Andrew Florial
Cameron Farzaneh

## Background

Understanding biological structures proves to be a difficult problem. We particularly are interested in amino acid prediction. For our particular task, we implement a general framework that applies “EdgeConv” on a Dynamic Graph CNN neural network model for structure-based classification. In our specific implementation, we configured it for amino acid structure-based classification. These amino acids and atoms are obtained from the Protein Data Bank (PDB).

By improving our ability to classify amino acids we enable progress in that of understanding protein structure and function. With traditional methods, this is an almost intractable problem that requires many years of study and expensive lab equipment. We explore neural networks ability to perform amino acid type prediction.

Point clouds are geometric point representations of elements in 3D space. This provides a flexible and scalable representation for 3D structures. We propose an implementation configured for amino acids. The novel method used, EdgeConv, works well for CNN-based high-level tasks on point clouds including classification.

Point cloud representation of the 3D structure is interesting for this problem domain for several reasons. In a 3D plane, there are many ways an amino acid structure can be represented. Thus, there are many configurations for an amino acid structure. For our classification task, it is desirable to maintain learning that is orientation invariant. DGCNN with EdgeConv provides this by incorporating local neighborhood information; it can be stacked to find global structure properties.

## Dataset and Format

The data is formatted with each line representing the information in a box, e.g., a sample. The first column is the true label (Y), e.g., 20 type of amino acids represented by three letter notation. Then the rest part is separated by tab as atoms. For each atom, we have the information including atom-label, atom-name, residue-name where this atom comes from, a notation tell if this atom from target chain or environment chain (TC/NTC), a notation tell if this atom from the target residue (TR/NTR), and the coordinate X, Y, Z. All these information are separated by commas. For our project, we simply needed the coordinates (X, Y, Z). The samples of the data consist of an all-atom model. This means that all the atoms near the target amino acid residue are contained in the box.

## Network architecture

The architecture that we used was taken from the paper Dynamic Graph CNN for Learning on Point Clouds. In this paper, a convolutional neural network is proposed called DGCNN that utilizes an operation called EdgeConv that is suitable for CNN-based high-level tasks on point clouds including classification and segmentation.

Andrew Florial
Cameron Farzaneh

EdgeConv captures local geometric structure while maintaining permutation invariant. Instead of generating points features directly from their embeddings, EdgeConv generates edge features that describe the relationships between a point and its neighbors. The model architectures proposed in DGCNN has two branches, one used for classification (top branch) and segmentation (bottom branch). For the use of our project, the segmentation branch is not needed and removed. The classification model, which we use, takes as input n points represented as X, Y, Z values, calculates an edge feature set of size k for each point at an EdgeConv layer, and aggregates features within each set to compute EdgeConv responses for corresponding points. The output features of the last EdgeConv layer are aggregated globally to form a 1D global descriptor, which is used to generate classification scores for c (in our case 20) classes.

**Inputting our data**

Inputting our data into the DGCNN model was a nontrivial task. This is because the original project used as a reference was coded to be trained on the ModelNet40 dataset. Although this dataset also contained X, Y, Z values, the way in which the data was handled and preprocessed is very different from how our dataset was organized. Because of this, it was necessary to implement a method of extracting our data and process it to be in a similar format that ModelNet40 was.

Originally, the code simply downloads the ModelNet40 dataset from Standford's website. This data was already preprocessed in that the X, Y, Z values were neatly organized, and the labels were placed in an HDF5 file. The same code segment in which the ModelNet40 data was downloaded and loaded is where the primary code of our project was placed. Because our data was not preprocessed already, we had to load the data in an efficient manner and generate the labels on the fly. The way we have gone about doing this was constructing a Python generator. Python generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop. Every time the function is called, it *yields* a value instead of returning. The state is remembered and on the next function call, the code loads its previous state and *yields* the next values. We used a generator in order to load the files in batches. We needed to do this because our dataset is very large, and if we were to load all the data at once, we would run into memory limitations and only be able to load a fixed amount of data at once. Another reason why we had to do this was because our data was split up across multiple files, versus just simply having one HDF5 file like ModelNet40 had.

Our generator works as follows:

In a loop, we open up the first protein which is 1a1x. We parse the string and grab the amino acid label and we split the line based on a newline. Here is where we can extract the X, Y, Z values that we need. The next step is very important for our architecture. When writing the code, we discovered that for each amino acid, the number of atoms in the volume is variable

and not fixed. This is a problem for our architecture. The reason why is because there is a fully connected layer at the end of the convolutions in the top branch of DGCNN. Fully connected layers require a fixed size input to be passed in so the appropriate amount weights can be built. Although there is a global max pooling right before the fully connected layer, this technically would allow variable sized input but actually implementing variable sized input is no easy task. If we were to pass variable size inputs, we would have to pass them one at a time and we cannot store large batches in a data structure like a numpy array. Because of this, we experimented with having fixed size inputs per amino acid. So for each amino acid, we duplicate (or remove) atoms from the volume to have the same amount for each sample (default is 2048). Another approach we could have done was place a bunch of atoms at the origin instead of duplicating. We have considered this option, however, we have not had enough time to go about experimenting this.

In addition to grabbing 2048 atoms per amino acid, we grab, in total, 2048 amino acids. This results in us having a numpy array of shape (2048,2048,3). The reason why we chose this number is simply that samples were being extracted of the same shape from the ModelNet40 dataset. In the meantime, to grab 2048 samples, we must open up files of multiple different proteins. After we have grabbed the samples, we must construct the labels. Fortunately, the way the code was written, the labels were not stored in one-hot-array values, they were simply stored as the numbers and one hot encoding was done afterward. This is an advantage to us because we did not have to go about building the one hot encodings inside the generator function. Once we grab all the amino acids, we just return the class index in an array of classes and we have built our labels. This concludes our generator.

As mentioned before, the generator returns samples of shape (2048,2048,3). We must now utilize our generator for training. The next step was writing the code to use the number of total samples we wanted to use for training and evaluation, number of epochs, add code to save the model and save figures.

## Difficulties

Some difficulties that may arise in our computations involves understanding the way EdgeConv is used. An assumption used when using this model is that the 3D points represent the surface area. The amino acid 3D structure includes internal 3D points as well. The learning of the locality of others points to maintain invariance works well for real-world object surface points, but for amino acid 3D coordinates that represents the entire structure and not just the surface may be a bottleneck for effectively classifying with this system.

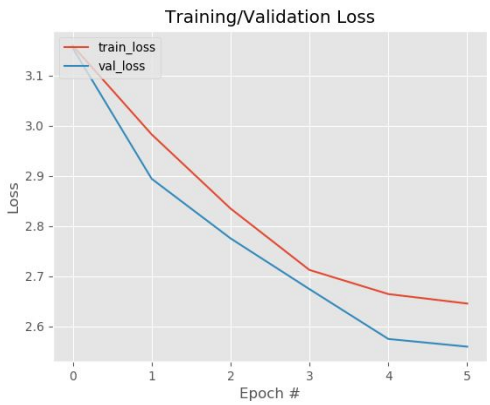
The data necessary for training this model requires a fixed number of points, to overcome this we use zero padding of points at the origin. Due to the varying points sizes, we recognize that the advantage of representation by considering the neighborhood of points instead of each point independently may not prove so effective. The results are discussed below.

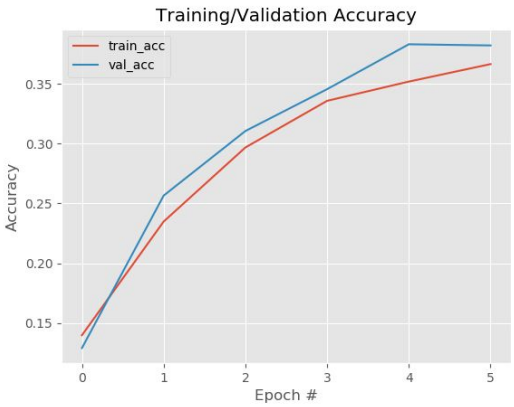
Andrew Florial
Cameron Farzaneh

## Results

We ran our experiment multiple times with different sized inputs and we've come to conclude that the default DGCNN architecture does not perform poorly on our dataset. Without modifying the architecture, we were able to obtain at maximum 56% accuracy on the training set and with 110,000 samples. At this time we had not figured out how to write the code for validation yet. Without specifying our own edge function, and not making many changes to the architecture, we believe that there is room for improvement. The challenge about this problem, however, is how long the computation takes. When we ran the experiment that obtained 56% training accuracy, we ran this for a week during Thanksgiving break and we had only obtained 150 epochs. This is due to the generator running very slow. In addition, at that time, we did not write the code to save the model and to plot the results. After thanksgiving break, we went about doing this and we trained the model for 6 epochs using samples of size (1024,1024,3) with training on 51200 samples and validating on 8192 samples so we could train in a more reasonable time frame. In just 6 epochs, we were able to obtain 39% training accuracy and 38% validation accuracy. Given how difficult this problem is, for using such small sample size and not utilizing additional information such as atom type, we believe that these results are promising. In addition, there is a lot more work that can be done. We believe that the amount of trainable parameters needs to be increased, as the model was not able to fully learn the training set. Also, Dr. Jinfeng Zhang says he has an edge function we could implement. We could also explore placing atoms at origin instead of repeating atoms. In addition, we would need to find a better and faster way of loading the data so training would be much faster. We can do this by preprocessing the coordinates once and saving the numpy arrays so we don't have to open multiple files every time. This would dramatically increase increase the training time. We could also parallelize the generator to be running on it's own process, which would also increase training time. All in all, DGCNN appears to be a promising architecture for this problem. With more time and effort, we believe we can increase the accuracy by adjusting the architecture to incorporate other features such as atom type, custom edge function, and increase in parameters.

## Loss and Validation Plots



Andrew Florial
Cameron Farzaneh

## References

*Dynamic Graph CNN for Learning on Point Clouds,* liuziwei7.github.io/projects/DGCNN.