# Learning Latent Mappings to Satisfy Constrained Optimisation Problems

**Alexander Atack - 27745449**

April 2019

Supervised by Dr. David Toal

9969 words

This report is submitted in partial fulfillment of the requirements for the Aeronautics & Astronautics MEng, Faculty of Engineering and the Environment, University of Southampton.

I, Alexander Atack, declare that this thesis and the work presented in it are my own and has been generated by me as a result of my own original research. I confirm that:

1. this work has been done wholly or mainly while in candidature for a degree at this University;
2. where any part of this thesis has previously been submitted for any other qualification at this University or any other institution, this has been clearly stated;
3. where I have consulted the published work of others, this is always clearly attributed;
4. where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. none of this work has been published before submission.

**Abstract**

Optimisation algorithms which are often used in engineering are not well suited to avoiding specific disallowed solutions such as those with degenerate geometries. This project introduces an adaptation of the generative adversarial network architecture in which a discriminator is trained to determine the viability of solutions, and a generator is trained to sample from a range of solutions that satisfy the discriminator. The architecture of the generator is also adapted so that its output can be parameterised by a constraint describing the space of viable solutions. In so doing, a latent space parameterised by a constraint vector is learned, allowing optimisation algorithms to explore only those solutions which will satisfy the constraint.

Promising results are obtained, showing that the generator is capable of learning effective mappings that satisfy a range of constraints, including equality constraints. Some limitations of the proposed architecture are also explored, and the reasons behind them identified for improvement by future work.

# Contents

# List of Figures

# List of Tables

# Nomenclature

## Symbols

| | |
|---|---|
| $f(s)$ | Objective function |
| $h(c, s)$ | Constraint satisfaction function |
| $s$ | Solution vector |
| $c$ | Constraint vector |
| $m$ | Dimension of the solution space |
| $n$ | Dimension of the constraint space |
| $S$ | Solution space |
| $C$ | Constraint space |
| $a_X, b_X$ | Lower and upper bounds of a space in each dimension |
| $V_c$ | Viable space of a constraint |
| $g(l, c)$ | Generator function |
| $l$ | Coordinate in the latent space |
| $L$ | Latent space |
| $k$ | Dimension of the latent space |
| $V_c'$ | Learned viable space of a constraint |
| $g'(l, c)$ | Learned generator function |
| $g'^{-1}(s, c)$ | Inverse of the learned generator function |
| $h'(c, s)$ | Discriminator |
| $p(g')$ | Precision of a generator function |
| $r(g')$ | Recall of a generator function |
| $w$ | Recall weight |
| $\mathcal{L}(g')$ | Generator loss function |
| $\hat{X}$ | Probability distribution function for a space |
| $\gamma$ | Probability density of valid solutions to a constraint |
| $D_{\mathrm{KL}}$ | KL-divergence |
| $\hat{p}(g')$ | Precision proxy defined for distributions |

| | |
|---|---|
| $\hat{r}(g')$ | Recall proxy defined for distributions |
| $\epsilon$ | Small value to prevent numerical errors |
| $q_{\text{id}}$ | Identity loss |
| $q_{\text{sep}}$ | Separation loss |
| $R$ | Relevant space |
| $\rho$ | Relative density |
| $\beta(x, y)$ | Branin function |
| $\beta'(s)$ | Rescaled Branin function |

## Abbreviations

| | |
|---|---|
| CSF | Constraint satisfaction function |
| ANN | Artificial neural network |
| DNN | Deep neural network |
| RNN | Recursive neural network |
| GAN | Generative adversarial network |
| VAE | Variational autoencoder |
| p.d.f. | Probability density function |
| DQN | Deep $Q$-Network |
| ESF | Expected satisfaction probability |
| MSF | Mean satisfaction probability |
| MCMC | Markov chain Monte Carlo |

# Chapter 1

# Introduction

## 1.1 Motivation

Many engineering challenges can be expressed as optimisation problems, in which the objective is to search a space of possibilities for a value which optimises some objective function. In real world problems, the objective function is typically complicated enough that its maximum cannot be found analytically; in these cases, it is necessary to explore the solution space in order to find the optimal solution. A number of algorithms exist to search this space efficiently, but often converge slowly when the error surface of the objective function changes abruptly. This will often be the case when trying to optimise continuous properties, such as minimising the mass of a design, while simultaneously adhering to constraints, such as avoiding degenerate geometries.

Machine learning excels in exploiting underlying patterns in data to make predictions in the face of uncertainty. This project explores the use of machine learning in constructing a differentiable mapping, parameterised by a constraint, between an arbitrary latent space and the space of solutions to a problem such that any point in the latent space, when mapped to the solution space, satisfies the constraint.

## 1.2 Engineering problems

For the purposes of this project, an engineering problem is defined as an optimisation problem in which a solution $s$ is sought which optimises an arbitrary objective function $f(s)$, while also satisfying a constraint $c$. Constraint satisfaction is determined by some function $h(c, s)$ which maps a constraint and solution to $\{\text{satisfied}, \text{unsatisfied}\}$. A solution which satisfies $h$ is referred to as viable.

The functions $f$ and $h$ are deterministic and belong to an engineering environment, which is thought of as an idealised mathematical model of a particular real-world problem. While each environment has a unique constraint satisfaction function $h$, they may define a number of different continuous objective functions.

It is also assumed that both the solution and constraint are parameterised by a vector bounded by an hypercube in $m$ or $n$ dimensions respectively, denoted $S$ and $C$. $h(c, s)$ is defined as long as $c \in C$ and $s \in S$.

$$S = \{[s_1, ..., s_m]^T \mid a_S \leq s_i \leq b_S \; \forall \; 1 \leq i \leq m\} \tag{1.1}$$

$$C = \{[c_1, ..., c_n]^T \mid a_C \leq c_i \leq b_C \; \forall \; 1 \leq i \leq n\} \tag{1.2}$$

where $a$, $b$ are the arbitrary bounds of the hypercube in each dimension.

Since both $S$ and $C$ are hypercubes, it is trivial to generate vectors which lie within them (Appendix B). As such, solving an engineering problem as defined here for a given constraint $c$ can be broken down into two steps: finding a solution $s$ for which $f(s)$ is suitably large; and ensuring that $s$ is viable.

## 1.3 Learned latent mapping

The first step can already be achieved with conventional optimisation algorithms which have no knowledge of the objective surface. When certain regions of the solution space do not satisfy the constraint, however, optimisation algorithms which work by exploring the solution space are unable to work efficiently since there is no natural way of reconciling the satisfaction of a constraint with a continuous objective function other than evaluating $h$ for every candidate solution. If the regions satisfying the constraint are sparse, it may be a considerable amount of time before even a single viable solution is found, before then having to optimise its value according to the objective function.

One way around this dilemna is to utilise the fact that $h$ is constant within an environment to learn its underlying patterns using data from previous experiments. This knowledge can be used to create a space of viable solutions $V_c$, specific to the constraint, such that any solution vector which is contained within $V_c$ satisfies $c$ by definition.

$$V_c = \{s \mid s \in S, \; h(c, s) = \text{satisfied}\} \tag{1.3}$$

While envisaging such a set is trivial, sampling from it is not. One can imagine a generator function $g(c)$ which maps from an arbitrary latent space $L$ (from which samples can be easily drawn) to somewhere in $V_c$:

$$g(l, c) \; : \; L, C \mapsto V_c \tag{1.4}$$

$$L = \{[l_1, ..., l_k]^T \mid a_L \leq l_i \leq b_L \; \forall \; 1 \leq i \leq k\} \tag{1.5}$$

$g$ can therefore be viewed as the inverse of $h$. Once $V$ and $g$ have been learned, solving the engineering problem for a constraint is a matter of performing optimisation over $l \in L$ of $(f \circ g)\ (l, c)$ using any exploratory optimisation algorithm. As such, finding a way of modelling $V_c$ and $g$ for any environment is the subject of this project.

# Chapter 2

# Previous Research

## 2.1 Naïve optimisation techniques

A number of algorithms exist for finding the maximum of a function. Many of these are gradient descent algorithms [1] which use information about the local gradient of the function to iteratively maximise it. Because the gradient is used to update the current guess, these algorithms often fail to converge quickly on functions whose surfaces are flat. Algorithms such as RMSProp [2] and Adam [3] have been developed to overcome this, but still fail to converge if the function is discretised.

Gradient-free, stochastic optimisation algorithms such as particle swarm optimisation [4] and genetic algorithms [5] solve this problem for functions of small dimensionality by rapidly exploring the input space, then focusing on areas which give good results. But the expected runtime of such algorithms drops off exponentially as the dimensionality of the function increases.

These algorithms are typically suitable for finding the maximum of a function given that any input is viable, and can therefore be used to solve the first part of the generic engineering problem described in §1.2. Crucially, they do not utilise information learned about the engineering environment, instead treating every new instance of its engineering problem as unseen, and as such are considered to be naïve.

## 2.2 Artificial neural networks

A function $g$ mapping from $L$ to $V_c$ takes two arguments: the constraint to be satisfied, and a point in the latent space. This function is a property of the engineering environment, and individual engineering problems within that environment can be considered instances or parameterisations of this function for which the constraint is fixed. It is therefore expected that a universal function approximator will be capable of learning $g$.

Artificial neural networks (ANNs) are known to be universal function approximators [6]; in particular, deep neural networks (DNNs) [7] are capable of approximating highly intractable functions.

The performance of ANNs and DNNs is limited by the quantity of data available [8]. While larger networks with more capacity are able to learn more complex mappings, they also run a higher risk of overfitting [9], a problem exacerbated when the quantity of training data is limited.

Regularisation techniques have been developed [10, 11, 12, 13] to prevent the overfitting of neural networks. These techniques generally rely on reducing the reliance of the network on singular large weights, making it more difficult for the network to memorise individual data points, thereby encouraging a more general mapping.

Adding noise to the network's input [14] has also been shown to decrease overfitting and increase the generality of the network's estimates, especially when the available data are sparsely spread throughout the feature space. The effect of adding noise to the input is to simulate a greater quantity of data, but assumes that inputs which are nearby in the feature space will produce similar outputs. The level of noise is another hyperparameter to be optimised: too little noise will have no effect on overfitting, while too much noise will make it impossible for the network to distinguish between noise and genuine information, causing a drop in accuracy.

## 2.3 Function inversion

As well as approximating mappings between spaces, ANNs are capable of learning the inverse of an existing function. Promising results have been obtained using ANNs to invert intractable functions in image processing, for uses such as deblurring [15] and colourisation [16].

Autoencoders are an unsupervised learning method used to pretrain layers of neural networks that will later be applied for supervised learning [17], in doing so training an encoder $f(x)$ and a decoder $f^{-1}(x)$. This can be used to extract the more important latent features of the feature space, or remove noise from corrupted data [18]. When used for pretraining, the decoder is normally discarded; but if the encoding function $f(x)$ is already known, the autoencoder architecture can be used to learn its inverse.

Other architectures have been proposed which perform the same task as a traditional autoencoder using a generative model. Variational autoencoders (VAEs) allow sampling from the latent space by parameterising a probability distribution over the it [19], predicting the latent variables most likely to explain the visible data. The most common use for this is learning a robust latent mapping for a dataset, but the method could be adapted to create a generative model that inverts a many-to-one function such as $h$.

A common problem faced by traditional autoencoders is that not every point in the latent space has a meaningful equivalent in the feature space. A regularisation term is imposed on a VAE in the $\beta$-VAE architecture [20] that encourages a higher standard deviation of the latent distribution. The result is that inputs are mapped to a greater variety of the latent space, and therefore a greater

proportion of the latent space will have a meaningful inverse mapping.

## 2.4 Adversarial algorithms

More recently, algorithms have been designed to train generative models using two competing networks in a minimax game. The most common adversarial networks, generative adversarial networks (GANs), can be trained to sample from an arbitrary distribution [21, 22]. These have been applied to a range of problems, including image resolution upscaling [23] and creating cross-domain transfer functions [24]. GANs may therefore be adapted to sample from a distribution modelling $V_c$, if a parameterisation of the distribution by the relevant constraint can be learned.

Some research has been done into the using GANs to invert functions [25]; while autoencoders can also accomplish this task and are generally considered to be easier to train [26], the generative nature of GANs is appealing considering that $V_c$ contains more than one point. Sampling from it is therefore a necessity.

## 2.5 Representation learning

All ANNs discussed previously require the input data to be input as a fixed-size vector of real values. While many practical data structures can be meaningfully encoded in this form (images, coordinates), many others cannot (words, time series).

Recursive neural networks (RNNs) merge information about pairs of objects, allowing the construction of representations of tree-like data structures [27]. Because the information capacity of the final state vector is finite by virtue of its fixed size, the network is forced to either generalise or forget unimportant information.

Some data structures, known as tokens, cannot be represented by real values and do not consist of sub-elements that can. The prime example of these data structures are words: each word carries meaning which is not encoded by its letters. Vector space models represent tokens as points in a vector space, and methods have been developed for learning these embeddings. One method commonly used is *word2vec* [28], which embeds words into a dense vector space, though the same principle has been adapted for use on other tokens [29].

# Chapter 3

# Theoretical Formulation

## 3.1 Solution and constraint spaces

As stated in §1, the object of this project is to find a way of sampling the space of viable solutions for a constraint, $V_c$. Let $g'(l, c)$ be the learned generator function, as opposed to the true generator function $g$. Then let $V_c'$ be the learned viable space: the space of all solutions which can be obtained by passing a valid latent coordinate through $g'$:

$$V_c' = \{s \mid g'^{-1}(s, c) \in L\} \tag{3.1}$$

If $g'$ is modelled by an ANN then the condition that samples from $V_c$ can be drawn efficiently is satisfied when $g \approx g' \implies V_c \approx V_c'$, simply by sampling a point from $L$ and passing it through $g'$.

It is therefore necessary to define a distance metric for two spaces, which the ANN can be trained to minimise. No single metric was found, however, which is both tractable and measures the distance between two general vector spaces. The performance of the generator function will therefore be measured in two dimensions, each describing a desired trait of the learned viable space:

**Precision**: the proportion of samples from $g'$ which are actually members of $V_c$.

$$p(g') = \int_C \int_L 1_{g'(l,c) \in V_c} \, \mathrm{d}l \, \mathrm{d}c \tag{3.2}$$

**Recall**: the proportion of solutions which satisfy the constraint that are members of $V_c'$.

$$r(g') = \int_C \int_{V_c} 1_{g'^{-1}(s,c) \in L} \, \mathrm{d}s \, \mathrm{d}c \tag{3.3}$$

Methods for estimating these quantities will be discussed in §4.

Given that the network capacity is limited, a fundamental tradeoff exists between $p$ and $r$. Too heavily weighting $p$ will result in a generator which picks from a small selection of promising solutions but misses many potential solutions; prioritising $r$ will result in a much larger range of solutions of which fewer satisfy the constraint.

This tradeoff is parameterised by a hyperparameter $w$ describing the weight of $r$ in the loss function $\mathcal{L}$ that the ANN is trained to minimise. The value of $w$ can be tuned to determine how confident the generator should be that its suggestions satisfy the constraint versus how confident it is that it is sampling from all viable solutions.

$$\mathcal{L}(g') = p(g') + w \cdot r(g') \tag{3.4}$$

## 3.2 Modelling spaces as distributions

One limitation of using an ANN to model $g'$ is that $V_c'$ will always be unimodal (given that $L$ is unimodal), while $V_c$ will almost always be multimodal. This arises as a consequence of the differentiability of ANNs, which is a desirable quality for this application so that gradient-based optimisation can be used in the latent space. Therefore $p$ and $r$, as defined above, will carry very little meaning because almost all valid solutions will be in $V_c'$, even though small distances in the $L$ can be mapped to large distances in $S$ (effectively changing the frequency of solutions sampled from $g'$).

This problem is solved by modelling each space as a probability distribution and defining proxy measures for precision and recall. The probability density of a distribution at a point is interpreted fuzzily as the likelihood that that point belongs to the space modelled by the distribution. If the density at point $a$ is greater than the density at point $b$, we say that $a$ is far more likely to belong to the space than $b$. The probability distribution function (p.d.f.) representing a space $X$ is denoted $\hat{X}$.

Arbitrarily, $L$ is defined to be a uniform distribution within its bounds, which are set to 0 and 1 for the sake of simplicity.

$$\hat{L}(l) = \begin{cases} 1 & \text{if } 0 \leq l_i \leq 1 \; \forall \; \{i \mid 1 \leq i \leq k\} \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

Since $V_c'$ is formed by passing $L$ through $g'$, its p.d.f. can be defined as the integral of $\hat{L}$ over all points in $L$ which map to that solution:

$$\hat{V_c'}(s) = \int_{\{l \mid g'(l,c)=s\}} \hat{L}(l) \, \mathrm{d}l \tag{3.6}$$

Finally, the true viable space's p.d.f. $\hat{V_c}$ is defined for any solution, and is 0 if the solution does not satisfy the constraint, and $\gamma$ if it does. The value of $\gamma$ is chosen such that the probabiliy mass of $\hat{V_c}$ is unity.

$$\hat{V_c}(s) = \begin{cases} \gamma & \text{if } h(c,s) = \text{satisfied} \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

The continuity of ANNs should therefore not threaten the feasibility of this approach, because while regions of $S$ which are not part of $V_c$ may be sampled occasionally, if the generator is trained well then these regions will be much less probable than regions highly likely to satisfy the constraint (Figure

3.1). Intuitively, this can be thought of as $g'$ pulling together the distinct modes of $V_c$, reducing the distance between them and thereby making it easier to find a global maximum by exploration.

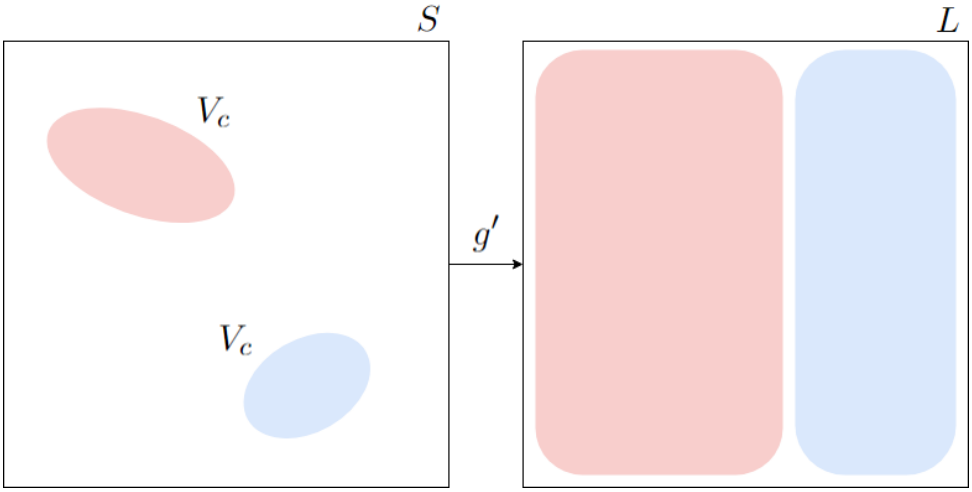


Figure 3.1: An example of how two distinct regions of $V_c$ in $S$ might be transformed by the generator to be much larger and closer together in $L$.

Proxy metrics can now be defined which are intuitively similar to $p$ and $r$ but are functions of p.d.f.s rather than of spaces. It is also possible to consider a distance metric between two distributions – which is much easier to define than a distance metric between two spaces – and minimise this directly, to prompt $\hat{V}_c'$ to match $\hat{V}_c$. These avenues are explored in §4.

## 3.3 Adapting the GAN architecture

A probabilistic interpretation of vector spaces has been introduced which provides information on the density of points within the space, thereby working around the limitation of ANNs that prevents discontinuous jumps between modes. This will also later allow the definition of tractable loss functions that can be used to train $g'$. The network architecture, however, is still undefined.

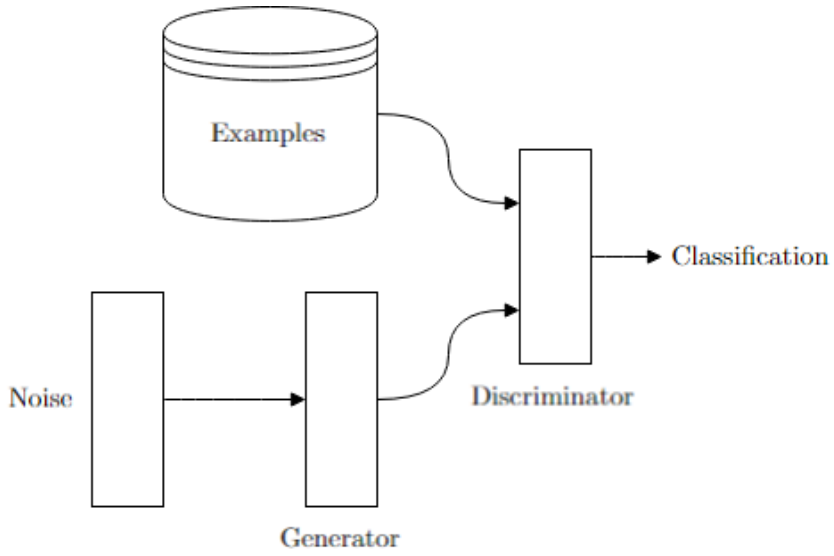


Figure 3.2: A typical GAN architecture.

GANs, introduced in §2.4, are a family of ANNs which allow samples to be drawn from a distribution. A typical GAN architecture has a generator, mapping samples from the latent space to the feature space, and a discriminator, which classifies feature vector as real or fake (Figure 3.2). This presents many similarities to the functions required to solve engineering problems, where the GAN's generator is similar to $g'$ and its discriminator to $h$. As such, the GAN architecture was adapted slightly for this project.

### 3.3.1 Pretrained discriminator

While training GANs, both the discriminator and generator are updated iteratively. This forces the generator to learn a range of outputs − effectively punishing low recall − by having the discriminator adapt continuously to the changing output of the generator.

In an engineering problem, however, $h$ is fixed and does not change. Whether $h$ is known in advance or learned from data, it does not make sense for it to be updated alongside the generator because of its immutability. This has already been accounted for by defining loss as a combination of $p$ and $r$, forcing the generator to not overspecialise.

### 3.3.2 Curried generator

Distributions learned by GANs are normally fixed. The distribution $\hat{V}_c$ is parameterised by the constraint vector, however, and so any adapted GAN needs to be capable of taking a constraint as input and adapting its output accordingly.

Perhaps a more natural way of thinking about this is to consider $g$ as a curried function; that is, it takes a constraint and then returns another function which maps from the latent space to the solution

space. This is a subtle semantic difference to a dyadic function, but may provide direction as to the best way of integrating the constraint vector into the GAN.

Consider an ANN for multi-class classification [30]. When fed a feature vector, the activations of units in the hidden layers do not depend on the class; information about the class is only added in the output layer. Likelihood estimates for each class are produced by multiplying the weight matrix of the final layer by the activations of the final hidden layer, and so are independent of each other. The likelihood of each class can be viewed as the dot product of the final hidden layer's activations with a vector embedding of the class (§2.5).

$Q$-learning performed by Deep $Q$-Networks (DQNs) on a discrete action space work similarly [31], except in a regression rather than a classification setting. Crucially, the DQN is a function of two parameters (state and action) which has been curried to take only one parameter (state) with the other built into its weights: the $Q$-values produced for each action are independent of each other; the resulting $Q$-values represent an unbounded scalar instead of a probability; and each $Q$-value is dependent only on the dot product of the final hidden layer's activations and a vector embedding of the related action. So DQNs prove that it is possible to use ANNs as curried functions where the weights of the last layer only are varied as an embedding of the first parameter.

To adapt this to approximate $g'$, one needs only realise that a constraint embedding can be produced as a function of the constraint vector, which could feasibly be approximated by another ANN. So information about $c$ can be passed into $g'$ by estimating the weights and biases of its output layer using two separate networks, the weight and bias embedders (Figure 3.3), which takes only $c$ as input.

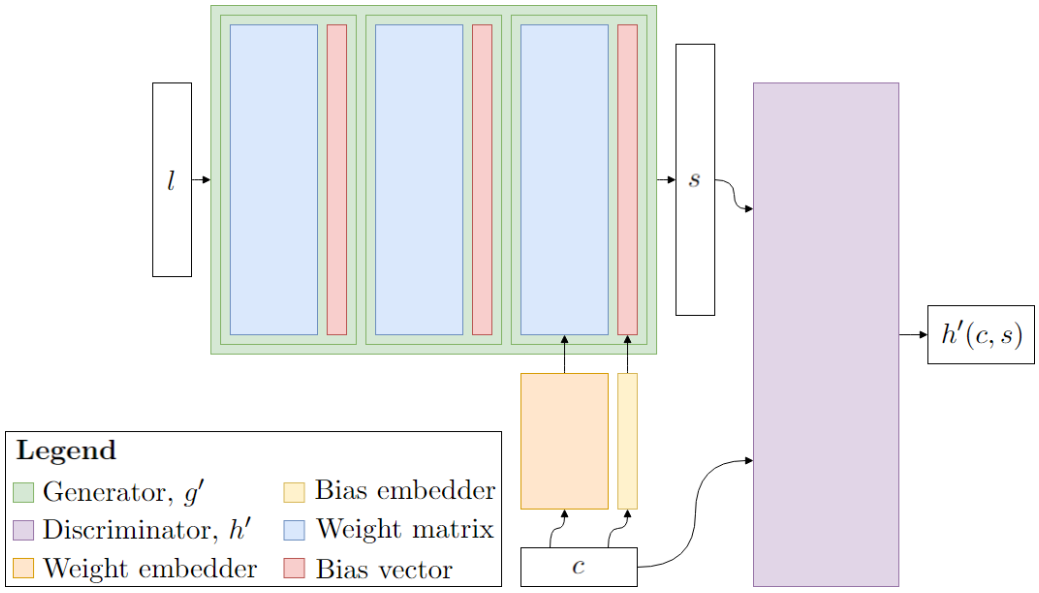


Figure 3.3: Proposed architecture, with two networks providing the weights and biases of the generator's output layer.

### 3.3.3 Data-free training

If the discriminator has already been trained, or is known, no data are needed to train $g'$ because samples from $L$ can be used as training examples. Hence, training data are only used as examples for training an approximation of $h$, $h'$, used as the discriminator; no additional data are needed to train $g'$. The advantages to this approach are twofold: data are only needed to train $h'$ individually rather than adversarially in a GAN (which are notoriously hard to train [26]), and $g'$ is only capable of overfitting to the extent that $h'$ has overfit its data.

When $h$ is known analytically, any overfitting of $g'$ can be combatted by increasing its capacity or training for longer. When $h$ is modelled by a discriminator $h'$ learned from data, knowledge of the engineering environment can be used to augment data to alleviate overfitting (see §5). In any case, moving all responsibility for overfitting from $g'$ to $h'$ − where it can be more easily measured and avoided − is an exceptional advantage to adapting the GAN architecture in this way.

# Chapter 4

# Generator Training

## 4.1 Distribution matching

### 4.1.1 KL-divergence

Because using ANNs requires a space similarity problem to be framed as a distribution similarity problem (§3.2), distribution distance metrics can be used. If $g'$ is modelled by a network whose p.d.f. can quickly be found, the KL-divergence between $\hat{V}_c'$ and $\hat{V}_c$ can be estimated by drawing samples from $\hat{V}_c'$. The p.d.f. of an ANN can be found by inverting it or using a constrained architecture such as a radial basis function [32]. Whether or not inverting a network is plausible is discussed in §4.3. The integral form of the KL-divergence is:

$$D_{\mathrm{KL}}(\hat{V}_c' \mid\mid \hat{V}_c) = \int_L \hat{V}_c'(g'(l,c)) \log\left(\frac{\hat{V}_c'(g'(l,c))}{\hat{V}_c(g'(l,c))}\right) \mathrm{d}l \tag{4.1}$$

Since integration over the entirety of the latent space is unlikely to be tractable, this equation can be adapted such that the divergence is written as an expectation across a batch of samples, each weighted by their probability of occurring. Let $\{l_1, l_2, ..., l_{j-1}, l_j\}$ be a batch of samples drawn from the latent space. An estimation of the KL-divergence can then be calculated:

$$\mathbf{E}\big[D_{\mathrm{KL}}(\hat{V}_c' \mid\mid \hat{V}_c)\big] = \frac{1}{j}\sum_{i=1}^{j} \log\left(\frac{\hat{V}_c'(g'(l_i,c))}{\hat{V}_c(g'(l_i,c))}\right) \tag{4.2}$$

The term before the logarithm disappears in the summation equation because each sample is weighted by the inverse of its probability density under $\hat{V}_c'$.

Since all operations in this equation are differentiable, the partial derivative of the KL-divergence with respect to the weights of the generator can be found by a library with automatic differentiation such a Tensorflow. Then $g'$ training is simply a matter of minimising $D_{\mathrm{KL}}$.

### 4.1.2 Potential issues

While theoretically very simple, there are several practical problems that could arise during training with this approach.

**Exploding gradients**: a problem frequently encountered when minimising functions containing a logarithm is the occurrence of `NaN` values resulting from taking the logarithm of zero [33]. While this will not be an issue since the numerator of the fraction will be non-zero by virtue of the fact that that point has been sampled, very small values are possible. Zeros in the denominator are also possible. These will produce large negative logarithms, whose derivatives may cause exploding gradients. Because it is known that large positive gradients are likely to result from a logarithm, but large negative ones are impossible, it may be possible to clip the derivatives to prevent exploding gradients while still obtaining correct results.

**Disjoint manifolds**: the points comprising real data distributions are often described in a much higher-dimensional space than the intrinsic dimension of the distribution that created them. This will likely be the case for $h$, and so when $g'$ is first initialised it is highly likely that $\hat{V}_c'$ and $\hat{V}_c$ will be disjoint (§6.1.3). If this is the case then the KL-divergence will always be 0 and no training will occur. Methods of solving this problem, such as minimising the Wasserstein distance instead of KL-divergence, were considered but judged to be beyond the scope of this project.

## 4.2 Metric proxies

As discussed in §3.2, modelling spaces as distributions invalidates the equations for $p$ and $r$ that were introduced in §3.1. Proxy metrics will now be defined for $p$ and $r$, which are intuitively similar to the concepts described for spaces but applicable to distributions.

### 4.2.1 Precision

Optimising precision is intended to ensure that as many elements from $V_c'$ as possible are also in $V_c$. When these are viewed instead as distributions, $\hat{V}_c'$ will likey cover almost the entirety of $S$ but with differing densities.

An adequate precision proxy $\hat{p}(g')$ is one which heavily penalises samples from $\hat{V}_c'$ which are in low-density regions of $\hat{V}_c$. Infrequently drawn samples from $\hat{V}_c'$ may have high penalties, but these will not have a large impact on the overall precision proxy because they will be outnumbered by more probable samples with low penalties.

Letting $\{l_1, l_2, ..., l_{j-1}, l_j\}$ be a batch of samples drawn from $L$, the precision proxy can be defined as:

$$\mathbf{E}\big[\hat{p}(g')\big] = -\frac{1}{j}\sum_{i=1}^{j}\log\big(1 - h(c, g'(l, c)) + \epsilon\big) \tag{4.3}$$

Note that $h(c, s)$ is proportional to $\hat{V}_c$ but is a probability instead of a p.d.f. and so will always be between 0 and 1.

Intuitively, this is equivalent to taking a batch of points from $L$, mapping them into $S$, calculating $h$ for each point, and taking the mean of the negative logarithm of 1 minus each of these values. Solutions for which $\hat{V}_c(s) \ll \hat{V}_c'(s)$ will occur frequently (because their density in $\hat{V}_c'$ is high) but be severely penalised (because $h(c,s)$ is small and so $\log(1-h(c,s))$ will be large). The result of this is that $g'$ will learn weights that avoid this situation as much as possible.

### 4.2.2 Recall

In many ways, recall is the opposite of the precision. Solutions which are highly likely to satisfy the constraint, but which occur very infrequently in $\hat{V}_c'$, will be penalised.

$$\mathbf{E}\big[\hat{r}(g')\big] = -\frac{1}{j}\sum_{i=1}^{j}\log\big(1 - g'^{-1}(s,c) + \epsilon\big) \tag{4.4}$$

where $\{s_1, s_2, ..., s_{j-1}, s_j\}$ is a batch of solutions drawn from $\hat{V}_c$. Such a batch can be produced using a Markov chain Monte Carlo (MCMC) method such as the Metropolis-Hastings algorithm [34]. Like direct minimisation of the KL-divergence, $\hat{r}$ assumes that $g'^{-1}$ is tractable.

### 4.2.3 Spread

Optimisation for $\hat{p}$ is dependent on high-density regions of $\hat{V}_c$ being sampled at least occasionally by $\hat{V}_c'$, and so it stands to reason that convergence will occur slowly if $\hat{V}_c'(s) \approx 0$ for most $s$.

Training might therefore be accelerated if $g'$ can be pretrained such that it samples from a wide range of $S$ at least sometimes, allowing the highest density regions of $\hat{V}_c$ to be quickly identified. Some metrics are proposed below which, when minimised, discourage $g'$ from sampling only a small subset of $S$.

**Similarity loss**: penalises according to the squared distance between a point in $L$ and its accompanying point in $S$, when both spaces are linearly scaled such that they are unit hypercubes. This effectively forces $g'$ to model the identity function after its initialisation.

$$q_{\text{id}} = \left| \frac{l - a_L}{b_L - a_L} - \frac{g(l,c) - a_S}{b_S - a_S} \right|^2 \tag{4.5}$$

Since any point in $L$ can be sampled, if $g'$ closely approximates the identity function then all points in $S$ will be roughly equally likely to be sampled. One clear drawback, however, is that this spread metric requires $m = k$, which may not be desirable for large $m$.

**Separation loss**: a more weakly defined spread metric for use when $m \neq k$. Intuitively, it is equivalent to forcing the mean distance between points in $\hat{V}_c'$ towards $q_{\text{target}}$.

$$q_{\text{sep}} = \left( q_{\text{target}} - \frac{1}{b^2}\sum_{i,j=1}^{b} \left| B_i - B_j \right|^2 \right)^2 \tag{4.6}$$

where $B = \{B_1, ..., B_b\}$ is a batch sampled from $\hat{V}_c'$. This spread metric will typically be quite slow because its calculation is of the order $\mathrm{O}(b^2)$ as opposed to $\mathrm{O}(b)$ for $q_{\mathrm{id}}$.

As well as being used in pretraining for efficient generator initialisation, these spread metrics could be used as a weak substitute for $\hat{r}$ in the event that evaluating $\hat{V}_c'$ is intractable. While not strictly measuring $\hat{r}$, they prevent $g'$ from only choosing a single point repetitively.

## 4.3 Neural network inversion

Some of the proposed training metrics depend on evaluating $\hat{V}_c'$ at a particular point in $S$. If $g'$ is modelled by an ANN, this is not an obviously tractable problem; this section will explore methods of solving it.

By viewing $g'$ as a series of transforms applied to $\hat{L}$, which is known to be uniform, the space of points which produce a desired output − referred to as the relevant space $R$ − can be backpropagated through each transform. Integrating over $R$ in $\hat{L}$ will yield the probability density of the desired output.

An ANN is made up of a series of layers, each of which applies three transforms to its input:

$$y = \sigma(Wx + b) \tag{4.7}$$

where $y$ is the layer output, $x$ is the layer input, $\sigma$ is the activation function, $W$ is the weight matrix, and $b$ is the bias vector. Depending on the number of rows in $W$, the layer will expand, reduce, or maintain the dimension of the input. The input dimension is $n_i$ and the output dimension is $n_o$.

### 4.3.1 Bias addition

The input which will produce a specific output after the addition of a bias can be found trivially by subtracting the bias vector from the desired output.

$$y = x + b \implies x = y - b \tag{4.8}$$

As such, the space of relevant inputs is equivalent to the space of relevant outputs translated by $-b$.

### 4.3.2 Square weight multiplication

A layer in which $W$ is a square matrix has $n_i = n_o$. There exists exactly one input which produces the desired output assuming that the output is not a null vector and $|W| \neq 0$; both of these are generally valid assumptions. Therefore $R_{\mathrm{in}}$ can be found by rotating $R_{\mathrm{out}}$ according to $W^{-1}$.

In general, evaluating $W^{-1}$ computationally expensive, especially when $n_i$ is large. One special case in which this is not true is when the $W^{-1} = W^T$, which is much more efficient to evaluate, and occurs when $W$ is orthogonal (pure rotation). Ensuring that $W$ is always orthogonal is discussed in Appendix C.

### 4.3.3 Expansive weight multiplication

When $n_o > n_i$, the input is overdefined by the output, hence not every output will have an input which creates it. An alternate way of looking at this is that the input is fully defined by the first $n_i$ dimensions of the output. The latter $n_o - n_i$ dimensions of the output are therefore redundant when it comes to producing $R_{\text{in}}$ from $R_{\text{out}}$.

$W$ is therefore constrained such that it is constructed by concatenating a $n_i \times n_i$ orthogonal matrix $W_{\text{orth}}$ with a $n_o - n_i \times n_i$ matrix whose elements can vary freely, $W_{\text{free}}$ (Figure 4.1). Then to invert a space of relevant points, any constraints on the latter $n_o - n_i$ dimensions can be ignored, while any remaining constraints are rotated by $W_{\text{orth}}^{-1} = W_{\text{orth}}^{T}$.

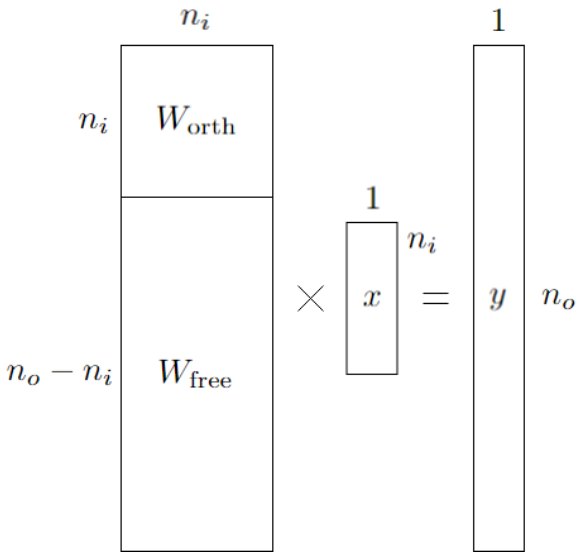


Figure 4.1: Weight matrices which make up an expansive layer in an invertible ANN.

### 4.3.4 Compressive weight multiplication

When $n_o < n_i$ the output underdefines the input. Generally, the relevant input space is unconstrained and therefore infinite in $n_i - n_o$ dimensions, which is why finding the probability density of a distribution transformed by an ANN requires that each step be able to handle backpropagating relevant spaces as opposed to only points.

Normally, the size of the weight matrix would be $n_o \times n_i$, where $n_i > n_o$. In this case, however, the weight matrix will be an orthogonal matrix of size $n_i$. The output vector is obtained by multiplying $W$ by the input, and then retaining only the first $n_o$ components of the resulting vector (Figure 4.2).

Consider the intermediate step, after multiplying $W$ by the input but before clipping the vector. The space of vectors which produce $y$ when retaining only the first $n_o$ components is constrained such that:

$$x'_j = y_j \; \forall \; 1 \leq j \leq n_o \tag{4.9}$$

where

$$x' = Wx \tag{4.10}$$

while the latter $n_i - n_o$ dimensions can assume any value. This creates a space which is infinite in some dimensions, but constrained to a single point in others. The relevant input space can then be obtained by rotating this intermediate space by $W^T$.

Note that the infinite relevant space can be approximated by a finite polytope which extends in each unconstrained dimension to a suitably large distance from the origin, under the reasonable assumption that the activations of hidden units will never greatly exceed 1. Therefore $R_{\text{in}}$ can be described by a set of simplices, also known as a simplicial complex, since any polytope can be constructed from a finite number of simplices [35]. Being able to propagate these simplices backwards through the various transforms described in this section is therefore sufficient for the backpropagation of $R_{\text{out}}$, and thereby finding its probability density.

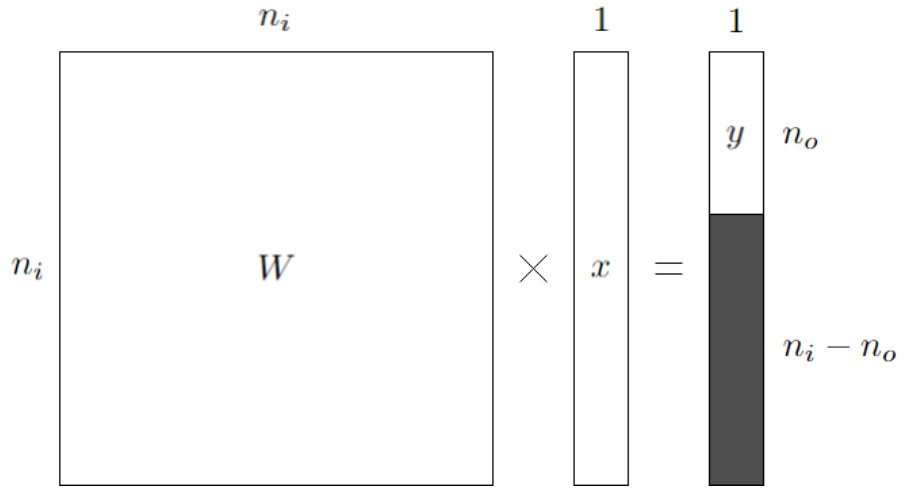


Figure 4.2: Weight matrices which make up a compressive layer in an invertible ANN.

### 4.3.5 Activation

Several activation functions are commonly used in ANNs, but here only the ReLU activation function [36] will be considered. It consists of two linear segments and is applied elementwise to a vector at the end of each layer.

Because the activation function is applied elementwise, the way in which $R_{\text{out}}$ propagates backwards through a ReLU unit can be considered in terms of each dimension independently. The following rules can be applied to each dimension:

1. Any region which is above 0 can propagate backwards unchanged, since the ReLU function mimics the identity function above 0;

2. Any region which is below 0 can be ignored, since the ReLU function will never produce an input which is below 0;

3. If 0 is a possible value in a dimension, all negative values must be added to $R_{\text{in}}$ in that dimension. This is because any negative value in that dimension would have produced an output of 0. Again, simplices extending suitably far from the origin can be used to approximate an extension to infinity.

### 4.3.6 Obstacles

While theoretically sound, propagating a relevant space backwards through an ANN in order to compute its p.d.f. like this is not practical.

Problems arise because compressive layers require that a space be propagated back rather than a point. A simplicial complex can be used to describe this space, but inverting a simplicial complex through a ReLU activation function is impractical for two reasons.

The first is that inverting each simplex requires determining how it intersects with an axis, if at all. While algorithms exist for doing this in three dimensions or fewer, it appears that an algorithm for $n$ dimensions does not yet exist [35].

Secondly, the inversion requires that each simplex splits in two upon meeting the zero of an axis. If it intersects with the zeros of $i$ axes simultaneously, it must be split into $2^i$ different simplices. While this split will not occur every time a simplex is propagated back through a ReLU activation, the potential for such an exponential growth could quickly become unmanageable.

Other activation functions which do not consist of linear segments will suffer even greater problems since a simplex is a fundamentally linear shape. For this reason, methods which require the evaluation of the p.d.f. of an ANN will not be pursued further in this report.

## 4.4 Training procedure

This section assumes that $h$ is either already known, or approximated by $h'$. That process is explored in greater depth in §5.

Training broadly consists of two steps: pretraining, which involves initialising $g'$ to explore a reasonable subset of $S$; and training, in which $g'$ is trained to balance $\hat{p}$ and $\hat{r}$.

### 4.4.1 Pretraining

A batch of samples are taken from $\hat{L}$ and passed through $g'$ into $S$. Unless otherwise required by the environment, constraints can also be sampled uniformly from $C$. Using these values, the spread loss of $g'$ is calculated according to either $q_{\text{id}}$ or $q_{\text{sep}}$, whichever is deemed more appropriate.

In the backward pass, the partial derivative of $q$ is calculated with respect to each generator parameter. Gradients are applied using a gradient descent algorithm such as Adam [3].

This process is repeated until either the parameters converge or $q$ is sufficiently low.

### 4.4.2 Training

In the main training phase, latent points are sampled uniformly and constraints are sampled as in the pretraining phase. They are matched into arbitrary pairs, each one comprising a data point. The constraint vector is fed to both $h'$ and the embedders (which in turn feed their values to the last layer of $g'$), while $l$ is fed to $g'$.

From these inputs, $s$ is calculated, and the likelihood of its satisfying $c$ is calculated from $h'$. Given all these values, $\hat{p}$ and $\hat{r}$ (or one of $q_{\text{id}},\ q_{\text{sep}}$ in place of $\hat{r}$) can be calculated and combined into a weighted loss term. Again, the derivatives of this value are used to minimise the weights of $g'$. If $h'$ is an ANN, its weights remain fixed throughout this process.

# Chapter 5

# Discriminator

Previous chapters have assumed that a discriminator, $h'$, has already been trained and is readily available. This chapter details the exact role of $h'$ and some rules of thumb for training it from data.

## 5.1 Formal definition

$h'(c, s)$ is a differentiable approximation of the constraint satisfaction function $h(c, s)$. While in some environments $h$ may be known, it is assumed that most environments will be too complex for an analytical form and so it will be learned from data by an ANN.

Differentiability is required of $h'$ because gradients will pass through it during training of $g'$. Because $h$ gives binary output, gradients will always be zero, so $h'$ is trained to output a probability that the solution satisfies the constraint, smoothing the gradients. It is for this reason that even $h$ is known analytically, an ANN should be used to accelerate training of $g'$ by ensuring that gradients are non-zero.

## 5.2 Training procedure

Data are provided in the form of $(c,\ s,\ h(c, s))$ tuples where the first two vectors are network inputs and the third is the label. The discriminator will normally take the form of a standard multi-layer perceptron, taking its arguments as a concatenated vector and outputting a single scalar probability. As such, the output layer should be sigmoid activated, but the internal activations are not constrained.

Training is accomplished by backpropagation [17]. It is also strongly recommended that a proportion of the data (nominally, 20%) is set aside for the purposes of validating $h'$ after training.

## 5.3 Overfitting

Because $g'$ is trained off $h'$ and not from data, ensuring that $h'$ does not overfit is essential. A number of standard techniques should be employed, including dropout [11] and L2 regularisation [37]. The

size of the dataset should be considered when choosing the network depth and width, and training should be stopped before validation error begins to rise (a sign that the network is overfitting).

## 5.4 Practical considerations

A number of different considerations should be taken into account on a case-by-case basis when training the discriminator.

### 5.4.1 Incorporating environment knowledge

Many engineering environments are understood quite well from an analytical perspective, and this knowledge can sometimes be leveraged to augment the training data. A frequently valid assumption is that data points which are nearby in the feature space will produce similar outputs. Adding noise to the data before feeding them into the network simulates a much greater quantity of data, thereby reducing overfitting, but may cause a loss of fidelity in environments in which the output is highly sensitive to the input.

Some environments are also very sparse: that is, only a miniscule subset of all valid environments will satisfy the constraint. In such cases, training data may have to be hand-picked to ensure that enough positive training examples are observed. If carrying out experiments to obtain data is also costly, a dataset of only positive examples could be gathered. Negative samples could then be produced by taking the solution from one datum and combining it with the constraint from another datum, under the assumption that that solution will very likely fail to satisfy the constraint.

Sparse environments may also be amenable to performing dimensionality reduction on the constraints, solutions, or both, by means of an autoencoder or VAE.

### 5.4.2 Limiting network capacity

Training a perfect discriminator is not always desirable: a perfect discriminator will have near-zero gradients by virtue of the fact that it is trying to approximate a binary function. Deliberately forcing the discriminator to cope with greater uncertainty can reduce the chance of the discriminator's output saturating, thereby giving clearer indications to the generator by backpropagating gradients of a greater magnitude.

Adding noise to the inputs as aforementioned is one way of achieving this, with another possibility being the limitation of the capacity of the network. Using fewer hidden nodes than would usually be used may force the discriminator to learn a simplified function, which is technically less accurate but may induce faster training of the generator. These changes also make the discriminator less susceptible to overfitting, which is possibly a greater threat to the success of the discriminator than a slight reduction in accuracy.

# Chapter 6

# Results

A number of experiments were conducted with two broad goals: verifying that the proposed architecture works on simplified abstract engineering environments, and proving that it has desirable qualities in more complex environments. Some experiments were also carried out to visualise and explore the latent space mappings in an intuitive way.

## 6.1 Approach verification

### 6.1.1 Distribution distance

Due to the potential issues with minimising KL-divergence discussed in §4.1.2, a simplified environment was devised in which the target distribution $\hat{V}_c$ is known. $\hat{V}_c$ was fixed as a standard normal distribution (constraints were not considered at this stage) and $\hat{V}_c'$ was modelled as a normal distribution whose mean and standard deviation were learned parameters.

Although KL-divergence is analytically tractable for two normal distributions, it was estimated through sampling of $\hat{V}_c'$ as described in §4.1.1. Initial standard deviation $\sigma_0$ was always set to 1, while initial mean $\mu_0$ was varied.

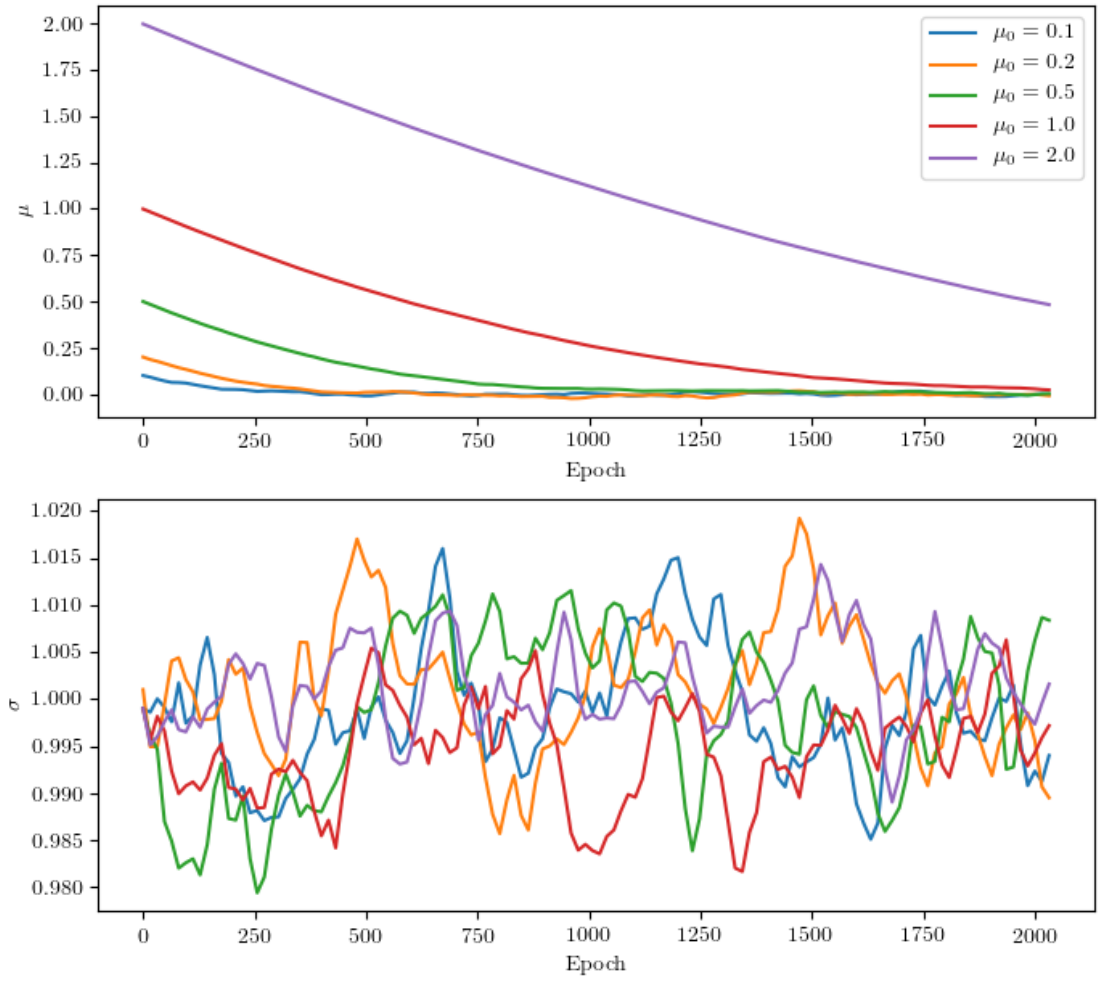


Figure 6.1: Change in $\mu$ and $\sigma$ over time when training arbitrary normal distributions with different starting means to match the standard normal distribution. Training parameters are reproduced in Appendix D.

The mean $\mu$ and standard deviation $\sigma$ after each batch are shown in Figure 6.1. Convergence does occur, but the time taken for the mean to approach 0 appears to increase exponentially as the initial mean moves away from the target. Two one-dimensional normal distributions whose means only differ by two and whose standard deviations are both one could still be considered to be quite similar, so the experiment was repeated with $\sigma_0 = 0.1$ for both distributions to better simulate disjoint distributions.

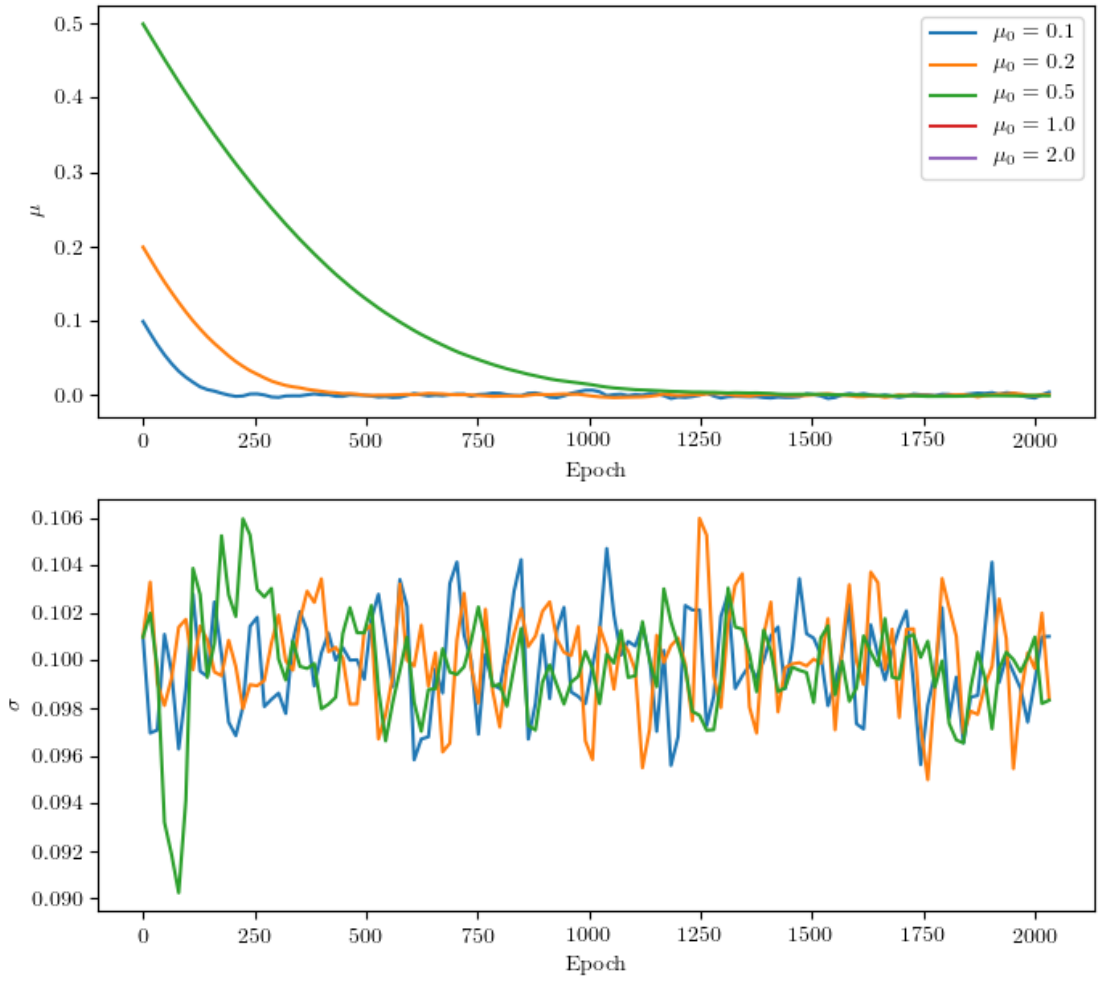


Figure 6.2: Change in $\mu$ and $\sigma$ over time when training arbitrary normal distributions with different starting means to match a normal distribution with $\mu = 0$ and $\sigma = 0.1$. Training parameters are reproduced in Appendix D.

Figure 6.2 shows that the variables converge at approximately the same rate as when $\sigma_0 = 1$. Both $\mu$ and $\sigma$ went to `NaN` after the first epoch and so are not shown. This was thought to have been caused by an excessively large KL-divergence resulting in exploding gradients due to the limited precision of floating point numbers.

Changing to 64- instead of 32-bit values lent further evidence to this hypothesis, as increasing the precision allowed the $\mu = 1.0, 2.0$, $\sigma = 0.1$ cases to be optimised without numerical errors (Figure 6.3). Similarly, using 16-bit precision always resulted in `NaN` errors occurring.

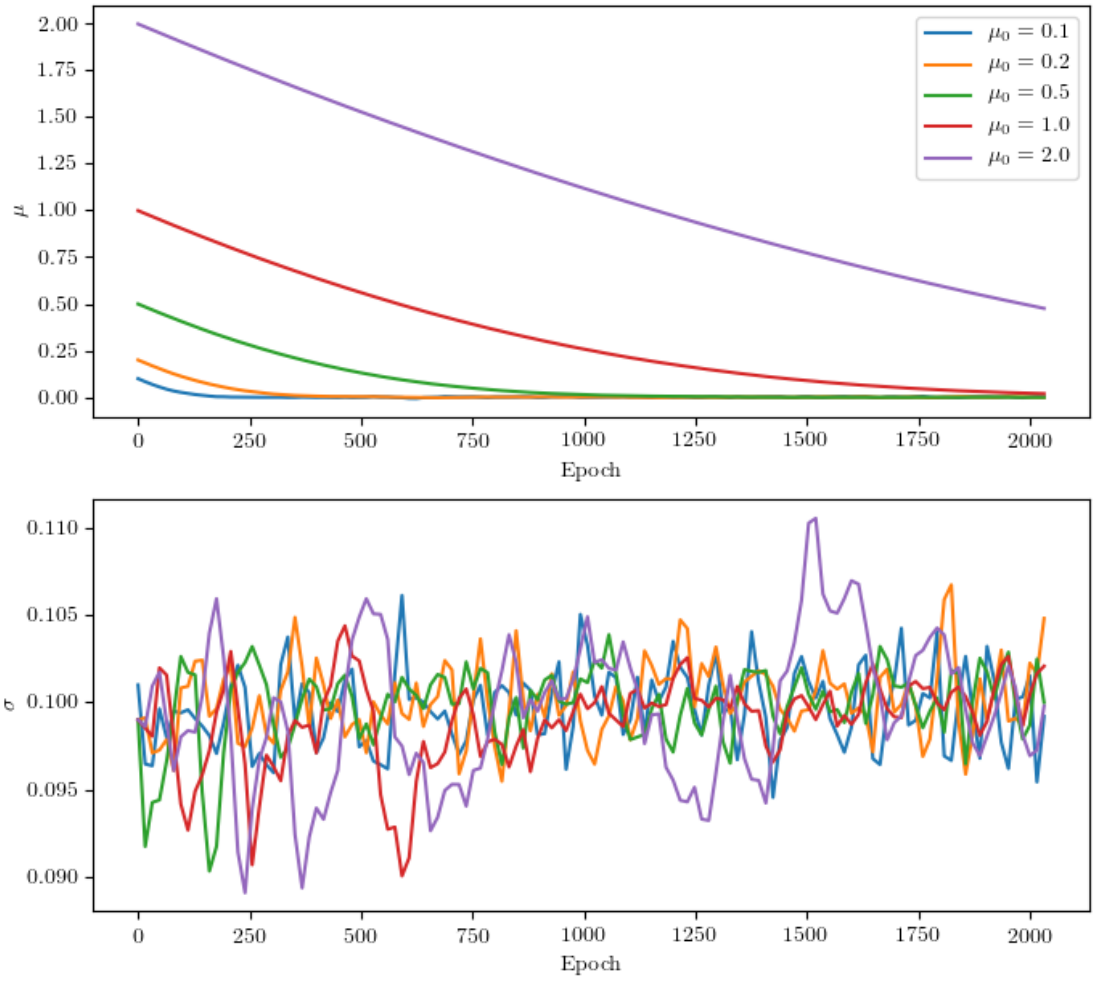


Figure 6.3: Change in $\mu$ and $\sigma$ over time when training arbitrary normal distributions with different starting means to match a normal distribution with $\mu = 0$ and $\sigma = 0.1$. Training parameters are those in Appendix D, but with 64-bit precision.

It was therefore concluded that minimising KL-divergence is too inconsistent for practical use, as real distributions − which are frequently disjoint − would very likely cause numerical issues due to lack of the required precision.

### 6.1.2 Precision proxy optimisation

All experiments in this section used the parameters in Appendix F.

Potential issues with directly estimating $\hat{r}$ were discussed in §4.3. It was also hypothesised that optimising only $\hat{p}$ would result in $g'$ sampling only a small subset of $V_c$. A short experiment was devised to prove that this is the case and so a substitute for $\hat{r}$ would be needed. $h$ was defined arbitrarily for

$m = 1,\ n = 0$, described in detail in Appendix E. A graph of the constraint satisfaction function on $s \in [-1,\ 1]$ is shown in Figure 6.4.

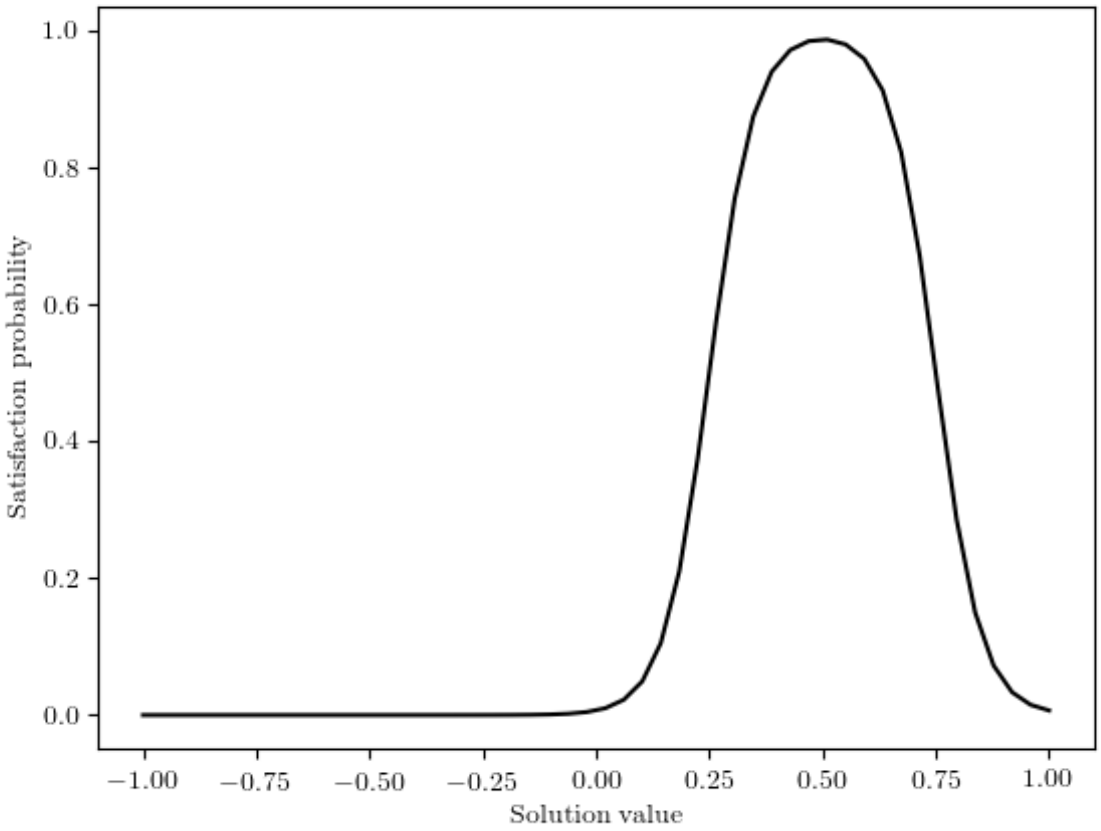


Figure 6.4: A test function estimating the probability that a solution satisfies a singular constraint.

Optimising $g'$ against $h(c, s)$ for $\hat{p}$ consistently yielded a distribution of generated solutions displayed in Figure 6.5. Only considering precision and disregarding recall produced a mapping which is of no use: it always mapped to a singular point in spite of the fact that around 15% of the solution space has a constraint satisfaction probability above 0.9.

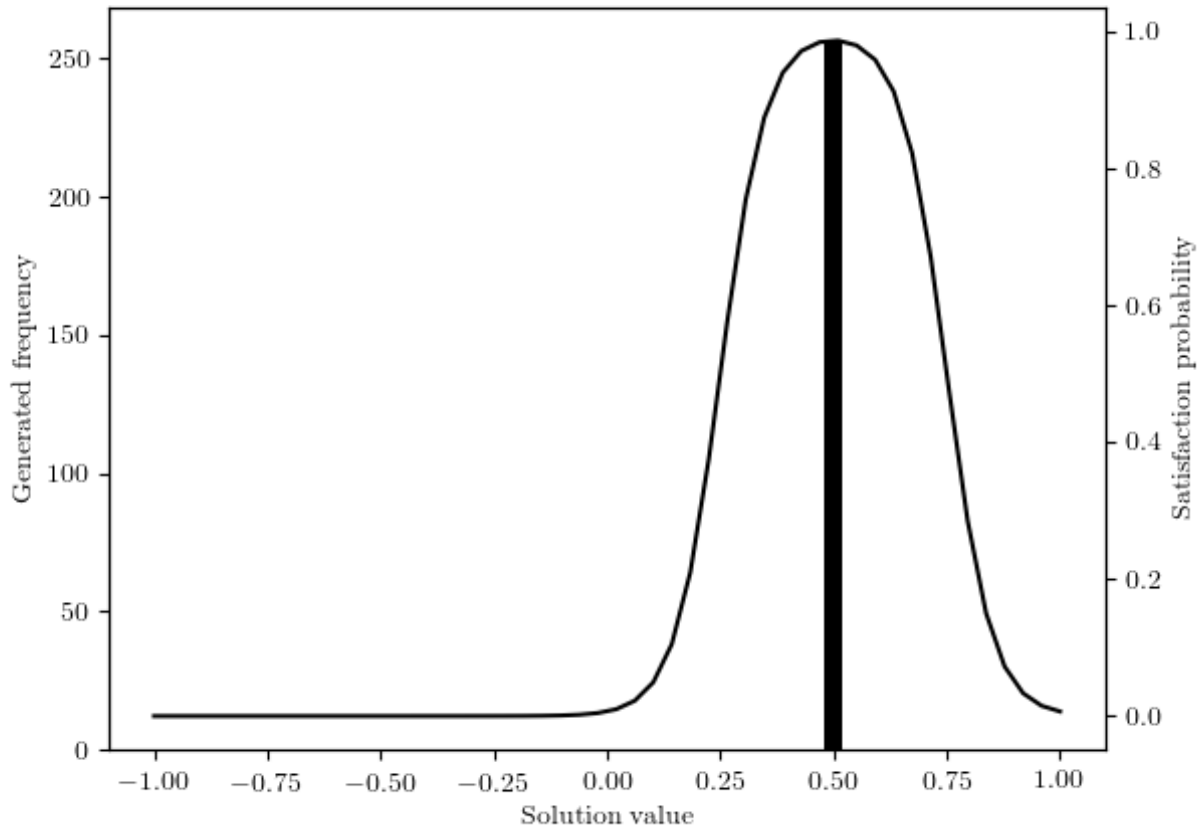


Figure 6.5: Histogram of samples from $\hat{V}_c'$ compared to $h$ after optimising only $\hat{p}$.

This issue becomes more pronounced when $h$ has two distinct modes (Figure 6.6, Appendix G). While a very high precision is obtained, optimising solely for precision fails to capture the bimodality of $h$ even in a simple environment, confirming that optimisation must take recall into consideration for satisfactory results.

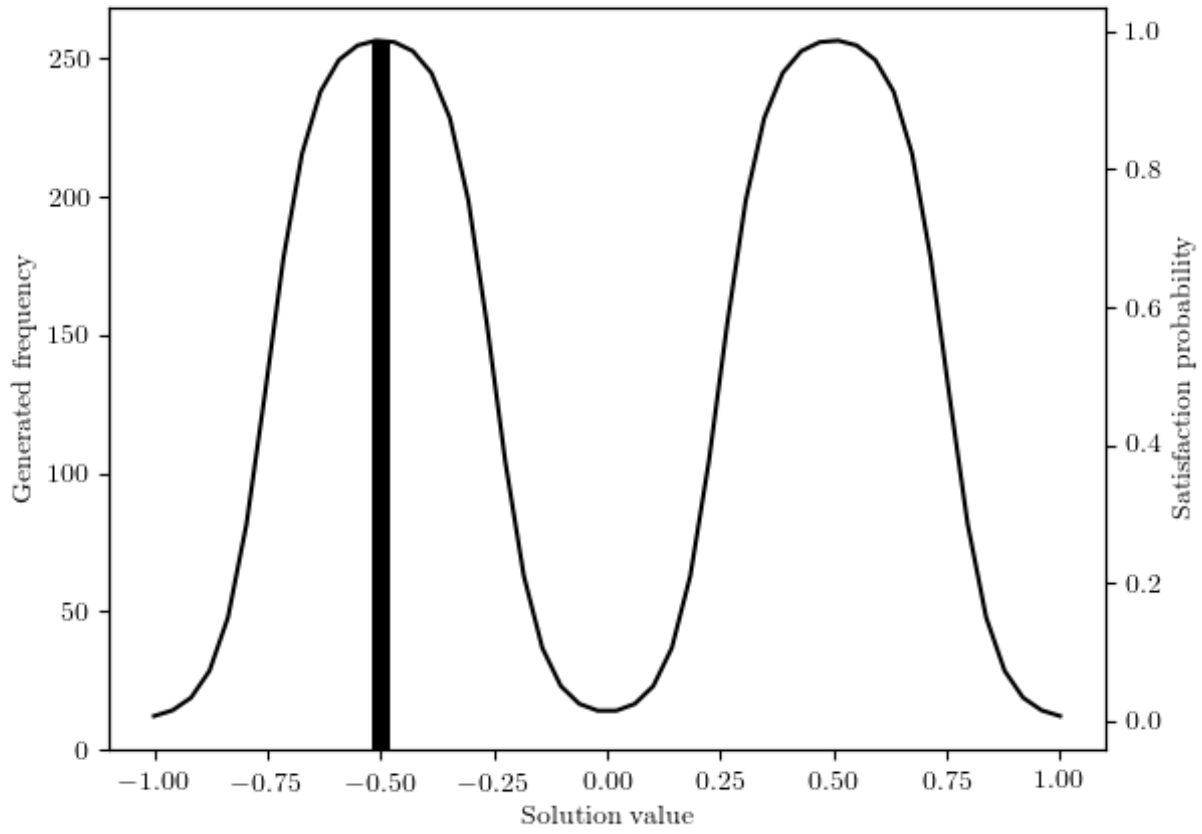


Figure 6.6: Histogram of samples from $\hat{V}_c'$ compared to $h$ after optimising only $\hat{p}$.

### 6.1.3 Pretraining

All experiments in this section used the parameters in Appendix F unless otherwise specified.

Without $\hat{r}$ being tractable, it was suggested that $g'$ be pretrained to ensure that its initial mapping covers as wide a range of $S$ as possible. This is based on two suppositions: that $g'$ initially only takes samples from an isolated region of $S$; and that a spread-out distribution would explore multiple high-satisfaction regions of $S$, thereby capturing a multimodal $\hat{V}_c'$ even when optimising only for precision.

That the first assumption holds is confirmed trivially by samples from $\hat{V}_c'$ before training (Figure 6.7).

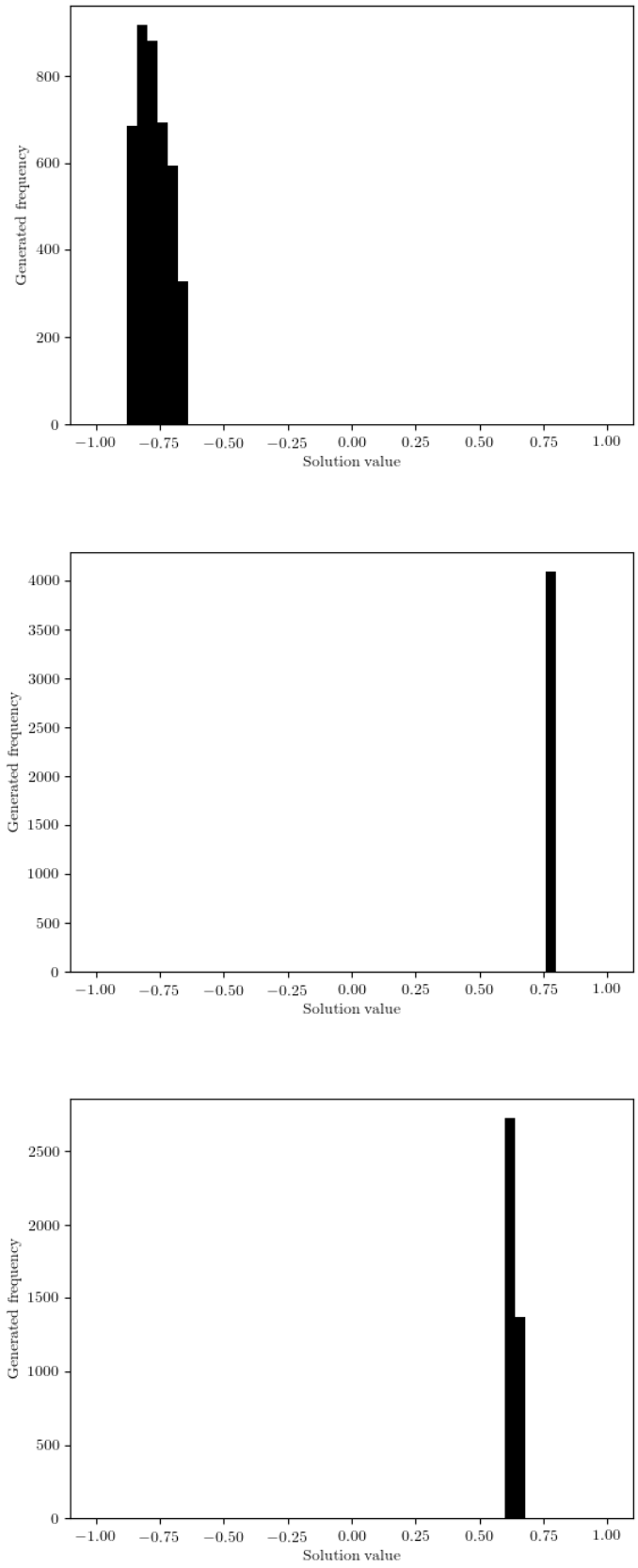


Figure 6.7: $\hat{V}_c'$ immediately after initialisation for three different seeds. 4096 histogram samples were taken instead of 256.

Pretraining is performed in the same way as normal training, but one of the spread losses $q_{\text{id}}$ or $q_{\text{sep}}$ is minimised instead of $\hat{p}$. Because the spread is a subjective quality which cannot be objectively measured, the success of each spread metric was judged on whether it enabled $g'$ to learn both modes of $\hat{V}_c$ simultaneously.

Below are examples of a typical $\hat{V}_c'$ after pretraining with each spread metric.

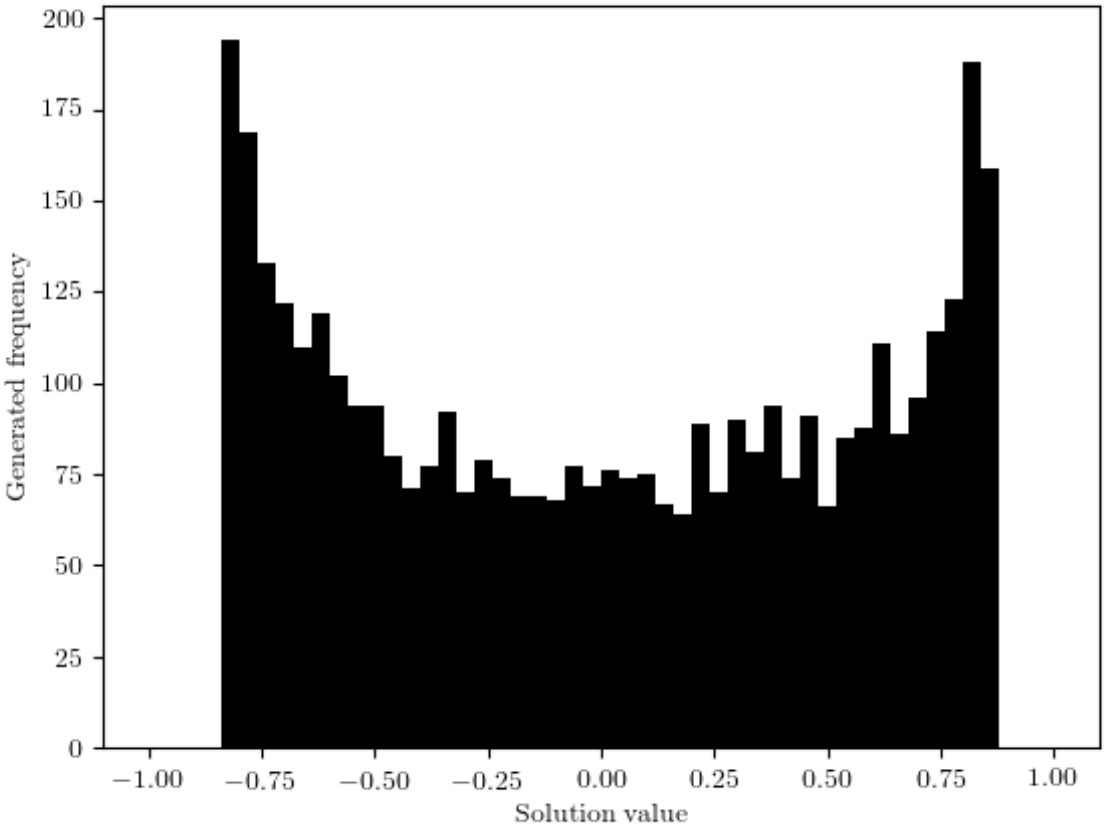


Figure 6.8: $\hat{V}_c'$ after pretraining until convergence using $q_{\text{id}}$. 4096 histogram samples were taken instead of 256.

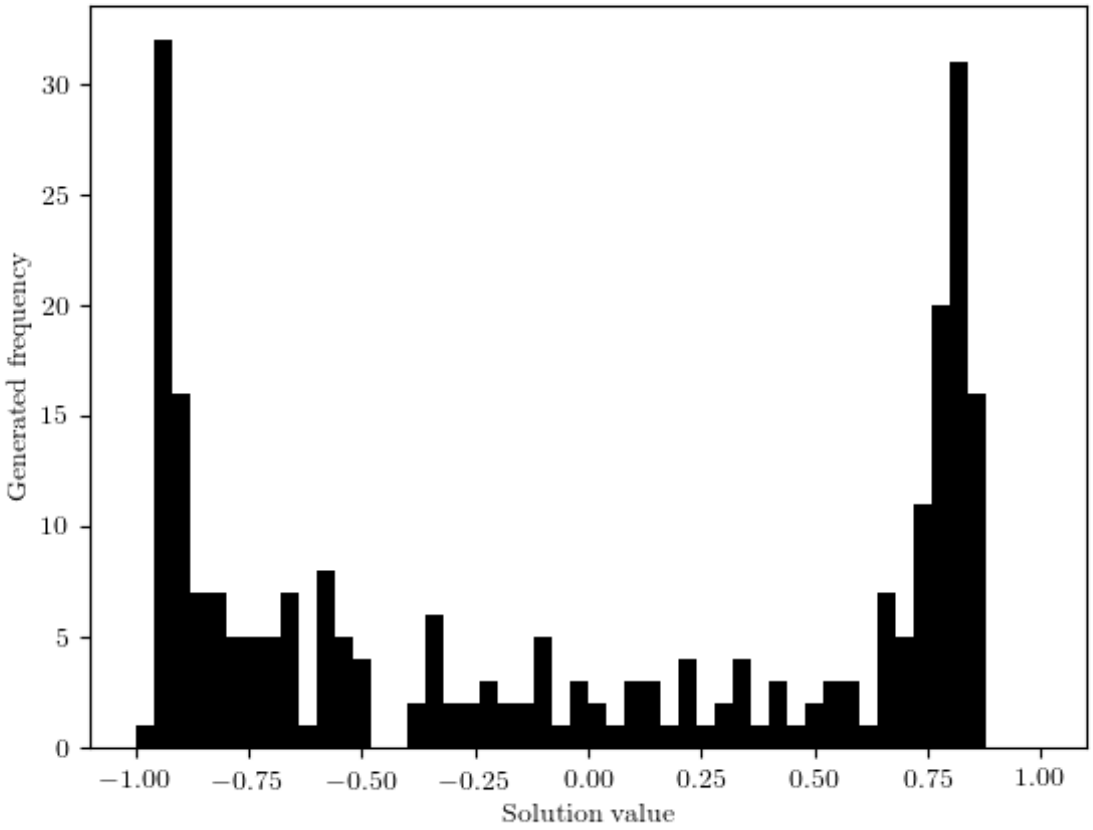


Figure 6.9: $\hat{V}_c'$ after pretraining until convergence using $q_{\text{sep}}$.

Both metrics show signs of causing $g'$ to explore a far greater proportion of $S$ than immediately after initialisation, but also have a tendency to over-explore the boundaries. This effect is much more pronounced with $q_{\text{sep}}$. The true success of each metric, however, is determined by the generator's ability to find different modes when precision is optimised after pretraining.

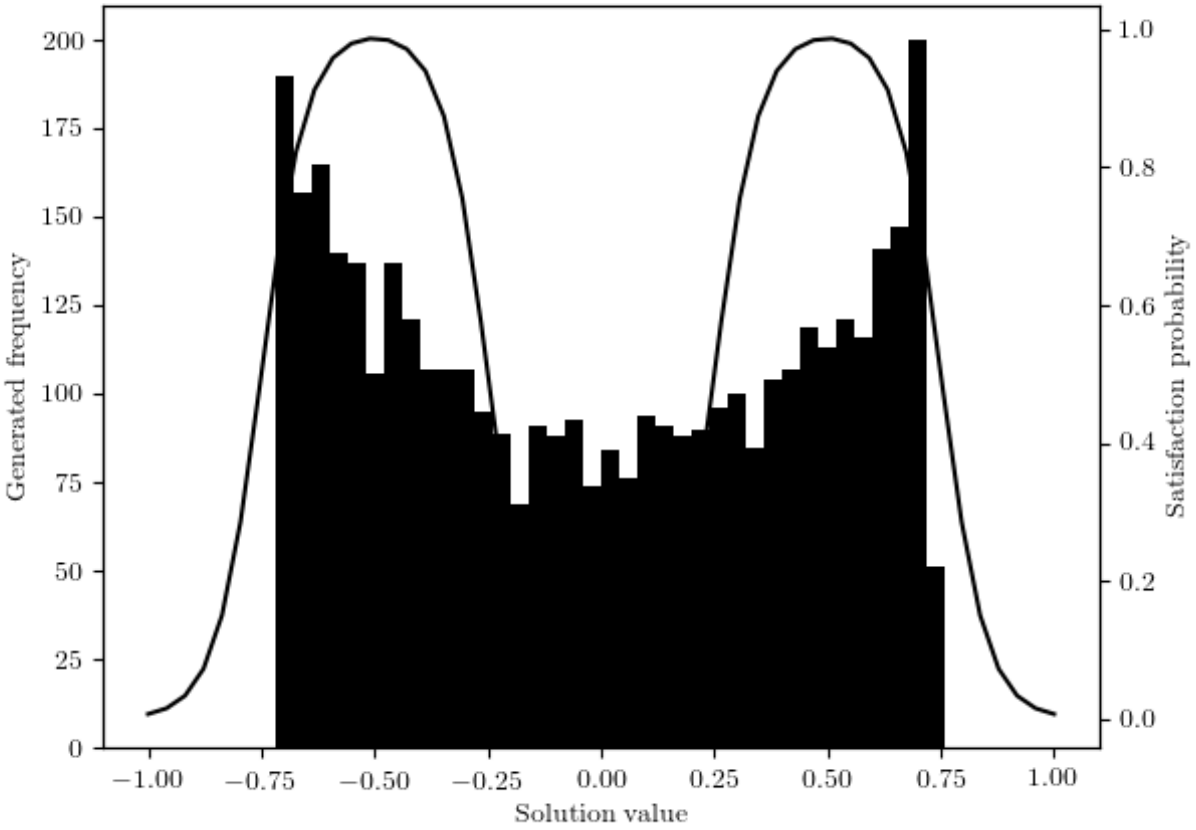


Figure 6.10: $\hat{V}_c'$ after pretraining until convergence using $q_{\text{id}}$, then trained to maximise $\hat{p}$ until convergence.

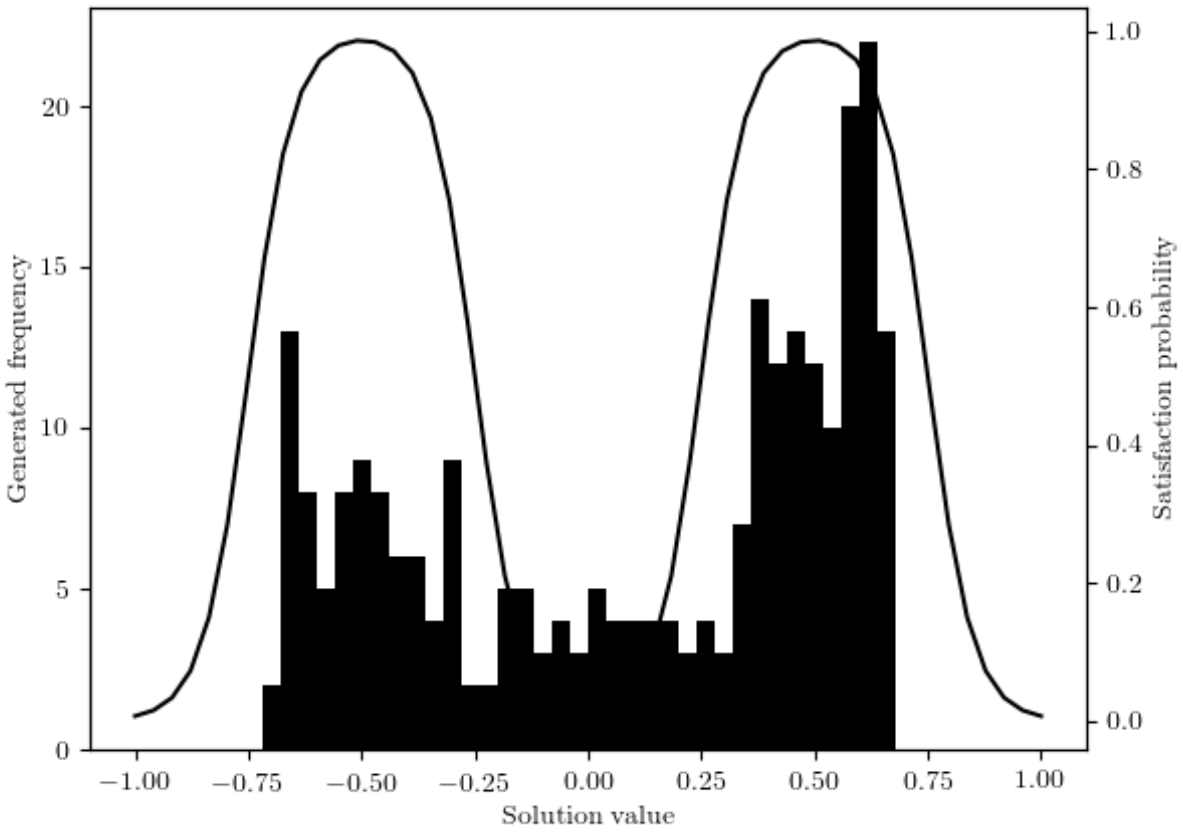


Figure 6.11: $\hat{V}_c'$ after pretraining until convergence using $q_{\text{sep}}$, then trained to maximise $\hat{p}$ until convergence.

Comparing Figure 6.6 to Figures 6.10 and 6.11 it is clear that pretraining greatly enhances the generator's ability to sample multiple modes. This comes at a cost, however; while the generator pretrained on $q_{\text{id}}$ samples approximately equally from both modes, it also frequently samples solutions from between the modes which are unlikely to satisfy $h$.

### 6.1.4 Maximising precision and spread

All experiments in this section used the parameters in Appendix F.

Another possibility is, after pretraining, to minimise a weighted sum of $\hat{p}$ and $q$, as if the spread loss were a substitute for $\hat{r}$. Using $q_{\text{id}}$, this has an intuitive explanation: $q_{\text{id}}$ penalises the relocation of probability mass between $L$ and $S$, thereby providing an incentive to $g'$ to seek out areas of high satisfaction probability which are more local.

Figure 6.12 shows $\hat{V}_c'$ after $q_{\text{id}}$ pretraining, then training to minimise $\hat{p} + q_{\text{id}}$ until convergence.

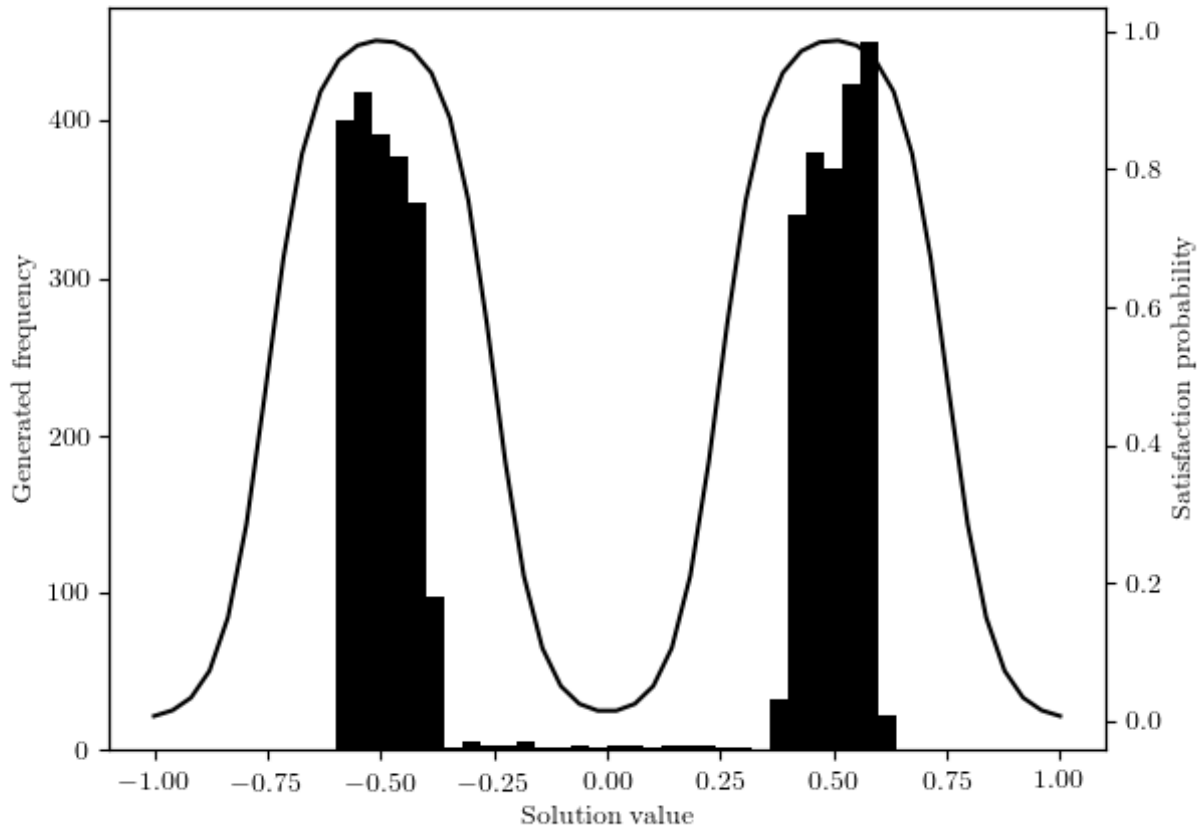


Figure 6.12: $\hat{V}_c'$ after pretraining, followed by training to maximise $\hat{p} + q_{\text{id}}$ until convergence.

This is clearly a result superior to any obtained previously, with $\hat{V}_c'$ covering both modes, and almost always in regions with a satisfaction probability above 0.8. Solutions less likely to satisfy $h$ are almost never sampled.

### 6.1.5 Constraint embeddings

The only part of the proposed architecture that has yet to be verified is the insertion of constraints in an embedded form at the output layer of the generator. This is achieved using a generalised alteration of $h$ introduced in Appendix G such that the position of each peak is controlled by one component of $c$. Code for replicating this function is reproduced in Appendix H and training was performed with parameters in Appendix I.

Generally, $\hat{V}_c'$ was able to adapt to account for the differences in $h$. Not only was it often able to distinguish between two modes, but also adjusted its mapping appropriately when the two peaks were near each other and so merged into one mode.

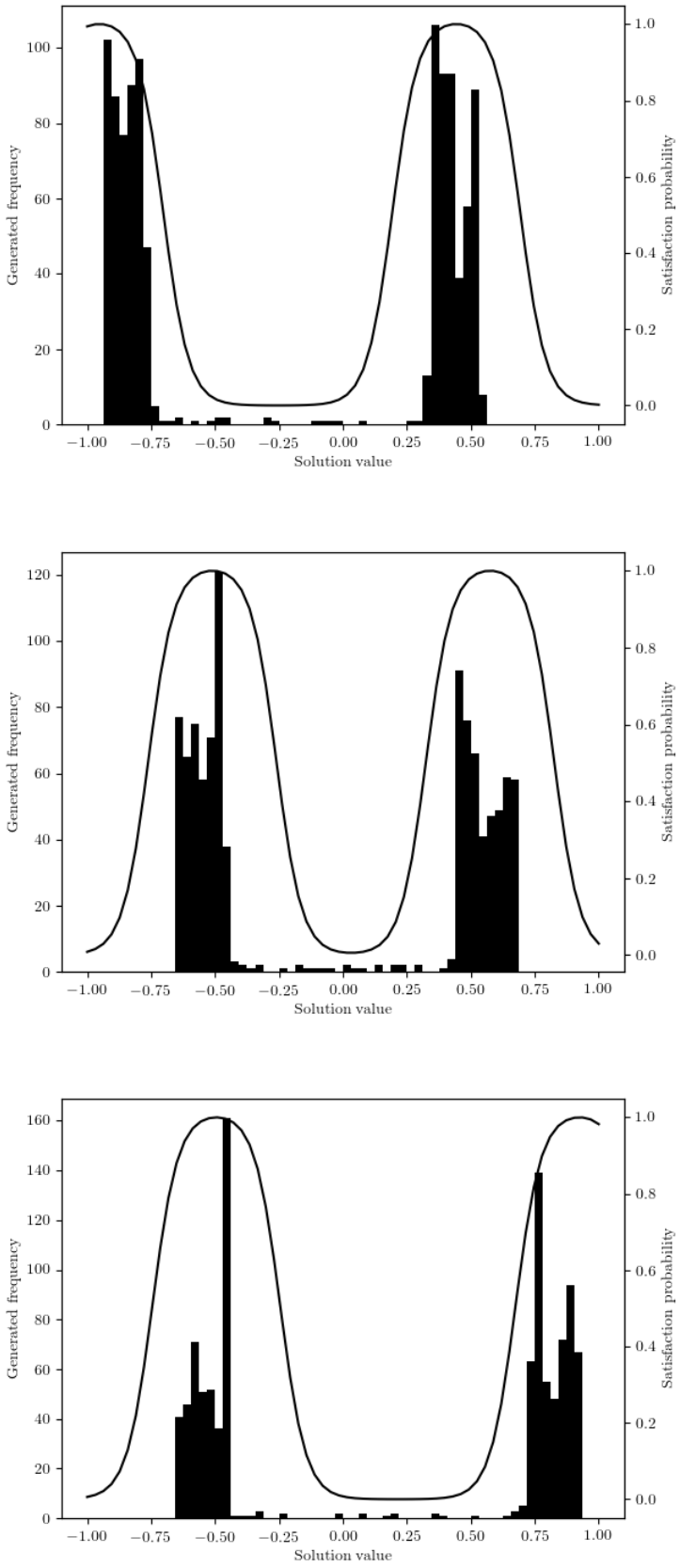


Figure 6.13: Examples of a generator learning to adjust its mapping to changing constraints.

Occasionally, when large regions of $S$ did not satisfy $c$, poor mapping choices such as those observed

in Figure 6.14 would occur. It is hypothesised that these are artefacts of the cost of moving probability mass from one side of $S$ to the other becoming greater than the gain in precision; this is partially confirmed by observations that this error happened less frequently once $w$ was reduced.

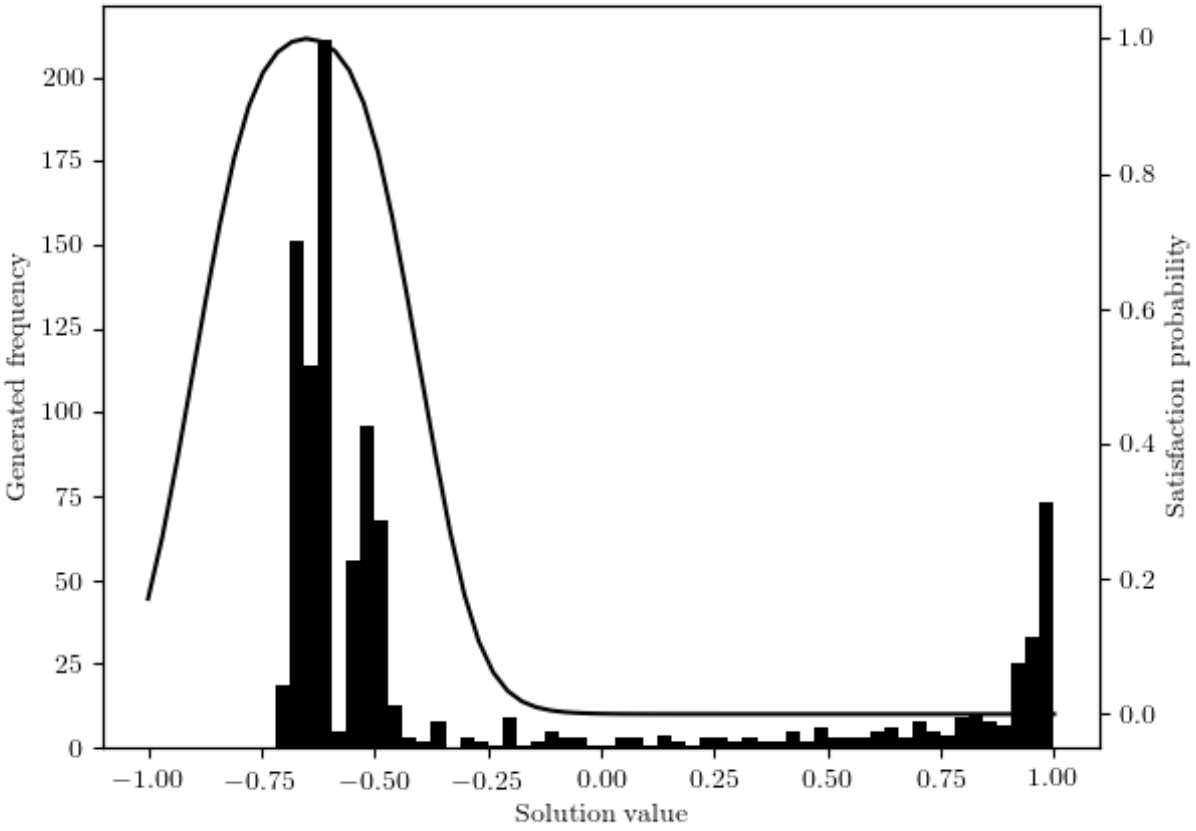


Figure 6.14: An example of a constraint whose viable space was so small compared to $S$ that $g'$ was forced to sample poor solutions to prevent penalties from $q_{\text{id}}$. This can be prevented by reducing $w$.

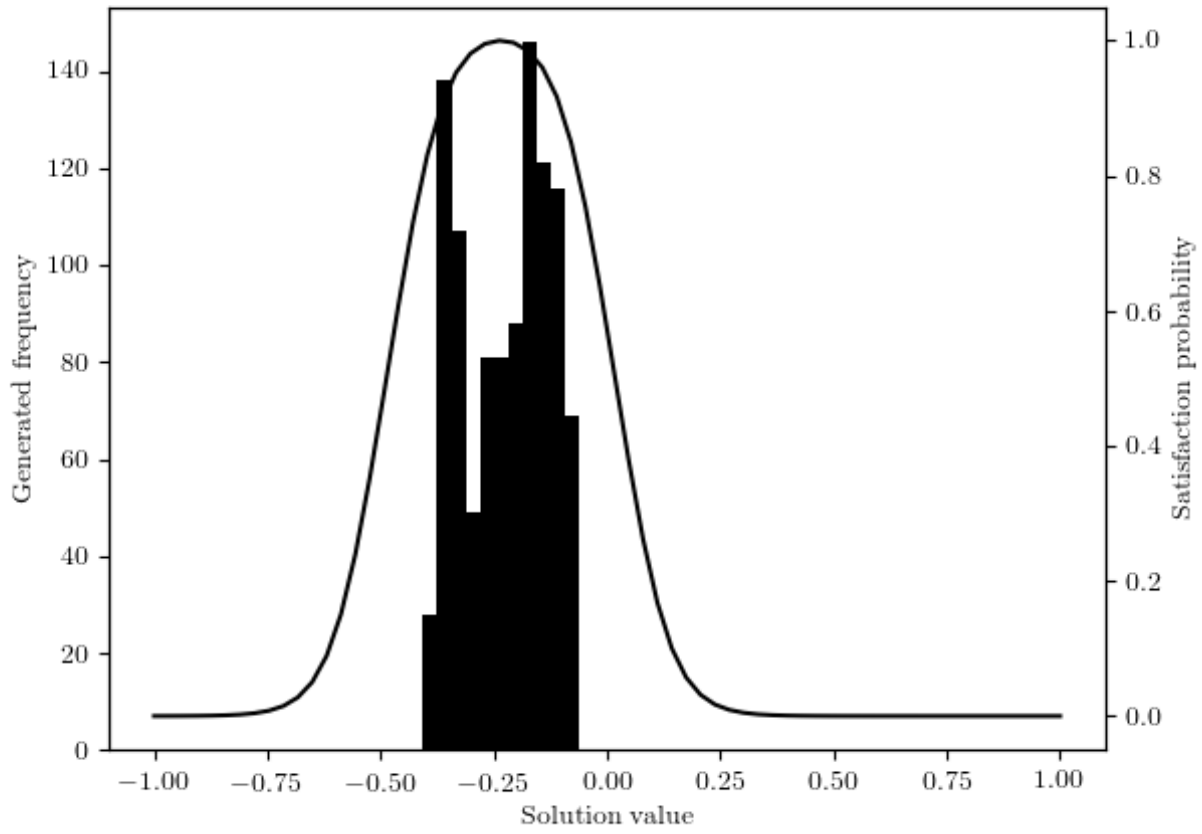


Figure 6.15: A constraint for which two peaks join together into a single peak, which is accurately reflected in $\hat{V}_c'$.

## 6.2 Method properties

The first half of this chapter described experiments performed on abstract, simplified engineering problems to inform the design of the project's proposed architecture. The remainder of this chapter will examine that architecture's performance in more depth and in more complex engineering environments.

### 6.2.1 Expected satisfaction probability

One crucial requirement of $\hat{V}_c'$ is that the solutions it produces should satisfy $c$. The expected satisfaction probability (ESF) will be used to measure the performance of the generator on various environments, and is calculated by taking the mean of $h'(c, s)$ over a batch of sampled solutions. Similarly, the median satisfaction probability (MSF) is the median $h'(c, s)$ of every solution in the batch.

### 6.2.2 Relative density of true solutions

Another requirement of $\hat{V}_c'$ is that it samples from the entire range of viable solutions. In all further experiments, either $q_{\text{id}}$ or $q_{\text{sep}}$ is used as a subtitute for $\hat{r}$ due to its intractibility. For the purposes of evaluating the generator's performance, the relative density of true solutions in $\hat{V}_c'$ will be calculated.

Calculating this value requires a sample of solutions from $\hat{V}_c$; this is achieved using the Metropolis-Hastings algorithm [34]. Finding the relative density of each of these solutions in $\hat{V}_c'$ requires creating an $n$-tree (a generalisation of a quadtree to an arbitrary number of dimensions) from a large number of samples from $g'$. For each true solution sampled from the Markov chain, the smallest bin within the $n$-tree containing that solution is found. The relative density of that solution is then defined as:

$$\text{relative density} = \rho = \frac{\text{population of bin/population of whole tree}}{\text{volume of bin/volume of } S} \tag{6.1}$$

Intuitively, if a solution has a relative density greater than 1 then it is being sampled more frequently than it would be if the generator simply output a uniform distribution; conversely, a relative density of less than 1 suggests it is being sampled less frequently. For a generator with good recall it could therefore be expected that the mean relative density over a sample of solutions taken from $\hat{V}_c$ by a MCMC algorithm should be above 1.

### 6.2.3 Data collection

All environments used in further experiments implemented a common interface that allowed the training of a generator upon them to output a large JSON file containing data about the experiment parameters and results. An exemplar JSON file is shown in Appendix N, but the results were not reproduced here in full due to their excessive size.

### 6.2.4 Effect of recall weight

A generator was trained on a more concrete engineering environment to examine the effect of tuning $w$ on the tradeoff between precision and recall. It was hypothesised that low $w$ causes high satisfaction probability but low relative density, whereas high $w$ causes the opposite.

The holes environment (Appendix J) was used to test this hypothesis. Datasets of $(c, s, h(c, s))$ tuples were created to train the discriminator, balanced such that roughly half of the examples were positives: samples were taken uniformly from $S$ and $C$, and the satisfaction probability was calculated for each pair. The training parameters used are reproduced in Appendix K.

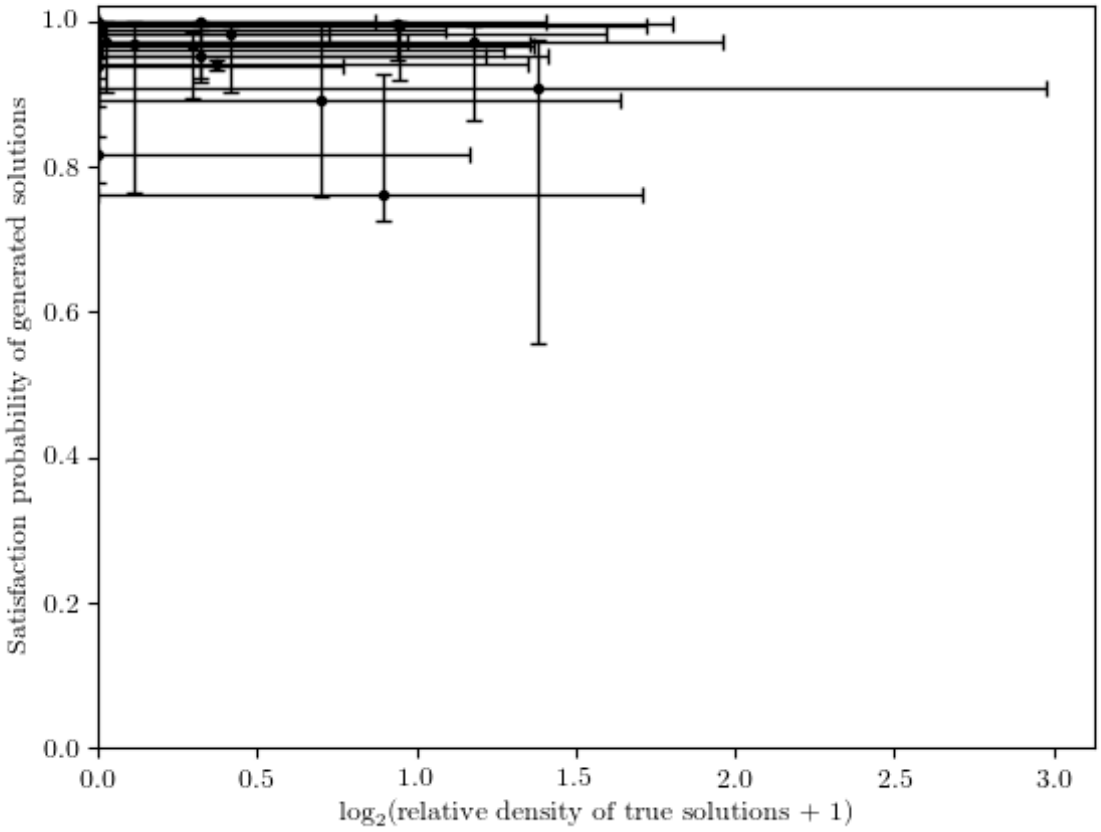


Figure 6.16: The median and interquartile range of the satisfaction probability and relative density of 128 generated and 128 true solutions to a sample of constraints. Relative density is given as $\log_2(\rho+1)$ to space the data while conserving the important lines $\rho = 0$ and $\rho = 1$. Generator trained with $w = 1$.

Immediately noticeable is the very high precision: only one of the constraints has a satisfaction probability of under 0.8 for at least half of its generated solutions. Recall is generally quite poor, however. Only a small proportion of the true solutions were sampled more frequently by $\hat{V}_c'$ than by a uniform distribution. The hypothesis suggests that increasing $w$ would shift the constraints to regions of higher $\rho$, at the cost of slightly decreasing $\hat{p}$.

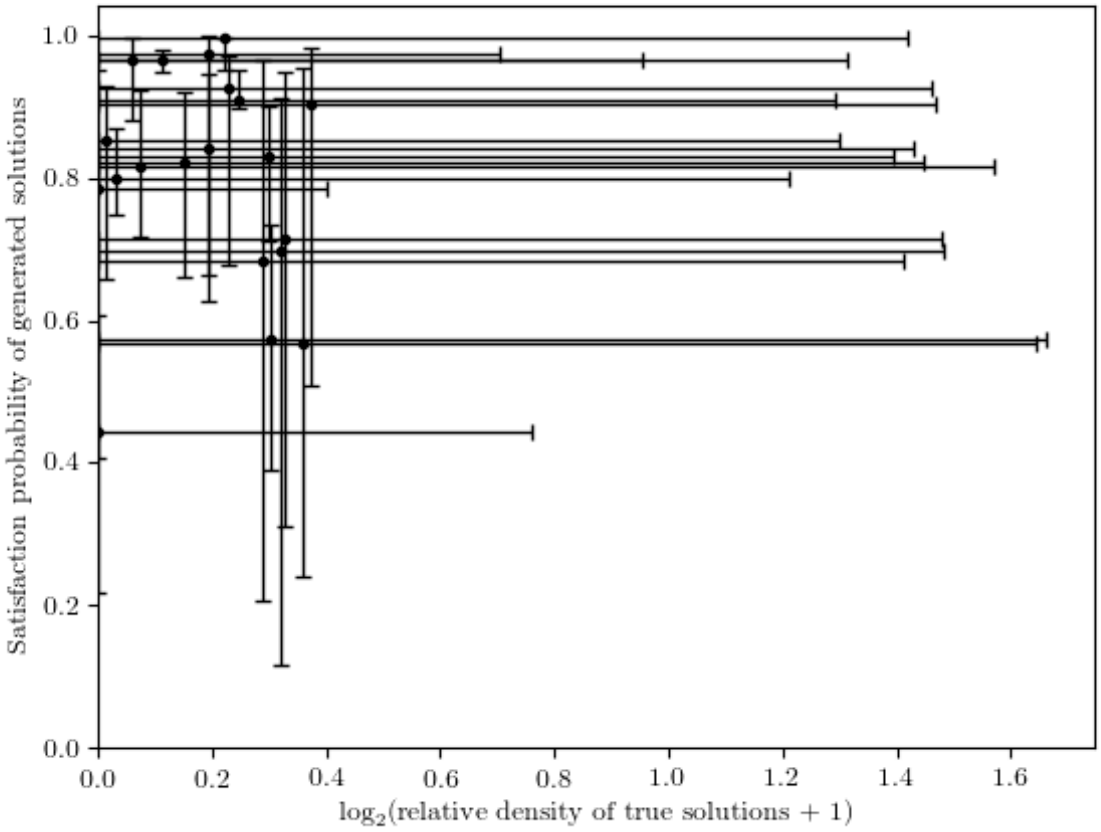


Figure 6.17: The median and interquartile range of the satisfaction probability and relative density of 128 generated and 128 true solutions to a sample of constraints. Generator trained with $w = 3$.

Figure 6.17 shows this to be partially incorrect, however: while a reduction in satisfaction probability is observed, increasing $w$ does not appear to have increased the relative density for the majority of constraints. But the spread loss does appear to be consistently reduced by increasing $w$ (Table 6.1), suggesting that $q_{\text{id}}$ is not an accurate gauge of recall. That increasing $w$ did trade precision for spread does suggest, however, that if a better substitute for recall could be obtained, tuning $w$ might have the intuitive effect hypothesised previously.

| | | After pretraining | | After training | |
|---|---|---|---|---|---|
| Experiment | $w$ | Spread loss | Precision loss | Spread loss | Precision loss |
| 1 | 0.500 | 0.116 | -0.452 | 0.213 | -0.650 |
| 2 | 0.500 | 0.092 | -0.503 | 0.156 | -0.643 |
| 3 | 0.500 | 0.098 | -0.466 | 0.225 | -0.647 |
| 4 | 0.500 | 0.103 | -0.391 | 0.300 | -0.638 |
| 5 | 0.500 | 0.119 | -0.410 | 0.229 | -0.637 |
| 6 | 1.000 | 0.103 | -0.420 | 0.173 | -0.632 |
| 7 | 1.000 | 0.093 | -0.324 | 0.181 | -0.659 |
| 8 | 1.000 | 0.091 | -0.537 | 0.174 | -0.644 |
| 9 | 1.000 | 0.136 | -0.457 | 0.136 | -0.631 |
| 10 | 1.000 | 0.113 | -0.438 | 0.143 | -0.634 |
| 11 | 2.000 | 0.121 | -0.452 | 0.113 | -0.549 |
| 12 | 2.000 | 0.104 | -0.431 | 0.132 | -0.600 |
| 13 | 2.000 | 0.085 | -0.435 | 0.105 | -0.515 |
| 14 | 2.000 | 0.090 | -0.435 | 0.164 | -0.617 |
| 15 | 2.000 | 0.110 | -0.429 | 0.124 | -0.591 |
| 16 | 3.000 | 0.074 | -0.442 | 0.100 | -0.558 |
| 17 | 3.000 | 0.088 | -0.443 | 0.121 | -0.544 |
| 18 | 3.000 | 0.123 | -0.444 | 0.108 | -0.551 |
| 19 | 3.000 | 0.100 | -0.423 | 0.099 | -0.511 |
| 20 | 3.000 | 0.103 | -0.427 | 0.111 | -0.487 |

Table 6.1: $\hat{p}$ and $q_{\text{id}}$ at different points throughout training.

### 6.2.5 Equality constraints

A common constraint in engineering problems is an equality constraint, specifying that a property of a design must equal a particular value. Such a constraint can be created using the proposed architecture by parameterising the constraint such that it limits the upper and lower acceptable bounds of a property, and then setting these two bounds to be arbitrarily close together.

The Branin function, $\beta(x, y)$, was chosen to test the ability of the generator to approximate equality constraints as it is a standard test function for which an equality constraint produces distinct contours (Appendix L). Having a solution dimension of 2 also allows simple visualisation of the results. To make passing the function and its arguments into the discriminator possible, both inputs and the output were scaled to be in $[0,\ 1]$. The constraint vector, in two dimensions, was then defined such

that the constraint is considered satisfied when:

$$c_1 \cdot c_2 \leq \beta(x, y) \leq c_2, \quad c_1, c_2 \in [0,\ 1] \tag{6.2}$$

and so:

$$h(c, s) = \begin{cases} \text{satisfied} & \text{if } c_1 \cdot c_2 \leq \beta(s_1, s_2) \leq c_2 \\ \text{unsatisfied} & \text{otherwise} \end{cases} \tag{6.3}$$

As with the environment in §6.2.4, datasets of various sizes were created and the proposed architecture was trained on the resulting data. Training parameters can be found in Appendix M. The generator produced good solutions over a large range of $V_c$ for most constraints, but was unable to find solutions to some constraints, as shown in Figure 6.18.

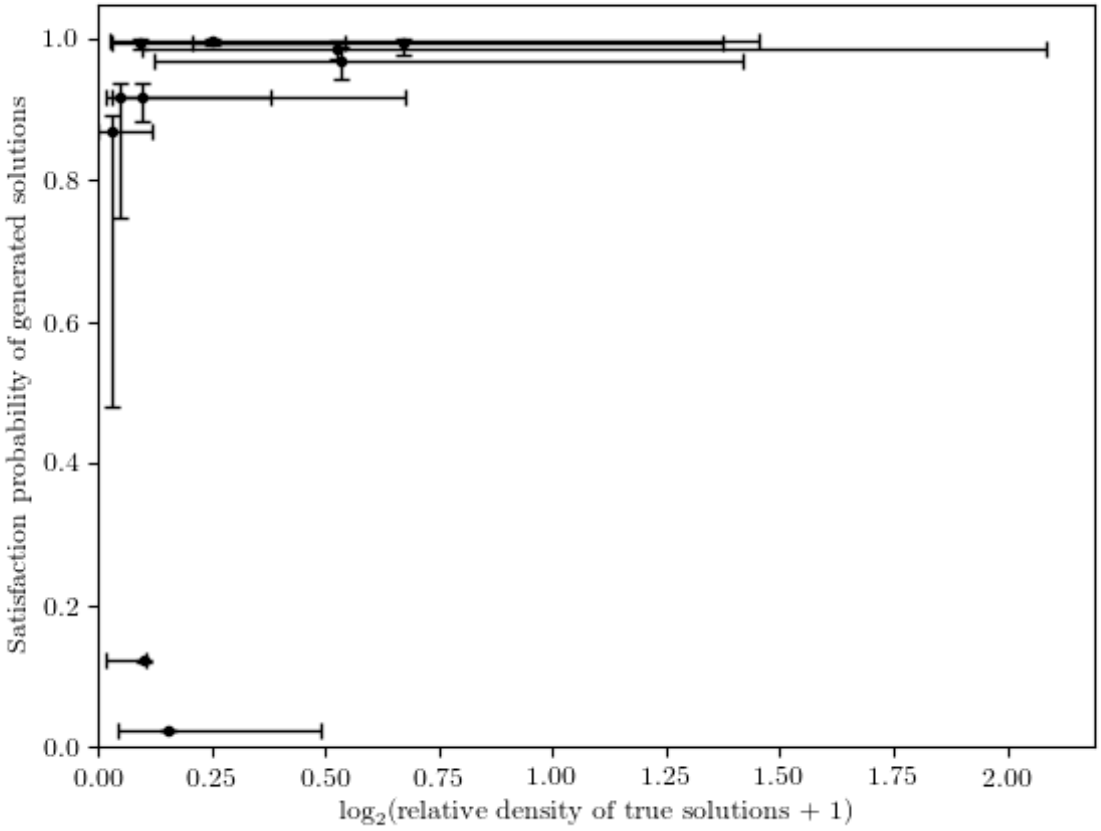


Figure 6.18: Solutions to a bounded Branin function, as sampled from 5 different instances of $\hat{V}_c'$.

After training, the constraint $c = [0.5,\ 0.5]$ was chosen, corresponding to regions of the solution space for which $\beta'$ outputs values in $[0.25,\ 0.5]$. Samples were then drawn from $\hat{V}_c'$ and plotted in $S$, with overlays for the output of both $h'$ and $h$, in Figure 6.19.

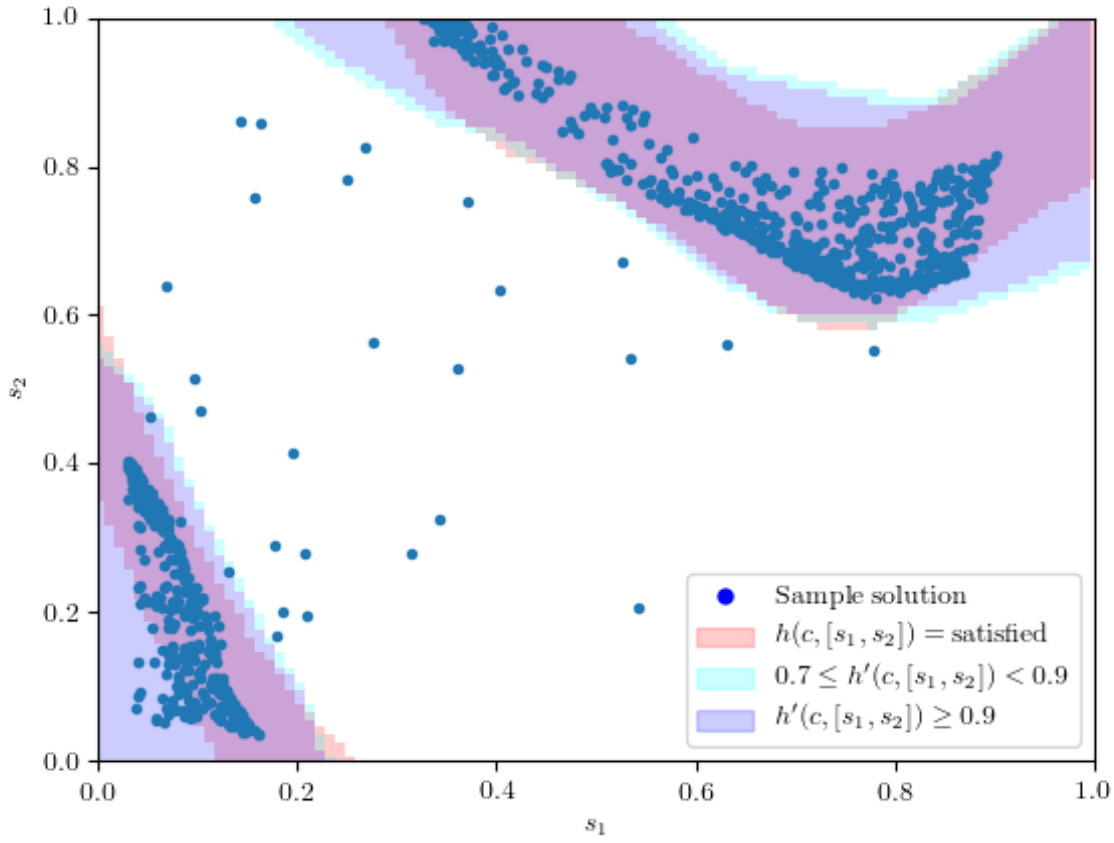


Figure 6.19: 1024 sample solutions, high-likelihood regions of the discriminator, and the regions containing true solutions for a generator trained on the bounded Branin function environment. $c = [0.5\ 0.5]$.

It is immediately clear that the vast majority of sampled solutions lie within regions of $S$ which the discriminator predicts are likely to satisfy $c$. Note also that $h'$ appears to have regions of very steep gradients (the cyan region in Figure 6.19 is thin compared to the blue region) and is generally accurate in its prediction of the boundaries of $V_c$. While the points are spread out sufficiently to cover much of $V_c$ and avoid coalescing into peaks, there are also regions near the periphery of $V_c$ which are undersampled, suggesting that some precision could be traded for an increase in recall.

Increasing $c_1$ to 0.7, constraining $V_c$ to those solutions for which $0.35 \leq \beta'(s) \leq 0.5$, yields similar results (Figure 6.20).

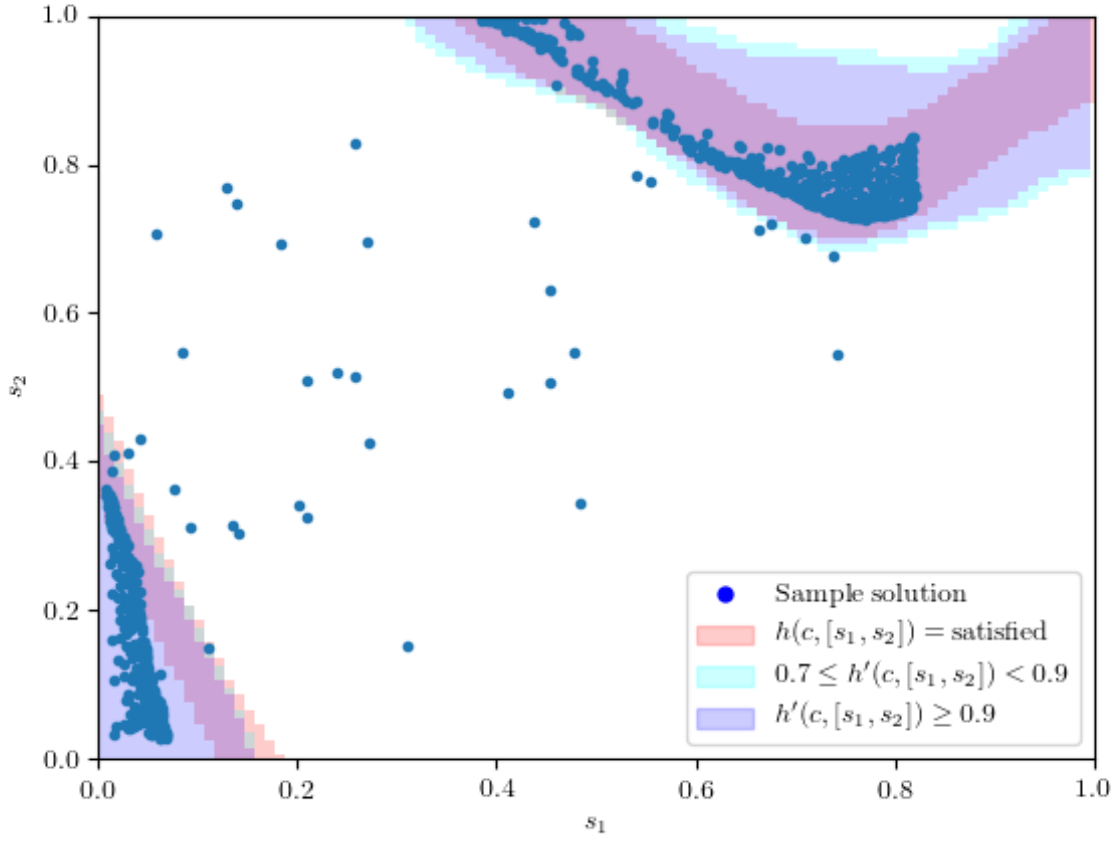


Figure 6.20: 1024 sample solutions, high-likelihood regions of the discriminator, and the regions containing true solutions for a generator trained on the bounded Branin function environment. $c = [0.7,\ 0.5]$.

The discriminator's predictions appear to be becoming less accurate as the bounded constraint approaches an equality constraint, failing to recognise that solutions near the origin do not satisfy $c$. While the discriminator made similar misclassifications for $c = [0.5,\ 0.5]$, the error is now affecting $\hat{V}_c'$ to the extent that a large number of samples are drawn from a region outside $V_c$.

$c_1$ was then increased to 0.9, representing regions for which $0.45 \leq \beta'(s) \leq 0.5$ (Figure 6.21). Of the two modes of $V_c$, one is fairly consistently sampled by $\hat{V}_c'$, while the other is missed almost entirely. It should be noted that the generator has learned to sample high-likelihood regions of $h'$, but that the discriminator misclassified the mode closer to the origin.

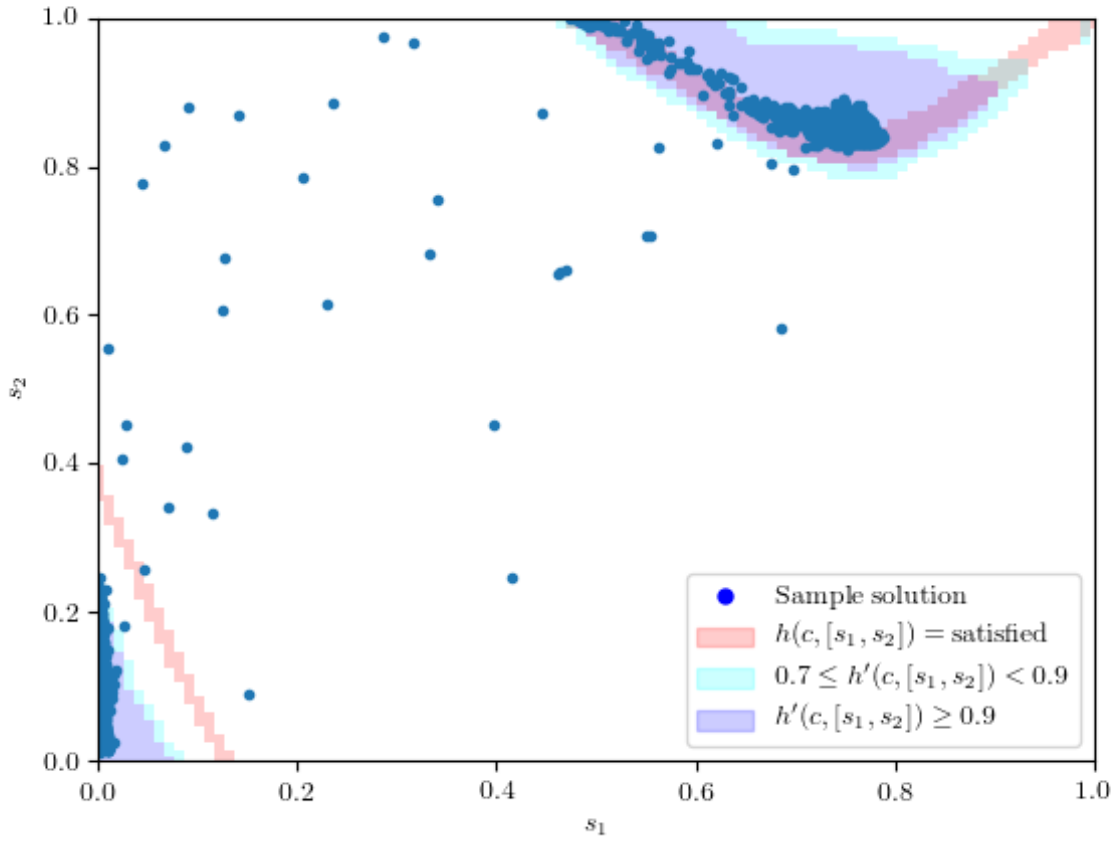


Figure 6.21: 1024 sample solutions, high-likelihood regions of the discriminator, and the regions containing true solutions for a generator trained on the bounded Branin function environment. $c = [0.9,\ 0.5]$.

$h'$ was therefore retrained using more data (16,384 examples as opposed to 1024) to see if a more accurate discriminator allowed sampling closer to an equality constraint. Figure 6.22 shows samples from $\hat{V}_c'$ for $c_1 = 0.98$, or $0.49 \leq \beta'(s) \leq 0.5$. This supports previous observations that the generator learns to sample high-likelihood regions of the discriminator, and therefore that if the discriminator is well trained, sampling solutions to constraints close to equality constraints is possible.

## 6.3 Latent space visualisation

To gain an intuitive understanding of how $g'$ maps $L \mapsto S$, a tool was written to plot invariants of each dimension of $L$ in $S$. $V_c$ was overlayed onto these plots. Because both its solution and constraint spaces are two-dimensional, the bounded Branin function environment was considered the most apt for visualisation purposes, and so the generator trained in that environment was used.

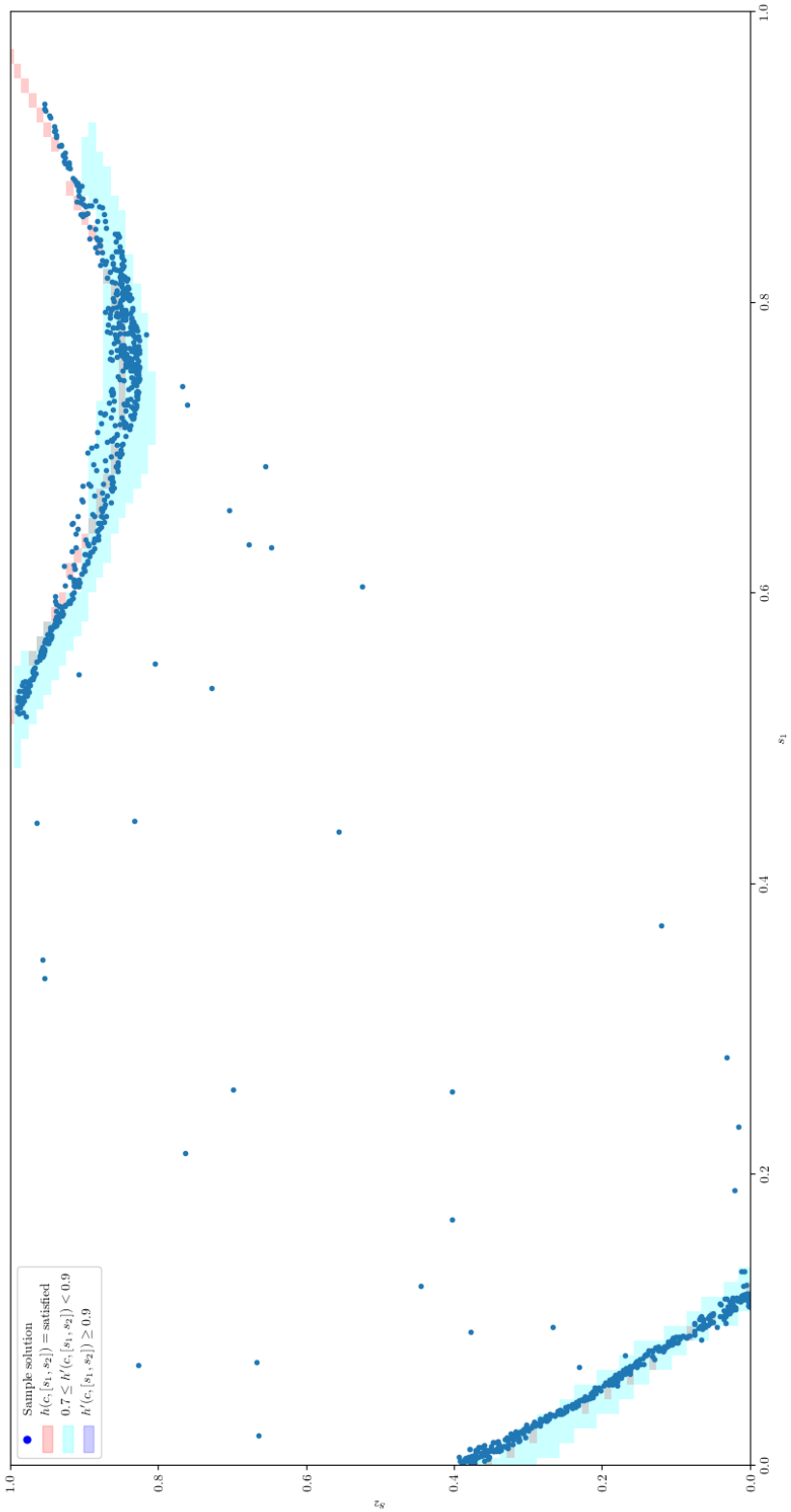


Figure 6.22: 1024 sample solutions for a generator trained on the bounded Branin function environment for $c = [0.98, 0.5]$.

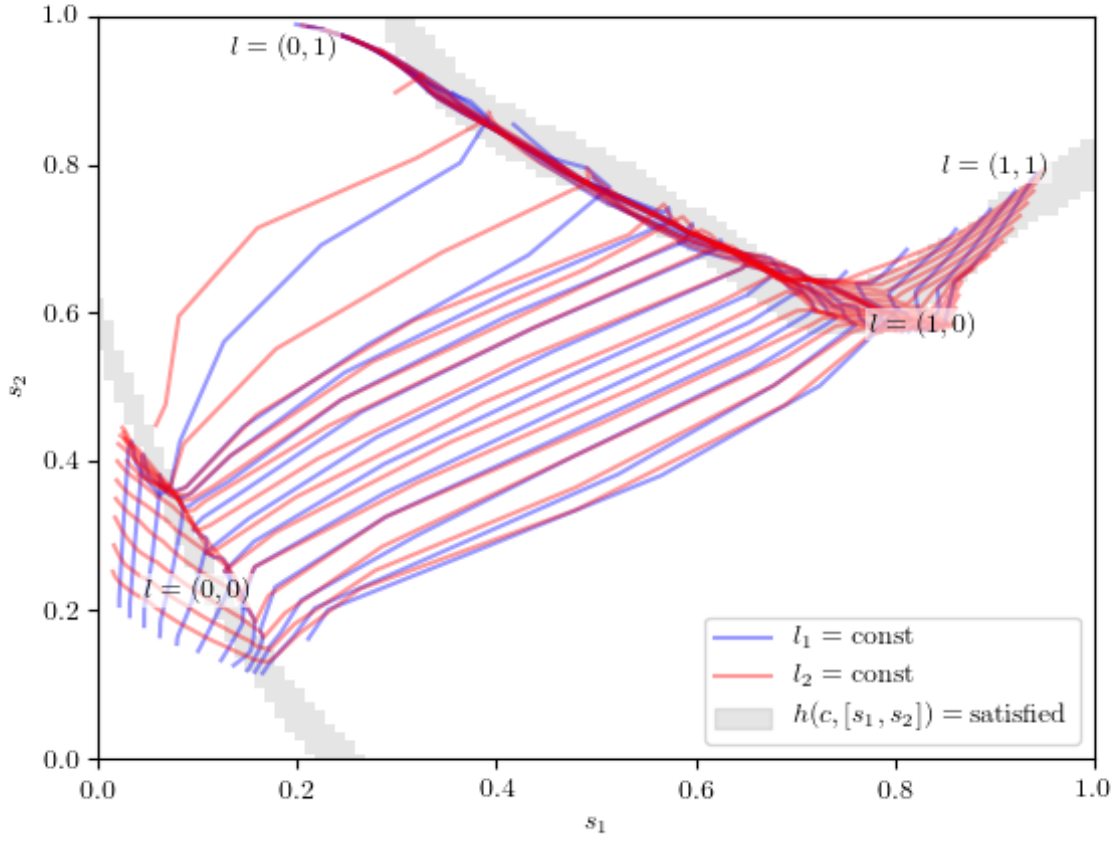


Figure 6.23: Latent invariants in the solution space of the bounded Branin environment for $c = [0.8,\ 0.3]$.

Figure 6.23 shows a highly constrained case where two distinct, narrow bands of viable solutions exist. As expected given previous results, the latent space is compressed within these two regions, and sparse between them.

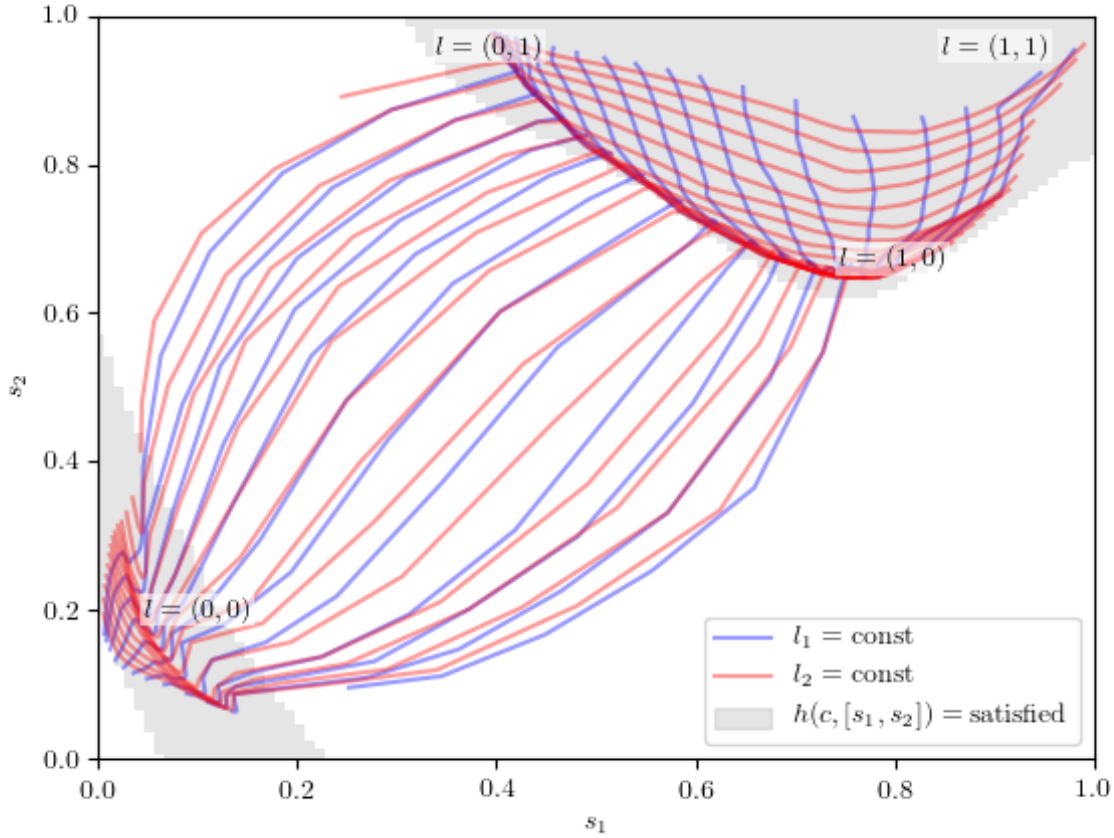


Figure 6.24: Latent invariants in the solution space of the bounded Branin environment for $c = [0.4,\ 0.7]$.

The constraint is relaxed somewhat in Figure 6.24, giving way to much larger areas of viable solutions. The latent space is, correctly, more spread out in this plot, but some parts of $S$ − including some viable solutions − exist completely outside $L$. This might explain why, in §6.2.4, so many true solutions were found to have no likelihood of occurring in $\hat{V}_c'$.

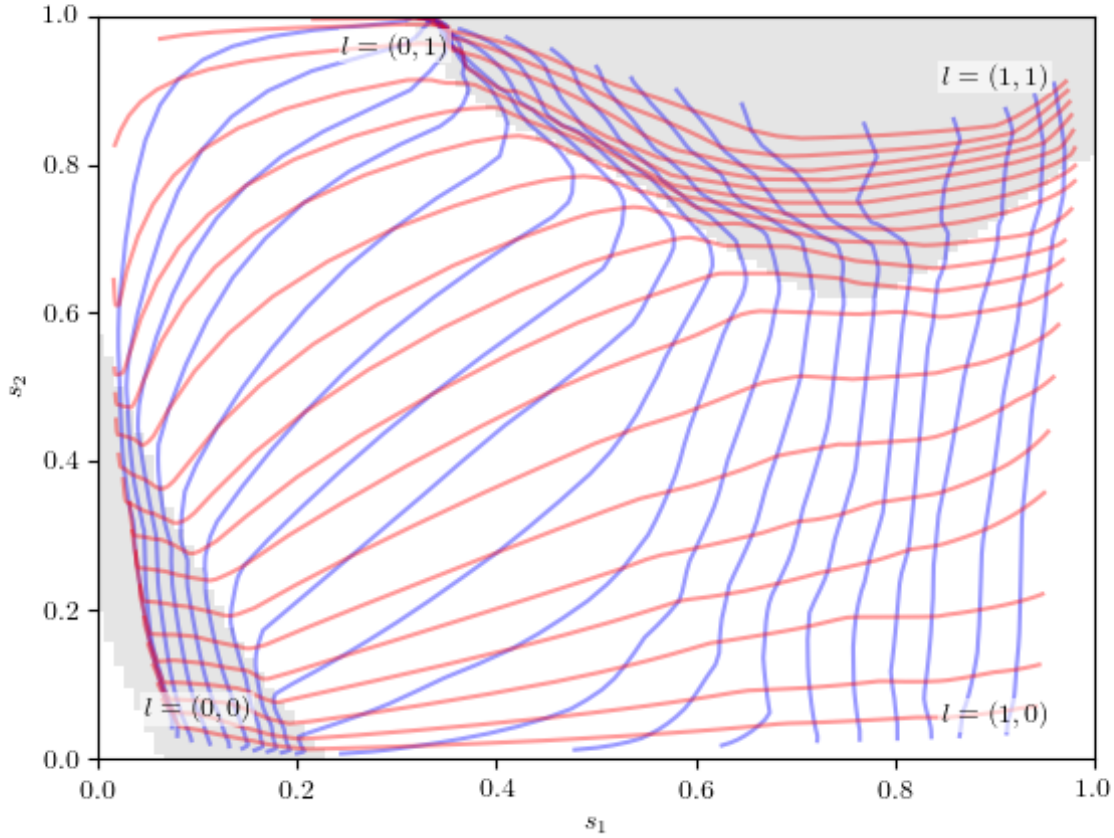


Figure 6.25: Latent invariants in the solution space of the bounded Branin environment for $c = [0.4,\ 0.7]$, trained with $w = 10$.

The same constraint was sampled, but using a generator trained with $w = 10$ (Figure 6.25). A latent space which covers a wider range of the solution space is clearly observed, allowing more viable solutions from the mode near the origin to be sampled. This comes at the cost, however, of more frequently sampling regions of low satisfaction probability.

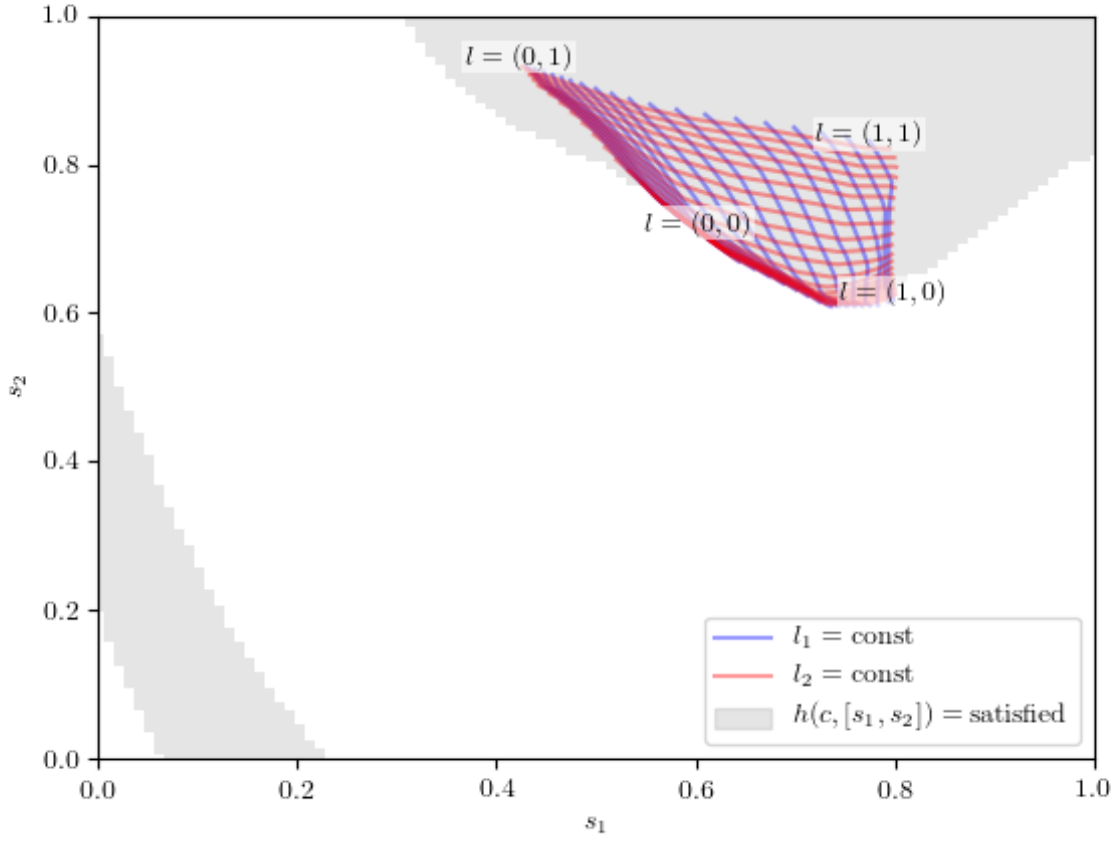


Figure 6.26: Latent invariants in the solution space of the bounded Branin environment for $c = [0.4,\ 0.7]$, trained with $w = 0.5$.

Correspondingly, reducing the recall weight to 0.5 shrinks the latent space considerably and causes one mode to be missed entirely, in spite of the fact that all sampled solutions do satisfy the constraint (Figure 6.26). These visualisations provide intuitive explanations for the tradeoff explored in §6.2.4.

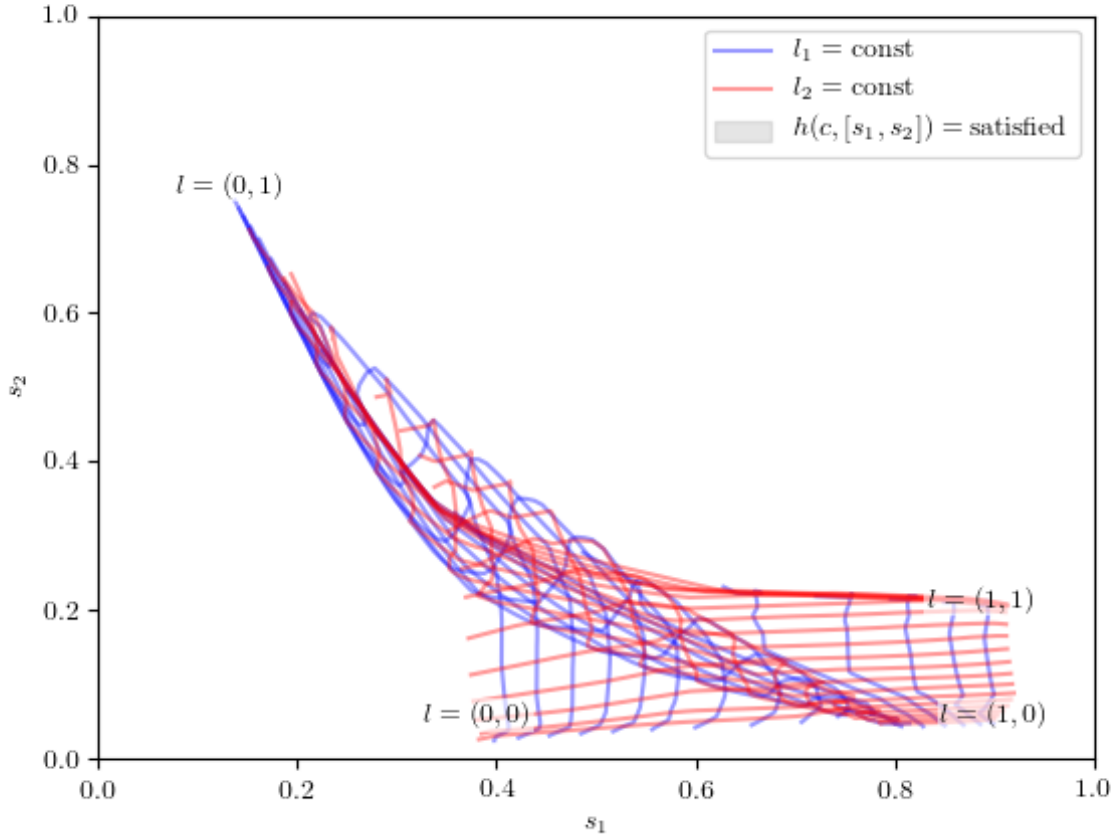


Figure 6.27: Latent invariants in the solution space of the bounded Branin environment for $c = [0.0,\ 0.0]$, a degenerate case with no solutions.

For the sake of completeness, Figure 6.27 shows the behaviour of $g'$ in a degenerate case with no solutions. Far fewer sparse regions appear here than in the other examples, with the entirety of $L$ relatively compressed. It also appears to frequently overlap itself, a phenomena not frequently seen for other constraints.

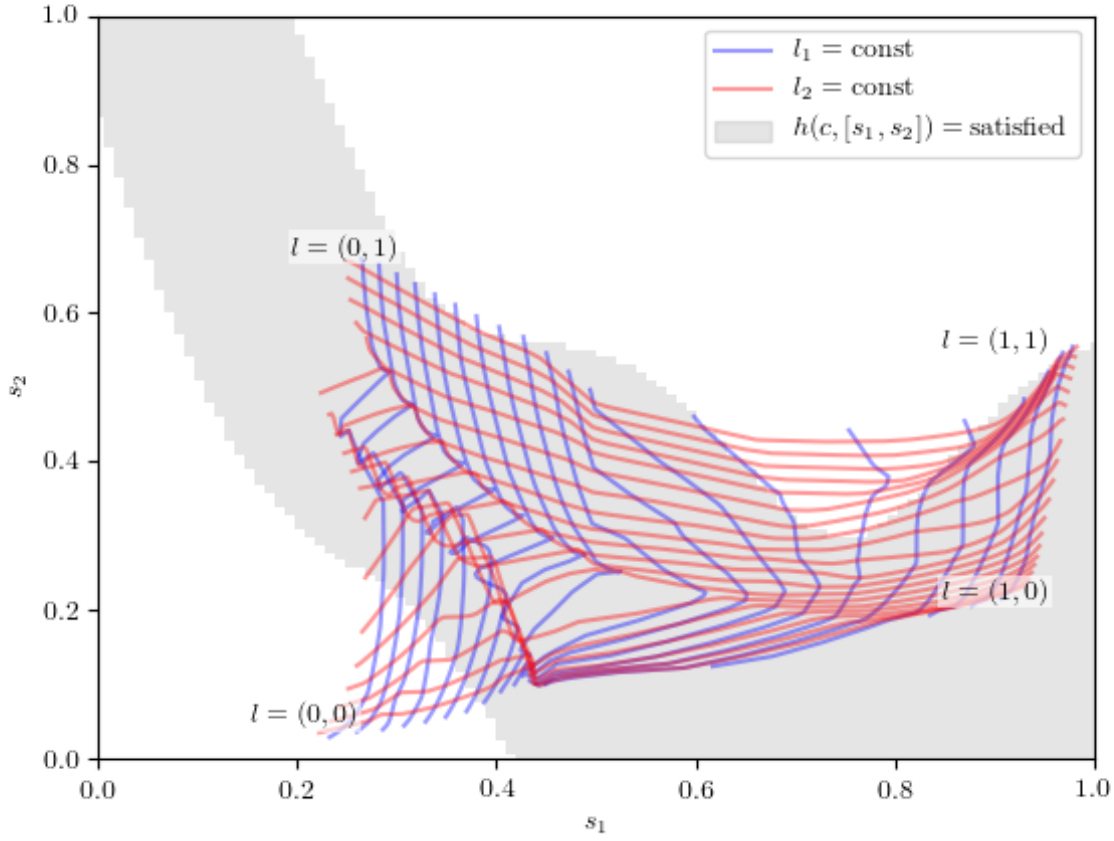


Figure 6.28: Latent invariants in the solution space of the bounded Branin environment for $c = [0.0,\ 0.1]$.

Finally, Figure 6.28 shows a fairly lenient case with a wide range of potential solutions. Here, poor recall is particularly evident, with relatively few of the potential solutions having latent equivalents in spite of the fact that most of $L$ maps to a viable solution.

# Chapter 7

# Overview

## 7.1 Further work

### 7.1.1 Other recall substitutes

Many of the limitations of the proposed architecture were found to be related to the inability to calculate $\hat{r}$. Though two metrics were introduced which provided acceptable results when used as a subtitute for $\hat{r}$, they were not designed for this purpose and it is hypothesised that better results could be obtained by creating a range of metrics specifically tailored for alleviating the effects of low recall.

### 7.1.2 Simplex intersection

The intractability of estimating the generator's recall was found to be related to a lack of a simplex intersection algorithm in an arbitrary number of dimensions. While research did not find such an algorithm, it also did not find any theses concluding that such an algorithm cannot exist. Further work into a simplex intersection algorithm could therefore allow direct optimisation of $\hat{r}$, eliminating a major flaw in the proposed architecture.

### 7.1.3 Composite constraints

Currently, only one constraint can be fed into $g'$ at once. It might be possible to train a RNN to combine constraint vectors in ways resembling set operations applied to $V_c$. This could allow multiple constraints to be combined, thereby making accessible the solutions to more complex, composite constraints.

### 7.1.4 Direct constraint embedding

The weights and biases of the output layer of $g'$ are directly estimated by a function of $c$ in the current architecture. It was not investigated whether $g'$ would be capable of learning effective mappings if $c$

were used directly as the weights of the final layer, without embedding. If this were the case, training times could be shortened by reducing the number of parameters used by the generator.

## 7.2 Conclusion

This project defined a generic engineering problem and postulated that such problems could be simplified by learning a latent mapping into the solution space. It was shown that in order for this mapping to be modelled by an artificial neural network it was necessary to view the space of viable solutions as a probability distribution over the space of valid solutions. A tradeoff between the precision and recall of the mapping was introduced and, while a method was derived for estimating the precision of a learned viable distribution, it was shown that estimating the recall of a distribution is intractable without further work. Two substitute metrics, referred to as spread metrics, were then introduced which, it was hypothesised, would encourage the mapping to exhibit features similar to if it had been trained on a direct estimation of the recall.

Using a simple, abstract environment, experiments were conducted which suggested that optimising for precision produces a mapping capable of sampling viable solutions. It was also shown that maximising a weighted sum of precision and one of the spread metrics was a suitable proxy for optimising precision and recall. Experimentation on more concrete, complex environments proved the robustness of the precision metric, but saw the substitutes for recall fail to force a complete mapping of all viable solutions.

The ability of the mapping to satisfy equality constraints was then examined, and was found to satisfy boundary constraints almost equivalent to equality constraints with good precision. Some nuanced solutions to equality constraints were missed, however.

Visualisations of the learned latent space were explored to gain an intuitive understanding of the generator. Observations were made which provided plausible explanations for the shortcomings of the proposed architecture discovered in earlier experiments, and areas of further research were suggested which could overcome them.

# Bibliography


[1] S. Ruder, *An overview of gradient descent optimization algorithms.* arXiv:1609.04747v2 [cs.LG], 15 Jun 2017.

[2] G. Hinton, N. Srivastava, K. Swersky, *Overview of mini-batch gradient descent.* University of Toronto [online lecture notes] 2017. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[3] D.P. Kingma, J.L. Ba, *Adam: A Method for Stochastic Optimization.* arXiv:1412.6980v9 [cs.LG], 30 Jan 2017.

[4] J. Kennedy, R. Eberhart, *Particle Swarm Optimisation.* Washington, DC, 2012.

[5] J. Carr, *An Introduction to Genetic Algorithms* [online]. May 16, 2014, `https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf`.

[6] K. Hornik, *Approximation Capabilities of Multilayer Feedforward Networks.* Neural Networks, 1991, Volume 4, Issue 2, Pages 251-257.

[7] S. Liang, R. Srikant, *Why Deep Neural Networks for Function Approximation?.* arXiv:1610.04161v2 [cs.LG], 3 Mar 2017.

[8] S.J. Raudys, A.K. Jain, *Small Sample Size Effects in Statistical Pattern Recognition: Recommendations for Practitioners.* IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 13, No. 3, 3 March 1991.

[9] R. Caruana, S. Lawrence, L. Giles, *Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping.* NIPS, 2001.

[10] I. Goodfellow, Y. Bengio, A. Courville, 2016. *Deep Learning* [online], MIT Press. `http://www.deeplearningbook.org/contents/regularization.html`.

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of Machine Learning Research, 15 (Jun): 1929-1958, 2014.

[12] S. Ioffe, C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv:1502.03167v3 [cs.LG], 2 Mar 2015.

[13] X. Li, S. Chen, X. Hu, J. Yang, *Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift*. arXiv:1801.05134v1 [cs.LG], 16 Jan 2018.

[14] R.M. Zur, Y. Jiang, L.L. Pesce, K. Drukker, *Noise injection for training artificial neural networks: A comparison with weight decay and early stopping*. Med Phys. 2009 Oct; 36(10): 4810–4818.

[15] S. Nah, T.H. Kim, K.M. Lee, *Deep Multi-scale Convolutional Neural Network for Dynamic Scene Deblurring*. arXiv:1612.02177v2 [cs.CV], 7 May 2018.

[16] T. Nguyen, K. Mori, R. Thawonmas, *Image Colourization Using a Deep Convolutional Neural Network*. ASIAGRAPH Conference, 2016.

[17] D.E. Rumelhart, G. Hinton, R.J. Williams, *Learning internal representations by error propagation*. Parallel Distributed Processing, Vol 1: Foundations, MIT Press, Cambridge, MA, 1986.

[18] P. Vincent, H. Larochelle, Y. Bengio, P. Manzagol, *Extracting and Composing Robust Features with Denoising Autoencoders*. Université de Montréal, 2008.

[19] D.P. Kingma, M. Welling, *Auto-Encoding Variational Bayes*. arXiv:1312.6114v10 [stat.ML], 1 May 2014.

[20] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinik, S. Mohamed, A. Lerchner, *beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*. ICLR, 04 Nov 2016.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, *Generaitve Adversarial Nets*. arXiv:1406.2661v1 [stat.ML], 10 Jun 2014.

[22] F. Horger, T. Würfl, V. Christlein, A. Maier, *Deep Learning for Sampling from Arbitrary Probability Distributions*. arXiv:1801.04211v2 [cs.LG], 10 Jul 2018.

[23] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, W. Shi, *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*. arXiv:1609.04802v5 [cs.CV], 25 May 2017.

[24] J. Zhu, R. Park, P. Isola, A.A. Efros, *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. arXiv:1703.10593v6 [cs.CV], 15 Nov 2018.

[25] R. Anirudh, J.J. Thiagarajan, B. Kailkhura, T. Bremer, *An Unsupervised Approach to Solving Inverse Problems using Generative Adversarial Networks*. arXiv:1805.07281v2 [cs.CV], 4 Jun 2018.

[26] D. Bang, H. Shim, *MGGAN: Solving Mode Collapse using Manifold Guide Training*. arXiv:1804.04391v1 [cs.CV], 12 Apr 2018.

[27] R. Socher, C.C. Lin, A.Y. Ng, C.D. Manning, *Parsing Natural Scenes and Natural Language with Recursive Neural Networks*. Stanford University, 2011.

[28] T. Mikolov, K. Chen, G. Corrado, J. Dean, *Efficient Estimation of Word Representations in Vector Space*. arXiv:1301.3781v3 [cs.CL], 7 Sep 2013.

[29] Q. Le, T. Mikolov, *Distributed Representations of Sentences and Documents*. arXiv:1405.4053v2 [cs.CL], 22 May 2014.

[30] G. Ou, Y.L. Murphey, *Multi-class pattern classification using neural networks*. Pattern Recognition, Volume 40, Issue 1, Pages 4-18, 2007.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*. 2015.

[32] J. Park, I.W. Sandberg, *Universal Approximation Using Radial-Basis-Function Networks*. Neural Computation, Volume 3, Issue 2, June 1991.

[33] Y. Bengio, P. Simard, P. Frasconi, *Learning long-term dependencies with gradient descent is difficult*. IEEE Trans Neural Netw., Volume 5, Issue 2, 1994.

[34] C.P. Robert, *The Metropolis-Hastings algorithm*. arXiv:1504.01896v3 [stat.CO], 27 Jan 2016.

[35] N.J. Wildberger, *Simplices and simplicial complexes*. `https://www.youtube.com/watch?v=Uq4dTjHfLpI`, 7 Oct 2012.

[36] A.F.M. Agarap, *Deep Learning using Rectified Linear Units (ReLU)*. arXiv:1803.08375v2 [cs.NE], 7 Feb 2019.

[37] A. Krogh, J.A. Hertz, *A Simple Weight Decay Can Improve Generalization*. NIPS, 1991.

[38] D. Bingham, *Test Functions and Datasets*. `https://www.sfu.ca/~ssurjano/branin.html`, accessed March 2019.

[39] R. Giryes, G. Sapiro, A.M. Bronstein, *Deep Neural Networks with Random Gaussian Weights: A Universal Classification Strategy?*. arXiv:1504.08291v5 [cs.NE], 14 Mar 2016.

[40] S. Hochreiter, J. Schmidhuber, *Long Short-Term Memory*. Neural Computation, Volume 9 Issue 8, 15 November 1997.

[41] O. Sigmund, *A 99 line topology optimization code written in Matlab*. Struct Multidisc Optim 21, 120-127, 2001.

[42] R. Luo, F. Tian, T. Qin, E. Chen, T. Yiu, *Neural Architecture Optimization*. arXiv:1808.07233v4 [cs.LG], 31 Oct 2018.

[43] Y. Yu, T. Hur, J. Jung, I.G. Jang, *Deep learning for determining a near-optimal topological design without any iteration*. Springer Nature, 2018.

# Appendices

## A Code

The experiments for this project were programmed in Python using Tensorflow as the primary machine learning library. All code used for the project can be found at `https://github.com/aatack/IP`, with some high-level wrappers for Tensorflow graphs implemented separately in `https://github.com/aatack/Wise`.

## B Sampling from axis-aligned hypercubes

While vector spaces in general are computationally expensive to sample, some special cases can be sampled by efficient algorithms. One such example is an axis-aligned hypercube, which is constrained in every dimension by a lower and upper bound. Any function that draws pseudo-random samples from a uniform distribution can be called multiple times and concatenated to produce a point which will always be inside that hypercube; this is also capable of producing any point inside the hypercube ($p = r = 1$). An example of this in Python is shown below.

```
>>> from numpy.random import uniform
>>> uniform(low=0.0, high=1.0, size=(5,))
array([0.0051103 , 0.60075015, 0.25896835, 0.56730558, 0.05386095])
```

Listing 7.1: Python code to sample from an axis-aligned hypercube.

A simple benchmarking function can be written to show that this process is also efficient:

```
>>> import time
>>> def benchmark(f, n_runs):
...     start = time.time()
...     for _ in range(n_runs):
...         f()
...     end = time.time()
...     return (end - start) / n_runs
...
>>> benchmark(lambda: uniform(low=0.0, high=1.0, size=(5,)), 10000)
```

```
1.8946647644042968e-06
```

Listing 7.2: A simple benchmarking function to prove the efficiency of sampling from an axis-aligned hypercube. Each sample takes an average of 1.89 microseconds for a 5-dimensional hypercube.

# C Parameterising orthogonal matrices

§4.3.2 describes the need for an orthogonal matrix whose values can be changed by an optimiser. Generally this is not possible since the majority of matrices will not be orthogonal. Hence a transformation is needed from some matrix whose values can vary freely to an orthogonal matrix.

Let $\ell$ be a function which takes a matrix and returns its lower triangle, while $W$ is a square matrix whose values vary freely. Therefore $\ell(W)$ is lower triangle while $\ell(W)^T$ is upper triangle, and their diagonals will be equal.

If $W_{\text{skew}}$ is defined as

$$W_{\text{skew}} = \ell(W) - \ell(W)^T \tag{7.1}$$

then:

$$W_{\text{skew}}^T = \ell(W)^T - \ell(W) \tag{7.2}$$

$$-W_{\text{skew}}^T = -\ell(W)^T + \ell(W) \tag{7.3}$$

and so $W_{\text{skew}} = -W_{\text{skew}}^T$, which is sufficient to prove that $W_{\text{skew}}$ is skew-symmetric. It is also known that the matrix exponential of a skew-symmetric matrix always results in an orthogonal matrix. So it is concluded that as long as $W$ is a square matrix,

$$W_{\text{orth}} = e^{\ell(W) - \ell(W)^T} \tag{7.4}$$

is orthogonal.

# D KL-divergence training parameters

The parameters used while performing the KL-divergence minimisation experiment in §6.1.1 are reproduced below.

```
{
    "values": {
        "type": "float",
        "precision": 32
    },
    "epsilon": 0.001,
    "batchSize": 64,
```

```
    "epochs": 2084,
    "startingParameters": {
        "means": [0.1, 0.2, 0.5, 1.0, 2.0],
        "stddevs": {
            "similar": 1.0,
            "dissimilar": 0.1
        }
    },
    "evaluationFrequency": 16,
    "optimiser": {
        "type": "adam",
        "learningRate": 0.001,
        "betaOne": 0.9,
        "betaTwo": 0.999,
        "epsilon": 10^-8
    }
}
```

Listing 7.3: Experimental parameters for minimising the KL-divergence of two distributions.

## E Unimodal constraint satisfaction function

A unimodal constraint satisfaction was defined for a solution dimensionality of 1 and a constraint dimensionality of 0, meaning that the function does not take into consideration any constraint parameters.

$$h(c, s) = \sigma(20s - 6) - \sigma(20s - 14) \tag{7.5}$$

where

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7.6}$$

Numerically integrating $h$ between $s = -1$ and $s = 1$ gives an area of $0.3999$, so the probability density function is $\hat{V}_c(s) \approx 2.5008\, h(c, s)$.

## F Precision optimisation training parameters

The parameters used to train a generator to match an arbitrarily defined constraint satifaction function are reproduced below. Because $h$ is not parameterised by constraints, no constraint embedder networks were used, and the last layer of the generator was trained normally.

```
{
    "histogramSamples": 256,
```

```
    "generator": {
        "layers": [
            {
                "nodes": 8,
                "activation": "leakyRelu"
            },
            {
                "nodes": 8,
                "activation": "leakyRelu"
            },
            {
                "nodes": 1,
                "activation": "tanh"
            }
        ],
        "initialisation": "glorot"
    },
    "batchSize": {
        "default": 4096,
        "identitySpread": 4096,
        "separationSpread": 256
    },
    "epochs": {
        "precisionOnly": 1024,
        "pretraining": 1024,
        "combinedTraining": 8192
    },
    "recallWeight": 1.0,
    "qTarget": 1.0,
    "optimiser": {
        "type": "adam",
        "learningRate": 0.001,
        "betaOne": 0.9,
        "betaTwo": 0.999,
        "epsilon": 10^-8
    }
}
```

Listing 7.4: Experimental parameters for training a generator to match an arbitrary constraint satisfaction function with no constraint inputs.

## G Bimodal constraint satisfaction function

A bimodal constraint satisfaction was defined for a solution dimensionality of 1 and a constraint dimensionality of 0, meaning that the function does not take into consideration any constraint parameters.

$$h(c, s) = \sigma(20s - 6) - \sigma(20s - 14) + \sigma(20s + 6) - \sigma(20s + 14) \tag{7.7}$$

where

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7.8}$$

## H Parameterised constraint satisfaction function

An adaptation of the previously introduced $h$ was defined within a Tensorflow computation graph, whose peak locations are controlled by each component of the constraint vector. The below Python code produces this function for any $n$.

```
import tensorflow as tf

constraint_dimension = 3

def parameterised_csf(solution_node, constraint_node):
    tiled_solution = tf.tile(solution_node, [1, constraint_dimension])
    summed_output = tf.reduce_mean(
        sigmoid_peak(tiled_solution, constraint_node), axis=1
    )
    return tf.reshape(
        summed_output,
        [tf.shape(summed_output)[0], 1]
    )

def sigmoid_peak(x, offset, width=5.0, thickness=0.05, height_scale=1.0):
    """Produces a single peak from two sigmoid functions."""
    after_offset = (x - offset) / thickness
    return height_scale * (
        tf.sigmoid(after_offset + width) - tf.sigmoid(after_offset - width)
    )
```

Listing 7.5: Python code to produce a constraint satisfaction function parameterised by a constraint vector within a Tensorflow computation graph.

# I Embedder verification parameters

The parameters used to train a generator to match an arbitrarily defined constraint satifaction function are reproduced below. Two neural networks, the weight embedder and bias embedder, produce the weights for the final layer of the generator.

```
{
    "batchSize": 64,
    "pretraining": {
        "epochs": 10,
        "stepsPerEpoch": 20000
    },
    "training": {
        "epochs": 150,
        "stepsPerEpoch": 20000
    },
    "generator": {
        "layers": [
            {
                "nodes": 32,
                "activation": "leaky-relu"
            },
            {
                "nodes": 32,
                "activation": "leaky-relu"
            },
            {
                "nodes": 1,
                "activation": "tanh"
            }
        ],
        "initialisation": "glorot"
    },
    "weightsEmbedder": {
        "layers": [
            {
                "nodes": 32,
                "activation": "leaky-relu"
            },
            {
                "nodes": 32,
```

```
                "activation": "leaky-relu"
            },
            {
                "nodes": 32,
                "activation": "leaky-relu"
            }
        ],
        "initialisation": "glorot"
    },
    "biasesEmbedder": {
        "layers": [
            {
                "nodes": 32,
                "activation": "leaky-relu"
            },
            {
                "nodes": 32,
                "activation": "leaky-relu"
            },
            {
                "nodes": 32,
                "activation": "leaky-relu"
            }
        ],
        "initialisation": "glorot"
    }
    "optimiser": {
        "type": "adam",
        "learningRate": 0.001,
        "betaOne": 0.9,
        "betaTwo": 0.999,
        "epsilon": 10^-8
    },
    "recallSubstitute": "identitySpread",
    "recallWeight": 0.7,
    "constraintDimension": 2,
    "embeddingDimension": 10
}
```

Listing 7.6: Experimental parameters for training a generator to match an arbitrary constraint satisfaction function parameterised by a constraint vector.

## J Holes environment

An engineering environment, referred to as the holes environment, was created for the purpose of testing the properties of the proposed architecture. It consists of a square plate with sides two units long whose origin is at its centre. Two holes are drilled into the plate, each described by the coordinates of their centre (which may be anywhere on the plate) and their radius (which may be in $[0, 1]$). One of the hole's parameters are fixed; this hole is taken to be the constraint. The other hole's parameters are varied as the parameters of the solution. As such, $m = n = 3$. Satisfaction of the constraint occurs when the holes do not meet.

$$h(c, s) = (c_1 - s_1)^2 + (c_2 - s_2)^2 > (c_3 + s_3)^2 \tag{7.9}$$

## K Training parameters for holes environment

The parameters used to train a generator to match an arbitrarily defined constraint satifaction function are reproduced below. Two neural networks, the weight embedder and bias embedder, produce the weights for the final layer of the generator.

```
{
    "discriminatorTrainingParameters": {
        "epochs": 20,
        "evaluationSampleSize": 256,
        "stepsPerEpoch": 20000,
        "batchSize": 32
    },
    "recallProxy": "identity",
    "epsilon": 1.0,
    "parametricGenerator": {
        "generatorArchitecture": {
            "internalLayers": [
                {
                    "activation": "leaky-relu",
                    "nodes": 4
                },
                {
                    "activation": "leaky-relu",
                    "nodes": 4
                }
            ],
            "internalActivation": "leaky-relu",
            "outputActivation": "tanh"
```

```
        },
        "discriminatorArchitecture": [
            {
                "activation": "leaky-relu",
                "nodes": 4
            }
        ],
        "generatorTrainingBatchSize": 64,
        "embeddingDimension": 2,
        "latentDimension": 3,
        "constraintSpace": {
            "upperBound": 1.0,
            "lowerBound": -1.0
        },
        "solutionDimension": 3,
        "embedderArchitecture": {
            "biases": {
                "activation": "leaky-relu",
                "internalLayers": [
                    {
                        "activation": "leaky-relu",
                        "nodes": 4
                    },
                    {
                        "activation": "leaky-relu",
                        "nodes": 4
                    }
                ]
            },
            "weights": {
                "activation": "leaky-relu",
                "internalLayers": [
                    {
                        "activation": "leaky-relu",
                        "nodes": 4
                    },
                    {
                        "activation": "leaky-relu",
                        "nodes": 4
                    }
```

```
                ]
            }
        },
        "latentSpace": {
            "upperBound": 1.0,
            "lowerBound": 0.0
        },
        "constraintDimension": 3,
        "solutionSpace": {
            "upperBound": 1.0,
            "lowerBound": -1.0
        }
    },
    "discriminatorValidationProportion": 0.2,
    "generatorTrainingParameters": {
        "epochs": 20,
        "evaluationSampleSize": 256,
        "stepsPerEpoch": 20000,
        "batchSize": 64
    },
    "recallWeights": [0.5, 1.0, 2.0, 3.0],
    "evaluationParameters": {
        "generatedSolutionsPerConstraint": 128,
        "trueSolutionsPerConstraint": 128,
        "constraintSamples": {
            "quantity": 32,
            "samplingMethod": "uniform"
        },
        "monteCarlo": {
            "burnIn": 1024,
            "sampleGap": 64
        },
        "nTree": {
            "bucketSize": 128,
            "population": 8192
        }
    },
    "precisionProxy": "precision",
    "dataset": "production/datasets/holes/256",
    "pretrainingLoss": "identity",
```

```
    "generatorPretrainingParameters": {
        "epochs": 20,
        "evaluationSampleSize": 256,
        "stepsPerEpoch": 20000,
        "batchSize": 64
    }
}
```

Listing 7.7: Experimental parameters for training a generator on the holes environment.

## L Branin function

The Branin function is a function designed for testing optimisation algorithms [38]. It is defined as

$$\beta(x, y) = a(y - bx^2 + cx - r)^2 + s(1 - t)\cos(x) + s \tag{7.10}$$

where

$$a = 1,\ b = \frac{5.1}{4\pi^2},\ c = \frac{5}{\pi},\ r = 6,\ s = 10,\ t = \frac{1}{8\pi} \tag{7.11}$$

For the purposes of this project, a rescaled version is used, such that:

$$\beta'(s) = \frac{1}{250}\beta(15s_1 - 5,\ 15s_2) \tag{7.12}$$

Finally, the Branin environment imposes a constraint upon the upper and lower bounds of $\beta'$, limiting them to $c_1 \cdot c_2$ and $c_2$ respectively. As such, a constraint in the Branin environment begins to approximate an equality constraint as $c_1 \to 1$.

## M Training parameters for bounded Branin function

An extract from the configuration file for the instance of the proposed architecture trained on the Branin environment is included below.

```
{
    "epsilon": 1.0,
    "recallProxyMetadata": {
    },
    "discriminatorTrainingParameters": {
        "evaluationSampleSize": 256,
        "batchSize": 32,
        "epochs": 64,
        "stepsPerEpoch": 20000
    },
```

```
    "precisionProxy": "precision",
    "recallWeight": 3,
    "generatorTrainingParameters": {
        "evaluationSampleSize": 256,
        "batchSize": 64,
        "epochs": 128,
        "stepsPerEpoch": 20000
    },
    "recallProxy": "identity",
    "parametricGenerator": {
        "generatorTrainingBatchSize": 64,
        "constraintDimension": 2,
        "latentDimension": 2,
        "generatorArchitecture": {
            "outputActivation": "sigmoid",
            "internalActivation": "leaky-relu",
            "internalLayers": [
                {
                    "nodes": 32,
                    "activation": "leaky-relu"
                },
                {
                    "nodes": 32,
                    "activation": "leaky-relu"
                }
            ]
        },
        "solutionDimension": 2,
        "embeddingDimension": 8,
        "embedderArchitecture": {
            "biases": {
                "activation": "leaky-relu",
                "internalLayers": [
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    },
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
```

```
                    }
                ]
            },
            "weights": {
                "activation": "leaky-relu",
                "internalLayers": [
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    },
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    }
                ]
            }
        },
        "repeatConstraints": false,
        "latentSpace": {
            "lowerBound": 0.0,
            "upperBound": 1.0
        },
        "constraintSpace": {
            "lowerBound": 0.0,
            "upperBound": 1.0
        },
        "discriminatorArchitecture": [
            {
                "nodes": 64,
                "activation": "leaky-relu"
            },
            {
                "nodes": 64,
                "activation": "leaky-relu"
            }
        ],
        "solutionSpace": {
            "lowerBound": 0.0,
            "upperBound": 1.0
        }
```

```
    },
    "pretrainingLossMetadata": {},
    "pretrainingLoss": "identity",
    "discriminatorValidationProportion": 0.2,
    "evaluationParameters": {
        "generatedSolutionsPerConstraint": 128,
        "constraintSamples": {
            "quantity": 16,
            "samplingMethod": "uniform"
        },
        "trueSolutionsPerConstraint": 128,
        "monteCarlo": {
            "sampleGap": 64,
            "burnIn": 1024
        },
        "nTree": {
            "bucketSize": 128,
            "population": 8192
        }
    },
    "precisionProxyMetadata": {},
    "generatorPretrainingParameters": {
        "evaluationSampleSize": 256,
        "batchSize": 64,
        "epochs": 16,
        "stepsPerEpoch": 20000
    },
    "dataset": "production/datasets/branin/1024"
}
```

Listing 7.8: Experimental parameters for training a generator on the bounded Branin function environment.

## N Exemplar results JSON

An example of a typical results dump produced by the common experiment interface used for all experiments in §6.2.

```
{
    "generatorPretraining": {
        "startTime": 1555341084.538,
```

```
        "before": {
            "loss": 0.096
        },
        "endTime": 1555341086.955,
        "duration": 2.417,
        "after": {
            "loss": 0.000
        }
    },
    "evaluation": {
        "constraintSamples": [
            {
                "solutions": [
                    {
                        "type": "generated",
                        "latent": [0.692, 0.607],
                        "relativeDensity": 7.625,
                        "solution": [0.649, 0.572],
                        "satisfactionProbability": 0.873
                    },
                    ...,
                    {
                        "type": "generated",
                        "latent": [0.915, 0.810],
                        "relativeDensity": 10.250,
                        "solution": [0.787, 0.549],
                        "satisfactionProbability": 0.980
                    },
                    {
                        "type": "true",
                        "relativeDensity": 0.688,
                        "solution": [0.137, 0.279],
                        "satisfactionProbability": 0.911
                    },
                    ...,
                    {
                        "type": "true",
                        "relativeDensity": 12.500,
                        "solution": [0.240, 0.023],
                        "satisfactionProbability": 1.000
```

```
                    }
                ],
                "constraint": [0.519, 0.260],
                "summary": {
                    "true": {
                        "satisfactionProbability": {
                            "median": 0.956,
                            "mean": 0.956,
                            "minimum": 0.911,
                            "maximum": 1.000
                        },
                        "relativeDensity": {
                            "median": 6.594,
                            "mean": 6.594,
                            "minimum": 0.688,
                            "maximum": 12.500
                        }
                    },
                    "generated": {
                        "satisfactionProbability": {
                            "median": 0.926,
                            "mean": 0.926,
                            "minimum": 0.873,
                            "maximum": 0.980
                        },
                        "relativeDensity": {
                            "median": 8.938,
                            "mean": 8.938,
                            "minimum": 7.625,
                            "maximum": 10.250
                        }
                    },
                    "all": {
                        "satisfactionProbability": {
                            "median": 0.945,
                            "mean": 0.941,
                            "minimum": 0.873,
                            "maximum": 1.000
                        },
                        "relativeDensity": {
```

```
                            "median": 8.938,
                            "mean": 7.766,
                            "minimum": 0.688,
                            "maximum": 12.500
                        }
                    }
                }
            },
            ...,
            {
                "solutions": [
                    {
                        "type": "generated",
                        "latent": [0.704, 0.663],
                        "relativeDensity": 8.125,
                        "solution": [0.669, 0.525],
                        "satisfactionProbability": 0.994
                    },
                    ...,
                    {
                        "type": "generated",
                        "latent": [0.213, 0.650],
                        "relativeDensity": 0.395,
                        "solution": [0.362, 0.734],
                        "satisfactionProbability": 1.000
                    },
                    {
                        "type": "true",
                        "relativeDensity": 0.234,
                        "solution": [0.621, 0.339],
                        "satisfactionProbability": 0.922
                    },
                    ...,
                    {
                        "type": "true",
                        "relativeDensity": 3.719,
                        "solution": [0.429, 0.666],
                        "satisfactionProbability": 1.000
                    }
                ],
```

```
                "constraint": [0.272, 0.279],
                "summary": {
                    "true": {
                        "satisfactionProbability": {
                            "median": 0.961,
                            "mean": 0.961,
                            "minimum": 0.922,
                            "maximum": 1.000
                        },
                        "relativeDensity": {
                            "median": 1.977,
                            "mean": 1.977,
                            "minimum": 0.234,
                            "maximum": 3.719
                        }
                    },
                    "generated": {
                        "satisfactionProbability": {
                            "median": 0.997,
                            "mean": 0.997,
                            "minimum": 0.994,
                            "maximum": 1.000
                        },
                        "relativeDensity": {
                            "median": 4.260,
                            "mean": 4.260,
                            "minimum": 0.395,
                            "maximum": 8.125
                        }
                    },
                    "all": {
                        "satisfactionProbability": {
                            "median": 0.997,
                            "mean": 0.979,
                            "minimum": 0.922,
                            "maximum": 1.000
                        },
                        "relativeDensity": {
                            "median": 2.057,
                            "mean": 3.118,
```

```
                            "minimum": 0.234,
                            "maximum": 8.125
                        }
                    }
                }
            }
        ],
        "summary": {
            "true": {
                "satisfactionProbability": {
                    "median": 0.961,
                    "mean": 0.958,
                    "minimum": 0.911,
                    "maximum": 1.000
                },
                "relativeDensity": {
                    "median": 2.203,
                    "mean": 4.285,
                    "minimum": 0.234,
                    "maximum": 12.500
                }
            },
            "generated": {
                "satisfactionProbability": {
                    "median": 0.987,
                    "mean": 0.962,
                    "minimum": 0.873,
                    "maximum": 1.000
                },
                "relativeDensity": {
                    "median": 7.875,
                    "mean": 6.599,
                    "minimum": 0.395,
                    "maximum": 10.250
                }
            },
            "all": {
                "satisfactionProbability": {
                    "median": 0.987,
                    "mean": 0.960,
```

```
                    "minimum": 0.873,
                    "maximum": 1.000
                },
                "relativeDensity": {
                    "median": 5.672,
                    "mean": 5.442,
                    "minimum": 0.234,
                    "maximum": 12.500
                }
            }
        }
    },
    "generatorTraining": {
        "startTime": 1555341087.703,
        "before": {
            "recallProxy": 0.000,
            "loss": -0.176,
            "precisionProxy": -0.176
        },
        "endTime": 1555341111.789,
        "duration": 24.087,
        "after": {
            "recallProxy": 0.040,
            "loss": -0.464,
            "precisionProxy": -0.585
        }
    },
    "discriminatorTraining": {
        "startTime": 1555341069.424,
        "before": {
            "loss": 1.398,
            "accuracy": 0.469
        },
        "endTime": 1555341083.675,
        "duration": 14.251,
        "after": {
            "validationAccuracy": 0.919,
            "validationLoss": 0.347,
            "trainingAccuracy": 1.000,
            "trainingLoss": 0.020
```

```
        }
    },
    "parameters": {
        "dataset": "production/datasets/example/256",
        "recallWeight": 3.000,
        "precisionProxy": "precision",
        "evaluationParameters": {
            "generatedSolutionsPerConstraint": 128,
            "monteCarlo": {
                "sampleGap": 64,
                "burnIn": 1024
            },
            "constraintSamples": {
                "quantity": 128,
                "samplingMethod": "uniform"
            },
            "nTree": {
                "bucketSize": 128,
                "population": 8192
            },
            "trueSolutionsPerConstraint": 128
        },
        "parametricGenerator": {
            "generatorArchitecture": {
                "internalActivation": "leaky-relu",
                "outputActivation": "sigmoid",
                "internalLayers": [
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    },
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    }
                ]
            },
            "solutionDimension": 2,
            "embeddingDimension": 8,
            "constraintSpace": {
```

```
            "lowerBound": 0.000,
            "upperBound": 1.000
        },
        "discriminatorArchitecture": [
            {
                "nodes": 64,
                "activation": "leaky-relu"
            },
            {
                "nodes": 64,
                "activation": "leaky-relu"
            }
        ],
        "solutionSpace": {
            "lowerBound": 0.000,
            "upperBound": 1.000
        },
        "latentDimension": 2,
        "latentSpace": {
            "lowerBound": 0.000,
            "upperBound": 1.000
        },
        "generatorTrainingBatchSize": 64,
        "repeatConstraints": false,
        "constraintDimension": 2,
        "embedderArchitecture": {
            "biases": {
                "activation": "leaky-relu",
                "internalLayers": [
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    },
                    {
                        "nodes": 32,
                        "activation": "leaky-relu"
                    }
                ]
            },
            "weights": {
```

```
                    "activation": "leaky-relu",
                    "internalLayers": [
                        {
                            "nodes": 32,
                            "activation": "leaky-relu"
                        },
                        {
                            "nodes": 32,
                            "activation": "leaky-relu"
                        }
                    ]
                }
            }
        },
        "generatorPretrainingParameters": {
            "stepsPerEpoch": 20000,
            "epochs": 16,
            "batchSize": 64,
            "evaluationSampleSize": 256
        },
        "pretrainingLoss": "identity",
        "generatorTrainingParameters": {
            "stepsPerEpoch": 20000,
            "epochs": 128,
            "batchSize": 64,
            "evaluationSampleSize": 256
        },
        "recallProxyMetadata": {},
        "recallSubstitute": "identity",
        "precisionProxyMetadata": {},
        "discriminatorValidationProportion": 0.200,
        "pretrainingLossMetadata": {},
        "epsilon": 1.000,
        "discriminatorTrainingParameters": {
            "stepsPerEpoch": 20000,
            "epochs": 64,
            "batchSize": 32,
            "evaluationSampleSize": 256
        }
    }
```

```
364 }
```

Listing 7.9: The output of a typical experiment, automatically logged when the experiment concludes.