information flow, this could result in the loss of past information. To address the issue, Moon et al. propose RNNDrop [33]. The Bernoulli mask is only applied to the hidden cell states and is kept the same for all time steps. Semeniiuta et al. proposed a variation of this method called Recurrent Dropout [34]. Alternatively, it applies dropout only to the memory state update rule. That way, the Bernoulli mask prevents some elements from contributing to the memory. But the memory itself is not altered. The last dropout method we discuss is RNN Dropout [35]. Gal and Ghahramani zero out the hidden states before the internal gates for multi-layer architectures. That results in applying dropout to different points of the LSTM or GRU.

2.1.8 Gradient Descent Algorithms

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function J. It repetitively takes steps in the opposite direction to the gradient of the objective function at a given point scaled by learning rate α. That results in following the steepest descent. The pseudo-code of gradient descent is present below.

$$\begin{aligned} &Repeat\ \{ \\ &\theta = \theta - \alpha \nabla J(\theta) \\ &\} \end{aligned} \tag{2.39}$$

The most important parameter to set for the gradient descent algorithm is the learning rate. The proper setup of α is essential for the network's learning performance. If the learning rate is too large, it could oscillate around a local minimum or even diverge. On the other hand, a small learning rate slows down learning and may converge in the shallow local minimum. We could also consider learning rate annealing to decrease the gradient scaling factor as the learning progresses. It preserves rapid initial improvement without sacrificing the final model performance. Calculating the exact direction requires extensive computing power. Therefore, it is more common to use an approximate gradient considering randomly selected samples for an update step. This approach is called stochastic gradient descent (SGD). It reduces the optimization time and prevents the algorithm from getting stuck in the shallow local minimum as the gradients are noisy.

Momentum

One extension of SGD is momentum proposed by Rumelhart, Hinton and Williams [5]. It incorporates a moving average to the gradient updates.

$$g_t = \mu g_{t-1} + \alpha \nabla J(\theta_t) \tag{2.40}$$

$$\theta_{t+1} = \theta_t - g_t \tag{2.41}$$

Momentum prioritises moving in the average direction, significantly damping oscillations promoting higher learning rates. It can speed up convergence as it does not slow down when the objective flattens, contrary to the standard SGD. However, that may also result in overshooting or oscillat-

ing around the local minimum. Thus, the momentum parameter μ may need to be reduced with the iteration count to ensure convergence.

Another variation of momentum is Nesterov's Accelerated Gradient (NAG) [36]. It uses a velocity vector to predict where the gradient is going to be. First, we move to the estimated future point. Then we correct in the gradient direction at the new position.

$$g_t = \mu g_{t-1} + \alpha \nabla J(\theta_t - \mu g_{t-1}) \tag{2.42}$$

$$\theta_{t+1} = \theta_t - g_t \tag{2.43}$$

That approach significantly improves convergence. It prevents overshooting the local minimum by introducing a notion of forecasting.

Adaptive gradients

Another problem of SGD is that it uses the same learning rate for all parameters. The low variable parameters will contribute less to the updates. One of the methods that address this problem is AdaGrad, published by Duchi et al. in 2011 [37]. It decreases the learning rate for the frequently occurring features and increases otherwise, making it well-suited for sparse data.

$$g_{t,i} = \nabla J(\theta_{t,i}) \tag{2.44}$$

$$G_{t,ii} = \sum_{\tau} g_{\tau,i}^2 \tag{2.45}$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i} \tag{2.46}$$

One problem with AdaGrad is that the accumulated gradients eventually make learning slow as the learning rate can not recover.

Adaptive Moment Estimation (Adam) [38] combines adaptive learning rate with momentum. It uses estimates of the first (mean) and second (unbiased variance) moments of the gradients through exponential moving averages, respectively m and v. If we would view momentum as a ball running down a hill, Adam also models friction. We can calculate the moments as follows

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla J(\theta_t) \tag{2.47}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla J(\theta_t))^2 \tag{2.48}$$

We initialize the moments with zeros, thus creating a bias towards zero. Therefore, we need to find unbiased estimates.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.49}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.50}$$

Finally resulting in an update rule

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \tag{2.51}$$

Adam is one of the most commonly used gradient descent algorithms due to its fast convergence. However, Loshchilov and Hutter point out that when using L2 regularization, SGD with momentum often produces models that generalize better [39]. When using adaptive gradients, the L2 regularization term is present in the moving averages. That can result in less regularized weights with high historic parameters or gradient amplitudes. To mitigate the problem, the authors propose AdamW. It implements weight decay with parameter λ, not in the loss function but directly in the update rule.

$$\theta_{t+1} = \theta_t - \alpha\Big(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} + \lambda\theta_t\Big) \tag{2.52}$$

Another remark is that even though the learning rate is scaled adaptively, we could still benefit from learning rate annealing.

2.1.9 Hyperparameter Optimization

Hyperparameters define a hypothesis space. Therefore, the choice of appropriate parametrization is crucial for optimal performance. There are several common approaches to hyperparameter search. The first approach is to use expert knowledge and manually tune a model. However, this can be time-consuming and intractable in complex search spaces. Another approach to consider is grid search that automates the process. However, the search space grows exponentially with the number of parameters, and it may be impossible to consider every combination of parameters. Alternatively, we could use random search by sampling the hyperparameter space. This method can acquire good results in low dimensional search spaces. However, due to the uninformed nature of the algorithm, the random search can spend considerable time evaluating not promising parameter configurations. For that reason, sequential model-based optimization (SMBO) is often the choice for complex search spaces.

SMBO algorithm creates a probability model ("surrogate") of the objective function to find the most promising hyperparameters to test with the objective function. It is often less computationally demanding to optimize the surrogate compared to training and evaluating a model. SMBO takes into account the previous outcomes performing an informed search. The algorithm outline:

1. Create a surrogate probability model of the objective function.
2. Find the best performing hyperparameters on the surrogate.
3. Test the objective function with the found hyperparameters.
4. Update the surrogate model with new results.
5. Repeat steps 2-4 until max iterations.

The SMBO algorithm maximizes expected improvement EI_{y^*} with respect to the given set of proposed hyperparameters [40].

$$EI_{y*}(x) := \int_{-\infty}^{\infty} \max(y^* - y, 0)p(y|x)dy \tag{2.53}$$

where

$y -$ score

$y^* -$ threshold score

$x -$ set of hyperparameters

$p(y|x) -$ surrogate

In this thesis we use Tree-structured Parzen Estimator (TPE) surrogate function, which does not model $p(y|x)$ directly. Instead, it models $p(x|y)$ as well as $p(y)$ and applies Bayes rule.

$$p(y|x) = \frac{p(x|y) * p(y)}{p(x)} \tag{2.54}$$

TPE algorithm [40]:

1. Create search space - define distributions describing hyperparameters.
2. Perform a random search to initialize the algorithm.
3. Split the observations into two groups based on y^* threshold: good ($y < y^*$) and bad ($y \geq y^*$).
4. Model the $p(x|y)$ as two density functions $l(x)$ and $g(x)$ based on groups:

$$p(x|y) = \begin{cases} l(x), & y < y^* \\ g(x), & y \geq y^* \end{cases} \tag{2.55}$$

   The density functions are modelled by Parzen estimators also known as kernel density estimators.

5. Model $p(y)$ so that:

$$p(y < y*) = \gamma \tag{2.56}$$

   where

   $\gamma -$ defines percentile split into two groups

6. Taking equations 2.54, 2.55, 2.56 and substituting them to equation 2.53 it can be shown that

$$EI_{y*} \propto \left(\gamma + \frac{g(x)}{l(x)}(1-\gamma)\right)^{-1} \tag{2.57}$$

   Which means, to maximize the expected improvement the algorithm must maximize the ratio $\frac{l(x)}{g(x)}$.

7. Iterate in the loop updating the surrogate function, and optimizing expected value to find the best hyperparameters x^*.

The main drawback of the TPE algorithm is that it does not model interactions between hyperparameters. Thus, it may perform worse than the Gaussian Process when two hyperparameters strongly interact.

2.2 Machine Learning in Proteomics

Due to recent advancements in mass spectrometry, we can comprehensively study protein structures in untargeted and targeted proteomics. For untargeted proteomics, proteins are digested into peptides and injected into liquid chromatography-tandem mass spectrometry (LC-MS). Then the detection can be performed with the data-dependent acquisition (DDA) or the data-independent acquisition (DIA) method. On the other hand, targeted proteomics require applying multiple reaction monitoring (MRM) or parallel reaction monitoring (PRM) methods to detect selected proteins of interest. DIA and DDA experiments can produce large volumes of MS/MS spectra. That gives rise to the need for sophisticated computational methodologies for proteomics data analysis. In this section, we discuss some of the recent machine learning applications. We first show how to extract features from amino acid sequences. Later, we follow with peptide detectability prediction, peptide retention time prediction and protein structure prediction.

2.2.1 Feature Extraction Methods From Amino Acid Sequences

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
Counts	3.00	2.0	1.00	1.00	1.00	2.0	0.0	0.0	0.0	2.0	0.0	1.00	0.0	4.0	0.0	0.0	1.00	1.00	0.0	1.00
Binary	1.00	1.0	1.00	1.00	1.00	1.0	0.0	0.0	0.0	1.0	0.0	1.00	0.0	1.0	0.0	0.0	1.00	1.00	0.0	1.00
Relative	0.15	0.1	0.05	0.05	0.05	0.1	0.0	0.0	0.0	0.1	0.0	0.05	0.0	0.2	0.0	0.0	0.05	0.05	0.0	0.05

Figure 2.19: Bag of words variations for peptide AACAQLNDFLQEYGTQGCQV.

One of the peptide representation methods is a bag of words (BOW). It encodes a peptide as a vector of amino acid counts. We could also consider normalizing the vectors by sequence length or producing binary encoding whether an amino acid is present in the sequence or not, see Figure 2.19. A disadvantage of this representation method is that we lose sequential information. Alternatively, we could assign a unique index to each amino acid and create a one-hot encoding. The one-hot encoding represents each amino acid as a vector of zeros with one at the index corresponding to a given amino acid. That representation is sparse and therefore inefficient. It is also high dimensional, so learning algorithms require considerable amounts of training data. In biology, the data sets are often relatively small as gathering this kind of data may be unpleasant for the patient (collecting samples of spinal fluids) or raise ethical concerns (experiments on animals). Also, inspecting the samples with methods such as MS is expensive. Hence, there is an apparent need for dense protein representations that could be used by machine learning models with relatively good sample efficiency, such as Gaussian Processes or Support Vector Machines. A substantial amount of research on unsupervised representation learning was carried out by the natural language processing (NLP) community. If we view amino acids as tokens, we can apply successful NLP methods to proteomics.

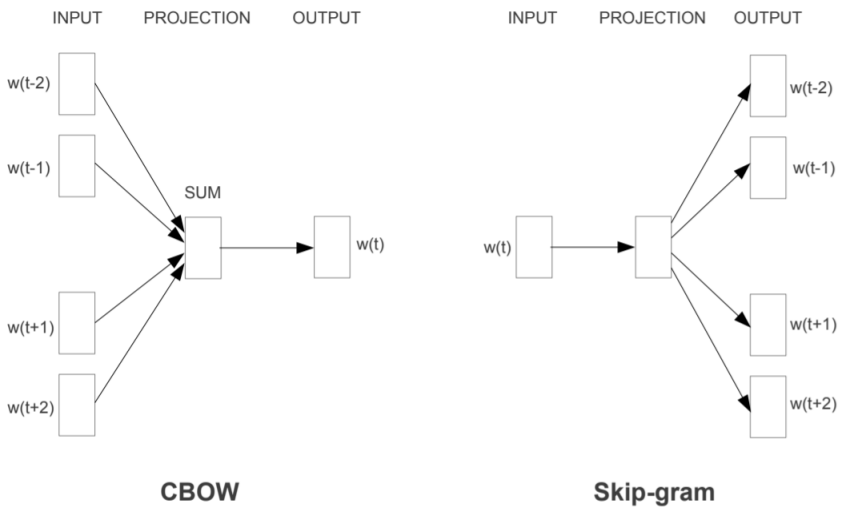


Figure 2.20: Word2vec models. (left) continuous bag of words model - CBOW. (right) skip-gram model [41].

In 2013 Mikolov et al. proposed a word2vec model for training word embeddings [41]. It relies on a distributional hypothesis that words, which often have the same neighbours, tend to be semantically similar. The authors use that assumption to create a mapping where these words are close to each other. The authors propose two architectures: a continuous bag of words model (CBOW) and a skip-gram model, see Figure 2.20. Both models are a single hidden layer neural network that takes one-hot encoded words as input and applies softmax activation function after the output layer. The CBOW model predicts a word given its neighbours. Its name comes from the fact that it uses a continuously distributed representation of the context. On the other hand, the skip-gram model inverts the problem and uses the current word to predict its neighbours. It is slower compared to CBOW but better handles infrequent words.

Original Sequence

$^{(1)}\overrightarrow{M}{}^{(2)}\overrightarrow{A}{}^{(3)}\overrightarrow{F}SAEDVLKEYDRRRRMEAL..$

Splittings

1) MAF, SAE, DVL, KEY, DRR, RRM, ..
2) AFS, AED, VLK, EYD, RRR, RME, ..
3) FSA ,EDV, LKE, YDR, RRR, MEA, ..

Figure 2.21: Creating words out of protein sequence [42].

Agari et al. used the word2vec model to encode protein sequences [42]. Each amino acid sequence is split into three training sequences, see Figure 2.21. Later, the 3-mers are passed as words to the word2vec skip-gram model with 100-dimensional word vectors. The authors found that approach to yield the best results. Other considerations included different windows sizes or creating the set of all mers instead of non-overlapping ones. To speed up the training, they used negative sampling to approximate the softmax calculation. Agari et al. state that their feature extraction method captures various physical and chemical protein properties. They also used this representation for protein family classification resulting in an average accuracy of 93%

or identification of disordered sequences with nearly 100% accuracy.

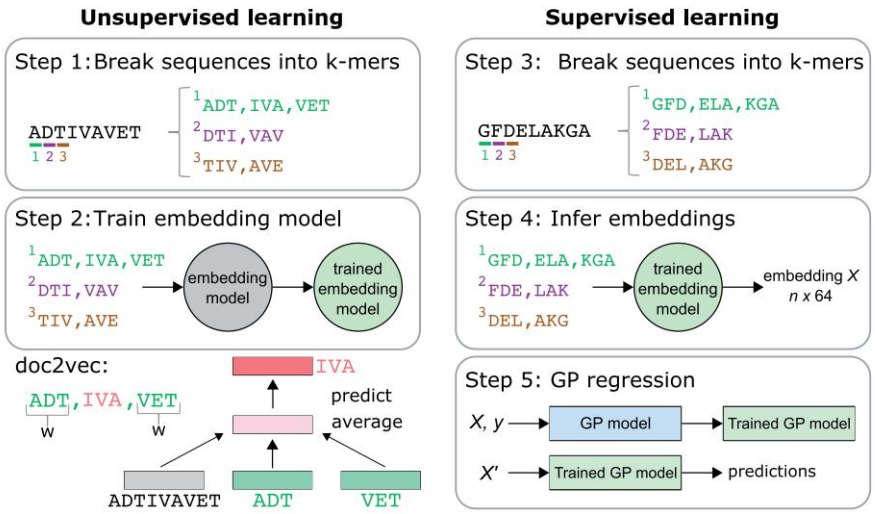


Figure 2.22: Doc2vec inspired embedding method [43].

Another example is the use of the doc2vec [44] embedding method applied to amino acid sequences to predict the functional properties of a small number of related proteins [43]. The doc2vec model tries to encode a document into a dense vector, as opposed to single words. It extends word2vec, additionally passing a unique document vector to the input of the CBOW architecture. That way, it simultaneously trains the word vectors as well as document vector representation. The authors of doc2vec call this variation a Distributed Memory version of Paragraph Vector (PV-DM) because it uses memory to remember what is missing from the current context. Yang et al. created words from amino acid sequences by running a sliding window with size k to break proteins into k-mers, similarly to Agari et al.. Then they trained the doc2vec with a whole protein sequence as a unique document entry, see Figure 2.22. The embedding model was trained in an unsupervised manner on 524 529 unlabeled sequences from the UniProt database. It allowed finding a low-dimensional dense primary protein structure representation. Next, they trained the Gaussian Process model on the learned embeddings. The authors contrasted the results with one-hot encodings of sequence, structural contacts, mismatch string kernels, and amino acid physical properties as input. The performance with the embeddings was comparable and often better to other representations proving that as few as 32 dimensions are sufficient to achieve competitive model performance.

Recent advancements in NLP have shown that self-supervised learning can extract complex knowledge from unlabelled data [45, 46, 47]. The most popular self-supervised pre-training methods are present in Figure 2.23. Embeddings from Language Model (ELMo) [48], uses a BiLSTM language model to produce word embeddings. Unlike word2vec, it considers context information to create a word vector. Hence, the same word can have different representations conditioned on a different context. A Generative Pre-trained Transformer (GPT) [47] is another contextual embedding model. It is a left-to-right Transformer that applies a triangular mask for the attention mechanism. The Bidirectional Encoder Representations from Transformers (BERT), as the name

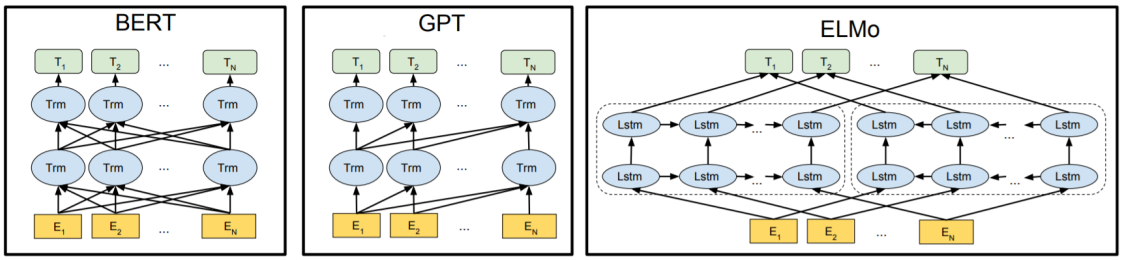


Figure 2.23: Differences in self-unsupervised pre-training of language model architectures [45].

suggests, is also a Transformer, but it does not mask the attention mechanism. Both ELMo and GPT are trained with next word prediction on a large text corpus of billions of words. However, as the BERT model considers past and future context to produce a word embedding, it requires a more sophisticated pre-training technique. It randomly masks the words that should be extrapolated from the context. In addition, BERT and GPT are fine-tuning approaches, whereas ELMO is a feature-based approach. The self-supervised models dominated natural language processing. They can be used for different downstream tasks and allow for knowledge transfer from large text corpora.

The success of self-supervised learning in NLP gave rise to many recent proteomic solutions. To mitigate the increasing need for systematic evaluation of protein embedding methods, Rao et al. propose a Tasks Assessing Protein Embeddings (TAPE) method [49]. It evaluates the models on five diverse supervised tasks highlighting three major areas of protein biology: structure prediction, detection of remote homologs, and protein engineering. Later the authors apply SOTA NLP embedding methods to primary protein structures treating each amino acid as a token. They evaluate three self-proposed models: BERT transformer, two LSTM networks with outputs concatenated similarly to ELMo, ResNet autoencoder [50]. They also consider a weakly supervised LSTM from [51] as well as mLSTM [52] with 1900 hidden units from [53]. The BERT model and Resnet were trained with masked-token prediction, while the LSTM networks with next-token prediction. All models were compared to the One-Hot amino acid sequence encoding with alignment as a baseline. The conclusion from the study is that self-supervised embedding models did not outperform SOTA alignment-based methods. Surprisingly, none of the proposed models has shown dominating performance. The transformer model outperformed others on the fluorescence prediction, LSTM on the homology task, and One-Hot with alignment on Structure prediction tasks. It shows the need for protein-specific embedding methods to capture the underlying relations.

An alternative approach to self-supervised training of embeddings is supervised training of siamese networks. However, to do so, we need a notion of similarity between sentences. In 2017 Facebook AI proposed InferSent [54]. They considered architectures based on GRUs, LSTMs, BiLSTMs with mean or max-pooling methods to produce the final sentence embeddings. On top of embeddings, they applied a 3-way softmax classifier, see Figure 2.24. The authors pre-trained the models on the Stanford Natural Language Inference data set. Then they evaluated them on 12 different transfer tasks. Their study showed that the BiLSTM network with max-pooling produced the best results. However, the mean pooling was superior for smaller embedding dimensions. Supervised embeddings outperformed models fine-tuned on unsupervised embeddings as

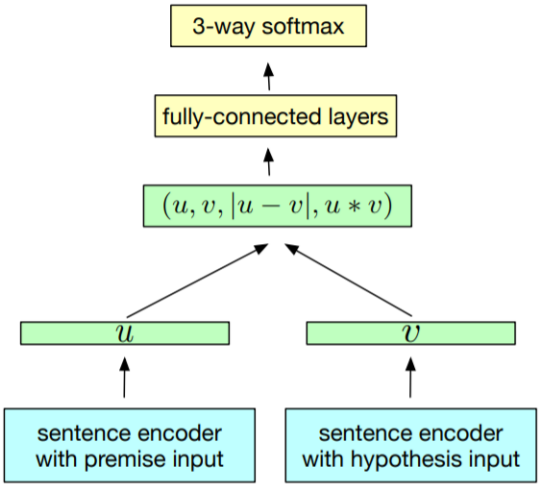


Figure 2.24: InferSent pre-training procedure [54].

well as supervised models without pre-training. Following the introduction of BERT, Reimers and Gurevych took a similar approach to InferSent but replaced recurrent layers with a bi-directional Transformer [55]. They also showed the importance of embedding transformations before the softmax classifier. Using concatenation of embeddings alone resulted in 66% accuracy on the Sentence Similarity benchmark. Extending the feature vector by the absolute difference of embeddings improved the model accuracy to above 80%.

It is not always apparent how to measure the similarity of proteins. However, Bepler et al. introduced a new structural similarity measure - soft symmetric alignment (SSA) [51]. They incorporate it as one of the targets for the representation learning model, see Figure 2.25. The encoder network uses a self-supervised pre-trained BiLSTM protein sequence encoder whose output is later concatenated with residue sequence and passed to another BiLSTM encoder. The procedure allows to improve sample efficiency of the model and reduce training time by leveraging transfer learning. The study has shown that incorporating information from structure improved upon previous methods for predicting structural similarity. It also proved beneficial to other tasks such as transmembrane region prediction. Hence, weakly supervised embeddings may provide a richer representation of the protein sequence compared to self-supervised methods.

2.2.2 Peptide Detectability Prediction

At the core of each proteomic research lies the accurate identification of all proteins in a cell. Shotgun proteomics provides a quantitative description of proteins through the analysis of peptides. The first step is to break proteins down with enzymes into peptide mixtures and analyze them with liquid chromatography-tandem mass spectrometry. Later, comparing the obtained spectrum with a reference database, we can simultaneously identify hundreds of proteins. Accurate protein detection and quantification is still challenging due to the stochastic nature of the peptide selection in the mass spectrometer. We could increase the accuracy by considering only peptides that are detectable with high probability. For that reason, we need methods to identify such peptides.

In 2017 Guan et al. compared several machine learning algorithms to predict peptide de-

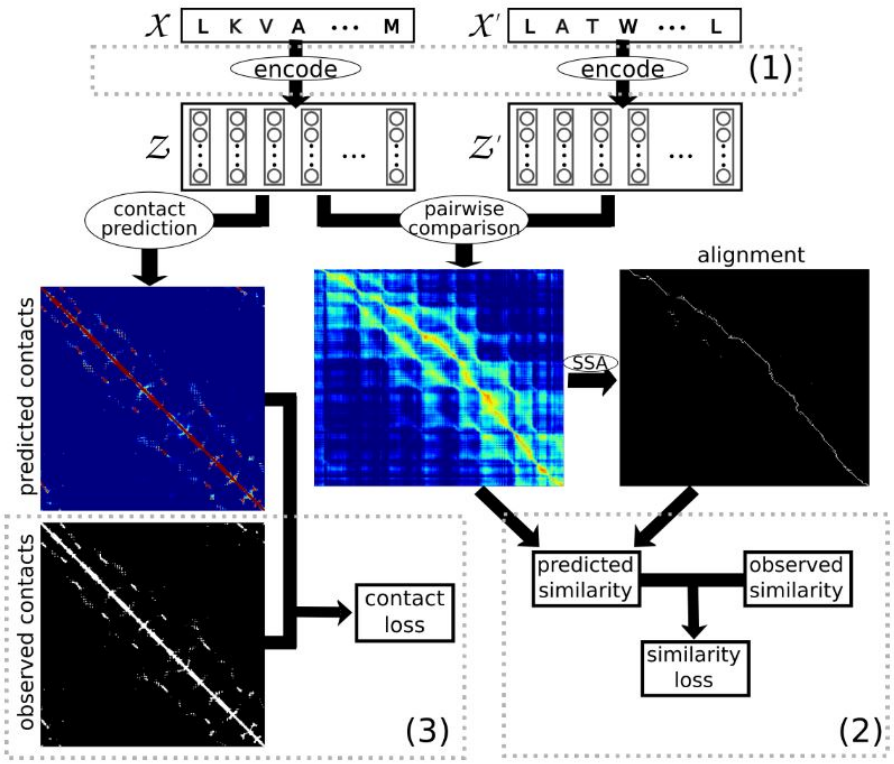


Figure 2.25: LSTM embedding model weakly supervised with information from the structure. (1) Encoder model computes the embeddings. (2) Embeddings are aligned based on L1 distance, and the similarity score is computed with ordinal regression. (3) The contact between amino acids is predicted for each protein, and the contact loss is calculated based on the available 3D protein structure. Finally, the parameters of the encoder are fit using the error signals from both tasks. [51].

tectability. The input consisted of the physicochemical properties of the peptides present in AAIndex and other features such as peptide length [56]. The study showed that Random Forrest (RF) provided the best results. Inspired by the NLP technology and similarity between protein sequences and natural language data, Serrano et al. proposed DeepMSPeptide [57]. They first represent each peptide as a padded sequence of amino acid indexes. Later, this data is passed through the embedding layer followed by 1 or 2 1D convolutional layers with 64 filters of window sizes 1 and 3 to extract sequential features. Finally, the encoded sequences were passed through 2 additional fully connected layers to return the probability of peptide detection. Recently, Cheng et al. proposed a siamese network architecture called PepFormer [58]. They encoded the peptides the same way as in DeepMSPeptide. The encoder is a hybrid of a 2-layer transformer followed by a 2-layer BiGRU, see Figure 2.26. The amino acid vectors are averaged to produce the peptide embedding. Later a fully connected neural network maps the embedding vectors to the unified vector representations. The siamese network module is then frozen to also fit the discriminator module for direct detectability prediction. Cheng et al. also carried out a comparison of the described approaches on two different data sets: homo sapiens (human) and mus musculus (house

mouse), see table 2.2. The definitions of the comparison metrics:

$$\text{accuracy (ACC)} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.58}$$

$$\text{specificity (SP)} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{2.59}$$

$$\text{sensitivity (SN)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.60}$$

AUC is the area under the Receiver Characteristic Operator curve. The perfect classifier has 1.0 AUC, and the always mistaken one has 0.0 AUC. The best performing model was PepFormer, followed by DeepMSPeptide and the Random Forest. The authors of PepFormer showed that the single network converged faster, but the siamese architecture was superior in performance, 80% vs 80.66% on homo sapiens data.

Homo Sapiens				
Model	**ACC**	**SP**	**SN**	**AUC**
RF	0.6743	0.5873	0.761	0.7364
DeepMSPeptide	0.7854	0.6875	0.8827	0.8513
Pepformer	0.8066	0.7213	0.8915	0.8640
Mus Musculus				
Model	**ACC**	**SP**	**SN**	**AUC**
RF	0.6218	0.5047	0.7398	0.6798
DeepMSPeptide	0.7228	0.6244	0.8220	0.784
Pepformer	0.7521	0.6421	0.8629	0.8108

Table 2.2: The results of different models on peptide detectability prediction [58].

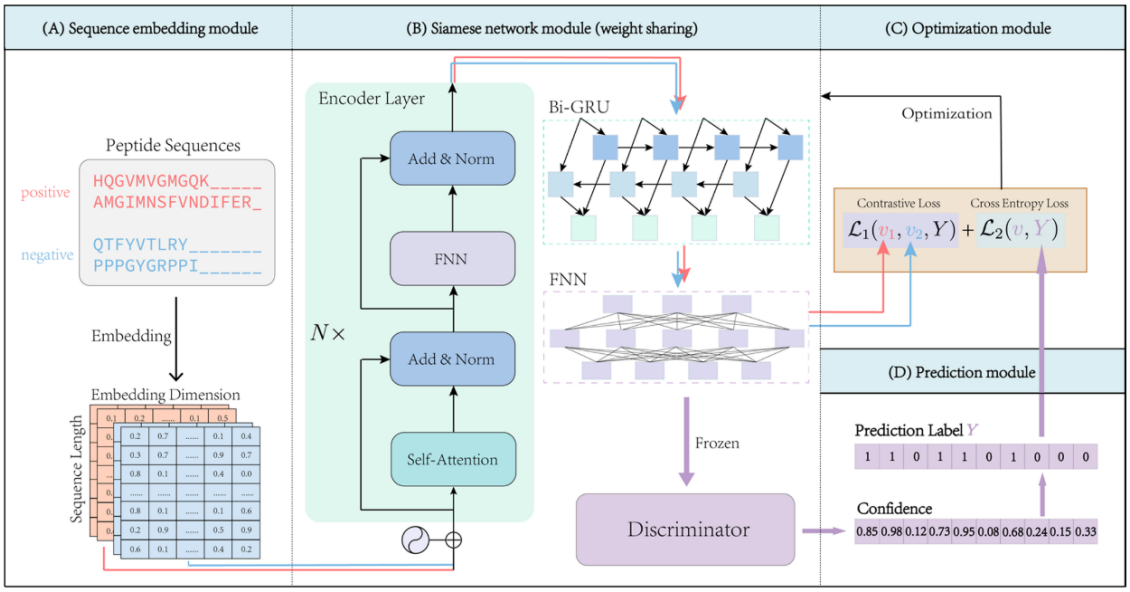


Figure 2.26: PepFormer architecture [58].

2.2.3 Retention Time Prediction

In proteomics experiments, before feeding the samples to the MS, peptide mixtures need to be separated via a liquid chromatography (LC) system. The peptide retention time corresponds to the time point when the peptide elutes from the LC column in LC-MS/MS system. It is also highly reproducible under the same LC conditions. The ability to accurately predict retention times can benefit facilitating targeted proteomics experiments, improving the sensitivity of peptide identification in database searching, building spectral libraries for DIA data analysis, and serving as a quality evaluation metric for peptide identification. Most deep learning approaches focus on recurrent, convolutional, or hybrid neural networks.

One of the RNN-based approaches is Prosit [59]. It represents a peptide as a one-hot encoding of an amino acid sequence of length 30, padded with zeros for shorter residue chains. The neural network has an embedding layer, a BiGRU layer followed by a GRU, and attention layers with decreasing dimensionality to find a compressed latent representation. It is then passed to a dense layer for RT prediction. The model showed substantial improvements to the previous index-based methods such as SSRCalc [60]. DeepMass [61] has similar architecture to Prosit with the difference that it uses a bidirectional LSTM with 40 units followed by another LSTM layer with 20 units compressing the representation. Then these latent vectors are fed into two dense layers with respectively 20 and 1 units. DeepMass has considerably better performance than SSRCalc. Another RT prediction method was proposed by Guan et al. [62]. Instead of padding the one-hot encoded sequences, it uses a masking layer during training and has two BiLSTM layers.

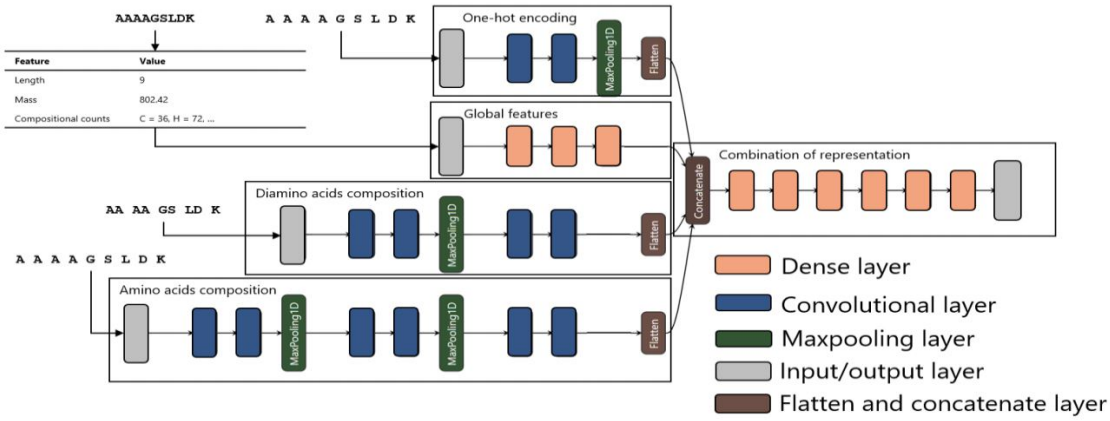


Figure 2.27: DeepLC architecture [63].

Authors of DeepRT [64] and DeepLC [63] used convolutional neural network as the base of their architectures. DeepRT has an embedding layer similar to Prosit and uses a capsule neural network (CapsNet) [65]. On the other hand, DeepLC uses a standard CNN and can predict RT for modified peptides not present in the training data. It is possible as DeepLC uses an encoding based on the atomic composition where each peptide is represented as a 60x6 matrix with a sequence in rows and atom counts in the columns (C, H, N, O, P, S). Furthermore, three additional peptide representations are considered to capture global and other position-specific information. Next, these four representations are passed to parallel paths and later concatenated and fed into a

multi-layer perceptron. The architecture schematic is present in Figure 2.27. DeepLC achieved similar performance as the SOTA RT prediction methods for unmodified peptides and performed comparably on modified peptides [63].

Authors of DeepDIA [66] and AutoRT [67] use hybrid architectures. DeepDIA uses one-hot peptide encoding, which is fed into a CNN network followed by a BiLSTM network. In contrast, AutoRT uses GRU rather than LSTM. Additionally, AutoRT employs automatic neural network architecture search (NAS) with a genetic algorithm. It then identifies best performing models for an ensemble. AutoRT also leverages transfer learning as base models are trained on the large peptide datasets and then fine-tuned for RT prediction.

2.2.4 Protein Structure Prediction

Protein 3D shape largely determines its function and how it works. Therefore, it is essential to know the protein structure to develop treatments for diseases or find enzymes that break down industrial waste. Unfortunately, the direct methods of establishing the protein 3D shape as X-ray crystallography are expensive. Furthermore, the DNA only encodes the primary structure of the protein, not the 3D shape. By Levinthal's paradox, the protein structure has too many possible configurations to use greedy search algorithms. Hence, approximate methods such as deep learning are a natural consideration for the protein folding problem. Recently DeepMind researchers made considerable advancement in protein structure prediction with a deep learning system called AlphaFold 2 [68].

The first version of the algorithm - AlphaFold [69], relied on predicting two physical properties of the protein structure: the distances between pairs of amino acids and the angles between chemical bonds connecting them. To do so, they encode a protein as an image where each pixel corresponds to pairwise features of amino acid sequence and Multiple Sequence Alignment (MSA). The MSA features are found by aligning the input residue sequence with a database of known protein structures. Correlated changes in the primary protein structure may indicate which residues are in contact. Next, the researchers feed the above to the dilated ResNet with 220 residual blocks with filter size 64 to predict the distance and torsion distributions. These distributions are sampled to initialize a protein structure search through gradient descent on the potential score determining the viability of a proposed protein structure, see Figure 2.28.

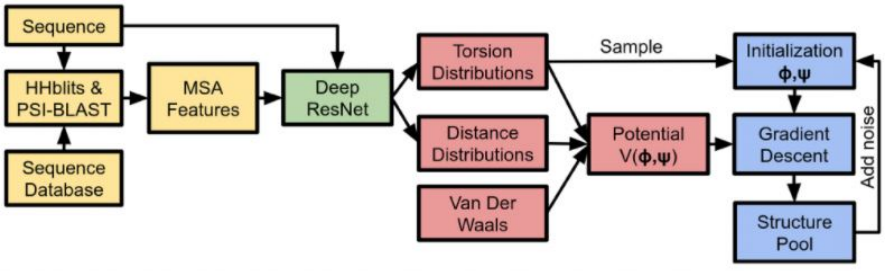


Figure 2.28: Schematics of the AlphaFold system [69].

The second version, called Alpha Fold 2, presented at the CASP14 competition, considerably outperformed other existing methods [70]. The new model employs an attention-based mechanism

that overcomes the problem of the locality of the ResNet filters to capture more distant dependencies. Furthermore, it uses evolutionarily related sequences, MSA, and amino acid features to refine the graphically represented protein structure. The model then iteratively refines the mentioned graph. Additionally, the AlphaFold 2 returns confidence of the predicted structural parts. The system was trained end-to-end on approximately 170 thousand protein structures from publicly available databases. The high-level architecture of the system is present in Figure 2.29.

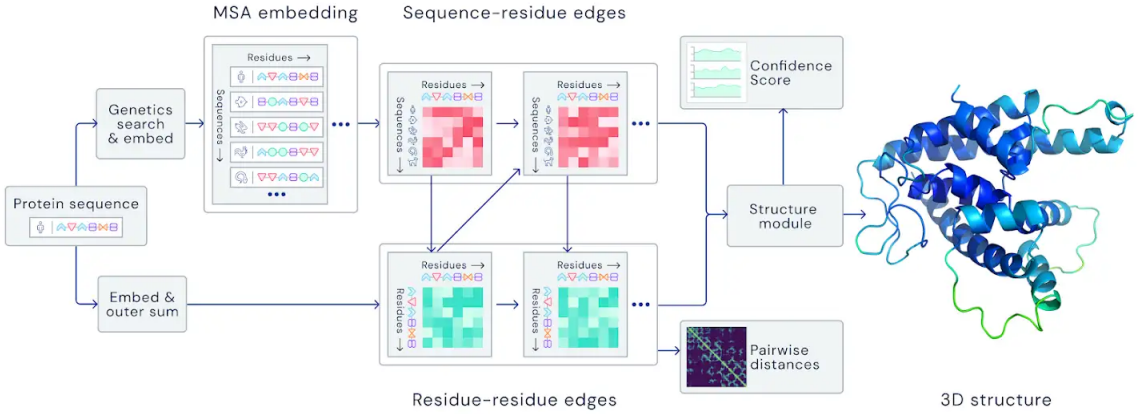


Figure 2.29: Schematics of the AlphaFold 2 system [71].

Chapter 3

Data

We consider two tasks relevant toshotgun proteomics. The first is peptide detectability prediction - a binary classification task. The second is peptide reproducibility prediction, with a data set provided by UCB Pharma.

3.1 Detectability

	peptide	detectability
0	LLSEVEELNMSLTALREK	0
1	ERMDEEQKLYTD	0
2	YVPRAVLVDLEPGTMDSIR	0
3	TAHYGSLPQKSHGR	1
4	KFVADGIFK	1

Figure 3.1: Detectability prediction data set sample.

We use the same two benchmark data sets as the authors of PepFormer [58]. These include information regarding mass spectrometry experiments on homo sapiens (human) and mus musculus (house mouse) peptides. They both consist of 90000 rows of peptides with the corresponding binary variable showing whether the peptide was detected in the mass spectrometry study, see Figure 3.1. The original data used to create the detectability data sets comes from the GPMDB database [72]. It holds experimentally validated peptide data from a variety of species. Here we focus on the homo sapiens and mus musculus data due to data abundance. The GPMDB data contains information about the minimum number of detections. Cheng et al. sort the peptides based on the number of detections and label the peptides as detectable or not so that the classes are balances, see Figure 3.2. Furthermore, the maximum peptide length in the homo sapiens data set is 79 and in mus musculus 69.

We split the data into train and test sets the same way as the authors of PepFormer, leaving 20000 samples for testing. It will allow us to compare our models to the previous studies using the methods comparison carried out by Chang et al., see Table 2.2.

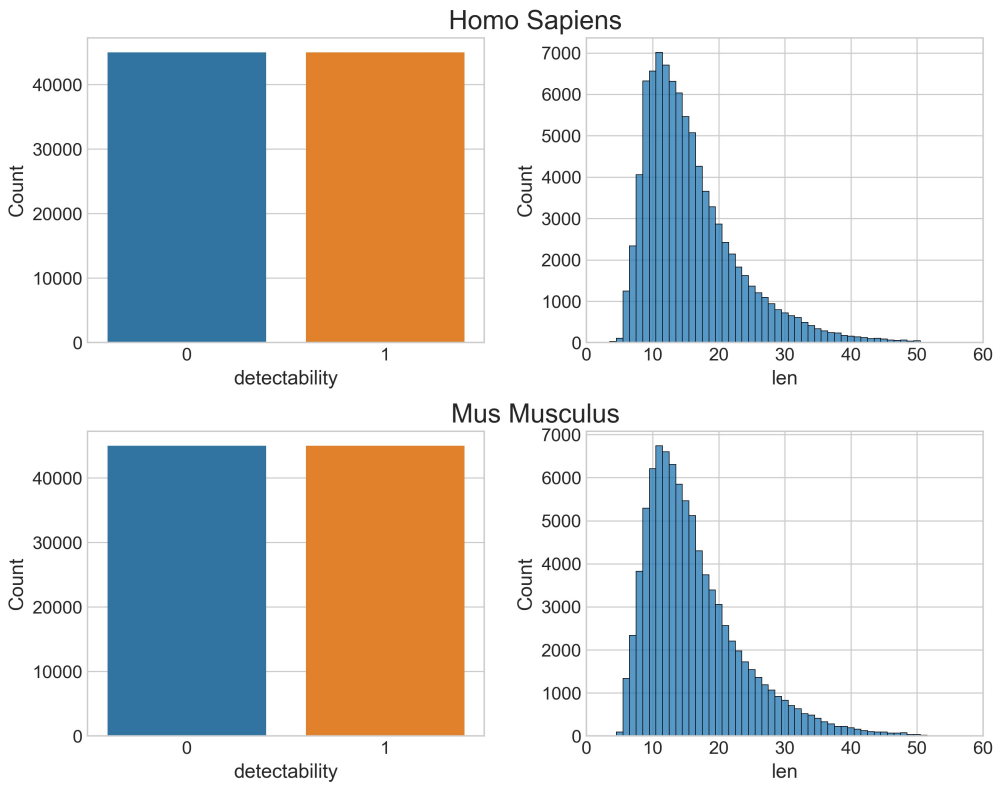


Figure 3.2: Distributions in the peptide detectability data sets.

3.2 Reproducibility

	peptide	peptide_len	peptide_mean_1	peptide_mean_2	protein_id	reproducibility
0	SELTQQLNALFQDK	14	28.516261	28.044687	P06727	0.134116
1	EETGQVLER	9	20.845601	18.876826	P43121	0.118270
2	QLYGDTGVLGR	11	22.800435	21.649654	P07360	0.730757
3	EVQVFEITENSAK	13	21.669861	20.489964	P12111	0.245870
4	EDYICYAR	8	23.403898	23.618924	Q92823	0.768097

Figure 3.3: Reproduciblity prediction data set sample.

The data set consists of 2911 rows. A data sample can be seen in Figure 3.3. Peptides are represented as sequences of letters, where each letter corresponds to one of 20 amino acids. The data was acquired by processing spinal fluid samples on two different MS machines. Later we take the log2 of the peptide abundance detections and calculate Pearson's Correlation as shown below

to acquire the reproducibility variable.

$$r_{xy} = \frac{\sum_i (x_i - \bar{x})((y_i - \bar{y}))}{\sqrt{\sum_i (x_i - \bar{x})^2}\sqrt{\sum_i (y_i - \bar{y})^2}} \tag{3.1}$$

Where x_i, y_i are data points and $\bar{x} = \frac{1}{n}\sum_i x_i$ is the sample mean (analogously for $\bar{y}$). The data for each peptide comes from 58 different patients. Peptide_mean_1 and peptide_mean_2 are the means of log2 transformed peptide abundance respectively from the first and second machine. The distributions of the variables are present in Figure 3.4. We also calculate 95% confidence intervals of the Pearson's Correlation calculation for reproducibility. On average, the differences between reproducibility and lower, upper bounds are respectively 0.22 and 0.18.

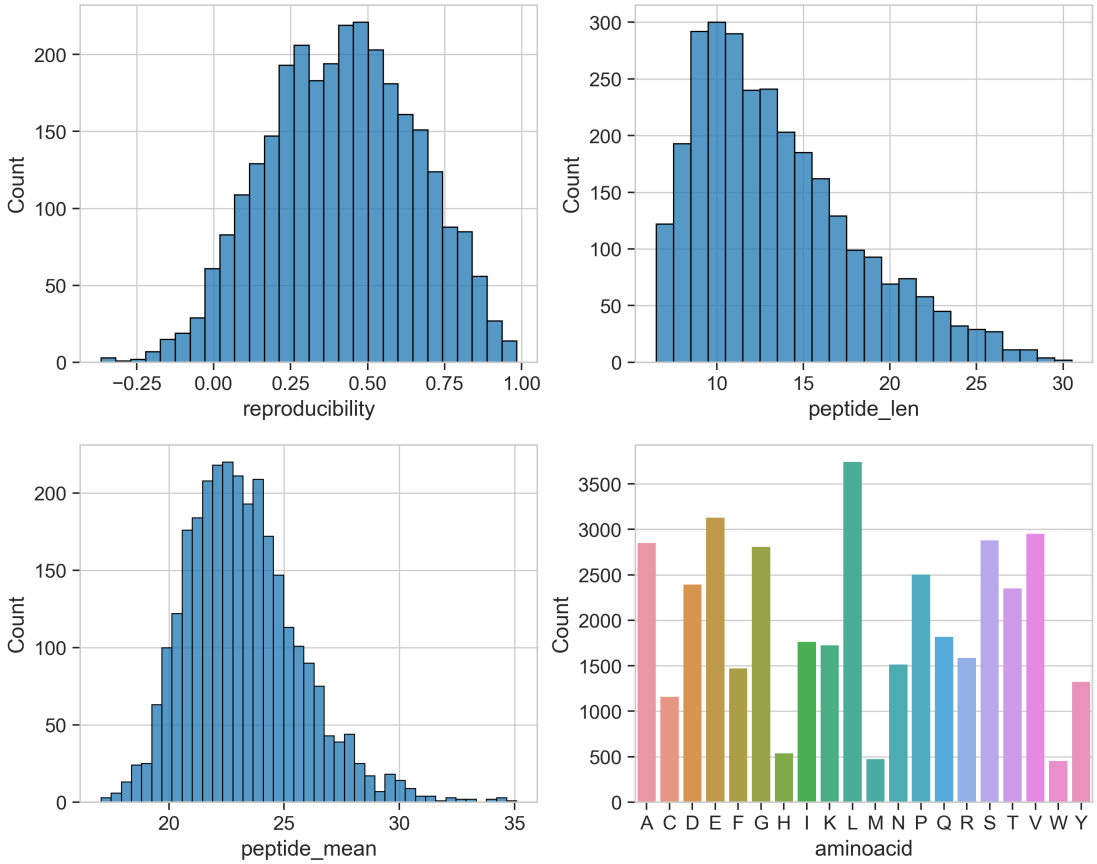


Figure 3.4: Distributions in the reproducibility data set.

Additionally, we know from which protein does each peptide come from - protein_id variable. There are 550 unique proteins in our data. Figure 3.5 shows reproducibility means grouped by proteins. The reproducibility distributions between proteins vary considerably. For that reason, we use group cross-validation to estimate the test error. It should prevent learning algorithms from matching peptides between train and test set based on proteins, providing honest error estimation. It will also ensure that our models generalize to peptides originating from proteins not present in the data set. We consider predicting with the training mean of reproducibility as a baseline. It results in a 0.05725 ± 0.00436 mean squared error from group cross-validation.

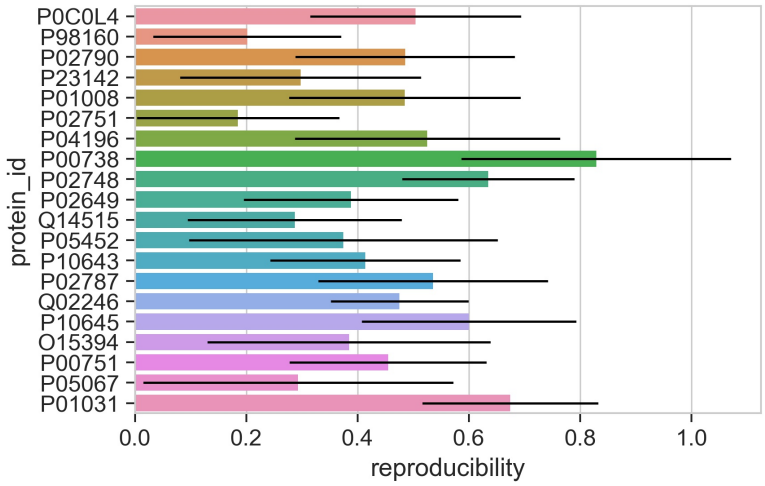


Figure 3.5: Mean reproducibility grouped by proteins. The error bars depict the standard deviations of reproducibility for peptides coming from the same protein.

Chapter 4

Detectability Experiments

4.1 Experimental Setup

To capture the sequential information from a residue sequence, we first set a unique index to each of the 20 amino acids and represent peptides as lists of indexes. However, we need to apply padding as peptides often have different lengths, and all inputs in the batch need to have equal dimensionality. Instead of padding all sequences to the maximum peptide length, we apply padding for each mini-batch to prevent unnecessary computations and speed up the training. Another optimization technique that we use is sequence packing for the recurrent neural networks. That way, we discourage the recurrent layer from performing operations on the padding embeddings.

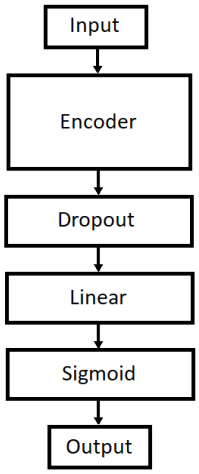


Figure 4.1: Model architecture.

The model architecture present in Figure 4.1 has in its core an encoder that returns a peptide encoding. It processes the sequential input to produce a contextual embedding for each of the amino acids. The residue embeddings are then averaged to yield the final peptide embedding. Moreover, in the pooling procedure, we account for peptide length in a mini-batch to discard padding from the final embedding computation. After the encoder, we apply Dropout (same dropout probability p throughout the whole network) and pass the peptide encoding through a

linear layer with a Sigmoid activation function to map the embedding to the 1-dimensional output. We use a binary cross-entropy loss as our training objective.

$$l(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}) \tag{4.1}$$

What is more, we use the AdamW gradient descent algorithm as it generalized better than Adam without decoupled weight decay and converges faster than SGD with momentum. We described it in detail in section 2.1.8 Gradient Descent Algorithms. Furthermore, we use exponential scheduling of the learning rate as follows

$$\text{scheduler(epoch)} = \text{lr} \cdot 0.97^{\text{epoch}} \tag{4.2}$$

That allows using a high learning rate at the beginning of the training to speed up convergence. Furthermore, a small learning rate in the later stages of the training ensures reaching the bottom of the local minimum. For the encoder module, we consider the following architectures: BiGRU, CNN-BiGRU, BERT, BERT-BiGRU, see Figure 4.2.

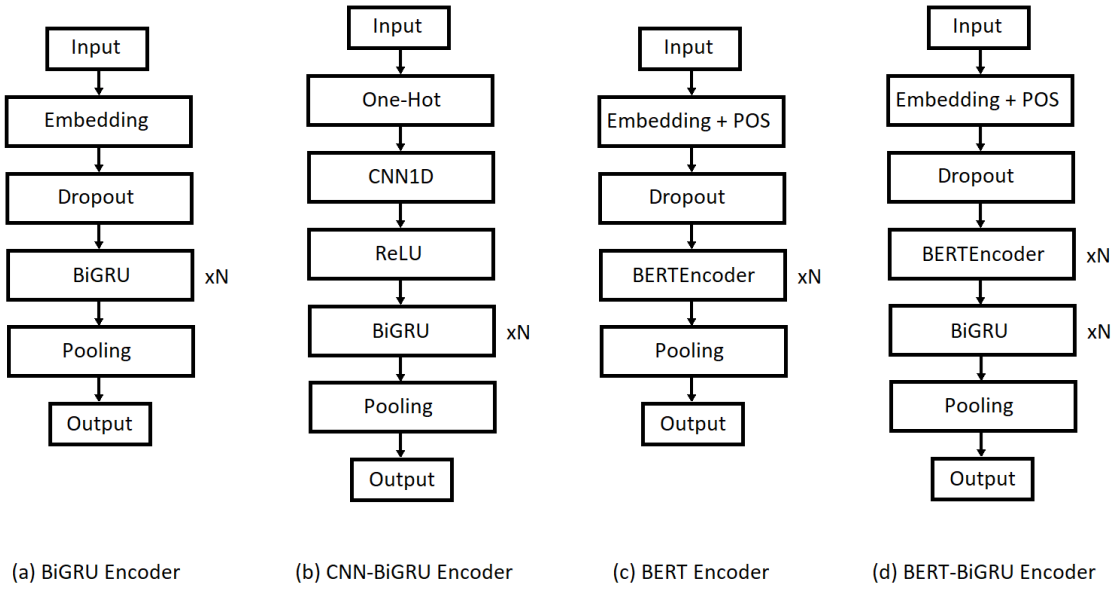


Figure 4.2: Encoder architectures.

The BiGRU Encoder first passes the input through the Embedding layer. It computes the n-dimensional amino acid embeddings based on the amino acid index. It can be viewed as a linear layer assuming the input is a one-hot vector. However, the Embedding layer can leverage that assumption to reduce calculations and discard the padding embedding weights from optimization. The inferred amino acid encodings are then passed through a Dropout Layer to regularize the network. Next, there are N stacked Bidirectional GRU layers with a hidden dimension of size n. Between the BiGRU layers, we also apply Dropout with probability p. Lastly, we perform average-pooling accounting for peptide lengths in the batch.

The CNN-BiGRU Encoder creates one-hot vectors from the lists of amino acid indexes. It then applies in parallel 1D convolutional layers with n filters of sizes 1 and 3. We next concatenate

the outputs from the two CNN1D layers to create $2n$-dimensiotnal amino acid encodings. Before propagating the information to the BiGRU layers, we apply a ReLU activation function to impute non-linearities. The hidden dimension size of the BiGRU layers is n. Finally, we use a Pooling layer as previously described.

The BERT Encoder also passes the input through the Embedding Layer with n-dimensional embeddings. We add a positional encoding (POS) to the embeddings the same way as described in chapter 2.1.6 Attention Mechanism and Transformers. Also, the Bidirectional Transformer Encoder architecture is the same as proposed by the authors of Attention is All You Need [21] depicted in Figure 2.17. Inside the transformer encoder we consider h heads for the multi-head attention mechanism and probability p for Dropout layers. The dimensionality of the hidden linear layer of the transformer encoder is $2n$. We also consider stacking N transformer encoder layers.

Finally, we consider BERT-BiGRU Encoder present in the PepFormer. It has a similar architecture to the BERT Encoder. The only difference is that N BiGRU layers follow a BERTEncoder module. The motivation for this architecture is that the Transformer can capture arbitrarily long dependencies to find global dependencies in the peptide sequence. Furthermore, the multiple heads can find different relations between amino acids in the peptide. We hope that it would enrich the peptide representation and could be beneficial for the following BiGRU network.

4.2 Hyperparameter Optimization

To choose the best architecture, we perform hyperparameter optimization for each proposed encoder using the homo sapiens data set. We split the training data set further to create validation data for the model performance estimation without data leakage from the test set. That results in the 60000 data points for model training and 10000 samples for validation. Furthermore, we use sequential model-based optimization (SMBO) with a Tree-structured Parzen Estimator (TPE) algorithm implemented in the hyperopt library [73] as described in section 2.1.9 Hyperparameter Optimization. We model the hyperparameter space the following way

- encoder output dimension n - qloguniform$(16, 256, 4)$
- number of layers N - quniform$(1, 4, 1)$
- number of heads h - $2^{\text{quniform}(0,2,1)}$
- dropout probability p - uniform$(0, 0.5)$
- batch size - qloguniform$(8, 256, 4)$
- learning rate - loguniform(1e-5, 0.1)
- weight decay - loguniform(1e-5, 1)

As a side note, we change the number of layers to quniform$(1, 2, 1)$ for the BERT-BiGRU Encoder because it has in total 2N layers as we stack both BERTEncoders and BiGRU layers. Here the q in front of the distribution means that we want integer output.

$$\text{qdistribution(low, high, q)} = \text{round}\Big(\frac{\text{distribution(low, high)}}{\text{factor}}\Big) \cdot \text{q} \tag{4.3}$$

The q parameter ensures that the output is divisible by q. It can be convenient for modelling the dimensionality of the encoder output for the BERTEncoder. We need the encoder output to be divisible by the number of heads in the multi-head self-attention mechanism. So as we consider the number of heads in $\{1, 2, 4\}$, we must ensure that the encoder output dimension is divisible by 4. Hence, we set $q = 4$. The benefit of modelling discrete variables using qdistributions instead of categorical distributions is that we can incorporate the underlying smooth objective assumption. It is important to properly model the search space to improve the surrogate model convergence.

We train each model for 20 epochs with 25 evaluation rounds due to the limited computational resources. The optimization process on average takes between 6 to 7 hours for a single architecture on the NVIDIA TESLA P100 Graphical Processing Unit. In Table 4.1 we can see the hyperparameter optimization results.

Encoder	BiGRU	CNN-BiGRU	BERT	BERT-BiGRU
val loss	0.448	0.453	0.502	0.454

Table 4.1: Validation BCELoss for different encoder architectures with the best found hyperparameters.

The model with the BiGRU Encoder was superior to others with 0.448 Binary Cross-Entropy validation loss. Following, both hybrids CNN-BiGRU and BERT-BiGRU show comparable performance. The bidirectional transformer encoder turned out to be the worst in the study. We list the found hyperparameters for the best performing model in Table 4.2.

encoder output dim	number of layers	dropout	batch size	learning rate	weight decay
140	3	0.45	132	4.84e-4	2.9e-5

Table 4.2: Found hyperparameters for the model with BiGRU Encoder.

4.3 Single Network

To evaluate the model on the test set, we train the model on 6700 data points from the training set, leaving the remaining 3000 samples for early stopping. We do so as training to convergence may result in overfitting since we run hyperparameter search training only for twenty epochs due to computational resources. Also, training for longer than twenty epochs may be beneficial for the model performance. We stop training if there was no improvement in the loss function for ten epochs. Furthermore, we save the best model only based on the validation loss. The training of the networks is present in Figure 4.3. For the homo sapiens data set after twenty epochs, we observe the lowest validation loss of 0.429. Further training did not result in validation loss decrease. Training on the mus musculus data follows a similar behaviour with the best validation loss of 0.506 obtained after 23 epochs. The classification metrics achieved on the test set are present in table 4.3.

Data Set	**ACC**	**SP**	**SN**	**AUC**
Homo Sapiens	0.8049	0.7091	0.9002	0.8654
Mus Musculus	0.7549	0.6349	0.8759	0.8159

Table 4.3: Performance metrics evaluated on the mus musculus test data.

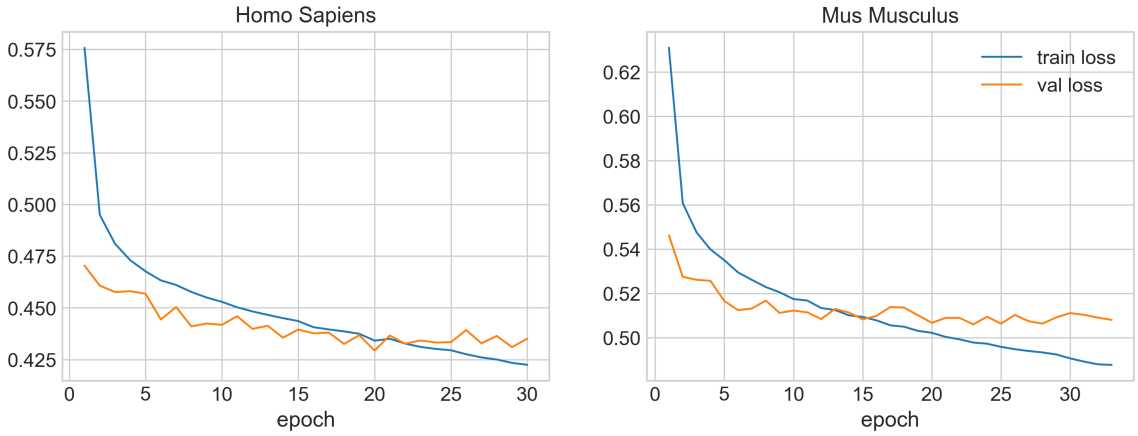


Figure 4.3: Model training.

We classified peptides as detectable if the probability that the peptide is detectable is more than 0.5. Comparing our results to the previous state of the art methods collected in Table 2.2, we outperform all of them on the mus musculus data set. Regarding the homo sapiens data, our model also has the highest AUC score. The PepFormer has better accuracy 80.66% vs 80.49%. What is also worth mentioning is that the authors of the PepFormer monitor the accuracy metric on the test set during training and return the highest seen accuracy. However, we do not use the testing data in any way for the model training. For that reason, the PepFormer may be overfitted to the test set.

Examining the homo sapiens results closer, we can see that based on the specificity and sensitivity, more than 70% of the positive predictions are correct, and 90% of the negative predictions are correct. Therefore, the model is more reliable in predicting undetectable peptides. We could steer this behaviour by considering different classification thresholds, which could also yield higher accuracy. For that reason, the AUC is often a better classifier performance measure.

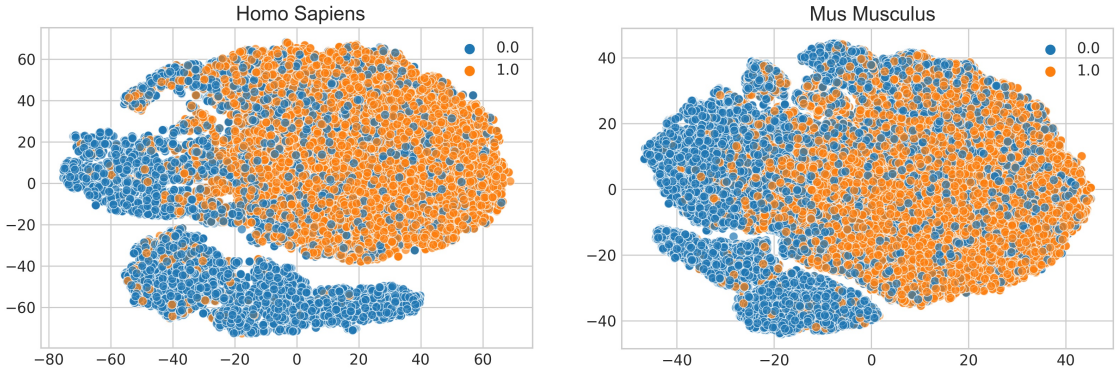


Figure 4.4: The TSN-E visualization of the test peptide embeddings from the BiGRU Encoder trained on the homo sapiens data.

We also test the inter-species transfer learning capabilities. For that reason, we infer the mus musculus peptide embeddings with the model trained on the homo sapiens data. In Figure 4.4 we

show the 2D TSN-E mapping from the 140-dimensional embedding space for two data sets. We can see the distinction between detectable and non-detectable peptides for both homo sapiens and mus musculus. That proves there exist general dependencies in the amino acid sequence responsible for peptide detectability. Also, we calculate the performance metrics for the mus musculus test data obtained from the model trained on the human peptides present in Table 4.4. We can achieve 74.32% accuracy without training on the mus musculus data set. Comparing the model trained on the homo sapiens and mus musculus data, we can see that the performance gap on the house mouse test data is not substantial. That further assures us that the peptide detectability generalizes well between species. Therefore, we could consider merging all the available data for training the peptide detectability prediction models.

ACC	**SP**	**SN**	**AUC**
0.7432	0.5789	0.9074	0.8084

Table 4.4: Performance metrics evaluated on the mus musculus test data after training on the homo sapiens peptides.

4.4 Ensemble

From the model training plots in Figure 4.3 we can see that we could still decrease the training loss. However, the validation loss would not improve further. That is an example of a high variance problem, meaning that the model is over-complex. To benefit from that behaviour, we create an ensemble of six neural networks with different recurrent cells and pooling methods.

$$\text{average pooling: } e_j = \frac{1}{l}\sum_{i=1}^{l} E_{i,j} \tag{4.4}$$

$$\text{max pooling: } e_j = \max_i(E_{i,j}) \tag{4.5}$$

$$\text{last pooling: } e_j = \begin{cases} E_{l,j} \text{ if } j < \frac{l}{2} \\ E_{1,j} \text{ if } j > \frac{l}{2} \end{cases} \tag{4.6}$$

Where $E \in \mathbb{R}^{140\times l}$ is a matrix of amino acid embeddings, $e \in \mathbb{R}^{140}$ is a peptide embedding vector, and l is a peptide length. The last pooling returns a concatenation of the last hidden states from the forward and backward passes of the bidirectional recurrent layer. $E_{i,j<\frac{l}{2}}$ corresponds to the amino acid embedding from the forward pass and $E_{i,j>\frac{l}{2}}$ to the residue embedding of the backward pass. Below we list the permutations of the encoder architectures in the ensemble:

1. GRU cell and average pooling
2. GRU cell and max pooling
3. GRU cell and last pooling
4. LSTM cell and average pooling
5. LSTM cell and max pooling
6. LSTM cell and last pooling

To even more de-correlate the predictors, we overfit the models training for 50 epochs on the whole training set. The ensemble model returns the averaged outputs. We evaluate the ensemble on the test set to acquire the performance metrics listed in Table 4.5.

The ensemble model is superior on all metrics compared to the single network architecture.

Data Set	ACC	SP	SN	AUC
Homo Sapiens	0.8137	0.7198	0.9071	0.8737
Mus Musculus	0.7570	0.6244	0.8906	0.8243

Table 4.5: Ensemble performance metrics evaluated on the test data sets.

Averaging the predictions of several mildly different models increased the accuracy from 80.49% to 81.37% and AUC from 0.8654 to 0.8737 for the homo sapiens data. The metrics for the mus musculus also improved from 75.49% to 75.70% ACC and from 0.8159 to 0.8243 AUC. We can also see that our ensemble model improved the ACC of the runner-up method PepFormer by 0.88% and AUC by 1.12% for the homo sapiens data set. Considering the mus musculus data, we have a 0.65% higher ACC and 1.67% higher AUC than PepFormer. Nevertheless, our model has 0.21% lower specificity for the human peptides and 2.76% lower SP on the house mouse peptides. We also outperform the PepFormer on the sensitivity with 1.75% improvement for the human peptides and 3.21% improvement for the mouse peptides.

We also test the ensemble knowledge transfer between species. As previously, we use a model trained on the human peptides to calculate the performance metrics on the mus musculus test set, see Table 4.6. All metrics improved compared to the single network model. The ACC (0.7505), SP (0.5857), SN (0.9153), AUC (0.8178) increased respectively by 0.98%, 1.17%, 0.87% and 1.16%.

ACC	SP	SN	AUC
0.7505	0.5857	0.9153	0.8178

Table 4.6: Ensemble performance metrics evaluated on the mus musculus test data set after training on the homo sapiens peptides.

In Figure 4.5 we include the results of the single network and ensemble sample efficiency study. To evaluate the single network model, we use 10% of training data for validation purposes. We use the best performing model on the validation data to calculate the accuracy of the held-out test set with 20000 samples. On the other hand, the ensemble model is trained for 50 epochs as previously to ensure mild overfitting of experts. From the accuracy against the number of training samples plot, we can see that the ensemble model consistently outperforms the single network. The most considerable performance gap between the models is apparent for the fewest training data samples. The single network achieved 71.53% accuracy when trained on 1000 samples, while the ensemble model obtained 74.88% accuracy. It is a 4.68% improvement in ACC. The ensemble model averages out the noise. That results in a decrease of the algorithm's variance, substantially improving performance for small data sets.

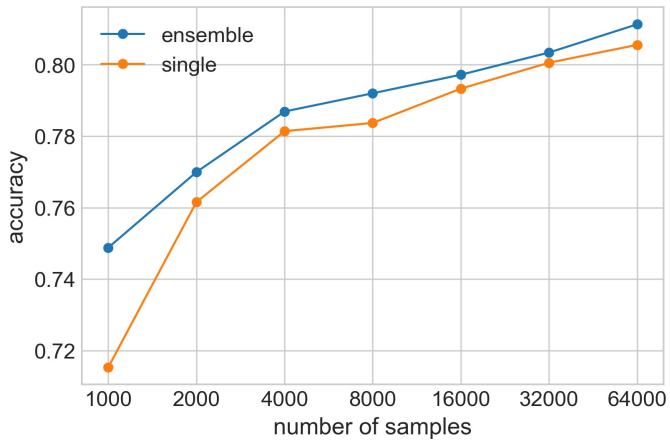


Figure 4.5: Sample efficiency comparison of the single network and the ensemble model. Mind the logarithmic scale.

4.5 Merged Data Sets

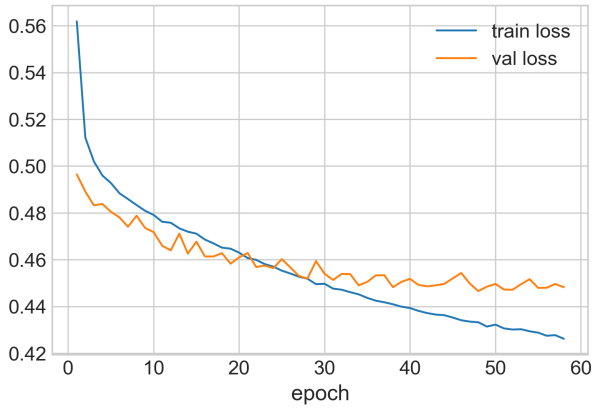


Figure 4.6: Single model training on merged homo sapiens and mus musculus data sets.

From the previous study of inter-species generalization, we know that the peptide detectability is highly transferable between human and house mouse peptides. For that reason, we merge the data sets resulting in 134000 training and 6000 validation samples. We will present the final metrics' estimators on the same held out test data sets as in the previous experiments - 20000 test samples for homo sapiens and mus musculus. The single model training is included in Figure 4.6. The lowest binary cross-entropy validation loss of 0.4465 occurs after the 48th epoch. The resulting test metrics are present in Table 4.7

We can see that training the single model on the merged data sets (Single Merged) considerably improved the single model performance. The accuracy for the homo sapiens and mus musculus increased by respectively 1.03%, 0.50% compared to the previous single network model. The AUC also improved by 1.01% for the human peptides and by 1,2% for the house mouse peptides. The single network trained on the merged data sets also slightly outperforms the previous best

Test Data Set	ACC	SP	SN	AUC
Homo Sapiens	0.8167	0.7294	0.9035	0.8741
Mus Musculus	0.7587	0.6084	0.9101	0.8258

Table 4.7: Performance metrics of the single model trained on the concatenated homo sapiens and mus musculus data sets. The metrics are evaluated on the test data sets.

Ensemble.

We also re-train our ensemble model on the merged data sets, which we refer to as Ensemble Merged. Considering the Figure 4.6, this time, we train each model for 100 epochs to overfit the experts. The resulting performance metrics are gathered in Table 4.8.

Test Data Set	ACC	SP	SN	AUC
Homo Sapiens	0.8245	0.7428	0.9059	0.8863
Mus Musculus	0.7720	0.6293	0.9159	0.8405

Table 4.8: Performance metrics of the ensemble model trained on the concatenated homo sapiens and mus musculus data sets. The metrics are evaluated on the test data sets.

Training the ensemble of experts on the merged data sets improves the ACC by 1.33%, 0.96%, 2.22% and AUC by 1.44%, 1.40%, 2.58% for the human peptides, respectively for the Ensemble, Single Merged, PepFormer. Comparing the results for the mus musculus data set, we have an increase in ACC by 1.98%, 1.75%, 2.65% and AUC by 1.97%, 1.78%, 3.66% respectively for the Ensemble, Single Merged, PepFormer. The sensitivity is above 90% for both test data sets, making the classification of undetectable peptides relatively reliable. On the other hand, only 74.25% (human), 62.96% (house mouse) of the detectable peptides are classified correctly. We also plot the Receiver operating characteristic curves in Figure 4.7. The Ensemble Merged is the best performing model based on the ROC curve and AUC. The following up model is Single Merged.

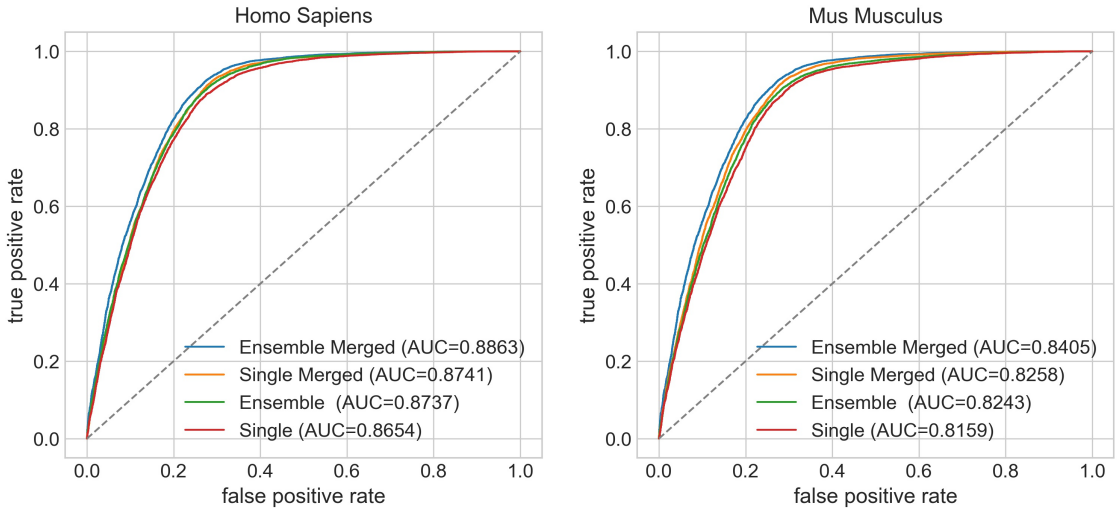


Figure 4.7: Receiver operating characteristic curves for the homo sapiens and mus musculus test data sets.

Chapter 5

Reproducibility Experiments

5.1 Protein Centre Hypothesis

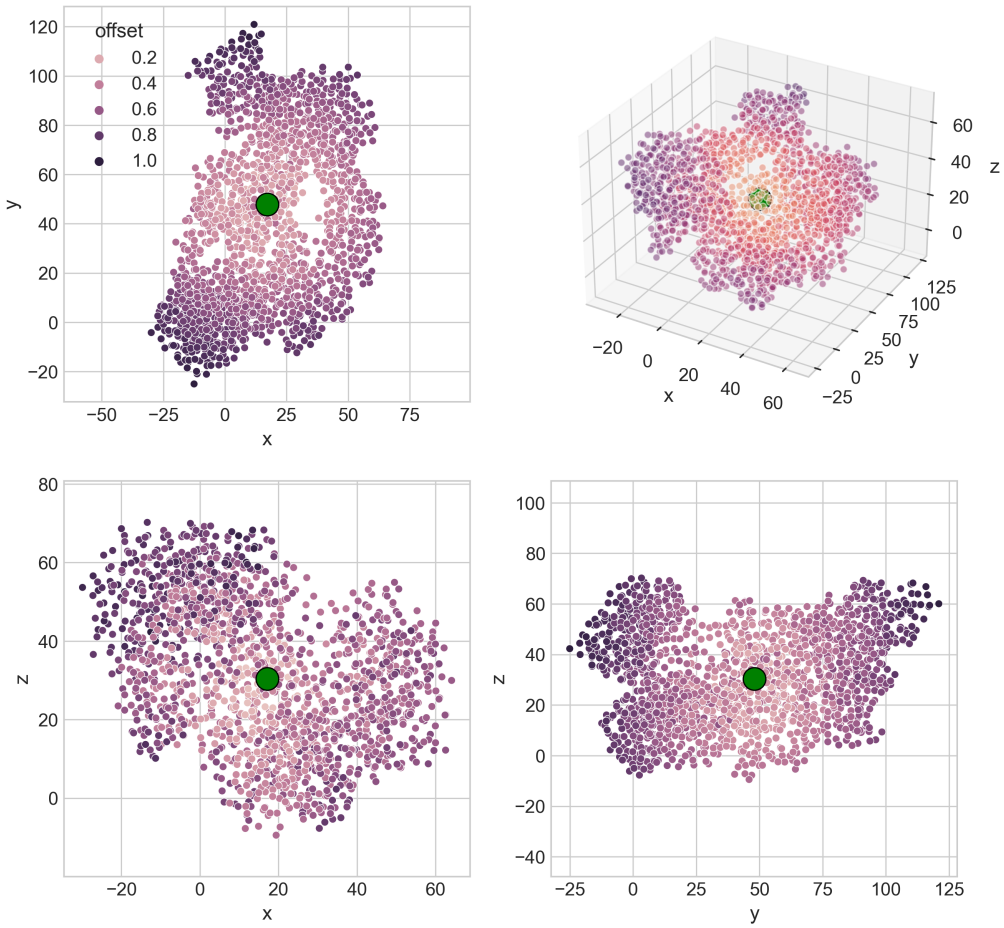


Figure 5.1: Isometric projection of the Complement C3 protein structure with amino acids coloured by the normalized offset from the protein centre (green point).

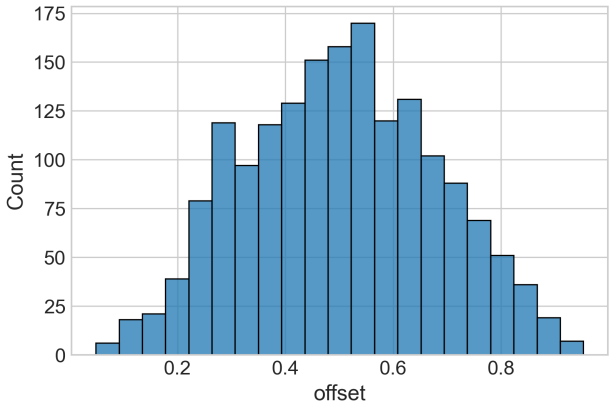


Figure 5.2: Distribution of peptide offset from the protein centre.

In this section, we test Professor Michael MacCoss's hypothesis that peptides further from the protein centre are easier to measure in mass spectrometry experiments. The reproducibility data set provides information about the protein from which a peptide originates. To calculate the peptide offset from the protein centre, we reference the 3D protein structure files from the SwissModel database. The files hold information about the 3D atom coordinates. To find the peptide location in the polypeptide chain, we first need to convert atom locations to amino acid locations. We do so by taking the average of the atom coordinates making up a particular residue. From there, we can find the peptide location in the primary protein structure with sequence matching. We again take the average of the residue locations to represent a peptide location as a single point in space. We calculate the protein centre as

$$c_x = \frac{\max(x) + \min(x)}{2} \tag{5.1}$$

$$c_y = \frac{\max(y) + \min(y)}{2} \tag{5.2}$$

$$c_z = \frac{\max(z) + \min(z)}{2} \tag{5.3}$$

where c is the coordinate vector of the protein centre and x, y, z are vectors of amino acid coordinates in x, y, z directions. Finally, we can compute the peptide offset applying the L2 norm to the difference of the peptide and protein centre coordinate vectors. We additionally normalize the offset by the maximum possible distance from the protein centre, mapping it to the 0-1 scale.

$$\text{offset}(p) = \frac{||p - c||_2}{\max_a(||a - c||_2)} \tag{5.4}$$

Here p is a point coordinate vector, and a is an amino acid coordinate vector. In Figure 5.1, we can see a 2D representation of a 3D protein structure with a marked protein centre and amino acids coloured by the offset as defined above. Proteins often have alternations in their amino acid sequence. For that reason, we were not able to find offsets for all peptides. The matching process reduced the data set to 1728 rows. Please refer to Figure 5.2 to see the offset distribution.

To test Michael MacCoss's hypothesis, we calculate 95% confidence intervals of the linear

regression coefficients for reproducibility prediction. The experiment assumes that if the confidence interval includes 0, there is no statistical evidence that the hypothesis holds. We use a pivotal bootstrap to calculate the confidence intervals described in Algorithm 1 box.

Algorithm 1 Pivotal Bootstrap Interval [74]

Input:
α such that $p(a \leq \theta \leq b) = 1 - \alpha$
n - number of bootstrap rounds
D - data
M - function such that $M(D) = \theta$

Output:
Confidence interval of the form $[a, b]$

Procedure:
$\hat{\theta} = M(D)$
for $i = 1 : n$ **do**
  Sample D^* from D with replacement such that size(D^*) = size(D)
  $\theta_i^* = M(D^*)$
end for
Define $H(r) = p(\hat{\theta} - \theta \leq r),\ \hat{H}^{-1}(r) = \theta_{nr}^* - \hat{\theta}$
$a = \hat{\theta} - H^{-1}(1 - \frac{\alpha}{2}) = 2\hat{\theta} - \theta_{n(1-\frac{\alpha}{2})}^*$
$b = \hat{\theta} - H^{-1}(\frac{\alpha}{2}) = 2\hat{\theta} - \theta_{n\frac{\alpha}{2}}^*$

We consider polynomial features up to and including order 3 to address possible non-linear dependencies. We do fit the intercept but do not calculate confidence intervals for it. The results are present in Table 5.1. From the experiment results, we can see that all confidence intervals include 0. Hence, there is no statistical evidence that the peptide distance from the protein centre influences reproducibility.

Order	**offset**	**offset^2**	**offset^3**
1	[-0.05, 0.07]	-	-
2	[-0.20, 0.40]	[-0.39, 0.19]	-
3	[-1.25, 0.50]	[-0.94, 2.85]	[-2.01, 0.62]

Table 5.1: 95% confidence intervals of polynomial regression coefficients.

5.2 Comparison of Peptide Representations

5.2.1 Experimental Setup

We consider three models: Kernel Ridge Regression (KR), Random Forest (RF), and artificial neural network (ANN). The below algorithm explanations are based on Mark Herbster's Supervised Learning lecture notes [75]. The KR optimizes the mean squared error loss with the L2 penalty.

$$\text{Objective}(w) = (y - Xw)^\top (y - Xw) + \lambda ||w||_2^2 \tag{5.5}$$

Where $y \in \mathbb{R}^{m \times 1}$ is the target vector, $X \in \mathbb{R}^{m \times n}$ is the matrix of input vectors with n features, $w \in \mathbb{R}^{n \times 1}$ is a column weight vector and λ is the L2 regularization parameter. The primal form solution minimizng the above objective is

$$w = (X^\top X + \lambda I_n)^{-1} X^\top y \tag{5.6}$$

where $I_n \in \mathbb{R}^{n \times n}$ is an identity matrix. Now consider that $m < n$, the above solution would be ill-posed. We have a relatively small data set and would like to consider high dimensional feature spaces. For that reason, KR uses a dual representation that is well defined when $m < n$. If we write

$$w = \alpha^\top X \tag{5.7}$$

then we can derive a closed form solution that minimizes the objective.

$$\alpha = (XX^\top + \lambda I_m)^{-1} y \tag{5.8}$$

Finally, the Kernel Ridge predicts with

$$f(x) = w^\top x = \alpha^\top X x \tag{5.9}$$

You can observe that solving for w in the primal form has a time complexity of $O(mn^2 + n^3)$, whereas solving for α has $O(nm^2 + m^3)$. Therefore, it is more efficient to use kernels when $m < n$, which may be the case with our data set. We can further improve the time complexity by exchanging the term $XX^\top$ with a kernel function. In this study, we will consider a polynomial kernel due to its success in NLP tasks [76].

$$K(x, t) = \phi(x)^\top \phi(t) = (x^\top t + 1)^d \tag{5.10}$$

Where d is the polynomial degree. As an example, the second-order polynomial kernel has the following feature map.

$$\phi(x) = [x_n^2, \ldots, x_1^2, \sqrt{2}x_n x_{n-1}, \ldots, \sqrt{2}x_n x_1, \sqrt{2}x_{n-1}x_{n-2}, \ldots, \sqrt{2}x_{n-1}x_1, \ldots, \sqrt{2}x_2 x_1, \ldots, \sqrt{2}x_n, 1]^\top \tag{5.11}$$

Bear in mind that the feature map is not unique. We can always multiply a feature map by an orthogonal matrix U so that

$$(U\phi(x))^\top (U\phi(t)) = \phi(x)^\top \phi(t) \tag{5.12}$$

Now we can see that the dot product of the second-order polynomial feature maps has $O(n^2)$ time complexity, while the kernel evaluation is in $O(n)$.

Another hypothesis space that we consider is partitioning the input space with hyper-rectangles $R_1, \ldots, R_P$, where $\bigcup_{p=1}^{P} R_P = \mathbb{R}^n \wedge R_a \cap R_b = \varnothing$ if $a \neq b$. Then predict with the mean of the

data points in the hyper rectangle.

$$f(x) = \sum_{p=1}^{P} c_p \mathbb{I}[x \in R_p] \tag{5.13}$$

$$c_p = \frac{\sum_{i=1}^{m} y_i \mathbb{I}[x_i \in R_p]}{\sum_{i=1}^{m} \mathbb{I}[x_i \in R_p]} \tag{5.14}$$

To learn the above function we want to solve

$$\min_{R_1,\ldots,R_P} \left\{ \sum_{i=1}^{m} \Big(y_i - \sum_{p=1}^{P} c_p \mathbb{I}[x_i \in R_p] \Big)^2 \right\} \tag{5.15}$$

However, this could be computationally intractable. Therefore, we use a Regression Decision Tree to learn the mentioned function. It iteratively partitions the space onto axis parallel half-spaces $R_1(j,s) = \{x | x_j \leq s\}$ and $R_2(j,s) = \{x | x_j > s\}$ so the objective becomes

$$\min_{j,s} \left\{ \min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right\} \tag{5.16}$$

The complexity of finding the best partitioning on feature j is $O(m)$ as we consider all possible splits between m training data points. Hence, we construct a node in $O(nm)$, as we have n features to consider. Then we recursively apply the above to grow the regression tree. During training, we grow the tree $\hat{T}$ to full depth so that there is a single data point in each leaf node. Later, we use cost complexity pruning (CCP) to prevent overfitting. CCP finds a subtree $T \subseteq \hat{T}$ which minimizes

$$\sum_{p=1}^{|T|} \sum_{x_i \in R_p} (y_i - c_p)^2 + \lambda |T| \tag{5.17}$$

To benefit from the wisdom of the crowd we consider ensembling the regression trees to create a Random Forest Regressor, see Algorithm 2.

Algorithm 2 Random Forest Regressor

Input:
$D = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ - training data
s - number of trees in the forest
k - number of features to consider at each node split
λ - cost complexity pruning parameter

Output:
Ensemble of trees $F(x) := \frac{1}{s} \sum_{i=1}^{s} T_i(x)$

Procedure:
for $i = 1 : s$ **do**
  $D_i :=$ m data points sampled from D with replacement
  $T_i :=$ regression tree trained on D_i considering a subset of k features at each split
  $\text{CCP}_\lambda(T_i)$ apply cost complexity pruning
end for

The ANN has a fully connected architecture with a single hidden layer followed by a ReLU activation function. To train the network, we use mini-batch SGD with Nesterov's momentum and L2 weight penalty.

The procedure for estimating the algorithm's test mean squared error is present in the Algorithm 3 box. Due to the small number of data samples, we perform in a loop model optimization followed by the error estimation on the current test set. Furthermore, we use a sequential model-based optimization (SMBO) with the TPE algorithm as previously. At each of the 30 optimization trials, we estimate the model performance through group cross-validation on the training data. The SMBO procedure present in the Algorithm 3 box returns the model with the best-found hyperparameters trained with the whole training data. Here we consider 5-fold group cross-validation for both testing and optimization loops.

Algorithm 3 Model Testing Procedure

Input:
$D = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ - data set
M - learning algorithm
k_1 - number of group cross-validation folds for the testing loop
k_2 - number of group cross-validation folds for the optimization loop

Output:
Model's test MSE estimate E

Procedure:
for $i = 1 : k_1$ **do**
  $D^{(i)}_{\text{train}}, D^{(i)}_{\text{test}}$:= data D split based on k_1-fold group cross-validation
  $M_i = \text{SMBO}(M, D^{(i)}_{\text{train}}, k_2)$
  $E_i = \text{MSE}(M_i, D^{(i)}_{\text{test}})$
end for
$E = \frac{1}{k_1} \sum_{i=1}^{k_1} E_i$

5.2.2 Results

First, we will see how much variance we can explain with a bag of words representation. For this reason, each peptide is encoded as a one-dimensional vector, discarding information about the residue ordering. We will consider three BOW variations: Counts, Binary and Relative. In the Counts representation, the sequence of amino acids is encoded as a vector where the amino acid occurrence is stored in the vector entry corresponding to that amino acid. The Binary representation instead of counts captures information about the presence of a residue in a sequence. Finally, the Relative encoding is a Counts representation divided by the peptide length to capture the relative presence of amino acids in a peptide, see Figure 2.19. Additionally, we normalise the data (zero mean and unit variance) for the KR and ANN as they use the L2 weight penalty. To prevent data leakage from the test set, we fit the scaler on the training data.

We show the hyperparameter search spaces for KR, RF, ANN respectively in Figures 5.3, 5.4, 5.5. We can see that for the Kernel Ridge, on average, the second-order polynomial kernel degree provided the best performance. Furthermore, we observe that the kernel degree positively

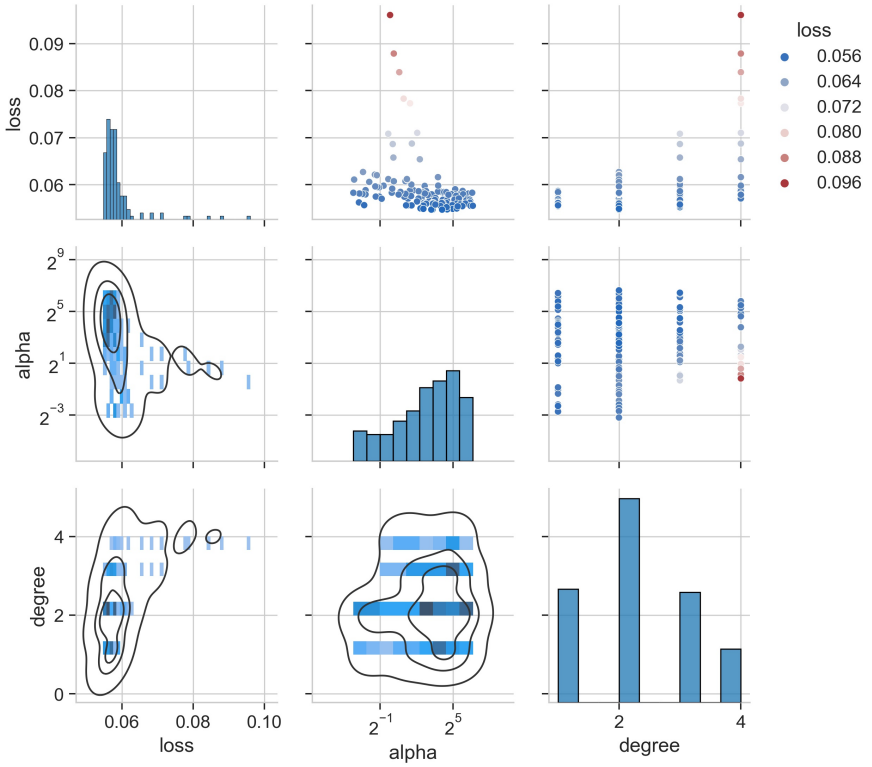


Figure 5.3: Kernel Ridge hyperparameter optimization search space for Counts representation.

correlates with the validation loss variance. The contrary is true for the L2 penalty parameter alpha. Increasing the regularization parameter reduces the algorithm's variance.

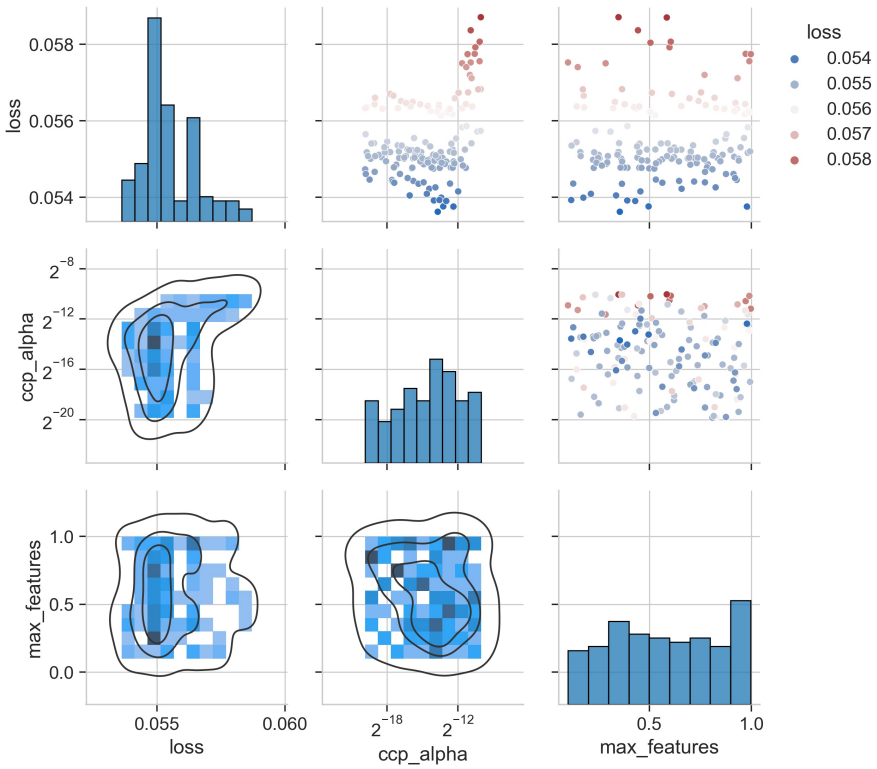


Figure 5.4: Random Forest Regressor hyperparameter optimization search space for Counts representation.

We use a Random Forest Regressor with 300 trees. Additionally, we fine-tune the percentage of features to consider at each node split as well as cost complexity pruning parameter ccp_alpha. We can see a clear minimum in validation loss for ccp_alpha at around 2^{-14}. When the cost complexity pruning parameter is too large, there is an increase in the validation mean squared error. Too large regularization for single trees results in an algorithm underfitting. On the other hand, too low ccp_alpha results in tree overfitting too substantial to be reduced by the ensemble. We could increase the number of trees in the forest to help reduce the variance further. However, it would harm the computational complexity of the algorithm.

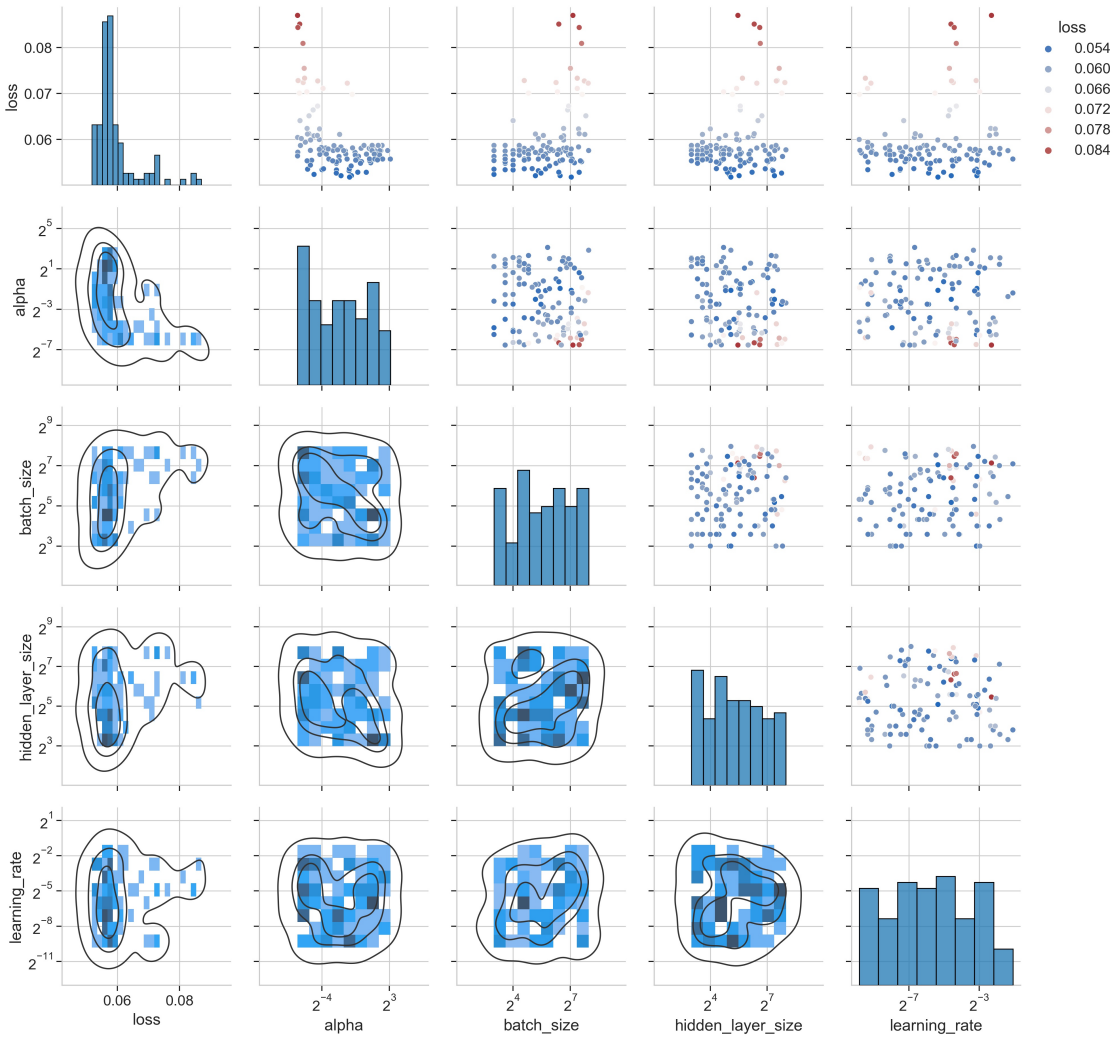


Figure 5.5: Artificial Neural Network hyperparameter optimization search space for Counts representation.

Examining the ANN hyperparameter search space, there is a dependency between the loss and L2 regularization parameter alpha. The optimal value of alpha is around 0.2. Furthermore, we can observe how batch size influences the loss. The larger the batch size, the higher the variance of the algorithm. Using small batch sizes smooths the training objective as the update gradients are too imprecise to converge to a narrow local minimum. Instead, it forces the SGD algorithm to find a wide minimum preventing fitting to the noise. The hidden layer size directly controls model complexity. Again, we can see that the more complex the model, the higher the variance of the loss. However, the best performing hyperparameters tend to be found for relatively large hidden layer sizes with high alpha. That combination ensures that the model can find complex data patterns while preventing overfitting.

Each parameter search loop estimates a mean squared error based on the average of 5-fold group cross-validation results. We perform 30 evaluations for each model. Then we train all the models with the found hyperparameters and evaluate them on the test set. The resulting test

mean squared errors for different algorithms and BOW representations are present in Table 5.2

Model	**Counts**	**Relative**	**Binary**
KR	0.05541 ± 0.00280	0.05605 ± 0.00290	0.05521 ± 0.00366
RF	0.05480 ± 0.00330	0.05472 ± 0.00331	0.05529 ± 0.00378
ANN	0.05188 ± 0.00242	0.05469 ± 0.00338	0.05485 ± 0.00370

Table 5.2: Comparison of test mean squared errors for models trained on different BOW representations.

All combinations of model and BOW representations improved upon the naive baseline with 0.5725 MSE. Artificial Neural Network achieved the best results with 0.05188 average test mean squared error explaining $9.36 \pm 2.32\%$ of the variance. Furthermore, the Counts representation provided the best performance for all models. That may stem from the fact that the Relative encoding discards information about peptide length and Binary only considers whether an amino acid is present in a peptide. To further investigate that hypothesis, we extend the BOW vector with a peptide length and repeat the experiment, see Table 5.3

Model	**Counts+len**	**Relative+len**	**Binary+len**
KR	0.05495 ± 0.00280	0.05483 ± 0.00308	0.05437 ± 0.00329
RF	0.05422 ± 0.00333	0.05456 ± 0.00332	0.05430 ± 0.00342
ANN	0.05202 ± 0.00214	0.05168 ± 0.00264	0.05371 ± 0.00277

Table 5.3: Comparison of the test mean squared errors for models trained on different BOW representations with peptide length feature.

With the peptide length feature, the results for Relative and Binary encodings improved considerably, especially for the ANN model with Relative representation. It decreased the MSE from 0.05469 to 0.05168, being the best in the study and explaining $9.71 \pm 2.74\%$ of the variance. However, bearing in mind the standard deviations, both representations yield similar results.

We also test if pre-trained protein embeddings can be beneficial to peptide reproducibility prediction. For that reason we consider Prot2Vec [43] and BioBERT [49] embeddings. We use a Doc2Vec inspired model with a window size of 3 and stride 1 to infer the 64-dimensional peptide embeddings. The BioBERT returns a 768-dimensional peptide representation. Both embedders were described in section 2.2.1 Feature extraction methods from amino acid sequences. As previously, we test KR, RF and ANN with the above inputs, see Table 5.4

Model	**Prot2Vec**	**BioBERT**
KR	0.05685 ± 0.00363	0.05567 ± 0.00352
RF	0.05674 ± 0.00367	0.05515 ± 0.00353
ANN	0.05689 ± 0.00358	0.05557 ± 0.00345

Table 5.4: Comparison of the test mean squared errors for models trained on different pre-trained embeddings.

The BioBERT embeddings are superior to Prot2Vec with an average MSE of 0.05515 vs 0.05674 for the best performing RF algorithm. However, they did not improve upon the simpler bag of words representation. BioBERT embedding explained $3.75 \pm 0.57\%$ of the variance. Which is almost three times less than the variance explained with the Counts BOW representation.

The last peptide representation that we consider is the one-hot amino acid encoding. It captures sequential information, contrary to BOW vectors. Here we train and optimize BiGRU and CNN-BiGRU recurrent neural networks. The architectures are the same as in the Detectability prediction study and can be seen in Figures 4.2, 4.1. The optimization setup is also similar to the peptide detectability experiments, with the difference that we considering up to three BiGRU layers and from 8 to 128-dimensional encoder output due to the small amount of available data. The testing procedure is the same as explained in Algorithm 3. The results are present in Table 5.5

Model	**One-Hot**
GRU	0.05671 ± 0.00329
CNN-GRU	0.05618 ± 0.00313

Table 5.5: Comparison of the test mean squared errors for models trained on one-hot peptide encoding.

The hybrid CNN-GRU model was superior to the GRU network. The hybrid reduced the 0.05725 baseline MSE by 1.87%. However, it has an 8.28% higher test MSE compared to the ANN with Counts representation. That may be due to the too few data samples being insufficient to capture the sequential information. In theory, the amino acid sequence encodes all peptide properties. Nevertheless, we should gather more data to benefit from the one-hot representation.

5.3 Feature Engineering

Here we augment the peptide Counts representation with additional peptide features. We add peptide detectability probability inferred from our SOTA ensemble model. Additionally, we pass peptides from our data set through AutoRT [67] to acquire peptide retention time predictions. The AutoRT model can predict peptide retention time with 0.47min mean absolute error. We also extend the features by peptide length and transform peptide_mean_1 and peptide_mean_2 as follows

$$\text{peptide\_mean} = \frac{\text{peptide\_mean\_1} + \text{peptide\_mean\_2}}{2} \tag{5.18}$$

$$\text{peptide\_mean\_diff} = \sqrt{|\text{peptide\_mean\_1} - \text{peptide\_mean\_2}|} \tag{5.19}$$

The intuition behind the above transformations is that peptide_mean estimates the peptide abundance in the sample, whereas peptide_mean_diff captures the difference in the peptide abundances between the two machines. The square root operation is used to normalize the feature. Furthermore, we calculate more physiochemical peptide features based on the average amino acid properties with Biopython library [77]. The definitions of calculated peptide properties are listed below.

- gravy - a grand average of hydropathy.
- helix, turn, sheet - the fraction of amino acids which tend to be in helix, turn or sheet secondary protein structure. Amino acids in helix: V, I, Y, F, W, L, turn: N, P, G, S, sheet: E, M, A, L.

- aromaticity - the relative frequency of F+W+Y.
- instability_index - instability index according to Guruprasad et al 1990 [78]. Any value above 40 means the protein is unstable, has a short half-life.
- isoelectric_point - the pH at which a molecule carries no net electrical charge (below - negative, above - positive).
- molar_extinction_coefficient - the molar extinction coefficient assuming cysteines (reduced) and cystines residues.

We will refer to the described set of features as Bio. We use a testing procedure as described in Algorithm 3. The resulting average test mean squared errors for different input sets are present in Table 5.6.

Model	**Counts**	**Bio**	**Counts+Bio**
KR	0.05541 ± 0.00280	0.05057 ± 0.00354	0.05090 ± 0.00311
RF	0.05480 ± 0.00330	0.04984 ± 0.00320	0.04945 ± 0.00271
ANN	0.05188 ± 0.00242	0.05027 ± 0.00353	0.04969 ± 0.00305

Table 5.6: Comparison of the test mean squared errors for models trained on different sets of features

We can see that the selected Bio features improve the performance of all models compared to the Counts representation. The test MSE loss decreased by 8.73%, 9.09%, 3.10%, respectively, for the KR, RF and ANN. What is more, combining Counts and Additional feature sets further improved the performance of RF and ANN. The Random Forest Regressor has the best performance in the study, explaining 13.76% of the variance with combined Counts and Bio feature sets.

We will further investigate the models by plotting the feature importances, see Figures 5.6, 5.7, 5.8. We use the permutation feature importance algorithm present in Algorithm 4 box to estimate the feature importances. We use explained variance score (EV) for S calculated as

$$\mathrm{EV}(\hat{y}, y) = 1 - \frac{\mathrm{Var}(\hat{y} - y)}{\mathrm{Var}(y)} \tag{5.20}$$

Furthermore, for each of the five train/test data splits from group cross-validation, we consider $K = 10$ random permutations. Finally, we aggregate the results to acquire 50 importance score estimates.

First, we will examine the feature importances for models trained on the Counts feature set shown in Figure 5.6. The amino acid L has the highest importance score for all models. Following in the decreasing importance order are amino acids F, I, V, H. Randomly permuting the amino acid L count feature reduces the explained variance by 12.6% for ANN. Notice that the explained variance by the ANN trained on the peptide Counts representation is 9.36%. Therefore, the random permutation of that feature results in the model worse than the baseline predicting with the mean.

In Figure 5.7 we can observe how the peptide retention time dominates over the other Bio features. The Random Forest Regressor heavily relies on the retention time followed by peptide mean and mean diff. The KR and ANN also find peptide length and helix relevant. These algorithms use the L2 penalty of weights' magnitude resulting in a smoother importance distribution

Algorithm 4 Permutation Feature Importance

Input:
$D = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ - data set where $x_i \in \mathbb{R}^n$
M - fitted model
S - scoring function
K - number of repetitions

Output:
Feature importances i.

Procedure:
$s = S(M, D)$
for $j = 1 : n$ **do**
  for $k = 1 : K$ **do**
    $D_{j,k} :=$ data set D with randomly shuffled feature j.
    $s_{j,k} = S(M, D_{j,k})$
  end for
  $i_j = s - \frac{1}{K} \sum_{k=1}^{K} s_{j,k}$
end for

across features comparing to RF. We can also see that all models did not find peptide detectability informative. However, keep in mind that permutation importances only describe feature relevancy to a model given all other variables.

Finally, we can see the feature importance distribution across models trained with Counts and Bio feature sets in Figure 5.8. Given the peptide retention time and peptide mean variables, the amino acid importance scores decreased considerably. The best performing RF almost solely relies on peptide retention time, peptide mean and mean diff. We can conclude that the reproducibility of mass spectrometry experiments depends on peptide retention time. For that reason, we plot the reproducibility against the peptide retention time in Figure 5.9

In the peptide detectability prediction study, we showed that ensembling results in a better sample efficiency. For that reason, we consider stacking several experts for peptide reproducibility prediction. We use the top two hyperparameter configurations for KR, RF, ANN found previously through SMBO, resulting in 6 experts. That procedure should de-correlate the models and improve the ensemble performance. What is more, we predict with a weighted average of experts. To find the relevant expert weights, we fit a Ridge Regression with a cross-validated L2 penalty parameter. The L2 penalty will smooth the weights' distribution to ensure that all experts are relevant to the final prediction. We evaluate the ensemble model with group-cross validation. The ensemble model resulted in a 0.04863 ± 0.00225 test MSE being the best in the study. It reduces the baseline MSE by 15.06% explaining $15.15 \pm 2.06\%$ of the variance.

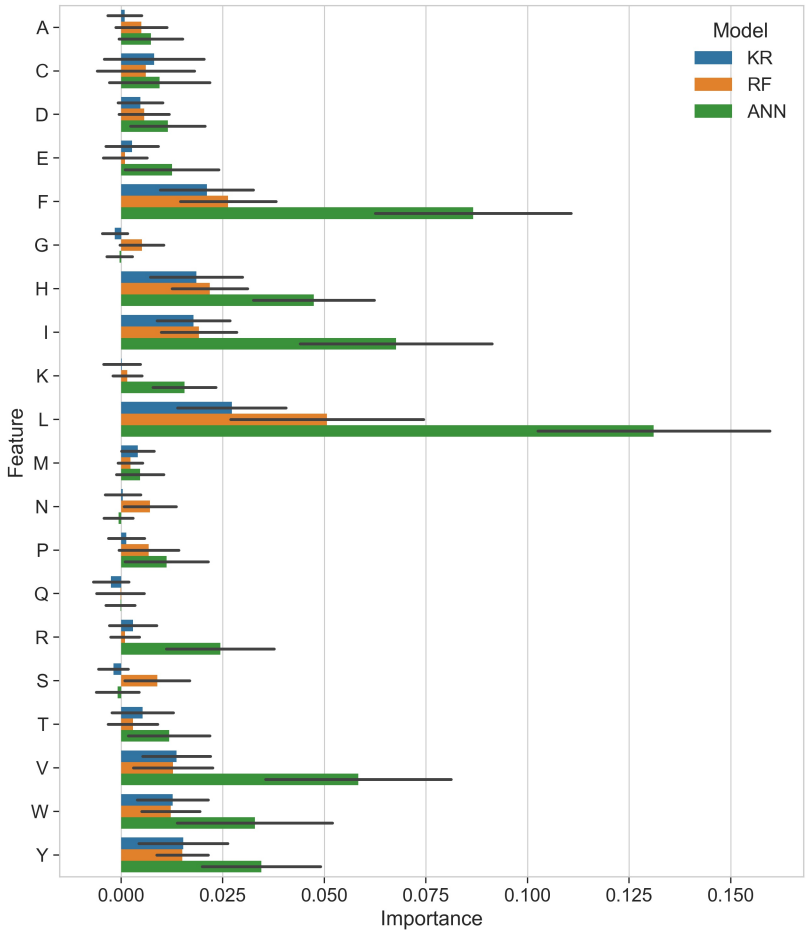


Figure 5.6: Feature importances for KR, RF and ANN models trained with Counts peptide representation. The bars show the mean importance score, whereas the error bars correspond to the standard deviations.

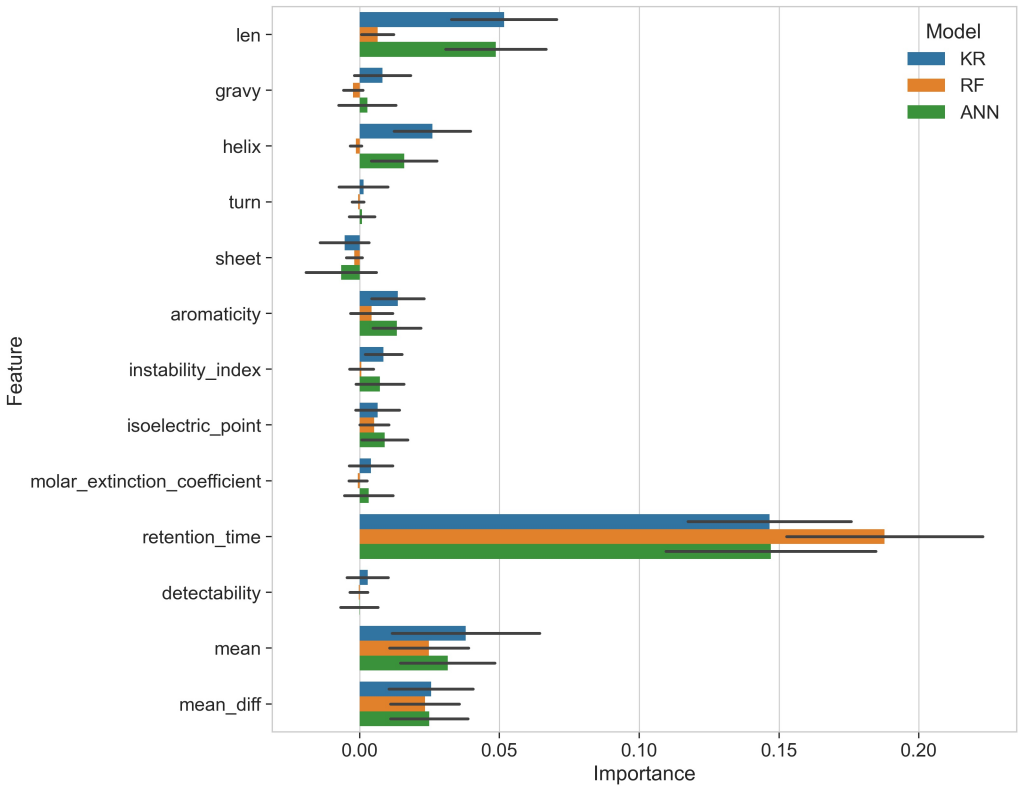


Figure 5.7: Feature importances for KR, RF and ANN models trained with Bio peptide feature set. The bars show the mean importance score, whereas the error bars correspond to the standard deviations.

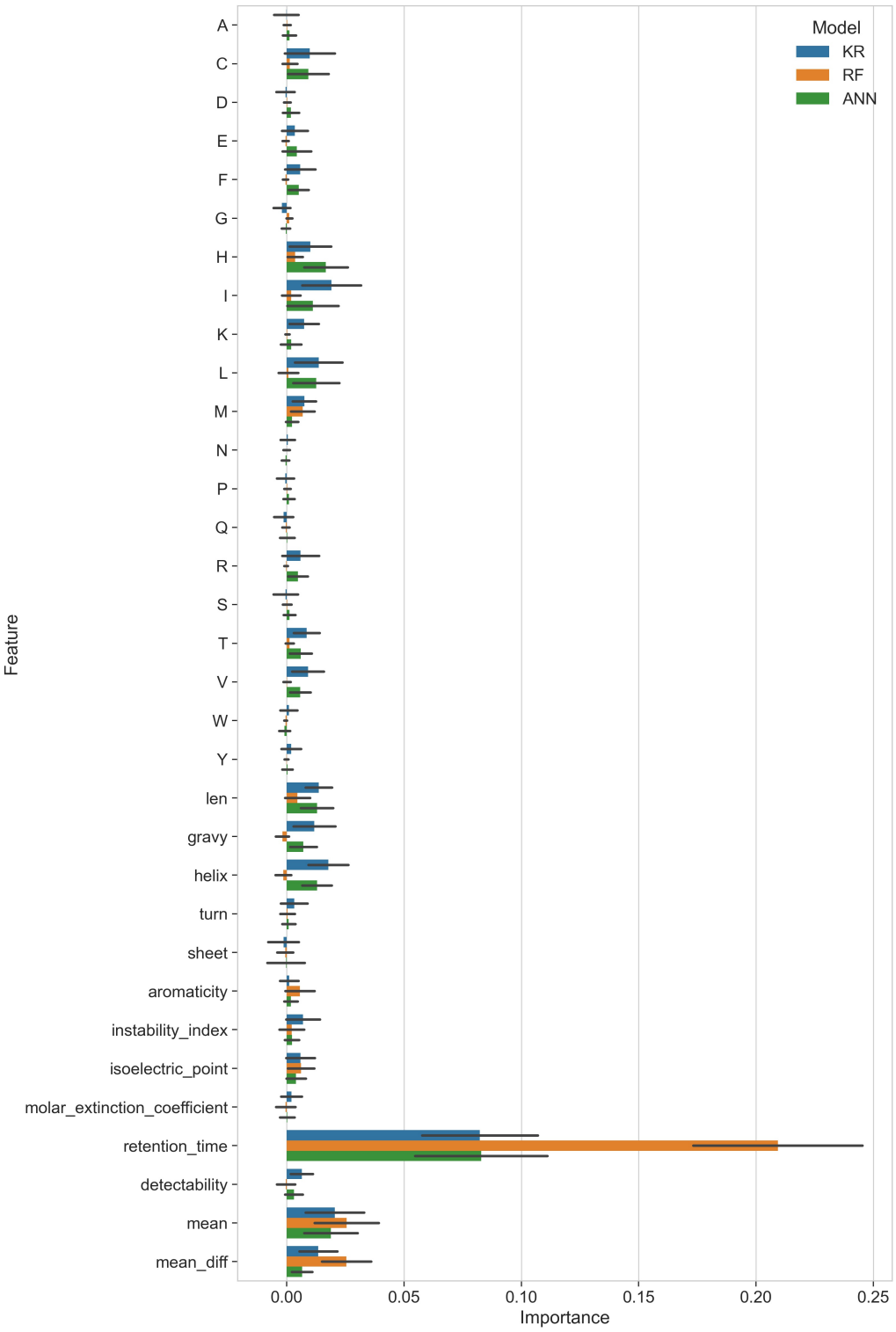


Figure 5.8: Feature importances for KR, RF and ANN models trained with Counts and Bio feature sets. The bars show the mean importance score, whereas the error bars correspond to the standard deviations.

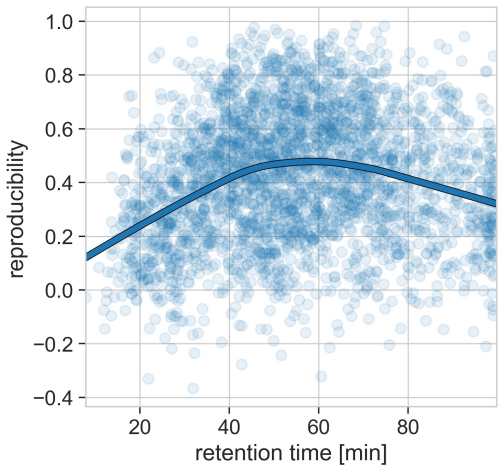


Figure 5.9: The connection between peptide retention time and reproducibility. The present line shows a locally weighted linear regression fit.

Chapter 6

Conclusions

In this thesis, we focus on improving the reliability of mass spectrometry experiments. We provide an overview of the most recent deep learning techniques for working with sequential data such as amino acid chains. Furthermore, we include the literature review of the state of the art machine learning methodologies for different problems in proteomics. Finally, we address peptide detectability and results' reproducibility in shotgun proteomics.

Our main contribution is improving upon the PepFormer model for homo sapiens and mus musculus peptide detectability prediction, respectively from 80.66% to 82.45% and from 75.21% to 77.20%. To our knowledge, the proposed ensemble of recurrent neural networks is the new state of the art method. We first considered four peptide encoder architectures: BiGRU, CNN-BiGRU, BERT, BERT-BiGRU. Later, the hyperparameter optimization with the Tree-Structured Parzen Estimator algorithm found the model with BiGRU encoder as best performing. To reduce the variance of the algorithm, we proposed an ensemble de-correlating the models by training to overfit and introducing alternations to the architecture. The study of peptide embedding vectors and inter-species generalization showed that the peptide detectability is highly transferable between homo sapiens and mus musculus peptides. Thus, to further improve the performance, we merged the data sets and re-trained the models. Future research could consider training the models on all available data for different species. Moreover, extensive hyperparameter optimization should improve upon our solution as we had limited computational resources. Another approach might be to consider training the models in a triplet network setup.

Another contribution is disproving the hypothesis that the relative peptide distance from the protein centre influences the reproducibility of shotgun proteomics experiments. We calculated normalized peptide offset from the protein centre referencing external databases and estimated confidence intervals of linear regression coefficients with polynomial offset features. All confidence intervals included 0, showing no statistical evidence in favour of the proposed hypothesis.

We also compared different peptide representation methods, including a bag of words variations, contextual embeddings and one-hot peptide encoding. Our results show that the vector of amino acid counts outperforms other encoding methods for peptide reproducibility data set. Moreover, we calculate other physiochemical properties from the amino acid sequence to extend our input representation, improving the predictive power of our models. The study of feature importance showed that there is a possible link between inferred peptide retention time and reproducibility

of MS results. Finally, we create an ensemble of Kernel Ridge, Random Forest, and Feedforward Neural Network to explain $15.15 \pm 2.06\%$ of the variance estimated with group cross-validation.

We should notice that the peptide retention time is calculated based on the amino acid sequence. Hence, there exist sequential dependencies that we were unable to capture with the limited number of data samples. For that reason, we could consider training a model for retention time prediction, which we might use for transfer learning. We should also keep in mind that the reproducibility data set derives from only two mass spectrometers. We can never assume that both machines were working perfectly, so we should further investigate our findings on data sets with better statistical significance. There is also a question of whether the retention time influences reproducibility directly or indirectly.

Bibliography

[1] Anon, 2021 proteins. https://bio.libretexts.org/@go/page/5309. Accessed: 2021-05-29.

[2] Neil G. Rumachik, Stacy A. Malaker, and Nicole K. Paulk. Vectormod: Method for bottom-up proteomic characterization of raav capsid post-translational modifications and vector impurities. *Frontiers in Immunology*, 12:830, 2021.

[3] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[4] AG Ivakhnenko and GA Ivakhnenko. The review of problems solvable by algorithms of the group method of data handling (gmdh). *Pattern recognition and image analysis c/c of raspoznavaniye obrazov i analiz izobrazhenii*, 5:527–535, 1995.

[5] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[6] Corinna Cortes and Vladimir Vapnik. Support vector machine. *Machine learning*, 20(3):273–297, 1995.

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

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[9] Berghout tarek, 2021. basic learning rules for rosenblatt perceptron. https://www.mathworks.com/matlabcentral/fileexchange/71948-basic-learning-rules-for-rosenblatt-perceptron. Accessed: 2021-08-26.

[10] Stefan Strauß. From big data to deep learning: A leap towards strong ai or 'intelligentia obscura'? *Big Data and Cognitive Computing*, 2(3), 2018.

[11] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[12] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[13] Anh H. Reynolds. Convolutional neural networks. `https://anhreynolds.com/blogs/cnn.html`, 2019.

[14] Christopher Olah. Conv nets: A modular perspective. `https://colah.github.io/posts/2014-07-Conv-Nets-Modular/`, 2014.

[15] Christopher Olah. Understanding lstm networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[16] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[17] Manuel P Cuéllar, Miguel Delgado, and MC Pegalajar. An application of non-linear programming to train recurrent neural networks in time series prediction problems. In *Enterprise information systems VII*, pages 95–102. Springer, 2007.

[18] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[20] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[22] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[24] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Proceedings of the 32nd international conference on neural information processing systems*, pages 2488–2498, 2018.

[25] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[26] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[27] Sida Wang and Christopher Manning. Fast dropout training. In *international conference on machine learning*, pages 118–126. PMLR, 2013.

[28] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. *Advances in neural information processing systems*, 28:2575–2583, 2015.

[29] Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout. *arXiv preprint arXiv:1705.07832*, 2017.

[30] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066. PMLR, 2013.

[31] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 648–656, 2015.

[32] Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[33] Taesup Moon, Heeyoul Choi, Hoshik Lee, and Inchul Song. Rnndrop: A novel dropout for rnns in asr. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 65–70. IEEE, 2015.

[34] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118*, 2016.

[35] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. *Advances in neural information processing systems*, 29:1019–1027, 2016.

[36] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady an ussr*, volume 269, pages 543–547, 1983.

[37] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

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

[39] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[40] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[42] Ehsaneddin Asgari and Mohammad RK Mofrad. Continuous distributed representation of biological sequences for deep proteomics and genomics. *PloS one*, 10(11):e0141287, 2015.

[43] Learned protein embeddings for machine learning. *Bioinformatics*, 34(15):2642–2648, 2018.

[44] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[46] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[47] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[48] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[49] Roshan Rao, Nicholas Bhattacharya, Neil Thomas, Yan Duan, Xi Chen, John Canny, Pieter Abbeel, and Yun S Song. Evaluating protein transfer learning with tape. *Advances in Neural Information Processing Systems*, 32:9689, 2019.

[50] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. Dilated residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 472–480, 2017.

[51] Tristan Bepler and Bonnie Berger. Learning protein sequence embeddings using information from structure. *arXiv preprint arXiv:1902.08661*, 2019.

[52] Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

[53] Ethan C Alley, Grigory Khimulya, Surojit Biswas, Mohammed AlQuraishi, and George M Church. Unified rational protein engineering with sequence-based deep representation learning. *Nature methods*, 16(12):1315–1322, 2019.

[54] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*, 2017.

[55] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[56] Elizabeth Guruceaga, Alba Garin-Muga, Gorka Prieto, Bartolome Bejarano, Miguel Marcilla, Consuelo Marín-Vicente, Yasset Perez-Riverol, J Ignacio Casal, Juan Antonio Vizcaíno, Fernando J Corrales, et al. Enhanced missing proteins detection in nci60 cell lines using an integrative search engine approach. *Journal of proteome research*, 16(12):4374–4390, 2017.

[57] Guillermo Serrano, Elizabeth Guruceaga, and Victor Segura. Deepmspeptide: peptide detectability prediction using deep learning. *Bioinformatics*, 36(4):1279–1280, 2020.

[58] Hao Cheng, Bing Rao, Lei Liu, Lizhen Cui, Guobao Xiao, Ran Su, and Leyi Wei. Pepformer: End-to-end transformer-based siamese network to predict and enhance peptide detectability based on sequence only. *Analytical Chemistry*, 93(16):6481–6490, 2021.

[59] Siegfried Gessulat, Tobias Schmidt, Daniel Paul Zolg, Patroklos Samaras, Karsten Schnatbaum, Johannes Zerweck, Tobias Knaute, Julia Rechenberger, Bernard Delanghe, Andreas Huhmer, et al. Prosit: proteome-wide prediction of peptide tandem mass spectra by deep learning. *Nature methods*, 16(6):509–518, 2019.

[60] Oleg V Krokhin, Robertson Craig, Vic Spicer, Werner Ens, Kenneth G Standing, Ronald C Beavis, and John A Wilkins. An improved model for prediction of retention times of tryptic peptides in ion pair reversed-phase hplc: its application to protein peptide mapping by off-line hplc-maldi ms. *Molecular & Cellular Proteomics*, 3(9):908–919, 2004.

[61] Shivani Tiwary, Roie Levy, Petra Gutenbrunner, Favio Salinas Soto, Krishnan K Palaniappan, Laura Deming, Marc Berndl, Arthur Brant, Peter Cimermancic, and Jürgen Cox. High-quality ms/ms spectrum prediction for data-dependent and data-independent acquisition data analysis. *Nature methods*, 16(6):519–525, 2019.

[62] Shenheng Guan, Michael F Moran, and Bin Ma. Prediction of lc-ms/ms properties of peptides from sequence by deep learning. *Molecular & Cellular Proteomics*, 18(10):2099–2107, 2019.

[63] Robbin Bouwmeester, Ralf Gabriels, Niels Hulstaert, Lennart Martens, and Sven Degroeve. Deeplc can predict retention times for peptides that carry as-yet unseen modifications. *BioRxiv*, 2020.

[64] Woochul Kang and Jaeyong Chung. Deeprt: predictable deep learning inference for cyber-physical systems. *Real-Time Systems*, 55(1):106–135, 2019.

[65] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. *arXiv preprint arXiv:1710.09829*, 2017.

[66] Yi Yang, Xiaohui Liu, Chengpin Shen, Yu Lin, Pengyuan Yang, and Liang Qiao. In silico spectral libraries by deep learning facilitate data-independent acquisition proteomics. *Nature communications*, 11(1):1–11, 2020.

[67] Bo Wen, Kai Li, Yun Zhang, and Bing Zhang. Cancer neoantigen prioritization through sensitive and reliable proteogenomics analysis. *Nature communications*, 11(1):1–14, 2020.

[68] Kathryn Tunyasuvunakool, Jonas Adler, Zachary Wu, Tim Green, Michal Zielinski, Augustin Žídek, Alex Bridgland, Andrew Cowie, Clemens Meyer, Agata Laydon, et al. Highly accurate protein structure prediction for the human proteome. *Nature*, pages 1–9, 2021.

[69] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[70] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Kathryn Tunyasuvunakool, Olaf Ronneberger, Russ Bates, Augustin Žídek, Alex Bridgland, et al. High accuracy protein structure prediction using deep learning. *Fourteenth Critical Assessment of Techniques for Protein Structure Prediction (Abstract Book)*, 22:24, 2020.

[71] Deepmind, 2020 alphafold: a solution to a 50-year-old grand challenge in biology. `https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology`. Accessed: 2021-06-07.

[72] Robertson Craig, John P Cortens, and Ronald C Beavis. Open source system for analyzing, validating, and storing protein identification data. *Journal of proteome research*, 3(6):1234–1242, 2004.

[73] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[74] Anthony Christopher Davison and David Victor Hinkley. *Bootstrap methods and their application*. Number 1. Cambridge university press, 1997.

[75] Mark Herbster. Lecture notes in supervised learning, 2020.

[76] Yin-Wen Chang, Cho-Jui Hsieh, Kai-Wei Chang, Michael Ringgaard, and Chih-Jen Lin. Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, 11(4), 2010.

[77] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 03 2009.

[78] Kunchur Guruprasad, BV Bhasker Reddy, and Madhusudan W Pandit. Correlation between stability of a protein and its dipeptide composition: a novel approach for predicting in vivo stability of a protein from its primary sequence. *Protein Engineering, Design and Selection*, 4(2):155–161, 1990.