Eindhoven University of Technology

MASTER

Robustness of discriminative and generative classifiers

Meeuwisse, Thijs P.

Award date:
2021

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science
Uncertainty in Artificial Intelligence Group

MASTER'S THESIS

Robustness of discriminative and generative classifiers

Thijs P. Meeuwisse

Supervisors: Cassio de Campos & Alvaro Correia

EINDHOVEN, DECEMBER 2021

So I appear to be wiser, at least than him, in just this one small respect: that when I don't know things, I don't think that I do either.

— Plato, *Apology* 21d

Abstract

As machine learning (ML) is becoming more ubiquitous, the demands placed on ML models are increasing too. We expect models to be *robust*, that is, insensitive to small irrelevant perturbations in the input. If models are not robust, an adversary can alter the model output to his advantage by perturbing the input. These perturbations are typically undetectable by humans.

In classification problems, we can either use a discriminative or generative approach. In the generative approach, the model captures the joint distribution of input and class. Interest in generative models is growing, since they are more powerful than discriminative models. For instance, they can fill in missing data and generate new examples. In this project, I investigate if generative models are more robust than discriminative models. I consider robustness to corruptions in the input and robustness to adversarial attacks. In the latter, an adversary deliberately perturbs the input. The experiments are performed using the formalism of probabilistic circuits (PCs), which can be trained both discriminatively and generatively. This makes them suitable for carrying out comparisons. The first results indicate that generative models are more robust than discriminative models, but only if the model was trained on a sufficient amount of data. The observation that differences in robustness between discriminative and generative models only become apparent as sample size grows, is novel.

Acknowledgements

This project would not have come to fruition if it wasn't for the tremendous support I received from the people around me. I would like to thank my supervisors, Cassio de Campos and Alvaro Correia, for their valuable insights and feedback on my work. Without your help, I would have been at a complete loss. I would like to thank my friends, who were incredibly supportive and provided the much needed distractions. I am also grateful to my housemates, who on many days during these confusing times were the only people I saw. They reminded me that there is more to life than writing a thesis.

Graduating while in lockdown was not easy for me. Stripped down of things to look forward to, I sometimes lost the motivation to continue with the project. Above all, I would like to thank my parents, Mieke and Annet, who were always there for me when I was having a hard time. Without your support, not just during the project, but my whole lifetime, I would not have been where I am now.

Thijs Meeuwisse
5 December 2021

Contents

List of figures

List of tables

Notational conventions

- Events are labeled with capital letters, e.g. A, B, C.
- A random variable (RV) is capital X and takes values small x. The connection between values and events: '$X = x$' is the event that X takes the value x.
- An *indexed set* of random variables $(X_1, X_2, \ldots)$ is bold capital $\mathbf{X}$ and takes values small bold $\mathbf{x}$. The connection between values and events: '$\mathbf{X} = \mathbf{x}$' is the event that $\mathbf{X}$ takes the value $\mathbf{x}$.
- The random variables and assignments within a set are indexed with a subscript, e.g. X_i and x_i.
- The probability of an event is capital $P(A)$.
- For conciseness, $P(\mathbf{X} = \mathbf{x})$ is shortened to $P(\mathbf{x})$, if this does not lead to ambiguity.
- The domain/state space of a random variable X is denoted as $\mathsf{val}(X)$.
- The state space $\mathsf{val}(\mathbf{X})$ of a set of n random variables $\mathbf{X}$ consists of all possible states $\mathsf{val}(X_1) \times \ldots \times \mathsf{val}(X_n)$

These conventions are adapted from Orloff and Bloom, 2014 and Y. Choi et al., 2020, p. 1.

1 Introduction

1.1 Motivation

HUMANS PERCEIVE differently than machines. If we see an image, and it is perturbed by noise, we are quite capable of ignoring the noise, as long as the signal to noise ratio is not too bad. Machines on the other hand may have great difficulty in drawing the right conclusions under a wide range of conditions; we say that their methods of *inference* are not *robust*. In other words, we can easily fool the machine.

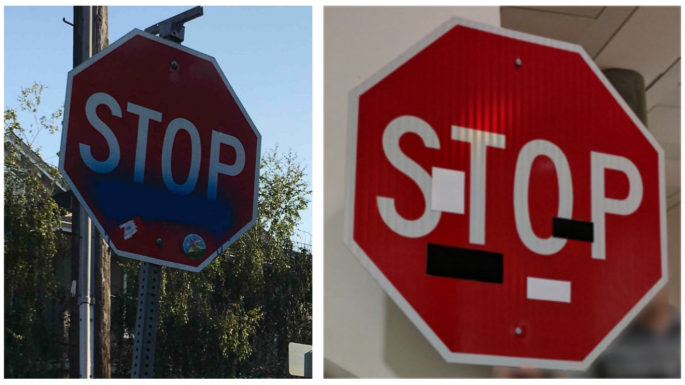
Perceptual tasks, such as vision and auditory perception, are not the only tasks prone to manipulation. Other tasks, such as inference from tabular data, can be manipulated as well, in ways that fool the machine—but not the human.

Ideally, we would want to understand the phenomenology of the machine. With this I mean: we would like to understand *how* the machine *experiences* the data it is fed. Unfortunately, we will never be able to know what it is like to be a bot.[1] What we can do however, is try to make the systems we use more robust, so that for *practical* purposes, the machine performance is comparable to that of humans.

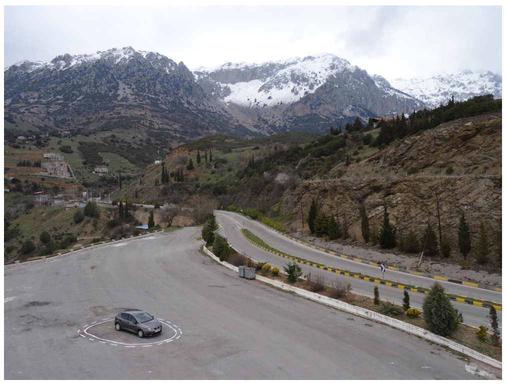
[1] With a nod to Thomas Nagel, who wrote the seminal paper 'What Is It Like to Be a Bat?' (1974).

Robustness in machine learning (ML) is important for two reasons. First of all, we would like models to perform well under sub-optimal conditions. Think of situations where the input data is noisy. It is undesirable if machine learning models are very sensitive to small perturbations in the input data, because these perturbations are meaningless to a human observer, and should be meaningless to the model too. Secondly, the lack of robustness can be exploited by an adversary. In *adversarial attacks*, a very small variation in the input leads to a completely different outcome. By attacking a model, the objectives of the model can be compromised. For instance, if a model designed for spam detection is attacked, it may no longer be able to detect spam.

There are many real-world situations that show the importance of robust models. As an example, consider the case of autonomous driving. An autonomous car should be able to learn about its environment in order to drive safely. This includes recognising traffic signs and road markings. Eykholt et al. (2018) succeeded in physically perturbing traffic signs, so that the computer vision model misclassified the signs. Figure 1.1a displays an example where the authors have



(a)



(b)

Figure 1.1: Illustrations of importance of robustness in autonomous driving.
(a) Left: a stop sign sprayed with real graffiti. Right: a physical adversarial perturbation applied to a stop sign that mimics graffiti. From Eykholt et al., 2018, p. 2.
(b) 'Autonomous Trap 001', an art project by James Bridle. From Mufson, 2017.

successfully introduced an adversarial perturbation to a traffic sign. While the perturbation would not yield suspicion with the human operator, the car can no longer interpret the stop sign, which can lead to dangerous situations. Artist James Bridle has created a project (see figure 1.1b) in which a car is trapped inside road markings. Although the car is not an actual self-driving vehicle, the presented image speaks volumes: by modifying the built environment, someone with nefarious intent can fool machines in ways that humans would never be fooled.

1.1.1 *The classification problem*

The example presented in the previous section is an instance of the *classification problem*. It can be defined as follows:

Definition 1.1 (Classification)**.** Classification is the problem of identifying which of a set of categories an observation belongs to.

The categories are commonly called 'classes'. The simplest scenario is binary classification. In this scenario, there are two classes and an observation always belongs to a single class. We can however extend the problem to *multiclass* settings, where there are multiple classes. An observation is then assigned to one of several classes. The problem can be extended further by allowing an observation to be assigned to multiple classes at the same time. We then speak of *multilabel* classification.

The classification problem is common in ML, and in the remainder of this thesis, this is the problem I focus on. The toolkit of classification algorithms that have been developed is large. I will not try to give an overview of all classification methods, but I will pick out two and discuss their most important properties.

First there is *naive Bayes*, which is a simple version of the (much larger) class of Bayesian network (BN) models. In this approach, each explanatory variable of the observation—also called 'feature' in machine learning—is treated as independent from all other explanatory variables. For an instance $\mathbf{x} = (x_1, \dots, x_n)$, the model assigns a probability

$$P(Y_k \mid \mathbf{X} = \mathbf{x})$$

for each of the K classes Y_k, with $k = 1, \ldots, K$. By assuming independence of the features, and using Bayes rule, the problem can be greatly simplified to

$$P(Y_k \mid x_1, \ldots, x_n) \propto P(Y_k) \prod_{i=1}^{n} P(x_i \mid Y_k).$$

$\propto$ indicates proportionality. In order to get the actual probability, we would need a scaling factor. But for the purpose of classification the actual probabilities are not that important; only the relative differences are.

The naive Bayes model is performant and highly explainable. A problem however is its limited *expressiveness*. Not every probabilistic distribution can be encoded by a naive Bayes model.

Another classification method is by using artificial neural networks (ANNs). Neural networks are universal approximators.[2] This means that in principle, they can model any distribution. This is an enormous advantage over methods like naive Bayes. The advantage comes at a price, though: ANNs are very hard to interpret. We need to develop elaborate methods in order to explain why a neural network came to certain conclusion. Methods like ANNs exhibit *deep opacity*, because their processes are opaque not just to laypeople, but to experts as well.[3]

[2] Cybenko, 1989.

[3] Müller, 2021.

1.1.2 Generative and discriminative models

There are two main approaches to the classification problem: the *generative* approach and the *discriminative* approach. In this section, I will explain these approaches and discuss their properties.

Classifiers like ANNs use the discriminative approach. In the context of classification, this means that they aim to learn a direct map from input $\mathbf{X}$ to a class Y_k.[4] In other words: the conditional distribution $P(Y_k \mid \mathbf{X})$ is modelled directly. *Generative approaches*, on the other hand, try to recover the data-generating distribution; they model the joint distribution $P(\mathbf{X}, Y_k)$ and then make predictions using the Bayes rule.[5] If we were able to recover the data-generating distribution perfectly, we could answer any question we may have about the data-generating process. Figure 1.2 gives an intuition of the difference between learning a decision boundary (discriminative) and learning the data-generating distribution (generative).

[4] Ng and Jordan, 2001.

[5] Ng and Jordan, 2001.

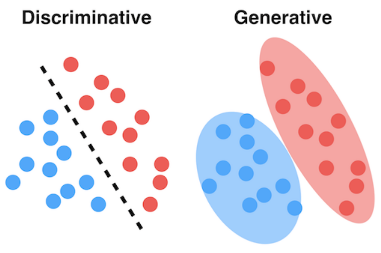


Figure 1.2: Discriminative *vs.* generative modelling. From https://dataisutopia.com/blog/discremenet-generative-models/.

Because generative approaches model the joint distribution, the range of queries we can pose to the model is far larger. Whereas a discriminative model can only yield predictions about the class, in a generative model, the class variable is treated in the same way as the input features in $\mathbf{X}$. This opens up new possibilities, such as generating new observations—hence the term 'generative'—, performing outlier detection, or filling in missing parts of an observation.[6] Because of these capabilities, generative models can do classification even if the value assignment in feature vector $\mathbf{x}$ is incomplete. These additional capabilities entail that, from the outset, the comparison between discriminative and generative models is not straightforward. Taking ANNs and BNs as examples, we should note that a neural network computes a *function* which we can interpret as a conditional probability distribution, whereas a Bayesian network is a *model* that

[6] Correia et al., 2020, p. 2.

can answer many different (types of) queries.[7] If we are to make a comparison, then, we should compare the *queries* of a Bayesian network, because we can interpret those as functions.

[7] A. Choi and Darwiche, 2018.

1.1.3 Tying it together: the robustness of generative models

The additional capabilities of generative approaches over discriminative approaches are nice and all, but if all we want to do is classification, one might ask: what is benefit do generative models bring over discriminative models? This sentiment is expressed succinctly by Vapnik (1998, p. 12), who states that one should "solve the problem directly and never solve a more general problem as an intermediate step." And indeed, many recent approaches in classification, such as the typical ANN, adhere to this principle. Discriminative approaches also tend to yield higher accuracy in classification tasks than generative approaches.[8]

[8] Ng and Jordan, 2001.

An aspect of generative models that has not received widespread attention, is their robustness. While it has been shown that some generative models are better *calibrated*, that is, the model "knows that it does not know",[9] this result does not necessarily imply that generative models are more robust. The aim of my thesis is to investigate the robustness of generative classifiers, compared to discriminative classifiers. The intuition is as follows. Because of their data-generating properties, generative models 'know more' about the probability distribution of their input. If the input is perturbed somewhat, the model is less easily fooled. I therefore formulate the following hypothesis:

[9] Peharz et al., 2018, p. 2.

> Generative models are more robust to noise and attacks than discriminative models.

If this hypothesis is confirmed, we will have an additional reason for preferring them over their discriminative counterparts.

There are many approaches to generative modelling and I cannot consider them all. Therefore, in the remainder of this work, I limit myself to probabilistic circuits (PCs). A PC is a promising formalism that aims to combine the strong aspects of both probabilistic graphical models (PGMs) such as Bayesian networks and computation-oriented approaches such as ANNs. In chapter 2, I will further explain the PC formalism.

1.2 Research questions

The research questions are aimed to find evidence to either reject or accept the hypothesis formulated in the previous section:

- How can we compare the robustness of different models?
- How do generative models perform compared to discriminative models, w.r.t. corruptions in the input?
- How do generative models perform compared to discriminative models, w.r.t. adversarial robustness?
- In what conditions are generative models more robust than discriminative models?

1.3 *Outline*

In chapter 2, I present the state of the art in the field of probabilistic circuits, the formalism used to answer the research questions. In the next chapter, I will discuss what is meant by 'robustness', and how we can quantify robustness. I will also here discuss the field of adversarial attacks, which are strongly related to robustness.

In chapter 4, I explain the experimental setup used for answering the research questions. I will present the datasets and models used for running experiments, as well as the methods for assessing the robustness of the models. The results of the experiments are presented in chapter 5. Finally, in chapter 6, I will draw conclusions and I will discuss the limitations and difficulties faced during the project. I will touch upon future work that can be done to corroborate my findings.

2 Probabilistic circuits

In contemporary machine learning, deep learning is all the craze.[1] Whereas the field of artificial intelligence (AI) traditionally focussed on symbolic, model-based approaches, the availability of larger datasets and greater computing power contributed to a shift of focus toward function-based approaches.[2] This shift is exemplified by the advent of artificial neural networks (ANNs) and deep learning. The popularity of models such as ANNs is understandable, given their astounding performance in areas like computer vision. Probabilistic graphical models (PGMs) on the other hand, with Bayesian networks as the prime example, nowadays seem to enjoy more popularity in cognitive science than in engineering.

[1] See e.g. Goodfellow et al., 2016.

[2] A. Choi and Darwiche, 2018.

Recently attempts have been made to combine the best of both worlds. Probabilistic circuits (PCs) are *computational graphs*, like ANNs, but have a full probabilistic interpretation of the nodes, unlike ANNs. A PC encodes a probability distribution over the input $(\mathbf{X}, \mathbf{Y})$. Poon and Domingos (2011) did seminal work in the field of PCs with their introduction of sum–product networks (SPNs).[3] Poon and Domingos (2011, p. 1) motivated their work on SPNs by the observation that "models with multiple layers of hidden variables allow for efficient inference in a much larger class of distributions", and wondered how we could create deep architectures that take advantage of this, while providing tractable inference. SPNs are directed acyclic graphs (DAGs) that, under some structural constraints, guarantee tractable inference. This is a clear advantage over Bayesian networks, which are typically intractable. PCs combine the advantages of artificial neural networks, especially their deep structure, and Bayesian networks, viz. their probabilistic interpretation.

[3] In the remainder of this paper, I will treat PCs and SPNs as synonyms. Strictly speaking this is inaccurate, because SPNs are a subset of PCs. For the purposes of my thesis, however, the SPN formalism provides all the ingredients I need, and other formalisms such as arithmetic circuits (ACs) can be converted into an SPN representation.

For the purposes of comparing discriminative and generative classifiers, PCs are ideal for two reasons. First, they can be trained both discriminatively and generatively. This allows for a more controlled comparison, because all other factors—which can potentially confound the result—are kept constant. Secondly, they can be trained relatively fast and, depending on the chosen structure, they are structurally similar to ANNs. This enables a comparison with a baseline ANN model.

In this chapter, I will first introduce the PC formalism. I will introduce the main 'ingredients' and then give them an appropriate interpretation. Using this formalism, I will then explain how adding certain structural constraints to the graph leads to tractability guarantees. I will then shift my attention to structure of PCs. Finally, I will investigate how PCs can be trained.

2.1 *What is a probabilistic circuit?*

A probabilistic circuit (PC) over random variables $\mathbf{X}$ consists of a computational graph $\mathcal{G}$, also known as the *circuit structure*, and is parameterised by $\boldsymbol{\theta}$, the *circuit parameters*. The circuit structure $\mathcal{G} = (V, E)$ is a rooted DAG. Let $\mathsf{in}(v)$ denote the set of all input nodes for some node $v \in V$, that is, all nodes $\{u \in V : (u, v) \in E\}$. An example of a PC is drawn in figure 2.1.

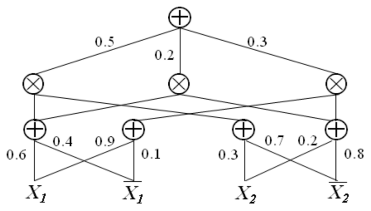


Figure 2.1: Example of a PC over two binary RVs X_1, X_2. The sum weights are printed along the edges. Note that arrowheads are not drawn but left implicit; we should understand all edges going upwards. From Poon and Domingos, 2011, p. 2.

There are three types of nodes:

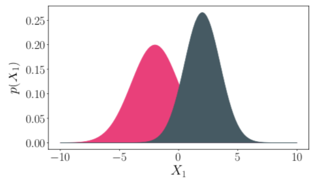
- A *leaf node* encodes some probability distribution. A leaf node takes as input some observed state and outputs the corresponding probability density function (PDF) for continuous distributions, or the probability mass function (PMS) for discrete distributions. A node $v \in V$ is a leaf node iff $\mathsf{in}(v) = \emptyset$. So the leaves of the circuit structure represent the random variables (RVs) $\mathbf{X}$.
- A *product node* $v \in V$ computes the product over its input nodes $\mathsf{in}(v)$. We can understand a product node as encoding a decomposition of a joint distribution into smaller factors. This is comparable to how Bayesian networks (BNs) capture the independencies between RVs.[4]
- A *sum node* $v \in V$ computes a weighted sum over its input nodes $\mathsf{in}(v)$. The weights are parameterised by $\theta_1, \dots, \theta_{|\mathsf{in}(v)|}$. The mixture density that results from this summation is more expressive than the original components. We can interpret a sum node as encoding a categorical latent variable (LV) Z.[5] The weights θ_i then represent the probability that Z takes value i. An example of the LV interpretation is shown in figure 2.2.

[4] Y. Choi et al., 2021.

[5] Peharz et al., 2017.

Furthermore let $\phi(v) \subseteq \mathbf{X}$ denote the *scope* of some node $v \in V$. For a leaf node, the scope corresponds to the RV associated with that leaf node. For a non-leaf node, the scope is defined as the union of the scopes of its input nodes: $\phi(v) = \cup_{u \in \mathsf{in}(v)} \phi(u)$.

The root node of a PC is the node $v_r \in V$ that has no outgoing edges. It captures the joint distribution over all RVs in $\mathbf{X}$. For the root, we always have $\phi(v_r) = \mathbf{X}$. Note that this definition of a PC lays down a *recursive* scheme. We can interpret each input node of the root node as a sub-PC, rooted in that node. The base case of the recursion is



$$\begin{aligned} p(X_1) = \theta_1 p_1(X_1) + \theta_2 p_2(X_1) = \\ = p(Z=1)p(X_1 \mid Z=1) + \\ p(Z=2)p(X_1 \mid Z=2) \end{aligned}$$

Figure 2.2: The latent variable interpretation of a sum node. The mixture model marginalises out the LV Z, which can take values 1 or 2. From Y. Choi et al., 2021, p. 14.

always a leaf node, because leaf nodes are the only nodes that have no children.

With these simple ingredients, we can build surprisingly powerful and expressive models. PCs may seem similar to PGMs, but there are some important differences. While both are probabilistic models that capture some joint distribution, the semantics are entirely different.[6] A node in a PGM represents a random variable whereas a node in a PC represents a unit of computation. An edge in a PGM stands for a dependency whereas an edge in a PC stands for the order of execution. The method of inference is different, too. PGMs use conditioning, elimination, and/or message passing for inference. PCs on the other hand make use of a feedforward pass and, depending on the query, a backward pass.

[6] Y. Choi et al., 2021, p. 32.

It seems that PCs are more similar to ANNs than to PGMs, as both PCs and ANNs are computational graphs. This intuition is correct. Like ANNs, PCs can also fit non-linear functions. In ANNs, the non-linearity stems from non-linear activation functions; in PCs the non-linearity stems from leaf distributions that potentially have non-linear behaviour, and from the product nodes. Martens and Medabalimi (2015) show that PCs are universal approximators of distributions if we allow the size of the model to be exponential to the dimensionality of the input.[7]

[7] In order to capture the probability distribution *efficiently*, however, the marginals and normalisers of the density function must be tractable. This is not the case for all distributions. Hence there is a class of distributions that can be tractably represented by a neural net, but not by a PC (Martens and Medabalimi, 2015).

But there are important differences with ANNs too. Most notably, ANNs have no probabilistic interpretation whereas PCs do. PCs also typically have a sparser computational graph than ANNs. This makes efficient tensorised GPU computation more difficult.[8] Finally, the set of available computation units is heavily constrained. There are no activation functions for the internal nodes, and we cannot perform operations such as max-pooling. Convolution in PCs is possible but non-trivial, because of some structural constraints we usually enforce on the circuit structure.[9]

[8] Work is being done to close this gap, however. Einsum Networks for instance, by Peharz et al. (2020) make computations on PCs more efficient. See § 2.1.3.

[9] Examples of convolution in PCs can be found in Butz et al., 2019 and van de Wolfshaar and Pronobis, 2020.See § 2.2.

2.1.1 *Query classes*

In order to assess the tractability of PCs and the structural constraints required to attain tractability guarantees, it is useful to distinguish between different *query classes*. A query is a 'question' we ask to a probabilistic model.[10] Not all queries are alike. We can distinguish different *query classes*, based on two questions. First, what do we already know? Do we have a value for all, some, or none of the RVs in $\mathbf{X}$? Secondly, what do we want to know? Do we want to find a probability of a certain state, or do we want to find the most probable state given a partial value assignment to the RVs in $\mathbf{X}$?

[10] Y. Choi et al., 2021, p. 6.

In some cases, we know the value assignment for all RVs, and want to find the probability of this state. This is the *complete evidence* class, defined as follows.

Definition 2.1 (Complete evidence query class)**.** The class of complete evidence queries consists of all queries that compute $P(\mathbf{X} = \mathbf{x})$, where P is a joint probability distribution over RVs $\mathbf{X}$, and $\mathbf{x} \in \mathsf{val}(\mathbf{X})$ is a complete state. [11]

[11] Adapted from Definition 1 in Y. Choi et al., 2021, p. 7.

Typically, however, we do *not* know the complete state. If we are interested in the probability of a partial state, we are dealing with a *marginal query*, defined as follows:

Definition 2.2 (Marginal query class)**.** For a joint distribution $P(\mathbf{X})$ over RVs $\mathbf{X}$, the class of marginal queries computes the following:

$$P(\mathbf{E}=\mathbf{e},\mathbf{Z}) = \int_{\mathsf{val}(\mathbf{Z})} P(\mathbf{e},\mathbf{Z})\mathrm{d}\mathbf{Z}$$

where $\mathbf{e} \in \mathsf{val}(\mathbf{E})$ is the observed evidence for some subset of RVs, $\mathbf{E} \subseteq \mathbf{X}$, and $\mathbf{Z} = \mathbf{X} \setminus \mathbf{E}$. Note that the integration is over a Cartesian product of $|\mathbf{Z}|$ intervals.[12]

[12] Adapted from Definition 11 in Y. Choi et al., 2021, p. 20.

Finally, we can consider queries where we are not interested in the *probability* of an event, but rather in the *state* that is most likely, given the partial evidence. This is known as *inference to the best explanation (IBE)*. In the context of PGMs, IBE can be formalised as the maximum a posteriori (MAP) problem. The corresponding query class is defined as follows.

Definition 2.3 (Maximal a posteriori query class)**.** For a joint distribution $P(\mathbf{X})$ over RVs $\mathbf{X}$, the class of MAP queries computes the following:

$$\arg\max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} P(\mathbf{Q}=\mathbf{q} \mid \mathbf{E}=\mathbf{e},\mathbf{Z}) = \arg\max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} \int_{\mathsf{val}(\mathbf{Z})} P(\mathbf{q},\mathbf{e},\mathbf{Z})\mathrm{d}\mathbf{Z}$$

where $\mathbf{Q}$, $\mathbf{E}$, and $\mathbf{Z}$ form a partitioning of RVs $\mathbf{X}$, and $\mathbf{e} \in \mathsf{val}(\mathbf{E})$ is the observed evidence.[13]

[13] Adapted from Definition 37 in Y. Choi et al., 2021, p. 46. Note the difference in terminology: the authors call this class marginal MAP instead of MAP. What the authors call 'MAP', I call 'MPE'.

The right-hand side follows by noting that the arg max operation is not affected by the normalisation constant $P(\mathbf{E} = \mathbf{e},\mathbf{Z})$ of the conditional probability. We are not interested in the values of $\mathbf{Z}$, so we marginalise these out. Again, the integral sign actually stands for a $|\mathbf{Z}|$-dimensional integral.

In the special case where $\mathbf{Z} = \emptyset$, the MAP problem is known as *most probable explanation (MPE)*. The corresponding query class is as follows:

Definition 2.4 (Most probable explanation query class)**.** For a joint distribution $P(\mathbf{X})$ over RVs $\mathbf{X}$, the class of MPE queries computes the following:

$$\arg\max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} P(\mathbf{Q}=\mathbf{q} \mid \mathbf{E}=\mathbf{e}) = \arg\max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} P(\mathbf{q},\mathbf{e})$$

where $\mathbf{Q}$ and $\mathbf{E}$ form a partitioning of RVs $\mathbf{X}$, and $\mathbf{e} \in \mathsf{val}(\mathbf{E})$ is the observed evidence.[14]

[14] Adapted from Definition 26 in Y. Choi et al., 2021, p. 37.

Naturally, the MPE query class is a subset of the MAP query class.

2.1.2 Structural constraints and tractability

Our goal is that the PC correctly computes the probability of evidence. A PC is 'valid' if it satisfies this goal. Formally:

Definition 2.5 (Validity). A PC $\mathcal{S}$ is valid iff it correctly computes the unnormalised probability of evidence Φ, that is, $\mathcal{S}(\mathbf{e}) = \Phi(\mathbf{e})$ for all evidence $\mathbf{e}$.[15]

[15] Adapted from Definition 2 in Poon and Domingos, 2011, p. 3.

Note that the probability distribution Φ is unnormalised. This means that there is some constant Z such that $\sum_{\mathbf{x}} \frac{\Phi(\mathbf{x})}{Z} = 1$.[16] It is possible to normalise the PC for each node. If we do so, the PC is *locally normalised*. The advantage of a locally normalised PC is in its interpretation: we can then understand the PC as computing a normalised probability distribution. It is not immediately obvious that a locally normalised PC has the same expressive power as an unnormalised PC. Peharz et al. (2015) show that normalising a PC does not affect the modelled distribution. We can thus safely assume locally normalised sum weights without hurting expressiveness. In the remainder of this report, I assume the PCs are normalised.

[16] Van de Wolfshaar, 2019.

In a PC, evaluating the joint probability for a complete state $\mathbf{x} \in \mathsf{val}(\mathbf{X})$ can be done in linear time w.r.t. model size. After all, we only need to execute the computation graph and read out the value in the root. This is no different from the tractability of BNs for complete evidence.

Things get interesting, however, if we consider other query classes, such as marginal probability. As can be seen in definition 2.2, computing the marginal probability involves a multiple integral. Computing a multiple integral over some function is often a very expensive operation. Generally this problem is #P-hard.[17] But the ingenuity of PCs lies in the imposing of *structural constraints* on the circuit structure, so that we can answer marginal queries in linear time, *while guaranteeing validity*.

[17] Baldoni et al., 2010.

First we note that for a fully factorised probabilistic model, we can break the multivariate integral into a product of simpler integrals. This only works however if the children of a product node depend on disjoint sets of variables. Otherwise, the factors are not independent and the decomposition would be invalid. The following equation illustrates the decomposition of an integral into a product of integrals, for some sets of random variables $\mathbf{X}$ and $\mathbf{Y}$:

$$\begin{aligned}\int_{\mathsf{val}(\mathbf{X})}\int_{\mathsf{val}(\mathbf{Y})} P(\mathbf{X},\mathbf{Y})\mathrm{d}\mathbf{X}\mathrm{d}\mathbf{Y} &\overset{*}{=} \int_{\mathsf{val}(\mathbf{X})}\int_{\mathsf{val}(\mathbf{Y})} P(\mathbf{X})P(\mathbf{Y})\mathrm{d}\mathbf{X}\mathrm{d}\mathbf{Y} \\ &= \int_{\mathsf{val}(\mathbf{X})} P(\mathbf{X})\mathrm{d}\mathbf{X}\int_{\mathsf{val}(\mathbf{Y})} P(\mathbf{Y})\mathrm{d}\mathbf{Y},\end{aligned}$$

where the star indicates that this equality only holds if $\mathbf{X}$ and $\mathbf{Y}$ are conditionally independent given the latent variables in the PC. In a PC, we can apply this trick recursively, as long as the scopes of the children of a product node do not overlap. This brings us to the first structural constraint:

Definition 2.6 (Decomposability)**.** A product node v is *decomposable* if the scopes of its input units do not share variables: $\phi(u_i) \cap \phi(u_j) = \emptyset, \forall u_i, u_j \in \mathsf{in}(v), i \neq j$. A PC is decomposable if all of its product nodes are decomposable.[18]

[18] Adapted from Definition 15 in Y. Choi et al., 2021, p. 24

Next, we should note that a mixture model, as encoded by a sum node, only makes sense if the components range over the same set of random variables. If they are not, summing them together would lead to mistakes. For instance, it makes no sense to add an RV representing people's height with an RV representing people's weight. If the set of random variables is equal, the integral of a sum equals the sum of the integrals of the components. If we write this out, using the definition of a sum node (w_i represents the weight for component i), we get the following:

$$\begin{aligned}\int_{\mathsf{val}(\mathbf{X})} P(\mathbf{X})\mathrm{d}\mathbf{X} &= \int_{\mathsf{val}(\mathbf{X})} \sum_i w_i P_i(\mathbf{X})\mathrm{d}\mathbf{X} \\ &= \sum_i w_i \int_{\mathsf{val}(\mathbf{X})} P_i(\mathbf{X})\mathrm{d}\mathbf{X}\end{aligned}$$

This observation leads to the second structural constraint:

Definition 2.7 (Smoothness)**.** A sum node v is *smooth* if its inputs all have identical scopes: $\phi(u) = \phi(v), \forall u \in \mathsf{in}(v)$. A PC is smooth if all of its sum nodes are smooth.[19]

[19] Adapted from Definition 16 in Y. Choi et al., 2021, p. 25. Some authors, including Poon and Domingos (2011), call this property 'completeness' instead.

If a PC is smooth and decomposable, this means that we can always 'push down' the integrals toward the leaf nodes. The integral of any valid normalised PDF or PMF is 1. From this it follows that we can perform marginal queries on PCs in time linear to the model size. We set the output of all leaf nodes for RVs in $\mathbf{Z}$ to 1, and then execute the computational graph. The result is a valid marginal probability for the given query. To summarise, we can formulate theorem 2.1.

Theorem 2.1. *A PC is* valid *if it is smooth and decomposable.*[20]

[20] Formal proof given in Poon and Domingos, 2011, p. 3
Note that the constraints presented here differ from those in Poon and Domingos (2011). The authors originally specified smoothness ('completeness') and *consistency* as constraints. Consistency is a strictly weaker requirement than decomposability, but Peharz et al. (2015) show that every smooth and consistent SPN can be transformed into a complete and decomposable SPN in polynomial time.

One can wonder if MAP inference in PCs is tractable too, if the PC satisfies the smoothness and decomposability constraints. On first sight, it seems that MPE queries, at least, can be computed in linear time. In their original paper, Poon and Domingos (2011, p. 8) indeed state that MPE inference can be achieved exactly and in linear time. The authors' approach works by pushing the maximisation problem down to the leaves, just like in answering marginal queries. More precisely, we replace the sum operations with max operations, and replace the distributions of the leaf nodes with their modes. In a bottom-up evaluation, we compute $\max_{\mathbf{q} \in \mathsf{val}(\mathbf{Q})} P(\mathbf{q}, \mathbf{e})$, that is, the probability of the MPE state. With some additional backtracing, for instance by means of the Viterbi algorithm, we can recover the state of the evidence variables too.[21]

[21] Peharz et al., 2017.

MPE inference in linear time! This almost sounds too good to be true. And as it turns out, it is.[22] The problem boils down to this: max

[22] Peharz et al., 2014; Peharz et al., 2017.

and $\sum$ do not commute. With the described algorithm, we do the following in the sum nodes:

$$\begin{aligned}
\max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} P(\mathbf{q},\mathbf{e}) &= \max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} P(\mathbf{q},\mathbf{e}) \\
&= \max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} \sum_i w_i P_i(\mathbf{q},\mathbf{e}) \\
&\overset{?}{=} \max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} \max_i w_i P_i(\mathbf{q},\mathbf{e}) \\
&= \max_i w_i \max_{\mathbf{q}\in\mathsf{val}(\mathbf{Q})} P_i(\mathbf{q},\mathbf{e})
\end{aligned}$$

The equality on the third line, marked with the question mark, only holds if at most one of the elements of the sum is non-zero. The smoothness constraint of a PC does not guarantee this. We need an additional constraint, next to smoothness, in order to have tractable MPE inference, called determinism:

Definition 2.8 (Determinism)**.** A sum node v is *deterministic* if, for any fully-instantiated input, the output of at most one of its children is nonzero. A circuit is deterministic if all of its sum nodes are deterministic.[23]

[23] Taken from Definition 30 in Y. Choi et al., 2021, p. 40.

To summarise, the MPE algorithm in Poon and Domingos (2011) is correct, but only if the SPN is deterministic.[24]

[24] Peharz et al., 2017.

Finally, I would like to quickly consider (partial) MAP inference in PCs, although not directly relevant for the classification task. MAP is inherently harder than MPE.[25] The structural constraints that allow for tractable MPE are insufficient for tractable MAP.[26] It *is* possible to formulate even stronger structural constraints, as is shown in Y. Choi et al. (2021), but these constraints introduce a dependency on query set $\mathbf{Q}$. This dependency hurts the expressiveness of the model.

[25] Kwisthout, 2011.

[26] Y. Choi et al., 2021.

2.1.3 Tensorised PCs

As noted in § 2.1, PCs typically have a sparser computational graph than ANNs. For instance, the distance from leaf to root can differ greatly across leaf nodes. This makes GPU-optimised computations with PCs challenging.

In 2020, Peharz et al. (2020) introduced a new implementation for SPNs, called *Einsum networks (EiNets)*. In this novel implementation, nodes are vectorised into vectors of length K. This leads to a denser (hence more efficient) layout for the PC. More precisely, EiNets work by redefining the leaf nodes as *vectors* of K densities over the scope of the node, instead of just a single density. Sum nodes and product are also redefined accordingly; sum nodes become vectors of K weighted sums and product nodes become outer products. These redefinitions are illustrated in figure 2.3. The arithmetic operations performed in the network can be combined into a single, monolithic einsum-operation, which makes the computations faster and more memory-efficient. The

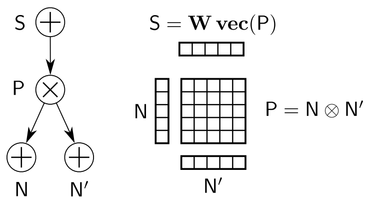


Figure 2.3: Basic einsum operation in EiNets: A sum node S, with a single child P, which itself has 2 children. All nodes are vectorised. In this illustration, $K = 5$. From Peharz et al., 2020, p. 4.

authors report speedups of up to two orders of magnitude compared to previous implementations.

Independently from Peharz et al., van de Wolfshaar and Pronobis (2020) have introduced `libspn-keras`, a layer-centred implementation of PCs.[27] This implementation builds upon the open-source Keras library for creating ANNs. In `libspn-keras`, PCs are built up layer-by-layer. By using padding, a tree structure is obtained that is fully balanced and homogenous. This enables GPU parallellisation. The authors have created a benchmark, comparing `libspn-keras` with EiNets.[28] They claim that `libspn-keras` is more flexible than EiNets, and twice as fast.

[27] Source code available at https://github.com/pronobis/libspn-keras.

[28] See https://git.io/JGoBB.

2.2 *Probabilistic circuit structures*

In PCs, different circuit structures can be used. Three main approaches can be distinguished. First, there are *predefined structures*. These are based on the shape of the input data, and possibly some hyperparameters. Secondly, there are *random structures*. These are comparable to predefined structures, but their structure is not deterministic; at least part of the structure is built randomly. Again, the structure depends on the shape of the data and possibly some hyperparameters. Thirdly, there are *structures learned from data*. These are distinct from the other two approaches, in that the structure is based not only on the shape of the input data, but also on the input data itself.

In the remainder of this section, I give examples for each of the approaches.

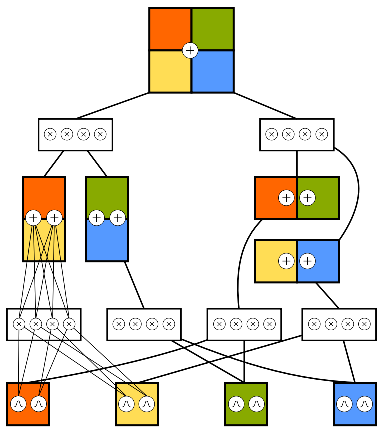


Figure 2.4: Poon–Domingos structure. From From Vergari et al., 2020, p. 116.

2.2.1 *Predefined structures*

In their original paper on SPNs, Poon and Domingos (2011) used a predefined structure, optimised for image data. It works by recursively decomposing the image into sub-rectangles using axis-aligned splits, displaced by a certain step-size Δ. Figure 2.4 illustrates the process. Horizontal splits are alternated by vertical splits, until we are left with single pixels. Note that each rectangular subregion is decomposed in *all* possible ways.[29] This leads to a PC with very high depth; for a $d \times d$ image, the corresponding PC has $2(d-1)$ layers.

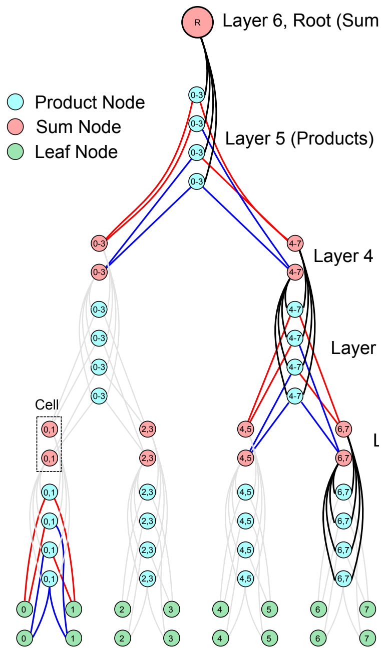


Figure 2.5: Example of a 'vanilla' 1-dimensional convolutional SPN. This architecture does not allow for spatially overlapping patches. Some connections are highlighted to improve readability. Adapted from van de Wolfshaar and Pronobis, 2020, p. 5.

[29] Poon and Domingos, 2011, p. 7.

In ANNs, it is common to use *convolution*. Convolutional layers in an ANN make use of a filter that slides over the input. This way, parameters can be shared, leading to a vast reduction of the number of parameters needed in the network. Convolutional neural networks (CNNs) are very successful in image recognition tasks. One can wonder if it is possible to build a convolutional structure in PCs too. The first work on convolutional PCs was done by Sharir et al. (2018) and Butz et al. (2019). They were faced with the following challenge: how to implement a convolutional structure, while maintaining validity of the PC? In particular, satisfying the decomposability requirement (see definition 2.6) is tricky, because the scopes of the input of a

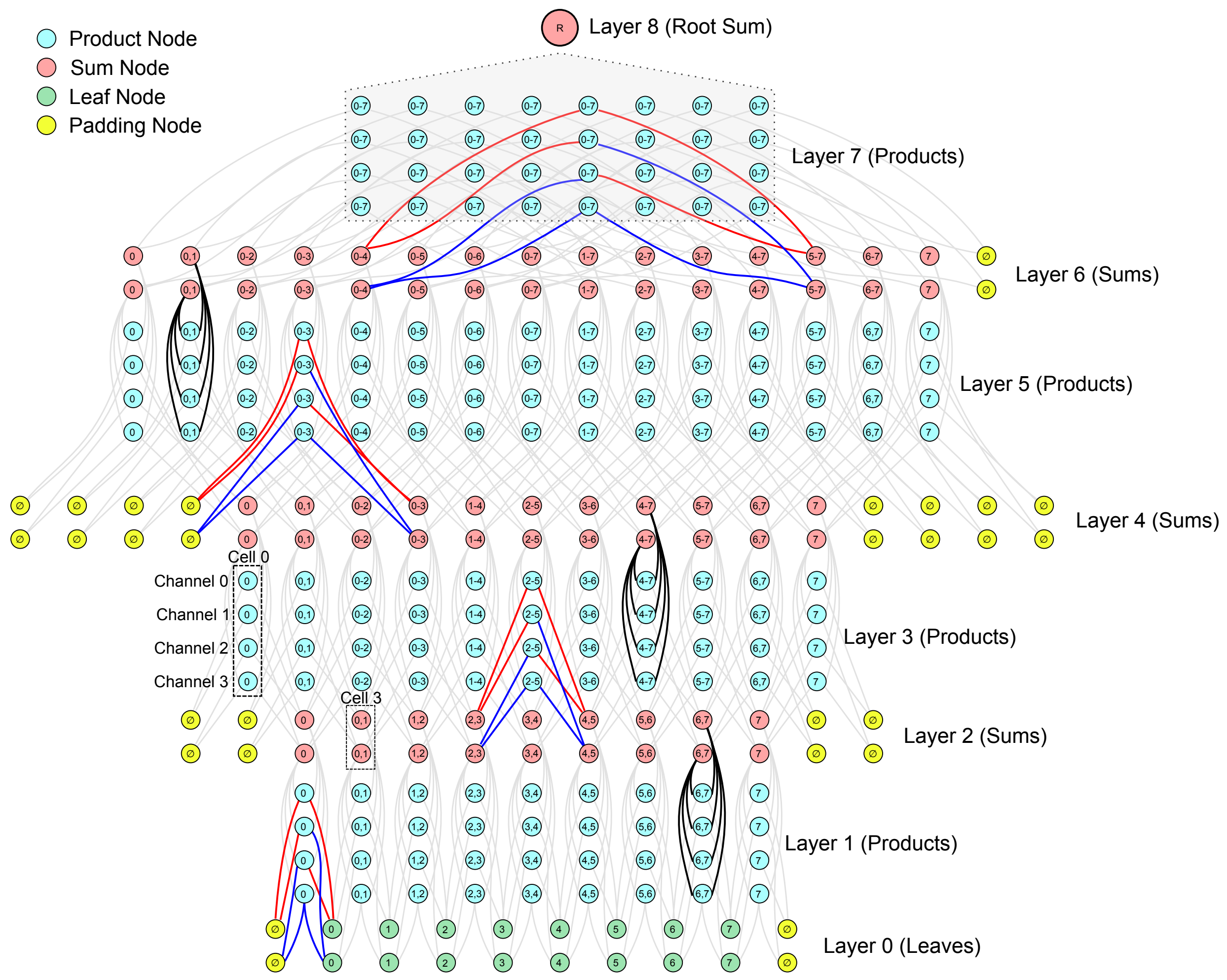


Figure 2.6: Example of a 1-dimensional DGC-SPN. Some connections are highlighted to improve readability. Layer 0 contains leaf distributions, where each channel corresponds to an indicator for discrete variables or a distribution component (e.g. Gaussian) for continuous variables. Every product layer doubles the dilation rate, starting at the rate of 1. The scopes are indicated by the numbers within each node. Padding nodes have a fixed probability of 1 (or 0 in log-space). The nodes of a single cell share the same scope. All children of the root node *R* have a scope that contains all input variables. As opposed to the vanilla convolutional SPN in figure 2.5, this architecture does allow for spatially overlapping patches. From van de Wolfshaar and Pronobis, 2020, p. 6.

product node may not overlap. The sliding filter in a convolutional layer typically does have overlap in scope, unless the stride is at least as large as the filter. Sharir et al. (2018) and Butz et al. (2019) solved this issue by using non-overlapping image patches in the product layers.[30] See figure 2.5 for an example. This solution however leads to a loss of feature resolution. Furthermore, many spatial relations are lost in the input, because the convolutional filter may never be correctly positioned to detect them.

Van de Wolfshaar and Pronobis (2020) address this lack of generalisability in their proposed structure, called *deep generalised convolutional sum–product networks (DGC-SPNs)*. DGC-SPNs do make use of overlapping image patches in the product layers. Decomposability is satisfied by using exponentially growing dilation rates for the product nodes. This way, a compromise is reached where the convolutional filters use partially overlapping input, but the decomposability constraint is maintained. See figure 2.6 for an example.

[30] van de Wolfshaar and Pronobis, 2020

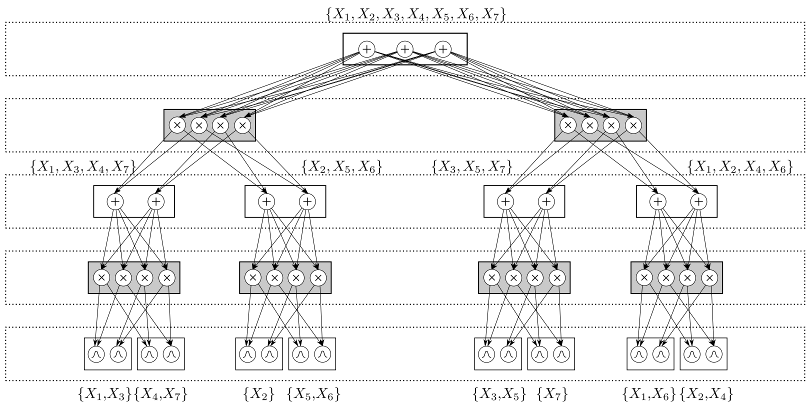


Figure 2.7: Example RAT-SPN over 7 RVs $\{X_1, \ldots, X_7\}$, using parameters $C = 3, D = 2, R = 2, S = 2$, and $I = 2$. From Peharz et al., 2018, p. 4.

2.2.2 *Random structures*

Peharz et al. (2018) have introduced *random tensorised sum–product networks (RAT-SPNs)*. RAT-SPNs are designed for classification problems. In RAT-SPNs, the random variables in the input set $\mathbf{X}$ are recursively split into a 2-partition D times. The resulting hierarchy of partitions, which is a DAG, is then used to construct a corresponding SPN. The root of the hierarchy is equipped with C sum nodes, one for each class. The leaves of the hierarchy are equipped with I leaf nodes. Finally, the remaining partitions in the hierarchy are equipped with S sum nodes. The sum nodes are tied together by using cartesian products of the nodes in the connecting partitions in the hierarchy. This entire procedure is repeated R times, each with a different random partitioning of the random variables. Figure 2.7 shows an example RAT-SPN. Since RAT-SPNs offer 5 hyperparameter at our disposal, the number of parameters in the model can be precisely controlled.

2.2.3 *Structures learned from data*

For completeness, I would like to mention that it is also possible to learn the structure of a PC based on the data. This approach was introduced by Gens and Domingos (2013). Their method, called *LearnSPN*, works by first clustering the instances in the input data $\mathbf{X}$. Each cluster is assigned a sum node. Then, the algorithm search for independent groups of random variables. Within the identified clusters, the RVs are split in independent groups, which are given a product node. The procedure is repeated recursively until splitting is no longer possible; then a leaf is created for the single variable.

I will not pursue the topic of structure learning further in this thesis.

2.3 *Classification with probabilistic circuits*

There are two main ways of solving the classification problem with PCs. The first option is the approach used in RAT-SPNs, as explained in the previous section. The root node is a sum node with K children,

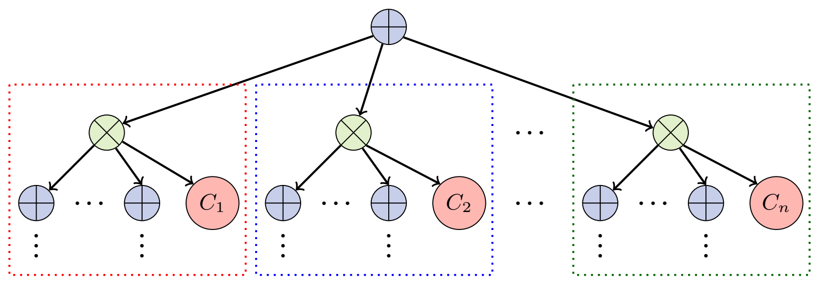


Figure 2.8: Illustration of a class-selective SPN. In the graph, C_n is a leaf node applying the indicator function. From Correia and de Campos, 2019, p. 5.

one for each of the K classes. The model should then be configured to output a vector of size K with the probabilities for each of the classes—one value for each child. This output is fed to a loss function that is used during training. If we use this approach, we can interpret the sum weights of the root sum as the prior probabilities $P(Y = k)$ of an instance belonging to class $k, k = 1, \ldots, K$. Each child of the root sum computes the likelihoods $P(\mathbf{X} \mid Y = k)$ of the instance belonging to class k. Note that k is also the index of the child. The outputs of the root sum are the joint probabilities

$$P(\mathbf{X}, Y = k) = \underbrace{P(\mathbf{X} \mid Y = k)}_{\substack{\text{likelihood}\\ \text{(inputs)}}} \underbrace{P(Y = k)}_{\substack{\text{prior}\\ \text{(sum weights)}}},$$

for each k. Finally, to get the conditional probabilities, the joint probabilities are normalised:

$$P(Y = k \mid \mathbf{X}) = \frac{P(\mathbf{X}, Y = k)}{P(\mathbf{X})} = \frac{P(\mathbf{X}, Y = k)}{\sum_{i=1}^{K} P(\mathbf{X}, Y = i)}$$

The other way is by constructing a *class-selective SPN*, as introduced by Correia and de Campos (2019). Here the model outputs a single joint probability $P(\mathbf{X}, Y = k)$. The root sum has K product nodes as its children. Each of these product nodes has a leaf node attached to it, applying an indicator function: 1 if $Y = k$ and zero otherwise. Using this approach, we are in fact training a different SPN per class, possibly with non-identical structures. Figure 2.8 illustrates a class-selective SPN.

2.3.1 *Tractability of classification*

Is classification in PCs tractable? In order to answer this question, we need to know which query class classification belongs to.[31]

[31] The different query classes are discussed in § 2.1.1.

Most generally, we can consider classification as a special case of MPE, as defined in definition 2.4. $\mathbf{Q}$ then is the set of indicator variables for each of the K classes. As discussed in § 2.1.2, that would imply that the PC needs to satisfy the determinism constraint. However, the structures in § 2.2 are valid but not (necessarily) deterministic.

Does that mean we cannot do tractable classification with these structures? Not quite. Recall from the beginning of this section how PCs can be used for classification. In both ways discussed, we can understand the model as getting the probabilities for the K classes by running K *marginal queries*. By comparing these scores, a verdict of most likely class can be made. Typically in classification, the number of classes is very small compared to model size. Therefore, classification in PCs can still be done in linear time w.r.t. model size, even though the general classification problem is an instance of the MPE problem.[32]

[32] This also means that if K does grow very large, it may be interesting to look at MAP/MPE algorithms for solving the problem. Since this is not the case for the datasets used in this thesis, I set aside discussion on these algorithms.

2.4 *Training a probabilistic circuit*

As stated in the chapter introduction, PCs can be trained both discriminatively and generatively, making them very suitable for answering the formulated research questions. Whether the model is trained discriminatively or generatively depends on the loss function chosen. A loss function $L : \mathbb{R}^K \times \mathbb{Z}^+ \to \mathbb{R}^+$ maps the model predictions and true label a non-negative number that is minimised during optimisation.

For generative training, negative log–likelihood (NLL) loss is used:

Definition 2.9 (Negative log likelihood loss)**.** For a K-sized vector $\hat{\mathbf{y}}$ of predicted (joint) probabilities, and true class y, the NLL loss computes

$$L_{\text{NLL}}(\hat{\mathbf{y}}, y) = -\sum_{k=1}^{K} \log(\hat{y}_k)\mathbb{1}\{y = k\},$$

where $\mathbb{1}\{A\}$ is the indicator function which takes a value of 1 if the logical condition A is true and zero otherwise. That is, the loss function selects the predicted probability of the correct class and returns the negative log.

The log transformation is included to penalise predictions that are close to zero and reward those that are close to one. For discriminative training, cross-entropy (CE) loss is used:

Definition 2.10 (Cross-entropy loss)**.** For a K-sized vector $\hat{\mathbf{y}}$ of predicted (joint) probabilities, and true class y, the cross-entropy loss computes

$$L_{\text{CE}}(\hat{\mathbf{y}}, y) = L_{\text{NLL}}(\sigma(\hat{\mathbf{y}}), y),$$

where $\sigma : \mathbb{R}^K \to [0,1]^K$ is the softmax function:

$$\sigma(\mathbf{z}) = \frac{1}{\sum_{k=1}^{K} e^{z_k}} e^{\mathbf{z}}, \text{ for } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K.$$

In other words, CE loss is identical to NLL loss, except that the input is normalised first, so that the sum of the values in $\hat{\mathbf{y}}$ is 1.

The addition of the softmax function in the CE loss is crucial, because it entails that the loss function does not maximise the actual joint probability $P(\mathbf{x}, y)$ that the model outputs. Instead, a model trained with CE loss maximises the conditional probability $P(y \mid \mathbf{x})$, leading to a discriminative model. In discriminative training, there is no incentive for $P(y)$ or $P(\mathbf{x} \mid y)$ to be accurate in themselves.

For optimising the model, different strategies are available depending on the chosen loss function. Stochastic gradient descent (SGD) is always available whereas expectation maximisation (EM) is only available for models trained generatively. This is because EM algorithms need to compute the expectations of conditional distributions. These expectations can only be computed accurately if $P(y)$ and $P(\mathbf{x} \mid y)$ are accurate, which is only the case in models trained generatively. EM can be used for models that estimate the likelihood function directly, such as PCs, but also variational autoencoders (VAEs), flow-based generative models, and BNs. Generative adversarial networks (GANs) on the other hand, while trained generatively, cannot be trained with EM, because they do not represent the likelihood function explicitly.

3 Robustness

The goal of this thesis is to investigate the robustness of generative versus discriminative classifiers. In the previous chapter, I introduced the PC modelling formalism used for running the experiments. I will now turn my attention to what it means for a model to be robust. The importance of robustness was illustrated in the introduction. In this chapter, I only consider notions of robustness with respect to the classification problem (§ 1.1.1).

On the most general level, I consider two kinds of robustness: *non-adversarial* and *adversarial*. With non-adversarial robustness, I mean robustness to perturbations in the input, without ill intent. I call these corruptions of the input data. A typical example is the presence of random noise in the data. I address this topic in § 3.1 From § 3.2 onwards, I turn my attention to adversarial robustness, in which an adversary deliberately alters the input, typically in a way that is (nearly) imperceptible to humans.

3.1 Robustness to corruptions

A straightforward method for assessing the robustness of a model, is by corrupting the test input and assessing the model performance. This way the model can be tested against known perturbations that might occur. The perturbations are not necessarily imperceptible, but might arise in the data collection phase. Think for instance of noisy sensors that are used as the input to a model.

Mu and Gilmer (2019) have introduced MNIST-C, a corrupted version of the original MNIST dataset by Lecun et al. (1998). The fifteen corruptions available in MNIST-C, shown in figure 3.1, can be used to benchmark out-of-distribution (OOD) robustness. The authors show that several approaches to defend models against adversarial attacks in fact degrade performance on MNIST-C. This is a strong argument for considering performance on corrupted datasets when assessing model robustness.

An even simpler way of assessing robustness is by introducing random noise to the test dataset. In this approach, the amount of noise can be precisely controlled, so that the effect of different noise levels on model performance can be plotted.

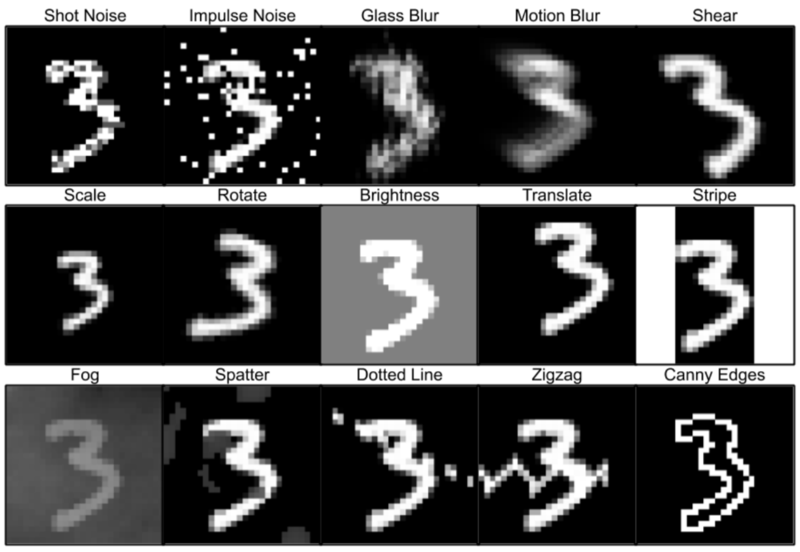


Figure 3.1: Randomly sampled applications of all 15 corruptions comprising MNIST-C. From Mu and Gilmer, 2019, p. 2.

3.2 *Adversarial attacks*

In an attack, an adversary tries to change the output of a model for his benefit. This is distinct from robustness to corruptions, where there is no a priori goal of changing the output. Before discussing different types of attacks, it is useful to make the following distinctions: (1) What is the goal of the adversary? And (2) What are the capabilities of the adversary?

Regarding (1) we can identify the following attack goals: [1]

- *Confidence reduction*: reducing the confidence of prediction for the target model, e.g., an adversarial sample of a '5' digit is predicted with lower confidence.
- *Non-targeted misclassification*: changing the classification to any class distinct from the true class, e.g., an adversarial sample of a '5' digit is predicted to be some other digit.
- *Targeted misclassification*: changing the classification to a specific target class, e.g., any adversarial sample is classified as a '7' (alternatively, we could even have source–target mappings).

[1] Based on Qiu et al., 2019

Regarding (2) we can identify the following capabilities:

- *Testing stage*: the adversary can manipulate either training data or the learning algorithms. This type of attack I will not consider further.
- *Training stage*: the adversary cannot manipulate training stage, but can manipulate testing stage. There are two main types: *white-box* and *black-box* attacks.[2] In a white-box attack, the adversary has full knowledge about the target model h_θ. Typically, the gradients are exploited. In a black-box model, the adversary has no knowledge about h_θ, but can manipulate input–output pairs. In the *non-adaptive* setting, a substitute model h'_θ is trained. In the *adaptive* setting, the adversary can only access h_θ as an oracle and knows nothing else about the model.[3] He can however change the inputs to observe

[2] In § 3.3, I provide examples for both white-box and black-box attacks.

[3] Qiu et al., 2019, p. 7.

changes in outputs. Within the adaptive setting, we can further distinguish between *soft-label* attacks, in which the adversary can obtain probability outputs for the model for all classes, and *hard-label* attacks, in which the adversary only knows the final decision (e.g. top-1 predicted class).[4]

[4] Cheng et al., 2018, pp. 1–2.

Black-box attacks are harder for the adversary than white-box attacks, because he has access to less information. They however have the great advantage that they are model-agnostic. This means that the same attack strategy can be used on different types of models (of course, the performance of this attack may differ across models).

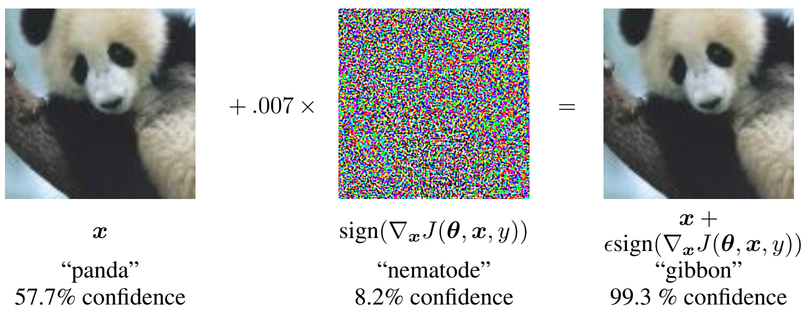

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 3.2: A demonstration of a generated adversarial example. The right image looks like a panda to us, but is classified as a gibbon. From Goodfellow et al., 2015, p. 3.

What does an attack look like? A domain in which attacks are easy to explain, is image classification. Figure 3.2 shows an example in which an attack is employed. In this example by Goodfellow et al. (2015), we see an example of a (white-box) attack on an image classifier. By introducing some noise—for visualisation purposes, the noise in the middle image has been greatly amplified—we can find an image that still looks like a panda to us, but is classified by the machine as a gibbon, with very high confidence. Naturally the possibility of such adversarial examples is undesirable: the semantic content of the adversarial example is unchanged (we still see a panda), so the computer should be able to ignore this noise.

3.2.1 *Formalisation of adversarial robustness*

Every adversarial attack and defence strategy is a method for (approximately) solving the following problem:[5]

[5] Kolter and Madry, 2018.

$$\min_{\boldsymbol{\theta}} \hat{R}_{\text{adv}}(h_{\boldsymbol{\theta}}, D_{\text{train}}) \equiv \min_{\boldsymbol{\theta}} \frac{1}{|D_{\text{train}}|} \sum_{(\mathbf{x},y)\in D_{\text{train}}} \max_{\delta\in\Delta(\mathbf{x})} L(h_{\boldsymbol{\theta}}(\mathbf{x}+\delta)), y) \tag{3.1}$$

Where

- $h_{\boldsymbol{\theta}} : \mathcal{X} \to \mathbb{R}^K$ is the hypothesis function, i.e. *the model* (K represents the number of classes being predicted).
- $\boldsymbol{\theta}$ is a vector representing the parameters of the model.
- $\hat{R}_{\text{adv}}$ is the empirical adversarial risk.
- D_{train} is some training set consisting of $(\mathbf{x}, y)$ tuples, for input $\mathbf{x} \in \mathcal{X}$ and true class $y \in \mathbb{Z}$.

- δ is the perturbation applied to create the adversarial example $\tilde{\mathbf{x}} = \mathbf{x} + \delta$.
- $\Delta(\mathbf{x})$ is the set of allowable perturbations. Note that Δ explicitly depends on $\mathbf{x}$, i.e., the allowable perturbations may differ depending on the input. A trivial situation where we can see this dependence, is when we need to ensure the resulting perturbed input $\tilde{\mathbf{x}} = \mathbf{x} + \delta$ is a valid point in $\mathcal{X}$ (e.g. ensure all values are bounded in some interval $[a, b]$), but we could also change the set of allowed perturbations based on other criteria.
- $L : \mathbb{R}^K \times \mathbb{Z}^+ \to \mathbb{R}^+$ is a loss function, i.e. a mapping from the model predictions and true labels to a non-negative number.[6]

[6] Loss functions are further discussed in § 2.4.

While this equation may look daunting at first, we can intuitively understand it as follows: we try to find the best adversarial example within the allowed set of perturbations (the inner maximisation function, which maximises the loss of the true label), and then we try to minimise the empirical risk, i.e. find the parameters $\boldsymbol{\theta}$ such that the overall loss is as low as possible. From this follows that there are two main tasks within the field of adversarial robustness: (1) finding good adversarial examples, i.e. good attack methods, and (2) finding good defence strategies.

In solving the first task, a common approach is to find a *lower bound* on the inner optimisation objective.[7] This means that one tries to find *some* adversarial example. There may be better examples, but as long as we solve the inner maximisation problem well enough, we can work with it.

[7] Kolter and Madry, 2018.

Closely related to the inner maximisation problem is the concept of *adversarial robustness*. Adversarial robustness is defined as the minimum adversarial perturbation $\delta \in \Delta(\mathbf{x})$ that enables $h_{\boldsymbol{\theta}}(\mathbf{x} + \delta) \neq h_{\boldsymbol{\theta}}(\mathbf{x})$.[8] In order to quantify adversarial robustness, we need some distance measure for δ, of the form $\mathcal{X} \to \mathbb{R}$. A common distance measure is the ℓ_p-norm, which is defined as

[8] Qin et al., 2020. Note that access to the loss function L used by the model is not required in order to find an adversarial example, although it may help to have this access, such as in white-box attacks (see § 3.2).

$$\|\delta\|_p := \left(\sum_{i=1}^{n} |\delta_i|^p \right)^{1/p}, \tag{3.2}$$

but in principle any other distance measure can work too. For $p = 1$, the ℓ_p-norm is equal to the taxicab norm (Manhattan distance), for $p = 2$, the ℓ_p-norm is equal to the Euclidean norm, and for $p = \infty$, it is equal to the maximum norm.

3.2.2 *Why are adversarial examples possible?*

A lot of research is being done on adversarial attacks on artificial neural networks. It would be interesting to step back and ask: how come adversarial attacks exist? Carlini and Wagner (2017) offer the following:

- A possible explanation is that due to the highly non-linear nature of artificial neural networks, they are likely to have blind spots. Adversarial examples may lie in these blind spots. Defensive distillation is a strategy to reduce overfitting on the training data, thereby removing these blind spots. This strategy works by using label smoothing.

- But defensive distillation does not work well at all. Carlini and Wagner (2017) devised new attack strategies that are practically immune to defensive distillation. So it appears that these blind spots do not provide a (full) explanation of adversarial examples.
- As an alternative, Goodfellow et al. (2015) introduce an explanation based on the locally linear nature of neural networks. This is the so-called linearity hypothesis and appears to be supported by evidence.

3.3 *Attack algorithms*

In this section, I discuss some of the attack techniques used for solving the inner maximisation problem in eq. (3.1).

3.3.1 *Fast gradient method*

One of the earliest white-box attack methods was proposed by Goodfellow et al. (2015). This technique, called fast gradient method (FGM), works by taking the gradient of the loss function L, and then taking a step of size ϵ in the direction that maximises the loss.[9] FGM is a non-targeted attack, because the goal is misclassification without a specific target class. For the ℓ_∞ norm, the perturbation δ is given by the formula:

$$\delta = \epsilon \operatorname{sign}(\nabla_{\mathbf{x}} L(h_{\boldsymbol{\theta}}(\mathbf{x}), y)), \tag{3.3}$$

or for an ℓ_p norm where $p \neq \infty$:

$$\delta = \epsilon \frac{\nabla_{\mathbf{x}} L(h_{\boldsymbol{\theta}}(\mathbf{x}), y)}{\|\nabla_{\mathbf{x}} L(h_{\boldsymbol{\theta}}(\mathbf{x}), y)\|_p}. \tag{3.4}$$

[9] The authors use the term fast gradient sign method (FGSM) instead. Nicolae et al. (2018) generalise the method to other norms than ℓ_∞, hence the term FGM is more appropriate.

The main advantage of the fast gradient method is that it is, well, fast. While iterative approaches may find a better lower bound on the inner optimisation problem in eq. (3.1), they need far more computational resources. The method is surprisingly effective too. Goodfellow et al. (2015, p. 3) find that with $\epsilon = 0.25$, they can achieve an error rate of 89.4% on MNIST. These properties make FGM suitable for quickly assessing the adversarial robustness of a model.

In practical settings, using the FGM attack is often unfeasible, because the attack needs access to the model gradients, which are typically unavailable.

3.3.2 *Carlini–Wagner method*

A more advanced, iterative version of the FGM is the Carlini–Wagner (C&W) method.[10] This attack is considered the state of the art in white-box attacks.[11] The authors reformulate the problem of finding

[10] Carlini and Wagner, 2017.

[11] Nicolae et al., 2018; Chen et al., 2020.

an adversarial example as follows:

$$\begin{aligned} &\text{minimise} && \|\delta\|_p + c \cdot f(\mathbf{x} + \delta) \\ &\text{such that} && \mathbf{x} + \delta \in [0,1]^n \end{aligned} \tag{3.5}$$

where n is the number of features of input $\mathbf{x}$, $c > 0$ is a suitably chosen constant, and f is a chosen objective function.[12] For image input, the box constraint ensures that the resulting adversarial example is a valid image. The optimisation problem is then solved iteratively.

[12] The authors consider different objective functions and choose the best one; see Carlini and Wagner, 2017, pp. 44–45.

Like the FGM attack, the C&W attack is white-box, limiting its usefulness in practical settings. Still, because of its very high performance, it is a relevant attack to consider when assessing model robustness. Furthermore, one can intuit that a white box offers a better lower bound on the inner optimisation objective in eq. (3.1). After all, it is unlikely that an adversary without access to the model parameter performs a more effective attack than an adversary with this access. This is an additional consideration for studying white-box attacks, despite their limited usefulness in real-life settings.

3.3.3 *Gradient estimation methods*

The attack methods discussed in the previous subsections need access to the gradients of the model in order to find an adversarial example. While this is feasible in experimental settings, in real-life situations it is rather uncommon to have full access to the model f. Instead we can only access the model as an oracle and observe the prediction for a given input. To make things even harder for the adversarial, typically only the top-1 prediction is returned. Hence we are in the black-box hard-label setting. In this project I consider black-box attacks for two reasons. Firstly, they are more realistic in real-life settings. Secondly, they more easily allow for the model to be swapped by another model. After all, the model is only ever accessed as an oracle. This latter property is desirable, given that I aim to compare different models.[13]

[13] There is another practical consideration. Some models under test are trained with EM in `libspn-keras` (see § 2.1.3 and § 4.2). The EM implementation in this framework makes use of custom gradients in TensorFlow. Due to this, existing implementations of white-box attacks do not work on these models out of the box. Black-box attacks do not suffer from this problem because the gradients are not requested.

Why is the hard-label setting so hard? The main difficulty is that in this setting, the decision boundary is *discrete* and discontinuous, not continuous as in the soft-label setting.[14] This means we cannot use strategies for solving continuous optimisation problems, such as gradient descent. A strategy that *does* work is performing a random walk on the decision boundary.[15] Unfortunately this approach suffers from exponential search time, and lacks convergence guarantees.

[14] Cheng et al., 2018.

[15] See e.g. Brendel et al., 2018

Cheng et al. (2018) have developed a new black-box attack that addresses these issues, dubbed 'Opt attack'. By reformulating the attack problem as an optimisation problem, they define an objective function $g(\boldsymbol{\phi})$ that represents the distance from the original input $\mathbf{x}$ to the nearest adversarial example in the direction $\boldsymbol{\phi}$:

$$g(\boldsymbol{\phi}) = \underset{\lambda > 0}{\arg\min}\, h_{\boldsymbol{\theta}}(\mathbf{x} + \lambda \frac{\boldsymbol{\phi}}{\|\boldsymbol{\phi}\|}) \neq y, \tag{3.6}$$

where λ is the smallest possible distance in direction $\boldsymbol{\phi}$ to arrive at an adversarial example. The optimisation problem then is given by

$$\min_{\boldsymbol{\phi}} g(\boldsymbol{\phi}), \tag{3.7}$$

i.e., finding the direction for which an adversarial example is closest. Even if the classifier function is not continuous, like in the hard-label setting, $g(\boldsymbol{\phi})$ *is* continuous.

The adversarial example $\tilde{\mathbf{x}}$ becomes

$$\tilde{\mathbf{x}} = \mathbf{x} + g(\boldsymbol{\phi}^*) \frac{\boldsymbol{\phi}^*}{\|\boldsymbol{\phi}^*\|},$$

where $g(\boldsymbol{\phi}^*)$ is the optimal solution of eq. (3.7).

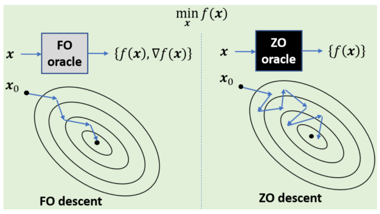


Figure 3.3: First order *vs.* zeroth order optimisation. From Liu et al., 2020.

Now the optimisation problem can be solved. Because we do not have access to the gradients, we need to use *zeroth order optimisation*. Figure 3.3 gives an illustration in which first order and zeroth order optimisation is compared. The zeroth order optimisation strategy only needs access to the function value, not the gradient.[16]

[16] Liu et al., 2020 provide a primer on how zeroth order optimisation works.

Cheng et al. (2018) add some refinements, by first finding a general direction, then performing a binary search to find a solution within the region identified in the first step. The authors then go on to prove some properties of their proposed attack method. In particular they prove that their lower bound for an adversarial example is within an adjustable ϵ-precision, and the algorithm has guaranteed convergence.

IMPROVING ON THE WORK by Cheng et al. (2018), Chen et al. (2020) have introduced the HopSkipJump (HSJ) attack. Their approach improves query efficiency over Opt attack. That is, the method can attain similar results with fewer calls to $h_{\boldsymbol{\theta}}$. Each iteration of the HSJ algorithm involves three steps. First, the direction of the gradient is estimated. Then the step size is determined. Finally, a boundary search is performed via a binary search. For mathematical details, I refer to the original paper.

HSJ is considered the state of the art in black-box attacks, with performance similar to the C&W attack.[17]

[17] Chen et al., 2020, p. 2.

3.4 *Measuring adversarial robustness*

In § 3.2.1 I introduced a quantification for adversarial robustness, viz. a norm that measures the minimum distance in the input needed to alter the model's classification. Next to this distance measure, there are some related measures, proposed by Qin et al. (2020):

- *Calibration* measures the alignment between predicted probability and accuracy. Informally, we would like the model to be confident about samples it can predict well, and have low confidence for for samples it has difficulty to predict. It is typically measured in expected calibration error (ECE). This approach divides the data in buckets, sorted by predicted probability (i.e. confidence) of the predicted class. ECE is low if accuracy and confidence are close. Hence for ECE, a lower score is better.

- *Model uncertainty* measures how random initialisation affects training. It can be quantified by the variance in predictions of n independently trained models. A lower variance is better. For generative models, we could alternatively look at the density of a data point, and reject points that have a density below a threshold, since such data points are unlikely to have come from the same distribution as the training data.

Calibration is especially interesting here. When a model is *miscalibrated*, the "predicted confidence is not indicative of the true likelihood of the model being correct".[18] This is problematic in practical applications where the input is never seen by the model before. In these situations there may a *distributional shift*, which means that the input samples come from a different distribution than the training samples. Ideally the model should be able to predict these as well, or at least be honest in its confidence of predictions.

[18] Qin et al., 2020, p. 1.

While the three metrics are distinct and measure different concepts, Qin et al. (2020) found that they are strongly aligned. This suggests that in order to improve model calibration, we could explicitly address examples that are adversarially unrobust and train them differently.

4 Experimental setup

In this chapter, I explain the methodology used for answering the research questions (§ 1.2). I take an empirical, experimental approach in this project. By training a large family of models, varying relevant independent variables, I aim to discover what factors influence model robustness. In addition, the models are attacked in different ways. I monitor several dependent variables for quantifying robustness.

4.1 Datasets

I train models on three types of datasets: (1) image data, (2) empirical tabular data, and (3) synthetic tabular data. I choose a mix of image and tabular datasets in order to assess if differences in robustness arise between the two. The inclusion of synthetic dataset in my experiments allows for precisely controlling the characteristics of the dataset, and assessing the impact of these characteristics on model robustness.

For an overview of the non-synthetic datasets used, see table 4.1. For image data, I pick the MNIST[1] dataset because it is relative small compared to other commonly used image datasets in the field, such as CIFAR-10 and ImageNet; small both in number of input features and in number of training examples. This allows for faster training and attacking with limited computational resources. In addition I use the Fashion-MNIST[2] dataset, for which the same model architectures can be reused.

For tabular datasets, I select a subset of the UCI[3] datasets, as provided by the OpenML[4] dataset library.[5] The selected datasets are a subset of those used by Correia et al. (2020). I only consider

[1] Lecun et al., 1998.

[2] Xiao et al., 2017.

[3] Dua and Graff, 2019.

[4] Vanschoren et al., 2013.

[5] See https://openml.org/search?type=data.

Dataset name	$\lvert\mathbf{x}\rvert$	K	$\lvert D_{\text{train}}\rvert$	$\lvert D_{\text{val}}\rvert$	$\lvert D_{\text{test}}\rvert$	Balanced
MNIST	784	10	54 000	6 000	10 000	✓
Fashion-MNIST	784	10	54 000	6 000	10 000	✓
authent	4	2	960	206	206	
diabetes	8	2	537	115	116	
gesture	32	5	6 911	1 481	1 481	
jungle	6	3	31 373	6 723	6 723	
robot	24	4	3 819	818	818	
segment	18	7	1 617	346	347	✓
texture	40	11	3 849	826	825	✓
vehicle	18	4	592	127	127	✓
wdbc	30	2	398	85	85	

Table 4.1: Overview of non-synthetic datasets. $\lvert\mathbf{x}\rvert$ is the number of input features (flattened) and K is the number of classes.

Configuration	# samples	# informative features	# redundant features	# useless features
Default	10 000	5	3	12
Fewer samples	1 000	5	3	12
More samples	30 000	5	3	12
More informative ft.	10 000	17	3	0
No redundant ft.	10 000	5	0	15
More redundant ft.	10 000	5	15	0

Table 4.2: Overview of configurations for the synthetic tabular datasets. Note that given the number of classes ($K = 10$), reducing the number of informative features is not possible—then there would be too few features to put each cluster on a distinct corner of the hypercube.

tabular datasets where all input features are continuous and there is no missing data.

I generate synthetic tabular datasets with the `sklearn` method `datasets.make_classification`.[6] I use six named configurations, listed in table 4.2. For each configuration, I vary one parameter w.r.t. the default configuration. The number of classes K is fixed at 10 and the number of features is fixed at 20. The classes are balanced and each class has a single cluster on a hypercube with sides of length 4. All features have a mean of zero and standard deviation of one. There are no repeated features. For 1 percent of samples, the class is assigned randomly.

[6] Pedregosa et al., 2011.

For image data, I use the train–test split as provided in the original dataset. The original training partition is randomly split in a training (90%) and validation (10%) dataset. For tabular data, both empirical and synthetic, I use a random split of train (70%), validation (15%), and test (15%) dataset.

4.1.1 *Preprocessing*

All input features are *standardised* to have zero mean and unit variance. For image datasets, standardisation is performed globally over all input features. For tabular datasets, standardisation is performed per input feature. The synthetic datasets do not need to be standardised, because their input features have zero mean and unit variance by definition. Standardisation allows adversarial distance measures across datasets to be compared to each other.

For the empirical tabular datasets, I drop features with near-zero variance, since they are not informative. The number of features reported in table 4.1 is *after* these features are dropped.

4.2 *Models*

I train two families of models: *baseline models* and *PCs*. The baseline models are ANN classifiers. For image data I use CNNs as a baseline. For tabular data I use multi-layer perceptrons (MLPs) instead. The baseline models are only trained discriminatively, whereas the PCs are trained both discriminatively and generatively. While not directly relevant for comparing discriminative and generative classifiers, the baseline models serve as a check to assess if the PCs were trained correctly. Additionally, these models serve as a ‘zero measurement’ for adversarial robustness. All baseline models are built with Keras, as

			Fashion-MNIST						MNIST					
	# parameters		Baseline		Discriminative		Generative		Baseline		Discriminative		Generative	
Architecture	ANN	PC	Acc.	Loss	Acc.	Loss	Acc.	Loss	Acc.	Loss	Acc.	Loss	Acc.	Loss
Reference CNN	34 826	—	0.908	0.257	—	—	—	—	0.992	0.024	—	—	—	—
Conv. SPN (large)	1 008 394	25 579 674	0.912	0.253	0.875	0.357	0.742	594	0.986	0.047	0.975	0.085	0.905	412
Conv. SPN (small)	48 414	175 262	0.881	0.324	0.889	0.324	0.679	639	0.978	0.074	0.962	0.114	0.881	445
DGC-SPN (large)	50 234	1 214 498	0.893	0.299	0.890	0.306	0.685	948	0.982	0.071	0.978	0.071	0.905	987
DGC-SPN (small)	102 430	230 050	0.893	0.294	0.878	0.342	0.701	636	0.985	0.058	0.967	0.109	0.893	444
RAT-SPN	918 490	922 250	0.891	0.350	0.852	1.565	0.784	139	0.981	0.086	0.970	0.472	0.895	-119

Table 4.3: Overview of models for image data. Reported accuracy and loss are on the test datasets.

provided in Tensorflow v2.5.0 [7]. All PCs are built with `libspn-keras`.[8]

[7] Abadi et al., 2015

[8] Van de Wolfshaar and Pronobis, 2020.

4.2.1 *Model training*

All models are trained on a high number of epochs (1 000), but with an early stopping stopping rule to prevent overfitting. If after 3 epochs the loss on the validation dataset has not decreased by at least 0.005, training is halted. An additional callback ensures that when a plateau is reached, the learning rate is decreased. Specifically, the original learning rate is multiplied by 0.5. For this callback too, validation loss is monitored.

Discriminative models are trained using SGD with an initial learning rate of 0.001 for ANNs and 0.01 for PCs. Generative models with a convolutional structure (for image datasets) are trained using EM with an initial learning rate of 0.05. Generative RAT-SPNs are trained using SGD with an initial learning rate of 0.01.[9]

[9] I chose the values for learning rates and optimisation methods by testing different different values, and selected the values that yield the highest model performance.

ANNs are initialised with the default initialisation offered by Keras. For dense layers, this is a Xavier uniform initialiser for the weights and zeros for the biases. For PCs, I tested different initialisers, both for the leaf distributions and for the sum weights. I found best performance by initialising the location and scale of the distributions in the leaf nodes with truncated normal distributions. The weights of the sum nodes are initialised with truncated normal distributions as well.

For PCs trained on image data, the fact that classes are balanced is leveraged: the weights of the root sum are uniformly initialised and are marked as non-trainable. For tabular datasets, the weights of the root sum *are* trainable.

4.2.2 *Models for image datasets*

I train five different PC architectures for image data. Two architectures use a convolutional structure *without partially overlapping scopes*; a large model and a smaller model. I call these 'Conv. SPN (large)' and 'Conv. SPN (small)'. These architectures are comparable to those used in Sharir et al. (2018) and Butz et al. (2019). Two architectures use a DGC-SPN structure as introduced by van de Wolfshaar and Pronobis (2020). These models *do have partially overlapping scopes*. Again I create a larger and a smaller model with this structure: 'DGC-SPN (large)' and 'DGC-SPN (small)'.[10] Finally, I add one RAT-SPN structure as introduced by Peharz et al. (2018). Contrary to the convolutional structures, the

[10] See § 2.2.1 for more information on these structures.

RAT-SPN "do not exploit the neighbourhood correlations present in images".[11] For all models, I manually tweak the details of the structure—such as the number of layers and layer types—in order to attain acceptable performance with generatively trained PCs on MNIST.[12] With 'acceptable' I mean a classification accuracy of at leat 85%. Attaining such performance is not trivial, because generative models are not naturally good classifiers.

[11] Peharz et al., 2018, p. 5.

[12] I do *not* tweak model structure based on performance on Fashion-MNIST. Instead I take the model structure found with the MNIST dataset and train it on Fashion-MNIST too.

For each PC, I train a baseline ANN that is *structurally similar* to the PC. For the PCs with convolutional structures, this is a CNN. For the RAT-SPN, this is an MLP. In order to structurally match convolutional PCs and CNNs, I use a translation table which is included in appendix A.[13] The precise structures used for each model are laid out in appendix B.

[13] Note that this method of structure matching may still lead to large differences between the models in number of parameters. I decided structural similarity to be more important than similarity in number of parameters.

Finally, I include a single reference CNN for each dataset that is not structurally matched to any PC. It merely serves as a check to benchmark model performance against.

Naturally, each PC is trained both discriminatively and generatively. In total, there are 3 'modes' (baseline, discriminative, generative) $\times$ 5 architectures $\times$ 2 datasets + 2 reference CNNs = 32 models for image data. See table 4.3 for an overview of these models.

4.2.3 Models for tabular datasets

For tabular datasets, I use RAT-SPNs for the PCs and MLPs for the baseline ANNs.[14] I vary: (1) the learning mode, (2) the depth of the model, and (3) the dataset on which the model is trained.

[14] See § 2.2.2 for more information on RAT-SPNs and their hyperparameters.

In total, there are 3 'modes' (baseline, discriminative, generative) $\times$ 3 depth levels $\times$ 9 datasets = 81 models trained on empirical tabular datasets. The actual number of trained models is slightly lower, viz. 69. The discrepancy arises because in RAT-SPNs, the decomposition into D layers must satisfy the decomposability criterion. At some point, all leaf partitions contain only a single random variable, so they cannot be split further. Therefore, depth D is bounded by the number of input variables $|\mathbf{x}|$ in the following way: $D \leq \mathcal{O}(\log|\mathbf{x}|)$. More precisely, 2^{D-1} must not exceed $|\mathbf{x}|$. The datasets 'jungle', 'diabetes' and 'authent' have too few input features to build a RAT-SPN of depth 4. The dataset 'authent' also has too few input features to build a RAT-SPN of depth 3.

Following Peharz et al.'s (2018) approach, I performed a grid search in order to find the hyperparameters that yield highest accuracy for generative models. For each $D \in \{2, 3, 4\}$, I tested every combination of $R \in \{10, 20, 30, 40, 50\}$, $I \in \{4, 6, 8, 10, 12, 14, 16, 18\}$, and $S \in \{6, 9, 12, 15, 18, 21\}$. In case of draws in accuracy, I selected the model with fewest params. For each selected model, I then trained the same model discriminatively, and I constructed an MLP with same depth and similar number of parameters. I list the selected hyperparameters along with their performance in appendix C.

FOR THE SYNTHETIC DATASETS, I take a different approach. I do not perform a grid search for finding optimal parameters. Instead, I keep the RAT-SPN hyperparameters fixed at $R = 32, I = 10, S = 6$.[15] For MLPs I keep the number of hidden units H fixed at 256.

[15] I took this decision because for synthetic data, model accuracy can be more easily controlled by changing the *dataset* hyperparameters. This approach needs far fewer computational resources than performing a grid search.

I *do* vary model depth $D \in \{2, 3, 4\}$. I also train the models on synthetic tabular data with 3 different random seeds. This decision is informed by the results of the attacks run on the empirical tabular datasets; see chapter 5. In total, there are 3 'modes' (baseline, discriminative, generative) $\times$ 3 depth levels $\times$ 6 dataset configurations $\times$ 3 random seeds = 162 models trained synthetic tabular datasets. For model performance, see appendix D.

4.3 *Corruptions*

I consider two types of corruptions for assessing non-adversarial robustness. For all empirical datasets I look at the effects of adding Gaussian noise to the test dataset. I vary the amount of noise by setting the scale parameter σ of the Gaussian distribution. I use values $\sigma \in \{2^i \mid i \in \mathbb{Z} \land -2 \leq i < 3\}$. This way of perturbing the input data makes sense, because all input features are standardised (see § 4.1.1).

For models trained on MNIST, I additionally consider performance on MNIST-C. This dataset is introduced in § 3.1.

I measure classification accuracy on the corrupted datasets.

4.4 *Attacks*

In order to assess adversarial robustness, I run different attack algorithms on the trained models. All attacks are *non-targeted*, i.e., the goal of the adversary is to misclassify, not to change the output classification to a specific target class.

For running the attacks, I use the implementations as provided in the Adversarial Robustness Toolbox, by Nicolae et al. (2018). I use default settings unless noted otherwise.

For models trained on empirical tabular data, I run the FGM attack, C&W attack, and the HSJ attack, as introduced in § 3.3. The first two attacks are white-box whereas the latter is black-box. Running white-box attacks for these models is possible, because all are fully differentiable. Hence the gradients, which are needed for these white-box attacks, are accessible. All attacks are run with both an ℓ_2 and ℓ_∞ minimisation objective, in order to assess if robustness differences arise between the two. I set the step size ϵ in FGM to 0.3. For ℓ_∞ C&W attacks ϵ is set to 0.3 too, but for this attack ϵ is an *upper bound* for the adversarial perturbation. HopSkipJump and ℓ_2 C&W have no notion of an ϵ step size/upper bound.

For models trained on image data, I use the same settings but only use the black-box HopSkipJump attack. This is because most generative models on image data are trained with the EM algorithm in `libspn-keras`, which uses custom gradients in TensorFlow. Due to this, the FGM and C&W implementations in the Adversarial Robustness Toolbox cannot be used, because these white-box attacks need access the original gradients—something `libspn-keras` does not presently offer.

For models trained on synthetic tabular data, I only perform the ℓ_2 HopSkipJump attack, in order to save computational resources.[16]

[16] The rationale is that in the results, no significant differences were found betweenℓ_2 and ℓ_∞ attacks.

The total number of attacks is:

Image data	32 models $\times$ 2 norms =	64
Empirical tabular data	3 attacks $\times$ 69 models $\times$ 2 norms =	414
Synthetic tabular data	162 models =	162
Total		640

For all attacks, I measure the mean ℓ_p distance of the adversarial examples, where p is the norm used as the objective when running the attack. I also save the generated adversarial examples. I measure running time as well. Finally, I measure loss and accuracy of the model on the adversarial test set, to assess the successfulness of the attack.

Following the approach by Chen et al. (2020, p. 8), I limit the number of adversarial examples generated, in order to speed up the attacks. I take the test dataset, which the model has not seen during training, for generating the adversarial examples. I only attack examples in the test set which are correctly classified by the model. After all, if the model already misclassifies the example, there is no need to attack. If there are fewer than 1 000 correctly classified examples in the test dataset, all are used for generating adversarial examples. If there are more, I take a stratified sample of the test set of size 1 000. Stratification ensures that in the adversarial test dataset, the classes are balanced.[17]

[17] For the Fashion-MNIST dataset, I generate 100 instead of 1 000 adversarial examples, to speed up computation.

4.5 *Practicalities*

4.5.1 *Computational resources*

I trained and attacked the models in using different computational resources. I ran most experiments in a virtual machine rented from Google Cloud Platform with a dedicated GPU.[18] Additional experiments which required fewer resources were run on a 2015 laptop with a four-core 2.2 GHz CPU and no dedicated GPU.

[18] See https://cloud.google.com.

4.5.2 *Data management*

For managing the large ensemble of models and attacks, I used the open source platform MLFlow.[19] With this platform, I store trained models including the model weights, the parameters used for the experiments, and the metrics monitored during training and attacking. I also store the generated adversarial test sets.

[19] See https://mlflow.org.

4.6 *Summary of independent variables*

For an overview of the independent variables (IVs) used, see table 4.4.

Dataset type	IV	# levels	Levels
Image	Mode	3	Baseline, discriminative, generative
	Dataset	2	MNIST, Fashion-MNIST
	Model architecture	6	Reference CNN$^{(1)}$, conv. SPN (large), conv. SPN (small), DGC-SPN (large), DGC-SPN (small), RAT-SPN
	Attack norm	2	ℓ_2, ℓ_∞
	Attack algorithm	1	HSJ
Empirical tabular	Mode	3	Baseline, discriminative, generative
	Dataset	9	Authent, diabetes, gesture, jungle, robot, segment, texture, vehicle, wdbc
	Model depth	3	2, 3, 4$^{(2)}$
	Attack norm	2	ℓ_2, ℓ_∞
	Attack algorithm	3	FGM, C&W, HSJ
Synthetic tabular	Mode	3	Baseline, discriminative, generative
	Configuration	6	Default, fewer samples, more samples, more informative features, no redundant features, more redundant features
	Model depth	3	2, 3, 4
	Model random seed	3	1, 2, 3
	Attack norm	1	ℓ_2
	Attack algorithm	1	HSJ

Table 4.4: Overview of independent variables used in the experiments. Notes: (1) The reference CNN is not attacked. (2) Some datasets lack a model with depth 3 or 4; see § 4.2.3 for explanation.

5 Results

In this chapter, I present the results of the experimental setup outlined in the previous chapter.

5.1 Corruptions

5.1.1 Random noise

As expected, adding noise to the test data degrades classification performance. Classification performance on the clean test datasets varies depending on dataset and 'mode'; see appendix C. In order to make a fairer assessment, I look at *relative* classification accuracies instead.[1] This value is computed by dividing the accuracy on the noisy datasets by the model accuracy on the clean dataset. The results are plotted in figure 5.1.

[1] The interested reader can consult a plot of the original, absolute accuracies in appendix E.

For all modes, the plots reveal a trend of decreasing accuracy as the noise level increases. When comparing the modes with each other, however, no unambiguous trend can be detected. For most datasets, accuracy on generative models is lower than accuracy on discriminative models (authent, diabetes, gesture, segment, texture). The jungle and Fashion-MNIST datasets reveal an opposite trend. For the other datasets (robot, vehicle, wdbc, MNIST), the difference

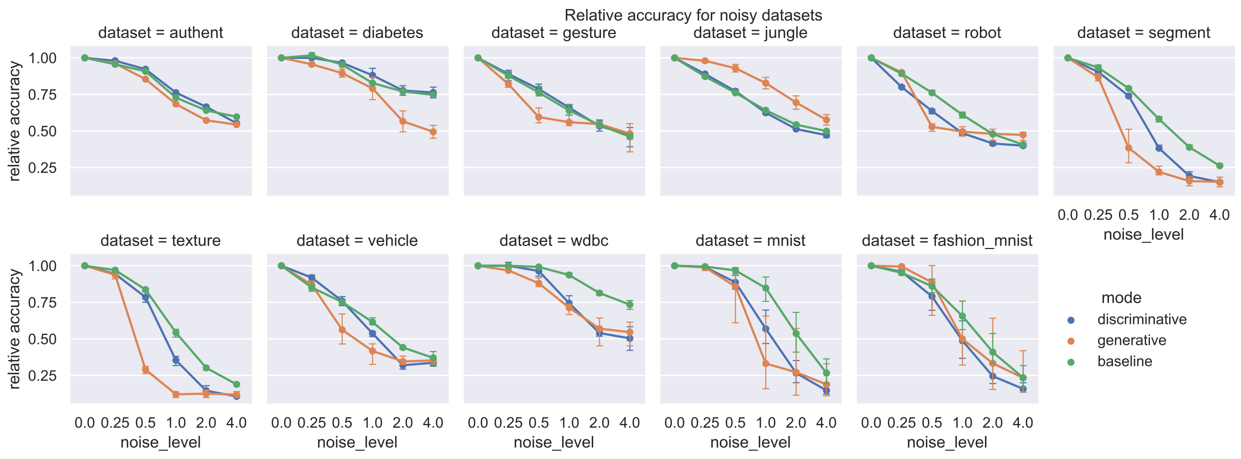


Figure 5.1: Relative classification accuracy on noisy datasets. The error bars represent the 95% confidence interval on the observations.

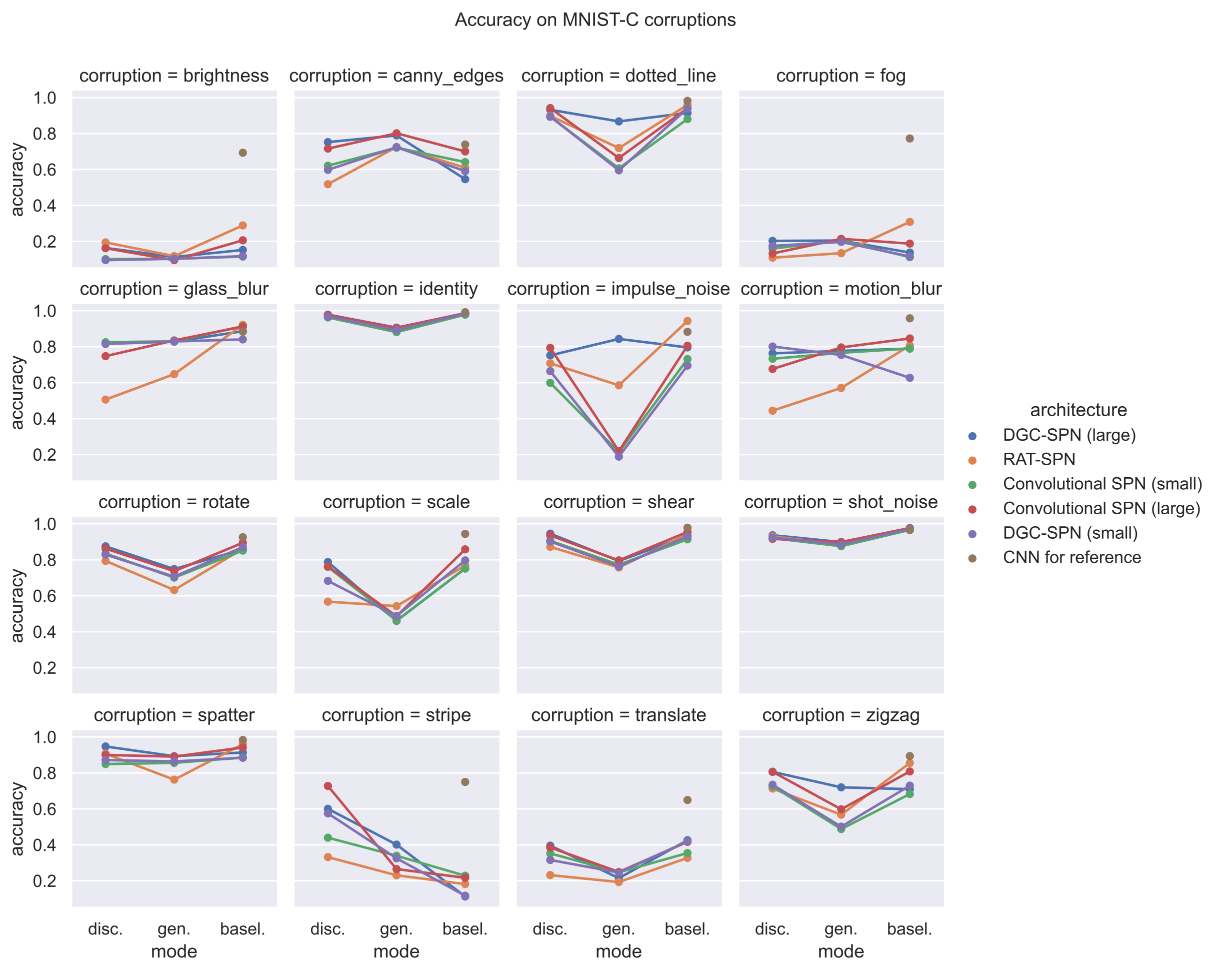


Figure 5.2: Classification accuracy on MNIST-C, split based on corruption type, architecture, and mode.

is unclear: at some noise levels, the discriminative model performs better whereas at other noise levels the generative model has highest relative accuracy.

How can we explain the absence of a clear trend? A possible explanation is the effect of sample size. The datasets where accuracy on generative models is lower than on discriminative models, all have train sample sizes smaller than 7 000. For the larger datasets on the other hand (> 30 000 training samples), jungle, MNIST, and Fashion-MNIST, generative models do seem to perform better—although this effect is not unambiguous for MNIST, where there is large variance between different models, as can be seen from the large error bars.

5.1.2 *MNIST-C*

The results for the MNIST-C corruptions are mixed too. It appears that model accuracy is highly dependent on the type of corruption applied to the input data. For the corruption `canny_edges`, the generative models have better performance than the discriminative and baseline models. For the other corruptions, the direction is reversed or the direction is unclear. It does appear that the different model architectures generally agree on the direction (and often also magnitude) of the effect of the corruptions. The most notable exception is the `impulse_noise` corruption for the DGC-SPN (large) architecture; the generatively trained DGC-SPN shows higher performance than its discriminative counterpart. This is different from all other architectures, which show a reverse effect.

It is possible that each model architecture has different inductive biases. That is, each architecture makes different assumptions that are used to predict the class for unobserved inputs. If this is so, it can explain why each model responds differently to each type of perturbation.

5.2 *Attacks and adversarial distance*

The attacks are generally successful. Figure 5.3 shows a histogram of the adversarial accuracy for the attacks run on empirical datasets. Half of the attacks (233 out of 478) result in an adversarial accuracy of lower than 10%. The remaining attacks are distributed evenly over attack norms, attack algorithms, learning modes, and datasets. For attacks on the synthetic tabular datasets, all adversarial datasets result in an accuracy below 10%. In figure 5.4, some example adversarial examples for the MNIST dataset are shown, along with their predicted classes. These are generated with the HopSkipJump (HSJ) attack.

From the examples shown, it already seems that the adversarial examples for the generative model need more perturbation in order to achieve misclassification. In other words, the generative model appears more adversarially robust than the discriminative model. This conjecture is confirmed when assessing the average adversarial distance, compared to the baseline models. In figure 5.5, I compare mean adversarial distances for the generative and discriminative models with the baseline models. For all models, norms and datasets, the generative model has a higher mean adversarial distance than the discriminative model.[2] There are differences between architectures in magnitudes of the effect, but the direction of the effect is clear. From this experiment it seems that indeed, generative models are more adversarially robust than discriminative models.

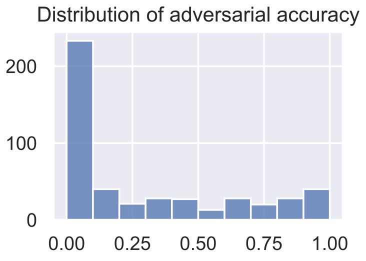


Figure 5.3: Distribution of adversarial accuracy for the attacks on empirical (image and tabular) datasets.

[2] There is one exception: for the RAT-SPN on MNIST with norm ℓ_2, the discriminative model has higher mean adversarial distance. But this difference is small compared to the other differences.

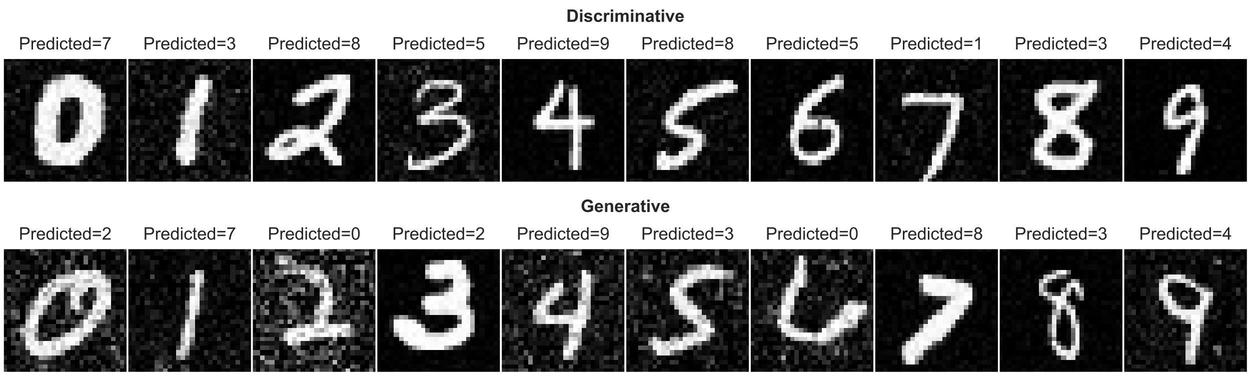


Figure 5.4: Adversarial examples for the MNIST dataset, generated with the HopSkipJump attack with ℓ_2 norm objective. These examples were generated while attacking DGC-SPN (small) models.

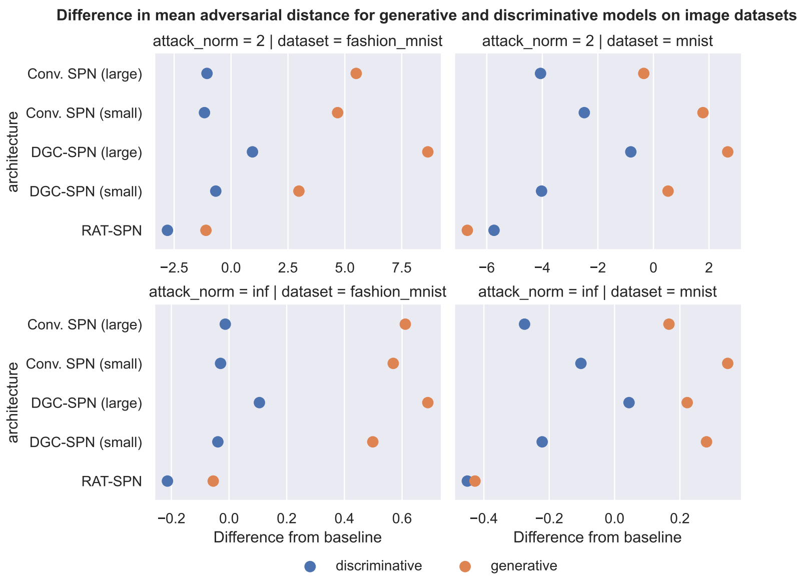


Figure 5.5: Difference in mean ℓ_2 and ℓ_∞ adversarial distance for generative and discriminative models on image datasets, using the HopSkipJump attack.

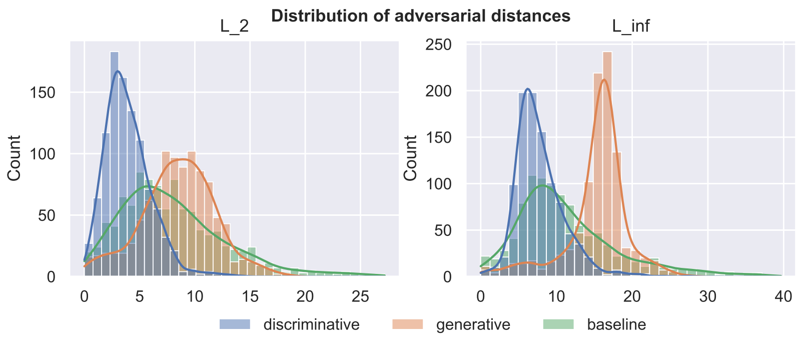


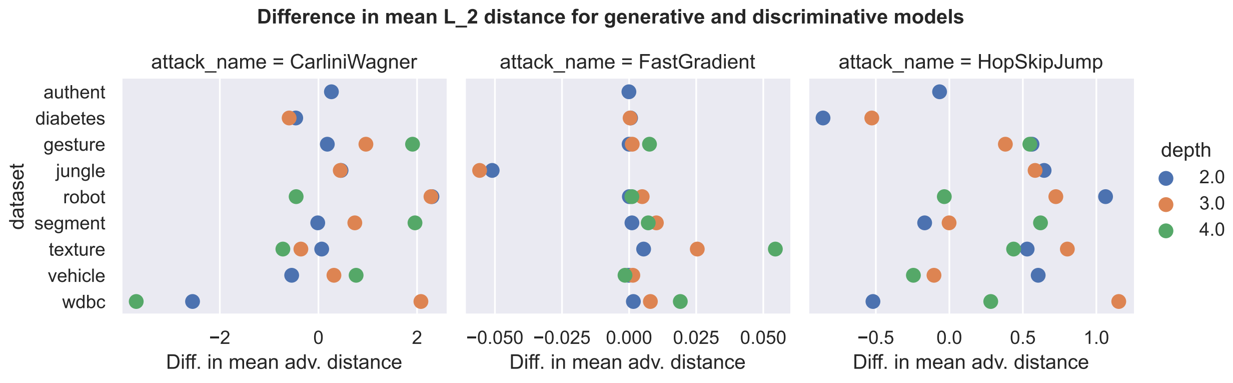
Figure 5.6: Histogram of the adversarial distances, for a DGC-SPN (small) trained on MNIST and attacked with HSJ (same models as in figure 5.4).

As the mean adversarial distances only provide point estimates, it can be worthwhile to look at the distribution of distances too. In figure 5.6, I plot these distributions, both for an ℓ_2 and an ℓ_∞ attack norm. The difference in mean of the discriminative and generative model is highly significant; for the ℓ_2 attack norm, $t(999) = -36.92, p < 0.001$. This histogram is for a DGC-SPN (small) architecture, but other architectures exhibit similar distributions.

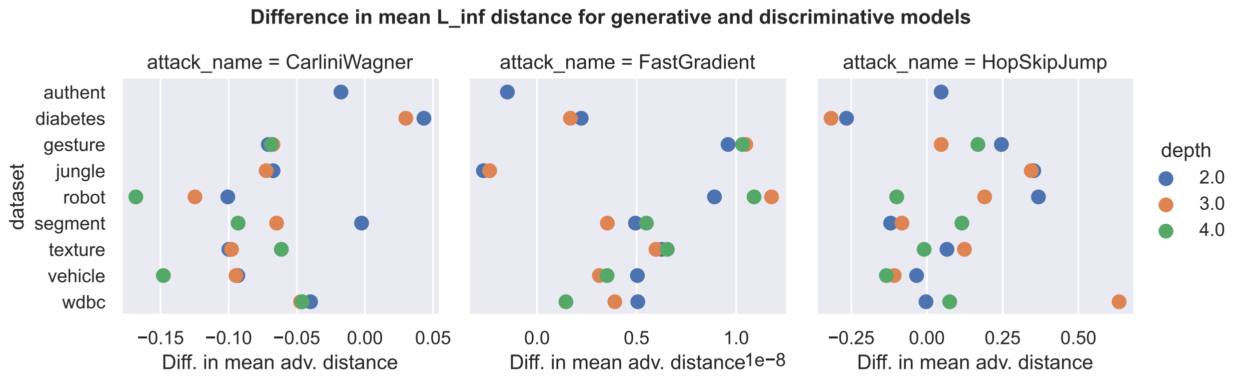
For models trained on empirical tabular datasets, the results are less clear-cut. In figure 5.7, I plot the differences in mean adversarial distance between discriminative and generative models trained on empirical tabular data. The results are further split out by

	ℓ_2 attack norm				ℓ_∞ attack norm			
Attack	M	SD	$t(22)$	p	M	SD	$t(22)$	p
CW	0.231	1.408	0.771	0.449	-0.068	0.049	-6.551	**<0.001**
FGM	0.002	0.021	0.400	0.693	<0.001	<0.001	5.752	**<0.001**
HSJ	0.279	0.513	2.548	0.018	0.070	0.216	1.521	0.143

Table 5.1: t tests for difference in mean adversarial distance for models on empirical tabular datasets. A positive value indicates that the generative model has higher adversarial distance than generative model. Significant results for $\alpha = 0.05$ after Bonferroni correction are boldfaced.



(a)



(b)

Figure 5.7: Difference in mean (a) ℓ_2 and (b) ℓ_∞ adversarial distance for models on empirical tabular datasets, plotting the *difference in adversarial distance* between discriminative and generative model. Interpretation: if point is on the left side (<0): discriminative model has higher adversarial distance than generative model. If point is on the right side (>0): generative model has higher adversarial distance than discriminative model.

model depth. For the raw adversarial distances, consult appendix F. The results for adversarial distance appear near-random. For some combinations of dataset, attack algorithm, attack norm, and model depth, the generative model is more robust (positive values) whereas for other combinations the effect is reversed. In order to assess the main effect of learning mode, I present the t tests in table 5.1. Only for CW and FGM attacks with an ℓ_∞ attack norm, the difference is statistically significant. For FGM there is no practical significance, because the mean and standard deviation of the effect is close to zero. Hence the only (statistically *and* practically) significant effect can be found for the CW attack with ℓ_∞, where discriminative models on average have higher adversarial distance than generative models. Since this effect is not present in the other combinations of attack norm and attack algorithm, I conclude that for models on empirical tabular data, there is no main effect of learning mode (discriminative *vs.* generative) on adversarial robustness.

There is also no main effect of model depth.

For some specific attacks and datasets, consistent results *can* be found. For the fast gradient method (FGM) attack, the difference in adversarial distance is always close to zero. This result is to be expected, because the FGM algorithm takes a single step of fixed size in the direction that most likely leads to misclassification. Furthermore, for the jungle dataset, generative models either outperform discriminative models, or the difference is very close to zero.

Considering the challenges interpreting the results for the empirical tabular datasets, I run a subset of the experiments for *synthetic* tabular datasets too. This opens up additional IVs that can be manipulated, in order to assess the effects. The configurations used for the synthetic datasets were already outlined in § 4.1. Since the results for image datasets do not reveal significant differences between attack norms, I only consider the ℓ_2 norm. I also limit myself to HSJ attacks, as these black-box attacks are most versatile and allow for comparison with the models trained on image data. Since the results for the empirical tabular datasets appear near-random, I use different random seeds when training the models, to assess variance between independently trained models.

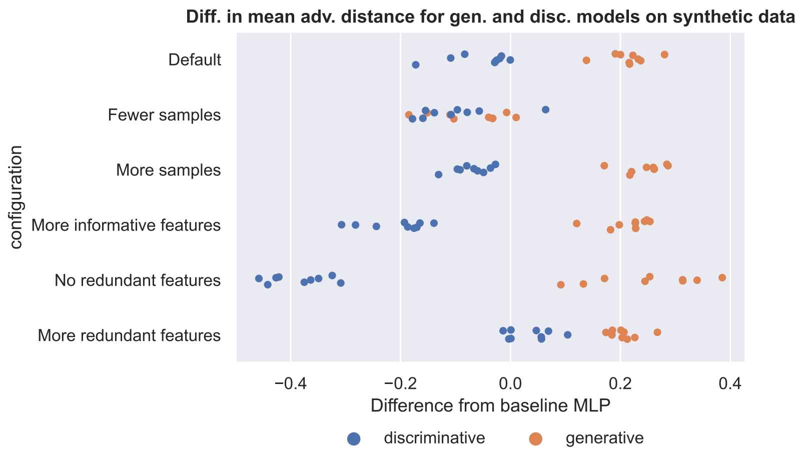


Figure 5.8: Difference in mean ℓ_2 adversarial distance for generative and discriminative models on synthetic tabular datasets.

In figure 5.8, the differences in mean adversarial distance for synthetic datasets are shown, relative to the baseline MLPs. The results for the synthetic datasets reveal again that there is no main effect of model depth on adversarial robustness. I therefore do not distinguish between them in this plot.

For all configurations, the difference between is statistically significant ($p < 0.001$), except for the configuration 'fewer samples'. For this configuration, $t(8) = 0.93, p = 0.38$. These observations confirm the results found with the image datasets: generative models are generally more adversarially robust than discriminative models. The finding that the difference in robustness disappears when the dataset size decreases is insightful too. It can serve as a candidate explanation for the results with empirical tabular datasets. These datasets are typically small, in the order of magnitude $|D_{\text{train}}| \approx 10^3$. It is possible that the differences in adversarial robustness only appear as the train dataset grows. This finding is in concordance with the results for random noise experiments too, where generative models outperform discriminative models only if sample size is large (see § 5.1.1).

6 Conclusions and discussion

In this chapter, I review the results from the previous chapter in light of the research questions formulated in the introduction, and draw my conclusions. Furthermore, I discuss the difficulties encountered during the study. Finally, I point out limitations of my work and indicate directions for future research.

6.1 Main conclusions

Recall the hypothesis formulated in the introduction:

> Generative models are more robust to noise and attacks than discriminative models.

I compared the robustness of discriminative and generative models using two methods: (1) corrupting the input data and assessing classification accuracy, and (2) attacking the model and assessing mean adversarial distance.

The results for robustness to corruptions are inconclusive. In the random noise experiments, the different datasets show different relative accuracy curves. Properties of the dataset are likely to affect robustness to noise.

The type of corruption applied has an effect on accuracy too. The MNIST-C experiments reveal that typically, classification performance with generative models is worse than with discriminative models. But there are corruptions for which the direction of this effect is reversed.

The attacks provide a clearer picture. Generative models are more robust, confirming the hypothesis, but only if the model was trained on a sufficient amount of data. The empirical tabular datasets are typically small. Hence for these, a main effect of learning mode could not be demonstrated.

The other independent variables I manipulated next to learning mode, namely model architecture, model depth, attack norm and attack algorithm, may influence the *magnitude*, but do not seem to influence the *direction* of the main effect on robustness (measured as either classification accuracy or adversarial distance). This result is satisfying, because it indicates that factors that should not influence (the direction of) robustness, indeed do not do so.

All in all, it seems that the robustness of discriminative *vs.* generative classifiers is more intricate than initially thought. In particular, the dependence on dataset size is striking.

6.2 Difficulties

During programming, I encountered several issues. In this section, I would like to point out four of these.

First, I discovered that finding a structure for generative models that achieves satisfactory classification performance is very hard. It took far more effort than for a discriminative model, for instance. This constrained the choice of models. After all, if I could not obtain adequate performance on a generative model whereas its discriminative counterpart was acceptable, the comparison would not be fair. One consequence of this difficulty is that some generative models on image data use a different optimisation method than all other models (viz. EM instead of SGD), for the reason that only with EM satisfactory performance could be achieved. It is known that EM converges faster than SGD for PCs and leads to more stable learning, because the log-likelihood is guaranteed to be non-decreasing during training.[1] So it is not surprising that I could achieve better results with EM than with SGD. What *is* surprising, however, is that I could not manage to train RAT-SPNs with satisfactory performance using EM. I made efforts to explain this phenomenon. I tried a large number of different initialisation strategies, as these are known to influence the quality of the solution dramatically.[2] Among them are truncated normal and Dirichlet distributions for the sum weights and Poon–Domingos and truncated normal distributions for the Gaussian leaves. This was to no avail; I did not find a way to properly train RAT-SPNs with EM.

[1] Zhao et al., 2016, p. 7.

[2] París et al., 2020, p. 10.

Secondly, some model architectures were not available in the `libspn-keras` framework I used. The most striking omission is the Poon–Domingos structure. It is theoretically possible to implement this structure in `libspn-keras`, as I conferred with the framework author, but I lacked the expertise and time to implement the structure myself.

Thirdly, I had difficulty managing the large number of models and attacks. There are over 250 models, 600 attacks, and 750 noise experiments incorporated *in this report*. The total number of experiments is far larger, considering that most models and attacks did not even make it into the report.[3] Good data management is key when dealing with such quantities of results. Logging the results is not very challenging, but querying these logs is. For example: what parameters did I use when running this model? Eventually, I mitigated this difficulty by using MLFlow, which manages recording and tracking of experiments. I only made this decision late into the project, however, so a lot of refactoring was required in order to use MLFlow in my experiments.

[3] Think only of the models considered in the grid search while finding the optimal RAT-SPNs.

Fourthly, I faced computational limitations while conducting the experiments. My personal computer is not fast enough for generating the millions of adversarial examples I needed. Therefore, I looked for other high-performance solutions. The high performance computing (HPC) cluster of the university satisfied my computing requirements, but the queueing times were prohibitively long, sometimes over a week. I ended up renting a virtual machine with Google Cloud Platform. The trial period lasted only 90 days, however. After that time, I could not afford to continue using this platform, so in the final months of my thesis, I was unable to run new experiments that required many computations.

6.3 Limitations

6.3.1 Experimental design

In testing my hypothesis, I chose a single formalism when comparing discriminative and generative classifiers, viz. probabilistic circuits (PCs). This inherently limits the representativeness of the results. One can wonder if results for PCs are valid for other types of (generative) models too. It is highly likely that the model type chosen influences robustness results.

On a related note, I only considered a limited set of attack algorithms for estimating adversarial distance. From my results, it is already apparent that the attack algorithm greatly affects the adversarial distances found.[4] Broadening the palette of attacks used may uncover new insights that remain hidden in my experimental design, improving generalisability of results.

[4] This is particularly clear in the results for FGM attacks, where a single step of fixed size is taken. Therefore, in this simple white-box approach, distances are always very close.

6.3.2 Confounds

As addressed in the previous section, training well-performing generative classifiers is not straightforward. I aimed to train all models with stochastic gradient descent (SGD), as this optimisation method is available for both discriminative and generative models. Since eventually some models on image data were trained with EM instead, this introduces a potential confound. After all, choice of optimisation method may influence adversarial robustness. In my study, I cannot rule out the presence of this effect.

6.4 Future work

My first suggestions for future work align with the points raised in the previous sections. In order to generalise beyond the particular models and attacks used in this study, it is advisable to extend the range of models and attacks. One could extend to different PC structures, in particular Poon–Domingos and class-selective SPNs. It is even possible to move beyond the PC formalism and consider other model types as well. Furthermore additional (black-box or white-box) attacks can be taken into account, to assess how different attack algorithms influence the results.

Secondly, the results of this study are not exhausted; they could be analysed further. For instance, in my study, I only looked at the hard-label output of the classifiers, that is, I looked at the top-1 predicted class. New insights may be uncovered, however, by looking at the predicted soft labels too. By considering the soft labels, we can gain information about the confidence the models have for the different classes. Perhaps for poorly performing PCs, the distribution of probabilities over the different classes was close to uniform, which may explain why the model performed poorly: the confidence of the model was low to begin with. Measurements worth considering here are expected calibration error (ECE) and top-k accuracy for $k > 1$.

Another interesting question to explore if what happens if we feed adversarial examples generated for one model into another model. This way we could perform some cross-check to see if the adversarial examples work for different models, or instead are strongly tied to a specific model.

My research has been mostly explorative, in the sense that I try to get a better understanding on the relation between learning mode and robustness. A logical follow-up question is: how can we make (generative) classifiers more robust? This question has been asked before in literature on adversarial robustness, but is seldom asked for PCs specifically. In future work, we can try to develop ways to robustify PCs against noise in the input and adversarial attacks. Some ideas that spring to mind are the use of regularisation in PCs, training on corrupted datasets (also known as adversarial training'), and the use of label smoothing. These things have been done before, but not in the specific setting of generative classifiers and PCs.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., . . . Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/

Baldoni, V., Berline, N., De Loera, J. A., Köppe, M. & Vergne, M. (2010). How to integrate a polynomial over a simplex. *Mathematics of Computation*, *80*(273), 297–325. https://doi.org/10.1090/S0025-5718-2010-02378-6

Brendel, W., Rauber, J. & Bethge, M. (2018, February 16). *Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models*. arXiv: 1712.04248 [cs, stat]. Retrieved January 29, 2021, from http://arxiv.org/abs/1712.04248

Butz, C. J., Oliveira, J. S., Dos Santos, A. E. & Teixeira, A. L. (2019). Deep Convolutional Sum-Product Networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*, 3248–3255. https://doi.org/10.1609/aaai.v33i01.33013248

Carlini, N. & Wagner, D. (2017). Towards Evaluating the Robustness of Neural Networks. *2017 IEEE Symposium on Security and Privacy (SP)*, 39–57. https://doi.org/10.1109/SP.2017.49

Chen, J., Jordan, M. I. & Wainwright, M. J. (2020, April 27). *HopSkipJumpAttack: A Query-Efficient Decision-Based Attack*. arXiv: 1904.02144 [cs, math, stat]. Retrieved August 7, 2021, from http://arxiv.org/abs/1904.02144

Cheng, M., Le, T., Chen, P.-Y., Yi, J., Zhang, H. & Hsieh, C.-J. (2018, July 12). *Query-Efficient Hard-label Black-box Attack: An Optimization-based Approach*. arXiv: 1807.04457 [cs, stat]. Retrieved January 8, 2021, from http://arxiv.org/abs/1807.04457

Choi, A. & Darwiche, A. (2018). On the relative expressiveness of bayesian and neural networks. In V. Kratochvíl & M. Studený (Eds.), *Proceedings of the ninth international conference on probabilistic graphical models* (pp. 157–168). PMLR. http://proceedings.mlr.press/v72/choi18a.html

Choi, Y., Vergari, A. & Van den Broeck, G. (2020). Probabilistic Circuits: Representation and Inference. http://starai.cs.ucla.edu/papers/LecNoAAAI20.pdf

Choi, Y., Vergari, A. & Van den Broeck, G. (2021). Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models. http://starai.cs.ucla.edu/papers/ProbCirc20.pdf

Correia, A. H. C. & de Campos, C. P. (2019). Towards Scalable and Robust Sum-Product Networks. In N. Ben Amor, B. Quost & M. Theobald (Eds.), *Scalable Uncertainty Management* (pp. 409–422). Springer International Publishing. https://doi.org/10.1007/978-3-030-35514-2_31

Correia, A. H. C., Peharz, R. & de Campos, C. P. (2020). Joints in random forests. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 11404–11415). Curran Associates, Inc. https://proceedings.neurips.cc/paper/2020/file/8396b14c5dff55d13eea57487bf8ed26-Paper.pdf

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

Dua, D. & Graff, C. (2019). UCI machine learning repository. http://archive.ics.uci.edu/ml

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T. & Song, D. (2018, April 10). *Robust Physical-World*

Attacks on Deep Learning Models. arXiv: 1707.08945 [cs]. Retrieved November 3, 2021, from http://arxiv.org/abs/1707.08945

Gens, R. & Domingos, P. (2013, May 26). Learning the Structure of Sum-Product Networks. In Sanjoy Dasgupta & David McAllester (Eds.), *Proceedings of the 30th International Conference on Machine Learning* (pp. 873–880). PMLR. https://proceedings.mlr.press/v28/gens13.html

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep learning*. MIT Press. http://www.deeplearningbook.org

Goodfellow, I., Shlens, J. & Szegedy, C. (2015, March 20). *Explaining and Harnessing Adversarial Examples*. arXiv: 1412.6572 [cs, stat]. Retrieved December 9, 2020, from http://arxiv.org/abs/1412.6572

Kolter, Z. & Madry, A. (2018). *Adversarial Robustness - Theory and Practice*. Retrieved December 9, 2020, from http://adversarial-ml-tutorial.org/

Kwisthout, J. (2011). Most probable explanations in Bayesian networks: Complexity and tractability. *International Journal of Approximate Reasoning*, *52*(9), 1452–1469. https://doi.org/10.1016/j.ijar.2011.08.003

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Liu, S., Chen, P.-Y., Kailkhura, B., Zhang, G., Hero, A. & Varshney, P. K. (2020, June 21). *A Primer on Zeroth-Order Optimization in Signal Processing and Machine Learning*. arXiv: 2006.06224 [cs, eess, stat]. Retrieved January 8, 2021, from http://arxiv.org/abs/2006.06224

Martens, J. & Medabalimi, V. (2015, January 22). *On the Expressive Efficiency of Sum Product Networks*. arXiv: 1411.7717 [cs, stat]. Retrieved March 12, 2021, from http://arxiv.org/abs/1411.7717

Mu, N. & Gilmer, J. (2019, June 5). *MNIST-C: A Robustness Benchmark for Computer Vision*. arXiv: 1906.02337 [cs]. Retrieved January 29, 2021, from http://arxiv.org/abs/1906.02337

Mufson, B. (2017). Meet the Artist Using Ritual Magic to Trap Self-Driving Cars [magazine]. *Vice*. Retrieved November 3, 2021, from https://www.vice.com/en/article/qkmeyd/meet-the-artist-using-ritual-magic-to-trap-self-driving-cars

Müller, V. C. (2021). Deep Opacity Undermines Data Protection and Explainable Artificial Intelligence. In C. Zednik & H. Boelsen (Eds.), *Overcoming Opacity in Machine Learning* (pp. 18–21).

Nagel, T. (1974). What Is It Like to Be a Bat? *The Philosophical Review*, *83*(4), 435–450.

Ng, A. & Jordan, M. (2001). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in neural information processing systems*, *14*, 841–848.

Nicolae, M.-I., Sinn, M., Tran, M. N., Buesser, B., Rawat, A., Wistuba, M., Zantedeschi, V., Baracaldo, N., Chen, B., Ludwig, H., Molloy, I. & Edwards, B. (2018). Adversarial robustness toolbox v1.0.0. *CoRR*, *1807.01069*. https://arxiv.org/pdf/1807.01069

Orloff, J. & Bloom, J. (2014). Notational conventions. https://ocw.mit.edu/courses/mathematics/18-05-introduction-to-probability-and-statistics-spring-2014/readings/MIT18_05S14_Reading13b.pdf

París, I., Sánchez-Cauce, R. & Díez, F. J. (2020, April 2). *Sum-product networks: A survey*. arXiv: 2004.01167 [cs]. Retrieved December 8, 2020, from http://arxiv.org/abs/2004.01167

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Peharz, R., Gens, R. & Domingos, P. (2014). Learning Selective Sum-Product Networks. *Proceedings of the 31st International Conference on Machine Learning*, *32*. https://pure.tugraz.at/ws/portalfiles/portal/2673084/selSPN.pdf

Peharz, R., Gens, R., Pernkopf, F. & Domingos, P. (2017). On the latent variable interpretation in sum-product networks. *IEEE Transactions on Machine Intelligence and Pattern Analysis (TPAMI)*, *39*(10), 2030–2044. https://doi.org/10.1109/TPAMI.2016.2618381

Peharz, R., Lang, S., Vergari, A., Stelzner, K., Molina, A., Trapp, M., Van den Broeck, G., Kersting, K. & Ghahramani, Z. (2020). *Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits*. arXiv: 2004.06231 [cs, stat]. http://arxiv.org/abs/2004.06231

Peharz, R., Tschiatschek, S., Pernkopf, F. & Domingos, P. (2015). On Theoretical Properties of Sum-Product Networks. *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, 744–752.

Peharz, R., Vergari, A., Stelzner, K., Molina, A., Trapp, M., Kersting, K. & Ghahramani, Z. (2018). *Probabilistic Deep Learning using Random Sum-Product Networks*. arXiv: 1806.01910 `[cs, stat]`. http://arxiv.org/abs/1806.01910

Plato. (2015). *Socrates' defence* (C. J. Rowe, Trans.). Penguin Books.

Poon, H. & Domingos, P. (2011). Sum-product networks: A new deep architecture. *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 689–690. https://doi.org/10.1109/ICCVW.2011.6130310

Qin, Y., Wang, X., Beutel, A. & Chi, E. H. (2020, June 29). *Improving Uncertainty Estimates through the Relationship with Adversarial Robustness*. arXiv: 2006.16375 `[cs, stat]`. Retrieved December 3, 2020, from http://arxiv.org/abs/2006.16375

Qiu, S., Liu, Q., Zhou, S. & Wu, C. (2019). Review of Artificial Intelligence Adversarial Attack and Defense Technologies. *Applied Sciences*, *9*(5), 909. https://doi.org/10.3390/app9050909

Sharir, O., Tamari, R., Cohen, N. & Shashua, A. (2018, March 25). *Tensorial Mixture Models*. arXiv: 1610.04167 `[cs, stat]`. Retrieved November 8, 2021, from http://arxiv.org/abs/1610.04167

van de Wolfshaar, J. (2019, June 11). *Tensor-Based Sum-Product Networks: Part I*. Retrieved July 8, 2021, from https://jostosh.github.io/spn01/

van de Wolfshaar, J. & Pronobis, A. (2020, September 22). *Deep Generalized Convolutional Sum-Product Networks*. arXiv: 1902.06155 `[cs, stat]`. http://arxiv.org/abs/1902.06155

Vanschoren, J., van Rijn, J. N., Bischl, B. & Torgo, L. (2013). OpenML: Networked science in machine learning. *SIGKDD Explorations*, *15*(2), 49–60. https://doi.org/10.1145/2641190.2641198

Vapnik, V. N. (1998). *Statistical learning theory*. Wiley.

Vergari, A., Choi, Y., Peharz, R. & Van den Broeck, G. (2020, September 14). *Probabilistic Circuits: Representations, Inference, Learning and Theory*. Ghent, Belgium. https://www.youtube.com/watch?v=2RAG5-L9R70&t=5639s

Xiao, H., Rasul, K. & Vollgraf, R. (2017, September 15). *Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv: 1708.07747 `[cs, stat]`. Retrieved November 13, 2021, from http://arxiv.org/abs/1708.07747

Zhao, H., Poupart, P. & Gordon, G. J. (2016). A unified approach for learning the parameters of sum-product networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon & R. Garnett (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/6c9882bbac1c7093bd25041881277658-Paper.pdf

Glossary

AC arithmetic circuit. 23
AI artificial intelligence. 23
ANN artificial neural network. 19, 20, 23, 46–48
BN Bayesian network. 18, 19, 24, 35
C&W Carlini–Wagner. 41–43, 49
CE cross-entropy. 34
CNN convolutional neural network. 30, 46, 48
DAG directed acyclic graph. 23, 24, 32
DGC-SPN deep generalised convolutional sum–product network. 31, 47
distributional shift Input samples come from a different distribution than the training samples. 44
ECE expected calibration error. 43, 62
EiNet Einsum network. 29, 30
EM expectation maximisation. 35, 47, 49, 60, 61
FGM fast gradient method. 41, 42, 49, 57, 61
FGSM fast gradient sign method. 41
GAN generative adversarial network. 35
HPC high performance computing. 61
HSJ HopSkipJump. 43, 49, 55, 58
IBE inference to the best explanation. 26
IV independent variable. 50, 58
LV latent variable. 24
MAP maximum a posteriori. 26
ML machine learning. 5, 17, 18
MLP multi-layer perceptron. 46, 48
MPE most probable explanation. 26, 33
NLL negative log–likelihood. 34
OOD out-of-distribution. 37
PC probabilistic circuit. 5, 20, 23, 24, 29, 30, 46, 47, 61
PDF probability density function. 24
PGM probabilistic graphical model. 20, 23, 25, 26
PMS probability mass function. 24
RAT-SPN random tensorised sum–product network. 32, 47, 48
RV random variable. 15, 24, 25
SGD stochastic gradient descent. 35, 47, 60, 61
SPN sum–product network. 23, 29
VAE variational autoencoder. 35

Appendix A *Model translation table*

For SPNs with a convolutional structure, I create a CNN that is structurally similar. This table explains which CNN equivalent layer I use for each SPN layer.

SPN layer	CNN layer
`NormalLeaf`	`Dense`
`Conv2DProduct` with depthwise convolutions, followed by `Conv2DSum`	`SeparableConv2D`
`Conv2DProduct` with depthwise convolutions, otherwise	`DepthwiseConv2D`
`Conv2DProduct` without depthwise convolutions	`Conv2D`
`Local2DSum`	`LocallyConnected2D` with kernel size 1
`Conv2DSum`	—[1]
`LogDropout`	`Dropout`
`DenseSum`	`Dense`
`SpatialToRegions`	`Flatten`
`RootSum`	—

1. There is no CNN equivalent for `Conv2DSum`, but this is unproblematic because `Conv2DSum` is always preceded by `Conv2DProduct`, which *has* an equivalent.

Constructing matching CNNs for the SPN with similar structure *and* similar number of parameters is very hard. It turns out that models with `LocallyConnected2D` layers, which would be the sensible equivalent for `DenseSum` layers, converge very poorly. Hence in the source base, I use `Dense` instead of `LocallyConnected2D`. But `Dense` only operates on the final dimension of the tensor—the number of channels—which leads to far fewer params.

Appendix B
Models on image data

Reference CNN

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 1600)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1600)              0
_________________________________________________________________
dense_18 (Dense)             (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

Convolutional SPN (large)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
normal_leaf_29 (NormalLeaf)  (None, 28, 28, 4)         6272
_________________________________________________________________
conv2d_product_88 (Conv2DPro (None, 14, 14, 256)       4096
_________________________________________________________________
local2d_sum_68 (Local2DSum)  (None, 14, 14, 256)       12845056
_________________________________________________________________
conv2d_product_89 (Conv2DPro (None, 7, 7, 256)         4
_________________________________________________________________
local2d_sum_69 (Local2DSum)  (None, 7, 7, 512)         6422528
_________________________________________________________________
zero_padding2d_8 (ZeroPaddin (None, 8, 8, 512)         0
_________________________________________________________________
conv2d_product_90 (Conv2DPro (None, 4, 4, 512)         4
_________________________________________________________________
local2d_sum_70 (Local2DSum)  (None, 4, 4, 512)         4194304
_________________________________________________________________
conv2d_product_91 (Conv2DPro (None, 2, 2, 512)         4
_________________________________________________________________
local2d_sum_71 (Local2DSum)  (None, 2, 2, 1024)        2097152
_________________________________________________________________
conv2d_product_92 (Conv2DPro (None, 1, 1, 1024)        4
_________________________________________________________________
log_dropout_5 (LogDropout)   (None, 1, 1, 1024)        0
_________________________________________________________________
dense_sum_44 (DenseSum)      (None, 1, 1, 10)          10240
_________________________________________________________________
root_sum_29 (RootSum)        (None, 10)                10
=================================================================
Total params: 25,579,674
Trainable params: 25,575,552
Non-trainable params: 4,122
_________________________________________________________________
```

CNN equivalent:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 14, 14, 256)       1280
_________________________________________________________________
dense_10 (Dense)             (None, 14, 14, 256)       65792
_________________________________________________________________
depthwise_conv2d_9 (Depthwis (None, 7, 7, 256)         1280
_________________________________________________________________
dense_11 (Dense)             (None, 7, 7, 512)         131584
_________________________________________________________________
zero_padding2d_8 (ZeroPaddin (None, 8, 8, 512)         0
_________________________________________________________________
depthwise_conv2d_10 (Depthwi (None, 4, 4, 512)         2560
_________________________________________________________________
dense_12 (Dense)             (None, 4, 4, 512)         262656
_________________________________________________________________
depthwise_conv2d_11 (Depthwi (None, 2, 2, 512)         2560
_________________________________________________________________
dense_13 (Dense)             (None, 2, 2, 1024)        525312
_________________________________________________________________
depthwise_conv2d_12 (Depthwi (None, 1, 1, 1024)        5120
_________________________________________________________________
flatten_3 (Flatten)          (None, 1024)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_14 (Dense)             (None, 10)                10250
=================================================================
Total params: 1,008,394
Trainable params: 1,008,394
Non-trainable params: 0
_________________________________________________________________
```

Convolutional SPN (small)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
normal_leaf_14 (NormalLeaf)  (None, 28, 28, 4)         6272
_________________________________________________________________
conv2d_product_49 (Conv2DPro (None, 14, 14, 4)         4
_________________________________________________________________
local2d_sum_37 (Local2DSum)  (None, 14, 14, 32)        25088
_________________________________________________________________
conv2d_product_50 (Conv2DPro (None, 7, 7, 32)          4
_________________________________________________________________
local2d_sum_38 (Local2DSum)  (None, 7, 7, 64)          100352
_________________________________________________________________
zero_padding2d_5 (ZeroPaddin (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_product_51 (Conv2DPro (None, 4, 4, 64)          4
_________________________________________________________________
```

```
conv2d_sum_28 (Conv2DSum)    (None, 4, 4, 128)        8192
_________________________________________________________________
conv2d_product_52 (Conv2DPro (None, 2, 2, 128)        4
_________________________________________________________________
conv2d_sum_29 (Conv2DSum)    (None, 2, 2, 256)        32768
_________________________________________________________________
conv2d_product_53 (Conv2DPro (None, 1, 1, 256)        4
_________________________________________________________________
dense_sum_14 (DenseSum)      (None, 1, 1, 10)         2560
_________________________________________________________________
root_sum_14 (RootSum)        (None, 10)               10
=================================================================
Total params: 175,262
Trainable params: 175,232
Non-trainable params: 30
_________________________________________________________________
```

CNN equivalent:

```
_________________________________________________________________
Layer (type)                 Output Shape             Param #
=================================================================
depthwise_conv2d_29 (Depthwi (None, 14, 14, 4)        20
_________________________________________________________________
dense_33 (Dense)             (None, 14, 14, 32)       160
_________________________________________________________________
depthwise_conv2d_30 (Depthwi (None, 7, 7, 32)         160
_________________________________________________________________
dense_34 (Dense)             (None, 7, 7, 64)         2112
_________________________________________________________________
zero_padding2d_19 (ZeroPaddi (None, 8, 8, 64)         0
_________________________________________________________________
separable_conv2d_8 (Separabl (None, 4, 4, 128)        8576
_________________________________________________________________
separable_conv2d_9 (Separabl (None, 2, 2, 256)        33536
_________________________________________________________________
depthwise_conv2d_31 (Depthwi (None, 1, 1, 256)        1280
_________________________________________________________________
flatten_9 (Flatten)          (None, 256)              0
_________________________________________________________________
dense_35 (Dense)             (None, 10)               2570
=================================================================
Total params: 48,414
Trainable params: 48,414
Non-trainable params: 0
_________________________________________________________________
```

DGC-SPN (large)

```
_________________________________________________________________
Layer (type)                 Output Shape             Param #
=================================================================
normal_leaf_20 (NormalLeaf)  (None, 28, 28, 16)       25088
_________________________________________________________________
conv2d_product_66 (Conv2DPro (None, 14, 14, 16)       4
_________________________________________________________________
local2d_sum_49 (Local2DSum)  (None, 14, 14, 16)       50176
_________________________________________________________________
conv2d_product_67 (Conv2DPro (None, 7, 7, 16)         4
_________________________________________________________________
local2d_sum_50 (Local2DSum)  (None, 7, 7, 32)         25088
_________________________________________________________________
conv2d_product_68 (Conv2DPro (None, 8, 8, 32)         4
_________________________________________________________________
local2d_sum_51 (Local2DSum)  (None, 8, 8, 32)         65536
_________________________________________________________________
conv2d_product_69 (Conv2DPro (None, 10, 10, 32)       4
_________________________________________________________________
local2d_sum_52 (Local2DSum)  (None, 10, 10, 64)       204800
_________________________________________________________________
conv2d_product_70 (Conv2DPro (None, 14, 14, 64)       4
_________________________________________________________________
local2d_sum_53 (Local2DSum)  (None, 14, 14, 64)       802816
_________________________________________________________________
conv2d_product_71 (Conv2DPro (None, 8, 8, 64)         4
_________________________________________________________________
spatial_to_regions_11 (Spati (None, 1, 1, 4096)       0
_________________________________________________________________
dense_sum_35 (DenseSum)      (None, 1, 1, 10)         40960
_________________________________________________________________
root_sum_20 (RootSum)        (None, 10)               10
=================================================================
Total params: 1,214,498
Trainable params: 1,201,920
Non-trainable params: 12,578
_________________________________________________________________
```

CNN equivalent:

```
_________________________________________________________________
Layer (type)                 Output Shape             Param #
=================================================================
depthwise_conv2d_16 (Depthwi (None, 14, 14, 16)       80
_________________________________________________________________
dense_19 (Dense)             (None, 14, 14, 16)       272
_________________________________________________________________
depthwise_conv2d_17 (Depthwi (None, 7, 7, 16)         80
_________________________________________________________________
dense_20 (Dense)             (None, 7, 7, 32)         544
_________________________________________________________________
zero_padding2d_10 (ZeroPaddi (None, 9, 9, 32)         0
_________________________________________________________________
depthwise_conv2d_18 (Depthwi (None, 8, 8, 32)         160
_________________________________________________________________
dense_21 (Dense)             (None, 8, 8, 32)         1056
_________________________________________________________________
zero_padding2d_11 (ZeroPaddi (None, 12, 12, 32)       0
_________________________________________________________________
depthwise_conv2d_19 (Depthwi (None, 10, 10, 32)       160
_________________________________________________________________
dense_22 (Dense)             (None, 10, 10, 64)       2112
_________________________________________________________________
zero_padding2d_12 (ZeroPaddi (None, 18, 18, 64)       0
_________________________________________________________________
depthwise_conv2d_20 (Depthwi (None, 14, 14, 64)       320
_________________________________________________________________
dense_23 (Dense)             (None, 14, 14, 64)       4160
_________________________________________________________________
zero_padding2d_13 (ZeroPaddi (None, 16, 16, 64)       0
_________________________________________________________________
depthwise_conv2d_21 (Depthwi (None, 8, 8, 64)         320
_________________________________________________________________
flatten_6 (Flatten)          (None, 4096)             0
_________________________________________________________________
dense_24 (Dense)             (None, 10)               40970
=================================================================
Total params: 50,234
Trainable params: 50,234
Non-trainable params: 0
_________________________________________________________________
```

DGC-SPN (small)

```
_________________________________________________________________
Layer (type)                 Output Shape             Param #
```

```
=================================================================
normal_leaf_26 (NormalLeaf)  (None, 28, 28, 4)        6272
_________________________________________________________________
conv2d_product_76 (Conv2DPro (None, 14, 14, 4)        4
_________________________________________________________________
local2d_sum_58 (Local2DSum)  (None, 14, 14, 32)       25088
_________________________________________________________________
conv2d_product_77 (Conv2DPro (None, 7, 7, 32)         4
_________________________________________________________________
local2d_sum_59 (Local2DSum)  (None, 7, 7, 64)         100352
_________________________________________________________________
conv_overlapping_1 (Conv2DPr (None, 8, 8, 64)         4
_________________________________________________________________
conv2d_sum_51 (Conv2DSum)    (None, 8, 8, 64)         4096
_________________________________________________________________
conv_overlapping_2 (Conv2DPr (None, 10, 10, 64)       4
_________________________________________________________________
conv2d_sum_52 (Conv2DSum)    (None, 10, 10, 64)       4096
_________________________________________________________________
conv_overlapping_3 (Conv2DPr (None, 14, 14, 64)       4
_________________________________________________________________
conv2d_sum_53 (Conv2DSum)    (None, 14, 14, 128)      8192
_________________________________________________________________
conv_overlapping_final (Conv (None, 8, 8, 128)        4
_________________________________________________________________
spatial_to_regions_17 (Spati (None, 1, 1, 8192)       0
_________________________________________________________________
dense_sum_41 (DenseSum)      (None, 1, 1, 10)         81920
_________________________________________________________________
root_sum_26 (RootSum)        (None, 10)               10
=================================================================
Total params: 230,050
Trainable params: 230,016
Non-trainable params: 34
_________________________________________________________________
```

CNN equivalent:

```
_________________________________________________________________
Layer (type)                 Output Shape             Param #
=================================================================
depthwise_conv2d_22 (Depthwi (None, 14, 14, 4)        20
_________________________________________________________________
dense_25 (Dense)             (None, 14, 14, 32)       160
_________________________________________________________________
depthwise_conv2d_23 (Depthwi (None, 7, 7, 32)         160
_________________________________________________________________
dense_26 (Dense)             (None, 7, 7, 64)         2112
_________________________________________________________________
zero_padding2d_14 (ZeroPaddi (None, 9, 9, 64)         0
_________________________________________________________________
separable_conv2d_5 (Separabl (None, 8, 8, 64)         4416
_________________________________________________________________
zero_padding2d_15 (ZeroPaddi (None, 12, 12, 64)       0
_________________________________________________________________
separable_conv2d_6 (Separabl (None, 10, 10, 64)       4416
_________________________________________________________________
zero_padding2d_16 (ZeroPaddi (None, 18, 18, 64)       0
_________________________________________________________________
separable_conv2d_7 (Separabl (None, 14, 14, 128)      8576
_________________________________________________________________
zero_padding2d_17 (ZeroPaddi (None, 16, 16, 128)      0
_________________________________________________________________
depthwise_conv2d_24 (Depthwi (None, 8, 8, 128)        640
_________________________________________________________________
flatten_7 (Flatten)          (None, 8192)             0
_________________________________________________________________
dense_27 (Dense)             (None, 10)               81930
```

```
=================================================================
Total params: 102,430
Trainable params: 102,430
Non-trainable params: 0
_________________________________________________________________
```

RAT-SPN (large)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_5 (Flatten)          (None, 784)               0
_________________________________________________________________
flat_to_regions_5 (FlatToReg (None, 784, 10, 1)        0
_________________________________________________________________
normal_leaf_35 (NormalLeaf)  (None, 784, 10, 32)       501760
_________________________________________________________________
permute_and_pad_scopes_rando (None, 1024, 10, 32)      10240
_________________________________________________________________
reduce_product_10 (ReducePro (None, 128, 10, 32)       0
_________________________________________________________________
dense_sum_60 (DenseSum)      (None, 128, 10, 8)        327680
_________________________________________________________________
reduce_product_11 (ReducePro (None, 16, 10, 8)         0
_________________________________________________________________
dense_sum_61 (DenseSum)      (None, 16, 10, 8)         10240
_________________________________________________________________
dense_product_20 (DenseProdu (None, 8, 10, 64)         0
_________________________________________________________________
dense_sum_62 (DenseSum)      (None, 8, 10, 8)          40960
_________________________________________________________________
dense_product_21 (DenseProdu (None, 4, 10, 64)         0
_________________________________________________________________
dense_sum_63 (DenseSum)      (None, 4, 10, 8)          20480
_________________________________________________________________
dense_product_22 (DenseProdu (None, 2, 10, 64)         0
_________________________________________________________________
dense_sum_64 (DenseSum)      (None, 2, 10, 8)          10240
_________________________________________________________________
dense_product_23 (DenseProdu (None, 1, 10, 64)         0
_________________________________________________________________
dense_sum_65 (DenseSum)      (None, 1, 10, 1)          640
_________________________________________________________________
undecompose_5 (Undecompose)  (None, 1, 1, 10)          0
_________________________________________________________________
root_sum_35 (RootSum)        (None, 10)                10
=================================================================
Total params: 922,250
Trainable params: 912,000
Non-trainable params: 10,250
_________________________________________________________________
```

MLP equivalent:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 356)               279460
_________________________________________________________________
dense_1 (Dense)              (None, 356)               127092
_________________________________________________________________
dense_2 (Dense)              (None, 356)               127092
_________________________________________________________________
dense_3 (Dense)              (None, 356)               127092
```

```
_________________________________________________________________
dense_4 (Dense)              (None, 356)               127092
_________________________________________________________________
dense_5 (Dense)              (None, 356)               127092
_________________________________________________________________
dense_6 (Dense)              (None, 10)                3570
=================================================================
Total params: 918,490
Trainable params: 918,490
Non-trainable params: 0
_________________________________________________________________
```

Appendix C
Models on empirical tabular data

In the grid search explained in § 4.2.3, these are the optimal paramet-
ers I found:

		MLP				RAT-SPN							
				Baseline						Discriminative		Generative	
Dataset	D	H	# params	Acc.	Loss	R	I	S	# params	Acc.	Loss	Acc.	Loss
authent	2	215	47 947	1.000	0.007	30	4	21	48 182	1.000	0.029	1.000	3.073
	3	—	—	—	—	—	—	—	—	—	—	—	—
	4	—	—	—	—	—	—	—	—	—	—	—	—
diabetes	2	391	157 575	0.750	0.457	40	14	9	158 162	0.776	0.472	0.784	9.291
	3	183	69 359	0.750	0.452	10	8	12	69 842	0.724	0.513	0.784	8.920
	4	—	—	—	—	—	—	—	—	—	—	—	—
gesture	2	150	28 355	0.559	1.133	10	12	6	28 365	0.534	1.179	0.436	14.528
	3	356	267 717	0.594	1.124	50	4	12	268 005	0.544	1.157	0.446	11.438
	4	1 368	5 670 365	0.599	1.150	50	18	21	5 675 755	0.544	1.157	0.447	11.544
jungle	2	194	39 773	0.875	0.245	20	12	6	40 243	0.864	0.285	0.685	9.104
	3	235	113 273	0.889	0.188	10	8	15	113 933	0.858	0.308	0.696	9.227
	4	—	—	—	—	—	—	—	—	—	—	—	—
robot	2	668	466 268	0.938	0.173	50	18	12	466 804	0.949	0.137	0.806	0.333
	3	767	1 200 359	0.946	0.182	50	12	18	1 201 204	0.944	0.169	0.783	0.525
	4	627	1 199 455	0.933	0.190	50	16	9	1 201 304	0.949	0.149	0.748	0.778
segment	2	233	60 587	0.942	0.168	40	6	9	60 927	0.937	0.231	0.876	-2.287
	3	699	996 781	0.960	0.184	40	6	21	997 807	0.954	0.222	0.844	-2.066
	4	926	2 599 289	0.954	0.204	40	6	21	2 600 847	0.951	0.194	0.850	-1.820
texture	2	682	501 281	0.999	0.005	50	10	21	502 561	0.995	0.033	0.898	6.238
	3	672	939 467	0.999	0.003	50	16	12	940 411	0.993	0.070	0.886	6.602
	4	1 173	4 192 313	0.999	0.004	40	16	21	4 196 531	0.895	0.172	0.891	5.563
vehicle	2	341	124 469	0.819	0.311	50	8	12	124 604	0.787	0.469	0.701	8.343
	3	847	1 455 997	0.858	0.282	50	10	21	1 457 104	0.780	0.495	0.693	8.792
	4	1 110	3 725 164	0.858	0.310	50	10	21	3 729 704	0.748	0.516	0.677	8.628
wdbc	2	303	102 113	0.977	0.151	30	14	6	102 482	0.965	0.121	0.977	24.627
	3	645	854 627	0.977	0.132	50	16	12	857 202	0.977	0.138	0.977	23.717
	4	617	1 164 281	0.977	0.261	30	12	15	1 165 562	0.965	0.168	0.965	24.300

The number of hidden units H is set in order to closely match the number of parameters in the RAT-SPN. RAT-SPN hyperparameters are explained in § 2.2.2. Empty lines denote models that cannot be built due to there being too few input features.

Appendix D
Models on synthetic tabular data

For models trained on synthetic data, the following hyperparameters are used:

	MLP		RAT-SPN			
D	*H*	# params	*R*	*I*	*S*	# params
2	256	73 738	32	10	6	65 930
3		139 530				118 282
4		205 322				222 986

This yields the following performance:

		Baseline		Discriminative		Generative	
Configuration	*D*	Acc.	Loss	Acc.	Loss	Acc.	Loss
Default	2	0.935	0.243	0.928	0.267	0.901	25.698
	3	0.936	0.239	0.930	0.268	0.904	25.524
	4	0.936	0.251	0.926	0.276	0.906	25.420
Fewer samples	2	0.891	0.334	0.898	0.424	0.753	26.742
	3	0.889	0.396	0.900	0.387	0.744	26.714
	4	0.887	0.409	0.871	0.441	0.740	26.864
More samples	2	0.961	0.171	0.951	0.182	0.894	25.748
	3	0.962	0.169	0.953	0.179	0.897	25.824
	4	0.963	0.171	0.950	0.190	0.898	25.817
More informative features	2	0.981	0.116	0.959	0.162	0.930	25.234
	3	0.982	0.119	0.962	0.153	0.936	25.063
	4	0.981	0.127	0.960	0.156	0.929	25.065
No redundant features	2	0.945	0.237	0.944	0.211	0.889	13.186
	3	0.946	0.223	0.944	0.212	0.876	12.729
	4	0.948	0.216	0.946	0.208	0.886	12.625
More redundant features	2	0.920	0.309	0.906	0.340	0.875	28.054
	3	0.918	0.322	0.907	0.340	0.878	28.024
	4	0.912	0.333	0.911	0.329	0.839	28.074

Appendix E
Accuracy on noisy datasets

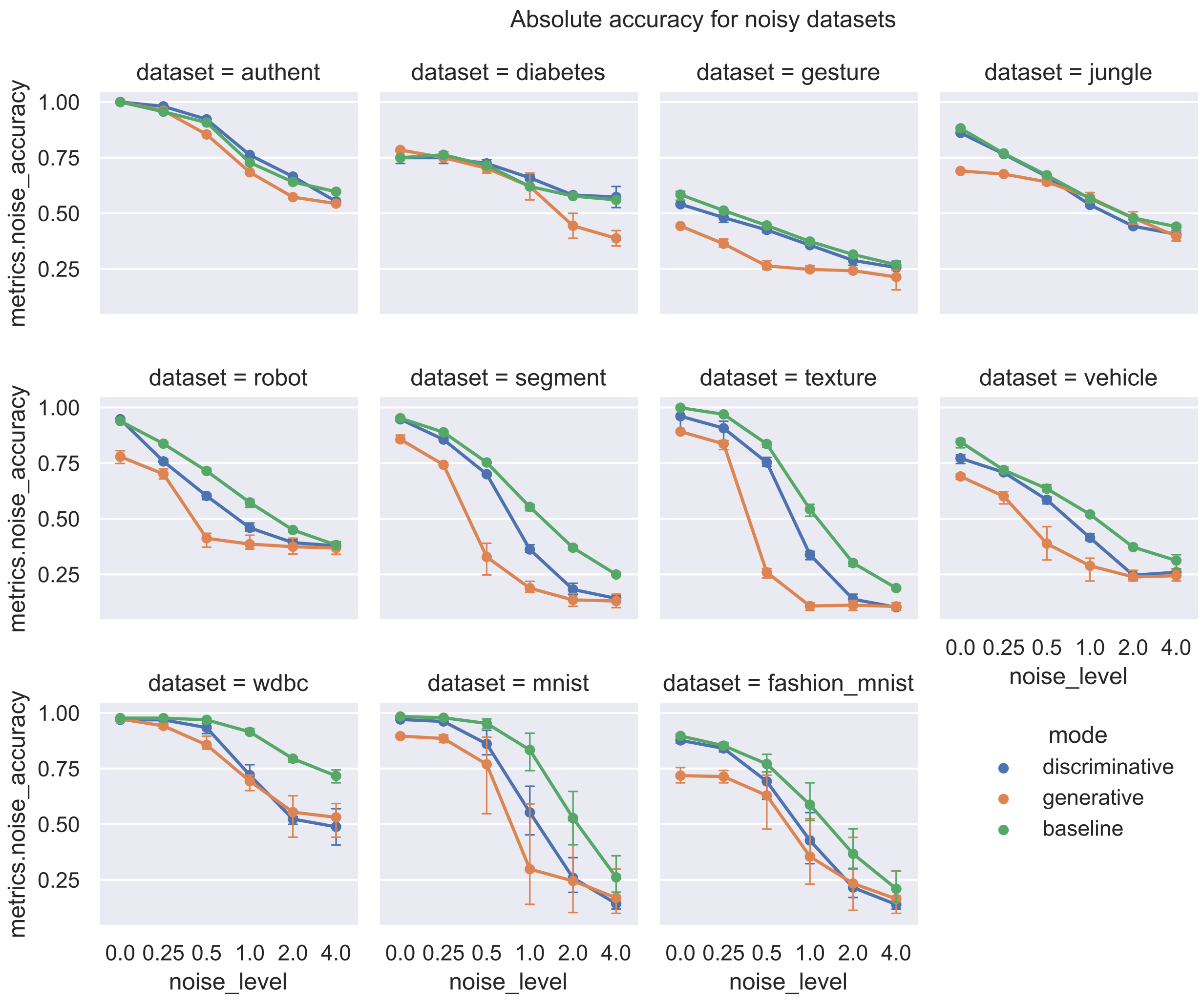


Figure E.1: Absolute classification accuracy on noisy datasets. The error bars represent 95% confidence interval on the observations.

Appendix F *Adversarial distance for empirical tabular datasets*

These are the adversarial distances found for the empirical tabular datasets, for different model depths D.

		CarliniWagner			FastGradient			HopSkipJump		
Dataset	D	Base.	Disc.	Gen.	Base.	Disc.	Gen.	Base.	Disc.	Gen.
authent	2	1.14	1.16	1.43	0.29	0.30	0.30	0.84	0.97	0.90
diabetes	2	1.23	1.34	0.88	0.30	0.30	0.30	1.36	1.95	1.09
	3	1.20	1.44	0.84	0.30	0.30	0.30	1.30	1.74	1.22
gesture	2	0.55	1.62	1.80	0.30	0.30	0.30	0.79	0.90	1.47
	3	0.54	1.08	2.04	0.30	0.30	0.30	0.67	0.85	1.23
	4	1.01	0.67	2.58	0.29	0.29	0.30	0.72	0.75	1.30
jungle	2	0.51	0.50	0.96	0.27	0.28	0.23	0.61	0.61	1.26
	3	0.46	0.54	0.98	0.23	0.29	0.23	0.55	0.59	1.18
robot	2	0.84	0.47	2.77	0.30	0.30	0.30	0.83	0.66	1.72
	3	0.92	0.83	3.12	0.27	0.29	0.30	0.83	0.62	1.34
	4	1.08	0.82	0.37	0.29	0.30	0.30	0.83	0.72	0.69
segment	2	1.15	2.62	2.61	0.30	0.30	0.30	1.42	1.25	1.09
	3	3.13	5.91	6.65	0.26	0.29	0.30	1.36	1.13	1.13
	4	3.24	4.20	6.16	0.20	0.29	0.30	1.44	1.15	1.77
texture	2	4.75	2.33	2.40	0.26	0.29	0.30	1.36	1.29	1.81
	3	5.91	3.81	3.45	0.21	0.27	0.30	1.36	1.25	2.06
	4	5.91	4.46	3.74	0.18	0.25	0.30	1.24	1.18	1.61
vehicle	2	0.91	1.74	1.20	0.30	0.30	0.30	0.68	0.90	1.51
	3	1.69	1.45	1.77	0.30	0.30	0.30	0.70	0.95	0.85
	4	1.98	1.25	2.02	0.27	0.30	0.30	0.67	0.81	0.57
wdbc	2	4.57	7.39	4.84	0.23	0.30	0.30	1.80	2.94	2.42
	3	2.39	4.70	6.77	0.16	0.29	0.30	1.42	2.61	3.77
	4	2.42	6.77	3.08	0.09	0.28	0.30	1.37	3.08	3.36

Note that for FastGradient, the results are consistently very close to 0.3. This is explained by the nature of the attack algorithm (see § 3.3.1), in which a single step of size ϵ is taken. In my experiments, $\epsilon = 0.3$ (see § 4.4).