Making Rational Protein Design Artifically Intelligent

Richard Chen
rchen40@jhu.edu

Jimmy Kim
jkim551@jhu.edu

Stewy Slocum
sslocum3@jhu.edu

Jason Wong
jwong62@jhu.edu

Abstract

Computational Protein Design (CPD) could play an integral role in the next generation of medical treatments. CPD tries to, given a three-dimensional protein structure, find an amino acid sequence that will fold to that shape. Current methods include dead-end elimination followed by a branch and bound algorithm to search the amino acid solution space. We propose using a deep neural network (DNN) to learn a branching heuristic that will minimize the number of nodes searched by branch and bound. We will modify the existing branch and bound algorithm used by toulbar2, a cost function network solver, by training the DNN to imitate a branching heuristic we call *small branching*.

1 Introduction

Protein design holds promise to treat some of todays toughest diseases and to cure certain forms of cancer. Rational protein design makes protein-sequence predictions that will fold to specific structures. Because of the close relationship between structure and function, protein design offers a way to get from a desired behavior to a sequence of amino acids that can be used in treatment.

But predicting sequences that will fold to a specific structure is difficult. The number of possible protein sequences grows exponentially with the size of the protein chain. The task of computational protein design is to find a subset of sequences that will quickly and reliably fold to the desired shape, which can then be verified experimentally through archived results. Since proteins tend to fold into a conformation that minimizes the molecules free energy, protein design can be cast as an optimization problem that is the inverse of protein folding. Since accurately predicting protein-sequences is such a difficult and computationally expensive problem, an existing protein is often chosen to be modified

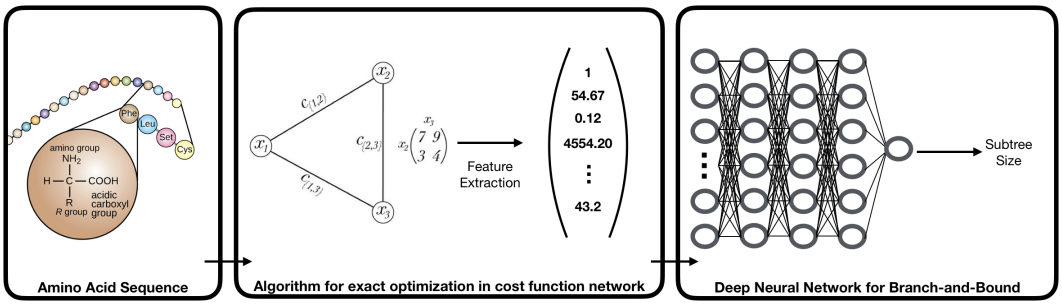

Figure 1: Pipeline from biological problem, to computational model, to training set for DNN which will then in turn be used to solve the computational model to inform the biological problem.

instead of designing a whole protein de novo. This process is called protein redesign and is far more tractable [10].

2 Related Work

2.1 Protein Design

The problem of scoring and ranking amino acid sequences by how well they fold into the target structure of a protein is a NP-hard discrete optimization problem that grows exponentially with the size of the protein sequence, with many approaches being developed. One of the first combinatorial optimization algorithms developed was by Leach and Lemon 1998, who used the Branch-and-Bound algorithm to explore the conformational energy surface of protein side chains, in which the problem is described as searching for a single path through through the branches in a combinatorial tree that corresponds to the GMEC (global minimum energy conformation) set of rotamers. In this early work, the Branch-and-Bound algorithm is used to simultaneously prune the tree while searching, with each branch tested with a quantitative bounding expression [8]. Gordon and Mayo 1998 describes a bounding function that maximizes the efficiency of pruning for problems in which the total energy can be decomposed into interactions between rotamers, which they called termination, and how the energetic information produced by termination can be used to determine to the optimal search order for the remainder of the tree [9]. Allouche et al. 2010 presented toulbar2, an exact solver for cost function networks (CFNs) a valuable formulation of the optimization problem of protein redesign. In order to select variables, toulbar2 relies on both a weighted constraint heuristic, and on last conflict reasoning [12]. Moreover, Allouche et al. 2014 tested toulbar2's performance on 35 protein design problems, formulated as cost function networks, and found toulbar2 to significantly outperform contemporary protein design software [4].

2.2 Deep Learning

In the past decade, deep learning has revolutionized the fields of computer vision, machine learning and automation, achieving remarkable performance on previously-difficult tasks such as image classification, semantic segmentation and machine translation. More recently, they have been extended to solve more complex problems such as drug discovery and protein folding. In the seminal work by DeepMind, Evans et al. 2019 were able to use deep networks to predict 3D structure of a protein based solely on its genetic sequence. Following the works by Evans, AlQuraishi et al. 2019 intro-

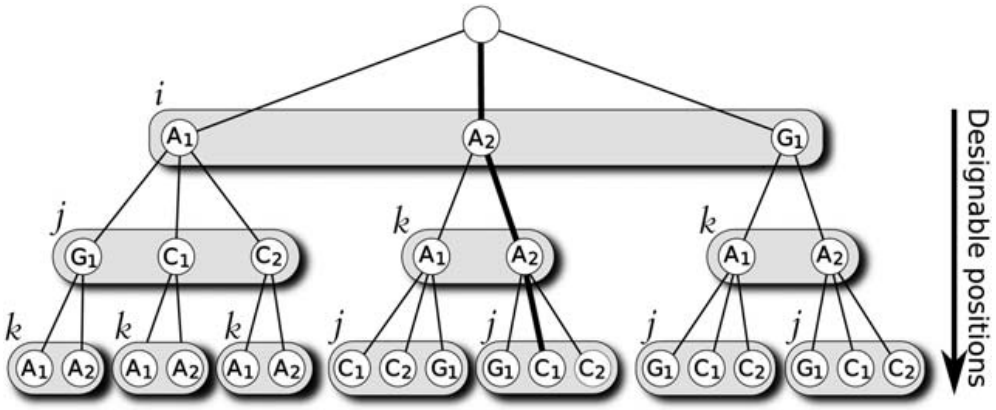

Figure 2: CPD search as a branch and bound tree.

duced an end-to-end differentiable model for protein structure learning that did not rely on using co-evolution [11].

3 Branch-and-Bound Problem Formulation in Protein Design

3.1 Protein Design as a WCSP

Definition 3.1. Weighted constraint satisfaction problems: A weighted constraint satisfaction problem (WCSP) can be expressed as a tuple (X, D, F), where $D = \{D_1, ..., D_n\}$ is a set of finite domains, and $X = \{x_1, ...x_n\}$ is a set of discrete variables in which for X_i, every value $X_i = i_r \in D_i$. F is a set of cost functions that can be used to indicate preferences between solutions to the constraint problem.

For the sake of our problem, each function $f \in F$ may map either $f : D_i \to [0, k]$ or $f : D_i \times D_j \to [0, k]$, where k is an integer value representing infinite cost, for illegal value assignments. Our problem then is to find an assignment of variables for which the k-bounded sum of all our cost functions is minimized.

To frame our protein design problem as a WCSP, we say that $\forall x_i \in X, x_i$ represents the ith amino acid in the amino acid chain, and D_i is the set of all possible amino acid-rotamer pairs that can occupy the ith spot in the chain. As for our set F of cost functions, a function $f : D_i \to [0, k]$ represents a unary energy function, i.e. the energy that a single amino acid-rotamer pair contributes to the total internal energy of the protein. Similarly, a function $f : D_i \times D_j \to [0, k]$ represents a binary energy function, i.e. the energy that the interaction between two amino acid-rotamer pairs adds to that of the protein. Our problem of finding the minimum energy configuration of amino acids given a backbone structure translates to minimizing the sum of the energy functions.

3.2 Branch-and-bound

Definition 3.2. Integer Linear Programming: An integer linear programming problem can be expressed as minimizing the objective function $\mathbf{c}^T\mathbf{x}$, subject to:

$$\mathbf{Ax} \leq \mathbf{b}$$

$$\mathbf{x} \geq 0$$
$$\text{and } \mathbf{x} \in \mathbb{Z}^n$$

Branch and bound algorithm is commonly used to solve combinatorial optimization problems, like ILP, and by toulbar2 to solve WCSPs. For each node in the tree, we branch on a variable X_i to generate two child nodes. One child node represents the assignment $X_i = a_i, a_i \in D_i$, while the other represents the restriction $X_i \in D_i - \{a_i\}$. Lower bounds on the objective function are stored and modified by the algorithm to inform the search as to which nodes are promising to search. Higher lower bounds to our objective function's minimum are understood to promise better searches, since this corresponds to more possible minimum values being ruled out.

Here we introduce the strong branching heuristic. At each branching decision, the restriction of the domain D_i alters the lower bound of our objective function. So for each variable-value pair, new lower bounds on this objective function are computed, one case in which $X_i = a_i$, and the other in which $X_i \neq a_i$. The changes in lower bound from when X_i was unassigned are then computed, and are used to compute a strong branching score, with higher scores being affiliated with greater increases in lower bound. Each variable-value pair has a strong branching score associated with it, and the pair with the highest score is the one chosen to branch.

It is worth noting that both before and while toulbar2 is solving the WCSP, the search is cut off altogether at certain nodes, eliminating a massive number of nodes from our search. A node is pruned from the tree when the least possible energy of an amino acid sequence somewhere down the search tree from the current node is known to be greater than a known minimum that we have already encountered. Even before the search begins, this logic can be used to significantly decrease our search space. Allouche et al. 2014 uses the technique of removing rotamer r at position i (denoted by i_r) if there exists another rotamer u at the same position such that:

$$E(i_r) - E(i_u) + \sum_{j \neq i} \min_s [E(i_r, j_s) - E(i_u, j_s)] \geq 0,$$

where E represents the unary or binary energy function associated with the rotamers passed as arguments [3].

4 A Machine Learning Approximation of Small Branching

4.0.1 Existing Branching Heuristics for WCSPs

While toulbar2 and ILP solvers like CPLEX both use branch and bound to search a solution space, the way in which they do so is quite different. CPLEX uses a best-first search, meaning that apart from occasional probes into the tree, the solver chooses to expand the node with the best (lowest) lower bound. This allows CPLEX to spend more time searching promising parts of the search tree. Meanwhile, toulbar2 uses a depth first search, continuing until it either finds a solution, or a conflict (dead end). Toulbar2 uses a depth first search to keep memory usage low as it has the capacity to explore more much larger branch and bound trees than CPLEX. Toulbar2 can expand search tree nodes several orders of magnitude more quickly than CPLEX because rather than solving an entire LP (a costly operation), it simply enforces local consistency (relatively cheap operation) [4].

CPLEX's best-first node selection policy can be paired with strong branching so that the best lower bound is not only explored, but then raised as much as possible during the branching decision. Raising lower bounds means moving closer to estimating the optimal solution, helping best-first search to spend more time where the optimal solution might be. Meanwhile, because toulbar2 is a

depth first search, once it enters a node, it cannot explore any other part of the search tree until it searches through that node's entire subtree. Rather than raise lower bounds, toulbar2's branching heuristics (last conflict and weighted degree) try to make branching decisions that will result in as many dead ends as possible [4]. Both of these heuristics to branch on variables that have resulted in dead ends in the past [5, 6]. Toulbar2 uses a separate heuristic to select a branching value a for x_i based on unary costs that also aims to cause dead ends [4].

Continuing to compare ILP and WCSP branch and bound methods, it is evident that the weighted degree heuristic is analogous to pseudocost branching in ILP. Both are efficient heuristics that have some sort of cost or counter associated with each variable, and as the program runs, these costs and counters are updated to reflect how effective each variable was in the past as a branching choice. For pseudocosts, the metric is related to the lower bound increase, for weighted degree, the metric keeps track of how many dead ends the variable has been involved in. Pseudocost branching approximates strong branching. So then what does weighted degree approximate?

4.1 Small Branching

We propose *small branching* as a theoretical analog to strong branching for WCSP. In small branching, we would test every possible variable-value pair to branch on, compute a small branching score, and choose to branch on the variable-value combination that has the best (lowest) score. The small branching score is the size of the subtree created as a result of the branching choice.

Weighted degree estimates the probability of dead ends in the search tree, which is in turn an estimate of how much of the search tree will be pruned off. Without any pruning, all subtrees would be of the same size. Weighted degree reasons using past conflicts, aiming to choose a branching variable that will minimize the size of the subtree to be explored (since the entire subtree must be explored before another part of the tree can be searched).

But in order to find the size of the subtree as a result of a branching decision, we have to visit every node in the tree, which is equivalent to searching through that portion of the solution space. Therefore, it is unreasonable to compute an exact strong branching score, but it would make sense to find a way to effectively approximate it.

4.2 Learning Branching Decisions

Machine learning is one approach to approximate a small branching score. Machine learning has been used to approximate strong branching scores for ILP, efficiently imitating the decisions that strong branching takes [7]. We used a similar approach to Alvarez et. al., handcrafting a set of features associated with a branching decision and node, and training a model to compute a small branching score given this set of features.

4.3 Feature Extraction

We divided the features into three subsets, static, local dynamic, and global dynamic features. Static features include the number of variables, binary costs, and number of constraints across all variables. Local dynamic features include the current variable's domain size, degree, weighted degree, unary cost, and binary costs. Global dynamic features include number of unassigned variables, product of the domains of all unassigned variables, unary cost of unassigned variables, and the global upper and

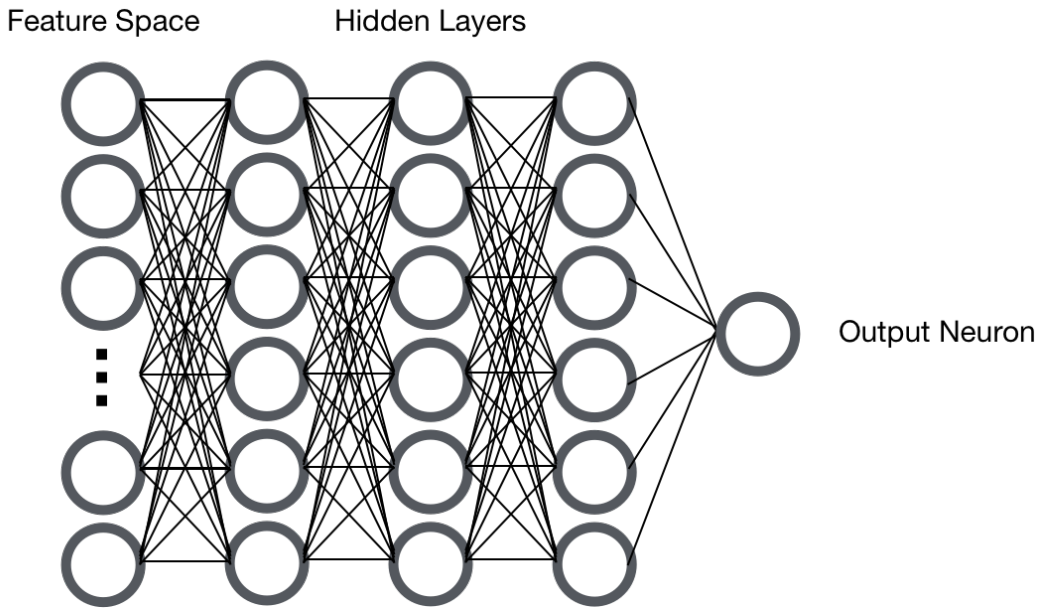

Figure 3: Multilayer Perceptron with 3 hidden layers.

lower bounds. For each quantity for which there were many values, we instead included statistics (mean, median, standard deviation, min, max, first quartile, and third quartile).

5 Deep Learning for Protein Design

In this section, we outline the model architecture for our deep network, how it was designed for subtree size regression, and the training procedure used.

5.1 Multilayer Perceptron Definition and Model Architecture

A multilayer perceptron (MLP) is a feedforward network consisting of three types of layers - an input layer, hidden layer(s), and output layer - in which information is forward propagated through sequentially from the input layer, to each hidden layer and then the output layer. Compared to single layer perceptrons, a which is a linear classifier that tries to find a hyperplane to best linearly separate data, a multilayer perceptron (MLP) tries to learn a representation of the input (that usually isn't immediately separable), that when projected into a different space through various (non-linear) transformations, would then become linearly separable. Feedforward neural networks are described as a directed acyclic graph, $G = (V, E)$, and a weight function over the edges, $w : E \to \mathbb{R}$, and every node in G correspond to neurons. Each neuron receives as input a weighted sum of outputs of the neurons connected to its incoming edges (usually from neurons in the previous layer), followed by a scalar "activation" function $\sigma : \mathbb{R} \to \mathbb{R}$.

In this work, we adopted a 3-layer multilayer perceptron, with each hidden layer having a size of 6 neurons. Because the input feature space as small, we chose to limit the size of the hidden layers to be small. Each hidden layer is activated by a ReLU function, followed by a linear activation function on the 3rd layer. Mathematically, we can write out the forward-propagation calculation for performing subtree regression as:

$$\text{Subtree Size} = \sum_{l=1}^{6} w_{lm}^{(4)} \left(\sum_{k=1}^{6} w_{kl}^{(3)} \sigma \left(\sum_{j=1}^{6} w_{jk}^{(2)} \sigma (\sum_{i=1}^{6} w_{ij}^{(1)} x_i + b^{(1)}) + b^{(2)} \right) + b^{(3)} \right) + b^{(4)}$$

Using linear algebra, we can generalize the above expression as:

$$\text{Subtree Size} = W^{(4)} \left(W^{(3)} \left(W^{(2)} (W^{(1)} X + b^{(1)}) + b^{(2)} \right) + b^{(3)} \right) + b^{(4)}$$

5.2 Training Details

We trained our network for 10,000 epochs using the Adam optimizer with a learning rate of 0.01and a mini-batch size of 128.

6 Result, Discussion, and Challenges

6.1 Experimental Setup

To generate the training set, we weigh each variable based on the depth of the node. If the depth from the root node is higher, we give it a smaller probability of being added to the set. The probability decreases in half each layer we go down ($P_{selected} = \frac{1}{2^d}$ where d is depth). The reason is that we value decisions at the top of the tree more, as they are more crucial to reducing the overall size of the problem. However, only training on nodes in the top of the tree will reduce the amount of training data we have, so we settled for the middle. After if we want to add a certain variable to the dataset, we choose a random variable-value pair to branch on and measure how well the value performed based on the size of the subtree it generated.

We generated data from five different CPD instances based on the proteins 1BRS, 1CDL, 1DKT, 1ENH, and 2DRI, chosen for the size of the search trees that they would generate (upwards of a thousand nodes). Using the selection criteria above, we obtained a data set for each problem ranging averaging several hundred nodes. We used four of the datasets for training and one for testing. Root mean squared error (RMSE) was used to measure the accuracy of our proposed network for subtree size regression.

6.2 Results

In our results, we achieved a training RMSE of 107261.323, and a testing RMSE of 4796.269. In monitoring the training performance across 10000 epochs, we noticed no substantial drop in RMSE loss on either the training set or the testing set. With our observation, we conclude that our proposed methodology for doing subtree size regression for branch-and-bound is not yet sufficient for protein design. Further work needs to be done to explore and design a tractable approach.

6.3 Challenges and Future Direction

In this work, we propose a deep learning approach to speed up the WCSP approach to protein design, learns to predict subtree size to inform branching decisions. In order to improve results, it could be beneficial to tweak existing features, and investigate entirely new ones to describe the state of the problem and quality of the branching choice. Because deep networks are completely data-driven

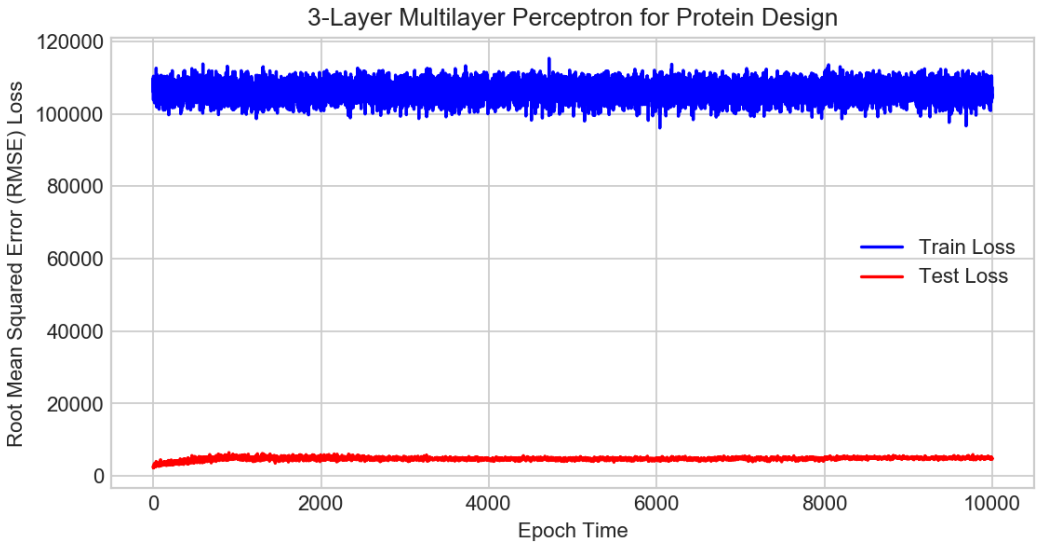

Figure 4: Training performance

and our training performance did not improve, it could be that not enough data was generated to to find a pattern for branch-and-bound. In validating this methodology, our team ran into substantial problems generating the full set of problems in a feasible time, which was why the training set was limited. Future work would be generating more annotated data for subtree regression, then testing and validating whether the features extracted and the model architecture is able to improve training performance.

References

[1] Rumelhart D., Hinton G.E., Williams R.J., (1986) GLearning representations by back-propagating errors. *Nature*

[2] Cortes, Mohri, & Rostamizadeh (2010) Generalization Bounds for Learning Kernels, *arXiv*

[3] Allouche et. al. (2014) Computational protein design as an optimization problem, *Artificial Intelligence*

[4] F. Boussemart, F. Hemery, C. Lecoutre, L. Sais (2004) Boosting systematic search by weighting constraints, *ECAI 2004*

[5] C. Lecoutre, L. Sas, S. Tabary, V. Vidal (2009) Reasoning From Last conflict(s) in Constraint Programming, *Articial Intelligence*

[6] T. Achterberg, T. Koch, A. Martin (2005) Branching Rules Revisited, *Operations Research Letters*

[7] Alvarez, Louveaux, Wehenkel (2017) A Machine Learning-Based Approximation of Strong Branching, *INFORMS Journal on Computing*

[8] Leach,A.R. and Lemon,A.P. (1998) *Proteins*, 33, 227239.

[9] Gordon and Mayo (1999) Branch-and-Terminate: a combinatorial optimization algorithm for protein design. *Structure*

[10] Wikipedia contributors. (2019, March 5). Protein design. In Wikipedia, The Free Encyclopedia. Retrieved 05:53, April 4, 2019.

[11] AlQuraishi (2019) End-to-End Differentiable Learning of Protein Structure, *Cell Systems*

[12] Allouche et al. (2010) Toulbar2, an open source exact cost function network solver. *The Rachel and Selim Benin School of Computer Science and Engineering*