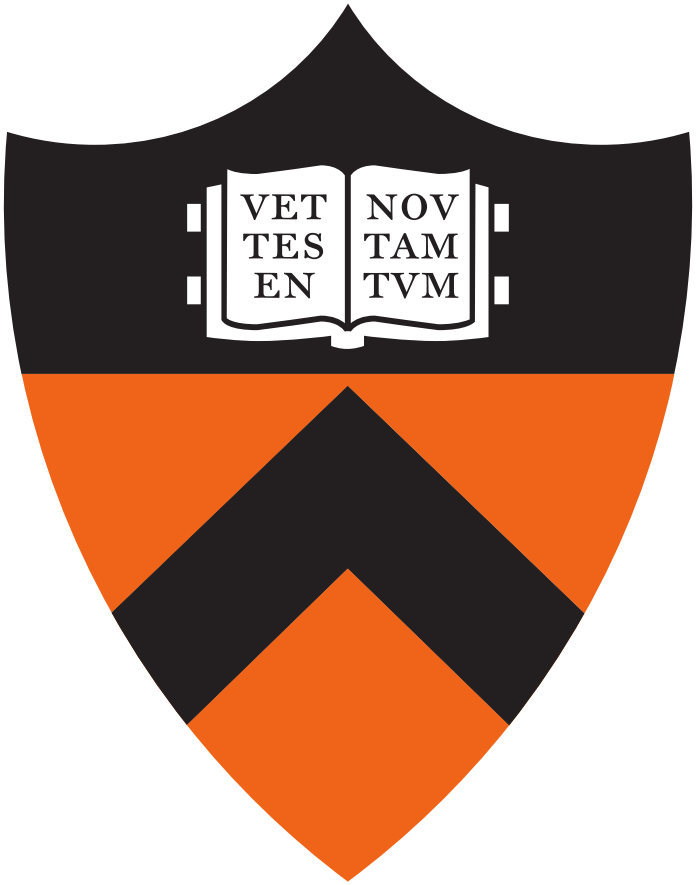
– Independent Work Report, 2017 –

Encoder / Decoder Neural Networks for Protein Secondary Structure Prediction

Christopher Ferri
Adviser: Professor Mona Singh

Abstract

In this thesis, I attempt to apply Encoder / Decoder neural networks to the problem of protein secondary structure prediction. I explain the design of basic Recurrent Neural Networks and the history of secondary structure prediction algorithms. I then describe the three network models that I will train and test, along with a simple test dataset built from the Protein Data Bank. Through this study, I hope to provide a simple, accurate system for researchers to predict the secondary structure for a novel protein.



Acknowledgements

Over the course of this year, many people have helped me along the way with the writing of this thesis. Whether their contribution was large or small, I would like to take the time to thank them for their contributions.

I would first like to thank my advisor, Professor Mona Singh, in the Lewis-Sigler Institute for Integrative Genomics for supporting my research. I would also like to thank my second reader, Professor Sebastian Seung, of the Princeton Neuroscience Institute.

I would like to thank my friends, who have been great resources. My friend, Stephen Timmel, for helping me debug my preprocessing code. My friends Matthew Penza and Stephen Bork, who have been great working buddies and proofreaders.

I would like to thank my family. My mother and Father, Roseann and John Ferri. Thank you for all of the times you were willing to let me rant and rave about the latest step of producing a thesis, along with all the countless other ways you have supported me from a young age. I wouldn't be here with out you guys. My brother, Nicholas Ferri, for always knowing how to put a smile on my face and helping me to forget about my work, even for just a moment.

I would like to thank the Roman Catholic church. My years of being a practicing Catholic have taught me not only how to reason, but how to turn even my very work into a prayer. You have giving me strength through the toughest of times, even when writing a thesis.

Ad majorem Dei gloriam inque hominum salutem.

J.M.J

Contents

List of Figures

List of Listings

Introduction

Protein Secondary Structure prediction is one of the many open problems in Structural Biology. Due to the inherent complexity of the problem, it is not solvable by any known closed-form equation; in fact, we will most likely never find any such solution. Unfortunately, this information can be incredibly valuable to structural biologists, as the structure of the protein affects nearly all of its properties. Due to the recent boom in Machine Learning methods, biologists have turned to Computer Science to help advance the current state of Secondary Structure prediction. Many methods have pushed the accuracy into the 80% range. However, considering the complexity of chemical reactions that proteins participate in, this is not nearly accurate enough. In this paper, state of the art techniques from Natural Language Processing and Deep Learning are put to use to help advance the state of the art in the field, and improve the accuracy of secondary structure prediction. Using techniques such as word embeddings and Encoder / Decoder networks, we hope to increase the Q8 accuracy of prediction, resulting in a new method for protein secondary structure prediction.

Currently, the process for discerning protein structure, secondary or higher, is a complex, expensive, and time consuming process. In the age of high throughput sequencing, this is a bottleneck. Research in the field of structural biology depends on understanding the complex form of a protein to perform research on the metabolic pathways that the respective protein is a part of. By improving secondary structure prediction methods, we give biologists the ability to produce working drafts of the protein that they are studying, which can be used until experimental methods can be used to correct and verify the protein structure.

Background

In biology, all cells consist of a few classes of molecules: Lipids, Carbohydrates, Nucleic Acids, and Proteins. Lipids and Carbohydrates are used for the structure of the cell, such as the cell wall, and energy. Nucleic Acids make up our DNA and RNA, which encode Proteins and other regulatory information in the permutation of the nucleotides contained within them. Proteins are the workhorses of the cell. Their functions range from acting as catalysis for chemical reactions happening within

cells, to forming complex channels in the cell membrane to allow specific molecules such as water to flow in and out of the cell.

In what is considered the Central Dogma of Biology, genetic information is transcribed from DNA into mRNA. From here, mRNA is translated from the sequence of chemical labels, called codons, into a long polymer, called a polypeptide. Polypeptides are the building blocks of proteins; they go on to form proteins by organizing themselves into complex geometries that allow very specific proteins or chemicals interact with them. Each of these polypeptides are composed of small, discrete chemical units called amino acids. Like links in a chain, these amino acids chemically bond to form the long polypeptide. In total, there are 20 canonical amino acids, each having unique properties that when combine in a specific order, determine the final geometry of the polypeptide, and in turn, the protein. These 20 amino acids are displayed below in Figure 1, along with some of their properties, and their unique alphabetical letter which identifies each respective protein. While there are 20 canonical amino acids, there exist a handful of non-canonical amino acids that do not correspond deterministically to a specific set of RNA codons. Selenocysteine is one such of the non-canonical amino acids. This translation is dependent upon the addition of a Selenocysteine Insertion Sequence (SECIS) in the mRNA [15]. For the purposes of this study, we will ignore all but the canonical amino acids. The unique structure of a protein is the result of countless electrostatic forces between the atoms within individual amino acids. Through random changes in the protein shape, the take on their final configuration, which is an equilibrium between all of the atomic forces, to form a stable protein.

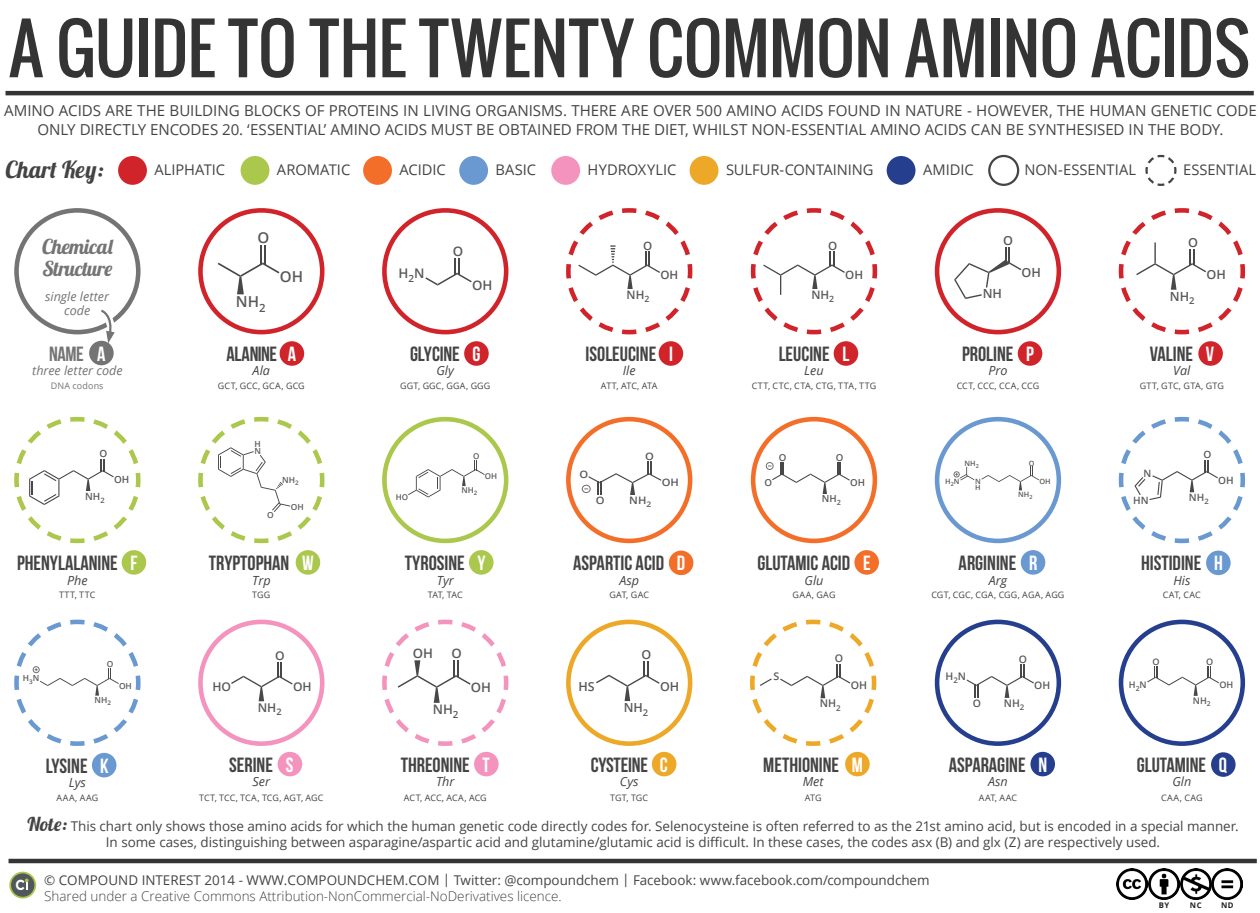


Figure 1: The 20 canonical amino acids and their properties [7]

Protein Structure

Protein structure can be divided into four levels: Primary, Secondary, Tertiary, and Quaternary. Primary Structure is the sequence of amino acids that make up the polypeptide. As previously stated, there are 20 canonical amino acids. Secondary Structure is the formation of geometrical structures, such as α-Helices and β-sheets, in the polypeptide. Tertiary Structure is the large-scale configuration of the individual secondary structures and how they fit together to form a working protein. Quaternary structure is the assembly of multiple polypeptides together to form a larger, more complex protein. However, not all proteins are made up of multiple polypeptides; as such, not every protein has Quaternary Structure. Figure 2 below depicts the relationship between each of these levels of protein structure.

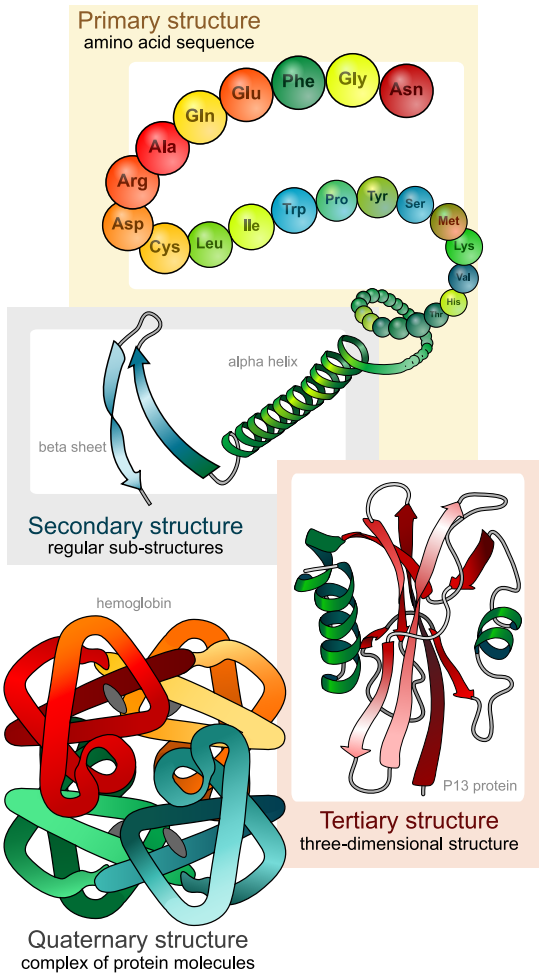


Figure 2: A graphical breakdown of the four levels of protein structure [9]

To discover the structure at each respective level, scientists must use a combination of different methods to pool the data. Primary structure is easily found through either protein sequencing techniques or by transcribing sequenced DNA which encodes the protein. Much like DNA sequencing technology, it has become cheaper, faster, and more widely available. Furthermore, the accuracy of current sequencing methods are well within the acceptable amount of error for scientific studies. This leaves no need for any sort of predictive methods. Conversely, Quaternary, Tertiary and Secondary structure are much more difficult. The reason for this lies in the fact that both of these levels are related directly to the geometric layout of the polypeptide vs. the chemical composition. To find the geometrical structure of a protein, scientists must crystallize the protein and use techniques such as X-Ray Crystallography, Nuclear Magnetic Resonance Spectroscopy (NMR),

or Cryo-electron microscopy to find the positions of atoms with respect to each other. For example, in the case of X-Ray Crystallography, large amounts of X-Ray radiation from a Synchrotron are shined through a crystallized protein. By recording the diffraction pattern, scientists are able to reassemble the 3 dimensional structure of the protein. From there, they are able to hand annotate the protein to create a finalized model of the protein. However, there are a couple of downsides to this process, specifically the process of crystallizing a protein and the requirement of a Synchrotron. Crystallizing a protein is a process that can take months at a time, and requires the protein to form without any imperfections, otherwise there won't be a diffraction pattern. Moreover, Synchrotron sources are very large and expensive, requiring researchers to travel to a national laboratory to run the experiments. NMR does not crystallization of the protein, and the equipment to run the experiment can be found in many large research universities. By using the same technology as MRI scanners, NMR machines use large magnetic fields to direct the molecule in two directions parallel to the magnetic field. From there, large radio waves bombard the molecules, causing them to selectively flip direction. By recording the frequency when these flips occur, we can identify the structure of the molecule.

Neural Networks

Neural networks are a machine learning technique that originated with the invention of the Perception by Frank Rosenblatt [22]. The Perceptron is a device for binary classification. Its input consists of a single vector $\vec{x}$ and a binary output y. Inside, it contained a weight vector $\vec{w}$ which was learned during the training phase. Mathematically, it can be represented by Equation 1:

$$y = f\left(\sum_{i=0}^{n} w_i x_i + b\right) \tag{1}$$

The function $f(u)$ is usually either the Heaviside step function, the Sigmoid / Logistic function, tanh function, or RelU. Each of these functions operate such that their range is $\{u \in \mathbb{R} : -1 \leq f(u) \leq 1\}$, with the exception of ReLU, which has a range of $\{u \in \mathbb{R} : 0 \leq f(u)\}$. This allows for easy binary classification. However, it has been shown that Perceptrons, including multi-neuron versions,

have very strong limitations. For example, while a Perceptron can learn most boolean functions, it has been proven that there is no way for a Perceptron to internally represent the XOR function with any set of values for $\vec{w}$ and b. Neural networks take this idea of a Perceptron and generalize them with more of them and by stacking them together. To train them, we use an optimization technique known as Stochastic Gradient Descent. To allow the Gradient Descent to update all layers of the network, an algorithm known as backpropagation, which is based off of the derivative chain rule, allows weight updates to be propagated backwards through the network. The algorithm first calculates δ_i^L, where subscripts represent individual neurons, and the superscript L represents the last layer. This value is the gradient of the output layer, for any general error function e. This is calculated via the equation $\delta_i^L = f'\left(u_i^L\right)\frac{\partial e}{\partial x_i^L}$, where u is the preactivation of the neuron, the result of the dot product between the $\vec{w}$ and $\vec{x}$ vectors. Given this first gradient, we are able to iteratively calculate the weight updates for each layer by using Equation 2 [26]:

$$\delta_j^{l-1} = \left(\sum_{i=1}^{n_l} \delta_i^l W_{ij}^l\right) f'\left(\vec{u}^l\right) \tag{2a}$$

$$\Delta \mathbf{W}_{ij}^l = \eta\, \delta_j^l x_i^{l-1} \tag{2b}$$

$$\Delta b_i^l = \eta\, \delta_j^l \tag{2c}$$

One of the main difficulties with neural networks is processing long sequences of data. As it is unable to take the entire input in at once, the network must process small blocks of the sequence at a time. This leads to a problem; the network is unable to take previous information / context into account when making inferences on the data. Recurrent Neural Networks (RNN) were created to solve this problem. An RNN is designed to process long sequences of data, such that the network can take into account information from a previous part of the sequence, at the current moment. The key to this ability is that each RNN neuron produces not only an output, but a hidden state that is then passed back into the cell. This information contains the necessary information for the cell to take context into account during classification. Below, Figure 3 is a diagram that shows this feedback loop, and how it is equivalent to information being passed down to cells that process

inputs of the sequence later on.

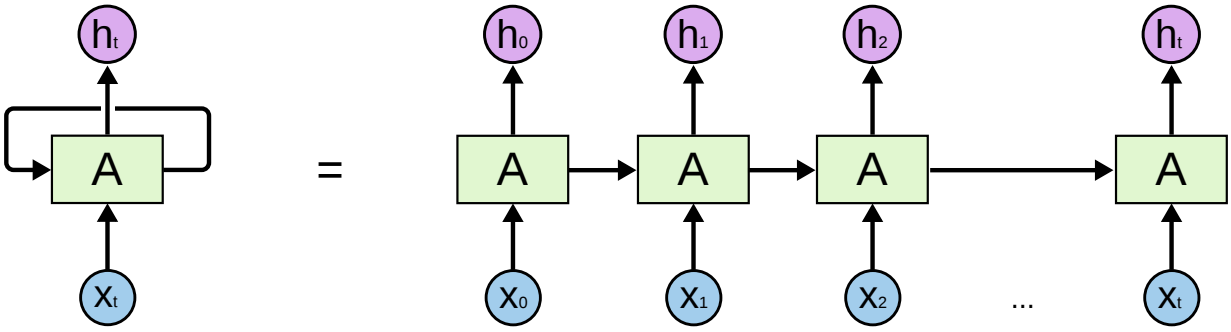


Figure 3: An Unrolled RNN [20]

To train the network, we must use a modification of backpropagation, called backpropagation through time. This algorithm propagates errors not only backwards through the network, but backwards through the feedback loop, so as to allow the network to update through the feedback loops. In terms of taking the gradient, the feedback loop is treated as a deeply nested function, requiring only the chain rule to provide the correct gradient. To improve training time and improve accuracy, backpropagation through the feedback loop is truncated to a finite number of passes.

However, there is an inherent problem with this model of an RNN. Called the Vanishing Gradient Problem, as the backpropagation through time algorithm proceeds through though the feedback loop, the activation function causes the gradient to exponentially decay [4]. This causes the network to have problems recognizing long term relationships in sequences. In modern RNN's, the problem is solved by using cells that are very different from that of a standard feedforward neural network. The most common type of cell is called a Long Short Term Memory cell [13]. Each cell contains multiple logic gates, called forget gates, which allow it to control the flow of previous information in future time steps of the network. They also prevent the exponential decay of the gradient, allowing the network to be properly trained.

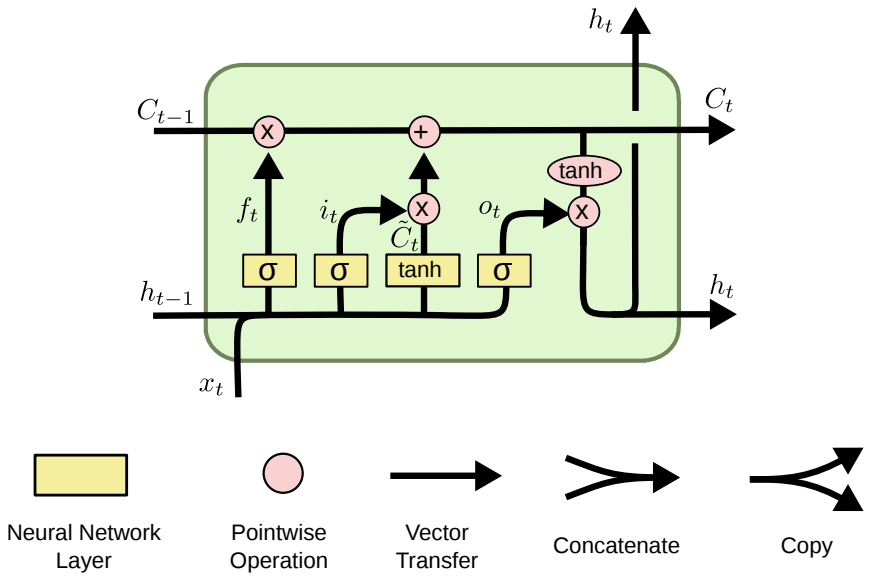


Figure 4: An LSTM Cell [20]

As depicted above in Figure 4, an LSTM cell has two hidden states that each uniquely transmit information through pointwise operations with the input and the previous hidden states. Most RNN's today use some version the LSTM architecture, but as of 2014 a simpler architecture called a Gated Recurrent Unit (GRU) has been gaining popularity in the research community [8].

Secondary Structure Prediction

Secondary structure prediction is the process of predicting the conformation of a polypeptide string given any valid polypeptide string. This conformation takes the form of either a α-helix or β-sheet. For secondary structure, each amino acid can take on only 1 conformation, out of a specified set of conformations. This makes secondary structure prediction an inherently more tractable problem than tertiary structure prediction. While tertiary structure prediction requires an algorithm to predict the 3 dimensional location of each atom of the polypeptide, secondary structure prediction is only required to apply a label of conformation to each amino acid. This is a well known problem in machine learning, known as classification.

One of the earliest methods for secondary structure prediction was the Chou-Fasman method, proposed in 1974 [27]. This method simply calculated the probability of every amino acid being part of an α-Helix, a β-Sheet, or neither, normalized for the number of times the respective amino

acid appears. This method, while a good beginning step, was very simplistic and naive. First and foremost, it does not take the surrounding amino acids into account for a particular protein, making it impossible to use previously predicted structure states to make the current prediction. Furthermore, it does not account for long range interactions that help to stabilize the α-helix and β-sheet structures.

Later, in 1978, the GOR method improved the approach and created a mathematical formalization of the problem [27]. The key was to define the problem in terms of information theory; each amino acid is viewed in context of the whole amino acid sequence, with the goal for each amino acid being to predict the conformation type with the highest information [10]. Mathematically, this goal appears as $\arg\max_{S_i} I(S_i : R_{i-j}, R_{i-j+1}, R_{i-j-2} \ldots R_{i+j-2}, R_{i+j-1}, R_{i+j})$, defining information as the following in Equation 3:

$$I(y : x) = \log\left(\frac{\Pr(y|x)}{\Pr(x)}\right) \tag{3}$$

To fully calculate this value would be impossible even on todays computers. To over come this, a simple approximation is put forward in Equation 4:

$$I(S_i : R_{i-j}, R_{i-j+1}, R_{i-j-2} \ldots R_{i+j-2}, R_{i+j-1}, R_{i+j}) \approx \sum_{j=-n}^{n} I(S_i : R_{i+j}) \tag{4}$$

This simplification allows us to calculate the information given by each amino acid surrounding the current amino acid. To further simplified the problem, instead of taking all other amino acids into account, one can specify a small window of size m such that $m < n$, dropping the total number of information residues to be at max $2m$. In the original paper, they choose $m = 8$ as their window size, which means the surrounding 13 amino acids inform the prediction of the current amino acid configuration. While this method addresses the problem of taking surrounding information into context, it fails to examine the configuration of the surrounding amino acids.

Other approaches that have been tried include nearest neighbor algorithms, which attempt to use nearest neighbor based classifiers by taking the current amino acid with the surrounding amino acid, and using the algorithm to identify other amino acid sequences that are similar and have

been labeled with their conformation. By examining these sequences, the algorithm is able to give a predicted conformation for the amino acid. This algorithm and extensions of the previously mentioned algorithms have achieved higher than 60% accuracy, which is a significant improvement over the accuracy achieved with random guessing, but not enough to become a viable alternative to experimental structural prediction methods [27].

Starting in 1989, neural network based methods started to come into practice. These networks were very shallow and simple in contrast to today's ever deepening networks. One of the most successful neural network approaches was that of Rost and Sander [24]. This method used a 2 layer neural network that included a multiple alignment of the protein against other similar proteins. The first layer would examine different sections surrounding the current amino acid being classified, and proceeds to classify the structural conformation of these independent, surrounding sections. The second layer takes these independent predicted conformations and then processes them to predict the conformation of the current amino acid. Finally, a jury decision layer to help reduce noise in the neural network architecture, after which the highest conformation class is used as the predicted conformation. This method was able to achieve an accuracy of 70.8% accuracy for basic 3-state secondary structure prediction.

Currently, neural networks are replacing previous generation methods such as Hidden Markov Models. One of the most recent attempts is that of Wang, Peng, and Xu [31]. This proposed an algorithm called DeepCNF (Conditional Neural Field); this algorithm was derived from standard neural networks and a Conditional Random Field (CRF), an algorithm associated with prediction and classification on a sequence. A traditional CRF allows for prediction, while taking previous and forward context into account. Using this information, it uses a simple linear classifier to make its prediction. In the case of DeepCNF, the linear classifier is replaced with a convolutional neural network. In their paper, Wang et al. test multiple network architectures, of 1, 3, 5, and 7 layers, with the 7 layer model giving the best 8-state accuracy. In summary, they were able to achieve a 3-state accuracy of 84.7% and an 8-state accuracy of 72.3%.

Despite this constantly improving accuracy, there are limits to using algorithmic based methods

to predict secondary structure in proteins. In fact, it is estimated that there exists an upper limit of accuracy around 88% [23].

Approach

Encoder / Decoder Architecture

While Recurrent Neural Networks as described above are great for processing sequences, they are only able to output a single value for the entire sequence. For example, in the case of sentiment analysis, a Recurrent Neural Network is able to process a sentence then provide a single number to rate the sentiment inherent to the sentence as either good or bad and how good or bad it is. This is useful for many problems. However, in the field of machine translation, more than one output is required. In fact, an entire sequence is the output. This lead to the creation of Encoder / Decoder, known more colloquially as Seq2Seq models, which allows translation from one sequence to another [8]. It consists of two Recurrent Neural Networks placed together, with one focused on learning the input sequence representation, called the Encoder, and the second RNN, called the Decoder, is focused on outputting a translation of the sequence. This technique has been shown to be highly successful at Machine Translation, and is being used as the new translation algorithm for Google Translate [32]. This network is shown depicted below in Figure 5.

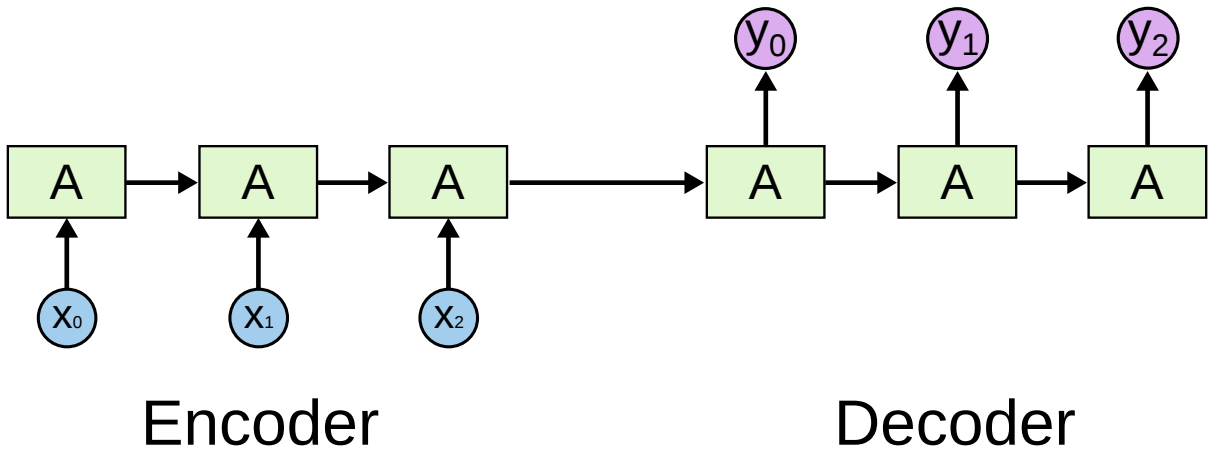


Figure 5: An Encoder / Decoder RNN a single layer for each [20]

As we can see, Figure 5 depicts the Encoder processing the input data X_1, X_2, X_3 and learns the

inherent meaning of the input. From there, the Encoder feeds that information as a vector into the Decoder, where the Decoder is able to take the meaning of the sequence and translate it into the language that it has been trained for. As it is a Recurrent Neural Network, it is trained by the Backpropagation through Time algorithm (BTT), but usually takes longer than a standard RNN, due to the fact that two networks that must be trained. For the problem of translation, we have the Equation 5 as the loss function for the network.

$$\max_{\theta} \frac{1}{N} \sum_{i=1}^{N} \log(P_{\theta}(y_i|x_i)) \tag{5}$$

With this loss function, we can see that for the model parameters θ, we are trying to optimized the log loss probability that y_i will appear given x_i for all i. Overall, this means that the neural network will learn to mimic the translated output, given the input.

For this study, we will be using a modification of the basic Encoder / Decoder algorithm. One of the problems with a standard Recurrent Neural Network is the fact that it can take past context into account, but not future context. Future information can be just as important as past information, especially in the case of secondary structure, where all of the surrounding amino acids affect the structure of a given amino acid. To allow for RNNs to account for this future context, Bidirectional RNNs were created. First proposed in 1997, the network incorporates future context by doubling the number of hidden states in each RNN / LSTM cell, with half being for past conxtext and the other half being for future context [25]. This means that an individual RNN / LSTM cell recives a forward and a backward hidden state, increasing the accuracy of the prediction. This technique has been used successfully in protein secondary structure prediction algorithms, and have been found to do better than state of the art methods in 2005 [21]. This type of network is used within the Encoder of the Encoder / Decoder model.

Given this recent architecture, I believe it will prove successful at predicting the secondary structure of proteins and rank very high in terms of accuracy of prediction.

ProtVec

An inherent problem processing vocabulary sequences is information sparsity. Any given sequence may only contain a very small subset of possible words, each of which posses a complex relationship with many other words. To counter this, a technique known as word embedding tries to compress this relational information into a continuous vector space [17]. In a word embedding system, each word is represented by a high dimensional vector, where the vector represents its relationship to other words in the high dimensional vector space. One of the unique properties of a good embedding is the fact that relationships between words can be represented through vector arithmetic. For example, in a good embedding the equation $\vec{w}_{Berlin} - \vec{w}_{Germany} + \vec{w}_{France} \approx \vec{w}_{Paris}$ holds, showing that within a word embedding, there exists a linear relationships between each word. For this project, we hope to capture the relationships between three-mers in the protein sequences.

While word embedding techniques have traditionally been used exclusively for natural language processing and machine translation tasks, they have been successfully applied to biological sequences. Called BioVec, DNA and protein sequences can be successfully embedded in a vector space using the Skip-Gram model using three-mers as the words that make up the vocabulary of proteins [2]. In this paper, researchers were able to represent the protein sequence using the three-mer embeddings as input features to an Support Vector Machine classifier, for the problem of protein family classification. Researchers were able to achieve 93% accuracy on this task. This demonstrates that proteins sequences, like sentences, contain relational information between their three-mers / words.

To build this embedding, we build a linear (without an activation function) neural network with a single hidden layer. The input layer contains a number of neurons equal to the size of the vocabulary. The hidden layer is represented by the embedding matrix. Given this network, we use one of two possible algorithms, Continuous Bag of Words and Skip-Gram, to train the network and build the embedding matrix. For this project, we will be using the Skip-Gram algorithm, as it produces better results with words in the vocabulary that are rare. The algorithm can be described as training the neural network to predict the surrounding words, given an input word. In the first variation of this

algorithm, the log loss of the softmax equation was used as the error function, J in Equation 6

$$P(w_c|w_t) = \frac{e^{\vec{w}_c \cdot \vec{w}_t}}{\sum_{v \in Words} e^{\vec{v} \cdot \vec{w}_t}} \tag{6a}$$

$$\log(P(w_c|w_t)) = (\vec{w}_c \cdot \vec{w}_t) - \log\left(\sum_{v \in Words} e^{\vec{v} \cdot \vec{w}_t}\right) \tag{6b}$$

$$J(W) = \frac{1}{T}\sum_{t=1}^{T} \sum_{-c<j<c, j\neq 0} \log(p(w_{t+j}|w_t)) \tag{6c}$$

However, this error function becomes intractable for large vocabularies, requiring on the order of 10^5 to 10^7 terms to be computed for each gradient [16]. To combat this, two methods, Negative Sampling and Noise Contrastive Estimation, were put forward as simpler error functions that achieve good embedding results [18]. Although Negative Sampling might be considered to be the better methods, we will be using Noise Contrastive Estimation, due to its native implementation in TensorFlow. Noise Contrastive Estimation reduces the number of parameters to learn by requiring that individual words be tested against the input word. This can be represented by Equation 7 below, where the probability distribution produces the probability of a given word pair occurring together. The equation uses the notation provided to us in the original paper by Minh and Kavukcuoglu.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{7a}$$

$$\Delta s_\theta(w,h) = s_\theta(w,h) - \log(kP_n(w)) \tag{7b}$$

$$P^h(D=1|w,\theta) = \sigma\left(\Delta s_\theta(w,h)\right) \tag{7c}$$

$$J^h(\theta) = E_{P_d^h}\left[\log\left(\sigma\left(\Delta s_\theta(w,h)\right)\right)\right] - kE_{P_n}\left[\log\left(1-\sigma\left(\Delta s_\theta(w,h)\right)\right)\right] \tag{7d}$$

This new error function approximates the logarithmic softmax output. The only impediment to this is the requirement of the noise probability distribution, P_n. This must be manually calculated by

Implementation

Models

For this study, we test three models, a small model, a medium model, and a large model. The first three models are pure Encoder / Decoder models and are the core of this study. Each of these models are based off of the example models provided in the tf-seq2seq source code. For all models, we will use an embedding size of 128 for all models, as we do not need as large of a word embedding as networks used for language translation. Moreover, we are using LSTM based cells in all of the Decoder networks and the first three Encoder networks. All model configuration files can be found in the Appendix section of this paper. The files contain a few other parameters that are not mentioned in this paper, but are taken from the configuration files that were included in the tf-seq2seq package.

The small model was built with the primary motivation being to show basic evidence for the success of an LSTM based Encoder / Decoder system on the problem. It consists of one Encoder layer and one Decoder layer. This model will be very easy to train, and will only take a few hours.

The medium model attempts to deepen the Decoder network to increase accuracy. I consists of one Encoder layer and two Decoder layers. This model will take a bit longer to train, and will probably take a whole day to train on the compute cluster.

The large model increases both the Encoder and Decoder depth. It consists of two layers in the Encoder and four layers in the Decoder. I expect to get the highest accuracy on this model, as it is the most complex. However, this accuracy comes at the cost of a long training time. This model will take between one to three days to train for 5,000,000 steps.

Moreover, there are multiple ways to test the accuracy of a secondary structure prediction algorithm. The two most popular are Q3 and Q8 accuracy. Simply put, these two accuracy measures simply tally the percent amino acids the algorithm correctly predicted. Now Q3 and Q8 each use different labeling classes, with Q3 using the standard three secondary structure classes, α-helix,

β-sheet, and coil. Q8 on the other hand, uses eight individual classes, which are subclasses of the Q3 structure types. The Q8 classes are described in the Table 1 below.

Table 1: Q8 Classes as defined by PDB [5]

Structure Letter	Structure Type
H	α-Helix
B	Residue in isolated β-bridge
E	Extended Strand, participates in β-ladder
G	3-Helix ($\frac{3}{10}$ Helix)
I	5-Helix (π Helix)
T	Hydrogen Bonded Turn
S	Bend
C	Coil

Attention Mechanism

A common mechanism that is used to enhance the predictive power of Encoder / Decoder architectures is the use of a technique called Attention. First introduced by Dzmitry Bahdanau in 2014, this allows Encoder / Decoder networks to focus individual outputs from the Decoder on the pertinent inputs to the encoder [3]. When translating a sentence, the translator focuses mostly on the current word and slightly on a few surrounding words, attention mechanisms attempt to mimic this by deemphasizing the output of Encoder cells that are farther away from the current word being translated. This is accomplished by passing a vector, called the query vector, from the Decoder to the encoder. Then we take the dot product of the query vector and the output vectors from the Encoder network, and put it through a softmax layer. The output of the softmax is called the attention distribution, and decides which parts of the input are the most pertinent to the current decoder state. From here, each scalar value is multiplied by their respective Encoder vector and summed together to get the final input to the the Decoder cell. This process is depicted below, in Figure 6. Note how for the Decoder cell certain Encoder vectors are weighted much more strongly.

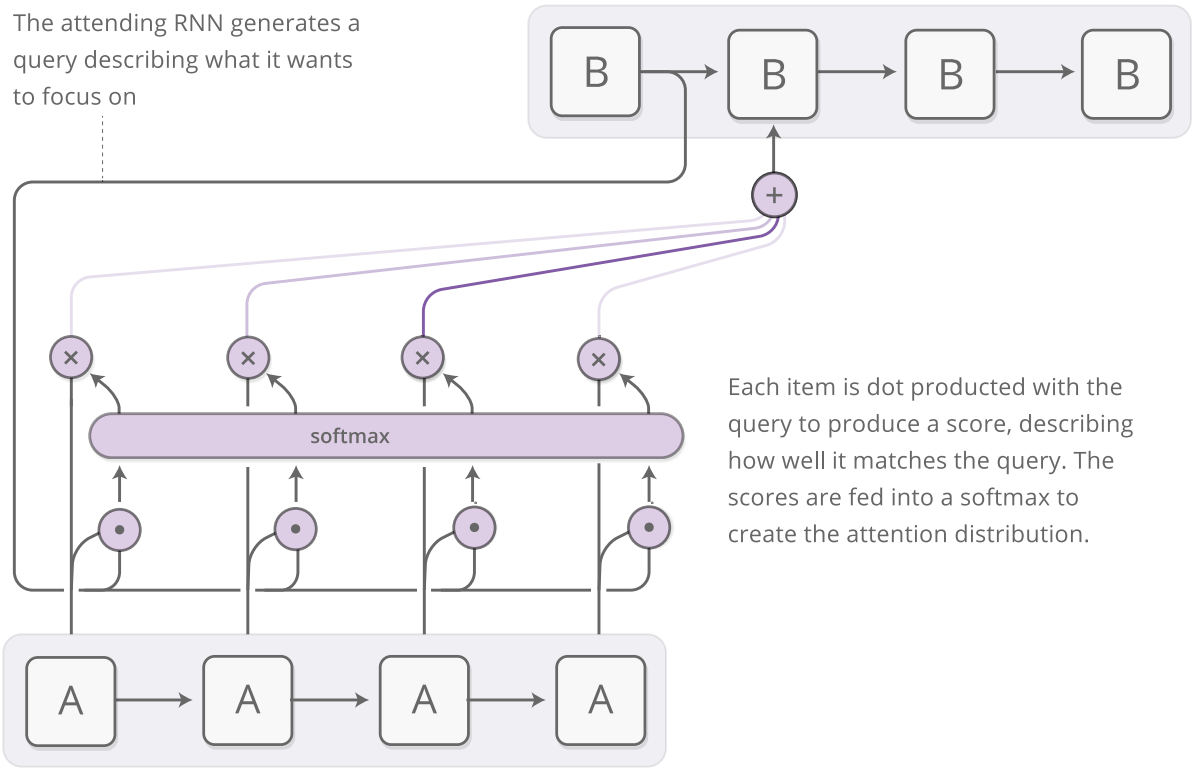


Figure 6: Attention Mechanism in an Encoder / Decoder network [19]

The addition of the attention mechanism has the benefit of allowing each Encoder state to output a vector of information for the Decoder instead of a single vector from the final output of the Encoder. This increases the amount of information that is given to the decoder, allowing it to make better predictions. In the three models we will be testing, the attention vector will be of length 128.

TensorFlow

One of the keys to implementing this project has been the Tensorflow library [1]. Released by Google in 2015, the library focuses on providing an all inclusive neural network library. The user is given many primitive operations to construct the network; this network must be defined before beginning the training phase, so that the library is able to pre-calculate the derivatives and compile the math expressions required for backpropagation through the network. By doing this, the library is able to optimize the computation of both forward and backwards passes through the network. As with every other major neural network library, it includes CUDA support, allowing GPU acceleration to greatly reduce training time. One of the more unique features of the library is its new XLA

accelerator. XLA is a Just-in-Time (JIT) compiler, designed specifically for compiling high level graph operations together, so as to speed up both the training and inference times of the network. Furthermore, all training data can be recorded, and easily accessed through the Tensorboard interface. From here, we are easily able to export all scalar data (learning rates, gradient magnitude, and network cell parameters) via a CSV. Instead of installing the vanilla TensorFlow package available from the Python package manager, I chose to compile TensorFlow from source, gaining me extra vector extensions and the XLA accelerator; these features will help to reduce training and inference time. For this project, we will be using version r1.0, released on February 15, 2017.

Running on top of Tensorflow is the newly released tf-Seq2Seq package [6]. Open-sourced on April 11, 2017, this package allows the user to build Encoder / Decoder networks with little, to no code. The network architecture is passed into the program via a single YAML file that contains specifications such as the number of hidden layers, the type RNN cells to be used, learning rate, etc. This helps to make model easy to distribute, as only a small set of files need to be provided to run a trained model. One of the main benefits to using this package is that it natively supports multi-GPU acceleration, along with distributed computing among multiple compute nodes. This allows the use of large clusters for training the model quickly. Furthermore, this package helps to improve reproducibility, by standardizing the code and providing the precise hyper-parameters in a small number of configuration files. The data is fed into the network via two separate files, the source and target files, which are in Parallel Text Format. With the release of this package, all but my preprocessing code was made obsolete. As such, I quickly re-factored my entire software pipeline to incorporate this package.

Regularization

One of the problems we come across when training a neural network is the problem of overfitting. This occurs when the network begins to learn specific examples in the training dataset instead of learning the ground truth behind the data it is classifying. This is easily detected through the use of a validation dataset. If the training error and the validation error diverge, with the validation

error beginning to plateau or increase, then overfitting has occurred. To prevent this, we must apply regularization to the network. Regularization can be liken to applying Occam's Razor to the neural network, it forces the network to simplify its These methods range from changing the error function to stopping the training when the validation and training errors diverge.

One of the most successful regularization methods, called dropout, is also one of the simplest. First introduced in 2012, dropout is the process of randomly zeroing out neuron outputs from each layer in the network during the training phase [28]. The probability that any one neuron is zeroed out, is mutually independent of all other neurons. This means that the total number of neurons that get zeroed out in any training step is distributed according to a binomial distribution. Using the Maximum Likelihood Estimator, we are able to realize that the average fraction of neurons that are zeroed out in any single step is approximately equal to the dropout probability. In most cases, a dropout probability of $\frac{1}{2}$ is the most effective.

In the case of this project, we will be using dropout on the medium and large models, with a dropout probability of 0.8, recommended by tf-seq2seq. I chose to not apply dropout to the small model because of the low depth, making overfitting not as much of a problem. This is also the only type of regularization that we will apply, as something like L_2 regularization add to the complexity of the network, and increase the training time. Since the large network will take a number of days as is to train, it is best to stick to methods that don't increase the runtime.

Optimizers

As explained earlier, neural networks are trained through stochastic gradient descent, minimizing the error function for the given training dataset. On most modern neural networks, pure stochastic gradient descent is not powerful enough and can easily get trapped in saddle points or deep valley in the topology of the error function. To prevent this, we modify SGD to prevent the algorithm from becoming trapped in these topological features. The most well known modification, called momentum optimization, treats the algorithm as if it were a physical system [29]. It uses the gradient as a force, while having a momentum parameter, which holds a cumulative sum of all previous

gradient updates, much like momentum in classical mechanics. In the case of the algorithm falling into a large valley or saddle point, momentum allows the algorithm to escape these false minima by slowly growing the magnitude of the momentum vector, until it is able to escape the false minima.

For this study, I will be using the Adam optimization algorithm. This algorithm combines the power of momentum based optimization with dimensional scaling, a technique that changes the learning rate for each dimension, independently of all other dimensions [14]. These two improvements to stochastic gradient descent help to make Adam one of the most popular optimization algorithms for training neural networks. In the algorithm, there are four hyper-parameters: β_1, β_2, ε, and η. The two β parameters both act as decay rates for the momentum factors. The η parameter is the actual learning rate of the optimizer, which affects the magnitude the weight updates in the algorithm. The ε parameter is not clearly described, but has been called a fuzz factor. If we examine the expression it appears in $\theta_t = \theta_{t-1} - \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}$, where $\hat{m}_t$ and $\hat{v}_t$ are the two momentum parameters and θ is the parameter/s being optimized, we can see that ε affects the parameter which the authors call the, "Signal to Noise Ratio". This means that the ε parameter affects how noise associated with gradient of a complex function, then goes on to update the parameter. The values we will be using for them are the default values in TensorFlow, which are the recommended values in the original paper. For the two β values, we have the following settings $\beta_1 = 0.9$ and $\beta_2 = 0.999$, implying almost no momentum decay. For ε, we use $\varepsilon = 0.8 \times 10^{-6}$ for all models. For the learning rate, we are using the default value prescribed by the tf-seq2seq package, $\eta = 10^{-4}$.

Dataset

The data for training and testing these models comes from the Protein Data Bank (PDB) [5]. Started in 2000, PDB now contains over 129,367 protein sequences, including their 3 dimensional (tertiary) structure. The data, given in the form of a FASTA file, consists of protein sequences and their 8-state secondary structure labeling. However, we can not just train directly on the PDB data. Because many protein sequences are highly similar, except for a few amino acids, the algorithm can return highly inflated accuracies. To prevent this, we must first cull sequences that are too similar from the

dataset. To do this, we use the Culled PDB service PICES [30]. I used the following settings for culling, shown in Table 2.

Table 2: PISCES Culling Parameters

Parameter	Value
Maximum percentage identity	25%
Maximum Resolution	3.0
Maximum R-value	0.6
Minimum Chain Length	40
Maximum Chain Length	10000

I processed this data by simply parsing the FASTA file and removing the excess information included, such as sequence names. From this point, I split the sequences into their source and target sequences and placed them into two files, each in Parallel Text Format. As in any standard machine learning algorithm, I created two sets of data, testing and training sets. The testing and training sets were split randomly after parsing out the sequence names and delimiting marks. The test dataset is approximately 10% the size of the training dataset. I chose not to build a validation dataset, as the initial dataset is already on the smaller end of those used to train neural networks. The longest sequence in the dataset is 1739 amino acids long.

Computing Resources

Neural networks are inherently computationally intensive to train, as there is no closed form equation to optimize the weight matrices and other hidden variables. On simple desktop machines, state of the art neural networks would take months to train, and would probably strain the memory of the machine. That is where the use of Graphics Processing Units (GPUs) are essential. Due to their vector heavy design, they are able to do large amounts of vector and matrix math quickly, making them the perfect tool for accelerating neural network training and inference.

For this project, we used multiple clusters, each with their own configuration. This allowed us

to train almost all of the networks at once. However, the downside to this is the fact that speed comparisons will not be as useful. However, for the purposes of comparison, we can assume that the addition of another GPU gives a linear speed-up. Therefore, we can just multiply the total training time by the number of GPUs to give us a reasonable value to compare the training times. The main clusters consisted of Amazon Web Services EC2 instances, Google Cloud Compute Engine instances, and running models on the Department of Computer Science's Ionic cluster. Each of these services / clusters used the same model GPU, an Nvidia K80 Tesla. This model GPU is currently the standard model used for training neural networks. It offers 8.74 TFlops of computing power and 24 GB of memory, allowing it to power through large numbers of matrix multiplications. The first clusters we will be using are the 16 GPU cluster and 8 GPU cluster. On Google Cloud Compute Engine, we will be using another 8 GPU instance. For small test runs, we have the Ionic cluster with a total of 4 GPUs split over 2 nodes.

Results

The results were assembled by first training the models for a reasonable number of epochs over the dataset, and then examining the results from the most recent training output file. At that point, I had the model process the input test sequences and output a file containing all of the predicted secondary structure sequences. The models were trained to output Q8 prediction based on the culled dataset. All of the models output save files every 1000 time steps, allowing us to use models from a previous training step. This helps to prevent against the possibility of overfitting, by allowing us to jump back to the point before the model started overfitting.

All of the models require a large amount of helper code provided in the tf-seq2seq package. Figure 7 below, generated by TensorBoard, depicts this, as each node is a major section of the program.

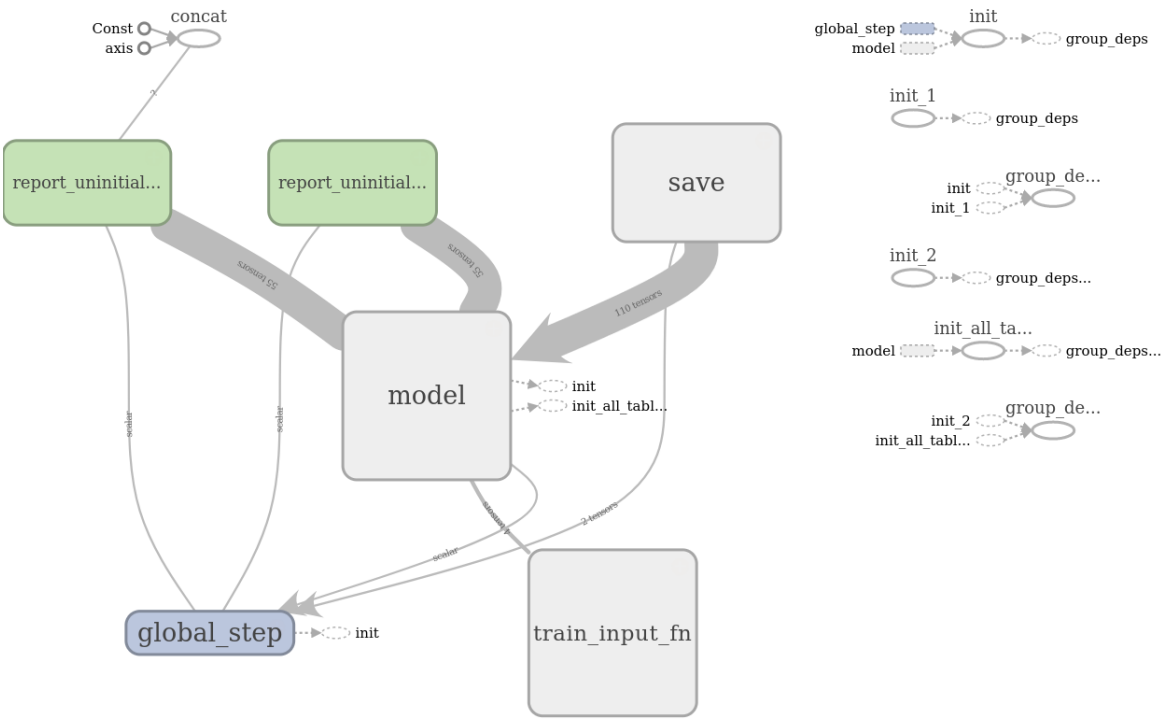


Figure 7: Computational Graph of tf-seq2seq model

We can see that the neural network consists only of the model block, but we have assisting code that helps to make the model flexible and failure resistant. In the Appendix of the paper, I include diagrams of the internals of the neural network, such as the layout of the Encoders and Decoders for some of the models.

Small Model

The small model was quite easy to train, and

Figure 8: Training Loss Curve for Small Model

Medium Model

The medium model was the hardest model to train, for reasons that will be explained in the Computation Time section. Unfortunately, due to this, I was only able to train the medium model for time steps. While this initially sounds bad, the medium model has some of the fastest convergence behavior, converging to a loss in the low 2's. This can be seen in the graph below

Large Model

The large model was surprisingly easy to train, but time consuming. This model took approximately 16 hours to train to 2400 time steps. As it was the largest of the three models, I ran it on the 16 GPU machine to give it the most GPU memory and processing power.

Figure 9: Training Loss Curve for Large Model

When examining the embedding average weights, I noticed that they trained particularly quickly in comparison to the rest of the model. This is displayed in Figure 10 below.

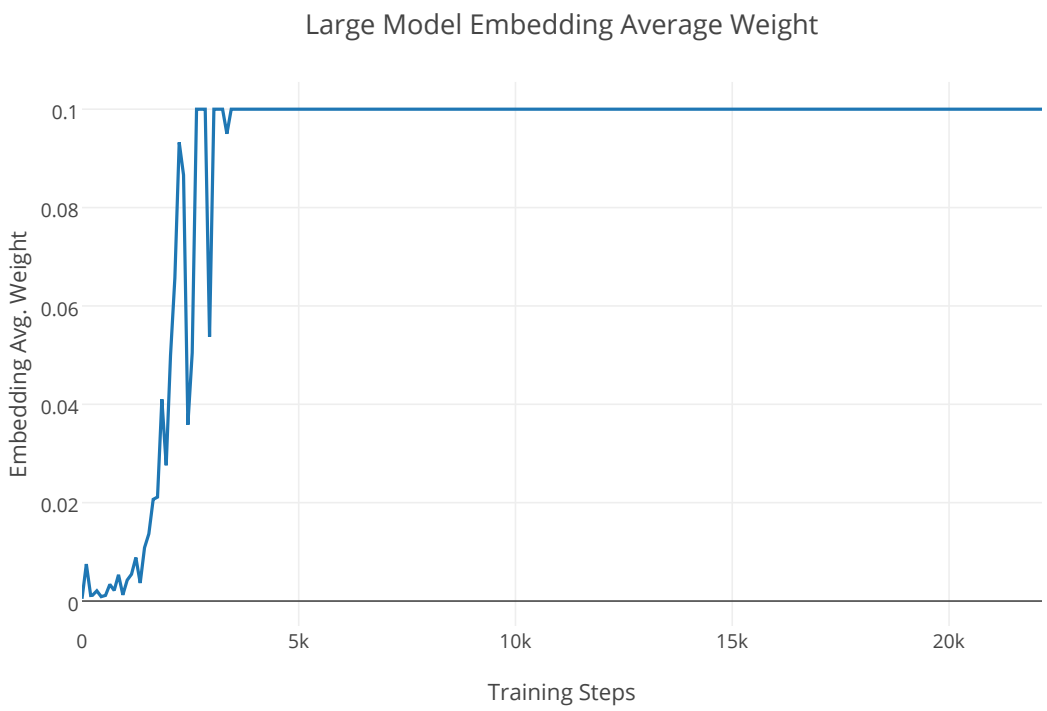


Figure 10: Embedding Avg Weight Training Curve

Computation Time

As mentioned earlier, to complete the training and inference of the networks, I had to run them on multiple machines, each with their own unique set of hardware specifications. Therefore, any of the following time results must be viewed with this fact in mind. However, provided below in the table are the number of training steps executed per second and how many GPU's available for training the model.

Table 3: Computation Time for the three models

Model	Service	# GPUs	Avg. Steps / Sec	Steps / (GPU * Sec)
Small	AWS	16	3.3	0.20625
Medium	Google	8	0.08	0.01
Large	AWS	16	0.32	0.02

As we can see, the small model was able to train the fastest, training approximately 10 times faster than the large model on the same configuration and 20 times faster than the medium model.

This is expected as the Backpropagation through Time algorithm must only backpropagate the errors one layer back for each network. Overall, I found that the small and large models trained in a reasonable amount of time, given the configuration of the server they were running on. The medium model

As a small note, I found the models trained on Google Compute Engine to be significantly slower for some unknown reason. It seemed as if the Google instances had more lag in comparison to the Amazon Web Services instances. I was unable to debug the problem, and as such the amount of time that the medium model could train for was reduced. This is highly unexpected, since the tf-seq2seq code advertised a training time of 2-3 days for the large model, on 8 Nvidia K80 GPUs [6]. Given that they considered 5,000,000 training steps fully trained, and that the medium model is half the size of the large model, I find these training time claims to be highly inflated. While the actual training times are still reasonable compared to other state of the art neural networks, I believe that the Google team should reexamine the runtime of the code in order to provide a much more reasonable estimate, or qualify their runtime with further data.

Summary

Although I did not train these models to the 5,000,000 training steps as done in the tf-seq2seq tutorial, I believe that the amount of training done was sufficient to extrapolate data from the models. However, due to the flexibility of the tf-seq2seq software, the models can be trained further for any number of training steps, given the running files. All code, datasets, and TensorFlow model files can be found in the github repository located at `https://github.com/PrincetonUniversity/ProtNet`. By the time this paper is finished, it will be available to the public. The final Q8 accuracies are displayed in Table 4 below.

Table 4: Q8 Accuracy of the Models

Model	Q8 Accuracy
Small	
Medium	
Large	

Conclusion

Overall, I am quite satisfied with the results of this study. From the accuracies achieved, we can clearly see that these models are capable of predicting secondary structure. It would be worth studying these models further by tuning the hyperparameters better, finding ways to expand the dataset by using a better culling algorithm, or

For future work, I would like to investigate the incorporation of multiple alignment data in the input to the network. Integrating multiple alignment data has been shown to be critical in improving the accuracy of secondary structure prediction. It was originally suggested that I use this as my main architecture for this study; however, I wanted to focus on the ability for this type of network architecture to inference using only the input sequence. One of the main difficulties is that it would require protein BLAST (BLASTP) to be packaged offline, as the network would prediction code would need to call BLASTP for every new protein sequence. However, once this is done, one would have to change the network architecture to accept this extra alignment information. I propose the following modifications to allow for this. First, instead of using the embedding vectors on three-mers in the amino acid sequence, we use the embedding vectors to represent a single column of a protein alignment. Consequently, this would mean that the number of embedding vectors to represent the entire sequence would be equal to the length of the amino acid sequence. This would give the extra benefit of allowing the network to perform conformation prediction for individual amino acids instead of individual tri-mers. One of the pitfalls, however, is that it will accept only a constant number of alignment proteins for a given input protein. Furthermore, total vocabulary size

of the embedding would increase significantly, as it must now include alignment symbols in each embedding "word".

A convolutional model would be worth trying, as it strays from the standard LSTM based Encoder architecture.. Convolution is a complex mathematical operation that involves two functions. In the case of neural networks, we treat the input as one of the functions. The other input, called the kernel, is learned during training. The benefit to using convolution in a neural network is that compared to more traditional neural networks, the number of learned parameters per layer, making the network more memory efficient and easier to train. Convolutional based Encoder networks were first introduced in 2016 as a way to bring the benefits to convolution to Encoder / Decoder networks [11]. A simple starter model to test this concept would be one where the Encoder would have a total of six convolutional layers, each with a kernel size of three. Moreover, the Decoder would consist consists of one LSTM layer.

One of the key difficulties training machine learning algorithms on secondary structure data is the comparably small size of the dataset in comparison to other tasks. A simple way to compare this is by looking at the raw file size of the dataset. While the entire PDB secondary structure dataset can be fit in a single 196.7 MB FASTA file, where as standard text corpora range from as small as 580 MB to as large as 2.3 GB. Even compared to the small text corpus, the secondary structure dataset is $\frac{1}{3}$ the size. This might hinder the neural network, as it has to generalize over a smaller dataset. In most cases, the easiest way to solve this problem is to gather more data. As mentioned earlier, this is not feasible, due to the complex process of experimentally discovering the conformation of a protein. A possible method to overcome this is a recent technique called generative adversarial networks, or adversarial learning. First introduced by Ian Goodfellow in 2014, this technique uses two neural networks playing against each other in a zero-sum game [12]. One network focuses on classification while the other tries to generate new datapoints. Through this competition, both networks are able to learn the data not only through the provided training dataset, but through new data created by the generative network, given to the discriminative neural network. The new data provided by the generative network is designed to maximize the probability

that the discriminative network will mis-categorize the data. This competition causes the generative network to provide harder and harder data to the discriminative network, which helps to increase the training accuracy of the discriminative network. What makes this technique so enticing to the problem of secondary structure prediction is the generative capabilities of the network. Because the network is able to generate example datapoints, we are able to train the network on a much larger, practical dataset. One of the difficulties in this approach is the fact that adversarial networks built so far have been found to perform poorly on text-based problems. Therefore, this would be a study to be performed once the network architecture improves.

References

[1] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. Available: http://tensorflow.org/

[2] E. Asgari and M. R. K. Mofrad, "Continuous distributed representation of biological sequences for deep proteomics and genomics," *PLOS ONE*, vol. 10, no. 11, pp. 1–15, 11 2015. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0141287

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014. Available: http://arxiv.org/abs/1409.0473

[4] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar 1994. Available: http://ieeexplore.ieee.org/document/279181/

[5] H. M. Berman *et al.*, "The protein data bank," *Nucleic Acids Research*, vol. 28, no. 1, p. 235, 2000. Available: http://dx.doi.org/10.1093/nar/28.1.235

[6] D. Britz *et al.*, "Massive Exploration of Neural Machine Translation Architectures," *ArXiv e-prints*, Mar. 2017. Available: https://arxiv.org/abs/1703.03906v2

[7] A. Brunning. (2014) A brief guide to the twenty common amino acids. Available: http://www.compoundchem.com/2014/09/16/aminoacids/

[8] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014. Available: http://arxiv.org/abs/1406.1078

[9] W. Commons. (2008) Main protein structure levels. Available: https://upload.wikimedia.org/wikipedia/commons/c/c9/Main_protein_structure_levels_en.svg

[10] J. Garnier, D. Osguthorpe, and B. Robson, "Analysis of the accuracy and implications of simple methods for predicting the secondary structure of globular proteins," *Journal of Molecular Biology*, vol. 120, no. 1, pp. 97 – 120, 1978. Available: http://www.sciencedirect.com/science/article/pii/0022283678902978

[11] J. Gehring *et al.*, "A convolutional encoder model for neural machine translation," *CoRR*, vol. abs/1611.02344, 2016. Available: http://arxiv.org/abs/1611.02344

[12] I. J. Goodfellow *et al.*, "Generative Adversarial Networks," *ArXiv e-prints*, Jun. 2014. Available: https://arxiv.org/abs/1406.2661

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. Available: http://www.mitpressjournals.org/doi/pdfplus/10.1162/neco.1997.9.8.1735

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. Available: http://arxiv.org/abs/1412.6980

[15] R. Longtin, "A forgotten debate: is selenocysteine the 21st amino acid?" 2004. Available: https://doi.org/10.1093/jnci/96.7.504

[16] T. Mikolov *et al.*, "Distributed Representations of Words and Phrases and their Compositionality," *ArXiv e-prints*, Oct. 2013. Available: https://arxiv.org/abs/1310.4546

[17] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. Available: http://arxiv.org/abs/1301.3781

[18] A. Mnih and K. Kavukcuoglu, "Learning word embeddings efficiently with noise-contrastive estimation," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 2265–2273. Available: http://dl.acm.org/citation.cfm?id=2999792.2999865

[19] C. Olah and S. Carter, "Attention and augmented recurrent neural networks," *Distill*, 2016. Available: http://distill.pub/2016/augmented-rnns

[20] C. Olah. (2015) Understanding lstm networks. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[21] G. Pollastri and A. Mclysaght, "Porter: a new, accurate server for protein secondary structure prediction," *Bioinformatics*, vol. 21, no. 8, pp. 1719–1720, 2005. Available: https://doi.org/10.1093/bioinformatics/bti203

[22] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, pp. 386 – 408, 1958. Available: http://search.ebscohost.com/login.aspx?direct=true&db=pdh&AN=1959-09865-001&site=ehost-live

[23] B. Rost, "Review: Protein secondary structure prediction continues to rise," *Journal of Structural Biology*, vol. 134, no. 2, pp. 204 – 218, 2001. Available: http://www.sciencedirect.com/science/article/pii/S1047847701943369

[24] B. Rost and C. Sander, "Prediction of protein secondary structure at better than 70pp. 584 – 599, 1993. Available: http://www.sciencedirect.com/science/article/pii/S0022283683714130

[25] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov 1997. Available: https://doi.org/10.1109/78.650093

[26] S. Seung, "Lecture notes in cos 495 neural networks: Theory and applications," February 2017. Available: https://cos495.github.io

[27] M. Singh, *Predicting Protein Secondary and Supersecondary Structure*. Available: https://www.cs.princeton.edu/~mona/Chapter29.pdf

[28] N. Srivastava *et al.*, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. Available: http://jmlr.org/papers/v15/srivastava14a.html

[29] I. Sutskever *et al.*, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. III–1139–III–1147. Available: http://dl.acm.org/citation.cfm?id=3042817.3043064

[30] G. Wang and R. L. Dunbrack, "Pisces: recent improvements to a pdb sequence culling server," *Nucleic acids research*, vol. 33, no. suppl 2, pp. W94–W98, 2005. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1160163/

[31] S. Wang *et al.*, "Protein secondary structure prediction using deep convolutional neural fields," *Scientific reports*, vol. 6, 2016. Available: http://rdcu.be/qAjf

[32] Y. Wu *et al.*, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *ArXiv e-prints*, Sep. 2016. Available: https://arxiv.org/abs/1609.08144v2

Appendix

```
# small_config.yaml
# Based off of Google tf-seq2seq nmt_small configuration


model: AttentionSeq2Seq
model_params:
  attention.class: seq2seq.decoders.attention.AttentionLayerBahdanau
  attention.params:
    num_units: 128
  bridge.class: seq2seq.models.bridges.ZeroBridge
  embedding.dim: 128
  encoder.class: seq2seq.encoders.BidirectionalRNNEncoder
  encoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 128
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 1
  decoder.class: seq2seq.decoders.AttentionDecoder
  decoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 128
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 1
  optimizer.name: Adam
  optimizer.params:
    epsilon: 0.0000008
  optimizer.learning_rate: 0.0001
  source.max_seq_len: 578
  source.reverse: false
  target.max_seq_len: 578
```

Listing 1: YAML configuration file for small network model

```
# medium_config.yaml
# Based off of Google tf-seq2seq nmt_medium configuration

model: AttentionSeq2Seq
model_params:
  attention.class: seq2seq.decoders.attention.AttentionLayerBahdanau
  attention.params:
    num_units: 128
  bridge.class: seq2seq.models.bridges.ZeroBridge
  embedding.dim: 128
  encoder.class: seq2seq.encoders.BidirectionalRNNEncoder
  encoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 128
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 1
  decoder.class: seq2seq.decoders.AttentionDecoder
  decoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 256
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 2
  optimizer.name: Adam
  optimizer.params:
    epsilon: 0.0000008
  optimizer.learning_rate: 0.0001
  source.max_seq_len: 578
  source.reverse: false
  target.max_seq_len: 578
```

Listing 2: YAML configuration file for medium network model

```
# large_config.yaml
# Based on Google tf-seq2seq large model

model: AttentionSeq2Seq
model_params:
  attention.class: seq2seq.decoders.attention.AttentionLayerBahdanau
  attention.params:
    num_units: 128
  bridge.class: seq2seq.models.bridges.ZeroBridge
  embedding.dim: 128
  encoder.class: seq2seq.encoders.BidirectionalRNNEncoder
  encoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 128
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 2
  decoder.class: seq2seq.decoders.AttentionDecoder
  decoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 128
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 4
  optimizer.name: Adam
  optimizer.params:
    epsilon: 0.0000008
  optimizer.learning_rate: 0.0001
  source.max_seq_len: 578
  source.reverse: false
  target.max_seq_len: 578
```

Listing 3: YAML configuration file for large network model

```
# conv_config.yaml
# Based off of Google tf-seq2seq conv_small configuration

model: AttentionSeq2Seq
model_params:
  attention.class: seq2seq.decoders.attention.AttentionLayerBahdanau
  attention.params:
    num_units: 128
  bridge.class: seq2seq.models.bridges.ZeroBridge
  embedding.dim: 128
  encoder.class: seq2seq.encoders.ConvEncoder
  encoder.params:
    attention_cnn.units: 128
    attention_cnn.kernel_size: 3
    attention_cnn.layers: 6
    output_cnn.units: 128
    output_cnn.kernel_size: 3
    output_cnn.layers: 3
    position_embeddings.enable: true
    position_embeddings.combiner_fn: tensorflow.multiply
    position_embeddings.num_positions: 52
  decoder.class: seq2seq.decoders.AttentionDecoder
  decoder.params:
    rnn_cell:
      cell_class: LSTMCell
      cell_params:
        num_units: 128
      dropout_input_keep_prob: 0.8
      dropout_output_keep_prob: 1.0
      num_layers: 1
  optimizer.name: Adam
  optimizer.learning_rate: 0.0001
  source.max_seq_len: 578
  source.reverse: false
  target.max_seq_len: 578
```

Listing 4: YAML configuration file for convolutional network model

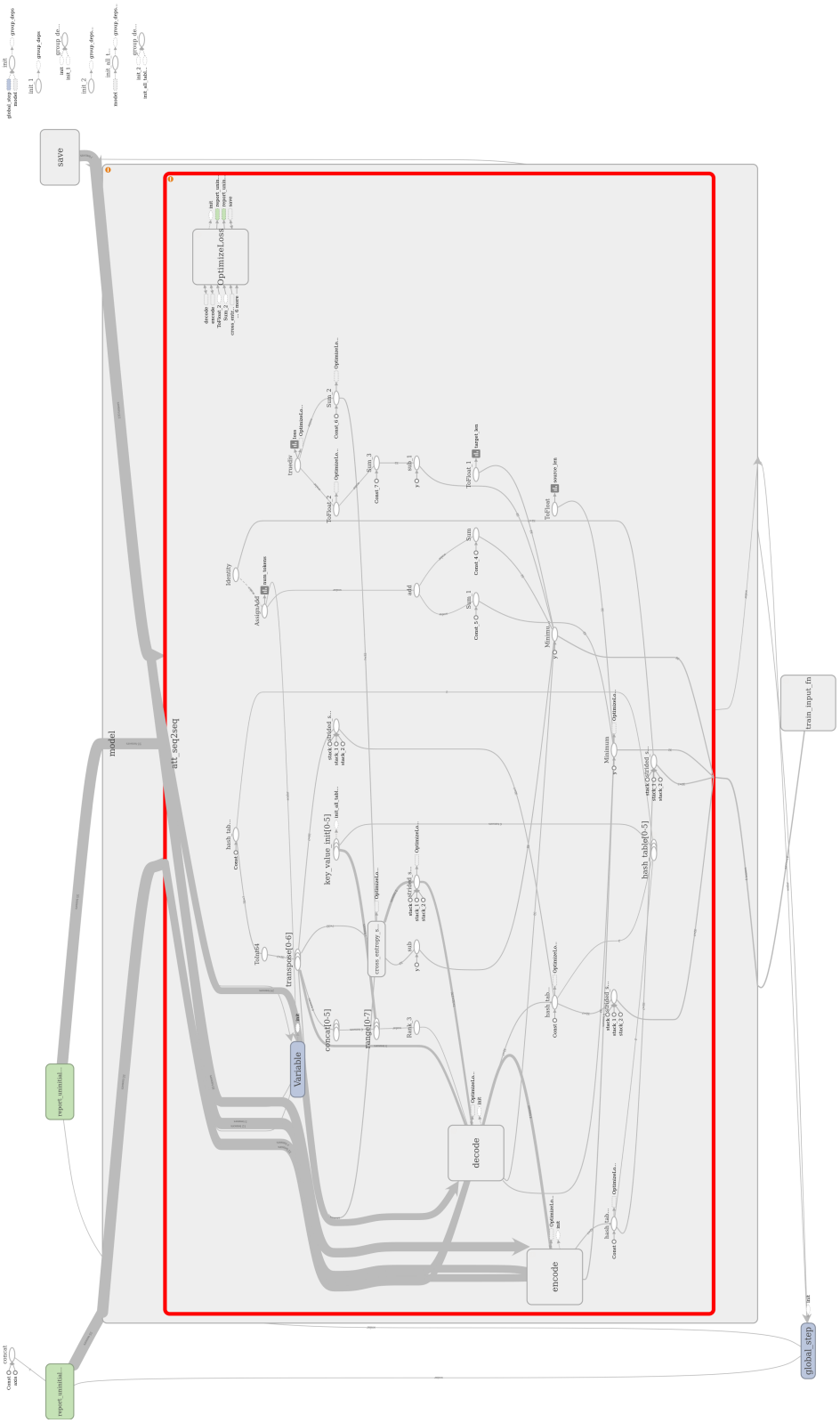


Figure 11: Computational Graph of Large Model

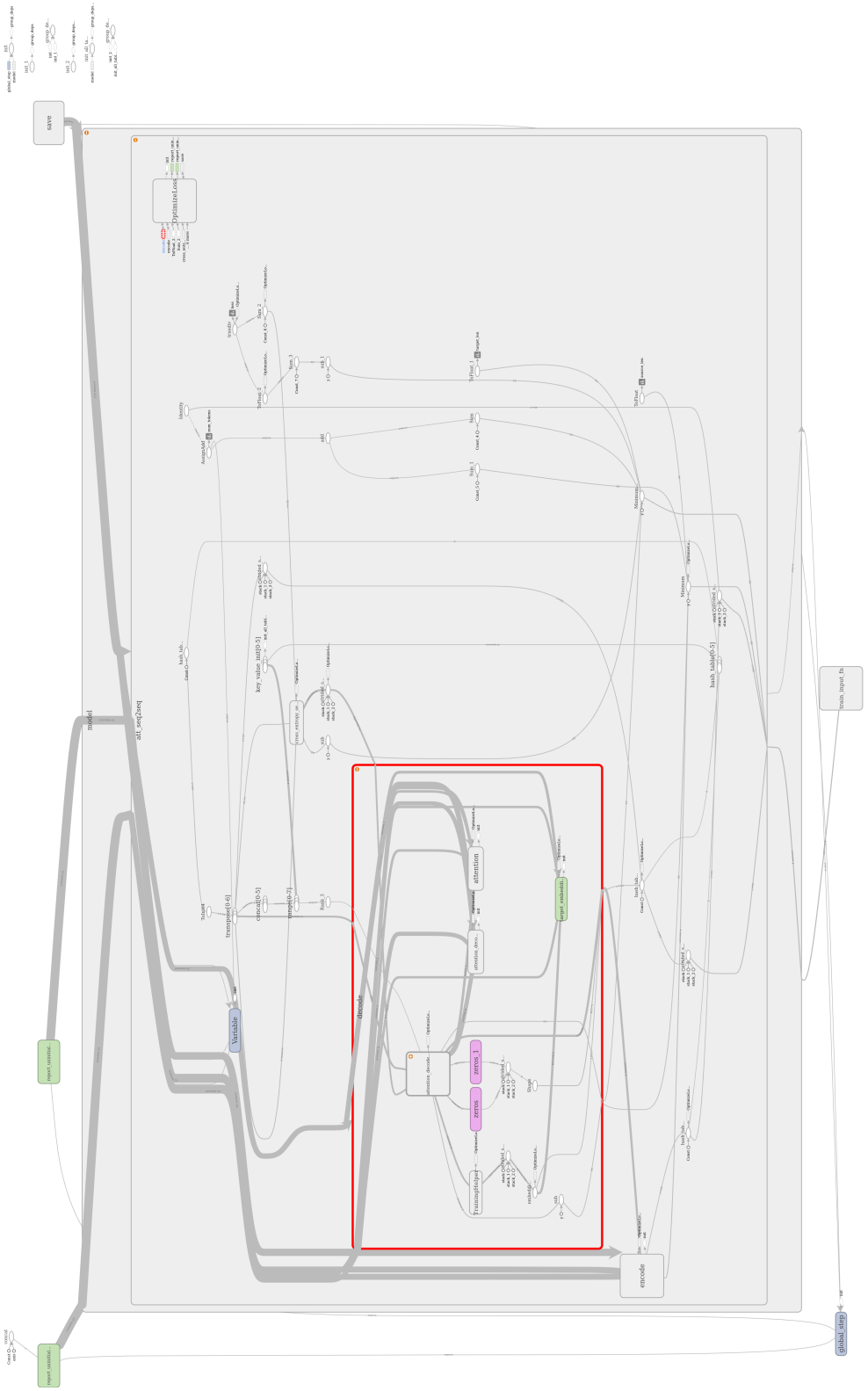


Figure 12: Computational Graph of Large Model Decoder