On the Relative Expressiveness of Bayesian and Neural Networks

Arthur Choi AYCHOI@CS.UCLA.EDU

Adnan Darwiche DARWICHE@CS.UCLA.EDU

Computer Science Department, University of California, Los Angeles

Abstract

A neural network computes a function. A central property of neural networks is that they are "universal approximators:" for a given continuous function, there exists a neural network that can approximate it arbitrarily well, given enough neurons (and some additional assumptions). In contrast, a Bayesian network is a model, but each of its queries can be viewed as computing a function. In this paper, we identify some key distinctions between the functions computed by neural networks and those by Bayesian network queries, showing that the former are more expressive than the latter. Moreover, we propose a simple augmentation to Bayesian networks (a testing operator), which enables their queries to become "universal approximators" as well.

Keywords: Bayesian networks; neural networks; universal approximation.

1. Introduction

The field of artificial intelligence (AI) has seen two major milestones throughout its history. Shortly after the field was born in the 1950s, the focus turned to *symbolic, model-based* approaches, which were premised on the need to represent and reason with domain knowledge, and exemplified by the use of logic to represent such knowledge (McCarthy, 1959). In the 1980s, the focus turned to *probabilistic, model-based* approaches, as exemplified by Bayesian networks and probabilistic graphical models more generally (first major milestone) (Pearl, 1988). Starting in the 1990s, and as data became abundant, these probabilistic models provided the foundation for much of the research in machine learning, where models where learned either generatively or discriminatively from data. Recently, the field shifted its focus to *numeric, function-based* approaches, as exemplified by neural networks, which are trained discriminatively using labeled data (deep learning, second major milestone) (Goodfellow et al., 2016; Hinton et al., 2006; Rosenblatt, 1958; McCulloch and Pitts, 1943). Perhaps the biggest surprise with the second milestone is the extent to which certain tasks, associated with perception or limited forms of cognition, can be approximated using functions (i.e., neural networks) learned purely from labeled data, without the need for modeling (Darwiche, 2018).

While this evolution of the field has increased our abilities, the emerging techniques have been pursued by somewhat independent research communities. The price has been a lack of enough integration/fusion of the various methods, which are limiting us in realizing their collective power. *Logic* provides a rich framework for representing knowledge in the form of domain constraints and comes with profound reasoning mechanisms. *Probabilistic graphical models* excel at capturing uncertainty, causal knowledge, and independence information. Both of these frameworks provide a foundation for capturing domain knowledge of various types. *Neural networks,* as a most general class of functions, revealed the promise of capturing certain tasks through the conceptually-simple approach of function fitting (or "curve fitting"). Using data in the form of input-output pairs, we managed to capture certain tasks that defied modeling for a long time, by simply fitting a complex function (deep neural network) to this data (e.g., identifying and localizing objects in images).

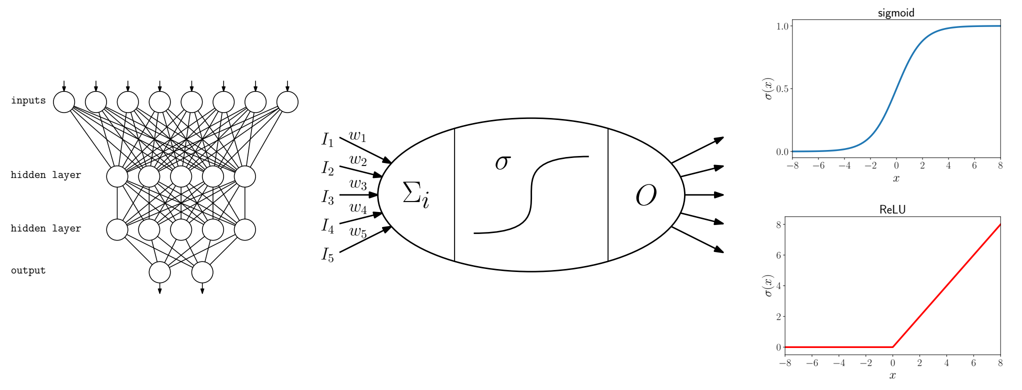
Each of these frameworks has its shortcomings, which triggered the corresponding milestone. Symbolic models are too coarse for capturing certain phenomena and, in their pure form, miss out on exploiting data and the wealth of information it may contain. Probabilistic graphical models, in their generative and discriminative forms, have been recently outperformed by function-based approaches which are optimized for specific/narrow tasks. Moreover, we are now starting to realize the limits of function-based approaches: they are sometimes unable to generalize beyond the given data, can be quite brittle, and are not interpretable. Ironically, it is these shortcomings that models, whether symbolic or probabilistic, can help alleviate. Hence, a key challenge/opportunity for AI today lies in *integrating* and *fusing* these approaches to realize their collective benefits.

We make a distinction here between integration and fusion. *Integration* may refer to an intelligent agent architecture in which the components may be based on different approaches, but work together in a harmonious way to complement each other. *Fusion* may refer to a model-based approach that is empowered by functions, or a function-based approach that is empowered by models. Our focus in this paper is on fusion, particularly the empowerment of function-based approaches with domain knowledge in the form of models. While our ultimate goal is to consider models that integrate both logic and probability (e.g., (Kisa et al., 2014; Poole, 2003; Halpern, 1990; Nilsson, 1986)), we focus in this paper on probabilistic models in the form of Bayesian networks, while tackling a specific and fundamental question that is highlighted by recent developments in AI.

In particular, our contribution is based on the following observations, which are known in the literature but together lead to a dilemma. First, a query posed to a Bayesian network model can be viewed as inducing (and evaluating) a function, which can be represented using an Arithmetic Circuits (AC) (Darwiche, 2002, 2003). Second, Bayesian networks (and, hence, ACs) can be trained discriminatively using labeled data, leading to an approach that is very similar to neural networks—except that a neural network represents only one function, while a Bayesian network represents many functions (one for each query). Third, neural networks appear to outperform Bayesian networks, even when the latter are trained discriminatively, which begs an explanation.

We shed some light on this question by first observing that the class of functions induced by Bayesian and neural networks have different expressiveness. It is known that neural networks are *universal approximators,* which means that they can approximate any functions to an arbitrary small error. However, we show that the functions induced by Bayesian network queries are polynomials (in fact, multi-linear functions); an observation that is well known but never discussed in this context. To address this expressiveness gap, we propose a simple extension to Bayesian networks, showing that it leads to inducing functions that are also universal approximators. When the newly induced functions are represented by circuits, and trained using labeled data, we obtain a function-based approach that is empowered by models. That is, not only can we now induce and train functions that are as expressive as neural networks, but we can also integrate domain knowledge.

This paper is structured as follows. We review in Section 2 the class of functions induced by neural and Bayesian networks, while identifying the corresponding gap in expressiveness. We then propose a new class of Bayesian networks in Section 3, called *Testing Bayesian Networks (TBN),* whose queries induce functions in the form of *Testing ACs (TAC).* We then show in Section 4 that these functions are universal approximators. We further show in Section 5 that TBNs can efficiently simulate neural networks with step activation functions, and use this observation to develop a TBN for classifying digits with performance that matches neural networks. We finally close in Section 6.



(a) A neural network structure (b) A mathematical model of a neuron (c) Two activation functions

Figure 1: A neural network, a neuron, and two activation functions. A sigmoid $\sigma(x) = \frac{1}{1+\exp\{-x\}}$ acts as a soft threshold which tends to 0 as x goes to $-\infty$ and tends to 1 as x goes to ∞. A ReLU $\sigma(x) = \max(0, x)$ is equal to 0 if $x < 0$ and is equal to x otherwise.

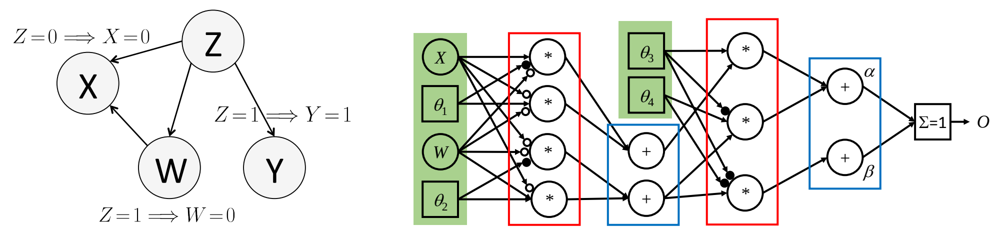
2. Technical Background

We will now review the class of functions represented by neural networks. We will also highlight results from the literature, which allow us to view Bayesian network queries as inducing, and evaluating, functions. The main goal is to pinpoint an expressiveness gap between the two classes of functions, which we address in the following section.

2.1 Neural Networks as Functions

A (feedforward) neural network is a directed acyclic graph (DAG); see Figure 1(a). The roots of the DAG are the neural network inputs, call them $X_1, \ldots, X_n$. The leaves of the DAG are the neural network outputs, call them $Y_1, \ldots, Y_m$. Each node in the DAG is called a *neuron* and contains an *activation function* σ; see Figure 1(b). Each edge I in the DAG has a *weight* w attached to it. The weights of a neural network are its *parameters,* which are learned from data. Consider a neuron with activation function σ, inputs I_i and corresponding weights w_i. The output of this neuron is simply $\sigma(\sum_i w_i \cdot I_i)$. Thus, one can compute the output Y_j of a neural network by simply evaluating neurons, parents before children, which can be done in time linear in the neural network size.

To simplify the discussion, we will assume that a neural network has a single output Y. Hence, a neural network represents a function $Y = f(X_1, \ldots, X_n)$. The question now is what class of functions can be represented by a neural network? If one does not restrict the type of activation functions, then any function can be represented. In practice, one uses specific types of activation functions, such as the *sigmoid;* see Figure 1(c). A neural network with a single hidden layer and only sigmoid activation functions can approximate any continuous function to within an arbitrary error ϵ. Such neural networks are called *universal approximators* of continuous functions (Hornik et al., 1989; Cybenko, 1989; Leshno et al., 1993). A *shallow* network (single hidden layer) is sufficient for universal approximation, but may require exponentially many neurons. A *deep* neural network may be more succinct for this purpose though.


$Z=0 \Longrightarrow X=0$
Z
X
$Z=1 \Longrightarrow Y=1$
W
Y
$Z=1 \Longrightarrow W=0$
X
θ_1
W
θ_2
θ_3
θ_4
α
β
Σ=1
O

(a) Bayesian network with logical constraints ($0/1$ parameters)

(b) function for query $O = Pr(y|x, w)$. Here, $\alpha = Pr(y, x, w)$ and $\beta = Pr(\bar{y}, x, w)$

Figure 2: The function uses adders ($+$), multipliers ($*$), inverters ($\circ$), $1-\theta$ units ($\bullet$), and normalizing units ($\Sigma = 1$). Excluding the Σ unit (division), we get an Arithmetic Circuit (AC) by emulating $\circ$ and $\bullet$ units using adders. The function implicitly integrates the Bayesian network's $0/1$ parameters. Moreover, $\theta_1, \ldots, \theta_4$ are Bayesian network parameters.

Most of the recent neural networks are based on the *ReLU* activation function, which is simpler than the sigmoid; see Figure 1(c). Interestingly enough, networks with ReLUs are also universal approximators of continuous functions (Leshno et al., 1993).

2.2 Bayesian Network Queries as Functions

Consider now a Bayesian network, some evidence $\mathbf{e}$ on variables $\mathbf{E}$ (e.g., symptoms), and let Y be a query variable (e.g., a disease). The probability $Pr(y, \mathbf{e})$ can be viewed as the output of a function $f(\mathbf{E})$ that maps evidence $\mathbf{e}$ into a number in $[0, 1]$. The function inputs are discrete values of variables $\mathbf{E}$, but can be continuous values in $[0, 1]$ if one uses soft evidence (Chan and Darwiche, 2005) (universal approximation results for neural networks assume that inputs/outputs are in $[0, 1]$).

It is known that the function $f(\mathbf{E})$ can be represented by an *Arithmetic Circuit (AC)* containing only multipliers and adders; see Figure 2. It is also known that classical inference algorithms for Bayesian networks can be interpreted as constructing this circuit on the fly (Darwiche, 2009, Chapter 12), while some other approaches construct this circuit explicitly through a compilation process (Darwiche, 2003, 2009). There are two key points here. First, each Bayesian network query $Pr(\alpha)$ is the result of evaluating a function. Second, such functions are *polynomials;* in fact, multi-linear functions to be more specific (Darwiche, 2003, 2009).[1] That is, the class of functions induced by Bayesian networks are less expressive than the ones represented by neural networks. Hence, Bayesian networks are limited in the kind of relationships they can capture between evidence (input) and beliefs (output), as compared to the kind of relationships that can be captured by a neural network. This potentially explains why a Bayesian network that is trained discriminatively using labeled data may not outperform a neural network that is trained for the same query.

Here is our major insight for addressing this expressiveness gap, which comes down to a very simple but consequential observation. It is known that if each activation function σ of a neural network is a polynomial, then the neural network can only represent polynomials. Hence, one needs non-polynomial activation functions to yield a universal approximator (Leshno et al., 1993). Consider now the ReLU activation function $\sigma(x) = \max(0, x)$, which leads to a universal approximator.

1. The conditional probability $Pr(y|\mathbf{e})$ is the quotient of two multi-linear functions.

This function equals 0 if $x < 0$ and equals x otherwise; see Figure 1(c). Hence, it is a linear function augmented with a simple test, $x < 0$. What this tells us is that one can turn ACs into universal approximators by only integrating testing units (in addition to multiplier and adders). We show this in the next section, leading to *Testing ACs (TAC)*. In fact, the notion of *testing* can be integrated directly into Bayesian networks, leading to *Testing Bayesian Networks (TBN).* A TAC will then be the result of compiling a TBN query, just like an AC can be compiled for a Bayesian network query.

3. Testing Bayesian Networks

The concept of a Testing Bayesian Network (TBN) is relatively simple. Consider a node X in the network with parents $\mathbf{U}$ and suppose we are specifying the conditional probability table (CPT) for node X. Instead of having a fixed parameter $\theta_{x|\mathbf{u}}$ for the conditional probability $Pr(x|\mathbf{u})$, node X will choose this parameter depending on a test of the form $Pr(\mathbf{u}|\mathbf{e}_X) \geq T_{X|\mathbf{u}}$. Here, $T_{X|\mathbf{u}}$ is a threshold that is specific to node X and its parents state $\mathbf{u}$, and $\mathbf{e}_X$ is the evidence pertaining to the ancestors of node X. That is, the CPT for node X is determined *dynamically,* based on the distributions $Pr(\mathbf{u}|\mathbf{e}_X)$ and thresholds $T_{X|\mathbf{u}}$. We will now give two illustrative examples of TBNs.

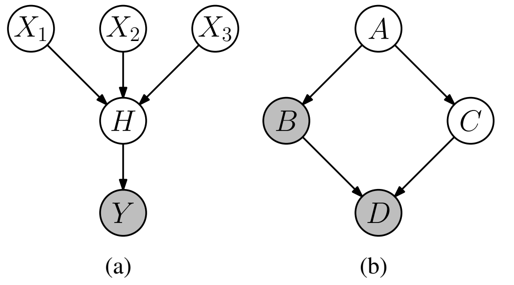


Figure 3: TBNs (testing nodes are shaded).

Consider Figure 3(a), while ignoring variable Y for now. Variables X_1, X_2, X_3, H are the basis of a noisy-or classifier as in (Vomlel, 2006). That is, we classify an instance x_1, x_2, x_3 positively iff $Pr(h|x_1, x_2, x_3) \geq T$, where T is the classification threshold. We can implement this classifier using a TBN, by adding the single *testing node* Y having the following *testing CPT*:

$$\theta_{y|h} = \begin{cases} 1 & \text{if } Pr(h \mid x_1, x_2, x_3) \geq T \\ 0 & \text{otherwise} \end{cases} \qquad \theta_{y|\bar{h}} = \begin{cases} 1 & \text{if } Pr(\bar{h} \mid x_1, x_2, x_3) \leq 1 - T \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

That is, $Pr(y|x_1, x_2, x_3) = 1$ iff instance x_1, x_2, x_3 is positive, and $Pr(y|x_1, x_2, x_3) = 0$ otherwise.

More generally, we may have multiple testing nodes in a TBN. Figure 3(b) depicts an example where all variables are binary and nodes B and D are both testing. In a classical Bayesian network (BN), we need 9 independent parameters to fully specify the network (1 for A, 2 for each of B, C and 4 for D). For the TBN, we need 15 independent parameters (2 additional parameters for B and 4 additional parameters for D). We also need 2 thresholds for B and 4 thresholds for D.

3.1 TBN Syntax

Formally, a TBN is a directed acyclic graph (DAG) with two types of nodes: *regular* and *testing.* The CPT of a regular node is defined as in BNs and its parameters are said to be *static.* Root nodes of a TBN are always regular. A testing node X with parents $\mathbf{U}$ has the following *testing CPT:*

$$\theta_{x|\mathbf{u}} = \begin{cases} \theta^{+}_{x|\mathbf{u}} & \text{if } Pr(\mathbf{u} \mid \mathbf{e}_X) \geq T_{X|\mathbf{u}} \\ \theta^{-}_{x|\mathbf{u}} & \text{otherwise} \end{cases}$$

Here, $\theta^{+}_{x|\mathbf{u}}$ and $\theta^{-}_{x|\mathbf{u}}$ are called *dynamic parameters,* $T_{X|\mathbf{u}}$ are called *thresholds,* and $\mathbf{e}_X$ is the evidence pertaining to the ancestors of node X. A testing CPT corresponds to a set of classical CPTs, one of which is selected based on the evidence $\mathbf{e}_X$. If a testing node has m states, and its parents have n states, its testing CPT will have m thresholds and $2 \cdot m \cdot n$ dynamic parameters, while a classical CPT will have $m \cdot n$ *static* parameters. As we shall discuss later, the thresholds and parameters of a TBN can be learned discriminatively from labeled data (as in deep learning).

3.2 TBN Semantics

A BN represents a single distribution. However, a TBN represents a set of distributions, one of which is *selected* based on the given evidence. In particular, once each testing node has selected its classical CPT based on the given evidence, the TBN transforms into a BN that has the same DAG as the TBN and that specifies the selected distribution.

Definition 1 *The selected distribution $P(.)$ of a TBN given evidence $\mathbf{e}$ is defined inductively:*

- *For a root node X, the distribution over X is defined as $P(x) = \theta_x$.*
- *For node X with parents $\mathbf{U}$, let $P(\mathbf{A})$ be the selected distribution over its ancestors $\mathbf{A}$, let $\mathbf{e}_X$ be the evidence on the ancestors of node X, and let $\mathbf{u}$ be the parent instantiation compatible with an ancestor instantiation $\mathbf{a}$. The selected distribution over $X\mathbf{A}$ is defined as follows:*

  If X is a regular node, then $P(x, \mathbf{a}) = \theta_{x|\mathbf{u}} \cdot P(\mathbf{a})$.

  If X is a testing node, then $P(x, \mathbf{a}) = \begin{cases} \theta^+_{x|\mathbf{u}} \cdot P(\mathbf{a}) & \text{if } P(\mathbf{u}|\mathbf{e}_X) \geq T_{X|\mathbf{u}} \\ \theta^-_{x|\mathbf{u}} \cdot P(\mathbf{a}) & \text{otherwise.} \end{cases}$

With no testing nodes, Definition 1 reduces to the classical one for Bayesian network semantics.[2]

Definition 2 *For a TBN, query variables $\mathbf{Q}$ and evidence variables $\mathbf{E}$, the selected probability of $\mathbf{q}$ given $\mathbf{e}$ is defined as $\mathbf{q}||\mathbf{e} = P(\mathbf{q}|\mathbf{e})$, where $P(.)$ is the selected distribution given evidence $\mathbf{e}$.*

That is, we first select a distribution based on evidence $\mathbf{e}$ and then use it to compute the conditional probability of $\mathbf{q}$ given $\mathbf{e}$. Hence, the probabilities $\mathbf{q}||\mathbf{e}_1$ and $\mathbf{q}||\mathbf{e}_2$ may be computed based on distinct distributions, if $\mathbf{e}_1 \neq \mathbf{e}_2$. This is also why we prefer the notation $\mathbf{q}||\mathbf{e}$ in contrast to $Pr(\mathbf{q}|\mathbf{e})$.

3.3 Compiling TBN Queries into TACs

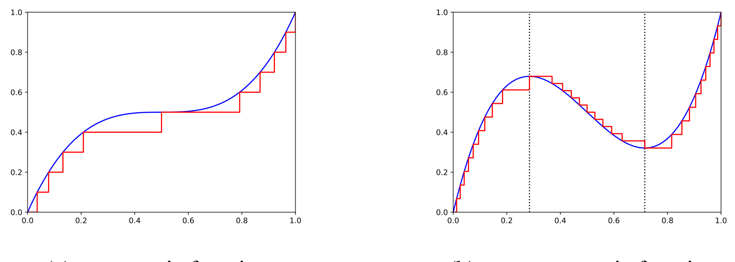
A Testing Arithmetic Circuit (TAC) is an Arithmetic Circuit (AC) that includes *testing units.* Such units have a single input (X), a single output $Y = f(X)$ and three parameters T, θ^+, θ^-:

$$f(x) = \begin{cases} \theta^+ & \text{if } x \geq T \\ \theta^- & \text{otherwise} \end{cases}$$

A testing unit with $\theta^+ = 1$ and $\theta^- = 0$ corresponds to the *step* activation function used in neural networks. Figure 5(b) depicts a TAC with five such testing units. Moreover, a testing unit $f(x)$ with $T = 0$, $\theta^+ = 1$ and $\theta^- = 0$ can emulate a ReLU $g(x)$ as follows: $g(x) = x \cdot f(x) = max(0, x)$.

A BN query can be compiled into an AC; see Figure 2. Similarly, a TBN query can be compiled into a TAC. A BN query can be compiled into an AC by inducing a symbolic trace from a variable

2. One can generalize Definition 1 so $\mathbf{e}_X$ is the evidence on non-descendants of node X, assuming evidence $\mathbf{e}$ does not include descendants of any testing node. That is, Definition 1 will lead to a unique selected distribution in this case.



(a) monotonic function (b) non-monotonic function

Figure 4: Two functions and their approximations.

elimination or jointree algorithm, as described in (Darwiche, 2009, Chapter 12). We next sketch a similar, albeit inefficient, approach for compiling a TBN query into a TAC.

Consider a TBN over variables $X_1, \ldots, X_n$ that are topologically sorted. Consider also a corresponding jointree C_1—C_2—…—C_n where cluster C_i contains variable X_i and its parents (among other variables). Let X_j be our query variable. If we pass messages towards cluster C_j, the following is guaranteed. For a testing node X_i with parents $\mathbf{U}_i$, the message passed from cluster C_{i-1} to cluster C_i will contain the marginals $Pr(\mathbf{u}|\mathbf{e}_{X_i})$ needed to perform the tests for node X_i. Hence, node X_i can pick its classical CPT, and then pass an appropriate message to cluster C_{i+1}. One can extract an AC from this message passing process (Park and Darwiche, 2003; Darwiche, 2009), and perform local modifications to incorporate the needed testing units, leading to a TAC for the TBN.

3.4 Discussion

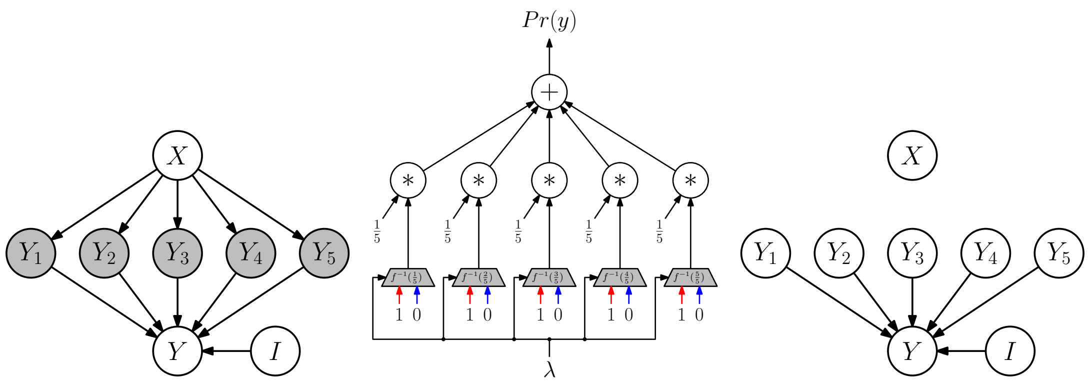
There is perhaps much work needed to fully analyze and understand the properties of TBNs, but we already know of a few desirable properties. For example, inference on a TBN can be tractable even when it is intractable on a BN with the same structure; see Section 5. Moreover, the distributions represented by a TBN all respect the independence properties captured by the TBN structure, and all satisfy the constraints imposed by the TBN known parameters. In fact, the values of any known parameters will be integrated into TACs that are induced by TBN queries. In Figure 2, for example, we already know the values of some parameters due to domain knowledge in the form of logical constraints. Any TAC (or AC) that is induced from this network will satisfy these constraints. Hence, we can now synthesize functions and train their unknown parameters ($\theta_1, \ldots, \theta_4$ in this case), while providing some guarantees on the function properties based on the inducing model.

4. Universal Approximation

We now show that any continuous, monotonic function $f(x)$ from $[0, 1]$ to $[0, 1]$ can be approximated by a TBN query. We later generalize our result to multivariate, non-monotonic functions.

4.1 Monotonic and Univariate Functions

Our proof is constructive and based on Jones (1990), and uses the TBN and TAC in Figure 5. The TBN has regular nodes X, Y and I, which are discrete. Nodes X and Y are binary with states $x, \bar{x}$ and $y, \bar{y}$. Node I is called a *selector* with values $1, \ldots, N$, with better approximations for larger



(a) TBN (b) TAC for query $Pr_\lambda(y)$ (c) BN after testing is done

Figure 5: The first layer of the TAC has testing units labeled with their testing threshold.

N. The TBN has testing nodes $\mathbf{Y} = Y_1, \ldots, Y_N$, which are binary with states $y_i, \bar{y}_i$. The TAC computes the query $Pr_\lambda(Y{=}y)$: the probability of y given soft evidence on x that is quantified by $\lambda \in [0,1]$, i.e., λ is the posterior on x once soft evidence is asserted (Chan and Darwiche, 2005).[3]

Nodes X and I have uniform priors. Node Y has the following CPT:

$$Pr(Y{=}y \mid \mathbf{y}, I{=}i) = \begin{cases} 1 & \text{if } \mathbf{y} \text{ sets variable } Y_i \text{ to value } y \\ 0 & \text{otherwise} \end{cases}$$

That is, Y is equivalent to node Y_i when selector I is set to index i. The testing CPT for node Y_i is:[4]

$$\theta_{y_i|x} = \begin{cases} 1 & \text{if } Pr_\lambda(x) \geq f^{-1}(\frac{i}{N}) \\ 0 & \text{otherwise} \end{cases} \qquad \theta_{y_i|\bar{x}} = \begin{cases} 1 & \text{if } Pr_\lambda(\bar{x}) \leq 1 - f^{-1}(\frac{i}{N}) \\ 0 & \text{otherwise} \end{cases}$$

We now have the following result.

Theorem 3 *For a continuous and monotonic function $f(x)$ from $[0,1]$ to $[0,1]$, and error ε, we have $|f(x) - Pr_\lambda(y)| \leq \varepsilon$ if the TBN in Figure 5(a) uses $N = \lceil \frac{1}{\varepsilon} \rceil$.*

Proof The CPT of each node Y_i depends on the posterior distribution on X given soft evidence λ (the posterior probability of $X{=}x$ and $X{=}\bar{x}$ is just λ and $1-\lambda$ in this case). Once each testing node Y_i selects its CPT, our network simplifies to the structure depicted in Figure 5(c). This is now a classical Bayesian network; denote its distribution by Pr_λ. We now have

$$\begin{aligned} Pr_\lambda(y) &= \sum_{\mathbf{y}} \sum_{i=1}^{N} Pr_\lambda(y \mid I{=}i, \mathbf{y}) Pr_\lambda(I{=}i, \mathbf{y}) \\ &= \sum_{\mathbf{y}} \sum_{i=1}^{N} Pr_\lambda(y \mid I{=}i, \mathbf{y}) \cdot \theta_{I=i} \cdot \prod_{i=1}^{N} \theta_{y_i} = \sum_{i=1}^{N} \theta_{I=i} \cdot \theta_{y_i}. \end{aligned}$$

3. The symbol x is overloaded. In the TBN, x is a value of binary variable X. In function $f(x)$, it is a value in $[0,1]$.
4. We assume, without loss of generality, that $f(0) = 0$ and $f(1) = 1$.

Since $\theta_{I=i} = \frac{1}{N}$, we $Pr_\lambda(y) = \frac{1}{N}\sum_{i=1}^{N}\theta_{y_i}$. Intuitively, each test for node Y_i is either *activating* or *de-activating* that node. If it is activated, then $\theta_{y_i} = 1$; otherwise it is de-activated and $\theta_{y_i} = 0$. Hence, the probability $Pr_\lambda(y)$ equals the proportion of activated nodes. By construction (of the testing CPTs), as the value of λ increases, the more nodes Y_i get activated. In fact, exactly $\lfloor N \cdot f(x) \rfloor$ will be activated, leading to $Pr_\lambda(y) = \frac{1}{N}\lfloor N \cdot f(x) \rfloor$ and $|f(x) - Pr_\lambda(y)| \leq \varepsilon$. ■

4.2 Non-Monotonic and Multivariate Functions

Figure 4(b) depicts a non-monotonic function $f(x)$ from $[0, 1]$ to $[0, 1]$. To approximate $f(x)$ with a testing BN, we first split our function into monotonic pieces. In Figure 4(b), we have used vertical dotted lines to mark the points where the sign of the first derivative $\frac{df}{dx}$ changes.

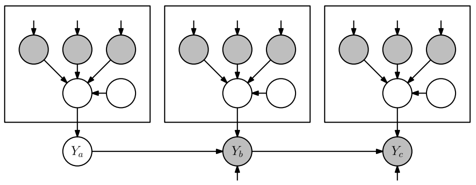
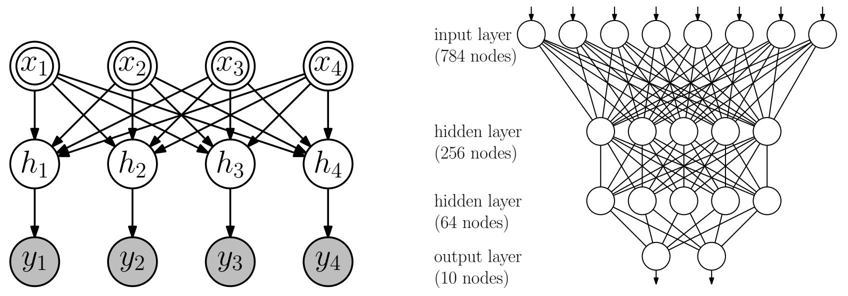


Figure 6: A chain of TBNs.

We can use the testing BN for approximating monotonic functions, as a building block for approximating non-monotonic functions. Consider the testing BN of Figure 6 that approximates the function $f(x)$ of Figure 4(b). For clarity, we do not draw the edges starting from X, and instead draw incoming edges without the tail node X. Here, each of the three sub-networks enclosed in a box approximates each of the three monotonic components of our function $f(x)$. Let a, b and c denote the three components, and let T_{ab} and T_{bc} denote the values of x at the two borders. We have a chain of nodes $Y_a \rightarrow Y_b \rightarrow Y_c$ on the bottom of our network. Node Y_a simply copies the value of its parent (its CPT is an equivalence constraint). Node Y_b will either copy the value of Y_a or the value of its component b, depending on whether the input x is below or above the threshold T_{ab}. Node Y_c will either copy the value of Y_b or the value of its component c, depending on whether the input x is below are above the threshold T_{bc}. This sequence of threshold tests will, given an input x, select the approximation of $f(x)$ from the appropriate component, which is finally obtained as the probability of Y_c. Again, the error of our approximation depends on the size of N used to approximate each component. The size of the construction is also linear in the number of times that the sign of the first derivative changes.

The generalization to multivariate functions is analogous to the approximation of functions using ridge and bump functions (Jones, 1990; Lapedes and Farber, 1987). First, we use our construction for approximating a univariate function via a testing BN as a building block to approximate a function $f(x_1, x_2)$ over two variables. In particular, we construct a testing BN for N univariate functions $f_{x_2}(x_1) = f(x_1, x_2)$ for N values of x_2 from 0 to 1. As we did previously for approximating non-monotonic univariate functions by pieces, we construct a chain of these N testing BNs and copy the output of the appropriate component based on the input value of x_2. To approximate a function $f(x_1, \ldots, x_n)$ over n variables, we construct N testing BNs that approximate functions $f_n(x_1, \ldots, x_{n-1})$ over $n - 1$ variables, and then perform a similar construction.

The error in the approximation can be improved arbitrarily by increasing N (under some assumptions, i.e., the change in f is bounded for small changes in the input). Moreover, this construction is exponential in the number of input variables n. Related constructions for showing neural networks (with one or two hidden layers) are "universal approximators" are also exponential in the



(a) TBN (b) neural network

Figure 7: A TBN that simulates a layer of a neural network.

dimension of the function. However, the expectation is that deeper networks, with more hidden layers, are able to approximate increasingly broader classes of functions succinctly.

5. Simulating Neural Networks

We will discuss three results in this section. First, we will show that a neural network with step activation functions can be simulated by a query on a TBN of the same size (within a constant factor). Second, we will show that this simulation is efficient in the sense that the TBN query can be evaluated in time linear in the TBN size—even though the TBN structure is connected to the point where it is not tractable when quantified by classical CPTs. Third, we will utilize our construction to simulate a neural network for classifying digits by a TBN, and show a matching performance.

The simulation result is based on (1) the observation we made in Section 3 about using TBNs for representing noisy-or classifiers, and (2) a result by Vomlel (2006), showing that a neuron with a step activation function can be simulated by a noisy-or classifier.

Consider a neuron f with a step activation function. Let $\mathbf{x}$ be the neuron's input vector, x_i the i-th input, w_i the i-th weight, and $T^\star$ the threshold. Then $f(\mathbf{x}) = 1$ if $\sum_i w_i x_i \geq T^\star$ and $f(\mathbf{x}) = 0$ otherwise (we assume $\mathbf{x}$ is a binary vector). We can simulate this neuron using the TBN in Figure 3(a), having testing node Y and the testing CPT in (1). We can determine the TBN parameters from the weights w_i, and the TBN threshold T from both the weights w_i and threshold $T^\star$ (Vomlel, 2006), giving us $f(\mathbf{x}) = 1$ iff $y \| \mathbf{x} = 1$.[5] Now that we can simulate the neuron using a TBN, we simply cascade these TBNs to simulate a neural network; see Figure 7(a).

We applied this reduction to a neural network for the MNIST digit classification task, a standard benchmark in the neural networks literature (`http://yann.lecun.com/exdb/mnist`). The MNIST dataset is composed of 28×28 grayscale images of handwritten digits, which we binarized using a threshold of 0.5 (out of a range from 0.0 to 1.0). Hence, each example of the dataset is a vector of $28 \times 28 = 784$ zeroes and ones. Each example is labeled with its corresponding digit (from 0 to 9) and our goal is to learn a classifier that can predict the digit of a given 28×28 image. MNIST contains $60,000$ training examples and $10,000$ testing examples.

Training neural networks with step functions can be challenging. A step function is not differentiable at the threshold, and has a zero derivative everywhere else. Hence, classical gradient methods do not directly apply. In practice, one can train with sigmoid or ReLU functions, using gradient

5. More specifically, we get $f(\mathbf{x}) = 1$ iff $Pr(h \mid \mathbf{x}) = \prod_i \frac{\exp\{w_i x_i\}}{1+\exp\{w_i\}} \geq \exp\{T^\star\} \prod_i (1+\exp\{w_i\})^{-1} = T$.

methods. At test time, however, we can use step functions, as we did in our experiment. We used TensorFlow to train a neural network with sigmoid activation functions and the architecture depicted in Figure 7(b).[6] We used the resulting weights and thresholds to parameterize our TBN, leading to 95.22% test set accuracy. The neural network with sigmoid activations obtained a 95.21% accuracy.

We now get to our last result, which pertains to the complexity of inference on TBNs. The TBN we constructed in the above experiment is very connected, with a treewidth that cannot be handled by classical Bayesian network algorithms. However, inference on this TBN is efficient for the following reason. Consider a testing node Y with parents $\mathbf{U}$ in a TBN. It is possible that the CPT selected by node Y is such that the distributions $Pr(Y|\mathbf{u})$ are all the same for any $\mathbf{u}$. This means that node Y will no longer depend on its parents $\mathbf{u}$, once the tests by node Y are concluded. In fact, this property holds for the TBN that simulates a neural network, and also for the TBN we used to prove Theorem 3. If a testing node has this property, we can disconnect it from its parents after the test is conducted, making it a root node with prior $Pr(Y|\mathbf{u})$ (for any $\mathbf{u}$); see Figure 5(c). As a result, a TBN that is very connected may end up being very sparse once testing is done. This is why we can efficiently evaluate the TBN that simulates a neural network (as in the MNIST TBN).

Since we now know that TBNs are as powerful as neural networks as far as approximating functions, our next step is to (1) consider more general methods for compiling a TAC from a TBN and a given query, and (2) learn the TAC parameters and thresholds directly from labeled data. We can think of the compiled TAC as a neural network, but one that we can provide guarantees on. In particular, the TAC will satisfy certain properties that hold regardless of which parameters are learned (by virtue of the TBN properties). Moreover, the TAC will be more interpretable as its structure is synthesized from a model, instead of being either engineered or chosen arbitrarily.

6. Conclusion

We considered the relative expressiveness of Bayesian and neural networks. Neural networks are "universal approximators" of continuous functions, whereas Bayesian networks can only represent multi-linear functions for marginal probability queries, and a quotient of multi-linear functions for conditional probability queries. We proposed Testing Bayesian Networks (TBN) whose queries are also "universal approximators," and Testing Arithmetic Circuits (TAC) for representing these queries. Moreover, we showed how to simulate neural networks with step activation functions using queries on TBNs, whose TACs can subsequently be learned from labeled data using gradient methods (as in deep learning). We finally argued that TBNs and TACs move us a step forward towards fusing model-based and function-based approaches to AI.

Acknowledgments

This work has been partially supported by NSF grant #IIS-1514253, ONR grant #N00014-18-1-2561 and DARPA XAI grant #N66001-17-2-4032.

6. Our network had 784 input nodes, two hidden layers of 256 and 64 nodes each with sigmoid activations, and an output layer of 10 nodes with linear activations, leading to a neural network and testing BN with $(784+1)\cdot 256 + (256+1)\cdot 64 + (64+1)\cdot 10 = 218,058$ parameters. We used the Adam optimizer with default settings, for 1,000 steps and a batch size of 512. The predicted digit is taken from the output of maximum value.

References

H. Chan and A. Darwiche. On the revision of probabilistic beliefs using uncertain evidence. *Artificial Intelligence*, 163:67–90, 2005.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

A. Darwiche. A logical approach to factoring belief networks. In *KR*, pages 409–420, 2002.

A. Darwiche. A differential approach to inference in Bayesian networks. *J. ACM*, 50(3):280–305, 2003.

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

A. Darwiche. Human-level intelligence or animal-like abilities? *Communications of the ACM*, 2018. To appear (`http://arxiv.org/abs/1707.04327`).

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

J. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, 1990.

G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

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

L. K. Jones. Constructive approximations for neural networks by sigmoidal functions. *Proceedings of the IEEE*, 78(10):1586–1589, 1990.

D. Kisa, G. Van den Broeck, A. Choi, and A. Darwiche. Probabilistic sentential decision diagrams. In *KR*, 2014.

A. S. Lapedes and R. M. Farber. How neural nets work. In *NIPS*, pages 442–456, 1987.

M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, 1959.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

N. Nilsson. Probabilistic logic. *Artificial intelligence*, 28(1):71–87, 1986.

J. Park and A. Darwiche. A differential semantics for jointree algorithms. In *NIPS*, pages 299–307, 2003.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. MK, 1988.

D. Poole. First-order probabilistic inference. In *IJCAI*, pages 985–991, 2003.

F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

J. Vomlel. Noisy-or classifier. *Int. J. Intell. Syst.*, 21(3):381–398, 2006.