STAT 72556 Project

Brandon Cohen

May 11, 2024

Contents

1 Introduction

Determining the way a protein folds is synonymous with determining the stability of a protein. While there have been advancements in predicting protein folding with novel algorithms such as AlphaFold, determining protein folding is still an unsolved problem in the field of structural biology. The difficulty of predicting protein folding and hence protein stability comes from the complex process of folding that is influenced by many factors such as the sequence of the protein, temperature, pH, interaction between other molecules, etc. Currently, it is clear that if scientists want to understand how proteins fold, there needs to be more research into the properties of proteins especially with respect to their stability in different environments. This has inspired experimental techniques to measure protein stability which are cross-referenced with theoretical approaches such as molecular dynamics simulations to calculate protein stability. Recently, a new synthetic approach using de novo computational protein design on a large scale has been used

to determine stability scores [1, 2]. The scientists used parallel DNA synthesis and oligo library synthesis libraries to produce the protein of interest on the surface of cells in yeast. Using an EC50 protease resistance assay, the stability score was calculated based on the difference between the measured protease concentration that was needed to denature 50% of the protein and the predicted protease concentration in the unfolded state.

Since the two papers were successful in validating the stability scores, this report used their data to try to predict the stability score of the proteins using different machine learning models with the goal of better understanding the hidden representations between stability and protein sequences. The data was obtained from the **Protein Benchmarks** dataset from Kaggle. In the Data section, an analysis of the data distribution was analyzed. The Analyses section contained the implementation and results of training a deep neural network (DNN), long short-term memory (LSTM), gated recurrent unit (GRU), and convolutional neural network (CNN) model. Each model suffered from overfitting even when there was regularization and it was even observed that training accuracy severely decreased from regularization. Based on training each model, it was determined that the LSTM model suffered the least from overfitting and worked best at predicting the stability class.

2 Data

The data obtained from Kaggle was already sorted into a training, validation, and testing set. The training, validation, and testing data consisted of 53,614 (77.8%), 2,512 (3.6%), and 12,851 (18.6%) samples respectively with a total of 68,977 samples. As shown in Table 1, there were two columns: protein sequence and stability score. The protein sequence consisted of 19 unique amino acids and was represented by their single-letter code. For example, D represents the amino acid aspartic acid. The stability scores were rounded to four decimal places since the extra decimal places did not make a difference in modeling the data. The minimum stability score was recorded at -1.97 and the maximum stability score was recorded at 3.40. A higher stability score indicated greater resistance to protease cleavage and hence greater stability. If the stability score was less than 0, then it indicated that the protein sequence was less resistant to protease cleavage compared to its unfolded state which is not desirable if you want a stable protein.

Table 1: Training Set

Sequence	**Stability Score**
DQSVRKLVRKLPDEGLDREKVKTYLDKLGVDREELQKFSDAIGLESSGGS	-0.209999993
GSSDIEITVEGKEQADKVIEEMKRRNLEVHVEEHNGQYIDKASLESSGGS	-0.949999988
⋮	⋮

Originally, the goal was to predict the exact stability score, but after working with different models, it became increasingly apparent that it would not be possible to predict the stability score as the highest accuracy achieved was 0.0086. Instead, the goal of the project was to turn this regression task into a classification task by classifying each stability score into a class which is further explained in Section 2.3.

The protein sequences were encoded to a numerical representation of the amino acid which is further explained in Section 2.2. The sequences were also padded so that each sequence had a standard length of 50, because the maximum sequence from the data was 50 amino acids long. Table 2 contains the encoded data that was used to train each model.

Table 2: Training Set

Sequence	**Stability Score**
2, 13, 15, 17, 14, 8, 9, 17, 14, 8, 9, 12, 2, 3, 5, 9, 2, 14, 3, 8, 17, 8, 16, . . . , 15	4
5, 15, 15, 2, 7, 3, 7, 16, 17, 3, 5, 8, 3, 13, 0, 2, 8, 17, 7, 3, 3, 10, 8, 14, . . . , 15	5
⋮	⋮

2.1 Data Distribution

To understand how the data was distributed, a histogram of the frequency of amino acids and stability scores was obtained in Figure 1. The distribution showed there was a significantly high frequency of glutamic acid (E) and lysine (K) amino acids. Glutamic acid is negatively charged and lysine is positively charged making these amino acids incredibly important for protein stability because they are pH sensitive and experience many charge interactions. The high frequency of these amino acids makes it almost impossible to predict the protein stability, so this was another motivation to approach predicting the stability based only on the order of the sequence. The distribution of the stability scores indicated that there was a roughly Gaussian distribution of data which is desirable in building a model. The mean stability score was greater than 0 which indicated that on average the proteins were more stable than in their unfolded state when exposed to the protease EC50.

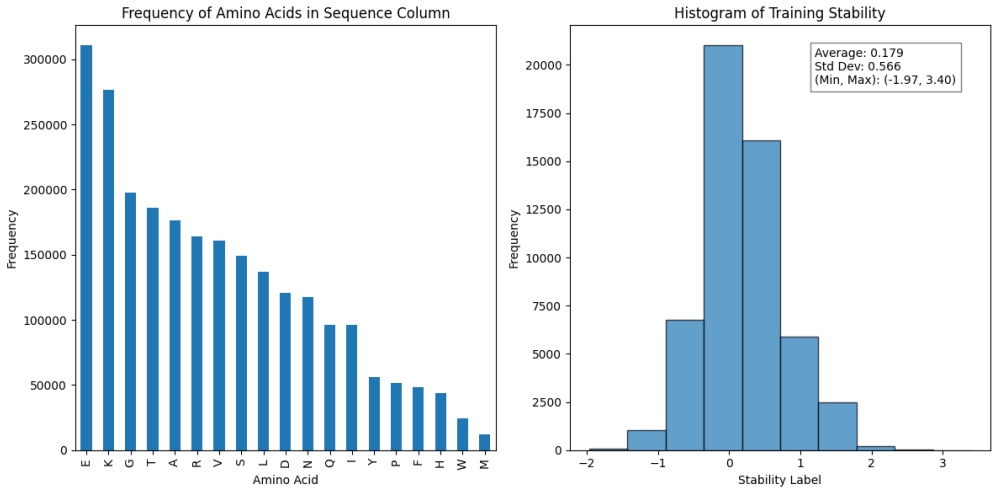

Figure 1: Distribution of Amino Acids and Stability Scores

2.2 Encoding and Padding

The encoding for the amino acid sequencing is below.

Table 3: Amino Acid Encodings

A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

2.3 Classifying Stability Scores

Below are the rules that were used to classify the stability score x. Note, that it was desired to have a protein that had moderate stability or higher since researchers often use more stable proteins.

- Class 0: $x \leq -1$: Highly Unstable
- Class 1: $-1 < x \leq -0.5$: Increasingly Unstable
- Class 2: $-0.5 < x \leq 0$: Moderately Unstable
- Class 3: $0 < x \leq 0.5$: Moderately Stable
- Class 4: $0.5 < x \leq 1$: Increasingly Stable

- Class 5: $1 < x \leq 1.5$: Highly Stable
- Class 6: $x > 1.5$: Extremely Stable

3 Analyses

3.1 Sparse Categorical Cross-Entropy Loss Function

The loss function $J(\theta)$ also known as the cost function or the objective function was minimized in this report. Since determining the stability class is a multi-class classification task, sparse categorical cross-entropy was used. It also made sense to use this loss function since the classes were represented numerically and not as one-hot encoded vectors. Another benefit of this loss function was it reduced memory usage compared to categorical cross-entropy. The loss function is shown below. Note: N is the number of samples which was 53,614 for training, C is the number of classes which was 7, y_{ij} is 0 or 1 depending on if class j was the correct classification for sample i, and $\hat{y}_{ij}$ was the predicted probability of that sample.

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

3.2 Regularization

Since there were significantly more parameters than there were data points, it was very easy to overfit. Overfitting occurs when the model learns the noise and fluctuations leading to less generalizability of the model. To prevent overfitting, regularization is often used in machine learning tasks to improve the generalization performance of a model. Since regularization methods introduce additional constraints or penalties on the model's parameters, it is advantageous to find a balance between fitting the training data and avoiding overfitting. By finding the right balance, regularization can (1) reduce the variance of the model, (2) improve its ability to generalize, and (3) control the model's complexity. Four common techniques of regularization used in this report were ℓ_1 regularization, early stopping, batch normalization, and dropout.

3.2.1 ℓ_1 Regularization

ℓ_1 regularization, also known as Lasso (Least Absolute Shrinkage and Selection Operator), is a penalty term that is added to the loss function and is proportional to the absolute values of the model's coefficients. This penalty encourages sparsity in the model by driving some coefficients to zero effectively performing feature selection. This can lead to faster computational times but it can also significantly diminish the model's performance. Strong ℓ_1 regularization is used to achieve sparsity. No matter what loss function $J(\theta)$ is used, there will be an extra term that will penalize large values and drive some parameters θ to zero based on the hyperparameter λ. By penalizing large parameter values θ, the model is discouraged from fitting the noise in the training data which encourages it to generalize better. Ridge (ℓ_2) regularization is similar to ℓ_1 regularization but it squares the magnitudes of the model's coefficient encouraging smaller coefficients.

$$J(\theta) + \boldsymbol{\lambda}\sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} |\boldsymbol{\theta_j}|$$

3.2.2 Early Stopping

Early stopping is a simple concept where training is halted when no progress has been made on the validation set for a number of epochs (patience argument). This saves time as it prevents the model from continuing to be trained. Figure 2 displays the implementation of early sopping for each model. The patience was set to 10 epochs and the weights were restored to the best weights once halted. The validation loss was monitored each epoch.

```
# Define early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
```

Figure 2: Early Stopping Code

3.2.3 Batch Normalization

While there are methods for reducing the vanishing/exploding gradients problem, it does not guarantee that it will not come back during training. Batch Normalization is a technique that considerably improves the gradients by adding an operation of scaling and shifting the data before or after the activation function. The data is zero-centered and normalized and then scaled and shifted.

The first part of the algorithm is to estimate the input's mean and standard deviation to zero-center and normalize. This is performed for each mini-batch. The second part of the algorithm is to standardize the input and rescale and offset them. The final mean and standard deviation are estimated using an exponential moving average of the layer's input means and standard deviations. There are four important parameters that are learned for each batch-normalized layer: $\boldsymbol{\gamma}$ (output scale vector), $\boldsymbol{\beta}$ (output offset vector), and $\boldsymbol{\mu}$ (final input mean vector), and $\boldsymbol{\sigma}$ (final input standard deviation vector). $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are estimated during training but they are only used to replace the batch input means and standard deviation after training. With the scaling and shifting of the input, the vanishing gradients problem is strongly reduced. The biggest problem with batch normalization is the increased computational complexity but it converges faster.

Algorithm 1: Batch Normalization Algorithm

Input: Mini-batch of activations $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$

// Compute mini-batch mean
$\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$;
// Compute mini-batch variance
$\sigma^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$;
// Normalize activations
$\hat{x}^{(i)} \leftarrow \frac{x^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}}$;
// Scale and shift normalized activations
$z^{(i)} \leftarrow \gamma\hat{x}^{(i)} + \beta$;
// Output normalized mini-batch
return $\{z^{(1)}, z^{(2)}, ..., z^{(m)}\}$;

3.2.4 Dropout

Dropout is a regularization technique that prevents overfitting by randomly dropping out a fraction of neurons during training. This forces the network to learn redundant representations and learn more robust features preventing it from relying too heavily on specific neurons. The result is a better generalization performance on unseen data. The algorithm works by temporarily ignoring a probability p neurons excluding output neurons at every training step which means every disabled neuron has a chance of being reactivated again in the next training step.

3.3 Adaptive Momentum Estimation (Adam) Optimizer

Choosing the right optimizer for machine learning algorithms is imperative for adjusting the parameters of the model to minimize the loss function. Each optimizer learns from the training data differently which results in different parameter updates which can lead to different convergence and performance. The optimizer used in this report was adaptive momentum estimation (Adam) which is a combination of Root Mean Square Propagation (RMSProp) and Momentum optimization. Similar to RMSProp, Adam keeps track of

an exponentially decaying average of past squared gradients, and similar to Momentum optimization, Adam keeps track of an exponentially decaying average of past gradients.

Algorithm 2: Adam Algorithm

Input: Learning rate η, β_1, β_2, Initial parameter θ
Output: Updated parameter θ
$m \leftarrow 0$
$s \leftarrow 0$
$t \leftarrow 0$
while *not converged* **do**
  $t \leftarrow t + 1$
  $g \leftarrow \nabla_\theta J(\theta)$
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$
  $s \leftarrow \beta_2 s + (1 - \beta_2) g \otimes g$
  $\hat{m} \leftarrow \frac{m}{1-\beta_1^t}$
  $\hat{s} \leftarrow \frac{s}{1-\beta_2^t}$
  $\theta \leftarrow \theta - \eta \frac{\hat{m}}{\sqrt{\hat{s}}+\epsilon}$

The Adam algorithm is displayed in Algorithm 2. The beginning steps of the algorithm are to initialize the learning rate η, two exponential decay rates β_1 and β_2, initial parameter values θ, first and second moment (m and s respectively), and the iteration count t. In step 1, Adam is computing an exponentially decaying average. The iteration count is incremented by 1 when it enters the while loop and the gradient g of the loss function $J(\theta)$ is calculated. Once the gradient is calculated, the first and second moments are estimated. The first moment is calculated by adding the past estimated first moment m weighted by β_1 with the current gradient weighted by $1 - \beta_1$ which smooths fluctuations in the gradient. The second moment is calculated by adding the previous estimated second moment s weighted by β_2 with the current squared gradient weighted by $1 - \beta_2$. Since the first and second moments were initialized to zero in the beginning which is biased, an adjustment of the first and second moment estimates is performed by dividing by the $1 - \beta_1$ and $1 - \beta_2$ term respectively. The parameters θ are then updated by taking into account the historical gradients and variances to reduce the objective function. The smoothing term ϵ is initialized to a tiny number to help smooth the data. These steps are repeated until convergence.

3.4 Activation Functions

There were two activation functions that were used in each model: rectified linear unit (ReLU) and softmax. ReLU was preferred since it is a commonly used activation function for all the machine-learning algorithms used in this paper. ReLU is also a very simple and computationally efficient model as it replaces negative values with zero as seen below. One of the main benefits of ReLU is that it helps mitigate the vanishing gradient problem and it is a non-linear non-saturating activation function making it ideal.

$$\phi(x) = \max(0, x)$$

The softmax function was essential in the output layer for each model as it produced predicted probabilities for each class. This is done by converting a vector of numbers into a vector of probabilities that add up to 1 as shown below. Note: $\boldsymbol{x}$ is a vector of scores that were of length 7 for this report since there were 7 unique classes.

$$\hat{p} = \phi(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

3.5 Embedding Layer

Each model except the DNN model began by becoming embedded by converting the integer-encoded amino acid sequences into dense vectors of size 32. The Embedding layer learned to represent each unique

amino acid by a dense vector in a continuous vector space which helped the model understand relationships and patterns between the amino acids more effectively. A larger output layer was not chosen even though it would contain more complex representations because it would result in more overfitting which was a major problem in this report. A smaller output layer was not chosen because it would not contain enough complex representations. Note, the length of the amino acid dictionary was 19 since there were 19 unique amino acids.

3.6 Deep Neural Network (DNN)

A deep neural network (DNN) is a type of artificial neural network (ANN) with three types of layers: input, hidden, and output. The hidden layers of a DNN are capable of learning complex relationships from the input. Since it is well understood that amino acid sequences contain high-dimensional relationships and patterns, DNNs were the first models chosen to model these intricate patterns and relationships. After training a DNN with 4 hidden layers, it became clear that the model suffered from extreme overfitting as can be seen in the Jupyter notebook on GitHub, so the model in Figure 3 was trained using batch normalization and 50% dropout to try to mitigate the overfitting. By using batch normalization, dropout, and four hidden layers, the goal was to find the hidden relationships between the order of peptide sequences and the stability class. After training, the accuracy of the second DNN model was diminished compared to the first DNN model which contained no regularization. The second model which is displayed in Figure 3 was only able to predict 44.3% of the training data in 14 minutes with 26 epochs. It was clear that even with the regularization techniques the model was still severely overfitted as it was only able to predict 15.5% of the testing data correctly.

```
dnn2 = Sequential([
    Flatten(input_shape=(50,)),
    Dense(1024, activation='relu'),
    BatchNormalization(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(124, activation='relu'),
    BatchNormalization(),
    Dense(64, activation='relu'),
    Dense(7, activation='softmax')
])

# Compile the model
dnn2.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Fit the model with early stopping
history2 = dnn2.fit(encoded_df_train['seq'].tolist(), encoded_df_train['label'].tolist(),
                    epochs=100, batch_size=32,
                    validation_data=(encoded_df_valid['seq'].tolist(),
                    encoded_df_valid['label'].tolist()), callbacks=[early_stopping])
```

Figure 3: Deep Neural Network with Batch Normalization and Dropout

3.7 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) designed to capture long-term dependencies in sequential data while being resistant to the vanishing gradient problem. LSTM networks model temporal relationships, handle variable-length sequences, and retain information over long time lags which make them incredibly useful for learning sequential data with particularly complex patterns. LSTM networks are well known for learning sequential data, especially for natural language processing (NLP). LSTM networks consist of memory cells and gates that selectively remember or forget information over time steps using three types of gates: input gate, output gate, and forget gate. The number of units was chosen to be 128 for the LSTM since it was believed that the hidden relationships of the protein sequence and stability class could be modeled in 128 units. Since many models were trained on the data and each experienced overfitting, there was 50% dropout in the next layer which was 32 neurons. The decision to

add a hidden layer between the LSTM and output layer was made to ensure that the hidden representations of the data would be learned well. This turned out to be successful as the model had **the most trustworthy training accuracy** with 65.3%, but it took the longest to train with 53 minutes and 19 epochs. It also experienced overfitting since the testing accuracy was 51.1%.

```
# len(amino_dict) = vocab_size
# input_length = max_length
lstm2 = Sequential([
    Embedding(input_dim=len(amino_dict), output_dim=32, input_length=50),
    LSTM(128),
    Dropout(0.5),
    Dense(32, activation='relu'),
    Dense(7, activation='softmax')
])

# Compile the model
lstm2.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy']
                                                                                )

# Fit the model with early stopping
history9 = lstm2.fit(encoded_df_train['seq'].tolist(), encoded_df_train['label'].tolist(),
                     epochs=100, batch_size=32,
                     validation_data=(encoded_df_valid['seq'].tolist(),
                     encoded_df_valid['label'].tolist()), callbacks=[early_stopping])
```

Figure 4: Long Short-Term Memory with Dropout

3.8 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) networks are very similar to LSTM networks since they are a variant of LSTM networks, but they excel in reduced computational complexity while retaining similar performance. GRUs are designed to have simplified architectures that make them more interpretable than an LSTM network. GRUs have the same benefits as LSTM because they can learn long-range relationships in sequential data, but they have fewer parameters and are easier to train. GRUs have fewer gates and do not have separate memory cells. GRUs are often considered if memory efficiency and computational speed are a concern when training data which was important since training was performed on a laptop. The GRU performed similarly to the LSTM model as it was able to predict 66.3% of the training data in 37 minutes and 18 epochs. The GRU model converged faster than the LSTM model in terms of time elapsed and number of epochs, but it severely overfitted the data with a 43.7% testing accuracy. Interestingly, there was another GRU model trained after the GRU model displayed in Figure 5 with regularization, but the regularization severely diminished the training/validation accuracy making it unusable as a model. For more information about the second GRU model, please refer to the Jupyter notebook on GitHub.

```
# len(amino_dict) = vocab_size
# input_length = max_length
gru1 = Sequential([
    Embedding(input_dim=len(amino_dict), output_dim=32, input_length=50),
    GRU(128),
    Dense(32, activation='relu'),
    Dense(7, activation='softmax')
])

# Compile the model
gru1.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Fit the model with early stopping
history4 = gru1.fit(encoded_df_train['seq'].tolist(), encoded_df_train['label'].tolist(),
                     epochs=100, batch_size=32,
                     validation_data=(encoded_df_valid['seq'].tolist(),
                     encoded_df_valid['label'].tolist()), callbacks=[early_stopping])
```

Figure 5: Simple Gated Recurrent Unit

3.9 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are another type of ANN that gained inspiration from how our eyes work. The goal of CNNs is to determine high-level features through multiple levels of convolutional layers where a kernel similar to a receptive field scans each data representation learning complex patterns. CNNs are commonly associated with image recognition and classification tasks, but they have been increasingly used for NLP tasks due to their ability to understand long-range complex patterns. CNNs excel at learning hierarchical representations by capturing spatial patterns. CNNs are also translational invariant meaning they are robust to translations, rotations, and scale changes. The CNN model consisted of three one-dimensional convolutional layers with two max pooling. The kernel size for each was 2×2. As stated earlier, the machine learning models severely suffered from overfitting so ℓ_1 regularization was used for each convolutional layer. A dense layer was added before the output layer to make sure all hidden representations were learned from the data. A 43.0% training accuracy was achieved in 19 minutes with 72 epochs, but severe overfitting was evident when evaluating the model on the testing set as the testing accuracy was 10.7%. Another CNN model that used more regularization and deeper layers was trained in the Jupyter notebook on GitHub, but its training accuracy was significantly lowered similar to the second GRU model indicating that the CNN overfitted whether it had regularization or not. The simpler CNN model was chosen in this report because it was faster to train and had an overall better training and testing accuracy compared to the second model.

```
cnn1 = Sequential([
    Embedding(input_dim=len(amino_dict), output_dim=32, input_length=50),
    Conv1D(128, kernel_size=2, activation='relu', kernel_regularizer=tf.keras.regularizers.
                                                  l1(0.01)),
    MaxPooling1D(),
    Conv1D(64, kernel_size=2, activation='relu', kernel_regularizer=tf.keras.regularizers.l1
                                                  (0.01)),
    MaxPooling1D(),
    Conv1D(32, kernel_size=2, activation='relu', kernel_regularizer=tf.keras.regularizers.l1
                                                  (0.01)),
    Flatten(),
    Dense(16, activation='relu'),
    Dense(7, activation='softmax')
])

# Compile the model
cnn1.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Fit the model with early stopping
history6 = cnn1.fit(encoded_df_train['seq'].tolist(), encoded_df_train['label'].tolist(),
                    epochs=100, batch_size=32,
                    validation_data=(encoded_df_valid['seq'].tolist(),
                    encoded_df_valid['label'].tolist()), callbacks=[early_stopping])
```

Figure 6: Deep Convolutional Neural Network

4 Plots and Tables

In Figure 7, the loss curve of the DNN showed that on average when training, the validation loss differed from the training loss showing there was overfitting during the training. Ideally, it is desired to have the training and validation loss to have very identical losses. The accuracy curve in Figure 7 showed that the accuracy increased as the epochs increased almost linearly but the accuracy of the validation was more unpredictable and the accuracy differed significantly starting around 10 epochs indicating severe overfitting of the training started around there.

In Figure 8, the loss and accuracy curve indicated that the LSTM did a good job of not overfitting until around 10 epochs. The training loss and validation loss were very similar until around 10 epochs at which the validation loss looked to grow quadratically indicating overfitting occurred after 10 epochs. This was also confirmed in the accuracy curve as the validation accuracy started to decrease around 10 epochs while the training accuracy continued to increase creating a very large difference in final accuracies.

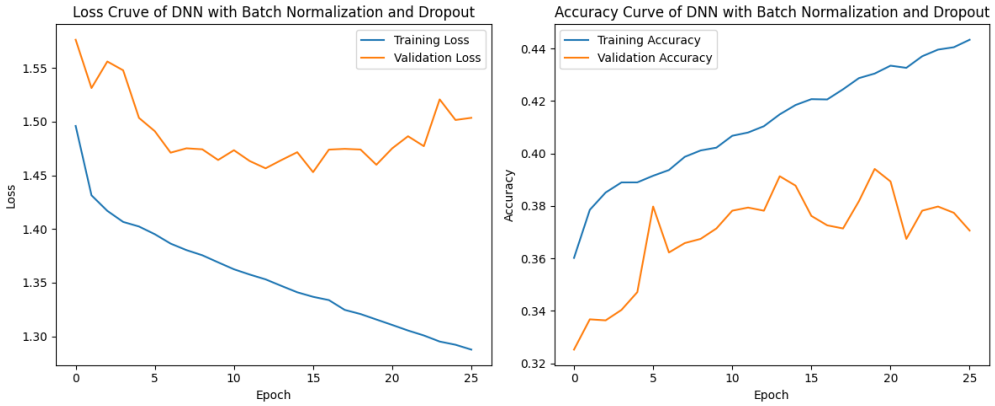

Figure 7: Loss and Accuracy Curve for DNN

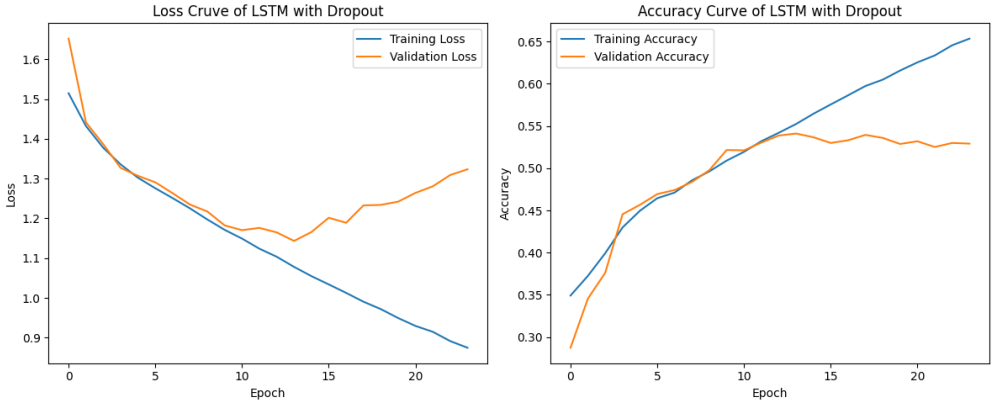

Figure 8: Loss and Accuracy Curve of LSTM

In Figure 9, similar overfitting as in the DNN model was seen in the GRU model, but severe overfitting started after 5 epochs for the GRU. The loss increased quadratically after 5 epochs and the difference between validation and training accuracy increased significantly after 5 epochs indicating severe overfitting.

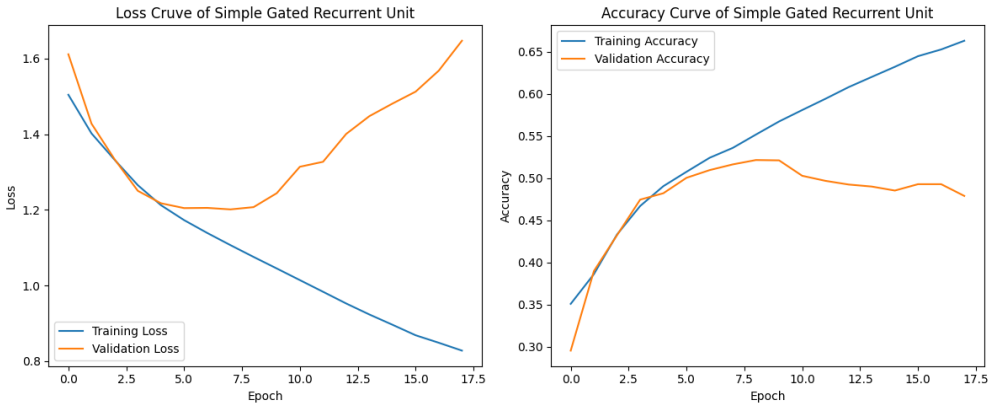

Figure 9: Loss and Accuracy Curve for GRU

In Figure 10, the training and validation loss for the CNN model seemed to decrease exponentially in 72 epochs. The validation loss and training loss did not diverge at any point and they were relatively close to each other for every epoch. The accuracy of the training and validation set had a bigger difference between the curves but they still followed the same trend which is ideal. Interestingly, it was assumed that there was not much overfitting in this model but the training accuracy was 0.430 while the testing accuracy was 0.107 indicating severe overfitting of the training data.

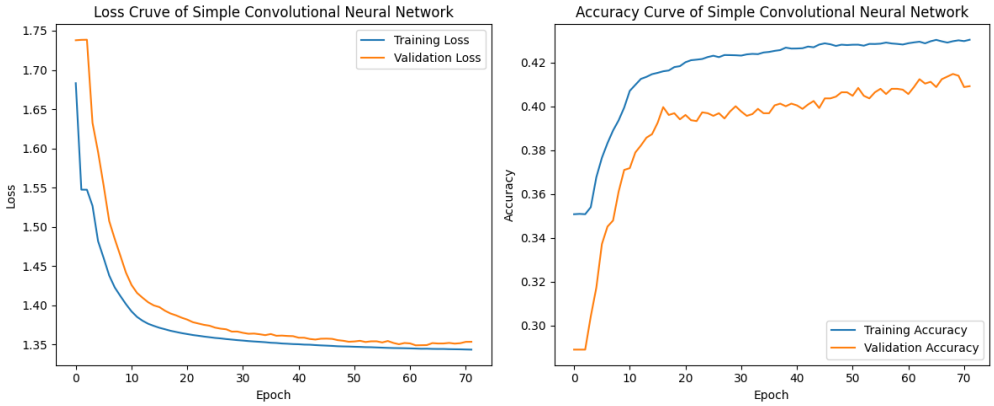

Figure 10: Loss and Accuracy Curve for CNN

In Table 4, it was clear that the LSTM with dropout model was **the best performing**. The GRU model achieved **the best training accuracy**, but it experienced severe overfitting. The LSTM model was able to predict slightly above random guessing which is not ideal, but this was by far **the best testing accuracy**. The DNN and CNN were the fastest to train but they experienced almost the same amount of severe overfitting on the training data as there was around a 0.3 difference in accuracy between the training and testing set for both.

5 Conclusions

In conclusion, four machine learning models were trained with the goal of learning the complex hidden relationships between protein sequences and their respective stability classes. Unfortunately, each model

Table 4: Model Comparison

Model	Training Accuracy	Testing Accuracy	Time Elapsed
DNN with BN and Dropout	0.443	0.155	14 mins
LSTM with Dropout	0.653	0.511	53 mins
GRU	**0.663**	0.437	37 mins
CNN	0.430	0.107	19 mins

suffered from severe overfitting for many epochs which was interesting since early stopping was implemented for each model. If these models were to be studied again, it would be more beneficial to decrease the patience to around 5 since each model experienced severe overfitting. The most trustworthy model was the LSTM with dropout model. This was not unexpected since LSTM models are great at learning sequential data, but the training time of 53 minutes make this model not scalable to train more proteins. From the analysis of each model, it became very clear that even with a dataset of 68,977 proteins there was not enough proteins in the dataset to accurately model the hidden representations of the protein sequences. If this experiment was performed on more data, then there would be an emphasis on regularizing the GRU enough so that it does not overfit or hinder the training accuracy significantly. A GRU would be better for a much larger model because it is also great at learning sequential data and it converged much faster than the LSTM model.

6 Code/Appendix

The major libraries that were used in this paper was pandas and numpy for the dataframe, matplotlib for visualizing data, keras.preprocessing for data preprocessing, and tensorflow and keras for implementing each model. More models were trained in the Jupyter notebook on GitHub. For more information on the code, please check out the link https://github.com/BrandonCohen1/Machine-and-Deep-Learning/blob/main/Protein_Stability_Project.ipynb.

References

[1] R. Rao, N. Bhattacharya, N. Thomas, Y. Duan, X. Chen, J. Canny, P. Abbeel, and Y. S. Song. *Evaluating protein transfer learning with TAPE.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[2] G. J. Rocklin, T. M. Chidyausiku, I. Goreshnik, A. Ford, S. Houliston, A. Lemak, L. Carter, R. Ravichandran, V. K. Mulligan, A. Chevalier, C. H. Arrowsmith, and D. Baker. Global analysis of protein folding using massively parallel design, synthesis, and testing. *Science*, 357(6347):168–175, 2017.