COS551 Intro. to Computational Biology	**11/16/2000**
Lecturer: Mona Singh	*Scribe: Carl Kingsford*

1 Secondary structure prediction

Given a protein sequence with amino acids $a_1 a_2 \ldots a_n$, the secondary structure prediction problem is to predict whether each amino acid a_i is in an $\alpha-$helix, a $\beta-$sheet, or neither.[1] If you know (say through structural studies), the actual secondary structure for each amino acid, then the *3-state* accuracy is the percent of residues for which your prediction matches reality. It is called "3-state" because each residue can be in one of 3 "states": α, β, or other (O). Because there are only 3 states, random guessing would yield a 3-state accuracy of about 33% assuming that all structures are equally likely.

As an example, if the correct secondary structure of a protein is $\alpha\alpha O\beta$, but we predict $\alpha\beta\beta\beta$, then the 3-state accuracy is 50%.

Note that the 3-state accuracy measure does not convey many useful types of information–for example, it doesn't say where the errors are, or in what way the algorithm failed. (Do you over predict α or β structures? Do you make more errors at residues along the boundaries of secondary structure units?). Nevertheless, it is a concise, useful measure that is commonly used to compare the performance of different methods.

Early secondary structure prediction methods (such as Chou-Fasman and GOR, outlined below) had a 3-state accuracy of 50–60%. (They initially reported higher accuracy, but this was found to be inflated once they were tested against proteins outside of the training set.) Today's methods have an accuracy of $> 70\%$.

We briefly describe several older methods, which are quite intuitive and natural, before moving on to one of the most successful recent methods, PhD, which is based on neural nets.

[1] It is possible to have more detailed descriptions of secondary structure (e.g., different types of helices, turns, etc.) but for simplicity, most secondary structure prediction algorithms just characterize structures that are neither helices or strands as either "other" or "random coil.")

1.1 The Chou-Fasman method

If you were asked to determine whether an amino acid in a protein of interest is part of a α-helix or β-sheet, you might think to look in a protein database and see which secondary structures amino acids in similar contexts belonged to. The **Chou-Fasman** method (1978) is a combination of such statistics-based methods and rule-based methods. Here are the steps of the Chou-Fasman algorithm:

1. Calculate propensities from a set of solved structures. For all 20 amino acids i, calculate these propensities by:

   $$\frac{Pr[i|\beta\text{-sheet}]}{Pr[i]} \quad \frac{Pr[i|\alpha\text{-helix}]}{Pr[i]} \quad \frac{Pr[i|\text{other}]}{Pr[i]}$$

   That is, we determine the probability that amino acid i is in each structure, normalized by the background probability that i occurs at all. For example, let's say that there are 20,000 amino acids in the database, of which 2000 are serine, and there are 5000 amino acids in helical conformation, of which 500 are serine. Then the helical propensity for serine is: $\frac{\frac{500}{5000}}{\frac{2000}{20,000}} = 1.0$. Note that often you will see propensities defined as:

   $$\frac{Pr[\beta\text{-sheet}|i]}{Pr[\beta\text{-sheet}]} \quad \frac{Pr[\alpha\text{-helix}|i]}{Pr[\alpha\text{-helix}]} \quad \frac{Pr[\text{other}|i]}{Pr[\text{other}]}$$

   and that these two formulations are equivalent.

2. Once the propensities are calculated, each amino acid is categorized using the propensities as one of: helix-former, helix-breaker, or helix-indifferent. (That is, helix-formers have high helical propensities, helix-breakers have low helical propensities, and helix-indifferents have intermediate propensities.) Each amino acid is also categorized as one of: sheet-former, sheet-breaker, or sheet-indifferent. For example, it was found (as expected) that glycine and prolines are helix-breakers.

3. When a sequence is input, find **nucleation sites**. These are short subsequences with a high-concentration of helix-formers (or sheet-formers). These sites are found with some heuristic rule (e.g. "a sequence of 6 amino acids with at least 4 helix-formers, and no helix-breakers").

4. Extend the nucleation sites, adding residues at the ends, maintaining an average propensity greater than some threshold.

5. Step 4 may create overlaps; finally, we deal with these overlaps using some heuristic rules.

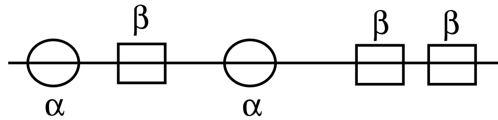


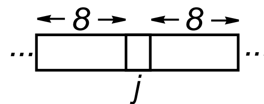
Figure 1: In the Chou-Fasman method, nucleation sites are found along the protein using a heuristic rule, and then extended.

1.2 The GOR method

To determine the structure for a given amino acid position j, the GOR method (named for the authors Garnier, Osguthorpe, Robson) looks at a window of 8 amino acids before and 8 after the position of interest. Suppose a_j is the amino acid that we are trying to categorize. GOR looks at the residues $a_{j-8}a_{j-7}\dots a_j\dots a_{j+7}a_{j+8}$. Intuitively, it assigns a structure based on probabilities it has calculated from protein databases. These probabilities are of the form

$$Pr[\text{amino acid } j \text{ is } \alpha | a_{j-8}, \dots, a_j, \dots, a_{j+8}]$$
$$Pr[\text{amino acid } j \text{ is } \beta | a_{j-8}, \dots, a_j, \dots, a_{j+8}]$$

That is, each corresponds to the probability that an amino acid has a particular structure, given the sequence around it. The GOR method thus looks at a window of 17 amino acids:



There are far too many possible sequences of length 17 to make calculating the above probabilities feasible. Instead, it is assumed that these probabilities can be estimated using just pairwise probabilities. We omit details but the overall idea is similar to the log-odds ratios we have studied before, except that pairwise dependencies are considered.

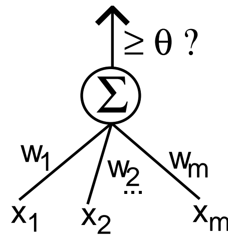


Figure 2: A perceptron.

2 Introduction to Neural Networks

The most successful methods for predicting secondary structure are based on neural nets. The overall idea is that neural nets are *trained* to be able to recognize amino acid patterns in known secondary structure units (e.g., helices), and to use these patterns to distinguish between the different types of secondary structures. Before considering one of these modern secondary structure prediction methods, we introduce the ideas behind neural networks. Neural networks classify input vectors or examples into two or more categories. They are loosely based on biological neurons.

2.1 Perceptrons

Perceptrons, also called **threshold units**, are a simple method for classifying input vectors, or examples, into one of two categories. They function similarly to one-layer neural networks — in fact, we will see that full neural networks are essentially built from many threshold units.

Think of a perceptron as a node in a directed graph with m input edges and one output edge. See figure 2. The inputs are $x_1, \ldots, x_m$, and each of the input edges has a weight, $w_1, \ldots, w_m$. Each input is multiplied by the corresponding weight of its edge. Then, the perceptron sums all these weighted inputs. If the result is greater than some threshold, a $+1$ is placed on the output edge. If the result is less than the threshold, -1 is put on the output edge.

More formally, if θ is the threshold, then the perceptron computes $A = \sum_1^m w_i x_i$ and outputs $+1$ if $A \geq \theta$, -1 otherwise. In vector notation, the input can be represented by a vector $\vec{x}$ of length m, and the weights correspond to a vector $\vec{w}$ of length m, and

$A = \vec{w} \cdot \vec{x}$. Thus, the perceptron categorizes the input $\vec{x}$ as belonging to one category $(+1)$ or the other (-1). Note that if $m = 1$, a perceptron just corresponds to a line, and checks whether input points lie above or below the line; in higher dimensions, a perceptron corresponds to a hyperplane.

If the weights are not known in advance, the perceptron must be **trained**. For that we need a **training set**: a set S of input vectors for which we know the desired (target) answer. Ideally, the goal of training is to find a set of weights such that the perceptron returns the correct answer for all of the training examples, with the hope that such a perceptron will have good performance on examples it has never seen. The training set should contain both positive and negative examples. For example, if we were to build a perceptron to recognize α-helices, then we should have sequences that are part of α-helices (positive examples), as well as sequences that are not (negative examples).

Perceptron training procedure

If there is a perceptron that can return the correct answer for *all* training examples, then a very simple rule (due to Rosenblatt) can be used to find one such setting of the weights. We can train the perceptron to output 1 for the positive examples, and -1 for the negative examples using the following procedure:

1. Run input vector $\vec{x}$ through the perceptron with the current weights $\vec{w}$; that is compute $\vec{w} \cdot \vec{x}$, and use the threshold to categorize it as positive or negative.
2. If the perceptron is wrong then update the weights with the rule:
$$\vec{w}' \leftarrow \begin{cases} \vec{w} - \vec{x} & \text{if false positive (i.e., } \vec{x} \text{ should be negative but } +1 \text{ is output)} \\ \vec{w} + \vec{x} & \text{if false negative (i.e., } \vec{x} \text{ should be positive but -1 is output)} \end{cases}$$
3. Repeat until the perceptron correctly classifies all the training examples.

This rule is sensible because the weights are moving in the right direction. Suppose, for example, the perceptron returned a false positive. Then after the training rule, if we supply the same input, the perceptron returns $(\vec{w} - \vec{x}) \cdot \vec{x} = \vec{w} \cdot \vec{x} - \vec{x} \cdot \vec{x}$. Since $\vec{x} \cdot \vec{x}$ is positive, the weighted sum is reduced, making it more likely that it is below the threshold.

Theorem 1 *If all examples have unit length and a set of weights exists that labels all examples correctly (in such a way that for each example, its weighted sum is not equal to the threshold θ), then the perceptron training procedure converges.*

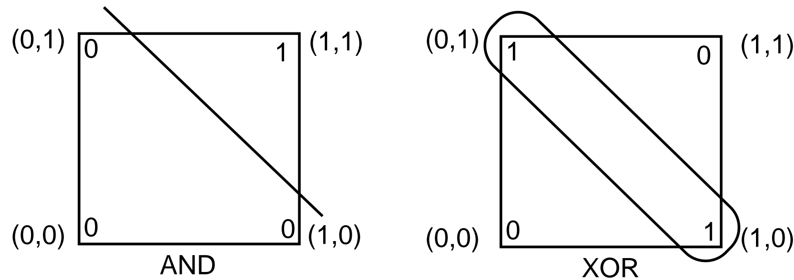


Figure 3: Each example corresponds to a point, and positive examples are labelled with "1" and negative examples are labelled with "0." AND is a linearly separable function (i.e., you can draw a line to separate the positive examples from the negative ones), but XOR is not (i.e., no such line exists). Given these inputs, a perceptron will learn to produce the correct output for AND, but cannot learn XOR.

Interested readers can find the (simple) proof to this theorem in the Appendix.

Continuous threshold units

But there is no guarantee that such a set of weights exists. In particular, they will not exist if there is no set of weights that can correctly classify all positive and negative examples in the training set. Since a threshold unit is computing a linear function, this means that the training data is not linearly separable. Linearly inseparable data sets are easy to find. The exclusive OR function (XOR) is linearly inseparable. See figure 3.

Rather than fitting the examples exactly, we can try to find a set of weights that minimizes some error function. Normally, to minimize a function we find where the derivative is 0, but we cannot compute the derivative of the output of a perceptron as defined above because the output is not differentiable everywhere (it's a step function). So we modify the perceptron to use a continuous, differentiable threshold function σ. An often used σ is:

$$\sigma(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

where the derivative of $\tanh(u) = \text{sech}^2(u)u' = 4u'/(e^u + e^{-u})^2$. The perceptron computes $\sigma(\vec{w} \cdot \vec{x})$. Figure 4 shows a graph of the tanh function and its derivative. You can see that tanh looks like the discrete threshold function, except it is smooth.

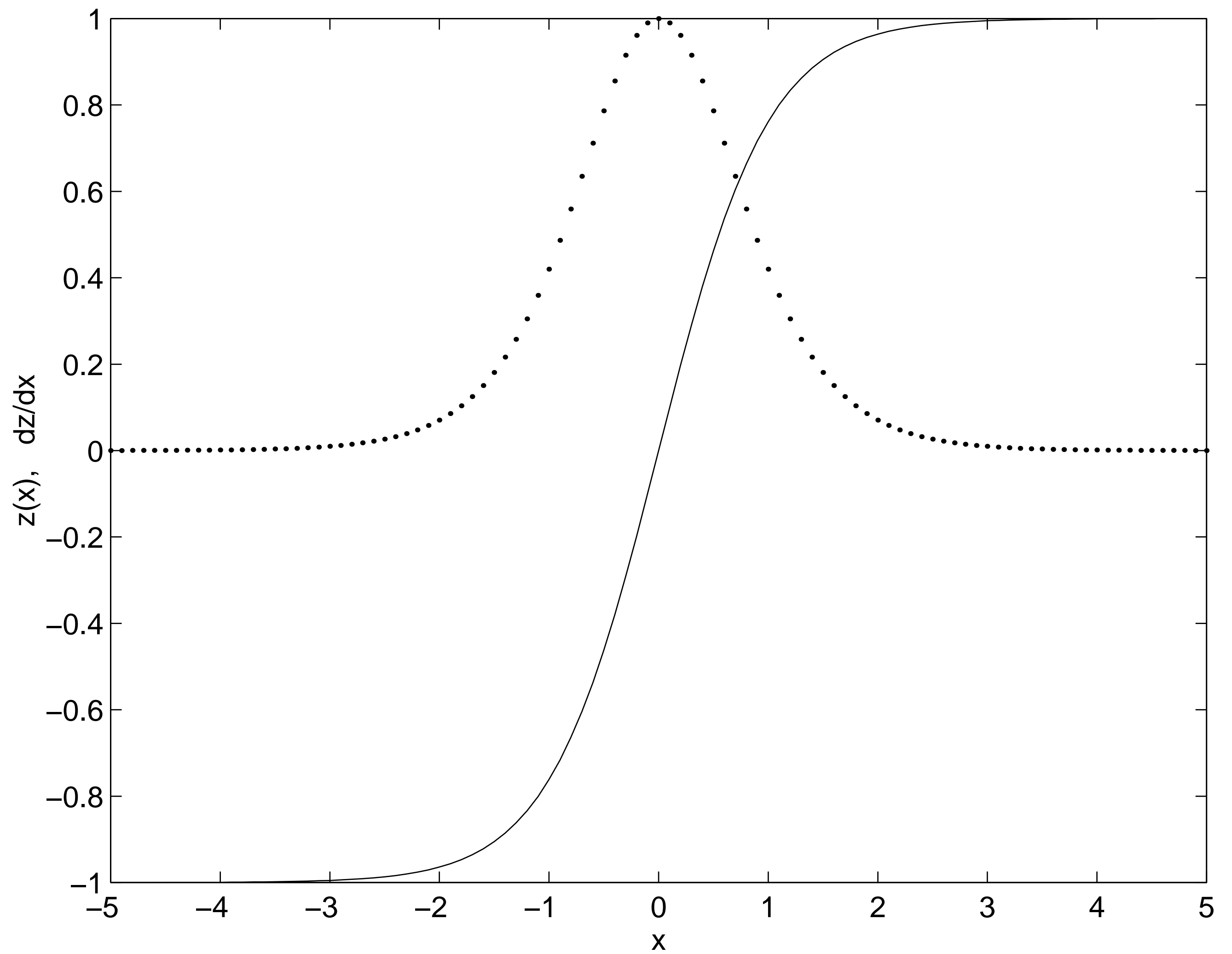


Figure 4: The $\tanh(x)$ function is shown by the solid line, and its derivative is shown by the dotted line.

2.2 Squared error

We need a notion of how close the actual output of a threshold unit (or neural net) is to the target output. The squared error is one such measure. Let S be the set of training examples. Let $t(\vec{x})$ be the target value of example $\vec{x} \in S$. Then the **squared error** is:

$$E(\vec{w}) = \sum_{\vec{x} \in S} \frac{1}{2}(\sigma(\vec{w} \cdot \vec{x}) - t(\vec{x}))^2$$

This measure is good because it is differentiable if σ is differentiable, though in practice "outliers" have a large effect on it.

The goal of training now is to find a $\vec{w}$ that minimizes $E(\vec{w})$.

2.3 Gradient Descent

To minimize squared error, we start with an initial weight vector $\vec{w}_0$ and take small steps in the direction which reduces the error the most. This process is called **gradient descent**. The idea (if we were maximizing instead of minimizing) is that if you were standing in a valley and wanted to get to the highest point, but couldn't see the tops of the mountains around you, you might decide to climb the steepest mountain. Because of this analogy, the method is sometimes called **hill climbing**.

The gradient of a (multivariate) function f, denoted ∇f is direction in which f is increasing fastest. If we take $f = E$ from above then we have:

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_k} \right)$$

To minimize E, we should take small steps in the direction opposite ∇E. Thus the update rule is:

$$\vec{w}^{(i+1)} \leftarrow \vec{w}^{(i)} - \epsilon \nabla E$$

where ϵ is the **learning rate**. That is, our $i+1$-st estimate for the weight vector comes from the i-th estimate, updated by the gradient, as given above. The parameter ϵ determines how big a step the update rule takes. If ϵ is too small, the gradient decent will take a long time to converge; if ϵ is too big, the process may "overshoot" the minima and have trouble converging. There is no systematic way to choose ϵ, but often it is set large at the start of training, and reduced as training progresses.

There is no systematic way to choose a good initial weight vector. It is often chosen randomly.

The gradient of E with respect to w is

$$= \sum_{x \in S} (\sigma(\vec{w} \cdot \vec{x}) - t(\vec{x}))\, \sigma'(\vec{w} \cdot \vec{x})\vec{x}$$

Using the continuous threshold tanh, and substituting in, we get:

$$\sum_{x \in S} (\tanh(\vec{w} \cdot \vec{x}) - t(\vec{x}))\, \mathrm{sech}^2(\vec{w} \cdot \vec{x})\vec{x}$$

Typically, the weights are updated one example at a time. So, the j-th example $x^{(j)}$ causes the weights at the $i+1$-st update to be changed as follows:

$$\vec{w}^{(i+1)} = \vec{w}^{(i)} - \epsilon \left((\tanh(\vec{w}^{(i)} \cdot \vec{x}^{(j)}) - t(\vec{x}^{(j)}))\mathrm{sech}^2(\vec{w}^{(i)} \cdot \vec{x}^{(j)})\vec{x}^{(j)} \right)$$

2.4 Complete Neural Networks

A complete neural network is a set of continuous threshold units interconnected in some topology so that the outputs of some units become the inputs of other units. See figure 5 for a simple example with one "hidden" layer. Input is applied along those edges entering the graph, and output is read from the edges leaving the graph. Each unit in the hidden layer, as well as each unit in the output layer, computes a weighted sum that is also thresholded using a continuous threshold unit. Neural nets can have much more complicated topology, with multiple layers of such "hidden" units, with outputs from one layer feeding in as inputs to the next layer.

In the one-layer networks (i.e., continous threshold units) we have previously focused on, each network computes some linear activation function, but a multiple-layer network can compute more general functions (i.e., we are no longer computing linear functions on the input). Adding multiple layers makes the networks more expressive.

Training neural nets: Backpropagation

Again, the goal of training is to find a set of weights such that the squared error is minimized:

$$E = \sum_{x \in S} \frac{1}{2} (NN(\vec{x}) - t(\vec{x}))^2$$

where $NN(\vec{x})$ is the output of the net on input $\vec{x}$. So, again, we can do gradient descent to minimize this error function; however, now since these units are layered,

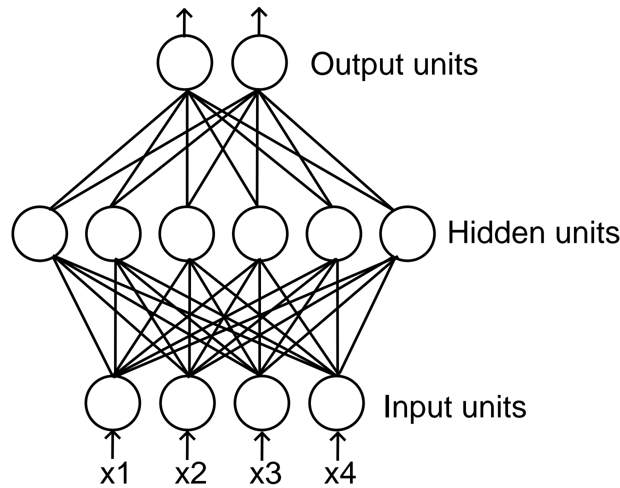


Figure 5: A neural network with 4 input nodes, 2 output nodes and one hidden layer.

changing one weight can affect other weights as well. As with continuous threshold units we want to move the weights in the direction opposite the gradient. To find this direction, we need to compute $\partial E/\partial \vec{w}$, where E is the squared error function discussed above. It turns out we can compute this partial derivative for each node using only information found in the given node and those nodes directly connected to the output of the given node. In this way, we are "propagating" errors down the network and the whole procedure of gradient descent to minimize errors on neural nets is known as backpropagation. Really, backpropagation is just a clever way to do gradient descent. The basic idea is to repeatedly apply the chain rule. It turns out that each weight in the neural net can be updated using a simple rule; we omit the rule and its derivation, but see [1] and [2] for details.

To summarize: a neural net is a more complicated version of the continuous threshold units we studied in detail. A neural net is given a set of positive and negative training examples, and its goal is to find the weights that minimize the error over the training set. This minimization is done using gradient descent, and a simple procedure known as backpropagation performs this gradient descent. One thing to note is that the backpropagation procedure is susceptible to local minima.

3 Secondary structure prediction with PhD

A more recent system for secondary structure prediction is called PhD [Rost, Sander, 1993]. We review just the main ideas, as the implementation has improved over time. The PhD system uses a combination of multiple sequence alignments and neural networks. When a protein is input, PhD finds all the homologs and builds a profile using a MSA and the profile techniques we've discussed in other lectures. It then feeds the profile into a series of neural networks.

The designers of PhD made 3 observations that guided their design:

1. Using multiple sequences is important.
2. In predicting what is happening at a residue, it is useful to consider a local window around it.
3. α-helices and β-sheets occur in runs (e.g., you don't see $\alpha\beta\alpha\beta\alpha$, and typically you expect to see at least 4 α helical residues in a row to form an α-helix).

Using multiple sequences is important because you can extract more information from multiple sequences than from a single sequence. For example, you can identify conserved residues readily. As another example, suppose you were looking at the sequence with an Ala residue at the position of interest. Then you might guess that this position favors small amino acids (since Ala is small). But you might also guess that it favors hydrophobic amino acids (since Ala is also hydrophobic). You couldn't be sure which one or both of those guesses was true. However, if you had the multiple sequence alignment

```
------S------
------A------
------A------
------S------
```

then since Ser is a small residue, but not hydrophobic, you would have more evidence that the *size* was the characteristic that was important.

PhD is an architecture with three levels. The output of each previous level is the input for the next level:

1. sequence → structure neural network

2. structure → structure neural network
3. jury system

Each of these levels is explained in more detail below.

Sequence → structure neural network

PhD begins by aligning the input sequence to its homologs and calculating a profile from the alignment. For example, with the following MSA:

```
KELN
KEFS
-DFA
HAEA
KEFS
```

we get the probabilities

col 1	col 2	col 3	col 4
K = .75	E = .6	E = .2	N = .2
H = .25	D = .2	L = .2	A = .4
	A = .2	F = .6	S = .4

Of course, in the real world, we might correct for zero probabilities with the "add-one-rule" or similar approaches.

To determine the secondary structure of residue a_j, PhD first looks at a window of the 13 residues $a_{j-6} \dots a_j \dots a_{j+6}$. The probabilities for these columns are fed into the sequence → structure neural network, which produces 3 probabilities: the probability that the amino acid a_j is in the secondary structure α ($P_{j\alpha}$), β ($P_{j\beta}$), and other (P_{jO}). These probabilities become the input for the next neural network.

So, taking a 13-long window from the training set (i.e., from an annotated structural database), if the central amino acid is helical, then the target for this example is $P_\alpha = 1$, $P_\beta = 0$, and $P_O = 0$.

The composition of each amino acid in the entire protein sequence is also feed into the neural network at this stage.

The sequence → structure network reflects the belief that the secondary structure of a residue can be predicted from a local window around it.

Structure → structure neural network

The probabilities $P_{k\alpha}$, $P_{k\beta}$, and P_{kO} are calculated using the network above for $k = j-8 \ldots j+8$. That is, it looks at a 17-long window of predictions. These probabilities are input to a second neural network that again produces probabilities that amino acid a_j is in each of the 3 possible structures.

This network tries to capture the fact that structures occur in runs.

Jury system

Because neural networks are very sensitive to topology, the set of training data, the order of training as well as other parameters, the PhD system independently trains several networks at each of the first two layers, varying these parameters. The jury system level takes as input the results from each of these nets and averages them. The secondary structure with the highest average score is output as the prediction.

4 Folklore and Miscellaneous

For all secondary structure prediction methods, β-sheets are thought to be harder to predict than α-helices because they tend to be shorter and are defined by longer range interactions.

These methods can have trouble at the ends of the sequences.

An excellent feature of PhD is that it in addition to a prediction, it also reports a reliability index from 0–9, where 9 indicates approximately 90% confidence.

References

[1] Rivest, R. Lecture 14, *Machine Learning class notes*, edited by M. Singh. See `theory.lcs.mit.edu/~mona`.

[2] Rivest, R. Lecture 15, *Machine Learning class notes*, edited by M. Singh. See `theory.lcs.mit.edu/~mona`.

[3] Rivest, R. Chapter 3. *Lecture Notes in Machine Learning* 1989.

[4] Rost B, Sander C, *Combining Evolutionary Information and Neural Networks to Predict Protein Secondary Structure*. JMB 1993.

Appendix

We include the proof of Theorem 1, taken from [3].

Proof. Without loss of generality, let $\theta = 0$, and assume that all examples are positive (you can transform your input vectors so that both things will be true). By assumption, a set of weights that fits the examples exists; let $\vec{v}$ be that set of weights, normalized so that $|v|$=1. This means that for all inputs $\vec{x}$, $\vec{v} \cdot \vec{x} > \delta$ for some $\delta > 0$ (since we assumed that the weighted sum using $\vec{v}$ for any example cannot be equal to θ). Let $\vec{w}$ be the *current* weight vector, and let $f(\vec{w}) = \vec{v} \cdot \vec{w}/|\vec{w}|$ — that is $f(\vec{w})$ is the cosine of the angle between the current weights $\vec{w}$ and the correct (normalized) weights $\vec{v}$. Since $f(\vec{w})$ is a cosine, $f(\vec{w}) \leq 1$ for all $\vec{w}$.

Consider what happens during an update. Let $\vec{w}'$ be the new weight vector. Since we only have positive examples, and thus update only when we get false negatives, by the perceptron training rule $\vec{w}' = \vec{w} + \vec{x}$. So, $\vec{v} \cdot \vec{w}' = \vec{v} \cdot \vec{w} + \vec{v} \cdot \vec{x}$, and we then have

$$\begin{aligned} |\vec{w}'|^2 &= \vec{w}' \cdot \vec{w}' \\ &= (\vec{w} + \vec{x}) \cdot (\vec{w} + \vec{x}) \\ &= |\vec{w}|^2 + 2\vec{w} \cdot \vec{x} + |\vec{x}|^2. \end{aligned}$$

We know that $2\vec{w} \cdot \vec{x} < \theta = 0$ since we got a false negative and since $|\vec{x}|^2 = 1$, we have

$$|\vec{w}'|^2 < |\vec{w}|^2 + 1$$

This means that the square of the weight vector is increased by less than 1 after each update.

If we start with the weights all zero, then after t updates $|\vec{w}| < \sqrt{t}$ and $f(w) > t\delta/\sqrt{t}$. For $t > 1/\delta^2$, $f(\vec{w}) > 1$, which is a contradiction since this function computes a cosine. Hence, the number of updates t is bounded. ■