Meet & Join Lattice Convolutional Neural Networks

Hugo Polsinelli

April 5, 2021

Contents

1 Introduction

The recent success of Deep Learning methods applied to data lying in a Euclidean space, such as image and audio, has led to the emergence of Geometric Deep Learning (GDL). The main motivations behind GDL is to transpose to irregular data domains proven Deep Learning techniques such as Convolutional Neural Networks (CNN). Indeed, CNNs have shown to be a very powerful tool to operate on N-dimensional grid-like structures. This can be partly explained thanks to their ability to extract localized features from the data by using shared weights, often called *filters*, and leverage the shift-invariance property of common classification tasks. However, some real life problems cannot be easily represented in an Euclidean space, often making the use of Convolutional Neural Networks impractical. Such tasks include: link prediction on a social graph [KW17], protein-protein interactions networks [Vas+20], modeling physical systems [San+18] and learning combinatorial optimization algorithms [Dai+18]. The non-Euclidean structure of these problems implies that the data cannot be described in a unified coordinate system and there exists no vector space structure nor any translation-invariance property. As a consequence, the convolution operation is not defined. In this thesis we will explore a special case of non-Euclidean domains: partially ordered sets and lattices. In order to gain more insight on how to define a convolution operation on non-Euclidean data domains, we will first expose the algebraic signal processing framewrok [PM08] (ASP). ASP is an axiomatic approach to signal processing with linear shift invariant filters on an algebra. We will see that from this framework, many graph signal processing frameworks can be derived.

Contribution The focus of this thesis are lattices and partially ordered sets. Partially ordered sets are directed acyclic graphs, and lattices are directed acyclic graphs with some additional properties. The meet and join of two elements in a lattice return the largest lower and the smallest upper bound respectively. [PSW20] utilize the meet and join operators to define lattice convolutions. This thesis provides three main contributions: first we implement the Discrete Lattice Signal Processing (DLSP) framework as laid out by [PSW20]. Second, we evaluate the implementation's correctness by assessing its performance on two synthetic tasks for which the Lattice Convolutional Layers should theoretically be able to solve the task perfectly, i.e, one can derive a closed form solution to these problem for the filters in the lattice domain. Lastly we make use of the framework on two real world structured output learning tasks. The hope is that using a lattice CNN adds enough prior about the output space into the model, such that it can improve the classification metrics. We benchmark our results against related work.

2 Background

In the first subsection, we will formally define Geometric Deep Learning and present an overview of the current techniques in the field. This will allow us to introduce the convolution on graphs. Then, we will motivate the need for a new signal processing framework that operates on lattices, which we will present in the following subsection.

2.1 Geometric Deep Learning

According to [Bro+17], geometric deep learning can be described as an umbrella term for emerging techniques attempting to generalize structured deep neural models to non-Euclidean domains such as graphs and manifolds. Geometric learning problems can be distinguished into two types. In the first class of problems the task is to *characterize the structure* of the underlying data whereas the second set of problems deals with *analyzing functions* defined on non-Euclidean spaces. In this thesis we will focus on problems falling in the second category, i.e, analyzing functions on a fixed non-Euclidean space.

Structure of the domain A common setting in the first class of problems is often referred to as *manifold learning* or *dimensionality reduction*, and is an instance of unsupervised learning. Assume the data is given as a set of points with an underlying lower-dimensional structure embedded in a higher dimensional non-Euclidean space, the task is then to recover the hidden low-dimensional structure. Examples of manifold learning include locally linear embedding (LLE) [Row00], stochastic neighbor embedding (t-SNE) [MH08] and various deep models [HCL06]. In network analysis applications such as social graphs, where nodes represent users and edges are drawn between two nodes to indicate the presence of a relation, an common task can is to detect communities.

Data on the domain The second task of analyzing functions on a non-Euclidean space can be further broken down into two sub-tasks: where the domain is *fixed* or when *multiple domains* are given. For example, assume we are given the traffic in a city represented as a time-dependant signal on the vertices of a city's road graph. The task is to predict the traffic given the past activity. In this scenario the road graph is assumed to be fixed. The focus of this thesis will be on analyzing functions on fixed non-Euclidean domains. An intuitive way to extend signal processing to any fixed algebra is provided by Algebraic Signal Processing framework (ASP).

[PM06] and [PM08] introduce ASP as an axiomatic approach to signal processing with linear shift invariant filters on an algebra. At its core, ASP is defined as a tuple (A, M, ϕ), where the signal space is cast as an algebra algebra A, the filter space as a module M, and ϕ generalizes the concept of z-transform to a bijective linear mapping from the signal to the filter space. Once the signal model and the shift operator are chosen, ASP provides the basic tools for filtering, convolving and computing the Fourier transform. Using ASP, one can derive many of the current graph signal processing (GSP) frameworks by defining the appropriate shift operator, such as graph adjacency matrices [SM13], or graph Laplacian filtering [Shu+13].

Methods and applications of signal processing on graphs have already been reviewed in [Zho+19], in particular in order to define an operation analogous to the convolution in the spectral domain which allows to generalize Convolutional Neural Networks models to graph signal processing. This thesis focuses on problems falling into the second sub task of the second task, i.e, analyzing functions living in a fixed non-Euclidean space, hence Graph Neural Networks important thus we will outline their motivations.

2.2 Case Study: Deep Learning on Graphs

Having exposed the main challenges of GDL, we will now review the current techniques in order to analyze functions on a fixed graph. Graphs are a structure used to model objects, as nodes, and their respective interactions, as edges. A number of non-Euclidean real world problems can be meaningfully represented as graphs such as social networks, protein-protein interactions and knowledge graphs, hence the aim is to develop machine learning techniques that can leverage this structure to predict relations (i.e link prediction), node classification or clustering techniques.

It is important to gain some intuition on how graph convolutional networks operate, since the aim of this paper is to define, implement and use lattice convolutional networks. The latter can, and should, be viewed as an analogy to convolutions on graphs, but instead of having the input indexed by a graph they operate of data indexed by a lattice. Hence this subsection is key to gain some insight into what we hope to archive using lattice convolutional neural networks.

Graph Neural Networks The concept of GNN was first proposed by [Sca+09]. The target is to learn a state embedding $\mathbf{h}_v \in \mathbb{R}$ which contains the information of neighborhood for each node. The state embedding $\mathbf{h}_v$ is an s-dimension vector of node v and can be used to produce an output $\mathbf{o}_v$ such as the node label. Let f and g be respectively the *local transition function* and the *local output function*.

f is shared among all nodes and updates the node state to its neighborhood and g describes how the output is produced. Formally we have:

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]})$$
$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v)$$

where $\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}$ respectively denote the features of v, the features of its edges, the states and the features of the nodes in the neighborhood of v. Let $\mathbf{H}$, $\mathbf{O}$, $\mathbf{X}$, and $\mathbf{X}_N$ be the vectors constructed by stacking all the states, all the outputs, all the features, and all the node features, respectively. The we can rewrite the above equations in a compact form:

$$\mathbf{H} = F(\mathbf{H}, \mathbf{X})$$
$$\mathbf{O} = G(\mathbf{O}, \mathbf{X}_N)$$

Using the following classic iterative scheme for iteratively computing the state at the next timestep:

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X})$$

GNNs provide a good intuition as to how one might interact with graphs, but they still lack concrete analogies with convolutions in the Euclidean space. Ideally, we would like to exploit a shift invariance property of graphs, which would in turn give rise to the notion of convolution on a graph.

Convolution on graphs Convolutional graph neural networks find their motivations in convolutional neural networks in the Euclidean space. The latter have produced impressive results on Euclidean domains due to their ability to extract localized spatial features through the use of a shared filter bank. Stacking multiple layers on top of each other allows the network to compose locally extracted features at a higher level and thus increasing the upper layer's field of view. However CNNs can only operate on N-dimensional grids such as images or sequences which makes their usage impractical for non-Euclidean data. By comparison, graphs have no natural coordinate system nor vector space structure. This implies that there is no clear way to define the shift operation on graphs, hence the convolution does not exist.

If we assume the data domain to be fixed, i.e, the input signal always lies in same space, one can define an operation analogous to convolution in the spectral domain. This in turn allows for the generalization of CNNs to graphs. The general idea behind graph convolutional neural networks is to define a so-called *spectral network* [Bru+14]. They allow for the generalization of convolutions on graph through the use of a Graph Fourier transform, which stems from a generalization of the Laplacian operator on the grid to the graph Laplacian [Shu+13]. Observe that graph neural networks are more general, i.e., one can write a GNN update rule that is equivalent to a graph convolutional network.

We have seen how using the spectral network allows us to define the convolution operation on graphs, giving rise to all kinds of graph convolutional neural networks using different shift operators. Observe that, after suitable ordering of the vertices, the adjacency matrix of a direct acyclic graph (of which lattices are a special case) are stricly triangular. Consequently, the eigenmatrix is always zero, hence it cannot be diagonalized. This provides further motivation to our lattice processing framework.

2.3 Lattice Signal Processing

In this thesis we consider signals indexed by a meet/join lattices, and use the discrete lattice signal processing framework as defined by [PSW20]. Consider a finite set $\mathscr{L}$ with a partial order $\leq$ also called a *partially ordered set* or *poset* and denote the elements of $\mathscr{L}$ as lowercase a, b, c... Formally a poset satisfies for all $a,b,c \in \mathscr{L}$:

(a) $a \leq a$

(b) $a \leq b$ and $b \leq a \implies a = b$

(c) $a \leq b$ and $b \leq c \implies a \leq c$

We say that b covers a, noted as $a \prec b$ if $a < b$ and there is no $c \in \mathscr{L}$ such that $a < c < b$. A meet-semilattice is a poset $\mathscr{L}$ for which a meet operation returning the greatest lower bound of a and b exists for every pair in $\mathscr{L}$. We denote this operation by $a \wedge b$. By symmetry we can also construct a *join semilattice* as a poset where the join $\vee$ operations returns the lowest upper bound for each pair of nodes in $\mathscr{L}$. A poset which is both a meet and join semilattice is called a lattice.

For example the powerset, i.e the set of all subsets of any finite set S, with $\cap$ as the meet and $\cup$ as the join operation is a lattice [WAP20]. Note that any semi lattice can be extended to a full lattice simply by adding a maximal (minimal) element that is larger (lower) than all $x \in \mathscr{L}$. This guarantees the existence of a join (meet) for each pair in $\mathscr{L}$.

Shifts and Convolutions on a Lattice Next we introduce the lattice signal processing framework as outlined in [PSW20]. Discrete Lattice Signal Processing (DLSP) is defined through the shift operation, which then allows to derive the notions of convolution, Fourier transform and frequency response. We consider real signal $\mathbf{s}$ indexed by the elements of a finite meet semilattice $\mathscr{L}$ of size n:

$$\mathbf{s} : \mathscr{L} \to \mathbb{R}, \quad x \mapsto s_x \tag{1}$$

and we write $\mathbf{s} = (s_x)_{x \in \mathscr{L}}$. The set of signals is an n-dimensional vector space, for which the ordering is arbitrarily defined as the topological order of the nodes in $\mathscr{L}$.

The construction of the shift as defined by [PSW20] uses the meet $\wedge$ operator as the shift. For any $a \in \mathscr{L}$ the shift by a is defined as $(s_{x\wedge a})_{x \in \mathscr{L}}$. One can easily show that the shift operator is a linear mapping on the space of signals: for any pair of signals $\mathbf{s}$ and $\mathbf{s}'$ and any scalars $\alpha, \beta \in \mathbb{R}$ shifting by $a \in \mathscr{L}$ yields, for all $x \in \mathscr{L}$:

$$(\alpha\mathbf{s} + \beta\mathbf{s}')_{x\wedge a} = \alpha(\mathbf{s}_{x\wedge a}) + \beta(\mathbf{s}')_{x\wedge a} \tag{2}$$

The main difference between discrete time signal processing and DLSP is that in the former a signal can be shifted by any integer value m but any shift can be obviously decomposed as $1 \times m$, hence any shift can be expressed as a repeated shift by 1, thus we say the shift by 1 *generates* all others. In the case of Discrete Lattice Signal Processing, Lattice theory shows that the generators of the meet semilattice are all the meet-irreducible elements of $\mathscr{L}$; i.e all $a \in \mathscr{L}$ such that there exists no $b, c \in \mathscr{L}$ with $b, c \neq a$ such that $a = b \wedge c$. Namely, an element of $\mathscr{L}$ is meet-irreducible (or join-irreducible) if and only if it cannot be expressed as the meet (join) of two other distinct elements of $\mathscr{L}$.

Now that we can shift by any $a \in \mathscr{L}$ we can define the notion of convolution. Let $\mathbf{h} = (h_q)_{q \in \mathscr{L}}$ be a filter, then we have:

$$\mathbf{h} * \mathbf{s} = \left(\sum_{q \in \mathscr{L}} h_q s_{x \wedge q} \right)_{x \in \mathscr{L}} \tag{3}$$

Since the meet operation is commutative; i.e $a \wedge b = b \wedge a$ for any $a, b \in \mathscr{L}$, all shifts commute with each other and with all filters hence the Discrete Lattice Signal Processing framework is shift-invariant.

Pure Frequency To compute the Fourier transform associated to the shift operation defined above, we need to find the signals that are eigenfunctions of all shifts; i.e the pure frequencies. Because the shifts commute, we are guaranteed that these signals exists. In our Lattice Signal processing setting the pure frequency, noted $\mathbf{f}^y$, is defined for every $y \in \mathscr{L}$ as:

$$\mathbf{f}^y = (\mathbb{1}_{y \leq x})_{x \in \mathscr{L}} \tag{4}$$

Were $\mathbb{1}_{y \leq x}$ is the indicator function of $y \leq x$; i.e: $\mathbb{1}_{y \leq x} = 1 \iff y \leq x$. To show that $\mathbf{f}^y$ is indeed the pure frequency associated to the DLSP shift, we shift the pure frequency by some $q \in \mathscr{L}$. Observe that if $y \leq q$ then $y \leq x \wedge q \iff y \leq x$ and thus $(\mathbb{1}_{y \leq x \wedge q})_{x \in \mathscr{L}} = \mathbf{f}^y$. Otherwise $y \leq x \wedge q$ never holds and $\mathbf{f}^y = 0$.

Frequency Response Using equation (4), we can now derive the frequency response of a filter $\mathbf{h} = (h_q)_{q \in \mathscr{L}}$ at frequency $y \in \mathscr{L}$ as follows:

$$\bar{h}_y = \sum_{q \in \mathscr{L},\, y \leq q} h_q \tag{5}$$

Notice that the frequencies are indexed by $y \in \mathscr{L}$ this means that the frequency response lies in the same space as the signal, i.e, it follows the partial order induced by $\mathscr{L}$.

Discrete Lattice Transform Let $\hat{\mathbf{s}} = (\hat{s}_y)_{y \in \mathscr{L}}$ denote the Fourier coefficient of a signal $\mathbf{s}$. Using equation (4) we can compute the inverse Fourier transform:

$$s_x = \sum_{y \in \mathscr{L}} \mathbb{1}_{y \leq x}\, \hat{s}_y = \sum_{y \in \mathscr{L},\, y \leq x} \hat{s}_y \tag{6}$$

Using the Moebius inversion formula from [SF99] to inverse equation (6) we can compute the Fourier transform:

$$\hat{s}_y = \sum_{x \leq y} \mu(x, y) s_x \tag{7}$$

Where $\mu(\cdot, \cdot)$ denotes the Moebius function defined by recursion as:

$$\begin{aligned} \mu(x,x) &= 1, && \text{for any } x \in \mathscr{L}, \\ \mu(x,y) &= -\sum_{x \leq z < y} \mu(x,z), && x \neq y, \end{aligned}$$

We call the DLSP Fourier transform defined in equation (7) the *discrete lattice transform*, noted $\mathrm{DLT}_{\mathscr{L}}$. Hence the spectrum of a signal $\mathbf{s}$ can now be computed as a matrix vector product, provided that both the DLT and the vector representation of $\mathbf{s}$ use the same ordering (unless stated otherwise we will always use the topological order of the poset). Further observe that the frequency response from equation (5) can be rewritten in matrix form.

$$\hat{\mathbf{s}} = \mathrm{DLT}_{\mathscr{L}}\, \mathbf{s} \tag{8}$$

$$\bar{\mathbf{h}} = \mathrm{DLT}_{\mathscr{L}}^{-1\,\top}\, \mathbf{h} \tag{9}$$

Convolution theorem Putting everything together, we obtain the following convolution theorem for the Discrete Lattice Signal Processing framework:

$$\widehat{\mathbf{h} * \mathbf{s}} = \overline{\mathbf{h}} \odot \hat{\mathbf{s}} \tag{10}$$

where $\odot$ denotes a vector point-wise multiplication. Observe that contrary to conventional Discrete Time signal processing, the frequency response and the Fourier transform are computed differently, using respectively equations (5) and (7).

Notice how the convolution theorem does not make use of the shift operations but only the discrete lattice transform. This means that we can relax our condition of requiring the meet and join to be defined for every pair in the poset. Hence our framework, while only theoretically proven for lattices, can be applied to partially ordered sets which are ubiquitous in machine learning tasks. Indeed, any direct acyclic graph defines a partially ordered set.

Relation with DTSP and GSP DSLP can be viewed as a form of GSP for lattices. Since lattices are a special kind of direct acyclic graphs, this implies that we can always find an ordering of the nodes such that the adjacency matrix is upper triangular. On the other hand, one major difference is that, in DLSP all the irreducible elements are generators, whereas in GSP there is only one generating shift. Moreover, the DSLP shifts do not always operate among the neighbors, but in a way that captures the partial order structure of the domain.

One major difference between DLSP and DTSP is that, in the latter one can leverage the fact that there exists and intuitive way to define subspaces of the input domain, i.e, 3×3 convolutional filter can be viewed as a filter with the same shape as the input padded with zeros. On the other hand, in DLSP there is no natural way to extend a sublattice. This means that contrary to convolutions on Euclidean data, we cannot perform operations that will alter the shape of the signal, such as pooling or similar transformations. This unfortunately imposes the constraint that our input domain must be fixed.

Table 1 summarizes the main equations of DLSP and compares them with DTSP in order to give more intuition on how in relates to the convolution in the Euclidean space.

Concept	DTSP	DLSP	graph adjacency
Signal	$(s_k)_{k\in[n]}$	$(s_x)_{x\in\mathscr{L}}$	$(s_v)_{v\in V}$
Basic shift	$(s_{k-1})_{k\in[n]}$	$(h_{q\wedge b})_{q\in\mathscr{L}}$, b meet irred.	$(A^k s)_{v\in V}$
Convolution	$\sum_{0\leq m<n} h_m s_{k-m}$	$\sum_{q\in\mathscr{L}} h_q s_{x\wedge q}$	$\sum_k h_k A^k s$

Table 1: Comparison of DLSP, finite DTSP and graph adjacency GSP. For simplicity we denote the index range with $[n] = (0,...,n-1)$

3 Lattice Convolutional Neural Networks

In the next section, we will show how we used the convolution theorem and equations (8) and (9) to implement lattice convolutional layers both in the *lattice domain* and *spectral domain*. We will extend the idea of *localized filters* to lattice signal processing, analogous to *one-hop* filters typically used in GNNs [Sca+09]. We will expose the optimizations techniques that have been implemented to decrease the computational overhead during both training and evaluation.

Lastly, we will validate our implementation by running our framework on two synthetic tasks. The aim of the first one is to assess the correctness of our implementation, whereas the second experiment can be thought of as a transition problem. Namely, data points are sampled from the true distribution hence it is synthetic but the problem of classifying ranked data is a common real world application.

3.1 Definitions

In order to implement the DLSP framework, we will heavily rely on equation (10), hence we will try to analyze it in detail. Observe that using successively equations (5), (8) and (9), we can rewrite the convolution as:

$$\begin{aligned}
\mathbf{h} * \mathbf{s} &= \mathrm{DLT}^{-1}\, \widehat{\mathbf{h} \star \mathbf{s}} \\
&= \mathrm{DLT}^{-1}\, (\overline{\mathbf{h}} \odot \hat{\mathbf{s}}) \\
&= \mathrm{DLT}^{-1}\, (\overline{\mathbf{h}} \odot (\mathrm{DLT}\, \mathbf{s})) \qquad (11) \\
&= \mathrm{DLT}^{-1}\, ((\mathrm{DLT}^{-1^{\top}}\, \mathbf{h}) \odot (\mathrm{DLT}\, \mathbf{s})) \qquad (12)
\end{aligned}$$

Notice that in order to compute a forward pass, we do not need to know the value of the filter in the lattice domain, hence this leads us to the following definition. If the network is trained to learn the filter without computing its frequency response, we say that it is operating in the *spectral domain*, otherwise we say that it operates in the *lattice domain*

Spectral and Lattice domain Optimizing the filter in the *spectral domain* (using equation 11) has the advantage of requiring less computational overhead. Indeed, it saves one matrix multiplication per forward pass to compute the frequency response. This is useful when we do not care about the value of the filter in the *lattice domain*. On the other hand, if we want to impose a constraint on the filter, it will most likely be in the *lattice domain* (using equation 12). Hence the framework must be able to learn the filter in both domains. Next we look at a way to add some constraints to the filter in the lattice domain.

Localized filters Similarly to techniques like *one-hop* filters in Graph Convolutional Neural Networks [KW17], we extend the notion of localized filters to the Lattice signal processing framework by defining the *one-hop* filter h on a meet-semilattice $\mathscr{L}$ as follows:

$$(h * s)_{H_{\mathscr{L}_1}} = \left(\sum_{q \in H_{\mathscr{L}_1}} h_q s_{x \wedge q} \right)_{x \in \mathscr{L}} \qquad \text{where } H_{\mathscr{L}_1} = \{a : a \in \mathscr{L},\ a \text{ irreducible}\}$$

This can be naturally extended to *k-hop* filters by the following recursion:

$$H_{\mathscr{L}_k} = \{a \wedge b : a, b \in H_{\mathscr{L}_{k-1}}\} \qquad \text{for } k > 1$$

Intuitively, this means that in the case of one-hop filters we only consider the shifts associated to irreducible elements. In the case of 2-hop filters we extend H to the shifts of the irreducibles elements and all of their pairwise meets. This procedure is repeated iteratively for k-hop filters. Recall from section 2.2 that shifts by irreducible elements generate all others, thus there exists a large enough k such that $H_{\mathscr{L}_k} = \mathscr{L}$. Further observe that in order to implement these localized filters, we need to be able to impose some constraints to the filter in the lattice domain. Hence the framework provides the possibility to train the lattice convolutional layer both in the lattice and spectral domain as we will see later. *k-hop* filters on join-semilattices are defined analogously by replacing the meet by a join where appropriate in the equations above.

Lattice convolutional layer Formally, we define a lattice convolutional layer on a lattice $\mathscr{L}$ as follows:

1. The input is given by n_c signals $\mathbf{s} = (s^{(1)}, \ldots, s^{(n_c)}) \in \mathbb{R}^{|\mathscr{L}| \times n_c}$;
2. The output is given by n_f signals $\mathbf{t} = \mathscr{L}_\Gamma(\mathbf{s}) = (t^{(1)}, \ldots, t^{(n_f)}) \in \mathbb{R}^{|\mathscr{L}| \times n_f}$;

3. The layer applies a bank of set function filters $\Gamma = (h^{(i,j)})_{i,j}$ with $i \in \{1,...,n_c\}$ and $j \in \{1,...,n_f\}$ and a point-wise non-linearity σ resulting in

$$t^{(j)} = \sigma\left(\sum_{i=1}^{n_c} h^{(i,j)} * s^{(i)}\right)$$

Observe that the lattice convolutional layer never uses the meet and join operations, thus we can extend the lattice convolutional layer to operate on partially ordered sets, i.e, signals indexed by a directed acyclic graph.

In the following section, we will discuss in depth the DLSP framework implementation details.

3.2 Implementation

The Discrete Lattice Signal Processing framework described in section 2.2 is implemented as a fully functional pytorch layer [Pas+19]. The aim is to provide to an end-user a black box for interacting with lattices inside the pytorch framework. First we briefly outline the pytorch library and how it operates, next we provide an overview of the framework's architecture. Third, we dive into the details of each of the modules that compose our implementation. Finally we discuss the technical optimizations that have been implemented in order to minimize the computational overhead, as well as some possible improvements.

The pytorch library In pytorch, Deep Learning models are just Python programs: defining layers and composing modules are all expressed using the familiar syntax of Python. This solution insures that novel neural network architecture such as the discrete lattice signal processing framework can be implemented easily. Indeed, the Lattice CNN layer is simply expressed as a Python class, which is constructed from a Lattice object in order to initialize the layer's internal parameters and whose forward method processes signals from the lattice. Furthermore, any end-user can create a new model by composing individual Lattice CNN layers. Finally, we leverage the automatic differentiation capabilities of pytorch to compute the gradient on-the-fly for our forward pass, since it is just a sequence of matrix multiplications. Thus the backward method does need to be implemented.

Framework architecture The framework is divided into two components: the lattice objects and the pytorch lattice CNN layer. The purpose of the former is to store the topology of the lattice, order the elements according to the partial order defined by the lattice and compute the discrete lattice transforms, inverse transforms and frequency response. The Lattice CNN layer are constructed using a lattice object. This allows the layer to initialize its own internal parameters according to the lattice and other user-specified settings we will explore in the next paragraphs. The framework supports meet and join semi-lattices as well as full lattices.

Cover graph Lastly, we define the notion of *cover graph* of a lattice $\mathscr{L}$. Let $G = (V, E)$ be a directed graph where $V = \{x : x \in \mathscr{L}\}$ and $E = \{(b, a) : a, b \in \mathscr{L}$ such that $b \prec a\}$. Namely, G is a directed graph where the nodes are the elements of $\mathscr{L}$ and we draw an edge between two nodes a and b if and only if b covers a. This is a useful tool to intuitively represent the partial order induced by a partially ordered set, hence the implementation will make an extensive use of it to instantiate lattice objects in our implementation. We also define the adjacency matrix $\mathbf{A}_{\mathscr{L}}$ of the cover graph, which we call the *cover matrix*, as follows:

$$\mathbf{A}_{\mathscr{L}} = (a_{\mathscr{L}})_{ij} = \begin{cases} 1, & \text{if } i \text{ covers } j \\ 0, & \text{otherwise} \end{cases}$$

where we use by convention the topological order of the lattice to order the columns of $\mathbf{A}_{\mathscr{L}}$. Using this ordering implies that the covers matrix is upper triangular.

Lattice object The lattice objects API is subdivided into three classes: meet and join semi-lattices and lattices. The latter simply inherits its properties from the two previous. These objects can be constructed in two ways: either using a networkX graph representing the covers graph of the (semi)-lattice, or by using the covers matrix. At creation, the lattice class will compute the partial ordering of the lattice elements, as well as the Fourier transform and the inverse Fourier transform. Both will be stored as matrices, with their rows and columns ordered by the topological ordering defined by the lattice.

The purpose of these objects is to provide the backbone structure to the lattice convolutional layers. Indeed, as our problem setting is that of fixed domains we assume that the lattice will not change during computation. Notice that this class only serves as the skeleton to store the topology data, whereas the signal is processed by the lattice CNN layers. Once the data domain has been computed and the lattice object encapsulating that domain created, we can now initialize as many lattice CNN layers as we want simply by passing the lattice object to the layer's constructor. Next we detail how the layers operate on lattice signals once they have been created.

Lattice convolutional layer Our framework provides two types of convolutional layers: *MeetConv* and *JoinConv*. By default these layers operate in the spectral domain, i.e, they train the spectral representation of the filter, never making use of the frequency response by using equation (11). This is done to avoid unnecessary computational overhead during the forward and backward passes. However, this behavior can nonetheless be modified to explicitly train the filters in the lattice domain by using equation (12). This can be useful in the case where the user wants to use localized filters or apply any other kind of explicit constraint to the filter in the lattice domain.

Akin to discrete time signal processing implementations, such as the one in the pytorch framework, the user can specify the number of input and output channels to the *MeetConv* and *JoinConv* layers as well as the presence or absence of a bias term. The user can also provide a mask when constructing a layer and, in so doing, the mask is applied as a point-wise multiplication on the lattice domain representation of the filter during the forward pass. Hence, for both *MeetConv* and *JoinConv* layers the framework implements the convolution in the spectral domain, the lattice domain and localized filtering by masking the filter in the lattice domain.

Thanks to the modularity of pytorch, the Lattice CNN layers behave exactly like any other in the pytorch framework. Indeed, they can be easily composed by stacking multiple of them in a module, but also with layers that are not part of this framework, provided they do not change the signal, such as activation functions, dropout, batch normalization or linear layers. Lattice convolutional layers operate similarly to any other convolutional layer: the output value of a meet or join convolutional layer with input size (b,c,n) and output (b,d,n) can be precisely described as:

$$\mathrm{out}(b_i,d_j) = \mathrm{bias}(d_j) + \sum_{k=0}^{c-1} \mathrm{filter}(d_j,k) * \mathrm{signal}(b_i,k) \tag{13}$$

where b is a batch size, c and d respectively the number of input and output channels, n the size of the (semi)-lattice and $*$ denotes the DLSP convolution operator as defined in equation (3). Next we look at some implementation tricks that allowed us to optimize the execution of the framework.

Optimization The *Einstein summation convention* (einsum for short) is a notational convention that implies summation over a set of indexed terms in a formula. By rewriting equation (13) using the einsum notation, we can leverage the optimized einsum package [SG18]. It can significantly reduce the overall execution time of einsum-like expressions by optimizing the expression's contraction order. We observe a five fold speedup on the forward pass against the conventional matrix multiplication. For example, the convolution in the spectral domain, $\mathrm{DLT}^{-1}\left(\overline{\mathbf{h}} \odot (\mathrm{DLT}\,\mathbf{s})\right)$, can be rewritten using the einsum notation as follows:

```
from opt_einsum import contract
out = contract('nm,com,ml,bcl->bon', DLT_inv,F,DLT,x)
```

where DLT and DLT_inv respectively denote the lattice Fourier transform and inverse Fourier transform, F denotes the filter bank with multiple input and output channels and x is the input signal. Note that this piece of code also extends the above equation to multiple input and output channels as well as batch processing.

An other optimization can be made by observing that all the layers store the Fourier transform and inverse Fourier transform. By sharing both matrices across all the layers of a same module we can save some space on the GPU and gain some time, as pytorch does not have to load the DLT stored in the next layer during the forward pass. Using this technique we observe a 10% speedup as well as a memory economy proportional to the number of layers in the module.

Finally, because we are using the matrix representation of the Fourier transforms we are limited to lattices with at most 2^{15} elements, otherwise the matrix multiplication becomes intractable. If the adjacency matrix of lattice is sparse, one can further improve the implementation by using the *torch.sparse* API. It provides a way to store and interact with sparse matrices in an efficient manner, thus increasing the maximum number of elements supported by the framework to 2^{18}. An other possible improvement could be to implement the fast Fourier transform algorithm described in [Bjö+16]. This lowers the computational complexity of the forward pass to $\mathcal{O}(\mu n)$ where μ is the number of irreducible elements in the lattice.

3.3 Framework Validation and Experiments

Now we evaluate our implementation of the DLSP framework using two synthetic tasks and benchmark our method against a fully-connected neural network for completeness. The aim of the first one is to provide justification of our implementation's correctness, since the lattice convolutional layer should be able to converge to the global optimum. The second problem explores how the lattice convolutional layers performs when classifying signals on a permutation lattice, a common setting in real-world problems.

3.3.1 Sums on a Powerset Lattice

This first task has been engineered such that there exists a closed form solution to the problem in the lattice domain, hence a lattice convolutional layer with one filter and no activation function nor bias should converge to a loss near zero. This tasks allows us to confirm that the code implementation of the DLSP is indeed correct and that is behaves as expected on a problem for which we know it should be able to reach a global optimum.

Problem definition The aim is to show that the framework is able to solve a problem for which there exists a perfect solution; i.e there is a filter for which the mean square error is exactly zero. Consider the powerset of a finite set S of size n: $S = \{x_1, x_2, x_3, ..., x_n\}$. Let us denote the powerset of S as 2^S. Recall from Section 2.2 that the powerset with the intersection as meet and the union as join is a lattice [WAP20]. The task is the following: given an input signal $\mathbf{s}$ such that:

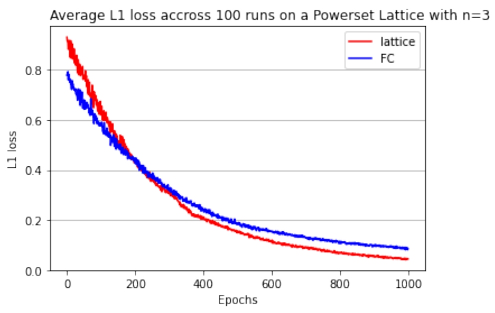
$$\begin{aligned} \mathbf{s}_{\{x_i\}} &= \text{some number} && \text{such that } \sum_{i=1}^{n} \mathbf{s}_{\{x_i\}} = 1, \\ \mathbf{s}_A &= 0 && \text{for any } A \in 2^S \text{such that } |A| \neq 1, \end{aligned}$$

The task is to extend the signal by propagating the sums; i.e: output a signal $\hat{\mathbf{s}}$ such that:

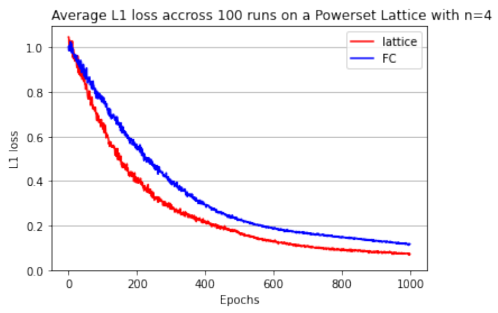
$$\hat{\mathbf{s}}_A = \sum_{x \in A} s_{\{x\}} \qquad \text{for all } A \in 2^S$$

Experimental setup A single *MeetConv* layer is trained without any bias, one input and one output channel and no activation function. We generate random pairs $(\mathbf{s}, \hat{\mathbf{s}})$ and optimize using a Mean Square Error loss. We benchmark our results against a fully-connected neural network without any bias term nor activation function. Observe our model only has $\mathscr{O}(|2^S|)$ trainable parameters whereas the fully-connected (FC) baseline has $\mathscr{O}(|2^S|^2)$ parameters, this is because in our model the filter lies in the same space as the signal. We run the experiment for $n = 3, ..., 8$ and repeat the experiment one hundred times.

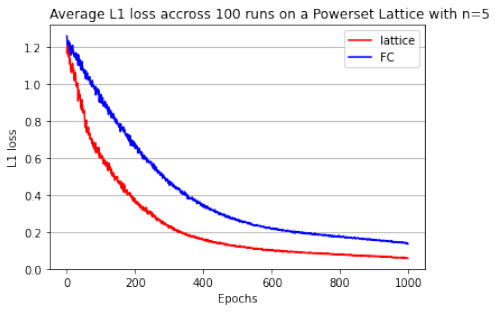
Results Figure 1 shows the average loss across all runs, as a function of the number of training steps. Observe that both models converge eventually to zero, but the *MeetConv* converges faster to its global optimum. This can be explained by the fact that it has orders of magnitude less trainable parameters, hence this behavior magnified as n grows as shown in Figure 1. Namely for $n = 8$ the *MeetConv* layer has $|2^8| = 256$ trainable parameters whereas the fully-connected baseline has $(2^8)^2 = 65,536$.



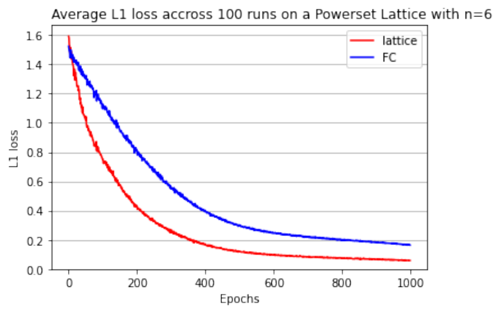
(a)



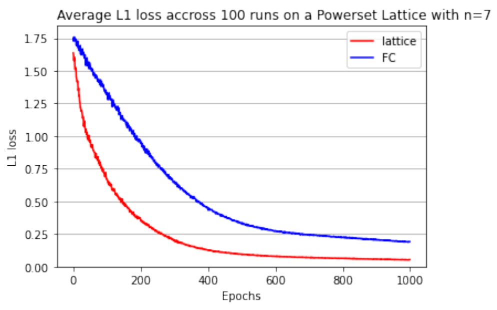
(b)



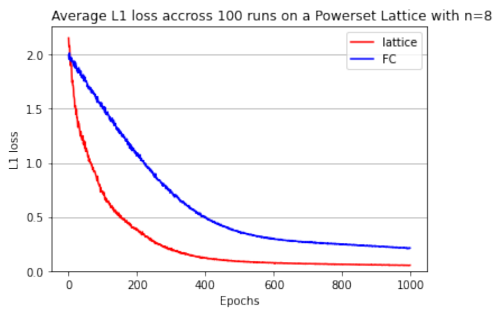
(c)



(d)



(e)



(f)

Figure 1: Average L1 loss of 100 runs as a function of training steps for $n = 3, \ldots, 8$. Fully-Connected baseline in blue, Lattice CNN in red.

3.3.2 Ranked data classification on a Permutation Lattice

This task can be viewed as a transition from synthetic to real world problems: we try to classify signals on the lattice of permutations. Namely, the signals represent ranked data which are common in real world tasks but we still generate the data from a known distribution. Observe that because in both problems we are sampling directly from the true distribution we do not need to make the distinction between training and testing error, as would be standard with problems for which we are sampling from the empirical distribution.

Motivations The permutations of a list of n elements forms a lattice of size $n!$. Formally, if we have two permutations $a = (a_1, ..., a_n)$ and $b = (b_1, ..., b_n)$, we say that b covers a if and only if there exists a transposition τ_i that exchanges two consecutive elements in positions $i, i+1$ such that $a_i < a_{i+1}$ and

$$b = a \cdot \tau_i = (a_1, ..., a_{i-1}, a_{i+1}, a_i, a_{i+2}, ..., a_n)$$

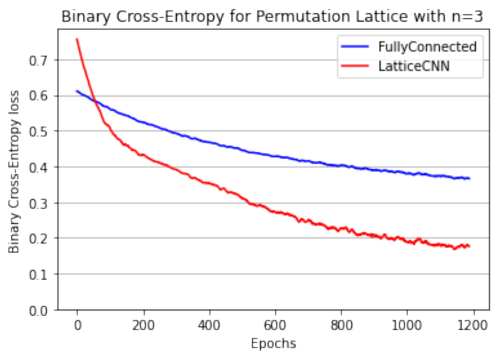
This equation allows us to construct the covers graph, which we can then use in our framework to create a lattice. Hence the hope is that we can gain a structural advantage by using a lattice CNN to classify ranked data. In this case, the signal would simply be the number of votes each permutation has received, and the target would be to recover the reference permutation from the mallows mixture model by producing a dirac delta on the true permutation, i.e, a one-hot vector corresponding to the reference permutation.

Problem definition Next we investigate the capability of our framework to classify strict-order complete list (SOC) ranked data. SOCs define a complete, transitive and symmetric relation over a group of objects. For example assume we have three objects A, B and C. We write B,A,C to specify that B is strictly preferred to A which is strictly preferred to C. Using [MW13] we can generate SOC ranked data indexed by the lattice of permutations. Namely we generate signals by sampling votes from an underlying hidden Mallow Mixture Model [Citation Needed].

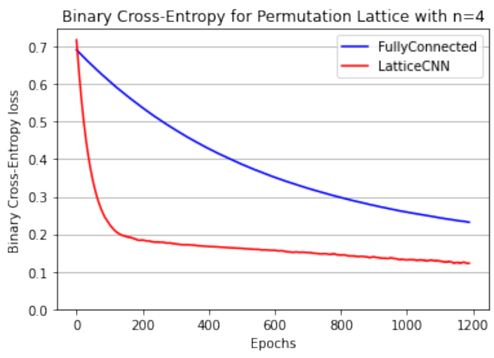
Experimental setup The task can be described as the following: m voters generate a random permutation (i.e they cast their vote) from a set of n candidates objects sampled from k hidden reference permutations using a Mallows Mixture Model. The signal input to the network is simply the number of votes that each permutation has received, and the target is to produce a dirac delta signal on the hidden source permutations. That is, train on pairs (permutation vote count lattice signal, source permutations lattice signal). The model is optimized using a Binary Cross-Entropy loss without any bias nor activation function. For this task, our model consists of 1 layer of *MeetConv* and *JoinConv*. The output of both layers is summed together. We benchmark our results against a one layer fully-connected neural network with the same activation function. For this experiment we set $m = 100, k = 1$ and $n = 3, ..., 7$.

Results Results are displayed in Figure 2. It shows that the Lattice CNN clearly beats the fully connected neural network benchmark. This can be explained for two main reasons. First as in the previous task the Lattice CNN has orders of magnitude less trainable parameters: $\mathcal{O}(n!)$ where as the FC has $\mathcal{O}(n!^2)$. Because the factorial operator grows exponentially fast, at $n = 7$ the fully connected network already has 25 million parameters per layer which becomes quickly intractable. Second, the Lattice CNN has, by design, a structural advantage over the baseline which further increases the gap between both models. We found that increasing the depth of both models did not yield any significant improvements.

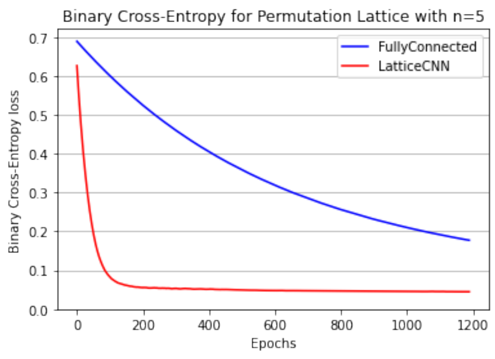
Because the number of voters is fixed ($m = 100$), the vote count for each node in the lattice becomes sparse as n increases. Namely, for $n = 3$ the permutation lattice contains 6 elements which means there is a high probability that each node in the graph gets some votes. On the other hand, for $n = 7$ there are 5,040 possible permutations for only 100 voters hence the signal is very sparse. This would tend to explain why the error is higher for smaller n, as there is less noise in the signal as n grows.



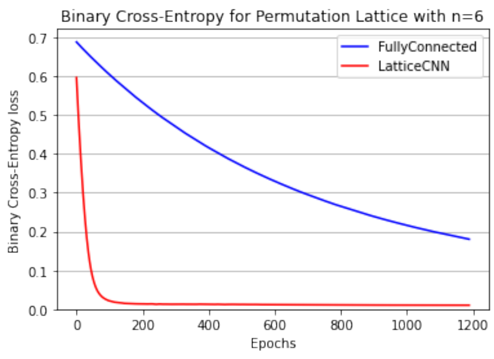
(a)



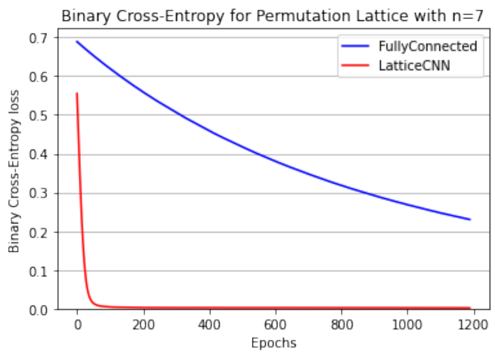
(b)



(c)



(d)



(e)

Figure 2: Binary Cross-Entropy loss for permutation lattice of $n = 3, \ldots, 7$ objects. Fully-Connected baseline in blue, Lattice CNN in red.

4 Structured Output Learning using Lattice Convolutional Neural Networks

Having described in detailed our framework and validated its performance on synthetic data, we now want to know how it performs on real world datasets. In the first subsection, we start by defining structured output learning (SOL) and expose our motivations for evaluating the DLSP framework in this particular domain. Next, we explore how lattice CNNs perform on a protein classification task where the output space is a tree. Finally, we try to leverage the powerset lattice to improve multi-label classification tasks.

4.1 Structured output Learning

Definition In traditional machine learning algorithms, the task can be summarized as follows: we seek to learn a parameterized function f_θ such that it maps our input $x \in X$ to our targets $y \in \mathbb{R}^n$:

$$\begin{aligned} &f_{\theta^*} : X \to \mathbb{R}^n, \\ &\theta^* = \arg\min_\theta \; \ell(f_\theta(x),\, y), \qquad\qquad \text{for } x \in X,\ y \in \mathbb{R}^n \end{aligned}$$

where ℓ is the loss function. This can be any kind of task like classification, regression or density estimation. In contrast, in *structured output learning* the outputs $y \in Y$ are complex structured objects. This is the case for example in tasks such as part of speech (POS) tagging [ZLS18] where we seek to produce a sentence syntax tree or protein function prediction where the output is a path in a tree.

Task output structures The label space of first experiments consists of a direct acyclic graph (DAG), hence it is a poset. We are interested in predicting the functions of a proteins given its amino-acid sequence. Namely, each function is a node in the DAG, where the root is the most general function and leafs are the most specific ones. We are interested in predicting the *functional subgraph* of a protein, i.e, the path from the root to the most specific functions of a given protein. Thus in this problem setting, the output structure is a partially ordered set, and the signal is a binary prediction on each node in the DAG such that the predicted functions form a connected subgraph of the label space. The root node is always active.

In the multi-label classification setting, we consider the powerset lattice of all the classes. The target is a Dirac delta on the node corresponding to the set of classes that are true. Formally, if we have a multi-label classification problem with k classes, we view $y \in \mathbb{R}^{2^k}$ as a signal indexed by the powerset lattice.

Motivations In both experiments, we consider structured output spaces where the outputs are indexed by a lattice (or a poset). Hence we hope that by using a lattice CNN, we can incorporate enough prior knowledge about the output space into the model such that it improves our prediction metrics. Indeed, the problems were specially chosen such that the output space can be modeled as a lattice or a direct acyclic graph.

In multi-label classification, the main focus is to learn the underlying dependencies between labels, and to take advantage of this during classification. Consider the set of all labels $\mathbb{L}$ and its powerset $2^{\mathbb{L}}$. Recall that the powerset with the meet as intersection and the join as union is a lattice. Thus we can leverage the DLSP framework to model all the dependencies between the labels and hope to learn the one that are meaningful while dropping the ones that are not.

In both experiments we compare our work against established benchmarks in order to validate our approach. Although results show that the improvements against previous work remains limited, our framework still archives results on par with other state-of-the-art techniques.

4.2 Protein Function Prediction

In this experiment, we have a sequence of amino acids that describe proteins, and the goal is to predict all its functions. In the next sub-sections we describe in depth the problem setting and try to motivate why the DLSP framework is a relevant tool to solve this problem. Next we talk about related work and introduce their findings. Finally we present our experimental setup and show our results.

4.2.1 Problem description

In this setting, we already know the structure of the output space and the relation between the labels. We seek to leverage the structural advantage of the Lattice CNNs to make some predictions on a DAG. Namely, we have multi-label classification task in which we have a known inductive biases over the individual labels, i.e, if a protein has a given function, then it must also have all the function's predecessors of it in the functional ontology tree. This task was previously investigated in related work where the authors tried to leverage graph neural network to make some predictions on the ontology [Spa+20], hence this provides a good benchmark for our lattice signal processing framework.

Gene Ontology The problem of predicting protein function prediction can be boiled down to the task of predicting a subgraph of a directed acyclic graph describing the hierarchy of protein functions, called the *Gene Ontology* (GO)

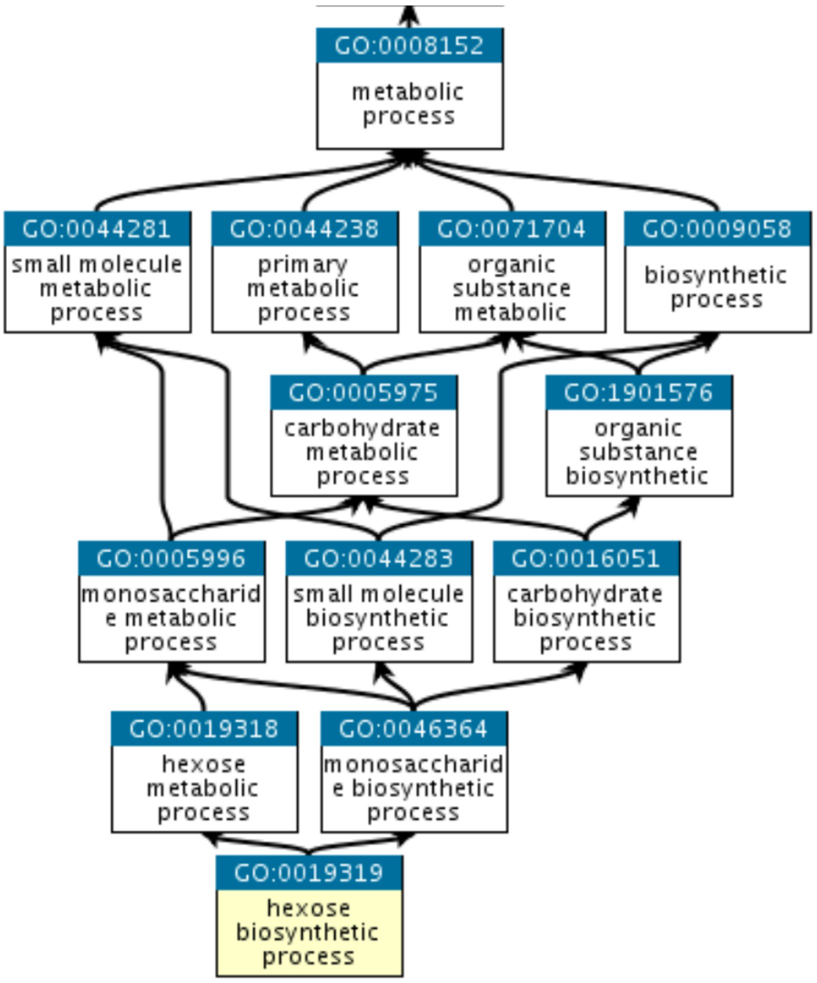

GO:0008152
metabolic process
GO:0044281
small molecule metabolic process
GO:0044238
primary metabolic process
GO:0071704
organic substance metabolic
GO:0009058
biosynthetic process
GO:0005975
carbohydrate metabolic process
GO:1901576
organic substance biosynthetic
GO:0005996
monosaccharide metabolic process
GO:0044283
small molecule biosynthetic process
GO:0016051
carbohydrate biosynthetic process
GO:0019318
hexose metabolic process
GO:0046364
monosaccharide biosynthetic process
GO:0019319
hexose biosynthetic process

Figure 3: Subgraph of the Molecular Function Gene Ontology. In this figure "metabolic process" is the root node hence every protein must inherit this property. Figure taken from [Ash+00]

[Ash+00]. The The Gene Ontology (GO) describes our knowledge of the biological domain with respect to three aspects: *Molecular Function*, *Cellular Component* and *Biological Process*. The structure of ontology can be described in terms of a graph, where each GO term is a node, and the relationships between the terms are edges between the nodes (see Figure 3). GO is loosely hierarchical, with 'child' terms being more specialized than their 'parent' terms, but unlike a strict hierarchy, a term may have more than one parent term.

CAFA dataset The protein sequences and functional annotations were provided by the CAFA3 dataset [NYa19]. It contains 31,243 proteins, the average amino acid sequence length is 431 and the average number of functions per protein is 10.

Label space structure The aim of this task is to predict the functional subgraph of a protein given its amino acid sequence, hence the output of the protein function prediction problem is a subgraph of a hierarchically-structured graph. As pointed out in the previous section, the output space is a direct acyclic graph, and the signal we wish to output is a binary prediction on each node of the DAG, such there is a path from the root to all the protein's function. Namely all the nodes in the protein's functional subgraph must be reachable from the root, i.e, the functional subgraph is (weakly) connected. Observe that unlike most graph representation learning task, the graph is not specified in the input space but only in the label space: it is a structured output learning problem. Namely, we are given a multilabel classification task in which we have a known hierarchy between the labels, i.e, if protein P has function F, then it must also have all the functions that are a predecessor of F in the ontology graph. In this task we will only focus on the *Molecular Function* subgraph of the Gene Ontology which contains 11,113 nodes. The ontology is a direct acyclic graph hence it is a meet semi-lattice and the root function, named *molecular function*, is the smallest node in the tree.

4.2.2 Related work

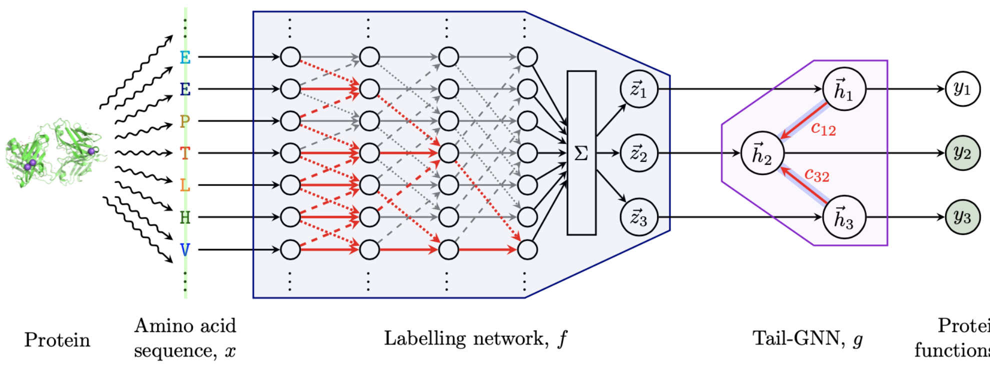


Figure 4: Overview of the model proposed by [Spa+20]. The inputs, as amino-acid sequences, are passed to the labeller network which produces the latent embeddings $\vec{v}_i$ for each label $i \in \mathbb{L}$. These are then passed to the Tail-GNN which computes an updated version of the latent embeddings, $\vec{h}_i$. Finally the prediction are made by passing the updated latent embeddings to a linear layer, which produced y_i for each $i \in \mathbb{L}$. Figure taken from the authors.

Let X denote the input space, and Y_i the output space for each label $i \in \mathbb{L}$, where $\mathbb{L}$ denotes the label space. In this case, $x \in X$ are protein sequences of one-hot encoded amino acids, and outputs $y_i \in Y_i$ are binary labels indicating presence or absence of individual functions for those proteins.

The authors of [Spa+20] propose a solution in which they subdivide their network in two main sub-components: the *labeller network* and the *Tail-GNN*. The former is tasked to project the inputs, i.e, the sequence of amino acids provided by the CAFA3 challenge [NYa19], into higher-dimensional embeddings living in the label space $\mathbb{L}$. Next, the *Tail-GNN* leverages prior knowledge of the output space, by using the ontology graph's adjacency matrix, to predict each protein's functional subgraph. Next we dive into more details about both sub-modules of the architecture proposed by [Spa+20].

Labeller Network We denote by $\mathbf{Z} = \{\vec{z}_1, ..., \vec{z}_{|\mathbb{L}|}\}$ the latent space in which the inputs are projected by the *labeller network*, called f. Typically, these will be k-dimensional real-valued vectors. The labeller network consists of a deep dilated convolutional neural network [Kal+17]. This allows for a efficient way of processing amino-acid sequence, without the performance drop induced by recurrent neural network cells. It outputs latent embeddings of shape $\mathbb{L} \times k$ for each amino-acid in the protein sequence, followed by global average pooling and reshaping to obtain a vector of length k for each label, i.e, each node in the ontology.

Tail-GNN The latent embeddings produced by f are the passed to the *Tail-GNN*, noted g. It treats each label $i \in \mathbb{L}$ as a node in the ontology graph and $\vec{z}_i$ as its corresponding node feature. The graph hierarchy is encoded using an adjacency matrix $\mathbf{A} \in \mathbb{R}^{\mathbb{L} \times \mathbb{L}}$ such that $\mathbf{A}_{ij} = 1$ implies that the prediction for label j can be related with the prediction for label i. The Tail-GNN is closely related to the GCN model [KW17]. At each step, the latent features $\vec{h}_i$ are updated in each label by aggregating neighborhood features across edges:

$$\vec{h}'_j = \mathrm{ReLU}\left(\sum_{i \in N_j} c_{ij} \mathbf{W} \vec{h}_i\right)$$

where N_j is the one-hop neighborhood of label i in the GO, $\mathbf{W}$ is a shared weight matrix parametrising a linear transformation, and c_{ij} is a coefficient of interaction from node j to node i, for which the authors experiment with several values:

1. *sum-pooling* X: $c_{ij} = 1$ for all $i, j \in \mathbb{L}$
2. *mean-pooling* X: $c_{ij} = \frac{1}{|N_j|}$ for all $i, j \in \mathbb{L}$

3. *graph attention*: $c_{ij} = a(\vec{h}_i, \vec{h}_j)$ for all $i, j \in \mathbb{L}$, where a is an attention function described in [Vel+20]

Finally the authors also to attempt to explicitly account for the label space topology by leveraging max-pooling:

$$\vec{h}'_j = \text{ReLU}\left(\max_{i \in N_j} \mathbf{W}\,\vec{h}_i\right)$$

The model can be summarized by the following relations:

$$\begin{aligned}
&f : X \to (Z_1 \times Z_2 \cdots \times Z_{\mathbb{L}})\\
&g : \mathbb{R}^{|\mathbb{L}| \times k} \times \mathbb{R}^{|\mathbb{L}| \times |\mathbb{L}} \to (Y_1 \times Y_2 \cdots \times Y_{\mathbb{L}})\\
&g(f(x), \mathbf{A}) = g(\mathbf{Z}, \mathbf{A}) = \mathbf{Y} = (y_1, \ldots, y_{|\mathbb{L}|})
\end{aligned}$$

The final layer of the network is a shared linear layer, followed by a logistic sigmoid activation. It takes the latent label representations produced by Tail-GNN and predicts a scalar value for each label, indicating the probability of the protein having the corresponding function. Figure 4 shows the architecture from input to output.

The entire network is optimized end-to-end using a Binary Cross-Entropy loss. Obsolete nodes and functions occurring in less than 500 proteins in the original CAFA3 dataset are discarded, which yields a reduced ontology with 123 nodes and 145 edges. Proteins whose function subgraph contain only the root node and those who are longer than 1000 amino acids are dropped. The dataset was randomly split into training/validation/test sets, with a rough proportion of 68:17:15 percent. Table 2 summarizes the results as reported by the authors.

Model	Validation F1	Test F1
Labeller network	0.582 ± 0.003	0.584 ± 0.003
Tail-GNN-mean	0.583 ± 0.006	0.586 ± 0.004
Tail-GNN-GAT	0.582 ± 0.004	0.587 ± 0.005
Tail-GNN-max	0.581 ± 0.002	0.585 ± 0.004
Tail-GNN-sum	$\mathbf{0.596} \pm 0.003$	$\mathbf{0.600} \pm 0.003$

Table 2: Values of F1 score for validation and test datasets for the proposed models, aggregated over five random seeds. Results taken from [Spa+20]

4.2.3 Experimental Setup

Similarly to what the author's setup in [Spa+20], we re-use the labeller network architecture to produce the latent embeddings $\mathbf{Z}$ and pass it's output to a lattice convolutional neural network in order to make a prediction in the label space. We reproduce the author's experimental setup by switching the Tail-GNN for the lattice CNN, and we compare performance of the labeller network alone against a lattice CNN on top of the labeller network in place of the Tail-GNN. We experiment by training the whole architecture end-to-end, both in the spectral and lattice domain for the lattice CNN. We also vary the number of layers in the lattice CNN, as well as the number of filters. Finally we evaluate the performance of the architecture using localized one-hop layers.

We also try to increase the size of the ontology. The authors of [Spa+20] report a reduced ontology size of 123 edges by filtering the nodes that occur less than 500 times in the CAFA3 dataset. We try to decrease this limit to 10, thus increasing the output space to 1,731 nodes. The incentive behind this is to increase the difficulty of the task for the labeller network when trained alone, and hope that the lattice CNN it will be able to leverage the bigger ontology size due to its structural advantage on the output space, and thus improve the test f1 accuracy. We validate our results using 5 fold cross-validation.

4.2.4 Results

The results are depicted in Table 2. Unfortunately, we find that the little increase in f1 accuracy reported by [Spa+20] was not able to resist against increased statistical scrutiny. We find that, when trained on the reduced ontology, the labeller network alone, the labeller network and the Tail-GNN and the labeller network with the lattice CNN produce similar f1 test accuracies (Table 3). We suspect this might be due to the expressiveness of the deep dilated convolutional network structure. Indeed, this is a very powerful model that is able to learn complex representations, such that stacking on top of it a lattice CNN or a GNN only marginally improves the prior knowledge on the output space structure. Namely, if the labeller network successfully projects the input domain (an amino acid sequence) to a latent representation $\mathbf{Z}$, one can doubt that it will benefit from another network to project $\mathbf{Z}$ to $\mathbb{L}$. In order to verify this hypothesis we increase the ontology size. We expect the task to be harder as the ontology grows, and the lattice CNN to significantly affect the results in the case where the labeller network would lack the expressiveness to exploit the increase in size of the output space.

Increased ontology Interestingly, although the results are lower on the augmented ontology size, we find that training on the bigger output space and computing the test f1 on the reduced ontology yields a significant increase of 10% of the evaluation metric. This is true for all three models compared, which would tend to indicate that the bigger the output space the more information the labeller network can extract from the ontology. This also suggests that the deep dilated convolution model is already a very powerful model, and that the use of GNNs or Lattice CNN to incorporate some prior into the model is not very successful. Indeed, these results tend to confirm our prior hypothesis that the labeller network is able to maximally exploit the information in the output space, i.e, it is not a bottleneck in the performance. Otherwise, increasing the ontology size would produce a lower increase in performance for the labeller network alone than the lattice CNN on top of the labeller. Since it is not the case, we can suspect that the deep dilated CNN is already enough to capture all the meaningful information in the output space, thus adding a GNN or a lattice CNN on top of it is redundant.

Localized filters Observe that for a lattice, an element is irreducible if and only if it has at most one parent in the covers graph. We extend this definition to direct acyclic graphs and reuse the definition of k-hop filters from section 3.1. We find that one-hop filters yield very poor results on the gene ontology DAG. Our analysis shows that when using a binary cross entropy loss, due to the last sigmoid activation layer, the gradient fades to zero very rapidly, thus prevents the model to train for longer than one epoch. Our attempts to mitigate this effect by removing the binary cross entropy loss and the sigmoid activation layer and replacing them with a L_2 loss seem to slow this phenomenon. However, this solution does not completely solve the problem. Stacking multiple layers of one-hop filters increases the depth of the network so much that it prevents the model from producing results on par with the other evaluated techniques.

Fixed Labeller Network Finally, we try to fix the weights of the labeller network and fine-tune a lattice CNN on top of the labeller's predictions. Although the training seems to hint a promising accuracy, when evaluated on the test set results suggest that the lattice CNN is overfitting on the labeller network's embeddings, and yields worse results than training the labeller and a lattice CNN end-to-end.

Model	Reduced Ontology	Full Ontology	Trained on full, Evaluated on reduced
Labeller network	0.584 ± 0.003	0.502 ± 0.009	0.666 ± 0.002
Tail-GNN-sum	0.600 ± 0.003	0.505 ± 0.007	0.661 ± 0.008
Ours	0.594 ± 0.005	0.503 ± 0.009	0.664 ± 0.009
Fixed labeller network + Ours	0.572 ± 0.009	0.488 ± 0.001	0.635 ± 0.002
Ours + localized filters	0.207 ± 0.004	0.124 ± 0.005	0.249 ± 0.003

Table 3: Test F1 score for our model against the best model from [Spa+20]

4.3 Multi-Label classification

Next we evaluate our framework in the context of multi-label classification. First we revisit our Mallows mixture model experiment by increasing the number of reference permutations. This is interesting because it allows to us validate our experimental setup on multi-label tasks before moving on to real world classification problems. Second, we try to leverage the powerset lattice to evaluate our proposed method on the Pascal VOC multi-label classification benchmark [Eve+09].

Assume we have a binary classification task for k labels. The goal of a multi-label classification task is to output a vector in $y \in \mathbb{R}^k$ such that $0 \leq y_i \leq 1$ for $i = 1, \dots, k$ the k labels, where y_i denotes the probability of the i^{th} label to be true. The specificity of this task is that more than one label can be true for each data point.

Contrary the the previous task, we do not have any prior on the relationships between the labels. In this context, we construct the powerset of all the combinations of labels, noted $2^{\mathbb{L}}$, with the intersection as meet and the union as join. The idea is to add a lattice CNN on top of our classification model just before the prediction. Assume we have a multi-label model that outputs a vector $y \in \mathbb{R}^k$, then we construct our method as follows. First we add a linear layer to project y into $\mathbb{R}^{2^k}$, then we pass the output of the linear layer to sequence of powerset convolutional layers. We then apply a softmax activation function to the last layer. Our prediction is the set of labels corresponding to the maximal element in the output vector.

Notice this methods is limited by the number of labels we can support, since we need first to project the output into $\mathbb{R}^{2^k}$ and then construct the powerset convolutional lattice. Since these quantities grow exponentially with respect to k, we are limited to small values (at most 15 labels).

4.3.1 Revisiting the Mallows Mixture Model experiment

Motivations Recall the experimental setup of section 3.3.2, where we generate ranked data from a reference permutation. The task was to predict the reference permutation by observing the votes sampled from a Mallows mixture model. We used $k = 1$ for the experimental setup. Now we evaluate how the lattice CNN performs as we increase the number of reference permutations. The objective of this experiment is similar to the ones of section 3, namely we wish to observe how our framework behaves in the context of a synthetic task, before we benchmark it against real-world experiments.

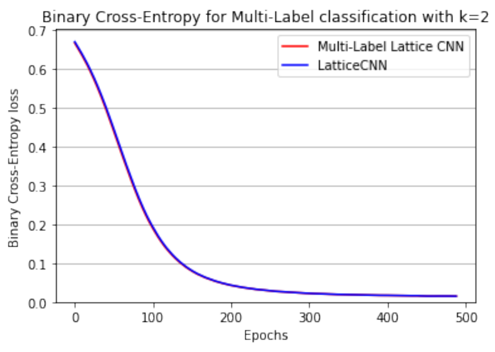
Experimental setup The setup is straightforward: we re-run our permutation lattice CNN from section 3.3.2 and increase the number of reference permutations. We compare this model against the same model, on which we stack a linear layer to project the outputs into $\mathbb{R}^{2^k}$ and then pass it to a one layer powerset lattice CNN. The networks are trained to minimize the binary cross entropy loss. We evaluate both models as a function of k, the number of reference permutations.

Results As Figure 5 shows, both models produce exactly the same output no matter the number of reference permutations. This can probably be explained by the fact that a single lattice CNN layer can already solve the task to a near zero binary cross-entropy loss. It seems that the powerset lattice CNN layer learns to copy the prediction of the permutation CNN since it is already able to solve the task by itself

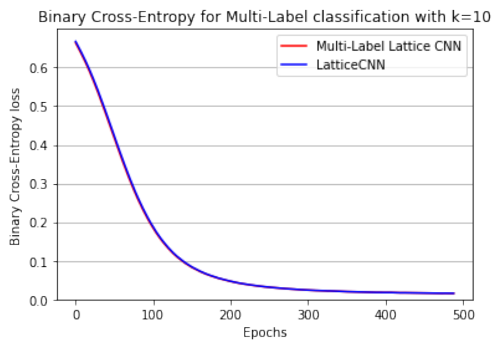
4.3.2 Pascal VOC

Problem description The Pascal VOC [Eve+09] is a visual object challenge. The dataset contains 8,776 images in which 20,739 objects appear, grouped into 20 classes. Thus it is a multi-label classification problem. Models are evaluated using the maximum a posteriori metric. Since we can only compute the powerset lattice for a small number of elements, we chose this challenge because it has a limited number of classes, unlike tasks like MS-COCO [Lin+15] which contains 80 classes.

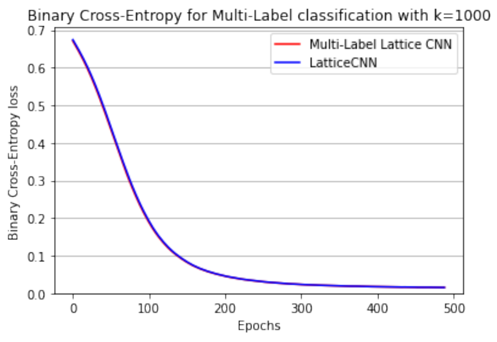
Related work The problem of predicting multiple labels per image can be approached by multiple angles. Some success was reported by exploiting label correlation via GNNs which represent the label relationships [Che+19b], or word embeddings based on knowledge priors [Che+19a]. Other authors have explored modeling image parts and attentional regions [You+20] as well as using recurrent



(a)



(b)



(c)

Figure 5: Binary Cross-Entropy loss as a function of the number of epochs on the permutation lattice of size $n = 3$. We vary the number of reference permutation to evaluate the capabilities of the powerset lattice on multi-label tasks. Both curves are exactly the same thus only one purple curve is seen.

neural networks [Nam+17]. Finally [Ben+20] propose using a focal loss to handle class imbalance in multi-label problems.

Experimental setup We use the pretrained model from [Ben+20] on Pascal-VOC 2007, and apply the same procedure as the multi-label permutation classification task above. Namely, we append to the pretrained model's output layer a linear module to project the outputs to $\mathbb{R}^{2^k}$ and pass it to a powerset lattice CNN. Since this procedure is computationally viable for small k only, we sample the datapoints from ten random classes. We fine tune end-to-end the newly obtained architecture. Since we want our results to be as statistically significant as possible, we repeat this procedure one hundred times by sampling ten random classes from the Pascal VOC dataset and re-run the training procedure. We report the mean average precision (mAP) across all runs.

Results Results are shown in Table 4. Stacking a powerset lattice on top of a pretrain model yields a loss of accuracy. This is probably due to the fact that TResNet-L is a very deep model. Recall we need to add a fully connected layer between the powerset CNN and the reset of the architecture. This is a full rank matrix of shape $\mathbb{R}^{10\times1024}$. This is probable that this matrix acts as a bottleneck in the model hence it slightly degrades the performance of the original model.

Model	MAP
TResNet-L	95.8
Ours	93.4

Table 4: Test MAP for our model vs TResNet-L from [Ben+20] pretrained on MS-COCO

5 Conclusion

In this thesis we introduced the discrete lattice signal processing framework from [PSW20] that stems from the algebraic signal processing theory [PM08]. DLSP leverages the meet and join operations to define a shift, and in so doing, ASP provides the tools to construct a full signal processing framework on lattices and partially ordered sets.

Framework implementation Using the DLSP theory, we built a fully functional lattice processing implementation based on the pytorch framework. The proposed lattice CNNs behaves like any other layer in the pytorch library, hence it can be easily mixed with other layers to produce a customized model. We introduced an API to construct and interact with lattices using the cover graph, which allows for the seamless creation of meet and join convolutional layers. The forward pass was implemented both in the lattice and spectral domain to accommodate for every possible usage. We also introduced some optimizations that helped increase the framework's overall performance. Finally, we observed that the proposed implementation could compute the convolution between a signal and a filter without the need of the shift operator (meet and join) , hence we were able to relax the lattice condition such that the framework can also process partially ordered sets, i.e, directed acyclic graphs, which are ubiquitous in real-world datasets.

Synthetic tasks The lattice CNN being implemented, we proceeded to evaluate the correctness of our code by running some synthetic experiments. The first one showed that our implementation indeed converged to the global minimum and that the lattice CNN was capable of solving tasks for which we could compute by hand closed form solutions in the lattice domain. Next, the second synthetic task allowed us to bridge the gap between synthetic and real-world tasks. Indeed, classifying permutations is a common problem in real-world experiments and we successfully showed that our implementation was able to leverage a signal on the permutation lattice to make some predictions.

Real-world tasks Our experiments on real-world datasets showed that there still remains some work to be done in order for the lattice signal processing framework to beat conventional deep learning methods. Even though our implementation was able to perform similarly to current state-of-the-art benchmarks, we have not been able to demonstrate a significant advantage from using the lattice CNN. This is partly due to the fact that current state-of-the-art models have been explored by deep learning academics for more than a decade, hence performing better than convolutional neural networks remains a very difficult task.

Limitations Finally, contrary to DTSP, our framework does not provide any way to extend sublattices, hence the filter and the signal must lie in the same space. Thus it is not possible to apply transformations such as pooling using our proposed implementation. This also yields the constraint that the input domain must be fixed. It was a surprisingly difficult task to find and conceive experiments such that the use of the lattice CNN was justified and the input domain was fixed. Also, due to some implementation requirements, we were not able to process lattices with more than 2^{18} elements. Furthermore, for the multi-label tasks, the use of the powerset lattice limited the number of classes to $n = 10$ due to the obvious exponential complexity of the powerset lattice.

Acknowledgements

I would like to thank Chris Wendler and Markus Püschel, for taking the time to answer my questions as well as their very insightful feedback and ideas. Their help was crucial in the process of building, improving and debugging this framework. I thank also Stefan Spalevic who provided the code of the labeller network used in Section 3.2. This was very helpful to ensure that the all experiments were ran as fairly as possible against the all benchmarks. It also allowed for precious time saving. Finally I thank ETH Zürich and the Advanced Computing Laboratory for providing me with this opportunity to do my master thesis in their laboratory, as well as a state-of-the-art computational infrastructure that allowed me to run my experiments in the best conditions.

References

[SF99] Richard P. Stanley and Sergey Fomin. *Enumerative Combinatorics*. Cambridge University Press, Jan. 1999. DOI: 10.1017/cbo9780511609589. URL: https://doi.org/10.1017/cbo9780511609589.

[Ash+00] Michael Ashburner et al. "Gene Ontology: tool for the unification of biology". In: *Nature Genetics* 25.1 (May 2000), pp. 25–29.

[Row00] S. T. Roweis. "Nonlinear Dimensionality Reduction by Locally Linear Embedding". In: *Science* 290.5500 (Dec. 2000), pp. 2323–2326. DOI: 10.1126/science.290.5500.2323. URL: https://doi.org/10.1126/science.290.5500.2323.

[HCL06] R. Hadsell, S. Chopra, and Y. LeCun. "Dimensionality Reduction by Learning an Invariant Mapping". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. 2006, pp. 1735–1742. DOI: 10.1109/CVPR.2006.100.

[PM06] Markus Püschel and José M. F. Moura. "Algebraic Signal Processing Theory". In: *CoRR* abs/cs/0612077 (2006). arXiv: cs/0612077. URL: http://arxiv.org/abs/cs/0612077.

[MH08] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[PM08] M. Puschel and J. M. F. Moura. "Algebraic Signal Processing Theory: Foundation and 1-D Time". In: *IEEE Transactions on Signal Processing* 56.8 (2008), pp. 3572–3585. DOI: 10.1109/TSP.2008.925261.

[Eve+09] Mark Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2 (Sept. 2009), pp. 303–338. DOI: 10.1007/s11263-009-0275-4. URL: https://doi.org/10.1007/s11263-009-0275-4.

[Sca+09] F. Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[MW13] Nicholas Mattei and Toby Walsh. "PrefLib: A Library of Preference Data HTTP://PREFLIB.ORG". In: *Proceedings of the 3rd International Conference on Algorithmic Decision Theory (ADT 2013)*. Lecture Notes in Artificial Intelligence. Springer, 2013.

[SM13] Aliaksei Sandryhaila and José M. F. Moura. “Discrete Signal Processing on Graphs”. In: *IEEE Transactions on Signal Processing* 61.7 (Apr. 2013), pp. 1644–1656. ISSN: 1941-0476. DOI: 10.1109/tsp.2013.2238935. URL: http://dx.doi.org/10.1109/TSP.2013.2238935.

[Shu+13] D. I. Shuman et al. “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains”. In: *IEEE Signal Processing Magazine* 30.3 (May 2013), pp. 83–98. ISSN: 1053-5888. DOI: 10.1109/msp.2012.2235192. URL: http://dx.doi.org/10.1109/MSP.2012.2235192.

[Bru+14] Joan Bruna et al. *Spectral Networks and Locally Connected Networks on Graphs*. 2014. arXiv: 1312.6203 [cs.LG].

[Lin+15] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[Bjö+16] Andreas Björklund et al. “Fast Zeta Transforms for Lattices with Few Irreducibles”. In: *ACM Transactions on Algorithms* 12.1 (Feb. 2016), pp. 1–19. DOI: 10.1145/2629429. URL: https://doi.org/10.1145/2629429.

[Bro+17] Michael M. Bronstein et al. “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Processing Magazine* 34.4 (July 2017), pp. 18–42. ISSN: 1558-0792. DOI: 10.1109/msp.2017.2693418. URL: http://dx.doi.org/10.1109/MSP.2017.2693418.

[Kal+17] Nal Kalchbrenner et al. *Neural Machine Translation in Linear Time*. 2017. arXiv: 1610.10099 [cs.CL].

[KW17] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG].

[Nam+17] Jinseok Nam et al. “Maximizing Subset Accuracy with Recurrent Neural Networks in Multi-label Classification”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/2eb5657d37f474e4c4cf01e4882b8962-Paper.pdf.

[Dai+18] Hanjun Dai et al. *Learning Combinatorial Optimization Algorithms over Graphs*. 2018. arXiv: 1704.01665 [cs.LG].

[San+18] Alvaro Sanchez-Gonzalez et al. *Graph networks as learnable physics engines for inference and control*. 2018. arXiv: 1806.01242 [cs.LG].

[SG18] Daniel G. A. Smith and Johnnie Gray. “opt_einsum - A Python package for optimizing contraction order for einsum-like expressions”. In: *Journal of Open Source Software* 3.26 (June 2018), p. 753. DOI: 10.21105/joss.00753. URL: https://doi.org/10.21105/joss.00753.

[ZLS18] Yue Zhang, Qi Liu, and Linfeng Song. *Sentence-State LSTM for Text Representation*. 2018. arXiv: 1805.02474 [cs.CL].

[Che+19a] Zhao-Min Chen et al. *Multi-Label Image Recognition with Graph Convolutional Networks*. 2019. arXiv: 1904.03582 [cs.CV].

[Che+19b] Zhao-Min Chen et al. “Multi-Label Image Recognition with Joint Class-Aware Map Disentangling and Label Correlation Embedding”. In: *2019 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, July 2019. DOI: 10.1109/icme.2019.00113. URL: https://doi.org/10.1109/icme.2019.00113.

[NYa19] Zhou N, Jiang Y, and Bergquist T.R et al. “The CAFA challenge reports improved protein function prediction and new functional annotations for hundreds of genes through experimental screens”. In: *Genome Biology* (2019). URL: https://doi.org/10.1186/s13059-019-1835-8.

[Pas+19] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[Zho+19] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. 2019. arXiv: 1812.08434 [cs.LG].

[Ben+20] Emanuel Ben-Baruch et al. *Asymmetric Loss For Multi-Label Classification*. 2020. arXiv: 2009.14119 [cs.CV].

[PSW20] Markus Püschel, Bastian Seifert, and Chris Wendler. *Discrete Signal Processing on Meet/Join Lattices*. 2020. arXiv: 2012.04358 [cs.IT].

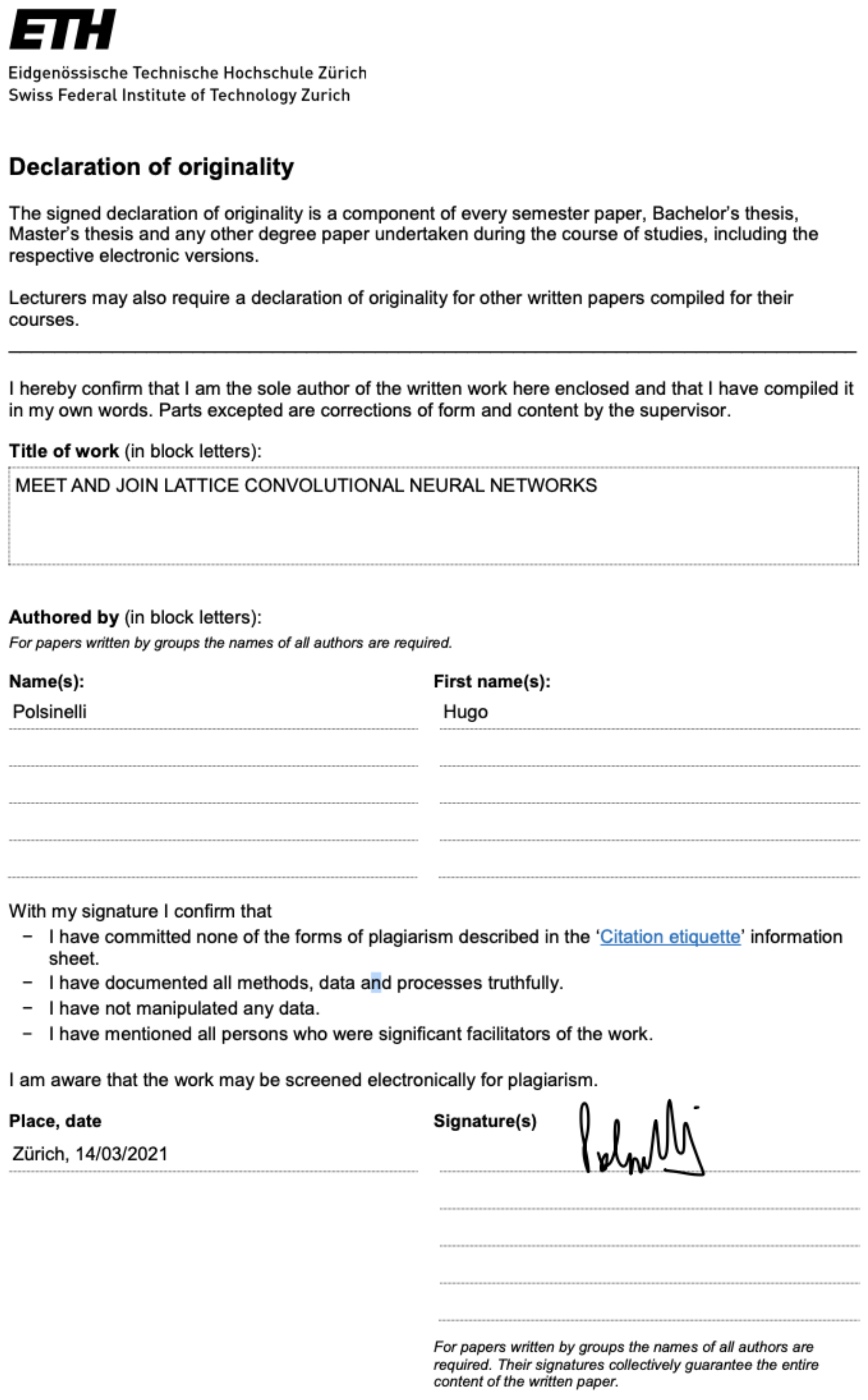
[Spa+20] Stefan Spalević et al. *2*. 2020. arXiv: 2007.12804 [cs.LG].

[Vas+20] Shikhar Vashishth et al. *Composition-based Multi-Relational Graph Convolutional Networks*. 2020. arXiv: 1911.03082 [cs.LG].

[Vel+20] Petar Veličković et al. *Neural Execution of Graph Algorithms*. 2020. arXiv: 1910.10593 [stat.ML].

[WAP20] Chris Wendler, Dan Alistarh, and Markus Püschel. *Powerset Convolutional Neural Networks*. 2020. arXiv: 1909.02253 [cs.LG].

[You+20] Renchun You et al. *Cross-Modality Attention with Semantic Graph Embedding for Multi-Label Classification*. 2020. arXiv: 1912.07872 [cs.CV].



ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

MEET AND JOIN LATTICE CONVOLUTIONAL NEURAL NETWORKS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Polsinelli	Hugo

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 14/03/2021	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.