Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Group Equivariant Neural Networks for Games

Oisin Carroll
[redacted]

April 30, 2020

A Dissertation submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Supervised by Dr. Joeran Beel

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Games such as go, chess and checkers have multiple equivalent game states. That is, multiple board positions where symmetrical and opposite moves should be made. These equivalences are not exploited by current state of the art neural agents which instead must relearn similar information, wasting computing time. Group equivariant CNNs in existing work create networks which can exploit symmetries to improve learning, however, they lack the expressiveness to correctly reflect the move embeddings necessary for games.

I introduce two methods for creating agents with an innate understanding of these board positions; Game Graph Convolutional Networks (GGCNs) and Finite Group Neural Networks (FGNNs). These are shown to improve the performance of networks playing checkers (draughts), and can be easily adapted to other games.

Additionally, FGNNs can be created from existing network architectures. This includes, for the first time, those with skip connections and arbitrary layer types. I demonstrate that an equivariant version of U-Net (FGNN-U-Net) outperforms the unmodified network in image segmentation.

Acknowledgements

Thanks to my supervisor; Joeran, for his advice and especially for his guidance in writing this document and the subsequent paper. Thanks to Lucy Deacon for helping break my head with some math concepts used in this and that will hopefully tie into future work. Thanks to Alec Barber and Eleanor Windle for their help proof-reading and suggesting improvements.

Thanks also to the Adapt Centre for allowing me access their HPC for training networks, and the dev-ops team; Graziano and others who kindly helped me with technical problems.

Finally, I'd like to extend a special mention to Millie, my dog, who kept me company during long nights of coding, debugging and writing.

Contents

List of Figures

1 Introduction

The leading computer algorithms for playing many board games are deep-learning based. Google's DeepMind created the first Go program able to beat a world champion; AlphaGo [1] and their subsequent papers discuss AlphaZero; a generic algorithm that was trained to become a top go, shogi or chess engine [2]. LeelaChessZero — a community lead effort to replicate AlphaZero for chess (based on LeelaZero, a go program) recently won the computer chess world championship[1]. The primary innovations in these papers are the methods used to train the networks stably through self-play; loss functions, genomes and randomness, as well as a much improved weighted Monte Carlo tree search.

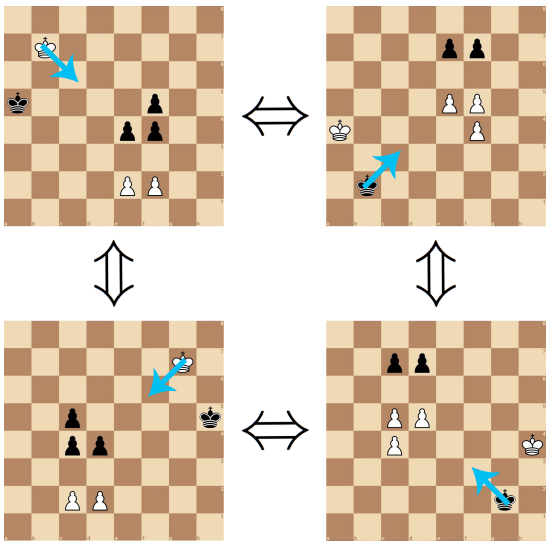


Figure 1.1: Chess positions with white to play upwards (left) and black to play downwards (right). These are all equivalent through reflecting left-to-right and/or swapping colours, and the only drawing move is shown in blue on all boards.

The neural network architectures used at the core of these algorithms are similar however; variations on standard Convolutional Neural Networks (CNNs). They aren't able to understand symmetries of the game rules or board and because of this, relearn similar information multiple times. For example, solving for the best move in any of the 4 boards shown in Fig. 1.1 is an equivalent problem; knowing the move in any one means you should play the symmetrically-equivalent move in the rest. We assert a better neural network architecture would be able to treat these 4 positions as the same, and would never play different moves on equivalent boards.

[1] https://www.chess.com/news/view/lc0-wins-computer-chess-championship-makes-history

1.1 Research Problem

Networks which treat these equivalent positions as the same may reduce overfitting and improve training times as each position must be learned only once, no matter the orientation it appears on the board.

For some network predicting a move from a board-state $N : X \to Y$ and an operation g which reflects the board and g' which reflects a move, $N(gx) = g'N(x)$ for all $x \in X$. This property is *equivariance*; some transformation of the input leads to an equivalent transformation of the output. If g' is the identity function it is *invariance*; where the output is unaffected by some transformation of the input.

Neural networks which are invariant or equivariant to some property of their input is a rapidly expanding field, with many influential works being published in recent years. These include invariances over sets [3, 4, 5, 6]; where the order of the input sequence is ignored, graphs [6, 7, 8, 9]; where perturbations of the input nodes and edges are ignored, and spatial equivariance to a variety of group operations. These are generally CNN-based and can be divided into networks which are equivariant over finite groups such as rotations of 90° [10, 11, 12, 13, 14, 15] and methods which are approximately equivariant over continuous groups, such as affine transformations [16, 17, 18, 19, 20, 21].

Of these, the most relevant are the CNN architectures over finite groups. Many of these are applicable to equivariances over of rotations or reflections of the input. Games however require a different kind of equivariance. Reflecting the input board state should result in the network predicting the symmetrically opposite move which, in many cases, is not the same as reflecting the output tensor. *AlphaZero* [2] for chess encodes moves in the output where each layer corresponds to a specific direction (North/North-East/East/...) and distance to move the piece. Reflecting this move tensor won't correctly reverse the move direction. The research into group-equivariant CNNs cannot easily be extended to handle this case.

Additionally, these methods are not applicable to neural network architectures with skip connections; an indispensable component of many popular neural network architectures [22, 23, 24, 25, 26].

1.2 Research Question

Can neural networks for games which can 'understand' the equivalence of different board positions perform better than those which don't?

1.3 Research Goal

This dissertation aims to investigate the existence of an equivariant deep-learning neural network architecture for games — that is, one which would treat all board positions in Fig. 1.1 as the same. We hypothesize that this may reduce training times, improve performance, and/or reduce model sizes when compared existing architectures.

1.4 Contributions

I propose two novel methods for creating equivariant networks for learning board games. The first; *Game Graph Convolutional Networks (GGCNs)* uses a network architecture equivariant over equivalent graphs, and the second; *Finite Group equivariant Neural Networks (FGNNs)* are a general method for creating equivariances in arbitrary existing neural network architectures.

GGCNs use *Graph Convolutional Networks*, and define a novel method of representing and reasoning about the board state as a graph. Since equivalent board states form isomorphic graphs, the method is equivariant over equivalent boards.

FGNNs can be used to derive equivariant versions of existing neural network architectures including, for the first time, those with skip connections and arbitrary layer types. The derivations are arguably easier to reason about than prior work, and this allows for easier extension of this method. FGNNs can be viewed as a generalization of the work of Dieleman et Al. [11] to arbitrary groups and with an additional proof for skip connections, although the approach differs.

For each of the network architectures proposed, I verify they create equivariant networks in theory and practice. Next I demonstrate that FGNNs reduce overfitting and improve performance when compared to equivalent networks without this equivariance. This work uses checkers, but the methods presented are general enough to be used in a variety of board games including chess, go and shogi.

I conclude with an FGNN-based implementation of a popular architecture; U-Net [26] over several finite groups, and demonstrate that these can demonstrate equivalent performance with 4-8 times fewer weights as compared to the baseline implementation.

2 Background

Convolutional Neural Networks (CNNs) are widely used for many kinds of machine learning, such as image classification, segmentation, and games since their resurgence in 2012 with AlexNet [27]. These networks are trained on a large sample set, and subsequently predict on unseen examples.

As introduced previously, this dissertation considers networks which are 'unaffected' by some transformation of the input. These transformations may be rotations, reflections, permutations, or others. It is useful to talk about these transformations in the context of *Group Theory*. Group Theory is, as the name implies, the study of algebraic structures known as Groups. It allows us to derive general expressions over 'types' of transformation.

A group consists of a set of *elements*, and some binary operation acting upon them. We primarily consider Transformation Groups, where elements may represent 'actions' such as a rotation of 90° or a permutation of (1,2,3). The group operation then is composition, where we simply perform the actions in succession. Equivalent actions, such as doing nothing (the identity element) and rotating 360° correspond to the same element in the group. A group must also be *closed*, meaning that the composition of any two elements in a group must also be a member of the group.

Specific groups referred to in this work are:

Dihedral groups (D_{2n}) The D_{2n} group describes the symmetries of a polyhedra with n sides. For example, D_8 ("The dihedral group of order 8"), describes the symmetries of the square; the elements are all rotations of 90° and reflections that map a square unto itself.

Lie groups ($SE(n)$) Groups which describe isometries of n dimensional euclidian space. These include rotations and translations, and preserve distance between points.

Affine group This group consists of all affine transformations. It is effectively the $SE(2)$ group with additional scaling and skewing.

Typically we use G to denote a group, and $g \in G$ to denote the group elements. A finite group is simply one with a finite number of elements, such as the dihedral groups.

The term **invariance** is used throughout the paper, loosely meaning 'unchanged by'. A function $f : X \to Y$ is invariant with respect to a group G iff:

$$f(g \cdot x) = f(x)$$

for all $x \in X$ and all $g \in G$

Similarly, a function $f : X \to Y$ is **equivariant** iff:

$$g \cdot f(x) = f(g \cdot x)$$

for all $x \in X$ and all $g \in G$

This is equivalent to saying that each $g \in G$ commutes with f, as $g \circ f = f \circ g$.

Approximate invariance or equivariance is where these properties hold approximately. For example, if they are empirically observed.

3 Related Work

Research into deep learning algorithms which have useful equivariances is broad and divides somewhat into different approaches, generally tailored to the type of problem. Here I provide a summary of influential works considering invariance or equivariance across a broad range of problems in deep learning, with focus on those most relevant to games.

3.1 Set Invariance

Many works in deep learning focus on data represented as ordered sequences, such as audio or text. For some problems, however, it is important to ignore permutation of the input and treat it as an unordered set.

For 3D data, often works quantize data into voxels - effectively 3D pixels, which allows researchers to closely mimic the approaches used for images. Qi et al. [3] present an alternate approach; *PointNet*, a network which directly consumes pointclouds. Pointclouds are simply sets of points, and are often generated by 3D scanners or structure-from-motion systems. A problem with feeding pointclouds into traditional networks is that the order of these points in memory changes the result, and it is difficult to not overfit to this order. *PointNet*'s predictions are robust to this the reordering of points, however its invariance is approximate and is primarily learned through training of an RNN on randomly perturbed input sequences.

Zaheer et al. [4] provide a neural architecture which is permutation invariant by definition. Their overall strategy; nicknamed *DeepSets*, enforces equality in weights of the neural network layers. In the simplest case, this means the number of nodes in each layer remains consistent and all 'horizontal' edges and all 'diagonal' edges are each equal to each other. This forms a *permutation-equivariant* layer which may be incorporated into other network architectures. Their results are competitive to many specialized algorithms in different domains, including text concept set retrieval (finding a missing word in a set), image tagging (suggesting descriptions for an input image), and set anomaly detection (detecting outliers). The full model for anomaly detection is permutation-equivariant, while this property is only approximate for other tested applications.

3.2 Graph Equivariance & Invariance

A similar problem to invariance or equivariance over set equality is graph-problems. Many problems are represented naturally as a graph, such as network routing, knowledge graphs, and CPU-register allocation. On disk these graphs may be stored as an adjacency matrix, which is changed by reordering the nodes. Algorithms consuming an adjacency matrix would

then need to learn this equivariance through learning. These learning algorithms for graphs have an equivariance to reordering of the graph nodes at an architecture level.

Graph Convolutional Networks (GCNs) are presented by Kipf and Welling and used for classification in graph structures [7]. They provide an efficient convolution for graphs with is similar to that of CNNs. Their equivariance properties can be seen in the layer-wise propagation rule for GCNs, defined as:

$$H^{(l+1)} = \sigma(D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}H^{(l)}W^{(l)})$$

Here, $\tilde{A} = A + I_N$ is the adjacency matrix of the graph G with added self-connections, $D_{ii} = \sum_j \tilde{A}i$ is a normalization term, and $W^{(l)}$ is the trainable weight matrix for layer l. $\sigma(\cdot)$ is an activation function such as ReLU.

The values for each node in the graph, $H_i^{(l)}$, in each layer are updated based on the sum of the nodes sharing an edge with node i, normalized by some factor. The weights, $W(l)$, applied to each node are the same, and the adjacent nodes remain the same under perturbation of the adjacency matrix. This means the architecture is equivariant to reorderings of the nodes and edges in the adjacency matrix.

The normalization used ($D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}$) is the core innovation of GCNs' spectral graph convolution. When calculating the values for node i, this allows the network to effectively normalize by both the number of edges of node i but also by node j if i and j share an edge.

Schlichtkrull et al. [28] present *Relational Graph Convolutional Networks (RGCNs)* which extend GCNs to add multiple edge types. This is used to model relational data, and each edge type has its own weight matrix.

Gilmer et al. [8] propose *Message Passing Neural Networks (MPNNs)*, an architecture which represents the graph structure literally and nodes send 'messages' along shared edges. Each update loop, nodes first pass messages and update their hidden state. The state of a node is updated as a function of the hidden state of that node and the sum of all incoming messages. Next, a readout function generates a feature vector for the whole network which is the output prediction. Due to this sum over incoming messages, the hidden state is order-invariant. However their best performing network uses *set2set* [5] as the readout function which, similar to *PointNet* [3], only learns an approximate equivariance to order.

The MPNN architecture is shown to generalize 8 popular graph deep-learning algorithms including GCNs [7]. MPNNs are originally used to solve quantum chemistry problems by representing atoms as nodes and chemical bonds as edges, and are later used by NeuroSat [29] to solve SAT problems, similarly embedding the problem as a graph. MPNNs can run for any number of iterations (update/propagate) until convergence; NeuroSat can solve harder problems than it was trained on simply by running for longer.

Maron et al. [9] consider graph learning algorithms specifically in the light of equivariant and invariance. They characterize the collection of linear invariant and equivariant layers that may be used in graph classification networks. They also provide orthogonal bases vectors for the spaces of these layers - allowing them to be parameterized without constraint. In their appendices they show that their model can represent any MPNN to arbitrary precision.

3.3 Equivariance to Spacial Transforms

Some works generalize or extend the 2D convolution operation typical in a CNN in such a way that the network gains an equivariance or invariance.

Gens et al. [16] propose *Deep Symmetry Networks*; a CNN variation which is approximately invariant to any given symmetry group. They focus especially on the affine transformation group, which consists of all translations, reflections, rotations, or skews of the input/output. Deep Symmetry Networks or *symnets* have parallels to a CNN; in a CNN each feature map (in each layer) is stored as a function $M : \mathbb{R}^2 \to R$, and the value at point $P \in \mathbb{R}^2$ of a feature map layer i is calculated from a weighted sum of points 'near' P in the previous layer. This is the convolution operation and it is due to this process that a CNN is equivariant to translations of the input.

Deep Symmetry Networks store feature maps in higher-dimensional space, where each point represents a unique transformation in the group, i.e. in D dimensional space where D depends on the group representation. Since a point in the translation group can be parameterized by $P \in \mathbb{R}^2$, $D = 2$, the implementation for this group is analagous to a CNN. However a transformation in the affine group can be expressed as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{1}$$

Which is parameterized by $\mathbb{R}^6$, so $D = 6$ for this representation of the affine group. Each feature map then is also a mapping $\mathbb{R}^D \to R$. Layers in a Deep Symmetry Network sample points in the 'k-neighbourhood' of the point in the previous layer, i.e. those points which are 'nearby' in this D-dimensional representation of the group. Storing these maps on disk is an issue, as even with 10 steps in each direction there are 10^6 points. Instead each feature map is evaluated at several control points, which are chosen to maximize the feature in symmetry space by Gauss-Neuton optimization.

3.3.1 Group Convolutions

Cohen and Welling [10] propose a method for creating CNNs which are equivariant to finite groups. These G-CNNs utilize a G-Convolution in place of the traditional one. They write this regular convolution as an operation taking a stack of feature maps $f : \mathbb{Z}^2 \to \mathbb{R}^{K^l}$, (where K^l is the number of features in layer l) and convolving it with a set of K^{l+1} filters $\psi^i : \mathbb{Z}^2 \to \mathbb{R}^{K^l}$.

$$[f * \psi](x) = \sum_{y \in \mathbb{Z}^2} \sum_{k=1}^{K^l} f_k(y) \psi_k(x - y)$$

Considering the first sum as a sum over elements of the translation group G, this can be rewritten as:

$$[f * \psi](x) = \sum_{h \in G} \sum_{k=1}^{K^l} f_k(y) \psi_k(hx)$$

This allows for a similar reasoning as before, as the network becomes equivariant to the group G. Subsequent proofs extend to arbitrary groups with the same convolution equation. Performance is improved by calculating the correlation instead of the convolution (this is the

inverse operation), and implementations for the roto-transformation group are demonstrated to have SOTA performance (for the time; 2016) on a rotated-MNIST dataset.

Several works extend or generalize G-CNNs. Cohen et al. [20] extend a similar approach to G-CNNs for defining CNNs equivariant to transformations in different input spaces, with specific mention of Euclidean and the sphere. These properties are proven using fiber bundles and fields.

Lenssen et al. [15] extend G-Convolutions to Lie groups and use them as part of a Capsule Network [30, 31], which is itself a generalization of a CNN. Using a similar approach to G-CNNs [10], their *group equivariant capsule network* outperforms non-equivariant equivalents. Equivariance may be especially useful in capsule networks, where the degree of parameter sharing in a G-Convolution could reduce the large number of trainable weights typically in a capsule network.

Romero et al. [14] define the *attentive group convolution*, which allows them to consistently outperform baseline G-CNNs. The addition of attention [32] accentuates meaningful symmetry during training and the learned concepts can be human-interpreted from the resulting attention maps.

Some apply G-CNNs to 3D problems [12, 13] and the 3D-Roto-Translation group over voxel data. These demonstrate that exploiting equivariances during learning, and specifically the G-CNN architecture has the ability to improve performance.

More generalize G-CNNs to the SE(2) group. Weiler et al. [17] use *Steerable Filters* which share weights to guarantee equivariance in each layer. This is improved upon by Bekkers et al. [18] who lighten the restrictions on the kernel functions and so show improved performance. Smets et al. [21] present a PDE-based framework which generalizes G-CNNs, applying known ideas for reasoning about CNNs using PDEs [33].

These methods notably solve an important problem in the standard G-CNN; that it can only provide equivariance over those groups which transform input exactly over the pixel grid. For board games however, the transformation groups are usually finite and map square-to-square and as such the improvement offered by these methods isn't as applicable.

Cohen et al. [19] solve the problem of CNNs applied to spherical signals such as 360° images and video. Definition for *spherical cross-correlation* allows their *Spherical CNNs* to exhibit equivariance over translations on the surface of a sphere. They show how this correlation can be computed efficiently using a generalized Fast Fourier Transform (FFT) algorithm.

Dieleman et al. [11] approach the problem of an equivariant CNN from a different angle. Other methods generally work by replacing the convolution operation, instead these create equivariance from a higher level. This is done by applying each layer multiple times to different transformations of the input. Their approach applied to CNNs "describes the same kind of models" as G-CNNs, although may be extended to arbitrary layer types. The method is specifically demonstrated for equivariance over cyclic symmetries, i.e. rotations of multiples of 90°, and issues with extending the approach to arbitrary groups are briefly commented on.

3.4 Equivariances in a Broader Context

There are other works considering equivariances or invariances of existing neural networks or unifying different approaches.

Lenc et al. [34] investigate the extent to which CNNs may naturally learn invariant or equivariant representations through training. They demonstrate that early layers of a trained AlexNet [35] network are approximately invariant to symmetries. They further evaluate how rotations or reflections of a layer affect subsequent layers. Since performance is roughly maintained, this is a strong indicator that networks learn approximate transformation equivariance through training.

Bloem-Reddy et al. [6] use tools from probability and statistics to define a general representation which characterizes the structure of many invariant and equivariant neural networks. They show that *Deep Sets* [4], *Message Passing NNs* [8] and others can be found using their program; a method for finding these symmetric representations. Further, several of their theorems generalize those in previous literature.

While 2D convolutions are equivariant to translations of the input, Zhang et al. [36] show that many downsampling methods commonly used in CNNs are not. These include max-pooling, strided-convolution and average pooling. In practice demonstrating that predictions made by AlexNet [35] and VGG [37] can vary by 90%+ for single pixel translations of the input. They propose variations of these operations, typically by adding an additional blurring step, which alleviates this. Their modified networks have improved translation invariance, but the invariance is still approximate.

4 Game Graph Convolutional Networks

In order to improve the neural architecture for games, we are looking for one which is equivariant over reflections or rotations of the game board. Here we define Game Graph Convolutional Networks (GGCNs), which achieves this equivariance by using *Graph Convolutional Networks (GCNs)* [7] or *Relational Graph Convolutional Network (R-GCNs)* [28]. These have inherent invariances and equivariances that are possible to exploit by representing the problem in specific ways. Here we apply this approach to checkers but the applications are more broad.

4.1 Boardgames as Graphs

In order to use a Graph Convolutional Network to predict moves for a game, we first represent the board state as a graph. Subsequently, we can show that this representation is equivariant to symmetries.

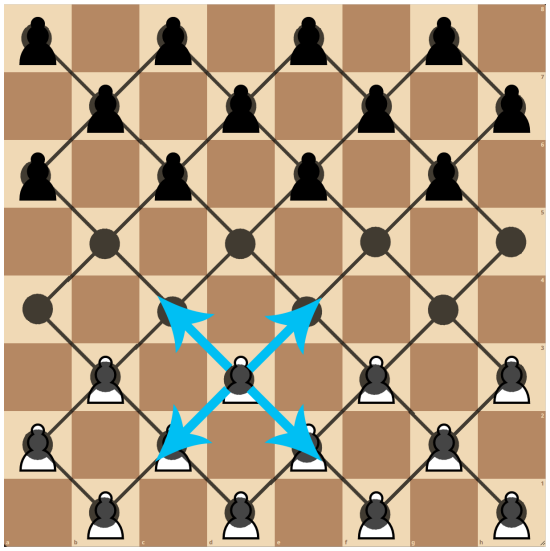


Figure 4.1: The starting checkers position represented as a graph. The arrows in blue represent the possible moves for a piece.

We have a node for each reachable square on the board, and edges between adjacent reachable squares. This representation can be seen in Fig. 4.1. At each node we store 1, 0, or -1 to denote the presence of a player's piece, an empty square, or an opponent piece respectively. Every possible move corresponds to exactly one directed edge, including where captures involve

'jumping' an extra square. The network then predicts a value for each directed edge indicating the probability of that move.

The graph representation of a board alone doesn't contain enough information to reasonably play checkers. This is because the graph doesn't give the orientation of the board. Each node doesn't 'know' which edges face 'up' or 'down' the board, nor is this information calculable from its neighbours. To fix this, the board can be represented as a directed graph with typed edges. The edges moving toward the opponent's side are one 'type' while edges moving towards the player's side are another. RGCNs extend the GCN architecture to add support for multiple edge types in a similarly equivariant way.

4.2 Equivariances of GGCNs

It can be seen, by taking two symmetrically equivalent boards and representing them both as a graph, that both are equal in the following sense: For each node in one graph there is an equivalent one in the other graph sharing the same value. There is also an equivalent edge for any edge in one graph. This is enough to say both graphs are isomorphic.

GCNs and their relational variant are equivariant over isomorphic graphs [9], so they will create a symmetry equivariant learning algorithm when the board state is represented this way.

5 Finite Group Neural Networks

In this section we derive *Finite Group Neural Networks* or FGNNs. FGNNs extend from an approach where an existing neural network architecture is taken and equivariance to a group is emposed layer-by-layer. The simple approach, discussed first, attempts to create layers which are equivariant to the group directly. The problems with this method are briefly discussed to motivate the FGNN approach, which creates equivariance over a higher level.

5.1 A Simple Approach

We take a neural network $N : X \to Y$ which we wish be equivariant over some group G:

$$N(gx) = gN(x)$$

for all $g \in G$ and $x \in X$

We can express networks using the same methods used by B. Fong et al. [38]. In their paper they define a category theoretical model for supervised learning algorithms, with some focus on neural networks. Of interest to us; they express a neural network as a sequence of k layers with corresponding input/output sizes:

$$(I_1 : \mathbb{R}^{C_1} \times \mathbb{R}^{n_0} \to \mathbb{R}^{n_1}), (I_2 : \mathbb{R}^{C_2} \times \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}) \dots (I_k : \mathbb{R}^{C_k} \times \mathbb{R}^{n_{k-1}} \to \mathbb{R}^{n_k})$$

Each function $I_i : i \in 1 \dots k$, is parametric with $|C_i|$ weights. In a neural network, these correspond to the neural weights and biases associated with any input, and it is these weights which are updated by back-propagation. A full, trained neural network is then expressed as this sequence of functions $I_1, I_2 \dots I_k$ along with the parameters $C_1, C_2 \dots C_k$.

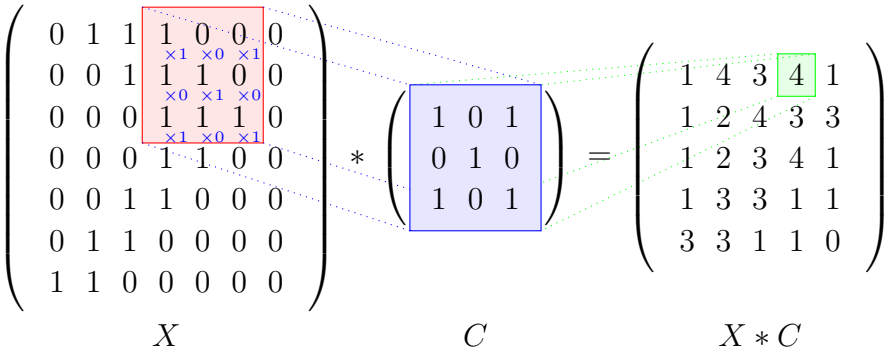
In order create a network which is equivariant to some group G, we could show that each parametric layer ($I(C, \cdot)$) individually is equivariant to that group, or equivalently commutes with any element of the group:

$$g \circ I(C, \cdot) = I(C, \cdot) \circ g$$

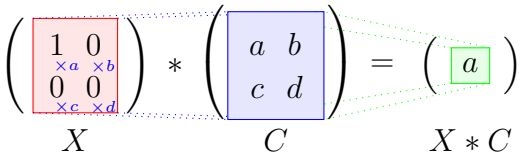
for all $g \in G$

As a consequence, the composition of many layers (i.e. the entire network) will also be group equivariant. We look at the restrictions on C such that the resulting layer is group equivariant. When applying this to a 2D convolutional layer, the titular component of CNNs, we can see that the weight kernel is too restricted.

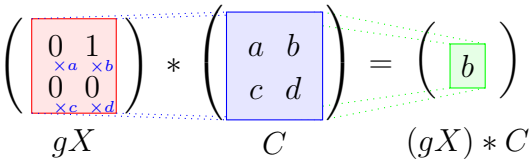
A 2D convolutional layer with input X simply has some weight kernel C applied at each point of the input to give the resulting matrix. This is visualized below, as the product of the red section convolved with the blue kernel gives the green value in the output.



The restriction required for equivariance can be calculated by taking the simplest possible 'interesting' network; a layer with a 2x2 input and kernel. Here, writing the weights of the network as a, b, c, d. The input, I, is a matrix with a single 1.



If our layer is equivariant over the group of 90° rotations, then rotating the input (applying g) should simply give a rotation of the output. Since the output is a matrix with just a single value, rotating it has no effect.



Here, rotating the input prior to applying the kernel gives b. Since the layer must be equivariant to g.

$$a = (gX) * C = g(X * C) = b$$

The approach can be continued to show that $a = b = c = d$. This is a big problem for our network, as many weights in each kernel must be the same. Applying the approach to a 3x3 kernel shows the kernel must be of the form:

$$C = \begin{bmatrix} a & b & a \\ b & d & b \\ a & b & a \end{bmatrix}$$

Generally, the kernel must be invariant over the chosen group. This makes the resulting architecture similar to the GCN approach shown previously. We can express the GCN network architecture applied to an infinite[1] board as several convolutional layers with restrictions on the weights. Since edges are only one square long, they fit into a 3x3 kernel.

$$C = \begin{bmatrix} a & 0 & a \\ 0 & b & 0 \\ a & 0 & a \end{bmatrix}$$

[1] In this case the normalization term is constant at all nodes.

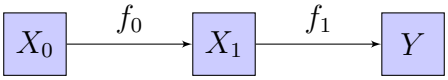
Similarly to GCNs, these layers can't learn to activate on non-symmetrical features and the resulting network will struggle to recognise lines and edges, or specific piece arrangements. Ideally, we want to avoid this, and create a network which can maintain a richer internal relationship of features while still having the same equivariance properties.

5.2 A Better Approach

Enforcing equivariance over the group directly in each layer, as seen previously, limits the ability of the layer to represent non-symmetrical features. Instead, we define an operation T_g for each $g \in G$ and enforce each layer's equivariance to this instead. This method results from a definition of T_g given formally in the subsequent section, and a general solution for creating an equivariance over it for an arbitrary layer.

In order to aid description, this section gives a practical example of FGNNs derived from a simple network and equivariance over horizontal reflections. The formalization of this which applies to arbitrary layers, network architectures and groups follows.

A neural network without recurrent layers or memory is a pure function $N(X) = Y$. It can be written as a composition of layers $f_i, i \in [1..k]$, which are linearly composed for now.



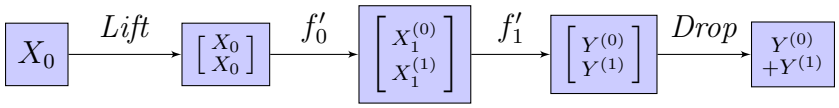
We add the restriction that the input tensor X must have an even number of 'layers', and write it as two stacked subtensors of equal size.

$$X = \begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}$$

In order to modify this network, we replace all functions f_i with their modified version f_i', where g is a matrix which horizontally reflects the input. Note the reordering of the subtensors in the second call of f_i.

$$f_i'(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) = \begin{bmatrix} f_i(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) \\ g f_i(g \begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix}) \end{bmatrix}$$

Then resulting equivariant network N' can be written as:



Along with switching out the layers, we add two more here. The $Lift$ layer at the start simply duplicates the input. The $Drop$ layer at the end adds the two subtensors.

$$Lift(X) = \begin{bmatrix} X \\ X \end{bmatrix} \tag{1}$$

$$Drop(\begin{bmatrix} X^{(0)} \\ X^{(1)} \end{bmatrix}) = X^{(0)} + X^{(1)} \tag{2}$$

Note that the dimensionality of the component layers may need to be changed, i.e. for convolutional layers the number of features should be halved in all layers except the last to retain the same model size and maintain layer inter-connectivity.

In order to be reflection equivariant, the following property should hold for all inputs:

$$N'(X) = Y \implies N'(gX) = gY = gN'(X)$$

This can be verified by defining an operation $T(X)$ which both applies g and reverses the order of the component tensors. Since g reflects the tensors, it can be applied equally to each sub-tensor $X^{(i)}$.

$$T(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) = g \begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix} = \begin{bmatrix} gX^{(2)} \\ gX^{(1)} \end{bmatrix}$$

We can then show that applying g the input leads to the following network values after each layer is applied, with the result correctly showing equivariance.

$$gX_0 \xrightarrow{Lift} T\begin{bmatrix} X_0 \\ X_0 \end{bmatrix} \xrightarrow{f'_0} T\begin{bmatrix} X_1^{(0)} \\ X_1^{(1)} \end{bmatrix} \xrightarrow{f'_1} T\begin{bmatrix} Y^{(0)} \\ Y^{(1)} \end{bmatrix} \xrightarrow{Drop} g\begin{pmatrix} Y^{(0)} \\ +Y^{(1)} \end{pmatrix}$$

The $Lift$ and $Drop$ layers can both be easily verified from the definitions. Reordering identical subtensors and reordering tensors before adding them, respectively, have no effect. The commutativity of T over f'_i is slightly more challenging.

$$f'_i(T\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) = \begin{bmatrix} f_i(T\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) \\ gf_i(gT\begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix}) \end{bmatrix} = \begin{bmatrix} f_i(g\begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix}) \\ gf_i(g^2\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) \end{bmatrix}$$

Then using the fact that g is its own inverse, i.e. reflecting something twice has no effect.

$$= \begin{bmatrix} f_i(g\begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix}) \\ gf_i(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) \end{bmatrix} = Tg^{-1}\begin{bmatrix} gf_i(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) \\ f_i(g\begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix}) \end{bmatrix} = T\begin{bmatrix} f_i(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}) \\ gf_i(g\begin{bmatrix} X^{(2)} \\ X^{(1)} \end{bmatrix}) \end{bmatrix} = Tf'_i(\begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix})$$

Hence each f_i commutes with T and so the full network is equivariant to flips.

5.3 T-Equivariance

The example above shows a simple network modified to be equivariant to horizantal reflections. Here this method is extended to work over arbitrary groups (with minor restrictions), and for networks with skip connections. There are also more efficient implementations for some layer types such as pooling.

For this the input tensor of the layer, denoted X, must have a multiple of $|G|$ layers (this is later enforced by the architecture) and the group elements must apply to slices of the tensor. For simple rotations and reflections of the input, this is evidently true.

$$gX = \begin{bmatrix} gX^{(1)} \\ gX^{(2)} \\ \vdots \end{bmatrix} \tag{3}$$

T_g is defined for each element of the group $g \in G$, and consists of splitting the input tensor into $|G|$ slices, reordering them before concatenating them, and applying g. We denote this reordering as R_g. Since the reordering of slices must commute with the group elements G:

$$T_g := R_g \circ g = g \circ R_g \tag{4}$$

Definition 5.3.1. T-Equivariance A function is T-Equivariant if it commutes with all of the resulting operations.

$$\forall g \in G : T_g \circ f = f \circ T_g$$

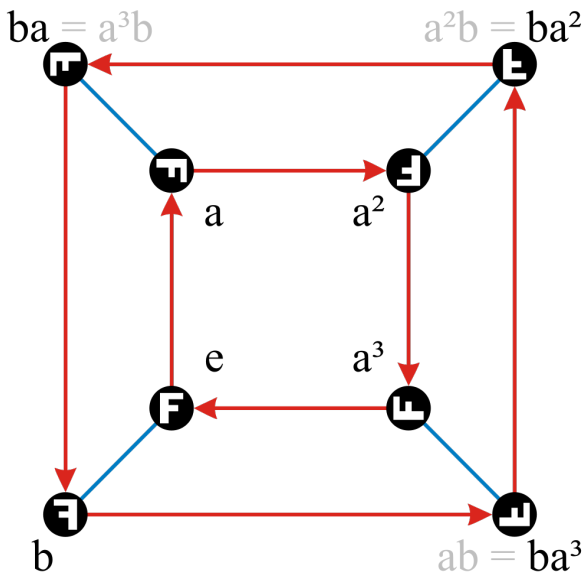


Figure 5.1: The cayley graph of the dihedral group of the square; D_8. This defines two generating functions; a (red) and b (blue), for 90° rotation and horiztontal reflection respectively. The nodes then represent all elements of the group/ all possible transformations of an image.

The exact reordering of pieces is defined similarly to a cayley graph representation of the group. We assign these pieces to nodes in the graph then permute them according to the outgoing edges. An example of this mapping is shown in Fig 5.1. This can be viewed as representing this group as a subgroup of an equivalently sized permutation group (S_n). Formally:

Let $m(\cdot)$ be a function which arbitrarily assigns each $g \in G$ a unique index $\in [1..|G|]$. Let $[g_1, g_2, \dots]$ be the resulting mapping applied to the group. Note that $m(g_i) = i$ by definition.

Consider the input tensor X as divided into $|G|$ sub-tensors denoted $[X^{(1)}, X^{(2)} \dots X^{(|G|)}]$. $R_s(X)$ reorders the input such that in the output tensor X', $X'^{(i)} = X^{(m(g_i s))}$.

In full, this mapping can be written as:

$$R_s\left(\begin{bmatrix} X^{(1)} \\ X^{(2)} \\ \vdots \\ x_{(|G|)} \end{bmatrix}\right) = \left(\begin{bmatrix} X^{(m(g_1))} \\ X^{(m(g_2))} \\ \vdots \\ X^{(m(g_{|G|}))} \end{bmatrix}\right) \tag{5}$$

Lemma 5.3.1. *$T_h T_s = T_{hs}$ for all $h, s \in G$*

Proof. This can be proven by considering the way which R_h and R_s permute the subtensors.

R_h maps each subtensor at index i to index $m(g_i h)$. Therefore $R_h R_s$ then maps index i to index $m(g_{m(g_i s)} h)$. Since $m(g_i) = i$, $g_{m(h)} = h$.

$R_h R_s$ then maps i to $m(g_i hs)$, which is the same as R_{hs}. Hence $R_h R_s = R_{hs}$

Finally, $T_h T_s = R_h R_s hs = R_{hs} hs = T_{hs}$ □

5.4 T-Equivariant Layers

It is possible to redefine any layer so that it commutes with T_s, for any $s \in G$.

Given some function f; a layer in our network, we can define its T-equivariant version as f'.

$$f'(X) := \begin{bmatrix} f(T_{g_1}X)g_1^{-1} \\ f(T_{g_2}X)g_2^{-1} \\ \vdots \\ f(T_{g_{|G|}}X)g_{|G|}^{-1} \end{bmatrix} \tag{6}$$

Theorem 5.4.1 (T-Equivariance of Simple Functions). *$f'(X)$ is T-Equivariant if $g \in G$ commutes with R_h.*

Proof. In order to prove that f' is equivariant to T, we need to show that $\forall s \in G$: $T_s(f'(X)) = f'(T_s(X))$.

$$T_s(f'(X)) = T_s \begin{bmatrix} f(T_{g_1}X)g_1^{-1} \\ f(T_{g_2}X)g_2^{-1} \\ \vdots \\ f(T_{g_{|G|}}X)g_{|G|}^{-1} \end{bmatrix} = s \circ R_s \begin{bmatrix} f(T_{g_1}X)g_1^{-1} \\ f(T_{g_2}X)g_2^{-1} \\ \vdots \\ f(T_{g_{|G|}}X)g_{|G|}^{-1} \end{bmatrix}$$

Element i in the matrix is $f(T_{g_i}X)g_i^{-1}$, and R_s maps each element in the output such that $x'_i = x_{m(g_i s)}$. Then the resulting section at index i is $f(T_{g_{m(g_i s)}}X)g_{m(g_i s)}^{-1}$. Since $g_{m(f)} = f$ this simplifies to $f(T_{g_i s}X)(g_i s)^{-1}$. Hence:

$$f'(X) = g \begin{bmatrix} f(T_{g_1 s}X)(g_1 s)^{-1} \\ f(T_{g_2 s}X)(g_2 s)^{-1} \\ \vdots \end{bmatrix}$$

From our restriction that the group G must commute with taking slices of the input, $s \in G$ must distribute across slices of the matrix. Further, from Lemma 5.3.1 $T_{g_i s} = T_{g_i}T_s$.

$$T_s f'(X) = \begin{bmatrix} f(T_{g_1}T_sX)g_1^{-1}s^{-1}s \\ f(T_{g_2}T_sX)g_2^{-1}s^{-1}s \\ \vdots \end{bmatrix} = \begin{bmatrix} f(T_{g_1}(T_sX))g_1^{-1} \\ f(T_{g_2}(T_sX))g_2^{-1} \\ \vdots \end{bmatrix} = f'(T_sX)$$

Since f' commutes with any arbitrary $T_s, s \in G$, it is T-Equivariant. □

5.5 Lift & Drop

Now that we have the core of our network, we can consider how to enter and remove data from either end. These are layers; $Lift$ to enter data and $Drop$ to remove, and are defined by the following properties.

$$Lift \circ g(X) = T_g \circ Lift(X) \tag{7}$$

And...

$$Drop \circ T_g(X) = g \circ Drop(X) \tag{8}$$

For simple transformation groups we use the definitions: $Lift(X)$ simply stacks $|G|$ copies of X, and $Drop(x)$ divides X into $|G|$ pieces and sums them.

$$Lift(X) = \begin{bmatrix} X \\ X \\ \vdots \end{bmatrix} \tag{9}$$

$$Drop\left(\begin{bmatrix} X^{(1)} \\ X^{(2)} \\ \vdots \end{bmatrix}\right) = X^{(1)} + X^{(2)} + \dots \tag{10}$$

The defining properties for these can be verified from properties of g (3) and T_g (4). Importantly this means that other definitions of these layers are possible for different groups, as is important for groups acting on the action spaces of board games.

5.6 Skip connections

Skip-connections make the training of very deep networks more stable, and are an indispensable component of a variety of popular neural network architectures [22, 23, 24, 25, 26].

$Merge$ layers which maintain T-Equivariance can be defined simply by splitting and zipping together the input tensors.

$$Merge\left(\begin{bmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(|G|)} \end{bmatrix}, \begin{bmatrix} B^{(1)} \\ B^{(2)} \\ \vdots \\ B^{(|G|)} \end{bmatrix}\right) = \begin{bmatrix} A^{(1)} \\ B^{(1)} \\ A^{(2)} \\ B^{(2)} \\ \vdots \\ A^{(|G|)} \\ B^{(|G|)} \end{bmatrix} \tag{11}$$

Lemma 5.6.1. $Merge(TA, TB) = T \circ Merge(A, B)$

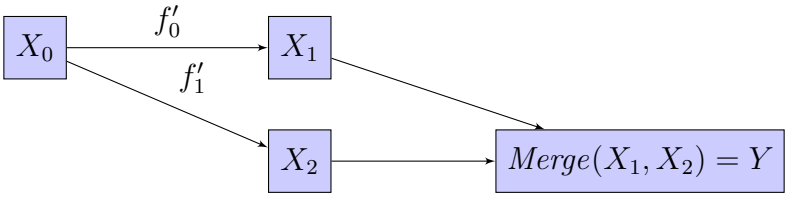
Proof. The proof of the merge layer's T-equivarience follows from the definition; the shuffling of the input results in the same shuffling of the output. For all T_s, $s \in G$:

$$\begin{aligned} Merge\left(T_s\left(\begin{bmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(|G|)} \end{bmatrix}\right), T_s\left(\begin{bmatrix} B^{(1)} \\ B^{(2)} \\ \vdots \\ B^{(|G|)} \end{bmatrix}\right)\right) &= Merge\left(\begin{bmatrix} A^{(m(g_1 s))_s} \\ A^{(m(g_2 s))_s} \\ \vdots \\ A^{(m(g_{|G|} s))_s} \end{bmatrix}, \begin{bmatrix} B^{(m(g_1 s))_s} \\ B^{(m(g_2 s))_s} \\ \vdots \\ B^{(m(g_{|G|} s))_s} \end{bmatrix}\right) \\ &= \begin{bmatrix} A^{m(g_1 s)_s} \\ B^{m(g_1 s)_s} \\ A^{m(g_2 s)_s} \\ B^{m(g_2 s)_s} \\ \vdots \\ A^{(m(g_{|G|} s))_s} \\ B^{(m(g_{|G|} s))_s} \end{bmatrix} = T_s\left(\begin{bmatrix} A^1 \\ B^1 \\ A^2 \\ B^2 \\ \vdots \\ A^{|G|} \\ B^{|G|} \end{bmatrix}\right) \end{aligned} \tag{12}$$

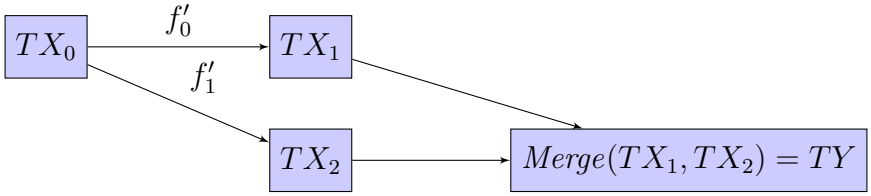
□

Lemma 5.6.2. *Any skip connection over T-Equivariant functions followed by a $Merge$ layer is itself a T-Equivariant funciton.*

Proof. Any skip connection in a network can be visualized as the following graph. Both f_0' and f_1' are T-Equivariant, and may represent the composition of multiple layers and skip connections.



$T : T \in \{T_g, g \in G\}$ can commute with this structure, using the definitions of T-Equivariance and Lemma 5.6.1. Hence it is T-Equivariant.



□

5.7 Pooling

The previous definition allows for arbitrary layers to be made T-Equivariant, which includes up or down-scaling layers. However, maximum, minimum, and average pooling layers are symmetric by definition, and applied pointwise. Over the D_8 group or any sub-group this means they are T-Equivariant without modification.

5.8 Group Equivariance

Finally, let our network be N; the T-equivariant function f' composed at either end with a $Lift$ and a $Drop$ layer.

$$N = Drop \circ f' \circ Lift$$

Theorem 5.8.1 (Group Equivariance). *$N = Drop \circ f' \circ Lift$ is equivariant over group G.*

Proof. Due to the definitions of $Lift$ and $Drop$, $\forall g \in G$:

$$\begin{aligned} N \circ g &= Drop \circ f' \circ Lift \circ g \\ &= Drop \circ f' \circ T_g \circ Lift \\ &= Drop \circ T_g \circ f' \circ Lift \\ &= g \circ Drop \circ f' \circ Lift \\ &= g \circ N \end{aligned} \tag{13}$$

□

Hence, N is equivariant over G, since $g : g \in G$ commutes. All together this allows us then to 'upgrade' all layers in any network to commute with T, and by adding a layer at the start and end the whole network will be equivariant to any chosen finite group.

5.9 Move Embeddings

For board-games, the network must output a move, or a policy π over all possible moves. The reflection of this policy tensor or move embedding may not correspond to the symmetrically opposite move.

For example, the *AlphaZero* [2] chess network outputs a policy representing the probability of all possible moves. This is a tensor of size $8 \times 8 \times 73$. Each of the 8×8 positions identifies where to "pick up" a piece, while each of the corresponding 73 planes encode how that piece should be moved. The first 56 of which move a piece 1–7 squares along one of the 8 compass directions. The next 8 planes correspond to possible knight moves. Finally, the remaining 9 planes encode possible pawn under-promotions or pawn captures.

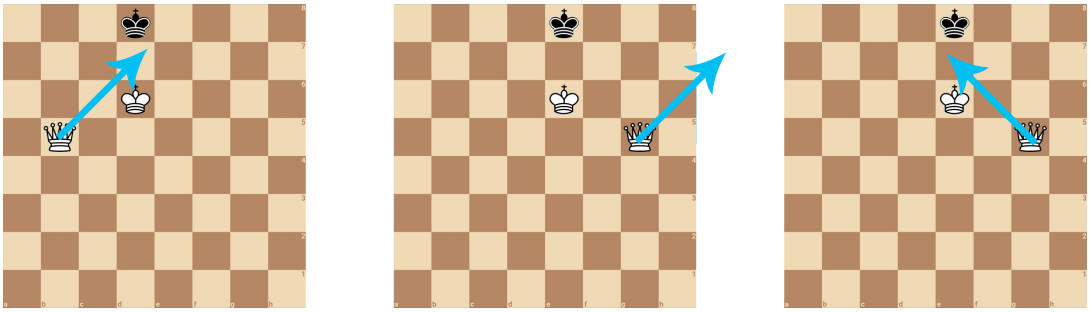


Figure 5.2: A move (blue arrow) on a board state (left). The resulting move and board state after naively reflecting both tensors (middle). The correctly reflected board and move (right).

Simply reflecting a move represented in this embedding gives the incorrect move as shown in Fig. 5.2. Since we can rotate the board and recolour pieces in the input such that white is to play in all cases, we only need the network to be equivariant over horizontal reflections.

We consider the group G as acting upon this space in such a way that it maps between symmetrically opposite moves. Hence a new $Drop$ layer must be defined for which its defining property holds.

$$Drop \circ T_g(X) = g \circ Drop(X)$$

This can be done by splitting the move space into subtensors relating to symmetrical; S_0 moves, non-symmetrical moves moving left; X_0 and non-symmetrical moves moving right; X_1. If g horizontally reflects the board

$$X = \begin{bmatrix} S_0 \\ X_0 \\ S_0' \\ X_1 \end{bmatrix}$$

$$Drop'\left(\begin{bmatrix} S_0 \\ X_0 \\ S_0' \\ X_1 \end{bmatrix}\right) = \begin{bmatrix} \frac{S_0+S_0'}{2} \\ X_0 \\ X_1 \end{bmatrix}$$

Then equivariance over horizontal reflections can be seen by:

$$Drop'\left(T_g \begin{bmatrix} S_0 \\ X_0 \\ S_0' \\ X_1 \end{bmatrix}\right) = Drop'\left(\begin{bmatrix} gS_0' \\ gX_1 \\ gS_0 \\ gX_0 \end{bmatrix}\right) = g \begin{bmatrix} \frac{S_0+S_0'}{2} \\ X_1 \\ X_0 \end{bmatrix}$$

Since this resulting network has X_0 and X_1 switched — each symmetrically relating to opposite moves, the move embedding is correctly flipped. This $Drop'$ layer then, when used in conjunction with any FGNN, will correctly reflect the move embeddings when the input tensor is reflected.

6 Methodology

This section outlines the methods and architectures used in testing GGCNs and FGNNs. I train versions of both on a novel checkers dataset, and compare results with equivalent baseline CNNs. Finally I create an equivariant version of U-Net (FGNN-U-Net), and evaluate it on an image segmentation task.

6.1 Dataset

In order to enable quick iteration of ideas and smaller neural networks to be tested, I choose an equivalently smaller/ simpler game. Checkers (or draughts) is a 2 player game played on an 8×8 board. It is a solved game [39], and has a relatively small number of possible states ($\sim 5 \times 10^{21}$ [39]) compared to chess ($\sim 10^{43}$ [40]), or go ($\sim 2 \times 10^{170}$ [41]). Still it has many of the properties which makes creating equivariant networks for board-games hard; pieces which 'move' resulting in more complicated move or policy embeddings, and multiple piece types (after promotion).

A dataset of 22 thousand tournament games compiled by the Open Checker Archive[1] is used to train and evaluate the networks. This dataset contains moves in Portable Draughts Notation (PDN), so this must first be parsed for board positions and moves. When a move consists of multiple captures, called jumps, these are added to the dataset as multiple board positions and moves each consisting of a single jump. During play, the network can be called multiple times to produce a long string of captures as long as it is valid. The resulting training and test sets together contain almost 1 million board states along with the next move made from that state. Training a network on this, then, is effectively teaching it to predict the moves made by high level checkers players.

The way the board states are passed to the network varies by method but for both each square on the board is represented as a single value. If the square is empty this value is 0, otherwise for regular pieces it is -1 or 1 for black or red respectively. Squares containing kings similarly are -3 or 3 for black or red. The moves made are one-hot encoded into an vector of size 128, with a space for each square on the board (32) times each of the directions (4) that a piece may move from that square. These are shown in blue/red in Fig 6.1.

The moves are shown using a simple matplotlib diagram which is useful to visualize the thinking of the network. In the case of a mistake in the input format or the network architecture, the network would often simply converge to predicting the average moves. This could easily be seen by evaluating several random positions from the training set.

[1] Open Checker Archive: http://www.fierz.ch/download.php

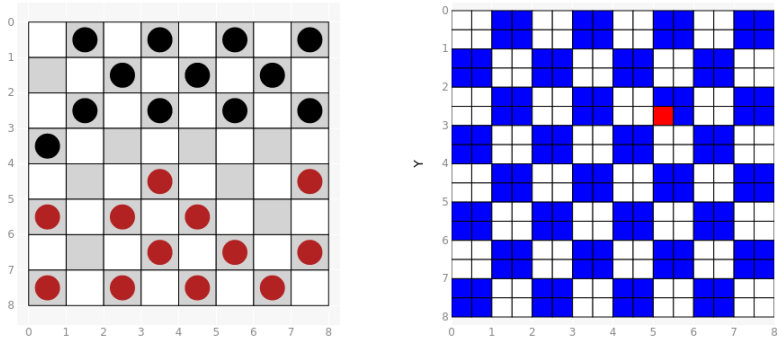


Figure 6.1: The current board state (left) is given as input to the network and the next move to be played, in red (right), is predicted by the network.

6.2 Baseline

In order to evaluate the performance of the methods, it is important to have a benchmark. A multi-layer CNN is used, with differing numbers of filters in each layer to create different sized models, for the following reasons.

The models are easily scalable: Equivariance may only benefit small or large networks. It is therefore important to evaluate across a wide variety of model sizes which vary from under-fitting to over-fitting. CNN models can be scaled up or down small amounts, simply by adding or removing filters.

There are equivalent FGNNs to any network: Since any network architecture has an equivalent FGNN, it isn't as important which is chosen for comparison. A minimal network is then logical to eliminate other implementation details which may make the comparison unfair.

Multiple definitions for normalisation or dropout are possible: For example, whether dropout should be applied layer-wise (i.e. to f'_i, breaking equivariance during training) or per function (i.e. to f_i where a single dropout causes multiple values to be lost in the next layer) is unclear and should be established experimentally. Using a network without these empirically motivated techniques is arguably a fairer comparison.

It performs well: Through informal empirical observation across a wide variety of model types and depths, it was found that a CNN architecture converges more consistently than models with multiple fully connected layers, performs well, and is quick to train.

To compare against GGCNs, CNNs with comparable numbers of trainable weights are used. These contain 3 or 5 layers and a small number of features. Both models are small due to the fact that GCNs are unoptimized for this type of problem and are slow to train.

To compare against FGNNs, CNNs with 10 layers are used. Models deeper than this were found not to notably improve performance, and 10 layers allows for models which both under and over-fit the dataset.

The CNN has the same number of filters in each layer. Each layer has a 3×3 convolution, with *ReLU* activation [42], and is zero-padded. The number of filters is varied to give a different number of trainable weights in a variety of models. The final layer applies 4 filters, resulting in 4 layers. The then model masks out the 32 squares which correspond to the reachable

squares in a checkers game, and flattens the values into a single vector of size 128. A final softmax layer ensures the vector is normalized.

6.3 GGCNs For Checkers

My implementation of a Relational Graph Convolutional Network is heavily based on the implementation provided in open-source library *Spektral*[2]. I add the ability to maintain the adjacency matrix in the layer, removing the need to pass it on each iteration.

The GGCN for checkers model defines 3 edge types. One moving away from the current player, one towards, and one connecting each node to itself. Each edge type in each layer has its own set of trainable weights. The feature vector for each node in each layer then is the sum of the activations along each layer type. Each edge type for each layer has 100 filters, meaning there are $3 \times 100 \times 100 = 30000$ trainable weights per layer. The activation function in these layers is *relu*.

The final layer only has edges from each node to itself with 4 filters - one per move from that node. It then flattens the input into a single vector of size 128 before applying softmax activation. The models are trained using categorical-crossentropy loss. Together this trains the model to predict a probability distribution over possible moves.

All models are trained for 50 epochs on the training set, and evaluated on the previously-unseen validation set.

6.4 FGNNs For Checkers

As discussed previously, an issue with using existing methods for equivariant networks for games is their inability to 'mirror' more complex representations such as moves. One strength of our method is that it is easier to reason about. Transformations of the input (g, where $g \in G$) transform the output prior to the $Drop$ layer by T_g. Defining $Lift$ and $Drop$ as defined previously allows for networks to be equivariant to g when both input and output are acting in the same space.

6.4.1 Equivariance

There are effectively four ways the that the same board state can occur in checkers. Two of these are for each player, times two reflections along the horizontal axis of the board. We rotate the board so that the current player is always playing upwards, and 'recolour' them so the current player is black. Because of this, the architecture only needs to only be equivariant over horizontal flips of the game board.

For checkers, we choose to output a tensor which is $8 \times 8 \times 4$. Similar to *AlphaZero* each of the 8×8 vertical vectors each correspond to a starting square for a piece. The 4 layers represent the 4 directions to move a piece from that square. Both single-square moves and captures, moving 2 squares, are represented by the same layer in the output.

By choosing the order of these 4 layers to be the compass directions: NE SE NW SW, there is a simple method of creating an equivariant architecture. Reflecting a move involves swapping

[2]https://github.com/danielegrattarola/spektral

any NE move with NW, and SE with SW. This is the same as swapping the corresponding layers in the output. Finally, the full tensor can simply be horizontally reflected. For example, a move to the NE at (0,0) becomes a move to the NW at (7,0), which is exactly what we want.

This way to 'reflect' a move is the exact definition for T_g over the group of horizontal reflections. The architecture can be used by simply defining the $Drop$ layer to be the identity function.

6.4.2 Implementation

The implementation of FGNNs uses the concept of generating elements. A 'group' has one or more generating elements. For example, the D_8 group can be generated using horizontal reflections and rotations of 90°. By applying these operations multiple times one can construct any of the 8 elements of the group.

First, define a group to be equivariant over. For each generating element, g, give corresponding tensorflow functions. Then, define the period of that generating element — what power of that element corresponds to the identity. Finally, define R_g for that element, which represents how the element permutes all elements of the group.

This group object can be passed to $Lift$, $Drop$, $Merge$ and any other layer. The resulting network will be equivariant over that group provided the commutativity rules hold.

6.4.3 Model sizes & Training

Analogous to the baseline, the architecture chosen is based on a 10-layer CNN, with the same number of filters in each layer, with 3×3 convolutions and *relu* activations.

However, the final layer only applies 2 filters, resulting in 4 layers in the output due to the FGNN architecture. Then, as before the model masks out the 32 squares which correspond to the reachable squares in a checkers game, and flattens the values into a single vector of size 128. A final softmax layer ensures the vector is normalized.

The size of the output from a FGNN layer with a certain number of trainable weights isn't the same as it's equivalent non-equivariant version. Therefore the sizes of the FGNN networks don't match exactly with the baselines. All models are trained for 50 epochs with a categorical crossentropy loss.

6.5 Metrics

For the checkers dataset, to compare models we use accuracy. This is the percentage of times the model correctly predicted the move made on a board. We also compare based on the Top-3 accuracy. Here the prediction is correct if the correct move is within the 3 highest probability moves from the network. This is common for evaluating predictions in classification problems [35], such as this. Top-3 is especially applicable here as often in the same board position there are many playable moves.

These metrics measure the playing strength of the resulting networks by evaluating how closely their play would mirror those of high level checkers players.

7 Results

In this section we measure how well GGCNs and FGCNs perform in practice. Here we outline the results for these equivariant and baseline networks trained on the Checkers dataset, as well as a subsequent test applying FGCNs to the problem of biomedical image segmentation.

7.1 Game Graph Convolutional Networks

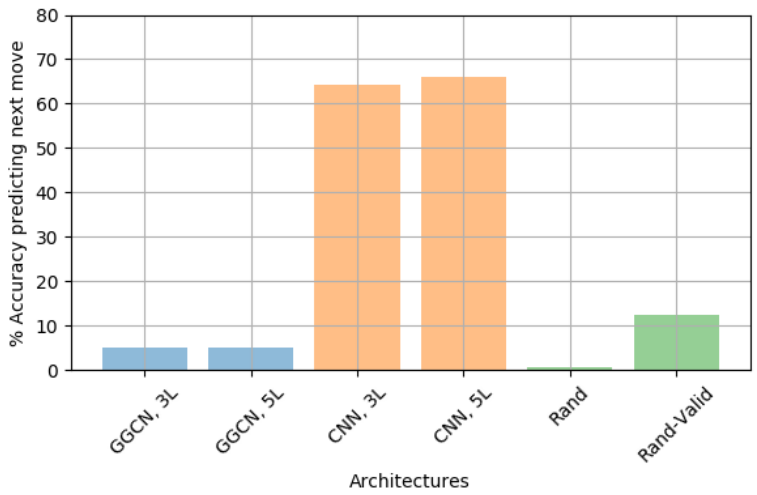


Figure 7.1: GGCN networks with 3 and 5 layers (3L, 5L) compared with simple equivalent CNNs and other methods. Rand is a random output from the network, and Rand-Valid is a random move taken which is legal in the current board state[2].

Both GGCN networks tested show ∼5.3% accuracy in predicting the next move made in an unseen high level checkers game. Fig. 7.1 shows this compared baseline CNNs with comparable numbers of weights, as well as simple non-ml strategies.

7.2 Finite Group Neural Networks

We first verify that FGNNs for several groups are equivariant in practice. This is done by generating random input tensors and testing if FGNN networks commute with transformations. We test groups of various transformations, including the D_8 group, and subgroups of horizontal

[2]Estimated from an average branching factor of 8 for checkers [43].

and/or vertical reflections, and of rotations of 90 degrees. In all cases the output never differs in any single value by more than 10^{-13}.

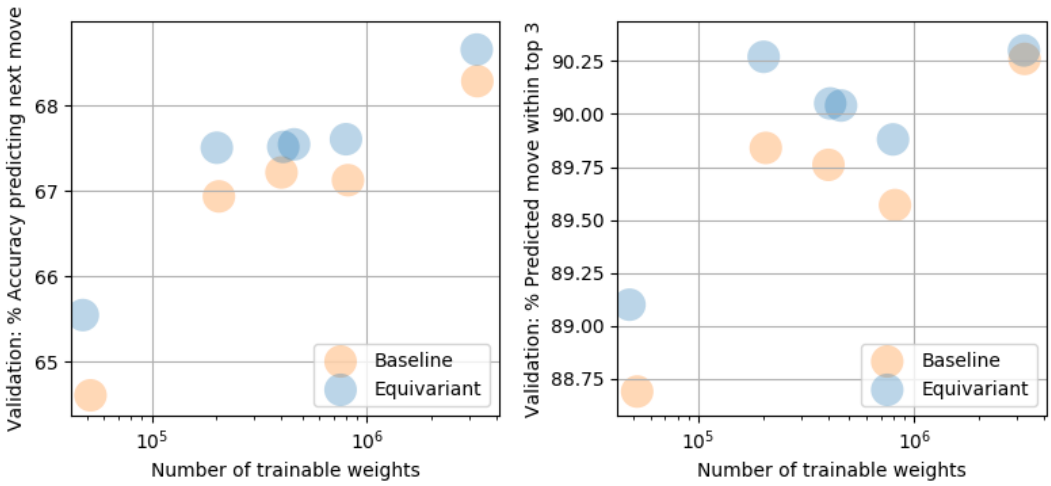


Figure 7.2: The accuracy (left) and top-3 accuracy (right) of equivariant and baseline networks of different sizes on unseen board states.

For checkers, we compare a horizontally equivariant FGNN to the baseline CNN. Both architectures have 10 layers, while the number of features per layer is varied to create different sizes of model. Because the equivariant architecture applies the same layer multiple times, these feature counts don't correspond to model sizes directly. To accommodate this, we compare between architectures based on the number of trainable weights per model. Fig. 7.2 shows a comparison on the validation set; after training models for 50 epochs, the model is tested on approximately 100k previously unseen board states and moves.

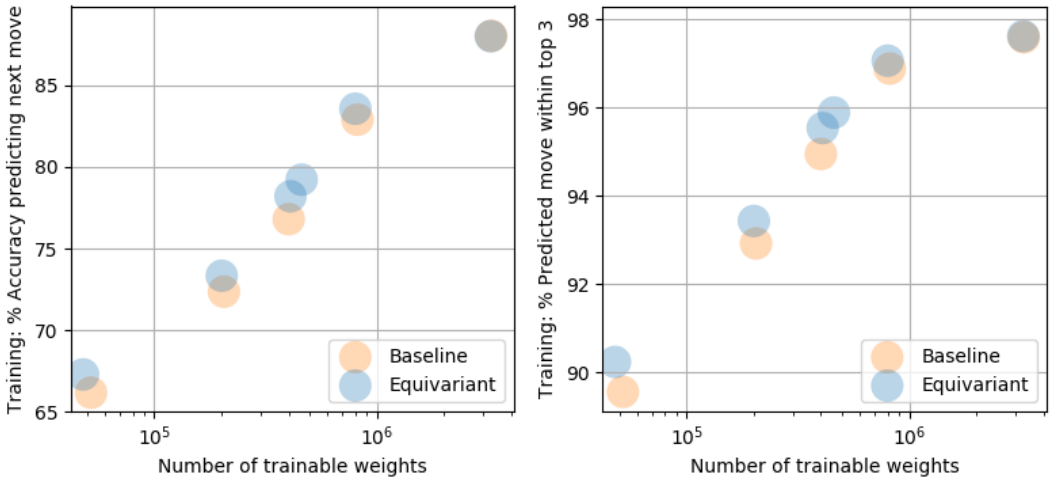


Figure 7.3: The accuracy (left) and top-3 accuracy (right) of networks over the training set.

Fig. 7.3, shows the equivalent results on the training set. This can be interpreted to measure the model's learning capacity. If an equivariant network only affected over-fitting, the models would score similarly here.

Comparing the ability of this method to combat overfitting is more challenging. Fig. 7.4

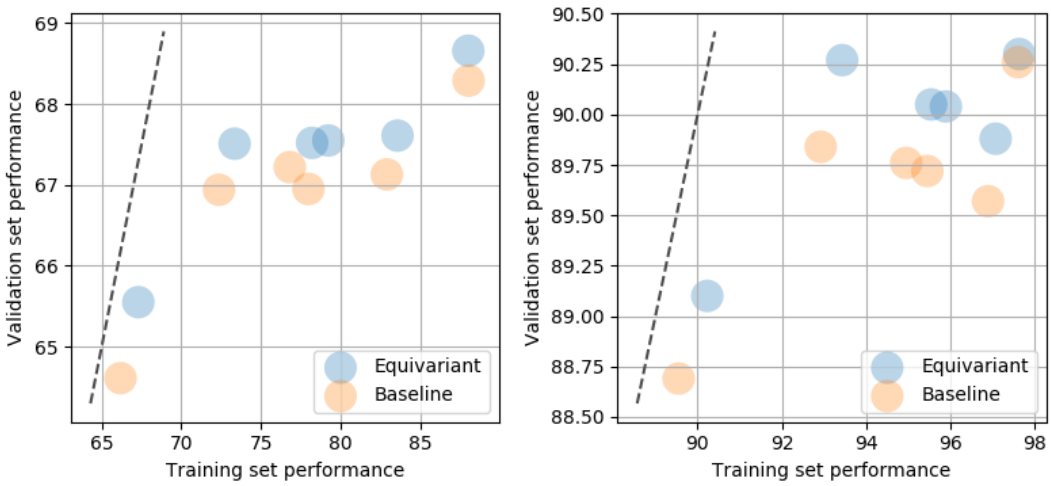


Figure 7.4: The models' performance on the training set vs the test set. The dashed line is where both are equal, and hence no overfitting occured. Nearer this line then equates to less overfitting.

compares each model's accuracy on the training set (seen examples) against the unseen examples of the validation set. Models which over-fit more will have a larger difference between training and validation set performance, which can be seen as a larger distance to the dashed line.

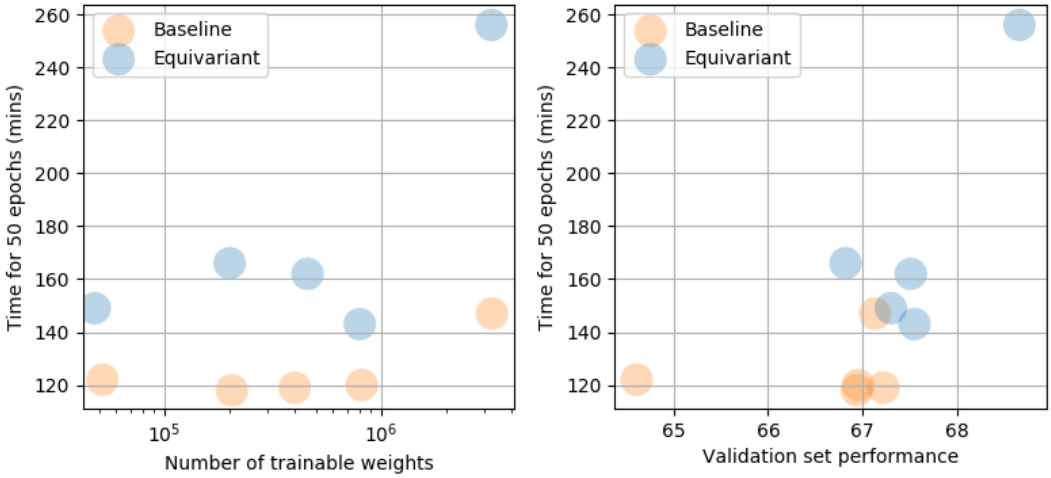


Figure 7.5: The training time in minutes compared against the models' number of trainable weights (left) and resulting performance (right).

The training time (for 50 epochs) in minutes is shown in Fig. 7.5. FGNNs in their current implementation seem to take 20–80% longer to train than comparable CNNs with the same number of trainable weights. When comparing based on the performance of the resulting network, FGNNs take 20–50% longer to train.

7.3 U-Net

Finally, to see how this approach may extend to other domains an network architectures, we test FGNN variants of U-Net [26] (FGNN-U-Net) on the ISBI 2012 challenge dataset. In this dataset, models must segment biomedical cells in an image. The problem is independent of rotations or reflections, and in 2015 U-Net had SOTA results on this benchmark/dataset. Additionally, U-Net is an architecture which makes heavy use of skip connections.

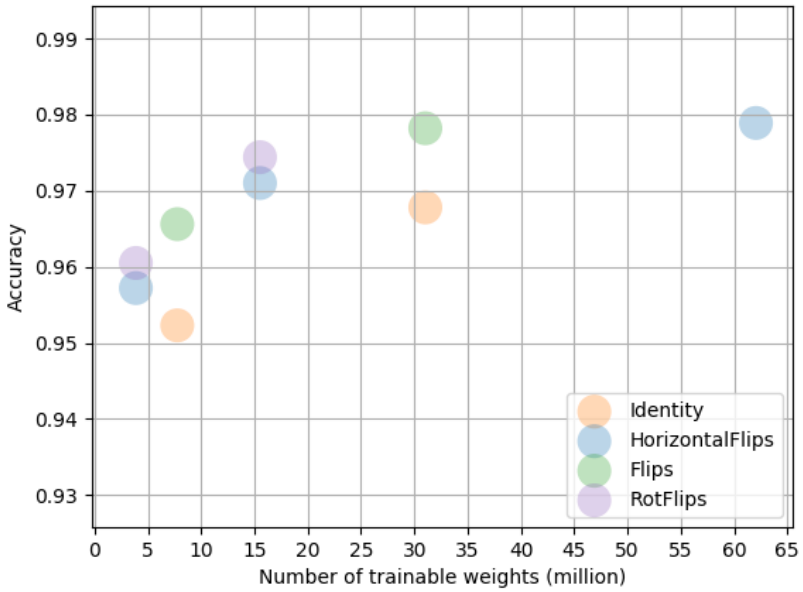


Figure 7.6: The performance of FGNN-U-Net variants of different sizes and symmetry groups compared to the baseline (Identity) model.

We create variants of FGNN-U-Net which are equivariant over different groups. These include horizontal reflections, vertical and horizontal reflections, and rotations of 90° and reflections (the D_8 group). By scaling up/down the number of filters in all the layers in the network, we also create variants with differing numbers of trainable weights. These can only easily be scaled by factors of two, doubling or halving the number of weights in the network. Due to this and the FGNN architecture it isn't always possible to create comparable networks over different groups. Variants which are equivariant to 3 groups are compared to the unmodified *Identity* U-Net in Fig 7.6.

8 Discussion

8.1 GGCNs

The game-graph networks are both outperformed by picking random valid moves, and are vastly outperformed by CNN architectures with equivalent numbers of trainable weights and layers.

An issue with this architecture is the ability to understand captures. A capture requires jumping over a piece and into an empty square beyond it. By representing the board as a graph, it is difficult for a node to work out what may lie 2 squares diagonally from itself. As such, the model is unlikely to be able to even determine valid moves, as evident in its poor performance.

8.2 FGNNs for Checkers

The affect of transforming the input according to some element of the group before feeding it to the network as compared to afterwards is shown to give the same results. The small difference between methods is likely to be a synonym of floating point maths rather than an error in the architecture. This shows that the network is equivariant in practice.

The results in Fig. 7.2 show that the FGNN-based models are stronger checkers players than baselines an equivalent numbers of trainable weights. Fig. 7.3 show the same on the training set, albeit less noticeable. The improvement FGNNs provide on the validation set seems to be present from small models up to the largest ones, and in all cases the most performant model is a FGNN.

The results in 7.4 demonstrate that FGNN models are able to marginally reduce overfitting. For every baseline model there is an equivariant model which outperforms it or is nearer to the dashed line denoting an ideal learner.

The training times shown in Fig. 7.5 are unfortunately somewhat noisy. Networks for each architecture seem to take the same time to train regardless of the number of trainable weights. This may mean that another factor is limiting their speed, such as cpu-gpu bandwidth or data preparation speed. Still, FGNNs of equal size seem to take %20–50 longer to train the same number of epochs when compared to baselines. This is unsurprising, as FGNNs introduce extra internal complexity. When compared against their resulting performance however, FGNNs are more competitive.

8.3 FGNNs for Biomedical Segmentation

Equivariant versions of U-Net outperform the baseline, and this performance improvement seems to scale with the larger groups. This shows that FGNNs can effectively be used in networks with skip-connections, a property that isn't possible in existing methods for creating equivariant networks. The fact that equivariance over larger groups further improves performance demonstrates that equivariant networks may show increased performance gains when used on problems which have more symmetries.

9 Conclusion

In this work we propose and evaluate two architectures which create symmetry equivariant learning algorithms for games.

The first, *Game Graph Convolutional Networks*, may not be a reasonable approach for creating these learning algorithms. The loss of spacial reasoning appears to make it hard for these networks to reason about the wider board state and even determine valid moves.

The second, *Finite Group Neural Networks* show far more promising results. A horizontally equivariant network reduces over-fitting and outperforms baselines at playing checkers. Additionally, FGNNs have a strong theoretical foundation and are arguably easier to reason about and extend than existing equivariant architectures. They are the first equivariant architecture which supports skip connections and arbitrary layer types. This is demonstrated by FGNN-U-Net, which also outperforms the unmodified U-Net network in the task of biomedical image segmentation.

Overall, equivariance is shown to improve the performance of neural networks for playing checkers, and may be prove to be a promising avenue of research in a variety of games. The methods presented may be readily adapted to improve the performance of neural architectures for Chess, Go, and Shogi, as well as a variety of wider learning tasks.

10 Summary

In this work we propose and evaluate two architectures which create symmetry equivariant learning algorithms for games.

The first, *Game Graph Convolutional Networks* (GGCNs), are an application of Graph Convolutional Networks to games. They create equivariance by representing the board state as a graph, in such a way that the graph representations of symmetrical or equivalent boards are isomorphic.

The second, *Finite Group Neural Networks* (FGNNs) are derived from the notion of T-Equivariance. Instead of creating equivariance to a group layer-by-layer directly, FGNNs enforce it over a more complex operation derived from this. FGNNs are the first neural architecture which can exhibit equivariance over complex move embeddings used for chess and checkers. Additionally the method used to create FGNNs can be applied to create equivariant versions of networks over arbitrary finite groups, as well as, for the first time, networks with arbitrary layer types and skip connections.

In the evaluation, while GGCNs' performance is lacking, FGNNs both reduce over-fitting and improve the performance of neural networks at playing checkers. This shows that equivariance over equivalent positions may be a useful property of board-game agents, and that FGNNs effectively exploit this equivariance.

Finally, to show the flexibility of FGNNs, equivariant versions of U-Net (FGNN-U-Net) are proposed, and outperform baselines on a segmentation dataset, even with 2–4 times fewer trainable weights.

11 Future Work & Limitations

The clear continuation of this work is FGNNs' applications to other games. While checkers was a decent choice to test a variety of approaches, without any external baselines the performance of models are hard to assess. In the future I would like to create equivariant versions of existing network architectures used for Chess and Go. Go in particular is a game equivariant over the full D_8 group so equivariant networks are likely to provide a significant improvement over current approaches.

One of the limitations of the work is that it inherently involves more operations per layer than non-equivariant networks. This slows down training and inference, especially for larger groups, and also scales linearly with the number of elements in the group (this is the same as existing work). However, there may be ways to improve this by using other representations of the group. Often a group can be faithfully represented as, for example, the permutation of a much smaller number of elements. This may provide significant improvements for the speed of FGNNs.

Another avenue of research is the impact of renormalization or dropout on FGNNs. This is not evaluated in this work, and multiple possible implementations would need to be compared.

For other applications of FGNNs, several works rely on RNNs to be equivariant over the order of input sequences [3, 8]. While an equivariant strategy exists in *DeepSets* [4] the application of T-Equivariance or a similar idea 'over time' may be an interesting avenue of research. Additionally, pose recognition works often produce a vector field DBLP:journals/corr/CaoSWS16 for the direction along limbs, which FGNNs may be able to elegantly model equivariance over. Many deep neural networks which use skip connections may similarly be a strong application for FGNNs [22].

Bibliography

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[3] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[4] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, pages 3391–3401, 2017.

[5] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[6] Benjamin Bloem-Reddy and Yee Whye Teh. Probabilistic symmetry and invariant neural networks. *arXiv preprint arXiv:1901.06082*, 2019.

[7] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[8] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[9] Haggai Maron, Heli Ben-Hamu, Nadav Shamir, and Yaron Lipman. Invariant and equivariant graph networks. *arXiv preprint arXiv:1812.09902*, 2018.

[10] Taco Cohen and Max Welling. Group equivariant convolutional networks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2990–2999, New York, New York, USA, 20–22 Jun 2016. PMLR.

[11] Sander Dieleman, Jeffrey De Fauw, and Koray Kavukcuoglu. Exploiting cyclic symmetry in convolutional neural networks. *arXiv preprint arXiv:1602.02660*, 2016.

[12] Marysia Winkels and Taco S Cohen. 3d g-cnns for pulmonary nodule detection. *arXiv preprint arXiv:1804.04656*, 2018.

[13] Daniel Worrall and Gabriel Brostow. Cubenet: Equivariance to 3d rotation and translation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 567–584, 2018.

[14] David W Romero, Erik J Bekkers, Jakub M Tomczak, and Mark Hoogendoorn. Attentive group equivariant convolutional networks. *arXiv preprint arXiv:2002.03830*, 2020.

[15] Jan Eric Lenssen, Matthias Fey, and Pascal Libuschewski. Group equivariant capsule networks. In *Advances in Neural Information Processing Systems*, pages 8844–8853, 2018.

[16] Robert Gens and Pedro M Domingos. Deep symmetry networks. In *Advances in neural information processing systems*, pages 2537–2545, 2014.

[17] Maurice Weiler, Fred A Hamprecht, and Martin Storath. Learning steerable filters for rotation equivariant cnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 849–858, 2018.

[18] Erik J Bekkers, Maxime W Lafarge, Mitko Veta, Koen AJ Eppenhof, Josien PW Pluim, and Remco Duits. Roto-translation covariant convolutional networks for medical image analysis. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 440–448. Springer, 2018.

[19] Taco S. Cohen, Mario Geiger, Jonas Köhler, and Max Welling. Spherical cnns. *CoRR*, abs/1801.10130, 2018.

[20] Taco S Cohen, Mario Geiger, and Maurice Weiler. A general theory of equivariant cnns on homogeneous spaces. In *Advances in Neural Information Processing Systems*, pages 9142–9153, 2019.

[21] Bart Smets, Jim Portegies, Erik Bekkers, and Remco Duits. Pde-based group equivariant convolutional neural networks. *arXiv preprint arXiv:2001.09046*, 2020.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015. cite arxiv:1512.03385Comment: Tech report.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks, 2016. cite arxiv:1603.05027Comment: ECCV 2016 camera-ready.

[24] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[25] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[26] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[28] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[29] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

[30] Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International conference on artificial neural networks*, pages 44–51. Springer, 2011.

[31] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in neural information processing systems*, pages 3856–3866, 2017.

[32] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[33] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. *arXiv preprint arXiv:1710.10121*, 2017.

[34] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 991–999, 2015.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[36] Richard Zhang. Making convolutional networks shift-invariant again. *arXiv preprint arXiv:1904.11486*, 2019.

[37] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[38] B. Fong, D. Spivak, and R. Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019.

[39] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[40] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[41] John Tromp. The number of legal go positions. In *International Conference on Computers and Games*, pages 183–190. Springer, 2016.

[42] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[43] Chien-Ping Lu. Parallel search of narrow game trees. 1993.