Computer Science Extended Essay:

Investigating the efficiency and effectiveness of neural networks on computational biology non-polynomial time tasks

Research Question:

How accurate and efficient are different neural networks in solving the protein folding problem?

Word Count: 3944

Table of Contents

1. Introduction

The protein folding problem has remained as one of the most ubiquitous unsolved problems in modern computational biology. However, with growing innovation in the field of neural networks and their unprecedented ability to solve complex tasks, there have been major attempts and advances in using neural networks to solve the protein folding problem. As most recently demonstrated by Google's AlphaFold neural network, which can accurately predict how a protein folds 100 times better than conventional computational methods (Alphafold, 2020), there is serious upside to using neural networks over conventional algorithms to solve this problem. Determining which neural network architectures and neural network features to use for optimal performance is still unclear however.

This essay seeks to compare the accuracy and efficiency of various neural networks on their ability to solve the protein folding problem. This essay evaluates four neural network architectures, a multilayer network, a recurrent neural network, a long short term memory network and a transformer on their accuracy on the problem, and conversely, the time and space complexity in order to perform predictions.

In order to investigate the accuracy and efficiency of each of the various neural networks, each network was programmed with the open-source machine learning framework Tensorflow (Abadi et al, 2015) and trained and tested with data from The Protein Data Bank. Runtime and accuracy based on the distance of each amino acid of the predicted structure to the real structure was used to evaluate each network's efficiency and accuracy.

This research question is worthy of investigation because it justifies the use of neural networks in the context of the protein folding problem, but in a broader context, in computational biology and shows the capacity to which they can be leveraged. Moreover, this research question helps to clarify the existing use of certain neural networks over others for their superior effectiveness.

2. Background Information

2.1 The Protein Folding Problem

Proteins are molecules within cells composed of amino acids that perform most functions that allow the cell to live and thrive. The structure of a protein directly determines how that protein will function. Protein folding is a complex biological process that turns a connected string of amino acids into the complex structure (Simmons, 2018). Thus, the protein folding problem is concerned with finding the three-dimensional atomic structure of a protein given its amino acid constituents.

Chemically, proteins fold into a structure which minimizes the net energy distribution among the entire protein. As amino acids are characterized by individual molecules, the folded structure minimizes the repulsion and attraction between each of the individual atoms that make up the protein (Lieff, 2012). This chemical process materializes quite quickly in the real world, however, simulating each individual atom and its relation to each other individual atom is computationally complex. To contextualize this complexity, consider the average protein in the human body: the average protein in the human body has 144 amino acid constituents, with each amino acid having around 136 atoms. To minimize the energy of each atom, one would have to

compare each atom to each other, leading to roughly 19584^{19584} calculations to compute the most ideal structure. In Big O notation, given the number of atoms n, it would take $O(n^n)$ time to fold a protein. This is considered a non-deterministic polynomial-time problem. It takes an exponential amount of time to solve for the structure of a protein, as the number of atoms scales. Thus it is infeasible to use a simulative method to solve this problem, which is why the necessity for other solutions exist.

This is the goal of neural networks in the context of this problem, to approximate the structure of proteins in a feasible runtime. The rest of this section further explains the mechanism in which this is achieved.

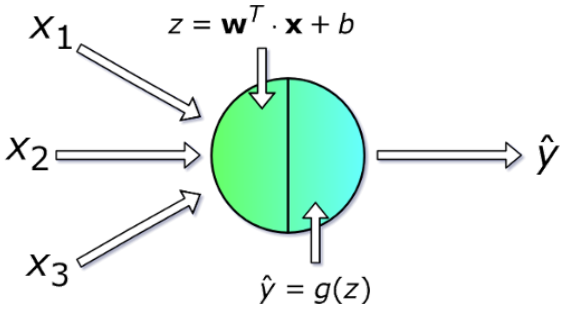
2.2 Multilayer Neural Networks

The overarching goal of neural networks is to take some input and map it to an output, by using a differentiable function with changeable parameters that map the input to the output (usually through addition or multiplication). These parameters, known as weights and biases, are learnt over time by providing the network with examples of known input-output pairs and having it iteratively get better (Brownlee, 2016).

These networks are composed of singular neurons, or simple functions that accept some number of inputs and produce a singular output value. Each input is multiplied by its own weight—some parameter, and all the values are added together, to produce the output value. The weights are optimized over time to create the best outputs. This operation is represented by taking the dot product of two vectors, an input vector X, and a weight vector W, to produce a singular value, z.

This singular value is then entered into an activation function, which is a nonlinear function that allows the output to be better characterized, which will greatly increase the speed of learning.

Figure 1: Single neuron input and output graph



Each of these individual neurons are stacked alongside other neurons to comprise a layer. This mimics much of how computation in brains occurs, with multiple neurons firing together simultaneously to perform some function in the brain. By having multiple neurons in a layer, the input and intermediary outputs can be represented in a higher dimensionality (by more weights), and thus, more value can be extracted from them. To illustrate this point, Figure 1 depicts an input being mapped to a layer 2 times its size, which allows the network to create better informed outputs. Since one weight vector is needed to represent all the weights for a single neuron, each weight vector for each neuron can be transposed and stacked together to create a weight matrix that represents the weights for the entire layer.

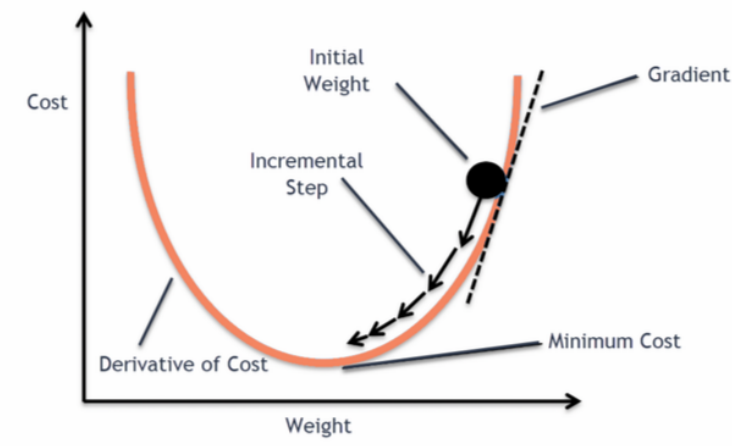
These layers are finally connected together, creating a multilayer network. Each neuron in one layer maps its output to each other neuron in the layer next to it, effectively creating relationships between each of the layers. Once again, this is similar to how brains perform complex functions, by firing interconnected neurons together in sequence. Layers are important because after each layer, some parts of the input can be determined to be more important than the others, and

following layers can use that information to better compute an output. In the context of the protein folding problem, some amino acids may be determined to be more important than others, which the network can pick up over time and use to create better predictions about the structure of the protein.

To create accurate predictions, the weights of the entire neural network must be tuned to optimal values such that they produce optimal predictions. How does one know what optimal predictions look like? By letting the network predict values for certain inputs, and comparing the predicted values against the ground-truth values, one can get any idea of how far off the model is from accurate predictions. By using a loss function, the model's accuracy can be quantified and tangible. Most loss functions take the predicted values and subtract them from the ground-truth values, leading to a number which represents how inaccurate the model is, with a larger number representing more inaccurate.

By optimizing the loss function, such that it is the lowest value possible, the model will perform better. By considering how changing the weights affects the loss function, the weights can be tuned to optimal values. By taking the derivative of the loss function with respect to the weights of the model, it can be precisely determined how the weights affect the loss function value. By descending down the derivative of the loss with respect to the weights, it can be determined what values of the weights will yield a lower value of the loss, as seen depicted in Figure 2. Thus, by descending down the derivative, or gradient, it will lead to optimized weights and thus, a better performing model.

Figure 2: Performing gradient descent on a "cost" (loss) function



2.3 Recurrent Neural Networks

One downside of multilayer neural networks is that they can not consider the relationships between certain parts of the data in relation to each other. It cannot use the characteristics of sequential data, where one part of the data directly affects the next part of the data (Akkaya, 2019). In the context of the protein folding problem, one amino acid that is adjacent to another amino acid directly affects how that one amino acid will fold. Sequential information is key in figuring out how a protein will fold, thus incorporating this information is vital to a neural network's performance.

Recurrent neural networks allow sequential data to be better analyzed and considered in a model's predictions. The simple idea of a recurrent neural network is to feed data sequentially, as opposed to everything together, and supply the output of the previous prediction into the current one. By doing so, there's explicit reference to past data, and explicit use of the past data in the calculations for the current one. For the protein folding problem, one could predict the position of each amino acid, as opposed to all of them together, and supply the position of each past

amino acid when predicting the current amino acid. This relates how the position of one amino acid directly affects the other, and allows the model to pick up on the chemical interactions between all the amino acids.

Additionally, by putting two recurrent neural networks together, and having one predict the position of an amino acid in the future, and then using that information to predict the position of the current amino acid, the model can use both past and future positional information in its calculations. This is known as a bidirectional recurrent neural network, because it incorporates information in both directions, in the past and future.

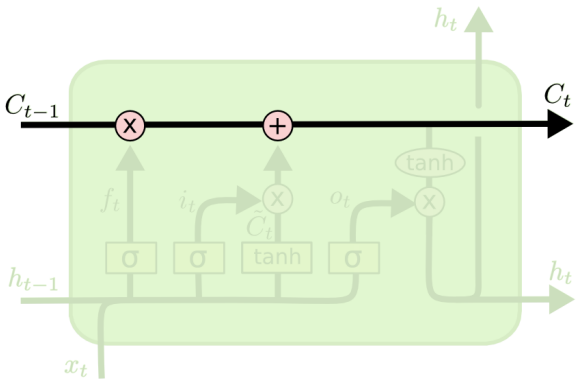
One key assumption of recurrent neural networks is that it assumes the information from the past in a sequence are all the same in importance. The first input in the sequence is treated as valuable as the second input in the sequence. This is problematic for protein folding, as one amino acid may have more of a bond to the current amino acid as opposed to another one. The weightage of importance is not considered. Furthermore, information about data seen a long time ago by the network is gradually lost. This is known as the vanishing gradient problem, and does not allow the network to use the entire breadth of information for use (Bohra, 2021). Long-short term memory networks aim to improve this downside.

2.4 Long Short Term Memory Neural Networks

Long short term memory networks (LSTMs) run on the same fundamental idea as RNNs, rather, they create a more complex memory state that is passed onto the next neural network to multiply against the input of the next input in a sequence.

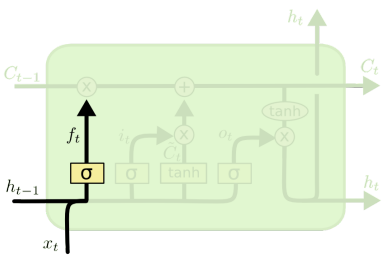
LSTMs comprise of 4 gates, or series of operations that produce a long-term hidden state, which is a vector of information that propagates the long-term context of some data, and an output, which is used to compute the predictive value (Singhal, 2020).

Figure 3: Long-term gate representation of an LSTM



The core gate to an LSTM is the long-term gate which runs unaffected by many independent operations. This passes long-term information, and directly interacts with other gates to alter the long term state. The multiplication operation is done with the output of the forget gate, and affects the long-term state by forgetting and strengthening certain information composed in the long-term state, as depicted in Figure 3. The addition operation adds new information from the long term state with the output of the new gate, which can create new long-term dependencies that are key to predicting the output of new sequences.

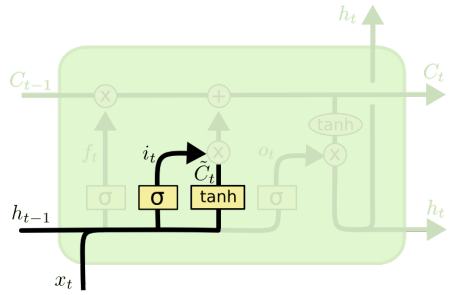
Figure 4: Forget gate representation of an LSTM



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The forget gate determines what information to remove in long-term memory. It does this by the last hidden state and current input through a neural network layer, using some optimized weights. The following output is then run through a sigmoid function, which gives each value of the resulting vector a number between 0 and 1, where 0 means to forget that certain information, and 1 means to keep it, Each value in the matrix is multiplied as a scalar against the current long term state, to compose a new long term state, or long term information, as shown in Figure 4.

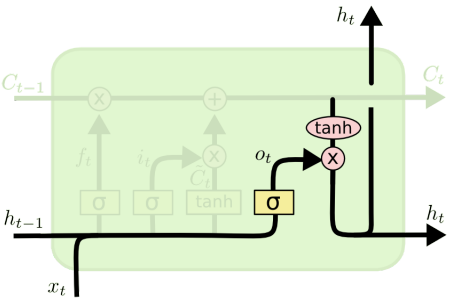
Figure 5: Add state representation of an LSTM



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

The add state adds new information by running the input and past hidden state through two neural network layers, one sigmoid layer, which is similar to the forget gate, which determines how important each piece of the new information is, and a tanh layer, which determines what new information to add, as depicted in Figure 5. Once again, weights perform these operations, and learn overtime what values to be to provide accurate outputs and information.

Figure 6: Short-term memory state representation of an LSTM



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Finally, the output of the LSTM is determined by multiplying the last hidden state with the long-term memory state to produce a short term memory state. This encodes short-term information about the sequence that is being provided to the network, and is propagated and changed to reflect how the information changes.

It's important to note that the final output of an LSTM network is some combination of the long-term state with the short-term state, which can be interpreted by more standard layers to produce an output. This mimics encoding important information and decoding it to produce a new sequence. In fact, the process of explicitly encoding and decoding information is what occurs in Transformer networks.

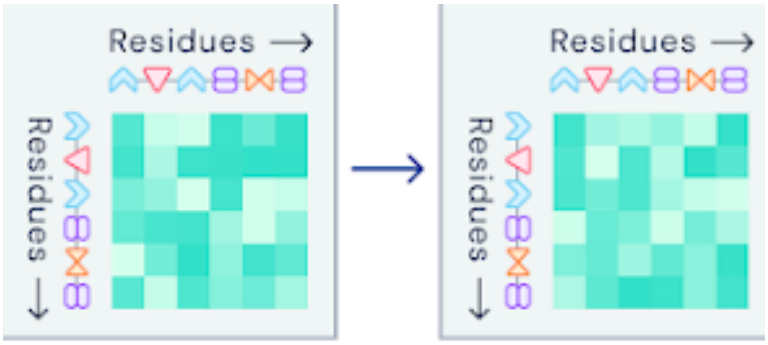
2.5 Transformers

Both LSTMs and RNNs try to incorporate the relationship between pieces of information in a sequence together. The biggest downside of LSTMs and RNNs is that these relationships are not explicit, instead it is more or less upto the neural network to determine what relationships are important in predicting the output. Especially in protein folding, each amino acid has an explicit relationship to any other amino acid—in the form of repulsion or attraction—and by not considering even a singular interaction can yield a widely different result from the ground truth.

Transformers solve this by explicitly determining how important the relationship of each amino acid is to any other amino acid. The mechanism it uses to do so is called attention (Alammar, 2018). The transformer uses weights to create an attention matrix, which is a N by N matrix (where N represents the number of pieces of data in the sequence) where the cell N_{kJ} has a

singular value between 0 and 1 (where 0 is representative of unimportant and 1 is important) that represents the importance of the relationship between K and J, as depicted in Figure 7. By doing so, the matrix has information about the relationship of each piece of data against every other piece of data. The only downside to this is that one would need to allocate N^2 memory to store all the information (which can scale up very quickly).

Figure 7: Attention matrix for amino acid sequences (amino acids as "residues")



Transformers are also unique in the way they decode the attention matrix. They first create multiple attention matrices, and then decode them asynchronously adding them all up at the end. The decoding process of the information happens multiple times independently, which reduces the randomness and accuracy of the neural network to create correct attention matrices. Finally, regular neural network layers are dotted against these matrices to produce an output.

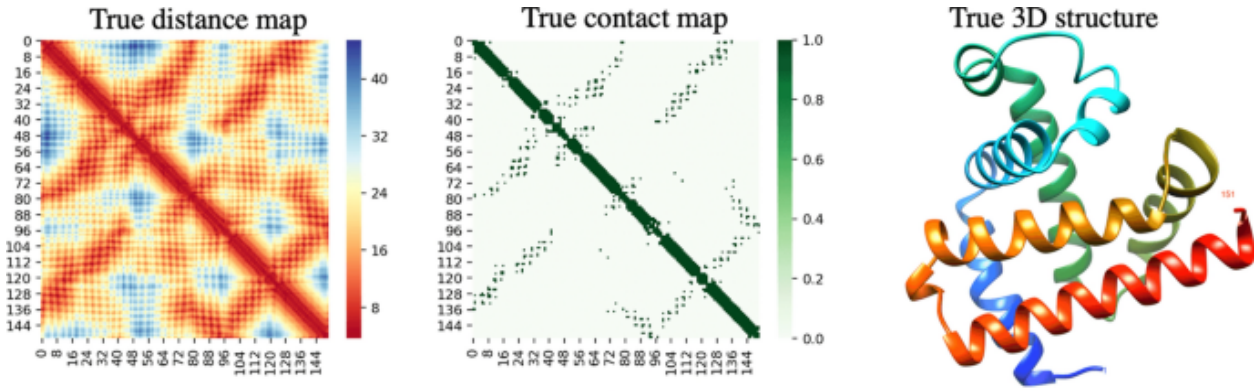
3. Experimentation

3.1 Methodology

Using Python 3 and the open-source machine learning framework Tensorflow, I programmed a multilayer network, a recurrent neural network, a long-short term memory network and a transformer that predicts protein structures based on data from the Protein Data Bank. The networks were made using Tensorflow's Functional API, which allows for more complex neural networks. The code for these models can be found in the appendix.

The task for each model is to accept a sequence of amino acids, and produce a pairwise distance matrix, where each column and row represents a different amino acid, as shown in Figure 8. A position in this matrix represents the distance between two amino acids in angstroms (the typical unit of distance when measuring on this scale). This matrix holds the information to construct a 3D representation of the protein sequence, and since in a matrix form, makes it ideal for networks to work with.

Figure 8: Output representation of a protein from neural network model



Instead of using the entirety of the Protein Data Bank, a criteria was used to select proteins for the dataset that are ideal for training. Firstly, only protein structures determined by X-ray

diffraction were used as data points within the dataset. This was to ensure the accuracy of the dataset, as X-ray diffraction produces the most accurate models for proteins (Smyth, 2000). Secondly, only proteins that were 100 to 400 amino acids in length were used in the dataset, as to reduce the complexity for the models and to reduce the training time. Lastly, only proteins found in mamillians were used, to reduce amino acid selectivity bias in other types of organisms. The resulting dataset had 24113 proteins.

19290 data points (80%) of the dataset was used for training the models, and the rest, 4823 data points (20%), for testing. In the training dataset, a further breakdown of 5787 data points (30%), was used for validation, data which the model has never seen, to ensure that it is not just memorizing the training dataset, and instead learning to make predictions. The last 13503 data points (70%) of the training dataset was used to train the models.

To measure how good a model is doing—the loss function—the root mean squared deviation function is used to determine how accurate a prediction is. It works by taking the predicted distance matrix, and compares each ground truth distance by how close the predicted and real value was, producing a final difference over all the distances. A lower value means a more correct prediction. This loss function is used for accuracy and also as the optimization function for the models. This loss function was optimized with the use of gradient descent.

To measure the runtime and memory complexity of each model, at every epoch, the runtime and memory used will be taken, and averaged across all epochs. This is done through Tensorflow.

Multiple instances of the model will be trained, using increasing dataset sizes, to see how the runtime and memory scale.

3.2 Results & Analysis

The models were run, and as they were running, the loss of the models and accuracy was measured after every epoch. The runtime and memory usage was also recorded. This data was exported, and graphed using Google Sheets, showing the trends of the accuracy overtime. The graphs and tables for each of the models are below:

Figure 9: The accuracy of the multilayer neural network over a training time of 300 epochs

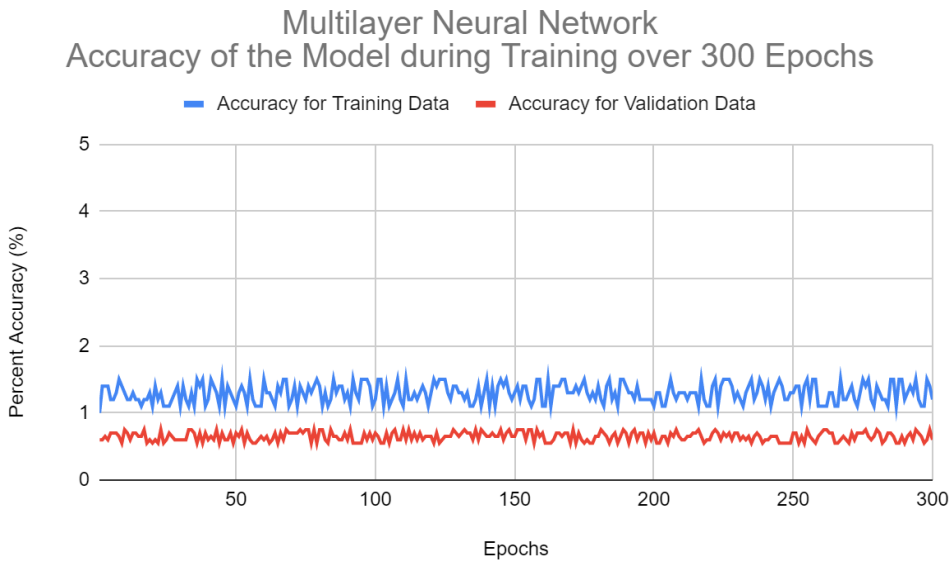


Table 1: Runtime and memory usage for multilayer neural network

Number of Training Examples	Average Runtime per Epoch (s)	Average Memory Usage per Epoch (mega bytes)
1000	15.8	5
2000	16.3	22
3000	14.3	37

4000	15.4	48
5000	18.2	69
6000	13.5	84
7000	14.5	93
8000	15.6	113
9000	16.1	129
10000	16.1	142

Firstly, the multilayer network didn't make any improvement whatsoever while training, as seen in Figure 9, meaning that it is likely that the model is unable to pick up on the chemical interactions between amino acids and find any complex relationships. Due to this, the model effectively had a 0% accuracy when predicting amino acid structures on the training and the test set, as seen in Figure 9. The runtime was constant, and the memory usage was linear, indicating a very efficient model, as seen in Table 1.

Figure 10: The accuracy of the recurrent neural network over a training time of 300 epochs

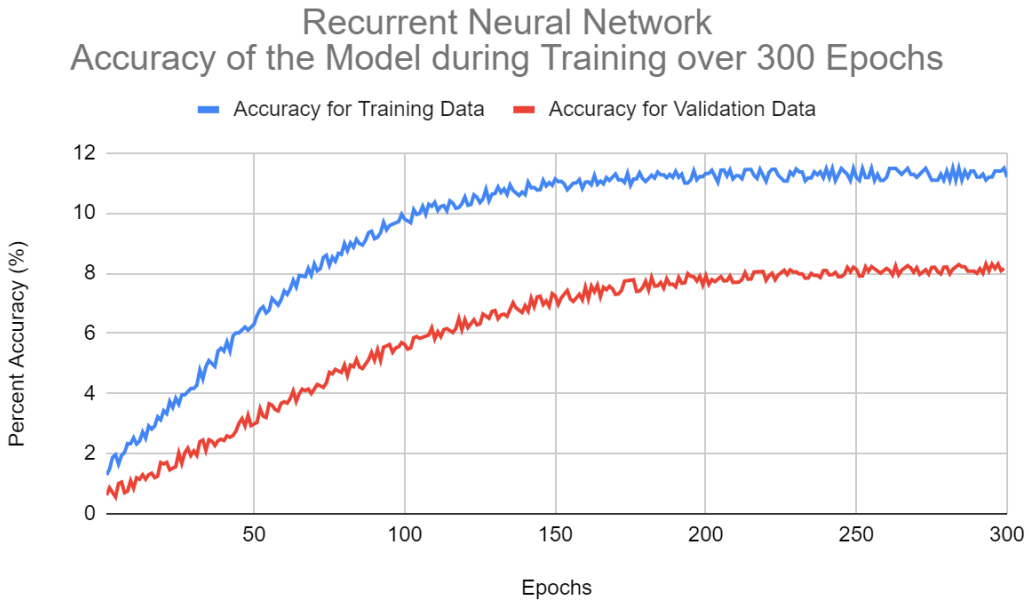


Table 2: Runtime and memory usage for recurrent neural network

Number of Training Examples	Average Runtime per Epoch (seconds)	Average Memory Usage per Epoch (megabytes)
1000	21.3	23
2000	23.5	55
3000	26.1	85
4000	28.3	119
5000	30.4	142
6000	32.7	174
7000	34.6	209
8000	36.9	240
9000	39.1	265
10000	41.2	296

The recurrent model starts to learn and pick up on relationships during the beginning of it's training phase. However, it quickly plateaus around the 10% mark, meaning it can accurately predict the amino acids positions 10% of the time, as seen from Figure 10. On the test set, it performed expectedly worse, at around 8%, but since it wasn't a large drop, it indicates that these relationships that the model developed are somewhat representative of general chemical interactions. The runtime and memory usage scaled linearly, meaning that the model was somewhat efficient, and did not scale up unfeasibly, as seen in Table 2.

Figure 11: The accuracy of the long short term memory network over a training time of 300 epochs

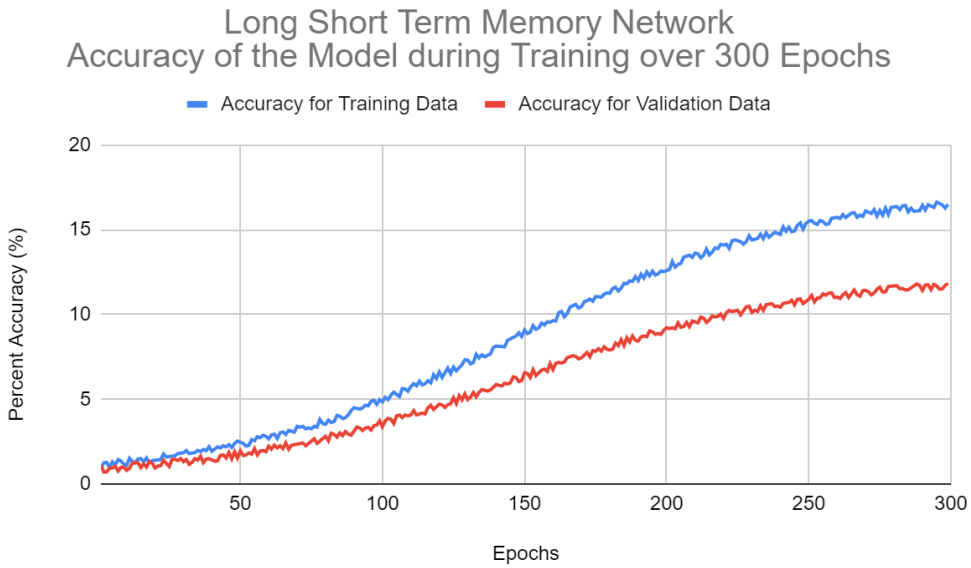


Table 3: Runtime and memory usage for long short term memory network

Number of Training Examples	Average Runtime per Epoch (seconds)	Average Memory Usage per Epoch (bytes)
1000	21.3	43
2000	26.2	95
3000	31.4	145
4000	37.1	198
5000	42.5	245
6000	47.9	301
7000	53.2	348
8000	58.9	398
9000	64.2	444
10000	69.8	492

The long-short term memory model interestingly performs only a little better then the recurrent model, with a final accuracy of around 15%, as seen in Figure 11. This is likely due to the fact that the model is more nuanced in it's relationships between amino acids, but it is still not significant enough to be a good predictor of amino acid positions. The distinct advantage of this model was it's efficiency, having it use the least amount of memory and its runtime being linear, as seen in Table 3.

Figure 12: The accuracy of the transformer network over a training time of 300 epochs

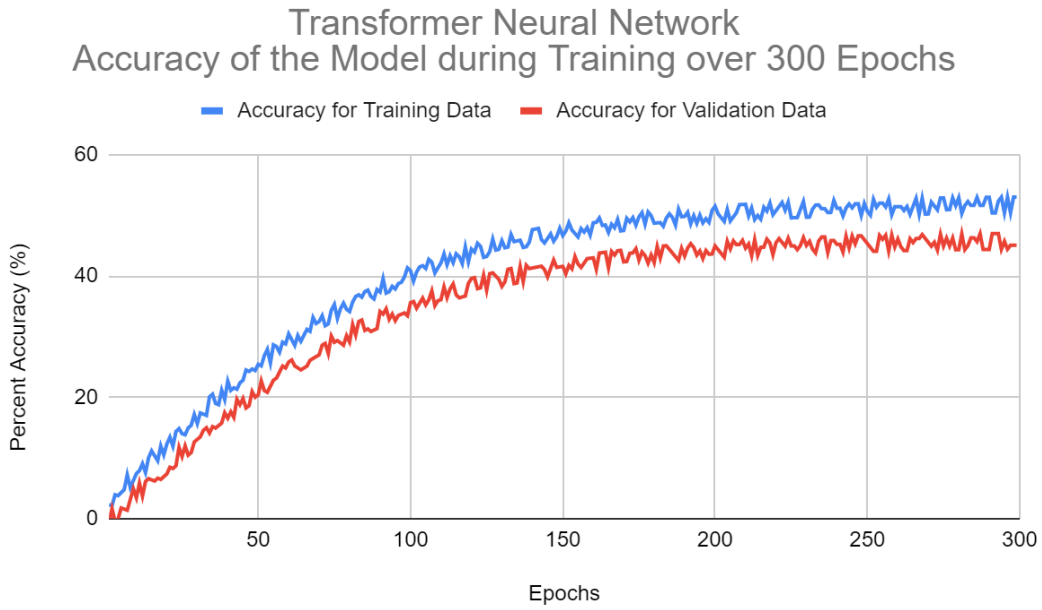


Table 4: Runtime and memory usage for transformer network

Number of Training Examples	Average Runtime per Epoch (seconds)	Average Memory Usage per Epoch (megabytes)
1000	31.3	20
2000	32.7	50
3000	37.2	140

4000	44.7	290
5000	68.7	500
6000	85.2	770
7000	104.7	1100
8000	127.2	1490
9000	152.7	1940
10000	181.2	2450

The transformer outperformed all the models by a significant margin. It had an accuracy of around 50%, with the testing accuracy being 43%, as seen in Figure 12. This is likely due to the fact that the model creates explicit relationships between each amino acid sequence, which is ideal for the protein-folding problem and better learns the interactions in general chemistry. It can be seen that for a model that predicts chemical interactions, it needs to factor in every interaction possible, as this greatly determines the final output of an interaction. The largest downside of this complete interaction-based model is it's memory usage. It had the most memory use and largest run time for training as compared to all the other models, almost 3 times per epoch more compared to the long-short term memory network model, as seen in Table 4.

From the results of this experiment, it is evident that the artificial intelligence approach that factors in explicit interactions between molecules yields more accurate predictions on the protein-folding problem, yet sacrifices efficiency overall.

3.3 Evaluation

In retrospect, the hyperparameters of the models (eg. weights in layers, number of epochs trained, batch size for gradient descent, etc.) may have not been optimized prior to beginning to train the models. For future experimentation, by running multiple instances of the models with different hyperparameters, and quickly seeing which models started to learn better, a model with more optimized hyperparameters could have been selected and used for training, likely yielding better results. Additionally, due to technical limitations with memory scalability, only proteins with amino acids of 100 to 400 length were used. This is best illustrated with the transformer, which scales up at $O(n^2)$, and to rectify this limitation in the future, leveraging cloud computing and instances can be used to automatically scale memory capacity when needed for the models.

4. Conclusion

The experiment that was conducted confirms that neural networks lead to more accurate predictions on the protein-folding problem, and some neural networks were more accurate due to specific features of the networks. It is clear that some models were more accurate than others, and were more efficient. In particular, networks that emphasized the interaction between amino acids performed better than other models—as seen with the long-short term memory network and transformer. However, these networks become more inefficient as the interactions between amino acids become more explicit in the model's function. The runtime and memory usage begins to scale up very quickly as the model's structure stores more information about the interactions between amino acids. Even with more computational power, when working with large biological molecules—that can span up to hundreds of thousands of atoms—these models become less feasible in an applicative context. However, it is evident that these models still

outperform classical algorithmic methods, being faster and more efficient. The major bottleneck for the use of neural networks in computational biology tasks still remains to be better computation. Through conducting this investigation, I learnt a lot about neural networks and how they function on a mathematical level, and how to apply neural networks to difficult problems using programming languages and technical skills.

Neural networks still can further be improved to reduce the time complexity, and memory usage complexity when working on non-polynomial computational biology tasks. As the landscape of biochemistry is immensely complex, a model that encapsulates this complexity will perform the best, which is evident through the experiment, with each better-performing model having more complex functionality. In order to capture this complexity, and reduce the time required to run these models, new methods in neural networks will need to be developed. However, it is abundantly clear that neural networks are the best approach to solving these problems, and have immense potential to radically change how we think about biological systems.

5.0 References

Abadi et al. (2015) *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45166.pdf

Akkaya, B. (2019, September 3). *Comparison of Multi-class Classification Algorithms on Early Diagnosis of Heart Diseases*. Research Gate. https://www.researchgate.net/figure/Multilayer-Perceptron-Advantages-and-Disadvantages_tbl4_338950098

Alammar, J. (2018, June 27). *The Illustrated Transformer*. https://jalammar.github.io/illustrated-transformer/

Bohra, Y. (2021, June 18). *The Challenge of Vanishing/Exploding Gradients in Deep Neural Networks*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

Brownlee, J. (2016, May 27). *Crash Course On Multi-Layer Perceptron Neural Networks.* Machine Learning Mastery. https://machinelearningmastery.com/neural-networks-crash-course/

Lieff, J. (2012, December 10). *Protein Folding in the Neuron.* https://jonlieffmd.com/blog/protein-folding-and-the-mind#:~:text=Chain%20Entropy%20%E2%80%93%20This%20is%20a,is%20a%20very%20stable%20state.

Singhal, G. (2020, September 9). *Introduction to LSTM Units in RNN*. Pluralsight. https://www.pluralsight.com/guides/introduction-to-lstm-units-in-rnn

Simmons, W. (2018, November 19). *Protein Folding and Machine Learning: Fundamentals*. Arxiv. https://arxiv.org/abs/1811.09536

Smyth, M. (2000, February). *X-Ray crystallography.* National Center for Biotechnology Information https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1186895/

The Alphafold Team. (2020, November 19). *AlphaFold: a solution to a 50-year-old grand challenge in biology*. Deepmind. https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology`

6.0 Appendix

The following program has the code that defines each of the models, and runs them using Tensorflow, and Numpy. The model was run using a 1060 TI GPU, with 16GB of RAM.

```
# Packages and libraries for programs and models to run
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras.layers import *
from tensorflow.keras.models import Model
import csv

# Getting the Protein Databank dataset
def get_dataset():
    # Load the dataset using tensorflow built-in dataset function
    dataset = tfds.load('pdb', split="train[80%:]", shuffle_files=True)
    refined_dataset = np.array()
    # Processing data
    for data in dataset:
        # Only using data that is 100 - 400 amino acids long, and is determined
using x-ray crystallography
        if ((len(data[0]) < 400 and len(data[0]) > 100) and (data.method=="xray")):
            refined_dataset.append(data)
    # Each Datapoint is 400 characters long in order for models to run
    refined_dataset=tf.pad(refined_dataset)
    return refined_dataset

# Multilayer model structure
def multilayer_model():
    input = Input(shape=400)

    # Defining multilayer model architecture
    multilayer_model = Dense(units=10000, activation='relu')(input)
    multilayer_model = Dense(units=5000, activation='relu')(multilayer_model)
    multilayer_model = Dense(units=2500, activation='relu')(multilayer_model)
    multilayer_model = Dense(units=5000, activation='relu')(multilayer_model)
    multilayer_model = Dense(units=10000, activation='relu')(multilayer_model)
    multilayer_model = Dense(units=50000, activation='relu')(multilayer_model)
    output = Dense(160000, activation='sigmoid')(multilayer_model)
    model = Model(inputs=input)

    # Compiling the Model (setting loss function, optimizer, and metrics)
    model.compile(loss='rmsd',optimizer='adam',metrics=['accuracy'])
```

```
    return model

# RNN model structure
def rnn_model():
    input = Input(shape=400)

    # Defining RNN model architecture
    rnn_model = RNN(units=500, return_sequences=True)(input)
    rnn_model = RNN(units=250, return_sequences=True)(rnn_model)
    rnn_model = RNN(units=125, return_sequences=True)(rnn_model)
    rnn_model = RNN(units=100, return_sequences=True)(rnn_model)
    rnn_model = RNN(units=75, return_sequences=True)(rnn_model)
    rnn_model = RNN(units=75, return_sequences=True)(rnn_model)
    output = Dense(160000, activation='sigmoid')(rnn_model)
    model=Model(inputs=input, outputs=output)

    # Compiling the model (setting loss function, optimizer, and metrics)
    model.compile(loss='rmsd',optimizer='adam',metrics=['accuracy'])
    return model

# LSTM model structure
def lstm_model():
    input = Input(shape=400)

    # Defining LSTM model architecture
    lstm_model = LSTM(units=100, return_sequences=True)(input)
    lstm_model = LSTM(units=100, return_sequences=True)(lstm_model)
    lstm_model = LSTM(units=100, return_sequences=True)(lstm_model)
    lstm_model = LSTM(units=100, return_sequences=True)(lstm_model)
    lstm_model = LSTM(units=100, return_sequences=True)(lstm_model)
    lstm_model = LSTM(units=100, return_sequences=True)(lstm_model)
    lstm_model = LSTM(units=100, return_sequences=True)(lstm_model)
    output = Dense(160000, activation='sigmoid')(lstm_model)
    model=Model(inputs=input, outputs=output)

    # Compiling the model (setting loss function, optimizer, and metrics)
    model.compile(loss='rmsd',optimizer='adam',metrics=['accuracy'])
    return model

# Transformer model structure
def transformer_model():
    input = Input(shape=400)

    # Defining Transformer model architecture
    inputs = Input(shape=(None, 400))
```

```
    start = LSTM(20, return_state=True)
    e_outputs, h, c = start(inputs)
    states = [h, c]
    d_inputs = Input(shape=(None, 20))
    d_lstm = LSTM(400, return_sequences=True, return_state=True)
    outputs, _, _ = d_lstm(d_inputs, initial_state=states)
    decoder_dense = Dense(160000, activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)
    model = Model([inputs, d_inputs], outputs)

    # Compiling the model (setting loss function, optimizer, and metrics)
    model.compile(loss='rmsd',optimizer='adam',metrics=['accuracy'])
    return model

# Function that runs each model and saves the results
def run_model(epochs, model, dataset):
    model.fit(dataset, epochs=epochs)
    return model.history

# Function that writes the results to a csv file
def save_data(history, model_name):
    with open(model_name+'.csv', mode='w') as csv_file:
        fieldnames = ['loss', 'accuracy']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerow({'loss': history.history['loss'], 'accuracy':
history.history['accuracy']})
        writer.writerow({"memory": history.memory, "time": history.time})

# Running Each model and saving the results
data_collection_mlm = run_model(300, multilayer_model(), get_dataset())
save_data(data_collection_mlm, 'multilayer_model')

data_collection_rnn = run_model(300, rnn_model(), get_dataset())
save_data(data_collection_mlm, 'rnn_model')

data_collection_lstm = run_model(300, lstm_model(), get_dataset())
save_data(data_collection_mlm, 'lstm_model')

data_collection_trans = run_model(300, transformer_model(), get_dataset())
save_data(data_collection_mlm, 'transformer_model')
```