Protein Secondary Structure Classification

Problem Statement:

Proteins are the workhorses that make life possible. They're responsible for nearly all the physiological processes that happen in and around us. In biology (and in other disciplines), form dictates function. Therefore, to understand protein structure, is to understand its function. The hope is to get such an understanding of proteins, that we can begin to understand diseases better, prevent and treat illnesses, develop biotechnology, solve agriculture/ecological problems, find novel applications for proteins, etc. With so much possibility and value, this has been a long-standing center of research and serves as the basis for this final project. Typically, regression and classification based on tabular data use classical ML methods. Since this data is complex spatial data that does not adhere to the classic tabular format, deep learning is a great approach for solving this problem. Using 3D atomic spatial data from the CATH database, the goal is to develop a deep learning model that can accurately predict secondary class structures in proteins.

CATH Input Data:

Specifically, the data used in this project is sourced from a Numpy NPZ file hosted in this public repository. In its "raw" form, the spatial data is an array of shape (16962, 1202, 3). Where 16962 is the number of proteins in the dataset, 1202 is the max number of positional data (aka carbon-alpha positions of every amino acid in the protein) per protein, and 3 is the 3D coordinates of the atomic positions. Each row (protein record) is an array of shape (1202, 3). A protein with only a few amino acids will be a sparse array of 3D coordinates, while the largest protein with 1202 amino acids, will have all columns filled with the atomic positions.

In addition, this CATH dataset gives equal classes, so class imbalance is not a concern. However, in terms of preprocessing, different considerations had to be made for both neural networks (NNs) and convolutional neural networks (CNNs).

For the NN, the original (16962, 1202, 3) array had to be flattened to a single array of size (16962, 3606). The number of rows is preserved but the original added dimensional data (1202 x 3) is flattened to a columnar format of length 3606.

Without going into too much technical detail, preparing the data for the CNNs required different processing. Instead of flattening the original array, I added a spatial dimension, a channel dimension, and converted the final array to a tensor. This yielded a tensor of shape (16962, 1, 3, 1). In general, adding the spatial and channel dimensions was done to help the model increase its ability to learn more complex features.

For all instances of the data, the data was split into training, validating, and a final testing set. An example of these inputs can be found in the eda_traning.ipynb Jupyter notebook.

Data Labels:

The class labels for the specific CATH data used in this project have three possible choices. Again, these structures refer to a protein's secondary structure. The labels are "Mainly Alpha", "Mainly Beta", and "Alpha-Beta". As the label data is simply an array of arrays of size 3, not much processing (other than shaping) had to be done. The label data contained binary data at the index of the array that corresponds to the class label. For example [1, 0, 0] is equivalent to "Mainly Alpha", [0, 1, 0] is equivalent to "Mainly Beta", and [0, 0,1] is equivalent to "Alpha-Beta".

Building the NNs:

To begin, I started off with building a base-line neural network. This was the first experiment and consisted of one layer, made up of 4 neurons. I used a "relu" activation function and the "softmax" activation function for the output layer (with 3 neurons). These activation functions and outputs are standard throughout all the experiments. In addition, the model compile method for all future models used the "adam" optimizer, "categorical_crossentropy" for loss, and "accuracy" for the scoring metric. This first model

trained very fast (~6s) and had promising training curves (Fig. 1). However, it performed very poorly with a test accuracy of ~37% (Fig. 5). As a baseline, this was expected.

With these results, I decided to create a more complex NN. The second NN had the same overall architecture as the first NN but with 2 layers, both containing 64 neurons. The training curves for this model showed a growing divergence of train/validation accuracy and loss over the epochs (Fig. 2). This was starting to suggest overfitting. The model did increase its test accuracy to about ~62% (Fig. 5). At this point and based on my research/class lecture, it felt that the NNs were probably at their limit with spatial data, so I decided to pursue CNNs.

Building the CNNs:

The first CNN that I built had two 3D convolutional layers with padding, two pooling layers with padding, and two dense layers before the final output. The first convolutional layer had a filter of 32, kernel size of (3,1,3), and an activation of "relu". The second convolutional layer had filter of 64, kernel size of (3,1,3), and an activation of "relu" (these parameters are the same with the final CNN). The pooling layers are meant to prevent overfitting by essentially reducing the spatial data to the most important features. To keep things simple, these layers had a pool size of (2,1,1) and "same" padding (which function to maintain and preserve spatial information during the convolutional steps). The final two dense layers were made up of 128 and 64 neurons respectively. Similar to the second experiment (2^{nd} NN), the training curves showed diverging accuracy and loss over the epochs, suggesting overfitting (Fig. 3). However, the overall test accuracy of the model did improve to ~85% but with a training time of ~22 minutes (Fig. 5).

I had two goals in mind for the last CNN. I wanted to reduce complexity and reduce the computational overhead. With these two ideas in mind, I decided to simplify the original CNN and implement early stopping. To simplify the CNN, I maintained the main CNN architecture/pooling parameters, removed the final dense layers altogether, and implemented early stopping.

With “early stopping” implemented, I used “val_loss” as the monitor and a patience of 2. These results yielded great results. The training accuracy and loss curves were looking great (both improving and mirroring one another), the training time was cut by nearly 250% to ~6 minutes, and the test accuracy improved to ~86% (up to ~89% on subsequent runs). Refer to Fig. 4 and Fig. 5.

Final Outputs:

Each model outputs an array of arrays of size 3. Like the input arrays of the class labels, each index of the array had the probability of the corresponding class being correct. The index position with the highest value was the predicted class label. For example, an output of [0.1, 0.8, 0.1] would predict the second class, “Mainly Beta”. From the probabilities of each class-index position, the final output can easily be processed to the final class labels of 0, 1, 2 (which map to “Mainly Alpha”, “Mainly Beta”, and “Alpha-Beta”). An example of these outputs can be found in the eda_training.ipynb Jupyter notebook.

Using the mlflow API:

To track the work, modularize the code, version it with source control, containerize it with Docker, and make it re-producible, I created a public code repository that can be forked locally and ran with all the necessary utilities, modules, models, and dependencies for logging/experiment tracking this entire final project. Reference the mlflow documentation for more info.

The repository is structured in such a way that each aspect of the project is modular and abstracted into logical components. The end result is that within the main.py Python module, the utility functions are imported, the different experimental models are imported, and then the final script is accessed which runs each deep learning model sequentially

(training, testing, and logging of experiment artifacts) and writes the results to a local mlflow server and into local run artifacts.

Reference this experimental_models.py module for the specifics of the mlflow API usage, the architectures/parameters of each model, and seeing how the experiments are logged.

Conclusion:

In this final project, I was able to use and understand spatial protein data, iterate and build NNs/CNNs with different architectures, create a software project with engineering best-practices, and a build up a deep learning project that executes and modularizes the training, testing, and logging of the different experimental results with the mlflow API. In addition to this, I was able to successfully get a fairly accurate (86-89% test accuracy) prediction of secondary protein structures by feeding 3D atomic spatial data into a CNN model that took 1/3 the time of the original CNN to train and compute. The next steps to further enhance this project are to add new models that have different CNN architectural choices as well as possibly add new feature transformations to the 3D spatial data.

Figures

Fig. 1 – Single Layer NN Training Curves

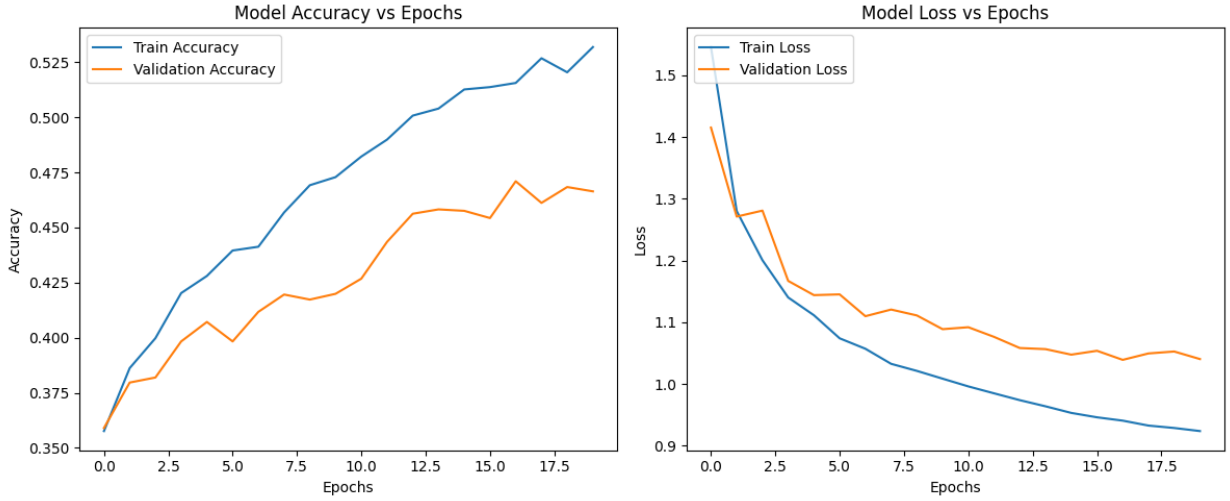

Fig. 2 – Double Layer NN Training Curves

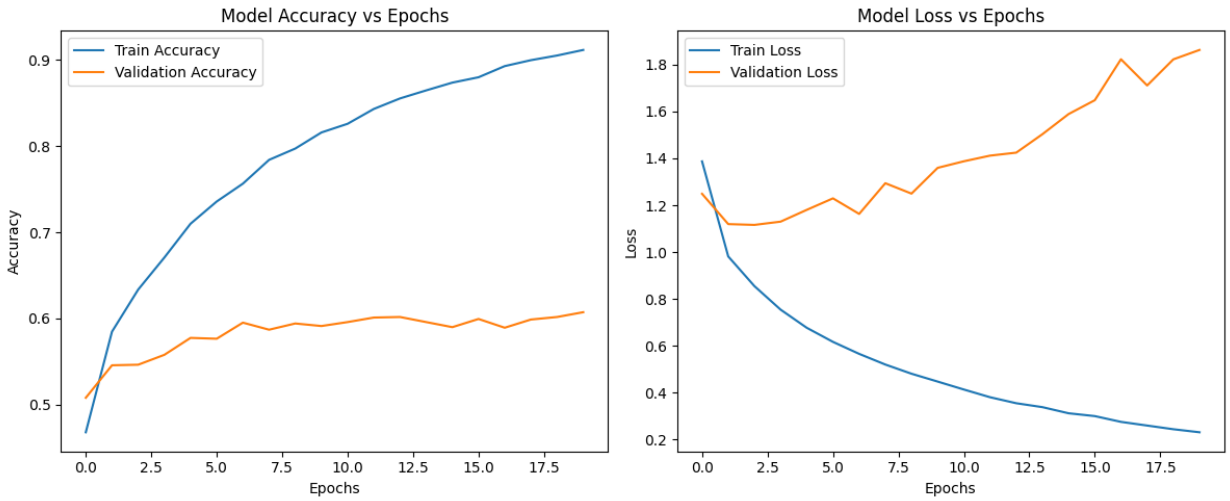

Fig. 3 – CNN Training Curves

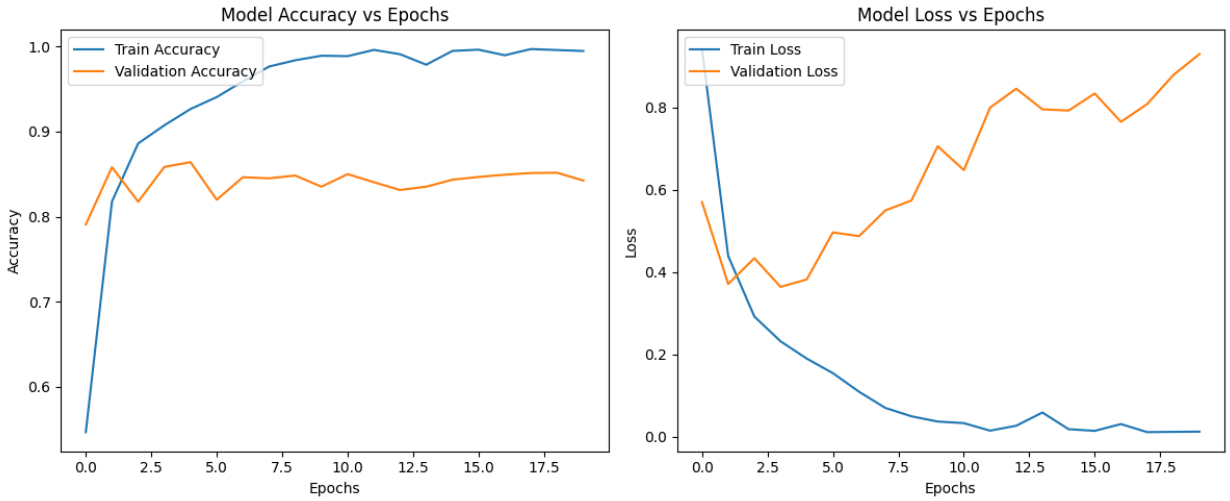

Fig. 4 – Simplified CNN w/ Early Stopping Training Curves

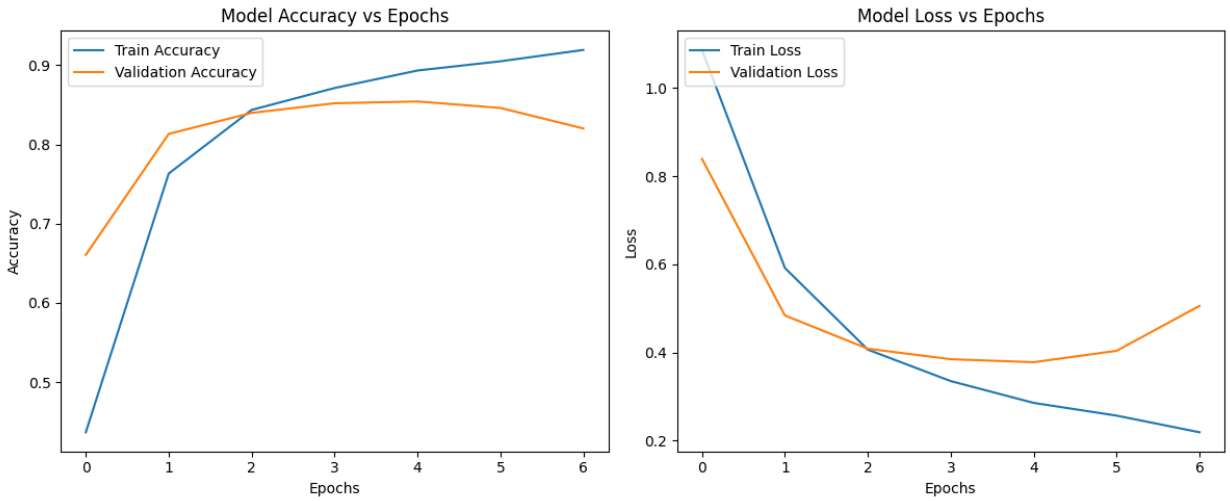

Fig. 5 – Model Results

Run details

Run ID:	77eb0bf931d84c638825091555cc6991	b8444e199a5a408ebdbeed48798dcb0f	05d2f45ce16c475e93f9cfab6f813844	b054d14d51b84041ac491d1adf22b883
Run Name:	Simplified CNN w/ Early Stopping	CNN	Basic NN - Two Layers	Basic NN - One Layer
Start Time:	2025-02-19 20:45:33	2025-02-19 20:24:02	2025-02-19 20:23:52	2025-02-19 20:23:46
End Time:	2025-02-19 20:53:09	2025-02-19 20:45:33	2025-02-19 20:24:02	2025-02-19 20:23:52
Duration:	7.6min	21.5min	10.1s	6.0s

Parameters

Metrics

Show diff only

test_accuracy	0.856	0.852	0.615	0.37
test_loss	0.386	0.832	1.679	1.185