Machine Learning from Evolution

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Oskar Taubert

Tag der mündlichen Prüfung: 9. Februar 2024

1. Referent/Referentin: Prof. Dr. Achim Streit
2. Referent/Referentin: Prof. Dr. Alexander Schug

Declaration

Erklärung zur Selbstständigen Anfertigung der Dissertationsschrift
Hiermit erkläre ich, dass ich die Dissertationsschrift mit dem Titel

Machine Learning from Evolution

selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich oder inhaltlich übernommenen Stellen Stellen als solche kenntlich gemacht und die Regeln zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT) beachtet habe.

Ort und Datum Oskar Taubert

Abstract

This dissertation broadly draws inspiration from biology for solving issues in the rapidly developing field of machine learning in the natural sciences. Nature and biology have long served as foundation for models and algorithms in computer science. Well-known examples include artificial neural networks, the attention mechanism, or population based optimization. In the opposite direction, biology has seen rapid progress driven by high throughput data acquisition methods and the surge in powerful machine learning methods enabled by them. Evolution in particular provides a mechanism that has been adapted to a family of optimization algorithms in computer science, but it also leaves behind machine-learnable patterns in biological records.

Adaptation of machine learning techniques primarily developed for default applications like image or language processing, to natural sciences poses diverse challenges, but have been recently successful in solving long standing problems. Tailoring the model architecture and inductive biases to the specific problem at hand promises greater performance but often requires expert domain knowledge and specialized model components. One area where machine learning has driven rapid advance is bio-molecular structure prediction. This advance has been enabled by the algorithmic advances at the intersection of image processing, language processing and geometric learning, by the rapid growth of public databases, and finally by the availability of compute resources.

While these recent successes have been relegated to areas, where labeled training data is relatively ubiquitous, there are many, where labels or data in general are sparse. One such example is structured ribonucleic acids. There are few resolved structures in the public databases, since their structural resolution remains challenging. While the datasets are expected to grow, experiments will remain expensive and time consuming and thus data efficiency is a valuable model property. My contribution for better contact prediction models as a proxy for structure prediction is a composite model consisting of two modules. The first module encompassing the bulk of the model parameters is pre-trained in a self-supervised manner, in order to utilize as much information from unlabeled data as possible. The second is a combination of shallow artificial neural network and classical random forest model that further processes the latent representations of the bulk model, since it is more robust than a neural network when faced with limited training data.

Another problem, especially present in the sparse data regime, is the biases present in the distribution of the data. While there are more subtle phenomena, the most easily visible one is class imbalance. Often, the less interesting background classes are overrepresented in comparison to the interesting foreground or anomalies. This imbalance in particular has an effect on the optimization landscape that is traversed during training of a neural network. To manipulate the optimum the model finally converges on, I dynamically deform this landscape by scheduling from one training loss to another.

Finding suitable architectures or other parameters governing the training of a model is a growing problem as models become increasingly complex and training more expensive. To search these parameter spaces efficiently, evolution can be leveraged. Evolutionary optimization is a well established approach, however it has several properties that make it useful for modern neural architecture search. Its population based nature has inherent potential for parallelization, that makes it scalable to modern large scale computing infrastructure and its sampling mechanism does not only forego utilization of gradients but also distances altogether, which allows searching categorical parameter spaces. I introduce a new communication scheme based on lazy synchronization to further adapt evolutionary optimization to neural architecture search in high performance computing environments.

Zusammenfassung

Diese Dissertation beschäftigt sich mit biologisch inspirierten Methoden für das sich rapide entwickelnde Feld des maschinellen Lernens angewendet auf die Naturwissenschaften. Die Natur und insbesondere die Biologie dienen seit langem als Grundlage für Modelle und Algorithmen in der Informatik. Bekannte Beispiele sind künstliche neuronale Netze, der Attention Mechanismus, oder populationsbasierte Optimierungsverfahren. Im Gegenzug hat die Biologie in verschiedenen Bereichen große Fortschritte durch moderne Hochdurchsatzverfahren und die dadurch ermöglichten Datenanalysemethoden erfahren. Insbesondere das Konzept der Evolution bietet einen Mechanismus der für ganze Familien von Optimierungsverfahren adaptiert wurde, aber sie hinterlässt auch maschinenlernbare Muster in evolutionsbiologischen Datenbeständen.

Die Anpassung von Techniken des maschinellen Lernens, die für Standardanwendungen wie Bild- oder Sprachverarbeitung entwickelt wurden, an naturwissenschaftliche Fragestellungen wirft eine Reihe von Herausforderungen auf, erlaubte aber auch in jüngster Zeit lange zuvor ungelösten Problemen zu lösen. Modelarchitekturen und den enthaltenen induktiven Bias auf spezifische Probleme maßzuschneidern erlaubt eine bessere Leistung des Modells, setzt üblicherweise aber Expertenwissen und angepasste Modellkomponenten voraus. Ein beispielhaftes Gebiet der Biologie, auf dem maschinelles Lernen großen Fortschritt ermöglicht hat, ist die biomolekulare Strukturvorhersage. Dieser Fortschritt wurde ermöglicht durch algorithmische Entwicklungen an der Schnittstelle zwischen Bildverarbeitung, Sprachverarbeitung, und geometrischem Lernen, durch das rasante Wachstum von öffentlichen Datenbanken und nicht zuletzt durch die Verfügbarkeit von großen Mengen an Rechenressourcen.

Während diese Fortschritte sich auf Gebiete beschränken, in denen die erwünschten Vorhersageergebnisse für die Trainingsdaten bereits in großer Menge bekannt sind, gibt es viele, in denen Beispielergebnisse oder Trainingsdaten insgesamt knapp sind. Ein solches Beispiel sind funktionale Ribonucleinsäuren mit assoziierter dreidimensionaler Struktur. Es gibt nur wenige experimentell aufgelöste Strukturen in öffentlichen Datenbanken, da die entsprechenden Experimente aufwändig und teuer sind. Obwohl diese Datensätze im Laufe der Zeit anwachsen sollten, werden Strukturdaten ein limitierender Faktor bleiben und die effiziente Datennutzung wird weiterhin eine notwendige Modelleigenschaft sein. Mein Beitrag für eine bessere Kontaktvorhersage als Proxy für Strukturvorhersage ist ein Kompositmodell bestehend aus zwei Modulen. Das erste, das den Großteil der Modellparameter enthält, wird selbstüberwacht auf Gensequenzen trainiert, so dass die vorhandenen Daten ohne assoziierte Strukturen genutzt werden können. Das zweite Modul benutzt anschließend die latenten Repräsentationen des vorigen Moduls um die eigentliche Vorhersage zu produzieren. Es besteht aus einem flachen neuronalen Netz und einem robusten, klassischen Random Forest, die gemeinsam die wenigen Strukturdaten effizient nutzen können.

Ein weiteres Problem, das oft insbesondere in datenarmen Szenarien auftaucht, ist ein Bias in der Verteilung von Trainingsdaten. Während es subtilere Phänomene gibt, ist eine einfach zu beobachtende Eigenschaft das Klassenungleichgewicht. Die weniger interessanten Hintergrundklassen sind oft überrepräsentiert im Vergleich zu dem interessanteren Vordergrund oder gesuchten Anomalien. Dieses Ungleichgewicht hat insbesondere einen Einfluss auf die Optimierungslandschaft, die während des Trainings eines neuronalen Netzwerks durchquert wird. Um das Optimum auf dem das Modell letztendlich konvergiert zu manipulieren, verzerre ich diese Landschaft, indem ich verschiedene Trainingssignale dynamisch miteinander kombiniere.

Das Finden von geeigneten Architekturen oder anderen Parametern, die das Training eines Modells beeinflussen, ist ein an Relevanz zunehmendes Problem, da Modelle komplexer und Training ressourcenhungriger werden. Evolution bietet einen Ansatz, um diese Parameterräume effizient zu durchsuchen. Evolutionäre Optimierung ist ein etabliertes Vorgehen, es hat jedoch einige Eigenschaften, die es besonders geeignet für neurale Architektursuche machen. Die populationsbasierte Basis hat inhärentes Potenzial für Parallelisierung, was gute Skalierungseigenschaften auf modernen großskaligen Recheninfrastrukturen verspricht. Ausserdem verläßt sich der Navigationsmechanismus nicht auf Gradienten oder Distanzen im Suchraum, und ist daher geeignet sowohl kategorische als auch kontinuierliche Parameter zu verarbeiten. Ich stelle ein neues Kommunikationsprotokoll und darauf angepasste Algorithmen vor, die auf asynchronem Austausch zwischen einzelnen Evaluations- und Trainingsschritten basieren, um populationsbasierte Optimierung für neurale Architektursuche auf Hochleistungsrechnern anzupassen.

Acknowledgements

This work would not have been possible without a host of people, who I would like to thank here.
First and foremost my thanks goes to to my supervisors Achim Streit and Alexander Schug. Their support and advice over the last six years made all of this possible.
Thank you to my family. In particular Martha and Sebastian.
Thank you to the members of the Research Group Multi-Scale Biomolecular Simulation and the NIC Research Group Computational Structural Biology at JSC, past and present, that I shared some time with: Claude Sinner, Ines Reinartz, Mehari Zerihun, Arthur Voronin, Linda Neubrand, Lars Franke, Momin Ahmad, Fabrizio Pucci, Emanuel Peters, Julian Herold, Fathia Idiris, Christian Faber, Eric Behle, and Ali Sharifian.
Thank you to Max Fischer, Eileen Kühn, Peter Krauß and all the staff at SCC who kept the machines livable and the offices running.
Thank you to Terry Cojean, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Fritz Göbel and Hartwig Anzt for all the hikes and discussion.
Thank you to everyone at Helmholtz AI Energy: Daniel Coquelin, Anis Farshian Abbasi, James Kahn, Katharina Flügel, Mishal Benz, and Juan Pedro Gutiérrez Hermosillo Muriedas.
Thank you to my future colleagues at RAI: Charlotte Debus, Arvid Weyrauch, David Li, Nicholas Kiefer, Jonas Dressner, and Deifilia To for making me stay on.
Thank you to my coauthors and collaborators Fabrice von der Lehr, Alina Bazarova, Philipp Knechtges, and Stefan Kesselheim. Without your help BARNACLE would still be unfinished.
Thank you to Max Neukum, Alex Neukum, Ewa Götz and all the RPG people. All the joy and escapism you provided would have been missed.
Thank you to Ina Rottmann.
Thank you Yin-Chen Hsieh, for being my friend and helping me hold on to a vestige of a semblance of sanity. Most of all, thank you Markus Götz and Marie Weiel. Your example and friendship is an ever present inspiration and without it, I would have given up a long time ago.

List of Publications

This thesis contains work previously published in peer reviewed venues.

- **Oskar Taubert**, Fabrice van der Lehr, Alina Bazarova, Christian Faber, Marie Weiel, Philipp Knechtges, Charlotte Debus, Daniel Coquelin, Achim Basermann, Achim Streit, Stefan Kesselheim, Markus Götz, and Alexander Schug, 2023. *RNA Contact Prediction by Data Efficient Machine Learning. (Communications Biology)*, doi: 10.1038/s42003-023-05244-9.

- **Oskar Taubert**, Markus Götz, Alexander Schug, and Achim Streit, 2020. *Loss Scheduling for Class-Imbalanced Image Segmentation Problems. (ICMLA)*, doi:10.1109/ICMLA 51294.2020.00073.

- **Oskar Taubert**, Marie Weiel, Daniel Coquelin, Anis Farshian, Charlotte Debus, Alexander Schug, Achim Streit and Markus Götz, 2023. *Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations. (ISC)*, doi:10.1007/978-3-031-32041-5_6.

While conducting my doctoral studies, I also contributed to several other publications, that will not be discussed in this thesis.

- **Oskar Taubert**, Ines Reinartz, Henning Meyerhenke, and Alexander Schug, 2019. *diSTruct v1.0: generating bioloecular structures from distance constraints. (Bioinformatics)*, doi:10.1093/bioinformatics/btz578

- James Kahn, Ilias Tsaklidis, **Oskar Taubert**, Lea Reuter, Giuilio Dujany, Tobias Boeckh, Arthur Thaller, Pablo Goldenzweig, Florian Bernlochner, Achim Streit, and Markus Götz, 2022. *Learning tree structures from leaves for paticle decay reconstruction. (MLST)*, doi:10.1088/2632-2153/ac8de0

- James Kahn, **Oskar Taubert**, Ilias Tsaklidis, Lea Reuter, Giulio Dujany, Tobias Boeckh, Arthuar Thaller, Pablo Goldenzweig, Florian Bernlochner, Achim Streit, and Markus Götz, 2022. *Lowest Common Ancestor Generations (LCAG) Phasespace Particle Decay Reconstruction Dataset.* doi:10.5281/zenodo.6983258

Contents

1. Introduction

1.1. Motivation

Molecular biology is the study of life on the level of biological macromolecules in terms of their chemical and physical properties. Detailed understanding of molecular origins and function is the foundation of progress in a wide range of fields like medicine, chemical engineering, or material science. Supplementing and augmenting experimental and theoretical biology, data science and simulation have been established as a third pillar of scientific endeavor. This has lead to the establishment of bioinformatics as its own discipline at the intersection between biology and computer science. High throughput experiments producing terabytes of data necessitate efficient computational methods in their analysis and interpretation. One example is molecular structure prediction. High throughput sequencing made entire genomes cheaply accessible, but the three dimensional structures of the derived macromolecules are still only determined through expensive and time consuming experiments. Modern deep learning models applied to all the available data have enabled prediction of protein structures from their genetic sequences. The function of these molecules, that act as nano-scale machinery, governing life on a cellular level, is tightly linked to their structure. Hence, progress in the understanding of molecular structure enables further research into many cellular and also industrialized processes. The inverse problem, protein design, is equally aided by machine learning.

In Return, computer science draws inspiration from nature. Processes optimized through evolution over millennia are often a reasonable first approach for a given problem with some potential for adaptation. Artificial neural networks and population based optimizers are just two examples of leveraging biological understanding for building general models in computer science. However, there is a second mechanism inspiring generalizable problem solving. Sometimes technical solutions for specific domain problems show utility beyond their originally intended application. Sequence processing algorithms or statistical principles and heuristics for the systematic, quantitative study of populations have emerged in biological use cases before being propagated through computer science in general. In this vein, we wanted to develop machine learning tools for the investigation of biological problems. We can then leverage the findings of these investigations to design improved computational tools in general and in particular for machine learning. The central biological application we focus on is molecular contact map prediction from genetic sequences for ribonucleic acids. Contact map prediction serves as a proxy for full structure prediction. This constraint is given primarily by the extreme sparsity of labels used to train supervised machine learning models, which is a problem hampering many scientific deep learning applications. More technical problems encountered here are class imbalance and model calibration. Since contact maps are a sparsified and lossy representation of structure they contain little signal and a lot of background. This influences the resulting model trained with such labels. Mitigating resulting harmful biases while preserving the interpretability of its prediction, e.g. by maintaining its calibration, is an open problem. Finally, determining the non-learnable parameters of deep learning models is its own area of machine learning research.

We explored further measures to maximize data efficiency, which are technically available to other domains, like self-supervised learning, and composite deep and traditional models. Relatively deep neural networks can be trained as self-supervised models using the larger amounts of unlabeled data. The representations such a model generates can then be used by a traditional machine learning model, in this example a gradient boosted forest classifier, better than the usually used neural network. The drawback is that the large self-supervised model cannot be fine-tuned by backpropagating the error through the entire composite model. To address this shortcoming, we fine-tune the neural network using a small surrogate network standing in for the forest model [1]. To traverse the loss landscape in search for an optimum resulting in a well calibrated model, we introduce loss scheduling [2]. Varying different loss contributions over the course of training of a neural network allows for the trading off of the benefits and drawbacks of different loss functions. To perform hyperparameter search, we adapt the well established evolutionary optimization to the HPC setting [3], where large scale machine learning takes place. Minimizing synchronization, we dispense with the notion of generations as synchronization barriers and instead adopt a more flexible and efficient communication scheme.

1.2. Research Questions

The work presented in this thesis tries to address and is guided by several central questions. The problem posed at the root is that of structure prediction of RNA molecules.

- How can modern deep learning be applied to improve RNA structure prediction?

In the pursuit of a solution to this problem a series of follow-up questions arises:

- How can the extreme data sparsity for RNA be addressed?
- How can evolutionary information be leveraged to maximize the exploitable information content of the available data?
- When using contact prediction as a proxy for full structure prediction, how can the class imbalance problem be addressed?
- Does the combination of multiple self-supervised upstream tasks improve downstream performance and model robustness?
- Can the use of classical machine learning methods improve data efficiency?
- How can classical machine learning methods be combined with modern deep neural networks?

- How can evolution be leveraged to find suitable combinations of model and training parameters?
- How can high performance computing environments be exploited for hyperparameter search?

1.3. Contributions

Here we give a brief overview over the individual contributions making up this thesis. Molecular structure prediction is a problem in molecular biology [7], where the three dimensional atomic structure of a molecule is to be determined from easier to access information, commonly genetic sequences. In the complete prediction pipeline a model solving this problem receives a genetic sequence as input and generates a point cloud with the coordinates of the atoms making up the molecule. For one class of biomolecules, proteins, a range of end-to-end models solving this problem have been recently introduced [8, 9, 10].

For others, like ribonucleic acid (RNA), training these types of end-to-end models is not feasible, since there is several orders of magnitude less data available, requiring a more complex workflow. Instead of end-to-end structure prediction, contact prediction [11, 12] is used as a proxy, in order to simplify the problem. For each possibly interacting pair of building blocks in the sequence, the model predicts, whether they are in proximity in the 3D structure or not. All predictions taken together form a contact map, that can serve as an input to bias a downstream modeling algorithm. Contact prediction thus transforms the problem to a binary semantic segmentation problem.

To use the still small amount of available data most efficiently, we combine self-supervised pre-training of artificial neural networks with decision forests trained with extreme gradient boosting. The neural network extract patterns from the unannotated data that serve as more useful input features for the tree model. The forest model uses the labeled data more efficiently than a neural network would. Even though this composite model is not end-to-end trainable, it performs the best out of an unsupervised baseline model and different types of neural networks [1], almost doubling Matthew's correlation between prediction and label compared to a less sophisticated baseline.

Class imbalance is a problem for classification tasks [13], not only when evaluating the model using metrics based on assumptions of class balance, but also for training itself, since strongly underrepresented classes are sometimes ignored by the model entirely. Using resampling techniques to balance the classes do not use the scarce data efficiently and are also not trivially implementable for segmentation problems. For the aforementioned contact map prediction the foreground class is strongly underrepresented. Using different loss functions from the default cross-entropy [14], e.g. focal loss, can produce better results, with the trade-off that they are not necessarily proper scoring functions, i.e. we have to expect the resulting model to be not well calibrated. Since the predictions of the model are only one step in a complex process, interpretable results are relevant. To train well calibrated neural networks on class imbalanced segmentation datasets, we use loss scheduling. For this chapter, we examine established semantic segmentation datasets [15, 16, 17]. The training begins with a better performing loss function in terms of the target metric and is gradually scheduled towards a proper scoring function. Models trained in this way converge consistently on a different local minimum in the properly scoring loss landscape for a semantic image segmentation problem [2].

Training deep, artificial neural networks, in particular large language models, consumes larger and larger amounts of compute resources in terms of time and energy [18, 19, 20]. This does not only limit training of the final model, but also the exploration of the space of possible model configurations. During this hyperparameter optimization (HPO) or neural architecture search (NAS) many model candidates have to be at least partially trained. Evolutionary optimization is an established, gradient free, population based method for global optimization [21]. The gradient free aspect makes this optimization method suitable for NAS and the population based aspect makes it easier to parallelize. With the rise of of deep learning, NAS grows both in relevance and resource consumption. For evolutionary optimization, a population of candidates is evaluated. Since all candidates can be treated independently, this can be done in parallel. After evaluation selection, crossover, and mutation generate a new population. Especially in the case of NAS each candidate evaluation might take very different amounts of time and workers idle while waiting for the last evaluation in one population to complete. We proposed a lazy synchronization and breeding algorithm using soft generations, that produces new candidates to evaluate ad-hoc using the currently available set of already evaluated candidates. It compares favorably against a widely used competitor both in terms of performance and runtime [3].

1.4. Outline

This thesis is structured as follows:

- Chapter 1: This chapter. Outlines motivation, research questions, and methodology of this thesis.
- Chapter 2: Provides background and theoretical basis of the relevant aspects from both biology and computer science.
- Chapter 3: Presents my work on RNA contact prediction by data efficient deep learning, an example of a data sparse machine learning problem.
- Chapter 4: Presents my work on loss scheduling, a technique for manipulating the traversal of the loss landscape during training of a neural network.
- Chapter 5: Presents my work on high performance computing adapted population based hyperparameter optimization.
- Chapter 6: concludes this thesis and presents an outlook for future avenues of investigation.

2. Background

This chapter introduces the background from biology and machine learning, that we build on in later chapters. The relevant biological aspects include terminology and concepts from molecular biology and evolution in particular. we also discuss some bioinformatics concepts as they touch on sequence processing and structure determination. The discussed machine learning background includes neural networks, hyperparameter optimization and high performance computing.

2.1. Biology

2.1.1. Evolution and Molecular Biology

The theory of evolution describes the adaptation of populations of organisms to environmental pressures posed by their environment over the course of generations [22]. Individuals better adapted to their environment have better chances of survival and more opportunity to procreate – they are "fitter". Apart from the environment, the other variable going into the adaptation procedure is the initial setup of the organism itself. In a simplified view, this setup is given by the genes the organism inherited from its parents. The organisms' genes serve as the blueprints for the machinery and infrastructure it uses to interact with its environment and itself. Genes are encoded on sequential chains of molecules, deoxyribonucleic acid (DNA), comprised of a limited alphabet of building blocks. Figure 2.1 and fig. A.1 show the nucleic acid and protein alphabets, respectively. Genes in the genome, the collection of all genes of the organism, can be read, copied, and executed by other complex molecules present in the organism, which are themselves described by their respective genes. Because of their sequential nature and finite set of recurring building blocks, DNA sequences (and similar biological sequences) are often represented and digitally processed simply as strings of letters. To synthesize these molecular machines and others that are necessary or useful to the survival of the organism on a cellular level, the DNA blueprint is first transcribed to ribonucleic acid (RNA) fig. 2.2 for transport. The transcribed RNA can then be optionally modified during post-processing and/or translated to a protein. Proteins are, next to the aforementioned RNA and DNA, another class of biological macro-molecules. Other major groups of biomolecules include carbohydrates and lipids, which are not discussed further here. Like DNA or RNA macro-molecules, proteins are formed by linear chains of smaller building blocks from an alphabet of (canonically) twenty different amino acids fig. 2.2. Proteins perform a wide range of functions, like catalysis, signaling, and providing structure. However, protein blueprints make up only a small part [24] of an entire genome. A larger part of the identified genes implement regulatory instructions, or their function is entirely unknown.

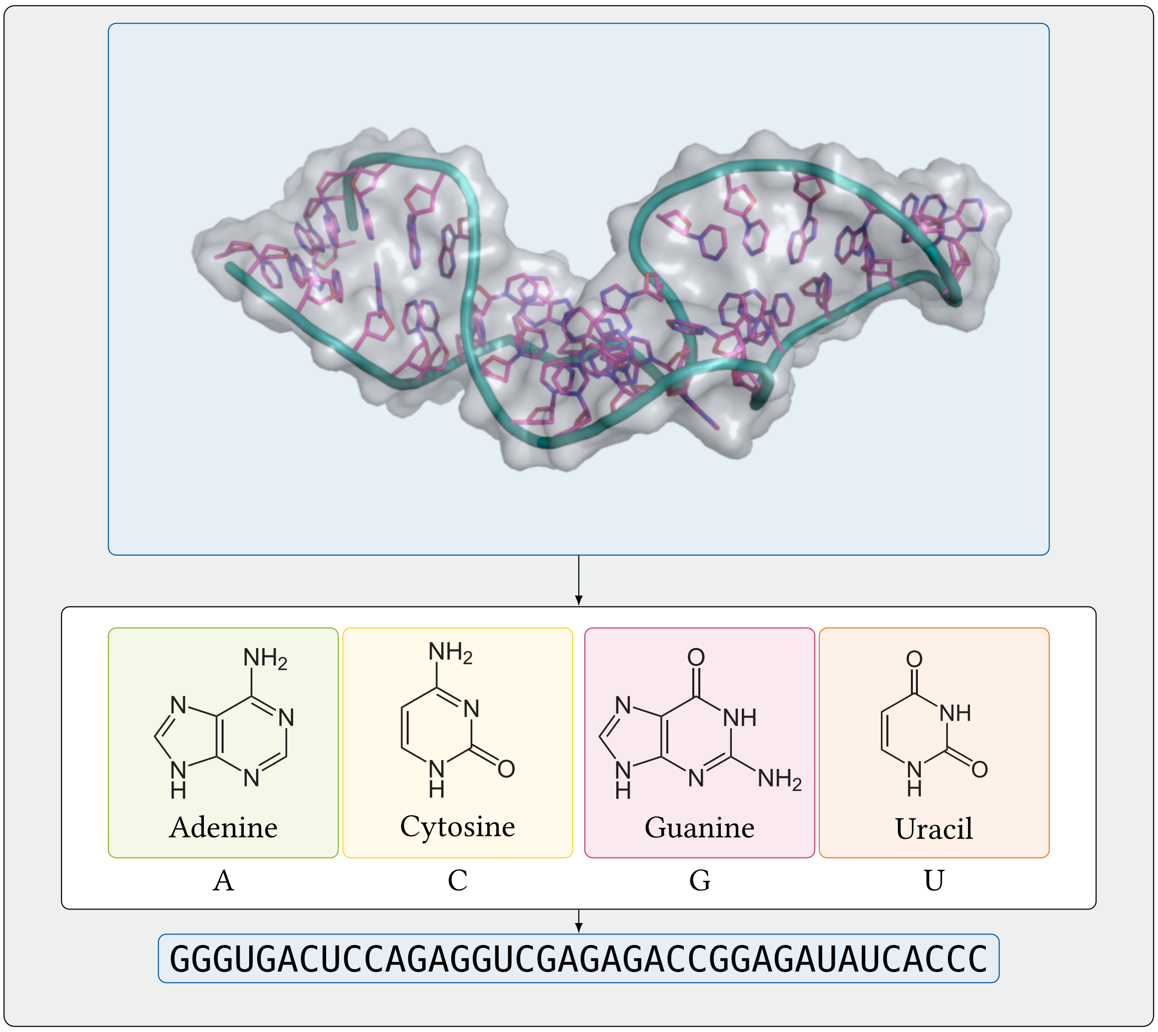


Figure 2.1.: In yellow: Structure and sequence of `1a9l` [23]. Green, red, blue, and orange: RNA residues (bases) and their corresponding tokens or letters. Bases in the structure are shown in purple.

2.1.2. Molecular Sequence, Structure, and Function

All processes that when combined enable life on the molecular level involve molecular machinery. Like machines on the macroscopic scale, the function and use of these microscopic machines is tightly coupled to their structure. However and perhaps most remarkably, unlike their macroscopic counterparts, proteins and functional RNA are typically at least partially self-assembling: Their genetic code does not only describe the ingredients that are required to build component parts, but this sequence contains sufficient information to reversibly self-assemble into a specific functional 3D-shape.

Up to this point we avoided making sweeping statements while maintaining a high level of abstraction. In the following we will employ some simplifications for illustrative examples. A life-cycle of a protein might look as follows: as response to a cellular signal, its

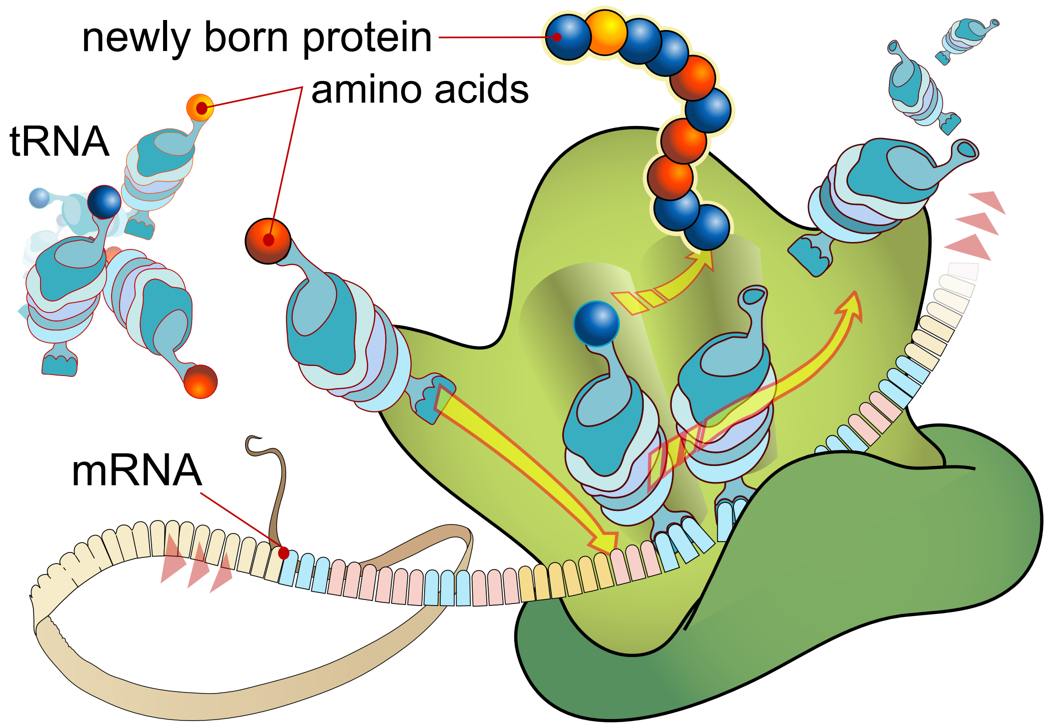


Figure 2.2.: Protein synthesis in the ribosome (green) translating an mRNA sequence into an amino acid sequence using the translation matrix implemented by tRNA. Adapted from wikimedia [25].

gene is transcribed into the corresponding RNA strand by a polymerase, another protein. The transcript is then transported to the ribosome, a complex made up of protein and RNA, where the sequence is read and translated three RNA bases (called a "codon") to one amino acid at a time, producing a protein chain. As the protein is synthesized it folds into the structure through interaction with the surrounding environment. Once the protein is fully assembled it can perform its function, until it is recycled into its component parts, which are returned to the ribosome, where they can be used to build new proteins. DNA can usually be thought of in terms of just their sequence. Their sequence stores information, which can be read and executed by more actively functional molecules, like RNA and protein. For a protein catalyst for example, the folded structure is central to their function, the reactants have to fit into binding pockets or be able to reach the active site of the catalyst for it to function correctly. One example for a functional RNA is transfer RNA (tRNA), which ferries the amino acids to the ribosome, where it performs the protein translation process. It needs to selectively bind to a triplet of RNA bases on one end and to the corresponding amino acid on the other (cf. fig. 2.2), while holding it in such a way that the ribosome can link it to the emerging protein chain. In general the sequence, structure, and function of molecules are tightly coupled. A detailed understanding of a molecules structure usually precedes detailed understanding of its function. Efforts furthering this understanding are mostly driven by two practical motivators next to scientific curiosity: dysfunction, i.e. disease and molecular design. Understanding of the function of a molecule involved in a disease enables the development of treatment. Molecular design on a larger scale, i.e. starting with a function and finding a corresponding sequence, enables the design of biochemical processes[26].

Putting this in terms of data structures, a genetic sequence can be represented by a string of tokens over an alphabet of monomers or residues, while the three dimensional structure can be represented by a point cloud, or a graph, if including chemical information.

Returning to evolution, sequences have to be copied at many stages throughout the processes making up life. While there are mechanisms in place preventing or correcting mistakes, when they do happen, mutations can occur, and an altered sequence can propagate. Mutations and constraining pressure from the environment are the drivers of evolution. Over the course of generations genomes evolve and separated populations diverge into different species [27], as these mutations can be beneficial or detrimental in the context of the organisms environment.

2.1.3. Structure Determination

In this context, structure determination describes the process of determining the three dimensional point cloud of the atomic coordinates of a biological macromolecule. It encompasses experimental structure determination and computational structure prediction. Experimental structure determination includes X-ray crystallography [28], nuclear magnetic resonance (NMR) spectroscopy [29, 30], and cryo-electron microscopy [31]. All of these require extensive sample preparation and are expensive and lengthy. At the same time high throughput sequencing techniques have made genetic sequences abundant. To bypass the often involved and expensive experimental structure determination and to further the understanding of molecular structure and function, predicting a molecular structure directly from its genetic sequence has been the goal of computational biology for a long time [7]. Early attempts would utilize a molecular dynamics (MD) simulation based on physical principles, where the atoms constituting the molecule and the solvent were subjected to the forces from chemical bonds and electrostatic interactions [32]. In addition to MD simulations there are also monte-carlo driven approaches, which neglect the dynamics of the folding process and search the folded structure by search for an optimum in the free energy landscape of the molecule [33]. Molecular simulations are computationally expensive and involve a trade-off between runtime and accuracy. Capturing the entire folding process in an ab initio simulation is still only possible for fast folding proteins on specialized hardware [34].

If the approximation of the force field and the environment is good enough and given enough time MD simulations can yield insight not only in the structure, but also the folding and interaction dynamics of the molecule in question. However these two requirements are limiting the efficacy of molecular dynamics as a structure prediction tool. The fastest process to simulate, the vibration of hydrogen atoms, occurs on the scale of femtoseconds, which determines the time step of the simulation. Folding processes can take up to seconds, which sets the average number of required time steps to 10^{15}. Even with approximations enabling larger time steps makes the computational cost prohibitive. Coarse-graining models [35] to increase the size of time steps also makes parameterizing a force field more difficult.

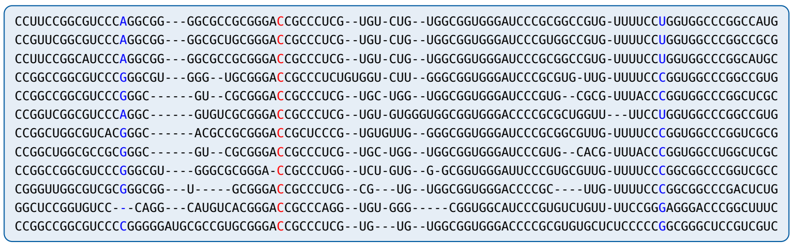
More recently, data driven approaches have enabled advances in structure prediction. In turn, these were enabled by the ubiquity of both sequence [36, 37, 38] and structure databases [39].

Contact prediction predicts partial structural information as a proxy for full structure prediction usually as one part in a larger software pipeline [12, 40, 41]. In a contact map, the ij-th entry indicates whether residue i and residue j are in spatial contact, akin to an adjacency matrix. Contact maps are an edge case of discretized distance maps with just two bins. Predicting full contact maps can be considered the precursor to structure prediction, as its output was initially intended to bias e.g. simulations [42] for more efficient sampling.

Mutual Information

The simplest approach we present here is co-evolutionary mutual information (MI). The underlying assumption of MI for contact prediction is, that function, structure, and sequence of a molecule are subject to evolutionary pressure. Mutations that are deleterious are unlikely to be present in the evolutionary record. If e.g. the interaction between to positions in the sequence is beneficial to the molecules functioning, a mutation weakening that interaction might reduce the molecules effectiveness. This change can still propagate, unless it is outright lethal. A subsequent mutation, that stabilizes the interaction again is then much more likely to survive than one, that destabilizes it further. This concept gives rise to the idea, that correlated mutations between pairs of sequence positions indicate spatial adjacencies. Sampling the evolutionary record means collecting sequences for the same molecule from several different species. To set these sequences, that have undergone point mutations, insertions, and deletions over the course of evolution, in relation, they are integrated into a common reference frame, a multiple sequence alignment (MSA). An alignment algorithm like Needleman-Wunsch [43] (also known as Wagner-Fischer [44]) or Smith-Waterman [45] inserts gap characters in the sampled sequences, such that all sequences end up the same length and the corresponding positions of the sequences fall in the same column of the resulting matrix. There exist a number of frameworks that are commonly used both for assembling the set of sequences and for alignment [46, 47]. These algorithms are related to the Hunt-Szymanski derived algorithms used in version control systems, however they are in a sense less strict, since e.g. point mutations should remain in the same column despite not being represented by the exact same character. This fuzziness is usually achieved by including a gap penalty to the final alignment score. Figure 2.3 shows a small example MSA. Assuming the co-evolutionary patterns present in an MSA contain information about the structure of the molecule, the MI between two positions that are in contact should be elevated. Given such an MSA the MI between two positions i and j in the sequence is given by[40]:

$$\mathrm{MI}_{ij} = \sum_{A,B \in V} f_{ij}(A,B) \ln \frac{f_{ij}(A,B)}{f_i(A) f_j(B)} \tag{2.1}$$



```
CCUUCCGGCGUCCCAGGCGG---GGCGCCGCGGGACCGCCCUCG--UGU-CUG--UGGCGGUGGGAUCCCGCGGCCGUG-UUUUCCUGGUGGCCCGGCCAUG
CCGUUCGGCGUCCCAGGCGG---GGCGCUGCGGGACCGCCCUCG--UGU-CUG--UGGCGGUGGGAUCCCGUGGCCGUG-UUUUCCUGGUGGCCCGGCCGCG
CCUUCCGGCAUCCCAGGCGG---GGCGCCGCGGGACCGCCCUCG--UGU-CUG--UGGCGGUGGGAUCCCGCGGCCGUG-UUUUCCUGGUGGCCCGGCAUGC
CCGGCCGGCGUCCCGGGCGU---GGG--UGCGGGACCGCCCUCUGUGGU-CUU--GGGCGGUGGGAUCCCGCGUG-UUG-UUUUCCCGGUGGCCCGGCCGUG
CCGGCCGGCGUCCCGGGC------GU--CGCGGGACCGCCCUCG--UGC-UGG--UGGCGGUGGGAUCCCGUG--CGCG-UUUACCCGGUGGCCCGGCUCGC
CCGGUCGGCGUCCCAGGC------GUGUCGCGGGACCGCCCUCG--UGU-GUGGGUGGCGGUGGGACCCCGCGCUGGUU---UUCCUGGUGGCCCGGCCGUG
CCGGCUGGCGUCACGGGC------ACGCCGCGGGACCGCUCCCG--UGUGUUG--GGGCGGUGGGAUCCCGCGGCGUUG-UUUUCCCGGUGGCCCGGUCGCG
CCGGCUGGCGCCGCGGGC------GU--CGCGGGACCGCCCUCG--UGC-UGG--UGGCGGUGGGAUCCCGUG--CACG-UUUACCCGGUGGCCUGGCUCGC
CCGGCCGGCGUCCCGGGCGU----GGGCGCGGGA-CCGCCCUGG--UCU-GUG--G-GCGGUGGGAUUCCCGUGCGUUG-UUUUCCCGGCGGCCCGGUCGCC
CGGGUUGGCGUCGCGGGCGG---U-----GCGGGACCGCCCUCG--CG---UG--UGGCGGUGGGACCCCGC----UUG-UUUUCCCGGCGGCCCGACUCUG
GGCUCCGGUGUCC---CAGG---CAUGUCACGGGACCGCCCAGG--UGU-GGG-----CGGUGGCAUCCCGUGUCUGUU-UUCCGGGAGGGACCCGGCUUUC
CCGGCCGGCGUCCCCGGGGGAUGCGCCGUGCGGGACCGCCCUCG--UG---UG--UGGCGGUGGGACCCCGCGUGUGCUCUCCCCCGGCGGGCUCCGUCGUC
```

Figure 2.3.: MSA for RF00957[48], a microRNA family. The red column highlights a conserved position, the blue columns highlight a co-evolving pair.

with the single column frequency,

$$f_i(A) = \frac{1}{M_{\text{eff}} + \lambda} \left(\frac{\lambda}{q} + \sum_{s=0}^{E-1} \frac{1}{m^s} \delta_{A,A_i^s} \right) \tag{2.2}$$

the column pair frequency,

$$f_{ij}(A,B) = \frac{1}{M_{\text{eff}} + \lambda} \left(\frac{\lambda}{q^2} \sum_{s=1}^{E-1} \frac{1}{m^s} \delta_{A,A_i^s} \delta_{B,A_j^s} \right) \tag{2.3}$$

the inverse sequence weight,

$$m^s = |\{z \in \{0,\ldots,E-1\}\,|\text{seqid}(A^s, A^z) > 80\%\} \tag{2.4}$$

i.e. the number of sequences in the alignment, that has a sequence identity larger than 80%, the alphabet V, the alphabet size $q = |V|$, the effective number of sequences $M_{\text{eff}} = \sum_{s=0}^{E} m^s$, and the pseudo count $\lambda \geq 0$, which is usually set to the same value as M_{eff}. Note that if there is no correlation between the sites, the pair-wise frequency factorizes into the product of the single-site frequency such that the MI value is 0.

Large values of MI_{ij} indicate, that the residues represented by the tokens at positions i and j are in spatial contact in the structure. However, MI produces a lot of false positives, since the correlations found by it could also be caused by indirect couplings, i.e. if position i is in contact with position j, and j with k, then MI_{ik} might also be large, or by functional correlations.

Direct Coupling Analysis

Direct coupling analysis [40] (DCA) is an improvement on contact prediction aimed at increasing the precision of MI. It is based on a statistical inverse Potts model and quantifies the strength between two columns of an MSA, excluding effects from other positions.

The underlying idea of the model is to formulate an evolutionary energy landscape for sequences of a set length. This energy term consists of contributions from pair-wise interactions and of single-site biases. The energy of a sequence is defined as:

$$E(A_1, \ldots, A_{L-1}) = -\sum_{i<j}^{L-1} e_{ij}(A_i, A_j) - \sum_{i}^{L-1} h_i(A_i) \tag{2.5}$$

The amount of MI resulting only from the direct coupling can then be computed with:

$$\mathrm{DI}_{ij} = \sum_{AB} P_{ij}^{\mathrm{dir}}(A,B) \ln \frac{P_{ij}^{\mathrm{dir}}(A,B)}{f_i(A) f_j(B)} \tag{2.6}$$

with the term replacing the pair-wise frequency:

$$P_{ij}^{\mathrm{dir}}(A,B) = \frac{1}{Z_{ij}} \exp\left(e_{ij}(A,B) + h_i(A) + h_j(B)\right) \tag{2.7}$$

The parameters occurring in this pair model can be found using different approximations like message passing, mean field [40], or pseudo-likelihood maximization [49].

2.2. Machine Learning

Machine learning (ML) describes the field of scientific research and application, where machines, algorithms or models are generated that perform tasks or give predictions without having been given explicit rules how to do this by a human designer. Instead the ML model discovers the patterns from data it is *trained* on. This is especially but not exclusively used for tasks, which are usually intuitively easy to humans but difficult for computers, like deciding, if a digital picture contains the image of a cat or a horse, or not. Next to these well established paradigms of image and language processing, there are applications in the natural sciences, which are not feasible for an untrained person or are even beyond any human as a matter of principle. In general ML systems are non-general (they can only perform the task they were trained on) and learn sample-inefficiently compared to a human (where a human might need few samples to understand the distinguishing features of a cat or a horse, an ML model might need thousands).

ML has seen a boom in recent years primarily driven by the advent of deep learning and deep neural networks enabled by the availability of large amounts of (labeled) training data and specialized hardware like general-purpose graphics processing units (GPUs) and tensor processing units (TPUs) [20, 50]. In the following we will give an overview on the background relevant to this thesis. As ML is a vast field, it can not be and therefore is not intended to be comprehensive.

2.2.1. Learning Paradigms

Usually, one distinguishes between three types of ML models: supervised, unsupervised, and reinforcement learning. All of these learning procedures are usually iterative. The model is initialized randomly and the training refines the model parameters until a convergence criterion is reached.

Supervised models like most artificial neural networks receive some input and produce an output that is compared against a label. The training process then minimizes the prediction error to produce the final model. Unsupervised methods do not require labels. One example for an unsupervised model is k-means clustering, where the data, a set of vectors of real values, is split into k subsets. The training process here minimizes the distance of the sample vectors to their current class mean, which does not require any knowledge about which cluster it should belong to. Since labels are usually expensive, not requiring any is a major advantage, however unsupervised models are often limited to specific applications.

As a hybrid between supervised and unsupervised, there is also self-supervised training. Here a label is generated from the input data, by distorting or augmenting the original input. The model is then tasked with identifying or reversing the augmentation. By forcing the model to learn, what a “proper” input sample looks like, it is thought to extract patterns, that are useful for a range of downstream tasks. Examples for this can be found in a range of domains like inpainting [51] or jigsaw puzzles [52] for image processing or masked language modeling [53] for language processing. In some instances, particularly in text generation the self-supervised model can be used for the final task without supervised finetuning.

Reinforcement learning is focused on the training of intelligent agents navigating a scenario as successfully as possible through the application of policies, which are optimized through a reward, accumulated until its conclusion.

In practice the distinction between supervised and unsupervised is not as clear and more paradigms have been introduced. Self-supervised learning is often used to pre-train a supervised model. Rather than training the supervised model starting from a randomly initialized state, instead a part of the model is first trained on unlabeled data. To this end, the unlabeled data is augmented or distorted in some way [51, 52, 54, 19] and the un-augmented data serves as label. By teaching the model what a proper input looks like, it is already sensitized to relevant patterns in the data, before having been exposed to a single label. However, it does still need some labels during a fine-tuning or downstream training.

2.2.2. Classification and Regression

One of the classical supervised ML problems is classification. Given a sample of a dataset, the model should predict, which of a pre-defined set of discrete categories this sample belongs to. The already mentioned problems of image classification, e.g. identifying an animal or object displayed [55], or handwritten digit recognition [56] are examples. So much so, that digit recognition problem MNIST [56] is considered the “Hello World” of ML. There are sub-categories like binary, multi-class [55, 56], and multi-label classification, or semantic segmentation [15].

In regression problems (a term originally coined in a biological context [57]), the model should predict a continuous value instead. Examples are time-series forecasting of temperature or relative gas pressure for weather forecasting [58].

In practice classification tasks are usually realized as regression of the probability or confidence of the model, that the sample in question belongs to each of the possible classes. This has several advantages: it gives a rudimentary level of uncertainty quantification [59], inherently allows for a ranking of the predictions, and it makes such a model trainable through gradient descent, as this requires a differentiable loss function.

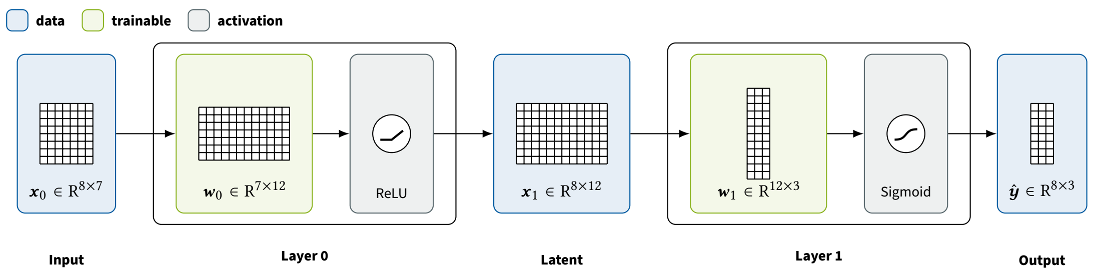


Figure 2.4.: Feed-forward network with two layers processing a batch of eight input vectors with an input dimension of seven. The latent and output dimension are twelve and three, respectively, and the activation functions are a ReLU and a sigmoid function.

2.2.3. Neural Networks

Deep artificial neural networks have become all but synonymous with ML. Large language models [60] (large language models (LLMs)) and stable diffusion [61] are not anymore only of interest to researchers, but utilized by the general public. Several frameworks have found widespread adoption. The ones used in this work are `Tensorflow` [62] and `PyTorch` [63].

The basis for neural networks are multi-layer perceptrons [64, 65, 66] (multilayer perceptrons (MLPs)) also called a feed-forward networks (cf. fig. 2.4). They are inspired by biological neural networks and process information by propagating it forwards through layers of artificial neurons from an input layer towards an output layer. The complete network is a universal function approximator [67]:

$$f(\boldsymbol{x}^{\mathbf{0}}|\boldsymbol{w}) = \hat{\boldsymbol{y}} \tag{2.8}$$

the network f with the weights $\boldsymbol{w}$ makes a prediction $\hat{\boldsymbol{y}} \in \mathbb{R}^{d_N}$, given the input $\boldsymbol{x}^{\mathbf{0}} \in \mathbb{R}^{d_0}$. In a network with N layers, the n-th single layer of the MLP performs the following transformation:

$$\boldsymbol{x}_i^{n+1} = \sigma^n\left(\sum_{j=0}^{d_n} \boldsymbol{w}_{ij}^n \boldsymbol{x}_j^n\right) \tag{2.9}$$

with the non-linear activation function $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$ and the layers weight or parameter matrix $\boldsymbol{w} \in \mathbb{R}^{d_n \times d_{n+1}}$. Note that we omit the bias vector at each layer for simplicity. Using this notation $\boldsymbol{x}^N = \hat{\boldsymbol{y}}$ is the output of the model. The output of one layer serves as input to the next and the operation the entire network performs is a concatenation of alternating linear and non-linear transformations. The non-linearities are essential, since without them the in-sequence applied linear transformations are equivalent to a single linear transformation. In the following, we will denote an element-wise application of a function to a vector like a vector: $\boldsymbol{\sigma} : \mathbb{R}^k \longrightarrow \mathbb{R}^k$ represents the application of σ to the k elements of an input vector $\boldsymbol{x} \in \mathbb{R}^k$:

$$\boldsymbol{\sigma}(\boldsymbol{x})_j = \sigma(\boldsymbol{x}_j)$$

There are several commonly used activation functions. For the internal, hidden layers, rectified linear unit(eq. (2.10)), exponential linear unit(eq. (2.11)), gaussian error linear unit [68](eq. (2.12)), and leaky rectified linear unit(eq. (2.13)) are often used (cf. fig. 2.5).

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.10}$$

$$\mathrm{ELU}(x|\alpha) = \begin{cases} x, & \text{if } x > 0 \\ \alpha \cdot (\exp(x) - 1), & \text{if } x \leq 0 \end{cases} \tag{2.11}$$

$$\mathrm{GELU}(x) = x \cdot \frac{1}{2}\left[1 + \mathrm{erf}\left(x/\sqrt{2}\right)\right] \tag{2.12}$$

$$\mathrm{LeakyReLU}(x|\alpha) = \max(0, x) + \alpha \cdot \min(0, x) \tag{2.13}$$

The activation of the final layer depends on the task of the model. The sigmoid (eq. (2.14) and softmax(eq. (2.15) functions are used for binary and multi-label classification and for multi-class classification, respectively.

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + 1} \tag{2.14}$$

$$\boldsymbol{\sigma}(\boldsymbol{x})_i = \frac{\exp(\boldsymbol{x}_i)}{\sum_j \exp(\boldsymbol{x}_j)} \tag{2.15}$$

Identity (eq. (2.16)), tanh (eq. (2.18)), and softplus (eq. (2.17)) are used for regression tasks depending on the required interval of the output value.

$$f(x) = x \tag{2.16}$$

$$\mathrm{Softplus}(x) = \frac{1}{\beta} \cdot \log(1 + \exp(\beta \cdot x)) \tag{2.17}$$

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{exp(x) + \exp(-x)} \tag{2.18}$$

Gradient Descent and Back-Propagation

Finding the best or a just good set of values for the parameters of a neural network can be viewed as an optimization problem. During the training process an optimization algorithm searches for an optimum in the parameter space, given the training dataset. While there are many optimization algorithms for different kinds of problems, gradient descent type algorithms [69] like stochastic gradient descent, RMSprop [70], or ADAM [71] are usually employed due to the high dimension of the search space, assumptions about convexity of the search space, and the efficiency of their parallel implementations. To improve the model output iteratively, the optimization algorithm first conducts a forward pass, by computing the output with the current weights as in eq. (2.8). Then it compares the output of the model to the label y using a loss or cost function. The most common loss functions are mean square error for regression, e.g.:

$$\mathrm{MSE}(\hat{y}, y) = (y - \hat{y})^2 \tag{2.19}$$

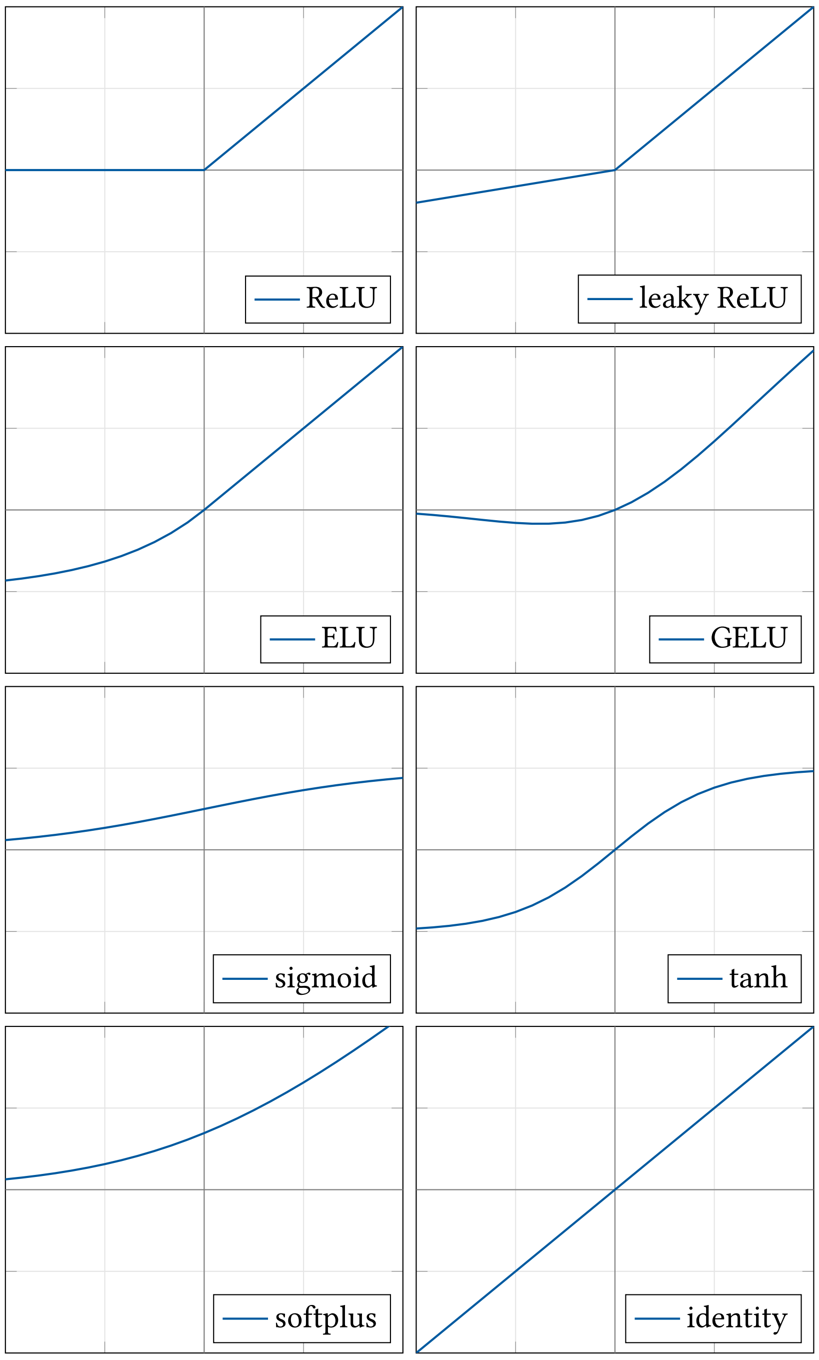


Figure 2.5.: Commonly used activation functions

for a scalar regression and cross entropy loss (CE) for classification, e.g.:

$$\mathrm{CE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{c \in \text{classes}} \boldsymbol{y}_c \log \hat{\boldsymbol{y}}_c \tag{2.20}$$

for a multi-class classification with C classes. Assuming the target $\boldsymbol{y}$ is one-hot encoded, this simplifies to the predicted log-likelihood of the target class. The loss J of the model is then the loss averaged over the training dataset with N samples:

$$J(\boldsymbol{w}) = \frac{1}{N} \sum_{i=0}^{N-1} J\left(\hat{y}(\boldsymbol{w}|\boldsymbol{x}), y\right) \tag{2.21}$$

And the optimal set of weights minimizes this loss.

$$\hat{\boldsymbol{w}} = \arg\min\left(J(\boldsymbol{w})\right) \tag{2.22}$$

To improve upon the current weights, the optimizer follows the negative gradient towards a local minimum:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - l \cdot \boldsymbol{\nabla}_{\boldsymbol{w}} J(\boldsymbol{w}) \tag{2.23}$$

with a learning rate l.

For practical reasons, the gradient is usually not averaged over the entire dataset, but over a batch of samples. For a model with only a single layer, computing these gradients is straightforward. For deeper networks, backpropagation [72] i.e. repeated application of the chain rule is used. The gradient of the loss with respect to the weights in the last layer is given by:

$$\frac{\partial J}{\partial \boldsymbol{w}_{ij}^{N-1}} = \frac{\partial J}{\partial \boldsymbol{x}_j^N} \cdot \frac{\partial \boldsymbol{x}_j^N}{\partial \boldsymbol{w}_{ij}^{N-1}} = \frac{\partial J}{\partial \boldsymbol{x}_j^N} \cdot \frac{\partial \boldsymbol{\sigma}_j^{N-1}}{\partial \boldsymbol{a}_j^{N-1}} \cdot \boldsymbol{x}_i^{N-1} \tag{2.24}$$

For the hidden layers:

$$\frac{\partial J}{\partial \boldsymbol{w}_{ij}^n} = \frac{\partial J}{\partial \boldsymbol{a}_k^n} \frac{\partial \boldsymbol{a}_k^n}{\partial \boldsymbol{w}_{ij}^{n-1}} = \frac{\partial J}{\partial \boldsymbol{a}_k^n} \cdot \boldsymbol{x}_i^{k-1} = \boldsymbol{\delta}_j^n \cdot \boldsymbol{x}_i^{k-1} \tag{2.25}$$

with the layer error

$$\boldsymbol{\delta}_j^n = \frac{\partial J}{\partial \boldsymbol{a}_j^n} = \frac{\partial J}{\partial \boldsymbol{a}_k^{n+1}} \frac{\partial \boldsymbol{a}_j^{n+1}}{\partial \boldsymbol{a}_k^n} = \boldsymbol{\delta}_k^{n+1} \cdot \boldsymbol{w}_{jk}^{n+1} \cdot \frac{\partial \boldsymbol{\sigma}^n}{\partial \boldsymbol{a}_j^n} \tag{2.26}$$

and the activation $\boldsymbol{a}^n = \boldsymbol{w}^n \cdot \boldsymbol{x}^n$. The layer errors can be backpropagated from the error of the final layer, which is given by:

$$\boldsymbol{\delta}_j^{N-1} = \frac{\partial J}{\partial \boldsymbol{x}_j^N} \cdot \frac{\partial \boldsymbol{\sigma}_j^{N-1}}{\partial \boldsymbol{a}_j^{N-1}} \tag{2.27}$$

The intermediate results (often called latent representations) needed to efficiently compute weights updates can be cached during the forward pass. For more complex model architectures, backpropagation is still applied using an automatic differentiation engine [63, 73]. Linear units as activation functions in the bulk of the network are used to combat one of the common problems of early neural network: vanishing gradients. Since activation functions like sigmoid have gradients tending toward 0 for activations with large absolute values, a model training can become stuck in certain regions of the parameter space. Next to the problem of vanishing gradients, there are also shattered gradients. For deep neural networks the gradients become smaller at each layer during backpropagation, such that the early layers converge very slowly [74]. To remedy this, skip-connections are introduced [74] (sometimes called highways).

$$\boldsymbol{x}_i^{n+1} = \boldsymbol{x}_i^n + \boldsymbol{\sigma}^n \left(\sum_{j=0}^{d_n} \boldsymbol{w}_{ij}^n \boldsymbol{x}_j^n \right) \tag{2.28}$$

This way each layer only learns the residual, and the gradient can flow through the identities from the output layer all the way back to the input layer unimpeded.

The Attention Mechanism

Next to pure feed-forward models there are several archetypal network building blocks replacing the pure linear layer for certain applications where the input has a more complex structure, than a one-dimensional feature vector. Exploiting known structure of the data often makes the network more performant, weight efficient, or versatile. The major paradigms are image processing, sequence processing, and geometric learning. For image processing tasks, the relationship between neighboring pixels is important for low-level features, and convolutional neural networks [75] (convolutional neural networks (CNNs)) consider patches of pixels in their receptive field to compute their output. CNNs drastically increase efficiency compared to a feed-forward network, by applying the same convolutional filter to the entire input image, sharing the weights. For sequences, especially sequences of variable length, recurrent neural networks (recurrent neural networks (RNNs)), e.g. long short term memory [76] (LSTM), are popular. These models iterate over an input sequence of tokens. At each iteration they take the input token and a memory of the previous iteration as input to generate an output token and the memory state for the next iteration as output. Geometric neural networks [77] (geometric neural networks (GNNs)) are more diverse. One variant is geometric or graph convolutions, which takes a vertex and the neighborhood of that vertex in the graph as input to generate an output representation of that vertex. Since unlike in the image processing case, the size of the neighborhood is variable, the weights of the transformation are shared for all vertices in the neighborhood.

Attention networks [78], popularized by the transformer architecture [79], are a relatively new type of neural network originally designed for the processing of sequences in language processing tasks. However, attention can process any structured data consisting of tokens of the same type, like sequences of tokens, images of pixels [80], volumes of voxels, or graphs of vertices [81]. Attention uses query-key information retrieval to model the interaction between individual tokens (cf. fig. 2.6). To this end, querying the latent state has to be made processable for a neural network by making it differentiable and thus learnable. The model performs a query represented by a vector $\boldsymbol{q} \in \mathbb{R}^{d_q}$, by computing

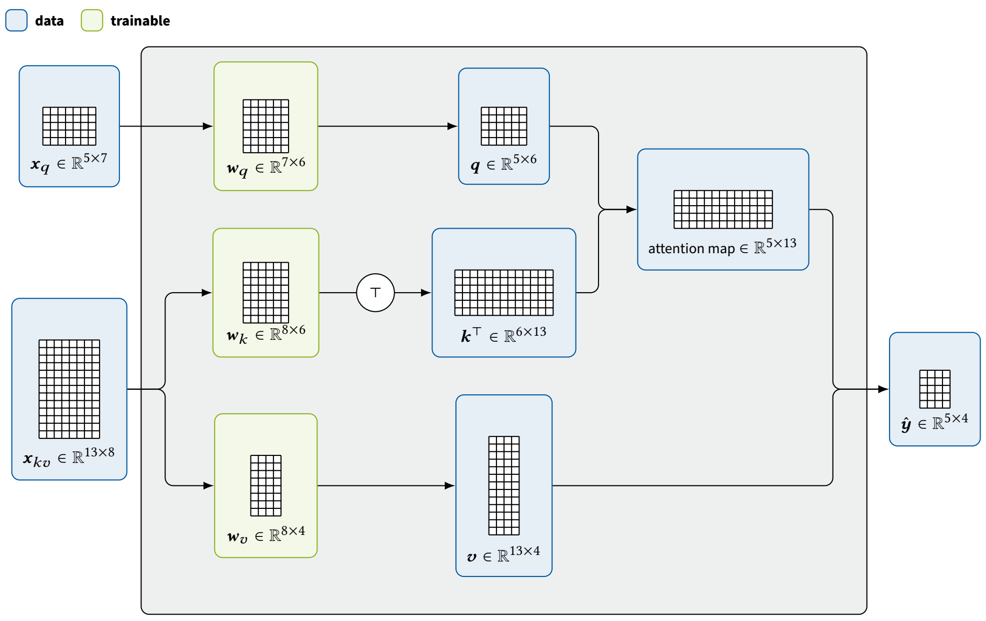

data
trainable
$\boldsymbol{x}_q \in \mathbb{R}^{5\times7}$
$\boldsymbol{w}_q \in \mathbb{R}^{7\times6}$
$\boldsymbol{q} \in \mathbb{R}^{5\times6}$
attention map $\in \mathbb{R}^{5\times13}$
$\boldsymbol{w}_k \in \mathbb{R}^{8\times6}$
$\top$
$\boldsymbol{k}^\top \in \mathbb{R}^{6\times13}$
$\boldsymbol{x}_{kv} \in \mathbb{R}^{13\times8}$
$\hat{\boldsymbol{y}} \in \mathbb{R}^{5\times4}$
$\boldsymbol{w}_v \in \mathbb{R}^{8\times4}$
$\boldsymbol{v} \in \mathbb{R}^{13\times4}$

Figure 2.6.: Attention mechanism: Query, key and value

the scalar product with a set of vectors representing keys $\left\{\boldsymbol{k}_i \in \mathbb{R}^{d_k} | i = 0, \ldots, L\right\}$ for the key-value pairs, that are queried against. The results is a set of L_{kv} relevance scores. These are softmax activated to get an attention score. The result of the query is the sum of the values weighted by their attention score. Usually, a set of queries are performed in parallel, such that the full attention layer can be expressed as:

$$\boldsymbol{y}(\boldsymbol{x}_q, \mathbf{x}_{kv} | \boldsymbol{w}_q, \boldsymbol{w}_k, \boldsymbol{w}_v) = \text{softmax}\left(\frac{\boldsymbol{w}_q \boldsymbol{x}_q \cdot (\boldsymbol{w}_k \boldsymbol{x}_{kv})^\top}{\sqrt{d_k}}\right) \boldsymbol{w}_v \cdot \mathbf{x}_{kv} = \text{softmax}\left(\frac{\boldsymbol{q} \cdot \boldsymbol{k}^\top}{\sqrt{d_k}}\right) \boldsymbol{v} \quad (2.29)$$

with the learned projection matrices of the layer $\boldsymbol{w}_{\{q,k,v\}} \in \mathbb{R}^{d \times d_{\{q,k,v\}}}$, $d_k = d_q$, the query input tensor $\boldsymbol{x}_q \in \mathbb{R}^{L_q \times d}$ and the key-value input tensor $\boldsymbol{x}_{kv} \in \mathbb{R}^{L_{kv} \times d}$. To attend to multiple input tokens in one layer, multi-head attention performs several attention operations in parallel on the same inputs, but separate projections, before mixing the values using another projection matrix on the concatenated value tensors of each head. For many applications the queries and key-value pairs are constructed from the same set of input vectors. This is called self-attention. Since this attention only operates on pairs of tokens and can not take the location of the tokens into account implicitly like convolutions or recurrent networks, the input to a transformer model are augmented with explicit positional information. Usually this is implemented either as a learned positional embedding or a fixed sinoidal embedding [79]. This is one of the limiting factors for the number of tokens a transformer can process in a single step, since a learned embedding can only embed sequences of a size it has seen during training. Another bottleneck is memory. A self-attention layer has both time and space complexity in $\mathcal{O}\left(L^2\right)$, since an attention value for every pair is computed and the intermediate result is stored for backpropagation. Since both compute time and accelerator memory is limited, there are efforts to make attention more efficient [82, 83]. However, large models require model parallelism [84, 85], i.e. distributing the model parameters and latent representations over multiple compute units.

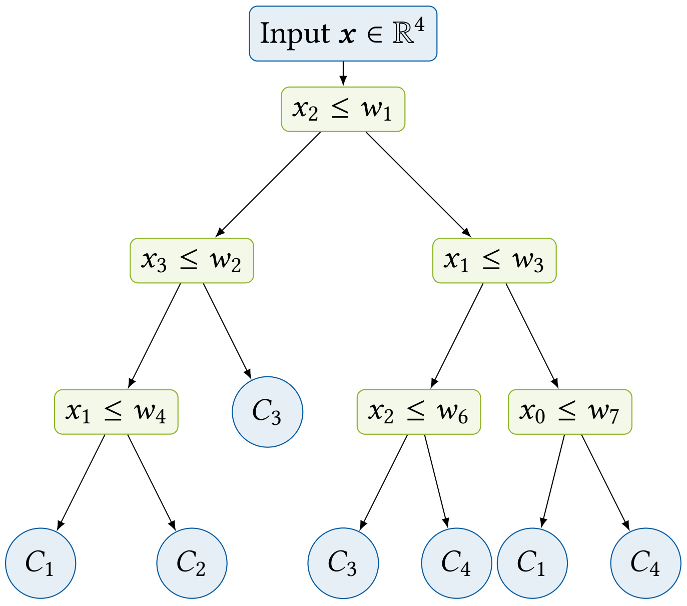


Figure 2.7.: Decision tree example

2.2.4. Random Forest Models

Random forest models are a traditional ML model used for both classification and regression. They are an example of an ensemble model, combining many weak learners, e.g. classification and regression trees (CARTs) [86], into one stronger predictor. A tree model is trained on a set of n samples x_i with d features and a label y_i, each. At each vertex, the trained model makes a binary partition of the training samples arriving at that node. The root vertex of the three represents the entire training set. The split samples follow the branches depending (cf. fig. 2.7) on the value of the split feature at each vertex until they reach a leaf node. The average label of the training samples that arrive at a leaf node is the prediction of a single tree. An optimal split at a vertex minimizes the diversity of the resulting partitions. There are different metrics used to quantify diversity. Examples are summed square error (SSE) for regression

$$\text{SSE} = \sum_{i}^{n} (y_i - \overline{y})^2 \tag{2.30}$$

and Gini impurity for classification with C classes and p_i fraction of samples with class i

$$I_G = \sum_{i}^{C} \left(p_i \sum_{k \neq i} p_k \right) = 1 - \sum_{i}^{C} p_i^2 \tag{2.31}$$

Since the training set contains only a finite number of samples, all split boundaries along all samples and all features can be computed and the one resulting in the optimal split selected.

Single tree models without constraints are prone to overfitting, when the partitions at the vertices of the tree are allowed to become too small. Regularization addresses this. For tree models, bagging (bootstrap aggregating), i.e. training only on a subset of the samples, and feature bagging [86], training only on a subset of the features, are used. To still use the discarded data, the complete forest contains multiple tree models, who are trained on overlapping subsets of features and samples.

Instead of building the trees completely independently, boosting [87] builds the weak learners successively reducing the residual of an error metric. For gradient boosted trees [88, 89] the output of the individual trees gets then weighted and the weight is optimized with respect to the error metric with gradient descent. A complete forest regression model F_m is built additively

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)\gamma = \arg\min_\gamma \sum_i L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)) \tag{2.32}$$

with $F_0 = 0$, the m-th tree h_m and the i-th labeled sample (x_i, y_i). The initial classification model instead assigns each class the same score.

2.2.5. Model Evaluation

To quantify and evaluate the performance of a model, a range of metrics is used. For regression models a mean square error (MSE) or mean absolute error (MAE) between label and prediction of the model is often used.

For classification models more intuitive metrics, that are not differentiable are available. Most of these metrics are originally defined for binary classification and then extended for multi-class problems. For the multi-class case there are often two versions of the extension: macro-averaging and micro-averaging. For macro-averaging the metric is computed separately for each class. The mean of these scores is then the total score. For micro-averaging, the underlying statistics of the predictions are used to compute the score directly instead. Micro-averaged scores run the risk of neglecting underrepresented classes.

The basis for most of the relevant metrics [90] is the confusion matrix $C \in \mathbb{N}^{N \times N}$ with the number of classes N, where C_{ij} is the number of times the model predicted class i for a sample belonging to class j. In the following, TP denotes a classifier's true positive, FP the false positive, FN the false negative , and TN true negative predictions.

$$\mathrm{TP}_i = C_{ii} \tag{2.33}$$

$$\mathrm{TN}_i = \sum_{j \neq i} C_{jj} \tag{2.34}$$

$$\mathrm{FP}_i = -C_{ii} + \sum_j C_{ij} \tag{2.35}$$

$$\mathrm{FN}_i = -C_{ii} + \sum_j C_{ji} \tag{2.36}$$

The global accuracy is the fraction of correctly classified samples:

$$\mathrm{ACC} = \frac{\sum_i C_{ii}}{\sum_i \sum_j C_{ij}} = \frac{\sum_i \mathrm{TP}_i}{|\mathrm{samples}|} \tag{2.37}$$

Accuracy on its own is often not a good metric for class-imbalanced problems since mistakes for an underrepresented class contribute little to the overall score. Instead, the class precision may be employed, which is the fraction of pixels predicted to belong to class i that actually belong to class i:

$$\mathrm{PPV}_i = \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i} \tag{2.38}$$

Here, PPV is the class-averaged (macro) precision. The class recall (sensitivity) is the fraction of pixels belonging to class i that are correctly predicted to belong to class i:

$$\mathrm{SEN}_i = \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FN}_i} \tag{2.39}$$

SEN denotes the class-averaged recall.

$$\mathrm{F}_{1i} = \frac{2 \cdot \mathrm{TP}_i}{2 \cdot \mathrm{TP}_i + \mathrm{FN}_i + \mathrm{FP}_i} = 2 \cdot \frac{\mathrm{PPV}_i \cdot \mathrm{SEN}_i}{\mathrm{PPV}_i + \mathrm{SEN}_i} \tag{2.40}$$

F_1 is the class-averaged F_1-score or macro-F_1-score.

2.3. Hyperparameter Optimization

Neural networks also contain parameters, which can not be trained through the normal gradient descent mechanism during the training process. E.g. the number of layers or the number of neurons per layer are usually a design decision decided on by the operator. This problem is a special case of hyperparameter optimization (HPO) called neural architecture search (NAS). Manual search for optimal hyperparameter combinations proves to be quite inefficient [91]. Because the search space may contain integer, ordinal, or categorical variables, gradients can not be computed and gradient-free optimizers have to be used. E.g. in the search for neural network parameters the learning rate is continuous (and usually logarithmic), the number of layers is integer, and the choice of activation function is categorical. The easiest commonly used algorithms are grid search and random search. They have the advantage of being easily implementable and trivially parallelizable. However they do not tend to search the space very efficiently and usually devote just as much resources to more interesting or promising regions as to overexplored ones. For grid search in particular the true optimum might fall in between the grid spaces searched without returning a lot of useful information on how to refine the search. More elaborate search algorithms can broadly be categorized into NAS specific approaches using reinforcement learning agents and neuro-evolution, and black-box optimization approaches, which use more or less generic optimization algorithms to sample the parameter space. NAS is a rapidly evolving field [92, 97, 98, 99, 100, 101, 102, 103, 104, 93, 94, 95, 91, 96] and the overview we give here is far from comprehensive.

2.3.1. Particle Swarm Optimization

particle swarm optimization [105] (PSO) is an established optimization algorithm inspired by the behavior of swarms of living organisms in their search for food. Each particle moves

Algorithm 1: Pseudocode for a genetic optimization algorithm

```
Input: Search-space limits, population size $P$, termination_condition, selection_policy,
       crossover_probability, mutation_probability.
1  Initialize population pop of $P$ individuals within search space.
2  while not termination_condition do                                   // OPTIMIZE
3      Evaluate individuals in pop.                                      // EVALUATE
4      Choose parents from pop following selection_policy.               // SELECT
5      foreach individual in pop do                                      // VARY
6          if random $\leq$ crossover_probability then                    // RECOMBINE
7              Recombine individuals randomly chosen from parents.
8          if random $\leq$ mutation_probability then                     // MUTATE
9              Mutate.
10         Update individual in pop.
Result: Best individual found
```

through the search space with a velocity, which is determined by its own knowledge of the space and communication with the other members of the swarm. At each iteration g the velocity of each particle i is updated and then applied to the position:

$$\boldsymbol{v}_i^{g+1} = \omega \boldsymbol{v}^g + \phi_\mathrm{p} \cdot r_1 \cdot \left(\boldsymbol{x}^g_{\hat{\mathrm{personal}},i} - \boldsymbol{x}_i^g\right) + \phi_\mathrm{s} \cdot r_2 \cdot \left(\boldsymbol{x}^g_{\hat{\mathrm{global}}} - \boldsymbol{x}_i^g\right) \tag{2.41}$$

$$\boldsymbol{x}_i^{g+1} = \boldsymbol{x}_i^g + \boldsymbol{v}_i^{g+1} \tag{2.42}$$

where ϕ_p and ϕ_s represent the cognitive and social coefficients, r_1 and r_2 are random numbers drawn from a normal distribution centered around 1, and ω is the inertia factor. $\boldsymbol{x}_{\hat{\mathrm{personal}},i}$ and $\boldsymbol{x}_{\hat{\mathrm{global}}}$ are the personal and global best points, respectively. There exist several variants of PSO [106], which modify this update rule to adapt it to different situations.

2.3.2. Evolutionary Optimization

Genetic or evolutionary optimizers are based on the idea of using the same mechanism as Darwinian evolution. An individual candidate combination of hyperparameters is modeled as a gene sequence, either as an array of bits or directly as a list of parameters. Several individuals form a population which is used to breed new candidates. To drive the population towards better solutions, the worst individuals in the population are discarded or the best ones *selected*. Their genes are then recombined through *crossover*, i.e. two or more parent individuals exchange values for the same genes to create the same number of child individuals. Finally, single genes are *mutated*. Each gene selected for mutation is assigned a random new value, either uniformly selected from the range of possible values or biased towards the proximity of the previous value. Algorithm 1 shows the algorithm as pseudocode.

Over the course of generations, the population trends towards better solutions.

Algorithm 2: Pseudocode for a CMA-ES optimization algorithm

```
  Input: Search-space limits, population size $\lambda$, $termination\_condition$, $\mu$, $c_\sigma$.
1 Initialize $\bar{x}$, $\sigma$, $C = I$, $p_\sigma = 0$, $p_c = 0$ while not $termination\_condition$ do      // OPTIMIZE
2     for $i < \lambda$ do
3         $x_i = \bar{x} + \sigma \cdot \mathcal{N}^d(0, C)$ $f_i$ = Evaluate$x_i$
4     Sort solutions $x_i$ $\bar{x}\prime \leftarrow \bar{x}$                                         // MEAN
5     $\bar{x} \leftarrow \sum_{i<\mu}(w_i x_i)$                                           // ISOTROPIC
6     $p_\sigma \leftarrow$ update_iso($p_\sigma, \sigma^{-1} C^{-1/2(\bar{x} - \bar{x}\prime)}$)              // ANISOTROPIC
7     $p_c \leftarrow$ update_aniso($p_c, \sigma^{-1}(\bar{x} - \bar{x}\prime), |p_\sigma|$)             // COVARIANCE
8     $C \leftarrow$ update_C($C, p_c, (x_1 - \bar{x}\prime)/\sigma, \ldots, (x_\lambda - \bar{x}\prime)/\sigma$)      // STEP SIZE
9     $\sigma \leftarrow$ update_step($\sigma, |p_\sigma|$
  Result: $x_1$ or $\bar{x}$
```

2.3.3. Covariance Matrix Adaptation Evolution Strategy

covariance matrix adaptation evolution strategy (CMA-ES) is an optimization algorithm related to genetic optimization. It also generates a population, which is refined over the course of generations to converge towards an optimum. The difference lies in the propagation mechanism. CMA-ES explicitly models the distribution samples for the next generation are drawn from. This distribution is approximated with a multivariate normal distribution and the mean vector and covariance matrix are updated each generation such, that the likelihood of well performing individuals in previous generations is maximized. Additionally, the algorithm (see algorithm 2) keeps track of an anisotropic and an isotropic evolution path.

$$x_i^{g+1} = \mathcal{N}^d(\bar{x}^g, \sigma_g^2 C^g) = \bar{x}^g + \sigma_g \cdot \mathcal{N}^d(0, C^g) \tag{2.43}$$

The mean of the distribution is the weighted mean of a subset of the population of the previous generation, where the weights w_i are positive and sum to one and depend on the value of the objective function at those points. The update rules are simplified here to:

$$p_\sigma \leftarrow (1 - c_\sigma) p_\sigma + \sqrt{1 - (1 - c_\sigma)^2} \sqrt{w} C^{-1/2} \frac{\bar{x} - \bar{x}\prime)}{\sigma} \tag{2.44}$$

$$p_c \leftarrow (1 - c_c) p_c + \Theta(|p_\sigma| - \alpha \cdot \sqrt{d}) \sqrt{1 - (1 - c_c)^2} \sqrt{w} \frac{\bar{x} - \bar{x}\prime)}{\sigma} \tag{2.45}$$

$$C \leftarrow (1 - c_1 - c_\mu - c_s) C + c_1 p_c p_c^\top + c_\mu \sum_{i=0}^{\mu-1} w_i \frac{x_\lambda - \bar{x}\prime}{\sigma} \left(\frac{x_\lambda - \bar{x}\prime}{\sigma} \right)^\top \tag{2.46}$$

$$\sigma \leftarrow \sigma \cdot \exp\left(c_\sigma \left(\frac{|p_\sigma|}{\mathrm{E}|\mathcal{N}_d(0, I)|} - 1 \right) \right) \tag{2.47}$$

with the remaining parameters either being constants or depending on the problem space dimension d. Again, there are several variants of CMA-ES adding additional features [107, 108].

Algorithm 3: Pseudocode for a bayesian optimization algorithm

```
  Input: Search-space limits, population size, termination_condition, initial population.
1 while not termination_condition do                                  // OPTIMIZE
2     $\hat{x} \leftarrow \text{argmin}_{x \in \text{population}}(a(x))$ evaluate $y = \hat{x}$
3     update posterior
4     update acquisition function
  Result: Best point found
```

2.3.4. Bayesian Optimization

Bayesian optimization [109] uses Bayes' theorem to construct a surrogate of the objective function to be optimized. Using the information of previously evaluated points in the search space as evidence, the posterior probability of a surrogate model is proportional to the likelihood of the evidence given the model times the prior probability of the model

$$P(\text{model}|\text{evidence}) \propto P(\text{evidence}|\text{model}) \cdot P(\text{model}) \tag{2.48}$$

The surrogate of the objective function is combined with an acquisition function a, which drives where to sample the search space next. To suggest the next point to be evaluated it takes into account the expected value of the posterior and its variance. Areas close to already sampled points have low variance and are unlikely to be sampled. Areas that are expected to have far from optimal values of the objective function are not interesting and are also less likely to be sampled. Algorithm 3 shows the generic bayes optimization algorithm. For NAS, tree structured parzen estimator [110, 111] (TPE) are often used to compute the posterior, rather than gaussian processes. Here, the population of already evaluated points is split into the best and worst set of points. The posterior distribution is then estimated by fitting Gaussian kernels to these samples resulting in a "good" distribution g and a "bad" distribution b. The acquisition function is then given by b/g.

2.3.5. Reinforcement Learning

Next to generic gradient free optimization techniques, there are also efforts to utilize machine learning to solve the hyperparameter optimization problem. Here an agent trained with reinforcement learning [112, 113] suggests new parameter combinations. The suggested model is then constructed and trained and the agent receives a reward based on the performance of the model. The agents in the cited examples are long short term memorys (LSTMs), which produce a sequence of parameters until they terminate or hit a budget threshold.

2.3.6. Neuro-evolution

Neuro-evolution (not to be confused with evolutionary optimization) algorithms [114, 115, 116] combine hyperparameter search and network training into a single process. Instead of generating an architecture, which is then constructed, trained, and evaluated, they train a model continuously while adding, removing, or swapping building blocks or components.

2.4. High Performance Computing

To process the large amounts of data repeatedly to fit the amount of parameters of a complex neural network, a single GPU on a single compute node is no longer sufficient. To provide the considerable compute resources required by the training of large machine learning models, high performance computing (HPC) environments are required. An HPC cluster consists of hundreds to thousands of compute nodes which are connected through a low-latency network. Each with its own central processing units (CPUs), memory, disk space and, optionally, hardware extensions, e.g. (general purpose) graphics processing units ((GP)GPUs). Through the network infrastructure the individual processes distributed over the nodes can collaborate on a single problem. Since in such a distributed memory architecture the different processes can not coordinate and exchange information directly through local memory, software has to be designed for this hardware or adapted to it.

The message passing interface [117] (MPI) is a communication standard with multiple implementations and bindings in several languages. Within an MPI program a specified number of processes are launched on the assigned compute nodes. All processes run the same program and are grouped into a *communicator*. Each process participating in a communicator has a unique rank, which can be used to address it for message passing or for control flow within the program being executed. MPI provides a range of different operations, depending on the situation at hand. To name a few: synchronous and asynchronous, point-to-point and collective, buffered and unbuffered.

To store the results of a program, all compute nodes also have access to a distributed file system. Writing to and especially reading from the distributed file system is again slower than reading from locally mounted storage and should generally be avoided inside performance relevant sections of a program.

In the context of neural networks, one commonly distinguishes between model parallelism and data parallelism. Model parallelism is usually relevant, when a single neural network would require more memory than a single GPU can provide. There are several different possible ways of realizing model parallelism, e.g. vertical or horizontal distribution of the network. For the following chapters only data parallelism is relevant. It is based on the fact, that the model output for each sample in a batch can be computed independently. Each GPU hosts a copy of the same model. During a single training iteration, a global batch of data is distributed over the available GPUs to compute forward and backward step. The gradient updates are exchanged and after one update, all models are synchronized again. This speeds up training and allows for larger batch sizes.

Next to MPI there are also dedicated GPU communication interfaces like NCCL [118] or RCCL [119], which enable efficient inter-GPU communication within a node and across node boundaries.

In the following, each chapter contains a section describing the hardware setup used to produce the results described in that chapter. The parallelization strategies used are also described in each chapter, if applicable.

3. Data Efficient RNA Contact Prediction

This chapter describes our work on data efficient RNA contact prediction. Only little labeled data for RNA is available in the form of experimentally resolved molecular structures. Our approach to make the most out of this limited data is two-pronged. First, we use self-supervised learning to utilize unlabeled data as much as possible. Second, we use boosted decision trees to improve the prediction over a pure neural network. The chapter adapts and builds on our publication in Communications Biology [1].

3.1. Introduction

As illustrated in chapter 2, the problem of predicting biomolecular structures from their genetic sequences has rapidly advanced by leveraging high throughput data acquisition (in particular high-throughput sequencing) via machine learning. A sequence in this context is a string of tokens, where each token represents one residue of the linear molecular structure. One of the main challenges in transferring the progress achieved for protein structure prediction to RNA is the stark difference in data availability. Successful protein structure prediction models utilize hundreds of thousands of experimentally determined structures as training labels [120]. Meanwhile, the number of available RNA structures is quite sparse and amounts to approximately one hundred, three orders of magnitude less [121]. To address these challenges, we forgo the end-to-end prediction of molecular structures in the form of atomic-resolution point clouds, which is possible for proteins, and return to the prediction of contact maps as a proxy. The predicted contact maps can then serve as bias for a simulation or other modeling to perform the full contact prediction. RNA sequences are more abundant and as discussed in chapter 2 also contain structural information.

Building on the MSA Transformer [122] already used for protein data, we employ a backbone network that is trained in a self-supervised fashion. While traditional supervised learning requires labeled datasets to train a model, self-supervised training can harness the larger amounts of unlabeled data in addition to the few labeled samples. During self-supervised pre-training, training samples are *augmented*, i.e. perturbed in some fashion and the model is tasked with recovering the original sample or predicting information about the applied augmentation. A range of different pre-or upstream tasks can be conceived of. This way the model learns patterns and context that represent the data. The most common implementation of this is masking out elements of the input, which the model then recovers. We describe the self-supervised tasks in more detail in section 3.3. On top of this inpainting or masked language modeling used to train the MSA Transformer, we explore different upstream tasks to expose the model to patterns at different scales. The secondary motivation for different upstream tasks is the inefficiency of random inpainting masks. The chance that a randomly generated augmentation is informative for the model is low. However, implementing a mechanism akin to semi-hard example mining [123, 124] without already knowing a lot about the considered molecule so far has proved elusive.

To use the labeled data as efficiently as possible, we use decision forest models for the downstream contact prediction. These models, in particular XGBoost [89], have been shown to outperform neural networks in certain contexts [125].

3.2. Related Work

In this section, we will provide an overview of the most important approaches to predicting biomolecular contact maps from their sequences. As protein and RNA models are very similar on a technical level, we will not differentiate between them in the following.

3.2.1. Direct Coupling Analysis

Direct coupling analysis [40] (DCA) is an unsupervised technique to find evolutionary couplings between co-evolving positions in a molecular sequence. The underlying theory is based on the idea that two residues forming a structurally or functionally relevant interaction in the molecular structure will leave behind a pattern in the evolutionary record, as mutations weakening the interaction are selected against (cf. chapter 2). Some of the possible mutations immediately make the organism non-functional and do not occur in the evolutionary record. In less extreme cases, a mutation might only weaken the strength of an interaction. On its own this mutation would then be expected to be less prevalent but existent. However subsequent mutations might compensate and stabilize the function of the molecule again. These correlated mutations leave behind a clear pattern when comparing many related sequences of the same molecule across species. Often the interaction that determines the fitness of the molecule is direct physical one between two residues in the RNA or protein chain. This allows to infer a spatial proximity from a high mutational correlation score between two tokens in the genetic sequence over the course of evolution.

This method still requires many sequences of the target molecule over a wide range of related organisms. However, sequencing is a lot cheaper than experimental structure determination.

The input for DCA is a multiple sequence alignment (MSA) of the target sequence and a set of, ideally, related but diverse sequences. Different algorithms can be used to produce an alignment from such a set [46, 47] queried from a sequence database [36, 38]. Given an MSA, DCA models the correlations between position in the target sequence, while accounting for indirect interactions to improve the true positive rate on structural contacts. The resulting model fitted to an alignment has parameters for the bias for each position and for the coupling strength between positions, with the bias representing the distribution of tokens at that position. Historically, DCA is rooted in so called Pott's models that used to explain magnetism in statistical physics.

Ordering the coupling parameters according to their magnitude and selecting the strongest ones allows for a first contact prediction, albeit without a quantitative measure for the model's confidence. An in-depth explanation is given in chapter 2.

3.2.2. Neural-Network Based Approaches

Early neural networks for contact prediction were CNN which used the output map of DCA as input and refined it, thus treating the task almost entirely like an image processing problem. These CNN-based models [126, 127] predict entire contact maps, i.e. a contact likelihood for each possible pair of residues. To include features of the underlying sequences directly, an outer concatenation or sum of embedded tokens and the so-called sequence profile containing information on alignment statistics for each position of the sequence were added to the input [126]. On the output side, these CNNs would soon predict discretized distance maps also known as distograms [41] instead of binary contacts.

With the advent of transformers [79] and large language models [128], the focus shifted away from image towards language processing models. The MSA Transformer [122] is such a protein language model. It extracts co-evolutionary patterns through self-supervised training and can be used to predict contacts from its latent attention maps. A successive single sequence model [129] embeds the evolutionary context inside its own model parameters instead of the input MSA and latent representations.

A third paradigm is geometric or graph models which extract information from generated or sampled structures, e.g., an improved structure score [130]. Since these models do not generate structure candidates themselves, we consider them beyond the scope of this thesis.

3.2.3. End-to-End Models

Ideally, a model would predict the atomic coordinates as a point cloud directly. This would save a lot of post-processing and often computationally involved modeling and give the model potentially useful inductive biases [120]. While a first end-to-end model, the recurrent geometric network (RGN) [131], was still based on long short-term memory [76], the more recent AlphaFold2 [120] and RosettaFold [9] are attention-based. The latter use a similar architecture of different sub-modules, including a token-level attention network and a geometric structure module to model appropriate inductive biases, e.g., the SE(3)-Transformer [132]. Interestingly, the token-level sub-module is also trained with a self-supervised masked language task as auxiliary loss. The referenced models are trained on protein data, as they require hundreds of thousands of samples.

For RNA a the RNAformer [133], a transformer model for secondary structure prediction, a related problem with more available data, was recently published.

3.3. Method

In this section, we describe our own contribution to solving ML-based RNA contact prediction.

3.3.1. Data

The entire data used for training originates from several sources. Self-supervised pre-training uses unlabeled MSAs, while finetuning requires structure labels. For the latter the first sequence in the alignment is the one the label structure belongs to. For the self-supervised backbone training, we use 4070 MSAs from the RFam 14.6 [134, 135] database and 43 MSAs from ZWD [136]. Figure 3.1 shows the distributions of MSA depth and width, i.e., the number of sequences and sequence length over the entire dataset. The labels for the upstream training are generated stochastically on the fly during the training and described in the respective section for each task. For downstream training, we use a set of

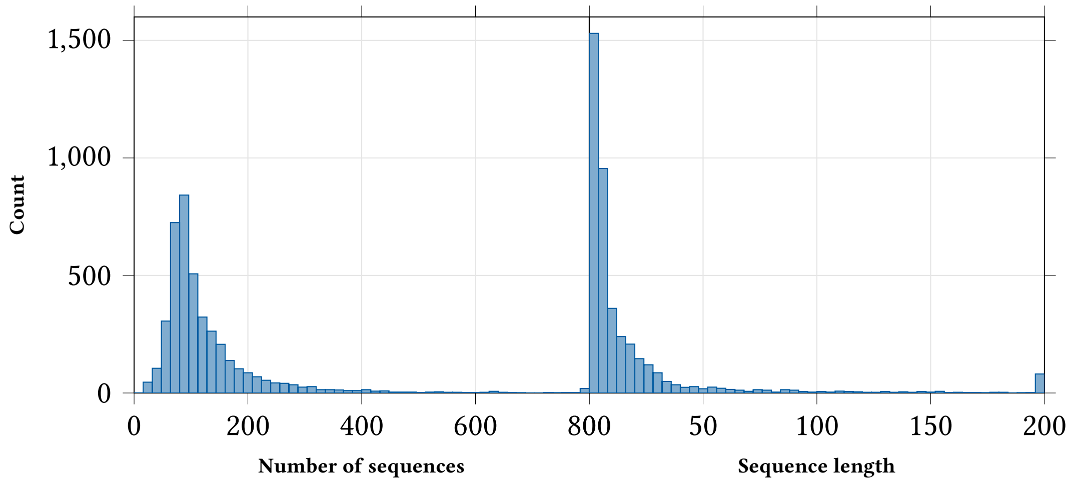


Figure 3.1.: Distribution of number of sequences (left) and sequence length (right) per MSA in the upstream dataset.

RNA MSAs and structures [121] previously used to train the simple convolutional model CoCoNet [127]. A contact map label $\boldsymbol{T}$ is constructed from a structure, by creating a binary tensor of dimension $L \times L$, where L is the sequence length. If the shortest inter-residue distance between atoms of two residues i and j is shorter than 10 Å then the $\boldsymbol{T}_{i,j}$ is 1, otherwise 0. These MSAs are selected from the same set of molecule families comprising RFam but filtered with the following constraints:

- Only RNA is present. Complexes with RNA or other molecules are excluded.
- The resolution is better than 3.6 Å.
- The minimum sequence length is 40 residues.
- For multiple structures with sequence identity ≥ 50%, only the higher resolution one is included.

In these MSAs, the columns containing gap characters in the target sequences are removed since this has been shown to improve the performance of DCA.

As labels, the full atomic structure is available. To generate a target contact map, we measure the shortest distance between all inter-residue atom pairs. All pairs closer than 10 Å are considered ‘in contact’.

3.3.2. Metrics

Some metrics for evaluating contact map prediction models go beyond the default classification metrics described in chapter 2. The Top-$L_{0.5}$-precision is the precision computed over the L most confident predictions where L is the sequence length. The Top-L-precision is the precision over the L most confident predictions, assuming all these predictions are positive. This implies that the decision boundary might be adjusted to be lower than 0.5, which is why we chose to denominate the "normal" precision with the fixed decision boundary to avoid ambiguity and use nomenclature consistent with most references. Using this metric has two benefits. First, it is applicable to DCA, which only provides an order of contacts but no transferable scores (i.e. the score is system-specific). Second, optimizing for it forces the model to make a number of positive predictions, avoiding the pathologic case where the model only predicts very few contacts and thus renders itself useless.

For a global metric, which takes into account all predictions and not just the most confident, we use the Matthews correlation coefficient (MCC):

$$\mathrm{MCC} = \frac{\mathrm{TP} \cdot \mathrm{TN} - \mathrm{FP} \cdot \mathrm{FN}}{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})} \tag{3.1}$$

This correlation score lies between −1.0 and 1.0 and is particularly useful for imbalanced binary classification problems. A score of 0 means, there is no correlation between target and prediction. A score of 1 means, that prediction and target are identical.

3.3.3. Self-Supervised Upstream Training

The model used for pre-training consists of several distinct stages (cf fig. 3.2). First, the original input samples are augmented and the corresponding labels are generated. The augmented inputs are then passed through the model backbone, which comprises several blocks of tied axial self attention [122] and outputs a latent representation. A sub-model for each task, called a task head, produces the prediction for each task from this latent representation. Finally, the loss between the previously generated labels and the model predictions is used to back-propagate through the entire model and update its weights. The task heads are kept deliberately shallow so as to contain most of the model capacity in the backbone and best possibly exploit a regularization between the different tasks when using multiple tasks at once.

The tasks themselves can, and in some cases are expected to, be combined with each other. Accordingly, the total upstream training loss is calculated as the sum of the respective individual task losses:

$$L = \sum_{\text{task}} L_{\text{task}} \tag{3.2}$$

It should be noted for the tasks considered here, we did not find a need to rescale the losses to be on the same scale (cf. section 3.4.1).

Model Backbone

The bulk of the model consists of a sequence of tied axial attention blocks similar to the MSA Transformer [122]. To conserve memory, this architecture does not compute

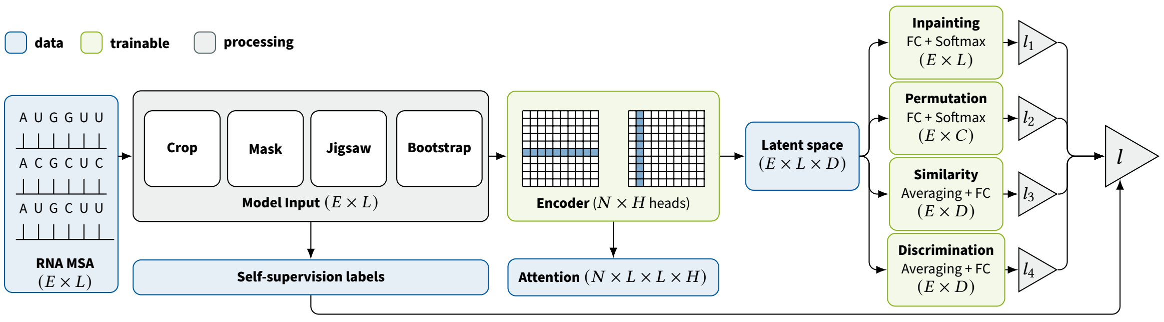

data
trainable
processing
A U G G U U
A C G C U C
A U G C U U
RNA MSA $(E \times L)$
Crop
Mask
Jigsaw
Bootstrap
Model Input $(E \times L)$
Self-supervision labels
Encoder ($N \times H$ heads)
Attention $(N \times L \times L \times H)$
Latent space $(E \times L \times D)$
Inpainting FC + Softmax $(E \times L)$
Permutation FC + Softmax $(E \times C)$
Similarity Averaging + FC $(E \times D)$
Discrimination Averaging + FC $(E \times D)$
l_1
l_2
l_3
l_4
l

Figure 3.2.: Illustration of the upstream training including augmentations and label generation for pre-training tasks, backbone model, and task heads.

attention between all tokens in the MSA. Instead each transformer block first applies attention across a sequence (row) and then across the column. Additionally the row attention maps before the softmax for row attention are summed over the column, such that they effectively express attention between two columns. Before reaching the attention blocks, an input MSA is prepended a column of start sequence tokens, one-hot encoded, and embedded such that each token is represented by a vector of dimension d_h, the hidden size of the network. Since transformers natively process sets of input tokens rather than sequences, we also add a positional embedding as a set of L_{max}, the maximum sequence length, learned vectors added to the embedded token vector before the first attention block. Since the positional embedding only carries information about which position the token has in the sequence, but not to which of the sequences in the alignment it belongs to, the position of the sequence within the alignment is not explicitly known to the model. However, since the first attention operation processes the sequences entirely, it has implicit access to sequence-global context, which can be meaningful to subsequent operations.

The final output of this backbone is a latent representation of the MSA, i.e., a tensor of shape $[B, E, L, d_h]$, with batch size B and number of sequences E. Each task head receives this tensor as input and reduces it according to its requirements if necessary. Table 3.1 shows the full upstream model and training parameters.

Inpainting

The inpainting task is inspired by the inpainting used in self-supervised training of image processing models [51]. First, the augmentation replaces a selected fraction of all tokens in the input MSA with uniformly selected other legal tokens.The inpainting head at the end of the neural network takes the latent embedding of each masked token and predicts which original token was replaced. Several variants of the inpainting augmentation can be implemented. The most successful masking selection in terms of downstream performance is the simplest one, where each token has the same chance of being masked. Unintuitively, masking schemes taking into account the structure of the data, i.e., masking out single entire columns or consecutive blocks of columns, perform worse even though they should make the task harder and more informative.

As already mentioned, masked tokens are not replaced with a special masking token, like in other applications [122, 53], but with random legal tokens. We did not explore different distributions for sampling these replacement tokens. Conceivable versions are sampling the replacement tokens according to the letter frequencies in the entire MSA or the column where the token is being replaced. Besides these choices, the most important hyperparameter is the masking frequency, i.e., the fraction of tokens to mask out.

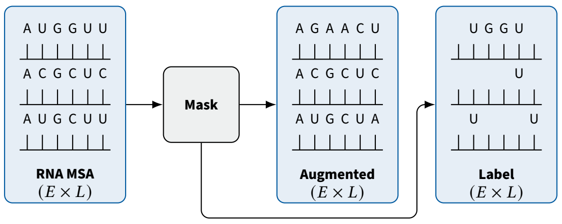


Figure 3.3.: Illustration of the inpainting task.

Jigsaw

The jigsaw tasks is intended to capture information on a larger scale than inpainting. In its original image processing version [52], random patches of the input image are given to the model in a particular order and a classifier task head predicts the permutation applied to this order. Adapting this approach to MSAs, we partition the sequences in the MSA along the same columns and permute these partitions independently of each other per sequence.The task head receives the `start sequence` special token and predicts the permutation applied to each sequence.

To make the jigsaw training more informative, the permutations are embedded in a Euclidean space with distances approximating their Hamming distance instead of a purely categorical one-hot representation usually employed for classifiers.

Hyperparameters for this task are the number of partitions, whether to apply the same permutation for each sequence in the MSA, and how much, if any, of the border regions to exclude. The rationale behind this is that the borders of the MSA are often the most volatile and thus contain a lot of gap characters, which makes them trivial to identify. We set the number of partitions to 4 and the number of permutations to 24.

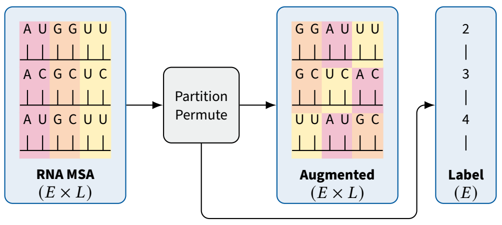


Figure 3.4.: Illustration of the jigsaw task.

Contrastive Embedding

For contrastive tasks, the model has to embed different augmentations of the same input samples closer than those of different samples in an output space [54]. For the MSA-adapted version, each sequence is embedded, while the augmentations applied are given by the other tasks that are trained concurrently. The loss is similar to the NT-Xent loss used in SimCLR [54] computed over all positive pairs of sequences i and j. Positive pairs originate from the same MSA.

$$J_{i,j} = -\log \frac{\exp\left(\mathrm{sim}(\hat{\boldsymbol{y}}_i, \hat{\boldsymbol{y}}_j)/T\right)}{\sum_k (1-\delta_{i,k}) \exp\left(\mathrm{sim}(\hat{\boldsymbol{y}}_i, \hat{\boldsymbol{y}}_k)/T\right)} \tag{3.3}$$

The temperature T is set to 100 for the models whose results are shown here. k indexes over all samples in the global batch, i.e. it includes the sequences from the local batches on other GPUs.

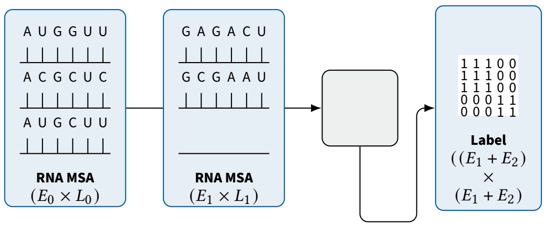


Figure 3.5.: Illustration of the contrastive task.

Bootstrap

The bootstrap task bears most similarity to generative adversarial networks (GANs) [137]. The difference is that the generator is not a neural network. Instead the augmentation randomly generates a new artificial sequence according to the per column frequencies of each letter. The generated sequence then replaces one of the original sequences in the MSA. The model being trained plays the role of the discriminator network, classifying for each sequence whether it is sampled from evolution or generated by the augmentation. The only parameter for this task is the bootstrapping ratio, which determines how many sequences in the MSA contribute to the bootstrapped sequence. The loss is a per token CE between the generated and the replaced sequence.

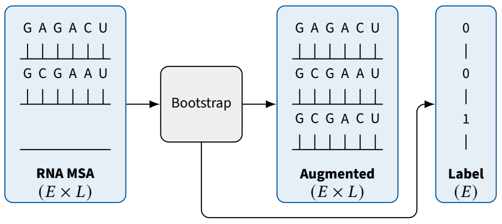


Figure 3.6.: Illustration of the bootstrapping task.

3.3.4. Finetuning and Downstream Training

Figure 3.7 illustrates how training for the contact prediction task is performed. We use

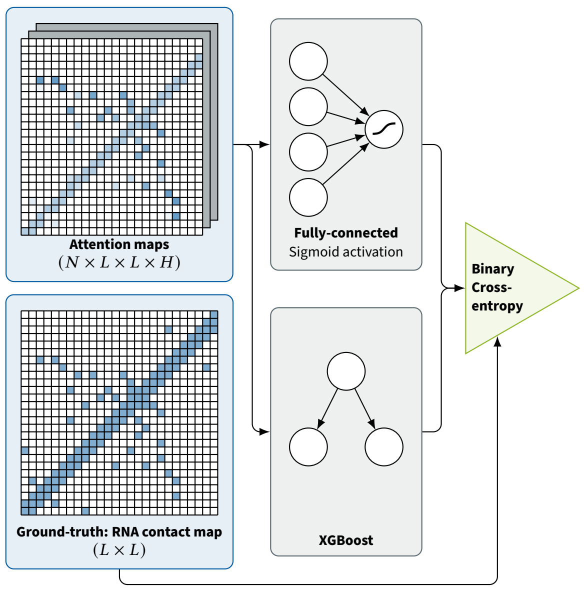


Figure 3.7.: Downstream training.

four different versions of downstream training for the actual contact prediction:

- **Frozen regression:** Training a single-layer regression model receiving the attention maps of the pre-trained and then frozen backbone as input.
- **Fine-tuned regression:** Training the same simple regression architecture but leaving the backbone parameters trainable, thus finetuning the backbone parameters to the contact prediction task.
- **Frozen XGBoost:** Training an XGBoost random forest model using the frozen backbone attention maps.
- **Fine-tuned XGBoost:** Training an XGBoost model using the attention maps of the previously regression-fine-tuned backbone.

All of these models take the stack of attention maps produced by the backbone as an input, rather than the latent output representation. Their receptive field is limited to one pixel of this image with the number of color channels determined by the number of blocks and the number of attention heads per block. We only use the row attention maps at this point, since column attention would require an additional transformation. For both regression and XGBoost, we hold back 20% of the training data as random validation set to use for early stopping since over-fitting is a likely problem, especially for the finetuning with unfrozen backbone. We explore different early-stopping metrics:

- cross entropy loss

- top-$L_{0.5}$-precision or positive predictive value
- top-L-precision or positive predictive value
- F_1-score
- MCC

End-to-end training random forest downstream models is easily possible since the backbone model is trained using gradients and the individual trees are constructed without gradients. While the gradient-boosted forests fit the weights for the weak learners' individual contributions using gradients, the trees themselves are still trained the same way. This is why we use the weights from the backbone model fine-tuned through regression to generate the inputs for the fine-tuned XGBoost model.

Parameter	**Pre-training**	**Finetuning**
Architecture		
# Attention blocks	10	10
# Heads per block	12	12
# Features per head	64	64
Batch size (local)	1	1
Optimizer	Adam	Adam
Learning rate	3×10^{-5}	1×10^{-4}
Warm-up	linear, 400 epochs	–
Decay	inverse square root, after 400 epochs	–
Drop-out ratio	0.3	0
Pre-processing		
Cropping mode	random	random
Cropping size	400	400
Subsampling mode	random	diversity-maximizing
Subsampling size	50	50
Inpainting		
Mode	token-wise	–
Masking ratio	0.15	–
Replacing tokens	regular seq. tokens	–
Jigsaw		
# Chunks	4	–
# Permutations	24	–
Bootstrapping		
Mode	token-wise	–
Replacement ratio	0.5	–
Contrastive		
Temperature	100	–
Validation data size	100 MSAs	20%
Float precision	16-bit mixed	16-bit mixed
Training time	2 days	< 30 minutes (incl. early stopping)

Table 3.1.: Hyperparameters of the neural networks for pre-training and finetuning.

Regression Model

The contact prediction model is a simple single-layer fully connected model processing one pixel of the input stacked attention maps generated by the backbone at a time. This is implemented as a one-by-one convolution with $n_{\text{heads}} * n_{\text{blocks}}$ input channels, which corresponds to the total number of attention heads. Since the problem is a binary classification, there is a single output channel followed by a sigmoid activation function. The loss function used is focal loss (FL).

3.3.4.1. XGBoost

Before passing the stacked latent attention maps to XGBoost, the backbone symmetrizes them like in the regression case and only passes the upper diagonal. XGBoost expects a list of samples with labels as training input. Since the entire XGBoost training set is a list of attention maps consisting of individual pixels, we flatten the upper triangle of each tensor and concatenate them to a list of potential contacts. The per-pixel outputs of the XGBoost model are then reconstructed into a contact map prediction.

Parameter	Value
# Trees (max)	300
Tree depth (max)	16
Learning rate	1.0
Booster	DART [138]
Drop-out ratio	0.1
Subsampling	
Mode	gradient-based
Rate	0.9
Colsample	
By-tree	0.7
By-level	0.7
Minimum split loss	0.7
Objective	binary:logitraw
Tree method	gpu_hist
Training time	< 10 minutes (incl. early stopping)

Table 3.2.: XGBoost model hyperparameters.

3.3.4.2. Finetuning

Finetuning a pre-trained model usually simply means training the neural network with the task-specific data of the downstream task. However, when the downstream model is a random forest, it is not easily possible to back-propagate the loss through the entire stack of models. For gradient-boosted trees, it might be possible to build an end-to-end trained compound model, but in this work we went a different route.

To tune the backbone to the downstream task, we use the regression model. Since it is a neural network the contact prediction loss can be propagated from the regression head through the entire backbone. For the regression model, the difference between the the untuned and the tuned training amounts to unfreezing the backbone weights during contact prediction training. The tuned XGBoost model then uses tuned backbone from the unfrozen regression training.

3.3.5. Hyperparameter Search

We performed a random search for the backbone model's hyperparameters for 300 iterations within the limits given in table 3.3.

For the downstream regression model, we briefly manually explored adding hidden layers and biases but then stuck to the same configuration as the MSA Transformer.

The hyperparameter search for the downstream XGBoost model was conducted using Propulate [3], the evolutionary search algorithm discussed in chapter 5. Table 3.4 shows the limits of the respective parameter search space. The evolutionary mechanism applied is a combination of selection, single-point crossover, and mutation operations. Using four nodes with four GPUs each, 16 workers are distributed over four islands with a migration probability of 0.1, a mating probability of 0.7, a mutation probability of 0.4, and a random initialization probability of 0.1.

Parameter	**Limits**
# Blocks	{6, 8, 10}
# Heads	{8, 12, 16}
d_{head}	{16, 32, 64, 128}
Learning rate	$[10^{-6}, 10^{3}]$
Dropout	$[0, 0.5]$
Inpainting masking mode	{token, column}
Jigsaw partitions	{3, 4, 5}
Contrastive Temperature	$[10, 100]$

Table 3.3.: Backbone hyperparameter search limits.

Parameter	**Limits**
# Trees (max)	$[1, 500]$
Tree depth (max)	$[4, 16]$
Learning rate	$[0.01, 1.0]$
Drop-out ratio	$[0, 0.5]$
Subsampling	
Mode	{uniform, gradient-based}
Rate	$[0.4, 1]$
Colsample	
By-tree	$[0.4, 1]$
By-level	$[0.4, 1]$
Minimum split loss	$[0, 1]$

Table 3.4.: XGBoost hyperparameter search limits.

3.3.6. Parallelization Strategy of Pre-Training

Using tied axial attention is primarily a memory saving measure, even though it also provides some inductive bias. Without further optimizations, HoreKa's 40 GB GPUs can only hold the latent representations of a single sample. To speed up pre-training and increase the effective batch size, we employ data parallelism by distributing the samples of one training iteration to the four GPUs of one node. This parallelization scheme requires synchronization at certain points during training. At each training iteration, each GPU performs the forward and backward pass for its sample. The local gradients are shared and averaged so that each GPU can update its copy of the model synchronously with the same batch-global gradient. Since the input MSA of each sample might contain a different number of sequences of different length, i.e., a different number of total tokens, a GPU with a small sample might have to wait for a GPU with a larger sample to complete before it can move on to the next iteration.

Downstream training is significantly faster and a single GPU is sufficient for regression, backbone finetuning, and XGBoost training.

3.3.7. Computing Environment

We performed all reported experiments on the high-performance computing system "Hochleistungsrechner Karlsruhe" (HoreKa) operated at Scientific Computing Center (SCC), Karlsruhe Institute of Technology (KIT). HoreKa provides different kinds of compute nodes. The accelerator nodes we used are equipped with two 38-core Intel Xeon Platinum 8368 processors at 2.4 GHz base frequency and 3.4 GHz maximum turbo frequency, 512 GB main memory, two network adapters, and 4 NVIDIA A100-40 GPUs with 40 GB memory each. The nodes are connected with a low-latency, non-blocking NVIDIA Mellanox InfiniBand 4X HDR interconnect with 200 Gbit/s per port.

The operating system installed on all nodes is Red Hat Enterprise Linux 8.2. The relevant software package versions are `Python v3.8` with `biopython v1.79`, `numpy v1.20.3`, `torch v1.9.1.+cu111`, `torchmetrics v0.6.0`, `pytorch-lightning v1.5.1`, and `lightning-bolts v0.4.0`.

3.4. Results

3.4.1. Upstream Performance

Since inpainting is the task operating on the highest resolution, i.e., of individual tokens, and already established as the baseline, we report upstream training performance with respect to this task. The jigsaw, contrastive, and bootstrap tasks are always used in conjunction with inpainting. Table 3.5 shows upstream performance on the validation set. For inpainting and jigsaw, we report classifier accuracy. We observe that inpainting performance always suffers when including a secondary task and there is no synergy between tasks.

Figure 3.8 shows the training loss over the course of pre-training. The losses are on the same order of magnitude, with the contrastive contribution diverging from inpainting the most. Different loss weights did not yield any appreciable performance improvement.

Task	Inpainting Accuracy	Sec. Task Accuracy	Epochs	Energy / Wh
Inpainting	90.4 %	–	2222	52989.8
Jigsaw	83.1 %	98.3 %	2264	53837.6
Contrastive	89.9 %	–	1707	50540.8
Bootstrapping	83.1 %	95.1 %	2182	52874.2

Table 3.5.: Pre-training task performance and energy consumption

3.4.2. Downstream Performance

For evaluating downstream contact prediction performance, we use two metrics, i.e., the top-$k \cdot L$-precision or positive predictive value (PPV), where L is the sequence length of the molecule and k is a fraction usually set to 1, and the Matthews correlation coefficient. Note that for the precision metric, the decision threshold is adapted for each molecule such that the number of positive predictions is $\lfloor k \cdot L \rfloor$. Figure 3.9 shows averaged top-L-precision and MCC for all model types. Figure 3.10 shows top-$k \cdot L$-precision over k for the tuned XGBoost model. Since the DCA baseline does not produce likelihoods that are consistent between molecules, MCC can not be computed here.

Our first observation is that jigsaw drastically decreases downstream performance, even though it did not have a comparable impact on upstream inpainting accuracy. The other pre-training tasks are less impactful, but the overall best model ends up being trained purely with inpainting.

In terms of model architecture, the frozen regression inpainting model can be viewed as another baseline since it is equivalent to the MSA transformer adapted to RNA. This model is roughly on par with the DCA baseline but outperformed quite significantly by the CoCoNet baseline.

Neglecting the jigsaw case for now, finetuning the backbone parameters by unfreezing them during downstream training improves the score in terms of MCC to surpass CoCoNet performance. Applying XGBoost instead of the simple regression model to the pre-trained backbone output without finetuning increases performance even further than finetuning. A combined approach using the attention maps of the fine-tuned backbone as input for an XGBoost model yields the best results overall.

For top-L precision, the difference between fine-tuned regression and frozen XGBoost is much smaller, where the fine-tuned regression is even better in some cases. However, fine-tuned XGBoost remains the best model, at least as long as the correct early-stopping metric is used during the backbone finetuning. In fig. 3.9, sets of models, whose performance is averaged, are separated by whether a global metric or a top-L metric was used. I.e. the models where early stopping was performed with top-L metric as the monitor metric are grouped together. The impact on this is larger on the top-L precision of the final model but still observable for the (global) MCC score. I explore this effect in more detail later in this section.

Returning to the jigsaw task, particularly the frozen models' performance suffers. Fine-tuning can recover it partially, albeit not reaching the other models of the same training setup.

Early-Stopping Metric Impact

In conjunction with other regularization measures (cf. table 3.1), we use early stopping during downstream training. For this, we split off a validation dataset from the training set and stop the training when the validation performance begins to degrade. The validation performance can be measured in terms of several different metrics. For fine-tuned XGBoost models, early stopping comes into play at two stages: first during the finetuning training of the backbone and then during the training of the downstream forest model itself. Figure 3.11 shows the final test performance over the metrics used for both backbone finetuning and final XGBoost downstream training.

The finetuning early-stopping metrics are more impactful, with the top-L style metrics giving consistently better performance than the global metrics, including validation loss. The only notable downstream metric is F_1, which degrades the performance gain granted by the choice of finetuning metric and brings it down to the level of the global ones. In particular, optimizing exclusively for MCC does not result in the best model in terms of MCC.

Feature Importance

One of the usually touted advantages of XGBoost models is their interpretability. Note that since we use opaque, learned latent representations from a deep neural network as input, this benefit is diminished. Figure 3.12 exemplarily shows the absolute parameter values of two regression models and the feature importance maps of two XGBoost models. The XGBoost feature importance score is the number of times the feature is used to split the data across all trees.

While the frozen regression model focuses on a small set of features, the weights of the fine-tuned regression spread more evenly across more features, favoring features from later layers of the backbone. The XGBoost models reverse this pattern, i.e., the fine-tuned model favors earlier features than the frozen one.

RNA Accessible Surface Area Prediction

To examine the generalizability our approach of finetuning pre-trained latent inputs for a downstream model that is itself not back-propagatable, we consider a related task.

The task is predicting an RNA's accessible surface area (ASA), which also contains structural information. We use the same pre-training inpainting task and pre-training data here. However, the task structure differs from the contact prediction task solved before. Instead of a binary classification for each pair of residues, it is a regression task for each residue, where the input is the backbone's latent embedding, that the task heads also receive during pre-training, rather than the stacked latent attention maps. The downstream dataset for this task is adapted from the RNA temperature adaptation dataset [139], consisting of 182 MSAs. We split this into training and validation set and report the Pearson correlation coefficient between prediction and label ASA over the validation dataset. In the absence of a completely separate test set, we do not perform a hyperparameter search for this task and use the same parameters as for the contact prediction. Since the backbone model is limited to sequence lengths of 400, we only use the middle 400 residues for a prediction.

Table 3.6 shows the Pearson correlation coefficients for the ASA prediction over the different downstream models. With the frozen XGBoost model being worse or on par with the fine-tuned neural network, the pattern observed for the contact prediction task does not persist exactly. Also the jigsaw task is not as degrading as in the contact prediction task and performs better than bootstrapping. However, the fine-tuned XGBoost model pre-trained on just inpainting remains the best result. This model also beats the baseline performance of 0.63 reported by Yang et al. [139].

	Frozen NN	Tuned NN	Frozen XGB	Tuned XGB
Jigsaw	0.1731	0.4849	0.4877	0.6740
Contrastive	0.1832	0.5292	0.4930	0.7194
Bootstrap	0.1115	0.4938	0.4927	0.6984
Inpainting	0.1934	0.5222	0.4905	0.7443

Table 3.6.: Pearson correlation coefficients for ASA prediction.

3.4.2.1. Examples

Figures 3.13 and 3.15 show example structures with the predicted contacts overlayed. For the contact maps, the upper triangle shows the results for the best inpainting finetuned XGBoost model. The lower shows the best model with a frozen backbone.

For `3ndb` most of the false positives are slightly above the contact definition threshold. There is one prediction, which is not compatible with the structure. The frozen model does not make the same mistake here, but has more false positives clustered closer to the true contacts.

`5di2` shows a cluster of false positives for both models. However, the structure indicates, that there is either a transient interaction where the false positive contacts are predicted, or that something is missing from the structure. Either this part of the molecule interacts with a different RNA or protein, or a small ligant might bind in the pocket.

Figures 3.14 and 3.16 show the corresponding contact maps.

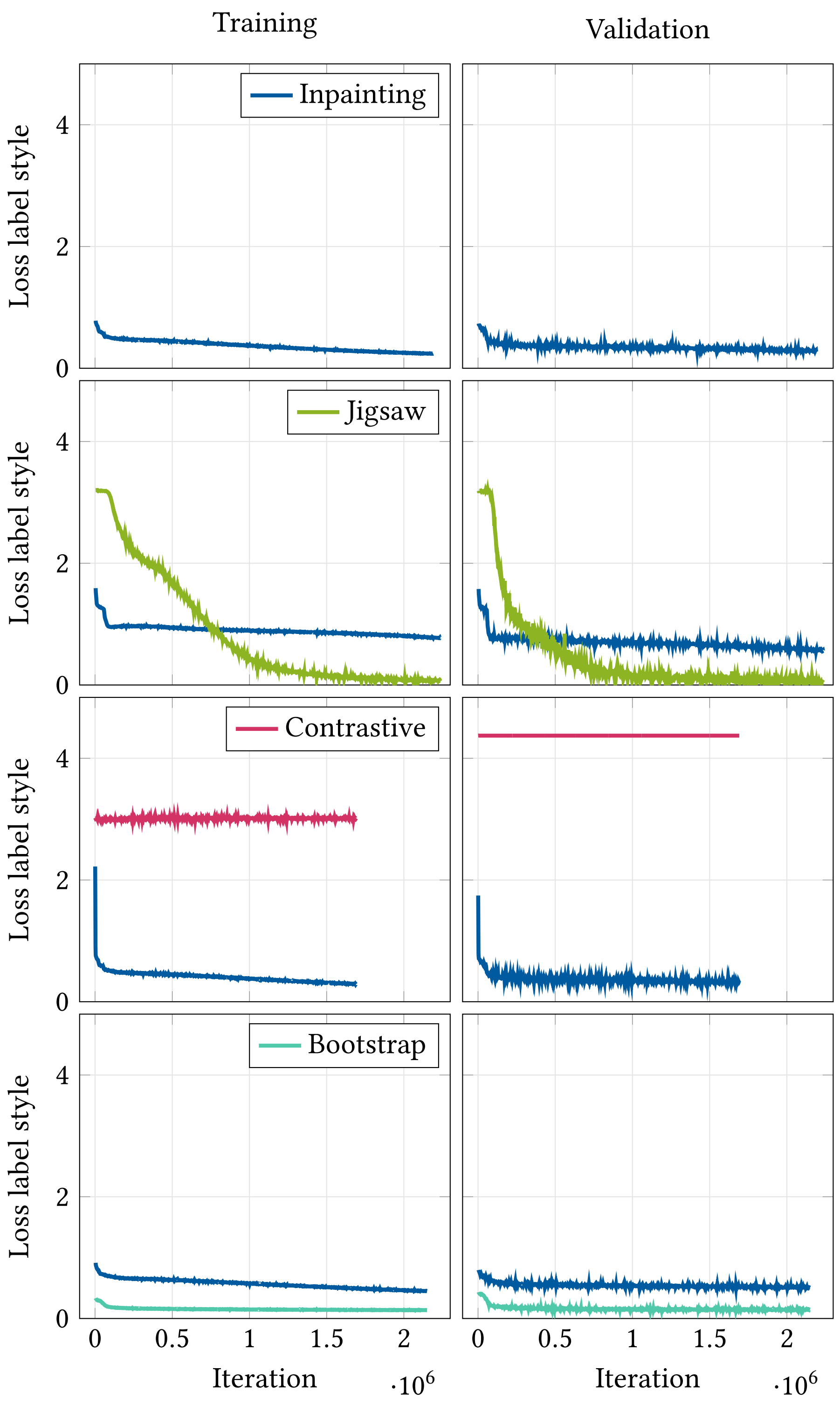


Figure 3.8.: Upstream loss components over the course of pre-training.

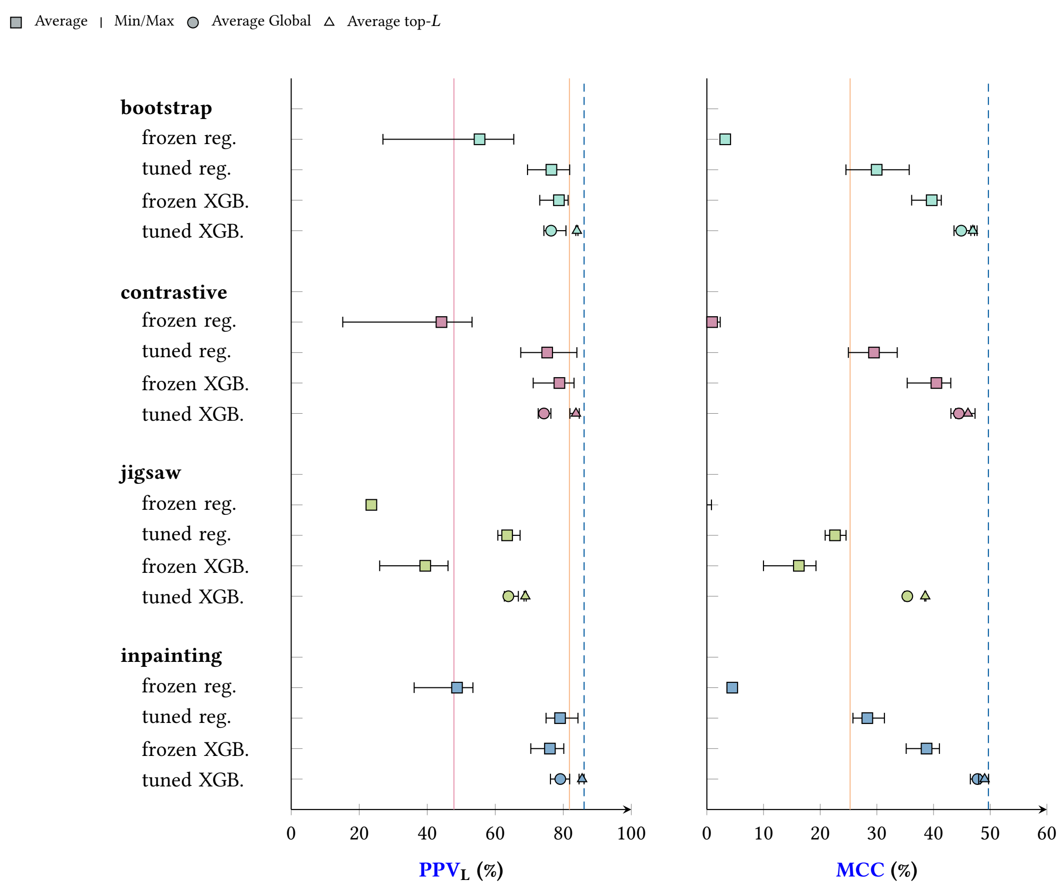


Figure 3.9.: Downstream model performance for different pre-training task combinations and downstream training procedures in terms of top-L precision (PPV_L) and MCC on the test set. The red, orange, and blue lines show the DCA baseline, the CoCoNet baseline, and our best model, respectively. Squares mark the score averaged over the different early-stopping metrics. The error bars show the best and worst score. Circles and triangles for the fine-tuned XGBoost instead show the score averaged over global and top-L style metrics, respectively.

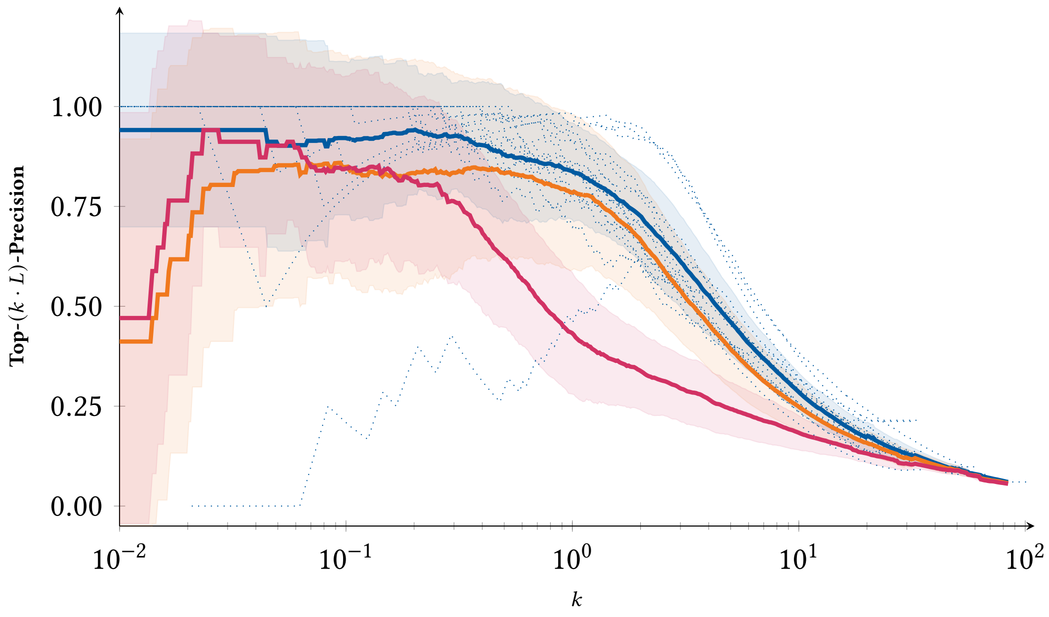


Figure 3.10.: Macro-top-$(k \cdot L)$ precision. The light dotted lines show individual samples of the test set. The thick lines show the average. DCA, CoCoNet, and inpainting tuned XGBoost are shown in red, orange, and blue, respectively. The band around the average shows standard deviation.

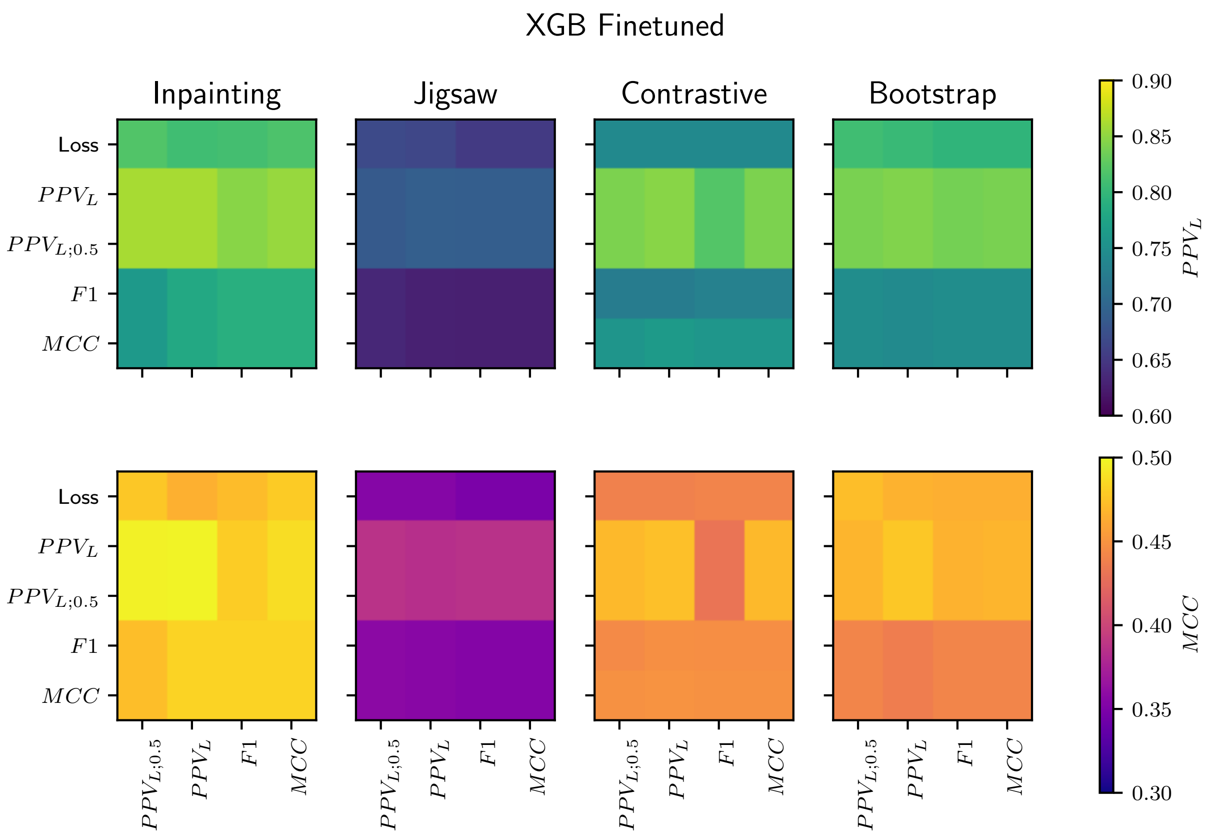


Figure 3.11.: Downstream performance for the fine-tuned inpainting XGBoost model in terms of top-L precision and MCC. The metrics used during finetuning are listed along the y-xis and the metrics used during XGBoost training are listed along the x-axis.

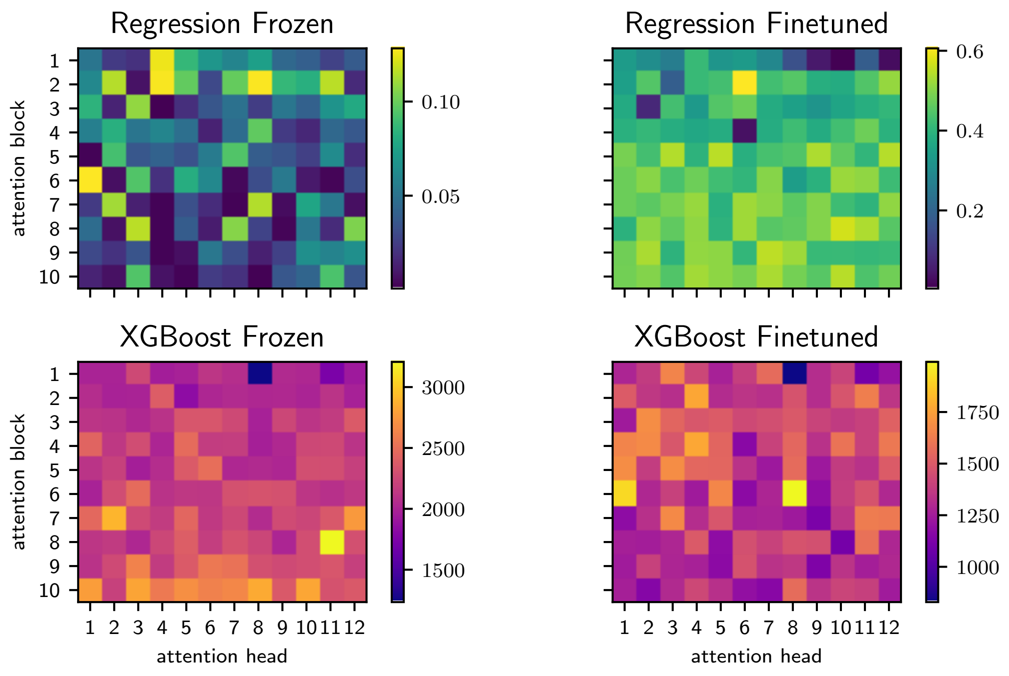


Figure 3.12.: Absolute values of regression model parameters (top) and XGBoost feature importance scores (bottom) for a selection of downstream models with frozen (left) and fine-tuned (right) backbone, respectively. The regression models are pre-trained with inpainting and bootstrapping and optimized for F_1 score. The XGBoost models are pre-trained with inpainting and optimized for top-L precision. The fine-tuned XGBoost model uses top-L precision during both finetuning and actual downstream training.

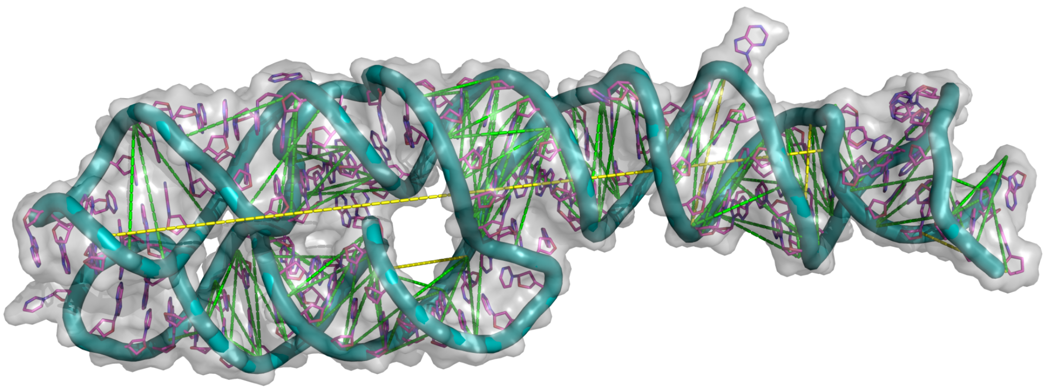


Figure 3.13.: 3D visualization of an RNA (PDB: 3ndb). Green dashed lines indicate correctly predicted inter-residue contacts, yellow ones refer to false positives.

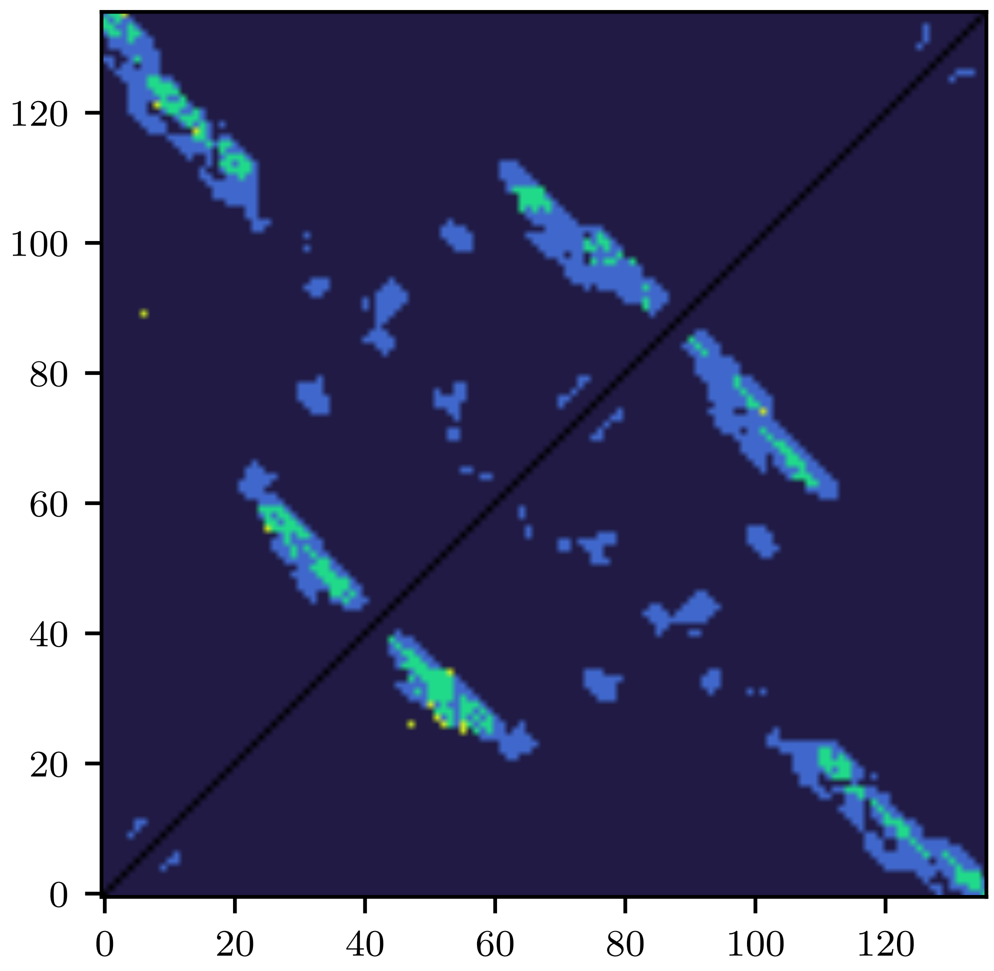


Figure 3.14.: Contact map (PDB: `3ndb`) – unfrozen vs. frozen. The upper left part shows the top *L* contact predictions for the best model with unfrozen backbone, the lower right one for the best model with frozen backbone. Green pixels refer to true positives, yellow to false positives, light blue to false negatives, and dark blue to true negatives.

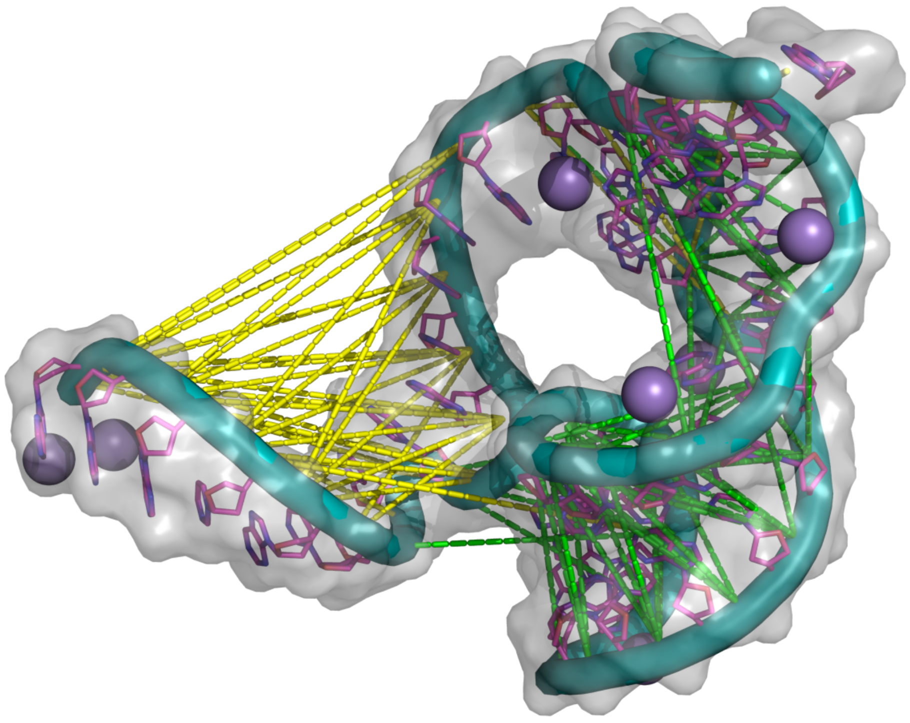


Figure 3.15.: 3D visualization of an RNA (PDB: 5di2). Green dashed lines indicate correctly predicted inter-residue contacts, yellow ones refer to false positives.

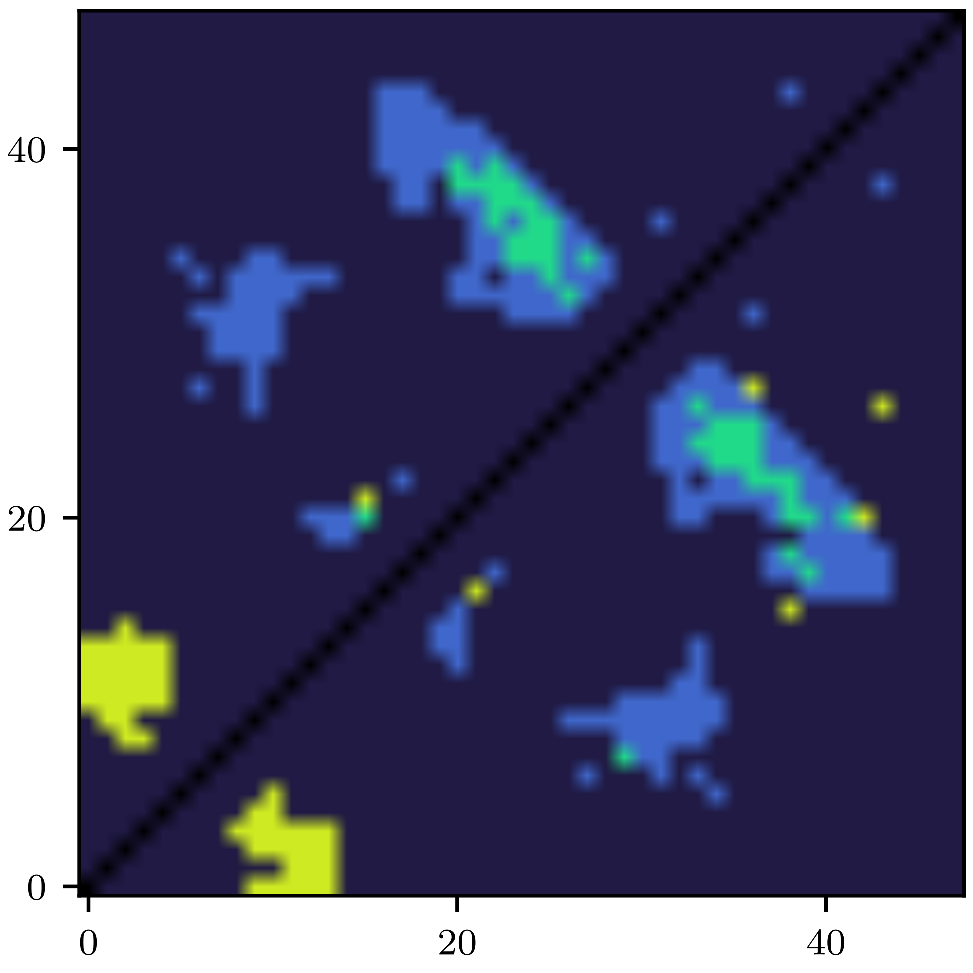


Figure 3.16.: Contact map (PDB: 5di2) - unfrozen vs. frozen. The upper left part shows the top L contact predictions for the best model with unfrozen backbone, the lower right one for the best model with frozen backbone. Green pixels refer to true positives, yellow to false positives, light blue to false negatives, and dark blue to true negatives.

3.5. Discussion

In summary, we demonstrated the benefit of combining two techniques, that is, self-supervised pre-training and random forest models, in order to extract structural information from evolutionary information. Despite not being end-to-end trainable, our approach improves upon the baseline using the sparse labeled data efficiently. More particularly, the finetuning of the pre-trained latent representation serving as inputs to the more data efficient random forest model is an approach that may prove effective in other label-sparse settings, which are common in ML tasks from life sciences. The generalizability to the related but different task of ASA prediction is a first indicator towards this direction.

3.5.1. Of Note

Comparing the models using the output of the frozen backbones as input, it may be intuitive that the XGBoost model with a higher capacity performs better than a single neural network layer. However, the improved performance of frozen XGBoost over the fine-tuned end-to-end trained model suggests that the former can inherently model aspects of the problem better.

Different pre-training tasks and combining them with multi-tasking did not result in measurable benefits. Maybe, the tasks as designed were not difficult enough or the model found a shortcut. Furthermore, we could confirm the observation noted by the authors of the MSA Transformer [122] that the masking for inpainting is more beneficial when performed on random single tokens rather than entire columns, even though this makes the task more difficult. Intuitively, the model can often insert the most common token in the column to find the correct one, making this masking regimen highly inefficient. The contrastive task seems to be too easy, trivially embedding sequences originating from different MSAs far apart.

3.5.2. Outlook

In the following, we want to briefly discuss avenues for possible future research building on the work described before.

As mentioned previously, our approach can be adapted to other settings. One such setting is the growing field of multiomics. While deep learning is already widely applied in genomics or meta-genomics, e.g. in transcriptomics, classical ML techniques like independent component analysis are often used [140]. A combined approach like the one we proposed here might offer an efficient route to learning transferable patterns from different sources of information, conceptually similar to ImageBind [141].

That the jigsaw task in particular does not seem to aid in contact prediction is somewhat disappointing. In the original image processing variant, there were additional augmentations intended to prevent the model from learning shortcuts to the pre-training task, like taking a random crop from each jigsaw partition. However, these kinds of augmentations are more difficult to design for MSAs than for images and the configuration space for up-stream tasks that has to be searched for viable hyperparameters grows even larger. While the hyperparameter optimizer presented in chapter 5 [3] makes such a search feasible, it was not available yet at the time of conducting this part of the project.

Why the other two pre-training tasks are not particularly useful is more intuitive. The contrastive task might benefit from more closely modeling it to the equivalent image processing task. Instead of embedding an average over the entire sequence, taking a fixed size crop would make the task more meaningful since the intra-class distance is not trivially low. In addition, this would remove the need for the currently required reduction operation.

For bootstrapping, a more sophisticated generator model might force the backbone to learn more relevant patterns. However, a model collapse, where the model only samples from the sequences already present in the MSA, would have to be prevented, possibly by combining it with an additional contrastive component. Still, MSAs with very few sequences would remain a problem for this task.

For inpainting, a mechanism akin to (semi-) hard example [142] mining or task difficulty scheduling might be beneficial.

On a more technical level, the synchronization overhead described in section 3.3.6 could be reduced by bucketing, i.e., building a super-batch during sampling and grouping samples of similar size together into the actual training batches. This is a commonly used technique for ML problems where inputs might be of variable size [143], and the samples in a single batch have to be padded to be able to be batched together. While usually applied to minimize the memory footprint, bucketing could improve the computational efficiency during pre-training at the cost of some sampling randomness.

Finally, foregoing the processing of entire MSAs and returning to single-sequence models would enable the utilization of larger pre-training datasets. Such a backbone model could and would need to be a lot larger than the one used here [122, 144], since it has to encode the relevant information from the evolutionary record within its model parameters rather than extracting it from the input and storing it in the latent representations.

The contact definition also has room for improvement. We used a simple distance cutoff, but the way two RNA bases can interact is limited by their geometry. Instead of a binary contact classification the model could be trained to predict which of the interactions is taking place.

4. Loss Scheduling

In this chapter we present work that has been published in the proceedings of the International Conference on Machine Learning and Applications [2]. *The main contribution is a dynamic loss scheduling mechanism, that deforms the optimization landscape during training in order to exploit properties of different loss functions and limited available training data as well as possible.*

4.1. Introduction

The training process of a classification model is–next to the model itself–to a large part governed by the choice of loss function and the sampling of training data. Class imbalance often inherent in semantic segmentation data can have adverse effects on the shape of the optimization landscape traversed during training, and thus the performance of the final model. For example the background class is often more prevalent than the classes of interest. In extreme cases, an easily found local optimum is one, where minority classes are completely ignored in favor of a majority class. Arbitrarily different loss functions can lead to optima in different positions in weight space, but different roughness or frustration, i.e., competition between local minima, of the loss landscape [145].

There are three possible adaption avenues: the model architecture, the data, and the loss function. Choosing better model hyperparameters will be the subject of chapter 5. As for data, it is in principle possible to resample the data to achieve a more balanced class distribution. In practice, however, this often not viable due to a drastic data reduction, especially infeasible in data sparse scenarios, posing the risk of not optimally utilizing the given, valuable information. The acquisition of additional data is usually challenging in its own right and usually dismissed on the basis of high time and cost investment. This leaves the loss function as the only remaining option.

Loss functions for class imbalance in classification [14] and for class imbalance for semantic segmentation problems [146] have been proposed. We present a more generic approach inspired by biology. In a nature vs. nurture view, where the model architecture represents the former and dataset and loss the latter, it might be beneficial to use different signals at different stages of the models training. The goal is to switch to a different training trajectory, which ultimately converges on a different optimum than the one found with gradient descent on a not distorted landscape. To maintain the minimum that the model finally converges on to be the one of the original loss function, the trajectory dynamically and gradually transitions from the distorted landscape back to the original one. We call the process of transitioning "*loss scheduling*".

4.2. Related Work

The proposed concept touches upon several different aspects, that have been subject to active consideration of general machine learning research. Here, we briefly present the preceding work on different metrics for model training and evaluation [147], compound or auxiliary losses [120], loss landscape theory [148], class imbalance [14], and model calibration [59] in more detail.

4.2.1. Class Imbalance

Class imbalance describes the phenomenon, where different class labels occur in a dataset used for machine learning with different frequencies. In many real-world problems class imbalance can degrade classifier performance for models trained on an imbalanced dataset, especially in terms of generalization [16, 15, 149]. In extreme cases, an underrepresented class might be ignored by the classifier altogether. As a rule of thumb classifiers should be initialized such, that their predictions reproduce the class distribution in the training set [150].

4.2.2. Semantic Segmentation

Semantic segmentation [15, 151] is an established problem class in image processing. The task is to partition an image into two or more regions. This is a special case of classification, since each pixel is classified rather than the entire sample image. Often there are one or more foreground classes, which are the pixels of interest, and a background class, containing the rest of the image. Usually, the background class is vastly overrepresented compared to the foreground classes. The contact map prediction described in chapter 3 can be viewed as a semantic segmentation problem. Depending on the contact definition used, the background class 'no contact' can make up up to 98% of the pixels in one contact map [126]. Because the class imbalance of image segmentation tasks is already manifest at a sub-sample level, it can not be addressed by re-sampling or building an unbiased dataset. In the context of segmentation, two types of class imbalance can be distinguished. At the sample level, where the class label applies to the entire sample image as a whole, and at the pixel level where the label only applies to a single element of a more complex data structure making up the complete sample. Sample-level class imbalance leads to a concentration of the underrepresented class to few batches and hinders training convergence. Similarly to classification tasks, this type of imbalance can be addressed during data collection by including class representatives uniformly [55]. Pixel-level class imbalance where only few pixels of a sample containing a particular class are harder to address at the data collection stage. One approach that does not involve modifying the loss function is stochastic sampling, which can also be used for sample-level class imbalance. For segmentation training, ground-truth pixels are masked out with a probability proportional to their class label frequency. This approach is not further explored in this work for two reasons: First, not training on large parts of the training data while still performing all computations makes the already expensive training process even less efficient both in terms of computational expense and data efficiency. Second,

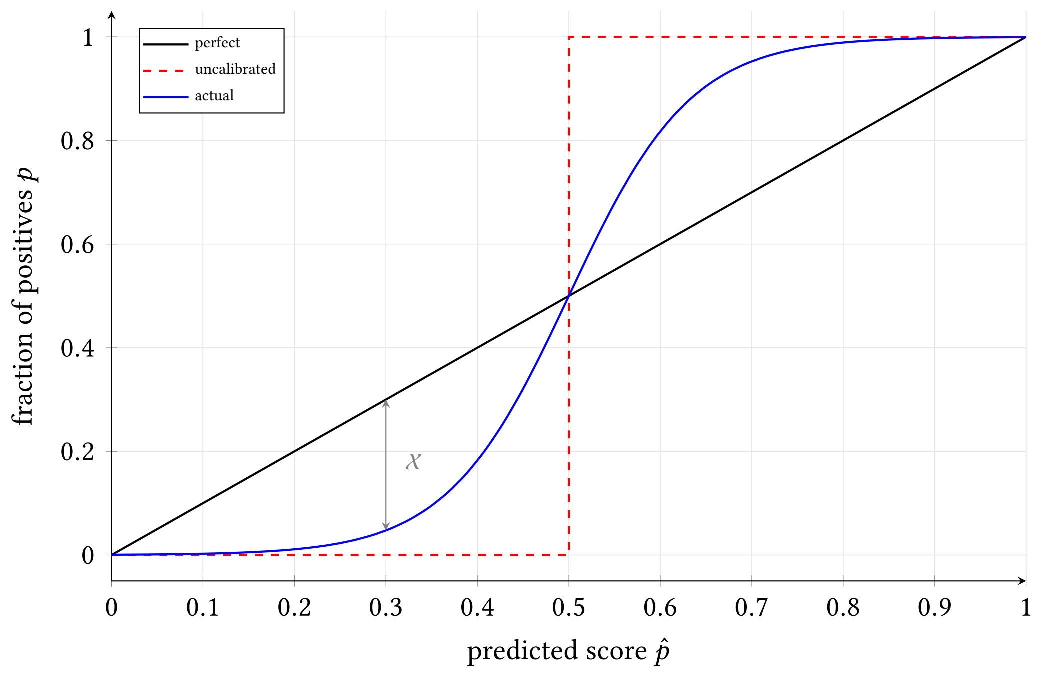


Figure 4.1.: Idealized schematic depictions of reliability diagrams to compare the confidence of the model output to the ground-truth accuracy for each probability.

it has been reported [152] that stochastic oversampling on classification tasks causes a model to overfit the underrepresented class.

4.2.3. Model Calibration

Neural network classifiers produce confidence scores, i.e., likelihoods, or posterior probabilities that the input sample belongs to one or more classes. For a well calibrated predictor, this confidence matches the frequency, with which the classifier is correct. In other words, if the model predicts an input belonging to a class with 90% confidence one hundred times, 90 of the samples should belong to the predicted class. Figure 4.1 shows an illustration of calibration and reliability.

Neural network classifiers have been shown to often be overconfident [59]. To quantify calibration of a binary classifier with parameters $\boldsymbol{w}$, the expected calibration error (ECE) [153] can be used. To this end, the B predictions over a set of samples are split into N bins. The contribution of each bin to the total error are then given by the difference of the average predicted score for that bin $\overline{\hat{y}}$ and the actual fraction of positive labels P/B_n, weighted by the number of samples in that bin:

$$\mathrm{ECE}(\boldsymbol{w}) = \sum_{n=1}^{N} \frac{B_n}{B} \cdot \left| \overline{\hat{y}}(\boldsymbol{x}_i|\boldsymbol{w}) - \frac{\mathrm{P}_n}{B_n} \right| \tag{4.1}$$

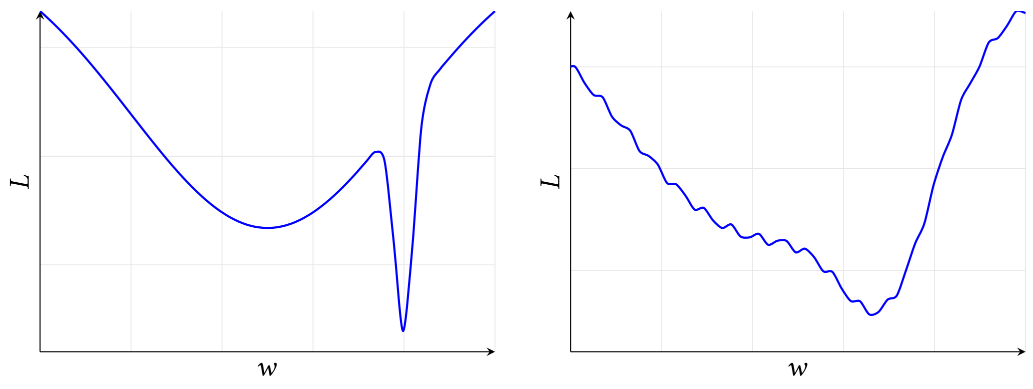


Figure 4.2.: Idealized loss landscapes. A landscape with a broad thus easy to find shallow minimum with a narrow global minimum (left) and a noisy landscape where the barrier between the two minima is removed, but the global minumum has shifted slightly (right).

4.2.4. Loss Landscapes

A loss landscape is the mapping of the model parameter space to the corresponding loss value given a model, a loss function, and a set of inputs. During the training of a neural network, this landscape is explored to find a minimum. Due to the batched nature of the employed optimization algorithms, the entire training dataset is not seen at once. Consequently, the landscape fluctuates slightly between steps. Such fluctuations can be a desired side-effect to allow the model escaping from a local minimum. Trapping in local minima is a potential problem, since neural networks are usually trained with gradient descent based algorithms [71, 69].

Several factors determine the shape of the loss landscape and how it is traversed, which in turn determines how well it can converge. Among them is the loss function itself. When the target metric can not be directly optimized, usually because it is not differentiable, there are often multiple possible surrogate functions, like MSE or cross-entropy loss that can be used to measure predictive performance. Figure 4.2 shows schematic illustrations of related loss landscapes with different roughness and barriers between local optima.

As mentioned in chapter 2, the default classification loss is CE, which can also be used for segmentation when viewing the task as a pixel-wise classification. CE is a proper scoring rule [154], in theory driving the model to predict the true probability distribution. A class weighted variant (WCE) gives more weight to underrepresented classes, by increasing the training signal from misclassified members of those classes.

$$\mathrm{WCE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{c \in \text{classes}} \boldsymbol{w}_c \boldsymbol{y}_c \log \hat{\boldsymbol{y}}_c \boldsymbol{w}_c = \left(\frac{N_c}{\sum_i N_i} \right)^{-1} \tag{4.2}$$

The class weight $\boldsymbol{w}_c$ is usually set to the inverse class frequency, either computed over the given batch or the entire dataset.

4.2.5. Dice Loss

Dice loss is a differentiable measure, related to the intersection over union (IoU), of the overlap between two areas, in this context usually the ground-truth mask and the predicted segmentation. A generalized version [146] includes a smoothing parameter and class weights to mitigate region size effects. The definition for class weighted dice loss (WDL) we use in the following is:

$$\mathrm{WDL}_x = 1 - 2\frac{\sum_{c=1}^{C} w_c \sum_x y_{cx}\hat{y}_{cx} + s}{\sum_{c=1}^{C} w_c \sum_n y_{cx}^2 + \hat{y}_{cx}^2 + s + \varepsilon} \tag{4.3}$$

with class weights w_c for class c, the smoothing parameter s, and ε for numerical stability. Dice loss is established for segmentation tasks, since it performs well for class imbalanced problems, however, it does not tend to produce well calibrated models [155].

4.2.6. Focal Loss

Reweighting individual pixel loss contributions is a more fine-grained approach compared to emphasizing the contribution of an entire class of pixels. Focal loss [14], originally a classification loss, modifies cross entropy, enhancing the loss of individual difficult-to-learn training samples.

$$\mathrm{FL}_x = -(1-\hat{y}_x)^\gamma y_x \log(\hat{y}_x) = (1-\hat{y}_x)^\gamma \mathrm{CE}_x\,, \tag{4.4}$$

where γ is an adjustable focusing parameter. The underlying assumption for the development of focal loss is, that the loss contribution of the easy, already correctly classified samples is overwhelming the signal from the few difficult ones by sheer volume. Focal loss is not a proper scoring function and thus does not produce well calibrated models.

4.2.7. Loss Max-Pooling

Similarly to focal loss, loss max-pooling [156] uses a modulating factor to enhance larger contributions to the overall loss while suppressing smaller ones. In short, the max-pooled cross entropy loss is then written as:

$$\mathrm{CEMP}(\boldsymbol{y}) = w_{\boldsymbol{y}} \cdot \boldsymbol{y} \log(\hat{\boldsymbol{y}}) = w(\mathrm{CE}(\boldsymbol{y})) \cdot \mathrm{CE}(\boldsymbol{y}) \tag{4.5}$$

The weight $w(\boldsymbol{y})$ emphasizes the largest pixel losses of a single sample image and ignores the smallest ones and is given by:

$$w(\boldsymbol{y}) = \begin{cases} \tau & \boldsymbol{y} \in \mathcal{J} \\ \tau \left(\frac{\mathrm{CE}(\boldsymbol{y})}{\alpha}\right)^{q-1} & \boldsymbol{y} \in \overline{\mathcal{J}} \\ 0 & \text{else} \end{cases} \tag{4.6}$$

with $\tau = n^{\frac{-1}{q}} \cdot m^{\frac{-1}{p}}$, $q = \frac{p}{p-1}$, the top m loss values of one image $\mathcal{J}$, $\alpha = \max\left(\overline{\mathcal{J}}\right)$. p and m are hyperparameters that have to be set by the model designer and n is the number of pixels in the sample. We apply the max-pooling over all pixels in an entire batch, rather than over the pixels of each image.

4.3. Method

4.3.1. Loss Scheduling

To manipulate the minimum the segmentation model converges on, we dynamically distort the optimization landscape, transitioning from one of the losses described previously to cross entropy loss. The total training loss is

$$L_x = w_{\mathrm{CE}}(i)\mathrm{CE}_x + w_{\mathrm{D/F}}(i)(\mathrm{D/F})L_x$$

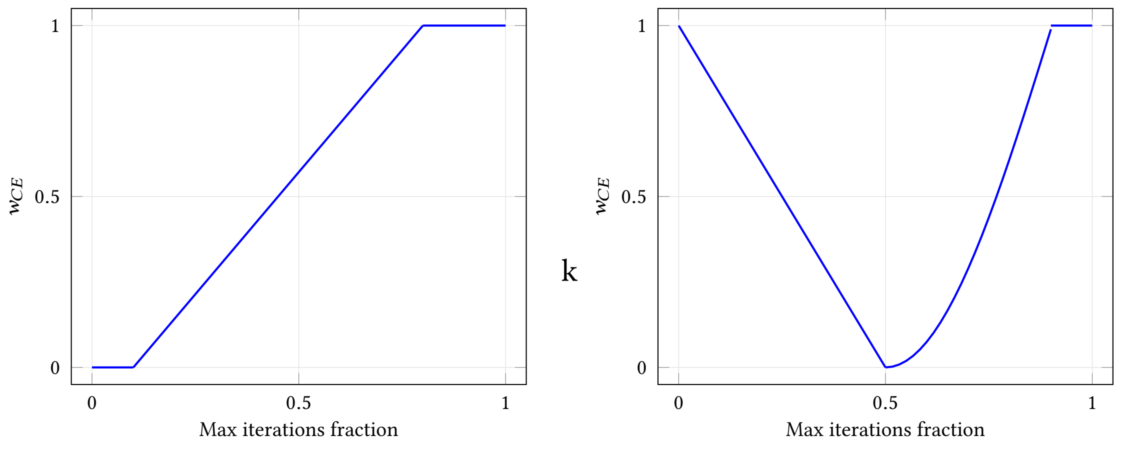
where the weights w always sum to 1. We explore two different transition schedules here. The "naive" schedule is a simple linear ramp with constant warmup and cooldown periods:

$$w_{\mathrm{CE}}(i) = \max\left(0, \min\left(1, \frac{i-s}{e-s}\right)\right) \tag{4.7}$$

at iteration i and beginning and end of the linear transition s and e. The "alternating" schedule begins with cross-entropy and linearly ramps to the other loss before returning to cross-entropy in a sinoidal curve (cf. fig. 4.3).

$$w_{\mathrm{CE}}(i) = \min\left(1, \max\left(0, 1 - \frac{i}{s}\right)\right) + 1 - \cos\left(\min\left(\frac{\pi}{2}, \max\left(0, \frac{\pi\,(i-s)}{2\,(e-s)}\right)\right)\right) \tag{4.8}$$

Here s and e are beginning and end of the sinoid.



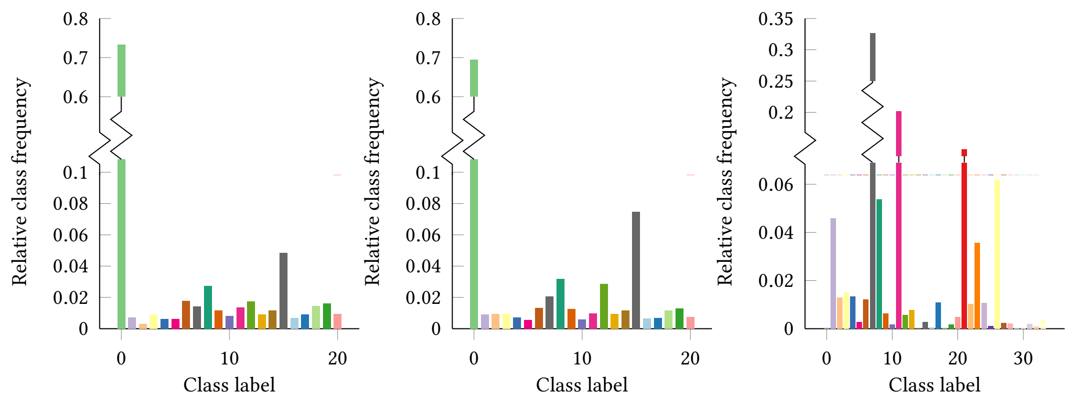
(a) Naïve (N).

(b) Alternating (A).

Figure 4.3.: Different loss scheduling schemes. Shown is the cross entropy loss contribution, i.e., weight w_{CE}, over the fraction of total optimization epochs.

4.3.2. Model Architecture

We explore two different convolutional model architectures for image segmentation to empirically validate the loss scheduling approach. The fully convolutional network [157]



(a) VOC. (b) SBD. (c) Cityscapes.

Figure 4.4.: Relative class frequencies of evaluated datasets. The class distribution of VOC is similar to the one of SBD

(FCN) consists of convolutional layers, batch normalization layers, ReLU activations, and a deconvolution to produce the per-pixel prediction. The final activation layer is a per-pixel softmax, which produces class likelihood predictions.

The second model is DeepLabV3 [158]. This model includes atrous or dilated convolutions in addition to the default operations included in the FCN. These are intended to widen the receptive field of a layer without incurring the cost of additional parameters.

For both models, we use the stock implementation that are part of the torchvision [63] library with the ResNet-50 backbone, but without pre-trained weights.

4.3.3. Data

For training and evaluation we use three different datasets. The Pascal Visual Object Classes (VOC) dataset [16] contains 11 540 images and 20 different classes. The Semantic Boundaries Dataset [149] (SBD) with a total of 11 355 images uses the same 20 different classes, The Cityscapes [17] dataset contains 25 000 images and 30 different classes. Figure 4.4 shows the class frequencies for the three datasets.

4.3.4. Losses

The losses we use for scheduling are weighted and unweighted versions of cross-entropy and dice loss. Focal loss and max pooled cross-entropy are included for comparison as other examples of strategies to address similar problems.

4.3.5. Metrics

We employ several metrics described in chapter 2 to quantify and compare the quality of the trained classifiers. Furthermore, we introduce a score to quantify the calibration of a model. As depicted in fig. 4.1, a well calibrated classifier would produce class-membership probabilities rather than an opaque confidence score. The calibration is estimated based on calibration curves: We first bin the model output for each pixel and class. Then we plot the fraction of positive labels for each bin over the mean output. As a measure of a model's reliability, we define a scalar calibration score (CAL), which is computed by a metric similar to the expected calibration error (ECE) introduced in [153]:

$$\text{CAL} = 1 - \frac{4}{N} \cdot \sum_i |p_i - \hat{p}_i| = 1 - \frac{4}{N} \cdot \sum_i |\mathcal{X}_i| \tag{4.9}$$

4.3.6. Computing Environment

All experiments have been performed on one of 15 computational nodes with commodity components hosted at the Jülich Supercomputing Centre (JSC). Each node is equipped with an Intel Xeon Gold 6126 CPU @ 2.60GHz as host processor, 256 GB of DDR3 main memory, and four NVidia Tesla V100 GPGPUs as AI accelerator with 32 GB VRAM per card. The GPUs communicate internally via an NVLink interconnect and are optimized for GPUDirect communication across node boundaries with 2x Mellanox 100 Gbit EDR InfiniBand links. A Redhat Enterprise Linux with kernel version 3.10.0 has served as operating system. The driver for the NVidia GPU was 418.87.00 and the CUDA runtime was in version 10.1. The GPU driver, and CUDA runtime were in versions 418.87.00 and 10.1, respectively. All models were implemented in PyTorch 1.4.0 [63] and torchvision [159].

4.4. Results

Table 4.1 contains the raw numbers and some additional data, including the performance for SBD and VOC models using pre-trained weights trained on the MS-COCO dataset [15]. These pre-trainend models perform consistently better than any other model we trained purely on one of the three smaller datasets. Focal loss and max pooled cross entropy produce performance metrics comparable with the other traditional losses but have the worst calibration scores. Models using a traditional loss or a compound loss with unweighted cross entropy have similar performance and calibration metrics. Compound or scheduled losses with class weighted cross entropy sacrifice global accuracy for class-averaged recall. On VOC and SBD, DeepLabV3 tends to have better overall accuracy, F_1, and recall but worse calibration compared to FCN with the exception of dice loss. On Cityscapes only losses scheduled towards weighted cross entropy and DLWCE compound loss specifically produce models that do not ignore at least one minority class. The different schedules do not display a consistent trend over the range of models. We also tested loss scheduling from focal loss to cross entropy with similar results as for the dice loss version. Figure 4.5 shows model recall over calibration score to make a comparison of the different scenarios easier. All models have in common that the single, static losses tend towards lower values for both recall and calibration, whereas the compound losses, scheduled or static trend towards the top right. Comparing the Cityscapes to the other two larger datasets, the values for recall are similar, but the calibration suffers. DeepLabV3 tends to trade in some recall for calibration on average. It also performs better with CEMP and FL than FCN. Again the choice of schedule has less of an impact than the component losses. For VOC and

SBD the scheduled losses tend to have slightly better calibration scores. In particular for models trained with weighted cross entropy as one component, rather than unweighted cross entropy. The clearest trend is shown in the case of DeepLabV3 and Cityscapes, i.e. the more complex model with the smaller dataset. Here, the cluster of models trained with weighted cross entropy as a component loss is the least scattered. Note that the model trained with pure weighted cross entropy is tied to the cluster with models trained with unweighted cross entropy as component loss as in the other scenarios. Within the XXWCE cluster the scheduled losses produce slightly better calibrated models.

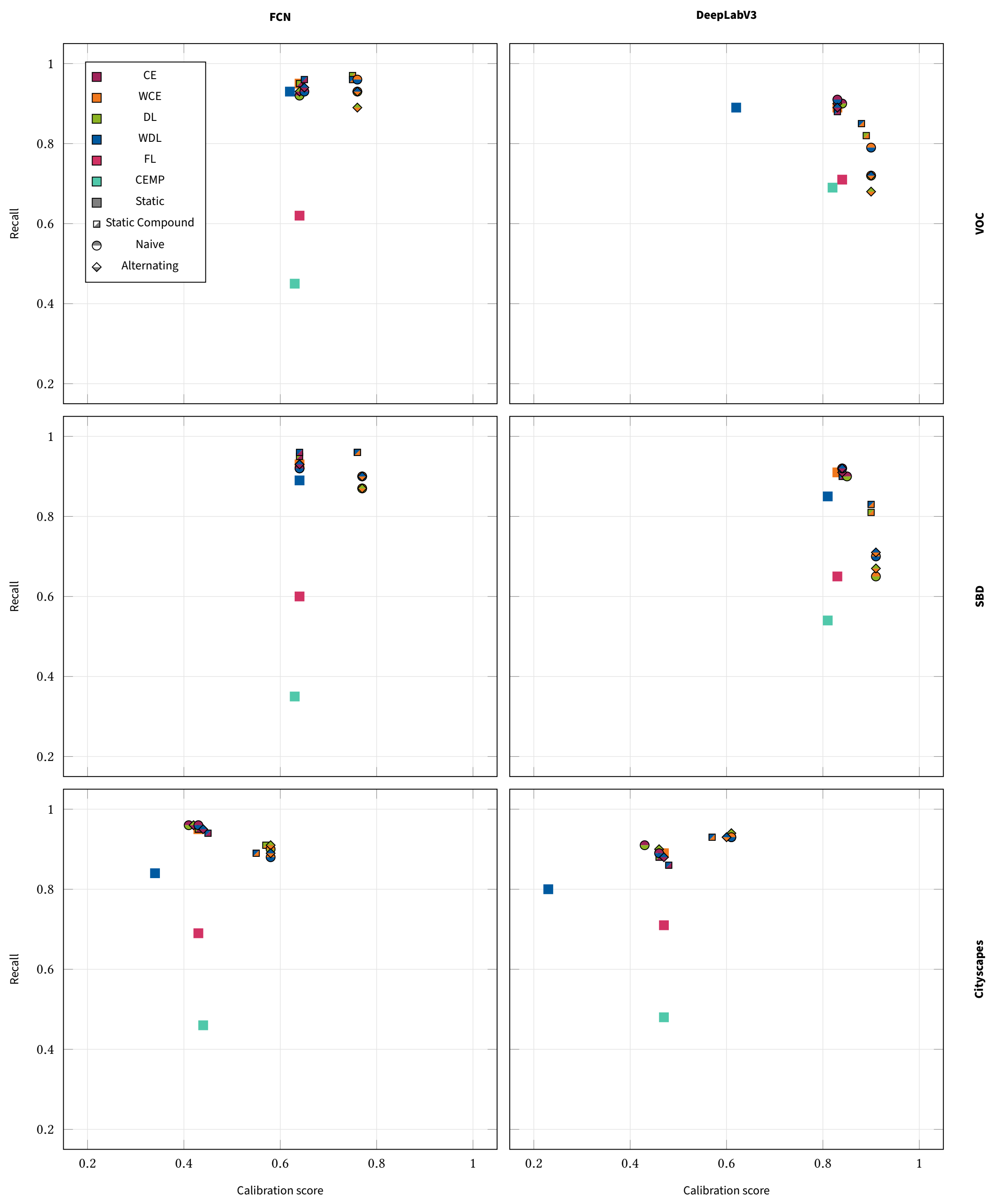


Figure 4.5.: Model performance in terms of recall over calibration score.

Table 4.1.: Results of the empirical evaluation of the FCN and DeepLabV3 neural network architectures using all loss functions on the VOC, SBD, and Cityscapes datasets. Losses prefixed with W are class-weighted. Losses suffixed with MP are max-pooled. Two losses separated by N or A are naïvely or alternatingly scheduled respectively. CE and DL denote cross entropy and dice loss, respectively.

		VOC						SBD						Cityscapes					
		ACC	F_1	SEN	CAL	S_{ACC}	S_{SEN}	ACC	F_1	SEN	CAL	S_{ACC}	S_{SEN}	ACC	F_1	SEN	CAL	S_{ACC}	S_{SEN}
FCN	pretrained	0.95	0.85	0.85	0.91	0.93	*0.87*	0.92	0.82	0.81	0.93	0.93	*0.87*	-	-	-	-	-	-
	WCE	0.89	0.68	0.64	0.95	0.92	*0.77*	0.87	0.68	0.64	0.93	0.90	*0.76*	0.84	nan	0.43	0.95	0.89	*0.59*
	WDL	0.89	0.65	0.62	0.93	0.91	*0.74*	0.87	0.67	0.64	0.89	0.88	*0.75*	0.83	nan	0.34	0.84	0.83	*0.49*
	FL	0.88	0.67	0.64	0.62	0.73	*0.63*	0.87	0.67	0.64	0.60	0.71	*0.62*	0.83	nan	0.43	0.69	0.75	*0.53*
	CEMP	0.88	0.66	0.63	0.45	0.60	*0.53*	0.86	0.66	0.63	0.35	0.50	*0.45*	0.83	nan	0.44	0.46	0.59	*0.45*
	DLCE	0.89	0.68	0.64	0.95	0.92	*0.77*	0.87	0.68	0.64	0.95	0.91	*0.77*	0.84	nan	0.43	0.95	0.89	*0.59*
	WDLCE	0.89	0.68	0.65	0.96	0.92	*0.78*	0.88	0.68	0.64	0.96	0.92	*0.77*	0.84	nan	0.45	0.94	0.88	*0.61*
	DLWCE	0.81	0.60	0.75	0.97	0.88	*0.84*	0.79	0.59	0.76	0.96	0.87	***0.85***	0.76	0.39	0.57	0.91	0.83	*0.70*
	WDLWCE	0.79	0.59	0.75	0.96	0.87	*0.84*	0.77	0.58	0.76	0.96	0.85	***0.85***	0.75	nan	0.55	0.89	0.81	*0.68*
	DLNCE	0.89	0.67	0.64	0.92	0.90	*0.76*	0.87	0.67	0.64	0.92	0.90	*0.75*	0.83	nan	0.41	0.96	0.89	*0.58*
	DLACE	0.89	0.68	0.64	0.93	0.91	*0.76*	0.87	0.67	0.64	0.93	0.90	*0.76*	0.84	nan	0.42	0.96	0.89	*0.59*
	WDLNCE	0.89	0.68	0.65	0.93	0.91	*0.76*	0.87	0.68	0.64	0.92	0.90	*0.76*	0.83	nan	0.43	0.96	0.89	*0.60*
	WDLACE	0.89	0.68	0.65	0.94	0.91	*0.77*	0.87	0.68	0.64	0.93	0.90	*0.76*	0.84	nan	0.44	0.95	0.89	*0.60*
	DLNWCE	0.68	0.51	0.76	0.93	0.78	*0.83*	0.64	0.50	0.77	0.87	0.74	*0.82*	0.70	0.37	0.58	0.90	0.79	*0.70*
	DLAWCE	0.67	0.51	0.76	0.89	0.77	*0.82*	0.64	0.50	0.77	0.87	0.74	*0.82*	0.69	0.37	0.58	0.91	0.79	***0.71***
	WDLNWCE	0.69	0.52	0.76	0.96	0.80	***0.85***	0.64	0.50	0.77	0.90	0.75	*0.83*	0.70	0.37	0.58	0.88	0.78	*0.70*
	WDLAWCE	0.68	0.52	0.76	0.93	0.79	*0.84*	0.65	0.50	0.77	0.90	0.75	*0.83*	0.69	0.37	0.58	0.89	0.78	*0.70*
DeepLabV3	pretrained	0.96	0.89	0.91	0.95	0.96	*0.93*	0.93	0.86	0.88	0.96	0.95	*0.92*	-	-	-	-	-	-
	WCE	0.92	0.80	0.83	0.89	0.90	*0.86*	0.91	0.80	0.83	0.91	0.91	*0.87*	0.85	nan	0.47	0.89	0.87	*0.61*
	WDL	0.88	nan	0.62	0.89	0.89	*0.73*	0.90	nan	0.81	0.85	0.87	*0.83*	0.81	nan	0.23	0.80	0.80	*0.36*
	FL	0.90	0.77	0.84	0.71	0.80	*0.77*	0.90	0.79	0.83	0.65	0.75	*0.73*	0.84	nan	0.47	0.71	0.77	*0.56*
	CEMP	0.92	0.79	0.82	0.69	0.79	*0.75*	0.90	0.79	0.81	0.54	0.68	*0.65*	0.84	nan	0.47	0.48	0.61	*0.47*
	DLCE	0.91	0.79	0.83	0.89	0.90	*0.86*	0.91	0.80	0.84	0.91	0.91	*0.87*	0.84	nan	0.46	0.88	0.86	*0.61*
	WDLCE	0.92	0.80	0.83	0.88	0.90	*0.86*	0.91	0.81	0.84	0.90	0.90	*0.87*	0.84	nan	0.48	0.86	0.85	*0.62*
	DLWCE	0.86	0.71	0.89	0.82	0.84	*0.85*	0.84	0.72	0.90	0.81	0.82	*0.85*	0.78	0.43	0.61	0.93	0.85	***0.74***
	WDLWCE	0.86	0.72	0.88	0.85	0.85	***0.87***	0.84	0.73	0.90	0.83	0.83	*0.86*	0.77	nan	0.57	0.93	0.85	*0.71*
	DLNCE	0.91	0.77	0.84	0.90	0.90	***0.87***	0.90	0.80	0.85	0.90	0.90	*0.87*	0.84	nan	0.43	0.91	0.87	*0.58*
	DLACE	0.91	0.79	0.83	0.90	0.90	*0.86*	0.91	0.80	0.84	0.91	0.91	*0.87*	0.84	nan	0.46	0.90	0.87	*0.61*
	WDLNCE	0.92	0.79	0.83	0.91	0.91	***0.87***	0.90	0.80	0.84	0.92	0.91	***0.88***	0.84	nan	0.46	0.89	0.87	*0.61*
	WDLACE	0.92	0.79	0.83	0.89	0.90	*0.86*	0.91	0.80	0.84	0.92	0.91	***0.88***	0.84	nan	0.47	0.88	0.86	*0.61*
	DLNWCE	0.74	0.60	0.90	0.72	0.73	*0.80*	0.74	0.63	0.91	0.65	0.69	*0.76*	0.73	0.39	0.61	0.93	0.82	***0.74***
	DLAWCE	0.76	0.62	0.90	0.68	0.72	*0.78*	0.74	0.63	0.91	0.67	0.70	*0.77*	0.73	0.40	0.61	0.94	0.82	***0.74***
	WDLNWCE	0.77	0.63	0.90	0.79	0.78	*0.84*	0.75	0.64	0.91	0.70	0.72	*0.79*	0.73	0.40	0.61	0.93	0.82	***0.74***
	WDLAWCE	0.76	0.63	0.9	0.72	0.74	*0.80*	0.75	0.64	0.91	0.71	0.73	*0.80*	0.73	0.41	0.6	0.93	0.82	*0.73*

4.5. Discussion

In this chapter we introduced loss scheduling as an approach to train models in the face of class imbalance. We demonstrated, that a model can be induced to converge on a different optimum, than if only trained on a pure loss and the scheduling fine-tunes properties of the resulting model compared to using a static compound loss. Compound losses in general seem to be capable of combining beneficial features of different losses, but if the properties of a single pure loss are required, loss scheduling still enables this. Surprisingly, the losses specifically designed for class imbalance did not necessarily produce the best performing models, even disregarding their calibration. Although, the inspiration here was the adversely shaped loss landscape due to class imbalance, the approach is generalizable to other problem settings and other losses with other schedules. In this regard especially the impact on DeepLabV3 trained on Cityscapes and the relative performance of the pre-trained models is relevant. The latter implies, that class imbalance is just a facet of data sparsity. Training on a larger dataset leads to better performance. Pre-training on a related dataset can transfer a lot of the performance to the downstream problem. For a relatively small dataset, as in the former, a brute force approach can not be used to train a more complex model, but a more elaborate training process, like loss scheduling could be used instead.

Very long training processes, like when grokking [160] would occur, likely necessitate custom schedules. Also, while the schedule choice only had limited impact on model performance and training process, in other settings this may not hold. More complex models than feed forward networks or convolutional models, e.g. transformers or geometric models, might also entail a more complex loss landscape. So far, we have considered, but not tested the combination of more than two loss components. One of the downsides to loss scheduling is the introduction of additional external parameters, which in turn increases the combinatorial space to be optimized in hyperparameter searches.

There is some room for further investigation of loss landscape deformation through scheduling. Even though schedules so far have had marginal impact on final model performance, a more complex landscape might exhibit more interesting behavior, e.g., in terms of convergence iterations, and merit different schedules. In the context of chapter 3, the concept could be extended to a task scheduling: oftentimes dialing in the task difficulty during self-supervised pre-training is non-trivial. Measuring the impact on the final model then requires full training of the entire model complex. Some models have employed ideas like semi-hard example mining to adapt the training signal to the models current capabilities. If this approach is not available due to lack of feedback, scheduling may be an alternative approach.

5. Propulate: Massively Parallel Population Based Optimization

In this chapter, we present our work on an HPC-adapted hyperparameter and neural architecture search platform. It is inspired by evolutionary optimization and similar population based optimizers. The central idea is a softened notion of the generation enabling lazy synchronization between parallel workers. When one worker finishes evaluating a candidate solution, instead of waiting for all other workers to finish with their current candidate, it uses only the new information that has been discovered up to this point to generate the next candidate immediately. This approach reduces the idle times when the candidates take different amounts of time to evaluate, as is common in neural architecture search. The work presented in this chapter has been published previously at the International Supercomputing Conference [3].

5.1. Introduction

Hyperparameter optimization (HPO) is the process of choosing not learnable parameters of a machine learning model. For neural networks this encompasses model architecture like number or width of layers, but also training parameters like learning rate or choice of optimizer. HPO, in particular in the context of neural architecture search (NAS), has been of increasing relevance [161] as the use of data-driven models and the resource footprint of those models increases [20]. Since evaluating a proposed set of hyperparameters usually involves at least a partial training, a more expensive model implies a more expensive hyperparameter search multiplied by a large factor. Therefore, smart search algorithms that minimuze the required number of evaluations are needed. Manual tuning or traditional approaches like grid search are not only inefficient in terms of time, but also energy and by extension carbon footprint [162].

As the name implies hyperparameter optimization is a hierarchical process. In an outer loop, a new set of candidate hyperparameters is proposed, which is evaluated by fitting the actual parameters or weights of the model in an inner loop. Figure 5.1 shows a sketch of the hyperparameter optimization process. To evaluate the performance of the hyperparameters, the model performance is measured on a held-out validation dataset. This validation dataset should be sufficiently diverse from the training data to prevent ground-truth bleeding and overestimating the performance. Since the HPO might overfit on the validation data, the final performance of the best model found is then estimated on a third test dataset not seen during the optimization. Evaluating the hyperparameters in the inner loop requires at least a partial training of the model, which makes this process computationally expensive. It is thus necessary to utilize sufficient computing resources to optimize the hyperparameters in a reasonable amount of time. Using these

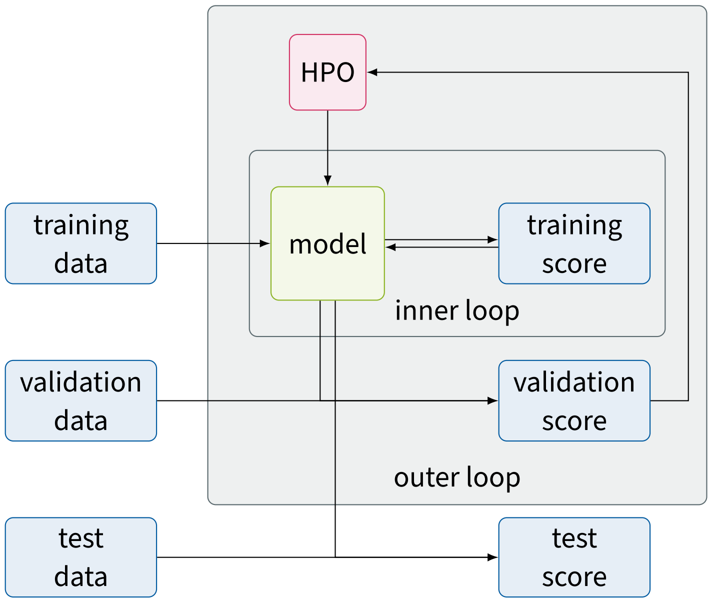


Figure 5.1.: Schematic illustration of the hyperparameter optimization process

resources efficiently requires scalable and high-performance computing adapted algorithms. Chapter 2 gives more background on the relevant concepts of HPC and HPO. The key to leveraging HPC resources is parallelization. As a general first approach we unroll the outer loop and train and evaluate many candidates in parallel, since they are independent of each other. Facilitating this parallelization is the primary task we are interested throughout this chapter.

Besides scalability, a general hyperparameter optimizer needs to feature certain properties owing to their specific usage:

- Black box or gray box optimization: The exact nature of the function to be optimized is not known in advance.
- Discrete parameter spaces: Possible types of hyperparameters go beyond continuous search spaces and include integer, ordinal, and categorical parameters.
- Gradient-free: Because of the properties of the search spaces commonly encountered, the optimization algorithm can not rely on gradients to navigate the search space.

Naive approaches like grid search and random search fulfill all these criteria. However, there are more intelligent, state-dependent search algorithms that require fewer evaluations of the black box function to meaningfully sample the search space. Chapter 2 contains an overview over different methods.

5.2. Related Work

We have examined the theory of relevant optimization algorithms in chapter 2. Here, we will briefly discuss implementations and frameworks as direct competitors to Propulate.

The recent AI boom has triggered heavy use of hyperparameter optimization techniques with Python as the de facto standard programming language. For NAS we thus limit our view to Python-based frameworks. Several surveys give a theoretical overview of hyperparameter optimization in general or neural architecture search specifically [92, 97, 98, 99, 100, 101, 102, 103, 104, 93, 94, 95, 91, 96].

Several frameworks, Optuna [163], Hyperopt [164], SMAC3 [165, 166], Spearmint [167], GPyOpt [168], and MOE [169], DeepHyper [170, 171] implement some form of Bayesian optimization as a surrogate model, e.g. tree-structured Parzen estimators or Gaussian processes (cf. chapter 2).

SMAC3 (**S**equential **M**odel-based **A**lgorithm **C**onfiguration) combines a random-forest-based Bayesian approach with an aggressive racing mechanism [165]. Its parallel variant, pSMAC, uses multiple collaborating SMAC3 runs which share their evaluations through the file system. Spearmint, GPyOpt, and MOE are Gaussian-process-based Bayesian optimizers. Spearmint enables distributed HPO via Sun Grid Engine and MongoDB. GPyOpt is integrated into the Sherpa package [172], which provides implementations of recent hyperparameter optimizers and the software infrastructure to run them in parallel via a grid engine and a database server. MOE (**M**etric **O**ptimization **E**ngine) uses a one-step Bayes-optimal algorithm to maximize the multi-points expected improvement in a parallel setting [169]. Using a REST-based client-server model, it enables multi-level parallelism by distributing each evaluation and running multiple evaluations at a time. Nevergrad [173] and Autotune [174] provide gradient-free and evolutionary optimizers, including Bayesian, particle swarm, and one-shot optimization. In Nevergrad, parallel evaluations use several workers via an executor from Python's concurrent module. Autotune enables concurrent global and local searches, cross-method sharing of evaluations, method hybridization, and multi-level parallelism. Open Source Vizier [175] is a Python interface for Google's HPO service Vizier. It implements Gaussian process bandits [176] and enables dynamic optimizer switching. A central database server does the algorithmic proposal work while clients perform evaluations and communicate with the server via remote procedure calls. Katib [177] is a cloud-native AutoML project based on the Kubernetes container orchestration system. It integrates with Optuna and Hyperopt. Tune [178] is built on the Ray distributed computing platform. It interfaces with Optuna, Hyperopt, and Nevergrad and leverages multi-level parallelism. Except for DeepHyper, which can use MPI for communication and parallelism, these tools use a database like MongoDB or SQL, i.e., the file system. PyHopper [179] is a scheduled Markov chain Monte Carlos sampler, which is intended particularly for high dimensional parameter spaces. However, so far, it is only parallelized for a single node with multiple GPUs.

Another family of solutions are bio-inspired approaches like particle swarms (e.g., FLAPS [180]) or evolutionary optimizers like DEAP [181] and MENNDL [182]. FLAPS (**F**lexible se**L**f-**A**dapting **P**article **S**warm) is an PSO, which dynamically scales the weights of different contributions to a compound loss function during the optimization. It was primarily developed to find hyperparameters for MD simulations. DEAP (**D**istributed **E**volutionary **A**lgorithms in **P**ython) [181] implements general evolutionary algorithms, evolution strategies, multi-objective optimization, and co-evolution of multi-populations. It enables parallelization via Python's multiprocessing or SCOOP module. EvoTorch [183] is built on PyTorch and implements distribution- and population-based algorithms. Using a Ray cluster, which uses a central head node for coordination and several worker nodes for evaluation, it can scale over multiple CPUs, GPUs, and computers. MENNDL (**M**ulti-node **E**volutionary **N**eural **N**etworks for **D**eep **L**earning) [182] is a closed-source MPI-parallelized hyperparameter optimizer for automated network selection. A coordinator node handles the evolutionary operations while the remaining worker nodes conduct the evaluations. However, global synchronization hinders optimal resource utilization [182].

5.3. Method

The problem of parallel hyperparameter search can be separated into two aspects. Firstly, candidates have to be generated and evaluated. Secondly, the computational load of these processes has to distributed and the relevant information has to be communicated between all participating resources.

5.3.1. Lazy Synchronization for Parallel Search

The basis for our neural architecture search approach is evolutionary optimization. In synchronized, parallel evolutionary optimization, if the cost for a single evaluation of the objective function varies depending on the hyperparameter candidate, resources are wasted as workers with a cheaper candidate have to wait for the evaluation of the more expensive ones to finish (cf. fig. 5.2). To alleviate this bottleneck inherent to synchronized parallel evolutionary algorithms, our massively parallel population based optimizer Propulate (***prop****agate* and ***pulate***) implements a lazy synchronization mechanism designed for large-scale HPC systems. Unlike other conventional population based algorithms, Propulate softens the notion of a generation. Instead, Propulate uses the entirety of the thus far evaluated solution candidates as the basis for breeding of new candidates. This enables *asynchronous* evaluation, variation, propagation, and migration of individuals.

The basic propagation mechanism is still that of Darwinian evolution, as in other evolutionary optimization algorithms. Well-performing individuals assumed to carry beneficial traits are selected from the overall population. Their traits are then recombined and mutated to breed new suggested candidates (see chapter 2). These different propagation building blocks can be assembled for the specific task at hand. Each one is represented by its own *propagator*, which receives a set of individuals and generates a different set of individuals. The construction of an evolutionary propagator for example, requires a selection, crossover, and mutation propagator. The evolutionary propagator applies these propagators with specified probabilities to generate a single new candidate. For example, to have a more aggressive optimization process, the selection operator selects only the best individuals from the already evaluated population, whereas a selection propagator closer to the traditional evolutionary optimization would select the most recently evaluated ones. Varying the mutation propagator might switch between mutating a single trait uniformly in the given limits and mutating multiple traits within a Gaussian distribution centered on the trait value of the parent individual.

Figure 5.3 illustrates the Propulate process on a higher level outside of the propagation process. Examining the example of the individual bred for generation 3 by the blue worker,

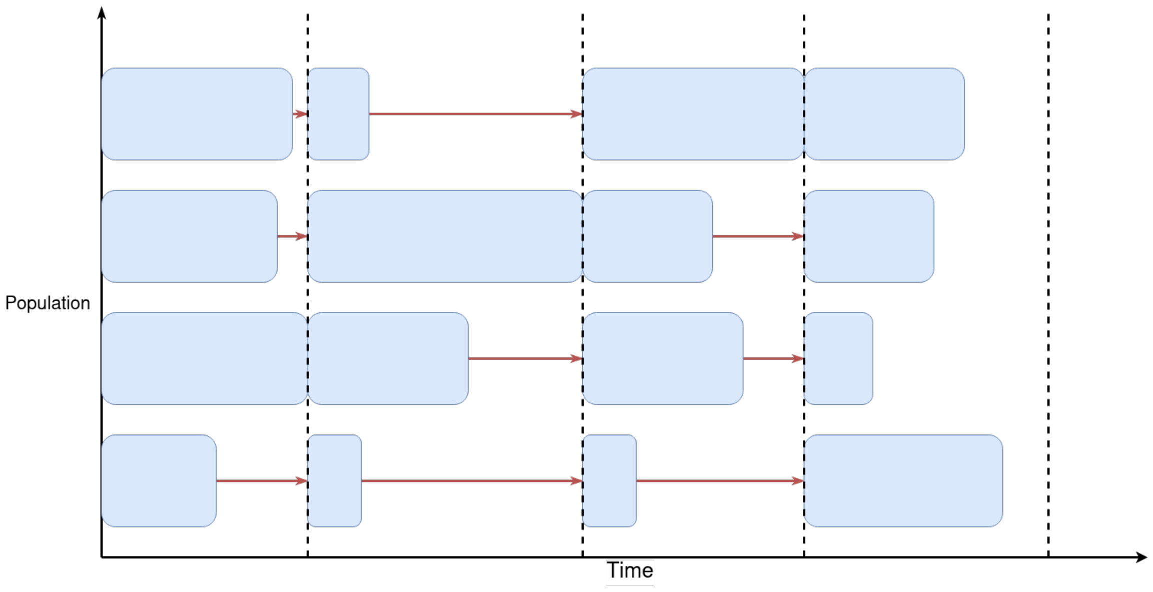


Figure 5.2.: Illustration of the evaluation progress of a naively parallelized evolutionary optimizer with four workers along the y-axis. Time progresses along the x-axis. The blue boxes represent the evaluation of a single candidate. The red arrows show idle time caused by the synchronization barriers at the dashed lines after each generation.

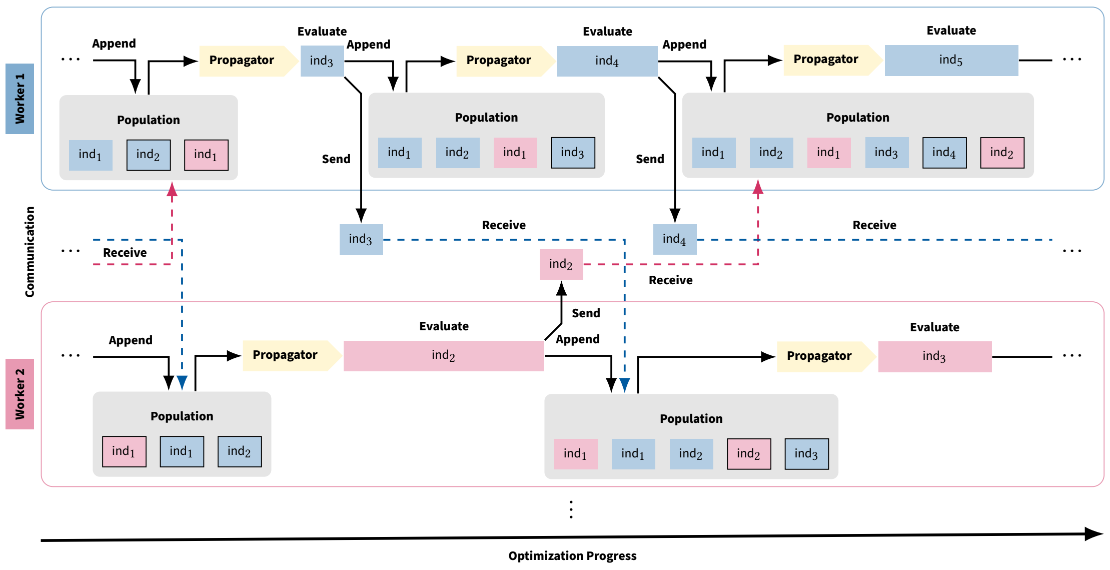


Figure 5.3.: **Propulate Asynchronous propagation.** Interaction of two workers during the search process. Individuals bred by worker 1 and 2 are shown in blue and red, respectively. Their origins are given by a generation sub- and an island (cf. section 5.3.2) superscript. Populations are depicted as round grey boxes, where most recent individuals have black outlines. Varying evaluation times are represented by sharp boxes of different widths.

ind^{i1}_{g3}: the worker first applies the propagator to generate the new candidate parameter set for this generation. It then evaluates the new individual by calling the user-supplied black-box function with the generated parameters. It then asynchronously sends the individual with its associated loss value to all other workers. Then, it checks whether the other workers have sent any individuals whose evaluation has completed in the meantime. In this example none have, so it integrates only ind^{i1}_{g3} in its breeding population and generates the next candidate. Once worker 2 has finished with its own currently ongoing evaluation of its own second generation and sent the result to the other workers, it receives the evaluated individual from worker 1. Worker 2 then updates its own population with those two newly evaluated individuals and continues with its own breeding step. All of these communication operations are realized as asynchronous MPI calls. Results are checkpointed and written to disk for later use, but during the run, nothing is read from disk inside the Propulate optimization itself.

The initial population that subsequent generations are bred from is generated randomly, and during the course of the search, there can be a chance set to randomly generate a fresh individual completely unrelated to the rest of the population instead of the crossover and mutation operations.

5.3.2. Splitting the Population

Population based optimizers sometimes separate the global population into several sub-populations, each on their own evolutionary island [184]. Individuals can be exchanged between islands intermittently. This approach has two advantages: firstly, if one population prematurely converges on a local optimum, there is a chance that other islands continue the exploration or even give the stuck sub-population the stimulus to restart its progress. Secondly, since all individuals on an island are exchanged between all workers of that island, splitting the population into several islands can reduce the communication costs and reserve more computational time for candidate evaluation. In addition to the basic migration mechanism already described, there is another variation called *pollination*. Whereas in true migration, emigrants are removed from their populations of origin for further breeding, pollination models retain that individual and only send a copy to the receiving island. Figures 5.4 and 5.5 illustrate the migration and pollination process, respectively. Comparing the mechanism for both schemes on worker N on island 1, the initial process is similar. After evaluation and (lazy) intra-island synchronization, it sends the chosen migrants to all workers of the target island, here island 2, who receive them asynchronously. It also informs all other workers on island 1 which individuals have emigrated and are no longer available for breeding. Afterwards, worker N receives migrants sent since the last synchronization from worker 1 on island 2 and integrates them into the breeding population. Then, the worker performs population bookkeeping, which depends on whether it is in pollination or true migration mode. After these migration-related intra-island population updates, worker N breeds the individual for the next generation with its normal propagator, now including information from island 2. After finishing this evaluation, worker N randomly decides with the specified migration probability not to emigrate any individual and skips to the immigration step. The main difference on a technical level between pollination and true migration is that pollination creates copies and thus inflates the total population over time. To alleviate this, the pollination includes an immigration operator that can remove individuals to keep the population size at the same level. One worker on the target island is responsible for performing this culling and communicating to the other workers on that island which individuals are no longer active in the breeding population. In return, the communication on the source island on which individuals are no longer available due to emigration is not required since only copies were sent and the original is still active on the source island. Again, there are no explicit synchronization barriers for inter-island communication. Instead, there is a chance after each evaluation that several individuals will be selected for migration and sent to be integrated into the population of the target islands. Figure 5.6 shows the hierarchical communication setup with several workers and islands. Typically, better performing individuals are selected for migration. With worse-performing islands receiving candidates from better-performing ones, islands communicate evolutionary information competitively, thus increasing diversity among the sub-populations compared to panmictic models [185]. Independent from the propagation mechanism of the evolutionary optimization, the following parameters determine the behavior and performance of the Propulate search process:

- **Island number and subpopulation sizes**
- **Migration (pollination) probability**
- **Number of migrants (pollinators):** How many individuals migrate from the source population at a time.
- **Migration (pollination) topology:** Directed graph of migration (pollination) paths between islands.
- **Emigration policy:** How to select emigrants (e.g., random or best) and whether to remove them from the source population (actual migration) or not (pollination).
- **Immigration policy:** How to insert immigrants into the target population, i.e., either add them (migration) or replace existing individuals (pollination, e.g., random or worst).

5.3.3. Alternative Algorithms

The HPOs considered here, which act as blackbox optimization functions, suggest new hyperparameter combinations only based on the objective function value of previously suggested hyperparameters. The sampling algorithm that suggests new candidate solutions for evaluation resides in a propagator on each worker rank. It takes the locally visible population with each individual identified by its worker of origin and its generation as input and produces a new vector in hyperparameter space:

$$P_{w\prime}(\{(\boldsymbol{x}^{wg}, y_{wg})\}) \rightarrow \boldsymbol{x}^{w\prime g\prime}. \tag{5.1}$$

with the propagator $P_{w\prime}$ on worker $w\prime$, the set of tuples of previous suggestions $\boldsymbol{x}^{wg}$ evaluated by worker w at generation g with associated objective function value y_{wg}. The

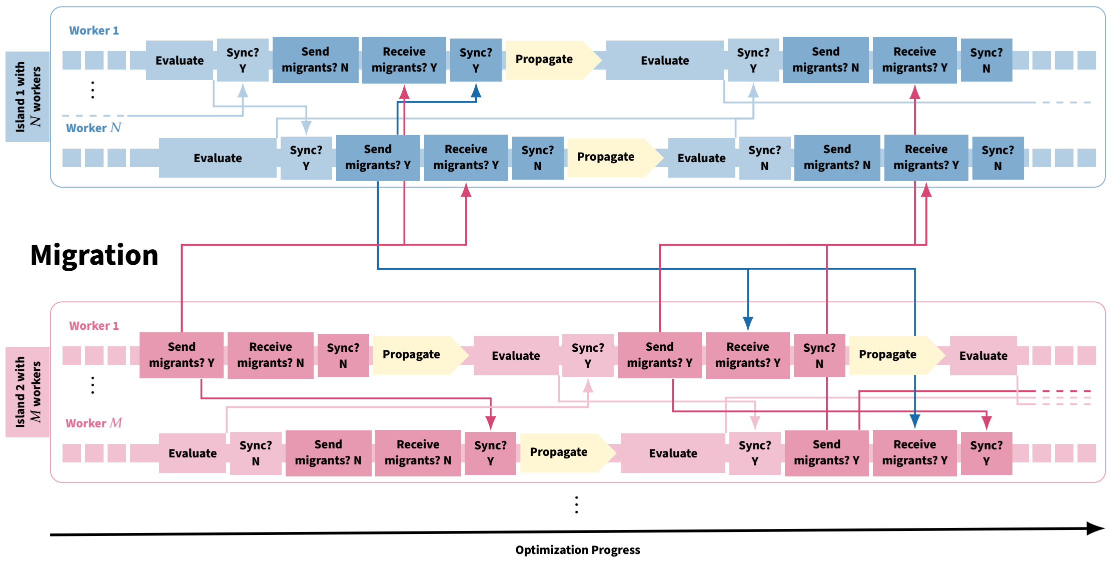


Figure 5.4.: **Asynchronous migration.** Consider two islands with N (blue) and M (red) workers, respectively. Intra-island communication and operations are shown in light colors and inter-island communication in full colors. During Sync blocks messages are received, that signal, which individuals have been emigrated by another worker.

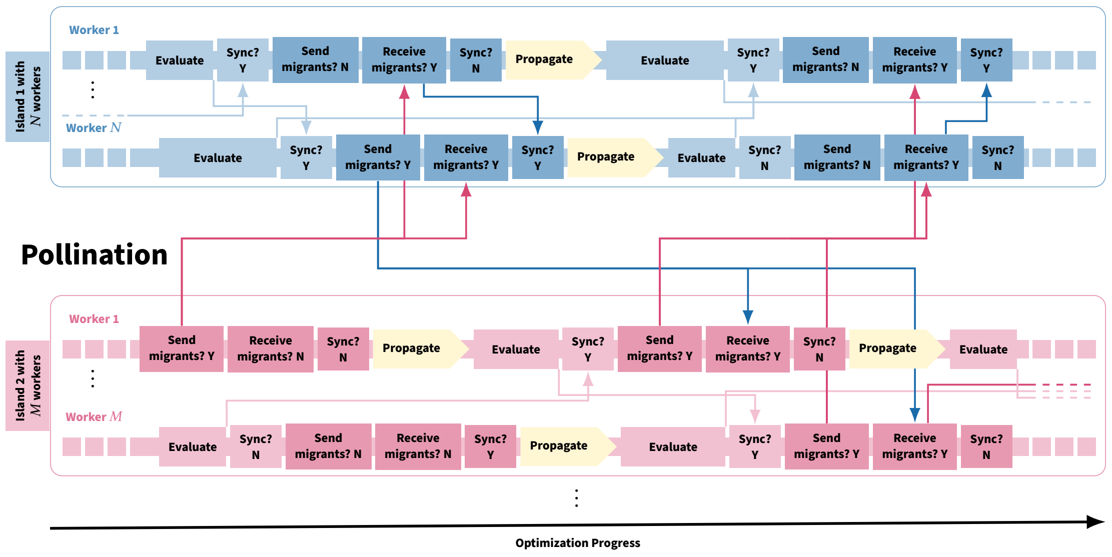


Figure 5.5.: **Asynchronous pollination.** Consider two islands with N (blue) and M (red) workers, respectively. Intra-island communication and operations are shown in light colors and inter-island communication in full colors. During Sync blocks messages are received, that signal, which individuals have been replaced by incoming pollinators by another worker.

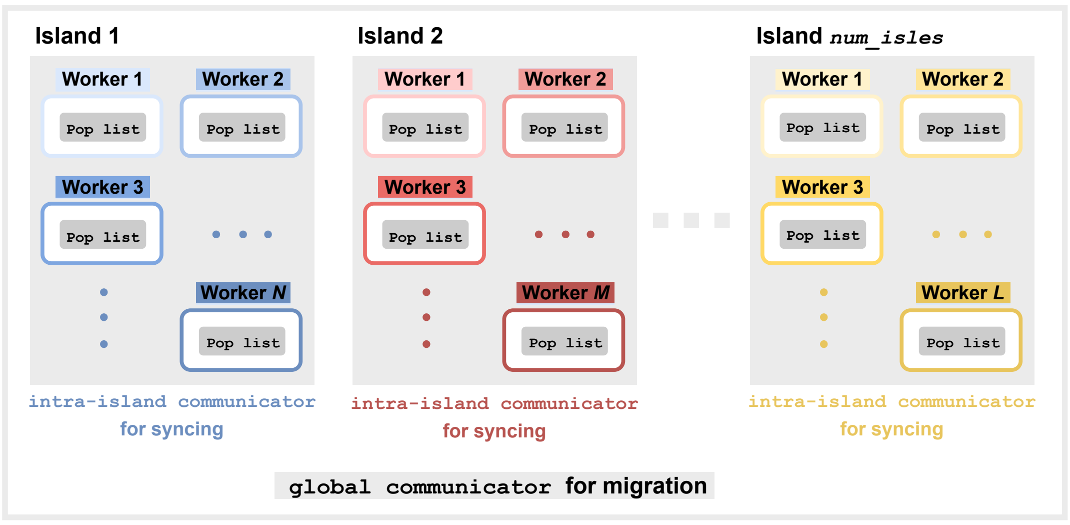


Figure 5.6.: Propulate communication setup. Workers on the same island share an island communicator. All workers share a global communicator, which is used for migration and pollination.

new candidate $\boldsymbol{x}^{w'g'}$ of the current generation $g\prime$ Note that any g in the input set does not necessarily have to be smaller than $g\prime$, since a different worker may already be several generations ahead and have provided its results. Evolutionary optimizers are only one example of such an algorithm that can be adapted to the Propulate platform. Evolutionary and depending on the surrogate model Bayesian optimizers can directly process discrete parameters. Since these other algorithms considered here can not process categorical or discrete search parameter spaces, we first project these parameters into a continuous space. To obtain an integer parameter, the internal continuous representation is rounded. To represent categorical variables without imposing an order, they are one-hot encoded and one dimension is added to the search space for each category. For these variables, the candidate parameter is retrieved from the internal continuous representation by finding the class vector closest to the candidate position, which amounts to taking the argmax as long as the algorithm is constrained to the interval of $[0, 1]$, e.g.,

$$\boldsymbol{x} = \left\{\begin{array}{cc} 0.0001 & \text{learning rate} \\ 4.920 & \text{number of layers} \\ 0.163 & \text{activation: ReLU} \\ 0.987 & \text{activation: ELU} \\ 0.618 & \text{activation: sigmoid} \end{array}\right\} \rightarrow \left\{\begin{array}{c} 0.0001 \\ 5 \\ \text{ELU} \end{array}\right\}. \tag{5.2}$$

This approach arbitrarily imposes that the distance between categories is the same, which might cause a bias. Depending on the algorithm different scales for different dimensions in parameter space can be rescaled to avoid this. For example, in the space of activation functions, one might expect that ReLU should be closer to ELU than to a sigmoid in an embedding of activation functions, which is not represented in a one-hot encoding.

5.3.3.1. Particle Swarm Optimization

PSO is an optimization algorithm inspired by the food-searching behavior of animal swarms (see section 2.3). Several flavors of PSO were integrated into Propulate by Paul Zanner within the scope of his bachelor thesis [186]. Each rank represents one particle in the swarm. Within the swarm, only the best individual discovered so far has to be distributed to other members to represent the social aspect of the optimization. The cognitive contribution (cf. chapter 2) is given by the previous candidates evaluated by the same worker. Currently, the swarm communicates all evaluation results, even ones that have no prospect of informing the trajectory of other particles, because it is worse than the best global results. The core of the swarm optimization is provided in the form of propagators similar to the evolutionary operators of the original Propulate. The only additional ingredient beyond that is an additional velocity field for individuals that contributes to the position in the search space of the candidate of the next generation.

5.3.3.2. Covariance Matrix Adaptation Evolution Strategy

CMA-ES adapts the distribution from which new candidates are sampled over the course of the optimization to better trade off exploration and exploitation. It is provided as an alternative to the default Bayesian optimizer in Optuna. An asynchronous adaption of CMA-ES was integrated into Propulate by Jonathan Roth within the scope of his bachelor thesis [187].

To adapt CMA-ES to the lazy synchronization scheme of Propulate, the algorithm has to be adapted such that in each iteration only a single new candidate is proposed. Since the sampling still has a stochastic component and each worker has a separate random number generator, no particular care has to be taken to avoid redundancy within the CMA-ES sampling. Also, each individual is likely generated from its own unique parent population since it is, at least initially, the only one containing the most recently evaluated individual from that worker.

5.3.3.3. Nelder-Mead Method

A simplex is the generalization of a triangle in two dimensions to an arbitrary number of dimensions. The downhill simplex or Nelder-Mead [188] method is a set of rules on how to choose the next candidate solution in an n dimensional optimization problem given $n + 1$ solutions forming the vertices of a simplex in the search space. The size of the population is then not entirely arbitrary anymore but rather determined by the dimension of the parameter space to be searched. Simplex optimization uses the following policies: reflection, expansion, contraction, and shrinking, with certain marked points:

$$\boldsymbol{x}_o = \underset{i \in \text{simplex}}{\text{centroid}}\, \boldsymbol{x}_i \tag{5.3}$$

$$\boldsymbol{x}_{n+1} = \max_{i \in simplex} (\boldsymbol{x}_i) \tag{5.4}$$

$$\boldsymbol{x}_n = \max_{i < n+1} (\boldsymbol{x}_i) \tag{5.5}$$

$$\boldsymbol{x}_r = \boldsymbol{x}_o + \alpha(\boldsymbol{x}_o - \max_{i \in \text{simplex}} (\boldsymbol{x}_i)) \tag{5.6}$$

$$\boldsymbol{x}_e = \boldsymbol{x}_o + \alpha(\boldsymbol{x}_r - \boldsymbol{x}_c) \tag{5.7}$$

$$\boldsymbol{x}_c = \boldsymbol{x}_o + \rho \begin{cases} \boldsymbol{x}_r - \boldsymbol{x}_o & f(\boldsymbol{x}_r) < f(\boldsymbol{x}_{n+1}) \\ \boldsymbol{x}_{n+1} - \boldsymbol{x}_o & f(\boldsymbol{x}_r) \geq f(\boldsymbol{x}_{n+1}) \end{cases} \tag{5.8}$$

With the reflection $\alpha > 0$, expansion $\gamma > 1$, contraction $\rho \leq 0.5$, and shrinking $\sigma < 1$ coefficients. During a single iteration of sequential Nelder-Mead, one or more of the reflected point $\boldsymbol{x}_r$, the expanded point $\boldsymbol{x}_e$, or a contracted point $\boldsymbol{x}_c$ might have to be computed in order to update the simplex for the next iteration. For the shrinking operation, the entire simplex is scaled to a smaller one of the same shape fixed to the best known vertex so far. As a consequence, potentially several evaluations have to be performed for a single iteration. This is not directly compatible with Propulate, which assumes only a single black box function evaluation at each generation. To adapt the algorithm, we change the how the operator is selected at each generation.

- First the current simplex is constructed from the best $n + 1$ solutions so far.
- If the last generation is the best solution so far: expand.
- If the last generation is the worst in the simplex: outer contract.
- If the last generation is not in the simplex: cycle through reflect, outer contract, inner contract, and shrink.
- Else: reflect

For initialization, the first $n+1$ points are sampled from a Gaussian around a point identical to all workers on the same island.

5.3.4. Implementation

Implementation-wise, to ensure interoperability with existing data science and ML workflows, which are expected to be the primary use-case, we maintain a Python implementation[1]. All communication between workers occurs directly over MPI. In most applications, evaluating the objective function represents the largest contribution to the total resource consumption. Performance-relevant paths inside the objective function evaluation are

[1] https://github.com/Helmholtz-AI-Energy/propulate

expected to be implemented and optimized in CUDA and C/C++ or Fortran. With the aforementioned workflows, this is typically already the case. Since we expect most of the time to be spent on evaluation, optimization here has great impact on the duration of the overall search.

5.3.5. Computing Environment

We ran the experiments on the distributed-memory, parallel hybrid supercomputer *Hochleistungsrechner Karlsruhe* (HoreKa[2]) at the Scientific Computing Center, Karlsruhe Institute of Technology. Each of its 769 compute nodes is equipped with two 38-core Intel Xeon Platinum 8368 processors at 2.4 GHz base and 3.4 GHz maximum turbo frequency, 256 GB (standard) or 512 GB (high-memory and accelerator) local memory, a local 960 GB NVMe SSD disk, and two network adapters. 167 of the nodes are accelerator nodes, each equipped with four NVIDIA A100-40 GPUs with 40 GB memory connected via NVLink. Inter-node communication uses a low-latency, non-blocking NVIDIA Mellanox InfiniBand 4X HDR interconnect with 200 Gbit/s per port. A Lenovo Xclarity controller measures full node energy consumption, excluding file systems, networking, and cooling. The operating system is Red Hat Enterprise Linux 8.2.

5.3.6. Performance Evaluation

We evaluate the performance of our proposed approach considering three different aspects: optimization performance, computational overhead, and relevance to neural architecture search. The obvious one is optimization performance, as in how good are the best solutions found by Propulate. Since optimization in general is a large and well-researched field of research, and evolutionary optimization is not expected to be the optimal choice for all optimization problems our expectation here is to at best match the baseline. Next, there is the computational performance and, specifically, the overhead borne by the communication setup and the propagation and sampling routines of Propulate. This is where we expect a drastic improvement over the baseline, since efficient parallelization was our primary consideration in the design of Propulate.

To examine the computational overhead and general optimization behavior, we first use often used mathematical benchmark functions (see table 5.1) rather than a more complex problem derived from neural architecture search. Evaluating the benchmark function is virtually free, especially compared to training a neural network. These test functions are primarily intended to test communication overhead under extreme circumstances to ensure correct implementation of and compare the different optimization algorithms. Additionally, this allows characterizing the optimization algorithms in terms of their suitability for different optimization problems, helping users choose the most suitable approach for their problem. Noise, roughness, or problem space dimension informs the choice of Propulate's parameters.

Finally, the most important aspect is neural architecture search. Compared to the mathematical benchmark functions evaluation for this task is more expensive, which

[2] https://www.scc.kit.edu/en/services/horeka.php

masks the communication overhead at least partially. If we assume, that the baseline is more intelligent and can find a better solution if given the same number of evaluations, can Propulate still outperform it by searching faster? An optimal solution to the NAS problem is not known, so only relative performance score is considered.

The selection of benchmark functions poses a specific challenge for each instance:

- **Sphere** is smooth, unimodal, strongly convex, symmetric, and thus simple.
- **Rosenbrock** has a narrow minimum inside a parabola-shaped valley.
- **Step** represents the problem of flat surfaces. Plateaus pose obstacles to optimizers as they lack information about which direction is favorable.
- **Quartic** is a unimodal function padded with Gaussian noise. As it never returns the same value on the same point, algorithms that do not perform well on this test function will do poorly on noisy data.
- **Rastrigin** is non-linear and highly multimodal. Its surface is determined by two external variables, controlling the modulation's amplitude and frequency. The local minima are located at a rectangular grid with size 1. Their functional values increase with the distance to the global minimum.
- **Griewank**'s product creates sub-populations strongly codependent to parallel GAs, while the summation produces a parabola. Its local optima lie above parabola level but decrease with increasing dimensions, i.e., the larger the search range, the flatter the function.
- **Schwefel** has a second-best minimum far away from the global optimum.
- **Lunacek's bi-sphere**'s [189] landscape structure is the minimum of two quadratic functions, each creating a single funnel in the search space. The spheres are placed along the positive search-space diagonal, with the optimal and sub-optimal sphere in the middle of the positive and negative quadrant, respectively. Their distance and the barrier's height increase with dimensionality, creating a globally non-separable underlying surface.
- **Lunacek's bi-Rastrigin [189]** is a double-funnel version of Rastrigin. This function isolates global structure as the main difference impacting problem difficulty on a well-understood test case.

To provide reasonable defaults, we also give results for a grid search over those parameters. Since the intended primary use-case is neural architecture search, we also compare Propulate against `Optuna` optimizing the hyperparameters of a training for a remote sensing application.

Table 5.1.: Benchmark functions

Name	Function	Limits	Global minimum
Sphere	$f_1 = x_1^2 + x_2^2$	± 5.12	$f(0,0) = 0$
Rosenbrock	$f_2 = 100\left(x_1^2 - x_2\right)^2 + (1 - x_1)^2$	± 2.048	$f(1,1) = 0$
Step	$f_3 = \sum_{i=1}^{5} \operatorname{int}(x_i)$	± 5.12	$f(x_i \leq -5) = -25$
Quartic	$f_4 = \sum_{i=1}^{30} \left(ix_i^4 + \mathcal{N}_i(0,1)\right)$	± 1.28	$f(0,\ldots,0) = \sum_i \mathcal{N}_i$
Rastrigin	$f_5 = 200 + \sum_{i=1}^{20} x_i^2 - 10\cos(2\pi x_i)$	± 5.12	$f(0,\ldots,0) = 0$
Griewank	$f_6 = 1 + \frac{1}{4000}\sum_{i=1}^{10} x_i^2 - \prod_{i=1}^{10} \cos\frac{x_i}{\sqrt{i}}$	± 600	$f(0,\ldots,0) = 0$
Schwefel	$f_7 = 10V - \sum_{i=1}^{10} x_i \sin\sqrt{\lvert x_i \rvert}$	± 500	$f\left(x_1^*,\ldots,x_{10}^*\right) = 0,$
	with $V = 418.982887$		$x_i^* = 420.968746$
Bi-sphere	$f_8 = \min\left(\sum_{i=1}^{30}(x_i - \mu_1)^2,\right.$	± 5.12	$f(\mu_1,\ldots,\mu_1) = 0$
	$\left.30 + s\cdot\sum_{i=1}^{30}(x_i - \mu_2)^2\right)$ with		
	$\mu_1 = 2.5, \mu_2 = -\left(s^{-1}\left(\mu_1^2 - 1\right)\right)^{1/2},$		
	$s = 1 - \left(2\sqrt{50} - 8.2\right)^{-1/2}$		
Bi-Rastrigin	$f_9 = f_8 + 10\sum_{i=1}^{30} 1 - \cos 2\pi(x_i - \mu_1)$	± 5.12	$f(\mu_1,\ldots,\mu_1) = 0$

5.3.6.1. Optuna

The aforementioned `Optuna` [163] is an optimization framework providing Bayesian-and CMA-ES-based optimizers (cf. chapter 2). We chose `Optuna` as a baseline since it appears to be the most widely adopted hyperparameter search suite. It is actively developed and maintained, with up-to-date documentation including a growing number of usage examples and practical guidance. Its primary reference has been cited over 3 200 times as of the time of writing. It is advertised as being easy to parallelize and lightweight. The parallelization is realized through a relational database, with the measurements shown for MySQL. The workers retrieve unevaluated candidates from the database and store their results after evaluating them. To conserve resources, `Optuna` also includes pruning mechanisms to abort unpromising trials. In our comparison, we use early stopping for both `Optuna` and Propulate instead of `Optuna`'s built-in pruning mechanism.

5.4. Results

5.4.1. Propulate Parameters

Propulate itself has several configurable parameters. The number of workers, the problem dimension, and the partition and communication setup between islands are algorithm-agnostic. The choice of optimization algorithm could be considered a parameter, and finally, each algorithm requires configuration. For example, the evolutionary optimizer requires a population size (independent of the number of workers), probabilities for mutation, crossover, random initialization, and variance if the mutation uses a Gaussian around the value of the parent rather than a uniform probability distribution. Each component might offer additional degrees of freedom, e.g., crossover might use n-point crossover rather than just drawing from two parents. Table 5.2 shows the search space. The absolute amount of resources is kept constant at two workers, but how they are distributed over islands is varied. The grid search is run over a subset of the previously discussed benchmark functions to avoid giving Propulate an unfair advantage when comparing it to Optuna in the next section. The results are averaged over five runs for the quartic, Rastrigin, and bi-Rastrigin benchmark functions (cf. table 5.1). These functions were chosen for their varying difficulty in optimization and high-dimensional parameter spaces (30, 20, and 30, respectively). Each run uses a different seed for each point in the searched grid.

Number of islands	2	4	8	16	32
Island population size	72	36	18	9	4
Migration (pollination) probability	0.1	0.3	0.5	0.7	0.9
Pollination		True		False	
Crossover probability	0.1	0.325	0.55	0.775	
Point-mutation probability	0.1	0.325	0.55	0.775	
Random-initialization probability	0.1	0.325	0.55	0.775	

Table 5.2.: Grid search parameters. Using 144 workers distributed over two nodes.

For quartic, Propulate found a minimum below 0.01 ± 0.005 for 80.12 % of all points across the five grid searches. This increases to 94.94 % for minima within 0.1 ± 0.05 of the global minimum. In comparison, the tolerances have to be relaxed considerably for the more complex Rastrigin and bi-Rastrigin. While only 18.57 % of all grid points had a function value less than 1.0 ± 0.5 for Rastrigin, only a single point resulted in an average value of less than 10 for bi-Rastrigin. Although the average value of bi-Rastrigin was only less than 10 once, we found the minimum across each of the five searches to be less than 1.0 for 3.31 % of the grid points.

Considering grid points with at least one result smaller than 1.0, 86.61 % used either 16 or 36 islands, while the remainder used eight. As Propulate initializes different islands at different positions in the search space, the chance that one of them is at a very beneficial position increases with the number of islands. The the higher migration probabilities of 0.7 or 0.9 for 61.41 % of these points, leads to a quick proliferation of fit individuals from island to island.

With every best grid point using pollination, we clearly find pollination to be favorable over real migration here. To determine the other hyperparameters, we compute the averages of the results for the top ten grid points across all three functions. The top ten were determined by grouping over the lowest average and standard deviation of the function values, sorting by the averages, and sorting by the standard deviations. This method reduces the chances of a single run simply benefiting from an advantageous starting seed. Average crossover, point-mutation, and random-initialization probabilities are 0.655±0.056, 0.363±0.133, and 0.423±0.135, respectively. The average number of islands was 28.800 ± 6.009, which equates to an island population of 5.00 ± 1.043. The average migration probability was 0.527 ± 0.150. These values provide a reasonable starting point for choosing default hyperparameters for Propulate (see Table 5.3). As the grid searches only considered functions with independent parameters, we assume a relatively high random-initialization probability to be useful due to the benefits of random search [190]. On this account, we chose to reduce the default random-initialization probability to 0.2. As the migration probability might also be lowered artificially by this phenomenon, we set its default to 0.7. The default probabilities for crossover and point-mutation were chosen as 0.7 and 0.4, respectively. The island size was set at four workers. This choice is motivated by technical considerations, as the test system has four accelerators per node, and the number of CPUs per node is a multiple of four.

5.4.2. Optimization Benchmark Functions

To evaluate Propulate in terms of optimization performance and communication overhead in an idealized setting, we ran ten optimizations for each benchmark function described previously for both Propulate and `Optuna`. We use the same amount of compute resources for both algorithms, with the same degree of parallelization and total number of evaluations, 38 912. Table 5.3 shows the parameters used for Propulate. For `Optuna`, the default parameters are used throughout. We use a tree-structured Parzen estimator instead of CMA-ES as sampling algorithm. Since this optimizer can also process categorical parameters natively, this seems a fair comparison.

Number of islands	38
Island population size	4
Pollination probability	0.7
Crossover probability	0.7
Point-mutation probability	0.4
Sigma factor	0.05
Random-initialization probability	0.2
Generations per worker	256
Selection policy	Best
Pollination topology	Fully connected
Number of migrants	1
Emigration policy	Best
Immigration policy	Worst

Table 5.3.: Propulate hyperparameters for benchmark function optimization.

Figure 5.7 shows function value over run time comparing Propulate to `Optuna`. In terms of solution quality, Propulate and `Optuna` are comparable for most functions. For some objective functions, e.g., Schwefel, Rastrigin, and bi-Rastrigin, Propulate even achieves an even better value. The difference is more drastic for walltime. Propulate is consistently faster than `Optuna`, between one and three orders of magnitude. This result is not entirely unexpected: since these benchmark functions are cheap to evaluate, the optimization algorithm itself and communication dominate the wallclock time. An example of an individual run for rastrigin, with the evolution of the objective function's value and distance to the optimum is shown in appendix A.4.

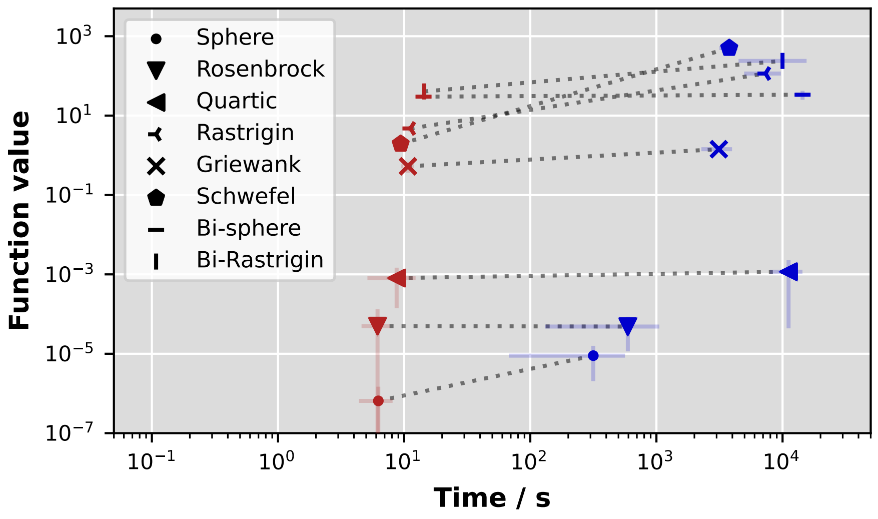


Figure 5.7.: Lowest function value found over wallclock time to reach them averaged over ten runs by Propulate (red) and `Optuna` (blue). On both axes lower is better. Propulate is consistently faster and occasionally finds a better result than `Optuna`.

This difference in efficiency also has implications for the energy consumption of the different approaches. Propulate measure only 46.27 W h compared to `Optuna`'s 2646.29 W h on average. This means `Optuna` consumes fifty times more energy than Propulate to solve this task.

5.4.3. Neural Architecture Search for Remote Sensing Classification

To evaluate the performance of Propulate on a use case closer to the intended application, i.e., relatively expensive function evaluation with varying run times, I use a typical neural architecture search problem. Since Propulate is already being used for this purpose and has contributed to several publications [191, 1, 192], I revisit one of these, an image classification problem by Coquelin et al. [191], focusing on the neural architecture search. It should be noted that the results presented in the original paper were achieved with an old version of the algorithm (cf. section 5.5). The results included in this section are representative of the current version of Propulate.

BigEarthNet [193] is a Sentinel-2 multispectral image dataset in remote sensing. It comprises 590 326 image patches, each assigned one or more of the 19 available CORINE Land Cover map labels [194, 193]. Multiple computer vision networks for BigEarthNet classification have been trained [193], with ResNet-50 [195] being closest to state of the art. The task is to optimize the model architecture and training process of BigEarthNet image classification in terms of validation F_1 score.

The search space is shown in table 5.4. The degrees of freedom are similar to the ones considered in the original paper [191]. The training parameters are learning rate, learning rate schedule, loss function, and the different optimizers and their parameters.

The model parameters are activation functions, number of filters per convolutional block, and the activation order inside a block [196]. To simplify the search space, only SGD-based optimizers are included and not Adam-like ones. These optimizers share a parameter subspace. This should not affect the performance of the final model since SGD with a well-tuned learning rate schedule usually performs better than Adam. Including Adam is possible, but (at least in the current version of the algorithm) introduces genes into the space that are only used sometimes, depending on other Genes. This makes the search less efficient and may have led to the problems outlined in the original paper. Each training is stopped early if the validation loss has not increased in the last ten epochs. The data preparation and processing is identical to the original. The network is implemented in TensorFlow [62].

For both Propulate and Optuna we average the results over three searches. Each search is run over 24 h on 32 GPUs. We use $1 - F_1^{\text{val}}$ with the validation F_1 score as the objective function to be minimized.

On average, Optuna achieves its best OF value of (0.39 ± 0.01) within (7.05 ± 3.14) h. Propulate beats Optuna's average best after (5.30 ± 2.41) h and achieves its best OF value of 0.36 within (13.89 ± 5.15) h.

Optimizers	Optimizer parameters		LR warmup parameters	
Adagrad	Initial accum. value	$[10^{-4}, 0.5]$	LR warmup steps	$[10^{0}, 10^{4}]$
SGD	Clipnorm	$[-1, -1000]$	Initial LR	$[10^{-5}, 10^{-1}]$
Adadelta	Clipvalue	$[-1, 1000]$	Decay steps	$[10^{2}, 10^{5}]$
RMSprop	Use EMA	Boolean	LR warmup power	$[10^{-1}, 10^{1}]$
	EMA momentum	$[0.5, 1.0]$		
	EMA overwrite	$[1, 10^{3}]$		
	Momentum	$[0.0, 1.0]$		
	Nesterov	Boolean		
	Rho	$[0.8, 0.99999]$		
	Epsilon	$[10^{-9}, 10^{-4}]$		

Loss functions			LR parameters	
Binary CE	Categorical CE	Categorical hinge	Decay rate	$[0.8, 0.9999]$
Hinge	KL divergence	Squared hinge	Staircase inverse time decay	Boolean
Activation functions			Decay rate	$[0.1, 0.9]$
ELU	ReLU	Softplus	Staircase polynomial decay	Boolean
Exponential	SELU	Softsign		
Hard sigmoid	Sigmoid	Swish	End LR	$[10^{-4}, 10^{-2}]$
Linear	Softmax	Tanh	Power	$[0.5, 2.5]$

Table 5.4.: Hyperparameter search space of ResNet-50 for BigEarthNet image classification.

5.4.4. Scaling

To explore Propulate's scaling behavior I continue to use the neural architecture use case. Figure 5.8 shows the results for weak and strong linear scaling. The baseline configuration used two full nodes. Since each node has four GPUs, speedup and efficiency is calculated with respect to eight workers. For strong scaling, the total number of evaluations is fixed at 512 and the number of workers, i.e., GPUs increases in steps of four. I average over three runs with different seeds and keep four workers per island while increasing the number of islands. Speedup increases up to 128 workers, where we reach approximately half the optimal value. This is an expected decline since each worker only processes few individuals, so the variance in evaluation times leads to larger idle times of the faster workers when they finish all of their assigned generations before the final population synchronization at the end. This inefficiency can partially be avoided, by letting workers run for a specified time rather than a set number of generations. Additionally, as the number of workers approaches the total number of evaluations, the randomly initialized evolutionary search in turn approaches a random search. At the extreme of a single generation no evolutionary operator can be used to inform the search. This means that the search performance is likely to be worse than what the pure compute performance might suggest. It is still possible to apply Propulate on these scales, but the other search parameters have to be adjusted accordingly as shown in the weak scaling plot (see Figure 5.8 top). The parameter that is normally given to Propulate by the user is the number of generations, independent of the worker count. The early super-scalar behavior is likely due to the non-sequential baseline. For small node counts, the performance is influenced by effects stemming from cluster utilization beyond the use case studied here, like file system congestion or inter-node distance in the network. With larger node counts relative to total cluster size, these effects average out or approach the worst case, which is consistent with the trend shown in Figure 5.8. Weak efficiency only drops to 95 % on average at our largest configuration of 128 workers.

5.4.5. Nelder-Mead

Figure 5.9 compares the adapted Nelder-Mead against a sequential reference implementation. All runs perform 1000 function evaluations, with the default parameters of $\alpha = 1.0$, $\gamma = 2.0$, $\rho = 0.5$, and $\sigma = 0.5$. On the sphere and Rosenbrock functions, the reference algorithm performs much better than the Propulate adapted one, with values smaller than 10^{-29}, which are not shown. For the rest of the functions, the search result is of similar quality. The adapted algorithm with a single worker is slightly slower than the reference one. The parallel ones, with two and four workers, respectively, accelerate the process without significant loss of performance. In fact, the parallel runs sometimes produced a result for Rosenbrock comparable to that of the reference algorithm, albeit less reliably. Island sizes much larger than the dimension of the search space are probably inefficient since they will result in a lot of redundant sampling around the starting point.

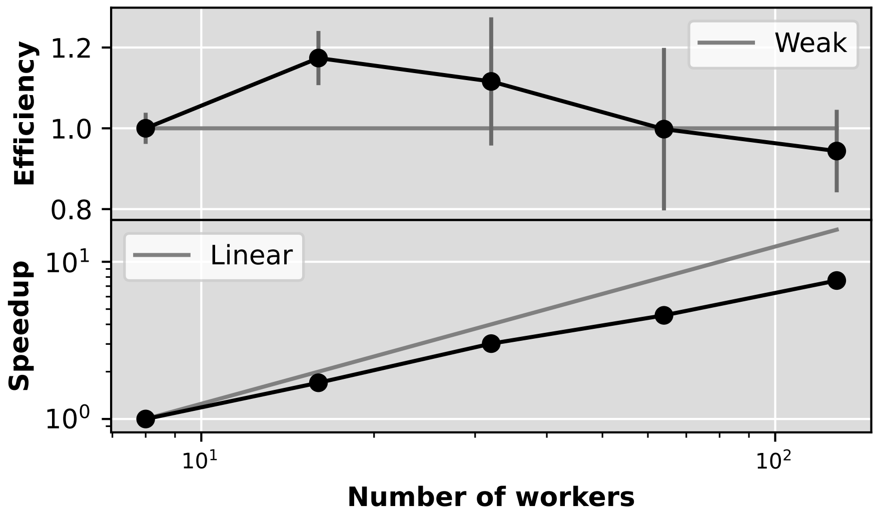


Figure 5.8.: Weak efficiency (top) and strong speedup (bottom) relative to a baseline of eight workers averaged over three runs.

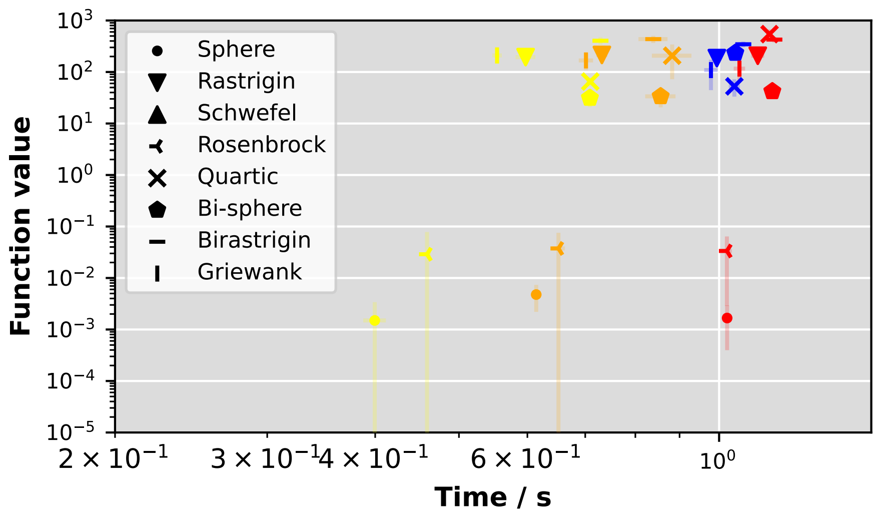


Figure 5.9.: Comparison of the reference Nelder-Mead (blue) against the Propulate adapted version with a single worker (red), two workers (orange) and four workers (red). The bars show standard deviation over ten runs.

5.5. Conclusion

5.5.1. Summary

I presented Propulate, our HPC-adapted platform for population based hyperparameter search coordination using lazy synchronization. While initially intended for neural architecture search, or hyperparameter search in general, its generic design makes it extensible and adaptable to any variable length. HPC task distribution workflow, where the dependencies between task steps can be fuzzy, such that hard synchronization barriers can be softened. I demonstrated Propulates optimization efficacy, its scaling behavior, and its versatility. Propulate is being actively developed and improved and being used beyond the limits of our own research group. Several published scientific works have made use of it already [191, 192] and several more are in preparation.

5.5.2. Discussion

There are several noteworthy aspects of the presented results. In a previous version of Propulate, used in referencing publications [191, 192], I used a coordinator-worker model. Instead of the current implementation, where each worker generates a new solution candidate locally and distributes its information to all other workers, there was a central coordinator rank that held the population and generated new candidates. The workers would request a new suggestion from the coordinator whenever they were free and report their result back once they finished evaluating the objective function. I abandoned this design because it is less user-friendly and robust. The most convenient implementation had one rank spawn an additional process, and the MPI rank hosting the coordinator would then simply send messages to itself, with the target process given by message tag. This exploits behavior that is dependent on the MPI implementation and may not be available on some clusters.

Having a distinguished coordinator rank creates different problems. For neural architecture search problems, the availability of accelerator hardware is the determining factor for how many worker ranks can be assigned to one node. If a single training is not parallelized, one worker is configured per accelerator. One of these ranks instead being used for coordination and breeding means the associated accelerator idles, which implies a relatively large inefficiency. Spawning a single additional rank on one of the participating nodes is not usually limited by computational resources, as the CPUs are not expected to be fully utilized. However, requiring the configuration of an asymmetric distribution of ranks over multiple nodes increases the barrier to entry. The default assumption for MPI programs is that of single-program-multiple-data. The current implementation, where all workers are equal is better adapted to this circumstance.

The current decentralized design avoids these problems. On top of that, it prevents congestion at the beginning of a run. When all workers request a new candidate at the same time, they may have to wait until the coordinator assigns them one.

The difference in wallclock time when comparing Propulate to `Optuna` stems from the used communication technologies. Propulate sends updates and candidate suggestions between workers using asynchronous MPI communication. It does not read from the file system at all during an optimization run. Optuna, however, implements most of its communication through a relational database on disk, which slows the process down drastically. It has to avoid corruption of the database while workers read from and write to it at arbitrary times. This effect is mitigated somewhat for the neural architecture use case since a single evaluation of the objective function is more expensive. Propulate uses parallel HDF5 for its checkpointing. In the beginning of a search, space for parameter data and metadata is allocated and the responsibilities for each rank are assigned. If a search is picked up from an existing checkpoint, the relevant data is read by each rank. After this setup, each rank only touches the areas of the file it has responsibility over and no further communication is required.

5.5.3. Outlook

Several more algorithms might be suitable for adaptation to lazy synchronization [197, 198, 199, 200, 201]. In particular, Bayesian optimization techniques, which are widely used for neural architecture search, still need to be included.

Beyond that, gray box techniques like successive halving, intelligent pruning, or performance prediction based on partial training are promising avenues to reduce computational cost and improve efficiency. The latter two are the object of another bachelor thesis by Vito Diercksen, which is currently in the process of being implemented. Another approach to speeding up the evaluation of a single candidate is to assign more resources. Currently, the degree of parallelization is capped by the limit of a single node. A third level in the communication hierarchy of Propulate below the current lowest, from island and worker to island, search worker, and evaluation worker for lack of better terms, would enable further scaling by enabling data and/or model parallelism during the search. These techniques all aim to improve the speed of evaluation, which only increases the impact of the efficient communication enabled by Propulate when compared to suites like Optuna.

Another established approach to increase the feasible number of evaluations is zero-shot learning, like NASWOT [202], where model performance is estimated without going through a training loop. While it is already possible to integrate these into the black box objective function, I have not systematically investigated them so far. Related to this is an approach uniquely enabled by Propulate. The training process of a neural network is significantly influenced by its weight initialization. In Propulate the initialization of individuals in later generations can be informed by the training of previous ones. Sometimes, the random seed used to set the initial weights is already treated as a hyperparameter to be optimized together with the model architecture and training setup, but here information is available on improved parameters after partial optimization. Transplanting a good initialization from a previously evaluated individual has the prospect of achieving final model performance in later generations with fewer training iterations. Measuring the model performance directly, without the need for surrogate model estimates or extrapolations or training at full cost throughout the search process again increases efficiency. The overhead of these proxies can be discarded and the final model does not have to be re-trained from scratch after obtaining a good set of hyperparameters.

Many challenges remain for solving the problem of hyperparameter optimization. The choice of algorithm from a pool of thousands simply escalates the problem of hyperparameter selection to the functionally identical problem of search algorithm selection. Propulate might be able to address this problem by employing a different algorithm on each island and letting them exchange information. Ideally, this could also include dynamically adapting search behavior, e.g., by scheduling the exploration vs. exploitation trade off, adapting island sizes and assigning more resources to better-performing samplers, or abandoning irrelevant regions of the search space. A robust, standardized neural architecture search benchmarking dataset would also help make transparent and informed recommendations on how to approach a search for a given problem.

On a technical level, the problem of the size of the search space depending on the other parameters in it remains. For example, different training algorithms for the inner training loop may have a different number of parameters. Current workarounds solve this problem through redundancy, which makes the search less efficient. Using a fixed size embedding for sampling, which is then projected into the actual hyperparameter space might be another viable approach.

6. Conclusion

This dissertation presented my work on biologically inspired machine learning techniques applied to biological machine learning problems. In this chapter we will summarize and recapitulate and provide some more context for the individual previous chapters with a focus on the overarching context. We will also discuss future directions in a broader context beyond what we have already provided in the individual chapters.

6.1. Discussion

Data science in general or machine learning more specifically, has been established as another pillar of the natural sciences next to experiment, theory, and simulation. Machine learning of molecular biology capitalizes on the progress of different domains of language processing, geometric learning, and image processing. However, these modern data science technologies, in particular deep neural networks, necessitate the utilization of large amounts of data. For example, the solution of some problems like protein structure prediction has been enabled by the availability of vast numbers of both genetic sequences and structurally resolved molecules. In areas where training data is less abundant, and this situation is expected to persist for the foreseeable future, modern approaches can still be applied. The sparse data that is available has to be utilized as efficiently as possible in order to transfer the progress of protein structure prediction to RNA structure prediction, molecular function characterization, or molecular design. Solving these problems in turn promises impactful new discoveries in industrial and medical applications.

The original research task of my scientific work was to leverage the attention mechanism and transformer models for molecular structure prediction. Evolution has been a main theme of my work in so far as machine learning approaches to biological questions oftentimes endeavor to exploit information left behind in the training data by the evolutionary processes that shaped them. However, traditionally used methods like direct coupling analysis are stateless in the sense, that they produce an output from only a single sequence alignment. Neural networks encode a memory of the data seen during training in their weights. This enables generalization, transfer learning, and zero-shot predictions.

Comparing the current performance of neural networks on RNA problems with those trained on proteins, or the difference in performance of the models presented in chapter 4 highlights the impact data sparsity has. In the more classical deep learning domains of image classification or text processing, there is at most a sparsity of labels. A language model presented with enough data and with enough capacity might learn translation, without having been explicitly shown corresponding pairs of expressions in different languages, given that this information is present in the training data. In more specialized areas like scientific machine learning applications, certain types of input training data are already sparse.

In these settings, we expect the approaches we propose in chapter 3 and chapter 4, based on self-supervised learning, robust and data-efficient models, and better understanding of the loss landscape to be useful beyond the biological applications we originally conceived of them for.

Adding more complexity to the already difficult problem of neural network training exacerbates the already escalating problem of neural architecture search. Particularly for a multi-stage training process as with a self-supervised pre-training, exploring the hyper-parameter space manually becomes less effective. This circumstance sparked the development of Populate. Evolutionary optimization seems a natural fit for hyper-parameter search next to bayesian models. This closes the circle and brings evolution back from being the data learned from to the driving mechanism.

Just like the wider machine learning field, neural architecture search is a rapidly developing area of research. There is a growing zoo of hyper-parameter suites, many of which are surprisingly difficult to deploy at as large a scale, as the problem of neural architecture search seems to demand.

6.2. Further Research

We explored avenues for further research in the context of the isolated problems in the respective individual chapters already. Tying everything together, loss scheduling or task scheduling could be a way of improving self-supervised pre-training. Dynamically adapting upstream task design and in particular task difficulty might make that search space easier to explore and make it unnecessary to find a harder to hit sweet spot.

To perform hyper-parameter searches for this multi-stage process in particular, a partitioning of the complete search space into a hierarchical structure promises large gains in efficiency. For a single generation pre-training, an exhaustive search of the much cheaper to evaluate downstream model can be performed to convergence. This way the much more expensive pre-training can be reused or rather not be undervalued by the algorithm, because the hyper-parameters selected for the second stage happened not to match the ones for the first stage well. The complete model in chapter 3 is even a three stage process: pre-training, finetuning, and building of the downstream forest model. There are similar multi-stage modeling processes distinct from self-supervised pre-training. One of these is currently in preparation to be published using Populate for its hyper-parameter search. To my knowledge, none of the available tools covers this use-case. Likely because this kind of partitioning would require coupling and communication between individuals or trials, which is easier to facilitate in a design like that of Populate.

In addition to the discussed topics, we have also conducted preliminary research on related biological applications. Namely, gene expression network prediction[203, 204]. Currently traditional unsupervised machine learning techniques like different clustering algorithms or signal processing inspired ones like independent component analysis [205] (ICA) are used. Independent component analysis was originally intended for source separation. For its applications, e.g. speaker separation, neural networks have mostly succeeded it [206]. As in the other use-cases discussed, the unsolved problem in bringing deep learning to this field is primarily one of data sparsity, but also of validation.

Since all extant forms of life are usually thought of to originate from a single common ancestor, there is only a single tree of life. How many degrees of separation have to be maintained to not over-estimate performance and how many are too much to consider the data in distribution often seems to be determined arbitrarily. On sequence data, a minimum sequence dissimilarity is often used to cluster the data [207]. One set of clusters is used for training and another for validation or testing. The cluster definition is then arbitrary, but for other types of data the boundaries are even more blurry.

The concept of evolution has been a driver on different aspects of machine learning progress. The two we examined in this thesis, extraction of structural information from patterns caused by molecular evolution and evolution inspired scalable optimization, will certainly cotinue to remain relevant.

Bibliography

[1] Oskar Taubert et al. “RNA Contact Prediction by Data Efficient Deep Learning”. In: *Communications Biology* (2023). DOI: 10.1038/s42003-023-05244-9.

[2] Oskar Taubert et al. “Loss Scheduling for Class-Imbalanced Image Segmentation Problems”. In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2020, pp. 426–431. DOI: 10.1109/ICMLA51294.2020.00073.

[3] Oskar Taubert et al. “Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations”. In: *International Conference on High Performance Computing*. Springer. 2023, pp. 106–124. DOI: 10.1007/978-3-031-32041-5_6.

[4] Oskar Taubert et al. “diSTruct v1.0: generating biomolecular structures from distance constraints”. In: *Bioinformatics* 35.24 (July 2019), pp. 5337–5338. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz578.

[5] James Kahn et al. “Learning tree structures from leaves for particle decay reconstruction”. In: *Machine Learning: Science and Technology* 3.3 (Sept. 2022), p. 035012. DOI: 10.1088/2632-2153/ac8de0.

[6] James Kahn et al. *Lowest Common Ancestor Generations (LCAG) Phasespace Particle Decay Reconstruction Dataset*. 46.21.04; LK 01. 2022. DOI: 10.5281/zenodo.6983258.

[7] John Moult et al. *A large-scale experiment to assess protein structure prediction methods*. 1995.

[8] John Jumper et al. “Highly accurate protein structure prediction with AlphaFold”. In: *Nature* 596.7873 (2021), pp. 583–589.

[9] Minkyung Baek et al. “Accurate prediction of protein structures and interactions using a three-track neural network”. In: *Science* 373.6557 (2021), pp. 871–876.

[10] Zeming Lin et al. “Evolutionary-scale prediction of atomic-level protein structure with a language model”. In: *Science* 379.6637 (2023), pp. 1123–1130.

[11] Itamar Kass and Amnon Horovitz. “Mapping pathways of allosteric communication in GroEL by analysis of correlated mutations”. In: *Proteins: Structure, Function, and Bioinformatics* 48.4 (2002), pp. 611–617. DOI: https://doi.org/10.1002/prot.10180.

[12] Martin Weigt et al. “Identification of direct residue contacts in protein–protein interaction by message passing”. In: *Proceedings of the National Academy of Sciences* 106.1 (2009), pp. 67–72.

[13] Alexandru Niculescu-Mizil and Rich Caruana. “Predicting good probabilities with supervised learning”. In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 625–632.

[14] Tsung-Yi Lin et al. “Focal Loss for Dense Object Detection”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2017, pp. 2980–2988. DOI: 10.1109/ICCV.2017.324.

[15] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *Computer Vision – ECCV 2014*. Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1. DOI: 10.1007/978-3-319-10602-1{_}48.

[16] Mark Everingham et al. “The PASCAL Visual Object Classes Challenge”. In: *International Journal of Computer Vision* 88 (2 2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[17] Marius Cordts et al. “The Cityscapes Dataset for Semantic Urban Scene Understanding”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 3213–3223. DOI: 10.1109/CVPR.2016.350.

[18] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).

[19] Victor Sanh et al. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter”. In: *arXiv preprint arXiv:1910.01108* (2019).

[20] Dario Amodei and Danny Hernandez. *AI and compute*. Last accessed 14 September 2023. 2016. URL: https://openai.com/research/ai-and-compute.

[21] Pradnya A. Vikhar. “Evolutionary algorithms: A critical review and its future prospects”. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. 2016, pp. 261–265. DOI: 10.1109/ICGTSPICC.2016.7955308.

[22] Charles Darwin. “On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life”. In: *London: John Murray* (1859).

[23] JL Diener and PB Moore. *SOLUTION STRUCTURE OF A SUBSTRATE FOR THE ARCHAEAL PRE-TRNA SPLICING ENDONUCLEASES: THE BULGE-HELIX-BULGE MOTIF, NMR, 12 STRUCTURES*. 1998. DOI: 10.2210/pdb1A9L/pdb.

[24] Roger P Alexander et al. “Annotating non-coding regions of the genome”. In: *Nature Reviews Genetics* 11.8 (2010), pp. 559–571.

[25] LadyOfHats. *A ribosome produces a protein using mRNA as template*. Public Domain, https://commons.wikimedia.org/w/index.php?curid=4889777. URL: https://en.wikipedia.org/wiki/Protein#/media/File:Ribosome_mRNA_translation_en.svg.

[26] Martin Peng et al. “Direct coupling analysis improves the identification of beneficial amino acid mutations for the functional thermostabilization of a delicate decarboxylase”. In: *Biological chemistry* 400.11 (2019), pp. 1519–1527.

[27] Sònia Casillas and Antonio Barbadilla. “Molecular population genetics”. In: *Genetics* 205.3 (2017), pp. 1003–1035.

[28] John C Kendrew et al. “A three-dimensional model of the myoglobin molecule obtained by x-ray analysis”. In: *Nature* 181.4610 (1958), pp. 662–666.

[29] Kurt Wüthrich. “Protein structure determination in solution by NMR spectroscopy.” In: *Journal of Biological Chemistry* 265.36 (1990), pp. 22059–22062.

[30] Boris Fürtig et al. “NMR spectroscopy of RNA”. In: *ChemBioChem* 4.10 (2003), pp. 936–962.

[31] Xueming Li et al. “Electron counting and beam-induced motion correction enable near-atomic-resolution single-particle cryo-EM”. In: *Nature methods* 10.6 (2013), pp. 584–590.

[32] Herman JC Berendsen, David van der Spoel, and Rudi van Drunen. “GROMACS: A message-passing parallel molecular dynamics implementation”. In: *Computer physics communications* 91.1-3 (1995), pp. 43–56.

[33] Alexander Schug and Wolfgang Wenzel. “Predictive in silico all-atom folding of a four-helix protein with a free-energy model”. In: *Journal of the American Chemical Society* 126.51 (2004), pp. 16736–16737.

[34] Kresten Lindorff-Larsen et al. “How fast-folding proteins fold”. In: *Science* 334.6055 (2011), pp. 517–520.

[35] Claude Sinner et al. “Simulating Biomolecular Folding and Function by Native-Structure-Based/Go-Type Models”. In: *Israel Journal of Chemistry* 54.8-9 (2014), pp. 1165–1175.

[36] The UniProt Consortium. “UniProt: the Universal Protein Knowledgebase in 2023”. In: *Nucleic Acids Research* 51.D1 (Nov. 2022), pp. D523–D531. DOI: 10.1093/nar/gkac1052.

[37] Ioanna Kalvari et al. “RFam 14: expanded coverage of metagenomic, viral and microRNA families”. In: *Nucleic Acids Research* 49.D1 (Nov. 2020), pp. D192–D200. DOI: 10.1093/nar/gkaa1047.

[38] “RNAcentral: a comprehensive database of non-coding RNA sequences”. In: *Nucleic acids research* 45.D1 (2017), pp. D128–D134.

[39] Helen M Berman et al. “The protein data bank”. In: *Nucleic acids research* 28.1 (2000), pp. 235–242. DOI: 10.1093/nar/28.1.235.

[40] Faruck Morcos et al. “Direct-coupling analysis of residue coevolution captures native contacts across many protein families”. In: *Proceedings of the National Academy of Sciences* 108.49 (2011), E1293–E1301. DOI: 10.1073/pnas.1111471108.

[41] Andrew W Senior et al. “Improved protein structure prediction using potentials from deep learning”. In: *Nature* 577.7792 (2020), pp. 706–710.

[42] Alexander Schug et al. “High-resolution protein complexes from integrating genomic information with molecular simulation”. In: *Proceedings of the National Academy of Sciences* 106.52 (2009), pp. 22124–22129.

[43] Saul B Needleman and Christian D Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[44] Robert A Wagner and Michael J Fischer. “The string-to-string correction problem”. In: *Journal of the ACM (JACM)* 21.1 (1974), pp. 168–173.

[45] Temple F Smith, Michael S Waterman, et al. “Identification of common molecular subsequences”. In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

[46] Sean R Eddy. “Accelerated profile HMM searches”. In: *PLoS computational biology* 7.10 (2011), e1002195.

[47] Martin Steinegger et al. “HH-suite3 for fast remote homology detection and deep protein annotation”. In: *BMC bioinformatics* 20.1 (2019), pp. 1–15.

[48] *RFam family 00957*. Last accessed 14 September 2023. URL: https://rfam.org/family/RF00957.

[49] Magnus Ekeberg et al. “Improved contact prediction in proteins: using pseudolikelihoods to infer Potts models”. In: *Physical Review E* 87.1 (2013), p. 012707. DOI: 10.1103/PhysRevE.87.012707.

[50] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[51] Deepak Pathak et al. “Context encoders: Feature learning by inpainting”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2536–2544.

[52] Mehdi Noroozi and Paolo Favaro. “Unsupervised learning of visual representations by solving jigsaw puzzles”. In: *European conference on computer vision*. Springer. 2016, pp. 69–84. DOI: 10.1007/978-3-319-46466-4_5.

[53] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[54] Ting Chen et al. “A simple framework for contrastive learning of visual representations”. In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR. 2020, pp. 1597–1607. URL: https://proceedings.mlr.press/v119/chen20j.html.

[55] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[56] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[57] Francis Galton. “Regression towards mediocrity in hereditary stature.” In: *The Journal of the Anthropological Institute of Great Britain and Ireland* 15 (1886), pp. 246–263.

[58] Jaideep Pathak et al. *FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators*. 2022. arXiv: 2202.11214 [physics.ao-ph].

[59] Chuan Guo et al. "On Calibration of Modern Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. JMLR, 2017, pp. 1321–1330.

[60] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[61] Robin Rombach et al. "High-resolution image synthesis with latent diffusion models". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 10684–10695.

[62] Martin Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.

[63] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[64] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[65] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. *Learning internal representations by error propagation*. 1985.

[66] Jürgen Schmidhuber. "Annotated history of modern AI and Deep learning". In: *arXiv preprint arXiv:2212.11279* (2022).

[67] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[68] Dan Hendrycks and Kevin Gimpel. "Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units". In: *CoRR* abs/1606.08415 (2016). arXiv: 1606.08415. URL: http://arxiv.org/abs/1606.08415.

[69] Ruo-Yu Sun. "Optimization for deep learning: An overview". In: *Journal of the Operations Research Society of China* 8.2 (2020), pp. 249–294.

[70] Alex Graves. "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850* (2013).

[71] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[72] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[73] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[74] David Balduzzi et al. *The Shattered Gradients Problem: If resnets are the answer, then what is the question?* 2018. arXiv: 1702.08591 [cs.NE].

[75] Purvil Bambharolia. "Overview of Convolutional Neural Networks". In: *Proceedings of the International Conference on Academic Research in Engineering and Management, Monastir, Tunisia.* 2017, pp. 8–10.

[76] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[77] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

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

[79] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017). URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[80] Niki Parmar et al. *Image Transformer.* 2018. arXiv: 1802.05751 [cs.CV].

[81] Petar Veličković et al. *Graph Attention Networks.* 2018. arXiv: 1710.10903.

[82] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning.* 2023. arXiv: 2307.08691 [cs.LG].

[83] Jonathan Ho et al. *Axial Attention in Multidimensional Transformers.* 2019. DOI: 10.48550/arXiv.1912.12180.

[84] Ludvig Ericson and Rendani Mbuvha. "On the performance of network parallel training in artificial neural networks". In: *arXiv preprint arXiv:1701.05130* (2017).

[85] Roy Schwartz et al. "Green ai". In: *Communications of the ACM* 63.12 (2020), pp. 54–63.

[86] L Breiman et al. "Classification and Regression Trees". In: (1984).

[87] Yoav Freund and Robert E Schapire. "A desicion-theoretic generalization of on-line learning and an application to boosting". In: *European conference on computational learning theory.* Springer. 1995, pp. 23–37.

[88] Jerome H Friedman. "Stochastic gradient boosting". In: *Computational statistics & data analysis* 38.4 (2002), pp. 367–378.

[89] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD '16. ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785.

[90] Tom Fawcett. "An Introduction to ROC Analysis". In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874. DOI: https://doi.org/10.1016/j.patrec.2005.10.010.

[91] Colin White et al. "Neural architecture search: Insights from 1000 papers". In: *arXiv preprint arXiv:2301.08727* (2023).

[92] Erick Cantú-Paz et al. "A Survey of Parallel Genetic Algorithms". In: *Calculateurs paralleles, reseaux et systems repartis* 10.2 (1998), pp. 141–171.

[93] Jakub Kudela. “The Evolutionary Computation Methods No One Should Use”. In: *arXiv preprint arXiv:2301.01984* (2023).

[94] Tansel Dokeroglu, Ayça Deniz, and Hakan Ezgi Kiziloz. “A comprehensive survey on recent metaheuristics for feature selection”. In: *Neurocomputing* 494 (2022), pp. 269–296.

[95] Bernd Bischl et al. “Hyperparameter optimization: Foundations, algorithms, best practices and open challenges. arXiv 2021”. In: *arXiv preprint arXiv:2107.05847* ().

[96] Christian Blum and Andrea Roli. “Metaheuristics in combinatorial optimization: Overview and conceptual comparison”. In: *ACM computing surveys (CSUR)* 35.3 (2003), pp. 268–308.

[97] Enrique Alba and José M. Troya. “A Survey of Parallel Distributed Genetic Algorithms”. In: *Complexity* 4.4 (1999), pp. 31–52.

[98] Enrique Alba and Marco Tomassini. “Parallelism and Evolutionary Algorithms”. In: *IEEE Transactions on Evolutionary Computation* 6.5 (2002), pp. 443–462. DOI: 10.1109/TEVC.2002.800880.

[99] Zhi-Hui Zhan et al. “A survey on evolutionary computation for complex continuous optimization”. In: *Artificial Intelligence Review* (2022), pp. 1–52.

[100] Leonora Bianchi et al. “A survey on metaheuristics for stochastic combinatorial optimization”. In: *Natural Computing* 8 (2009), pp. 239–287.

[101] Zhongqiang Ma et al. “Performance assessment and exhaustive listing of 500+ nature-inspired metaheuristic algorithms”. In: *Swarm and Evolutionary Computation* 77 (2023), p. 101248.

[102] Daniel Molina et al. “Comprehensive taxonomies of nature-and bio-inspired optimization: Inspiration versus algorithmic behavior, critical analysis recommendations”. In: *Cognitive Computation* 12 (2020), pp. 897–939.

[103] Alexandros Tzanetos and Georgios Dounias. “Nature inspired optimization algorithms or simply variations of metaheuristics?” In: *Artificial Intelligence Review* 54 (2021), pp. 1841–1862.

[104] Abdulaziz Alorf. “A survey of recently developed metaheuristics and their comparative analysis”. In: *Engineering Applications of Artificial Intelligence* 117 (2023), p. 105622.

[105] Yuhui Shi and Russell Eberhart. “A modified particle swarm optimizer”. In: *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*. IEEE. 1998, pp. 69–73.

[106] Riccardo Poli, James Kennedy, and Tim Blackwell. “Particle swarm optimization: An overview”. In: *Swarm intelligence* 1 (2007), pp. 33–57.

[107] Nikolaus Hansen. “The CMA evolution strategy: A tutorial”. In: *arXiv preprint arXiv:1604.00772* (2016).

[108] Christian Igel, Thorsten Suttorp, and Nikolaus Hansen. “A computational efficient covariance matrix update and a (1+ 1)-CMA for evolution strategies”. In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 2006, pp. 453–460.

[109] Eric Brochu, Vlad M Cora, and Nando De Freitas. “A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning”. In: *arXiv preprint arXiv:1012.2599* (2010).

[110] James Bergstra et al. “Algorithms for Hyper-Parameter Optimization”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[111] Shuhei Watanabe. “Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance”. In: *arXiv preprint arXiv:2304.11127* (2023).

[112] Barret Zoph and Quoc V Le. “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578* (2016).

[113] Hadi S Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. “Hyp-rl: Hyperparameter optimization by reinforcement learning”. In: *arXiv preprint arXiv:1906.11527* (2019).

[114] Dario Floreano, Peter Dürr, and Claudio Mattiussi. “Neuroevolution: from architectures to learning”. In: *Evolutionary intelligence* 1 (2008), pp. 47–62.

[115] Risto Miikkulainen et al. “Evolving deep neural networks”. In: *Artificial intelligence in the age of neural networks and brain computing*. Elsevier, 2024, pp. 269–287.

[116] Max Jaderberg et al. “Population based training of neural networks”. In: *arXiv preprint arXiv:1711.09846* (2017).

[117] David W Walker. *Standards for message-passing in a distributed memory environment*. Tech. rep. Oak Ridge National Lab., TN (United States), 1992.

[118] NVidia Corporation. *NVidia Collective Communication Library*. 2023. URL: https://github.com/NVIDIA/nccl.

[119] RCCL contributors. *ROCm Collective Communication Library*. 2023. URL: https://github.com/ROCmSoftwarePlatform/rccl.

[120] John Jumper et al. “Highly accurate protein structure prediction with AlphaFold”. In: *Nature* 596.7873 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2.

[121] Fabrizio Pucci et al. “Evaluating DCA-based method performances for RNA contact prediction by a well-curated data set”. In: *RNA* 26.7 (2020), pp. 794–802. DOI: 10.1261/rna.073809.119.

[122] Roshan M Rao et al. “MSA Transformer”. In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 8844–8856. DOI: 10.1101/2021.02.12.430858.

[123] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. “Training region-based object detectors with online hard example mining”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 761–769.

[124] Florian Schroff, Dmitry Kalenichenko, and James Philbin. “FaceNet: A unified embedding for face recognition and clustering”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2015. DOI: 10.1109/cvpr.2015.7298682. URL: https://doi.org/10.1109%2Fcvpr.2015.7298682.

[125] Ravid Shwartz-Ziv and Amitai Armon. *Tabular Data: Deep Learning is Not All You Need*. 2021. arXiv: 2106.03253 [cs.LG].

[126] Sheng Wang et al. “Accurate De Novo Prediction of Protein Contact Map by Ultra-Deep Learning Model”. In: *PLOS Computational Biology* 13.1 (2017), e1005324.

[127] Mehari B Zerihun, Fabrizio Pucci, and Alexander Schug. “CoCoNet—boosting RNA contact prediction by convolutional neural networks”. In: *Nucleic acids research* 49.22 (2021), pp. 12661–12672. DOI: 10.1093/nar/gkab1144.

[128] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[129] Zeming Lin et al. “Language models of protein sequences at the scale of evolution enable accurate structure prediction”. In: *bioRxiv* (2022).

[130] Raphael JL Townshend et al. “Geometric deep learning of RNA structure”. In: *Science* 373.6558 (2021), pp. 1047–1051.

[131] Mohammed AlQuraishi. “End-to-end differentiable learning of protein structure”. In: *Cell systems* 8.4 (2019), pp. 292–301.

[132] Fabian B. Fuchs et al. *SE(3)-Transformers: 3D Roto-Translation Equivariant Attention Networks*. 2020. arXiv: 2006.10503 [cs.LG].

[133] Jörg KH Franke, Frederic Runge, and Frank Hutter. “Scalable Deep Learning for RNA Secondary Structure Prediction”. In: *arXiv preprint arXiv:2307.10073* (2023).

[134] Ioanna Kalvari et al. “Non-coding RNA analysis using the Rfam database”. In: *Current protocols in bioinformatics* 62.1 (2018), e51. DOI: 10.1002/cpbi.51.

[135] Ioanna Kalvari et al. “Rfam 14: expanded coverage of metagenomic, viral and microRNA families”. In: *Nucleic Acids Research* 49.D1 (2021), pp. D192–D200. DOI: 10.1093/nar/gkaa1047.

[136] Zasha Weinberg et al. “Detection of 224 candidate structured RNAs by comparative analysis of specific subsets of intergenic regions”. In: *Nucleic acids research* 45.18 (2017), pp. 10811–10823. DOI: 10.1093/nar/gkx699.

[137] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems* 27 (2014).

[138] Rashmi Korlakai Vinayak and Ran Gilad-Bachrach. “DART: Dropouts meet Multiple Additive Regression Trees”. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*. PMLR. 2015, pp. 489–497. URL: https://proceedings.mlr.press/v38/korlakaivinayak15.html.

[139] Yuedong Yang et al. "Genome-scale characterization of RNA tertiary structures and their functional impact by RNA solvent accessibility prediction". In: *Rna* 23.1 (2017), pp. 14–22.

[140] Anand V. Sastry et al. "Mining all publicly available expression data to compute dynamic microbial transcriptional regulatory networks". In: *bioRxiv* (2021). DOI: 10.1101/2021.07.01.450581.

[141] Rohit Girdhar et al. "Imagebind: One embedding space to bind them all". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2023, pp. 15180–15190.

[142] Fabio Ferreira, Ivo Rapant, and Frank Hutter. "Hard View Selection for Contrastive Learning". In: *arXiv preprint arXiv:2310.03940* (2023).

[143] Viacheslav Khomenko et al. "Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization". In: *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP).* IEEE. 2016, pp. 100–103.

[144] Alexander Rives et al. "Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences". In: *Proceedings of the National Academy of Sciences* 118.15 (2021). DOI: 10.1101/622803.

[145] Ben Philps, Maria del C Valdes Hernandez, and Miguel Bernabeu Llinares. "Proper Scoring Loss Functions Are Simple and Effective for Uncertainty Quantification of White Matter Hyperintensities". In: *International Workshop on Uncertainty for Safe Utilization of Machine Learning in Medical Imaging.* Springer. 2023, pp. 208–218.

[146] Carole Sudre et al. "Generalised Dice Overlap as a Deep Learning Loss Function for Highly Unbalanced Segmentations". In: *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support.* Springer, 2017, pp. 240–248. ISBN: 978-3-319-67558-9. DOI: 10.1007/978-3-319-67558-9{_}28.

[147] Lena Maier-Hein, Bjoern Menze, et al. "Metrics reloaded: Pitfalls and recommendations for image analysis validation". In: *arXiv. org* 2206.01653 (2022).

[148] Jonathan Frankle and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks". In: *arXiv preprint arXiv:1803.03635* (2018).

[149] Bharath Hariharan et al. "Semantic Contours from Inverse Detectors". In: *2011 International Conference on Computer Vision (ICCV).* IEEE, 2011, pp. 991–998. DOI: 10.1109/ICCV.2011.6126343.

[150] Andrej Karpathy. *A Recipe for Training Neural Networks.* Last accessed 14 September 2023. 2019. URL: https://karpathy.github.io/2019/04/25/recipe/.

[151] Alberto Garcia-Garcia et al. "A Review on Deep Learning Techniques Applied to Semantic Segmentation". In: *Applied Soft Computing* 70 (2018), pp. 41–65. DOI: 10.1016/j.asoc.2018.05.018.

[152] Nitesh Chawla et al. "SMOTE: Synthetic Minority Over-sampling Technique". In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357. ISSN: 1076-9757. DOI: 10.1613/jair.953.

[153] Mahdi Pakdaman Naeini, Gregory Cooper, and Milos Hauskrecht. “Obtaining well calibrated probabilities using bayesian binning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 29. 1. 2015.

[154] Tilmann Gneiting and Adrian E Raftery. “Strictly proper scoring rules, prediction, and estimation”. In: *Journal of the American statistical Association* 102.477 (2007), pp. 359–378.

[155] Michael Yeung et al. “Calibrating the Dice loss to handle neural network overconfidence for biomedical image segmentation”. In: *Journal of Digital Imaging* 36.2 (2023), pp. 739–752.

[156] Samuel Rota Bulo, Gerhard Neuhold, and Peter Kontschieder. “Loss Max-Pooling for Semantic Image Segmentation”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 7082–7091. DOI: 10.1109/CVPR.2017.749.

[157] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 3431–3440. DOI: 10.1109/CVPR.2015.7298965.

[158] Liang-Chieh Chen et al. *Rethinking Atrous Convolution for Semantic Image Segmentation*. [accessed at 2020-02-26]. 2017. URL: https://arxiv.org/abs/1706.05587.

[159] Torch Contributors. *torchvision*. [online, accessed at 2020-02-04]. 2019. URL: https://pytorch.org/docs/stable/torchvision/index.html.

[160] Alethea Power et al. “Grokking: Generalization beyond overfitting on small algorithmic datasets”. In: *arXiv preprint arXiv:2201.02177* (2022).

[161] Pengzhen Ren et al. “A comprehensive survey of neural architecture search: Challenges and solutions”. In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–34.

[162] Charlotte Debus et al. “Reporting electricity consumption is essential for sustainable AI”. In: *Nature Machine Intelligence* (2023), pp. 1–3.

[163] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 2623–2631. DOI: 10.1145/3292500.3330701.

[164] 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*. PMLR. 2013, pp. 115–123. URL: http://proceedings.mlr.press/v28/bergstra13.pdf.

[165] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “Sequential Model-based Optimization for General Algorithm Configuration”. In: *International Conference on Learning and Intelligent Optimization*. Springer. 2011, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40.

[166] Marius Lindauer et al. “SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization.” In: *J. Mach. Learn. Res.* 23 (2022), pp. 54–1.

[167] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

[168] The GPyOpt Authors. *GPyOpt: A Bayesian Optimization Framework in Python*. http://github.com/SheffieldML/GPyOpt. 2016.

[169] Jialei Wang et al. "Parallel Bayesian global optimization of expensive functions". In: *Operations Research* 68.6 (2020), pp. 1850–1865.

[170] DeepHyper Development Team. *"DeepHyper: A Python Package for Scalable Neural Architecture and Hyperparameter Search"*. DeepHyper Team, 2018. URL: https://github.com/deephyper/deephyper.

[171] Romain Egele et al. "Asynchronous Decentralized Bayesian Optimization for Large Scale Hyperparameter Optimization". In: *eScience* (2023).

[172] Lars Hertel et al. "Sherpa: Hyperparameter Optimization for Machine Learning Models". In: *32nd Conference on Neural Information Processing Systems (NIPS 2018)* (2018). URL: https://github.com/sherpa-ai/sherpa.

[173] J. Rapin and O. Teytaud. *Nevergrad - A Gradient-free Optimization Platform*. https://GitHub.com/FacebookResearch/Nevergrad. 2018.

[174] Patrick Koch et al. "Autotune: A Derivative-free Optimization Framework for Hyperparameter Tuning". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 443–452. DOI: 10.1145/3219819.3219837.

[175] Xingyou Song et al. "Open Source Vizier: Distributed Infrastructure and API for Reliable and Flexible Blackbox Optimization". In: *Automated Machine Learning Conference, Systems Track (AutoML-Conf Systems)*. 2022. URL: https://github.com/google/vizier.

[176] Daniel Golovin et al. "Google Vizier: A Service for Black-Box Optimization". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2017, pp. 1487–1495. DOI: 10.1145/3097983.3098043.

[177] Johnu George et al. *A Scalable and Cloud-Native Hyperparameter Tuning System*. 2020. DOI: 10.48550/arXiv.2006.02085. arXiv: 2006.02085.

[178] Richard Liaw et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018). DOI: 10.48550/arXiv.1807.05118.

[179] Mathias Lechner et al. "PyHopper–Hyperparameter optimization". In: *arXiv preprint arXiv:2210.04728* (2022).

[180] Marie Weiel et al. "Dynamic Particle Swarm Optimization of Biomolecular Simulation Parameters with Flexible Ojective Functions". In: *Nature Machine Intelligence* 3.8 (2021), pp. 727–734. DOI: 10.1038/s42256-021-00366-3.

[181] Félix-Antoine Fortin et al. “DEAP: Evolutionary Algorithms Made Easy”. In: *The Journal of Machine Learning Research* 13.1 (2012), pp. 2171–2175.

[182] Steven R Young et al. “Optimizing deep learning hyper-parameters through an evolutionary algorithm”. In: *Proceedings of the workshop on machine learning in high-performance computing environments*. 2015, pp. 1–5.

[183] Nihat Engin Toklu et al. *EvoTorch: Advanced evolutionary computation library built directly on top of PyTorch, created at NNAISENSE*. https://github.com/nnaisense/evotorch. 2022.

[184] Dirk Sudholt. *Parallel Evolutionary Algorithms – Chapter in the Handbook of Computational Intelligence*. Springer, 2015. ISBN: 978-3-662-43505-2. DOI: 10.1007/978-3-662-43505-2_46.

[185] Erick Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Vol. 1. Springer Science & Business Media, 2000. DOI: 10.1007/978-1-4615-4369-5.

[186] Paul Zanner. *Asynchronous Particle Swarms for Solving Global Optimisation Problems in Parallel on Supercomputers*. Karlsruhe, 2023.

[187] Jonathan Roth. *Massively Parallel and Asynchronous Covariance Matrix Adaptation Evolutionary Strategy*. Karlsruhe, 2023.

[188] John A Nelder and Roger Mead. “A simplex method for function minimization”. In: *The computer journal* 7.4 (1965), pp. 308–313.

[189] Monte Lunacek, Darrell Whitley, and Andrew Sutton. “The Impact of Global Structure on Search”. In: *Parallel Problem Solving from Nature – PPSN X, LNCS 5199*. Springer, 2008, pp. 498–507. DOI: 10.1007/978-3-540-87700-4_50.

[190] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[191] Daniel Coquelin et al. “Evolutionary Optimization of Neural Architectures in Remote Sensing Classification Problems”. In: *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS*. IEEE. 2021, pp. 1587–1590. DOI: 10.1109/IGARSS47720.2021.9554309.

[192] Funk, Yannick and Götz, Markus and Anzt, Hartwig. “Prediction of Optimal Solvers for Sparse Linear Systems Using Deep Learning”. In: *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics. 2022, pp. 14–24. DOI: 10.1137/1.9781611977141.2.

[193] Gencer Sumbul et al. “BigEarthNet Dataset with a New Class-Nomenclature for Remote Sensing Image Understanding”. In: *arXiv preprint arXiv:2001.06372* (2020). DOI: 10.48550/arXiv.2001.06372.

[194] M Bossard, Jan Feranec, J Otahel, et al. *CORINE land cover technical guide: Addendum 2000*. Vol. 40. European Environment Agency Copenhagen, 2000.

[195] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[196] Kaiming He et al. “Identity Mappings in Deep Residual Networks”. In: *European Conference on Computer Vision*. Springer. 2016, pp. 630–645. DOI: 10.1007/978-3-319-46493-0_38.

[197] Mohammad Noroozi et al. “Golden search optimization algorithm”. In: *IEEE Access* 10 (2022), pp. 37515–37532.

[198] Benny Wong. “A new derivative-free optimization method: Gaussian Crunching Search”. In: *arXiv preprint arXiv:2307.14359* (2023).

[199] Tianyi Han et al. “A Region-Shrinking-Based Acceleration for Classification-Based Derivative-Free Optimization”. In: *arXiv preprint arXiv:2309.11036* (2023).

[200] Neeratyoy Mallik et al. “PriorBand: Practical Hyperparameter Optimization in the Age of Deep Learning”. In: *arXiv preprint arXiv:2306.12370* (2023).

[201] Danny Stoll, Frank Hutter, and Simon Schrodi. *Method and device for determining an optimal architecture of a neural network*. US Patent App. 18/184,379. 2023.

[202] Joe Mellor et al. “Neural Architecture Search without Training”. In: *International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Proceedings of Machine Learning Research. PMLR. PMLR, 2021, pp. 7588–7598. DOI: 10.48550/arXiv.2006.04647. URL: https://proceedings.mlr.press/v139/mellor21a.html.

[203] Kevin Rychel, Anand V Sastry, and Bernhard O Palsson. “Machine learning uncovers independently regulated modules in the Bacillus subtilis transcriptome”. In: *Nature communications* 11.1 (2020), p. 6338.

[204] Siddharth M Chauhan et al. “Machine learning uncovers a data-driven transcriptional regulatory network for the crenarchaeal thermoacidophile Sulfolobus acidocaldarius”. In: *Frontiers in Microbiology* 12 (2021), p. 753521.

[205] Aapo Hyvärinen. “Independent component analysis: recent advances”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 371.1984 (2013), p. 20110534.

[206] Zhongxin Bai and Xiao-Lei Zhang. “Speaker recognition based on deep learning: An overview”. In: *Neural Networks* 140 (2021), pp. 65–99.

[207] Mohammed AlQuraishi. “ProteinNet: a standardized data set for machine learning of protein structure”. In: *BMC bioinformatics* 20.1 (2019), pp. 1–10.

[208] Bert Hubert. *Proteinogenic amino acids*. CC BY-SA 4.0. URL: https://commons.wikimedia.org/wiki/File_talk:Amino_Acids.svg.

A. Appendix

A.1. Background

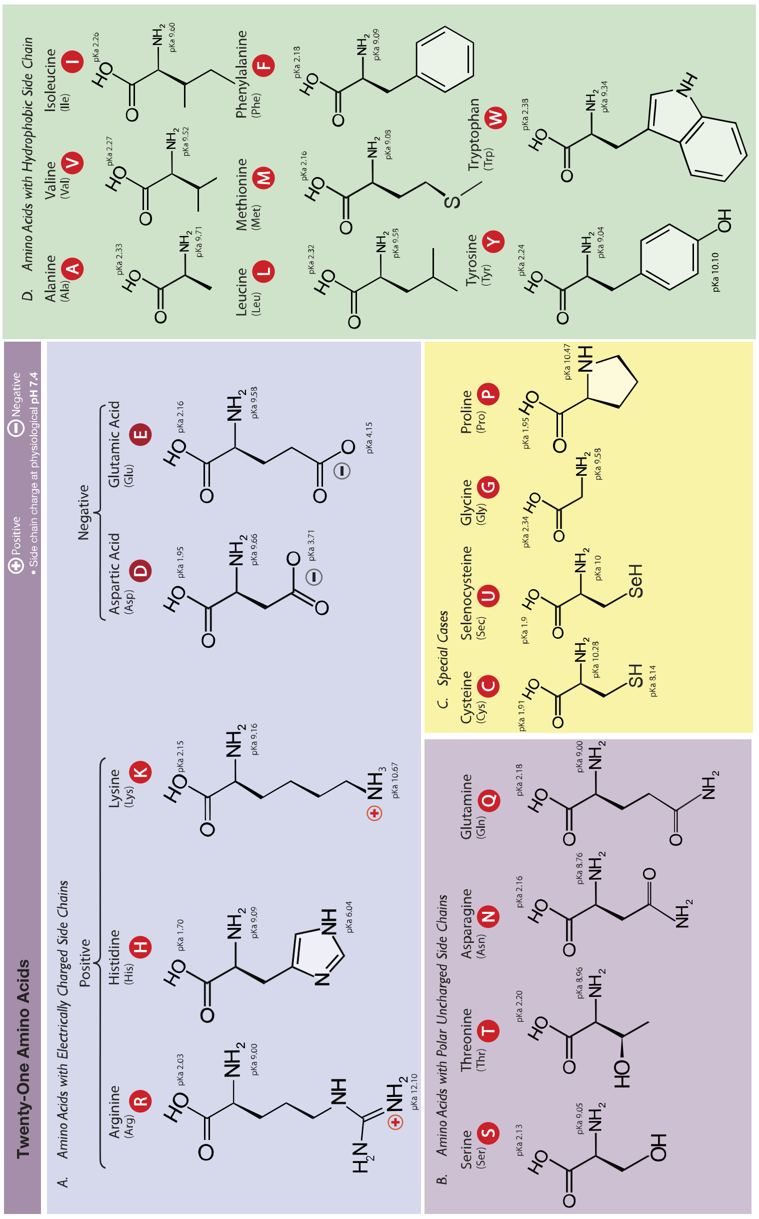


Figure A.1.: Protein building blocks by Bert Hubert [208]. Proteinogenic amino acids with their respective tokens in the alphabet.

A.2. RNA Contact Prediction

Table A.1.: Extended results for RNA contact prediction

contacthead frozen							
inpainting Loss	53.44	53.44	8.82	100.00	16.21	nan	3.84
inpainting PPV_L	53.44	53.44	8.82	100.00	16.21	nan	3.59
inpainting $\text{PPV}_{L;0.5}$	53.44	53.44	8.82	100.00	16.21	nan	4.50
inpainting F_1	47.33	47.33	9.05	97.62	16.57	3.59	3.57
inpainting MCC	36.16	36.16	10.31	59.92	17.59	5.37	4.74
jigsaw Loss	24.73	24.73	8.82	100.00	16.22	0.16	3.73
jigsaw PPV_L	22.93	22.93	8.44	77.57	15.22	-2.83	3.27
jigsaw $\text{PPV}_{L;0.5}$	22.93	22.93	8.44	77.57	15.22	-2.83	3.63
jigsaw F_1	23.57	23.57	8.82	100.00	16.22	0.16	3.59
jigsaw MCC	23.70	23.70	8.83	99.98	16.23	0.81	3.76
contrastive Loss	53.11	53.11	8.82	100.00	16.21	nan	3.59
contrastive PPV_L	53.18	53.18	8.82	100.00	16.21	nan	3.71
contrastive $\text{PPV}_{L;0.5}$	53.18	53.18	8.82	100.00	16.21	nan	3.79
contrastive F_1	46.31	46.31	8.93	98.55	16.38	2.36	3.65
contrastive MCC	15.16	15.16	8.68	50.96	14.84	-0.51	3.70
bootstrap Loss	65.45	65.45	8.82	100.00	16.21	nan	3.85
bootstrap PPV_L	65.38	65.38	8.82	100.00	16.21	nan	4.67
bootstrap $\text{PPV}_{L;0.5}$	65.38	65.38	8.82	100.00	16.21	nan	3.68
bootstrap F_1	53.76	53.76	8.95	99.12	16.42	2.92	4.06
bootstrap MCC	26.97	26.97	10.15	42.32	16.37	3.56	4.08
contacthead finetuned							
inpainting Loss	74.95	74.95	19.96	73.90	31.43	27.35	3.83
inpainting PPV_L	84.33	84.33	19.23	72.02	30.36	25.78	5.49
inpainting $\text{PPV}_{L;0.5}$	84.33	84.33	19.23	72.02	30.36	25.78	4.54
inpainting F_1	75.85	75.85	23.08	73.21	35.09	31.33	4.56
inpainting MCC	75.85	75.85	23.08	73.21	35.09	31.33	5.42
jigsaw Loss	61.34	61.34	16.50	75.10	27.05	22.17	4.28
jigsaw PPV_L	67.31	67.31	15.50	77.33	25.83	20.88	4.04
jigsaw $\text{PPV}_{L;0.5}$	67.31	67.31	15.50	77.33	25.83	20.88	4.93
jigsaw F_1	60.76	60.76	19.34	66.85	30.00	24.56	4.88
jigsaw MCC	60.76	60.76	19.34	66.85	30.00	24.56	5.41
contrastive Loss	72.00	72.00	21.85	75.38	33.88	30.39	3.05
contrastive PPV_L	84.01	84.01	18.84	71.18	29.79	24.97	3.80

Task + checkpoint metric(s)	$PPV_{L;0.5}$ / %	PPV_L / %	PPV / %	SEN / %	F_1 / %	MCC / %	Energy / Wh
contrastive $PPV_{L;0.5}$	84.01	84.01	18.84	71.18	29.79	24.97	3.68
contrastive F_1	67.50	67.50	26.35	67.90	37.96	33.52	3.65
contrastive MCC	68.85	68.85	26.15	68.82	37.90	33.59	3.80
bootstrap Loss	79.83	79.83	21.69	72.45	33.38	29.32	3.63
bootstrap PPV_L	81.89	81.89	18.52	71.38	29.41	24.54	3.72
bootstrap $PPV_{L;0.5}$	81.89	81.89	18.52	71.38	29.41	24.54	3.77
bootstrap F_1	69.49	69.49	30.49	61.95	40.87	35.71	3.74
bootstrap MCC	69.49	69.49	30.49	61.95	40.87	35.71	3.63
xgb frozen							
inpainting $PPV_{L;0.5}$	83.72	79.90	78.71	24.08	36.88	41.04	3.90
inpainting PPV_L	84.39	80.15	78.80	23.99	36.78	40.99	4.79
inpainting F_1	70.46	70.46	53.77	28.62	37.35	35.18	2.59
inpainting MCC	74.06	73.80	61.25	27.98	38.42	37.88	3.47
jigsaw $PPV_{L;0.5}$	87.08	43.93	87.08	4.78	9.07	19.26	3.81
jigsaw PPV_L	89.69	46.11	89.69	4.43	8.44	18.86	4.58
jigsaw F_1	26.01	26.01	18.35	16.90	17.60	9.99	3.63
jigsaw MCC	63.01	41.62	63.01	5.49	10.11	16.82	3.78
contrastive $PPV_{L;0.5}$	85.46	83.17	77.26	26.53	39.50	42.65	4.12
contrastive PPV_L	84.58	82.02	77.69	26.83	39.88	43.04	3.74
contrastive F_1	71.16	71.16	50.70	31.16	38.60	35.36	2.77
contrastive MCC	79.84	79.13	67.14	29.05	40.55	40.96	4.60
bootstrap $PPV_{L;0.5}$	87.07	81.44	79.90	23.68	36.53	41.06	3.78
bootstrap PPV_L	86.29	80.92	79.65	24.11	37.01	41.37	3.34
bootstrap F_1	73.43	73.09	56.12	28.54	37.84	36.14	3.17
bootstrap MCC	82.11	79.45	73.60	25.00	37.33	40.15	4.54
xgb finetuned							
inpainting Loss $PPV_{L;0.5}$	81.89	81.89	57.13	47.15	51.67	47.73	3.87
inpainting Loss PPV_L	80.80	80.80	54.94	47.08	50.70	46.50	4.57
inpainting Loss F_1	80.99	80.99	56.33	46.94	51.20	47.18	3.62
inpainting Loss MCC	81.63	81.63	57.22	47.23	51.75	47.82	3.67
inpainting PPV_L $PPV_{L;0.5}$	86.06	86.06	61.40	46.63	53.00	49.68	4.80

Task + checkpoint metric(s)	$\mathrm{PPV}_{L;0.5}$ / %	PPV_L / %	PPV / %	SEN / %	F_1 / %	MCC / %	Energy / Wh
inpainting PPV_L PPV_L	86.13	86.13	61.65	46.47	53.00	49.72	3.64
inpainting PPV_L F_1	84.65	84.65	57.20	47.49	51.90	47.96	3.75
inpainting PPV_L MCC	85.36	85.36	59.18	46.95	52.36	48.71	3.30
inpainting $\mathrm{PPV}_{L;0.5}$ $\mathrm{PPV}_{L;0.5}$	86.06	86.06	61.40	46.63	53.00	49.68	3.57
inpainting $\mathrm{PPV}_{L;0.5}$ PPV_L	86.13	86.13	61.65	46.47	53.00	49.72	3.64
inpainting $\mathrm{PPV}_{L;0.5}$ F_1	84.65	84.65	57.20	47.49	51.90	47.96	3.48
inpainting $\mathrm{PPV}_{L;0.5}$ MCC	85.36	85.36	59.18	46.95	52.36	48.71	3.65
inpainting F_1 $\mathrm{PPV}_{L;0.5}$	76.24	76.24	50.28	53.80	51.98	47.20	3.45
inpainting F_1 PPV_L	78.10	78.10	52.71	53.00	52.86	48.28	3.79
inpainting F_1 F_1	78.93	78.93	52.81	52.82	52.81	48.25	3.67
inpainting F_1 MCC	78.93	78.93	52.81	52.82	52.81	48.25	3.74
inpainting MCC $\mathrm{PPV}_{L;0.5}$	76.24	76.24	50.28	53.80	51.98	47.20	2.81
inpainting MCC PPV_L	78.10	78.10	52.71	53.00	52.86	48.28	3.79
inpainting MCC F_1	78.93	78.93	52.81	52.82	52.81	48.25	3.43
inpainting MCC MCC	78.93	78.93	52.81	52.82	52.81	48.25	3.73
jigsaw Loss $\mathrm{PPV}_{L;0.5}$	66.80	66.80	46.06	35.42	40.05	35.42	4.61
jigsaw Loss PPV_L	66.41	66.41	46.07	35.41	40.04	35.42	4.94
jigsaw Loss F_1	65.13	65.13	44.75	35.72	39.73	34.86	2.74
jigsaw Loss MCC	65.13	65.13	44.75	35.72	39.73	34.86	2.65
jigsaw PPV_L $\mathrm{PPV}_{L;0.5}$	68.46	68.46	46.20	41.47	43.71	38.65	4.56
jigsaw PPV_L PPV_L	69.11	69.11	45.74	41.57	43.55	38.43	4.91
jigsaw PPV_L F_1	68.91	68.91	45.90	41.58	43.63	38.53	2.78
jigsaw PPV_L MCC	68.91	68.91	45.90	41.58	43.63	38.53	3.64

Task + checkpoint metric(s)	$PPV_{L;0.5}$ / %	PPV_L / %	PPV / %	SEN / %	F_1 / %	MCC / %	Energy / Wh
jigsaw $PPV_{L;0.5}$ $PPV_{L;0.5}$	68.46	68.46	46.20	41.47	43.71	38.65	5.44
jigsaw $PPV_{L;0.5}$ PPV_L	69.11	69.11	45.74	41.57	43.55	38.43	4.50
jigsaw $PPV_{L;0.5}$ F_1	68.91	68.91	45.90	41.58	43.63	38.53	2.46
jigsaw $PPV_{L;0.5}$ MCC	68.91	68.91	45.90	41.58	43.63	38.53	2.75
jigsaw F_1 $PPV_{L;0.5}$	63.33	63.33	41.27	41.55	41.41	35.72	4.13
jigsaw F_1 PPV_L	62.75	62.75	40.91	41.73	41.32	35.58	4.60
jigsaw F_1 F_1	62.68	62.68	40.46	41.78	41.11	35.32	2.89
jigsaw F_1 MCC	62.68	62.68	40.46	41.78	41.11	35.32	2.79
jigsaw MCC $PPV_{L;0.5}$	63.33	63.33	41.27	41.55	41.41	35.72	4.07
jigsaw MCC PPV_L	62.75	62.75	40.91	41.73	41.32	35.58	4.71
jigsaw MCC F_1	62.68	62.68	40.46	41.78	41.11	35.32	3.62
jigsaw MCC MCC	62.68	62.68	40.46	41.78	41.11	35.32	3.82
contrastive Loss $PPV_{L;0.5}$	74.18	74.18	47.47	50.58	48.97	43.89	3.77
contrastive Loss PPV_L	74.18	74.18	47.47	50.58	48.97	43.89	3.65
contrastive Loss F_1	74.25	74.25	47.55	50.72	49.09	44.01	2.86
contrastive Loss MCC	74.25	74.25	47.55	50.72	49.09	44.01	3.73
contrastive PPV_L $PPV_{L;0.5}$	84.14	84.14	58.19	44.74	50.59	46.96	2.69
contrastive PPV_L PPV_L	84.71	84.71	59.83	43.91	50.65	47.32	3.62
contrastive PPV_L F_1	81.95	81.95	49.32	46.61	47.93	43.07	3.43
contrastive PPV_L MCC	84.14	84.14	58.19	44.74	50.59	46.96	3.67
contrastive $PPV_{L;0.5}$ $PPV_{L;0.5}$	84.14	84.14	58.19	44.74	50.59	46.96	3.63
contrastive $PPV_{L;0.5}$ PPV_L	84.71	84.71	59.83	43.91	50.65	47.32	3.85
contrastive $PPV_{L;0.5}$ F_1	81.95	81.95	49.32	46.61	47.93	43.07	3.68
contrastive $PPV_{L;0.5}$ MCC	84.14	84.14	58.19	44.74	50.59	46.96	3.04

Task + checkpoint metric(s)	PPV$_{L;0.5}$ / %	PPV$_L$ / %	PPV / %	SEN / %	F$_1$ / %	MCC / %	Energy / Wh
contrastive F$_1$ PPV$_{L;0.5}$	72.64	72.64	47.54	51.50	49.44	44.38	2.94
contrastive F$_1$ PPV$_L$	72.64	72.64	48.19	51.38	49.73	44.73	3.53
contrastive F$_1$ F$_1$	73.15	73.15	48.05	51.38	49.66	44.64	3.75
contrastive F$_1$ MCC	73.15	73.15	48.05	51.38	49.66	44.64	2.86
contrastive MCC PPV$_{L;0.5}$	75.59	75.59	47.49	52.51	49.88	44.83	2.97
contrastive MCC PPV$_L$	76.36	76.36	47.71	52.51	50.00	44.97	3.52
contrastive MCC F$_1$	75.72	75.72	47.45	52.49	49.85	44.79	4.54
contrastive MCC MCC	75.72	75.72	47.45	52.49	49.85	44.79	2.61
bootstrap Loss PPV$_{L;0.5}$	80.80	80.80	52.03	51.72	51.87	47.23	3.70
bootstrap Loss PPV$_L$	80.35	80.35	50.85	51.95	51.39	46.64	3.13
bootstrap Loss F$_1$	79.58	79.58	50.38	52.18	51.27	46.47	3.10
bootstrap Loss MCC	79.58	79.58	50.38	52.18	51.27	46.47	2.85
bootstrap PPV$_L$ PPV$_{L;0.5}$	84.01	84.01	59.26	43.42	50.12	46.75	3.44
bootstrap PPV$_L$ PPV$_L$	84.33	84.33	61.76	42.88	50.61	47.68	3.19
bootstrap PPV$_L$ F$_1$	83.69	83.69	58.69	43.63	50.05	46.58	2.71
bootstrap PPV$_L$ MCC	84.01	84.01	59.26	43.42	50.12	46.75	3.85
bootstrap PPV$_{L;0.5}$ PPV$_{L;0.5}$	84.01	84.01	59.26	43.42	50.12	46.75	3.71
bootstrap PPV$_{L;0.5}$ PPV$_L$	84.33	84.33	61.76	42.88	50.61	47.68	4.64
bootstrap PPV$_{L;0.5}$ F$_1$	83.69	83.69	58.69	43.63	50.05	46.58	2.69
bootstrap PPV$_{L;0.5}$ MCC	84.01	84.01	59.26	43.42	50.12	46.75	3.65
bootstrap F$_1$ PPV$_{L;0.5}$	74.69	74.69	49.47	48.51	48.99	44.11	3.72
bootstrap F$_1$ PPV$_L$	74.31	74.31	48.66	48.50	48.58	43.61	3.76
bootstrap F$_1$ F$_1$	74.69	74.69	49.47	48.51	48.99	44.11	3.66
bootstrap F$_1$ MCC	74.69	74.69	49.47	48.51	48.99	44.11	2.68

Task + checkpoint metric(s)	$\text{PPV}_{L;0.5}$ / %	PPV_L / %	PPV / %	SEN / %	F_1 / %	MCC / %	Energy / Wh
bootstrap MCC $\text{PPV}_{L;0.5}$	74.69	74.69	49.47	48.51	48.99	44.11	3.73
bootstrap MCC PPV_L	74.31	74.31	48.66	48.50	48.58	43.61	2.92
bootstrap MCC F_1	74.69	74.69	49.47	48.51	48.99	44.11	2.74
bootstrap MCC MCC	74.69	74.69	49.47	48.51	48.99	44.11	3.32

A.3. Loss Scheduling

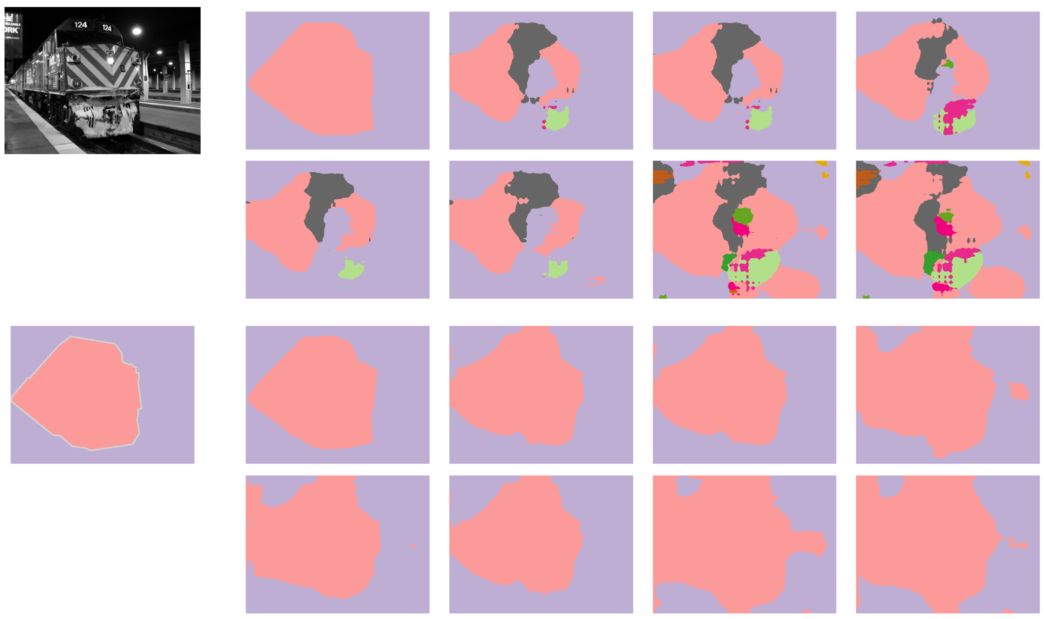


Figure A.2.: VOC images. The left-most column shows input image (top) and label (bottom). The rest are from left to right: pretrained, CE, WCE, DL, FL, CEMP, DLNWCE, DLAWCE. Top and bottom row show results from FCN and DeepLabV3 models, respectively.

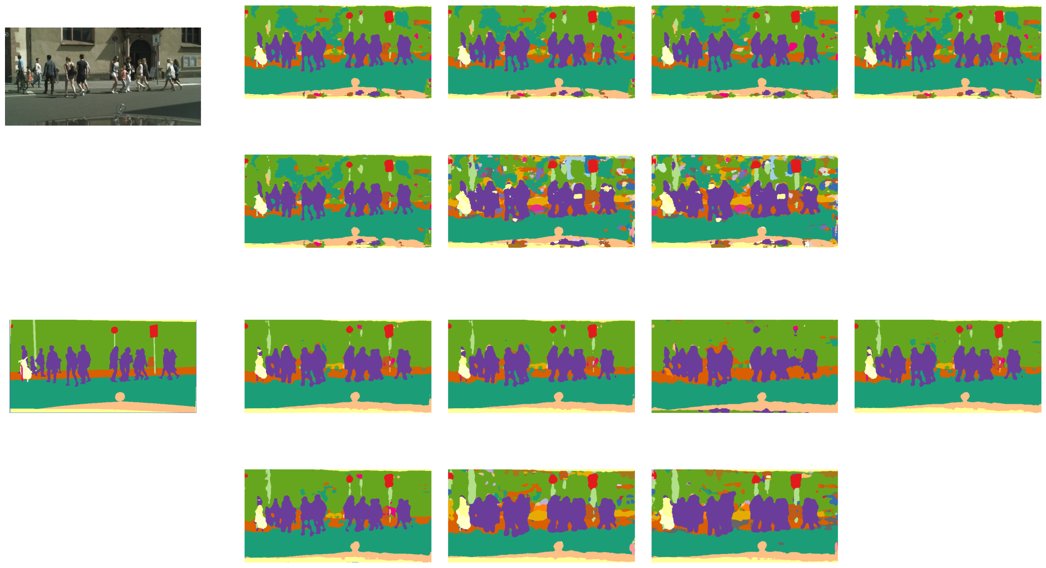


Figure A.3.: Cityscapes images. The left-most column shows input image (top) and label (bottom). The rest are from left to right: CE, WCE, DL, FL, CEMP, DLNWCE, DLAWCE. Top and bottom row show results from FCN and DeepLabV3 models, respectively. Since we did not have a larger dataset with the same annotated classes, there is no pretrained model.

A.4. Propulate

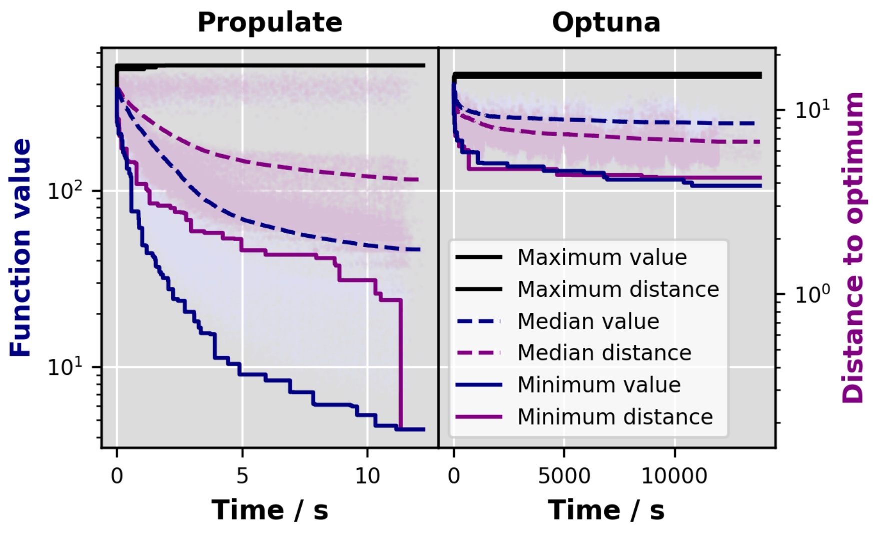


Figure A.4.: Evolution of the population over wallclock time. Propulate on the left versus `Optuna` on the right. Objective function values in blue use the left scale. Distances to the global optimum in purple use the right scale. Pastel dots show each individual candidate values. Solid (dashed) lines show the minimum (median) value achieved so far. Maximum function value and distance are shown in black. Both optimizers perform 38 912 evaluations.

List of Figures

List of Tables

List of Algorithms

Acronyms

F$_1$ F$_1$ score. 21, 36, 63, 108–113

ACC accuracy. 20, 63

ASA accessible surface area. 41, 42, 51, 121

CAL calibration score. 60, 63

CEMP max pooled cross entropy. 57, 63, 113, 114, 119

CE cross entropy loss. 16, 34, 56–58, 63, 113, 114, 119, 121

DCA direct coupling analysis. 10, 28–31, 40, 44, 45, 117

DI direct information. 11

DL dice loss. 63, 113, 114, 119, 121

ECE expected calibration error. 60

ELU exponential linear unit. 14

FL focal loss. 37, 57, 63, 113, 114, 119

FN false negative prediction. 20, 21, 31

FP false positive prediction. 20, 21, 31

GELU gaussian error linear unit. 14

IoU intersection over union. 57

MAE mean absolute error. 20

MCC Matthews correlation coefficient. 31, 36, 40, 44, 108–113, 117

MI mutual information. 9–11

MSA multiple sequence alignment. 9, 10, 27–34, 38–41, 51, 52, 117

MSE mean square error. 14, 20

PPV precision or positive predictive value. 21, 36, 40, 44, 108–113, 117

ReLU rectified linear unit. 13, 14, 59, 117

SEN	recall or sensitivity. 21, 63, 109–113
SSE	summed square error. 19
TN	true negative prediction. 20, 31
TP	true positive prediction. 20, 21, 31
WCE	class weighted cross entropy. 56, 63, 113, 114, 119
WDL	class weighted dice loss. 57, 63
AI	artificial intelligence. 60
CART	classification and regression tree. 19
CMA-ES	covariance matrix adaptation evolution strategy. 23
CNN	convolutional neural network. 17, 29
CPU	central processing unit. 25
DNA	deoxyribonucleic acid. 5, 7
GNN	geometric neural network. 17
GPU	graphics processing unit. 11, 25, 34, 38, 39, 60
HPC	high performance computing. 3, 25, 65, 66, 68, 85
HPO	hyperparameter optimization. 4, 21, 65–67, 71
ICA	independent component analysis. 90
LLM	large language model. 13
LSTM	long short term memory. 17, 24
MD	molecular dynamics. 8, 67
ML	machine learning. 11–13
MLP	multilayer perceptron. 13
MPI	message passing interface. 25, 67, 68, 70, 75, 85
NAS	neural architecture search. 4, 21, 24, 65, 67, 77
NCCL	NVidia collective communications library. 25

NMR	nuclear magnetic resonance. 8
PSO	particle swarm optimization. 21, 67
RCCL	ROCm communication collectives library. 25
RNA	ribonucleic acid. 3, 5–7, 10, 27–30, 40–42, 47, 49, 52, 117, 118
RNN	recurrent neural network. 17
SGD	stochastic gradient descent. 82
TPE	tree structured parzen estimator. 24
TPU	tensor processing units. 11