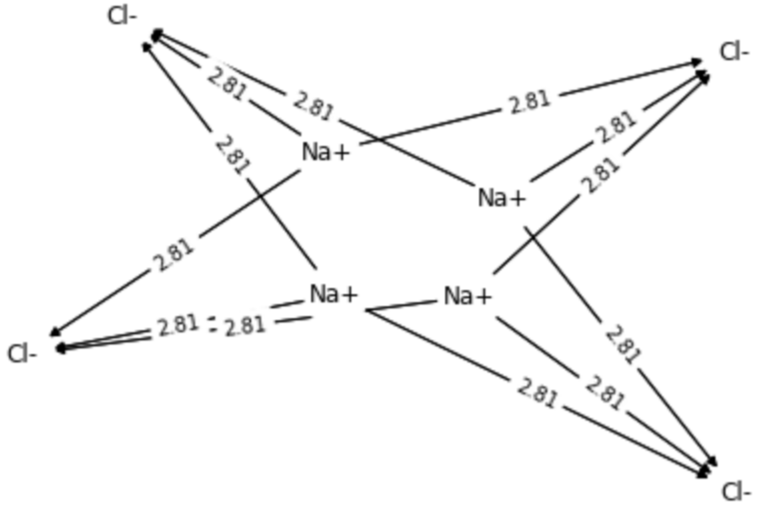
CS182 Project PDF

April 2023

1 Crystal Graph CNNs (CGCNNs)

Deep neural networks are being used everywhere. In this problem we will explore how materials science researchers have used them to predict material properties. You can view the paper for reference:
https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.120.145301.



The key idea of CGCNNs is to represent a crystal structure by a crystal graph, which encodes not only the atomic information but also bonding information between atoms based on the distance. Here we will let each atomic feature vector as v_i, which encodes the property of the atom corresponding to node i. The edge feature vector is represented as $u_{(i,j)k}$, which encodes k-th bond connecting atoms i and j. Note that there can be several bonds between atoms, which originate from the periodic nature of crystals.

Obviously, there are many ways to encode atomic and bond data. For the sake of simplicity, we will follow the methods used in the CGCNN paper, which used a pre-defined encoding vector to change atoms into vectors. These vectors have 92 dimensions, and for 100 atoms in the periodic table, they have different encodings which consist of 0s and 1s. For the edge feature vectors, there also can be many ways to encode bond information; bond length, angle, and covalency. However, we will only use bond-length information between nearest neighbors, by applying a Gaussian kernel to change it to encoding vectors.

1. The graph above represents the crystal structure of NaCl, otherwise known as table salt. Complete the "Encoding Crystal into Crystal Graph" section in **cs182_project_hw.ipynb** to see how we can embed this crystal data into vectors that we can process in a CGCNN.

2. Complete the **model.py** file, which will contain the elements of our CGCNN architecture. Then run the rest of the Python notebook to train and visualize model performance. How does training and validation loss differ with different hyperparameters?

3. In the encoding section of the notebook, we use information from nearby atoms and encode this into our data vector. Experiment with the number of nearest neighbors we look at when embedding. How does changing this affect model performance? Why do you think this is?

4. The most important part of our model is the ConvLayer module, which performs a "similarity check" on the layer inputs (similar to a convolution operation). Note that this layer does not actually convolute our data, but performs a function that achieves the same goal. Experiment with changing the activation functions and batchnorms in the ConvLayer class. How does this affect performance?

5. Take a look at the pooling function in **model.py**. We chose to use average pooling in our model, but how does using max pooling instead affect performance?