Protein Sequence analysis using sequential neural networks

Bachelor Project

Department of Computer Science
University of Copenhagen
Github: https://github.com/Cavtheman/proteinbachelor.git

Casper Lisager Frandsen <fsn483>,
Marc Pedersen <wfj327>

Version 1
Due: June 8th

Contents

1. Introduction

Proteins encode many different significant and more or less useful properties in the organisms they exist in. Proteins consist of long chains of up to several hundred amino acids, with some of these being more important for the proteins properties than others. Now, how a protein interacts with other proteins is primarily based on their 3D shape, which is generally called their tertiary structure. A Proteins primary structure consists of the amino acid chain, while their secondary structure is the shape that these chains form. (eg. helices or sheets). The tertiary structure would then describe the way that these are combined and folded in 3D space. Because their interactions with other proteins are primarily affected by their tertiary structure, finding this is very important for understanding a protein's function.

Finding a protein's tertiary structure has historically been a very slow and expensive process however, generally requiring significant amounts of lab work and time. On the contrary, finding their primary structures (amino acid sequences) is relatively very easy, and the amount of data available in this form is actually increasing exponentially. [1] This means that there is a significantly larger amount of data available with the primary structure of proteins. Thus, finding a meaningful representation that can be used to say something about other properties of a protein using only the primary structure is an active area of research. There are many parallels between tertiary structure prediction and natural language processing (NLP) in that the data consists of sequences of unknown length containing many long-range dependencies between subsequences with high importance.

Recurrent Neural Networks (RNNs) in various forms have been go-to architectures for most modelling tasks regarding these types of sequences for a long time now and with good reason. They provide good results and can, given proper architectures, remember many of the dependencies between subsequences and achieve state-of-the-art performance on many protein representation and prediction tasks. However, a recent paper has shown that in many cases, Convolutional Neural Networks (CNNs) can achieve as good, if not better results when applied properly. [4] We will in this paper reproduce an RNN that can learn some representation from these raw sequences by training it for next-token prediction. [2] The idea here is that if the model can accurately predict what the next amino acid is in these long chains, it is because it has learned something inherent in the structure of this protein.

We will then compare these results to a CNN that is trained to predict each token in the input sequence. This is an easy task, except the input will be changed to something incorrect. In this case, the idea is that the model will end up having to learn what each element is supposed to be from the context around it. We do this by encoding the primary structure of the protein into an arbitrarily small representation, and then decoding this representation to output the same primary structure. Having learned how to do this well must mean that it has also learned something inherent about the structure of these proteins, and thus the encoding of the protein must have some meaningful information about its structure.

To evaluate the very different results of these models, we will be training a simple model on the representation that our more complex models learn, against a dataset that describes how stable a protein is. Since this stability is entirely based on the proteins structure, having a good representation of this must mean that it is relatively easy to predict the proteins stability.

2. Theory

(a) General neural network models

A neural network is a deep learning technique. Deep learning is a branch of machine learning and is from a mathematical perspective - A method of representing differential functions mapping one type of variable x, into another type of variable y.

$$f(x) = y$$

Neural networks have high performance in detection and recognizing tasks. Through learning and classifying complex patterns and features, they can learn some representation of a given input. The neural network model is inspired by how biological neurons work in the brain. The objective of neural network models is to create a model that is teachable and can perform problem-solving tasks, which is easy for a human, but difficult for a computer.

As for the brain, a neural network has neurons. These neurons are called nodes. These neurons are clustered in different layers, and each layer of neurons is in some way connected with other neurons in the other layers. A simple version of a neural network is the feed-forward neural network. In this kind of neural network, the connections are structured in a way such that there are no cycles in the graph. This implies the data travels only in one direction. This kind of architecture is composed of an input layer, one or more hidden layers, and an output layer. Given any kind of input variable X, the model creates its estimate of the output_variable Y, after the input has been processed by the different layers. In general, if you feed a model $\mathbb{K}$ the input variable $\mathbb{X}$, it will produce the output value $\mathbb{Y}$:

$$\mathbb{K} : \mathbb{X} \rightarrow \mathbb{Y}$$

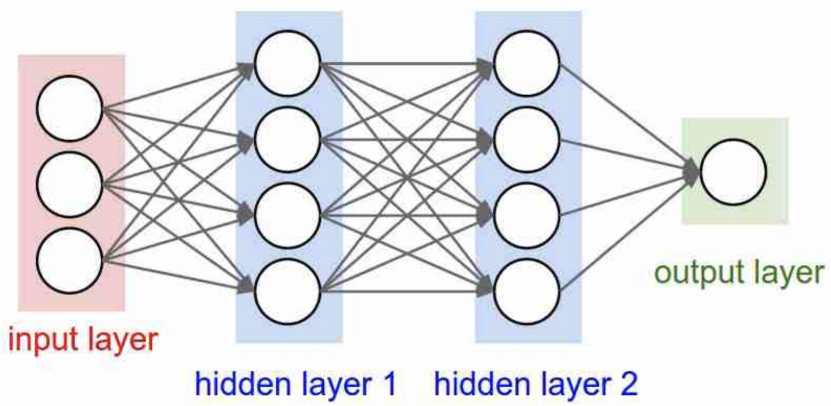

Figure 1: Feed forward model, with two hidden layers.

In our project we use two different types of neural networks, A convolutional neural network and recurrent neural network. These will be explained more in-depth later in the report.
In machine learning, we have two methods of learning; supervised and unsupervised. Supervised learning is that for each input $\mathbb{X}$, the data also contains a target output $\mathbb{Y}$. The goal is to use some input to predict one or more given labeled outputs. Making the model learn an unknown function, which can map the input to the labeled output. Unsupervised learning does not have a corresponding target $\mathbb{Y}$ output for any part of the data. The goal is to learn some useful insight into the data's characteristics, without having a supervising label. In our project, we focused mainly on unsupervised learning, given we had limited labeled data, but a great amount of unlabeled data.

(b) Hidden layers

A hidden layer in a neural network is one of the layers of neurons between the input layer and the ouput layer. Each layer consists of columns of neurons, and are connected to neurons in adjacent layers. The connections between these neurons are called weights, and are represented by a real number value. These weights are individual, meaning that two neurons in the same layer, recieving the same input, can have different weights associated to it. A neuron in a hidden layer, takes as input, all of the attached neurons multiplied by their weight, summed including a fixed bias and passed through an activation function to a scalar-output. This activation function is in our case a rectified linear unit.

Presenting this as a function, the sum of the multiplied weights and added bias we define as y_i. We then define the connected neurons x_j for $j \in \{1,...,N\}$, where N is the amount of the connected neurons. We define the associated weights as w_{ij} and added bias as b_i. Having the activation function as σ:

$$y_i = \sum_{j=1}^{N} x_j w_{ij} + b_i \tag{1}$$

$$a_i = \sigma(y_i) \tag{2}$$

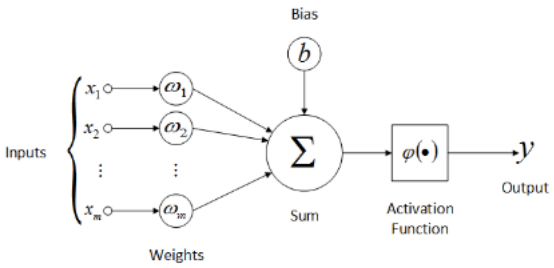

Figure 2: Diagram representing how a neuron calculates its value

When training a neural network, the only parameters that change are the weights. Since the weight from the neuron(1) is multiplied with the weight connecting to neuron(2), the magnitude of this weight defines how much influence neuron(1) has on neuron(2). Thus, the quality of the model depends on how good it is at adjusting the weights, and by that teaching the model to map an input to a correct output.

(c) Convolutional neural network

(c).1 Convolutional neural network

In our project, one of the model we use is a convolutional neural network(CNN). CNN's are strong tools for classification of feature maps, which will be described below. In a CNN, the input data goes through one or more layers, where each neuron in each layer is applied with convolutional operations, which are fed to adjacent layers - forming a fully connected neural network layer.

In our case, each amino acid will be represented as a neuron in the neural network's input layer. Since it's a fully connected network, each neurons' value, is calculated by looking at all neurons from the previous layer. This means that each neuron takes the entire sequence as input - applying its weights and then passes it through an activation function. By doing this, the model gets fit to detect important patterns in the sequence.

(c).2 Convolutions

The convolutional layers are what makes CNN's such a strong tool. In general, a convolutional layer consists of a set of learnable filters. Applying a convolutional operation on a neuron is done by using these filters to transform the input according to the filter, and output the transformed data. As mentioned earlier, CNNs can detect important patterns. The detections of these patterns are happening in these convolutional operations.

These filters also called 'kernels'. Kernels are simply just small matrixes initialized with some predefined dimensions and random values. When a kernel is applied on some feature map, the kernel simply

goes across all the feature-channels.

Presenting this as an example, we introduce a 3x3 kernel g with stride 1x1. The kernel will stride over all possible regions of the input, where the kernel can fit. Simply finding the dot-product between the regional values of the input f, and the kernel values of g.

$$g = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

This means, that if some were to use the filter kernel g, on some input image. This filter g would simply highlight the left vertival edges of some input; Rotating the filter, whould result in finding right vertical- and horicontal edges presented in the image, depending on how much the filter is rotated. Due to bounderies; using a filter greater than 1x1 will yield a smaller output matrix. It is possible to introduce padding to avoid dimension loss.

Lastly, the convolutional layers have something called stride. The stride value defines how the filter travels over the input. Meaning how many steps it takes before calculating a new dot-product. In the example given in figure(3 and 4), we have a stride size of 1x1. This means, that every single region where a 3x3 can fit in the input, the dot-product will be calculated. If we were to change the stride to size 2x1, the output would become smaller, since we take bigger steps, each time we are calculating the dot-product, see figure(6 and 5).

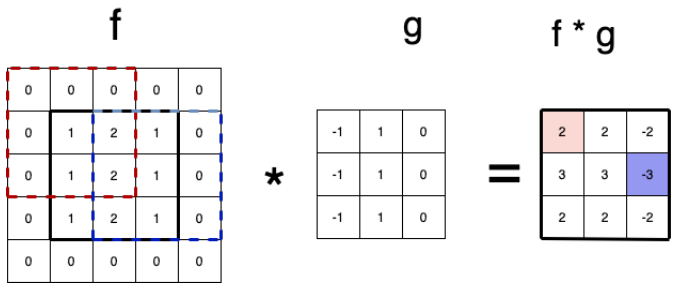

Figure 3: Shows how a 3x3 left vertical edge filter acts on a 5x5 pixel feature map, with stride 1x1

Given that we are looking at sequences which are one dimensional, the filtering will look like:

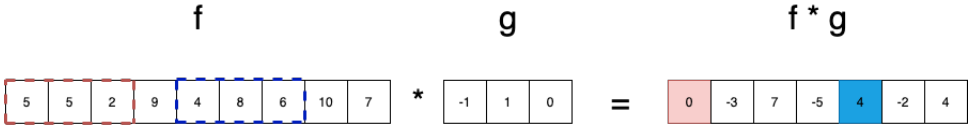

Figure 4: Shows how a 1x3 left vertical edge filter acts on a 1x9 feature map, with stride 1

The following figures represents how the output would look like if the stride was different from 1x1.

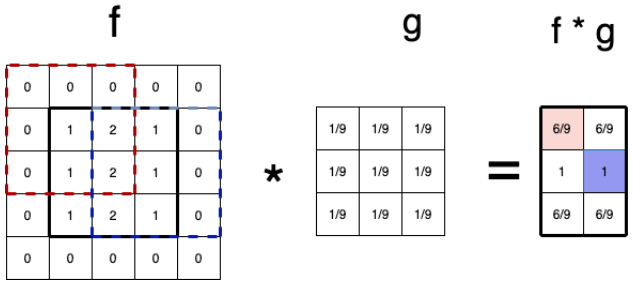

Figure 5: Shows how a 3x3 mean filter acts on a 5x5 pixel feature map, with stride 2x1

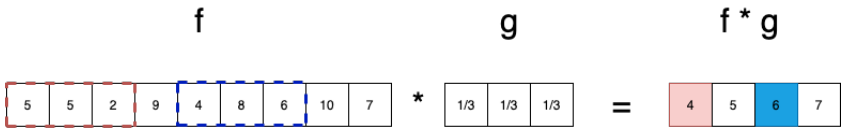

Figure 6: Shows how a 1x3 mean filter acts on a 1x9 feature map, with stride 2

(c).3 Pooling

Pooling is a way of shrinking the non-channel dimension of some input. The idea of pooling reminds a lot about how convolution works. Pooling also operates with a kernel, sliding over the features. Two popular pooling methods are Max-pooling and Average-pooling. In pooling methods, we have a kernel of some predefined size. This kernel will slide over the features, just like how the filter traveled through the features, during a convolution.

The output of Max-pooling (figure 7), will always take the maximum value, of the features where the kernel is currently at. The output of Average-pooling (figure 8), will take the average of the features where the kernel is currently at.

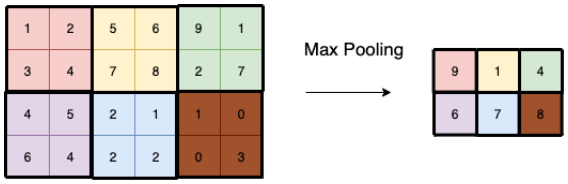

Figure 7: 2x2 kernel, max-pool

Defining the values within the striding kernel with size hxw as A, the Max-pooling function looks like:

$$Maxpool(A) = max(0, A)$$

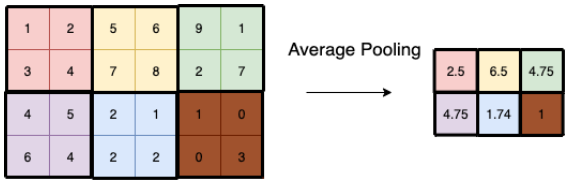

Figure 8: 2x2 kernel, average-pool

Defining the values within the striding kernel with size hxw as A, the Average-pooling function looks like:

$$Averagepool(A) = \frac{\sum_{i \in A} A_i}{h * b}$$

(c).4 Rectified Linear Unit (ReLU)

The Rectified Linear Unit (RelU) is a very simple activation function, none the less, still a very useful function. When ReLU is called with some input, it will go through all values of the input, turning all negative values to 0, and all positive numbers simply keep their value.

$$ReLU(x) = max(0, x)$$

or

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{3}$$

Figure 9: Example of how ReLU() works

(c).5 Fully connected layer

When taking convolutional layers, ReLU layers and pooling layers, stacking them, in way so the output of the first operation becomes the input of the next operation in the stack. When these operations get stacked like this, fully connected layers are often introduced at the end of the last convolution. These fully connected layers are the same as the hidden layers in a feed-forward neural network. In a CNN you can have non-convolutional layers, though at least one is required for it to be a CNN. CNNs can have multiple of these layers. Normally the last hidden layer before the output layer is a convolutional layer since the output of this only shows patterns that a fully connected layer can pick up on, to come up with a specific prediction of a class etc.

(d) Recurrent Neural Networks

Typical neural networks do not have a state that allows them to "remember" previous data points, for prediction. This is where the basic idea behind a recurrent neural network (RNN for short) comes from. It has a hidden state representing its memory regarding previous data. It will then be able to make decisions regarding new data, taking this previous data into account.

If we take a look at the sentences below, the idea is that the RNN would first be able to understand the nearby context; that the next word is supposed to be a reason for staying home. And also understand the larger context, (that the pandemic is that reason) without needing to remember unnecessary stuff like the words "is", "a" and so on. It should then be able to predict that pandemic would be the next word in the sentence.

- There is a pandemic. I had to stay home from university because of the ___

RNNs are extremely flexible in how they can be used. There are, in fact, four main ways they transform the input into something usable. The first is in a "one to many" fashion, in which they take a single input and output a sequence. This could be useful for tasks like generating descriptions based on fixed size inputs such as images.

The second is in a "many to one" fashion in which the input is a sequence, and the network outputs a single result a the end. This is useful in cases where an entire sequence has a single class to be predicted.

The last two both output in a "many to many" fashion, but how they do it is significantly different. The first case outputs a completely new sequence that is not necessarily the same length as the input sequence, after having seen the entire input. This is useful for networks trying to represent the data in a different way. The last case is the one that will be described further down in this report. This case outputs an element of a new sequence for each element in the input. This means that for each output element, the RNN only knows about previous elements. This makes it the perfect candidate for next token prediction.

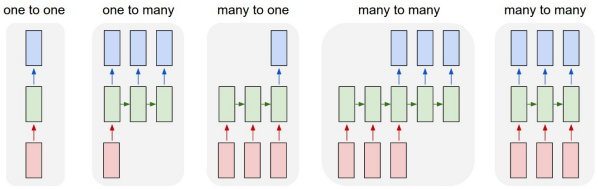

Figure 10: A visual representation of the output types described. The one-to-one is how fully connected networks work, for comparison. Image Source: [11]

The basic structure of an RNN is quite simple, consisting of a single linear layer and activation function, which is usually the *tanh* function. The network is interesting in that each "cell" passes along a hidden state as input to the next cell. This means that each cell takes both the current input element and the previous cell's output and concatenates them and uses this as input to the linear layer.

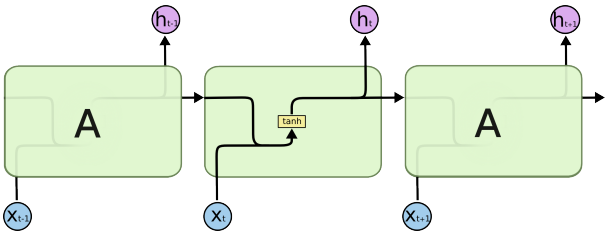

Figure 11: A visual representation of the inner structure of an RNN. Image Source: [11]

(e) The exploding/vanishing gradient problem

One major issue with using basic RNNs is the fact that they suffer quite heavily from the exploding/vanishing gradient problem. This problem arises from the fact that very large gradients and gradients very close to zero tend to approach infinity and zero respectively, as they are multiplied together during backpropagation.

To show this, let's consider the following example. We have the prediction vector h_t at each timestep t. The gradient when using the backprop through time algorithm looks as follows:

$$\frac{\partial E}{\partial W} = \sum_{t=0}^{T} \frac{\partial E_t}{\partial W} \tag{4}$$

With network weights W for the concatenated input vectors, and errors E through a sequence of length T. We will define the input vectors as follows. c_t for the recurrent vector that the network passes forward, and x_t for the input vector, both at timestep t. This gradient updates the model weights according to the following formula, with a learning rate α:

$$W \leftarrow W - \alpha \frac{\partial E}{\partial W} \tag{5}$$

If we use basic gradient descent, then the gradient of the error at an arbitrary step s is given as follows:

$$\frac{\partial E_s}{\partial W} = \frac{\partial E_s}{\partial h_s}\frac{\partial h_s}{\partial c_s}\cdots\frac{\partial c_2}{\partial c_1}\frac{\partial c_1}{\partial W} \tag{6}$$

$$= \frac{\partial E_s}{\partial h_s}\frac{\partial h_s}{\partial c_s}\left(\prod_{t=2}^{s}\frac{\partial c_t}{\partial c_{t-1}}\right)\frac{\partial c_1}{\partial W} \tag{7}$$

Now we want to show that as $s \to \infty$ the derivative $\to 0$. Since the network weights W are the concatenated c and x vectors, we can rewrite c_t:

$$c_t = tanh(W_c \cdot c_{t-1} + W_x \cdot x_t) \tag{8}$$

We find the derivative of this:

$$\frac{\partial c_t}{\partial c_{t-1}} = tanh'(W_c \cdot c_{t-1} + W_x \cdot x_t) \cdot W_c \tag{9}$$

Plugging these expressions in, we get the following expression:

$$\frac{\partial E_s}{\partial W} = \frac{\partial E_s}{\partial h_s}\frac{\partial h_s}{\partial c_s}\left(\prod_{t=2}^{s} tanh'(W_c \cdot c_{t-1} + W_x \cdot x_t) \cdot W_c\right)\frac{\partial c_1}{\partial W} \tag{10}$$

Since the derivative of the tangent is ≤ 1 as can be seen in Figure 12, the expression will tend towards zero as $s \to \infty$. However, if the weights W_c are significantly large, they may cause the expression to instead explode towards ∞ as $s \to \infty$. It should be noted that while the derivative at 0 is equal to 1, it is not very likely that this event will happen when backpropagating.

If we take a look back at the formula used to update the weights, we can easily see the problem:

$$W \leftarrow W - \alpha \frac{\partial E}{\partial W} \tag{11}$$

Since $\frac{\partial E}{\partial W} = \sum_{t=0}^{T}\frac{\partial E_t}{\partial W} \to 0$ as $T \to \infty$, we can see that the updates will eventually be incredibly small, to the point where they are basically nonexsistent.

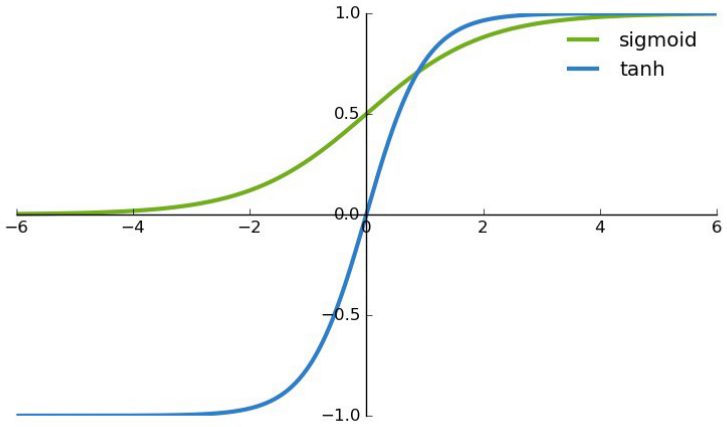

Figure 12: Showing the derivative of the tanh function. Image Source: [13]

(f) An LSTMs structure

First, we will go over how the LSTM architecture works, then we will go over how this solves the vanishing/exploding gradient problem.

An LSTM actually has an extra input and output vector compared to the base RNN architecture. Much like the base RNN, at time step t it takes an input vector x_t and concatenates it with a hidden vector from a previous output, which we will refer to as h_{t-1}. We will refer to the concatenated vector as $[x_t, h_{t-1}]$. The difference is that it also takes in a cell state, which we will refer to as c_{t-1} since it is also from a previous output. The idea behind this cell state is for it to act as a "highway" for information that can only be changed with linear operations. This means that the cell state is very helpful for remembering information that was learned a long time ago.

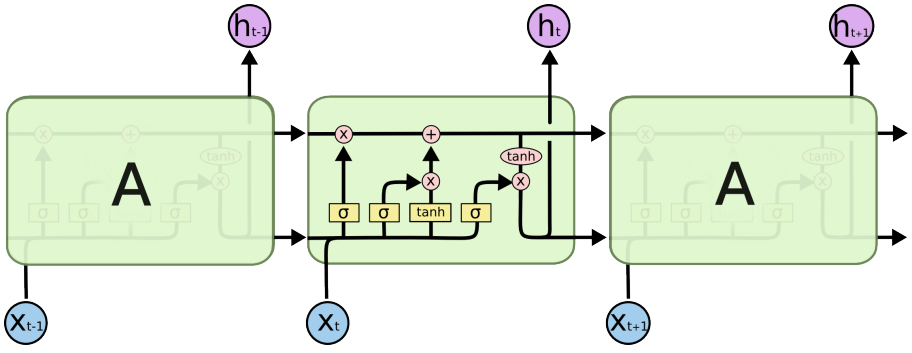

Figure 13: Visualisation of the LSTM architecture. Image Source: [12]

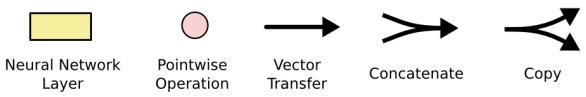

Figure 14: Notation for the image above. Image Source: [12]

Another major difference is the fact that there are four separate linear layers in an LSTM. These layers all take the $[x_t, h_{t-1}]$ vector as input. Going from left to right, we first have the "forget" layer. This

layer is activated with a sigmoid function, and controls which parts of the cell state to forget, given new information when training. It is also referred to as a forget "gate" because it controls the information that comes through. It does this by performing pointwise multiplication between the cell state and the output of this layer. The forget gate f_t at time step t with weights W_f is defined mathematically as follows:

$$f_t = \sigma(W_f \cdot [x_t, h_{t-1}]) \tag{12}$$

The next two layers also interact with the cell state, but are combined via pointwise multiplication to act as a single gate. The first layer has a sigmoid activation function, while the second has a *tanh* activation function. The output of this pointwise multiplication will then be combined with the cell state using pointwise addition. This gate acts as an input gate, in the sense that it controls which parts of the cell state get updated as per the new information. With weights W_c for the *tanh* layer and W_i for the sigmoid layer, the input gate i_t at time step t is defined as follows:

$$\tilde{c}_t = tanh(W_c \cdot [x_t, h_{t-1}]) \tag{13}$$
$$\tilde{i}_t = \sigma(W_i \cdot [x_t, h_{t-1}]) \tag{14}$$
$$i_t = \tilde{c}_t \otimes \tilde{i}_t \tag{15}$$

The final layer also has a sigmoid activation but is used quite differently. A copy of the cell state will be passed through a tanh activation and multiplied pointwise with the output of this layer. The output of this is then the new h_t that will both be passed forward and used as the output. This gate acts as an output gate because it is what controls which parts of the information in the cell state will be passed along as input for the next time step.

With weights W_o, the output gate o_t is defined as follows at time step t:

$$o_t = \sigma(W_o \cdot [x_t, h_{t-1}]) \tag{16}$$

The result of all these operations mean that there are two output vectors which are defined as follows:

$$c_t = c_{t-1} \otimes f_t \oplus i_t \tag{17}$$
$$h_t = \sigma(o_t \otimes tanh(c_t)) \tag{18}$$

(f).1 Why an LSTM structure solves the vanishing/exploding gradients

In an LSTM, the gradient of the error at each time step is calculated in almost exactly the same way. But the fact that the cell state, which corresponds to the basic RNNs output, is calculated differently means that it does not suffer from the problem to the same degree. We showed before that the following expression will tend towards zero:

$$\frac{\partial E_s}{\partial W} = \frac{\partial E_s}{\partial h_s}\frac{\partial h_s}{\partial c_s}\left(\prod_{t=2}^{s}\frac{\partial c_t}{\partial c_{t-1}}\right)\frac{\partial c_1}{\partial W} \tag{19}$$

However, in an LSTM, c_t is calculated differently, so let's look at how this affects the overall expression:

$$c_t = c_{t-1} \otimes f_t \oplus i_t \tag{20}$$
$$= c_{t-1} \otimes f_t \oplus \tilde{c}_t \otimes \tilde{i}_t \tag{21}$$
$$= c_{t-1} \otimes \sigma(W_f \cdot [x_t, h_{t-1}]) \oplus tanh(W_c \cdot [x_t, h_{t-1}]) \otimes \sigma(W_i \cdot [x_t, h_{t-1}]) \tag{22}$$

Because c_t is so different, the expression for the derivative changes:

$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial}{\partial c_{t-1}} c_t \tag{23}$$
$$= \frac{\partial}{\partial c_{t-1}} \left[c_{t-1} \otimes f_t \right] + \frac{\partial}{\partial c_{t-1}} \left[\tilde{c}_t \otimes \tilde{i}_t \right] \tag{24}$$
$$= \frac{\partial f_t}{\partial c_{t-1}} \cdot c_{t-1} + \frac{\partial c_{t-1}}{\partial c_{t-1}} \cdot f_t + \frac{\partial \tilde{i}_t}{\partial c_{t-1}} \cdot \tilde{c}_t + \frac{\partial \tilde{c}_t}{\partial c_{t-1}} \cdot \tilde{i}_t \tag{25}$$
$$\tag{26}$$

This can be written out as follows:

$$\frac{\partial c_t}{\partial c_{t-1}} = \sigma'(W_f \cdot [x_t, h_{t-1}]) \cdot W_f \cdot o_{t-1} \otimes tanh'(c_{t-1}) \cdot c_{t-1} \tag{27}$$
$$+ f_t \tag{28}$$
$$+ \sigma'(W_i \cdot [x_t, h_{t-1}]) \cdot W_i \cdot o_{t-1} \otimes tanh'(c_{t-1}) \cdot \tilde{c}_{t-1} \tag{29}$$
$$+ \sigma'(W_c \cdot [x_t, h_{t-1}]) \cdot W_c \cdot o_{t-1} \otimes tanh'(c_{t-1}) \cdot \tilde{i}_{t-1} \tag{30}$$

For brevity, we define each addend here as one variable:

$$W_t = \sigma'(W_f \cdot [x_t, h_{t-1}]) \cdot W_f \cdot o_{t-1} \otimes tanh'(c_{t-1}) \cdot c_{t-1} \tag{31}$$
$$X_t = f_t \tag{32}$$
$$Y_t = \sigma'(W_i \cdot [x_t, h_{t-1}]) \cdot W_i \cdot o_{t-1} \otimes tanh'(c_{t-1}) \cdot \tilde{c}_{t-1} \tag{33}$$
$$Z_t = \sigma'(W_c \cdot [x_t, h_{t-1}]) \cdot W_c \cdot o_{t-1} \otimes tanh'(c_{t-1}) \cdot \tilde{i}_{t-1} \tag{34}$$

Thus, the gradient can now be written as follows:

$$\frac{\partial c_t}{\partial c_{t-1}} = W_t + X_t + Y_t + Z_t \tag{35}$$

The fact that we now have a derivative consisting of several additions of number that are < 1 means that the LSTM can balance these out. Because of this, the addition in and of itself may result in the sum being ≥ 1. The full expression can be written as follows:

$$\frac{\partial E_s}{\partial W} = \frac{\partial E_s}{\partial h_s} \frac{\partial h_s}{\partial c_s} \left(\prod_{t=2}^{s} \left[W_t + X_t + Y_t + Z_t \right] \right) \frac{\partial c_1}{\partial W} \tag{36}$$

An interesting note here is the fact that the forget gate comes through all of this. This means that the LSTM can pick and choose information to be forgotten and information to be remembered at any given time step. This property is significant in preventing vanishing gradients. Take a look at this expression once again, and assume that for this $s < T$, we have:

$$\sum_{t=1}^{s} \frac{\partial E_t}{\partial W} \to 0 \tag{37}$$

If we don't want the gradient to vanish, we can then change the forget gate at $s + 1$ so the following is true:

$$\frac{\partial E_{s+1}}{\partial W} \not\to 0 \tag{38}$$

Because of this, we can say the following:

$$\sum_{t=1}^{k+1} \frac{\partial E_t}{\partial W} \not\to 0 \tag{39}$$

This is exactly what we wanted the LSTM to do. Now the gradients no longer vanish.

(g) Training the model

(g).1 Training What happens when training

The goal of training a neural network is for it to be as effective as possible. Meaning that the predictions it's making, is as close as the desired output. When training a model, various operations and methods are being introduced.

The training of a neural network is an iterative process. When a model is being trained, it is fed with some input x transforming it into an output $\hat{y}$, which is an estimation prediction of the desired value y. This prediction is then together with the desired input y, fed into an error function; which outputs some cost. An error- or cost function is, a way to estimate the differences between the estimated prediction $\hat{y}$ and y, by computing some cost value. This cost explains how much y and $\hat{y}$ differentiate from each other.

Using this cost function, in combination with backpropagation and some optimizer (which we will explain later in the report). It is possible to update, the model's parameters, in a way that minimizes the cost function's output. When the cost value becomes smaller, The amount that $\hat{y}$ deviates from y, also becomes smaller. This means that our model, becomes better at predicting a good estimation $\hat{y}$ compared to y.

(g).2 Error Function

As mentioned; a loss function describes the differents between $\hat{y}$ and y. In our project, we use the Cross-Entropy loss function. To formally understand how this function works, we first have to understand how the 'entropy' part works.

Given a probability distribution of a set of events where the probabilities sum to 1, the Entropy describes the average information value this distribution has, given the probability of the events happening. This information value is described in bits. Finding the information value for a single occurrence is defined as: $-log_2(p(x))$, where $p(x)$ is the probability of x occurring.

An example helps formally explaining this: If one was told that tomorrow there will have a 75% chance of rain and 25% chance of sun. If that person wakes up the next day, finding out it's not raining, this value this would have a 2-bit information value, because $-log_2(0.25) = 2$ bits. If he woke up the next day seeing it was raining, the value of that information is only 0.41 bits because $-log_2(0.75) = 0.41$ bits. Thus, the higher the probability of the occurrence, the lower the information bit-value will become. The intuition here is that if you are very sure that something is correct, but it isn't, then you learn more than if you were not sure.

The Entropy function h simply finds the average of the information contained in some occurences $X = \{x_0, x_1, ..., x_n\}$ where $\sum_i^n p(x_i) = 1$:

$$h(X) = \sum_{i=1}^{n} p(x_i)(-log_2(p(x_i))) \tag{40}$$

Using Entropy function from equation 40 on the weather example, it'll yield $(0.75 * 0.41) + (0.25 * 2) = 0.81$.

The Cross entropy function H measures the entropy between 2 probability distributions (P and Q) over the same set of events. We denote an event from the P distribution as $x_i^{(P)}$ and an event from Q as $x_i^{(Q)}$. Thus, Cross-Entropy function H will look like:

$$H(P, Q) = \frac{1}{n} \sum_{i=1}^{n} p(x_i^{(P)})(-log_2(p(x_i^{(Q)}))) \tag{41}$$

So in our case we want to predict the right amino acids. Say that we are only looking at sequences with 3 different amino acids {A, C, D}, and we want to predict the amino acid 'A'. We define the following y and an arbitrary $\hat{y}$.

	$p(A)$	$p(C)$	$p(D)$
y	1.0	0.0	0.0
$\hat{y}$	0.6	0.10	0.30

Using y and $\hat{y}$ distributions in the Cross Entropy model H, the function will yield the following cost:

$$H(y, \hat{y}) = 0.736 \tag{42}$$

	$p(A)$	$p(C)$	$p(D)$
y	1.0	0.0	0.0
$\hat{y}$	0.4	0.30	0.30

Table 1: example of a less confident model

Using this example of a less confident model, the function will yield the following cost: $H(y, \hat{y}) = 1.321$

	$p(A)$	$p(C)$	$p(D)$
y	1.0	0.0	0.0
$\hat{y}$	0.9	0.08	0.02

Table 2: example of a more confident model

Using this example of a more confident model, the function will yield the following cost: $H(y, \hat{y}) = 0.152$

Now it's clear to see that the greater the difference of $\hat{y}$ and y is, the greater the loss.

(g).3 Backpropagation

When a model is run on some input, it predicts some output $\hat{y}$, which the model wants to evaluate against the desired value y. This evaluation happens, with some error function. This error functions outputs some loss-value which is dependent on how much $\hat{y}$ deviates from y. The greater the deviation

is, the greater the loss/cost will become. Based on the size of the loss, the models know with how great magnitude it has to adjust its parameters (weights and biases). The goal is to adjust the parameters, so the predicted output $\hat{y}$, gets as close to the desired output y.

Trying every single combination of weights, in the pursuit of finding the best fitting weights for a specific problem, is a very comprehensive task, even for a computer. This problem is being solved with backpropagation. The idea is to find the minimum value of which the loss function can take. Calculating the gradient is a very effective way of finding this minimum. Finding this minimum can be achieved with various gradient descent algorithms. The gradient describes the slope of the function at a specific point in the graph. Assuming we have a function f, the following gradient of this function is f'. If f' takes some input x, and $f'(x)$ yields a negative result, it means that the graph currently has a negative slop at this current point. Thus, to get closer to a minimum, we have to increase the value of x.

But a gradient descent algorithm is an algorithm that finds the minimum given it has a gradient available, it is not the algorithm for finding the gradient. Backpropagation is the method of computing the magnitude each weights gradient has on the final error. The way this is done is by propagating backward through the layers, computing the gradient at each weight.

Each neuron's value is affected by the previous weights. This means, that the gradient of a single weight in one of the first layers, affects the value of all the following layers. If the gradient of this weight is either negative or positive, it will either affect the output in either a negative or positive way. The magnitude of this gradient, explains how great influence this weight has on the output. To formally understand what is happening, we will look at a very simple neural network, where each layer contains a single neuron, see figure(15). Explaining this as an equation, we define a^n as the predicted output, L and C as some loss-function and cost respectively, and y as the desired output.

$$a^{n-2} = \sigma(a^{n-3}w^{n-2} + b^{n-2}) \tag{43}$$
$$a^{n-1} = \sigma(a^{n-2}w^{n-1} + b^{n-1}) \tag{44}$$
$$a^{n} = \sigma(a^{n-1}w^{n} + b^{n}) \tag{45}$$
$$C = L(a^n, y) \tag{46}$$

by looking at this the above equation, it is clear to see that each weight affects the cost. But to find how much a specific weight (w^n) affects the cost, we will have to find the derivative of the cost with respect to w^n, which is achieved with the chain rule. For this notation, we define z as being the value of a neuron before the activation function.

$$\frac{\partial C}{\partial w^n} = \frac{\partial z^n}{\partial w^n}\frac{\partial a^n}{\partial z^n}\frac{\partial C}{\partial a^n} \tag{47}$$

The same counts for how much impact the bias and the activation of the previous layer, recpectivly has on the Cost:

$$\frac{\partial C}{\partial b^n} = \frac{\partial z^n}{\partial b^n}\frac{\partial a^n}{\partial z^n}\frac{\partial C}{\partial a^n} \tag{48}$$

$$\frac{\partial C}{\partial a^{n-1}} = \frac{\partial z^n}{\partial a^{n-1}}\frac{\partial a^n}{\partial z^n}\frac{\partial C}{\partial a^n} \tag{49}$$

With intuition we can simply keep iterating the same chain rule idea backward, to see how sensitive the cost is to previous weights and biases.

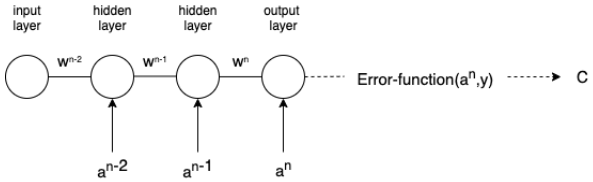

Figure 15: simple neural network

For a fully connected network which has more neurons per layer, the same idea can be applied. Having a neural network like the one seen in figure(14). For any neuron $i \in \{1, 2, ..., k\}$ in layer (n-1), and any neuron $j \in \{1, 2, ..., t\}$ in layer n, the impact w_{ji} has on the cost is calculated by:

$$\frac{\partial C}{\partial w_{ji}^n} = \frac{\partial z_j^n}{\partial w_{ji}^n} \frac{\partial a_j^n}{\partial z_j^n} \frac{\partial C}{\partial a_j^n} \tag{50}$$

Now since each neruon impact multiple neurons, which each impacts the cost, the impact of how much the activation function impacts the cost is calculated, like:

$$\frac{\partial C}{\partial a_k^{n-1}} = \sum_{j=1}^{t} \frac{\partial z_j^n}{\partial a_k^{n-1}} \frac{\partial a_j^n}{\partial z_j^n} \frac{\partial C}{\partial a_j^n} \tag{51}$$

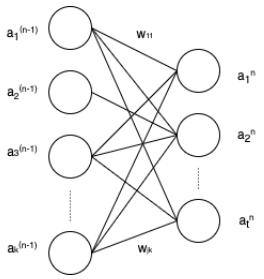

Figure 16: More complex neural network

(g).4 Gradient descent

There are various of different gradient descent algorithms. As mentioned in the backpropagation section, gradient descent is a technique which measures the degree of change w.r.t some parameter, at a certain point in a function. It is an iterative process that makes it path towards some minimum. This minimum point is defined as the local minima of the function; it is not necessarily a global minima. This means, gradient descent can be applied on functions which are differentiable w.r.t its parameters. This, makes gradient descent a relatively effiecient optimization method, when looking at neural networks.

There are variations of how the gradient descent is done, one of which is the Stochastic gradient descent (SGD) algorithm. When training on some dataset, usually what the programmer does is using batches, which refers to the number of samples in the dataset, which is used to calculate the gradient. Using the gradient of all individual samples results in a smoother way of finding the minima. Though, if the dataset is very big, this can quickly become a very comprehensive task. SGD relies on the random probability of randomly picking one sample from the batch in each iteration. This means that SGD takes a single example of a gradient, instead of the sum of all gradients of the error function. As can be

seen in figure(17), Doing this results in a more noisy way of finding the local minima. The noisy path doesn't matter, as long as the path it takes in finding the local minima, is actually finding the correct local minima. It's important to note, that because of the stochastic probability of picking some gradient, it is required that a lot of data is available.

(a) GD on each individual sample in the dataset

(b) SGD's path using randomly selected gradient

Figure 17: Two different gradient descent paths

(g).5 Adam

The optimizer we are using in our models is the Adam (adaptive moment estimation) optimizer. The Adam optimizer is based on stochastic gradient descent [8], therefor Adam also has a random probability of picking a gradient in the sample data. As well as SGD, Adam is well suited for objective functions, which require some scalar parameter maximization or minimization.

Adam also uses something called gradient descent with momentum. Momentum is a method, which computes an exponentially weighted average of previously calculated gradients, and uses this as gradient in the gradient descent step.
when talking about optimizers in neural networks, "learning rate" is a keyword in this context. The learning rate is a parameter, which defines the rate of change given a gradient. This means that the greater the learning rate is, the greater the change of some specific parameter will become; in the context of finding the local minima.

Adam differentiates from SGD, the way it dynamically can adjust the magnitude of how it updates the scalar parameters based on the momentum of the gradient. this means that Adam is capable of computing adaptive learning rates for different parameters [8]. Thus, for each iteration of updates in the neural network, the learning rate will be updated as well. To simplify what this means; if a gradient with respect to some parameter, has not changed for a few iterations, the learning rate for that specific gradient will be decreased.

3. Data

In this paper, we have used three different datasets, for three different purposes. The first dataset is a large dataset containing only the primary structures of proteins, which we use to train our models. The second and third datasets are significantly smaller, consisting also of the primary structures of proteins, and are labeled with structural class, and a value representing the stability of the protein respectively. The dataset containing the structural class is used as out validation set. It was never used for training, so it could be used to estimate the performance of the models. The stability dataset is used as a test set.

(a) Training data

As previously mentioned, we mainly focused on the training of our models with unsupervised learning. There is a lot more data explaining the primary structure of the protein, since this is historically very easy to find, compared to finding secondary structure.

Since we wanted to replicate the work of UniRep, we also chose to use UniRef50 as our training dataset. [16] This dataset contains the primary structures of roughly $\tilde{2}7$ million proteins. Due to hardware constraints, we have decided to randomly sample 100,000 sequences from the same dataset, and use this as our training data.
Since we do not know how the data has been generated, we chose to randomly sample from the data, instead of taking 100k adjacent samples. We did this to avoid any kind of bias the data could have. For example, the first 100k rows of the dataset could potentially only contain one kind, or simpler and easier to process proteins. This could have an effect on the performance of the network.

UniRep removed proteins containing amino acid symbols (X, B, Z, J), and sequences longer than 2,000 amino acids. [2] In our case, due to limited hardware and computer power, we chose to remove all sequences longer than 500 amino acids, and sequences containing the (X, B, Z, J) amino acids as well. The sequences containing (X, B, Z, J) are removed because they are considered "non-canonical", meaning that some of them are placeholders etc.

With the sample reduction, we ended up having roughly 78,000 protein sequences in our training set. We realize that only looking at sequences with a maximum length of 500 means that we have poten-

tially introduced some bias into the network. This bias may make itself visible when predicting longer proteins, where the model may think that the sequence has to end at around 500 amino acids. We argue that in our case at least, this tradeoff is worth it because training is significantly faster, especially for the LSTM.

The primary structure consists of amino acids. Each amino acid is represented as a char $\in \{A, C, D, E, F, G, H, I, K, L, M, N, O, P, Q, R, S, T, U, V, W, Y\}$. Meaning that each protein from the dataset is represented as a sequence of chars, with arbitrary length. Since the length of each sequence varies, we've added a padding element (padding element is '-'); padding this element on all sequences, resulting in all sequences has a 500 feature-length. Figure 18, shows the frequency of each amino acid in the whole dataset.

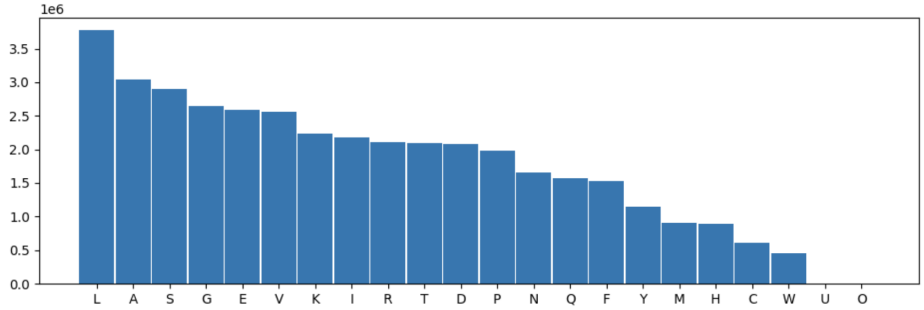

Figure 18: Histogram of amino acid frequency

Our models use embeddings, which is a method of converting a list of one-hot indices, to corresponding word embeddings. In our case, this meant that we had to transform our sequences into one-hot indices. A one-hot encoding is a way to represent the data, through a vector representation only containing {0,1}. Say each protein sequence only contain three amino acids (A, C, D), these amino acids will have the following one-hot encodings.

$$A = \begin{bmatrix}1\\0\\0\end{bmatrix}, C = \begin{bmatrix}0\\1\\0\end{bmatrix}, D = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

If we were looking at four amino acids (A, C, D, E), the one-hot encoding would look like:

$$A = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, C = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, D = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, E = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

We are having working with 22 amino acids and a pad-value each amino acid will be represented 1x23 onehot encoding. The onehot index describes at which index the 1 is placed. thus if we define the onehot index of any input as h, we get:

$$h(A) = 0 \tag{52}$$
$$h(C) = 1 \tag{53}$$
$$h(D) = 2 \tag{54}$$
$$h(E) = 3 \tag{55}$$

This means our data gets transformed to one-hot indices before we can use it in our models.

(b) Labelled test sets

Since our models' performance is not measured directly on the same kind of data that we trained on, we have two datasets to use as test sets. These datasets labeled, in contrast to the training set. The first dataset has labeling detailing the structural classification of each protein. [9] The second dataset has labels describing the stability of the given proteins. [15] Both of these datasets contain the proteins' primary structure in the same format as the training dataset.

(b).1 Structural classification

We do not use this dataset for quantitative performance analysis. Instead, we put the models' representation of the dataset through a TSNE dimensionality reduction to see whether the different structural classifications are clearly separated. We do also note down the next token prediction accuracy of the LSTM for this dataset, but this is not necessarily a representative parameter. The data is put in several classes, as follow:

- a: Alpha proteins
- b: Beta proteins
- c: Alpha/Beta proteins
- d: Alpha+Beta proteins. Not used
- e: Multi-domain proteins
- f: Membrane proteins
- g: Small proteins

(c) Protein Stability data

The stability dataset is also labeled dataset containing previously explained protein sequences. In this dataset, the label explains the stability of the protein. This label is explained with some real number value, which explains how stable some protein is. This dataset we use, to check whether or not our models have learned enough of the underlying structure of the proteins, to be used to check the stability of the protein.

4. Models and Architecture

(a) Model Selection

Choosing the right models, can in some cases be difficult. When training a neural network, there are a lot of hyperparameters that should be taken into consideration. in the pursuit of creating the most optimal neural network, it leads to a long row of hyperparameters which should be optimized. This includes the number of hidden layers, hidden neurons per each layer, learning rate, and regularization parameters.

There is no method for choosing an optimal architecture. A good architecture depends on the task - what are the model trying to achieve. Neural networks come in different forms, and each can vary a lot from each other. Tuning these hyperparameters, by trying every single combination, trying to find the ones that yield the best result, is a too comprehensive task for most computers.

in our case, we have experimented with two models, a simple bottleneck CNN, and an LSTM RNN where we are replicating the work done in unirep [2]. During this process, we used our expertise in neural networks, in choosing the correct hyperparameters for the task.

(b) LSTM reproduction of unirep

Going into this project, our first goal was to reproduce the results of the Unirep paper [2]. In most respects, we used the same architecture for our LSTM network. There are a few differences though. They used a 1900 feature, single hidden layer mLSTM, whereas we used a standard LSTM with 600 features and 2 layers with 0.5 dropout. They also performed some slightly different preprocessing of their data, though this should not have a large effect on the model's performance; They trained on sequences that were up to 2000 amino acids long, while we only trained on sequences with 500. Due to hardware constraints, we could also only load a dataset with 78k sequences, while they trained on 24 million sequences for three weeks on four separate GPUs. Meanwhile, our model that has trained the longest only trained for 8 hours, on a single MX250 laptop GPU.

(c) CNN architecture

(c).1 convolutional autoencoder

We also experimented with a CNN model, in the structure of a convolutional autoencoder. This model introduces encoding layers that reduce the dimensionality of the input x, down to a smaller latent representation z. From the latent representation, we increase the dimensions in decoding layers, with the goal of reconstructing $\hat{x}$, which is as close to the original input x. By doing this, the model should learn something which captures more properties of the data - more than a linear reduction method would.

We start by introducing embeddings which embeds our data from a 23 channel dimension to a 12 channel dimension. The reduction of dimensions consists of 3 encoding convolutional layers. Since the features of the data are one-dimensional, we're using one-dimensional convolutions. Each of these layers all uses the ReLU activation function, to get more reliable data. This counts for all layers in the network unless the last encoder layer. We do not use ReLU here to make z as general as possible. The two first encoding layers consist of average pooling, and the last consists of one max pool - reducing the dimensions to a latent space z.

From this point, our latent representation z is fed into the decoder. In the decoder, it goes through the same process, of 3 convolutional layers, in which we just increase the dimensionality back to the original size.
In each of the convolutional layers, we use convolutions with kernel_size=5 with stride=1, by having a kernel size 5 we increase our perceptive field, this means that each convolutional layer outputs greater amounts of amount of data since it's looking at more elements at a time. Due to the boundaries of the kernel, we use padding to avoid dimension loss.

Once the the the model reconstructed the original size, we use two extra convolutions, one with kernel_size=3 and one with kernel_size=1, both with stride=1. We do this to ensure that the model preserves the relation between the channels, once the final dimension is reached.

5. Experiments

In this section, we'll discuss some experiments that we have performed with the two different models. These experiments aimed to see which hyperparameters have significant effects on how well the models train. The models will primarily be evaluated based on how good their representation of the proteins is. To this end, we use the stability dataset described earlier. For each model, we then train a simple linear regression model on the model's representation of the proteins, against the given stability scores. How well this does is measured using spearman's rho.

(a) LSTM

(a).1 Layers vs no layers

Stacking LSTMs is a way to increase the non-linearity of the network. The idea is similar to increasing the depth of conventional neural networks. In a stacked LSTM, there are two or LSTM cells per element in the input sequence. The way this works is that the hidden state of the first cell is passed along as the input to the next cell and so on. For this experiment we have trained two models with the following identical hyperparameters:

- Character embedding size: 30
- Starting learning rate: $8e-4$
- Learning rate schedule: Multiply by 0.2 every 5 epochs
- Hidden layer size: 512
- Training time: 30 epochs

The only difference between these models is that one has two layers, and the other has only one.

We decided to perform this experiment because increasing the non-linearity of simpler networks has been shown to make them significantly easier to train. Since proteins have such complicated structures though increasing the non-linearity might help the network learn this structure better.

While adding layers to the network does increase the non-linearity it also makes the training significantly slower. Because each layer is essentially an entire LSTM in and of itself, going from one to two layers actually doubles the training time for the same amount of epochs. We hypothesize that due to the increased non-linearity, the network with two layers will learn a better representation of the protein structure.

(a).2 Dropout vs. no dropout

Because LSTMs are structured quite differently than many other networks, there are several different ways to apply dropout to them. [10] Pytorch has only implemented a simple version though, that requires stacked a network with more than one layer. With this implementation, the dropout happens on the output from the first layer that goes through to the second. For this experiment we trained two models with the following identical hyperparameters:

- Layers: 2
- Character embedding size: 30
- Starting learning rate: $8e-4$
- Learning rate schedule: Multiply by 0.2 every 5 epochs
- Hidden layer size: 512
- Training time: 30 epochs

The only difference was that one of the networks had a dropout value of 0.5 during training, corresponding to 50% of the features.

This test was laid out because in conventional fully connected networks we know that dropping out part of the features can significantly increase the robustness of the network. The reason that dropping information is that some of the easier features that help with classification can be dropped out, helping it learn from some features that may be less immediately informative. We wanted to test how much of

an effect this has on an LSTM.

We hypothesize that adding a dropout layer will increase the robustness of the network on new data. Because of this, we expect the model with dropout to get higher accuracy on both next token prediction and a higher spearman correlation.

(a).3 Feature size

The feature size is the hyperparameter which controls the size of the internal layers of the LSTM. Increasing this should mean that the network can learn more complex functions. However, it also increases training time in a linear way. The increase is linear because, for every feature you add, you have to backpropagate through one more. For this experiment we've trained three models with the following identical hyperparameters:

- Layers: 1
- Character embedding size: 30
- Starting learning rate: $8e-4$
- Learning rate schedule: Multiply by 0.2 every 5 epochs
- Training time: 6 hours

The only difference between the models is the hidden layer size or feature size. We trained three models, one with 256 features, one with 512 features, and one with 1024 features. Since training time is the most sparse resource for us, we decided that we would compare the results for equal real-time training time, not the number of epochs as with the other experiments.

For this experiment, we expect there to be some middle ground between the size of the model and how well it learns the representation. Since protein structures are very complex, we hypothesize that the larger models will perform better, even though they are not able to train for as many epochs.

(a).4 Learning rate

The learning rate of any kind of neural network is an incredibly important hyperparameter. Setting a learning rate that is too high will cause the model to have trouble converging because it overshoots any small minima. Setting it too low on the other hand will make it take a very long time to converge. For this experiment we have trained two models with the following identical hyperparameters:

- Layers: 1
- Character embedding size: 30
- Starting learning rate: $8e-4$
- Hidden layer size: 512
- Training time: 30 epochs

The only difference between the networks was that one of them had a learning rate schedule. This schedule meant that every 5 epochs, the learning rate would be multiplied by 0.2.

We decided to lay out this test because we noticed that when training, the training loss would vary significantly as training went on and that this variance increased significantly when training for long periods of time. We thought that reducing the learning rate could potentially alleviate this problem slightly. A learning rate schedule is a good way to do this because simply lowering the learning rate significantly at the beginning would mean that the model would take a very long time to reach the

point where it becomes helpful.

We hypothesize that the reason the loss is fluctuating so much is because of a learning rate that is too high, effectively "jumping over" a local minimum. Thus, decreasing the learning rate as training goes on should cause the model to converge to a better solution faster.

(a).5 Minimum loss model vs last model

This is a slightly unconventional experiment, in that it came about as a response to the same problem that was described in the learning rate experiment. The training loss begins varying significantly as training goes on. For this experiment, we use the models from all the previous experiments. Each model in the previous experiments has two "versions". One version is the "last" trained model. Meaning it is not necessarily the model that scored best on the training and validation sets. But the result of the last optimization in the training. The second version is the model that performed best during training. These will be referred to as the "final" and "minloss" models respectively.

The idea behind this experiment is that we wanted to see whether or not picking the minloss model would have any significant effect on performance. One could reasonably argue that when the difference between these models is so large, it could be because the minloss model is overfitting. Alternatively, it could be because the final model is underfitting.

Based on the fact that the models do not achieve very high next token prediction accuracy, we hypothesize that using the minloss models will improve both this accuracy and spearman correlation scores. This is because we suspect that the models are already underfitting.

(b) CNN

(b).1 Latent representation

Using the CNN we reduce the dimension of each sequence to some latent representation z. Using z in combination with t-sne, the goal is to get a two-dimensional representation of each sequence, showing that some separation could be seen in between the secondary structures of the protein.

while experimenting with this model, we worked with a learning rate of $lr = 1e - 4$ and 3 layers in each encoding- and decoding phase. We experienced early on that introducing a too steep bottleneck would yield worse results. So this experiment's goal is to see how much the latent space size, means to the final result.
The way our experiments vary from each other is how much we reduce the feature-dimension. Our two experiments work with the latent feature-dimensions 50 and 100. thus the z will either have a size of 2x50 or 4x100. the latent representation will then get flattened and used in t-sne.

The result of the CNN model will be evaluated using the model with the minimum loss, and the fully trained model.

This is a very simple test, with a very simple model. We are mainly using the CNN to compare how well it compares to the LSTM.

6. Results

In this section, we will go over the results of the various experiments we have concocted. The results will be evaluated in two separate ways. For each experiment, there are two models, the "final" model that was saved after all training was finished, and the "minloss" model which was saved during training when the model achieved the lowest loss. With the last experiment being the exception, all results

shown are from the "final" models.

The first evaluation will be on a structural classification dataset, which will be a qualitative analysis in which we perform t-distributed stochastic neighbor embedding (TSNE) dimensionality reduction on the data, and see whether it is able to cleanly separate the different types of protein structures. TSNE is preferred here over something like principal component analysis (PCA) due to this non-linearity of the reduction. This evaluation will be performed on the structural classification dataset. While using a classification method like k-nearest neighbors (KNN) might seem intuitive to quantitatively evaluate this, the non-linearity means that neither distance nor density is preserved between data points. We will also look at the next token prediction accuracy for the LSTM, but because we want to quantify how well it represents the proteins, this is not necessarily a good way to do that. It will mostly be used for comparison to see if there are any interesting insights.

The second evaluation will be on the stability dataset. This evaluation will be quantitative, using Spearman's rank correlation coefficient. This coefficient measures the degree to which the relationship between two inputs can be described using a monotonic function. This means that having a high spearman correlation is equivalent to having a highly descriptive representation. It does not necessarily mean that the linear regression model we have trained provides good results in and of itself since the coefficient does not describe a 1:1 correlation. For this evaluation, higher values are better, up to 1. This can be seen in figure 19.

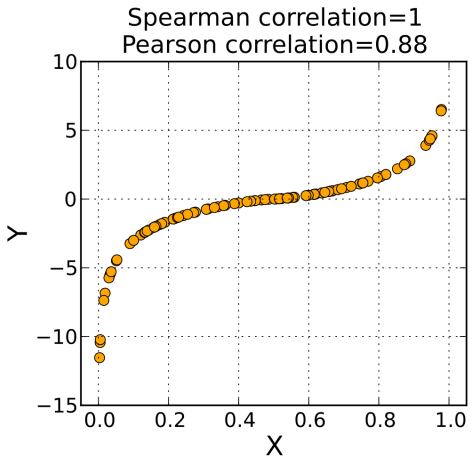

Figure 19: Graph showing that high spearman correlation does not necessarily mean a good score. Image source: [5]

(a) LSTM experiments

(a).1 Layers vs no layers

	Next token prediction accuracy	Test Loss	Spearman's rho
1-layer	12.43%	2.84	0.474
2-layer	13.03%	2.83	0.428

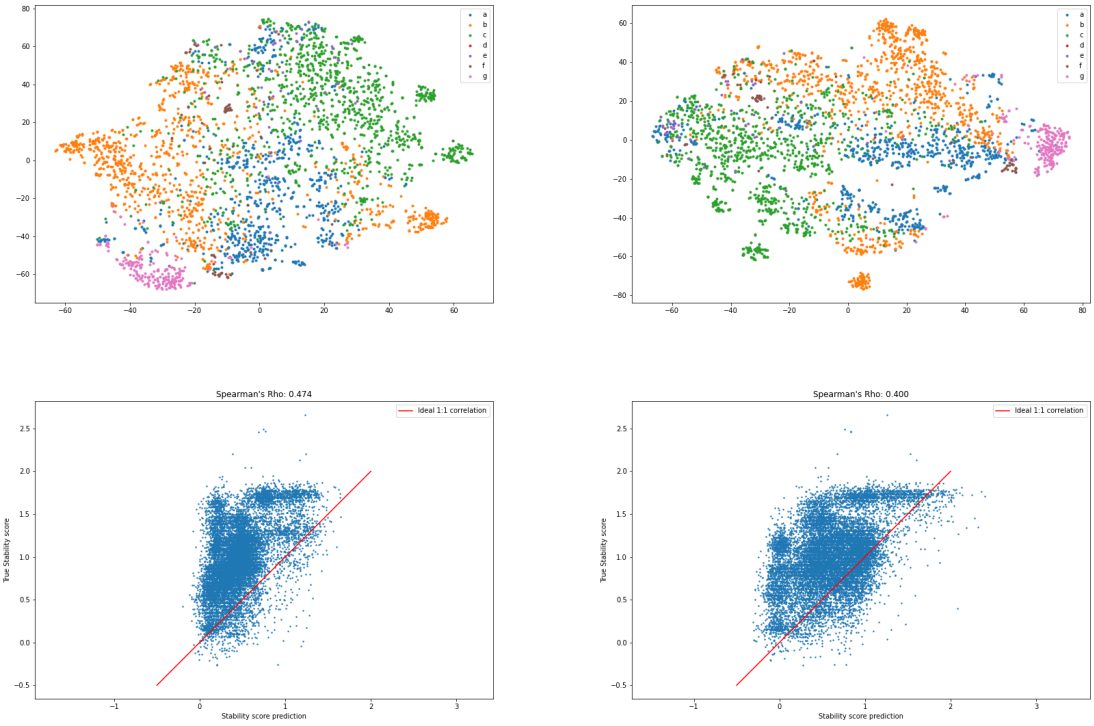

Figure 20: TSNE dimensionality reduction and Spearman's rho data plots of the two models. Left is 1-layer, right is 2-layer

	Next token prediction accuracy	Test Loss	Spearman's rho
No dropout	13.03%	2.83	0.400
50% dropout	13.02%	2.82	0.518

	Epochs trained	Next token prediction accuracy	Test Loss	Spearman's rho
256 features	60	11.41%	2.86	0.435
512 features	30	12.43%	2.84	0.474
1024 features	10	14.37%	2.79	0.507

	Next token prediction accuracy	Test Loss	Spearman's rho
No schedule	13.34%	2.83	0.605
With schedule	12.43%	2.84	0.474

(a).2 Dropout vs no dropout

(a).3 Feature size

(a).4 Learning Rate

(a).5 Minimum loss model vs last model

For this experiment the Spearman correlation and TSNE plots can be found in the appendices. Due to the nature of this experiment, the tables below also contain the results of all the previous experiments. The results can be seen in tables 3 and 4.

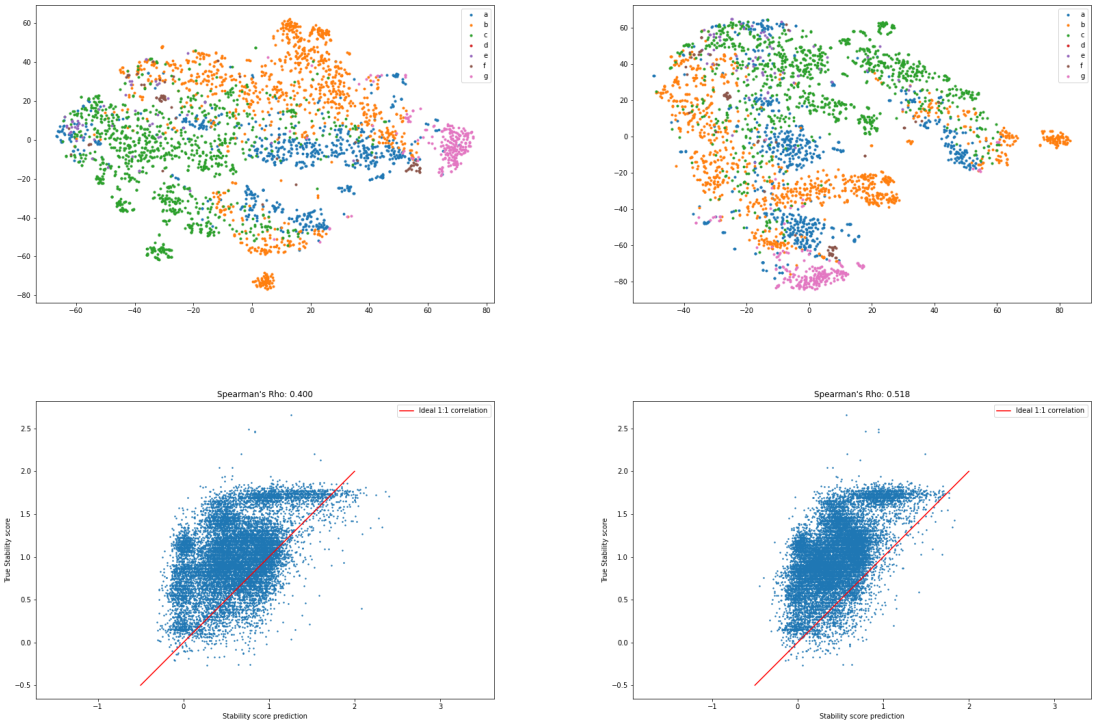

Figure 21: TSNE dimensionality reduction and Spearman's rho data plots of the two models. Left is without dropout, right is with 50% dropout.

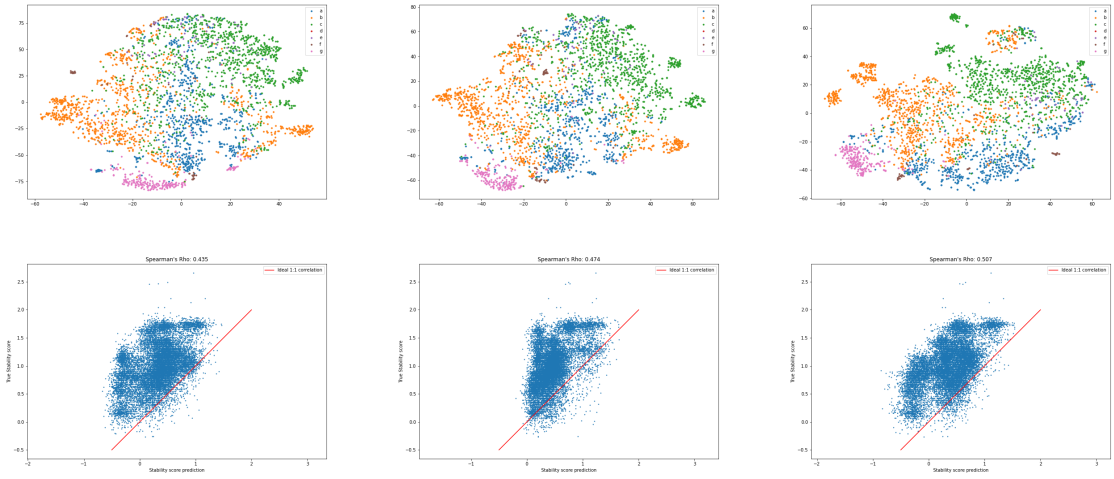

Figure 22: TSNE dimensionality reduction and Spearman's rho data plots of the two models. Left is 256 features, middle is 512 features, right is 1024 features.

(b) CNN

Training the model with different sizes of latent dimensions yielded some different results, in different aspects of the tasks the model had. We can see how well the model performed on the different tasks in Table 5

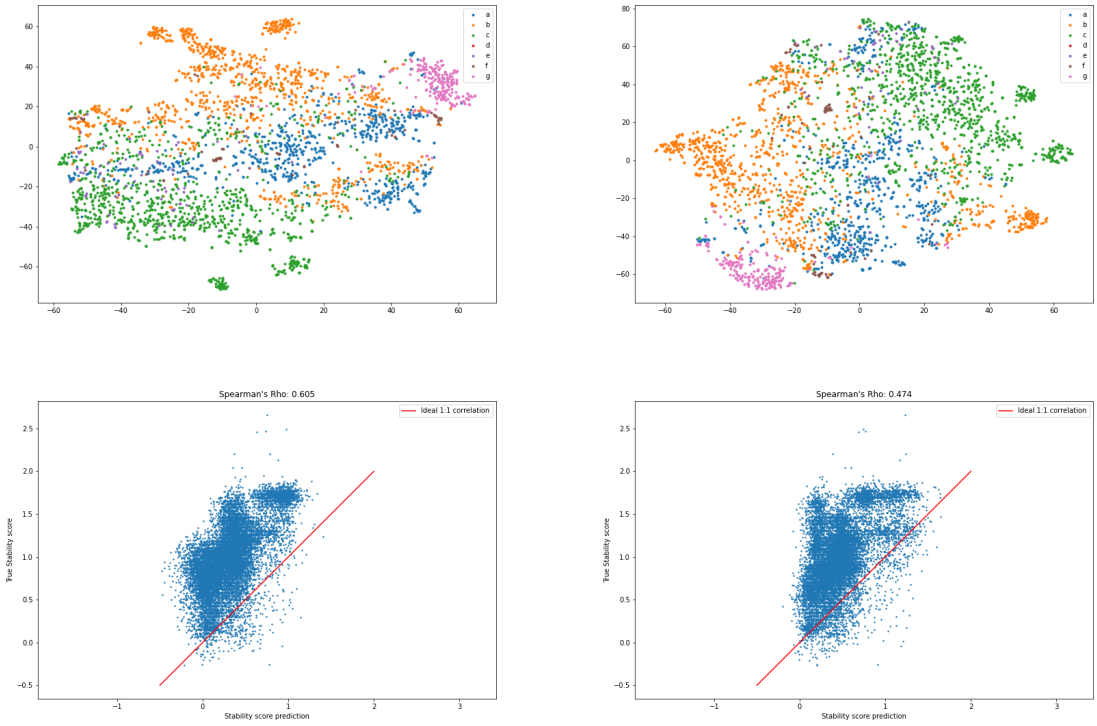

Figure 23: TSNE dimensionality reduction and Spearman's rho data plots of the two models. Left is without a learning rate schedule, right is with the schedule mention in the experiments section.

LSTM Results	Final models		
	Next token prediction	Test Loss	Spearman's rho
2-layer, 50% dropout, 512 features	13.02%	2.82	0.518
2-layer, no dropout, 512 features	13.03%	2.83	0.400
1-layer, no lr schedule, 512 features	13.34%	2.83	**0.605**
1-layer, 256 features	11.41%	2.86	0.435
1-layer, 512 features	12.43%	2.84	0.474
1-layer, 1024 features	**14.37%**	**2.79**	0.507

Table 3: Results for final models

LSTM Results	Minloss models		
	Next token prediction	Test Loss	Spearman's rho
2-layer, 50% dropout, 512 features	13.03%	2.82	0.539
2-layer, no dropout, 512 features	13.78%	2.81	0.427
1-layer, no lr schedule, 512 features	13.36%	2.83	0.592
1-layer, 256 features	11.40%	2.87	0.424
1-layer, 512 features	12.43%	2.85	0.598
1-layer, 1024 features	**14.22%**	**2.80**	**0.627**

Table 4: Results for minloss models

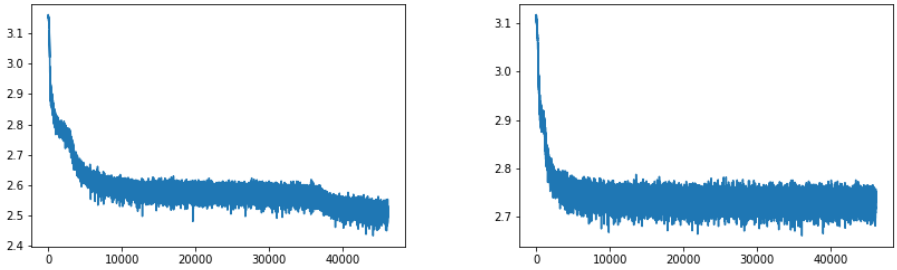

Figure 24: losses of the of cnn trained with latent dimension 4x100(left) and 2x50(right)

CNN Results	Final model		Minloss model	
	Reconstruction accuracy	Spearman correlation	Reconstruction accuracy	Spearman correlation
4x100	20.58%	0.340	20.39%	0.145
2x50	14.69%	0.351	14.73%	0.278

Table 5: Results from the CNN

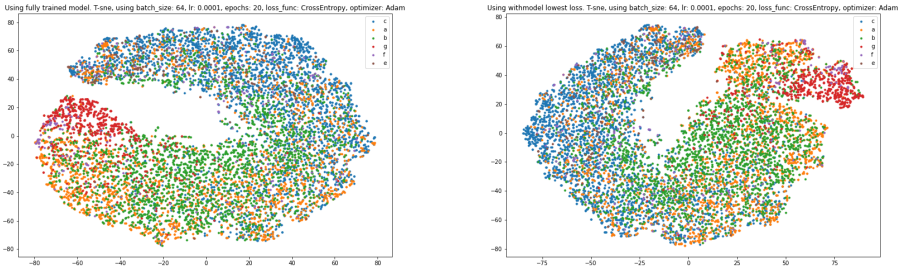

Figure 25: Plot showing secondary structure seperation with latent dimension 2x50. Fully trained model(left), min loss model (left)

Figure 26: Plot showing secondary structure seperation with latent dimension 4x100. Fully trained model(left), min loss model (left)

7. Discussion

In this section, we'll be discussing our experiments and their results. We will also be discussing the models and how they perform compared to each other with the task of finding any representation of the proteins' structure. We will discuss our final model, and also how our models' results compare to the results from UniRep [2] and Tape [14].

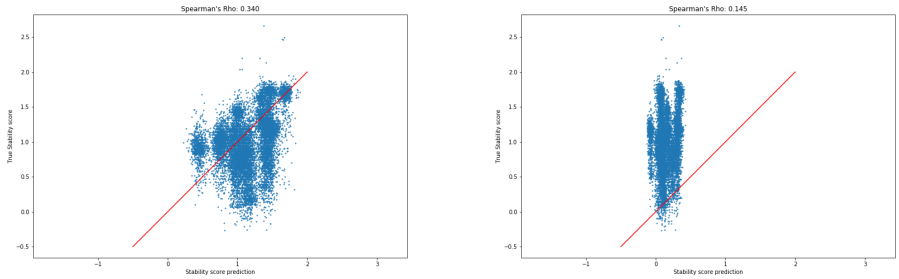

Figure 27: Graph showing the spearman correlation from model with latent dimension 4x100. Fully trained model(left), min loss model (left)

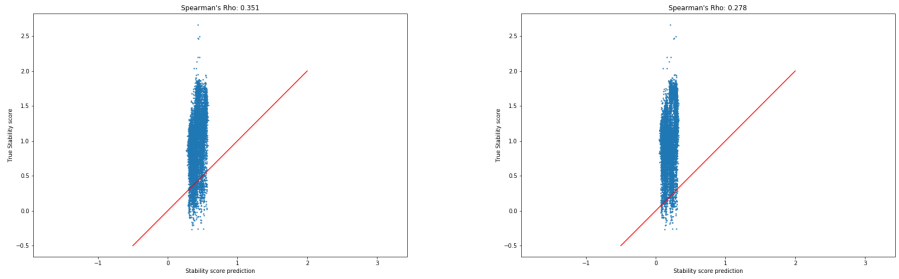

Figure 28: Graph showing the spearman correlation from model with latent dimension 2x50. Fully trained model(left), min loss model (left)

(a) LSTM Experiments

Because of the qualitative nature of the TSNE plots, it is difficult to judge how they compare to each other. Even between models with large differences in the Spearman correlation scores, the differences between their TSNE plots are quite subtle. They do all provide a quite good separation between the different categories though.

(a).1 Layers vs no layers

In this experiment we tried to figure out whether or not stacking an LSTM would help increase the performance of the network. The idea was that adding this extra non-linearity would make it able to find better patterns in the structures of these proteins and thereby get better results. The results show that simply stacking them without anything else does not provide any kind of improvement, quite the opposite in fact. The resulting Spearman's rho values of 0.474 for the single-layer and 0.400 for the double-layered models attest to this. One possible explanation for this could be that the model begins overfitting on the data when stacked in this way. It is interesting to note that while the 1-layer model gets a better Spearman correlation score, its next token prediction accuracy is slightly worse. Considering the fact that a 2-layer LSTM takes roughly twice as long to train as a 1-layer LSTM, stacking them like this does not seem like a good way to increase the performance of an LSTM for this task.

(a).2 Dropout vs. no dropout

In this experiment, we tested whether adding dropout between the layers of a stacked LSTM would help increase the robustness of the network. Now, considering the results from the previous experiment, we weren't expecting a significant improvement over the baseline single-layer model. The results were very interesting however; the next token prediction accuracy was basically the same, with only a 0.01%

difference on the test set. Meanwhile, the Spearman correlation score improved quite significantly, scoring 0.518, compared to the 0.400 of the 1-layer model. This may be because the dropout forced the network to learn a better general representation of the structure, rather than learning a few signs that may not always be there. This model takes a slightly shorter time to train than the network without dropout. Because of this, adding dropout between layers of a stacked LSTM seems to be a great way to increase both the performance and robustness of the network.

(a).3 Feature size

It is to be expected that reducing the number of features a model can use will limit how much it can learn. For this experiment, we decided to see whether the increased amount of data a smaller model can be trained on due to the increased speed would make up for that lack of features. For reference, the smallest model at 256 hidden features trained for 60 epochs, while the largest at 1024 hidden features only trained for 10. The model with 512 features trained for 30 epochs. We expected that there would be some middle ground between speed and the number of features, but actually saw that increasing the features was a boost to performance in all the cases we tested. Since we were limited by hardware, going over 1024 features was not possible. An interesting note is that there is a very large difference between how the next token predictions actually look. It seems like the 256 model has not learned much more than predicting which amino acids are the most common. Meanwhile, the larger 1024 model seems to have learned significantly more about the structure of the protein and varies its predictions much more. An example of this can be seen in Figure 29. This can also be seen in the Spearman correlation scores, in which the 1024 model scores significantly higher than the others.

Figure 29: Example of how different the predictions look. Left column uses the 256 feature model, right column uses the 1024 feature model. The model predictions are in the top row, the corresponding labels are in the bottom row.

(a).4 Learning rate

We observed quite early on that there was a very large variance in the loss values during training, and that the variance increased significantly as training went on. Thus, for this experiment, we used a learning rate scheduler to hopefully combat this phenomenon. An example of this variance can be seen in Figure 30. The results were very unexpected, however. First of all, the variance in loss did not decrease significantly, though it did seem to cause slightly fewer large outliers. The model that did not have a learning rate scheduler actually performed significantly better, to the point where it is the highest-scoring of the final models, with a Spearman correlation of 0.605. Meanwhile, the counterpart with a scheduler only scored a 0.474. We do not think that we can draw to the conclusion that this result means that using a scheduler makes the model perform worse. We think that the reason the model without scheduling performed so much better was that the scheduling was too aggressive. It is a tough parameter to adjust however since it requires you to know roughly where it is necessary to lower the learning rate for better results. As we see in the next experiment however, the scheduler may actually have caused the model to perform significantly better in the minloss version, however.

(a).5 Minimum loss model vs last model

Due to the large variance in the loss values, there are a few batches during training in which the model has significantly better loss values than the average. This led us to save the best of these models, to try and see how well they performed compared to the final models. The results were quite interesting, to say the least. In only two of the six models, we trained for experiments did the minloss model performs

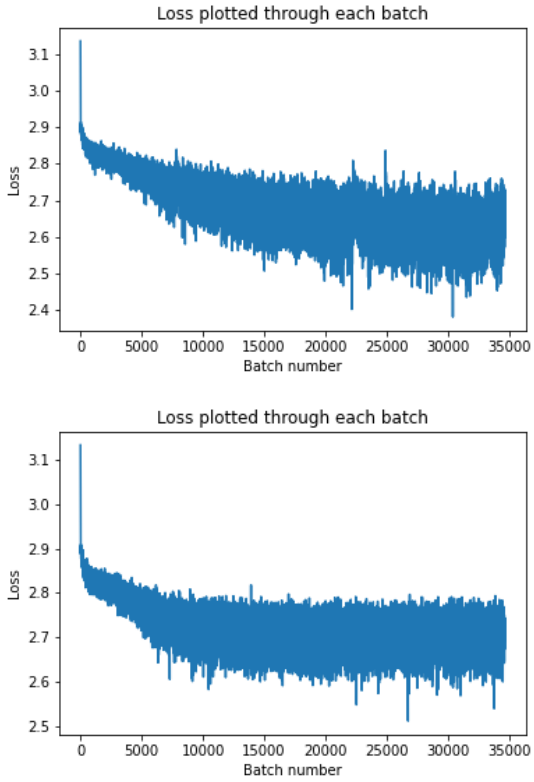

Figure 30: Example showing the increasingly large variance while training. The plot on the right uses a learning rate schedule where it multiplies the learning rate by 0.2 every 5 epochs.

worse than the final models. These two models were the 256 feature model with scheduling, and the 512 feature model without scheduling. In both cases the Spearman correlation was only slightly worse, however; decreasing by 0.011 and 0.013 respectively. The rest of the models experienced moderate to significant improvements in their Spearman correlation scores however, with the largest jump increasing the score by 0.124. We can only guess as to why two of the models performed worse, but we have a hypothesis that the reason the other models saw such a large increase was helped along by the learning rate scheduler.

(b) CNN Experiments

In the process of constructing the CNN, we had to do a lot of hyperparameter tuning before coming up with the final model. In the pursuit of getting the best performing model, we started with a sanity check. We did this by only introducing the bottleneck to the channel dimension, which led to a 100% accuracy in reconstruction. Once this was done, we started adding a bottleneck to the feature size as well. At this point, we started with tuning some of the hyperparameters: the number of hidden layers, and the sizes of each hidden layer.

It's important to note that the reconstruction accuracy is not representative of the model's performance on the separation or stability prediction tasks.

As can be seen in Table **??**, the model with a high latent dimension had a noticeably better loss than the one with a lower latent dimension. This makes good sense since the model with a greater latent dimension has more data to use in the reconstruction. It does mean that the model does not need to learn the structure of the protein as much, however. Since the loss was based on the model's ability to

reconstruct the image; it is no surprise that the one with higher dimensionality reduction, had better reconstruction accuracy.
Figures 25 and 26 show the structural plotting of the model's latent representation run through a TSNE dimensionality reduction. The model does find some separation between the different structural classes, but it does not seem to be nearly as clean as the LSTM results. As can be seen, both models find some separation between classes (C, B, G) with some overlap. The rest of the classes are mostly spread out between the groups. This result indicates that this model is not very good at finding much structural separation.

We experienced that when increasing the number of hidden layers, the loss would at some point start increasing. Yet, turning the learning rate down didn't solve the problem. Thus, the problem could be that the probability distribution in the cross-entropy loss function converged to all zeros. This could be solved by adding some epsilon in the calculation of the probability distribution. We ended up settling with the layers explained in the CNN architecture section. Thus, avoiding the increasing loss.

We started to tune the sizes of each hidden layer. Meaning the way the dimensionality changed over each specific hidden layer. This was simply a trial and error task, leading to what yielded the best results.
We only had one experiment with CNN, consisting of changing both the latent dimension and the channel size, which does not cover the full potential of a convolutional network. When checking how much impact the latent dimension had on the results, we should have made experiments with only reducing and increasing either the channel size or the feature size. Doing so would have made it easier to tell how each parameter affects the end result.

In general, throughout these experiments, the results from the minimum loss model doesn't perform that differently compared to the fully trained model.
During the trial and error phase, we failed to document how each of the variable tunings affected the model. This should have been documented since it would have made it easier to understand our path towards the final model.
The CNN ended up performing poorly compared to the LSTM. Our CNN was a relatively simple model, trying to solve a very complex problem. With the model only being able to reach a Spearman correlation score of 0.351, the results are not nearly as good as we had hoped. However, we can't rule out whether or not a CNN can be used to solve this specific problem. We chose to use a bottleneck type of architecture, but this might not be the optimal way of learning a useful representation of the structure.

(c) CNN vs LSTM

In this project, the LSTM clearly outperforms the CNN in finding any representation of the structure and in predicting the stability scores. This means that the LSTM has learned more insight into the internal structure of proteins. This does not necessarily mean that using a CNN does not work as well for protein structure prediction, however.

There are, in fact, several good reasons to use a CNN over an LSTM. One of the main reasons is the significantly increased parallelization of CNNs. Because LSTMs are sequential, it is necessary to finish calculating each sequence in order. This is not the case with a CNN, because each element is independent of the others in this context. Because of this, a CNN can be trained significantly faster than an LSTM.

Without special architectures or other measures, a CNN cannot handle variable-length sequences. This is of course something that an LSTM handles with ease, due to the recurrent design.

(d) Comparing Unirep results

When looking at both the TAPE paper [14] and the Unirep paper [2], they have some conflicting information regarding the Spearman correlation they achieve when predicting stability scores. The Unirep paper says that they get a Spearman correlation score of 0.59, but we could not find the dataset they used to get this on. We used the TAPE dataset, in whose paper they say that Unirep scores a significantly better 0.73. With our best model's score of 0.627 we would consider this a pretty good result. We also generated a TSNE plot with the output of this model, which can be compared to a similar Unirep model in Figure 31

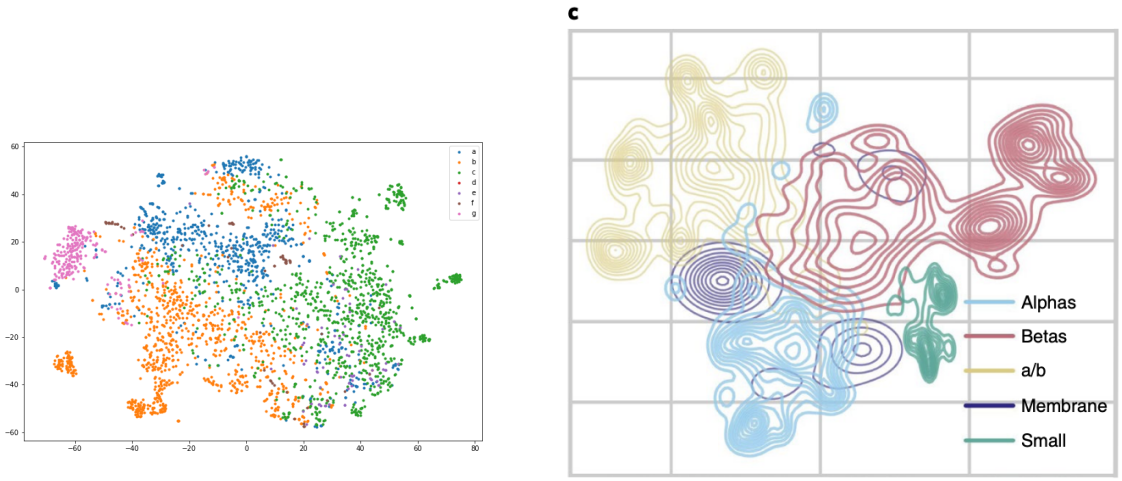

Figure 31: Left image is our best model's TSNE dimensionality reduction, the right image is an image taken from Unirep [2]

(e) Discussion of final model

Using the results gathered from the experiments, we set out to create one that was better than what we got there. We saw that there were quite a few things that improved the performance of the model. The things we decided seemed most crucial was adding a second layer with dropout, and increasing the number of hidden features in the model. Our model ended up with the following hyperparameters:

- Character embedding size: 30
- Starting learning rate: $8e-4$
- Learning rate schedule: Multiply by 0.2 every 5 epochs
- Layers: 2
- Hidden layer size: 600
- Training time: 15 epochs/8 hours

Unfortunately this model did not end up outperforming the best model from the experiments, and due to time constraints we could not train another. The resulting plots can be seen in Figure 32 and 33. The resulting scores can be seen in Tables 6 and 7. This results does make sense in hindsight. The very large increases in Spearman correlation did not happen in the 2-layer models, and this may be because the 2-layer model is less prone to this kind of increase.

	Final model		
	Next token prediction	Test loss	Spearman correlation
Last model results	13.68%	2.81	0.545

Table 6: Last model scores for final training

	Minloss model		
	Reconstruction accuracy	Test loss	Spearman correlation
Last model results	13.63%	2.81	0.478

Table 7: Last model scores for minloss

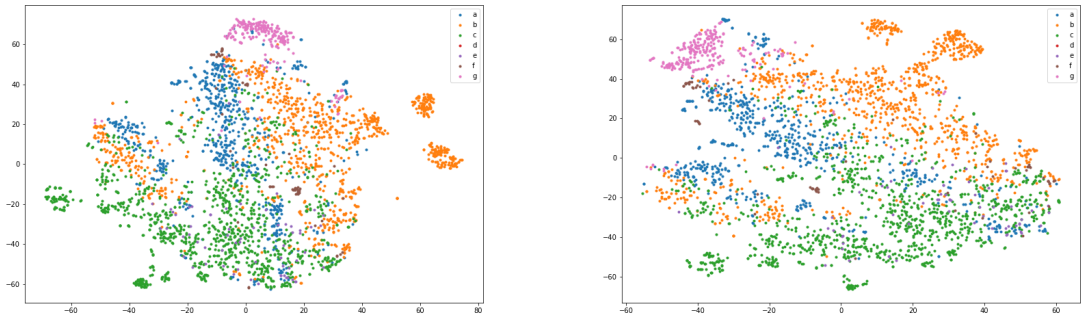

Figure 32: The TSNE plots generated from the last model. Left is final, right is minloss.

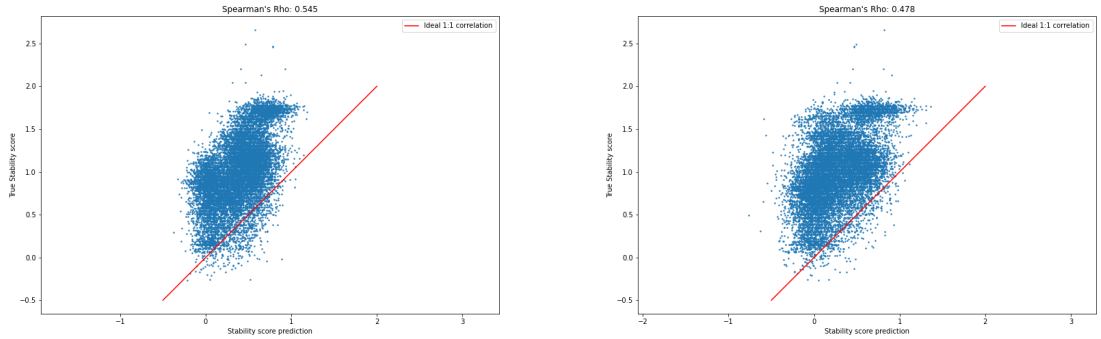

Figure 33: Spearman plots generated from the last model. Left is final, right is minloss.

(f) General discussion

(f).1 Spearman plots

One interesting thing to note about nearly all of the Spearman plots is that there seem to be several clusters of proteins. Unfortunately, the dataset with stability scores does not also have a structural classification or similar to compare with. It would be quite interesting to see whether some of the structural classes are more difficult to predict the stability of. This is especially clear in Figure 21. Another interesting note is that the better performing models seem to not have as many small clusters. This can be seen in Figure 23.

(f).2 Training loss

While working on this project, we noticed a weird phenomenon with both the LSTM and the CNN models while training. There seem to be large local minima at some specific points that nearly all the models hit during training. For the LSTM, this minimum seems to be when the model hits a training loss of around 2.8. If we stop a model that only got to this point, it will usually predict almost entirely the most common amino acid. One of the experiments only barely got past this point, as can be seen in Figure: 29. For comparison, the loss logs of this model, with 1-layer and 256 features and the 1024 model can be seen in Figure 34.

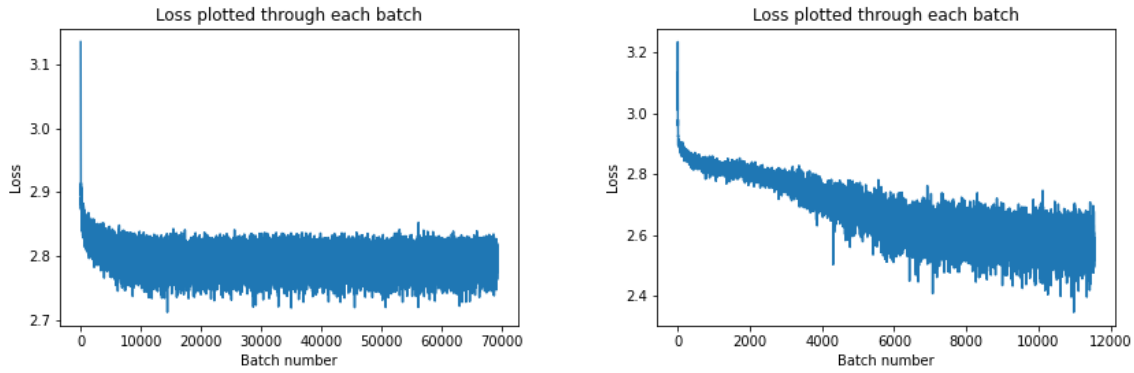

Figure 34: Example showing the very different loss logs. The left plot shows the 256 model, and the right shows the 1024 model. Note the y-axis values.

8. Potential future work

A potential improvement of the CNN could be working with dilated kernels. Dilated convolutional kernels are like regular convolutional kernels, but with some spacing between the values in the kernel. Dilated kernels have proved to significantly improve performance, compared to non-dilated counterparts in some cases. [6] This is because dilating the kernels makes the size of the receptive field increase exponentially, instead of linearly with non-dilated kernels. This could help the model learn some more long-distance dependencies more easily.

Instead of introducing a bottleneck as we do in the current CNN, potential future experiments could involve masking the input instead, and basing the training on how well it predicts the element based on the context around it. This method has proven to achieve high performance in many NLP tasks [7]. This mask could work by replacing the original token with another random token or a completely different value. This forces the model to learn some internal representation of the protein structure in a different, potentially more effective way.

One could also experiment with more complicated architectures, such as a Temporal Convolutional Network as described in the paper "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". [3] This architecture has two important distinguishing characteristics; that the convolutions do not take information from the "future" into account, and that it can take a sequence of any length and output a sequence of the same length. These two characteristics seem to make it uniquely suited to also be applied to protein structure prediction if applied similarly to an RNN.

The experiments performed with the LSTM network could be expanded upon significantly. One could try with more than two layers of LSTM, and see if this impacts the performance. While we ended up using a learned character embedding to represent the sequences, experimenting with the size of this

embedding could also lead to some interesting results. We have also realized that using such a high-dimensionality embedding may have hurt the performance of the mode, especially considering the fact that we haven't been able to experiment with it. The learning rate schedule experiment could also be expanded, by trying out different schedules. This is a difficult experiment though because intuitively the optimal schedule seems to vary significantly with other hyperparameters. We also only tested a dropout value of 50% between the 2-layer LSTMs, experimenting with only this value is perhaps slightly naive. So testing other values would likely provide better insight. One could also experiment with using an mLSTM instead of a regular LSTM.

Another interesting idea could be to work with a bidirectional LSTM. Since proteins are not sequential in the sense that one end always comes first, it makes sense intuitively that looking at it from both ends would help a model understand the structure better.

In this paper, we have only evaluated the models on their ability to predict the stability of proteins. Getting good results here may not necessarily mean that the models have learned representations that say a lot about other measures such as fluorescence, secondary structure, or homology, as has been suggested by the TAPE paper [14] as good benchmarks for protein structure representations.

9. Conclusion

In this paper, we first aimed to reconstruct the model described in the Unirep paper [2]. This went rather well, considering the significantly less hardware and training time available. Our best model scores a Spearman correlation within 0.11 of the results described in the Tape paper [14] and scores better by 0.03 compared to the values they reported in the Unirep paper. Our LSTM models also found good separation between the different structural classifications of the proteins in the Scope dataset. [9] Our attempt to create a CNN that could replicate these results did not go as well. It was only able to get a 0.351 Spearman correlation, and the structural classification plots did not have significant boundaries between the classes. This may be due to the simple model we utilized.

While writing this project we have become familiar with the fundamental of deep learning, and have learned a lot about the inner workings of both RNNs and CNNs. In the making of these models, we also got familiar with the basic terminology regarding proteins and amino acids. We gained this knowledge through both reading the papers cited throughout, and through practical experience, primarily using PyTorch to implement our models.

References

[1] Uniprotkb/trembl 2018_10. `https://www.uniprot.org/statistics/TrEMBL`, (Visited at 2020-06-03). 4

[2] Ethan C. Alley, Grigory Khimulya, Surojit Biswas, Mohammed AlQuraishi, and George M. Church. Unified rational protein engineering with sequence-only deep representation learning. *bioRxiv*, 2019. 4, 20, 22, 23, 31, 36, 39

[3] Shaojie Bai, J. Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. 03 2018. 38

[4] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, 2018. 4

[5] Wikipedia contributors. Spearman's rank correlation coefficient. `https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient`, (Visited at 2020-06-04). 27

[6] Greg Mori C.V.Jawahar, Hongdong Li. Computer vision - accv 2018, 2019. 38

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. 38

[8] Jimmy Lai Ba Diederik P. Kingma. Adam: A method for stochastic optimization. 2015. 20

[9] Naomi K Fox, Steven E Brenner, and John-Marc Chandonia. Scope: Structural classification of proteins—extended, integrating scop and astral data and classification of new structures. *Nucleic acids research*, 42(D1):D304–D309, 2013. 22, 39

[10] Adrian G. A review of dropout as applied to rnns. `https://medium.com/@bingobee01/a-review-of-dropout-as-applied-to-rnns-72e79ecd5b7b`, (Visited at 2020-06-04). 24

[11] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. 10

[12] Christopher Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, (Visited at 2020-06-03). 12

[13] Nallagoni Omkar. Activation functions with derivative and python code: Sigmoid vs tanh vs relu. `https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-sigmoid-vs-tanh-vs-relu-44d23915c1f4`, (Visited at 2020-06-07). 12

[14] Roshan Rao, Nicholas Bhattacharya, Neil Thomas, Yan Duan, Xi Chen, John Canny, Pieter Abbeel, and Yun S Song. Evaluating protein transfer learning with tape. In *Advances in Neural Information Processing Systems*, 2019. 31, 36, 39

[15] Gabriel J Rocklin, Tamuka M Chidyausiku, Inna Goreshnik, Alex Ford, Scott Houliston, Alexander Lemak, Lauren Carter, Rashmi Ravichandran, Vikram K Mulligan, Aaron Chevalier, et al. Global analysis of protein folding using massively parallel design, synthesis, and testing. *Science*, 357(6347):168–175, 2017. 22

[16] Baris E Suzek, Yuqi Wang, Hongzhan Huang, Peter B McGarvey, Cathy H Wu, and UniProt Consortium. Uniref clusters: a comprehensive and scalable alternative for improving sequence similarity searches. *Bioinformatics 31*, page 926–932, 2015. 20

10. Appendix

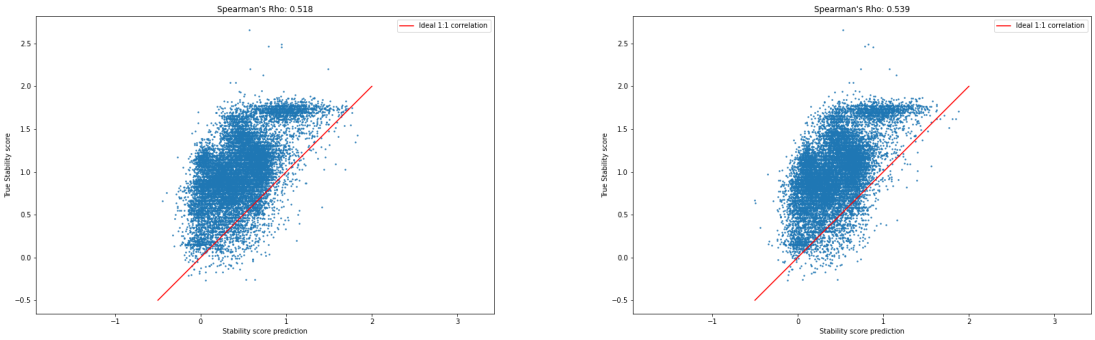

Figure 35: Final and minloss comparison of the Spearman's rho data plots of the 2-layer, 50% dropout, 512 feature model.

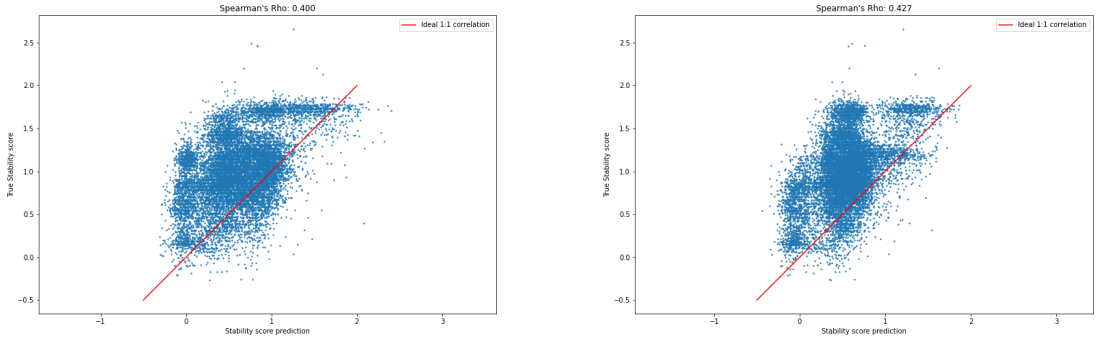

Figure 36: Final and minloss comparison of the Spearman's rho data plots of the 2-layer, no dropout, 512 feature model.

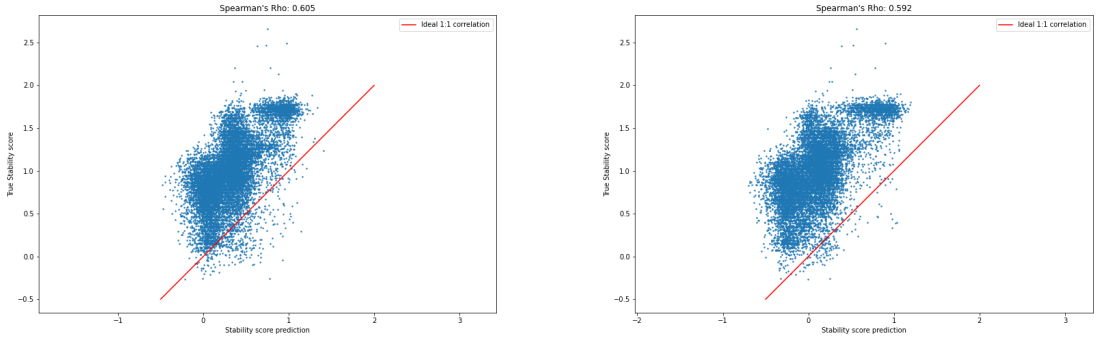

Figure 37: Final and minloss comparison of the Spearman's rho data plots of the 1-layer, no lr schedule, 512 feature model.

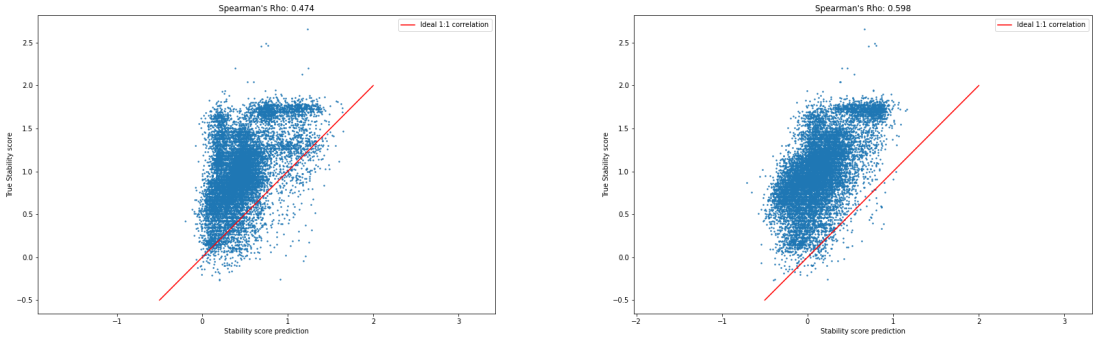

Figure 38: Final and minloss comparison of the Spearman's rho data plots of the 1-layer, 512 feature model.

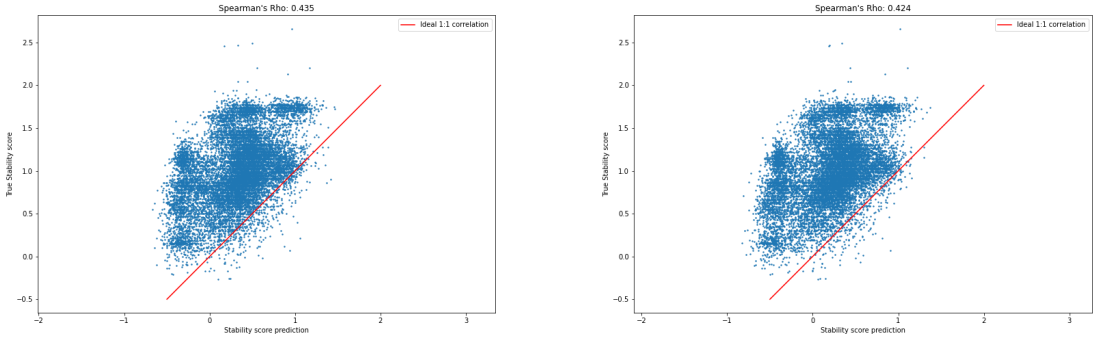

Figure 39: Final and minloss comparison of the Spearman's rho data plots of the 1-layer, 256 feature model.

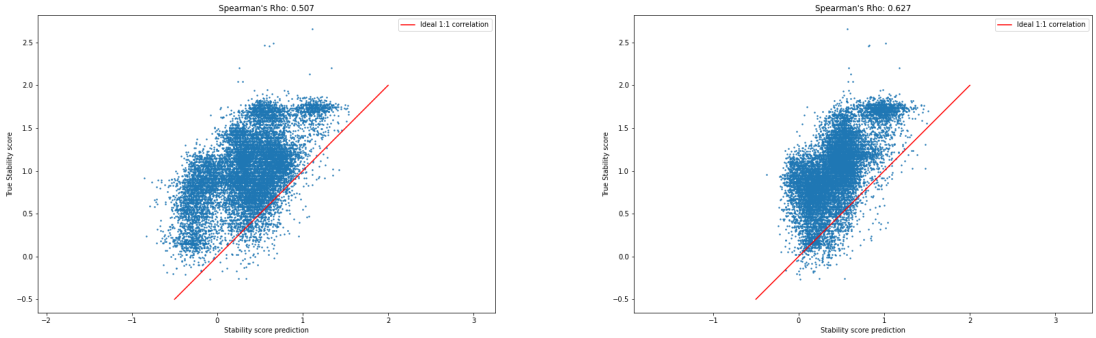

Figure 40: Final and minloss comparison of the Spearman's rho data plots of the 1-layer, 1024 feature model.

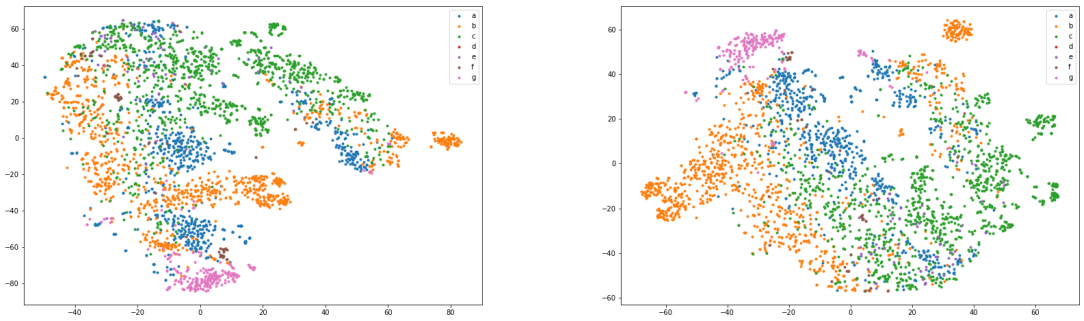

Figure 41: Final and minloss comparison of the TSNE dimensionality reduction plots of the 2-layer, 50% dropout, 512 feature model.

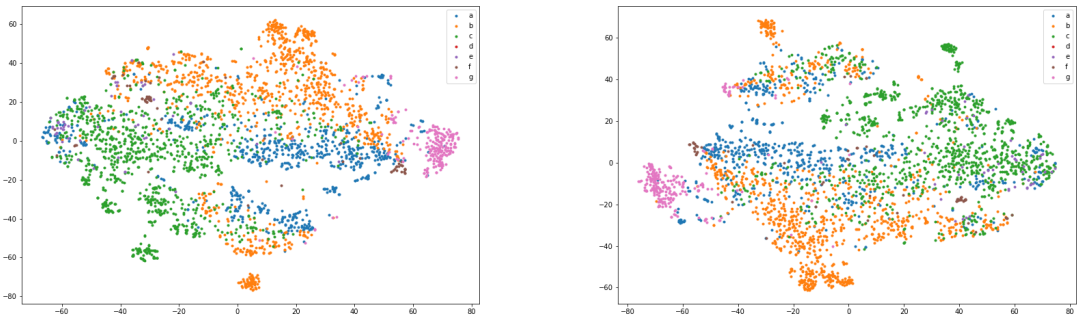

Figure 42: Final and minloss comparison of the TSNE dimensionality reduction plots of the 2-layer, no dropout, 512 feature model.

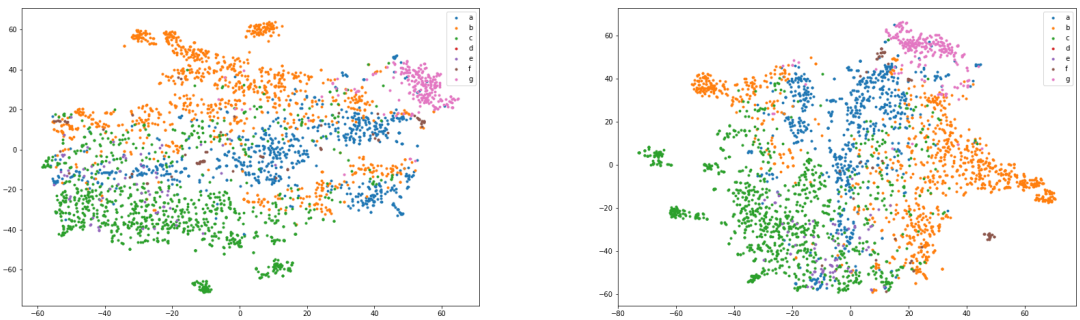

Figure 43: Final and minloss comparison of the TSNE dimensionality reduction plots of the 1-layer, no lr schedule, 512 feature model.

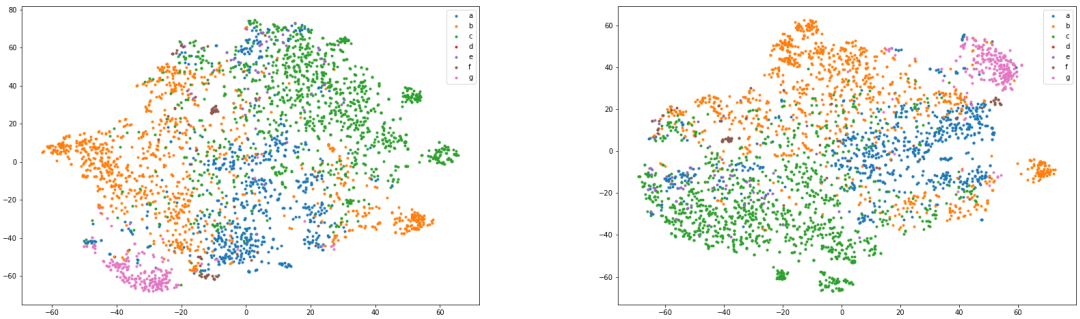

Figure 44: Final and minloss comparison of the TSNE dimensionality reduction plots of the 1-layer, 512 feature model.

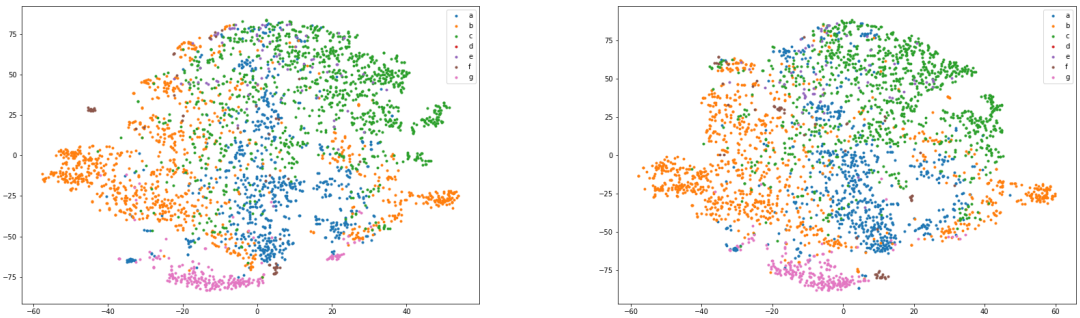

Figure 45: Final and minloss comparison of the TSNE dimensionality reduction plots of the 1-layer, 256 feature model.

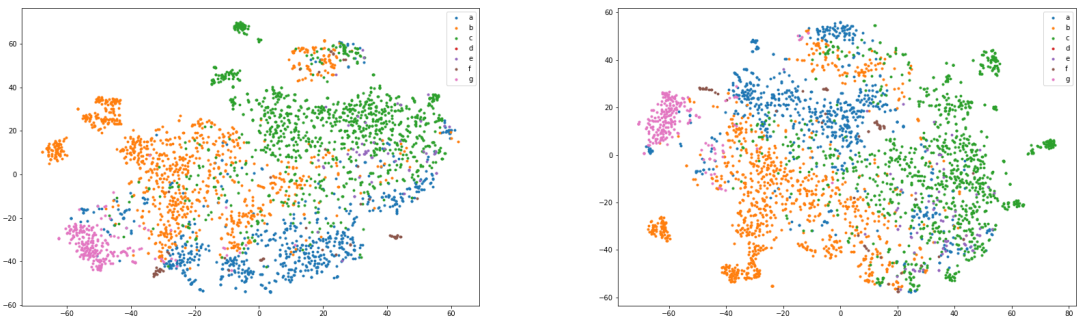

Figure 46: Final and minloss comparison of the TSNE dimensionality reduction plots of the 1-layer, 1024 feature model.