MPhys Project Notes

Benjamin Matsaert
Department of Physics and Astronomy, University of Manchester

October 7, 2025

Contents

1 Week 1

1.1 Day 1 : Tuesday, 30 September

Met with supervisor, assigned a number of papers to read through, initially:

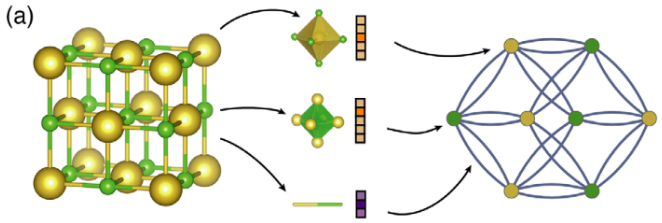
1. `https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.120.145301` (Crystal Graph Convolutional Neural Networks for an Accurate and Interpretable Prediction of Material Properties)
2. `https://www.nature.com/articles/s41524-021-00650-1` (Atomistic Line Graph Neural Network for improved materials property predictions)

1.1.1 Paper 1 Notes

- Most ML methods struggle with modelling infinite cystalline structure...

  ⇒ Represent the structure with a crystal graph, which encodes both atomic info, and bonding interactions between atoms.

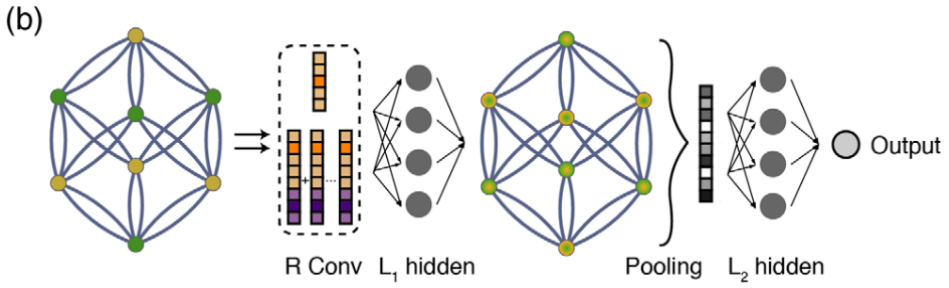
  ⇒ Nodes represent atoms, edges represent bonds between atoms; we **allow** for multiple edges between nodes to account for periodicity.



  ⇒ Each node is represented by a feature vector, $\mathbf{v}_i$, which encodes properties of the atom at that node. Each edge is represented by another feature vector $\mathbf{u}_{(i,j)_k}$ (kth bond between atoms i and j).

- Build convolutional NNs on top of these crystal graphs - achieves good level of accuracy to DFT (density functional theory) calculations.

  ⇒ Composed of two main components: convolution layers and pooling layers.



  ⇒ Convolution layer iteratively updates $\mathbf{v}_i$ with some **nonlinear** convolution function of surrounding atoms and bonds,

$$\mathbf{v}_i^{t+1} = \text{Conv}\left(\mathbf{v}_i^{(t)}, \mathbf{v}_j^{(t)}, \mathbf{u}_{(i,j)_k}\right)$$

  ⇒ The pooling layer then produces some overall crystal feature vector, $\mathbf{v}_c$, via some pooling function of the convolutions,

$$\mathbf{v}_c = \text{Pool}\left(\mathbf{v}_0^{(0)}, \mathbf{v}_1^{(0)}, \cdots, \mathbf{v}_N^{(0)}, \cdots, \mathbf{v}_N^{(R)}\right),$$

  which should satisfy permutational invariance wrt atom indexing, and size invariance wrt unit cell choice. They use a normalised summation as the pooling function.

$$\text{Softmax}\left(\boldsymbol{z}\right)_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)}$$

$$\tilde{\boldsymbol{v}}_i^{(t)} = \text{Softmax}(\boldsymbol{W}_t \boldsymbol{v}_i^{(t)} + \boldsymbol{b}_t)$$

$$\boldsymbol{v}_c = \sum_{t,i} \tilde{\boldsymbol{v}}_i^{(t)}$$

⇒ In addition, there are two *fully connected hidden layers*, with depths L_1 and L_2 - what are these? What do they achieve? Supposedly it helps with capturing complex mapping.

- Training is performed by minimising the difference between predicted properties and DFT calculated properties.
  - ⇒ The whole network can be considered as some function, f, parameterised by a set of weights, $\mathbf{W}$, which map a given crystal to some target property.
  - ⇒ Thus, we try to minimise the cost function wrt these weights, with the hope that the learned weights can be used to predict material properties. We use SGD to optimise for the weights.
  - ⇒ They train the NN on the full materials database (excluding 'ill-converged' crystals).
- An effective convolution function is found in terms of the concatenation of feature vectors of neighbours of i, $z_{(i,j)_k}^{(t)} = v_i^{(t)} \oplus \mathbf{v}_j^{(t)} \oplus \mathbf{u}_{(i,j)_k}$, with

$$\mathbf{v}_i^{(t+1)} = \mathbf{v}_i^{(t)} + \sum_{j,k} \sigma\left(\mathbf{z}_{(i,j)_k}^{(t)} \mathbf{W}_f^{(t)} + \mathbf{b}_f^{(t)}\right) \odot g\left(\mathbf{z}_{(i,j)_k} \mathbf{W}_s^{(t)} + \mathbf{b}_s^{(t)}\right),$$

  where $\oplus$ is element-wise multiplication, and σ is a sigmoid (softmax) function.
- Best MAEs achieved with this are 0.039 eV/atom.
- Does well on predicting band gaps.

1.1.2 General Comments

I won't spend too much time talking about the other papers, since the above is the most relevant for the time being. For completeness, the other papers I read through were:

- `https://advanced.onlinelibrary.wiley.com/doi/10.1002/advs.201801367`
  - ⇒ A review of several deep learning models (MLP, CNN, and DTNN) applied for finding excitation spectra of molecules.
  - ⇒ The NNs find the eigenvalues, and spectra are estimated with Gaussian broadening (of these eigenvalues).
- `https://pubs.acs.org/doi/10.1021/acs.jpclett.5c00169`
  - ⇒ Applying transfer learning methodologies to predict Raman spectra for complicated peptides and proteins.
- `https://www.mdpi.com/2673-4532/3/3/20`
  - ⇒ A nice review paper of various deep learning methods. I used this as a foundation for my understanding of the CNN (convolutional neural network).
- `https://www.nature.com/articles/s41597-025-04593-w`

⇒ The dataset we will be using for the purposes of this project. Once we make the 2D model, we'll use this as training data.

1.2 Day 2 : Thursday, 2 October

After meeting with our supervisor, we decided to proceed for now with the goal:

> Understand the CGCNN model, and attempt to apply it to 2D materials.

The first step, then, was to find the program that the paper uses; conveniently, this was all stored in the following GitHub repo:

CGCNN repo: `https://github.com/txie-93/cgcnn`

I forked this and stored it in our personal repo:

Personal repo: `https://github.com/bd-mat/raman_spectra`

1.2.1 CGCNN Structure

Atomic data is stored in `.cif` files, and loaded into the model with `cgcnn\data.py`. The CGCNN model is contained in `cgcnn\model.py`, and has the following basic structure:

1. **One embedding layer.**

   ⇒ This is a linear layer, so transforms inputs x to output y as some

   $$y = xA^T + b,$$

   where A^T is a matrix of (trainable) weights, and b is a vector of (trainable) biases.

   ⇒ Importantly, this layer acts as a *bottleneck*, and serves to transform $x \to y$, where the dimensionality of y is **lower** than x. Here, we take an input feature vector of length 92, and return an output vector of length 64, for each atom.

   ⇒ The input, x, is a sparse vector of integer values (binary, in fact), while the output is continuously valued (and dense).

2. ~ 4 **convolution layers.**

   ⇒ Each layer applies the convolution function from the CGCNN paper,

   $$\mathbf{v}_i^{(t+1)} = \mathbf{v}_i^{(t)} + \sum_{j,k} \sigma\left(\mathbf{z}_{(i,j)_k}^{(t)} \mathbf{W}_f^{(t)} + \mathbf{b}_f^{(t)}\right) \odot g\left(\mathbf{z}_{(i,j)_k} \mathbf{W}_s^{(t)} + \mathbf{b}_s^{(t)}\right), \tag{1}$$

   given inputs for each atom of `(atom_feature_vector,nbr_bond_vectors,nbr_indices)`. The atom feature vector in question has *already* passed through the embedding layer, so is just that for the first layer, and in further convolution layers, is simply the output of the previous layer.

   ⇒ Obviously, the atom feature vector is the only input to the convolution function which actually changes through these layers, so the other two inputs are 'effectively' constants.

   ⇒ We don't really need any more than 4 convolution layers, since our graph is relatively small, so this is sufficient for an atom to 'learn' about all other atoms in the structure.

3. **One pooling layer.**

   ⇒ We pool the set of *atom* feature vectors to a single *crystal* feature vector, which is achieved by just summing them [?]. The output has the same length as a single atom feature vector.

4. **Hidden layers.**

   ⇒ The output of the pooling layer is then passed through a series of hidden layers, each of which are modelled as linear transformations, which are then passed through a *softplus* function, before being sent to the next hidden layer.

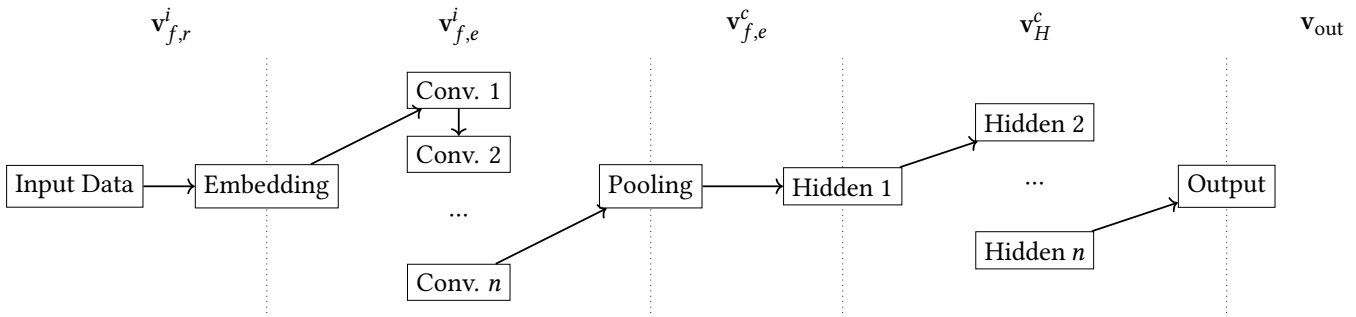
   ⇒ The hidden feature vectors have a **different** length to the crystal feature vector, and so the first layer (which is always there) acts in a similar way to the embedding layer, changing the size of the feature vector from 64 to some number, e.g. 32, which is then unchanged through the rest of the hidden layers.

5. **Output layer.**

   ⇒ Finally, the final hidden feature vector (of length 32) is passed to the output layer, which is again **linear**, and now maps to the **final** output of the neural network.

   ⇒ If we are estimating a single quantity, the output is just a scalar. Potentially, we can model multiple outputs by just increasing this value? Will require some testing ‿.

I will summarise the CGCNN structure in the following diagram:



Hopefully, I can now paste the definitions of the main CGCNN class, and the action of each line will be obvious:

Firstly, the initialisation of the class:

```
super(CrystalGraphConvNet, self).__init__()
self.embedding = nn.Linear(orig_atom_fea_len, atom_fea_len)
self.convs = nn.ModuleList([ConvLayer(atom_fea_len=atom_fea_len,
                                      nbr_fea_len=nbr_fea_len)
                            for _ in range(n_conv)])
self.conv_to_fc = nn.Linear(atom_fea_len, h_fea_len)
self.conv_to_fc_softplus = nn.Softplus()
if n_h > 1:
    self.fcs = nn.ModuleList([nn.Linear(h_fea_len, h_fea_len)
                              for _ in range(n_h-1)])
    self.softpluses = nn.ModuleList([nn.Softplus()
                                     for _ in range(n_h-1)])
self.fc_out = nn.Linear(h_fea_len, 1)
```

Secondly, the forward function (i.e. action of the NN):

```
atom_fea = self.embedding(atom_fea)
for conv_func in self.convs:
    atom_fea = conv_func(atom_fea, nbr_fea, nbr_fea_idx)
crys_fea = self.pooling(atom_fea, crystal_atom_idx)
crys_fea = self.conv_to_fc(self.conv_to_fc_softplus(crys_fea))
crys_fea = self.conv_to_fc_softplus(crys_fea)
if hasattr(self, 'fcs') and hasattr(self, 'softpluses'):
    for fc, softplus in zip(self.fcs, self.softpluses):
        crys_fea = softplus(fc(crys_fea))
out = self.fc_out(crys_fea)
return out
```

I will go into more detail into the pooling and convolution parts next week (hopefully).