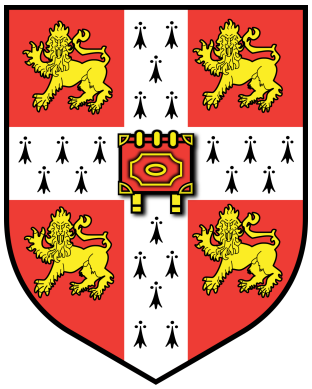
CMACE: An Expressive Equivariant Graph Neural Network with Higher Rank Cartesian Tensors

Candidate Number: 8205R
Supervisor: Pietro Liò
Word Count: 4971

May 15, 2023



Department of Physics

University of Cambridge

Abstract

In the past decade, Machine Learning has provided a wide range of computationally efficient models for calculating interatomic potentials for molecules whilst still retaining DFT-level accuracy. An example being Equivariant Graph Neural Networks which elevate scalar features to tensors enhancing the range of physical features they can model. This project presents CMACE, the first Cartesian tensor based model to have tensors of rank >1 and >3 many-body terms, drawing inspiring from it's spherical tensor analogue, MACE. Cartesian tensors allow for intuitive visual explanations of the operations of equivariant networks - a perspective currently absent from the literature. This approach simplifies equivariant models making them more accessible to audiences unfamiliar with spherical tensors. This work demonstrates CMACE maintains the same theoretical expressivity properties as MACE, making it more powerful than any current Cartesian architecture. CMACE also uses use tensor contraction path saving to speed up the contraction process $> 5\times$ in some cases. Finally, the codebase lays solid foundations with which to optimise CMACE for benchmarking and other applications.

Contents

1 Introduction

Accurate interatomic potentials are essential for applications in chemistry, molecular biology, and material sciences. At the lowest level, it is possible to solve these many-body systems *ab initio* using a simulation of Schrödinger's equation, though this is not computationally feasible at even small scales due to the exponential growth of the Hilbert space, therefore different alternatives have been devised at various levels of the speed vs. accuracy trade off. Treating all charges as points, we achieve the many-body expansion see in Equation 1. Again, in larger systems calculations quickly become intractable due to the number of terms growing combinatorially with body-order. Density Functional Theory (DFT) [1, 2] has long been the antidote to this poor scaling by offering an extremely accurate, tractable framework to calculate interatomic potentials. Although much faster than using Equation 1, DFT still takes $\sim 10^3$s for small molecules of ~ 20 atoms and on the order of days for larger molecules.

$$E_i = E^{(1)}(\mathbf{r}_i) + \frac{1}{2}\sum_j E^{(2)}(\mathbf{r}_i, \mathbf{r}_j) + \frac{1}{6}\sum_{j,k} E^{(3)}(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \frac{1}{24}\sum_{j,k,l} E^{(4)}(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k, \mathbf{r}_l) + \cdots \quad (1)$$

Machine Learning Interatomic Potentials (MLIP) emerged from the desire to gain more speed improvements while retaining DFT-level accuracy. Where DFT uses input positions and a computational method derived from physics to calculate output properties, MLIPs use input positions and training data from DFT to reconstruct a new computational method that approximates the original that is run at a fraction of the cost. After training, MLIPs are much quicker as DFT calculations scale cubically with the number of electrons whereas MLIPs often scale linearly with the number of atoms. Machine Learning models are trained by optimising the performance of a function, $f(x_i, w) = y_{i,\text{pred}}$ that predicts the output $y_{i,\text{true}}$ for the input x_i. We do this via minimising a loss function, such as the mean square error as seen in Equation 2.

$$\text{loss} = \text{mean}((y_{i,\text{pred}} - y_{i,\text{true}})^2) = \text{mean}((f(x_i, w) - y_{i,\text{true}})^2) \quad (2)$$

For simple models such as linear regression, it is possible to minimise the loss with respect to the parameters analytically via matrix inversion [3]. In the case of non-linear models such as neural networks, finding an analytical solution is not possible. Equation 3 shows how these models employ gradient descent to adjust the parameters *a little bit* (the learning rate η decides how big this is) in the parameter-space direction that minimises the loss the most. Following each adjustment, the loss is reassessed to determine the new optimal direction for the subsequent parameter shift. In recent years, many physically-inspired additions have been made to this algorithm such as the addition of 'momentum' in Adam [4].

$$w_{i+1} = w_i - \eta \frac{\partial \text{loss}}{\partial w_i} \quad (3)$$

Some MLIPs use Basis Function Regression, a type of linear regression that produces a hyperplane of best fit by finding the coefficients to a linear combination of basis functions of variables such as position, atom type etc. An example in 1D is the polynomial set of basis functions (see Equation 4) which is able to approximate any continuous function to arbitrary accuracy i.e. is complete.

$$y = \sum_{n}^{\infty} \alpha_n x^n = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots \tag{4}$$

Basis function selection needs to problem specific so is often a process informed by domain knowledge i.e. symmetries, physical laws etc. - the design of models to capture this domain knowledge is referred to as *inductive bias*. Inductive bias restricts the possible function space by not allowing functions that don't obey the symmetry/physical law and therefore reduces the amount of data needed to train

Over the past decade, many ways have been proposed to find these basis functions [5–7] but this has lacked the clarity of a unified framework which exist for similar problems in adjacent fields such as computer vision [8]. The Atomic Cluster Expansion (ACE) [9, 10] bridges this gap by offering a treatise in systematically creating sets of basis functions which are provably complete and allows for arbitrarily high-body order at constant cost. It has been shown that many atomic environment descriptors such as MTP, SOAP [6, 7] can be expressed in the ACE framework. Using these basis states as a basis for basis function regression has proved very effective and rivalled results of many far more complex architectures [11].

At the same time, the deep learning community was tackling MLIPs from a Graph Neural Network (GNN) perspective. Again, nodes share an edge if they are within some cutoff radius, r_c. GNNs use one/many rounds of message-passing in which nodes, which carry a list of scalar features, will received 'messages' from nodes they share an edge with (their neighbourhood, $\mathcal{N}$) as a function of these features. These messages are then aggregated in some permutation-invariant (e.g. sum) way and used alongside the current node features as an input to an update function which outputs the new features. Early architectures of this type started off with layers using two-body scalar (i.e. invariant) properties such as relative distance between neighbours [12, 13]. Other architectures [14–16] extended this idea by explicitly forming many-body invariants such as bond angles from the three body terms (in Figure 1 for nodes A, C in the neighbourhood of B: $\vec{x}_{BA} \cdot \vec{x}_{BC} \sim \cos\theta_{ABC}$) in DimeNet. These methods offered state-of-the-art results whilst having an inference time ~ 5 orders of magnitude less than DFT. As the many-body terms are calculated explicitly these models are subject to the many-body scaling of Equation 1.

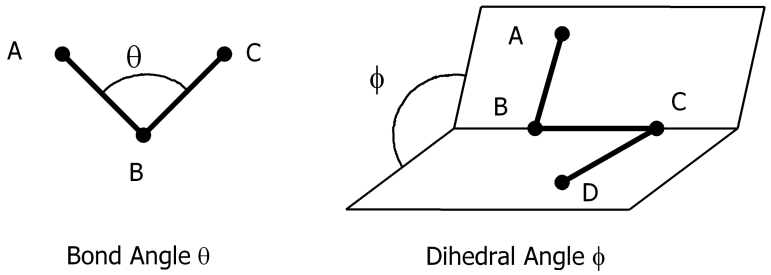


Figure 1: *left*: the angle θ is a three-body term, as positions of A, B, C are required to calculate via $\vec{x}_{BA} \cdot \vec{x}_{BC} \sim \cos\theta_{ABC}$. Note that both vectors $\vec{x}_{AC}, \vec{x}_{AC}$ can be generated in the neighbourhood of B.
right: the dihedral angle is a four-body term $\vec{x}_{AB} \cdot \vec{x}_{CD} \sim \cos\phi$. As these two vectors are from different graph neighbourhoods ϕ can only be calculate if vectors are passed between layers. [17]

Invariant scalar features in GNNs can be promoted to equivariant tensors e.g. vectors and matrices. This is on the intuition that equivariant features retain information about the coordinate system between layers which allows for the formation invariants involving geometric properties from multiple neighbourhoods i.e. dihedral angle as seen in Figure 1. Also, many physical properties of molecules are non-scalars such as vector dipole moments or polarisability matrices so its good to have features that transform in the same way. This led to the development of many equivariant architectures such as TFN, Comorant, E(n)-GNN [18–20]. Though these architectures were brilliant for propagating symmetries through multiple layers, only two-body terms were within a layer so these architectures struggled with some tasks that 'simpler' architectures such as Dimenet could solve. Figure 2b shows the timeline of both the invariant and equivariant architectures.

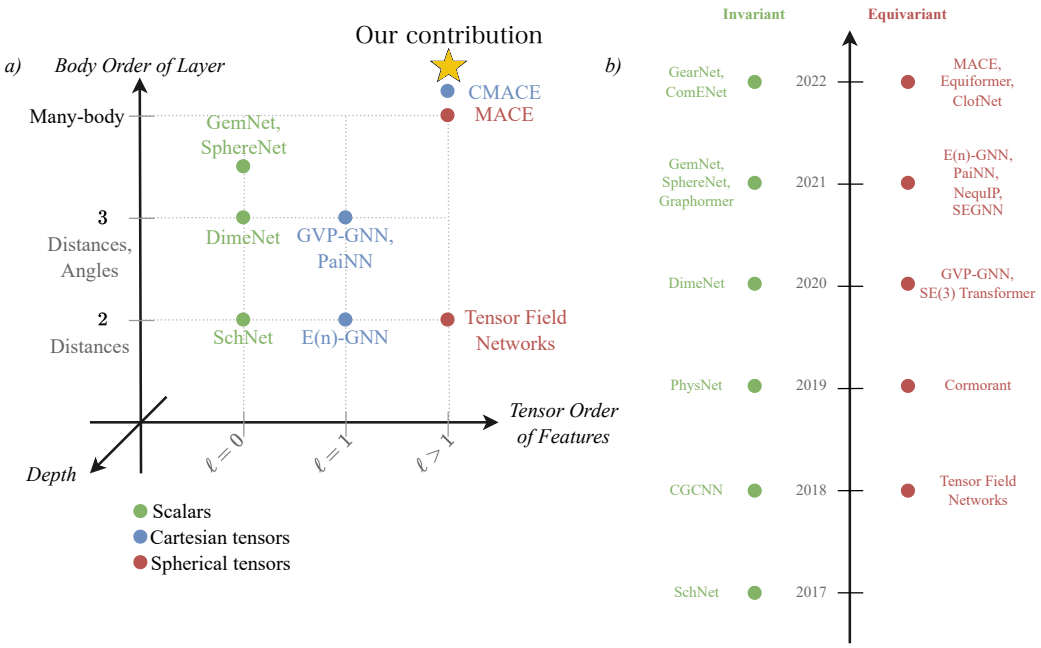


Figure 2: a) shows the body-order and tensor feature order of current invariant and equivariant architectures. **CMACE is the only Cartesian architecture with $\ell > 1$ and body-order > 3** b) shows the timeline of such architectures. Figure adapted from Joshi et al. 2023 [21]

Recently, the MACE architecture [22] unified the strong inductive bias of the Atomic Cluster Expansion with the highly-expressive framework of message-passing equivariant GNNs. This was made possible by elevating the one-hot encoding of nodes in ACE to continuous features embedding allowing for the message-passing in a equivariant networks. Although this marriage of two disparate disparate parts of the literature marks a great step forward, many researchers who find great use out of equivariant networks are in fields such as biology, chemistry and computer science. These practitioners may not have much/any experience with the spherical tensors which MACE and all other high-rank equivariant GNNs use as a basis for their tensors. Also, the necessary framework for tensor products and contractions for spherical tensors are highly non-trivial and is very computationally expensive requiring the inception of complex libraries such as `e3nn` to deal with Clebsch-Gordan coefficients. This motivated our contributions which are as follows:

1. Creation of **CMACE model in Cartesian basis** which takes the most important ideas from MACE. This is the **first Cartesian model both to have >3-body terms and/or use Cartesian equivariant tensors of rank >1.** As shown in Figure 2a.

2. The use of Cartesian tensors allows for the use of tensor networks notation which provide an **intuitive pedagogical resource for understanding equivariant models**.

3. Provide codebase[1] that doesn't require knowledge of extra machine learning packages making the model easily usable and modifiable by a wide range of the MLIP community.

4. Experiments that show **CMACE retains the theoretical expressivity properties of MACE**, this promise encourages the optimisation of the model for benchmarking in future.

This write-up starts with a brief background of message passing, equivariance and spherical/Cartesian tensors. Then the CMACE architecture is introduce via intuitive figures to explain how we use high rank tensors and why implicit many-body terms emerge. Then experiments of expressivity show that CMACE is as theoretically powerful as MACE. Finally, we look at the choice of code for this project and offer some suggestions for further work to take this project forward.

2 Background

2.1 Message passing

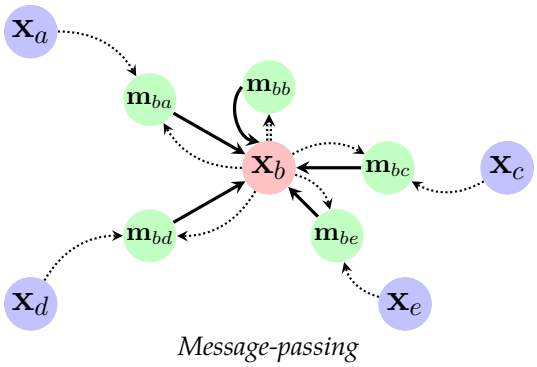


Figure 3: A schematic showing a single message passing step with node b aggregating messages that it receives from its neighbours $\mathcal{N}(b) = a, c, d, e$ taken from Veličković 2022 [23]

GNNs for MLIPs use the message-passing paradigm over one/many layers to update node features. The feature of an atom, along with the atom type and position make up the atom state given by the tuple:

[1]Code available on GitHub: `https://github.com/hws1302/partIII-project/tree/main/cartesian_mace`

$$\sigma_i^{(t)} = (\underbrace{\vec{r}_i, z_i}_{\text{constant}} , \mathbf{h}_i^{(t)}) \tag{5}$$

This process first starts with nodes (atoms) receiving messages (M) from nodes it shares an edge with (i.e. its neighbours) and aggregating these messages in some permutation-invariant way ($\oplus$), as in Equation 6. Two nodes will share an edge if their distance is less than some cutoff radius, r_c. The aggregated message for each node is then used with the current state of the atom to update (U) the features, as in Equation 7. Using the notation from the MACE paper [22].

$$m_i^{(t)} = \bigoplus_{j \in \mathcal{N}(i)} m_{ij}^{(t)} = \bigoplus_{j \in \mathcal{N}(i)} M(\sigma_i^{(t)}, \sigma_j^{(t)}) \tag{6}$$

$$h_i^{(t+1)} = U(h_i^{(t)}, m_i^{(t)}) \tag{7}$$

After the desired number of layers, there is graph-wide aggregation of features in some permutation invariant way and are mapped to a scalar such as energy by a readout function ($\mathcal{R}$), as in Equation 8.

$$E = \mathcal{R}(\bigoplus_i h_i^{(t)}) \tag{8}$$

2.2 Invariance and Equivariance

For a group element $g \in \mathcal{G}$ which can act on both the input space ($\{x\}$) and output space ($\{y\}$) of f via the representation $D(g)$. The form of $D(g)$ is dependent on what it is acting on. If a function is not affected by the symmetry group $\mathcal{G}$ it is said to be $\mathcal{G}$-invariant. Whereas if a function transforms in the same way as the symmetry group $\mathcal{G}$ it is said to be $\mathcal{G}$-equivariant. For a mathematical definition see Equation 9.

$$\begin{aligned} \mathcal{G}\text{-invariant:} \quad & f(x) = f(D(g) \cdot x) \\ \mathcal{G}\text{-equivariant:} \quad & D(g) \cdot f(x) = f(D(g) \cdot x) \end{aligned} \tag{9}$$

Invariant functions 'throw away' the symmetries related to group $\mathcal{G}$ where as equivariant functions propagate these symmetries forward to the output of the function, hence the output trasnform in the same way. See Figure 4 to see how this with how as we rotate the input (i.e the graphs) some output features i.e distances and angles are invariant whereas the vector quantity of relative position does transform in the same way. This project has layers equivariant to the Euclidean group O(3), rotations, and reflections.

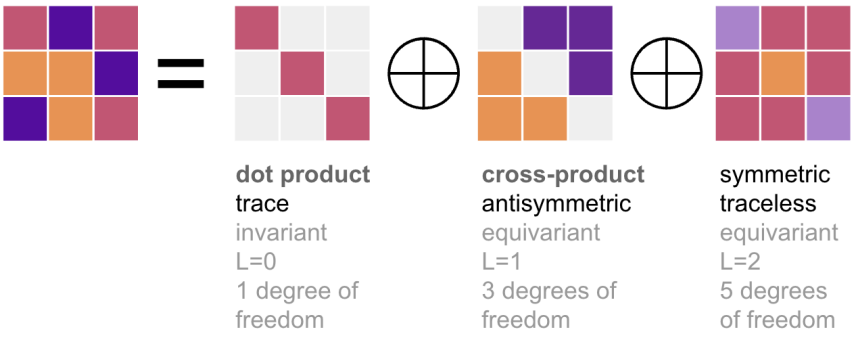


Figure 5: General matrix is a Cartesian tensor that can be decomposed into direct sum of spherical tensors with rank $0, 1, 2$ (Smidt 2021) [24]

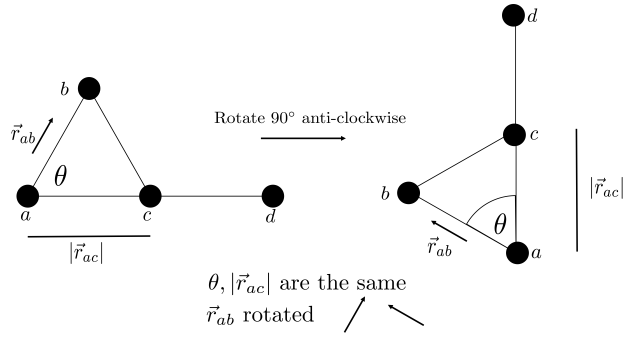


Figure 4: As we rotate the geometric graph 90° anti-clockwise, the invariant properties of distance $|\vec{r}_{ac}|, \theta$ do not change but the equivariant vector $\vec{r}_{ab}$ does change with the rotation

2.3 Tensor Operations and Abstractions

2.3.1 Cartesian vs. Spherical tensors

As discussed, the MACE architecture uses spherical tensors whereas our CMACE architecture uses Cartesian tensors. The reason that spherical tensors have been used up to this point is that they very nicely fall out from solving systems with spherical symmetry and have the same transformation properties of the O(3) group.

Representations. In previous equivariant models, rank-ℓ tensors have been represented by a linear combination of the $2\ell + 1$ spherical harmonics of the same rank, the coefficients can be put into a vector of the same size. Meanwhile Cartesian tensors of rank ℓ have shape $\underbrace{3 \times 3 \times \cdots \times 3}_{\ell \text{times}}$

Transformations. Cartesian tensors also have different representations of an orthogonal transformation matrix Q which we represent by $D_\ell(Q)$. For spherical tensors, these are the $(2\ell + 1) \times (2\ell + 1)$ Wigner-D matrix whereas we Cartesian tensors we must transform each index individually using ℓ lots of the 3x3 transformation matrix Q (as in Equation 10 for rank-3).

$$T'_{ijk} = Q_{ip} Q_{jq} Q_{kr} T_{pqr} \tag{10}$$

Decomposition. Figure 5 shows that a Cartesian tensor of rank-2 (matrix) can be decomposed into the trace, an anti-symmetric cross product, and a symmetric traceless matrix. From this point forward, we will be using Cartesian tensors unless stated otherwise.

2.3.2 Tensor Contractions

Throughout this report, the Tensor Network[2] notation will be used to visualise Cartesian tensors and their contractions. Much like Feynman diagrams, this notation allows for instances of combinatorial problems to be expressed with very few rules. This provides an intuitive tool in which each tensor is represented by a node and each *leg* represents an index, the legs may be either free or connected to another leg, in the latter case we *contract* between these two indices which in the case of Cartesian tensors amounts to setting them equal and summing (see figure 6). For spherical tensors we still have the idea of contracting but this involves the Clebsch-Gordan coefficients and thus doesn't have the same intuitive visualisations - making explanations and visualisations with Cartesian tensors far simpler.

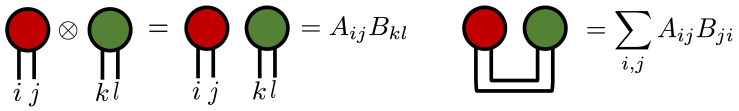


Figure 6: Tensor Network representation of Cartesian tensors. Here each node is a tensor and the number of legs is the number of indices. Joining two legs together is equivalent to summing over that pair of indices. Tensor products are implied when nodes are next to each other. Different node colours represents different value i.e. $A \neq B$

To understand the operation of our architecture we must define a `contract` operator which for a given tensor as input will produce all possible contractions of that tensor of desired output rank (the idea of the operator is not specific to Cartesian tensors). Figure 7 shows how to visualise this operator using tensor networks. As seen, it is only possible to get non-zero contractions when the difference between the initial and final ranks is even as contractions are done over pairs of indices. More generally, the number of different ways to do m contractions to a tensor with n free indices is given by equation 11 (proof in Appendix A.3). These contractions are symmetric and will give our layers $O(3)$ equivariance.

$$f(n, m) = \frac{n!}{(2-2m)! \cdot m! \cdot (2!)^m} \tag{11}$$

2.3.3 Channel mixing

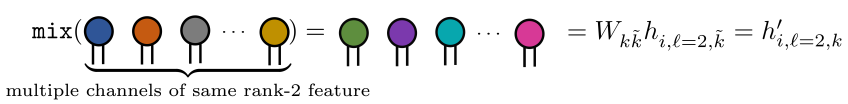


Figure 8: The action of the `mix` operator over the channels of a rank-2 feature. This is equivalent to linear transformation of an orthogonal weight matrix

In our models, as in many other machine learning models, each of the feature tensor is carried multiple times, with each copy being referred to as a 'channel'. At multiple stages

[2] `https://tensornetwork.org/`

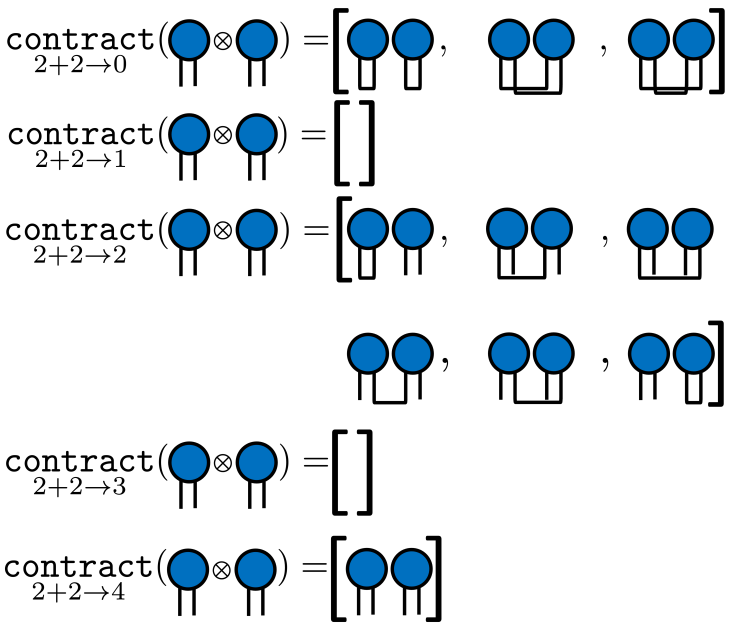


Figure 7: The action of the $\underset{\text{rank in}\to\text{rank out}}{\texttt{contract}}$ on a tensor of rank $2+2=4$ to ranks $0, 1, 2, 3, 4$. Output returned as a list of all possible contractions of that rank out. Contractions produce no output if rank in - rank out is not even.

in the model, the channels are mixed via an orthogonal weight mate by a matrix of learned weights (unique to each instance of `mix`) over the channel index, k. Figure 8 shows the channel mixing of some rank-2 tensor features (i.e. matrices), here different colours represent different values and thus the change of colour shows the mixing. The intuition for multiple channels being useful is that different channels can focus on distinct tasks e.g. one channel might be dealing with the charge whilst another will be dealing with the bond angles. Throughout this project, the channel index is k and can be ignored in all operations except for `mix`.

3 MACE in a Cartesian basis

The Atomic Cluster Expansion (ACE) [9] has played a crucial role in developing MLIPs. Whilst architectures like Dimenet explicitly account for high body-order terms (to great computational expense) ACE provides a framework for constructing a complete polynomial basis systematically at constant cost per basis function independent of body-order [10]. ACE takes advantage of a body-order expansion that implicitly generates higher-order terms MACE then uses channel mixing and learned weights to 'pick out' this most important terms, Figure 12 shows this process. The MACE/ACE architecture can be split up into four main parts, we will do this in the most general way that applies to both our Cartesian MACE and the MACE framework.

3.1 Particle Basis, ϕ

The particle basis is a two-body basis created between a central node i and peripheral node j that share an edge, where as in other models edges are shared between nodes within the cutoff distance, i.e. $r_{ij} < r_c$. They are a function of relative position, r_{ij}, and the features of the peripheral node h_{j,k,ℓ_1}. As the use of features is asymmetric $\phi_{ij} \neq \phi_{ji}$.

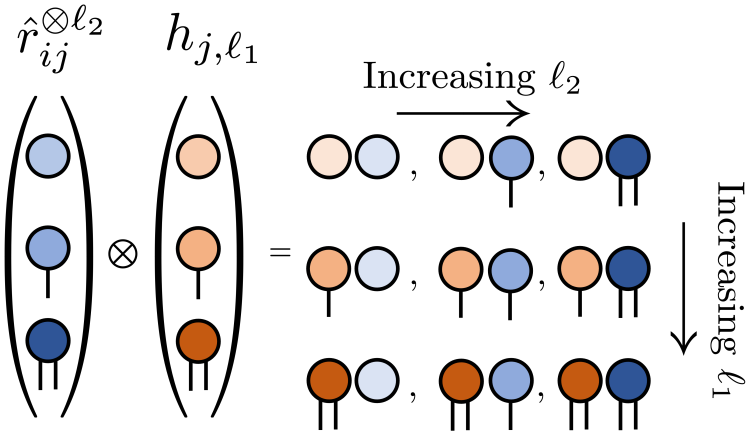


Figure 9: The 9 different tensors resulting from the projection of j's feature tensors, with rank-ℓ_1, onto the angular basis tensors of rank-ℓ_2. The angular basis is achieve via the outer product of the normalised displacement between i and j ℓ_2 time. These angular basis functions are inspired by the Moment Tensor Potentials [6].

The particle basis for node i is formed by, first, projecting the channel-mixed features $\texttt{mix}(h_{j,k\ell_1})$ of a neighbouring node $j \in \mathcal{N}(i)$ of rank-ℓ_1 onto a set of angular basis tensors, $\Theta_{\ell_2}(\hat{r}_{ij})$ of rank-ℓ_2 which are a function of the normalised displacement vector between nodes i and j, $\hat{r}_{ij}$. For MACE these consist of the spherical harmonics $\mathrm{Y}_\ell^m(\hat{r}_{ij})$ and for CMACE these are the outer product of the normalised displacement vector, $\hat{r}_{ij}^{\otimes \ell_2} = \underbrace{\hat{r}_{ij} \otimes \cdots \otimes \hat{r}_{ij}}_{\ell_2 \text{times}}$, these are called Moment Tensors [6]. The projection amounts to a tensor product. This results in tensors with rank-$(\ell_1 + \ell_2)$, Figure 9 shows the $3 \times 3 = 9$ projections for when $\ell_1, \ell_2 \in 0, 1, 2$. This action is shown in Equation 12 and produces the intermediate state $\Psi_{ij,k,\ell_1+\ell_2}$.

$$\Psi_{ij,k,\ell_1+\ell_2} = \Theta_{\ell_2}(\hat{r}_{ij}) \otimes \texttt{mix}(h_{j,k,\ell_1}) \tag{12}$$

Next, these projections $\Psi_{ij,k,\ell_1+\ell_2}$ of rank-$(\ell_1 + \ell_2)$ are contracted to the desired rank ℓ_3 via the `contract` operator. Figure 7 shows how the `contract` operator gives all possible contraction combinations as a list. In the case of the particle basis we sum up all these different paths and multiply by a scalar radial basis functions (RBFs), $\mathrm{R}_{k,\ell_1\ell_2\ell_3}(|r_{ij}|)$, which is a function of distance between a node i and its neighbour j. This operation is shown in Equation 13. These states are therefore two-body as they only include information about two nodes i (position) and j (position and features). The RBFs are indexed by $k, \ell_1, \ell_2, \ell_3$ as there are different learned weights for each combination of the indices, ℓ_1, ℓ_2, ℓ_3, but the more important thing is that we scale projections by some radially dependent function.

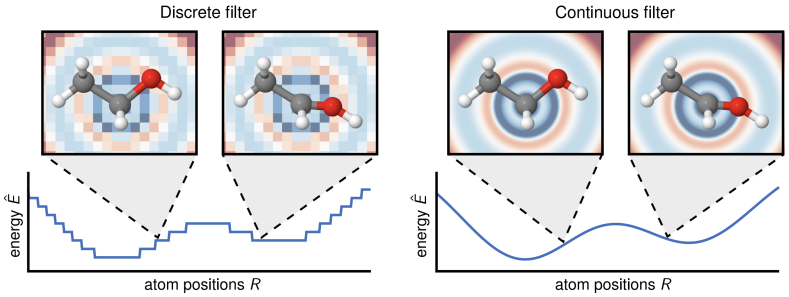


Figure 10: Schnet introduced the idea of smooth radial basis function. This provided smooth output making gradient descent easier

$$\phi_{ij,k,\ell_3} = \sum_{\ell_1,\ell_2} \mathrm{R}_{k,\ell_1\ell_2\ell_3}(|r_{ij}|) \underset{\ell_1+\ell_2\to\ell_3}{\texttt{contract}}(\Psi_{ij,k,\ell_1+\ell_2}) \tag{13}$$

In the first geometric GNNs [12] the value of these RBFs were discretised by dividing the distances into bins and assigning values to each of these bins. Schnet was the first model to propose the use of continuous RBFs, this was an essential leap forwards as no only were its outputs continuous and were therefore more physical (see Figure 10), this continuity propagates to the loss such that it has a continuous gradient which is essential for gradient descent. Equation 14 shows the 'Radial Bessel Basis' used by MACE/CMACE as introduced by Dimenet, the frequency grows with the channel number. This physically inspired solution gives an orthogonal set and requires 20× fewer RBFs than the equally-spaced Gaussians used in Schnet.

$$R_{\ell_1\ell_2\ell_3,k}(r) = w_{\ell_1\ell_2\ell_3}\sqrt{\frac{2}{c}}\frac{\sin(\frac{k\pi}{c}r)}{r} \tag{14}$$

3.2 Atomic Basis, A

Formation of the atomic basis is a simple step in which we sum up the two-body particle basis states of each rank (Equation 15), ϕ_{ij,k,ℓ_3} of node i with all of its neighbours.

$$A^{(t)}_{i,k,\ell_3} = \sum_{j\in\mathcal{N}(i)} \phi_{ij,k,\ell_3} \tag{15}$$

This leave each node with an atomic basis feature for every tensor rank. For example, if the maximum rank of A is two, $[A_{i,k,\ell_3=0}, A_{i,k,\ell_3=1}, A_{i,k,\ell_3=2}]$.

3.3 Product Basis, B

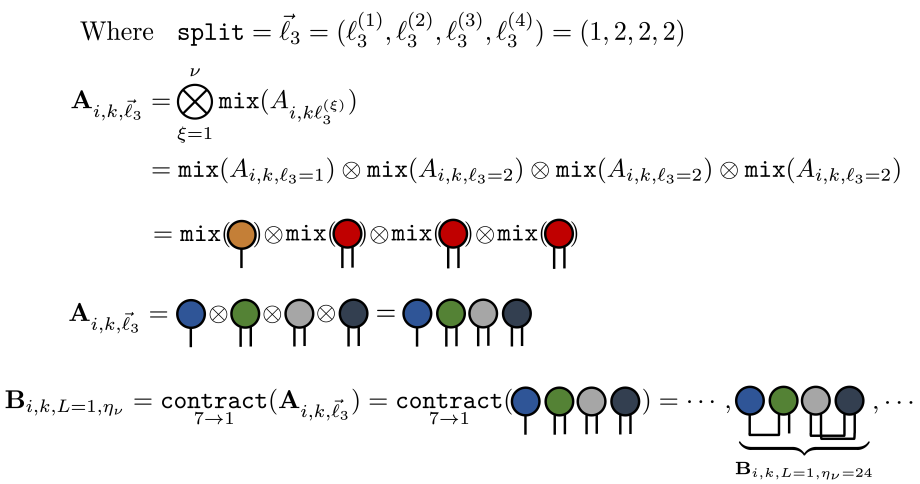


Figure 11: process of Equations 16 and 17 turning atomic basis functions for a given node, to the the basis functions $\mathbf{B}$ for a particular $\texttt{split} = \vec{\ell}_2 = (1, 2, 2, 2)$ (i.e. one vector and three matrices). The fact all three matrices that are red before the mix are different colours after shows that the weights for each mix is unique. The `contract` operator is then used as normal.

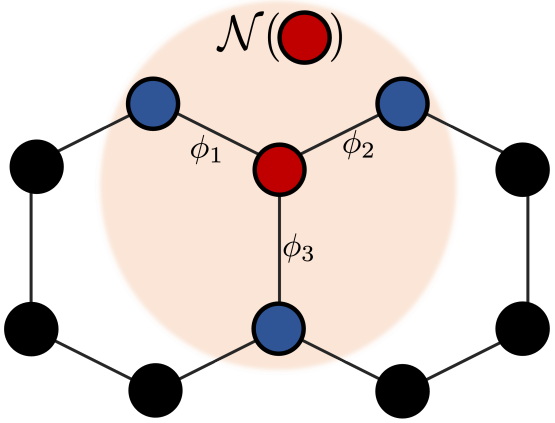
The formation of the product basis states, $\mathbf{B}$, is the key step that creates implicit higher body-order terms. The $\mathbf{B}$'s for each node are formed out of only the atomic basis states from that same node. The correlation order, ν tells us how many different atomic basis tensors will be involved in the creation of a given $\mathbf{B}$. For each ν every possible 'split' of atomic basis ranks is calculated and denoted by a vector of ℓ_3's, $\vec{\ell}_3$. For example, if $\nu = 2$ and the maximum rank of the atomic basis is 2 we can have six different splits $\vec{\ell}_3 = (0,0), (0,1), (0,2), (1,1), (1,2), (2,2)$. For each split the A's chosen have their channels mixed (unique to this split). This yields the intermediate tensor representation boldface $\mathbf{A}$ which has the rank $\texttt{sum}(\vec{\ell}_3)$ which is the sum of elements of ℓ_3, as seen in its definition in Equation 16.

$$\mathbf{A}^{(t)}_{i,k,\vec{\ell}_3} = \bigotimes_{\xi=1}^{\nu} \texttt{mix}(A^{(t)}_{i,k\ell_\xi}) \quad \text{where} \quad \vec{\ell}_3 = (\ell_3^{(1)}, \ell_3^{(2)}, \cdots, \ell_3^{(\nu)}) \tag{16}$$

After this, like in the production of the two-body particle basis, the `contract` operator is used to get $\mathbf{B}$ of the required rank L. Except in this case, we do not sum over the possible input splits and output combinations but except denote each unique contraction path by η_ν, this can be seen visually in Figure 11.

$$\mathbf{B}^{(t)}_{i,k,L,\eta_\nu} = \underset{\texttt{sum}(\vec{\ell}_3)\to L}{\texttt{contract}}(\mathbf{A}^{(t)}_{i,k,\vec{\ell}_3}) \tag{17}$$

Implicit high body-order. The correlation order, ν, is the number of atomic basis tensors are used in constructing each $\mathbf{B}$ with that correlation order. Figure 12 shows an example where we only consider scalar two-body particle states, with $\nu = 2$, here we see that we have implicit three-body terms. More generally, if the correlation order is ν, the highest body-order terms will be $\nu + 1$.



$$A_{\bullet} = \sum_{j \in \mathcal{N}(\bullet)} \phi_j = (\phi_1 + \phi_2 + \phi_3)$$

$$\mathbf{B}_{\bullet} = (\phi_1 + \phi_2 + \phi_3)^2 = \phi_1^2 + \phi_2^2 + \phi_3^2 + \underbrace{2\phi_1\phi_2 + 2\phi_1\phi_3 + 2\phi_2\phi_3}_{\text{implicit 3-body terms}}$$

Figure 12: This shows how in the neighbourhood of the red node the three particle basis states (which are scalars here for simplicity) with neighbours are aggregated. In this case $\nu = 2$ and thus we square the atomic basis states. This gives us terms like $2\phi_1\phi_2$ which are three-body as each particle basis state requires two nodes to make, the central and peripheral.

Computational advantage. The computational speed of MACE comes from the fact that the architecture never explicitly calculates $\mathbf{A}_{i,k,\vec{\ell_3}}$, as the contractions absorb tensor products. For example, if we have two arbitrary tensors $C_{ijk}, Dlmn$ and carry out the contraction over the indices $(i,l),(j,m),(k,n)$ of the tensor product of C and D. Equation 18 shows that the order of tensor product vs contraction doesn't matter.

$$\sum_{ijk} E_{ijkijk} = \sum_{ijk} C_{ijk} \otimes D_{ijk} \tag{18}$$

The first method needs calculation and storing of tensor E_{ijklmn} with $3^6 = 729$ elements where as the second method need us to access two tensors that are already stored with $2 \times 3^3 = 54$ elements. It is easy to see that this advantage scales exponentially so only becomes more important as the maximum rank of A and $\nu_{\max}$ grow.

This process can be conceptualised as extracting low-rank properties of the a high rank tensor (that is never actually calculated). Through the use of learned weights, the most important low-rank properties i.e. B's will becomes very important in the output. This idea is reminiscent of *the kernel trick* used for Support Vector Machines in which a high-dimensional space is implicitly accessed via inner products.

3.4 The Message, m

$$m_{i,L}^{(t)} = \sum_{\nu=1}^{\nu_{\max}} \sum_{\eta_\nu} W_{z_i L, \eta_\nu}^{(t)} B_{i,\eta_\nu L}^{(t)} \tag{19}$$

The *message* is formed via a (learned) linear combination of all the product basis states B (by considering all paths for all correlation order). Figure 13 illustrates what this message

the scalar and vector message would look like in tensor network notation, with the highest rank representation of the atomic basis being matrices and the maximum correlation order being 4, the different colours represent channel mixings.

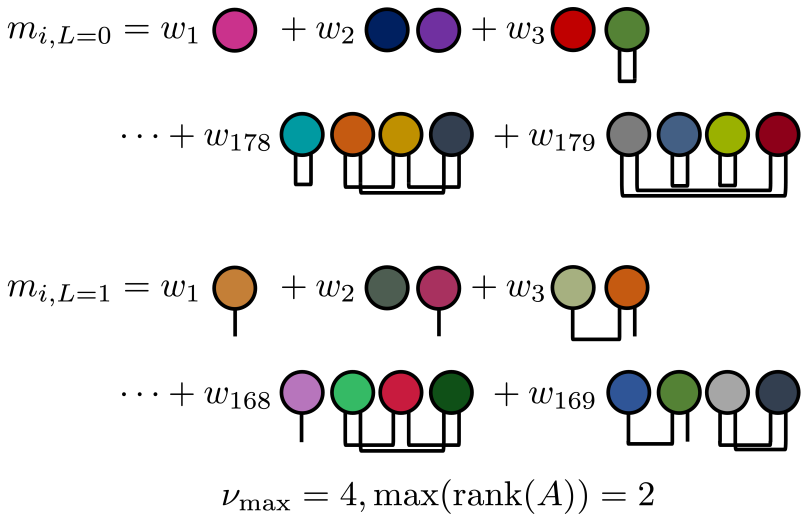


Figure 13: Message creation by a weighted sum of all **B**'s of the desired rank. Here we have two examples for $\nu_{\max} = 4$ and maximum atomic basis rank $= 2$. Production of scalar and vector messages respectively. **B** with coefficient w_{169} is produced in Figure 11

3.5 Updating Features

Features are then updated via Equation 20 in which the channels of the messages are mixed and are added together with the current features whose channels have also been mixed, this is called a residual connection [25] which works under the basic principle that to prevent vanishing gradient we should use 'skip' connections between layers.

$$h_{i,kL}^{(t+1)} = \texttt{mix}(m_{i,k,L}^{(t)}) + \texttt{mix}(h_{i,k,L}^{(t)}) \tag{20}$$

4 Experiments

This project's aim was to develop the theoretical landscape of equivariant GNN architectures. For this reason, synthetic tests of *expressivity* were chosen over performance benchmarking as this would better show the contributions of the CMACE architecture. Expressivity is a measure of the range and complexity of functions that an architecture can approximate. For example, the linear regression model $y = \alpha x + \beta$ is not expressive enough to approximate the function $y = x^2$. However, note that expressivity is a subjective measure highly dependent on the kind of task you are measuring against. For example, an architecture that is extremely capable in one domain but cannot complete a trivial task in another. Therefore choosing the right measures of expressivity is of the utmost importance.

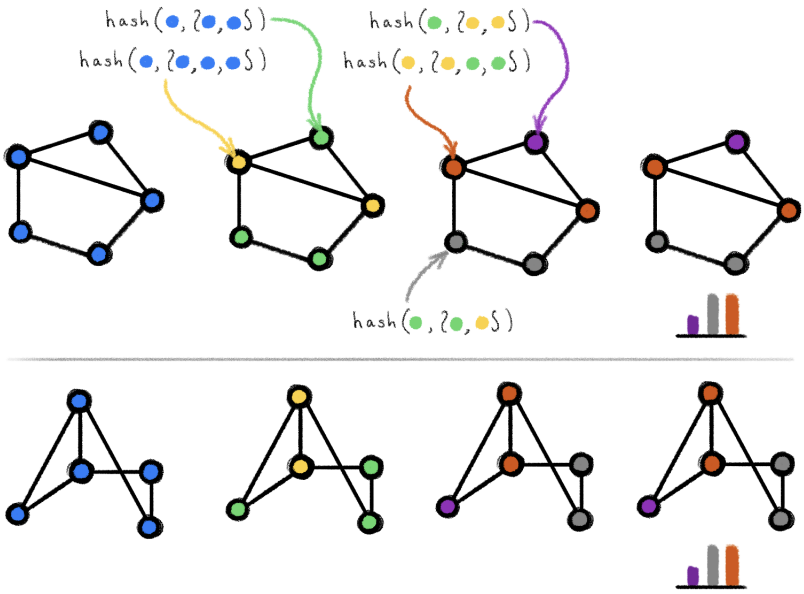


Figure 14: WL test for two graphs, which here, are isomorphic and thus have the same proportions of colours at the end [26]

When thinking about such tests for equivariant GNNs we ought to look at non-trivial problems on graphs - one classical example is a test for graph isomorphism. This is an enticing property for GNNs to have - if a graph doesn't have this property it will represent non-identical graphs in the same way which leaves us little place to classify aspects of them differently. The algorithm to solve this problem [27] is the Weisfeiler-Lehman (WL) test [28]. To see how the WL works, it is easiest to look at the example in Figure 14.

Initially, all nodes are given the same 'colour'. The current colour of a node and the multiset created by aggregating the colours of neighbouring nodes is used as an input to an one-to-one hash function which provides a new colour. Then repeat the process until the colouring is stable. If the stable colouring of two graphs is different then the graphs are not isomorphic. If the colouring of two graphs is the same they have met a necessary (but not sufficient) condition of isomorphism. The WL test is very reminiscent of message passing as the central node aggregates features of its neighbours, except in this process the propagation of information is lossless due to the fact that the hash function is one-to-one. With this perspective as the WL test as a perfect GNN testing for isomorphism, it is unsurprising that it has been shown that GNNs are at best as good the WL for isomorphism testing [29].

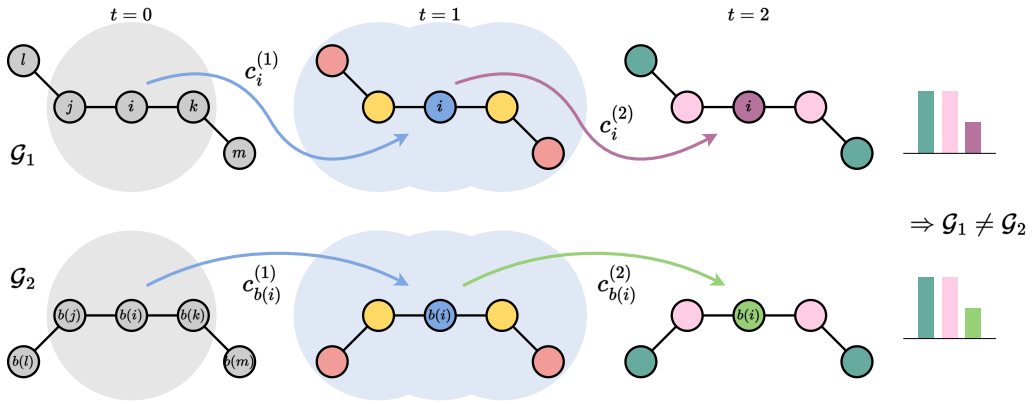
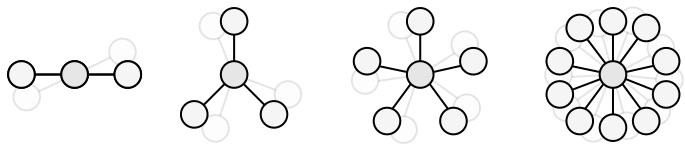


Figure 15: Geometric WL test for two graphs that are non-isomorphic once geometry is considered. Here, geometric information about the angles on the ends to the middle, once these equivariants 'meet' in the middle, the two graphs can be seen to be different by the GWL test [21]. This new found way of telling graphs apart motivates GWL

For Geometric GNNs the notion of isomorphism is stricter as the relative positions also have to be the same in addition to the graph structure. This required the creation of the Geometric WL (GWL) test [21] which like its analogue is shown as the upper bound for geometric graph isomorphism testing. Therefore, it is worth using the GWL as a baseline measure of expressivity. The action of GWL is shown in Figure 15 and will be used in the second of our experiments. Both experiments are from the testing environment in the `geometric-gnn-dojo`[3].

[3]Code available on GitHub: `https://github.com/chaitjo/geometric-gnn-dojo`

4.1 Rotational symmetry



	GNN Layer	Rotational symmetry 2 fold	3 fold	5 fold	10 fold
Cartesian	E-GNN$_{L=1}$	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
	GVP-GNN$_{L=1}$	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
	CMACE$_{L=1}$	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
	CMACE$_{L=2}$	**100.0 ± 0.0**	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
	CMACE$_{L=3}$	**100.0 ± 0.0**	**100.0 ± 0.0**	50.0 ± 0.0	50.0 ± 0.0
	CMACE$_{L=5}$	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	50.0 ± 0.0
	CMACE$_{L=10}$	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**
Spherical	TFN/MACE$_{L=1}$	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
	TFN/MACE$_{L=2}$	**100.0 ± 0.0**	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
	TFN/MACE$_{L=3}$	**100.0 ± 0.0**	**100.0 ± 0.0**	50.0 ± 0.0	50.0 ± 0.0
	TFN/MACE$_{L=5}$	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	50.0 ± 0.0
	TFN/MACE$_{L=10}$	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**

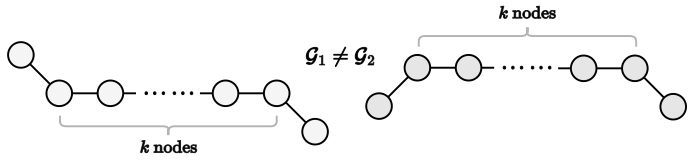
Table 1: *Rotationally symmetric structures.* Here expressive power is probed via training single layers of equivariant GNNs to tell apart an L-fold symmetric with have been rotated about the axis of symmetry. Previous work has shown that layers using tensors of rank-L are unable to differentiate between the rotated versions of graphs with greater than L-fold symmetry [21]. This experiment shows **CMACE is as powerful as MACE at telling apart rotationally symmetric environments**. Data from non-CMACE models and table adapted from Joshi et al. 2023 [21]. Anomolous results are marked in red and expected results in green.

This first task investigates how well different models could detect differences in rotationally symmetric environments that are orientated differently. This is useful as neighbourhoods like this appear very often in the material sciences in periodic structure.

Experiment. First, a graph with L-fold symmetry is created (a central node with L equally space spokes), and a second graph is created by rotating the first graph at an angle, $0 < \theta < \frac{2\pi}{L}$ about the central node (see examples in Figure on top of Table 1). Both graphs are then fed into the model, if it can tell the two graphs apart (i.e. has two different outputs) then the graph is expressive enough to detect this L-fold symmetry. The test is run on pairs of graphs with $L = 2, 3, 5, 10$.

Results. It has been previously shown empirically, a model using tensors with a maximum rank L is unable to tell apart the graph pairs described above for rotational symmetries greater than L-fold [21]. As expected, CMACE models with maximum rank tensors of varying L exhibited the same behaviour as TFN/MACE becoming the first Cartesian model to have the expressive power to tell apart these kinds of neighbourhoods.

4.2 k-chains



	($k = 4$-chains)	Number of layers				
	GNN Layer	$\lfloor \frac{k}{2} \rfloor$	$\lfloor \frac{k}{2} \rfloor + 1 = \mathbf{3}$	$\lfloor \frac{k}{2} \rfloor + 2$	$\lfloor \frac{k}{2} \rfloor + 3$	$\lfloor \frac{k}{2} \rfloor + 4$
Inv.	IGWL	50%	50%	50%	50%	50%
	SchNet	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00
	DimeNet	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00
Equiv.	GWL	50%	**100%**	**100%**	**100%**	**100%**
	E-GNN	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	**100.0 ± 0.0**
	GVP-GNN	50.0 ± 0.0	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**
	TFN	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	**80.0 ± 24.5**	**85.0 ± 22.9**
	MACE	50.0 ± 0.0	**90.0 ± 20.0**	**90.0 ± 20.0**	**95.0 ± 15.0**	**95.0 ± 15.0**
	CMACE	50.0 ± 0.0	**95.0 ± 15.0**	**90.0 ± 20.0**	**90.0 ± 20.0**	**80.0 ± 24.5**

Table 2: *k-chain geometric graphs.* This experiment investigates **how well different architectures propagate geometric information**, where the GWL test is an upper bound, propagating information perfectly. According to the GWL test, the k-chains graphs are $(\lfloor \frac{k}{2} \rfloor + 1)$-layer distinguishable, as it takes this long for the geometric information from both ends to meet in the middle. We train models with a varying number of layers for $k = 4$ and find that **like MACE, CMACE can faithfully propagate geometric information most of the time**, but sometimes struggles with bottle-necking. Data from non-CMACE models and table adapted from Joshi et al. 2023 [21].

Experiment. The k-chains experiment is a generalisation of previous analysis of equivariant models [30]. The aim is to investigate how well (if at all) architectures propagate geometric information across the graph during multiple message-passing steps (i.e. multiple layers). As discussed earlier GWL propagates this information perfectly, without loss therefore this experiment is a test of how close our model is to geometric information propagation. The synthetic data was made from graphs that have a straight chain of k nodes and then have two nodes on each end that can either point at an angle $\frac{\pi}{4}$ upwards or downwards. For one graph, both end nodes point downwards and in the other one points down, and the other points up (see schematic at top of Table 2 and Figure 15).

Figure 15 shows GWL (i.e. perfect propagation) on a pair of graphs for $k = 3$. After one layer ($t = 1$), both have the same colour, therefore these graphs appear the same after one layer. Whereas, after two layers, due the geometric from both ends mixing the middle nodes node gives a different colour for the two different graphs. After each layer, the information from a node is taken into account for the colour of the neighbouring nodes in the next layer, only when the information about the end nodes mixes in the middle do we get different results as they ends have different relative values in graph 1 vs graph 2. Therefore, at best, it takes $\lfloor \frac{k}{2} \rfloor + 1$ layers (floor function to account for even and odd k) for the geometric information about the two end nodes to mix causing the model to achieve different values.

Results. We used $k = 4$ to match the implementation of the paper where this test first appeared. As expected, when using fewer than $\lfloor \frac{k}{2} \rfloor + 1$ layers, our model couldn't tell apart the two graphs at all. When using more than the required number of layers, the performance shows that most of the time it was possible to tell them apart although

sometimes our model could not, this shows how our model is not a perfect GNN i.e. GWL performance is 100% when greater than minimum number of layers. This pathology is caused by bottle-necking, i.e. the bandwidth of the single path to the central node is not big enough to faithfully propagate the geometric information. These results were only comparable to MACE showing we have the same property of good but sometimes limited propagation of geometric information.

5 Computational Design

5.1 Code structure

The code[4] was built using the `pytorch` package in python which allows for the building and training of deep learning modules/architectures. I initially looked at using the `torch_geometric.nn.MessagePassing` class which has message passing functionality built-in but eventually decided to build a bespoke architecture using instances of the ubiquitous `torch.nn.Module` class. `torch_geometric` was used to store graph structured data in the `Data` class.

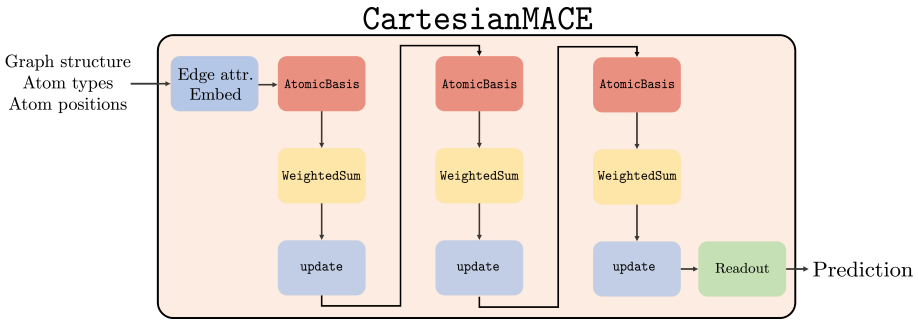


Figure 16: A CMACE model with 3 layers. To find initial features, embeddings are found using the atom type. Each layer is made from three modules: `AtomicBasis`, `WeightedSum` and `update` and outputs an updated set of features which can be the input for the next layer. After all the layers we use a readout function to get scalar output of model.

The model was assembled using three main classes. The `CartesianContraction` class carries out the exact same function as the Contraction operator (see Figure 7) and abstracts away all of nuance of contractions so that the code for the other classes is much more readable. The `AtomicBasis` class completes the operations of Equations 13 and 15 by taking in the features and edge attributes derived off relative positions, produces the particle basis states which are summed to yield the atomic basis. The `WeightedSum` class carries out Equations 16, 17 and 19 by taking in the set of atomic basis tensors for each node, calculates all the splits and their contractions, then creates a linear sum of all the paths to a certain rank, hereby forming the message as in Figure 13. The updating of Equation 20 takes place as a class method. These three parts are repeated in that order once per layer as seen in 16 which represents an instance of three layers. The final scalar feature are used as input to a readout function which in the case of our experiment was a `torch.nn.Linear` layer.

[4]Code available on GitHub: `https://github.com/hws1302/partIII-project/tree/main/cartesian_mace`

5.2 Optimisation

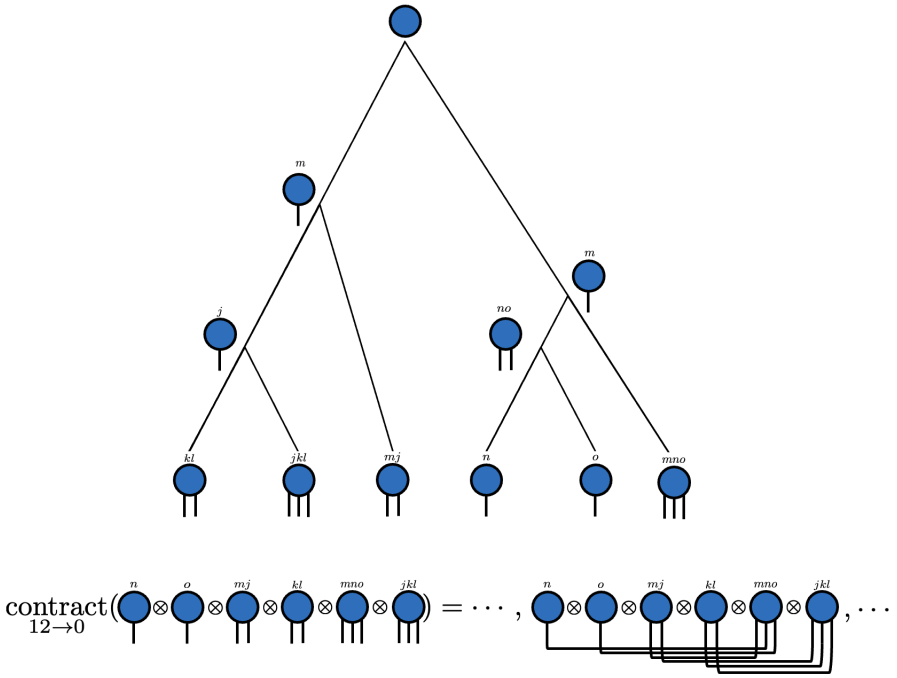


Figure 17: bottom: one contraction combination of a tensor product of rank-12 to a scalar. top: the optimal order to contract in, this optimises both compute and memory requirements.

The order in which the indices of a tensor are contracted doesn't affect the output values but makes a large difference to the computational cost. CMACE uses the `opt_einsum` package to calculate near-optimal contraction paths, like the one seen in Figure 17. As finding paths is also computationally non-trivial the path saving feature of `opt_einsum` is used. Table 3 shows that saving paths $> 5\times$ quicker vs. built-in PyTorch solution (find path at runtime). For a full discussion see Appendix A.4.

Contraction method	**Time /ms**	**Speed-up vs. naïve**
Theoretical	-	31.0
Saved path (`opt_einsum`)	0.34 ± 0.02	26.7
Find path at runtime (`opt_einsum`)	1.79 ± 0.03	5.3
Naive path (`numpy`)	9.52 ± 0.12	1.0

Table 3: The runtimes of different tensor contraction methods compared to the theoretical speed up of optimal path by `opt_einsum`. This shows us that **saving paths is >5** $\times quicker than calculating at runtime$

6 Further Work

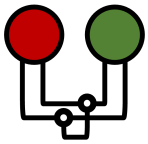
Benchmarking. This project sets the theoretical framework, therefore the next step is to optimise the model for benchmarking to prove real world usefulness. Benchmarking results

of CMACE should be impressive as it matches all the theoretical properties of MACE which provided state-of-the-art results. The main optimisation problems are as follows:

- Add in layer normalisation to ensure that values to not grow arbitrarily large after many layers
- Add in regularisation techniques such as penalising the size of parameters to prevent overfitting
- Tune the hyperparameters such as the number of layers, number of channels, maximum tensor rank via grid search
- Find computational improvements. For example, by compiling the model using PyTorch `TorchScript` framework

After these improvements are done, benchmarking against popular molecular benchmarks such as QM9 and MD17 can be used to gauge model performance.

Anti-symmetric Contractions. Currently in CMACE, all contractions are symmetric. This is a desired property when measuring physical properties of a single molecule such as energy which should not change on inversion as CMACE produces scalars with even parity which don't change on inversion. Although in multi-molecule applications it is useful to have pseudoscalars which flip sign on inversion. This is becuase, molecules with chiral carbons have two versions which are non-superimposable mirror images. For example, one version of limonene smells like lemons whereas its mirror image smells of pine trees illustrating how they differently interact with smell receptors. Therefore in this multi-molecule case is desirable to have pseudoscalars that have odd parity under inversion such that the model can vary its outputs depending on which mirror images are used.



$$= \sum_{i,j,l,m,k} A_{ij}\epsilon_{ijk}B_{lmk}\epsilon_{lmk}$$

Figure 18: Anti-symmetric contraction using the Levi-Cevita tensor - denoted by small white circle. Allows for the production of odd parity pseudoscalars which are important for representing interactions between multiple molecules with chirality

This requires the use of a 'dual' set of features that are contracted anti-symmetrically using the Levi-Civita tensor ϵ_{ijk}. Figure 18 shows how these anti-symmetric contractions can be represented in tensor network notation.

7 Conclusion

It is non-trivial and useful step forward to have CMACE in the literature as the first many-body equivariant model with Cartesian tensors of arbitrary rank. This project has produced the code for CMACE neatly split up into discrete modules and unlike MACE does not use any esoteric python packages allowing for anyone with a grasp of PyTorch to get a model up and running very quickly. This simplicity also allows for the easy addition novel modifications to CMACE that have not yet been thought of. The experiments have

shown the same theoretical properties as MACE in the k-chains and rotational symmetry experiments suggesting strong expressive power. Though expressivity is just one side of the story, there is future to be work done in optimising this model for benchmarking.

References

[1] P. Hohenberg and W. Kohn, "Inhomogeneous electron gas," *Physical review*, vol. 136, no. 3B, p. B864, 1964.

[2] W. Kohn and L. J. Sham, "Self-consistent equations including exchange and correlation effects," *Physical review*, vol. 140, no. 4A, p. A1133, 1965.

[3] E. W. Weisstein, "Moore-penrose matrix inverse," *https://mathworld. wolfram. com/*, 2002.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[5] J. Behler, "Atom-centered symmetry functions for constructing high-dimensional neural network potentials," *The Journal of chemical physics*, vol. 134, no. 7, p. 074106, 2011.

[6] A. V. Shapeev, "Moment tensor potentials: A class of systematically improvable interatomic potentials," *Multiscale Modeling & Simulation*, vol. 14, no. 3, pp. 1153–1173, 2016.

[7] M. A. Caro, "Optimizing many-body atomic descriptors for enhanced computational performance of machine learning based interatomic potentials," *Physical Review B*, vol. 100, no. 2, p. 024112, 2019.

[8] M. Uhrin, "Through the eyes of a descriptor: constructing complete, invertible descriptions of atomic environments," *Physical Review B*, vol. 104, no. 14, p. 144110, 2021.

[9] R. Drautz, "Atomic cluster expansion for accurate and transferable interatomic potentials," *Physical Review B*, vol. 99, no. 1, p. 014104, 2019.

[10] G. Dusson, M. Bachmayr, G. Csányi, R. Drautz, S. Etter, C. van der Oord, and C. Ortner, "Atomic cluster expansion: Completeness, efficiency and stability," *Journal of Computational Physics*, vol. 454, p. 110946, 2022.

[11] D. P. Kovács, C. v. d. Oord, J. Kucera, A. E. Allen, D. J. Cole, C. Ortner, and G. Csányi, "Linear atomic cluster expansion force fields for organic molecules: beyond rmse," *Journal of chemical theory and computation*, vol. 17, no. 12, pp. 7696–7711, 2021.

[12] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, pp. 1263–1272, PMLR, 2017.

[13] K. Schütt, P.-J. Kindermans, H. E. Sauceda Felix, S. Chmiela, A. Tkatchenko, and K.-R. Müller, "Schnet: A continuous-filter convolutional neural network for modeling quantum interactions," *Advances in neural information processing systems*, vol. 30, 2017.

[14] J. Gasteiger, J. Groß, and S. Günnemann, "Directional message passing for molecular graphs," *arXiv preprint arXiv:2003.03123*, 2020.

[15] Y. Liu, L. Wang, M. Liu, X. Zhang, B. Oztekin, and S. Ji, "Spherical message passing for 3d graph networks," *arXiv preprint arXiv:2102.05013*, 2021.

[16] J. Gasteiger, F. Becker, and S. Günnemann, "Gemnet: Universal directional graph neural networks for molecules," *Advances in Neural Information Processing Systems*, vol. 34, pp. 6790–6802, 2021.

[17] Hope College Department of Chemistry, "WebMO Help: Adjust Tool," 2023. [Online; accessed 15-May-2023].

[18] N. Thomas, T. Smidt, S. Kearnes, L. Yang, L. Li, K. Kohlhoff, and P. Riley, "Tensor field networks: Rotation-and translation-equivariant neural networks for 3d point clouds," *arXiv preprint arXiv:1802.08219*, 2018.

[19] B. Anderson, T. S. Hy, and R. Kondor, "Cormorant: Covariant molecular neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[20] V. G. Satorras, E. Hoogeboom, and M. Welling, "E (n) equivariant graph neural networks," in *International conference on machine learning*, pp. 9323–9332, PMLR, 2021.

[21] C. K. Joshi, C. Bodnar, S. V. Mathis, T. Cohen, and P. Liò, "On the expressive power of geometric graph neural networks," *arXiv preprint arXiv:2301.09308*, 2023.

[22] I. Batatia, D. P. Kovacs, G. Simm, C. Ortner, and G. Csányi, "Mace: Higher order equivariant message passing neural networks for fast and accurate force fields," *Advances in Neural Information Processing Systems*, vol. 35, pp. 11423–11436, 2022.

[23] P. Veličković, "Message passing all the way up," *arXiv preprint arXiv:2202.11097*, 2022.

[24] T. Smidt, "Tutorial 1: Euclidean symmetry in machine learning for materials science." Online, 2021.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[26] M. Bronstein, "How powerful are graph neural networks? expressive power of graph neural networks and the weisfeiler-lehman test," *Towards Data Science*, June 2020. Accessed: May 10, 2023.

[27] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *Journal of graph theory*, vol. 1, no. 4, pp. 339–363, 1977.

[28] B. Weisfeiler and A. Leman, "The reduction of a graph to canonical form and the algebra which appears therein," *nti, Series*, vol. 2, no. 9, pp. 12–16, 1968.

[29] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[30] K. Schütt, O. Unke, and M. Gastegger, "Equivariant message passing for the prediction of tensorial properties and molecular spectra," in *International Conference on Machine Learning*, pp. 9377–9388, PMLR, 2021.

[31] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM*, vol. 38, pp. 963–981, jan 2008.

A Appendix

A.1 Tensor Shapes

Table 4 gives the shapes for the tensors used in the CMACE process. The `tensor_shape` function is used as the Cartesian tensors require new dimensions as the rank increases (not the case for spherical tensors which are represented by $2\ell + 1$ vectors). For example, $\texttt{tensor\_shape}(\ell = 2) = [3, 3]$, therefore, $h_{i,k\ell_1=2}$ has shape $[\texttt{n\_nodes}, \texttt{n\_channels}, 3, 3]$.

Note that due to these strange shapes of Cartesian tensors, we could not just stack our features in a list like MACE. Instead we stack tensors of different rank into a list i.e. $[h_{i,k\ell_1=0}, h_{i,k\ell_1=1}, h_{i,k\ell_1=2}]$.

Tensor	Shapes	Equation(s)
$h^{(t)}_{i,k\ell_1}$	$[\texttt{n\_nodes}, \texttt{n\_channels}, \texttt{tensor\_shape}(\ell_1)]$	(12)
$\hat{r}_{ij}^{\otimes \ell_2}$	$[\texttt{n\_nodes}, \texttt{n\_channels}, \texttt{tensor\_shape}(\ell_2)]$	(12)
$R_{k\ell_1\ell_2\ell_3}(\lvert r_{ij}\rvert)$	$[\texttt{n\_edges}, \texttt{n\_channels}]$	(13, 14)
$\phi^{(t)}_{ij,k,\ell_3}$	$[2 \times \texttt{n\_edges}, \texttt{n\_channels}, \texttt{tensor\_shape}(\ell_3)]$	(13)
$A^{(t)}_{i,k,\ell_3}$	$[\texttt{n\_nodes}, \texttt{n\_channels}, \texttt{tensor\_shape}(\ell_3)]$	(15, 16)
$\mathbf{B}^{(t)}{}_{i,k,L,\eta_\nu}$	$[\texttt{n\_nodes}, \texttt{n\_channels}, \texttt{tensor\_shape}(L)]$	(17, 19)
$m^{(t)}_{i,k,L}$	$[\texttt{n\_nodes}, \texttt{n\_channels}, \texttt{tensor\_shape}(L)]$	(19, 20)

Table 4: Table adapted from Batatia et al. 2022 [22]

A.2 File Structure

The file structure of the project. The model was stored in the `models/model.py` file which puts together the blocks from the `modules` directory. Experiments were carried out in jupyter notebook in the `experiments` directory.

```
cartesian_mace
├── experiments
│   ├── geometric_gnn_101.ipynb
│   ├── issues.ipynb
│   ├── profile_model.ipynb
│   ├── rotsym.ipynb
│   ├── speed_tests.ipynb
│   ├── tensor-contractions.ipynb
│   └── test_path_saving.ipynb
├── models
│   └── model.py
└── modules
    ├── atomic_basis_block.py
    ├── tensor_contraction_block.py
    └── weighted_sum_block.py
```

```
├── test
│   ├── test_cotraction.ipynb
│   └── test_equivariance.ipynb
└── utils
    └── cartesian_contractions.py
```

A.3 Different ways to contract

A tensor of rank-n has n different indices e.g. scalars are rank-0 (s), vectors are rank-1 (v_i) and matrices are rank-2 (M_{ij}). It is possible to *contract* over a pair of indices setting two indices to the same letter and summing over them. For example, contracting a matrix is equivalent to finding the trace, $\sum_i M_{ii}$. Contractions will decrease the rank of a tensor by 2.

For our problem, it is useful to know how many ways a tensor can be contracted. If a tensor is of rank n (i.e. n indices) there are n choose 2, $\binom{n}{2}$ ways of doing this. Taking a second contraction, there are now $\binom{n-2}{2}$ ways. Therefore, for m contractions, we find the following formula:

$$|\text{contract}(n, m)| = \binom{n}{2} \cdot \binom{n-2}{2} \cdots \binom{n-2[m-1]}{2} \tag{21}$$

$$= \frac{n!}{(n-2)!2!} \cdot \frac{(n-2)!}{(n-4)!2!} \cdots \frac{n!}{(n-2[k-1])!2!} \tag{22}$$

$$= \frac{n!}{(n-2m)!m!(2!)^m} \tag{23}$$

This formula was used to check that the contract operator class was producing the correct amount of contractions.

A.4 Path finding and saving

Finding a path.
When contracting a tensor (by summing over the repeated indices), the resulting tensor is not dependent on the chosen order to sum over the indices, due to the associative nature of this operation. For example, to carry out the contraction in figure 6 $A_{ij}B_{ij}$ it is possible to sum over the i's before or after summing over the j's. Although paths do not affect the calculated result both the storage and compute are extremely sensitive to the path chosen to carry out a set of contractions [31]. Therefore, it is essential to find a (near) optimal path to limit unnecessary computation. The only problem is that path finding is also a highly non-trivial problem, to be certain you get the correct path you must check all permutations of the path i.e. O(N!). Therefore packages such as `opt_einsum` have been created to find near-optimal paths at a fraction of the computational cost. For example, `Dynamic Programming` or the `Greedy` algorithm are used by `opt_einsum` in this situation that suit it best.

Reusing a path.
In the *CMACE* architecture, the same tensor contractions occur every time a forward pass of the network is carried out, therefore it made sense to take advantage of the `opt_einsum`

path-saving functionality via the `ContractExpression` class. One instance of this class was needed for each contraction and was saved to a 'buffer' dictionary accessible with a unique key associated with that contraction. This allowed for a near-optimal path to be used every time without having to recalculate what this path way.

Example. To test the speeds of different path choosing methods we use the example of one of the possible contractions of 6 pairs of repeated indices across 6 tensors into a scalar (see Figure 17b). This contraction was implemented for a spatial dimension of 3 and a batch dimension of 1000. For this contraction, `opt_einsum`'s predictor estimated that the near-optimal path would be 31x quicker than the naïve implementation. We then tested this out in practice using 1) an instance of the class with the path saved ahead of time (path graphically shown in Figure 17) 2) `torch.einsum` which does not save paths but uses `opt_einsum` to calculate paths at runtime 3) `numpy.einsum` as the naïve implementation that doesn't use any optimisation. The results, listed in Table 3, show that finding an optimal path is much quicker than the naïve implementation and saving paths is $> 5\times$ quicker than calculating then at runtime. This shows that, given the majority of the models compute goes on contractions, we should certainly save paths and this is a useful optimisation to make.

A.5 Testing

Given this bespoke architecture was built from scratch, testing was important to ensure that our model and it's modules were behaving as expected. This was done through notebooks on which tested specific parts of the architecture. In future, this would be improved by using a proper testing framework such as `unittest`

Equivariance testing. Given many of the computational and theoretical benefits of our model are related to its equivariance, this was a very important feature to test. As discussed earlier, equivariance is the idea that we can either rotate our inputs or rotate our outputs but we will get the same result i.e. $D(Q) \cdot f(x) = f(D(Q) \cdot x)$. The `transform_tensors()` function detected the rank, n, of the input tensors and then transformed it via n orthogonal rotation matrices, this could be done for input or output tensors. We then checked the outputs were the same for rotating before or after. The errors seen for the both the model as a whole and the moduels tested individually were $\sim 10^{-8}$ which is of similar magnitude of the off-diagonal element of QQ^T where Q is an orthogonal matrix.

Tensor contractions. Another area of utmost importance was that of contraction calculation. This testing mainly occured on the `CartesianContraction` class as we checked that the numbers we got matched our analytic expression. We also checked that `AtomicBasis` and `WeightedSum` get the correct number of contractions especially when they came from different splits and in different numbers.

(lab book for project can be found in `zip` file)