Appendix A

Background

This appendix presents some background for the research presented in this dissertation. We first elaborate on the conceptual details of some standard deep neural network architectures that are extensively used in our research. Then, we provide some brief conceptual details on Inductive Logic Programming (ILP).

A.1 Deep Neural Networks

Deep neural networks (or DNNs) are artificial neural networks that consists of multiple layers between an input layer (that takes features describing a data-instance as input) and an output (that computes the prediction given the input). A DNN is defined by a structure and a set of parameters. The structure of a DNN refers to the organisation of various layers in the DNN and the interconnections among various layers. Each connection in a DNN associates a connection strength (a real-value) called the synaptic weight or weight. There are also weights of a different kind called biases. The set of all the weights and biases is called the parameters of a DNN. DNNs can model complex non-linear relationships in the data by expressing a data-instance as a layered composition of primitive features [STE13]. The hidden layers in a DNN are responsible for constructing a transformed representation for the data-instance suitable for the given problem.

Figure A.1 shows a diagrammatic representation of the brief process of learning a DNN model given data. A learner $\mathcal{L}$ takes as input the data D, a DNN model structure π relevant for learning D, the parameters $\boldsymbol{\theta}$ of the model corresponding to the structure π, an adequate loss function L for the problem, and constructs a DNN model M. Learning refers to finding a suitable set of parameters $\boldsymbol{\theta}$ for the DNN such that the model correctly represents the data D. This involves a procedure that iteratively updates the parameters in $\boldsymbol{\theta}$ given the data D by minimising the loss function L.

Data D consists of pairs of data-instances and their target outputs. That is, D consits of $(\mathbf{x}, y)$ pairs, where $\mathbf{x}$ represents a data-instance and y represents a category

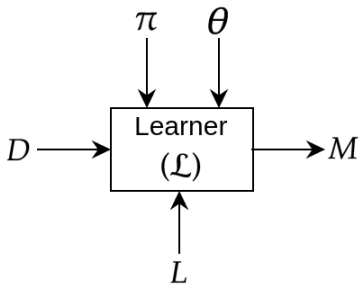

Figure A.1: Construction of a DNN model from data (Based on Figure 2.2; reproduced here for readability and completeness).

(for classification problems) or a real-value (for regression problems). $\mathbf{x}$ can be: a vector or tensor of properties describing a data instance, an image, a graph, or a sentence. For the applications studied in this dissertation, y is usually a class-label, although this is not necessary. Depending on what $\mathbf{x}$ is, there are different kinds of DNNs. For examples, if $\mathbf{x}$ is a vector (or in general, a tensor) there are multilayer perceptrons (MLPs), if $\mathbf{x}$ is an image there are convolutional neural networks (CNNs), if $\mathbf{x}$ is graph, there are graph neural networks (GNNs), etc.

Before describing what $\mathbf{x}$ is in this dissertation, we first summarise standard deep neural network architectures that are used in our research. We use two kinds of deep neural networks: (1) deep fully-connected feed-forward neural network, called Multilayer Perceptrons or MLPs, and (2) graph neural networks or GNNs.

Multilayer Perceptrons (MLPs)

MLPs are a class of deep neural network architectures consisting of multiple layers of neurons, each fully connected to those in the preceding layer (from which they receive input) and to those in the succeeding layer (which they, in turn, influence). The layer of neurons corresponding directly with the input features is called the input layer (or simply, "inputs") and the layer corresponding to the output(s) is called the output layer. The layers of neurons in between the inputs and the output layer are called hidden layers. A simple box diagram of a L-layered MLP structure is shown in Fig. A.2.

We now discuss some fundamental computations involved in an MLP. We start with a set of notations: Let $\mathbf{x} \in \mathbb{R}^d$ denote a data instance with d-features $x_1, \ldots, x_d$. This forms the input to an MLP. Let $m^{(\ell)}$ denote the size (number of neurons) of any layer ℓ. Clearly, $m^0 = d$. Let $\mathbf{z}^{(\ell)} \in \mathbb{R}^{m^{(\ell)}}$ denote the inputs and $\sigma^{(\ell)}$ be the activation for the hidden layer ℓ. Then, $\mathbf{h}^{(\ell)} \in \mathbb{R}^{m^{(\ell)}}$. The parameters of any layer ℓ is denoted by $\mathbf{W}^{(\ell)} \in \mathbb{R}^{m^{(\ell-1)} \times m^{(\ell)}}$. Let the bias parameters at layer ℓ be denoted by $\mathbf{b} \in \mathbb{R}^{m^{(\ell)}}$. The forward propagation in MLP consists of the following computations:

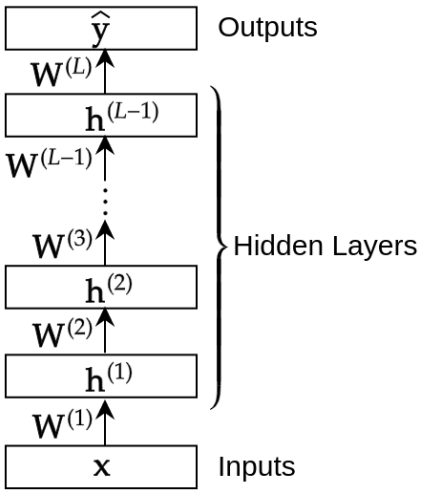

Figure A.2: Representing MLP with layers as boxes. No importance to be given to the width of the boxes. The *depth* of the MLP is L. $\mathbf{h}$ denotes a vector of hidden layer activations (also called hidden representation) and $\hat{\mathbf{y}}$ denotes the outputs. Superscript (ℓ) represents the layer index. The arrows show propagation of information (activations) from one layer to another. $\mathbf{W}^{(\ell)}$ denotes the parameters (a matrix of synaptic weights) at layer ℓ.

For layer $\ell = 0, \ldots, L-1$:

$$\mathbf{z}^{(\ell)} = \mathbf{h}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{b}^{(\ell)} \tag{A.1}$$

$$\mathbf{h}^{(\ell)} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}) \tag{A.2}$$

Clearly, $\mathbf{h}^{(0)} = \mathbf{x}$. For the output layer, the computations are then:

$$\mathbf{z}^{(L)} = \mathbf{h}^{(L-1)}\mathbf{W}^{(L)} + \mathbf{b}^{(L)} \tag{A.3}$$

$$\hat{\mathbf{y}} = \sigma^{(L)}(\mathbf{z}^{(L)}) \tag{A.4}$$

In the above computations, the activation function $\sigma^{(\ell)}$ is applied element-wise. It decides the degree of activation of a neuron based on the net input it receives. Below, we provide some common activation functions:

1. Linear activation: It results in an affine transformation of the input.

$$\sigma(z) = \alpha z \tag{A.5}$$

   where $\alpha \in \mathbb{R}$ is some constant.

2. Sigmoid or logistic activation: It squashes the input to a value in the range $[0, 1]$.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{A.6}$$

3. Hyperbolic tangent (tanh) activation: It squashes the input to a value in the range $[-1, 1]$.

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{A.7}$$

4. Rectified Linear Unit (ReLU) activation: A simple and popular non-linear function for deep networks.

$$\sigma(z) = \max(0, z) \tag{A.8}$$

5. Variants of ReLU:

   (a) Leaky ReLU:

   $$\sigma(z) = \max(0.01z, z) \tag{A.9}$$

   (b) Parametric ReLU:

   $$\sigma(z) = \max(\alpha z, z) \tag{A.10}$$

   where $\alpha \in \mathbb{R}$ is a learnable parameter.

Learning of an MLP refers to the update of the model parameters $\mathbf{W}$s given some labelled data in the form of $(\mathbf{x}, y)$ pairs. This process is referred to as "training". Training in a deep neural network is usually done using the popular *backpropagation* procedure [RHW86]. Backpropagation updates the parameters (synaptic weights and sometimes, other hyperparameters) using the chain-rule of derivative calculus for propagating the gradients of the loss from output layer towards the inputs.

Graph Neural Networks (GNNs)

MLPs described above can learn from data-instances that are described using numeric feature-vectors and cannot be adopted directly for graph-structured data where each data-instance is a graph. MLPs would require that the graph instances to be converted to fixed-length numeric vectors which can then be used as inputs. There is a class of DNNs that are suitable for learning directly from graph-structured data, called graph neural networks (GNNs). GNNs, however, do involve a structure similar to MLPs within their structure. We will discuss some implementational aspects of GNNs here. For a more complete technical overview, the reader is referred to [Ham20].

In Chapter 4, we discussed that GNNs, in their implementations, involve 3 procedures: (a) AGGREGATE: For every vertex, this procedure aggregates the information from neighboring vertices; and (b) COMBINE: This procedure updates the label of the vertex by combining its present label with its neighbors'; and (c) READOUT: This procedure constructs a vectorised representation of the entire graph. In mathematical forms, these procedures are described as follows:

For a graph G, at some iteration k, the labelling of a vertex v (denoted by h_v) is updated as

$$a_v^{(k)} = \mathsf{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right),$$
$$h_v^{(k)} = \mathsf{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

where, $\mathcal{N}(v)$ denotes the set of vertices adjacent to v. Initially (at $k = 0$), $h_v^{(0)} = X_v$.

The vector representation of the entire graph G is obtained in the final iteration ($k = K$) as

$$h_G = \mathsf{READOUT}\left(\left\{h_v^{(K)} \mid v \in G\right\}\right)$$

In practice, AGGREGATE and COMBINE procedures are implemented using graph convolution and pooling operations. The READOUT procedure is usually implemented using a global or hierarchical pooling operation [XHLJ19]. Variants of GNNs result from modifications to these 3 procedures: AGGREGATE, COMBINE and READOUT.

Implementation of AGGREGATE-COMBINE

There are several variants of GNNs of which some GNN variants are quite popular due to their successes in many different real-world problems. We provide some brief notes on their implementation, primarily, focusing on how the graph convolution operation is implemented in them. We focus mainly on the following variants of graph convolutions that are used in the research conducted in this dissertation: (1) GCN: spectral graph convolution [KW17], (2) k-GNN: multistage graph convolution [MRF+19], (3) GAT: graph convolution with attention [VCC+18], (4) GraphSAGE: simple-and-aggregate graph convolution [HYL17], and (5) ARMA: graph convolution with auto-regressive moving average [BGLA21].

Variant 1: GCN

Based on the spectral-based graph convolution as proposed by [KW17], this graph convolution uses a layer-wise (or iteration-wise) propagation rule for a graph with N vertices as:

$$\mathbf{H}^{(k)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\mathbf{H}^{(k-1)}\Theta^{(k-1)}\right) \tag{A.11}$$

where, $H^{(k)} \in \mathbb{R}^{N\times D}$ denotes the matrix of vertex representations of length D, $\tilde{A} = A + I$ is the adjacency matrix representing an undirected graph G with added self-connections, $A \in \mathbb{R}^{N\times N}$ is the graph adjacency matrix, I_N is the identity matrix, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, and $\Theta^{(k-1)}$ is the iteration-specific trainable parameter matrix, $\sigma(\cdot)$ denotes the activation function e.g. $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$, $\mathbf{H}^{(0)} = \mathbf{X}$, $\mathbf{X}$ is the matrix of feature-vectors of the vertices, where each vertex i is associated with a feature-vector X_i.

Variant 2: k-GNN

This graph convolution passes messages (vertex feature-vectors) directly between subgraph structures inside a graph [MRF+19]. At iteration k, the feature representation of a vertex is computed by using

$$h_u^{(k)} = \sigma\left(h_u^{(k-1)} \cdot \Theta_1^{(k)} + \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)} \cdot \Theta_2^{(k)}\right) \tag{A.12}$$

where, h_u^k denotes the vertex-representation of a vertex u at iteration k, $\mathcal{N}$ denotes the neighborhood function, σ is a non-linear transfer function applied component wise to the function argument, Θs are the layer-specific learnable parameters of the network.

Variant 3: GAT

This variant is based on aggregating information from neighbours with attention. This approach is popularly known as Graph Attention Network (GAT: [VCC+18]). This network assumes that the contributions of neighboring vertices to the central vertex are not pre-determined which is the case in the Graph Convolutional Network [KW17]. This adopts attention mechanisms to learn the relative weights between two connected vertices. The graph convolutional operation at iteration k is thereby defined as:

$$h_u^{(k)} = \sigma\left(\sum_{v \in \mathcal{N}(u) \cup u} \alpha_{uv}^{(k)} \Theta^{(k)} h_u^{(k-1)}\right) \tag{A.13}$$

where, h_u^k denotes the vertex-representation of a vertex u at iteration k; $h_u^{(0)} = X_u$ (the initial feature-vector associated with a vertex u). The connective strength between the vertex u and its neighbor vertex v is called attention weight, which is defined as

$$\alpha_{uv}^{(k)} = \text{softmax}\left(\text{LeakyReLU}\left(a^{\intercal}\left[\Theta^{(k)} h_u^{(k-1)} \parallel \Theta^{(k)} h_v^{(k-1)}\right]\right)\right) \tag{A.14}$$

where, a is the set of learnable parameters of a single layer feed-forward neural network, $||$ denotes the concatenation operation.

Variant 4: GraphSAGE

This graph convolution is based on inductive representation learning on large graphs, as proposed in [HYL17]. The convolution technique here is used to generate low-dimensional vector representations for vertices by learning how to aggregate feature information from the neighbourhood of the vertices. It adopts two steps: First, it samples a neighbourhood vertices of a vertex; Second, aggregate the feature-information from these sampled

vertices. GraphSAGE is used to found to be very useful for graphs with vertices associated with rich feature-vectors. The following is an iterative update of the vertex representations in a graph:

$$h_u^{(k)} = \sigma\left(h_u^{(k-1)} \cdot \Theta_1^{(k)} + \frac{1}{|\mathcal{N}(u)|} \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)} \cdot \Theta_2^{(k)}\right) \tag{A.15}$$

where, h_u^k denotes the vertex-representation of a vertex u at iteration k, σ is a non-linear transfer function applied component wise to the function argument, $\mathcal{N}$ denotes the neighborhood function, Θs are the layer-specific learnable parameters of the network.

Variant 5: ARMA

This graph convolution is inspired by the auto-regressive moving average (ARMA) filters that are considered to be more robust than polynomial filters [BGLA21]. The ARMA graph convolutional operation is defined as:

$$\mathbf{H}^{(k)} = \frac{1}{M} \sum_{m=1}^{M} \mathbf{H}_m^{(K)} \tag{A.16}$$

where, $\mathbf{H}^k$ denotes the vertex-representation matrix at iteration k, M is the number of parallel stacks, K is the number of layers; and $\mathbf{H}_m^{(K)}$ is recursively defined as

$$\mathbf{H}_m^{(k+1)} = \sigma\left(\hat{L}\mathbf{H}_m^{(k)}\Theta_2^{(k)} + \mathbf{H}^{(0)}\Theta_2^{(k)}\right) \tag{A.17}$$

where, σ is a non-linear transfer function, $\hat{L} = I - L$ is the modified Laplacian. The Θ parameters are learnable parameters.

Graph Pooling

In addition to the graph convolution methods mentioned above, graph pooling is a method that applies down-sampling to graphs. This operation allows to obtain refined graph representations at each layer. Like in convolutional neural networks, a (graph-)pooling operation follows a (graph-)convolution operation. The primary aim of including a graph pooling operation after each graph convolution is that this operation can reduce the graph representation while ideally preserving important structural information.

In the research conducted in this dissertation, we use a popular structural-attention based graph pooling method [LLK19]. This method uses the graph convolution defined in Equation (A.11) to obtain a self-attention score as given in Equation (A.18) with the trainable parameter replaced by $\Theta_{att} \in \mathbb{R}^{N \times 1}$, which is a set of trainable parameters in

the pooling layer.

$$Z = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\mathbf{X}\Theta_{att}\right) \tag{A.18}$$

Here, $\sigma(\cdot)$ is the activation function e.g. tanh.

Implementation of READOUT

The graph-convolution and graph-pooling operations described in the preceding subsection allows an iterative construction of vertex-representations. To deal with the problem of graph classification (as is the case in this dissertation), we need to represent an input graph as a "flattened" fixed-length feature-vector that can then be used with a standard fully-connected multilayer neural network (e.g. Multilayer Perceptron) to produce a class-label. To construct this graph-representation (mostly, a dense real-valued feature-vector, also called a *graph-embedding*), we use hierarchical graph-pooling method proposed by [CVJ+18].

The hierarchical pooling method is implemented with two operations: (a) global average pooling, that averages all the learnt vertex representations in the final (readout) layer; (b) augmenting the representation obtained in (a) with the representation obtained using global max pooling, that seek to obtained the most relevant information and could strengthen the graph-representation. The term "hierarchical" refers to the fact that the above two operations (a) and (b) are carried out after each "convolution-pooling" operation in the GNN. The final graph representation is an aggregate of all the layer-wise representations by taking their sum. The output graph after each convolution-pooling block can be represented by a concatenation of the global average pool representation and the global max pool representation as

$$H_G^{(k)} = \text{avg}(\mathbf{H}^{(k)}) \,||\, \max(\mathbf{H}^{(k)}) \tag{A.19}$$

where, H_G^k denotes the graph-representation at iteration k; $\mathbf{H}^{(k)}$ denotes the matrix of vertex-representations after convolution-pool operations at iteration k as mathematically described in the preceding subsection; avg and max denote the average and max operations, which are computed as follows:

$$\text{avg}(\mathbf{H}^{(k)}) = \frac{1}{N}\sum_{i=1}^{N}\mathbf{H}_i^{(k)} \tag{A.20}$$

$$\max(\mathbf{H}^{(k)}) = \max_{i=1}^{N}\mathbf{H}_i^{(k)} \tag{A.21}$$

Here, $\mathbf{H}_i^k$ denotes the representation for the ith vertex of the graph; N is the number of nodes in the graph.

The final fixed-length representation after iteration K for the whole input graph is

then computed by the element-wise sum, denoted as $\oplus$, of these intermediate graph-representations in Equation (A.19):

$$H_G^{(K)} = \oplus_{k=1}^{K} H_G^{(k)} \tag{A.22}$$

A.2 Inductive Logic Programming (ILP)

Here we provide some conceptual details on ILP. By no means, this section is intended to provide a complete technical overview of ILP; for which the reader could refer to [Mug91, Md94].

ILP is a symbolic machine learning method that construct models from data and background knowledge. It is a formal framework for symbolic machine learning and provides practical algorithms for inductively learning relational descriptions for data (in the form of programs in first-order logic) from training examples and background knowledge, both uniformly represented in relational form (first-order logic). We describe the learning problem in ILP with an illustration of the classic east-west train classification problem, proposed by Michalski [Mic80, MMPS94], which we have discussed extensively in Chapter 3.

Example A.1. *Michalski's east-west train classification problem has the following setting:*

- *Consider there are 10 trains, 5 going east and 5 going west as re-shown in the figure below. Each train can consist of more than one cars.*

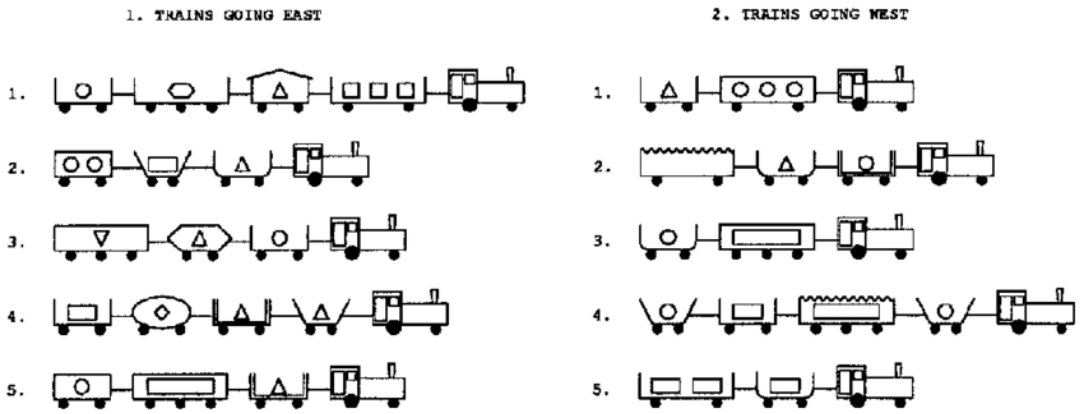

Figure A.3: Michalski's trains problem; adapted from [Mic80, MMPS94].

- *In a relational representation these trains are represented as facts:*

  Positive examples, E^+**:** $eastbound(east1)$, ..., $eastbound(east5)$;

  Negative examples, E^-**:** $eastbound(west6)$, ..., $eastbound(west10)$.

- *Each train comprises a set of locomotive pulling wagons; whether a particular train is travelling towards the east or towards the west is determined by some properties of that train.*

- *The learning task here is to determine what governs which kinds of trains are Eastbound and which kinds are Westbound.*

- *The following background knowledge about each wagon (or car) in the train are available: which train it is part of, its shape, how many wheels it has, whether it is open (i.e. has no roof) or closed, whether it is long or short, the shape of the things the car is loaded with. In addition, for each pair of connected wagons, knowledge of which one is in front of the other can be extracted.*

- *Let us assume the following background knowledge for the cars in train east*1*:*

  $short(car12)$, $short(car14)$, $long(car11)$, $long(car13)$,
  $closed(car12)$, $open(car11)$, $open(car13)$, $open(car14)$,
  $in\_front(car11, car12)$, $in\_front(car12, car13)$, $in\_front(car13, car14)$,
  $shape(car11, rectangle)$, $shape(car12, rectangle)$,
  $shape(car13, rectangle)$, $shape(car14, rectangle)$,
  $load(car11, rectangle, 3)$, $load(car12, triangle, 1)$,
  $load(car13, hexagon, 1)$, $load(car14, circle, 1)$,
  $wheels(car11, 2)$, $wheels(car12, 2)$, $wheels(car13, 3)$, $wheels(car14, 2)$,
  $has\_car(east1, car11)$, $has\_car(east1, car12)$,
  $has\_car(east1, car13)$, $has\_car(east1, car14)$.

  *Here cars are uniquely identified by constants of the form carxy, where x is number of the train to which the car belongs and y is the position of the car in that train. For example, car*14 *refers to the fourth car behind the locomotive in the first train.*

- *Then an ILP system could generate the following hypothesis:*

$$eastbound(X) \leftarrow has\_car(X, Y), short(Y), closed(Y)$$

  meaning, a train is eastbound if it has a car which is both short and closed.

The hypothesis about an eastbound train is simply a relational description. The core technique developed within ILP to constrain the search for such relational descriptions for data is mode-directed inverse entailment (MDIE), which was introduced by Muggleton in [Mug95]. More detailed description is provided in Chapter 5. For completeness, we briefly describe MDIE and the related concepts below.

Mode-Directed Inverse Entailment (MDIE)

Mode-directed Inverse Entailment (MDIE [Mug95]) is a technique for constraining the search for explanations for data in Inductive Logic Programming (ILP). Given a relational data instance e, background knowledge B, a set of modes M, a depth-limit d, and language restriction $\mathcal{L}$, MDIE identifies a most-specific logical formula $\bot_{B,\mathrm{M},d}(e)$ that contains all the relational information in B that is related to e.

In MDIE, Background knowledge B is given as a logic theory in the form of Horn clauses; data instances are provided as a set of positive and negative examples, $E = E^+ \wedge E^-$. Based on MDIE, a correct hypothesis H is a conjunction of definite clauses $H = D_1 \wedge D_2 \wedge \ldots$ which satisfies the following logical requirements:

- Prior necessity ($B \not\models E^+$): It forbids any generation of a hypothesis as long as the positive examples are explainable without it. More clearly, this requirement checks that at least one positive example cannot be explained by the background knowledge B alone.
- Posterior Sufficienty ($B \wedge H \models E^+$): It requires any generated hypothesis h to explain all positive examples E^+.
- Prior satisfybility ($B \wedge H \not\models \square$): This is a weak consistency requirement that forbids generation of any hypothesis h that contradicts the background knowledge B.
- Posterior satisfybility ($B \wedge H \wedge E^- \not\models \square$): This is a strong consistency requirement that forbids generation of any hypothesis that contradicts the negative examples (if given).

Here $\models$ denotes logical consequence and $\square$ denotes a contradiction. MDIE implementations attempt to find the most-probable H, given B and the data E^+, E^-. The key concept used in [Mug95] is to constrain the identification of the D_j using a *most-specific clause*. Let us look at the following example (re-used here from Chapter 5):

Example A.2. *In the following, capitalised letters like X, Y denote variables. Let B:*

$$parent(X,Y) \leftarrow father(X,Y)$$
$$parent(X,Y) \leftarrow mother(X,Y)$$
$$mother(jane, alice) \leftarrow$$

e:

$$gparent(henry, john) \leftarrow father(henry, jane), mother(jane, john)$$

The most-specific clause for the example $gparent(henry, john)$, denoted by $\bot_{B,\mathrm{M},d}(e)$ is:

$$gparent(henry, john) \leftarrow father(henry, jane), mother(jane, john), mother(jane, alice), parent(henry, jane), parent(jane, john), parent(jane, alice)$$

Language Restrictions

In this section we intuitively describe what "mode" means in ILP. Given a set of examples and background knowledge, the space of possible hypotheses in ILP tends to be very large [Mug91, Rae10]. To reduce the search space is to be more specific about how the predicates in the hypothesis (Horn) clauses will look like. There are two possible ways this can be achieved: (1) by limiting the number of existentially quantified variables allowed in the learned clauses; and (2) by explicitly specifying what the learned hypothesis will look like, in terms of restrictions on both their head and their body. In particular, for each predicate in the body of a hypothesis clause, it is possible to specify whether an argument in the predicate is to be a ground term, a new variable or a variable given in the head. This is done in some popular ILP systems such as Aleph [Sri01]. This is called a language restriction in ILP and it drastically reduces the search time by confining the search to be carried out in a limited search space. In implementations, this is carried out by providing a set of mode declarations [Mug95]. There are two kinds of mode declarations:

modeh declaration This is a mode declaration that dictates what the head of the hypothesis clauses will look like.

modeb declaration This is a mode declaration that stipulates the format of the predicates in the body of the clauses.

Let us look the following examples:

Example A.3. *For our grandparent example above, the following are some mode declarations:*

$$\boxed{\begin{array}{l} modeh(gparent(+person, -person)) \\ modeb(father(+person, -person)) \\ modeb(mother(+person, -person)) \\ modeb(parent(+person, -person)) \end{array}}$$

The first mode declaration, modeh is for the head of a hypothesis clause that stipulates that the head literal will have a predicate name gparent and that the first argument will take in a given person name (specified by '+', called input) and second argument will return a person name (specified by '-', called output). This means that the learned predicate will take in the name of a person and return its grand parent, as required. Similar description applies to the body literals of the clause, where there can be three kinds of body literals; father, mother, parent.

Example A.4. *There can be mode declaration with argument in the predicate being specified with '#'. Let us consider the following two mode declarations:*

$$\boxed{\begin{array}{l} modeh(class(+flower, \#category)) \\ modeb(colour(+flower, \#colourname)) \end{array}}$$

The modeh declaration here stipulates that the predicate name class will take a given flower (specified by '+') and the second argument will return a ground instance (specified by '#') of the type category. Similar description applies to the modeb declaration.

In addition to the mode declarations, the language restriction requires few more parameters such as the 'depth-limit', denoted by d, and the number of literals in the body of a clause. Informally, depth refers to the depth of a term (arguments) in the literals (predicates) appearing the body of a hypothesis clause. More details on this parameter is provided in Chapter 5. The language restriction is often referred to as the 'depth-limited mode language' in ILP.

Some ILP Learning Systems

There are several implementations of ILP learning systems. Probably one of the oldest is FOIL [Qui90], and the newest is Popper [CM21]. There are close to 20 implementations of ILP learning systems (as per [CD20, CDEM22]) in the last 3 decades that are based on considerably different techniques. It is difficult to provide details on these learning systems in this dissertation, and therefore, we direct the reader to some relevant discussions on some of these systems in the following survey: [CD20]. In this section, we briefly focus first on Progol [Mug95], an ILP system that is based on MDIE as discussed earlier. Progol is a precursors to one of the most popular ILP system, Aleph [Sri01]. In general, the technique of MDIE and the Aleph system are extensively used in this dissertation.

Progol

Progol is one of the most important ILP systems that has inspired development of many other ILP systems, including Aleph [Sri01]. Progol combines the technique of MDIE with general-to-specific search through a refinement graph representing the discrete hypothesis space. The space is bounded by an empty clause (at the top of the refinement graph) and a most-specific clause (at the bottom of the refinement graph) that entails a positive example, given a set of mode-declarations and background knowledge. A simple diagrammatic representation of the bounded search space in Progol is shown in Figure A.4. The search of an optimal hypothesis is performed by A*-search, over clauses which subsume the most-specific clause in the refinement graph.

For more details on Progol, the reader is directed to: [Mug95] and [Mug96]. A more concise description is available at [Rob97]. Next we describe a successor of Progol, called Aleph, in more detail.

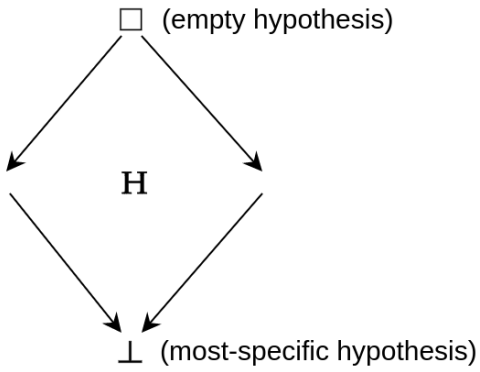

Figure A.4: Bounded search space in Progol.

Aleph

Aleph [Sri01] is based on the method of *inverse entailment* [Mug95] and uses a bottom-up learning mechanism where it restricts the hypothesis space by constructing a bottom-clause from the data and background-knowledge. Despite being at least 30 years old since its inception, Aleph remains to be the one of the most popularly used ILP system in relational learning research. The primary reason of its popularity could be its ease-of-use nature and the software is accompanied by a detailed user-manual [Sri01]. Aleph is written in Prolog, and its implementation is influenced by Progol [Mug95]. The learning in Aleph is based on the learning from entailment.

To construct a hypothesis, Aleph starts with an empty hypothesis and employs the following steps:

(1) Select a positive example to generalise. If none exists, stop and return the current hypothesis; otherwise proceed to the next step.

(2) Construct the most specific clause (the depth-limited bottom-clause) that is consistent with the mode declaration and entails the example.

(3) Search for a clause more general than the bottom-clause and has the best *score*.

(4) Add the clause to the hypothesis and remove all the positive examples covered by it. Return to Step 1.

Next we elaborate on the approaches to steps 2 and 3 below.

Step 2 The purpose of constructing a bottom-clause for an example is to bound the search in Step. (3) above. In general, a bottom-clause could be of infinite length (infinite cardinality). Aleph uses a depth-limited mode language to restrict them. During the search space, Aleph only considers the clauses that are generalisations of the bottom-clause, all of which entail the example.

Step 3 Aleph starts the search from the most general hypothesis and specialises it (by adding literals from the bottom clause) until it finds the best hypothesis. This results in a discrete space of clauses forming a search lattice. Figure A.5 shows the hypothesis space for the grandparent example. Aleph evaluates each clause in the lattice and assigns a score based on how well the training set is described by the clause. The default evaluation measure is the coverage as $P - N$, where P and N are the numbers of positive and negative examples, respectively, entailed by the clause. The users could also provide other predefined measures for search. Aleph tries to specialise a clause by adding literals to the body of the clause, which it selects from the bottom-clause or by instantiating variables. Each specialisation of a clause is called refinement. If Aleph finds the best clause that satisfies some specified constraint on the evaluation score, it adds it to the hypothesis, and remove all the positive examples covered by the new hypothesis, and returns to Step (1).

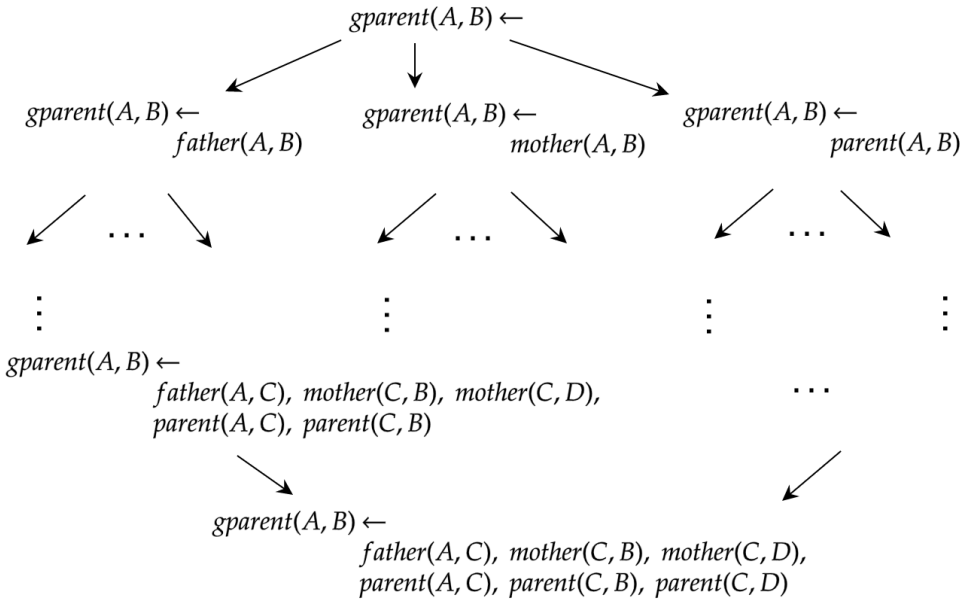

Figure A.5: A fragment of the hypothesis space in Aleph for the grandparent example, bounded by the most general hypothesis (at the top) and the most specific hypothesis (at the bottom).

Aleph offers several functionalities such as: constraint learning, mode learning, abductive learning, feature construction. In Chapter 3, the feature construction functionality of Aleph is extensively used. More details on Aleph can be found in [Sri01].

Appendix B

Additional Experimental Details

B.1 Details relevant to Chapter 3

Bottom-Clause Propositionalisation (BCP [FZG14]) constructs propositions using the most-specific clauses returned by the ILP system Aleph given the background-knowledge B, modes M and depth-limit d. In BCP, each data instance is represented using a Boolean vector of 0's and 1's, depending on the value of propositions (constructed manually or automatically) for the data instance (the value of the i^{th} dimension is 0 if the i^{th} proposition is false for the data-instance and 1 otherwise). The propositions represent the relational features constructed from the literals of a bottom-clause in ILP. For the construction of these Boolean features using BCP, we use the code available at [Jan20]. The resulting dataset is used to construct an MLP model. Our construction of MLP using BCP features is based on the following setup:

- The MLP is implemented using Tensorflow-Keras [C+15].
- The number of layers in MLP is tuned using a validation-based approach. The parameter grid for number of hidden layers is: $\{1, 2, 3, 4\}$.
- Each layer has fixed number of neurons: 10.
- The dropout rate is 0.5. We apply dropout [SHK+14] after every layer in the network except the output layer.
- The activation function used in each hidden layer is `relu`.
- The training is carried out using the Adam optimiser [KB15] with learning rate 0.001.
- Additionally, we use early-stopping [Pre98] with a patience period of 50 to control over-fitting during training.

B.2 Details relevant to Chapter 5

Mode-Declarations

We use the ILP engine, Aleph [Sri01] to construct the most-specific clause for a relational data instance given background-knowledge, mode specifications and a depth. The mode-language used for our main experiments in the chapter is given below:

```
:- modeb(*,bond(+mol,-atomid,-atomid,#atomtype,#atomtype,#bondtype)).
:- modeb(*,has_struc(+mol,-atomids,-length,#structype)).
:- modeb(*,connected(+mol,+atomids,+atomids)).
:- modeb(*,fused(+mol,+atomids,+atomids)).
```

The '#'-ed arguments in the mode declaration refers to type, that is, #`atomtype` refers to the type of atom, #`bondtype` refers to the type of bond, and #`structype` refers to the type of the structure (functional group or ring) associated with the molecule.

Experiments with ILP Benchmarks

The seven datasets are taken from [SKB03]. These datasets are some of the most popular benchmark datasets to evaluate various techniques within ILP studies. For the construction of BotGNNs, the following details are relevant:

- There is background knowledge available for each dataset.
- There are 10 splits for each dataset. Therefore, for each test-split we construct BotGNNs (all 5 variants), using 8 of rest splits as training set and the remaining 1 split as a validation set.
- Since these datasets are small (few hundreds of data instances), we could manage to perform some hyperparameter tuning for construction of our BotGNNs. The parameter grids for this are: $m : \{8, 16, 32, 64, 128\}$, batch-size: $\{16, 32\}$, and learning rate: $\{0.0001, 0.0005, 0.001\}$.
- Other details are same as described in the main BotGNN experiments.
- We report the test accuracy from the best performing BotGNN variant.

B.3 Details relevant to Chapter 6

A proxy for the results of hit confirmation assays is constructed using the assay results available for the target. This allows us to approximate the results of such assays on molecules for which experimental activity is not available. Of course, such a model is only

possible within the controlled experimental design we have adopted, in which information on target inhibition is deliberately not used when constructing the discriminator in D and generator in G2. In practice, if such target-inhibition information is not available, then a proxy model would have to be constructed by other means (for example, using the activity of inhibitors of homologues).

We use the state-of-the-art chemical activity prediction package Chemprop.[1] We train a Chemprop model using the data consisting of JAK2 inhibitors and their pIC50 values. The parameter settings used are: `class-balance` = TRUE, and `epochs` = 100 (all other parameters were set to their default values within Chemprop). Chemprop partitions the data into 80% for training, 10% validation and 10% for test. Chemprop allows the construction of both classification and regression models. The performance of both kinds of models are tabulated below:

Partition	Classification (AUC)	Regression (RMSE)
Valid	0.9472	0.6515
Test	0.8972	0.6424

The classification model is more robust, since pIC50 values are on a log-scale. We use the classification model for obtaining the results in Chapter 6 (see Figure 6.7), and we use the prediction of pIC50 values from the regression model as a proxy for the results of the hit-confirmation assays.

[1]It is likely that a BotGNN with access to the information in B_D along with the Chemprop prediction would result in a better proxy model. We do not explore this here.