Master's thesis

Supervised Bayesian network structure learning with equivariant models

Otto Rosenberg

May 16, 2023

Supervisor(s): professor Mikko Koivisto

Examiner(s): Etunimi Sukunimi
Etunimi Sukunimi

UNIVERSITY OF HELSINKI
FACULTY OF SCIENCE

PL 64 (Gustaf Hällströmin katu 2a)
00014 Helsingin yliopisto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Theoretical and Computational Methods

Tekijä — Författare — Author

Otto Rosenberg

Työn nimi — Arbetets titel — Title

Supervised Bayesian network structure learning with equivariant models

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	May 16, 2023	54

Tiivistelmä — Referat — Abstract

Bayesian networks (BN) are models that map the mutual dependencies and independencies between a set of variables. The structure of the model can be represented as a directed acyclic graph (DAG), which is a graph where the nodes represent variables and the directed edges between variables represent a dependency. BNs can be either constructed by using knowledge of the system or derived computationally from observational data. Traditionally, BN structure discovery from observational data has been done through heuristic algorithms, but advances in deep learning have made it possible to train neural networks for this task in a supervised manner. This thesis provides an overview of BN structure discovery and discusses the strengths and weaknesses of the emerging supervised paradigm. One supervised method, the EQ-model, that uses neural networks for structure discovery using equivariant models, is also explored in further detail with empirical tests. Through a process of hyperparameter optimisation and moving to online training, the performance of the EQ-model is increased. The EQ-model is still observed to underperform in comparison to a competing score-based model, NOTEARS, but offers convenient features, such as dramatically faster runtime, that compensate for the reduced performance. Several interesting lines of further study that could be used to further improve the performance of the EQ-model are also identified.

Avainsanat — Nyckelord — Keywords

deep learning, neural networks, Bayesian networks, machine learning, causal models

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Basics of deep learning **3**

2.1 Optimisation 5

2.2 Hyperparameters 8

2.3 Types of neural networks 10

3 Basics of Bayesian networks **11**

3.1 Motivation 15

3.2 Constructing a DAG 15

3.2.1 Constraint-based algorithms 17

3.2.2 Score-based algorithms 18

3.2.3 Hybrid algorithms 21

4 Supervised structure learning **23**

4.1 EQ-model 24

4.1.1 Evaluation and results 26

4.2 Other supervised models 27

5 Methodology **29**

5.1 Online training 30

5.2 Synthetic data-generation 30

5.3 Model configuration 31

5.4 Hyperparameter optimisation 32

5.5 Performance metrics 33

5.6 Sources of error 34

5.7 Acyclicity constraint 35

6 Experimental Results **37**

6.1 Offline and NOTEARS comparison 37

6.2 Transferability 37

6.3 Ensemble training 39
6.4 Confidence reporting 43
6.5 Visualisation 46
6.6 Runtime of algorithms 46

7 Conclusion **49**

Bibliography **50**

1. Introduction

Bayesian networks are popular tools for modelling a wide range of phenomena in fields as varied as cellular signalling networks in biology (Qin et al., 2012), genetic epidemiology (Su et al., 2013), climate change adaptation (Terzi et al., 2019), risk factors of transportation accidents (Zhao et al., 2012), and the economics of carbon emission trading (Wang and Zhao, 2021) to name only a few examples.

A Bayesian network describing a system can either be constructed using formal or intuitive knowledge of the domain, or it can be automatically inferred from the observed behaviour of the system. This latter approach is known as Bayesian network structure discovery. It is a computationally challenging problem, and its general formulation falls into the category of NP-hard optimisation problems (Chickering et al., 1996, 2012). Exact optimisation is only tractable for systems with up to two or three dozen variables, while many fields have the need for structure discovery for far larger systems. Approximative algorithms exist to fulfil this need to a certain extent and can reach a high degree of reconstruction accuracy despite having no explicit optimality guarantees, but demand still exists for ever-increasing accuracy and scalability to larger networks.

In recent years, the explosive development in the field of deep learning has enabled a new approach to this optimisation problem: supervised structure discovery using neural networks. Supervised neural networks learn a function using labelled training examples automatically, bypassing the need for heuristic algorithms to be manually crafted and optimised for best performance. There are already promising results in leveraging neural networks for solving other combinatorial optimisation tasks, such as the Minimum Vertex Cover and Travelling Salesman problems (Dai et al., 2018).

The potential benefits of this approach are numerous. As neural networks are universal approximators with myriad applications across a wide range of fields and industries, research into these tools is intense, and the rate of their development is rapid. This includes facets ranging from theoretical discoveries about the properties of neural networks to improvements in computational performance through software optimisation and widespread adoption of deep learning specific hardware acceleration.

Chapters 2 and 3 will establish background on deep learning methods and the theory of Bayesian networks, including an overview of conventional structure discovery algorithms. Chapter 4 explores the new paradigm of supervised structure discovery, including a deeper dive into the EQ-model (Li et al., 2020), among the earliest supervised structure discovery algorithms to be introduced. Finally, chapters 5 and 6 contain empirical analysis of the EQ-model and its performance in comparison to advanced conventional structure discovery algorithms, expanding on the limited testing done by the creators of the EQ-model.

2. Basics of deep learning

As the subject of this thesis is a supervised learning task, this section will only cover deep neural networks in a supervised learning setting. Supervised deep learning is the process of learning a function between the model inputs, also known as features, and the desired outputs, also known as targets. The other categories of deep learning that can be identified are semi-supervised learning, where a function is still learned but only some data points are labelled; unsupervised learning, where the model attempts to learn the distribution that generated the inputs; and reinforcement learning, where a model is incrementally improved through a feedback loop and an objective function. However, most of the basic principles explained here apply equally to these other categories of deep learning as well.

Neural networks are machine learning models that are comprised of a large number of elements called *neurons*. An individual neuron is generally a function $g : \mathbb{R}^k \to \mathbb{R}$ that is a combination of a linear function with learnable parameters and a fixed non-linear function, known as an *activation function*:

$$g(x) = \sigma\left(\sum_{i=1}^{k} w_i x_i + b\right), \tag{2.1}$$

where σ is the activation function and w and b are learnable parameters, also known as *weights* and *biases* respectively. The neurons can be arranged in parallel and sequentially, with each sequential set of neurons known as a layer. Neural networks are formed by stacking multiple layers into a deep network, which is why machine learning based on multi-layer neural networks is also known as deep learning. The first layer of a neural network is the input layer, and the last layer is the output layer, between which there can be any number of hidden layers, known as such because they are not exposed to the user. This structure is illustrated in figure 2.1 (Patterson and Gibson, 2017, Chap. 2).

Neurons often have many inputs, and the most basic fully connected networks have a connection from each neuron in the previous layer to each neuron in the next layer, which results in a very large number of weights in bigger neural networks. For example, in figure 2.1, the four orange neurons in the hidden layer each have two weights from their two inputs and a bias term, which means the hidden layer has in

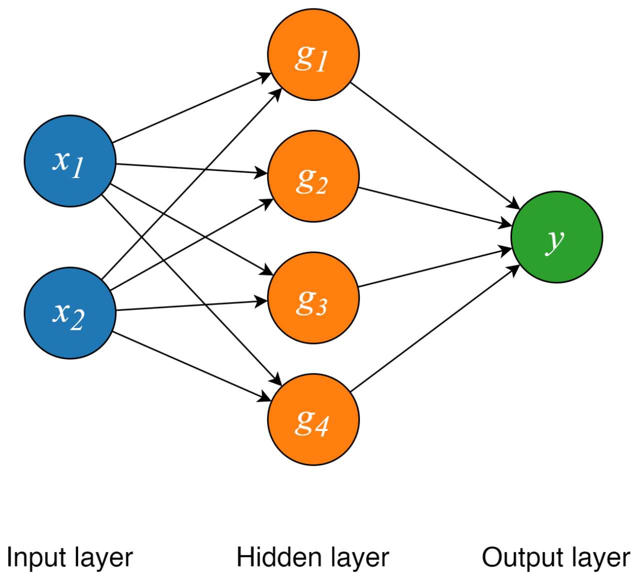

Figure 2.1: A simple neural network

total 12 learnable scalar parameters. A typical fully connected neural network can have millions of parameters across hundreds of neurons (Goodfellow et al., 2016, Chap. 6).

In modern deep learning applications, the default recommendation is to use the rectified linear unit (ReLU) activation function, defined as $\sigma(x) = \max\{0, x\}$ (Nair and Hinton, 2010), though others are also used in specific applications. A variant of ReLU sometimes used is leaky ReLU, defined as

$$\sigma(x) = \begin{cases} \alpha x, & \text{if } x < 0 \\ x, & \text{if } x \geq 0, \end{cases} \tag{2.2}$$

where $\alpha \in [0, 1]$. The sigmoid activation function, defined as $\sigma(x) = \frac{1}{1-e^{-x}}$, is often used in the output layer when a model is used to estimate probability, for the purposes of ensuring that output values are strictly confined within the $[0, 1]$ interval.

Because a sum of linear functions is also always a linear function, the non-linearity introduced by the activation function is the critical step that allows the neural network to approximate any function. According to the universal approximation theorem, a neural network with just a single hidden layer can approximate any continuous function as the number of neurons is increased (Hornik et al., 1989). However, the same model capacity can be achieved in practise with a far lower computational cost and fewer neurons by increasing the number of hidden layers between the input and output. A model that has the ability to replicate complex functions, such as deep learning models, is said to have high *capacity* (Goodfellow et al., 2016, Chap. 12).

2.1 Optimisation

A neural network learns its parameters during training using an algorithm called *back-propagation.* The first step of a training cycle is forward propagation, where the inputs x are simply carried through the network and the weights, biases, and activations are applied at each layer, which finally results in an output $p = f(x)$. The output is then compared to the true target value y and a *loss function* $\mathcal{L}(p, y)$ is used to measure the error between the two. Loss functions can be broadly divided between those used for regression problems and those used for classification problems, with mean squared error being a typical option for regression tasks and cross-entropy and hinge loss being common choices for classification problems. Of specific interest later on is the binary cross-entropy loss, also known as log loss, which is used for binary classification problems and defined as

$$\mathcal{L}(y, p) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \cdot \log(p_i)\right] + \left[(1 - y_i) \cdot \log(1 - p_i)\right], \tag{2.3}$$

where $y \in \{0, 1\}$ is the true value and $p \in [0, 1]$ is the predicted probability of y (Patterson and Gibson, 2017, Chap. 2).

Next, the gradients of the loss with respect to the parameters are computed procedurally, starting at the output layer and working backwards through the hidden layers until the input layer is reached. At each layer, the gradient of the loss with respect to the output of that layer is computed, as well as the gradient of the output with respect to the inputs of the layer. These two gradients are then multiplied together to obtain the gradient of the loss with respect to the inputs of the layer, which are then passed to the previous layer. This process continues until the gradients of the loss with respect to the weights and biases at each layer, $\nabla\mathcal{L}$, are obtained.

Once the gradients have been computed, the parameters are updated using an *optimiser* that employs gradient descent to optimise the weights. This involves taking small steps in the negative direction of the gradient towards reduced loss. This process is repeated for multiple epochs, with the hope that the loss will eventually converge to a minimum and the network will have learned the desired function. The size of the step along the gradient is called *learning rate*, and is one of the user-adjustable parameters in the model. A higher learning rate will increase the rate at which the model converges to a minimum loss state, but setting a value too high can also lead to instability and divergence. The basic gradient descent algorithm (Ruder, 2016) has the following weight update rule:

$$w_{n+1} = w_n - \eta\nabla\mathcal{L}, \tag{2.4}$$

where η is the learning rate. Some optimisation algorithms employ additional techniques to speed up and stabilise convergence, such as momentum. Momentum updates the weights using a combination of the current gradient and previous gradient, which helps the model move steadily towards minimum loss and can avoid getting stuck in local minima or saddle points in the loss manifold (Mukhopadhyay, 2018, Chap. 5). Some optimisation algorithms, such as the Adaptive Moment Estimation (Adam) optimiser (Kingma and Ba, 2014), can also dynamically adjust the learning rate or momentum.

The main metric for deep learning models is *generalisation accuracy*, which refers to the performance of the model on data that it has not been trained on. To measure generalisation accuracy, deep learning models split the available data into two separate sets, a training data set and a testing data set. The model is trained by giving it a portion of the training data set, called a *batch*, until the entire training data set has been used. This cycle through the training data set is called an *epoch* and training is continued for as many epochs as is needed or until performance stops improving. Updating weights in training is done using only the training data set, and the testing data set is kept unseen to the model so that it can be used after each epoch to gauge its generalisation accuracy.

Optimisation algorithms that use the entire batch each time are called batch gradient methods. Some algorithms only use a single data point or a subset of data points (called a minibatch) from each batch for training, and these are called stochastic methods. The benefit of stochastic optimisation algorithms is that by taking an unbiased sample from the training data set, training time can be lowered without a substantial hit to the accuracy of the gradient estimation. For this reason, stochastic gradient descent (SGD) is a common choice of optimisation method in deep learning (Ruder, 2016).

A further class of optimisation algorithms are online methods, which draw samples from a dynamically generated stream, rather than from a fixed-size training set that is iterated over multiple epochs (Goodfellow et al., 2016, Sect. 8.1). This method is necessary when the function between the input and output changes during training, an effect known as concept drift (Gama et al., 2014). This can happen in situations such as when modelling user preferences on social media websites or online shopping platforms based on their use history and other user data. However, online training can also be beneficial in static situations where concept drift is absent, as it allows synthetic training data to be generated on-demand. This enables the use of larger training data sets, beneficial for improving generalisation accuracy, without having to store every training sample in memory at once. This latter application of online training will be further investigated in chapters 5 and 6.

A common problem deep learning models face due to their high capacity is *overfitting*. Overfitting is a phenomenon where, rather than learning the desired, underlying patterns in the data, the model instead learns the random noise in the training data, essentially memorising it. Overfitting can be measured simply as the difference between the performance of the model on the training data and its performance on the testing data. Overfitting can be mitigated using *regularisation* methods such as early stopping, where the training is stopped before training loss converges to a minimum, or weight decay, where a penalty term that discourages large weights is added to the loss function. The online training paradigm can also be an effective way to curb overfitting, as increasing the training data set decreases the relative importance of any individual training data point and thus limits the ability of the model to memorise the training data.

The inverse problem is underfitting, where the effective capacity of the deep learning model becomes too constrained and it is unable to replicate the desired function with sufficient resolution. While it is easy to always keep a model large enough that it has, in principle, enough capacity to prevent underfitting, issues with the configuration of the model, such as a poorly selected learning rate or optimiser, can cause a large number or all of the parameters to get stuck, thus greatly lowering the effective capacity of the model. Underfitting can therefore be identified by the training accuracy improving little or not at all between training epochs (Patterson and Gibson, 2017, Chap. 1).

During the training process, models can face a runaway effect called exploding gradients, and are especially prone to encountering it if they are exceptionally deep or use certain activation functions. Exploding gradients involve the gradient of the loss function with respect to the weights growing very large, which in turn leads to the algorithm assigning very large values to the weights. This can lead to the training process failing to converge and instead increasing the training loss over epochs. A similar problem is vanishing gradients, where the gradients get very small, which causes many or all weights in the network to become zero. This can drastically slow training or lead to the training process getting stuck. Using weight decay can help avoid these effects, in addition to avoiding overfitting. Another regularisation technique that helps maintain reasonable weights is batch normalisation, where after every forward pass the weights are centred and normalised according to the mean and standard deviation of that batch. Also, an appropriately chosen activation function and learning rate can help avoid exploding and vanishing gradients (Goodfellow et al., 2016, Sect. 8.2.4).

While the iterative training process is computationally expensive and merely drives the loss to a small value instead of a guaranteed global minimum, it is necessary because the nonlinearities present in neural networks render their loss manifolds nonconvex. This means that the linear equation solvers or convex optimisation algorithms with exact convergence guarantees used to train traditional machine learning models, such as support vector machines and linear regressors, cannot be utilised. However, once the costly training process has been completed, use of the model only requires computing the forward propagation step once per input, which makes obtaining results from even very complex neural networks quite efficient (Goodfellow et al., 2016, Chap. 6.2).

2.2 Hyperparameters

While backpropagation with gradient descent makes for a very powerful method for optimising deep models, it cannot be used to optimise all parameters of a neural network. By its very nature, backpropagation pushes the model towards reduced loss on the training set, which is generally not a reliable measure of generalisation accuracy. This means any parameter that controls model capacity would be limitlessly increased by backpropagation, resulting in a high degree of overfitting and very low generalisation accuracy. These parameters controlling model capacity are called *hyperparameters* and include the optimiser learning rate, the depth of the network, the size of each layer, and regularisation. Because the hyperparameters need to be manually optimised in a separate process, usually a third separate set of data, called the validation set, is kept aside for this purpose (Goodfellow et al., 2016, Sect. 11.4).

The goal of hyperparameter search is to adjust the effective capacity of the model to match the complexity of the learning task. Effective capacity is determined by the combination of three factors. One of these is the representational capacity of the model, which is the maximum complexity of a function that the model can reproduce, and it is controlled primarily by the number and size of hidden layers. Another factor is the ability of the model to minimise the loss function used in training, which requires an appropriately tuned learning rate and optimiser settings. Finally, there is the degree to which the loss function and training process regularise the model, which is controlled by regularisation parameters such as the strength of weight decay. For example, even a model with a very large representational capacity could have a very low effective capacity if the learning rate is set too low or a too aggressive weight decay is used (Patterson and Gibson, 2017, Chap. 2).

Generalisation error typically follows a U-shaped curve when plotted as a function of one of the hyperparameters. At one end of the spectrum, the hyperparameter constrains the effective capacity of the model too much, rendering it unable to learn the desired function. This is the underfitting regime. At the other end of the spectrum is the overfitting regime, where the model capacity is too high and generalisation error

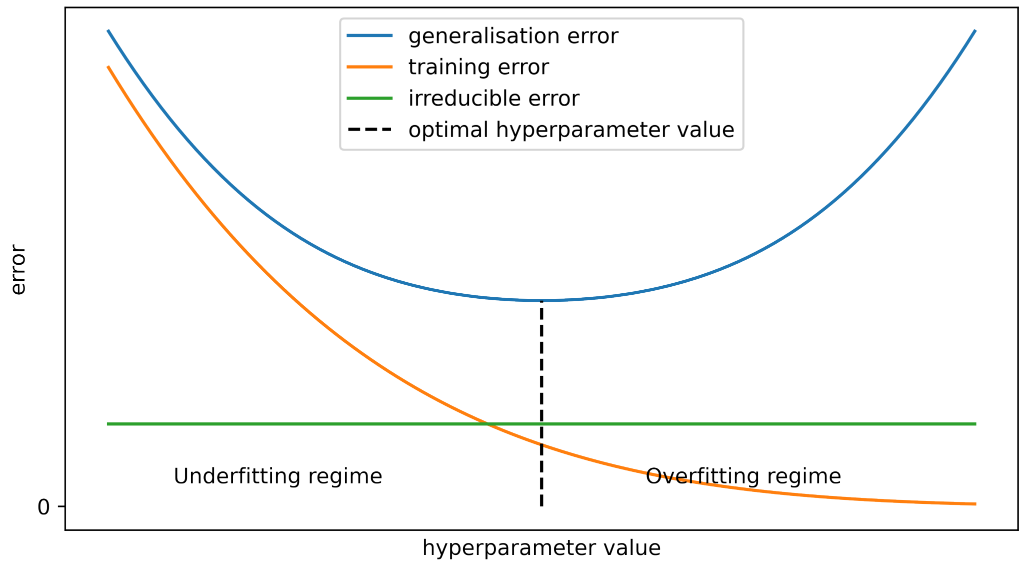

Figure 2.2: The relationship between hyperparameter values and model accuracy

is large due to the gap between training and test error. Between these two extremes lies the optimal value for the hyperparameter in question, where model capacity is appropriate for the learning task (Goodfellow et al., 2016, Sect. 11.4.1). This effect is illustrated in figure 2.2.

Not all hyperparameters can reach the full extent of this spectrum, however. For instance, the minimum value for weight decay is 0, at which point it has no effect on the model. Higher values only reduce model capacity by using a penalty term to reduce the effective range of values that weights and biases can take. Thus, the capacity of an already underfitting or appropriately tuned model that does not use weight decay cannot be increased to achieve an overfitting state by adjusting the strength of weight decay.

Among the most important hyperparameters to adjust is the learning rate. It controls the effective capacity of the model in a more subtle way than any other hyperparameter, since capacity is highest when the learning rate itself is appropriately tuned, not when it is particularly high or low. Setting the learning rate too low reduces capacity by slowing training and can lead training to get stuck at a point of high loss if the loss manifold is highly nonconvex. Setting the learning rate too high makes gradient descent prone to overshooting the optimum parameter value and can cause each iteration to increase total loss rather than reduce it (Goodfellow et al., 2016, Chap. 11.4.1).

Hyperparameters can either be tuned manually or by using an automatic optimiser. A common choice for simpler problems with few hyperparameters to optimise is a grid search, where the model is run using permutations of discrete, pre-binned values for the hyperparameters in question, and the model that yields the best validation set performance determines the optimal hyperparameter values. As the computational cost of grid search increases exponentially as the number of hyperparameters to optimise grows, random search is preferred for a larger optimisation task. A random search does repeated runs of the model and chooses the hyperparameter values randomly from a predetermined distribution. Random search has been shown by Bergstra and Bengio (2012) to perform significantly better than grid search and has compelling practical characteristics, such as the ability to be stopped at any time as opposed to the need of grid searches to complete the whole grid.

2.3 Types of neural networks

The simplest kind of neural network is a multi-layer perceptron, where each neuron in one layer is connected to every neuron in the following layer. This connectivity is why they are also called *fully-connected* (FC) neural networks, and it gives them high capacity, as each connection between a unique pair of neurons has a separately adjustable weight. While FC models can learn to approximate any function as per the universal approximation theorem, they are not ideally suited to certain kinds of workloads. Because FC models process inputs as tall column vectors, high-dimensional inputs like images must be flattened, which loses information about the potential correlations between adjacent values in the input array across all but one dimension.

For input data with strong local correlations in two or more dimensions, such as images, a common choice is a *convolutional neural network* (CNN), which first processes the input data using a filter that encodes local correlations into a lower-resolution array, which is then passed on to fully connected layers. The initial filter allows CNNs to retain structures in the input data after certain kinds of manipulations, such as shifting over the pixels in an image two steps to the right, a procedure that would significantly hurt the performance of an image classifier based entirely on fully connected layers.

Similarly to CNNs being particularly effective for image data, there are numerous other specialised neural network architectures that are engineered to perform well given the properties of other specific workloads. The case we will explore in chapter 4.1 is that of permutation invariant data, and how to build in constraints to the network architecture such that capacity and computational resources are not wasted on learning the permutations of the input data (Goodfellow et al., 2016, Chap. 13).

3. Basics of Bayesian networks

Bayesian networks (BN) are powerful representational devices introduced by Pearl and Verma (1987), that can be used to organise one's knowledge about a system of interdependent variables into a coherent whole. They combine probability theory and its ability to manipulate concepts such as likelihood, conditionality, and causality with graph theory into a mechanism that can effectively support reasoning under uncertainty.

A BN consists of two parts. The qualitative part is a *directed acyclic graph* (DAG), also known as the structure of the BN, and it encodes the dependencies between the variables of the system in question. The DAG is a set of nodes, each of which corresponds to one of the variables in the system under consideration, and a set of directed edges that connect certain nodes. The source of a directed edge is called a parent node, and the target is called a child node. If we make a causal assumption, then a directed edge $A \to B$ means that the variable A is a direct cause of B. The quantitative part is the set of *parameters* encoding the strength of the dependencies laid out in the DAG. The parameters take either the form of a conditional probability table, in the case of discrete-valued variables, or a conditional probability distribution, in the case of continuous-valued variables.

Using BNs, one can reason about the system by making predictions based on the conditional probabilities in the network. Formally, A and B are defined to be conditionally independent of C iff

$$P(A \cap B|C) = P(A|C)P(B|C). \tag{3.1}$$

Figure 3.1 shows an example DAG that models the relationship between select symptoms, medical conditions, and potential treatments. Conditional probability tells us that, for example, knowing whether a patient has a cough allows us to more accurately estimate their likelihood of getting an antibiotic prescription. This probability is written as $P(\mathit{Antibiotics}|\mathit{Cough})$ and could be computed if the conditional probability table for this DAG was known.

BNs represent conditional independence between variables through the graphical separation of their respective nodes in the DAG. This separation has special rules and

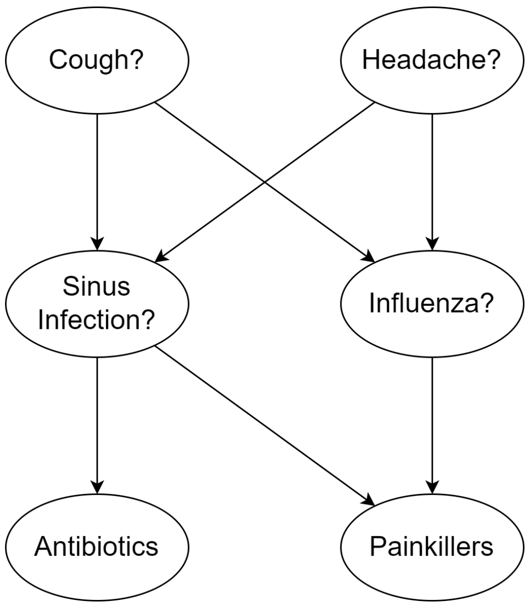

Figure 3.1: An example DAG

is known as *d-separation*. To determine whether two sets of disjoint variables X and Y are d-separated by a third set Z, we need to consider all possible paths from X to Y. D-separation occurs if at least one node on each of those paths fulfils either one of two conditions:

1. the node is a collider, and neither it nor any of its descendants are in Z, or
2. the node is not a collider, and it is in Z.

A collider is a structure where there are two incoming edges from parent nodes A and C to a child node B: $A \rightarrow B \leftarrow C$. If there is no edge between A and C, the structure is also known as a v-structure, or an unshielded collider.

To make sense of these conditions, consider the DAG in figure 3.1, where *Cough* and *Antibiotics* are connected through an intermediate node that is not a collider, meaning they are not d-separated. However, they become d-separated and thus conditionally independent if we condition on *Sinus infection*, because *Sinus infection* is on every possible path between the two nodes and is not a collider, blocking them (condition 2). The notation for this conditional independence is $Antibiotics \perp\!\!\!\perp Cough | Sinus\ Infection$. On the other hand, *Cough* and *Headache* are d-separated without conditioning because all possible paths between them go through a collider (condition 1). Conditioning on either *Sinus infection* or *Influenza*, however, creates a path and removes d-separation.

Using these rules, we can compute the probability of an event occurring given only partial information about the system, allowing us to make predictions even in situations where we do not have complete knowledge about the variables in the system. This ability to intuitively handle uncertainty in situations with noisy or incomplete data is one of the key benefits of BNs. Additionally, BNs allow us to easily incorporate new evidence into our model by updating the parameters, which makes them well suited for modelling highly dynamic systems.

The parameters and DAG together define a distribution that can be sampled to obtain random data points that represent a possible state for the underlying system. As we will see in section 3.2, this relationship can also be inverted in order to derive the parameters and DAG of a BN from a set of samples.

If we assume that every directed edge in the DAG signifies a direct causal relationship between the respective variables, every system with different causal relationships is defined by a unique DAG. However, some of these DAGs, when combined with appropriate parameters, define identical distributions. This means that data points obtained by sampling these distributions cannot be determined to have originated from any particular BN without further a priori knowledge of causal relationships in the system. For instance, a data set describing the values of two dependent variables cannot alone determine the direction of the causal link and thus could have been produced by the distribution defined by two different DAGs: $A \to B$ and $A \leftarrow B$.

DAGs that generate identical distributions in this fashion are said to belong to the same *equivalence class*. Any DAGs that share the same skeleton (same edges, ignoring directionality) and the same v-structures belong to the same equivalence class. A graph with the edge directions stripped from all but the v-structures is known as a partial DAG (PDAG). Some of the edge directions omitted by a PDAG can be inferred, given that one of the possible directions would create a v-structure, which would have had to be labelled by the PDAG. A graph where these edges have also been filled in is the most complete description of a given equivalence class and is known as a complete partial DAG (CPDAG). These graph types are illustrated in figure 3.2. Because the difference between the DAGs within an equivalency class cannot be determined from data alone, algorithms that learn the DAG from data can only distinguish between different equivalence classes and either return a randomly selected DAG within the equivalence class, or a CPDAG that conforms exactly to a single equivalence class.

It is also possible for a BN to produce data that has more independencies than the DAG implies. This situation occurs when the particular parameter values chosen for the original network are selected such that they precisely cancel out a dependence. For an example, we can turn back to the simple DAG $A \to B$, where we can assume the two variables take binary values. Its two variables are dependent, because there is a directed edge running between them. However, if we set the parameters to $P(B = 1|A = 1) = 0$

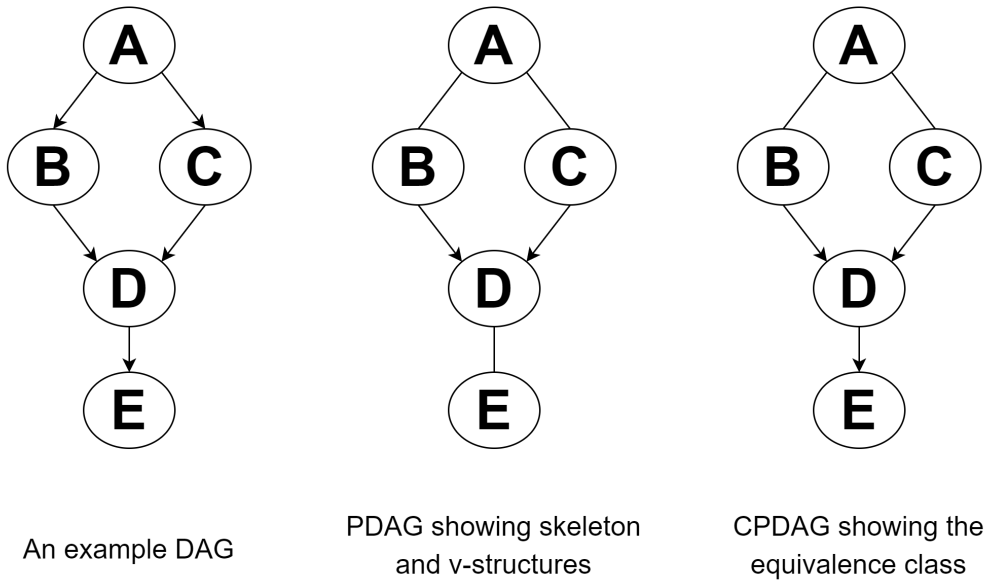
A
B
C
D
E
An example DAG
PDAG showing skeleton and v-structures
CPDAG showing the equivalence class

Figure 3.2: An example of a DAG and its corresponding PDAG and CPDAG

and $P(B = 1|A = 0) = 0$, then the variable B will always take the same value regardless of the value of A and thus create statistical independence between the variables, even though they are statistically dependent according to the BN structure.

A BN that has its parameters tuned this way is called *unfaithful* (Koski and Noble, 2011, Sect. 2.1). BN structure learning algorithms make the implicit assumption that the independencies present in the distribution are also reflected by the DAG; in other words, they assume that the distribution is faithful to the DAG. This assumption generally does not cause problems because, for a BN to exhibit unfaithfulness, careful tuning of the parameters is required, and even small perturbations will restore the conditional dependencies implied by the DAG. This can also be seen in the example above, where the independence is only created because the two parameters have exactly equal values, which is unlikely to occur in non-synthetic systems.

It is important to point out that if we do not make the causal assumption, the directed edges in a DAG do not necessarily correspond to the direction of the causal relationship between the connected variables. For example, in figure 3.1, there is a directed edge leading from head ache to influenza, even though causality actually flows the other way (diseases cause symptoms). It could be said that the edges denote the flow of information instead, such that the direction of an edge tells us that the state of the parent node can be used to infer the state of the child node. Sometimes this is equivalent to a causal link, but at other times the original cause itself, such as influenza, is not directly observable, and instead its state must be inferred from evidence, such as symptoms or the results of tests and examinations. Variables whose state can only be inferred from other, connected variables are called *latent variables* (Koski and Noble, 2011, Sect. 9.8).

3.1 Motivation

Given the explicit correspondence between the nodes in the DAG and variables in the system, BNs are useful for visualising and intuitively understanding complex, interdependent systems. This feature stands in particular contrast to deep learning models. While deep learning models can similarly be used to make accurate predictions about systems even given incomplete information and have uses in applications such as medical diagnostics, much like BNs, they are black boxes in nature. The values of the individual parameters in a neural network cannot be easily matched with specific variables, and this renders them impractical as devices for offering intuitive understanding and visualisation of complex, multi-variable systems. The ability of BNs to have their conditional probabilities easily adjusted in light of new evidence is also an advantage, as it avoids the need for the computationally expensive process of re-training a neural network.

Using a BN over a deep learning model is also beneficial in the case of limited data. While deep learning models are undeniably more accurate as the availability of training data grows, not having enough training data is a major weakness and can lead to overfitting. The lack of available data can be a particular concern in the supervised setting, where data also needs to be adequately labelled in order to be used for training. In some cases, data imputation methods, such as generative neural networks can be employed to fill in missing values, but these solutions can compromise the accuracy of the resulting model if used in excess. BNs, on the other hand, can organise even limited information into a usable model of the system. These factors mean that the modelling solution should be selected on a case-by-case basis. Furthermore, the dichotomy between these two methods has become less pronounced in recent years as a hybrid method called Bayesian neural networks (Lampinen and Vehtari, 2001), which incorporates Bayesian inference into deep learning models, has increased in popularity.

3.2 Constructing a DAG

Figure 3.3 shows the three main ways of constructing the DAG that encodes the structure of a BN. The first way is somewhat subjective, and involves a subject-matter expert simply laying out the structure according to their knowledge of the domain.

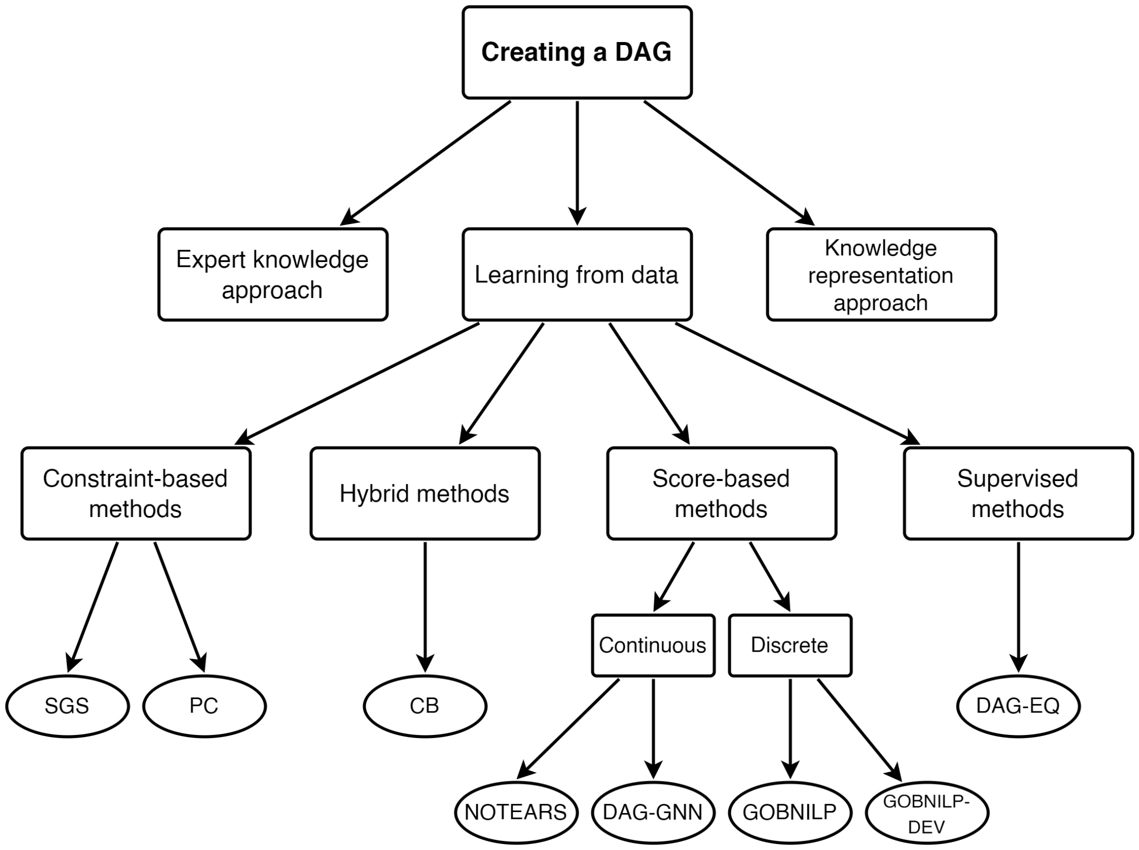

Figure 3.3: An overview of the various ways to construct a DAG

The second method, called the knowledge representation approach, uses formal knowledge such as system designs, genetic linkage analysis, or other explicit, a priori domain knowledge to automatically synthesise the DAG.

The topic of this thesis is the third approach to constructing the DAG: building it automatically from data, such as medical records or instrument readings. This entails having a set of data $D = [d_1, ..., d_m]$ consisting of m data instances $d \in \mathbb{R}^n$ that give a value to each of the n variables in the BN. The data can be discrete or continuous and is assumed to be independent and identically distributed in nature. BNs with continuous values are often evaluated as *linear Gaussian BNs*, where each value of a child variable is assumed to be a linear combination of the values of its parents with an added local noise component drawn from a normal distribution (Darwiche, 2009, Sect. 1.4.1). Most algorithms discussed here process DAGs and CPDAGs as *adjacency matrices*, which are square matrices where the rows and columns correspond to the nodes in the graph, with a value of 1 denoting a directed edge between the two nodes and a value of 0 denoting an edge not being present.

3.2.1 Constraint-based algorithms

Traditionally, there have been three different approaches to BN structure learning from data. The first one is *constraint-based* algorithms, which identify conditional independence constraints using statistical tests. The checks test whether nodes A and B are conditionally independent given a conditioning set of nodes, with the result determining whether or not the conditioning set needs to d-separate A and B. The algorithms make adjustments to the DAG based on the results of each pair of nodes tested. Constraint based algorithms can further be divided into local and global discovery algorithms, depending on whether they adjust the DAG immediately after testing a pair of variables for conditional independence or attempt to learn the graph structure as a whole. Popular conditional independence tests used by constraint-based algorithms for discrete BNs are G^2 and χ^2, and for Gaussian BNs, Fisher's z-test is commonly used.

An example of a constraint-based algorithm is the SGS algorithm (Spirtes et al., 1989), which falls into the global discovery category. It starts with a fully connected, undirected graph and works in three phases. First, each pair of nodes is tested for conditional independence, with every possible subset of remaining nodes being used as a conditioning set. If conditional independence occurs for any set, the edge between the pair of nodes is removed. Then, every set of three nodes A, B and C where A and C are connected to B but not to each other, is evaluated. A and C are tested for conditional independence using the same procedure as in the previous phase, and if conditional independence is detected, the triple gets marked as a v-structure $A \rightarrow B \leftarrow C$. Finally, the remaining undirected edges are evaluated, and if either possible orientation either introduces a cycle or a new v-structure, the orientation is known to be forbidden and the opposite orientation is assumed. Otherwise, the edge is left undirected. This process eventually results in a CPDAG. While quite inefficient, this algorithm is simple and is used in some form as the basis of many other constraint-based algorithms (Kitson et al., 2021, Sect. 2.2.1).

Adaptations of the SGS algorithm usually adjust the computationally expensive step of carrying out the conditional independence tests against every possible conditioning set. One such adaptation is the PC algorithm (Spirtes and Glymour, 1991), which checks pairs of nodes with conditioning sets of gradually increasing size and removes the edge between the pair if conditional independence is detected before moving to larger conditioning sets. Furthermore, the PC algorithm takes advantage of the fact that the minimum conditioning set d-separating the pair of nodes A and B must consist of only nodes directly connected to either A or B. At first, this grants no performance benefit as the algorithm starts from a complete graph, but it reduces the number of conditional independence checks more and more as the algorithm progresses and more edges are removed.

3.2.2 Score-based algorithms

Another approach is taken by *score-based* algorithms, which apply more general optimisation techniques. Score-based algorithms start by assigning a network score to each candidate graph using a score function. The scores form a high-dimensional manifold in the space of possible graphs, and an optimisation algorithm is then used to find the highest-scoring graph by searching for the global maximum of the manifold. Score-based search can be either exact or approximate, depending on the choice of the search algorithm and scoring function. Exact algorithms guarantee that the highest scoring DAG is returned, though this might still not be the graph that best matches the underlying distribution that generated the data. This disconnect can be caused by either limitations in the data, namely noise or a low sample size, or by biases introduced by the scoring function used.

The search space is often the space of possible DAGs or CPDAGs denoting equivalence classes. Since every DAG within an equivalence class entails the same conditional independences, there is no reason to prefer one over another based on observational data alone. For this reason, most scoring functions that operate in the DAG space are engineered to give the same score to every DAG in the equivalence class, a property known as score equivalence. Another important feature of scoring functions is decomposability, which refers to the ability of a score to be decomposed into a sum of scores associated with individual nodes in the graph.

Score functions can be broadly divided into Bayesian and information-theoretic scores. Bayesian scores allow the incorporation of prior knowledge and focus on goodness of fit. The Dirichlet prior is often assumed when working with discrete data, leading to the Bayesian Dirichlet (BD) score, which is generally not score-equivalent. BD is also known as the K2-score when its parameters are set to a specific value. The most common Bayesian score is BDeu, an adaptation of BD that is score equivalent (Heckerman et al., 1995). A downside to BDeu is the need to manually specify a value of a regularisation parameter that biases the score towards sparser or denser graphs.

Information-theoretic scores explicitly consider model complexity in addition to goodness of fit in an effort to avoid overfitting. These scoring functions are objective, featuring no prior parameters, which can make them more suitable than Bayesian scores when working with poorly understood target networks (Kitson et al., 2021, Sect. 3.1). Common information-theoretic scores include the Bayesian information criterion and the mutual information test. It is important to note that even without prior parameters, all scoring functions, both in the Bayesian and information-theoretic camps, use some distribution to weigh the candidate graphs, whether user-controllable with a parameter or not. This distribution introduces an inherent bias to the structure learning process and can manifest, for example, as a tendency to favour denser or sparser graphs. This built-in bias is one of the factors mentioned above that can lead even exact scoring algorithms to miss their mark and not correctly identify the correct graph structure.

The central problem for score-based algorithms to solve is that as the number of nodes in the graph increases, the search space quickly grows to an intractable size. This is particularly the case when searching the complete DAG space, which grows superexponetially with respect to the number of nodes in the DAG. To overcome this problem, the search space must be pruned. The pruning rules can either be sound, guaranteeing that the optimal solution is not pruned from the search space, or heuristic, which does not offer this guarantee. An otherwise exact algorithm that employs heuristic pruning of its search space therefore loses the guarantee of optimality and is rendered an approximate algorithm. Possible pruning methods include removing parent sets above a maximum size or parent sets with low scores (Kitson et al., 2021, Chap. 3).

An example of an exact score-based learning algorithm is GOBNILP (Bartlett and Cussens, 2017), which assigns each possible parent set of each variable a score-based on the correlations between these variables in the data matrix. The specific scoring algorithm is not dictated by the algorithm and can be freely selected by the user. The parent sets with the highest combined score, subject to the constraint that they encode a DAG, are then selected using integer linear programming (ILP). This involves the formulation of two constraints as linear functions: each variable must have only one parent set, and the resulting graph must have no cycles. The first condition is straightforward to encode, but enforcing the acyclicity of the graph requires the use of either auxiliary variables and constraints or an exponential number of cluster constraints. The benefit of formulating the search as a generic ILP problem is that it enables the use of pre-existing and highly optimised ILP solvers. It also allows the easy introduction of additional constraints through additional linear encodings, for example, to enforce a connection between specific nodes based on external domain knowledge. A recent adaptation of GOBNILP, called the GOBNILP-DEV (Liao et al., 2018), returns all DAGs that have a score within some threshold of the most optimal value. This provides a method for obtaining a set of plausible graphs.

Another score-based algorithm, called NOTEARS (Zheng et al., 2018), takes a fundamentally different strategy from previous algorithms. The authors note that traditional score-based algorithms, such as GOBNILP, attempt to find the minimum point on the discrete score manifold created by giving a score to each possible DAG with n nodes, which is a combinatorial optimisation problem. NOTEARS reformulates this problem by instead scoring all $n \times n$ matrices W and finding the minimum point on that continuous hyperplane, subject to the constraint $h(W) = 0$, where $h : \mathbb{R}^{d \times d} \to \mathbb{R}$ is a smooth function over real matrices and exactly zero for matrices that are adjacency matrices encoding a valid DAG. While these two formulations are equivalent, the solution employing the continuous function h eliminates the need for custom search algorithms that are tailored to search through the superexponential DAG space and allows leveraging standard numerical algorithms for smooth optimisation, such as L-BFGS.

Through this shift to a continuous constraint, NOTEARS overcomes the difficulty of combinatorial optimisation that discrete score-based methods face, but inherits its problems of nonconvex optimisation because of the nonconvexity of the constraint $h(W) = 0$. As such, it cannot be optimised using exact measures, as discussed in section 2.1, which renders NOTEARS an approximate algorithm that can only be guaranteed to return local minima of the constraint. However, these local minima are often found to be very close to or equal to the global minimum in practical tests, and therefore, NOTEARS is considered to be among the state-of-the-art solutions for structure learning in larger graphs that are beyond the capacity of exact algorithms such as GOBNILP to solve (Li et al., 2020).

Another example of the approximative continuous scoring approach is DAG-GNN (Yu et al., 2019), which uses a variational autoencoder (VAE). VAEs are generative deep learning models that consist of two neural networks called encoder and decoder, which are trained in unison. During training, the encoder processes inputs into an encoding of a lower dimension than the input, which is then used to reconstruct the input by the decoder. The lower dimension of the encoding means that some information from the input must be discarded, which encourages the model to learn the most information-dense encoding of the inputs as possible and avoid overfitting. Through regularisation, the space of possible encodings, also known as latent space, is enforced to be continuous and well-ordered in such a way that an arbitrary point within the latent space can be decoded into a new data point, which enables VAEs to be used for generative modelling (Kingma and Welling, 2013). The DAG-GNN model uses a graph neural network as the encoder and decoder of a VAE.

Instead of searching over possible DAGs or CPDAGs like the score-based algorithms reviewed so far, the search can also take place over ancestral graphs or node-ordering space. Searching through node-ordering space takes advantage of node-ordering, a linear topological sorting of variables such that parents are always higher up than descendants. This approach offers some benefits over the alternative search spaces, such as the fact that the size of the node-ordering space, $2^{\mathcal{O}(n \log n)}$, is much smaller than the size of the DAG space, $2^{\Omega(n^2)}$. An example of an algorithm that explores the node-ordering space is ordering-based search (OBS), a greedy hill-climbing algorithm that moves through the node-ordering space by swapping adjacent nodes in the current ordering (Teyssier and Koller, 2012). At each iteration, all adjacent node swaps are evaluated, with the best-scoring DAG consistent with each ordering compared, and the ordering consistent with the overall best-scoring DAG is adopted for the next iteration. This process is continued until the search reaches a local maximum.

3.2.3 Hybrid algorithms

The third traditional approach is a *hybrid* strategy, which combines constraint-based and score-based approaches with the goal of combining the best characteristics of each. An example of the hybrid strategy is the Constraint-Bayesian (CB) algorithm (Singh and Valtorta, 2013), which is a restrict/maximise algorithm. A restrict/maximise hybrid algorithm works by first using a constraint-based algorithm to limit the DAG space to some of the more likely possibilities, and then a score-based strategy is used to find the optimal solution in the restricted DAG space. The CB algorithm uses the PC algorithm for the restrict phase, using conditioning sets limited to a certain size for that iteration. Then the maximise step uses the K2 scoring algorithm to construct the optimum-scoring DAG from the resulting CPDAG. These two steps are repeated with increasing conditioning set sizes until the score of the resulting DAG no longer increases.

4. Supervised structure learning

In recent years, with advances in deep learning, a fourth technique of BN structure learning has emerged, where neural networks can be trained to learn BN structure in a supervised manner. This technique will be referred to here as the *supervised approach*.

While the supervised approach is a recent development in the field of BN structure learning, its potential benefits are numerous. Due to the intractable superexponential nature of the search space, score-based methods are forced to make various assumptions about the problem, such as the distribution of data or the structure of the underlying DAG, in order to limit the search space to a manageable size. While ingenious and generally well-performing, these assumptions are restrictive and may not always match real-life data. Neural networks, on the other hand, are universal approximators and need no such a priori assumptions.

Another benefit of the supervised approach is the flexibility it offers. A neural network can be trained for any length of time until the desired performance is reached. Regardless of the time spent on training, results can still be obtained in just the time it takes to do a single forward pass through the network. This is beneficial when deploying the model in situations where lots of DAGs must be learned quickly using limited computational resources. In cases where long training times are needed for either very large DAGs or limited compute time, an intermediate state of the model can be used with some performance hit until training is fully finished. It is also easy to store a copy of the optimiser state, which allows training to be continued further if extra performance is desired later down the line.

As a supervised deep learning workload, BN structure learning also has compelling characteristics. Because synthetically generating DAGs is easy in arbitrary quantities, model training does not have to be limited to a fixed, pre-generated data set. Training can instead be performed following the online paradigm as explained in section 2.1, using a dynamic data generator where each DAG is seen by the training process only once. This is more computationally expensive than a fixed training data set, but it also all but eliminates the prospect of overfitting and allows the models to be trained for any length of time.

A downside of using neural networks in a supervised capacity for BN structure learning is that there can be no guarantee of optimality for the resultant DAG, given the random nature of optimisation through backpropagation. Therefore, supervised algorithms are inherently approximative in nature, meaning that graphs small and regular enough to be handled by exact score-based algorithms are not the main application of the supervised approach. Instead, it promises to leverage the superior performance of neural networks for very large and dense graphs with upwards of hundreds of variables, which might be computationally infeasible for traditional algorithms.

4.1 EQ-model

One of the first supervised algorithms to emerge was DAG-EQ (Li et al., 2020), or simply the EQ-model, which uses a sparsely connected neural network for the learning task. The authors note that neither naive FC neural networks nor CNNs are well-suited to structure discovery due to a misalignment between the model characteristics and the problem domain. Much of the information in the input to a structure learning task records the parameters of the distribution used to generate the particular set of data, which is unnecessary for learning the underlying BN structure. FC models have too much capacity and are not capable of easily discarding irrelevant information, which results in the signal of the BN structure getting lost to the noise of the random values needed for generating concrete data samples. CNN models address this capacity problem by focusing on local correlations between individual values in the input data matrix. This constraint makes CNN models very powerful for many applications, such as image-based data, but they are equally ill-suited to BN structure learning since the patterns in the input data are global in nature, and thus a different constraint must be used.

As stated earlier, a structure learning task starts with a data matrix $D = [d_1, ..., d_m]$ consisting of m data instances $d \in \mathbb{R}^n$ that give a value to each of the n variables in the BN. If we define the data instances d as horizontal vectors, then the matrix can also be processed as a series of columns x_i that each define m values for the variable i. This type of input is poorly suited to being used as an input directly, since it encodes mostly values of randomly drawn samples, and only a small portion of the information is useful to the structure learning task. Therefore, an appropriate encoding is needed. Li et al. (2020) choose a featurisation using two square matrices to form the input tensor X. One of these is the $n \times n$ correlation matrix of the input data, with its entries defined as

$$r_{i,j} = \frac{cov(x_i, x_j)}{\sigma_{x_i}\sigma_{x_j}}, \tag{4.1}$$

and the other is a $n \times n$ diagonal matrix where the diagonal entries are defined as

$$s_i = \left(\sigma_{x_i}^2 - \mu\right) \frac{n}{\sum_{j=1}^n (\sigma_{x_j}^2 - \mu)^2}, \tag{4.2}$$

where μ is the mean of the standard deviations. In other words, s_i encodes the variances of the variables normalised to a mean of 0 and a standard deviation of 1. A natural encoding for the output is the adjacency matrix representing the input DAG. Thus, the model after the encoding scheme has been applied is a map $f_\theta : \mathbb{R}^{2\times n\times n} \to [0,1]^{n\times n}$.

The authors' key observation is that the problem is permutation equivariant with regards to variable ordering. Formally, this can be stated as a condition for the model f_θ given any input matrix X and permutation matrix P (Cohen and Welling, 2016):

$$f_\theta(PXP^T) = Pf_\theta(X)P^T.$$

In other words, when variables in the input set of matrices are permuted, the output adjacency matrix should also change according to the same permutation. Enforcing this equivariance introduces a powerful constraint to the neural network that enables significant performance improvements over baseline FC and CNN models. The authors call this new method DAG-EQ, short for DAG structure discovery by equivariant models.

At the heart of the EQ-model performing this enforcement is the exchangeable matrix layer (EQ-layer) (Hartford et al., 2018), which enforces the permutation equivariance of the neural network through its structure. The EQ-layer processes input X to output Y using the following procedure:

$$Y = \sigma\left(w_1 X + w_2 \mathbf{1}\mathbf{1}^T X + w_3 X \mathbf{1}\mathbf{1}^T + w_4 \mathbf{1}\mathbf{1}^T X \mathbf{1}\mathbf{1}^T + b\right),$$

where σ is the activation function, $\mathbf{1} = [1, ..., 1]^T$ and $w_1, ..., w_4, b \in \mathbb{R}$. We can see that regardless of the size of the input X, there are only five learnable parameters in this function, which addresses the capacity problem stated earlier and stands in stark contrast to the basic linear functions used in FC models, where each component of the input has an individually adjustable weight. For a graph with 50 nodes, for instance, this would mean a 500-fold increase in the number of parameters in each neuron.

The five learnable parameters, w_1 through w_4 and b can also be given intuitive explanations: w_1 is a constant weight applied to each element of the input matrix; w_2 and w_3 are weights that are applied to the sums of rows and columns, respectively; w_4 is a weight applied to the sum of the whole input matrix, and b is a bias term. Each of these components is equivariant to permutations of rows and columns, thus rendering the whole operation equivariant to matrix permutation. The layers formed using this procedure can be used like fully connected layers to form a deep network, they can be mixed with other types of layers, and their number and size can be adjusted to fine-tune the performance of the model.

4.1.1 Evaluation and results

To test their results, the authors carried out various tests of their model with various graphs. Their test models consisted of six EQ-layers with a hidden layer size of 300 and were trained using binary cross-entropy loss (equation 2.3) and the Adam optimiser. The leaky ReLU activation function was used with the exception of the last layer, which had sigmoid activation to make sure the outputs are within $[0, 1]$ and can be interpreted as the confidence of an edge being present. Their tests were performed using scale-free (SF) and Erdos-Renyi (ER) graphs. The graphs used in all the training of all models had the same number of nodes n and edges e, or in other words, a density of $\rho = e/n = 1$. The scale parameter k, explained in more detail in section 5.2, was set to $k = 1$. To train each model, a total of 1000 graphs were generated, out of which 80 percent were used for training and the remaining graphs reserved for testing. For each generated DAG, 1000 data points were sampled with either additive Gaussian $\mathcal{N}(0, 1)$ or non-Gaussian noise. Metrics used for measuring the performance of trained models were precision, recall, and structural Hamming distance, which will be explained in more detail in section 5.5. Their results will be briefly summarised here to provide context for the testing that will be conducted as a part of this thesis.

For a baseline comparison against naive supervised deep learning approaches, the authors deployed a FC model made up of six layers with 1024 neurons, as well as a CNN model that used six convolutional layers with 3×3 kernels and a padding size of 1. In addition, out of the common conventional structure learning algorithms, the constraint-based PC algorithm, as well as the score-based Greedy Equivalence Search (GES), Causal Additive Models (CAM) and NOTEARS were used. The score-based deep learning algorithms, the DAG-GNN model, and the reinforcement learning-based RL-BIC2 model were also used. These comparative tests were carried out with graphs of $n \in \{10, 20, 50, 100\}$ variables tested, with the baseline supervised FC and CNN models performing poorly in comparison to the EQ-model across the board, as expected. NOTEARS and DAG-GNN were the best performing models depending on the node count, with the EQ-model performing slightly worse than these two.

The authors also tested the transferability of their model, or the ability of a model trained with one kind of data to transfer the learning results to a different kind of data. Tests of transferability between scale-free and Erdos-Renyi graphs of $n = 10$ showed little decay in performance compared to models trained with the correct type of graphs, suggesting that the EQ-model is not sensitive to changes in input graph topology.

Transferability to larger graphs was also tested with models trained on $n = 100$ graphs and then tested with graphs of $n \in \{100, 200, 300, 400\}$. In this test, the EQ-model was compared to GES, and the EQ-model performed better or equivalently according to all metrics up to $n = 300$, only starting to fall behind GES on some metrics in the case of $n = 400$.

A further set of tests was conducted to investigate the transferability of different noise models. For these tests, an EQ-model trained with graphs of $n = 10$ nodes and using Gaussian noise was tested against graphs using exponential, Gumbel, and Poisson noise. This test shows little to no decay moving from one noise model to another, suggesting the model is also quite robust against changes in data noise patterns.

Finally, tests were conducted to investigate the transferability of the scale parameter k. A model trained with graphs of $k = 1, n = 10$ performed significantly worse when tested with graphs using $k = 2$ and $k = 4$, suggesting that this aspect of the EQ-model would benefit from more investigation and tuning.

The authors also conducted ensemble training by training a model with smaller graphs of $n \in \{10, 15, 20\}$ and testing it with graphs of up to $n = 80$. This method of training provided reasonably good results with larger graphs, potentially indicating that ensemble training might be an avenue to increase the transferability of other graph parameters as well.

4.2 Other supervised models

Petersen et al. (2022) introduce a supervised structure discovery algorithm called SLdisco, which aims to provide a method for CPDAG discovery for observational sciences. This involves the need to work with graphs of any density and with robust small-sample performance, noting specifically that the EQ-model as tested in Li et al. (2020) assumes a priori knowledge of graph density, which is not a realistic assumption in observational sciences. SLdisco is tested using graphs with 0–80 percent of edges missing from fully connected graphs, which for $n = 20$ graphs means a density of $\rho \in [3.61, 18.04]$. The model is a basic CNN with four convolutional layers followed by a fully connected layer, with correlation matrices used as inputs and CPDAG adjacency matrices as outputs. This structure was used merely as a proof of concept, as the objective of the SLdisco work was not to provide a sophisticated neural network architecture, but to focus on data generation and post-processing. Given this omission, it would be an interesting line of future study to combine SLdisco with a neural network built on the EQ-layers for improved performance.

AVICI (Amortised Variational Inference for Causal Discovery) is a supervised structure discovery algorithm introduced by Lorch et al. (2022). Given data D derived from a data generating process $p(D|G)$, the objective is to find the posterior $p(G|D)$ over the dependency structures G by approximating it with a variational distribution $q(G;\theta)$. This is formulated into a supervised learning task by training a model to minimise the expected KL divergence from the posterior to the variational distribution. The specific supervised model used for this optimisation task is a custom permutation equivariant neural network that differs from the EQ-model by using alternating multi-head self-attention and feed-forward residual sublayers. AVICI is shown to outperform competing structure learning approaches in many conditions, especially on out-of-distribution data sets.

5. Methodology

As a part of this thesis, a more detailed exploration of the performance of the EQ-model with various configurations and workloads was done, expanding on the original experiments done by its authors in Li et al. (2020) that were described in detail in section 4.1.1. The purpose of this testing is twofold: to explore the model itself for a potentially more optimal configuration and set of hyperparameters, and to explore the capabilities of the model outside of the limited circumstances employed in the original testing, such as with different graph densities.

While the authors built their original implementation of the model and ran their original tests on the Julia programming language, they also provide a Python implementation using the TensorFlow library on their GitHub-page, which was adapted for the code used in the testing done here. The adapted Python code used to obtain these results can be found on GitHub (https://github.com/ottoros/eq-model). The authors do not exhaustively report the configuration of their models, omitting factors such as optimiser hyperparameters, batch size, and number of training epochs, so identical reproduction of their results is not possible. For this reason and certain methodological differences explained later, direct comparisons with the original author results will be omitted, and testing will largely focus on scenarios not explored in the original article.

While the authors used several conventional structure learning models and basic deep learning models to gauge the relative performance of their model, most of these comparisons were omitted here as redundant. It is clear from the authors' results that the FC and CNN models are not viable approaches to BN structure learning. Furthermore, the performance of all of the conventional structure learning algorithms is well documented in relation to each other, meaning that further comparisons here with such an extensive suite of algorithms would yield little additional insight. Therefore, only the broadly best-performing conventional algorithm, NOTEARS, was used as a benchmark of the results of the testing done here. NOTEARS was implemented using unmodified code from the official GitHub page, with the exception of custom evaluation functions and data generation. The data generation pipeline used in NOTEARS comparisons was identical to the EQ-model tests in order to preserve comparability between the two models. In NOTEARS tests, the L1-regularisation parameter was set to 0, as NOTEARS was observed to benefit from it only in cases of extremely limited sample size, which were not explored in this testing.

5.1 Online training

A fundamental methodological update was applied to the testing conducted here, where all models are trained following the online paradigm from a constantly generated stream of new training samples, as described in section 2.1. Because each training example is only seen by the model once, overfitting is dramatically reduced, and training for indefinite periods becomes viable, which enables the model to utilise a larger portion of its effective capacity for the learning task.

Since online learning does not loop through a pregenerated data set, epochs do not have the same significance as with a traditional training paradigm. However, the training data used in these tests was generated in sets of 4096 graphs, and these sets will be referred to as epochs. For example, if a model was trained for 50 epochs, it will have been exposed to 204 800 unique graphs during training. This would be equivalent to training a model with a fixed training set (offline) with a training set size of 204 800 graphs for a single epoch.

Some models were also trained with a traditional fixed data set that was stored and reused over multiple epochs, which will be referred to as *offline* models. The size of these fixed training sets was set to 800 graphs, as this was also the size of the fixed data sets used by Li et al. (2020). In order to ensure a fair comparison between online and offline models, the offline models were trained for a correspondingly greater number of epochs (4096/800 = 5.12), such that both models went through the same number of iterations. When the duration of training is discussed, only the number of epochs for the online model is mentioned. For instance, when discussing a test run for 500 epochs, the offline model will have actually looped through its fixed sized data set 2560 times.

5.2 Synthetic data-generation

This procedure for synthetic data generation used for training the online models is built on and functionally the same as the Python code used by Zheng et al. (2018) in their testing of NOTEARS. This data generation pipeline is also equivalent to the one used by the authors in their original testing.

Scale-free (SF) graphs are commonly thought to be topologically similar to real-world systems found in fields such as genetics and social networking. Though evidence on this is contradictory (Holme, 2019; Broido and Clauset, 2019), SF graphs are a common choice in the literature for testing structure discovery algorithms, and were therefore also chosen as the primary graph type for these tests. SF graphs are generated using the Barabasi-Albert (BA) algorithm (Li et al., 2020; Barabasi and Albert, 1999), which builds a graph by starting with a small number of nodes and then repeatedly adding a node and drawing from it ρ new edges to nodes. When choosing destination nodes for these new edges, nodes that already have more connections are preferred. This preferential attachment creates a small number of *hub nodes*, which are connected to a large portion of all nodes in the graph. SF graphs require fixing a parameter β, the exponent used in the preferential attachment process, which was set to $\beta = 1$.

Some tests were also conducted on Erdos-Renyi graphs (Erdos and Rényi, 1984) to gauge the impact of changing the graph type. This is another common graph type used to evaluate the performance of structure discovery algorithms and is generated by randomly connecting n nodes with a fixed probability for an edge to appear between each node pair. This results in a dramatically different topology from SF graphs.

Once generated, the random graphs encoded as adjacency matrices $G \in \{0,1\}^{n\times n}$ are assigned random weights by multiplying each edge by a value drawn from $\mathrm{Unif}([-k,-0.5] \cup [0.5,k])$, where k is a parameter that controls the scale of the statistical dependencies in the resulting weight matrix W. The weight matrix can then be sampled for a single sample $X \in \mathbb{R}^n$ using a system of linear equations

$$X = W \times X + z,$$

where $z \in \mathbb{R}^n$ is a noise model. In these tests, the noise model used was Gaussian $z \sim \mathcal{N}(0, I_{n\times n})$. These samples are stacked to form the data matrix $D \in \mathbb{R}^{m\times n}$, from which the input matrix is finally constructed using the featurisation procedure described in section 4.1. All tests were conducted using a sample size of $m = 1000$.

5.3 Model configuration

The EQ-layers can be used like regular fully connected or convolutional layers, by stacking them one after another to form a deep network, as well as side by side to increase the number of neurons per layer. Each layer takes in a tensor of size (b, c_{in}, n, n) and outputs a tensor of size (b, c_{out}, n, n), where b is the batch size that determines the number of training samples fed to the model for one training iteration, c_{in} and c_{out} are the number of neurons in the input and output layers, and n is the number of nodes in the graphs. The total number of weights in one of these layers is $4c_{in}c_{out}$ and the number of biases is c_{out}.

The baseline configuration arrived at by the optimisation process described in section 5.4, and used in all tests unless otherwise mentioned was 6 EQ-layers with an input and output size of 1000, with the exception of the first layer having an input size

Hyperparameter	Min.	Max.	Sampling	Step	Optimal
Number of layers	4	12	discrete	1	6
Number of neurons	200	1000	discrete	100	1000
Leaky ReLU slope	0.0	0.4	linear		0.0602
Learning rate	10^{-5}	$1.5{\cdot}10^{-3}$	log		$4.26 \cdot 10^{-4}$
Optim. 1st moment	0.8	0.98	log		0.827
Optim. 2nd moment	0.95	0.9995	log		0.950

Table 5.1: Description of hyperparameters tuned with random search

of two (the matrices defined in 4.1 and 4.2) and the last layer having an output size of one. This means that the whole model has $8\,000 + 4\,000\,000 \cdot 4 + 4\,000$ weights and $1\,000 \cdot 5 + 1$ biases for a total of $16\,017\,001$ trainable parameters.

5.4 Hyperparameter optimisation

The hyperparameters of the EQ-model were optimised using an implementation of random search provided by the KerasTuner library (O'Malley et al., 2019). The hyperparameters tuned this way were the number and size of layers, the slope of the Leaky ReLU activation function (equation 2.2), the optimiser learning rate, and the 1st and 2nd momentum parameters of the Adam optimiser. The optimised values as well as the specific intervals and distributions used for the random sampling are included in table 5.1.

Some hyperparameters were left out of the tuning process for various reasons. To simplify the process and reduce the number of variables, the layer was tuned with a global variable that affected all layers, as opposed to tuning the size of each layer individually. The initialisation scheme for the network weights was also left at the TensorFlow default setting and not tuned for the same reason. Regularisation was deemed unnecessary and not used because using online training was expected to reduce the likelihood of overfitting. Batch size was not tuned from 32 because higher values would have made the model too large for hardware acceleration, while lower values would have slowed down the training process due to parallelisation going underutilised. Finally, the Adam optimiser was also selected without any tests with other optimisers and effort was put in instead to optimise the most important hyperparameters within the optimiser, the learning rate and the first and second moments of the gradients.

The hyperparameter search was done using $n = 20, k = 1$ graphs with density of $\rho = 3$ to obtain a middle ground between sparse and dense graphs. A total of 315

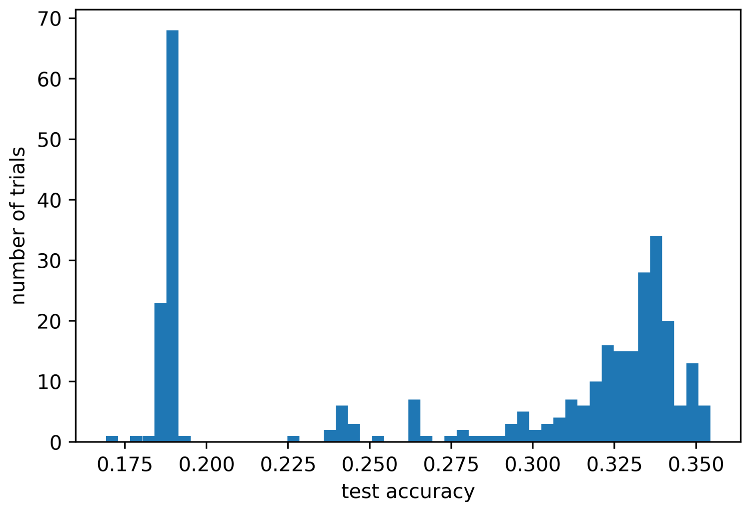

Figure 5.1: The spread of accuracies of the random search trials.

trials lasting 25 epochs were conducted, and the test accuracy distribution of the trials is shown in the histogram in figure 5.1.

5.5 Performance metrics

Some of the commonly used metrics for structure discovery algorithm performance are Structural Hamming Distance (SHD), precision, and recall. Precision P, and recall R are defined as

$$P = \frac{|E' \cap E|}{|E'|} \quad \text{and} \quad R = \frac{|E' \cap E|}{|E|},$$

where E' is the set of predicted edges and E is the set of true edges. In the special case where E' was the empty set, P was set to 0. Intuitively, precision is the proportion of correctly identified edges among all edges identified by the algorithm, while recall is the proportion of correctly identified edges among all the edges that should have been identified (true positives and false negatives). Sometimes the inverse of precision, false detection rate (FDR), and the inverse of recall, false negative rate (FNR), are also useful. All of these quantities are intensive, while SHD, defined as the number of elements that differ in value between the estimated and true CPDAG adjacency matrices (Peters and Bühlmann, 2014), is an extensive quantity.

A further metric designed to account for the shortcomings of SHD in a causal sense is the Structural Intervention Distance (SID) (Peters and Bühlmann, 2014). It computes the path between all pairs of variables and checks if the dependency, or lack thereof, between the variables is respected. However, for reasons of brevity, this study focuses on reporting results using precision, recall, and SHD, omitting the use of SID.

When applying the performance metrics, the comparisons are done between the CPDAGs of the predicted and ground truth graphs, with the CPDAG conversion done using the procedure described in chapter 3.

5.6 Sources of error

For simplicity of implementation, the testing data was not stored across tests and kept the same when comparing the performance of multiple models. When obtaining results for the EQ-model, the metrics were averaged from testing 1024 trial graphs, while metrics reported for NOTEARS comparisons are averaged from 128 trial graphs due to the longer runtime of NOTEARS, and averaging across this many unbiased random draws made random error small enough to not merit separate reporting.

It is important to note that while the reported results are the averages of 1024 tests, each EQ-model result is derived from a single trained model instance. This introduces its own source of error, as each new model is initialised with random weights and biases, and stochastic gradient descent takes an unpredictable path through the loss manifold. It is likely that this explains why some of the reported metrics deviate from the general behaviour of performance trending downwards as the node count in the graphs is increased, as can be seen in the results reported in the next chapter.

It is hard to judge generally how significant an error this variance between trained model instances causes. Some of the test scenarios can potentially be much more susceptible to variance between training runs, as adjusting the model configuration can introduce numerical instability and sensitivity to the initial parameter values, and changing the training data parameters changes the shape of the loss manifold. A sign of this was the failure of many EQ-model instances trained with 50 node graphs to converge, and requiring reruns, while no other graph sizes encountered the same problem.

To properly quantify this effect, numerous identical models would have had to be trained for every studied experimental scenario. While this would have been computationally unfeasible, a single experiment was conducted with a single model and training graph configuration to explore the variance between trained model instances. These results are reported in the histogram in figure 5.2. We can see that there is significant variance in precision and recall, while SHD values are more tightly clustered. From these values, the F1 scores were also calculated, and their spread was also much more tightly clustered, with a minimum of 82.2 and maximum of 83.6. This means that most of the variance in precision and recall between the models comes from the precision/recall tradeoff rather than significant overall performance differences. However, this test is likely not representative of all tested model and training graph configura-

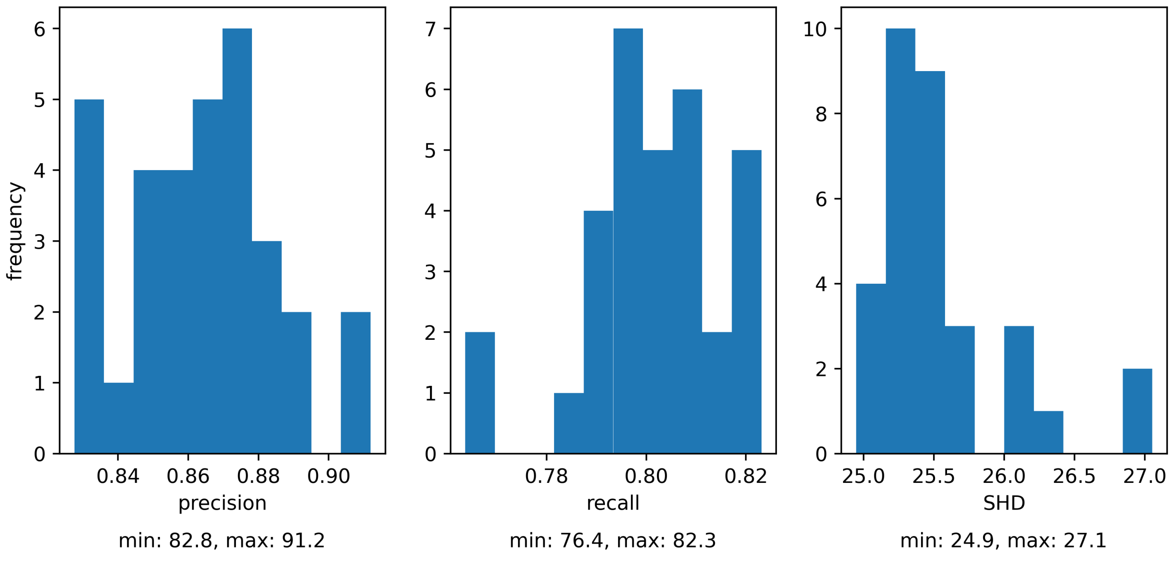

Figure 5.2: The spread of precision, recall, and SHD across 32 model instances trained and tested with 20 node SF graphs of $k = 1, \rho = 4$

.

tions.

5.7 Acyclicity constraint

Since the output of the model is a matrix with floating point values between 0 and 1, a further processing step is required to turn it into a binary adjacency matrix that describes a valid DAG. To do this, an edge corresponding to the largest value above 0.5 in the output matrix is iteratively added to a blank graph, unless it introduces a cycle. This process is continued until all edges with a probability 0.5 or higher have been processed. This is the same procedure used by the authors of the EQ-model. It is important to note that this iterative DAG construction process is only used for external performance evaluation after a model has already been trained. The internal performance metric used by the optimiser during the training of the EQ-model is the simple binary cross-entropy loss (equation 2.3), which does not explicitly enforce acyclicity.

Using this loss function appears to introduce bias into the training process when training models for larger graphs. The number of elements in the target adjacency matrix is n^2 while the number of non-zero elements in the target matrix is ρn, which means that for large graphs, especially with low densities, the model is increasingly incentivised to populate the output with zeros. In the test results presented in the next chapter, recall of discovered edges was observed to fall faster than precision when graph size was increased, which is likely related to this effect of adjacency matrices increasing in sparsity as n is increased.

It is possible that an explicit acyclicity condition baked into the loss criterion would improve performance at the limit of long training times by bridging the disconnect between the internal and various external evaluation metrics. However, the iterative acyclicity calculation is not a differentiable function and, as such, is not usable as a loss function in the optimisation of a neural network. However, a continuous function that takes acyclicity into account while also enabling optimisation through gradient-based methods, such as the smooth constraint h used by NOTEARS, could be potentially used as a component in the loss calculation. Another, simpler approach to level the sparsity bias could be to employ an asymmetric loss function that can help achieve a better balance between precision and recall in highly unbalanced data (Hashemi et al., 2019).

A further effect of the n^2 growth of the adjacency matrix is that models with larger graph sizes become quickly memory-hungry and difficult to fit into VRAM for hardware acceleration to be utilised. For this reason, models trained for graphs of $n = 100$ used a modified configuration with a reduced batch size of 8 and a reduced layer size of 500. To highlight this difference, these models are highlighted with an asterisk in the tables included in the next chapter.

For potential tests with larger graphs, the model would have had to be shrunk even further, which could negatively impact performance. However, due to larger adjacency matrices being sparser, an interesting potential alternative to shrinking the model would be using libraries and datatypes that optimise the memory footprint of sparse matrices. This could allow the structure discovery of vastly larger graphs, especially if L1 regularisation was additionally used to incentivise sparsity of weights. However, exploring the special characteristics of large graph discovery as well as the effectiveness of alternate loss functions was judged to be outside the scope of this work and left for future study.

6. Experimental Results

6.1 Offline and NOTEARS comparison

First, the potential performance improvement of online training over offline training was studied using graphs of various sizes and $k = 1, \rho = 1$. The size of the fixed training set for the offline model, 800 graphs, was the same as used in Li et al. (2020) and the models were trained for 25 epochs (or equivalent). Results of these tests, as well as NOTEARS results for further comparison, are included in table 6.1a. From these results, it appears that the case of low-density SF graphs is too simple to bring out any significant differences between the compared methods.

To increase the complexity of the discovery task, the same experiment was conducted with higher density graphs of $k = 1, \rho = 4$, with the results being displayed in table 6.1b. Overall, higher density graphs appear to be more challenging for the EQ-model. We can see the online models displaying a slight performance improvement over the offline models but NOTEARS is significantly above both implementations of the EQ-model. Overall, denser graphs can be seen to be a more challenging task for all studied methods.

6.2 Transferability

In the previous tests, new models were trained for each set of different graph parameters. However, a desirable property of a structure learning model is the ability to use a single model for graphs with different configurations than the ones it was trained with, such as graphs of varying densities and scale parameter values. This property is called transferability, and to gauge the transferability of the basic models trained with sparse graphs of $\rho = 1$, they were also tested with $\rho \in [2, 4]$ graphs with otherwise the same configuration. These results are reported in table 6.2a.

Transferability tests were also done on the the scale parameter k, which is another important BN parameter. The baseline value of k used in both these and the original author tests was 1, but the models trained with the baseline $k = 1$ graphs were also

	EQ (Offline)			EQ (Online)			NOTEARS		
Nodes	prec.	recall	SHD	prec.	recall	SHD	prec.	recall	SHD
10	99.9	99.9	0.02	99.8	99.6	0.04	99.9	99.9	0.01
20	99.7	99.4	0.20	99.7	99.8	0.06	99.9	99.7	0.04
50	96.3	99.4	2.38	96.9	99.2	1.83	99.7	98.6	0.81
100*	98.3	97.1	5.34	96.7	98.6	4.45	97.4	99.6	3.26

(a) Comparison of models trained and tested with $\rho = 1$ graphs

	EQ (Offline)			EQ (Online)			NOTEARS		
Nodes	prec.	recall	SHD	prec.	recall	SHD	prec.	recall	SHD
10	85.7	87.6	8.41	94.5	86.3	6.65	94.8	94.9	3.10
20	83.9	77.4	29.5	86.7	79.9	25.4	91.9	96.1	9.09
50	76.6	67.7	119	68.7	75.4	105	91.3	97.5	23.3
100*	49.8	78.9	257	55.2	76.4	250	88.9	97.6	59.9

(b) Comparison of models trained and tested with $\rho = 4$ graphs

Table 6.1: Comparison between offline models trained using a fixed data set of 800 graphs, online models, and NOTEARS.

tested with $k \in [2, 4]$. These results are displayed in table 6.2b. We can see that in both cases, performance degrades quickly as the parameters deviate from those used to generate the training graphs. In particular, these sparse models handle higher densities much worse than models trained with dense graphs from table 6.1b. The precision of discovered edges falls below 50 percent for all densities and graph sizes, which indicates that the majority of discovered edges are incorrect.

Nodes	ρ_{test}	prec.	recall	SHD	ρ_{test}	prec.	recall	SHD
10	2	51.7	88.9	10.1	4	24.3	73.5	26.5
20	2	45.9	88.3	23.9	4	16.8	74.6	64.3
50	2	40.3	88.1	68.1	4	14.5	74.3	178
100*	2	39.4	88.4	139	4	11.0	73.5	374

(a) Transferability of density

Nodes	k_{test}	prec.	recall	SHD	k_{test}	prec.	recall	SHD
10	2	94.4	94.2	1.22	4	81.0	82.1	3.88
20	2	93.1	93.8	2.90	4	73.6	80.9	9.34
50	2	86.4	86.1	15.5	4	59.9	66.6	39.9
100*	2	79.6	83.8	40.7	4	53.6	66.6	84.6

(b) Transferability of scale parameter

Table 6.2: Performance of models trained with $\rho = 1, k = 1$ tested on graphs with increased density and scale parameter

6.3 Ensemble training

To improve transferability, *ensemble training* was attempted as a potential solution. This involves training models with graphs of differing parameters (an ensemble), where the parameters in question are drawn randomly from pre-determined distributions. First, to improve density transferability, a set of models for $n \in \{10, 20, 50\}$ were trained with an ensemble of densities, while the scale parameter was kept constant at $k = 1$. The density of each graph was an integer drawn uniformly from the interval $[1, 6]$, with a discrete distribution being used because the BA algorithm used to generate SF graphs only allows integer densities. These models were then tested with graphs of different densities, with the results reported in table 6.3a. We can see that ensemble training improves performance dramatically in comparison to the models trained only on $\rho = 1$

graphs, with reasonable performance even in the case of very dense $\rho = 8$ graphs. Specifically, in the case of $\rho = 4$, the performance of the ensemble model is on par with the model from table 6.1a trained exclusively with those graphs, which indicates that for certain graph parameters, including at least density, ensemble training does not degrade performance in comparison to more narrowly trained models.

To investigate whether the discontinuities in the training data caused by integer densities limited model performance, further models were trained by graphs from a custom variant of the BA algorithm while still being tested using the standard data generation pipeline. In this custom version, the number of edges ρ drawn from each new node was randomised for each iteration, allowing the whole graph to obtain non-integer density values and thus enabling the generation of an ensemble of graphs with a smooth distribution of densities. However, none of these models were able to achieve a performance improvement over the models trained with integer density graphs, so the results will be omitted. This lack of improvement is likely because any potential gains from smoothing the granular sample space were undone by the introduction of a systematic difference between the training graphs and the test graphs due to the modifications to the graph generation algorithm.

Ensemble training was also done by varying the scale parameter k for graphs of size $n \in \{10, 20, 50\}$. In these tests, the value was sampled from $k \sim \text{Unif}(1, 4)$, while the density was kept at $\rho = 1$. The results can be seen in table 6.3b. As with density, we observe a significant performance uplift from ensemble training in comparison to models trained with a fixed value of $k = 1$.

While the non-ensemble models were trained for 25 epochs, the ensemble models were trained for 50 epochs in anticipation of slower convergence, since training the models using an ensemble of graphs of different configurations results in a more complex loss manifold. To find out if this increase was sufficient and to simultaneously illustrate the ability of online training to increase training time without having to worry about overfitting, both online and offline versions of a $n = 50$ density ensemble model were further trained for 450 epochs for a total of 500 epochs. The improvement obtained by the extra training epochs is listed in table 6.4. It is clear that this extended training duration is a compelling use-case for the online training paradigm. We can see that after 50 epochs, the difference between the online and offline models is fairly modest, but increasing the training duration to 500 epochs greatly improves the performance of the online model while actually degrading the performance of the offline model, as the limited training set size begins to cause overfitting.

During the training process of every model, the training loss after each epoch was also logged. These loss histories for the models shown in table 6.4 are shown in figure 6.1. We can see how, after roughly 150 epochs of training, the offline model has

Nodes	ρ_{test}	prec.	recall	SHD	ρ_{test}	prec.	recall	SHD
10	1	99.6	99.8	0.05	2	96.4	98.9	2.61
20	1	99.5	99.5	0.19	2	88.9	86.2	10.3
50	1	96.4	98.9	2.61	2	72.7	87.7	39.4
10	4	93.4	86.2	7.28	8	75.0	97.3	19.1
20	4	88.4	79.9	25.2	8	86.1	85.5	36.2
50	4	67.4	77.0	105	8	58.7	81.5	202

(a) Models trained with an ensemble of densities and $k = 1$

Nodes	k_{test}	prec.	recall	SHD	k_{test}	prec.	recall	SHD
10	1	99.9	99.9	0.01	2	99.4	99.3	0.12
20	1	99.3	99.7	0.18	2	98.5	98.8	0.57
50	1	97.3	98.8	2.13	2	93.7	97.9	4.62
10	4	97.9	97.9	0.42	8	91.4	93.5	1.58
20	4	93.6	93.8	2.85	8	78.0	81.2	8.81
50	4	84.1	93.5	12.1	8	72.6	81.0	25.1

(b) Models trained with an ensemble of scale parameter values and $\rho = 1$

Table 6.3: Performance of ensemble models tested on graphs of various densities and scale parameter values

		EQ (Offline)			EQ (Online)		
ρ_{test}	epoch	prec.	recall	shd	prec.	recall	shd
1	50	94.7	97.2	4.48	96.2	98.8	2.73
2	50	71.1	82.0	47.3	72.7	87.8	39.2
4	50	66.8	70.2	124	67.6	77.3	104
8	50	60.6	74.9	221	58.5	81.4	203
1	500	96.4	94.7	5.12	99.5	99.7	0.40
2	500	77.3	70.6	58.3	80.2	92.1	27.8
4	500	68.7	65.2	135	71.2	80.9	90.5
8	500	53.1	74.1	242	63.9	83.5	180

Table 6.4: Performance comparison of online and offline versions of the $n = 50$ density ensemble models trained for 50 and 500 epochs.

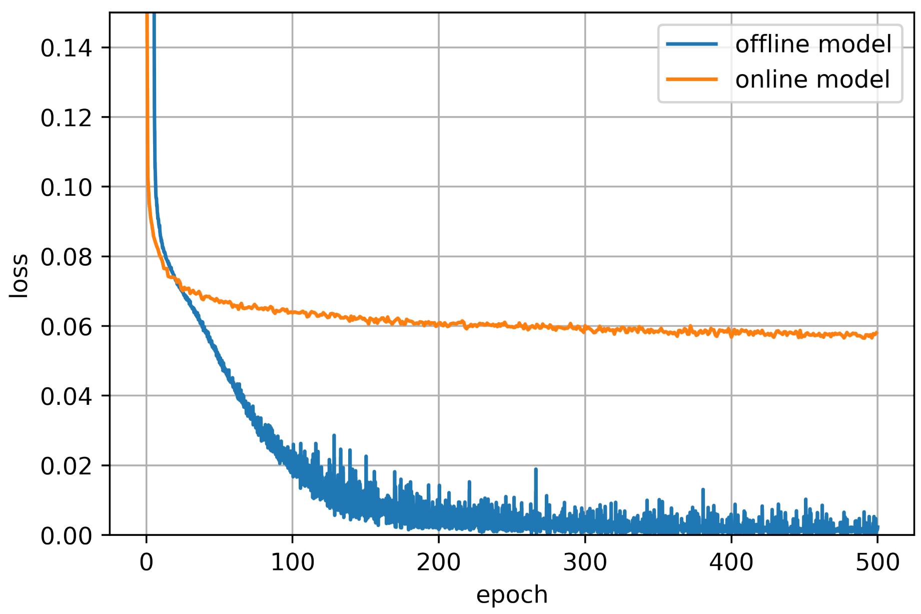

Figure 6.1: Loss during training of an online and an offline model trained with an extended number of epochs with 50 node graphs drawn from the density ensemble.

learned the training data set almost completely as training loss approaches zero and the model starts to overfit. Given how excessively long the model has been trained with a small training data set, demonstrated by the shape of this graph, the results for the 500-epoch offline model are still relatively good, which is a sign of the robustness of the EQ model's architecture to overfitting. At the same time, this graph illustrates another benefit of online training: the lack of need to carefully monitor the training process for overfitting and attempt to time early stopping correctly.

While the main focus of this analysis is SF graphs, the impact of graph type was also tested using ER graphs, with the results being displayed in table 6.5. Transferability from SF to ER graphs appears to be very poor, likely due to the dramatically different topologies of the different graph types. Like with the other studied cases of k and ρ, a graph type ensemble model was also trained to potentially increase transferability. In this ensemble, each batch of training graphs generated had an equal chance of being SF and ER graphs, and the results from this ensemble model are also included in table 6.5. We can see that the ensemble training resolves the issue of overfitting to a particular graph type. The ensemble model performs almost as well on ER graphs as the model trained entirely on ER graphs and shows no performance degradation on SF graphs in comparison to the model trained entirely on SF graphs.

This remarkable improvement in transferability gained through ensemble training across the various tested metrics, especially graph type, suggests ensemble training to

g_{train}	g_{test}	Nodes	prec.	recall	SHD	Nodes	prec.	recall	SHD
SF	SF	10	100	99.9	0.00	50	99.4	99.4	0.64
ER	ER	10	96.3	96.5	0.95	50	92.1	96.6	6.97
SF	ER	10	56.8	68.0	9.37	50	44.6	68.0	48.4
ER	SF	10	99.5	99.1	0.15	50	90.5	97.8	6.48
Ens	SF	10	99.9	99.9	0.00	50	98.9	99.5	0.87
Ens	ER	10	94.2	96.3	1.28	50	90.4	94.8	9.17

Table 6.5: Transferability of graph type g on ensemble and non-ensemble models trained on $\rho = 1, k = 1$ graphs for 50 epochs

be a very useful general method of obtaining better performance. One potentially interesting line of further study is composite ensembles, where models are trained with graphs that have multiple parameters randomised simultaneously, such as graph type, density, noise model, and graph size. It also seems likely that as the training ensemble becomes more complex, online training will continue to offer larger and larger benefits as the increased amount of information in the generating distributions struggles to fit into a training data set with a fixed size.

6.4 Confidence reporting

The raw output of the model is a matrix populated by floating point values in $[0, 1]$, which is then used to construct the final adjacency matrix guaranteed to correspond to a valid DAG using the iterative procedure described in section 5.7. The values in this raw output can be interpreted as a probability of an edge being reported correctly, with a value close to 0.5 meaning low confidence and a value close to 0 or 1 meaning high confidence in an edge either being present or absent. Having a reliable confidence estimate for the predicted DAGs would be useful in many situations, for instance when graphs are small enough that less certain edge cases can be verified with a slower exact structure discovery algorithm, while the bulk of predictions can still be made with the faster method of forward-propagation through the EQ-model. Another use for even a crude confidence metric would be scenarios involving poorly understood real-life data, where a priori assumptions about data-generating distributions are either mistaken or unavailable, and thus a confidence metric would be the only way to detect an out-of-distribution situation.

To test the validity and utility of this interpretation, the binary cross-entropy loss between the raw output of the EQ-model and the adjacency matrices produced by the

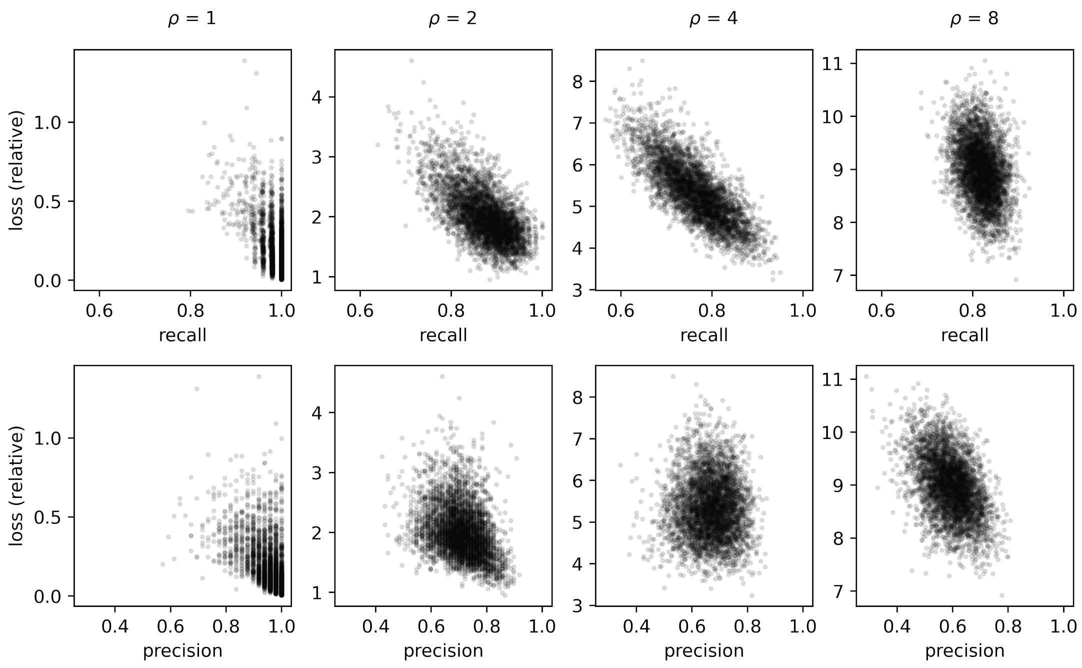

Figure 6.2: The relationship between the proposed confidence measure, binary cross-entropy loss, and objective performance metrics for graphs of various densities. Test conducted on the $n = 50$ density ensemble model trained for 50 epochs.

iterative post-processing was computed for various types of graphs. The scatterplot in figure 6.2 displays this loss as a function of the precision and recall values of the postprocessed adjacency matrices. This allows us to gauge the reliability of the loss as a confidence metric. From the recall scatterplots, we can see that for densities $\rho \in \{2, 4\}$ the trials form a fairly sharply defined diagonal cluster, with higher losses being correlated with a lower recall. This proves that the loss can be a good confidence measure, at least with certain graph types and parameters. Furthermore, it would appear that the reason we cannot see a correlation in the case of $\rho = 8$ is the narrower spread of recall in the tested graphs, possibly indicating that the confidence metric would also work with higher densities if a more significant spread of recall was present.

These results do not necessarily justify the need for a confidence measure, as there is no significant group of poorly performing outliers that would particularly benefit from being identified and treated separately. However, it is possible that different graph types, noise models, and non-Gaussian or non-linear data generation might result in a longer tail of outliers that would benefit from special treatment. Exploring the confidence metric with more complicated data and real-life examples is left for further study.

A further issue with the loss as a confidence metric is that even in cases where the losses and recall values are correlated, the loss values are not comparable between tests done on graphs of different densities. For instance, figure 6.2 shows that a loss value of 3.5 in the case of $\rho = 2$ indicates exceptionally poor performance, while in the case of $\rho = 4$ it indicates exceptionally good performance. This means that the confidence measure could likely not be very effectively utilised until a data set for contextualising the loss values similar to figure 6.2 has been compiled by analysing a large number of graphs generated from the same distribution, which in general is not analytically understood in real-life applications.

There is also a potential for refinement of the confidence metric. The sources of differences between the raw floating-point matrix output of the neural network and the post-processed adjacency matrix can be divided into two categories: the first case is the simple rounding difference from rounding the floats to the closest integer, thus confirming either the presence or absence of an edge in the final adjacency matrix and the second case is the difference from turning a value greater than 0.5 into a 0, because adding an edge in that position would have introduced a cycle. Whenever this second case is triggered by the interative post-processing, it indicates that the nebulous heuristics that the EQ-model built for itself in the training phase have failed, which could be a stronger indicator that the final output poorly matches the ground truth. Therefore, it is possible that weighting this second case more in the confidence metric, or even building a confidence metric that only takes into account the situations where this acyclicity check has to step in, would be a better predictor of the accuracy of the predicted graph. Exploring this is also left for further study.

6.5 Visualisation

In order to qualitatively gauge the outputs of the EQ-model, select test graphs were plotted in figure 6.3 as heatmaps of the adjacency matrices. Each graph is the median loss graph out of a sample of 32 graphs, and for each graph, four different metrics are shown. In all graphs, the hub nodes that are characteristic of SF graphs are clearly visible as light vertical lines in the true edges displayed on the top row of heatmaps. We can see that these hub edges are also responsible for the majority of the incorrectly predicted edges, as can be seen from the bottom row.

6.6 Runtime of algorithms

For some results, table 6.6 provides the training time and time to obtain results from a pre-trained model. All the tests were run on a desktop computer with a Ryzen 5 5600X processor and an Nvidia RTX 3080 graphics card, with CUDA acceleration enabled in TensorFlow for EQ-models.

It must be noted that the runtime of machine learning algorithms depends on numerous factors beyond simply the hardware configuration, such as the choice of drivers and other software, firmware versions, system load from other applications, and the specific way the algorithms have been implemented in code. Because of these factors, even on a system with similar hardware, runtimes could significantly differ from the values presented.

Nevertheless, these reported runtimes give a reasonable picture of how various algorithms and model configurations stack up against each other in a relative sense. Particularly of note is the dramatic speed difference between the EQ-models and NOTEARS for larger graphs. In the time of just six NOTEARS discovery operations, we can train a reasonably performant EQ-model that, once training is complete, performs structure discovery almost five orders of magnitude faster. While NOTEARS is certainly not the fastest conventional structure discovery algorithm, nor is this Python implementation likely the most optimal way to run the algorithm, this stark contrast still illustrates quite dramatically the arguably biggest strength of the supervised approach.

We can also see that there is no difference between the discovery time between online and offline models, as is to be expected from neural networks with identical configurations and likely only slight differences in parameter values. The differences

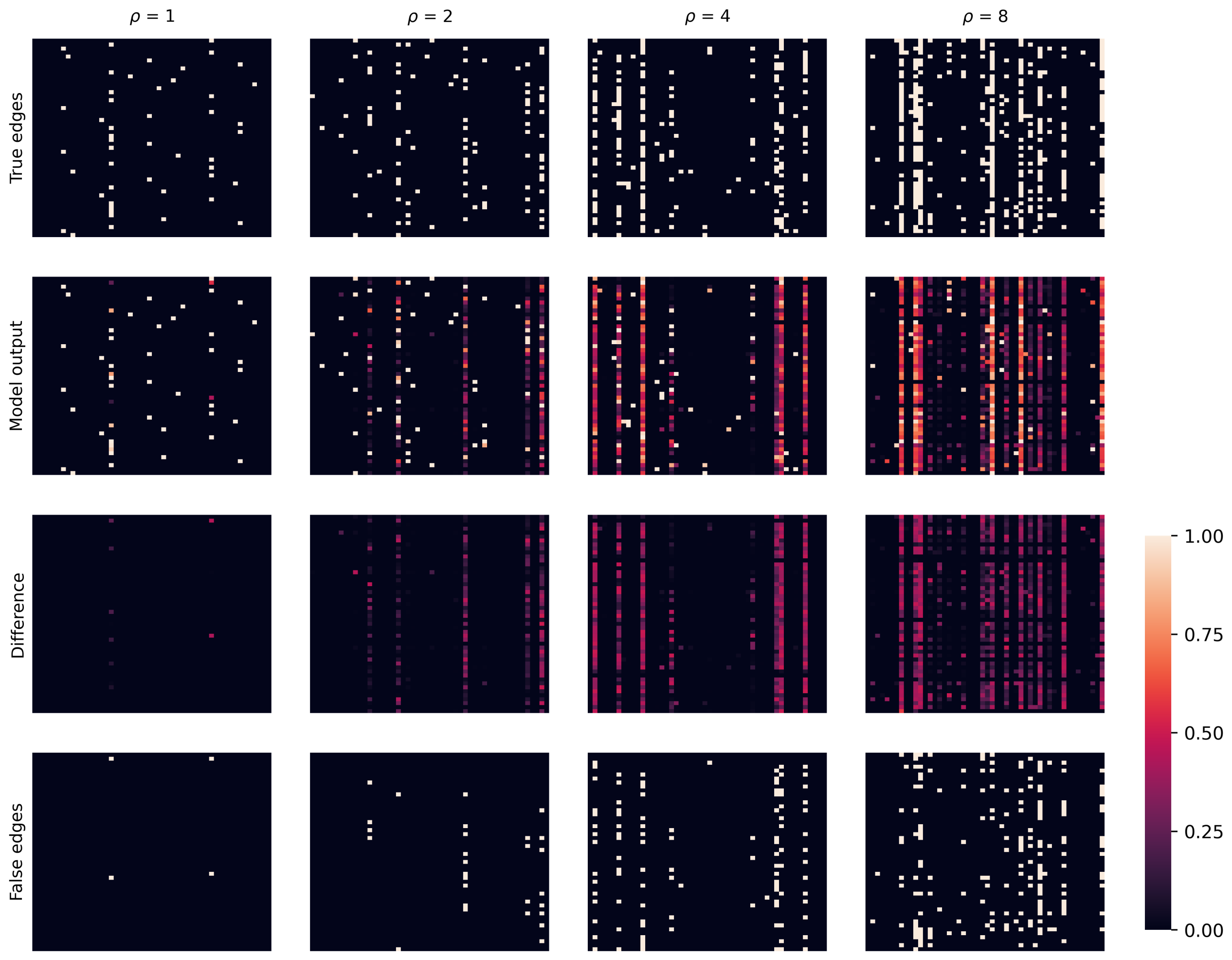

Figure 6.3: Four individual graphs of different densities tested on the 50 epoch, $n = 50$ density ensemble model, and visualised as adjacency matrices. The top row shows the true edges of the test graph, the second row shows the raw float-valued output of the model, the third row shows the difference between the raw model output and the DAG derived from the model output, and the bottom row shows the difference between the model output DAG and the true edges.

	EQ (Offline)		EQ (Online)		NOTEARS
Nodes	training	discovery	training	discovery	discovery
10	436	0.002	230	0.002	1.3
20	646	0.004	539	0.004	8.81
50	2265	0.021	2335	0.021	80.8
100*	3105	0.068	3852	0.068	538

Table 6.6: Comparison of model training and discovery times in seconds. Discovery times are the time to discover one graph structure, averaged from a set of 1024 tests for EQ-models and 128 tests for NOTEARS. All models were trained and tested with $k = 1, \rho = 1$ graphs. Offline models were trained with a fixed data set of 800 graphs for 128 epochs, and online models were trained for an equivalent duration of 25 epochs with 4096 graph dynamic data sets. Their results were reported in table 6.1a.

in training times can be explained by implementation differences. The offline models had to be trained for 5.12 times more epochs due to the smaller fixed training set size, which increased overhead for smaller node counts, while in the case of larger node counts, the online models are slowed down by the need to generate new training graphs continuously.

7. Conclusion

This thesis outlined the basics of deep learning and Bayesian networks and provided an overview of the various methods conventionally used for Bayesian network structure learning. Then, the prospect of supervised structure learning using neural networks was introduced, and one such method, the EQ-model, was explored in more detail and subjected to various empirical tests. Through a combination of more optimised hyperparameters and shifting to online training, the performance of the model was increased. Furthermore, the effectiveness of the model on a broader range of workloads, such as denser graphs, was demonstrated.

Aspects of the EQ-model left for further study include non-Gaussian noise models, non-linear data generation, further graph types like small-world graphs, and small sample sizes in the input data. Another case left unexplored was that of discrete or categorical data, which can be particularly challenging due to the increased difficulties introduced by a sparse parameter space (Gu et al., 2018). Also, despite a comprehensive hyperparameter search being done, certain hyperparameters were left out of the search, such as individual variations in the size of each layer and potentially mixing fully connected or convolutional layers into the network. Furthermore, certain potentially interesting lines for closer study were identified, including the use of composite ensembles for training.

In general, supervised methods built on neural networks promise major advantages over conventional algorithms. If data-generating distributions are known, models can be trained to learn the specific biases preset in the distributions and thus tailored to each use case for optimal performance. Additionally, these models offer scalability beyond the reach of most conventional algorithms and large improvements in discovery time once the one-time training process has been completed. It seems likely that, despite its recent introduction, supervised structure learning will continue to gain popularity over conventional algorithms in the future.

The paradigm of online training that was used for evaluating the EQ-model is also an interesting avenue for further study, not just with regards to causal discovery, but deep learning in general. There appears to be little literature on online training deep learning models, even as the popularity of deep learning has dramatically increased in the past decade. In particular, exactly how reduction of overfitting as a concern impacts hyperparameter optimisation and model configuration is an interesting question for all workloads, where training data can be easily synthetically generated, such as combinatorial optimisation problems, or where unbiased and reliable training data is otherwise available in massive quantities.

The weakness of this approach, of course, as also seen in certain results from testing the EQ-model, is that it does not improve transferability to out-of-distribution data sets as the model is still prone to overfitting to the data-generating distribution. At the same time, this weakness was shown to be partially solvable using ensemble training, which generalises the entire distribution and acts as an effective regularisation method. In future use, when branching out to new graph types and wider ensembles, the model would likely benefit from hyperparameters specifically optimised for that workload. Introducing traditional regularisation methods such as weight decay or batch normalisation, which were left out of consideration in this study, might also further help with transferability to out-of-distribution data.

Bibliography

A.-L. Barabasi and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, Oct 1999.

M. Bartlett and J. Cussens. Integer Linear Programming for the Bayesian network structure learning problem. *Artificial Intelligence*, 244:258–271, 2017.

J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, Feb 2012.

A. D. Broido and A. Clauset. Scale-free networks are rare. *Nature Communications*, 10(1), Mar 2019.

D. M. Chickering, D. Fisher, and H.-J. Lenz. Learning Bayesian Networks is NP-Complete. In *Learning from Data: Artificial Intelligence and Statistics V*, pages 121–130. Springer New York, New York, NY, 1996.

D. M. Chickering, C. Meek, and D. Heckerman. Large-Sample Learning of Bayesian Networks is NP-Hard. *arXiv: 1212.2468*, 2012.

T. S. Cohen and M. Welling. Group Equivariant Convolutional Networks. *arXiv: 1602.07576*, 2016.

H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning Combinatorial Optimization Algorithms over Graphs. *arXiv: 1704.01665*, 2018.

A. Darwiche. *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press, 2009.

P. L. Erdos and A. Rényi. On the evolution of random graphs. *Transactions of the American Mathematical Society*, 286:257–257, 1984.

J. a. Gama, I. Žliobaitundefined, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A Survey on Concept Drift Adaptation. *ACM Computer Survey*, 46(4), Mar 2014.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

J. Gu, F. Fu, and Q. Zhou. Penalized estimation of directed acyclic graphs from discrete data. *Statistics and Computing*, 29(1):161–176, Feb 2018.

J. Hartford, D. R. Graham, K. Leyton-Brown, and S. Ravanbakhsh. Deep Models of Interactions Across Sets. *arXiv: 1803.02879*, 2018.

S. R. Hashemi, S. S. M. Salehi, D. Erdogmus, S. P. Prabhu, S. K. Warfield, and A. Gholipour. Asymmetric Loss Functions and Deep Densely-Connected Networks for Highly-Imbalanced Medical Image Segmentation: Application to Multiple Sclerosis Lesion Detection. *IEEE Access*, 7:1721–1735, 2019.

D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian Networks: The Combination of Knowledge and Statistical Data. *Machine Learning*, 20(3):197–243, Sep 1995.

P. Holme. Rare and everywhere: Perspectives on scale-free networks. *Nature Communications*, 10(1):1016, Mar. 2019.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv: 1412.6980*, 2014.

D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *arXiv: 1312.6114*, 2013.

N. K. Kitson, A. C. Constantinou, Z. Guo, Y. Liu, and K. Chobtham. A survey of Bayesian Network structure learning. *arXiv: 2109.11415*, 2021.

T. Koski and J. Noble. *Bayesian Networks: An Introduction*. Wiley Series in Probability and Statistics. Wiley, 2011.

J. Lampinen and A. Vehtari. Bayesian approach for neural networks—review and case studies. *Neural Networks*, 14(3):257–274, 2001.

H. Li, Q. Xiao, and J. Tian. Supervised Whole DAG Causal Discovery. *arXiv: 2006.04697*, 2020.

Z. A. Liao, C. Sharma, J. Cussens, and P. van Beek. Finding All Bayesian Network Structures within a Factor of Optimal. *arXiv: 1811.05039*, 2018.

L. Lorch, S. Sussex, J. Rothfuss, A. Krause, and B. Schölkopf. Amortized Inference for Causal Structure Learning. *arXiv: 2205.12934*, 2022.

S. Mukhopadhyay. *Advanced Data Analytics Using Python: With Machine Learning, Deep Learning and NLP Examples.* Apress, 2018.

V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, et al. KerasTuner, 2019. `https://github.com/keras-team/keras-tuner`.

J. Patterson and A. Gibson. *Deep Learning: A Practitioner's Approach.* O'Reilly, Beijing, 2017.

J. Pearl and T. Verma. The Logic of Representing Dependencies by Directed Graphs. In *AAAI Conference on Artificial Intelligence*, 1987.

J. Peters and P. Bühlmann. Structural Intervention Distance (SID) for Evaluating Causal Graphs. *arXiv: 1306.1043*, 2014.

A. H. Petersen, J. Ramsey, C. T. Ekstrøm, and P. Spirtes. Causal discovery for observational sciences using supervised machine learning. *arXiv: 2202.12813*, 2022.

T. Qin, L. C. Tsoi, K. J. Sims, X. Lu, and W. J. Zheng. Signaling network prediction by the Ontology Fingerprint enhanced Bayesian network. *BMC Systems Biology*, 6 (3):S3, Dec 2012.

S. Ruder. An overview of gradient descent optimization algorithms. *arXiv: 1609.04747*, 2016.

M. Singh and M. Valtorta. An Algorithm for the Construction of Bayesian Network Structures from Data. *arXiv: 1303.1485*, 2013.

P. Spirtes and C. Glymour. An Algorithm for Fast Recovery of Sparse Causal Graphs. *Social Science Computer Review*, 9(1):62–72, 1991.

P. Spirtes, C. Glymour, and R. Scheines. Causality from probability. *Conference Proceedings: Advanced Computing for the Social Sciences, Williamsburgh*, 1989.

C. Su, A. Andrew, M. R. Karagas, and M. E. Borsuk. Using Bayesian networks to discover relations between genes, environment, and disease. *BioData Mining*, 6(1): 6, Mar 2013.

S. Terzi, S. Torresan, S. Schneiderbauer, A. Critto, M. Zebisch, and A. Marcomini. Multi-risk assessment in mountain regions: A review of modelling approaches for climate change adaptation. *Journal of Environmental Management*, 232:759–771, Feb 2019.

M. Teyssier and D. Koller. Ordering-Based Search: A Simple and Effective Algorithm for Learning Bayesian Networks. *arXiv: 1207.1429*, 2012.

Z.-J. Wang and L.-T. Zhao. The impact of the global stock and energy market on EU ETS: A structural equation modelling approach. *Journal of Cleaner Production*, 289, Mar 2021.

Y. Yu, J. Chen, T. Gao, and M. Yu. DAG-GNN: DAG Structure Learning with Graph Neural Networks. *arXiv: 1904.10098*, 2019.

L. Zhao, X. Wang, and Y. Qian. Analysis of factors that influence hazardous material transportation accidents based on Bayesian networks: A case study in China. *Safety Science*, 50(4, SI):1049–1055, Apr 2012.

X. Zheng, B. Aragam, P. Ravikumar, and E. P. Xing. DAGs with NO TEARS: Continuous Optimization for Structure Learning. *arXiv: 1803.01422*, 2018.