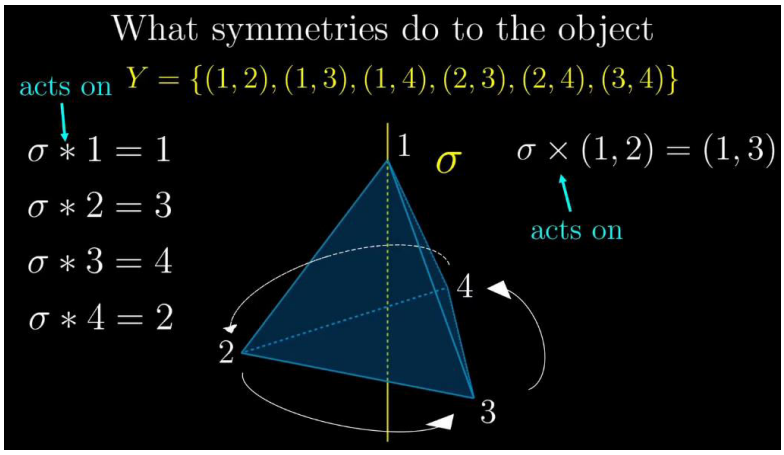
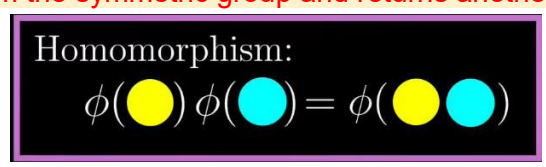
Preliminary Notes and Overview

- **Operation -** Ways of combining pairs of elements in a **set**
- **Group** - a **set** with an **operation**, sometimes called its **multiplication**
  - They describe the **symmetries of an object** (like a square, or R)
  - **Abelian Group -** group **G** with a composition operation that is **commutative** for all g, h in G
  - **Group Order (Cardinality) -** number of **elements** in the group's **set**
  - Group's operation shows how to replace any **two elements** of the group's set with a **third element** from the set in a useful way
    - e.g. the **group of integers with addition operation** replace two elements with their sum)
  - Satisfies the **4 Group Axioms** for a **group (G, *, e)**
    - **Closure -** for a, b in G, a * b is also in G
    - **Associativity -** a * (b * c) == (a * b) * c
    - **Identity -** for all a in G, a * e = e * a = a
    - **Inverse -** for all a in G, there is b in G, such that a * b = b * a = e
- **Injection / Injective Function -** f : A → B, where for every **b**, there is **at most one a** in domain
- **Surjection / Surjective Function -** f : A → B, where for every **b**, there is **at least one a** in domain
- **Bijection / Bijective Function -** f : A → B, where there is a **1-to-1 correspondence**
- **Composition -** $(f \circ g)(x) = f(g(x))$,
- **Symmetry** - **Invariant transformations** that **preserve distances** and the **object**
  - When interesting features (point positions/size/place) remain the same after a transformation
  - **Isometry -** When the points of a shape stay in the **same place**
    - Square has 8 isometries: <u>identity</u>, V/H flip, D/D', rotation 90, 180, 270
      - A **finite** group of **order 8**
      - A **non-abelian group -** makes a difference in which order we make transformations
  - We can have more symmetries depending on which features we find interesting
    - If position is not interesting, there is an infinite amount of symmetries
  - **Similarity -** two objects, that can have a different size, that have the same angles and proportional sides, i.e. things like squares of different sizes
- **Symmetry -** Symmetries move points around, so even if the shape looks the same, the indices change
  - Symmetries can be **noncommutative**, so order matters
  - Symmetry **composition g o h** is read **right-to-left**, so **h** is applied **first**, then **g**
  - Symmetries are **invertible**
  - *Since we can get rid of the shape of an object and just study the points, and how they move around under symmetries, it means that each symmetry corresponds to a permutation*
  - Used to study what symmetries do to an object
  - We can denote a **transformation** as **σ**
  - We can then say **σ * 1 = 1** to mean "sigma **acts on** pt 1 to move it to pt 1" (stays the same)
  - **σ** can also act on the sets of **edges** or **faces** to give the same set back
  - **σ** acts on something in the set to produce something **in the set**
  - **σ** is in the **group of symmetries**

- 

- **Permutation -** an ordering of indexes around an object that preserves the object and distances
  - A **symmetry corresponds to a permutation** of the vertices but **not** the other way round
    - If you simply permute the points, it doesn't have to be a symmetry (look at faces)
- **Symmetric Group Sym(X) -** set of ***every*** possible permutation of a **set X**
  - It's about permutations, **not symmetries**
  - **Permutation Group -** elements are permutations of a set, operator being the **composition** of ps
    - A **subgroup** of the **symmetric group**
- **Symmetry Group** - Set of all **invariant transformations** of an object where output is the same as input
  - E.g. rotating a square 90 degrees
  - Defines the structure of the domain **Ω**, a.k.a it's symmetry
  - It is the group of **permutations of n objects**
  - For images, it's the group of 2D translations that acts on points on **Ω**
  - **Operation of the group** on the points of the domain **is applied on the signals of the domain**
- **Group Action -** A group of symmetries, acts on a set of vertices/edges/faces, to give the same set back
  - It's important that the group action is a **homomorphism**
    - Allows us to use **isomorphism**
  - **Group homomorphism φ -** A **function** (aka map) that maps from a group **G** to a group **H**, such that for G = (R, +) and H = (R^+, *), φ(x + y) = φ(x) + φ(y)
    - Example is **$\varphi = e^x$**
  - A symmetry from a symmetry group takes a permutation from the symmetric group and returns another one.
    - Formulated as **φ : G -> Sym(X)**
      - **G -** group of symmetries
      - **X -** vertices
      - Sending **φ** any symmetry **g**, we get a permutation from **Sym(X)** symmetric group

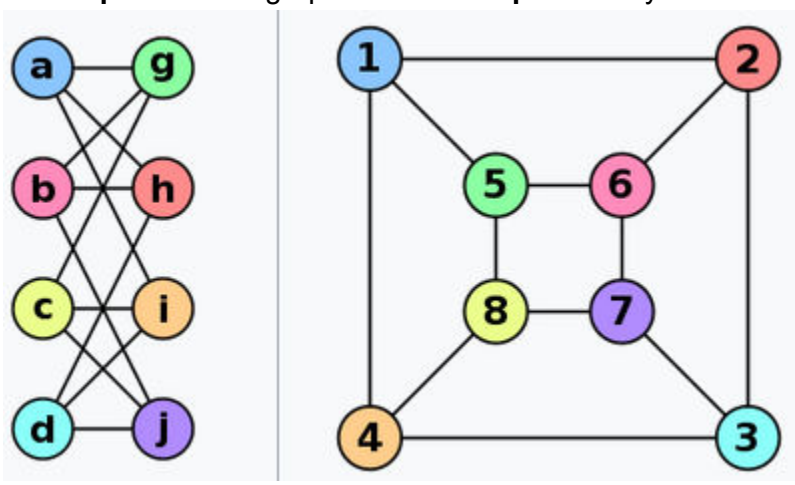


  - **φ** is a **homomorphism -** A permutation corresponding to **h** first then **g**, is the same as permutation corresponding to **hg**
    - $$\phi(h)\phi(g) = h^* g^* = (hg)^* = \phi(hg)$$
- **Morphism -** map between two structures
- **Invertible -** a **d*d matrix M**, for which there is an M^{-1} such that $MM^{-1} = I$
- **Group Representation -** a **group of symmetries** that has a **linear group action**, satisfying some axioms, meaning that we can think of it as warping **signals**, instead of just individual points on the domain
- We can exploit **physically-structured data**, by applying principles of **symmetry** and **scale separation**
  - Structure comes from the domain, and it is present in the inputs
  - **Ω -** Domain
  - **Signal -** functions on some **geometric domain Ω**, passed as input to a ML model
  - We can linearly combine signals (the signals form a vector space, called the *Hilbert space*)
  - **Hilbert Space -** a space with more than 3 dimensions with vector algebra that allows length and angle to be measured. It is also **complete**, meaning that you need to use **limits**
- **Invariance vs Equivariance**

- **Invariance -** when we transform something before passing it through a model, output **will not change**
    - If we translate a cat, it should still be labelled a cat
- **Equivariance -** when we transform something, output will change in the **same** way
    - If we translate a cat, segmentation pixels should translate the same way

- **Cartesian Product -** combination pairs of a set / two sets
    - For example, let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. Then:

      $$A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

- **Open set -** a set where all elements have the same properties
- **Topology -** study of how spaces are organised and how they are structured in terms of position
    - Studies how spaces are connected
    - No difference between square and circle, because they can be stretched into each other
    - A figure of 8 and a square are not the same though
- **Topological Manifold / Space T -** Set of points, with a set of neighbourhoods for each point describing closeness
    - They do not give distance
    - Neighbourhood = open set
    - **Axioms**
        - An intersection of two neighbourhoods is a neighbourhood
        - Empty set is in **T**
        - A union of any neighbourhoods is also a neighbourhood
- **Manifold** - A **topological space** that is/looks **locally Euclidean**
    - **n-Manifold -** something that looks like $\mathbf{R}^n$ up close, but might curve back on itself
    - A sphere, a structure that can be mapped by a series of 2D **maps**, is a **2-manifold**
    - Any object that can be **charted** is a manifold
    - **Map -** can have rules about making changes to structure around its edges, but they can't tear or overlap
- **Smoothness -**
- **Smoothness Function -**
- **Smooth / Differentiable Manifold -**
    - Infinitely differentiable, hence different form a normal manifold
- **Graph Isomorphism -** two graphs are **isomorphic** if they are the same graph represented differently
    - 

Overview

1

Representation Learning Architectures - discovering representations needed for **feature detection** or **classification**

- The work exploits symmetries in representation learning

2 - Learning in High Dimensions

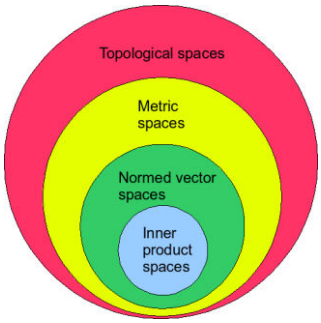
- **Parameterized Function -** a function that takes some input, but acts based on an **external constant**, i.e. model
- **θ ∈ Θ -** represents the network weights
- **Regularisation -** a technique for reducing overfitting by keeping the model function simple
- **Learnable Function Class -** set of functions for which an algorithm can be devised to minimise risk uniformly over all probability distributions
    - Related to **regularisation**

2.1 Inductive Bias via Function Regularity

- **Universal Approximation Theorem -** Neural networks (MLPs) can approximate any function

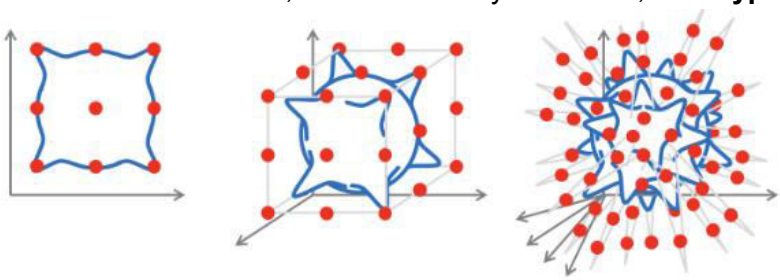
Universal Approximation, however, does not imply an *absence* of inductive bias. Given a hypothesis space $\mathcal{F}$ with universal approximation, we can define a complexity measure $c : \mathcal{F} \to \mathbb{R}_+$ and redefine our interpolation problem as

$$\tilde{f} \in \arg\min_{g \in \mathcal{F}} c(g) \quad \text{s.t.} \quad g(x_i) = f(x_i) \quad \text{for } i = 1, \ldots, N,$$



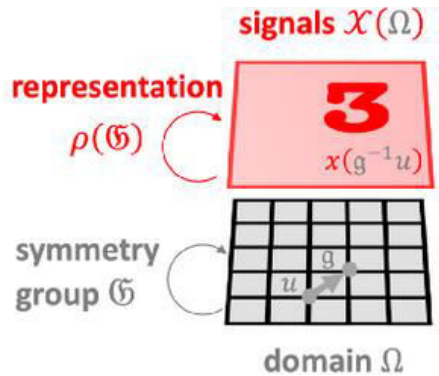
- **f~** is our **estimated** parametric function of the real f
- We want a **f~** that minimises some **complexity measure c(f~)** while keeping performance the same
- We are looking for the **most regular functions**
- **Differential Equation -** involves variables like **x** and **y**, and the rate at which they change
    - **Partial Differential equations -** contains more variables than just x and y
        - **Navier-Stokes Equations -** PDEs used to describe **fluid dynamics**
    - **Coupled Differential equations -** a world where several equations are obeyed at the same time
- **Mathematical Analysis -** Looks at functions, sequences and series
    - Study of continuous functions, differentiation and integration
    - Uses the concept of **limit**
- **Space -** A **set** with some added **structure**
    - **Metric Space -** a space with specific distances between objects
        - A **set of points**, where there is a **symmetric** function that gives you any pair's distance
        - **Cauchy Sequence -** sequence where distance between points progressively decreases
            - A sequence that tends to a point
        - **Complete Metric Space -** when every **Cauchy sequence** in M has a **limit inside M**
    - **Function Space -** a **set** of functions between **two fixed sets** (domain and codomain)
    - **Vector Space -** a **set** of vectors that can be multiplied by scalars
    - **Normed Vector Space -** a **vector space** over numbers, on which a **norm** is defined
        - **Norm -** a function on a vector to get nonnegative real numbers, **measures the size of something**
        - **Metric -** a function that measures the **distance between two things**
        - **Limit -** what a sequence tends to but never gets (except at the infinvalue)
    - **Numerical Analysis -** studying algorithms to get **approximations** for problems in mathematics
        - When the real solution is practically or actually impossible to obtain
        - **Functional Analysis -** Study of differential equations (PDEs) within numerical analysis
            - **Banach Space -** a vector space with a metric **(distance function)** that allows you to compute **vector length** and **distance**, and is complete in that a **Cauchy Sequence** always converges to a well defined **limit within the space**

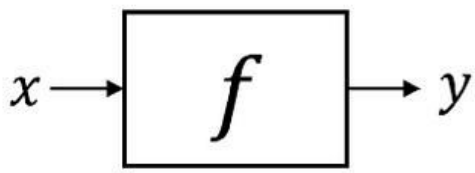
- **Complexity Measures**
  - In terms of **network weights**, $c(f_\theta) = c(\theta)$
  - **L2 Norm Regularisation of network weights (weight decay) -** we **add** a **regularising term** to our loss function, which is usually $\frac{\lambda}{2n}\sum_w w^2$, with **hyperparameter λ** between 0 and 1, 0 being no regularisation



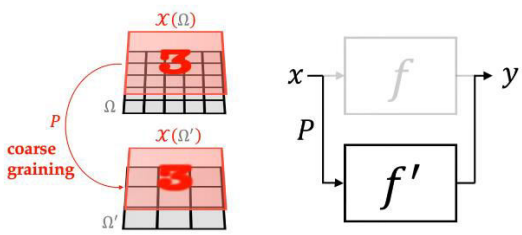
2.2 The Curse of Dimensionality

- **Interpolation** is difficult in high-dimensional problems
- **Class (Set Theory) -** a **collection of sets** that share some **property**
  - Sometimes used synonymously with **set**
- **Lipschitz function -** Functions of metric spaces where there is a constant **L** $|f(x) - f(y)| \leqslant L\,|x - y|$
  - **1-Lipschitz function -** Where **y** is the **derivative of x** (aka **x'**)
    - Results in **Local Smoothness -** if we change **x slightly**, **f(x)** can't change much
  - Once dimensionality grows, it takes **many observations** to prove that **f** is **1-Lipschitz**
- **Sobolev Space -** vector space of functions with derivatives that make the space **complete**, i.e **a Banach Space**
  - Curse of dimensionality also applies to Sobolev spaces
  - Characterised by **smooth functions**
  - **Sobolev class -**
    - Global smoothness hypothesis (an alternative to the Lipschitz class)
- **Neural Networks**
  - Neural networks define function spaces that enable more flexible notions of regularity, by considering **complexity functions** on their **weights**
  - **Sparsity-promoting regularisation** can break the curse of dimensionality
    - However, it is a **strong inductive bias** on the nature of **target f**
      - It suggests that **f** depends on a collection of **low-dimensional projections** of x
    - Functions tend to have complex long-range correlations that usually can't be expressed with LDPs
  - Instead, we can introduce alternative **sources of regularity**
    - **Spatial Structure** of the physical domain and **geometric priors** of **f**





3 Geometric Priors

- **Geometric Prior -** structure that comes from the **geometry** of the input signal
  - The input image is **no longer** just a **d-dimensional vector**
  - It's a **signal x(Ω)** defined on some **geometric domain Ω** (in the case of images, a 2D grid)
  - The **space** (set with structure) **of signals** is denoted by □**(Ω)**
- **Symmetry -** transformations that keep the image / segmentation the same when applied to input signal
- **Scale Separation -** ability to preserve important characteristics of a signal when transferring to a coarser domain
  - .e.g subsampling the image
  - We can assert that our function is **locally stable** if it can be approximated as a composition corresponding to the **coarse graining operator** (**P**) and the **coarse scale function f'**
    - Kind of like applying a classifier on a lower resolution image
  - 

    $f \approx f' \circ P$
- **CNNs**
  - **Convolutional filters** with shared weights exploit **translational symmetry**
  - **Pooling** exploits **scale separation**
- *We can show how other architectures like graphs and manifolds can utilise geometric priors*

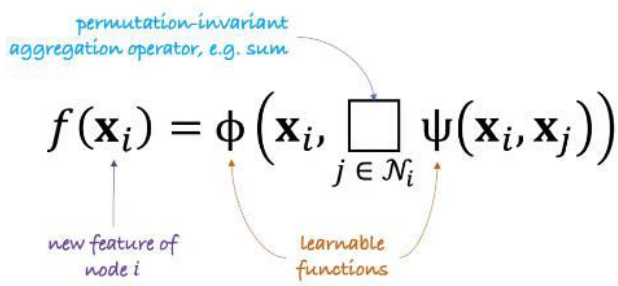
3.1 Symmetries, Representations, and Invariance

- **Symmetry -** invariant transformations
  - Can be smooth, continuous or discrete
    - **Discrete -** when something can be arbitrarily permuted (like particle systems)
- **Symmetry Groups -** set of symmetries of an object, with a composition operator
  - **Abelian group -** group that has **commutative** composition
  - **Group generator - basic elements** that are combined in a composition to create more elements
    - **Generator S - G** is **generated** by subset **S** if every **g** in **G** can be written as a composition of **S**
- **1D Translation Group -** generated by infinitesimal displacements
  - **Lie Group -** combines the ideas of a **group** and a **differentiable manifold**
    - Locally Euclidean space + defines abstract/generic concept of multiplication and inverses
- **Equivariant Function -** the output permutation changes in the same way the input permutation changes
- **Invariant Function -** the output stays the same regardless of input permutation
- **Group Action -** how the group acts on data **Ω** (translation points on plane), and obtain actions of the same group on the space of signals **X(Ω)**

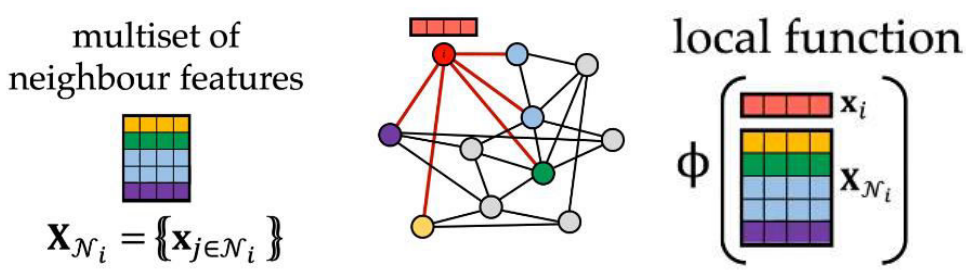
4 Geometric Domains: The 5 Gs

4.1 Graphs and Sets

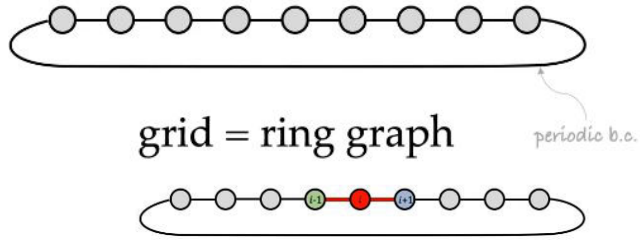
- Can be directed or undirected
- **Features -** nodes can have features which are **1-dimensional vectors**
  - *In more complicated models, edges can have features too*
- Graphs and sets **are unordered**
  - They can be **arbitrarily numbered**, then we can put them into a feature matrix or **adjacency matrix**
    - **Adjacency Matrix -** a truth or false matrix that tells us which two nodes share an edge
- **Invariant/Equivariant Graph Functions**
  - **Permutation Invariant -** The output of the function must be the same no matter the arbitrary numbering
    - E.g. classification
  - **Permutation Equivariant -** The permutation of the output of the function changes with input permutation
    - E.g. segmentation
- **Graph Neural Network -** we typically have a sequence of **permutation-equivariant** layers (**propagation / diffusion layers**) followed by a pooling layer
  - There are also **local pooling / graph coarsening layers**
- **General Blueprint for GNNs**

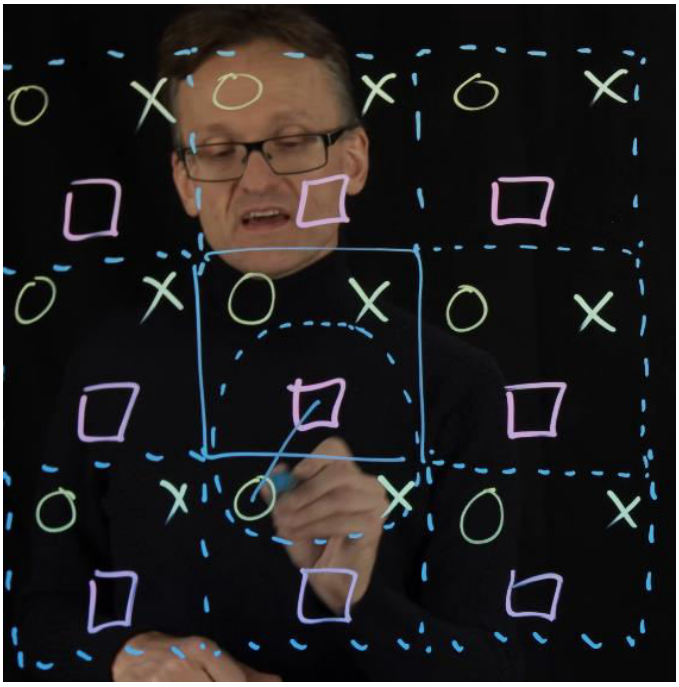


- **Message Passing Graph Neural Networks**
  - **Local Aggregation -** we look at the neighbours of each node in the graph
    - We make a **multiset** of their **feature vectors**
      - **Multiset -** a set that can have repeat elements
      - *Even though the indices of the neighbours are unique, their feature vectors don't have to be*



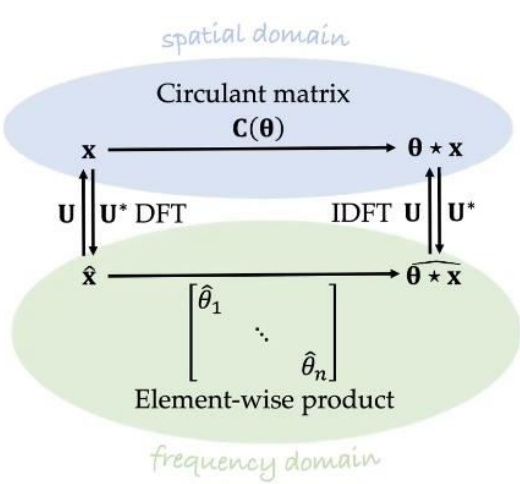
    -
    - **Φ - local aggregation** function that is **permutation equivariant**
    - We apply **Φ** for every node, then stack them into a matrix **F**
      - **Φ -** Learnable function that **updates** features of **node i** using aggregated features
    - We usually have a **permutation invariant aggregation operation**
      - Usually a **sum** or a **maximum**
    - **Ψ -** non-linear, learnable function that **transforms** neighbour features
      - It's output is seen as a **message** of node j to update node i





4.2 Grids and Euclidean Spaces

- **Grid -** a particular case of a graph
  - **Periodic Boundary Conditions -** a tile of smallest **unit cell** we need to simulate a large system
    - When an object passes through the boundary, it reappears on the opposite side with same velocity
  - **Ring Graph -** a grid with a **periodic boundary condition**
  - Grids have a **fixed neighbourhood structure**
  - They have a **fixed order of neighbours**
  - Since we have a prescribed order, we can have them **sequentially** as arguments instead of in a set, meaning that our function will no longer be permutation invariant/equivariant as order is important
  - **Grid Convolution -** linear local aggregation function
    - If we write a **convolution** as a **matrix vector**, we get a **circulant matrix**
    - **Circulant Matrix -** a **square** matrix, where each row is composed of same elements rotated by 1
      - Circulant matrices **commute** - $A^TB == B^TA$
      - Specially, they commute with a **shift operator** which has 1s in the start of the sequence
        - *Commuting with a **shift operator** means convolutions are **shift invariant***
          - *Therefore, convolution **emerges from translation symmetry***
      - **Commuting** matrices are **jointly diagonalisable**
        - **Diagonalising a matrix -**
        - **Eigen-value -**
        - **Eigen-vector -**
        - **Eigen-basis -**



        - **Jointly Diagonalisable** - a **common basis** for which all convolutions become pointwise multiplications
    - **Fourier Transform -**
    - **Discrete Fourier Transform -**
    - *All convolutions are diagonalised by the Fourier transform and the eigenvalues are given as a Fourier transform of this spectra(?) of the data that forms this convolution*
  - **Convolution Conversion Theorem in Signal Processing**
    - We can either perform a convolution by multiplying by a **circulant matrix**, corresponding to sliding a kernel along our signal, **or**, in the fourier domain, as element-wise product of fourier transforms of the signal and the filter, which is **efficient** because we can use **FFT** algs

4.3 Groups and Homogeneous Spaces

- *Convolution is like a sliding window pattern matching operator, by multiplying a patch with our filter*
- **Defining a convolution on a grid**
  - **Filter Ψ -** our kernel matrix
  - **Shift Operator T_u -** shifts the filter to position **u**
  - **Inner Product <a, b> -** scalar result vector dot p
    - How much of one vector is pointing in the direction of the other vector
    - Matches the filter to the signal (image **x**)
  - *This is sometimes known as the **convolutional correlation***

$$(x \star \psi)(u) = \langle x, T_u \psi \rangle = \int_{-\infty}^{+\infty} x(v)\psi(u-v)\mathrm{d}v$$

shift vector shift operator

- **Extension, defining a convolution on a group**
  - This is a **special case** where we **can identify** the **translation group** with the **domain**

$$(x \star \psi)(\mathrm{g}) = \langle x, \rho(\mathrm{g})\psi \rangle = \int_{\Omega} x(v)\psi(\mathrm{g}^{-1}v)\mathrm{d}v$$

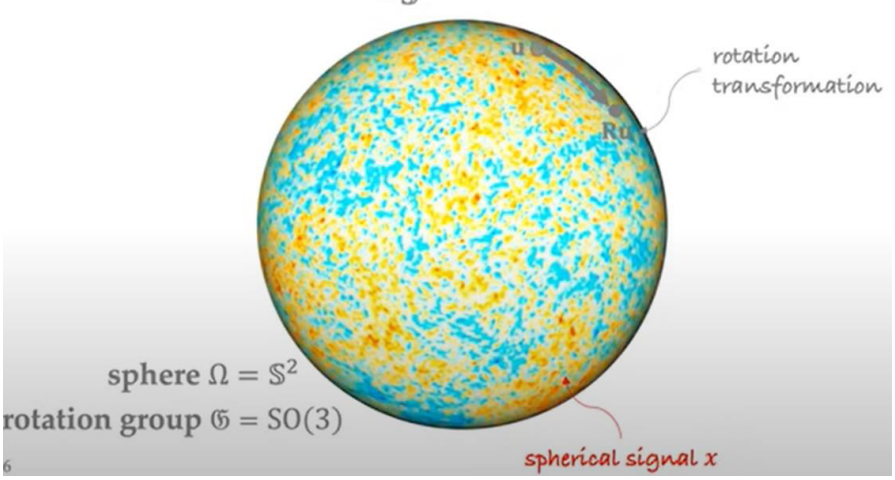
group element group representation

    - Every shift is represented by a **point on the domain**
      - Not a general case
    - In the general case, the filter is transformed by a **representation of our group ρ**
    - This will give us a group convolution for every element
    - We must assume that the groups are **small** (can't do it for a group of permutations)

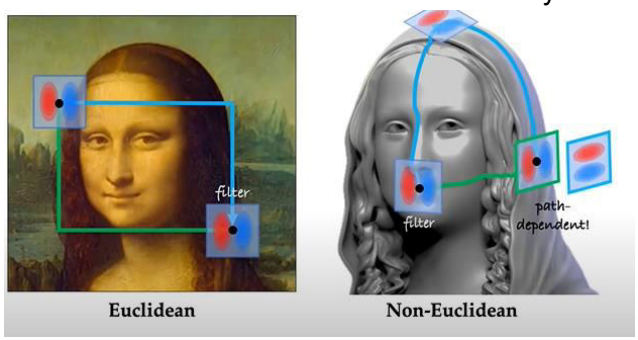
4.4 Geodesics and Manifolds

- **Extension, defining a convolution on a sphere**

$$(x \star \psi)(\mathrm{R}) = \int_{\mathbb{S}^2} x(\mathrm{u})\psi(\mathrm{R}^{-1}\mathrm{u})\mathrm{d}\mathrm{u}$$



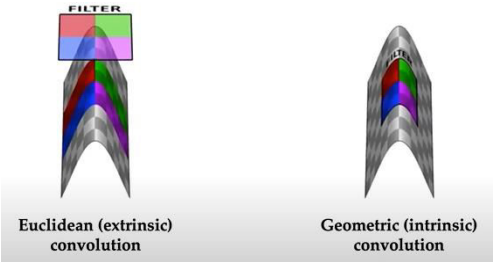
-

- Special orthogonal group **G = SO(3)**
    - Rotations that preserve orientation
    - **Orthogonal Matrix -** a matrix which has its transpose as its inverse
    - **Points Q** on the sphere are defined as **unit 3d vectors** (which are **rotations**)
    - The action of the group on the points is represented by an **orthogonal matrix R** with **det = 1**
        - Positive determinant = orientation is preserved
- The convolution is defined on **SO(3)** so we do **inner product** for every rotation **R**
- The input and output domain structure are different
    - Sphere is a **2-manifold**
    - Rotations are **3D**, the third being a rotation on itself
- Sphere = **homogeneous space -** For every **u, v** there is a **g in rotation group G** to go from u to v
        - Homogeneous spaces have a **global symmetry structure**
    - **Transitive action -** we say this group is transitive, i.e. the above
- **Parallel Transport** in **Differential Geometry -** when it comes to manifolds, if you are doing Euclidean translations, they aren't commutative, as your filter/object will rotate when it gets to the end point
    - We can't move a kernel on a manifold as you can on an image because they're only locally Euclid.
    - 

4.5 Gauges and Bundles

- **Tangent Space -** small neighbourhood of a point **u**
    - Equipped with additional structure, i.e. we can assign to them an **inner product**
        - **Riemannian Metric - inner product**, used to measure lengths, angles and volumes of a manifold
            - **Isometric Deformation -** manifold deformation preserving Riemannian metric
                - **Isometry -** metric-preserving deformation
                - They also form a group, **isometric group**
    - We can do convolutions on manifolds by doing them on points' **tangent spaces**
        - Using a local filter applied to a tangent space
        - If we do this construction **intrinsic**, or expressed in terms of the **metric**, we get what's called a **deformation invariance** with respect to the **isometric group**
            - Utilised by **geodesic CNNs**
    - Since we work locally, we **don't have global coordinate** system
        - **Gauge -** attaching a **local frame** to each point on a manifold
            - Came from Physics
        - **Gauge Transformation -** can **arbitrarily** transform a gauge at every point
            - **Structure Group -** Formed by a gauge transformation
                - The choice for the transformation depends on our assumption
        - **Assumptions for choice of gauge transformation**
            - **Manifolds** don't have much structure, so any transformation is possible
                - **G =** general linear group, an invertible matrix
            - If a manifold has **orientation**, we restrict the transformation **determinant** to be **positive**
            - If we have a **Riemannian metric**, we restrict the transformation to be **orthogonal**
                - **Orthogonal** transformations preserve **angles**
            - Manifold combining **orientation and metric**, we get special orthogonal group **SO(2)**
                - Rotations that preserve orientation
        - We need to **transform** the **filter** to **account for** the **gauge transformation** of a point

        - *This means that the **filter** is **gauge equivariant***
    - We care about **manifolds** because in computer vision and graphics, manifolds are the standard way to model 3d objects
        - Using **geometric perspective**, we get **intrinsic filters** which are **equivariant** to our surface, so the filter curves with the surface as if it was on it, instead of hovering
        - Gives us **invariance to inelastic deformation**
    - Used in **correspondence of deformable 3D shapes**

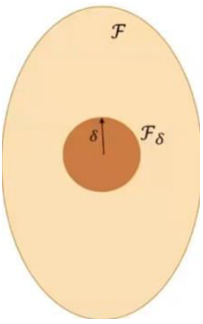


Online Lectures

Lecture 2 - Learning in High dimensions

- **Statistical Learning -** extracting information from possibly high dimensional a
- **Data Distribution -**
- **Approximation Model -**
- **Error Metric -** choose good models from bad models
- **Estimation Algorithm -** some procedure to find the estimate

- Assumptions in the data and our models are very important
  - If there is no assumption from the distribution or the target there's no way we can generalise
- **Model / Hypothesis class / Function Approximation**
  - Subset of functions
  - Mappings that go from input space **X** to target space
  - Examples
    - Polynomials, Linear regression, Neural networks
  - **Complexity Measure γ** - a **norm** / quantity that is meant to organize, and divide a hypothesis into being simple or complicated
    - E.g. number of neurons in a NN
    - E.g. in **harmonic analysis**, **Sobolev Norm -** if it's small, the function is simple
      - **Sobolev simple** = the function is smooth
- **Error Metric -** how far we are from the ground truth
  - Given point-wise convex measure, we can consider the average
  - **R(f) Population Average - expectation** for the point wise measure (average)
    - This is the mean error of the **whole data distribution v**
  - **R^(f) Empirical Average -** training loss, replacing expectation over data with **empirical expectation**
    - This is the mean error of the **training data**
    - A random quantity depending on the draw of the training set
  - In machine learning, we want to make the **population error small** but we only have access to **empirical**
  - If we **fix** the hypothesis **f**, **empirical avg** is just the **avg of the IID quantities** (distribution of the inputs)
  - **Empirical avg** is the **unbiased estimator** of the **population avg**
  - We can compute the variance bound of our loss using **uniform bounds**, e.g. **Rademacher complexities**
- **Empirical Risk Minimisation**
  - **Goal -** minimize **R(f)**, which is **deterministic**, having only access to **R^(f)**, which is **random**



    - We have to control the distance between R(f) and R^(f)
    - Instead of considering the **whole hypothesis space F**, we only consider those that have a complexity that is not too large
      - Only consider hypotheses that are **within some kind of max complexity norm δ from our f***
      - $\mathcal{F}_\delta = \{f \in \mathcal{F}; \gamma(f) \leq \delta\}$
    - **Empirical Risk Minimisation** - taking the **f** from **F_δ** that produces the smallest **empirical risk R^(f)** $\hat{f}_\delta = \arg\min_{f \in \mathcal{F}_\delta} \hat{\mathcal{R}}(f)$
      - **Constraint Form -**
        - $f \in \mathcal{F}_\delta$ is a **convex constraint** that may not be easy to use in practice
      - **Penalised Form -** $\arg\min_{f \in \mathcal{F}} \hat{\mathcal{R}}(f) + \lambda\gamma(f)$
        - An alternative to the constraint form where you use a **Lagrangian multiplier** where the constraint now becomes part of the optimisation objective
        - **Complexity δ** - hyperparameter that controls **regularisation** strength, keeps **cmpxty low**
      - **Interpolation Form -** $\arg\min_{f \in \mathcal{F}} \gamma(f) \; s.t. \hat{\mathcal{R}}(f) = 0$
        - Popular with large networks
        - You take the hypothesis with **lowest complexity that fits your training data**

        - You can only do this if you have **no noise** in the data / labels
    - *We can combine these ERM forms to give some guarantee of learning*
- **Minimum -** smallest actual value of a set
- **Infimum -** the **greatest lower bound -** it is the **lower limit** that may not be reached, i.e. 1/n, the infimum is 0
- **Supermum -** the **lowest greater bound -** it is a **upper limit** that may not be reached
- **A priori -** known before experience
- **A posteriori -** known after experience
- **Basic Decomposition of Error -** We break up error into numerous errors so that we can interpret them differently
    - We start from our **arbitrary hypothesis f^ in F_δ** with some **complexity δ**
    - We try to give a **guarantee**, we try to say that given a **hypothesis**, what is the **risk?**
        - I.e. what is the **population error R(f^)** that this hypothesis is going to increase?
    - **Steps**

1) $\mathcal{R}(\hat{f}) - inf_{f\in\mathcal{F}}\mathcal{R}(f)$

- We subtract the **baseline**, the **infimum $inf_{f\in F}R(f)$** which is the **lowest test error**
    - *If you had infinite computational resources, you can select best hypothesis, that's infimum*

2) $\left(\mathcal{R}(\hat{f}) - inf_{f\in\mathcal{F}_\delta}\mathcal{R}(f)\right) + \left(inf_{f\in\mathcal{F}_\delta}\mathcal{R}(f) - inf_{f\in\mathcal{F}}\mathcal{R}(f)\right)$

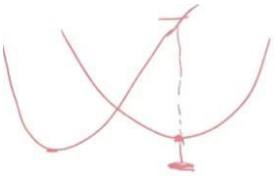
- We break down the error by introducing the term that describes the **minimum obtainable err in F_δ**
- The red term $\left(inf_{f\in\mathcal{F}_\delta}\mathcal{R}(f) - inf_{f\in\mathcal{F}}\mathcal{R}(f)\right)$ is the **difference** between minimum error in **F** and in **F_δ**
    - **Approximation Error -** how well we can approximate target **F*** with small complexity δ
        - the error introduced by restricting our complexity norm δ
        - Gets smaller as we increase δ
        - Is a **pure term** as there's no empirical data involved, it's across the whole space
            - Therefore, it must be **approximated**

3) $\left(\hat{\mathcal{R}}(\hat{f}) - inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f)\right) + \left(\mathcal{R}(\hat{f}) - \hat{\mathcal{R}}(\hat{f})\right) + \left(inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f) - inf_{f\in\mathcal{F}_\delta}\mathcal{R}(f)\right) + \varepsilon_{appr}$

- Introduce **training**, we will add+subtract infimum of **training error**, we don't just add+subtract infimum of test error
- $\hat{\mathcal{R}}(\hat{f})$ **-** training error
- $inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f)$ **-** best training error given our max complexity norm
- $\hat{\mathcal{R}}(\hat{f}) - inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f)$ **- optimisation error -** ability to efficiently solve empirical risk minimisation
    - *It captures how much does our current hypothesis cost if we could instead solve the empirical risk minimisation perfectly*
- $\left(\mathcal{R}(\hat{f}) - \hat{\mathcal{R}}(\hat{f})\right) + \left(inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f) - inf_{f\in\mathcal{F}_\delta}\mathcal{R}(f)\right)$
    - Something that **compares** the **population objective** with the **training objective**

$$= \left(\hat{\mathcal{R}}(\hat{f}) - inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f)\right) + \left(\mathcal{R}(\hat{f}) - \hat{\mathcal{R}}(\hat{f})\right) + \left(inf_{f\in\mathcal{F}_\delta}\hat{\mathcal{R}}(f) - inf_{f\in\mathcal{F}_\delta}\mathcal{R}(f)\right) + \varepsilon_{appr}$$

$$\leq \varepsilon_{opt} + 2sup_{f\in\mathcal{F}_\delta}|\mathcal{R}(f) - \hat{\mathcal{R}}(f)| + \varepsilon_{appr}$$



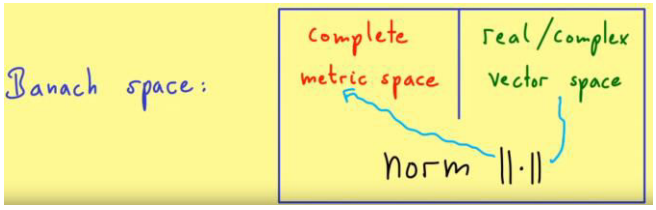
- We can **upper bound** the two terms above
- The **2** comes from the two terms in black
- If we have **two minimisers**, we can upper bound the difference by their difference at minima

$$= \varepsilon_{opt} + \varepsilon_{stat} + \varepsilon_{appr}$$

-
    - **We now get the above**
    - Our **test error** is a contribution of three different sources - **optimisation, statistical, approximation error**

- **Statistical Error** - term penalising uniform fluctuations over F_δ between true function and random function
- **Dense Network** - if the **infimum** in the whole hypothesis space **R(f) is = 0**
- How can we solve all sources of error at the same time?

- **Curse of Dimensionality** - first became a term in dynamic programming
  - Synonymous with statistical high dimensional statistics
  - **Principle of learning - interpolation** - finding patterns in things that are similar or nearby
    - This suffers a lot in high dimensions
  - **Lipschitz Function** - a function that has a **derivative lower** than some **hyperparameter ρ**
    - Tells me that the value of a function at one point is not going to be far from another point
    - Limits the frequency / amplitude of the wave $|f(x) - f(y)| \leq C\,|x - y|$
    - Even if we know the **target function f*** is n-Lipschitz, we still need an **exponential number of samples** to estimate it

- **How many numbers do we need to estimate target f*** ?
  - **Upper Bound of error**
    - We have **n samples** x_i, $f^*(x_i)$
    - *We now have to prove that **estimator** will do well given **n is large***
    - Our **hypothesis space F = {f : R^d -> R^j}**, set of all functions that map from dD -> jD real numbers
      - We will also assume that **f** is **bounded** and **Lipschitz**
      - **Bounded Function** - function with a bound, e.g. **|f(x)| < Lim**
      - As a result, **F** is a **Banach Space**
      - **Banach Space** - a **complete normed space (X, || .))**
        - Because it's **normed**, it naturally has a notion of complexity
          - Norm is actually the **Lipschitz constant ρ**, smaller **ρ** = simpler function
        - **Set** of numbers, with the **normal metric**, every **Cauchy sequence has a limit**
        - 

    - We will define the **estimator ERM** in the **interpolant form**
      - $\hat{f} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \{ \operatorname{Lip}(f);\ f(x_i) = f^*(x_i)\ \forall i \}$
      - **Estimator f^** - function **f** where **f(x_i) == f*(x_i) for all x~v**, with **lowest Lipschitz constant**
        - **Interpolant Form** - a model that passes through all training points
        - $|\hat{f}(x) - f^*(x)| \leq |\hat{f}(x) - \hat{f}(x_{i_0})| + |\hat{f}(x_{i_0}) - f^*(x_{i_0})| + |f^*(x_{i_0}) - f^*(x)|$
        - The above is our **computed error** between the **estimator** and the **ground truth**
        - We picked an **x~v** (sample **x** taken from distribution **v**), and its NN, x_{i_0}
          - These should be close, as we have the Lipschitz assumption
        - **|f^(x) - f*(x)|** - this is our error
        - Our error is **bounded** by the terms:
          - 2: **f^(x_{i_0}) - f*(x_{i_0}) = 0**, assumption is that our estimator goes through all **x** (train. set)
          - 3: **f*(x_{i_0}) - f*(x) = ρ**, because our assumption is that f* is Lipschitz
          - 1: **f^(x) - f^(x_{i_0}) = ρ**, since f^ is an interpolant of f*, it's also Lipschitz
        - **Hence, |f^(x) - f*(x)| = 2 || x_{i_0} - x ||** $2\,\|x_{i_0} - x\|$
          - We could get rid of the f^ and f* because we can factor them out

- **Lower bound of error**
  - $$\mathbb{E}_x \left| \hat{f}(x) - f(x) \right|^2 \leq 4\, \mathbb{E}_x \| x - x_{i_0} \|^2$$
    - We square the 2 to get 4, in this example the Lipschitz constant is 1 but it could be higher
  - $$\underbrace{\mathbb{E}_x \| x - x_{i_0} \|^2}$$
    - **Wasserstein Distance** - A quantity of how **far a new sample** is from **closest** training sample
      - $$\underbrace{\mathbb{E}_x \| x - x_{i_0} \|^2} = 4 W_2^2(\nu, \hat{\nu}_n)$$
      -
      - The distance on the left is the **square Wasserstein distance** of the ground truth gaussian distribution **v** and it's empirical version **v^$_n$**, where n is the number of samples?
      - This distance is of the order $n^{-1/d}$
        - Well known distance
        - Therefore, if we want $n^{-1/d} = \varepsilon$, it implies that $n = \varepsilon^{-d}$, meaning it's **exponential**
          - **Extra:** Think about why this number of samples is sufficient **and** necessary, and why we can't learn with less number of samples than **exponential**
- **Curse in Optimisation**
  - Finding global optima in high dimensional functions is **NP-hard**
  - In most **real life problems**, the topology of the loss w.r.t parameters is kind of like **mounts ranges**, with most paths leading down to a **good minimum**
    - This means that there's "**no bad local minima**"
  - Computing local minima is **relatively easy**
  - This means we can make use of **gradient descent** to find local minima efficiently
    - **Gradient Descent -** finding a point that is an **approximate second order stationary point**
      - This is another way of saying finding a **local minimum** that has **approximation error ε**
      - With gradient descent, to achieve this error, we need a number of iterations that is of the order $\tilde{O}(\beta/\varepsilon^2)$ iterations, where **Õ()** is a way of saying it is hiding log factors for dimensions, which is a low complexity dependency
      - This means our number of iterations **only scales with the error we want to achieve**
      - *This means even if our dimension is really high, this complexity won't change*
    - **Efficient -** this relates to iteration complexity
- **Summary**
  - **Lipschitz class is too large -** statistical error cursed by dimension because you have to evaluate all pts
  - **Sobolev / Barron classes too small -** approximation error cursed by dimension
    - We make the problem too easy and it destroys the original question
  - *We need to think about functions outside of the box*
  - Exploit the underlying **low-dimensional structure** that is hiding inside of a **high-dimensional space**
    - Such as a group, a grid, a graph, a mesh, etc
    - Geometric domains provide new notions of regularity for more efficient learning
  - High dimensional learning **impossible without assumptions** due to **curse of dimensionality**
  - **Classic regularity assumptions** are too weak / strong

Lecture 3 - Geometric Priors I

- **Domain -** things like grids, graphs, manifolds, etc
- **Signal -** data that lives on a domain

Recap

- **Supervised learning** in high dimensions is **intractable -** the number of samples required grows exponentially
- **Geometric Deep Learning -** exploit **structure of spaces of signals** to make high dimensional learning tractable
- Three sources of error in learning:
  - **Approximation -** if your considered **function class is too small**, and the true class is **far outside**
    - From this perspective, you'd like a **large function class**, to **maximise** likelihood of **finding f***
    - Measures **how much inductive bias** do we have
    - Doesn't depend on sample size
    - Under the **realizability assumption**, approximation error is **0**
      - In the **agnostic case**, it can be large
  - **Statistical -** given a **finite sample**, you are **not likely** to find the right function **inside the function class**
    - From this perspective, want **small function class**/many samples, **maximise** likelihood **finding f***
    - This error relates to the error between error on the **training set vs. all data**
  - **Optimisation -** given a **finite sample**, tells us how good we are at finding the right **local optimum** in search space / function class
    - Deep learning solves this problem quite well even if we don't fully understand it

- **Geometric Priors -** geometric priors **respect symmetries of our problem**, decreasing its size / complexity hopefully without discarding useful hypotheses

Geometric Domains

- **Domain Ω -** data lives on domains, e.g. 5 Gs: grids, groups, graphs, manifolds, gauges
  - Domains are **always a set**, but they may have different kinds of structure
    - **Grid -** set with a **neighbourhood structure**
    - **Graph -** set with a **connectivity structure** (possibly with a **metric structure**)
    - **Manifold -** set with **metric structure** for measuring distance on the surface
  - *Popular AI structure implicitly assume structure of the domain to work well*
- **Signal -** a function that takes a **point on a domain** to output a **point on a vector space**
  - **x : Ω -> C**, a **function** that takes **input as element of Ω domain**, and outputs vector in **vector space C**, with dimensions called **channels**
  - Example: our domain **Ω** can be a **grid**, which is a **cartesian product** of two sets to create coordinates, which is mapped to the vector space R^3, representing the **rgb values**
  - **- data space** of the **C signals** on the domain **Ω**
    - The **set of all signals**
    - $\mathcal{X}(\Omega, \mathcal{C}) = \{ x : \Omega \to \mathcal{C} \}$
    - It is a **Hilbert Space**, meaning we can always have the notion of length and angle between signals
      - It has an **inner product** and a **measure**
      - This means we can add not just signals that are images, but also those that are graphs whose structure doesn't allow for addition
      - **We can add signals and multiply by scalars**: (ax + by)(u) = ax(u) + by(u)
      - *a*, *b* are scalars (e.g. transformations), *u* is a point (e.g. pixel) and *x*, *y* is data (e.g. images)

We can add signals and multiply by scalars:

$$(\alpha x + \beta y)(u) = \alpha x(u) + \beta y(u), \qquad \text{where } \alpha, \beta \in \mathbb{R} \text{ and } u \in \Omega$$

- The space of signals is a **vector space**, which means we can compute inner products on a pair
  - $\langle x, y \rangle = \int_\Omega \langle x(u), y(u) \rangle_{\mathcal{C}} \, d\mu(u)$
  - *The **inner product of two signals** (e.g. two images), is given by the integral over the domain, of the inner product of the vectors of the **two signals x, y at a point u** (e.g. pixel position), with respect to the measure **μ(u)**, which is usually simply the **counting measure** of u, or the number of pixels that we have*
    - **Counting Measure -** counting measure is a simple measure on a set, gives the size
    - When we use the counting measure, the integral just becomes a **sum over omega**
    - We want to be able to carry out inner products so that we can do pattern matching by comparing our signals to some kind of filter / convolution

Fields of Geometric Features

- **Function -** Our **signals x** are functions on **Ω**, such that they take a point **u** on **Ω**, and output **x(u) ∈ C**
  - For example, an image takes a pixel position and outputs an **rgb vector** for every pixel
- **Field -** a generalisation of a function

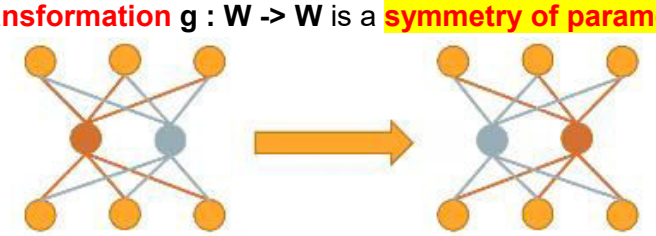
Domain as Data

- Usually, our data are the signals on our domain
- Sometimes, like with **meshes** and **point clouds**, our data is the domain
- **Solution**
  - **Graph Adjacency Matrix - n x n matrix**, where **n** is number of nodes, with **1s to represent edges**
    - This way, we can think of this matrix as a **signal** on a domain that is **Ω x Ω**
  - **Metric Tensor -** If you have a manifold or mesh, you can use a metric tensor to define length and angle between tangent vectors, meaning you can view them as **signals on the nodes Ω**
  - *There are methods to convert a domain into a signal of a different, more complex domain*

Symmetries in General

- **Symmetry -** of an **object**, is a transformation that leaves it unchanged
- **Symmetry Group -** the set of symmetries that we can compose to end up in the same place we started

Symmetries of the Parameterization

- Slightly more abstract because the word **object** is more abstract in Geometry
- **X =** input space, **Y =** label space, **W =** weight space
- **Model f : X x W -> Y** (e.g. NN)
- **Transformation g : W -> W** is a **symmetry of parameterization** if **f(x, g(w)) = f(x, w)** for all **x** and **w**
- 

  Swapping the incoming and outgoing connections of two neurons in the same layer does not change the input-output map $f(\cdot, w)$

Symmetries of the Label Function

- **X =** input space, **Y =** label space
- **Ground Truth Label Function** **L : X -> Y** mapping inputs to outputs
- **Transformation** **g : X -> X** is a **symmetry of the label function** if **L o g = L**
    - **g** is a function that transforms the input. We say that it is a symmetry if the **pre-composition of L and g** *(if we apply the transformation first, then compute the label)* outputs the same labels as **L**
- L() = L() = "dog"

Symmetries of Structured Domains

- Learning is about learning about the symmetries of our problem
- If we know all of the symmetries of a class, we can start at one point of the class and get to every other point by applying symmetries until we permute through every point
- If we knew these symmetries **a priori**, then the problem would be trivial
- What do we do if we don't know these symmetries?
- In a large class of problems in geometric deep learning, a lot of symmetries come from the **domain of our data**
- We say that a transformation **g : Ω -> Ω** is a **symmetry** if the **structure of Ω** is **preserved** (very vague)
- E.g.
    - **Permutation -** set membership is preserved
    - **Euclidean isometries -** rotation, translation, reflection preserve distances and angles in metric space
    - **General Diffeomorphism** (smooth warping) **-** preserving smooth structure of a **smooth manifold Ω**
- **Symmetry -** what you define it as depends on what is the **structure of your domain**

Groups of Symmetries

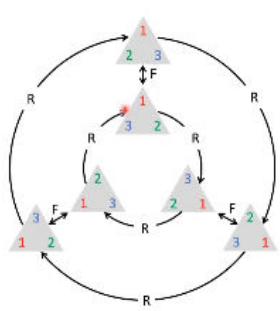
- We can consider the set of all symmetries of a single object / domain **Ω**
- The **identity transformation** is always a symmetry
- The **composition** of two symmetries is also a symmetry
- The **inverse** of a symmetry is also a symmetry

Symmetry Groups, Abstract Groups and Group Actions

- **Symmetry Group -** we formalize the set of symmetries as a group **G** with a **binary operation** **gh**
    - They satisfy **properties**: **associativity** (order), **identity**, **inverse** (equals e), **closure** (gh is element)
    - Elements are transformations **g : Ω -> Ω**
    - The **group operation** of the **symmetry group** is the **composition of maps**
- **Abstract Groups -** if we have an abstract group, we can't assume that we our elements are functions (maps), so we need some sort of a **composition rule** that **satisfies the group axioms**
    - **Group Action -** this gives us a **group action** that takes an element of **G** (a symmetry) and a point on the domain **Ω** to give us a new point on **Ω** satisfying axioms

Cayley Diagrams and Tables

- A way to represent how we can compose different symmetries together

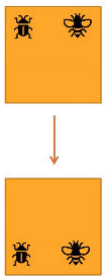


	id	R	R^2	F	FR	FR^2
id	id	R	R^2	F	FR	FR^2
R	R	R^2	id	RF	RFR	RFR^2
R^2	R^2	id	R	R^2F	R^2FR	R^2FR^2
F	F	FR	FR^2	id	R	FR
FR	FR	FR^2	F	FRF	FRFR	$FRFR^2$
FR^2	FR^2	F	FR	FR^2F	FR^2FR	FR^2FR^2

Kinds Of Groups

- **Discrete Group**
  - **Finite Groups** - like the group of rotational symmetries
  - **Countably Infinite Groups -** such as translations of the set of integers
- **Continuous & Lie Group**
  - **Compact groups -** 2D rotations
  - **Locally Compact Groups -** continuous translations
  - **Non-Locally Compact Groups -** Diffeomorphisms of manifolds
- All of these groups have **commutative** (*order doesn't matter*) or **non-commutative** (*order does matter*) cases

Symmetries of Ω acting on signals X(Ω, C)



- **Group Action -** a group of symmetries we can apply
  - $(\mathfrak{g}\, x)(u) = x(\mathfrak{g}^{-1}u)$
    - This makes sense, we can either move the signal forward, or we can bring the point back
  - The signal **x** transformed by **g**, evaluated at point **u**, is the same as the signal **x** evaluated at point **g^{-1}u**
  - **Linear Group Action = Group Representation**
    - They satisfy **linearity:** $\mathfrak{g}\,(\alpha x + \beta y) = \alpha \mathfrak{g} x + \beta \mathfrak{g} y$

Symmetries: Sets & Graphs

- **Group G = S_n - symmetric group** of all permutations on n elements
- **Domain Ω -** set of nodes or vertices or edges
- There are three features (different representations of the symmetric group **S_n**)
  - If we are classifying the graph, the output of the classifier is a single number which won't change
    - E.g. the output becomes **1 * s**, i.e. stays the same

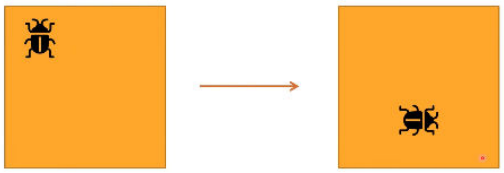
$$\rho_0(P_{12})s = 1 \cdot s \qquad \rho_1(P_{12})v = P_{12}v \qquad \rho_2(P_{12})M = P_{12}MP_{12}^T$$

Scalar feature e.g network output in graph classification | Vector feature e.g. one feature per node | Tensor feature e.g. one feature per edge or node pair

Symmetries of Graphs: Objects & their Descriptions

- A graph or a set is an **abstract object**
  - It's just a description of the set with some extra properties (like order)
    - We are generally more interested in the symmetries of this description (of the properties) than of the object itself
    - For example, if we have the **adjacency matrix** of a graph, we want our neural network to be **equivariant** to the **permutations** of the matrix, as they describe the same graph
      - Some graphs have the same adjacency matrix, meaning their representation is the same. This is different, it's a symmetry of a particular instance, which should also be equivariant

Symmetries: Grids



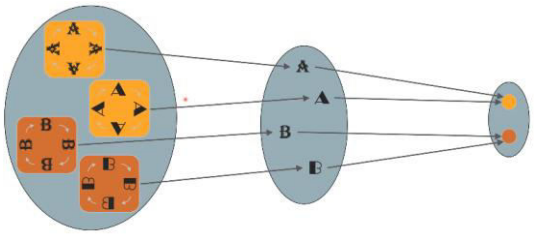
- **Groups G -** discrete translations, discrete rotations, flips
- **Domain Ω = V**, the grid points
- **Regular Representation -** $(\mathfrak{g}\, x)(u) = x(\mathfrak{g}^{-1}u)$
  - We can shift and rotate images

Symmetries: Groups & Homogeneous Spaces

- This can be generalised to the setting of general groups and homogeneous spaces
- Any groups that can be locally compact
- **Homogeneous Space -** for any two points in the space, there's min one **symmetry** that maps one to the other

Symmetries: Manifolds (Geodesics & Gauges)

- **Group G -** Gauge transformations
    - Changes in the **reference frames** of the feature spaces
    - *B-automorphisms of a principal bundle*

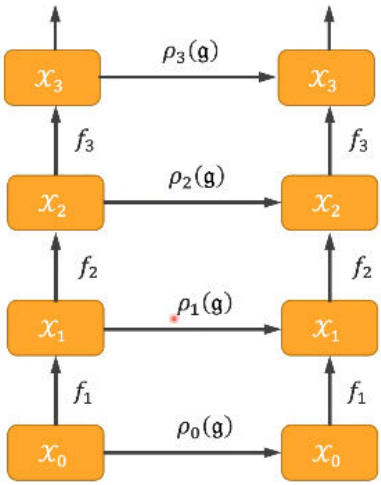


Problem: with Invariance in Deep Learning

- We might be tempted to make our model **invariant to symmetries**, for example, if we want to classify letters, we might want to form some **invariant representation** where each rotated version of a letter is represented by the same feature vector
- In **deep learning**, to recognise an object, we first need to recognise **individual parts**
- If we make the intermediate representations **invariant**, we lose critical information
    - *The **relative pose** of object parts contains critical information*
- Above, we may instead want to make the model invariant to the representation of the object as a whole

Solution: Equivariant Networks

- We have an equivariant network, if we can show that at each layer of a network, if we transform the input with a symmetry before putting it through the layer, we get the same output
    - If the above is true, we can also show that their composition satisfies this property as well

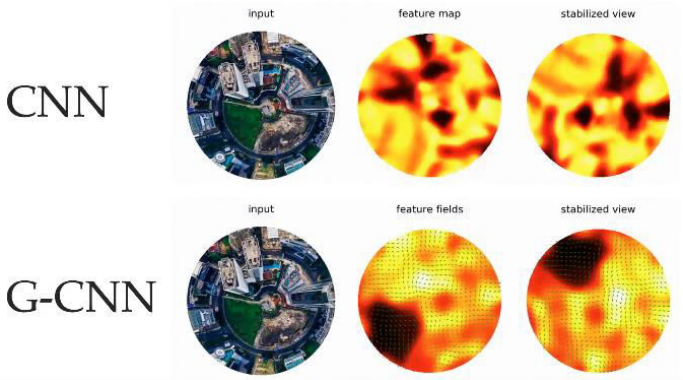


Translation Equivariance in CNNs

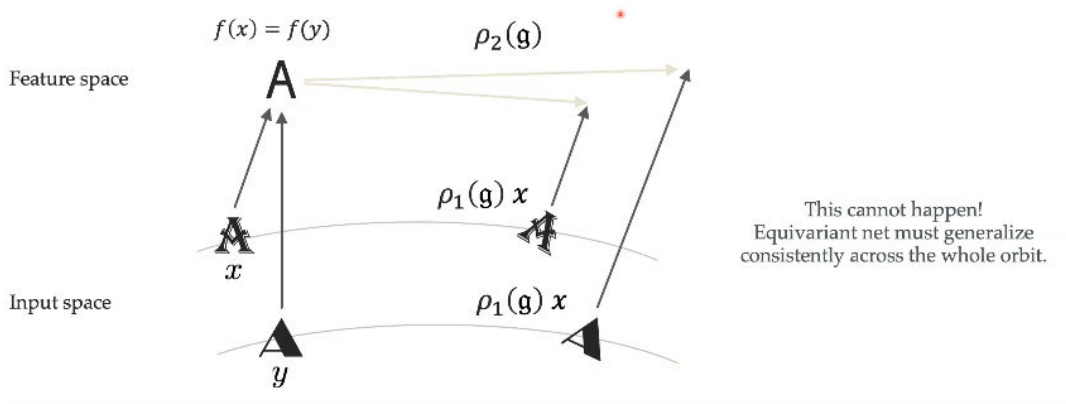
- Standard convolutional networks are not rotation equivariant
- CNNs can learn approximate equivariance in their weights

Rotation Equivariance in CNNs

- The CNN feature maps change with rotation even if we stabilise the view
    - This is as opposed to the G-CNN which stays the same



Equivariance: $f(\rho_{i-1}(\mathfrak{g})\,x) = \rho_i(\mathfrak{g})\,f(\mathbf{x})$



Equivariance as Symmetry-consistent Generalisation

- The above diagram is not an example of equivariance because if we translate the input before passing it through the model, the outputs end up in a different place than if we first pass it through the model, then translate it
- Each font creates a **manifold**, an **orbit** of all rotated variations of itself
- An equivariant network is generalised in a way that's consistent with the symmetry
    - *If x and y map to the same points, their transformed versions should also map to the same points*

Equivariance vs Data Augmentation

- Why do we not use data augmentation? We know that it is used in AI solutions, especially unsupervised learning
- There are related advantages and disadvantages
    - Equivariant networks perform 10x on medical data that is truly have translational and rotational symmetry
    - This could be because equivariant networks are equivariant **layerwise**, not just on the whole net
- They require more thought to implement
- We can do test time data augmentation for equivariance by applying all augmentations to test input, then average
- We can't use train time data augmentation as we can't show every input combined with every transformation so the network won't end up being equivariant
- Data augmentation **doesn't** work with **large symmetry groups**, e.g. graphs have **n! factorial large** symmetry group which us too large to rely on data augmentation
-

Property	Train augmentation	Test augmentation	Equivariance
Layerwise or whole-net constraint	Whole-net	Whole-net	Layerwise
Easy to implement, simple			
Guaranteed in/equivariance at training data			
Guaranteed in/equivariance at test data			
Works for large (e.g S_n) / infinite (e.g. SE(n)) groups			
Efficient at train time			
Efficient at test time			

Summary

- **Symmetries** are transformations that leave objects **invariants**
    - **Groups** formalise properties of the **set of symmetries** of an object (axioms such as associativity, identity, composition, closure on inverses)
- In **MLAI**, we care about **symmetries of parameterization, label function** (especially those that arise from the domain) **and domain**
    - **Symmetries of domain** act **linearly** on the **space of signals** on the domain, via a **Group representation**
- To exploit symmetry in **MLAI**, we use **equivariant networks**
    - **Equivariant network -** each feature space is associated with and equivariant w.r.t a group representation
        - **Invariance -** a special case where **trivial representation** is used

Lecture 4 - Geometric Priors II

Invariant Function Classes

- In MLAI, we want to learn an unknown function **f* : X -> R**, and **F** is our hypothesis class
- We can open up the input space as $\mathcal{X} = \{x: \Omega \to \mathcal{C}\}$
  - Allows us to look at **X** as a **space of signals** defined over a **domain Ω**, this helps reduce dimensionality
  - The domain can capture structure with interesting transformations that associate with signals
    - E.g. if the domain is a **set**, there's a natural transformation that allows it to **permute** elements
    - E.g. if the domain is a **euclidean space**, we can apply transformations and rotations
  - The transformations can be combined together
  - This action, combined with the set of symmetries, is called a **group**
  - The **linear transformation** of this group form the **group representation**

    $\mathfrak{g} : \Omega \to \Omega \quad \Rightarrow \quad \mathfrak{g} : \mathcal{X}(\Omega) \to \mathcal{X}(\Omega)$

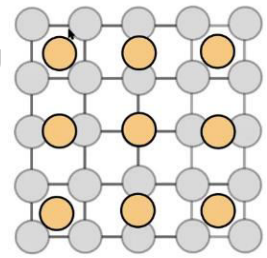
    - $(\mathfrak{g}\, x)(u) = x(\mathfrak{g}^{-1} u)$
    - **Group representations** allow us to look at transformations as warping signals instead of just coordinates on the domain
      - If we have a **transformation g** that transforms one point on **domain Ω** to another point, we can look at this **g** as it transforming signals instead of just coordinates
- Our promise, when learning, is that **f*** is **G-invariant** for all **x in X(Ω), g in G**
  - This means that **f*(g . x) = f*(x)**
- **Group-smoothing Operator (G-smoothing)** - it takes a hypothesis **f** from **F** and replace it with the **average** of the **group G** (*the average hypothesis over all possible transformations*)
  - $S_{\mathfrak{G}} f \stackrel{\text{def}}{=} \frac{1}{|\mathfrak{G}|} \sum_{\mathfrak{g} \in \mathfrak{G}} f \circ \mathfrak{g}$
  - This means that we average the prediction for each input over the **orbit of the group** (all possible transformations of an image)
    - **Group Orbit G.x -** {g . x; g ∈ G}
- If the **G-smoothing hypothesis** is true, if we apply the group-smoothing operator to **f***, it won't change
  - **S_Gf* = f***
- We can apply this hypothesis to our **hypothesis class F** to restrict our hypothesis space
- **Approximation error** is not affected by **G-smoothing**
  - $inf_{f \in \mathcal{F}} \|f - f^*\|^2 = inf_{f \in S_{\mathfrak{G}}\mathcal{F}} \|f - f^*\|^2$
  - The **approximation error** tells us that given our complexity restriction, what is the difference between our best possible hypothesis and the true function f*?
  - The reason why **g-smoothing** doesn't affect the approximation error is because a g-smoothed hypothesis is an **orthogonal projection**
- **Statistical error** is **reduced** by **G-smoothing**
  - Because our hypothesis space is smaller
  - We can measure the improvement quantitatively
  - $\mathbb{E}\mathcal{R}(\tilde{f}) \lesssim (|\mathfrak{G}|\, n)^{\frac{-1}{d}}$
  - It improves by the order of the **size** of the **group G**
  - The generalisation error is **upper bounded** by the number of samples times the size of the group
  - **Group size** can be exponential in dimension (e.g. all local translations)
  - **Group invariance** doesn't fix the curse of dimensionality because we still have the **-1/d** term, meaning if we want to halve the error, we have to increase n or |G| by an exponential amount
- Using **global symmetries** will give us better statistical error without loss in approximation error = **no brainer**
  - Guarantees improvements in sample complexity
- However, **insufficient to break curse of dimensionality**
- How do we build **invariant classes**?

Scale Separation

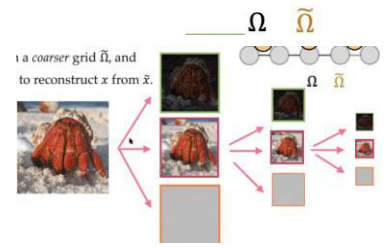
- Deep learning research suggests that it works because of **compositionality -** giving representations and meanings to parts that make up the whole input
  - How do we formalize this?
  - Looking at science, we usually try to understand things at different scales, making the scales smaller to add complexity as we learn more about the model

Basics of Multiresolution Analysis

- For simplicity, we can fix our domain **Ω** to be a 2D-grid
  - We can think of multi-resolution analysis on all domains
- **Multiresolution analysis -** breaking up a signal that lives in a **complicated domain Ω** in terms of a signal **x~** that lives in a **coarser domain Ω~**
  - E.g. it's like going from a high resolution domain to a smaller resolution while keeping all of the information
  - We **decompose** the image into a smaller image, plus all of the details that we need to go back
  - We do the **coarsening** through filters that are **localised in space**
    - **Wavelets -** filters that are localised in space



- Fundamental tool in signal processing in **Fourier analysis**

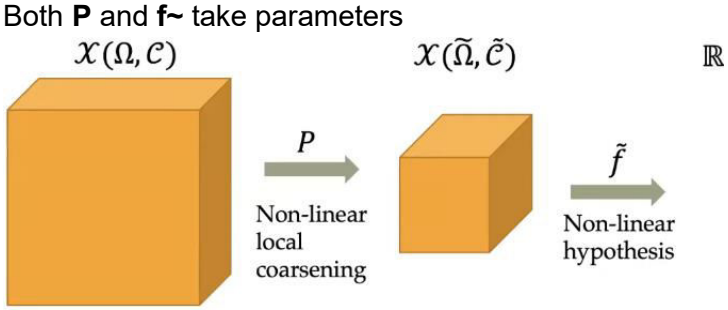


Scale Separation Prior

- **Coarse scales**
- We can make an **inductive prior** by supposing that **f*** is such that **f*(x) ≈ f*~(x~)**
  - This means that **f*** can be well-approximated by a coarser function **f*~** that takes a coarser, non-native input x to determine the right answer
  - E.g. a problem where we can smooth an image and still be able to classify it
- If we can do this, the complexity of our domain becomes much smaller, which reduces the curse of dimensionality
- This is because **d** in the number of samples we need to get some performance is determined by the size of our input, e.g. number of pixels in input, which is directly proportional to the complexity of our domain
- $\dim(\mathcal{X}) \propto |\Omega|$ and $|\widetilde{\Omega}| \ll |\Omega|$
- This is usually not possible, problems are usually too complicated to just smooth over our data, so this is usually too much of a **strong assumption**

- **Local fine scales**
- We can also say that **f*** can be well approximated by a **sum of local terms** $f^*(x) \approx \sum_u g(x_u)$
  - Each input $\mathbf{x_u}$ is a patch of our image
- In this case, local scales dominate the classification problem, so we reduce the size of our input to just a patch
- In general, this is also a **strong assumption**

- **Composition model**
- Ideally, we would rather make a **composition** of these two assumptions
  - We want to capture the data we have in both coarse and fine scales
- We think about approximating **f*** as a **composition** of two operators: **P** and **f~**
  - **P - nonlinear** local coarsening with some structure to approximate more complicated patterns
  - **f~ - nonlinear** function that extracts information at coarse scales, more complicated than averaging out
- Both **P** and **f~** take parameters



GDL Blueprint

Combining Invariance with Scale Separation

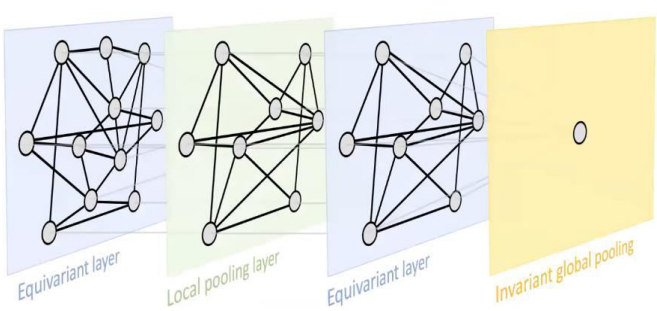
- We want our **function hypothesis class F_δ** to be:
  - **G-Invariant -** output same result regardless of which **g** we apply to **x**
  - **Multiscale Structure -** incorporates multiscale structure using local fine scales and coarse scales
  - **Rich Approximation -** potentially good performance, best solution in F_δ close to **f***
- *How do we design such an architecture?*
- **Invariants**
  - We start with **linear G-invariants**
  - Loses lots of information, as $\bar{x}$ becomes the **group average over the orbit**
  - Below, we can move **f** out of the sum as it is **linear** a priori

    $$f(x) = \frac{1}{|\mathfrak{G}|}\sum_{\mathfrak{g}} f(\mathfrak{g}.x) = f\left(\frac{1}{|\mathfrak{G}|}\sum_{\mathfrak{g}} \mathfrak{g}.x\right) = f(\bar{x})$$

    $$\text{Group average } Ax \stackrel{\text{def}}{=} \bar{x} = \frac{1}{|\mathfrak{G}|}\sum_{\mathfrak{g}} \mathfrak{g}.x$$
  - Loses a lot of information!
- **Equivariants**
  - We use **linear G-equivariants**, an operator that **commutes** unlike an invariant $B(\mathfrak{g}.x) = \mathfrak{g}.B(x)$
  - How do we extract invariants from equivariants?
  - Compose **linear G-equivariant** with a **nonlinear element-wise** function **p** to get non-linear G-equivariant
    - $\boldsymbol{\rho}: \mathcal{X} \to \mathcal{X}, with\ \boldsymbol{\rho}x(u) = \rho(x(u))$
      - Nonlinear, so each point is being computed separately

Geometric Deep Learning Blueprint

- We build **rich invariants with multiscale structure**, using building blocks:
  - *Linear $\mathfrak{G}$-equivariant layer* $B : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{X}(\Omega', \mathcal{C}')$, satisfying $B(\mathfrak{g}.x) = \mathfrak{g}.B(x)$ for all $\mathfrak{g} \in \mathfrak{G}$ and $x \in \mathcal{X}(\Omega, \mathcal{C})$.
  - *Nonlinearity* $\sigma : \mathcal{C} \to \mathcal{C}'$ applied element-wise as $(\boldsymbol{\sigma}(x))(u) = \sigma(x(u))$.
  - *Local pooling (coarsening)* $P : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{X}(\Omega', \mathcal{C})$, such that $\Omega' \subseteq \Omega$.
  - *$\mathfrak{G}$-invariant layer (global pooling)* $A : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{Y}$, satisfying $A(\mathfrak{g}.x) = A(x)$ for all $\mathfrak{g} \in \mathfrak{G}$ and $x \in \mathcal{X}(\Omega, \mathcal{C})$.
  - **Linear layers** are easier to understand
  - We compose linear layers with **element-wise non-linearity**
  - We combine the **non-linear equivariance** with **coarse graining** in multi-resolution analysis
  - We can extract **invariance** by applying **group smoothing**
- General blueprint that can be applied to any domain



Architecture	Domain Ω	Symmetry Group $\mathfrak{G}$
CNN	Grid	Translation
Spherical CNN	Sphere / SO(3)	Rotation SO(3)
Intrinsic / Mesh CNN	Manifold	Isometry Iso(Ω) / Gauge Symmetry SO(2)
GNN	Graph	Permutation Σ_n
Deep Sets	Set	Permutation Σ_n
Transformer	Complete Graph	Permutation Σ_n
LSTM	1D Grid	Time warping

- We can visualise what these networks are doing as a **composition of simple tools**

Summary

- Geometric priors can be talked about in terms of **symmetries** and **scales together** and allow us to define **hypothesis spaces** that can break the **curse of dimensionality**
- **Group invariance** is useful to define architectures through the language of **invariance** and **equivariance**
- One of the roles of **depth in NNs** is to propagate information to the **coarse scales**
- **Scale separation** helps us avoid the problem of processing all of the pixels together (curse of dimensionality)

Lecture 5 - Graphs and Sets I

Permutation invariance and equivariance and how it relates to processing data in graphs and sets

Recap

- We use a set of guiding principles:
  - **Symmetry -** a **transformation** that we want our model to be invariant to
  - **G-Equivariance - layers** that are equivariant to transformations of a certain **symmetry group**
    - They give us an output for each point on the domain
  - **G-Invariance - global pooling layers** aggregate all info and give a **single** answer for the whole domain
  - **Locality - convolutional layers** that work on the patches of a domain
  - **Scale Separation - Local pooling layers** can help you coarsen the domain
- Using these principles, we can steer the design of deep learning architectures to avoid dimensionality

  Let Ω and Ω' be domains, $\mathfrak{G}$ a symmetry group over Ω.
  Write $\Omega' \subseteq \Omega$ if Ω' can be considered a compact version of Ω.

  We define the following building blocks:

  Linear $\mathfrak{G}$-equivariant layer $B : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{X}(\Omega', \mathcal{C}')$,
  satisfying $B(\mathfrak{g}.x) = \mathfrak{g}.B(x)$ for all $\mathfrak{g} \in \mathfrak{G}$ and $x \in \mathcal{X}(\Omega, \mathcal{C})$.

  Nonlinearity $\sigma : \mathcal{C} \to \mathcal{C}'$ applied element-wise as $(\boldsymbol{\sigma}(x))(u) = \sigma(x(u))$.

  Local pooling (coarsening) $P : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{X}(\Omega', \mathcal{C})$, such that $\Omega' \subseteq \Omega$.

  $\mathfrak{G}$-invariant layer (global pooling) $A : \mathcal{X}(\Omega, \mathcal{C}) \to \mathcal{Y}$,
- satisfying $A(\mathfrak{g}.x) = A(x)$for all $\mathfrak{g} \in \mathfrak{G}$ and $x \in \mathcal{X}(\Omega, \mathcal{C})$.
- **Linear G-Equivariant layers** satisfy linearity and resistance to **group action**
- We introduce **nonlinearity** through **point-wise nonlinearities** like sigmoids, tans, relus
- This gives us **universal approximators** over the domains which is sufficient
- Sometimes we coarsen the domains to go from **Ω** to **Ω~**
- We use a **global pooling layer** to get an answer over the whole domain
- Using this, we can derive popular MLAI architectures

Starting with Graphs

- They have a **discrete** domain with minimum geometric assumptions
  - This makes them easy to analyse
  - If you squint hard enough, all domains can be seen as a graph
    - Architectures can be seen as an instance of a graph neural network

Processing data that lives on graphs

- We first look at (unordered) **sets -** graphs without edges
- It's a simpler domain, so it's simpler to analyse the architectures
- It's still very relevant - e.g. point clouds and LIDAR

Setup

- Graph with no edges, so **Ω = V**, the set of nodes
- We assume that each **node x** will have **k features**, $\mathbf{x_i} \in \mathbf{R^k}$, this is our feature space ($C = R^k$)
- We can stack the features of nodes into a **node feature matrix** **X** = |V| * k
  - This assumes some **order** to the nodes, even if it's unordered
  - We need to make sure that the model doesn't depend on the order of the rows
- This means that if we permute the order in which we give the model our nodes, the output is the same
  - E.g. **Symmetry Group G -** n-element permutation group $\mathbf{\Sigma_n}$
    - The different group elements **g** ∈ **G** are permutations

Permutation and permutation matrices

- There are **n!** permutations for a domain
- **Permutation -** an operation that changes the node order
  - $\text{permutation } (2, 4, 1, 3) \text{ maps } \mathbf{y}_1 \leftarrow \mathbf{x}_2, \mathbf{y}_2 \leftarrow \mathbf{x}_4, \mathbf{y}_3 \leftarrow \mathbf{x}_1, \mathbf{y}_4 \leftarrow \mathbf{x}_3$
  - Each permutation defines a **n*n permutation matrix** with **n** 1s representing new positions for each node
  - $$\mathbf{P}_{(2,4,1,3)}\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \mathbf{x}_2 & - \\ - & \mathbf{x}_3 & - \\ - & \mathbf{x}_4 & - \end{bmatrix} = \begin{bmatrix} - & \mathbf{x}_2 & - \\ - & \mathbf{x}_4 & - \\ - & \mathbf{x}_1 & - \\ - & \mathbf{x}_3 & - \end{bmatrix}$$

Permutation Invariance

- Functions **f(x)** that operate over these node features that won't depend on the order of the nodes
  - E.g. if we apply a permutation **p** from **P**, it shouldn't change the result
- **f(PX) = f(X)**
- This is similar to our definition of a **G-Invariant layer (global pooling)**
- **Permutation invariance** is good if we want to have an **output over the entire set**

Deep Sets

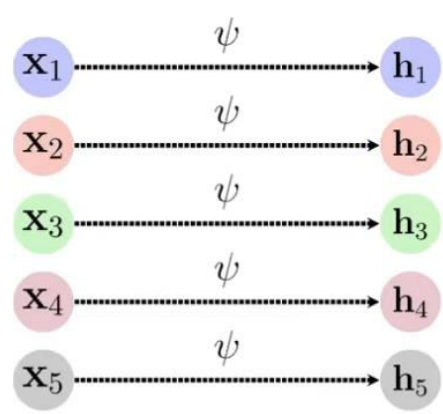
- **Deep Sets Model -** a model for representation learning that's permutation invariant
- $$f(\mathbf{X}) = \phi\left(\sum_{i \in \mathcal{V}} \psi(\mathbf{x}_i)\right)$$
- **ψ** and **φ** are learnable functions, e.g. MLPs
- We can substitute the **sum** operation for an **avg** or **max**, etc
  - The sum is important because it introduces **permutation invariance**
  - Since that operation can be any permutation invariant aggregation operation, we call it $\oplus$
- It applies a point-wise MLP **ψ** on every single node in isolation
- **Aggregate** all of the nodes' outputs and then pass it to another MLP **φ to perform the nonlinear activation**

Permutation Equivariance

- **Permutation equivariance** is good if we want to get an **output over individual nodes**
- It is a function **F(X)** where no matter what permutation we use, it doesn't matter if we apply it at input/output level
  - Capital F to make it clear we are using with a matrix, not a single vector/node
- This is the same as a **linear G-Equivariant Layer**

Important Constraint: Locality

- We want the signal to be **stable** under slight deformations of the domain
- We want to be resistant to slight noise in the data
- One method to do this is to compose **local operations** to model larger ones
  - Any **errors in a local operation** will **not be propagated globally**
    - E.g. super deep 3x3 CNNs



Locality on Sets

- We want an **equivariant layer** to be local as above
- We can have a **shared function ψ** that is applied to each node in isolation
- The output of each **$\psi(\mathbf{x}_i) = \mathbf{h}_i$**, we can stack all **h** to form a matrix **H = F(X)**

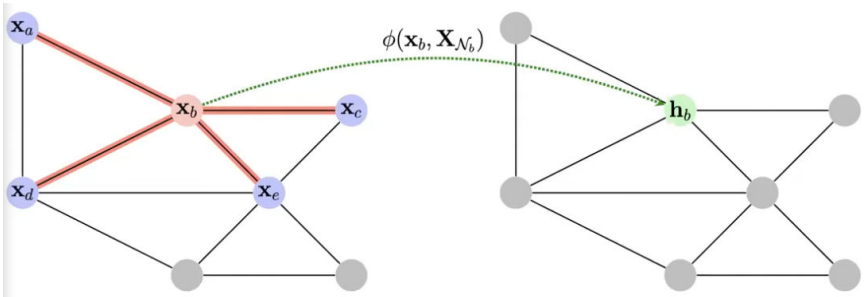
Learning on Graphs

- *How do we generalise invariance, equivariance and locality to graphs?*
- Graphs are just sets with edges **G = (V, E)**, where **E is the cartesian product of V x V**
- Represent the edges within a graph with an **adjacency matrix A**, which has $\mathbf{a_{ij} = 1}$ if nodes **i and j** share an edge

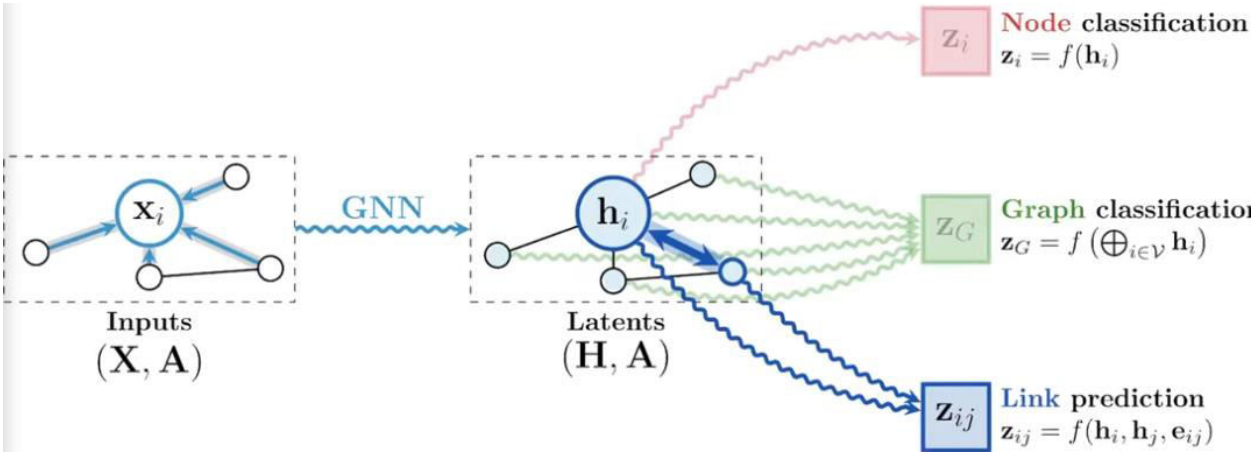
Permutation Invariance and Equivariance on Graphs

- When we permute our nodes, we need to permute the edges in the same way
- Permutations now act on the **edges**, i.e. the **adjacency matrix**
  - This means that permutations permute the rows and columns of the adjacency matrix at the same time
  - Therefore, when applying a **permutation P**, we permute **rows** and **cols**, leading to $\mathbf{PAP^T}$
  - $$\textbf{Invariance:} \quad f\left(\mathbf{PX}, \mathbf{PAP}^{\mathrm{T}}\right) = f(\mathbf{X})$$
    $$\textbf{Equivariance:} \quad \mathbf{F}\left(\mathbf{PX}, \mathbf{PAP}^{\mathrm{T}}\right) = \mathbf{F}(\mathbf{X}, \mathbf{A})$$
- The only difference here compared to sets is that we need to be mindful of the extra adjacency matrix for edges

Locality on Graphs: Neighbourhoods

- We define a **local function φ** that give us **broader context** - the node's **neighbourhood N**
- We can choose the **hop size**, how big of a neighbourhood we look at (*default at 1-hop*)
- $\mathcal{N}_i = \{ j : (i,j) \in \mathcal{E} \vee (j,i) \in \mathcal{E}) \}$
- We can take the features for these nodes and put them into a matrix $\mathbf{X_N}$:
  - $\mathbf{X}_{\mathcal{N}_i} = \{\{ \mathbf{x}_j : j \in \mathcal{N}_i \}\}$
- We then define the **local function $\varphi(x_i, X_{N\_i})$** that operates over all of the nodes in the neighbourhood in **isolation**, by taking a node as one argument, and its neighbourhood as the other
- **Permutation Equivariance**
  - As in sets, we get equivariant functions by applying **local function φ** to every single node+neighbourhood
  - We then stack the results into a feature matrix
  - $$\mathbf{F}(\mathbf{X}, \mathbf{A}) = \begin{bmatrix} - & \phi(\mathbf{x}_1, \mathbf{X}_{\mathcal{N}_1}) & - \\ - & \phi(\mathbf{x}_2, \mathbf{X}_{\mathcal{N}_2}) & - \\ & \vdots & \\ - & \phi(\mathbf{x}_n, \mathbf{X}_{\mathcal{N}_n}) & - \end{bmatrix}$$
  - **φ** is invariant if the ordering of the node features in $\mathbf{X_N}$ doesn't change the output of the
  - For **F(X, A)** to be **equivariant**, **φ** needs to be **invariant**. If we can prove the latter, we prove the former.
  - 

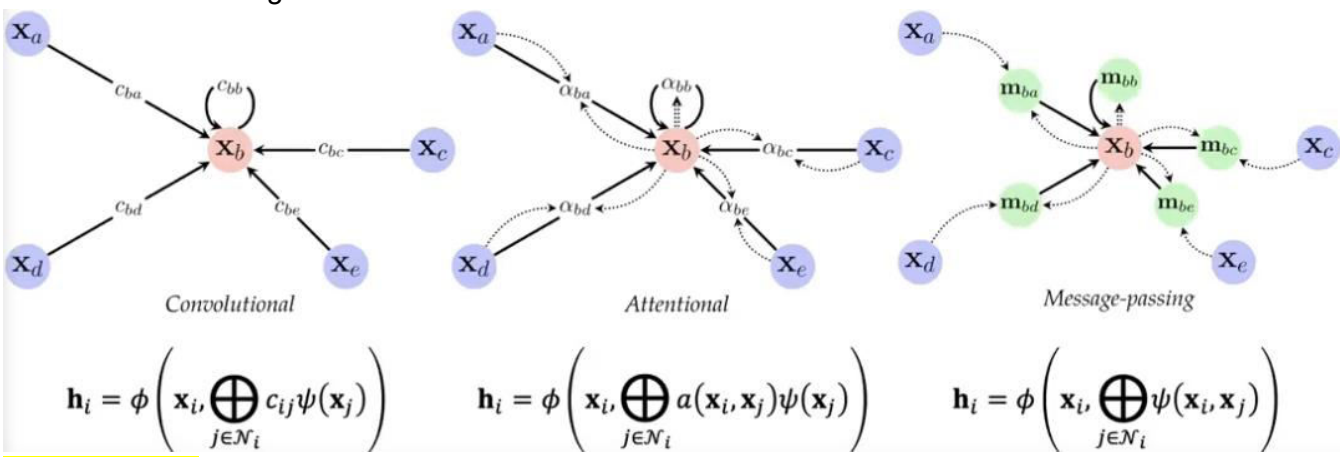
    $$\mathbf{X}_{\mathcal{N}_b} = \{\{\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d, \mathbf{x}_e\}\}$$



General blueprint for graphs

- We start off with a graph **G = (X, A)**
- We pass the graph to a GNN and update our node features to **H**, i.e. G becomes **G = (H, A)**
    - Structure usually doesn't change with this update, it's usually just the node features that change
- We can also, on the output, carry out several tasks:
    - **Node Classification -** pass a node's feature to a function **f(h_i)**
    - **Graph Classification -** pass a **graph** to an **<u>invariant</u>** function, which applies a node-wise function, then aggregates them with an **invariant operator** ⊕, then passes them to an invariant function to make a decision over the whole domain
    - **Edge Classification -** classify an edge
        - **Link Prediction -** An alternative is to predict whether an edge should exist between two nodes
        - We can have a directed edge, i.e. by considering one node to be a **sender** and one a **receiver**

What is in a GNN layer?

- Our **equivariant "GNN Layer" F** is formed by stacking a **local permutation-invariant function φ(x_i, X_{N_i})**
- **φ** can be called **diffusion**, **propagation**, or **message passing**
- Lots of research
- **Three flavours** of **φ** for GNNs
    - ***Convolutional*** *layers seen as a special case of* ***attentional****, which are special case of* ***message-passing***
        - The more general, the more complex a problem you can represent, but there is cost in scalability and overfitting
    - 

    - **Convolutional -** uses weights c_{ij} on the edges which are **pre-determined** in advance, **usually 1/|N|**
        - Point-wise transform all node features, multiplied by the weight
        - **Chebyshev Network**, **Kipf & Welling GCN**, **Wu et al SGC**
        - Simple layers
        - Useful for **homophilous graphs -** when edges tend to connect nodes of the same label
        - Highly scalable because the weights+edges are specified up front, this results in **sparse matrix multiplication**

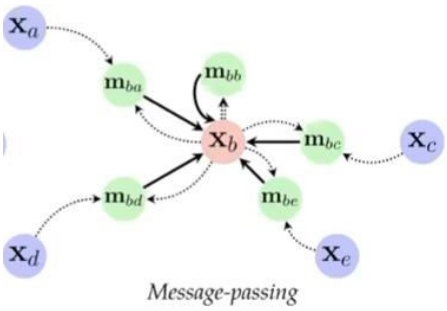
- **Attentional** - computing **attention** weight on edges between neighbours
    - Useful when we can't pre-specify the coefficient of interaction
    - Useful when edges don't encode **similarity**
    - Attentional layers try to take into account that different neighbours could be valued differently
    - The **coefficient of interaction** (weights) is computed **based on the features** of the receiver node
    - Replace $\mathbf{c_{ij}}$ with an **attention function** that takes features of $\mathbf{x_i}$ and $\mathbf{x_j}$ to produce a weight $\mathbf{a_{ij}}$
    - **MoNet**, **GAT**, **GATv2**
    - Allows you to learn different amount of interaction between neighbours
    - Still computing **one weight per edge** so it's just a sparse matrix multiply which is light-weight
    - This is where **transformers** live
    - Computes **importance** of the neighbour node, not the features
- **Message-Passing** -
    - Computes **arbitrary vectors (messages)** on each **edge**
    - Messages $\mathbf{m_{ij} = \psi(x_i, x_j)}$
    - Instead of computing importance as in attention, it computes a **latent representation of features**
    - Uses both the **receiver** and the **sender** node to collaborate and form a message vector
    - Your edges become a **recipe** for passing data around the graph
    - Involves **1-hop spatial GNNs**
    - Most **generic** GNN layer, there's a lot more parameters involved
    - May have **scalability** and **learnability** issues
    - A lot more expressive, used for chemistry, reasoning and simulation
    - **Interaction Nets**, **MPNN**, **GraphNets**

Summary

- We looked at **geometric deep learning blueprint** on sets and graphs
- We looked at **permutation invariance** and **equivariance**
- The **deepsets model** can be seen as a **universal blueprint** for neural networks
- Looked at various **GNN layers** - looked at **three flavours** with varying generalisation

Lecture 6 - Graphs and Sets II

- Coarsening is omitted as research hasn't shown its effectiveness in achieving good performance



$$\mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} \psi(\mathbf{x}_i, \mathbf{x}_j)\right)$$

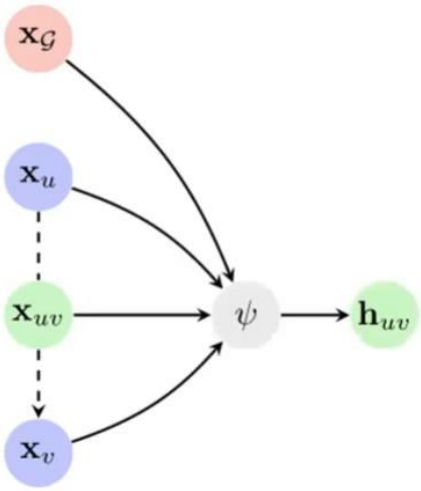
Message-Passing GNNs

- Compute **arbitrary vectors (messages)** sent across edges
- We consider the **features** of both the **receiver** and the **sender** node
- The **receiver** node aggregates all message vectors to compute next level **h**
- We can also consider **edge features**
  - **Node Features** = $\mathbf{x_u} \in R^k$
  - **Edge Features** = $\mathbf{x_{uv}} \in R^l$
    - Bond type, is in a ring?, etc
  - **Graph Features** = $\mathbf{x_G} \in R^m$
    - One vector, on the level of the **entire graph**, representing some global attributes
      - E.g. molecular weight, arthritis?, etc
    - Stores the features of the whole entire graph
    - Can be thought of as one node connected to all of the other nodes
  - **Hypergraphs…** etc, we can go further and implement it as above
    - Hypergraphs are made of **hyperedges** which connect **numerous nodes together**
    - You simply have to modify the way you carry out the **message passing** as a variation of the above
  - **Latents** are $\mathbf{h_u}$, $\mathbf{h_{uv}}$, $\mathbf{h_G}$ respectively
- **Graph Network Battaglia 2018** is used as a basis as it operates on **generic attributed graphs**
  - It implements the three flavours of features above and also includes **skip connections**
- **Updating the Graph Model**
  - *We update in this order as edges are **easiest to update**, always relying on just 2 nodes, nodes being a combination of edges, and graph features require all features*
    - Not a fixed order but makes sense implementationally

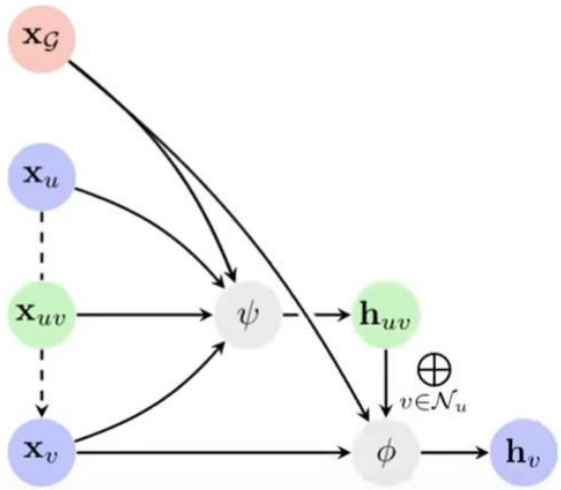
1. **Update Edges -** based on **global graph features** and relevant **adjacent node features**
   - $\mathbf{h}_{uv} = \psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{uv}, \mathbf{x}_{\mathcal{G}})$
   - Has **skipped connections** $\mathbf{x_{uv}}$
   - We send $\mathbf{h_{uv}}$ as **features** to the **nodes**
2. **Update Nodes -** propagate edge features into the **receiving nodes**
   - $\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{u \in \mathcal{N}_v} \mathbf{h}_{vu}, \mathbf{x}_{\mathcal{G}}\right)$
   - The nodes **aggregate** the **latent edge vectors**, and uses the global graph features to compute $\mathbf{h_u}$
     - Aggregation is **permutation invariant**, or the layer won't be **permutation equivariant**
   - Note the **skipped connection** $\mathbf{x_u}$ in the computation of $\mathbf{h_u}$
3. **Update Graph -** Use updated nodes and edges with a **permutation invariant function** to compute $\mathbf{h_G}$
   - $\mathbf{h}_{\mathcal{G}} = \rho\left(\bigoplus_{u \in \mathcal{V}} \mathbf{h}_u, \bigoplus_{(u,v) \in \mathcal{E}} \mathbf{h}_{uv}, \mathbf{x}_{\mathcal{G}}\right)$
   - Aggregates the **node and edge features** with a **permutation invariant aggregator**
   - Also has **skipped connections** $\mathbf{x_G}$

- We add **skip connections** to make sure our aggregations won’t lose information

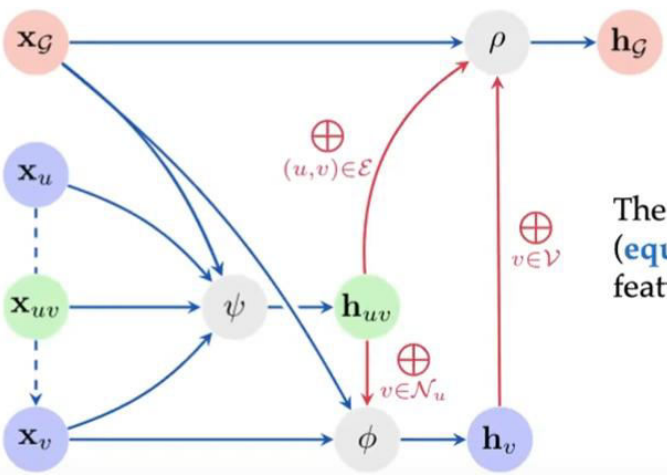
Visualising Graph Networks



$$\mathbf{h}_{uv} = \psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{uv}, \mathbf{x}_{\mathcal{G}})$$



$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{u \in \mathcal{N}_v} \mathbf{h}_{vu}, \mathbf{x}_{\mathcal{G}}\right)$$



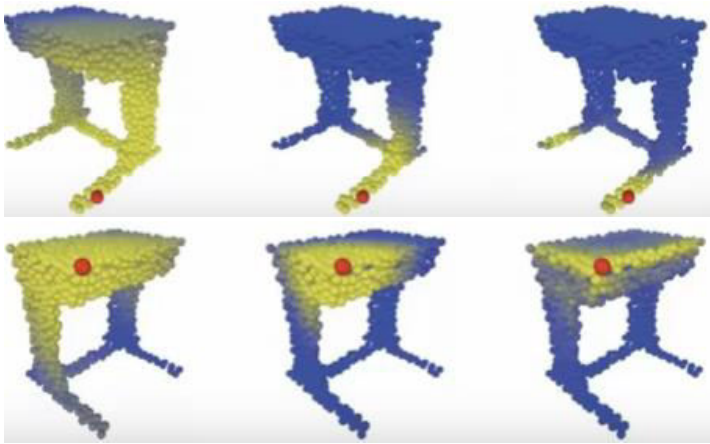
The *geometric deep learning blueprint* (**equivariant** and **invariant** layers) features extensively in GNs

$$\mathbf{h}_{\mathcal{G}} = \rho\left(\bigoplus_{u \in \mathcal{V}} \mathbf{h}_u, \bigoplus_{(u,v) \in \mathcal{E}} \mathbf{h}_{uv}, \mathbf{x}_{\mathcal{G}}\right)$$

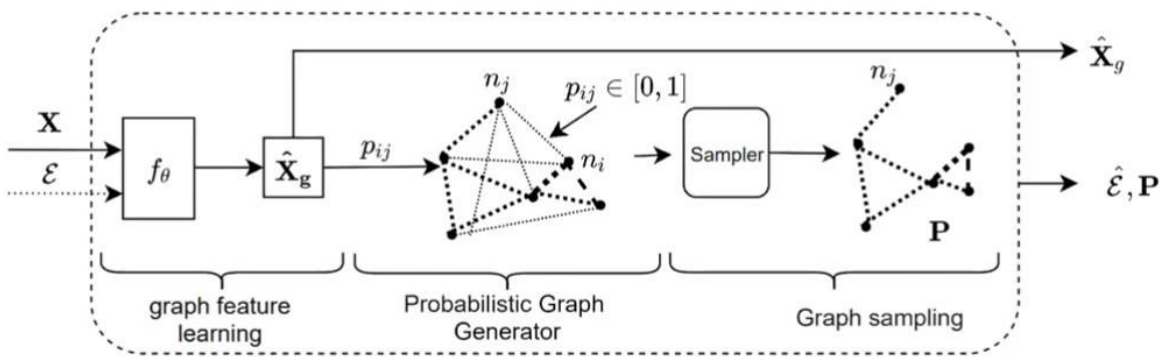
- *Using this blueprint, we can extend architectures, e.g.* ***GCN****,* ***GAT*** *and* ***MPNN*** *to using edge and graph features*

Latent Graph Inference

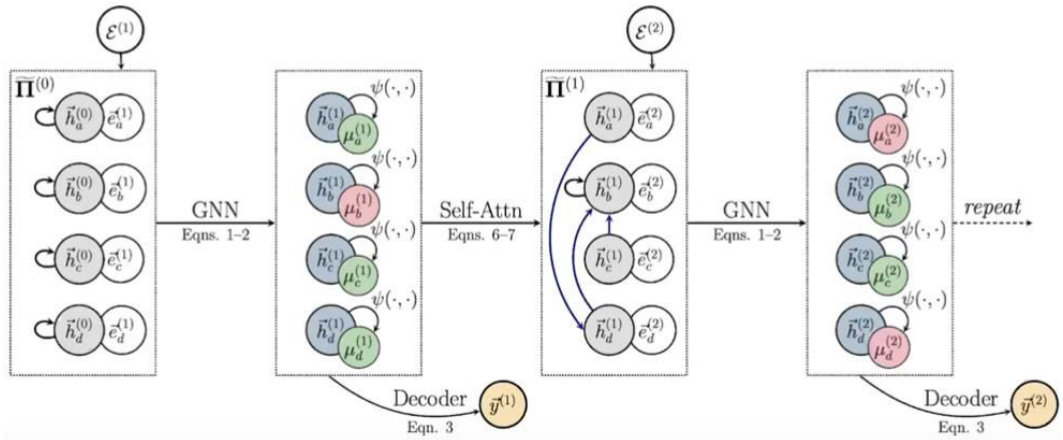
- The above assumes that we are given the **ground truth graph**, which isn't always the case
- This leads to the study of **latent graph inference**, *possibly very important area in graph representation learning*
- Sometimes a complete graph will be **suboptimal for our task**, so we need a simpler graph
    - E.g. using the **disjoint set union**
- This isn't always the case, and sometimes we need to build it up to varying extent (i.e. we don't know true edges)
    - **Option 1 - assume no edges (empty graph)**
        - If you assume the graph has an **identity adjacency matrix**, meaning that the graph has no edges, regardless of the architecture above, it will simplify to **deep sets**, where aggregating neighbourhood features amounts to deep sets taking just the node's itself's features
    - **Option 2 - assume all edges (fully connected graph)**
        - *This is the more popular option*
        - Assume that the graph is **fully-connected**, so the **adjacency matrix is full of ones**
        - This means that the **neighbourhood** of any node are **all nodes**
        - If the edges have no features, meaning we can't tell them apart, a **fully-connected** convolutional GNNs are **equivalent** to **deep sets** in this case
            - This is probably because the neighbours don't provide any unique information, the only information changing is the node's own information, just like in deep sets
    - **Transformers**
        - A **fully connected graph**, of the **attentional flavour**, where for every pair of nodes, we compute an **attention coefficient** and aggregate features based on those attention coefficients
        - As attention is just a single **scalar** value, think of attention as giving us a **soft adjacency matrix**
    - ***Empty graphs** ignore a lot of information, and **fully connected graphs** are hard to scale due to large neighbourhoods, so truth is probably somewhere in-between*
- **Latent Graph Inference - Inferring** the adjacency matrix **A** of a graph
    - Very difficult problem
    - Choosing a graph is a **binary** yes/no action
        - This makes **backpropagation very difficult**, as these actions are not differentiable
    - **Options on inferring adjacency matrices**
        - **Option 3a -** Inferring edges (**Variational** / **probabilistic**)
            - This approach is called **neural relational inference**
            - Our **prior** will be that the graph is fully connected
            - We make a **prior data distribution** of likelihood of edges
                - We can make it favour a sparse graph to get a simple structure
            - We use a GNN to compute the **posterior probability distribution**
                - Having observed the **node features**, we update the probabilities of edges existing, followed by a **sigmoid/softmax** (which are easily differentiable) to make a decision
                - We use the **Gumbel trick** to backpropagate through our decisions
                - Make **easy decisions** at **training time**, at training time we have fully connected graphs that we sample from
            - We **sample edges** from the **posterior** to decide which edges to create/destroy
            - We then update the graph and use the new graph for the next iteration
            - **Problem -** we still need to run over a fully connected graph
        - **k-NN Graphs**
            - We would like sparser graphs as they are **cheaper and more efficient**
            - **k-Nearest Neighbour Graph**
                - Each node has features $\mathbf{h_u}$
                - Connect a node only to **closest k nearest neighbours** of h, based on **distance**
                - *Usually how latent graphs are sparsely inferred nowadays*
                - May not represent the actual graph structure which may be important



- **Option 3b** - Inferring edges (**no learning**)
  - **Dynamic Graph CNN** (Want et al, ACM TOG 2018) - we use the **dot product** between node features to decide whether they are connected
    - We **recompute** edges every layer
    - **Non-parametric** - we don't infer the graph with any parameters
    - **Euclidean Distance** - in the first layer, we simply use **Euclidean distance** to connect nearby nodes with edges to compute the **latent vectors** for the node
    - In the beginning, we only **diffuse** the signal outwards, which isn't useful for finding semantically interesting parts of a graph (*first picture*)
      - As we go deeper into the layers, we use the **latent node features** to find similar nodes by $\mathbf{h_u}$ which **helps pick more relevant areas** (*last picture*)
      - As we can see by the legs on the table, spatially they are far away, but the features computed are similar so they are connected together
      - Similar features mean that one table leg can send information to the other

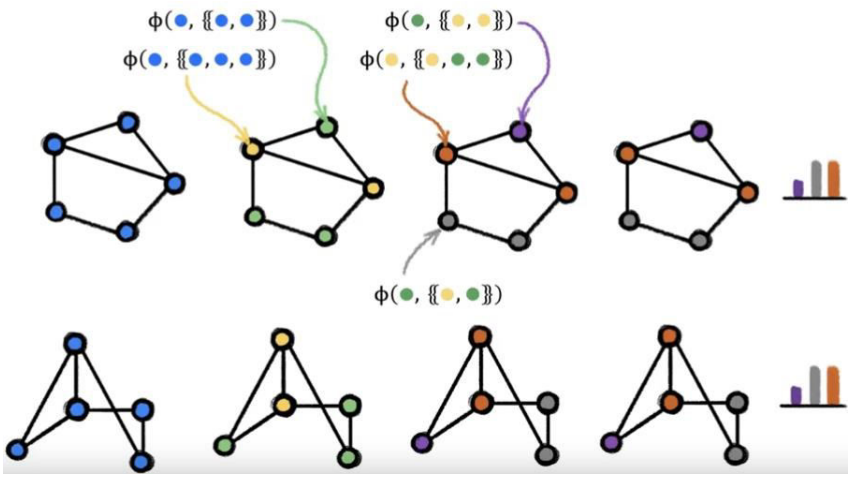


- **Option 3c** - Inferring edges (**reinforcement learning**)
  - **Differentiable Graph Module** (Kazi et al 2020)
    - As in 3b, we use the **dot product** $\mathbf{h_u}^T\mathbf{h_v}$ to measure similarity between nodes
      - Instead of using euclidean distance, we use an **RL agent** to select **k** edges
      - We get **better accuracy** at the cost of complexity
    - at every GNN layer, let $p(a_{uv} = 1) \propto \sigma\left(\psi(\mathbf{h}_u)^T \phi(\mathbf{h}_v)\right)$
      - The probability of some edge existing is **proportional** to some function over a **dot product** of transformed features
        - **φ** and **ψ** can be something like a **key** and **query** as in **transformers**
    - You can use some **performance measure** to say how good your edges were so that you can **reward** the RL agent
  - You have some **node features**, which are updated based on **pairwise similarities** which help you figure out **top k edges** which are passed to the **next layer**

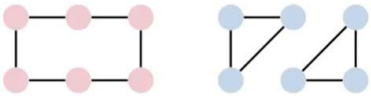


- **Option 4d** - Inferring edges (**supervised learning**)
  - **Pointer Graph Networks** (Velickovic et al, NeurIPS 20)
    - **Same as before -** at every GNN layer, let $p(a_{uv} = 1) \propto \sigma\left(\psi(\mathbf{h}_u)^T \phi(\mathbf{h}_v)\right)$
    - This time, however, you **directly supervise** to imitate some **ground-truth edges**
    - The network can **extrapolate** from the ground-truth edges to find better solutions

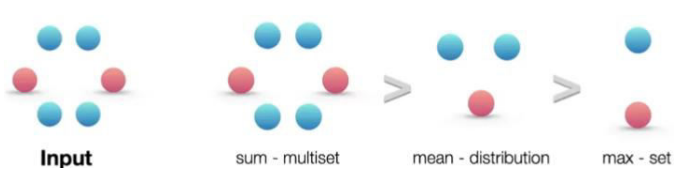
- **Graph Neural Networks**
  - **Decision of Graph Isomorphism** (test) **-** ask our GNN to **distinguish 2 graphs** that are **not isomorphic**
    - Estimates the neural network's capability through power to distinguish different graphs
  - If a network can't distinguish 2 different graphs, chances of discriminating them is hopeless because they will be represented by the same features, and get the same label



  - **Weisfeiler-Lehman Test (1-WL)** (Graph isomorphism test) **-** test to distinguish **non-isomorphic graphs**
    - Pass **random hashes** of **sums** along edges
    - You iterate the **colouring procedure** until the hashes don't change
    - If the **histograms** of colours is similar, then you say the graphs are **possibly isomorphic**
    - This is kind of similar to **convolutional GNNs**

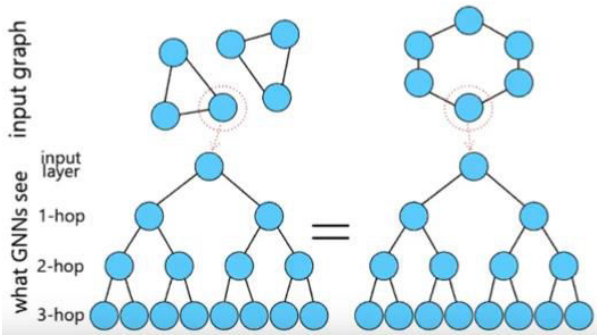


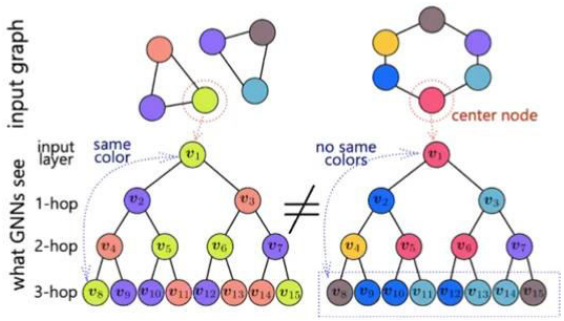
- As a result, **untrained GNNs** (where message passing is roughly equal to passing random hashes), can perform well, even though it can have some outlier cases causing issues
  - As a result, when we have **discrete features**, GNNs can only distinguish what the 1-WL test can distinguish! (GNNs can only perform as well as 1-WL)
    - To get close to achieving this performance, we need our **aggregation function** to be **injective**, meaning you want to **sum up the features**

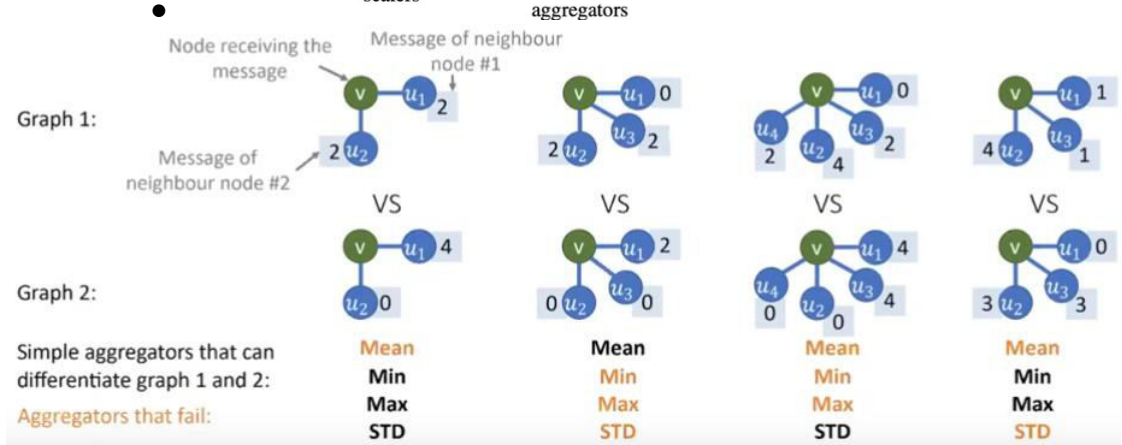


- **Summing** features **retains cardinality**, which **averaging** doesn't do
  - **Summing -** retains information about cardinality of the different kinds of nodes
  - **Averaging -** retains **ratio** of different nodes, not cardinality

    - **Element-wise Maximum -** only says which types are present
- **Problem with Weisfeiler-Lehman Test -** 6-cycle vs 2 triangles





- **Higher-Order GNNs** - Analysing and **fixing failure cases of the 1-WL** test to increase performance
  - GNNs can't detect closed triangles because nodes in a **cycle** and a **triangle** have the **same cardinality**
    - To fix this, we **augment nodes with random features**
      - This allows a node to **see itself k hops** away
        - *If you want to see k hops away, you need k layers*
      - We can extend to using **positional features** in the graphs instead of randomised features
        - E.g. RP-GNN (Murphy, 2019) or P-GNN (You, ICML 2019)
    - We can also **count interesting subgraphs**, e.g. we can count the number of rings as a feature
      - E.g. (Bouritsas, 2020)
  - In general, we can categorise **three** groups of **higher-order GNNs**
    - Modifying **features** (adding random / positional features as above)
    - Modifying **message passing rule** (DGNs using graph laplacian)
    - Modifying **graph structure** (1-2-3-GNNs)
      - Compute messages not just for nodes but also for tuples of nodes
      - More powerful than 1-WL but less scalable
- **Discrete vs Continuous Features**
  - We get **continuous features** either as a part of computing **latent features** or from **real life noise**
  - Continuous features **break proof of sum** being the best **injective** function to use
    - In some cases, other aggregators will be able to **distinguish graphs** that **summation cannot**
    - For continuous features, **PNA paper** (Corso, Cavalleri NeurIPS 2020) proves no best aggregator
    - If you want to **distinguish** two neighbourhoods of **size n**, you need at least **n aggregators**
    - You can **combine aggregators** to maximise performance
    - $$\bigoplus = \underbrace{\begin{bmatrix} I \\ S(D,\alpha=1) \\ S(D,\alpha=-1) \end{bmatrix}}_{\text{scalers}} \otimes \underbrace{\begin{bmatrix} \mu \\ \sigma \\ \max \\ \min \end{bmatrix}}_{\text{aggregators}}$$
    - 

- **Summary**
  - **Transformers** and **Deep sets** fitting within GNN flavours
  - **Latent Graph Inference** to infer the adjacency matrix
  - Breaking the limits of GNN expressive power: **Graph Isomorphism testing**, **higher-order GNNs**, **PNA**
  - Don't need lots of layer **depth** in GNNs because most real life graph **diameter** (max geodes dist) is **small**