Probabilistic Regression using Conditional Generative Adversarial Networks

Joel Oskarsson

Supervisor: Fredrik Lindsten
Examiner: Jose M. Peña
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Joel Oskarsson
Abstract

Regression is a central problem in statistics and machine learning with applications everywhere in science and technology. In probabilistic regression the relationship between a set of features and a real-valued target variable is modelled as a conditional probability distribution. There are cases where this distribution is very complex and not properly captured by simple approximations, such as assuming a normal distribution. This thesis investigates how conditional Generative Adversarial Networks (GANs) can be used to properly capture more complex conditional distributions. GANs have seen great success in generating complex high-dimensional data, but less work has been done on their use for regression problems. This thesis presents experiments to better understand how conditional GANs can be used in probabilistic regression. Different versions of GANs are extended to the conditional case and evaluated on synthetic and real datasets. It is shown that conditional GANs can learn to estimate a wide range of different distributions and be competitive with existing probabilistic regression models.
Acknowledgments

I would like to thank my supervisor Fredrik Lindsten for great support throughout my thesis work and for coming up with the original idea for the project. Thank you also to my opponent Daniel Björnander and my examiner Jose M. Peña for valuable comments on drafts of this thesis. Everyone at the Division of Statistics and Machine Learning have been very helpful and welcoming, which is greatly appreciated. A special thank you to Anders Eklund for assistance with the computer used for my experiments. Lastly I would like to thank friends and family for being very supportive throughout my thesis project and all of my time at Linköping University.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Aim</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Research questions</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Theory</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Neural Networks</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Training</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Activation Functions</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Optimizers</td>
<td>7</td>
</tr>
<tr>
<td>2.1.4 Regularization</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Implicit Generative Models</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Generative Adversarial Networks</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 Unconditional GANs</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Conditional GANs</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3 Theoretical Optima</td>
<td>12</td>
</tr>
<tr>
<td>2.3.4 (\phi)-divergence Minimization</td>
<td>13</td>
</tr>
<tr>
<td>2.3.5 Integral Probability Metrics</td>
<td>14</td>
</tr>
<tr>
<td>2.3.6 Generator Training Objective</td>
<td>15</td>
</tr>
<tr>
<td>2.3.7 Density Ratio Estimation</td>
<td>16</td>
</tr>
<tr>
<td>2.3.8 Least Squares GAN</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Generative Moment Matching Networks</td>
<td>17</td>
</tr>
<tr>
<td>2.4.1 GMMNs and MMD</td>
<td>17</td>
</tr>
<tr>
<td>2.4.2 Deep Kernel Learning</td>
<td>19</td>
</tr>
<tr>
<td>2.4.3 Conditional GMMNs</td>
<td>19</td>
</tr>
<tr>
<td>2.4.4 Joint GMMNs</td>
<td>20</td>
</tr>
<tr>
<td>2.5 Probabilistic Regression</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1 Basic Regression Models</td>
<td>21</td>
</tr>
<tr>
<td>2.5.2 Gaussian Processes</td>
<td>22</td>
</tr>
<tr>
<td>2.5.3 Mixtures of Regression Models</td>
<td>24</td>
</tr>
<tr>
<td>2.5.4 Mixture Density Networks</td>
<td>24</td>
</tr>
<tr>
<td>2.5.5 Energy-based Regression</td>
<td>25</td>
</tr>
<tr>
<td>2.5.6 CGAN as a Regression Model</td>
<td>27</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Car at an intersection example ............................................. 2
1.2 An n-layer neural network .................................................. 5
1.3 GAN and CGAN .................................................................. 6
1.4 Samples from an example model with heteroskedastic noise .......... 10
1.5 Example of a multimodal distribution ..................................... 22
1.6 Conceptual view of how a neural network can be used for probabilistic regression .................................................. 22
1.7 Conceptual view of a Mixture Density Network ....................... 24
1.8 The role of the neural network in the DCTD model ................. 25
1.9 Approximating a probability density function from samples using KDE .................................................. 28
2.1 An n-layer neural network .................................................. 5
2.2 Activation functions ................................................................ 6
2.3 GAN and CGAN .................................................................. 10
2.4 Samples from an example model with heteroskedastic noise .......... 22
2.5 Example of a multimodal distribution ..................................... 22
2.6 Conceptual view of how a neural network can be used for probabilistic regression .................................................. 22
2.7 Conceptual view of a Mixture Density Network ....................... 24
2.8 The role of the neural network in the DCTD model ................. 25
2.9 Approximating a probability density function from samples using KDE .................................................. 28
3.1 Double-input neural network architectures used in CGANs .......... 31
3.2 The noise-injection generator network architecture ..................... 32
3.3 Generated test sets for each one-dimensional synthetic dataset .......... 39
3.4 Generated test set for the swirls synthetic dataset ....................... 40
3.5 Samples trajectories from trajectories datasets ............................ 41
3.6 Examples of Weibull distributions with different shape parameter k .................................................. 42
3.7 Test split of microwave dataset ............................................. 43
4.1 Scatter plots for the trajectories dataset with $d_y = 4$ ............... 55
4.2 Scatter plots for the trajectories dataset with $d_y = 10$ ............... 55
4.3 Trajectories conditioned on different maximum turning parameters $t_{max}$ .................................................. 56
4.4 Examples of conditional probability density functions for the wmix dataset .................................................. 59
4.5 Samples from models trained on the microwave dataset ............. 61
A.1 Results for experiment on noise dimensionality ....................... 82
A.2 Resulting scatter plots for CGANs trained on the bimodal dataset with different activation functions ...................... 83
A.3 Resulting scatter plots for CGANs trained on the complex dataset with different activation functions ...................... 84
D.1 Samples from models trained on the const_noise dataset ............. 90
D.2 Samples from models trained on the laplace dataset ................... 91
D.3 Samples from models trained on the exponential dataset .......... 92
D.4 Samples from models trained on the butterfly dataset ................ 93
D.5 Samples from models trained on the heteroskedastic dataset ........ 94
D.6 Samples from models trained on the bimodal dataset .................. 95
D.7 Samples from models trained on the complex dataset ................ 96
E.1 Sampled trajectories for the trajectories dataset with $d_y = 4$ .......... 97
E.2 Trajectories with $d_y = 4$ conditioned on different values for the maximum turning parameter $t_{max}$ .................................................................................................................. 98
E.3 Scatter plots for the trajectories dataset with $d_y = 4$.

E.4 Sampled trajectories for the trajectories dataset with $d_y = 10$.

E.5 Trajectories with $d_y = 10$ conditioned on different values for the maximum turning parameter $t_{\text{max}}$.

E.6 Scatter plots for the trajectories dataset with $d_y = 10$.

E.7 Sampled trajectories for the trajectories dataset with $d_y = 20$.

E.8 Trajectories with $d_y = 20$ conditioned on different values for the maximum turning parameter $t_{\text{max}}$.

E.9 Scatter plots for the trajectories dataset with $d_y = 20$.

F.1 Examples of conditional probability density functions for models trained on the mix dataset with $d_x = 3$.

F.2 Examples of conditional probability density functions for models trained on the mix dataset with $d_x = 6$.

F.3 Examples of conditional probability density functions for models trained on the mix dataset with $d_x = 9$.

F.4 Examples of conditional probability density functions for models trained on the mix dataset with $d_x = 15$. 
List of Tables

3.1 Sizes of double-input CGAN networks

3.2 Sizes of noise-injection CGAN networks

3.3 f-divergences used to define CGAN models

3.4 Kernels used for the GP model

3.5 Sizes of neural networks

3.6 Parameters of the random process for generating $y$ in trajectories datasets

4.1 Results from experiments on const_noise and laplace synthetic datasets

4.2 Results from experiments on exponential and butterfly synthetic datasets

4.3 Results from experiments on heteroskedastic synthetic dataset

4.4 Results from experiments on bimodal synthetic dataset

4.5 Results from experiments on complex and swirls synthetic datasets

4.6 Training times for different models with different network sizes

4.7 Results for the network architecture experiment

4.8 Results from experiments on trajectories datasets

4.9 Results from experiments on wmix datasets with $d_x = 3$ and $d_x = 6$

4.10 Results from experiments on wmix datasets with $d_x = 9$ and $d_x = 15$

4.11 Results from experiment on the microwave dataset

4.12 Results from experiments on power and housing datasets

A.1 Test log-likelihoods for CGANs using different distributions for the noise fed to the generator
1 Introduction

An increasing number of computer systems make use of machine learning. The behaviour of such systems is not completely manually designed and tested by a programmer. Instead machine learning based systems have learned parts of their behaviour from large sets of data. Even safety-critical systems are making use of machine learning models, including neural networks. The behaviour of such a network is encoded in thousands or even millions of numerical weights. Using these weights, a network will produce an output for any given input. Most relationships in the real world are however not well captured by such deterministic one-to-one mappings. Data collected about real systems is noisy and full of uncertainty. When building machine learning models it is then important to properly take this uncertainty into account and not make oversimplifying assumptions.

There are two separate kinds of uncertainty that are mainly relevant when working with machine learning models \([30]\). The first is model uncertainty, also called epistemic uncertainty. For a machine learning model trained on some finite set of data, the predictions of the model will depend on which data samples that were used in training. This results in model uncertainty in the predictions, since another realization of the training data might result in a slightly different model and slightly different predictions. For models that are fully defined by a set of parameters, the model uncertainty is the uncertainty in the values of those parameters. Model uncertainty can be reduced by using more training data \([30]\).

The second type of uncertainty is data uncertainty, also called aleatoric uncertainty or risk. Real data can be thought of as made up of a signal component, containing the actual useful value, and a noise component. The noise component is the source of data uncertainty. Such noise can either be inherit to the real data generating process or introduced through measurement errors. Data uncertainty does not vanish when using more data samples for training \([30]\). Predictions made by a machine learning model will therefore always feature data uncertainty. This thesis will be exclusively concerned with data uncertainty, leaving model uncertainty as a separate problem.

As an example of the importance of properly modelling the uncertainty in the data, consider predicting the movement of a car in traffic. When the car arrives at an intersection, as illustrated in figure \[1.1a\], it should be predicted where it is going to turn next. A large set of historical data on similar situations might indicate that the likely directions for the car to turn follow the distribution in figure \[1.1b\]. Think of direction here as the heading angle of the car at the end of the turn. A too simplistic machine learning model trained on this data
would predict the mean of this distribution. For this example, it is clear that the mean is not a very useful prediction. It is about equally likely that the car will turn right or left, but the mean prediction of going straight forward is implausible. Modelling the uncertainty in the data more carefully would be necessary to get more useful predictions.

(a) A car at an intersection

Figure 1.1: Car at an intersection example

Generative adversarial networks (GANs) are a type of deep machine learning model that has gained a lot of traction for its ability to generate high fidelity synthetic data, such as fake images and audio. A GAN consists of two neural networks. A generator network takes random noise as input and produces an output. The other network, called discriminator, takes that output and compares it to training samples, trying to determine if it was generated by its fellow network or not. This defines a form of competition where the generator network tries to fool the discriminator and the discriminator tries to learn not to get fooled. In this way a GAN learns to approximate the underlying data distribution of the training samples. Conditional GAN (CGAN) is an extension that approximates a conditional distribution.

Normal GANs take noise as input to generate samples. CGANs take an additional non-random value as input. This has the effect that the output of a CGAN can be varied by varying the chosen value of $x$.

When training a GAN the objective is to minimize the difference between the distribution of generated samples and the distribution of real data. This difference can be measured in many different ways, each resulting in a separate formulation of the GAN training objective. This thesis focuses on the GAN variants known as $f$-GANs and Generative Moment Matching Networks (GMMNs).

Regression is the study of modelling relationships between two variables $x$ and $y$, where $y$ is numerical. Such models are trained using observed pairs of both variables. The trained models can then be used to predict values of $y$ when a value for $x$ is observed. An example of this is the "least squares" line-fitting method, used for drawing the straight line that best matches a collection of known samples. Probabilistic regression is a special case of regression where the focus does not lie on individual values of $y$. Instead, given a value for $x$, the problem is finding the probability distribution $p(y|x)$ of possible values for $y$. This conditional distribution changes with $x$ and a probabilistic regression model has to take this into account.

Since CGANs approximate conditional probability distributions they can be used as probabilistic regression models. Traditionally, GANs and CGANs have been used to mimic human creations, where $y$ is considered to be some complex data-type such as images or music. The focus often lies on the quality of this produced output, as determined by human observers. The focus of this thesis is instead on generating less complex data, on the level of single scalars. By doing this, CGANs are investigated as distribution estimators rather than sample gener-
1.1 Motivation

As use of machine learning systems is becoming more common in many real-world scenarios, new quality considerations emerge. When these systems are used in safety-critical environments such as self-driving cars or aircraft, it is not sufficient to simply come up with single predictions. It would be desirable that the machine also had some form of awareness of how certain the prediction is. Being able to use a set of samples to reason about uncertainty would then be very useful.

Existing probabilistic regression models typically only estimate data uncertainty as distributions from a known family. An example of this is the Gaussian Process (GP), that is typically used to model a Gaussian distribution matching the data [47, p. 16-19]. CGANs are not as restricted in the kinds of distributions that can be estimated. This would make a regression model based on CGAN useful in cases where little is known about the distribution of data. CGANs thus has the potential to work as black-box regression models for many different use cases. Note that many models, including GPs, can model very complex relationships between \(x\) and \(y\). The advantage of CGANs is not to model these relationships more accurately, but to better model the data uncertainty in predictions. In other words, CGANs provide a more flexible model for the conditional distribution \(p(y|x)\).

CGANs gain their ability to adapt to complex data from using neural networks. In their basic form, neural networks are deterministic function approximators. They can be directly applied to regression problems by letting \(x\) be the network input and minimizing the mean squared error between the true target value and the network output. In the probabilistic setting, this corresponds to assuming additive Gaussian noise on the network output. This is a strong assumption that will not hold in situations where the noise is more complex. CGANs on the other hand does not explicitly require such assumptions and can model a much more general class of distributions.

1.2 Aim

The aim of this thesis is to define a mathematical framework for using CGANs for regression problems and investigate the usefulness of the model. The output is a comparison of CGAN and other commonly used probabilistic regression models. Conclusions are drawn about suitable network architectures, hyperparameter values and training objectives.

1.3 Research questions

Based on the motivation and aim this thesis will answer the following research questions:

1. How well can CGANs approximate simple and complex data distributions?
2. How does the use of different training objectives and neural network architectures impact the training process and the capability of CGANs to approximate distributions?
3. How does CGANs compare to alternative probabilistic regression models on real world datasets?
2 Theory

The underlying theory of this thesis mainly comes from two subareas of machine learning. The first is theory related to CGANs, the model being considered. As CGANs are entirely based upon neural networks, some key aspects related to these will be covered. The second subarea central to this thesis is the greater study of regression problems. Regression is the problem the model will be applied to. Exploring regression is important both for understanding CGAN as a regression model, but also for contrasting it with other approaches proposed in the literature. Additional theory will also be presented related to the practical evaluation of probabilistic regression models.

A note on notation

Throughout this thesis bold letters will be used to denote vectors or vector-valued functions. Normal lower-case letters denote scalars or scalar-valued functions. Capital letters are used for matrices or higher order tensors. The log-function refers to logarithm with base $e$. All other notation is defined as it is introduced.

2.1 Neural Networks

Neural networks are machine learning models that combine linear transformations with non-linear functions to allow for approximating a vast set of mathematical functions. A neural network consists of multiple layers of simple computational units [19, p. 168-171]. These layers are divided into input and output layers as well as zero or more so-called hidden layers in between. The units of each layer are connected to the units of neighbouring layers via weighted connections, as is illustrated in figure 2.1. This allows them to take input from previous layers and pass it further along the network.

The operation of a neural network can be seen as a sequential computation from the input layer, through the hidden layers and ending in the output layer [19, p.168-171]. The result of this sequential computation at layer $i$ is a vector $u^i \in \mathbb{R}^{m_i}$. Here $m_i$ is the number of computational units in layer $i$ and thereby also the dimension of the vector. The first layer $u^0$ corresponds to the input vector $x$ and the last layer $u^n$ to the output vector $\hat{y}$. The network weights are stored as matrices $W_i \in \mathbb{R}^{m_{i+1} \times m_i}$ where $i$ is the layer the connections start from. Associated with each layer is also a bias vector $b_i \in \mathbb{R}^{m_i}$. The bias is a constant offset that
2.1. Neural Networks

Figure 2.1: An $n$-layer neural network. Biases are left out to avoid clutter.

... does not depend on values of previous layers. Each $u^i$ can then be computed as:

$$u^i = g(W_{i-1}u^{i-1} + b_i),$$

(2.1)

where $g: \mathbb{R}^{m_i} \rightarrow \mathbb{R}^{m_i}$ is the so-called activation function. Activation functions are typically non-linear functions applied entry-wise over a vector [14], p. 191-197]. This value for $u^i$ is then passed on to the following layer and used in the computation of $u^{i+1}$. This continues until the output layer is reached and the network outputs $u^n = \hat{y}$. The process is referred to as forward propagation.

2.1.1 Training

The trainable parameters $\theta$ in a neural network is the set of all weight matrices and all bias vectors:

$$\theta \overset{\text{def}}{=} \{W_0, W_1, \ldots, W_{n-1}, b_1, b_2, \ldots, b_n\}.$$  

(2.2)

For mathematical rigor it is sometimes useful to see $\theta$ not as a set, but as a long vector containing all the entries of all weight matrices and bias vectors. By choosing suitable values for the parameters the network can be made to output a desired value for a chosen input. The values of these parameters are most commonly determined by minimizing some loss function [19], p. 177-191]. A loss function $L(\theta)$ measures how poor a network with parameters $\theta$ is performing for a set of training data. Many different kinds of loss functions can be designed for different problems. The minimization of $L$ with respect to $\theta$ defines a generally non-convex optimization problem. Finding the global optimum of such an optimization problem is typically not possible. When training a neural network it is luckily often sufficient to find a good local optimum or simply to reduce the loss function to a low value. A common optimization algorithm to use for this is gradient descent [8], p. 236-249]. In gradient descent optimization all the model parameters are trained according to:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta),$$

(2.3)

where $\alpha$ is an adjustable learning rate hyperparameter. Gradient descent can be performed in a computationally efficient manner for neural networks by using the backpropagation algorithm to compute the gradients [19], p. 204-221].

From a statistical perspective the optimization problem should be to minimize the expected loss, $L(\theta) = \mathbb{E}[l(x, y, \theta)]$ [19], p. 151-153]. Here $l$ measures the error for a single $(x, y)$ pair. The expectation is over the data distribution $p_d(x, y)$. All data that the network works with...
2.1. Neural Networks

is assumed to come from this underlying distribution. The distribution is unknown, but the expected loss can be estimated using a training dataset \( \{(x^{(i)}, y^{(i)})\}_{i=1}^{N_{\text{train}}} \) as:

\[
\mathcal{L}(\theta) \approx \frac{1}{N} \sum_{i=1}^{N_{\text{train}}} l(x^{(i)}, y^{(i)}, \theta).
\] (2.4)

Applying gradient descent to this estimated loss results in each step using an estimate of the true gradient. Minimizing \( \theta \) with respect to the entire sum in eq. 2.4 does however become impractical for large datasets. Optimization can instead be carried out using a small batch of training pairs for each step. This is referred to as stochastic gradient descent.

### 2.1.2 Activation Functions

The use of an activation function \( g \) is important for allowing neural networks to approximate complex functions [19, p. 171-177]. In fact, if no activation function is used (or equivalently, using \( g(x) = x \)) the operation of an entire neural network is equivalent to just a single matrix multiplication [9, p. 229]. As noted before, \( g(x) \) is typically non-linear and applied entry-wise over the vector \( x \). In this section all activation functions will be described only by their operation on single entries of \( x \). For a visual reference of the discussed activation functions and their gradients, see figure 2.2.

![Figure 2.2: (a) Activation functions and (b) their derivatives](image)

Perhaps the most commonly used activation function is the Rectified Linear Unit (ReLU) [19, p. 193-194], defined by:

\[
\text{ReLU}(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  0 & \text{if } x \leq 0
\end{cases}
\] (2.5)

The function is not differentiable at \( x = 0 \), which might seem problematic when using gradient descent for training. In practice this turns out to not be an issue, since a suitable derivative can still be defined in the implementation [19, p. 192]. In practical numerical computation, the chance of \( x \) being exactly 0 is also very small. ReLU has consistent, large gradients for \( x > 0 \), but zero gradients for \( x < 0 \). The zeroed gradients can cause problems in training earlier layers in the network.

To combat the zero gradients problem in ReLU, Maas et al. [35] have proposed the Leaky ReLU, defined as:

\[
\text{LeakyReLU}(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha x & \text{if } x \leq 0
\end{cases}
\] (2.6)

In the original formulation \( \alpha = 0.01 \), but this parameter can easily be tuned to any desired value. Note that \( \alpha \) itself is the value of the gradient for \( x < 0 \).
Another noteworthy activation function is the Exponential Linear Unit (ELU) \cite{13}.

\[
\text{ELU}(x) = \begin{cases} 
x & \text{if } x > 0 \\
\beta(\exp(x) - 1) & \text{if } x \leq 0
\end{cases}
\]  
\tag{2.7}
\]

where $\beta \geq 0$ is a tuning parameter. A typical value would be $\beta = 1$. ELU pushes the mean of its output closer to zero during training. Clevert et al. \cite{13} have shown that this results in more useful gradients and faster training of the network.

### 2.1.3 Optimizers

When training neural networks using gradient descent the choice of learning rate, $\alpha$ in eq. \ref{eq:2.3}, is key to learning a good set of parameters \cite[p. 306-310]{19}. Tuning this hyperparameter manually can be challenging and a number of more sophisticated approaches have been proposed. These so called optimizers use an adaptive learning rate that changes throughout training and can also use an individual learning rate for each network parameter. They often also make use of past gradients in the computation of the current parameter update.

Many optimizers include a momentum mechanism \cite[p. 296-300]{19} to speed up learning. Momentum is inspired by the corresponding concept in physics. The model parameters can be seen as a point mass and the training process as this mass moving around with some velocity \cite[p. 296]{19}. The actual momentum mechanism uses an exponentially decaying sum of past gradients as velocity $v$. The velocity is then used to update the model parameters:

\[
v \leftarrow \gamma v - \alpha \nabla \theta \mathcal{L}(\theta)
\]
\[
\theta \leftarrow \theta + v.
\]
\tag{2.8}
\tag{2.9}

Here $\gamma \in [0,1)$ is an additional hyperparameter controlling how fast the influence of earlier gradients decays. A fixed learning rate $\alpha$ for the current gradient is still present in this basic formulation.

The optimizer AdaGrad \cite[14, p. 307]{19} lets the learning rate decay based on the size of earlier gradients. The learning rate of each parameter at training step $S$ is divided by $\sqrt{\eta_1^2 + \eta_2^2 + \cdots + \eta_{S-1}^2}$, with $\eta_i$ being the gradient for the parameter at training step $i$. This makes the learning rate decay over time, but with different rate for each parameter. The learning rate in AdaGrad always decreases during training. While this in theory helps with convergence, it has proven to be problematic in practical neural network training. The learning rate might be too small before the training arrives in a good local optimum. A modified version is the RMSProp \cite[p. 307-308]{60} optimizer, that instead of a sum involving all previous gradients uses an exponentially decaying average. That way the scaling of the learning rate depends more on the most recent gradients. Neither AdaGrad nor RMSProp include momentum in their original formulations. Another optimizer often used in practice is Adam \cite[19, p. 308-309]{31}. Adam combines the adaptive learning rate from RMSProp with the momentum mechanism.

There is no real consensus on which optimizer performs the best \cite[p. 309-310]{19}. The choice depends highly on the problem at hand and the exact network used. The use of adaptive learning rates rather than basic gradient descent does however increase performance in general.

### 2.1.4 Regularization

The key challenge in all of machine learning, and in particular when training neural networks, is that of generalizing to unseen data \cite[p. 110-116]{19}. A model should not just achieve a low loss on the training data, but also discover some general structure in the way the data is distributed. Consider the following loss function, measuring the squared distance between network predictions $\hat{y}$ and training data $y$:  

\[
\mathcal{L}(\theta) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \| \hat{y}(x^{(i)}; \theta) - y^{(i)} \|_2^2.
\]
\tag{2.10}


This loss could easily be driven to zero for the entire training dataset by the model simply memorizing all the training data, if the model has capacity enough to do so. However, a model that just memorizes the training data is not very useful and has not really discovered any of the underlying structure in the data. This situation is known as the model overfitting to the training dataset. To prevent this behaviour and make a model generalize to unseen data the representational capacity has to be limited through regularization techniques. The representational capacity can be regularized either by directly changing the capacity of the specified model or by alterations to the training procedure.

For many models the capacity depends on the number of free parameters \[19\], p. 110-116. This is true also for neural networks, although the relationship between the number of parameters and how well the network generalizes is not straightforward \[9\], p. 256-257. Many techniques exist for regularizing neural networks. One choice is to not limit the number of parameters, but rather penalize their magnitude. This is typically achieved through adding a regularizing term to the loss function:

\[
\Lhat(\theta) \overset{\text{def}}{=} \L(\theta) + \frac{\lambda}{2} \| \theta \|_2^2.
\]  

This is known as weight decay or L2-regularization \[19\], p. 231. The hyperparameter \(\lambda\) can be adjusted to change the amount of regularization applied.

Another commonly used regularization procedure for neural networks is early stopping \[9\], p. 259-261. Since neural networks are trained in an iterative manner using gradient descent, the overfitting to training data happens gradually throughout the training process. By stopping the training at an earlier iteration some of the overfitting behaviour can be avoided. To know when to stop the training there needs to be a way to measure the ability of the network to generalize to unseen data. This can be achieved by defining a validation error to be computed on a secondary dataset. The validation data should come from the same underlying data distribution as the training data, but it is never used to directly train the model. During training the validation error typically decreases for multiple iterations and then starts to increase as the network overfits to the training dataset. The training can therefore be stopped when the validation error is minimized to obtain a good model. In practice early stopping can be applied by training for a fixed number of epochs and then saving the model parameters from the epoch with the lowest validation error \[19\], p. 246-252. Epoch is here used to denote one pass through all of the training data.

### 2.2 Implicit Generative Models

Probabilistic machine learning models can be divided into prescribed and implicit models \[40\]. Prescribed models specify a likelihood function \(q(y; \theta)\) parametrized by \(\theta\), that can be used to compute the probability of a data point \(x\) according to the model. Implicit models have no such likelihood function, but are defined through some sampling process involving randomness.

A random process can be created by performing multiple operations on noise from some known distribution. The parameters in an implicit model parametrize the generation process instead of the actual distribution. The generation process defines an implicit distribution \(p_g\), that can not be explicitly computed. The distribution \(p_g\) is completely defined through the parametrized sampling process. While not having an explicit likelihood function complicates learning in implicit models, this also makes them very flexible. The implicit distribution \(p_g\) is not restricted to some closed form expression, making implicit generative models powerful and versatile.

A widely used approach to learning in prescribed models is maximum likelihood \[8\], p. 23]. Maximum likelihood aims to find parameters \(\theta\) such that the likelihood function is maximized for a training dataset. This is equivalent to minimizing the log of the likelihood, which is often a more convenient problem to solve. Finding parameters using maximum likelihood is not
possible in implicit models, since the likelihood function itself can not be computed or even written down explicitly.

Learning in implicit models instead has to be performed using some method that does not involve the likelihood function directly. Since samples can be drawn from the model distribution, any method that only needs a set of data samples and a set of samples generated by the model can be used \[40\]. Using these sample sets the difference between the model distribution and the data distribution can be estimated. The difference estimate can then be used to improve the implicit model distribution.

Conditional Implicit Generative (CIG) models extend the general idea of implicit models to conditional distributions. In these models, \(p_g(y|x)\) is implicitly defined through a generation process. The distribution of \(x\) is not modelled directly, so whether it is implicit or not is of no importance. The random generation process for CIG models is still parametrized by learnable parameters, but it also depends on the \(x\)-value.

Different methods can be considered for training CIG models. These are mainly distinguished by two design choices:

- How is the difference between the model distribution and data distribution measured?
- What is the practical process used to minimize this difference?

The first choice is a matter of finding a theoretically sound way to measure the difference between two probability distributions. The second choice involves finding a useful learning method. The generation process underlying \(p_g\) should be changed such that the distribution difference is minimized. This might include formulating a loss function that approximates the chosen difference measure. These choices are further complicated by the fact that \(p_g\) is a conditional distribution. Different values of \(x\) result in different distributions. The difference to the data distribution has to be minimized for all likely values of \(x\).

In the following sections, Conditional GANs will be introduced as a special case of CIG models. Also Generative Moment Matching Networks (GMMNs) will be described. GMMNs can either be viewed as a subset of CGANs or as a separate type of CIG model.

2.3 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a framework for training implicit generative models using neural networks \[18\]. To generate samples similar to some dataset the underlying distribution of the data is approximated by the GAN. Popularly, GANs have been used to create images mimicking a given collection of training examples \[64\].

2.3.1 Unconditional GANs

A GAN consists of a generator \(G\) and a discriminator \(D\), see figure 2.3a \[18\]. The generator \(G\) is a function that takes a sample of noise \(z \sim p_z(z)\) and outputs a generated sample \(G(z)\).

The noise distribution \(p_z\) can be chosen freely as any distribution that can easily be sampled from. Typically a standard Gaussian \[12\] \[39\] or uniform distribution \[18\] \[39\] is used. More complex choices like mixtures of t-distributions have also been proposed to improve diversity of generated samples \[57\].

The discriminator \(D\) is a function that tries to discriminate between fake samples generated by \(G\) and real samples from a dataset. The output of \(D\) in the standard GAN formulation is limited to \((0, 1)\) and can be interpreted as an estimated probability that a given sample is real.
2.3. Generative Adversarial Networks

Training of a GAN corresponds to searching for $G$ and $D$ to solve a minmax problem \[18\]. In the standard GAN formulation, this is described by the following expression:

$$ V(D, G) = \min_G \max_D V(D; G) $$ \[2.12\]

where $V(D, G)$ is referred to as the GAN training objective. Consider the role of maximizing with respect to $D$. The first term in the objective means that a good discriminator should assign high probabilities to real samples. The second term says that a good choice for $D$ should assign low probability to samples generated by $G$. Looking at the minimization with respect to $G$, the first term is just constant. The second term says that the best $G$ is such that $D$ assigns high probability to its samples. Intuitively this outlines a form of competitive game, where the generator is trying to fool the discriminator and the discriminator is trying to learn not to get fooled.

The generator $G$ induces an implicit generator distribution $p_g$. A specific choice of noise distribution $p_z$ and generator $G$ together define $p_g$ \[18\]. The generator distribution can be sampled from by sampling $z \sim p_z(z)$ and passing $z$ through the generator to get $G(z) \sim p_g$. Note however that $p_g$ is completely implicitly defined by this sampling process. It is typically not possible to actually compute the probability density $p_g(y)$ for any value of $y$. By defining $p_g$ it is possible to reformulate eq. 2.13 using only expectations over $y$:

$$ V(D, G) = \mathbb{E}_{y \sim p_d(y)} [\log(D(y))] + \mathbb{E}_{z \sim p_{z}(z)} [\log(1 - D(G(z)))] $$ \[2.14\]

Searching for any general functions $G$ and $D$ is not possible in practice. The search space can be restricted by parametrizing $G$ and $D$ as neural networks with parameters $\theta_G$ and $\theta_D$ \[18\]. The search is then over the space of network parameters and standard neural network training procedures, as explained in section 2.1, can be applied. The optimization in eq. 2.12 can then be reformulated:

$$ \max \theta_G \mathbb{E}_{y \sim p_d(y)} [\log(D(y; \theta_D))] + \mathbb{E}_{z \sim p_{z}(z)} [\log(1 - D(G(z; \theta_G); \theta_D))] $$ \[2.15\]

and

$$ \min \theta_D \mathbb{E}_{z \sim p_{z}(z)} [\log(1 - D(G(z; \theta_G); \theta_D))] $$ \[2.16\]
2.3. Generative Adversarial Networks

where the generator and discriminator training has been split up for clarity. The expectation over $y$ can be approximated with a training dataset as in eq. 2.4 before applying gradient descent. Turning the maximization into minimization by multiplying the objective with $-1$, the following loss functions are arrived at:

$$\mathcal{L}_D(\theta_D) \overset{\text{def}}{=} -\frac{1}{N_d} \sum_{i=1}^{N_d} \log(D(y^{(i)}; \theta_D)) - \mathbb{E}_{z \sim p_z} \left[ \log(1 - D(G(z); \theta_D)) \right]$$  \hfill (2.17)

$$\mathcal{L}_G(\theta_G) \overset{\text{def}}{=} \mathbb{E}_{z \sim p_z} \left[ \log(1 - D(G(z; \theta_G); \theta_D)) \right].$$ \hfill (2.18)

Each training step includes $N_d$ data samples $y^{(i)}$, each from the underlying data distribution $p_d$. Basic GAN training would take one gradient descent step minimizing eq. 2.17 and one minimizing eq. 2.18 for each batch of training data [18]. The remaining expectations are over the known noise distribution $p_z$. These can be estimated in each training step using samples drawn from $p_z$. Such a monte carlo estimate leads to an unbiased estimate of the true loss.

2.3.2 Conditional GANs

Conditional GAN (CGAN) is an extension to the GAN model where both discriminator and generator also take a conditioning variable $x$ as input, as can be seen in figure 2.31. A CGAN functions much like a normal GAN but with a slightly modified training objective [30]:

$$\min_G \max_D V_c(D, G)$$ \hfill (2.19)

$$V_c(D, G) \overset{\text{def}}{=} \mathbb{E}_{z \sim p_z} \left[ \mathbb{E}_{y \sim p_{y|x}(y|x)} \left[ \log(D(y|x)) \right] + \mathbb{E}_{z \sim p_z} \left[ \log(1 - D(G(z|x); \theta_D)) \right] \right],$$ \hfill (2.20)

or equivalently, letting $p_g$ be the implicit conditional distribution induced by the generator $G$ and choice of noise distribution $p_z$:

$$V_c(D, G) = \mathbb{E}_{z \sim p_z} \left[ \mathbb{E}_{y \sim p_{y|x}(y|x)} \left[ \log(D(y|x)) \right] + \mathbb{E}_{y \sim p_{y|x}(y|x)} \left[ \log(1 - D(y|x)) \right] \right].$$ \hfill (2.21)

Note the conditioning on $x$ in $G$ and $D$ that is not present in eq. 2.13. This means that the generator learns a conditional distribution $p_g(y|x)$ that approximates the true conditional data distribution $p_d(y|x)$.

When in practice using neural networks for $G$ and $D$ the conditioning on $x$ can be achieved by simply concatenating it to the network input [30]. The generator network then takes both noise $z$ and the conditioning variable $x$ as input. Similarly the discriminator network takes both a sample $y$ and the conditioning variable $x$. It is also possible to build more complex architectures where $x$ alone is passed through multiple hidden layers before being concatenated with the other network input $z$.

The training procedure for unconditional GANs presented earlier in this section translates to CGANs as well. A difference is that the training dataset needs to contain pairs $(x, y)$ instead of only single samples $y$. This is necessary for both networks to learn to adapt their outputs to the value of $x$. Loss functions corresponding to eq. 2.17 and 2.18 for the CGAN case are:

$$\mathcal{L}_D(\theta_D) \overset{\text{def}}{=} -\frac{1}{N_z} \sum_{j=1}^{N_z} \left\{ \frac{1}{N_d} \sum_{i=1}^{N_d} \log \left( D \left( y^{(j,i)}; \theta_D \right) \right) 
+ \mathbb{E}_{z \sim p_z} \left[ \log \left( 1 - D \left( G(z^{(j)}; \theta_G); \theta_D \right) \right) \right] \right\}$$ \hfill (2.22)

$$\mathcal{L}_G(\theta_G) \overset{\text{def}}{=} \frac{1}{N_z} \sum_{j=1}^{N_z} \left\{ \mathbb{E}_{z \sim p_z} \left[ \log \left( 1 - D \left( G(z^{(j)}; \theta_G); \theta_D \right) \right) \right] \right\},$$ \hfill (2.23)

where

$$x^{(j)} \sim p_d(x)$$ \hfill (2.24)

$$y^{(j,i)} \sim p_d(y|x^{(j)}).$$ \hfill (2.25)
Both $x^{(j)}$ and $y^{(j)}$ are available only as samples in a training dataset. They always occur as pairs sampled from the true joint distribution $p_d(x, y)$. The sum over $N_d$ training samples is where batching would be applied when performing stochastic gradient descent. The value of $N_d$ is determined by the dataset. If $x$ is continuous the only option is typically $N_d = 1$. Assuming that each value for $x$ only occurs once in the dataset, there is only one sample from $p_d\left(y\mid x^{(j)}\right)$, the corresponding $y$ in the training data pair. In cases where $x$ is categorical, such as in image generation from a class-label, there might be many samples of $y$ available in the training set for each $x$. Just like in unconditional GANs the expectations over $p_z$ can in practice be handled using Monte Carlo estimates. The number of samples used in such estimates is not restricted by the data, since any amount of noise easily can be sampled from $p_z$ (assuming the noise distribution is chosen appropriately).

The direct connection between GANs and CGANs can be made even more clear by considering not conditional distributions, but the joint distributions of $x$ and $y$. Since the generator does not estimate a distribution over $x$, the joint generator distribution factorizes as $p_g(x, y) = p_g(y\mid x)p_d(x)$. Each $x$ the generator is fed with comes from some external source and is always distributed according to $p_d$. For comparison also the joint data distribution can be factorized $p_d(x, y) = p_d(y\mid x)p_d(x)$. By reparametrizing $w \triangleq [x, y]^\top$ the joint distribution becomes $p_g(w)$, which matches the unconditional GAN case. This allows for viewing the generator training in CGAN the same as for the unconditional case, with the restriction that the $x$-part of $p_g(w)$ is restricted to always be equal to $p_d(x)$.

### 2.3.3 Theoretical Optima

It can be shown [18] that for a fixed generator $G$ the optimal discriminator is given by:

$$D^*(y) \triangleq \frac{p_d(y)}{p_d(y) + p_g(y)},$$

or in the conditional case:

$$D^*(y\mid x) \triangleq \frac{p_d(x, y)}{p_d(x, y) + p_g(x, y)} = \frac{p_d(y\mid x)}{p_d(y\mid x) + p_g(y\mid x)},$$

where the last equality follows from the factorizations of the joint distributions. Since neither $p_d$ nor $p_g$ can be computed this is purely a theoretical result. Note also that there are no guarantees that $D^*$ is in the set of possible discriminators once it has been restricted to neural networks parametrized by $\theta_D$ (this would indeed be highly unlikely). Inserting $D^*$ into eq. 2.13 yields [18]:

$$V(D^*, G) = -\log 4 + 2D_{JS}(p_d\|p_g)$$

$$D_{JS}(p\|q) \triangleq \frac{1}{2}D_{KL}(p\|\frac{p + q}{2}) + \frac{1}{2}D_{KL}(q\|\frac{p + q}{2})$$

$$D_{KL}(p\|q) \triangleq \int_\mathcal{U} p(u) \log \left(\frac{p(u)}{q(u)}\right) du,$$

where $\mathcal{U}$ is the domain of the variable $u$ and $D_{JS}$ is the Jensen-Shannon divergence [18], which in turn is defined through the Kullback-Leibler divergence [1 p. 55-58]. These divergences measure the difference between two probability distributions. In eq. 2.28 the Jensen-Shannon divergence expresses the difference between $p_d$ and $p_g$. This gives some intuition about the GAN training procedure. It is possible to interpret the training of $G$ as changing $p_g$ so as to minimize $D_{JS}(p_d\|p_g)$. This also holds for the CGAN case, through inserting $D^*$ in eq. 2.21. The Jensen-Shannon divergence is minimized when the distributions are the same, meaning that an optimal generator is such that $p_g = p_d$ [18]. Although this gives some nice intuition, it is worth noting that these arguments rely on the assumption that $D$ can reach this optimum. The limitations of using specific neural networks are not considered.
As can be seen, the GAN and CGAN training objectives theoretically lead to optimization problems with suitable optima. Despite this, the standard formulation of the GAN objective has shown to exhibit some less favourable properties in practice. Issues related to gradients used in training makes the GAN training process highly unstable \[42\]. This has motivated research efforts to come up with alternative GAN formulations \[14\] [38] [13].

The interpretation of the training as minimizing the Jensen-Shannon divergence offers a direct connection between CGANs and the general class of Conditional Implicit Generative models described in section 2.4. In the standard CGAN formulation, \(D_{JS}\) takes the role of the difference measure between distributions. The Jensen-Shannon divergence can however not be directly minimized, since the definition involves integrals and explicit probability density functions. Instead, an approximate training process based on the discriminator is used. This is a computationally costly approach, requiring the training of an additional neural network. Why such a complex approach is necessary will be further explored in the next sections.

### 2.3.4 \(f\)-divergence Minimization

A wide family of difference measures between probability distributions that is of particular interest to the analysis of GANs is \(f\)-divergences \[42\]. For any convex function \(f: \mathbb{R}_+ \rightarrow \mathbb{R}\) such that \(f(1) = 0\), given some mild continuity assumptions, the corresponding \(f\)-divergence is defined as:

\[
D_f(p\|q) = \int_{\mathcal{U}} q(u) f\left(\frac{p(u)}{q(u)}\right) du, \tag{2.31}
\]

between distributions \(p\) and \(q\) defined over \(\mathcal{U}\). An example of an \(f\)-divergences is the Kullback-Leibler divergence in eq. 2.30, with \(f(u) = u\log(u)\) \[42\]. Also the Jensen-Shannon divergence, minimized by the standard GAN formulation, is an \(f\)-divergence with

\[
f(u) = -\frac{u+1}{2} \log\left(\frac{1+u}{2}\right) + \frac{u}{2} \log(u). \tag{2.32}
\]

The standard GAN is thereby minimizing a particular \(f\)-divergence \[42\].

It turns out that by changing the training objective \(V(D, G)\) it is possible to design GANs that minimize many different \(f\)-divergences. Nowozin et al. \[12\] propose a general GAN formulation, referred to as \(f\)-GAN, for minimizing any \(f\)-divergence \(D_f\). To allow for this, let the discriminator be written as \(D(y) = g_f(T(y))\), where \(g_f\) is the activation function in the last layer of the neural network and \(T\) all previous layers. The training objective \(V_f\) for minimizing \(D_f\) then becomes:

\[
\min_G \max_T V_f(T, G) \tag{2.33}
\]

\[
V_f(T, G) \overset{\text{def}}{=} \mathbb{E}_{y \sim p_d(y)} [g_f(T(y))] + \mathbb{E}_{z \sim p_z(z)} [-f^*(g_f(T(G(z))))]. \tag{2.34}
\]

Here \(f^*\) is the conjugate function of \(f\), defined by \(f^*(t) \overset{\text{def}}{=} \sup_u \{ tu - f(u) \}\). There is some freedom in the choice of \(g_f\), but it needs to only take values in the domain of \(f^*\). Nowozin et al. provide a list of \(f^*\) and suitable \(g_f\) for many different \(f\)-divergences. For example, to minimize the Kullback-Leibler divergence \(D_{KL}\), the requirement is \(f^*(t) = \exp(t-1)\) and \(g_f(z) \in \mathbb{R}\) \[42\]. For example just \(g_f(z) = v\) could be used.

In the standard GAN formulation the connection to the Jensen-Shannon divergence arose from assuming an optimal discriminator. A different way to see the connection between the GAN training objective and \(f\)-divergence is through the result:

\[
D_f(p^\parallel q) \geq \sup_{T \in \mathcal{T}} V_f(T, G), \tag{2.35}
\]

where \(\mathcal{T}\) can be any class of functions \[41\] [12]. When training GANs in practice \(\mathcal{T}\) is simply the set of functions achievable using a specific neural network architecture and adjusting the
network parameters $\theta_D$. Training the discriminator corresponds to making this bound tighter. The generator training then minimizes the lower bound.

Also CGANs can be formulated for different $f$-divergences in a similar way. Inserting the joint generator and data distributions into the $f$-divergence in eq. 2.31 yields:

$$D_f(p_d(x,y)||p_g(x,y)) = \int_{x \times y} p_g(x,y) f \left( \frac{p_d(x,y)}{p_g(x,y)} \right) dx \ dy$$

$$= \int_x p_d(x) \int_y p_g(y|x) f \left( \frac{p_d(y|x)p_d(x)}{p_g(y|x)p_g(x)} \right) dy \ dx$$

$$= \mathbb{E}_{x \sim p_d(x)} \left[ \int_y p_g(y|x) f \left( \frac{p_d(y|x)}{p_g(y|x)} \right) dy \right]$$

$$= \mathbb{E}_{x \sim p_d(x)} \left[ D_f(p_d(y|x)||p_g(y|x)) \right].$$

This shows that there are two possible viewpoints of how $f$-divergence can be applied to the conditional GAN case, either as the divergence between the joint distributions or as the expected divergence of the conditional distributions. As shown these viewpoints are equivalent. The inequality in eq. 2.35 is equally true for the joint distributions $p_g$ and $p_d$, resulting in:

$$D_f(p_d(x,y)||p_g(x,y)) \geq \sup_{T \in \mathcal{T}} \left( \mathbb{E}_{(x,y) \sim p_d(x,y)} \left[ g_f(T(y|x)) \right] + \mathbb{E}_{(x,y) \sim p_g(x,y)} \left[ -f^*(g_f(T(y|x))) \right] \right)$$

$$= \sup_{T \in \mathcal{T}} \left( \mathbb{E}_{x \sim p_d(x)} \left[ \mathbb{E}_{y \sim p_g(y|x)} \left[ g_f(T(y|x)) \right] + \mathbb{E}_{y \sim p_g(y|x)} \left[ -f^*(g_f(T(y|x))) \right] \right) \right),$$

where $T(y|x)$ now corresponds to the CGAN discriminator without the last layer activation function $g_f$. This allows for defining the CGAN training objective corresponding to $D_f$ according to eq. 2.42:

$$\min_G \max_T \mathcal{V}_{c,f}(T,G)$$

$$\mathcal{V}_{c,f}(T,G) \overset{\text{def}}{=} \mathbb{E}_{x \sim p_d(x)} \left[ \mathbb{E}_{y \sim p_g(y|x)} \left[ g_f(T(y|x)) \right] + \mathbb{E}_{y \sim p_g(y|x)} \left[ -f^*(g_f(T(y|x))) \right] \right].$$

Nowozin et al. [14] note that the optimal $D(x) = g_f(T(x))$, for which the bound in eq. 2.35 is tight, is:

$$D^*(y) \overset{\text{def}}{=} f' \left( \frac{p_d(y)}{p_g(y)} \right),$$

where $f'$ is the first derivative of $f$. Analogously, the optimal $D$ in the conditional case:

$$D^*(y|x) \overset{\text{def}}{=} f' \left( \frac{p_d(x,y)}{p_g(x,y)} \right) = f' \left( \frac{p_d(y|x)}{p_g(y|x)} \right).$$

makes the bound in eq. 2.41 tight. As with the optimal discriminator in the standard GAN case, this is purely a theoretical result since $p_d$ and $p_g$ can not be computed.

Through the $f$-GAN framework, the role of CGANs as CIG models can be further generalized. As has been shown, the difference measure being minimized in CGANs is not just restricted to Jensen-Shannon Divergence, but can be any $f$-divergence. For any $f$-divergence, designing a learning process for the model is then straightforward by using eq. 2.43. Additionally, the supremum in eq. 2.41 offers an explanation of why the discriminator network is necessary in the training process.

### 2.3.5 Integral Probability Metrics

Apart from $f$-divergences there exist other difference measures that have been used for training GANs. One such family is the Integral Probability Metrics (IPMs) [5]. For a class of functions
$\mathcal{F}$, the corresponding IPM between two distributions $p$ and $q$ is defined as:

$$\gamma_{\mathcal{F}}(p,q) \overset{\text{def}}{=} \sup_{f \in \mathcal{F}}|E_p[f(y)] - E_q[f(y)]|.$$  

(2.47)

For example, with $\mathcal{F} = \{f : \max_x |f(x)| \leq 1\}$ the metric $\gamma_{\mathcal{F}}$ becomes the so-called Total Variation distance. Although they are not disjoint, there is very little overlap between the IPMs and $f$-divergences.

Other IPMs include the Wasserstein distance and $L_p$-metrics on function spaces, both of which have been used to define alternative GAN models. Optimizing the Wasserstein distance leads to the formulation of Wasserstein GAN (WGAN) [3]. WGANs use the training objective:

$$\min_D \max_G V_W(D,G) = E_{y \sim p_d(y)}[D(y)] - E_{z \sim p_z}(D(G(z))),$$

(2.48)

with the additional constraint that $D$ must be $k$-Lipschitz [3] for some $k$. In practice the Lipschitz constraint can be enforced by clamping the weights of the discriminator neural network to some fixed interval, for example $[-0.01, 0.01]$. WGANs provide more useful gradient information than standard GANs, making training more stable.

Due to the weight clamping in the WGAN discriminator network most weights end up around the limits of the clamping interval [22]. This restricts the capacity of the model. An alternative way to satisfy the Lipschitz constraint has been introduced by Gulrajani et al. [22] in their WGAN-GP model. They propose adding a gradient penalty to the training objective to directly constrain the gradients of $D$. This has shown to improve training stability further and in particular allow for using very deep neural networks in GANs.

### 2.3.6 Generator Training Objective

Early on in the training process the discriminator will typically outperform the generator [18]. Although this is to be expected, in the standard GAN formulation of eq. 2.12 this creates practical issues for learning a good generator. For a close to optimal discriminator, $D(G(z; \theta_D); \theta_D)$ will be close to 0 (low estimated probability that samples are real). At this value, the generator training objective $\mathbb{E}_{z \sim p_z}(\log(1 - D(G(z; \theta_G); \theta_D)))$ has a quite flat loss surface, meaning that near-zero gradients will be produced when optimizing with respect to $\theta_G$. These small gradients will make it very slow to learn a good generator using gradient descent. Plenty of methods have been proposed in the literature to get around this problem [34]. In their original paper Goodfellow et al. [13] propose the reformulation

$$\max_{\theta_G} \mathbb{E}_{z \sim p_z}(\log(D(G(z; \theta_G); \theta_D)))$$

(2.49)

for the generator training. This objective does not suffer from the small gradient problem. The reformulation results in the same optimum if $G$ is allowed to be any general function. When the generator search space is restricted to neural networks with parameters $\theta_G$ there are no theoretical guarantees of this.

Also for general $f$-divergences Nowozin et al. [42] note that using such an alternative training objective for the generator is beneficial. This means changing out the minimization of $\mathbb{E}_{z \sim p_z}(\log(D(G(z; \theta_D))))$ in eq. 2.34 for:

$$\max_G \mathbb{E}_{z \sim p_z}(g_f(T(G(z))))$$

(2.50)

or in the conditional case:

$$\max_G \mathbb{E}_{x \sim p_d(x)}[\mathbb{E}_{z \sim p_z}(g_f(T(G(z|x)|x)))]$$

(2.51)

\footnote{A function $f$ is $k$-Lipschitz if $\forall x\forall y, |f(x) - f(y)| \leq k |x - y|$ [19, p. 92].}
The discriminator training objective is left unchanged.

The alternative generator objectives have been proposed as a strategy to improve training, rather than being motivated by the underlying theory. Poole et al. provide a more theoretical interpretation of alternative generator objectives as minimizing a separate $f$-divergence. Assume that a GAN discriminator $D'$ has been trained to optimum estimating the $f$-divergence $D_{f_1}$. Any function $f_1$ could be considered, as long as it satisfies the criteria in section 2.3.4 for defining an $f$-divergence. If $f_1'$, the derivative of $f_1$, is invertible the expression for the optimal discriminator in eq. 2.40 can be used to compute:

$$r(y) \overset{\text{def}}{=} \frac{p_d(y)}{p_g(y)} = (f_1')^{-1}(D^*(y)),$$

and in the conditional case:

$$r(x,y) \overset{\text{def}}{=} \frac{p_d(x,y)}{p_g(x,y)} = \frac{p_d(y|x)}{p_g(y|x)} = (f_1')^{-1}(D^*(y|x)).$$

This gives a direct expression for the probability density ratio between $p_d$ and $p_g$. Any $f$-divergence, say for a function $f_2$, can then be expressed as:

$$D_{f_2}(p_d\|p_g) = \mathbb{E}_{y \sim p_g(y)}[f_2(r(y))] = \mathbb{E}_{y \sim p_g(y)}[f_2((f_1')^{-1}(D^*(y)))],$$

where the expectation can be approximated using samples from $p_g$. Compare this to the $f$-divergence definition in eq. 2.31 to see the correspondence. Minimizing eq. 2.54 then becomes the new generator training objective. In practice the optimal discriminator $D^*$ is not available, but the current discriminator at each step of training can be used as an approximation. This allows for training a discriminator to estimate $D_{f_1}$, computing an estimate for the density ratio $r(y)$ and then training the generator to minimize any $f$-divergence $D_{f_2}$. As an example Poole et al. show that from this viewpoint the alternative generator objective in eq. 2.49 corresponds to choosing

$$f_2(u) = \log \left(1 + \frac{1}{u}\right),$$

### 2.3.7 Density Ratio Estimation

As noted above, the GAN discriminator estimates an $f$-divergence and indirectly also the density ratio $r(y)$. Given any estimate of $r(y)$ it is possible to train the GAN generator by minimizing eq. 2.54. There are many different ways to estimate the density ratio $r(y)$.

Mohamed et al. propose three ways to learn an estimate for the density ratio. These all have clear connections to GANs. The first way is to consider a binary classification problem, where a probabilistic classifier is tasked with classifying if a sample comes from $p_d$ or $p_g$. This is very much in line with the original description of the GAN discriminator. Following the $f$-GAN approach, the density ratio can also be estimated by considering a lower bound on an $f$-divergence (as described in section 2.3.4, eq. 2.35 and 2.41). The final method proposed is to directly parametrize the density ratio as $r_\phi(y)$ and minimize the difference to the true ratio. This difference can be measured by something called Bregman divergence, which then leads to an objective to be minimized. This approach has been explored by Uehara et al., resulting in the model referred to as b-GAN.

The density ratio estimation methods are not completely distinct and there are close connections between them. The original GAN formulation, with the discriminator as a probabilistic classifier, can be seen as a special case of the $f$-divergence minimization approach. Also the Bregman divergence minimization has a direct connection to $f$-GANs. The only difference lies in whether the density ratio $r_\phi$ is parametrized directly (b-GAN) or indirectly through $(f')^{-1}(D(y; \theta_D))$ (f-GAN).
2.3.8 Least Squares GAN

Another version of GANs that has seen some use in practice due to its more stable training process is Least Squares GAN (LS-GAN) [38]. In contrast to the previously mentioned GAN variants, LS-GAN is not motivated by some difference metric between distributions. Instead the motivation comes from viewing the discriminator as a probabilistic binary classifier. In the original GAN formulation the discriminator was trained as a classifier using a binary cross-entropy loss function. LS-GAN changes out this loss function for a least squares loss. This type of loss function is commonly used for regression tasks, but can also be utilized for binary classification [38], p. 11-14. In the CGAN case this leads to the following training objective:

\[ V_{LS}(T, G) \overset{\text{def}}{=} \mathbb{E}_{x \sim p_d(x)} \left[ -\frac{1}{2} \mathbb{E}_{y \sim p_c(y|x)} \left[ (T(y|x) - 1)^2 \right] - \frac{1}{2} \mathbb{E}_{z \sim p_z(z)} \left[ T(G(z|x)|x)^2 \right] \right] \]

where \( T \) again is the discriminator without any final layer activation function. Discriminator training consists of maximizing \( V_{LS}(T, G) \) with respect to \( T \). The LS-CGAN generator is trained by the minimization:

\[ \min_G \mathbb{E}_{x \sim p_d(x)} \left[ \frac{1}{2} \mathbb{E}_{z \sim p_z(z)} \left[ (T(G(z|x)|x) - 1)^2 \right] \right]. \]

As with other versions of GANs, the discriminator and generator objectives do not completely match. With the LS-CGAN being mostly empirically motivated this is however neither a problem in practice nor in connection to any underlying theory.

It can be noted that the discriminator objective \( V_{LS}(T, G) \) takes the value 0 when the discriminator can successfully classify all samples (when the generator does not match the data distribution). On the other end of the spectrum, for an optimal generator such that \( p_g = p_d \), the best possible discriminator would always choose \( T(y|x) = 0.5 \). This results in the objective taking the value \( V_{LS}(T, G) = -0.25 \).

The least squares loss functions have less flat areas, leading to better gradients in the GAN training. Mao et al. [38] have shown experimentally that the LS-GAN formulation gives more stable training than the original GAN objective.

2.4 Generative Moment Matching Networks

Previous sections have described GANs and CGANs with slightly different training objectives. One version that differs more substantially from the standard formulation is Generative Moment Matching Networks (GMMNs). Section 2.4.1 will present GMMNs first for the unconditional case. Section 2.4.3 and 2.4.4 will then show how the model can be used when there is a conditioning variable \( x \).

2.4.1 GMMNs and MMD

GMMN is a Generative Implicit model that minimizes the Maximum Mean Discrepancy (MMD) between \( p_g \) and \( p_d \) [34]. MMD is a difference measure between distributions that is based on kernel methods [20]. MMD is actually another integral probability metric (see section 2.3.5), but one where the supremum in eq. (2.47) is taken over a very specific set of functions. The set \( \mathcal{F} \) in MMD is a unit ball in a Reproducing Kernel Hilbert Space (RKHS) [20]. An RKHS is a space of functions with some special properties [47], p. 129-132]. The exact definition is not necessary for the following presentation, so the interested reader is referred to Manton et al. [37] for details. Every specific RKHS is uniquely defined by a kernel function \( k : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R} \). A kernel function \( k(y, y') \) is a real-valued measurement of how similar two
objects $y, y' \in \mathcal{Y}$ are. The kernel function defines the inner product of the RKHS, which is central to the usefulness of MMD.

Through the kernel properties of the RKHS, the squared MMD can be formulated without the supremum in the IPM definition:

$$
\text{MMD}^2(p, q) = E_{y \sim p} [k(y, y)] + E_{y' \sim q} [k(y, y')] - 2E_{y \sim p, y' \sim q} [k(y, y')],
$$

(2.59)

here between two distributions $p$ and $q$ over $\mathcal{Y}$ \[20\]. The kernel function $k$ is here an important design choice. It is desirable that the kernel is chosen from a class known as characteristic kernels \[33\]. These kernels theoretically ensure that MMD$^2(p, q) = 0$ if and only if $p = q$. A common choice of characteristic kernel is the Gaussian kernel:

$$
k(y, y') = \frac{1}{\sqrt{2\pi\sigma^2_k}} \exp\left(-\frac{\|y - y'\|^2}{2\sigma^2_k}\right).
$$

(2.60)

The kernel parameter $\sigma^2_k$ is called scale or bandwidth. With this kernel choice, minimizing MMD is theoretically equivalent to minimizing the distance between all moments of the compared distributions \[34\]. Thus MMD can be called a Moment Matching distance.

Generative Moment Matching Networks \[34\] use a neural network generator exactly like other GAN versions. In the unconditional case, a noise vector $z$ is fed to the generator, propagated through the network and a sample $y = G(z)$ is generated. GMMNs do however not require a separate discriminator network. Instead the squared MMD takes the role of critiquing the generated samples. Let $\{y^{(i)}\}_{i=1}^N$ be a set of $N$ data samples and $\{\tilde{y}^{(i)}\}_{i=1}^N$ a set of $N$ generated samples. An estimate of the squared MMD between $p_g$ and $p_d$ can then be computed as \[34\]:

$$
\text{MMD}^2 \overset{\Delta}{=} \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N k(y^{(i)}, y^{(j)}) + \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N k(y^{(i)}, \tilde{y}^{(j)}) - 2 \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N k(y^{(i)}, y^{(j)}) + 2 \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N k(y^{(i)}, \tilde{y}^{(j)}).
$$

(2.61)

This provides a closed form, differentiable objective for the generator to minimize. Keeping in mind that each generated $\tilde{y}$ depends on the generator parameters $\theta_G$, training is straightforward through gradient descent.

An interpretation of the MMD is to think of it as answering the question: Given these two sets of samples, how unlikely is it that they come from the same distribution? Higher MMD-values signal that the sets are very different, whereas for very small MMD-values the samples seem to be from the same distribution. Minimizing MMD is thus an intuitive way to make $p_g$ more similar to $p_d$.

As can be seen in eq. \[2.61\], the computation of MMD$^2$ scales quadratically with $N$ due to the nested sums. It is therefore necessary to compute MMD batch-wise \[34\]. This additionally fits nicely with the generator training, as one batch of samples can be generated, MMD computed and then the generator network trained. For MMD to be useful the batches of samples have to be representative of the underlying distributions. Because of this, a somewhat larger batch size is usually used for GMMNs compared to other GANs \[33\].

GMMNs are also implicit generative models, since the generator is equivalent to the one used in GANs. The model formulation and training method maps nicely to the presentation in section 2.2, with MMD taking the role of the difference measure between distributions. It is possible to treat GMMNs as a separate model from GANs, but they can also be viewed as just a special case of the GAN framework. The MMD measurement takes the role of the adversary, trying to determine if the generated samples are real or not. The viewpoint that GMMNs fit under the general GAN definition is taken throughout this thesis. As GMMNs are extended with kernel learning methods in the next section the GAN-connection will be made even stronger.

\[2\] The $n$th moment of a distribution $p(x)$ is $E_p(x) \{x^n\}$. Moments in higher dimensions are defined similarly, but with possibly different powers for each entry in a random vector \[44\], p. 703-704.
2.4.2 Deep Kernel Learning

GMMNs rely heavily on the choice of kernel and its parametrization. When using a finite set of samples, the Gaussian kernel might not be powerful enough to correctly distinguish between distributions through MMD. Dziugaite et al. propose a way to improve on the GMMN model by jointly learning the generator and the kernel used in MMD. Deep kernel learning is a method for learning expressive kernels on complex data, by utilizing neural networks in the kernel formulation. Deep kernels can be constructed by the combination of a feature transformation and a base kernel. For a base kernel \( k \), for example the Gaussian kernel, a deep kernel can be defined:

\[
\tilde{k}_{\theta_\lambda}(y, y') = k_{\sigma_k^2}(\xi_{\theta_\kappa}(y), \xi_{\theta_\kappa}(y')).
\]  

Here \( \xi_{\theta_\kappa}(y) \) is the output of a neural network with parameters \( \theta_\kappa \), when \( y \) is fed as input. This is a parametrized feature transformation into some new feature space, where the base kernel is then applied. The deep kernel has parameters \( \theta_\kappa \) that includes both the parameters of the neural network and of the base kernel (the scale \( \sigma_k^2 \) in this case). When applied to GMMNs, the parameters of the kernel are learned by maximizing the estimate \( \tilde{\text{MMD}}^2 \). This has an intuitive interpretation as changing the kernel such that the two distributions \( p_g \) and \( p_d \) can be better distinguished (thus giving higher MMD values).

When the kernel is learned jointly with the generator, the GMMN training can be summarized:

\[
\min_G \max_k \tilde{\text{MMD}}^2.
\]  

The generator \( G \) should produce samples that minimize the MMD, but the kernel used in the MMD computation should result in the highest value possible. This formulation is very similar to the standard GAN training in eq. 2.12 to the point where this approach is referred to as MMD-GAN. The training of such an MMD-GAN also follows that of other GAN models, where the two networks involved are trained by taking alternating training steps. It should be noted that the auxiliary neural networks fill different roles in standard GANs and MMD-GANs. A discriminator network does itself act as the classifier between distributions. The network in MMD-GANs simply performs a feature transformation. The actual discrimination between \( p_g \) and \( p_d \) is still done through MMD.

In the previous section it was mentioned that the kernel used for MMD should be chosen from the class of characteristic kernels. This becomes a complication when deep kernel learning is applied. Even if the base kernel is chosen to be characteristic, the resulting deep kernel might not be. One way to guarantee this however, is to use a feature transformation \( \xi \) that is injective. Achieving this with a neural network is not trivial. Dziugaite et al. solve this problem by using an autoencoder to approximately enforce injectivity. In an autoencoder, both the feature transformation \( \xi \) and its inverse \( \xi^{-1} \) are learned jointly as neural networks. The autoencoder loss function is constructed to ensure that \( \xi^{-1}(\xi(y)) = y \) for all \( y \)’s. By adding on the autoencoder loss to the MMD-GAN training, \( \xi \) becomes approximately injective and the kernel approximately characteristic. It should be noted that even though injectivity is important in theory, empirical investigations have shown that good results are possible even without this constraint.

2.4.3 Conditional GMMNs

Recall from the beginning of section 2.4.1 that MMD is an IPM that includes a supremum over a set of functions in an RKHS (see eq. 2.47). This can be viewed as using the function that best distinguishes between the two distributions that the MMD is computed between. The function that attains the supremum is called witness function. Now consider the conditional case, where MMD is used to distinguish between two conditional distributions \( p_d(y|x) \) and \( p_g(y|x) \). These distributions both change with the value of \( x \), so the witness function is conditioned on \( x \).
function that best distinguishes between them must also do so. A separate witness function and MMD value is achieved for each $x$.

Building on this idea, through the concept of conditional Hilbert space embeddings, Ren et al. \cite{ren2017conditional} introduce a conditional extension to the GMMN model. The details underlying their Conditional GMMN (CGMMN) model are beyond the scope of this thesis, so the interested reader is referred to \cite{ren2017conditional} and \cite{gao2019conditional} for a more theoretical presentation. In the CGMMN model the standard MMD is replaced by a measurement of the difference between two conditional embedding operators. Let $C_{y|x}$ be the empirical conditional embedding operator associated with a set of data samples $\{(x^{(i)}, y^{(i)})\}_{i=1}^N$. Let $\hat{C}_{y|x}$ be the corresponding operator for generated samples $\{(\hat{x}^{(i)}, \hat{y}^{(i)})\}_{i=1}^N$. Typically, the $x$s in both sets would be the same, since the generator only generates $y$-values. To clarify, these empirical operators can be viewed as matrices in some high-dimensional vector space, defined by a chosen kernel.\cite{gao2019conditional}

The CGMMN model minimizes the squared Conditional MMD (CMMD), defined by \cite{gao2019conditional}:

$$\text{CMMD}^2 \overset{\text{def}}{=} \left\| \hat{C}_{y|x} - C_{y|x} \right\|_F^2$$

\hspace{1cm} (2.64)

The norm used on the operators is the Frobenius (or Hilbert-Schmidt) norm \cite[p. 40]{gao2019conditional}. Each $K$ and $L$ is a gram matrix, such that $\{K_{d,i,j}\} = k_d(x^{(i)}, x^{(j)})$ and $\{L_{d,i,j}\} = k_y(y^{(i)}, y^{(j)})$. The kernel $k_x$ is defined between $x$-values and $k_y$ between $y$-values. The subscripts of $d$ and $g$ denote that the gram matrices are computed for either the set of real data samples or the set of generated samples. The combined gram-matrices contain the kernel values between the sets, for example $\{L_{d,y}\}_{i,j} = k_y(y^{(i)}, y^{(j)})$. Matrices are regularized with a small value $\lambda$ as $\hat{K} \overset{\text{def}}{=} K + \lambda I$ before their inverse is computed.

The CMMD in eq. 2.65 defines a straightforward objective that can be minimized with respect to the parameters of the generator network. As with other CGANs, the generator needs to be modified to also take $x$ as input. Batching can be applied similarly to unconditional MMD \cite{gao2019conditional}.

It is worth noting that computing the matrix inverses in eq. 2.65 is computationally expensive, resulting in a complexity of $O(N^3)$. This is unbearably slow to perform during training. Ren et al. \cite{gao2019conditional} do however note that since the $K$-matrices do not change during training, it is enough to compute the inverses once. The matrix inverses for each batch can be computed during the first epoch and then simply stored in memory to be reused throughout remaining epochs.

### 2.4.4 Joint GMMNs

An alternative way to extend GMMN to the conditional case is to use MMD in the joint space of $x$ and $y$. This is a quite simple approach that requires very little changes to the GMMN model. Let $w = [x, y]^T$ be the concatenation of the $x$ and $y$-vectors. It is then possible to use the MMD objective from eq. 2.61, but where the kernels are computed over the values of $w$ instead of $y$. In the generated samples $\hat{w} = [\hat{x}, \hat{y}]^T$ the value of $x$ still comes from the training data. This reformulation corresponds to computing the MMD between the joint distributions $p_d(x, y)$ and $p_g(x, y)$. As before, the joint generator distribution factorizes as $p_g(x, y) = p_g(y|x)p_g(x)$, since each $x$ comes from the training data. This approach will be referred to as joint GMMN or just GMMN (since conditioning on $x$ is always used throughout upcoming chapters). This idea is similar to an approach proposed by Gao et al. \cite{gao2019conditional}, but they work in the joint space of $y$ and $z$ (the noise vector). The joint GMMN is just a straightforward special case of the GMMN model and needs no special considerations.

It might seem unnecessary to work in the joint space, when only a generator for $y$ should be learned. Note however that the distribution over $x$ does not have to be explicitly modelled by the generator. The joint MMD measures the difference between $p_d(y|x)p_d(x)$ and
For an example, the model in eq. (2.30) can be extended to heteroskedastic noise. See figure 2.4 for an example. The model in eq. (2.71) can be extended to heteroskedastic noise.
2.5. Probabilistic Regression

regression by letting the noise variance also depend on \( x \), that is \( \epsilon \sim \mathcal{N}(0,u(x)) \) for some function \( u \). This results in

\[
p(y|x) = \mathcal{N}(y|h(x),u(x)).
\]  

(2.72)

Note that \( \epsilon \) is still assumed to be Gaussian.

![Figure 2.4: Samples from an example model with heteroskedastic noise. Here \( y = 2x + \epsilon \) where \( \epsilon \sim \mathcal{N}(0,0.2x^2) \).](image1)

![Figure 2.5: Example of a multimodal distribution. The probability density function of the random variable \( x \) has modes at both \( x = -1.5 \) and \( x = 2 \).](image2)

In some real world problems the distribution of noise has peaks for multiple different values \([8]\). Formally such maxima of a probability density function are called modes of the distribution \([63]\). Depending on if a distribution has one or multiple modes it is referred to as unimodal or multimodal. Figure 2.5 shows an example of a multimodal distribution. Since the Gaussian distribution is unimodal it is not a good model when the underlying data distribution is multimodal.

![Figure 2.6: Conceptual view of how a neural network can be used for probabilistic regression. The output of the network defines a Gaussian distribution.](image3)

Neural networks can easily be used for regression by adopting the model from eq. 2.71 and letting the network represent \( h \). A more expressive model can be achieved by adopting the model from eq. 2.72, letting the network output both the mean \( h(x) \) and noise variance \( u(x) \). This allows for modeling heteroskedastic noise distributions, but \( p(y|x) \) is still limited to a unimodal Gaussian \([30]\). Figure 2.6 shows how a neural network can be used for probabilistic regression in this way.

2.5.2 Gaussian Processes

One popular probabilistic regression model is the Gaussian Process (GP). While GPs can be used in many settings, the standard regression model with additive Gaussian noise will be considered here. Assuming Gaussian noise substantially simplifies the estimation of \( p(y|x) \) for the GP model \([47], p. 16-19\).
A GP is any set, possibly infinite, of random variables such that any finite subset of the variables are jointly Gaussian distributed [17, p. 13]. In the one-dimensional case GPs use the model \( y = h(x) + \epsilon \) from eq. [2.71], but assume the function \( h \) itself to also be a random variable. The following presentation will assume that \( y \) is one-dimensional, but GPs can be applied in a similar way to higher-dimensional vectors. A Gaussian process defines a distribution over the function space \( h \) belongs to. This is written as [17, p. 13]:

\[
\begin{align*}
    h & \sim \mathcal{GP}(m(x), k(x, x')) \\
    m(x) & \overset{\text{def}}{=} \mathbb{E}[h(x)] \\
    k(x, x') & \overset{\text{def}}{=} \mathbb{E}[(h(x) - m(x))(h(x') - m(x'))],
\end{align*}
\]

where \( m(x) \) is called the mean function and \( k(x, x') \) the covariance function. Note that \( k(x, x') \) completely encodes any belief about the covariance of \( h(x) \) and \( h(x') \). This is an application of the commonly used kernel trick [17, p. 12].

Using this framework, the distribution of \( h^* \overset{\text{def}}{=} h(x^*) \) can be predicted for some \( x^* \) of interest, given an observed dataset \( \{(x_i, y_i)\}_{i=1}^n \). Let \( y \overset{\text{def}}{=} (y_1, \ldots, y_n)^\top \) and let \( X \) be the matrix with \( x_i \) as column \( i \). Assuming \( m(x) = 0 \) (a prior belief that \( h \) is zero-centered) it then holds that the stacked vector of \( y \) and \( h^* \) is Gaussian with mean zero and covariance matrix \( C^* \):

\[
\begin{bmatrix} y^* \\ h^* \end{bmatrix} \sim \mathcal{N}(0, C^*)
\]

\[
C^* = \begin{bmatrix} K + \sigma_n^2 I & k^\ast \\ k^\ast & k(x^*, x^*) \end{bmatrix} \in \mathbb{R}^{(n+1) \times (n+1)},
\]

where \( K \in \mathbb{R}^{n \times n} \) such that \( \{K\}_{i,j} = k(x_i, x_j) \) and \( k^\ast \overset{\text{def}}{=} (k(x^*, x_1), \ldots, k(x^*, x_n))^\top \). For prediction purposes the conditional distribution of \( h^* \) alone is of interest. Due to properties of the Gaussian distribution it follows that [17, p. 16–17]:

\[
\begin{align*}
    h^*(x^*, X, y) & \sim \mathcal{N}(\mu^*, \sigma^2) \\
    \mu^* & \overset{\text{def}}{=} k^\ast (K + \sigma_n^2 I)^{-1} y \\
    \sigma^2 & \overset{\text{def}}{=} k(x^*, x^*) - k^\ast (K + \sigma_n^2 I)^{-1} k^\ast.
\end{align*}
\]

Computing \( \mu^* \) and \( \sigma^2 \) using these formulations gives the distribution of \( h(x^*) \). Inserting this into the model in eq. [2.71] results in the distribution

\[
p(y|x^*) = \mathcal{N}(\mu^*, \sigma^2 + \sigma_n^2).
\]

The \( \sigma_n^2 \)-term in the variance comes from uncertainty in \( h \) and the \( \sigma_n^2 \)-term from the additive Gaussian noise \( \epsilon \).

A key design aspect in the use of GPs is the choice of kernel function [17, p. 79]. The choice of kernel function encodes a notion of similarity, requiring some knowledge of the nature of the dataset. Another concern with the GP model is that the Gaussian assumption might not always hold. The standard GP regression model is only able to estimate the distribution of \( h^* \) and \( y \) as Gaussian, which might not always be the case for real data.

In the basic GP formulation the additive noise is assumed to have constant variance. The model can be extended to heteroskedastic regression according to eq. [2.72]. Goldberg et al. [17] propose such a formulation, using a second GP for the input-dependent noise variance. This results in a versatile regression model, but unlike basic GPs does not offer a closed form solution for the distribution of \( y \). Instead approximate sampling methods have to be employed.
2.5.3 Mixtures of Regression Models

To achieve more complex conditional distributions combinations of multiple regression models can be used in the form of mixtures. A general form for such a mixture is:

\[
p(y|\mathbf{x}) = \sum_{i=1}^{K} \pi_i q_i(y|\mathbf{x}),
\]

(2.82)

where \(\{\pi_i\}_{i=1}^{K}\) are so called mixture coefficients and the components \(\{q_i\}_{i=1}^{K}\) a set of regression models [9, p. 666-674]. To make the mixture represent a proper probability distribution the mixture coefficients must satisfy

\[
\sum_{i=1}^{K} \pi_i = 1.
\]

(2.83)

The mixture coefficients are estimated from data either together with the regression model parameters or separately. An extension called mixture of experts lets also the mixture coefficients be functions of \(\mathbf{x}\) [9, p. 672-674]. This means that each \(\pi_i\) needs to be represented by a regression model as well.

The regression models \(q_i\) can be for example linear models [9, p. 667-670] [24], GPs [46] or even more complex models. Mixture models are quite expressive and can model many different kinds of distributions. Using mixtures is therefore a good approach when noise in the data is heteroskedastic or multimodal [9, p. 672] [24].

2.5.4 Mixture Density Networks

Mixture Density Network (MDN) is a mixture model based on neural networks [8, p. 272-277]. The mixture formulation in eq. 2.82 is adopted with each \(q_i\) being a Gaussian and also the mixture coefficients being dependent on \(\mathbf{x}\):

\[
p(y|\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\mathbf{x}) \mathcal{N}(y|\mu_k(\mathbf{x}), \sigma_k^2(\mathbf{x}) I).
\]

(2.84)

In this formulation the covariance of each Gaussian component is assumed to be diagonal and equal for each dimension of \(y\). A more general formulation for the covariance matrix can be used to achieve a more expressive model.

\[
\sum_{k=1}^{K} \pi_k(\mathbf{x}) = 1
\]

(2.83)

In MDNs all mixture coefficients \(\{\pi_k(\mathbf{x})\}_{k=1}^{K}\), means \(\{\mu_k(\mathbf{x})\}_{k=1}^{K}\) and variances \(\{\sigma_k^2(\mathbf{x})\}_{k=1}^{K}\) are the outputs of a neural network that takes \(\mathbf{x}\) as input [8]. These parameters then define the model distribution as a mixture of Gaussians, as can be seen in figure 2.7. The constraint that the mixture coefficients must sum to 1 (eq. 2.83) can be satisfied by not using the network outputs directly. Instead \(K\) network outputs \(\{o_k(\mathbf{x})\}_{k=1}^{K}\) are fed through the softmax function:

\[
\pi_k(\mathbf{x}) = \frac{\exp(o_k(\mathbf{x}))}{\sum_{j=1}^{K} \exp(o_j(\mathbf{x}))},
\]

(2.85)

guaranteeing that the constraint holds. Similarly the non-negativity of variances can be assured by applying the exponential function to network outputs. An MDN can then be trained by gradient descent, minimizing the negative log of eq. 2.84 evaluated at the training data points.
2.5. Probabilistic Regression

2.5.5 Energy-based Regression

Another approach to probabilistic regression comes from the area of energy-based models. This class of models assume the conditional distribution of \( y \) to be a Gibbs distribution (also called Boltzmann distribution):

\[
p(y|x) = \frac{e^{-\frac{1}{T}E(x,y)}}{\int_{y' \in Y} e^{-\frac{1}{T}E(x,y')} dy'}.
\]  

(2.86)

This is a very general kind of distribution, since it is parametrized by a general function \( E: \mathcal{X} \times \mathcal{Y} \to \mathbb{R} \). The sets \( \mathcal{X} \) and \( \mathcal{Y} \) contain all possible values the variables \( x \) and \( y \) can take. The energy function can be viewed as a measure of compatibility between \( x \) and \( y \), with small energy values corresponding to highly compatible \((x, y)\). Known families of distributions can be recovered from the general Gibbs distribution formulation by selecting an appropriate energy function. The constant \( T \) is referred to as temperature (originating from statistical physics).

The integral \( Z(x) \equiv \int_{y' \in Y} e^{-\frac{1}{T}E(x,y')} dy' \) in the denominator of the Gibbs distribution is a normalizer. It is necessary to make the distribution properly integrate to 1. This integral is however often intractable or even impossible to compute. Much work has been put into finding useful ways to approximate its value.

Gustafsson et al. propose a specific energy-based regression model that incorporates neural networks. In their Deep Conditional Target Densities (DCTD) the model distribution is given by:

\[
p(y|x) = e^{v_\theta(x,y)}
\]

(2.87)

Here \( v_\theta(x,y) = -\frac{1}{T}E(x,y) \) is a neural network with parameters (weights and biases) \( \theta \). The network takes a pair of \( x \) and \( y \) as input and outputs a single scalar value, as can be seen in figure 2.8. The energy function and temperature are combined as the network output. The model is then trained by maximizing the log-likelihood for the training data. The log-likelihood for a single data point \((x, y)\) is given by:

\[
\log(p(y|x)) = v_\theta(x, y) - \log(\int_{y' \in Y} e^{v_\theta(x,y')} dy') = v_\theta(x, y) - \log(Z(x)).
\]

(2.88)

A problem with this is that the value of \( Z(x) \) can not be computed. This computation would involve an integral over a function represented by a neural network. Gustafsson et al. instead propose approximating \( Z(x) \) by using importance sampling.

Importance sampling is a sample-based method for approximating expectations or normalizing constants \[ p. 532-534 \]. Using this method, the constant \( Z(x) \) can be approximated as:

\[
Z(x) \approx \frac{1}{M} \sum_{m=1}^{M} \frac{e^{v_\theta(x,y^{(m)})}}{p_{\text{prop.}}(y^{(m)}|x)}
\]

(2.89)

\[
y^{(m)} \sim p_{\text{prop.}}(y|x) \quad \forall m \in \{1, \ldots, M\}.
\]

(2.90)
The $M$ samples $y^{(1)}, \ldots, y^{(M)}$ are all drawn from a proposal distribution $p_{\text{prop}}$. This is a separate distribution from $p$ and can be chosen somewhat freely. There are however some requirements on the proposal distribution $p_{\text{prop}}$ [2, p. 532-534]:

1. It must be possible to sample from $p_{\text{prop}}$.
2. The probability density function $p_{\text{prop}}(y|x)$ must be possible to evaluate, at least up to a normalizing constant.
3. The probability density of $p_{\text{prop}}$ should not be too small in areas where $p$ has a large concentration of probability mass.

Requirement 3 is central for achieving good estimates using importance sampling.

The DCTD model uses a mixture with equally weighted components as proposal distribution [23]. The mixture contains Gaussians with different variances, all centered on the true $y$ from the training set. Note that the estimation of $Z(x)$ is done for each $x$ separately and has to be repeated for each data point the model is trained on.

By using importance sampling to estimate $Z(x)$, the log-likelihood of the model becomes tractable. It should be noted that both $y$ from the training set and each sample from $p_{\text{prop}}$ needs to be propagated through the neural network. This means that each training data point requires $M + 1$ passes through the network. The computational cost of this can however be reduced somewhat by utilizing the fact that all samples are conditioned on the same $x$. Values in the network that only depend on $x$ only have to be computed once [23]. This can be very useful if $x$ is high-dimensional or of some complex data-type, like an image.

The fact that each sample $y^{(m)}$ also is propagated through the network means that these are also taken into account during training. Gradients with respect to the network parameters will be computed through both $v_\theta(x,y)$ and $Z(x)$. The value of $Z(x)$ depends on the network parameters through the importance sampling formulation in eq. 2.83. This gives an interpretation of the training as adjusting the network parameters such that:

- $v_\theta$ outputs a large value for the true $y$ from the training set,
- $v_\theta$ outputs a small value for all other $y$:

A secondary issue with the DCTD model formulation in eq. 2.87 is that there is no straightforward way to sample from $q$. Also here, importance sampling comes in handy. A procedure called sampling-importance-resampling can be used to approximately sample from $q$ [2, p. 534-536]. First, $N_{\text{prop}}$ samples $y^{(1)}, \ldots, y^{(N_{\text{prop}})}$ are drawn from the proposal distribution $p_{\text{prop}}(y|x)$. Using these, a set of importance weights are computed:

$$w^{(i)} \equiv \frac{1}{S} \frac{e^{v_\theta(x,y^{(i)})}}{p_{\text{prop}}(y^{(i)}|x)}, \quad \forall i \in \{1, \ldots, N_{\text{prop}}\}$$

(2.91)

Next, $N_{\text{sample}}$ $y$'s are drawn with replacement from the set $\{y^{(1)}, \ldots, y^{(N_{\text{prop}})}\}$. These are however not sampled uniformly from the set, but weighted by the importance weights. That is, the probability of drawing $y^{(i)}$ is $w^{(i)}$. The resulting $N_{\text{sample}}$ samples are approximately distributed according to $q(y|x)$. As with estimating $Z(x)$, this entire procedure depends on the specific value of $x$. It should also be noted that $N_{\text{prop}}$ should be chosen quite large to get good approximations for the samples.

Importance sampling is key to making the DCTD model trainable. It is however well known that the technique scales poorly to higher-dimensional $y$ [2, p. 532-540]. Getting good estimates in high dimensions either requires a very well-designed proposal distribution or drawing a very large number of samples. For the DCTD model, increasing the number of samples $M$ also increases the training time.
2.5.6 CGAN as a Regression Model

In contrast to models like GP that restrict \( p(y|x) \) to a known family of probability distributions, CGANs offer the possibility to approximate a class of much more general distributions. It is important to note that all CGANs can be seen as regression models, even when generating complex data like images [29]. Consider a CGAN generating images conditioned on some \( x \) (for example a class label or another image). If the generated image contains \( 28 \times 28 = 784 \) pixels it can be seen as \( y \in \mathbb{R}^{784} \) being a vector in some very high-dimensional space [32]. The vector \( y \) is still the regression target, but the model has to estimate a conditional distribution over 784 dimensions. Working in high-dimensional spaces can be qualitatively different than probabilistic regression with low-dimensional \( y \) [9, p. 33-38] [59]. Although the work on CGANs for high-dimensional \( y \) is relevant, the focus here will lie on CGAN regression with low-dimensional regression targets. For clarity, consider low-dimensional to be \( \leq 20 \) dimensions. This is the typical case for regression problems. The target variable \( y \) commonly just has one or a few dimensions.

Some initial work on examining CGAN as a regression model has been done by Aggarwal et al. [2]. They experiment on synthetic datasets sampled from heteroskedastic and multimodal distributions. Their work shows that while these properties can not be captured by simple models like standard GP regression, the CGAN model can successfully approximate the distributions. It has also been shown that CGANs are competitive with common regression models on some real world datasets. One restriction of the work by Aggarwal et al. is that only one-dimensional regression targets are considered. The experiments also only consider the standard CGAN formulation (eq. 2.20). Considering \( y \) of more than one dimension as well as CGANs with different training objectives are mentioned as useful future research.

CGAN regression has been applied in the context of some specific applications. Chapfuwa et al. [11] apply CGANs to time-to-event modelling, a special kind of regression problem where \( y \) is the time until an event occurs. The use of CGANs in their work is motivated by the knowledge that the distribution of \( y \) is complex and not well represented by limited, parametric families of distributions. Pang et al. [44] apply CGANs to predict aircraft trajectories while taking into account weather conditions. In their application both \( x \) and \( y \) are time series data. To take this into account both the generator and discriminator make use of neural network architectures specifically designed for this type of structured data. The resulting model is able to predict aircraft trajectories and estimate uncertainty in the prediction.

There exists a tangential body of work where adversarial losses are used for regression problems [68] [52] [21]. These approaches differ in two main ways:

1. The generator does not take any noise as input, only the conditioning variable \( x \).
2. The discriminator does not take \( x \) as input, only a generated or real \( y \).

From a probabilistic perspective this can be seen as an unconditional GAN, where the noise \( z \) is replaced by \( x \). The underlying distribution of \( x \) in the data then takes the role of the noise distribution \( p_z \). The resulting model tries to learn a generator such that the marginal generator distribution \( p_g(y) \) is close to the marginal distribution of the data \( p_d(y) \). In this case \( p_g \) is implicitly defined by \( y = G(x), \; x \sim p_d(x) \). Since there is no conditioning on \( x \) in \( p_g \) additional training mechanisms are used in order to make the generated \( y \)'s depend on \( x \) [68] [52]. This approach has been applied to image-to-image translation [68], refining synthetic data [52] and human motion prediction [21].

2.6 Evaluation

To evaluate a probabilistic regression model a measure is needed for how well it has approximated the true distribution. For this discussion let \( p_d(y|x) \) be the true conditional distribution.
of the data and \( q(y|x) \) the distribution represented by the model. For example, for a CGAN \( q(y|x) \) would be the generator distribution \( p_{\phi}(y|x) \). The goal in the probabilistic regression setting is to learn a model distribution as close as possible to \( p_d \). Since \( p_d(y|x) \) describes the data uncertainty, this is equivalent to learning a good model for this type of uncertainty. For a good model, \( q(y|x) \) properly describes the data uncertainty in the prediction for the given \( x \).

Since the true probability density function \( p_d(y|x) \) is generally unknown it can not be evaluated. Instead a set of samples \( \{(x^{(i)},y^{(i)})\}_{i=1}^{N_{\text{test}}} \) from \( p_d \) is available. If \( p_d(y|x) \) would be specified explicitly, such as when creating a synthetic dataset, it might be possible to both evaluate the true probability density and to draw samples from \( p_d \).

A useful way to evaluate probabilistic models is to compute the likelihood on a set of test data. The test set should not have been used neither for training nor validation. Due to the implicit probability density function of CGANs, computing the likelihood is however not straightforward. Section 2.6.1 will describe a way to estimate the probability density also for implicit models. The log-likelihood metric will then be described in detail in section 2.6.2 and alternative evaluation metrics presented in section 2.6.3.

### 2.6.1 Kernel Density Estimation

For probabilistic regression models that use a known family of distributions to estimate \( p_d \) the model probability density can usually be evaluated. For example, the model in eq. 2.71 results in \( q(y|x) = N(y|h(x),\sigma_y^2) \) [8, p. 140]. For generative models like CGAN, where only samples are available, there is no explicit expression for \( q(y|x) \). \( q \) can be approximated by a set of samples from a generative model by the use of Kernel Density Estimation (KDE) [8, p. 122-124] [18]. The technique is also called the Parzen window method. KDE makes use of kernel functions to encode a notion of similarity. Commonly the Gaussian kernel is used for density estimation [8, p. 123]:

\[
k(y,y') = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp \left( -\frac{\|y-y'\|^2}{2\sigma_k^2} \right), \tag{2.93}
\]

where \( \sigma_k^2 \) is a tunable scale parameter. Using this kernel, an estimate of the model probability density can be computed as:

\[
q(y|x) = \frac{1}{N_{\text{kde}}} \sum_{i=1}^{N_{\text{kde}}} \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp \left( -\frac{\|y_i-y\|^2}{2\sigma_k^2} \right), \tag{2.94}
\]

where \( \{y_i\}_{i=1}^{N_{\text{kde}}} \) is a set of samples generated by the model of interest when conditioned on \( x \). See figure 2.9 for an example.

![Figure 2.9](image)

Figure 2.9: Approximating a probability density function from samples using KDE. (a) A set of 5 samples \( y_1, \ldots, y_5 \). (b) Gaussian kernels (eq. 2.93) centered at the samples. (c) Probability density estimate \( q(y|x) \) computed according to eq. 2.94.
2.6. Evaluation

2.6.2 Log-Likelihood

A common metric for evaluating probabilistic regression models is the log-likelihood of test data. Goodfellow et al. \[18\] use log-likelihood to evaluate GANs and Aggarwal et al. \[2\] apply it to CGANs. In the CGAN case the conditional log-likelihood is used. Typically this is still only referred to as log-likelihood, making it implicit that a conditional distribution is considered. The log-likelihood is calculated as:

\[
\mathcal{L} \overset{\text{def}}{=} \frac{1}{N} \sum_{i=1}^{N} \log \left( q(y^{(i)}|x^{(i)}) \right),
\]  

(2.95)

on a test set \(\{(x^{(i)}, y^{(i)})\}_{i=1}^{N_{\text{test}}}\) from \(p_d\). For generative models KDE can be used to estimate \(q\). Using log-likelihood estimates based on KDE has shown to be a poor evaluation measure for GANs \[59\]. These results do however assume that the generated samples are high-dimensional (for example images). For low-dimensional \(y\), as is the focus of this thesis, the issues with KDE log-likelihood estimation are not as severe. It should also be noted that the use of KDE-estimated log-likelihood is purely for getting a quantitative evaluation metric. When using an implicit generative model in practice to generate samples, there is no reason to estimate a probability density function.

Evaluation using log-likelihood can intuitively be seen as a test for how close the model distribution \(q(y|x)\) is to the data distribution \(p_d(y|x)\). If \(\log (q(y^{(i)}|x^{(i)})\) is high, the data is likely to come from the model distribution. Since the data is known to be from \(p_d\), this indicates that the two distributions are similar. The log-likelihood evaluation metric is additionally connected to the Kullback-Leibler divergence \[9\], p. 55-58]. A model distribution achieving high log-likelihood values is equivalent to \(D_{\text{KL}}(p_d(y|x)\|q(y|x))\) being small.

2.6.3 Alternative Evaluation Metrics for Generative Models

How to properly evaluate generative models that do not have an explicit likelihood function is a hard problem that does not have a clear solution \[59\]. The best choice of evaluation metric often depends on the application considered. In image generation, models can be evaluated by letting human observers rate the quality of samples \[51\]. Manual inspection can however be both subjective and time consuming. Evaluation metrics have also been proposed that make use of auxiliary neural networks, pretrained for some discriminative task. These include the Inception Score \[51\] and Fréchet Inception Distance \[27\]. The idea behind these approaches is to compare the values of learned feature representations. Samples from a good generative model should give similar features as the real data.

Im et al. \[28\] propose using the GAN objective directly as a metric of how well the data distribution has been estimated. This type of evaluation can be done for any generator \(G\) by retraining a new discriminator based on real samples and samples generated by \(G\). The value of the training objective \(V(D,G)\) on a separate test set, also consisting of both real and generated samples, is then used as the metric. Utilizing the theoretical framework of \(f\)-GANs, the metric can be interpreted as an estimate of some \(f\)-divergence between the data and generator distributions. Different \(f\)-divergences can be evaluated by using the corresponding formulation of the GAN objective, \(V_f(T,G)\). The metric can also be used for GAN objectives that do not have an interpretation as \(f\)-divergences. Note that the choice of divergence in the metric is completely separate from how the model has been trained. The model being evaluated does not even have to be a GAN. As long as samples can be generated from the model distribution a discriminator can be trained for evaluating this type of metric. On an image generation task, Im et al. \[28\] compare the metric formulations based on different GAN training objectives. They find that the metrics based on the training objectives of Least Square GAN \[18\] and Wasserstein GAN \[4\] best match human perception of sample quality. This evaluation procedure can readily be extended to the case of CGANs.
To investigate the capabilities of CGANs to approximate different data distributions a number of experiments were carried out. Different CGAN models were implemented, corresponding to different training objectives. These are presented in section 3.1. Also a number of baseline models were implemented for comparison. The baseline models used are described in section 3.2. To get an understanding of how CGANs behave for distributions with different properties, multiple synthetic datasets were created. All datasets used in experiments are presented in section 3.3. Evaluation metrics used are described in section 3.4. Details on individual experiments are given in the corresponding section in the Experiments chapter.

All models and experiments were implemented in Python 3.7. Models based on neural networks were implemented using PyTorch 1.4. Code and datasets are available in an online repository.

3.1 CGANs

A number of different CGANs were implemented, corresponding to different $f$-divergences and MMD. Note that the structure and size of the generator and discriminator neural networks are not impacted by the choice of divergence. The only differences lies in the loss functions used in the training process.

3.1.1 Network Architectures

Multiple different generator and discriminator networks were implemented with different architectures and sizes. The first type of architecture used was named double-input. The idea behind double-input was to use some initial layers to learn useful features for each network input separately. These would then be concatenated to let later layers learn joint representations. The double-input generator and discriminator networks are shown in figure 3.1. The exact sizes of the networks are listed in table 3.1. Activation functions were applied after each layer except the last. See section 2.1.2 for more details about activation functions. If nothing else is mentioned for a particular experiment Leaky ReLU was used (eq. 2.6).
Table 3.1: Exact sizes of the double-input networks used in CGANs. *First two layers 128 units, followed by three layers of 64 units.

Figure 3.1: Double-input neural network architectures used in CGANs.
In the generator network $x$ is fed through $l_{x,G}$ layers of $u_{x,g}$ units each. $z$ is fed through $l_{z,G}$ layers of $u_{z,G}$ units. The resulting vector representations are then concatenated to a vector of dimension $u_{x,G} + u_{z,G}$. This vector is fed through $l_{y,G}$ layers of $u_{y,G}$ units to create the generated $y$ as output.

Also in the discriminator $x$ is fed through $l_{x,D}$ layers of $u_{x,D}$ units. The input $y$ is also propagated separately through $l_{y,D}$ layers with $u_{y,D}$ units. The resulting vectors for $x$ and $y$ are concatenated and fed through $l_{d,D}$ layers of $u_{d,D}$ units, creating a final one-dimensional output. This discriminator output, that does not yet have any activation function applied, corresponds to $T$ in the $f$-GAN framework.

**An alternative architecture, named noise-injection, was also considered for the generator.** In the noise-injection generator the noise $z$ is not fed through its own group of network layers. Instead the noise vector is concatenated to the hidden representation at each layer in the network. This means that each layer takes as input the output from the previous layer concatenated with the noise vector, as can be seen in figure 3.2. ReLU activations were used everywhere except after the last layer.

![Figure 3.2: The noise-injection generator network architecture](image)

The idea behind the noise injection architecture was to let the generator learn how early in the network the noise should be included. If the noise needs to go through some complex transformation it can be included already in the first layer. If the data distribution is well modelled by just additive Gaussian noise the network can instead simply include the noise in the last layer. There is no clear reason for also using the noise-injection architecture for the discriminator, so in experiments a noise-injection generator was used together with a double-input discriminator.

The noise-injection generator network contains $l_{ni}$ hidden layers of $u_{ni}$ units each. It was also implemented in different sizes, listed in table 3.2 together with the sizes of corresponding discriminators.

<table>
<thead>
<tr>
<th>Generator</th>
<th>Discriminator</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_{ni} \times u_{ni}$</td>
<td>$l_{x,D} \times u_{x,D}$</td>
</tr>
<tr>
<td>small</td>
<td>4 × 16</td>
</tr>
<tr>
<td>medium</td>
<td>6 × 64</td>
</tr>
<tr>
<td>large</td>
<td>8 × 128/64*$^*$</td>
</tr>
</tbody>
</table>

Table 3.2: Exact sizes of the noise-injection networks used in CGANs. *128 units in the first two layers, 64 units in the rest.
3.1.2 Loss Functions

For the CGAN training loss functions were formulated for both the generator and discriminator. This will be presented first for a general $f$-divergence and then specified for specific choices of $f$. The MMD-based loss used in GMMNs is treated separately in section 3.1.3.

Let $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N_{\text{train}}}$ be the training dataset. The discriminator network parameters will still be denoted $\theta_D$, but the reformulation $D(y|x) = g_f(T(y|x))$ is used. Starting from eq. (2.44) in the theory:

$$V_{c,f}(T, G) = E_{x \sim p_x(x)} \left[ E_{y \sim p_y(y|x)} \left[ g_f(T(y|x)) \right] + E_{y \sim p_y(y|x)} \left[ -f^*(g_f(T(y|x))) \right] \right]$$

(3.1)

$$= \frac{1}{N_{\text{train}}} \sum_{j=1}^{N_{\text{train}}} \left[ g_f \left( T(y^{(j)}|x^{(j)}; \theta_D) \right) - (f^* \circ g_f) \left( T \left( G(z^{(j)}|x^{(j)})|x^{(j)}; \theta_G \right) \right) \right].$$

(3.2)

Batching was applied over the outermost sum, splitting the $N_{\text{train}}$ samples into batches of size $N_{\text{batch}}$. For the discriminator $V_{c,f}(T, G)$ should be maximized, so it needs to be negated to be turned into a loss function. The expectation over $p_z$ was approximated by drawing one noise sample $z^{(j)}$ for each $x^{(j)}$. All of this allows for defining the discriminator CGAN loss:

$$L_D(\theta_D) \overset{\text{def}}{=} -\frac{1}{N_{\text{batch}}} \sum_{j=1}^{N_{\text{batch}}} \left[ g_f \left( T(y^{(j)}|x^{(j)}; \theta_D) \right) - (f^* \circ g_f) \left( T \left( G(z^{(j)}|x^{(j)})|x^{(j)}; \theta_G \right) \right) \right].$$

(3.3)

For the generator loss function the alternative formulation presented in section 2.3.6 was used for all $f$-divergences, as suggested by Nowozin et al. [32]. By similar reasoning as above, the generator loss was formulated based on eq. (2.51), resulting in:

$$L_G(\theta_G) \overset{\text{def}}{=} -\frac{1}{N_{\text{batch}}} \sum_{j=1}^{N_{\text{batch}}} \left[ g_f \left( T \left( G(z^{(j)}|x^{(j)})|x^{(j)}; \theta_G \right) \right) \right].$$

(3.4)

With the loss functions formulated for general $f$-divergences, it is clear that the only difference between different $f$ is in $g_f$ and $(f^* \circ g_f)$. A number of different $f$-divergences were selected and corresponding CGANs implemented. The implementation for each CGAN consisted only of defining the correct $g_f$ and $(f^* \circ g_f)$ for the particular $f$. Table 3.3 lists the different divergences used.

<table>
<thead>
<tr>
<th>Divergence</th>
<th>$g_f(v)$</th>
<th>$(f^* \circ g_f)(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>$\log(\sigma(v))$</td>
<td>$-\log(1 - \sigma(v))$</td>
</tr>
<tr>
<td>KL</td>
<td>$v$</td>
<td>$\exp(v - 1)$</td>
</tr>
<tr>
<td>Reverse KL</td>
<td>$-\exp(v)$</td>
<td>$-1 - v$</td>
</tr>
<tr>
<td>Jensen-Shannon*</td>
<td>$\frac{1}{2} \log(2) + \log(\sigma(v))$</td>
<td>$-\frac{1}{2} \log(2) - \frac{1}{2} \log(1 - 2\sigma(v)^2)$</td>
</tr>
<tr>
<td>Pearson $\chi^2$</td>
<td>$v$</td>
<td>$v(\frac{1}{2} v + 1)$</td>
</tr>
<tr>
<td>Neyman $\chi^2$</td>
<td>$1 - \exp(v)$</td>
<td>$2 \left( 1 - \exp\left(\frac{v}{2}\right) \right)$</td>
</tr>
<tr>
<td>Squared Hellinger</td>
<td>$1 - \exp(v)$</td>
<td>$-1 - \exp(-v)$</td>
</tr>
</tbody>
</table>

Table 3.3: $f$-divergences used to define CGAN models. The table is adapted from Nowozin et al. [32]. The sigmoid function is defined $\sigma(v) \overset{\text{def}}{=} (1 + \exp(-v))^{-1}$. *Note that the standard GAN formulation results in a slightly modified Jensen-Shannon divergence, as can be seen in eq. (2.28). Because of this, the version corresponding exactly to $D_{JS}$ was also included.
In addition to the different $f$-GANs a Least Squares CGAN model was also implemented. Although the Least Squares CGAN has no direct connection to $f$-divergences, the loss functions used still follow the formats of eq. 3.3 and eq. 3.4. The least squares training objective can be recovered by setting $g_f(v) = -\frac{1}{2}(v - 1)^2$ and $(f^* \circ g_f)(v) = \frac{1}{2}v^2$.

### 3.1.3 GMMN

For an MMD-based CGAN, the joint GMMN approach presented in section 2.4.4 was used. The generator was trained to minimize eq. 2.70 for each batch of samples. Neural network architectures for the GMMN were exactly the same as for other CGANs, but no discriminator network was needed.

The kernel used for MMD was a composition of a simple feature transformation and a Gaussian kernel:

$$k([x; y]\top, [x'; y']\top) = \sigma^2_k k([x; y]\top, [x; y]\top) \defeq \text{diag}(\rho)[x; y]\top.$$  \hspace{1cm} (3.5)

Here $\rho$ is a vector that scales each component in $x$ and $y$ before the Gaussian kernel is applied. Since $\xi_\rho$ already performs component-wise scaling, $\sigma^2_k$ was just fixed to 1. The vector $\rho$ was learned during training. This can be viewed either as a very simple version of deep kernel learning or just as a way to automatically tune the scale parameter of the Gaussian kernel. The simple transformation $\xi_\rho$ is injective, making sure the kernel stays characteristic (see section 2.4.3). Gradient descent was used to minimize $\hat{-\text{MMD}}^2$ with respect to $\rho$. One step was taken each batch, at the point where the discriminator would be trained in other CGAN models. All computations of the MMD and kernel values were done batch-wise, utilizing fast matrix operations on the GPU.

### 3.1.4 Training Setup

For each batch a single gradient descent step was taken both for the discriminator and generator. A batch size of $N_{\text{batch}} = 100$ was used. Both the Adam and RMSProp optimizers have been used in the literature for training GANs \[42\] \[38\]. After some small initial tests the RMSProp optimizer was chosen for training the CGAN models. Adam was however used for the GMMN, since it showed good performance. Network parameters in all layers were initialized by sampling from a uniform distribution over $(-1/\sqrt{k}, 1/\sqrt{k})$ where $k$ is the dimensionality of the input to the layer.

MMD requires a larger batch size, so $N_{\text{batch}} = 200$ was used for GMMN. Learning rate for the kernel parameters was fixed at 0.01. The vector $\rho$ was initialized with each component set to 1.

The noise $z$ for each batch was sampled within the training loop. Unless explicitly mentioned otherwise, the noise distribution used was an isotropic standard Gaussian. This choice of distribution has proven useful both in existing literature \[39\] and in initial experiments. Noise dimensionality was varied between experiments to match the complexity of the current dataset.

### 3.2 Baseline Models

In addition to the CGANs a number of baseline models were implemented. The baseline models were chosen to have different properties to allow for useful comparisons with CGANs. In particular the kinds of conditional distributions that can be represented by each model is a key property. For a fair comparison some of the baseline models were chosen to have a similar capacity as the CGANs. This was achieved by using models based on neural networks of similar size as the CGAN generator.
3.2. Baseline Models

3.2.1 Gaussian Process Regression

A Gaussian process model was used as a simple baseline that can model homoskedastic Gaussian noise. GPs do not have an iterative training process in the same way as neural networks. Instead the computation of \( p(y|x) = \mathcal{N}(y|\mu(x), \sigma(x)I) \) at test time is a more involved process, as described in section 2.5.2. This process includes using the training data and chosen kernel function to determine covariances.

Four different kernel functions were considered for each experiment using GPs. These are listed in table 3.4. In each experiment separate GPs were fitted using each kind of kernel and evaluated on the validation data set. The best performing kernel was then used for the final GP model that would be evaluated on the test set. The kernels themselves contain hyperparameters that need to be selected appropriately. Since all chosen kernels are differentiable this could be done automatically through an optimization process. Kernel hyperparameters \( \theta_k \) were optimized by maximizing the log-marginal-likelihood \( p(y|x, \theta_k) \) of the training data \([47], \text{p. } 112-116\) . The L-BFGS-B optimization algorithm \([10]\) was used with 10 restarts at random initializations.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Hyperparameters ( \theta_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radial-basis function</td>
<td>( l &gt; 0 )</td>
</tr>
<tr>
<td>Matern</td>
<td>( l &gt; 0, \nu = 1.5^* )</td>
</tr>
<tr>
<td>Rational Quadratic</td>
<td>( l &gt; 0, \alpha &gt; 0 )</td>
</tr>
<tr>
<td>Exp-Sine-Squared</td>
<td>( l &gt; 0, p &gt; 0 )</td>
</tr>
</tbody>
</table>

Table 3.4: Kernels used for the GP model. See Rasmussen et al. \([47], \text{p. } 81-95\) for more complete descriptions. The hyperparameter \( \nu \) was kept fixed to 1.5 and not optimized.

To take into account the noise in the data, also the noise variance \( \sigma^2_n \) had to be estimated. It was treated as another hyperparameter and optimized together with the kernel hyperparameters. This could be done by changing the kernel used from \( k(x,x') \) to

\[
\tilde{k}(x,x') = k(x,x') + \sigma^2_n I_{\{x=x'\}},
\]

which is also a valid kernel. Here \( I_{\{\cdot\}} \) is the indicator function, taking the value 1 if the given condition is true and 0 otherwise.

The GP model was implemented using the `gaussian_process` module of the scikit-learn library.

3.2.2 Neural Network Regression

Another baseline model was a standard neural network used for regression. The network takes \( x \) as input and tries to predict \( y \). The probabilistic perspective of this model corresponds to assuming fixed homoskedastic additive Gaussian noise. The model distribution is then

\[
p(y|x) = \mathcal{N}(y|\mu(x), \text{diag}(\sigma^2_1, \sigma^2_2, \ldots, \sigma^2_{d_y})),
\]

with \( \mu(x) \) being the network output and \( d_y \) the dimensionality of \( y \). The noise dimensions were modelled as independent, resulting in a diagonal covariance matrix. The network was trained

\[ \text{https://scikit-learn.org} \]
to maximize the log-likelihood for the training data \( \{(x^{(i)}, y^{(i)})\}_{i=1}^{N_{\text{train}}} \). This corresponds to minimizing the Mean Squared Error (MSE) loss function:

\[
L_{\text{MSE}}(\theta) \overset{\text{def}}{=} \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \| \mu(x^{(i)}) - y^{(i)} \|^2_2
\]

with respect to the network parameters \( \theta \). The log-likelihood would theoretically also include a scaling factor depending on the variances in the model distribution. This factor can however be baked into the learning rate, with the small caveat that the noise variance for each dimension then is assumed to be equal. This approximation decouples the training of the network from estimating the noise variances.

Since the noise variances \( \sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2 \) were assumed constant and not dependent on \( x \) they could be determined separately from the network training. With the network parameters fixed, this was again done by maximum likelihood. This results in the following closed form expression:

\[
\sigma_j^2 = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left( \mu(x^{(i)})_j - y_j^{(i)} \right)^2 \quad \forall j \in \{1, \ldots, d_y\}. \tag{3.10}
\]

A set of neural networks of different sizes were used. Each network consisted of \( l_{\text{nn}} \) layers of \( u_{\text{nn}} \) units each. Exact sizes of the different networks are listed in table 3.5. The number of layers and units were chosen to match the capacity of corresponding CGAN generators. ReLU activation functions were used everywhere in the networks, except for the output layer. No activation function was used for the output.

<table>
<thead>
<tr>
<th>Network</th>
<th>( l_{\text{nn}} )</th>
<th>( u_{\text{nn}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>medium</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>large</td>
<td>7</td>
<td>128/64*</td>
</tr>
</tbody>
</table>

Table 3.5: Exact sizes of the different neural networks used for baseline models. * First two layers of 128 units, followed by five layers of 64 units.

The neural network regression model was trained using stochastic gradient descent and the Adam optimizer [31]. The batch size was set to 100 and \( L_2 \)-regularization was applied to prevent overfitting.

### 3.2.3 Heteroskedastic Neural Network Regression

Also a heteroskedastic version of neural network regression was implemented. This model predicts both mean and variance for each \( x \), corresponding to the model distribution:

\[
p(y|x) = \mathcal{N}(y|\mu(x), \text{diag}(\sigma_1^2(x), \sigma_2^2(x), \ldots, \sigma_{d_y}^2(x))). \tag{3.11}
\]

The variance \( \sigma_j^2(x) \) for each dimension \( j \) needs to be positive. This was enforced by not predicting the variance directly, but instead letting the network estimate \( \ln(\sigma_j^2) \). The variance could then easily be computed as \( \sigma_j^2(x) = \exp(\sigma_j^2(x)) \). This trick has been widely used in the literature [26] [28]. The resulting network has \( 2d_y \) output units, \( d_y \) for predicting \( \mu(x) \) and \( d_y \) for \( \sigma(x) \)

\[
\alpha(x) \overset{\text{def}}{=} \begin{bmatrix} \alpha_1(x), \alpha_2(x), \ldots, \alpha_{d_y}(x) \end{bmatrix}^T.
\]

For the heteroskedastic neural network model the same networks and training procedure were used as for the homoskedastic version. The difference lies in the number of network outputs and the loss function being minimized. In the heteroskedastic case both mean and variances are estimated by the network and needs to be part of the loss function. Maximizing the log-likelihood for the training dataset in this case results in the following loss function:

\[
L_{\text{Het}}(\theta) \overset{\text{def}}{=} \frac{1}{2N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \sum_{j=1}^{d_y} \left[ \alpha_j(x^{(i)}) + \ln(\sigma_j(x^{(i)})) \left( \mu(x^{(i)})_j - y_j^{(i)} \right)^2 \right]. \tag{3.12}
\]
See appendix A for an explanation of how this loss function can be derived.

### 3.2.4 Mixture Density Network

To allow for estimating distributions beyond single Gaussians an MDN baseline was implemented. The MDN model follows the presentation from section 2.5.4 with $K$ mixture components. A more general form was however adopted for the covariance matrices. A separate variance was predicted for each dimension of $y$, as in eq. 3.11. The same reformulation of $\alpha = \log(\sigma^2)$ was also utilized.

The general training setup and network architectures for the MDN model were the same as for the other neural network models. For the MDN model the network was given $2d_yK + K$ output units. Of these, $K$ were used for representing the mixture coefficients, $d_yK$ for the $K$ different mean vectors and another $d_yK$ for variances. The loss function used to train MDNs was:

$$\mathcal{L}_{MDN}(\theta) \overset{\text{def}}{=} -\frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \log \left( \sum_{k=1}^{K} \pi_k \left( x^{(i)} \right) \mathcal{N} \left( y^{(i)} \mid \mu_k \left( x^{(i)} \right), \Sigma_k \left( x^{(i)} \right) \right) \right),$$  \quad (3.13)

where

$$\Sigma_k \left( x^{(i)} \right) \overset{\text{def}}{=} \text{diag} \left( \sigma_{k,1}^2 \left( x^{(i)} \right), \sigma_{k,2}^2 \left( x^{(i)} \right), \ldots, \sigma_{k,d_y}^2 \left( x^{(i)} \right) \right)$$  \quad (3.14)

$$= \text{diag} \left( \exp \left( \alpha_{k,1} \left( x^{(i)} \right) \right), \exp \left( \alpha_{k,2} \left( x^{(i)} \right) \right), \ldots, \exp \left( \alpha_{k,d_y} \left( x^{(i)} \right) \right) \right).$$  \quad (3.15)

### 3.2.5 Energy-based DCTD Model

Deep Conditional Target Densities \cite{li2017dctd} was implemented as an energy-based baseline model. The model is presented in detail in section 2.5.2. In DCTD, a neural network $v_\theta$ is trained by minimizing the following loss function:

$$\mathcal{L}_{DCTD}(\theta) \overset{\text{def}}{=} \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left[ -v_\theta \left( x^{(i)}, y^{(i)} \right) + \log \left( \frac{1}{M} \sum_{m=1}^{M} e^{v_\theta \left( x^{(i)}, y^{(i)} \mid y \right)} \right) \right],$$  \quad (3.16)

for a dataset $\{ \left( x^{(i)}, y^{(i)} \right) \}_{i=1}^{N_{\text{train}}}$. The details of the training process are the same as for the other neural network baselines.

The log-term in eq. 3.16 contains the importance sampling estimate of the normalizing constant. Each sample $y^{(i,m)}$ was drawn from the proposal distribution $p_{\text{prop}} \left( y \mid x^{(i)} \right)$. For training $M = 1000$ samples were used and at test time $M = 5000$. The proposal distribution was a mixture of three isotropic Gaussians with variances 0.5, 1 and 5. Each Gaussian was centered at $y^{(i)}$, the ground truth $y$-value.

Note that the neural network $v_\theta$ takes both $x$ and $y$ as input. To allow for this, the network architecture used matches the discriminator architecture used for CGANs (see figure 3.1b). Exact network sizes matches those given for the noise-injection discriminators in table 3.2. In this architecture there are few intermediate values that only depend on $x$. The speed-up method of computing these only once for each $x$ (instead of recomputing them for each $y$) was therefore not implemented.

A complication with the DCTD model is what to use for proposal distribution at test time. There is no known true value for $y$ to center the mixture of Gaussians on. One idea is to use the most likely prediction, $y^* = \arg \min_y -v_\theta(x,y)$, in its place. This optimization can be solved approximately by performing gradient descent with respect to $y$ instead of the model parameters $\theta$. This is however not always a convex optimization problem, so it is likely to converge only to a local optimum. By rerunning the optimization from multiple starting points it is more likely that a global optimum is found. A local optimum is however a mode of the model distribution, so performing gradient descent once only was still deemed a useful method. In summary, the proposal distribution at test time was constructed as:
3.3. Datasets

1. Start at $y = 0$

2. Perform 100 gradient descent steps with respect to $y$, minimizing $-v_{\theta}(x, y)$. Use the Adam optimizer with a learning rate of 0.01.

3. Center the mixture of Gaussians on the optimized $y$-value to get a proposal distribution.

This process has to be performed for each $x$ separately, resulting in different proposal distributions. By batching together multiple $x$s from the test set, the entire optimization process for $y$ can however be performed concurrently. Utilizing GPU computation makes the method very fast in practice.

3.2.6 Hyperparameter Tuning

All models have some associated hyperparameters that change the model itself and its training process. To get fair comparisons between models, hyperparameters needed to be chosen appropriately in each experiment.

A two-step process was used for finding suitable hyperparameter values. For each dataset, some initial experiments were used to determine reasonable intervals for the hyperparameter values. These initial experiments mainly consisted of shorter training runs with manually set parameters. Results from these are not reported.

Based on the initial observations, a single value or a set of useful values was chosen for each hyperparameter. In this second step, a more principled approach was taken. A grid search was performed over the chosen sets, training the model once using each possible configuration of hyperparameter values. For example, when training the MDN model it might be useful to try values for $K$ in $(3, 5, 10)$ and learning rates in $(0.01, 0.001)$. A grid search over these parameters would result in the six configurations: $(3, 0.01)$, $(3, 0.001)$, $(5, 0.01)$, $(5, 0.001)$, $(10, 0.01)$ and $(10, 0.001)$. The hyperparameter configuration that achieved the highest log-likelihood on the validation data was then chosen for that model and dataset. This procedure was used both for baseline models and CGANs.

3.3 Datasets

This section presents the different datasets used in experiments. This includes both synthetic datasets, generated to test specific model properties, and real world datasets.

3.3.1 Small Synthetic Datasets

A number of small synthetic datasets with different properties were generated. These all used a one-dimensional $x$ and a small number of training samples. This allowed for easily visualizing the data to study the behaviour of the models. Each dataset was defined by a distribution or a random data-generating process. For each set 1000 samples were generated for training, 1000 for validation and 1000 for testing. The following is a brief description of each dataset in this group:

- **const_noise**: A simple dataset described by a linear model with additive Gaussian noise.
- **laplace**: A linear model but with additive laplacian noise. The input $x$ is addition-ally Gaussian instead of uniformly distributed.
- **exponential**: A quadratic model with additive exponential noise. The noise variance changes with $x$.
- **butterfly**: A dataset generated by passing both $x$ and sampled Gaussian noise through a complex non-linear function.
heteroskedastic A heteroskedastic Gaussian model where the variance is a non-linear function of x.

bimodal Two component conditional mixture with uniformly distributed noise.

complex A more complex dataset where the data is unimodal but heteroskedastic for negative x and multimodal for positive x.

swirls A conditional Gaussian mixture model with two-dimensional y, where both means and variances of the components depend on x. For this dataset 2000 samples were used for training.

In all datasets except swirls both x and y are one-dimensional. Figure 3.3 shows the test set of generated samples for these datasets. The test samples from swirls are plotted in 3 dimensions in figure 3.4. For a detailed description of how the data was generated for each set see appendix C.

![Generated test sets for each one-dimensional synthetic dataset](image)

Figure 3.3: Generated test sets for each one-dimensional synthetic dataset

3.3.2 trajectories

To better investigate model performance for different dimensionality of y the synthetic trajectories datasets were generated. Each of these was generated by simulating a two-dimensional movement trajectory according to a random process parametrized by x. The vector y is then the concatenated coordinates of each step in the trajectory. Adjusting the
3.3. Datasets

![Graph of generated test set for the swirls synthetic dataset](image)

---

Figure 3.4: Generated test set for the **swirls** synthetic dataset

The number of steps $s$ in the simulation thereby results in different dimensionality of $y$. While there is a sequential structure to $y$, this is not the key property of the dataset. The use of a step-wise random process is mainly just a convenient way to achieve complex high-dimensional distributions for $y$.

The entries in $x$ correspond to the parameters of the random process that generates $y$. These are listed in table 3.6, together with the distribution they were sampled from.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_{\text{init}} = [w_{\text{init},1}, w_{\text{init},2}]^\top$</td>
<td>Starting position</td>
<td>$\mathcal{N}(0, I)$</td>
</tr>
<tr>
<td>$l$</td>
<td>Step length</td>
<td>$\mathcal{U}(0.25s^{-1}, s^{-1})$</td>
</tr>
<tr>
<td>$\theta_{\text{init}}$</td>
<td>Initial orientation</td>
<td>$\mathcal{U}(-\pi, \pi)$</td>
</tr>
<tr>
<td>$t_{\text{max}}$</td>
<td>Maximum turning</td>
<td>$\mathcal{U}(0.2, 0.5)$</td>
</tr>
</tbody>
</table>

Table 3.6: Parameters of the random process for generating $y$ in trajectories datasets. The step length depends on the number of simulation steps, $s$. The distribution $\mathcal{U}(a,b)$ is the continuous uniform distribution over the range $[a,b]$.

The $x$-vector contains all of these parameters:

$$x = [w_{\text{init},1}, w_{\text{init},2}, l, \theta_{\text{init}}, t_{\text{max}}]^\top \in \mathbb{R}^5.$$ (3.17)

Based on $x$, the following discrete random process can then be simulated:

$$\theta_0 = \theta_{\text{init}}$$ (3.18)

$$w_0 = w_{\text{init}}$$ (3.19)

$$\Delta^\theta_i \sim \mathcal{U}(-t_{\text{max}}\pi, t_{\text{max}}\pi)$$ (3.20)

$$\theta_i = \theta_{i-1} + \Delta^\theta_i$$ (3.21)

$$\Delta^w_i = l[\cos(\theta_i), \sin(\theta_i)]^\top$$ (3.22)

$$w_i = w_{i-1} + \Delta^w_i.$$ (3.23)

for steps $i \in \{1, \ldots, s\}$. The $y$-vector is constructed as

$$y = [w_{1,1}, w_{1,2}, w_{2,1}, w_{2,2}, \ldots, w_{s,1}, w_{s,2}, \ldots]^\top \in \mathbb{R}^{2s}.$$ (3.24)
Examples of resulting trajectories can be seen in figure 3.5.

Figure 3.5: 20 sampled trajectories from trajectories datasets with 5 and 10 steps. All the trajectories in each plot were generated based on the same \( \mathbf{x} \). Note the difference in smoothness and starting positions between trajectories in plot (a) and plot (b). This is due to different \( \mathbf{x} \) parametrizing the generating random process.

Based on this framework, 20000 \((\mathbf{x}, \mathbf{y})\) pairs were sampled as training dataset. For the validation and testing sets 4000 samples were drawn. Trajectories datasets were generated for \( s = 2, 5 \) and 10 steps, corresponding to 4, 10 and 20-dimensional \( \mathbf{y} \).

### 3.3.3 \texttt{wmix}

Complementing the \texttt{trajectories} datasets, synthetic datasets were also generated for different dimensionality of \( \mathbf{x} \). For these, \( y \) was fixed to be one-dimensional. The \texttt{wmix} datasets were constructed as mixtures of Weibull distributions. The conditioning variable \( \mathbf{x} \) contains all the mixture parameters and \( y \) is a sample from the mixture. Datasets with different dimensionality for \( \mathbf{x} \) could then be generated by varying the number of mixture components.

The Weibull distribution \([63][65]\), denoted \texttt{Weibull}(\(\lambda, k\)), is parametrized by a scale parameter \(\lambda\) and a shape parameter \(k\). The scale parameter stretches the distribution over the real axis and the shape parameter manipulates the overall shape of the distribution. The Weibull probability density function is:

\[
p(y) = \begin{cases} 
\lambda k (\lambda y)^{k-1}e^{-(\lambda y)^k} & \text{if } y > 0 \\
0 & \text{if } y \leq 0
\end{cases} \tag{3.25}
\]

For \( k = 1 \) the distribution reduces to an exponential distribution. Figure 3.6 shows some examples of Weibull distributions with different shape parameters. The shape parametrization was the motivation for using a Weibull distributions in the \texttt{wmix} datasets, instead of something more common like Gaussians. It allows for creating more complex conditional distributions.

Each component in the Weibull mixture is associated with three parameters: an offset \(\omega_i\), a component weight \(\tilde{w}_i\), and a shape modifier \(\tilde{k}_i\). Each of these is sampled from the uniform distribution \(U(0, 1)\). The subscript \(i\) indexes the mixture component. The conditioning vector \(\mathbf{x}\) can then be constructed as:

\[
\mathbf{x} = \left[ \omega_1, \omega_2, \ldots, \omega_{n\text{comp}}, \tilde{w}_1, \tilde{w}_2, \ldots, \tilde{w}_{n\text{comp}}, \tilde{k}_1, \tilde{k}_2, \ldots, \tilde{k}_{n\text{comp}} \right]^\top \in \mathbb{R}^{3n\text{comp}}. \tag{3.26}
\]
3.3. Datasets

Here \( n_{\text{comp}} \) is the number of components in the mixture. The one-dimensional \( y \)-value is generated by normalizing the weights and sampling from the mixture:

\[
    w_i = \frac{\tilde{w}_i}{\sum_{j=1}^{n_{\text{comp}}} \tilde{w}_j} \quad \forall i \in \{1, \ldots, n_{\text{comp}}\}
\]  

(3.27)

\[
c \sim \text{Categorical}(w_1, w_2, \ldots, w_{n_{\text{comp}}})
\]

(3.28)

\[
\tilde{y} \sim \text{Weibull}(1, (1 + k_c))
\]

(3.29)

\[
y = \tilde{y} + 5\sigma_c.
\]

(3.30)

\( \text{Categorical}(w_1, w_2, \ldots, w_{n_{\text{comp}}}) \) is the distribution of a discrete random variable that takes value \( i \) with probability \( w_i \).

As \( n_{\text{comp}} \) is increased, the mixture requires more parameters and the dimensionality of \( x \) increases. Note however that this also makes the conditional distribution more complex, making it harder for a model to properly learn to estimate it.

Datasets were generated for \( n_{\text{comp}} = 1, 2, 3 \) and 5, corresponding to \( x \)-dimensionality 3, 6, 9 and 15. For each dataset 50000 samples were generated for training, 5000 for validation and 5000 for testing.

### 3.3.4 microwave

The microwave dataset originates from a survey of cosmic microwave background radiation\[6\]. It consists of 899 data points where both \( x \) and \( y \) are one-dimensional. The input \( x \) represents multipole moment and \( y \) the temperature angular power spectrum, but the physical meaning of the data is not of great importance. The data is non-linear and heteroskedastic, as can be seen in figure 3.7. For this reason it has previously been used for experiments with complex regression models\[24\]. 70% of the microwave dataset was used for training, 15% for validation and 15% for testing. The data was pre-processed by normalizing \( x \) to lie in \([0, 1]\) and standardizing \( y \) to have zero mean and unit variance.
3.3. Datasets

![Figure 3.7: Test split of microwave dataset](image)

3.3.5 power

The *power* dataset\(^4\) contains measurements on a combined cycle power plant over 6 years\(^5\). The \(y\) to be predicted is net hourly electrical energy output of the power plant. This is a one-dimensional real value. The \(x\)-vector is four-dimensional and contains physical measurement values from the power plant. The exact features are:

- Temperature
- Ambient pressure
- Relative humidity
- Exhaust vacuum

The dataset contains 9568 samples. 70\% of these were used for training, 15\% for validation and 15\% for the test set. Both \(x\) and \(y\) were standardized in pre-processing.

3.3.6 housing

The *housing* dataset\(^6\) contains data from 1990 about houses in the state of California, U.S\(^7\). The dataset does not contain information about individual houses, but averages and medians for entire housing blocks. The goal is to predict the median house value of the block. The features for a housing block, making up \(x \in \mathbb{R}^8\), are:

- Median income
- Median house age
- Average number of rooms
- Average number of bedrooms
- Total population
- Average house occupancy


\(^5\)The *housing* dataset can be found at [http://lib.stat.cmu.edu/datasets/houses.zip](http://lib.stat.cmu.edu/datasets/houses.zip) or downloaded through the scikit-learn library.
3.4 Evaluation

Models were evaluated using log-likelihood and by retraining a CGAN discriminator to convergence. In one-dimensional cases a set of generated samples were also plotted for qualitative, visual evaluations. Such plots contain one sample from the model distribution \(q(y|x)\) for each \(x\) in the test set.

3.4.1 Log-likelihood

Log-likelihood was used for quantitative evaluation in all experiments. Since there is no closed form expression for the log-likelihood in CGANs, KDE was used to estimate the value. For each \(x\) in the test set 1000 \(y\)’s were generated from the model. The log-likelihood \(\log(q(y|x))\) was then estimated from these samples using KDE and a Gaussian kernel, as described in section 2.6.1. The scale parameter of the Gaussian kernel was determined by testing a set of different values and using the scale that maximized the log-likelihood. In validation this was done on the validation set itself, but at test time the scaling parameter was determined on the validation set instead of the test set. The set of candidates for the scale parameter were chosen as 100 values logarithmically spaced between 0.001 and 0.5. For validation during training only 200 samples and 50 scale-values were used.

The KDE process includes randomness in which samples are drawn. To account for this the KDE at test time was repeated 10 times. From these 10 evaluations the mean and standard deviation for the log-likelihood estimate could be computed.

Most baseline models used have closed form expressions for the log-likelihood, so the true value could be computed. Baseline models were additionally also evaluated using KDE. The set of samples for \(y\) were generated by sampling from the resulting model distribution. The choice to also include the KDE-estimate for baseline models was motivated by the fact that KDE tends to underestimate the true log-likelihood. To make a fair and useful comparison between CGANs and the baseline models it is therefore more suitable to compare KDE-estimates for both. Having both the true log-likelihood and a KDE-estimate for some models additionally allowed for better understanding the estimation error.

3.4.2 Estimated Divergence

As a secondary evaluation metric, the value of the CGAN training objective for a discriminator trained to convergence was used. This type of metric was described in more detail in section 2.6.3. The specific CGAN training objective used was the formulation for Least Squares CGAN (\(V_{LS}(T, G)\) from eq. 2.56). This choice was motivated by the stable training process of LS-CGAN and its proven usefulness as an evaluation metric [28]. Computing the value of the LS-CGAN objective requires one set of real data and one set of generated samples. Training a discriminator and evaluating this metric therefore first requires sampling generated datasets corresponding to all the real data.

A new training dataset was generated by sampling new values of \(y\) from the model distribution. This creates a new dataset of \((x, y)\) pairs, where each \(x\) comes from the original data distribution and each \(y\) from the model distribution conditioned on that \(x\). The same was done for the validation and test datasets. Using both the real training data and generated training dataset an LS-CGAN Discriminator was then trained for 500 epochs. The RMSProp optimizer was used with a learning rate of 0.001. During training the value of the LS-CGAN
objective was also computed using the validation data (real and generated). Early stopping was applied at the epoch that maximized this value. The value of the LS-CGAN objective for the test data was then computed using the trained discriminator. To get values in a more convenient range, the final metric was defined: $\hat{D}_{LS} \overset{\text{def}}{=} (-100) \times V_{LS}(T, G)$. The metric then normally takes values in $[0, 25]$, with higher being better. The value 25 here represents that there is no way for the discriminator to determine between real and generated samples. Values above 25 are possible, but represent nothing different than the discriminator being completely fooled. For easier comparison these were therefore set to 25.0.

### 3.4.3 Early Stopping

For models based on neural networks, including CGANs, the best scoring model parameters (weights and biases for all layers in the network) were restored and used for testing. The best scoring parameters were taken from the step in training where the highest log-likelihood was achieved on the validation set. This corresponds to regularization by early stopping, as discussed in section 2.1.4. Evaluation on the validation data during training was done every 20th epoch for small datasets and every 5th epoch for larger datasets.

Using log-likelihood on the validation set for early stopping is intuitive for models with likelihood-based loss functions. The choice is not as clear in the CGAN case. Ideally, early stopping of CGAN training should be performed based on the value of the CGAN training objective (for validation data). Doing this properly would however include retraining a discriminator on the validation data at each validation step. This was deemed too computationally expensive for any possible benefit over using the KDE-estimated log-likelihood.
A number of experiments were conducted in order to better understand the properties of the CGAN regression model. Multiple CGAN models were trained and compared to baselines on synthetic datasets. This helped to get an initial understanding of the model and answer research question 1: How well can CGANs approximate simple and complex data distributions? The same was also done for some real world datasets to answer research question 2: How does CGANs compare to alternative probabilistic regression models on real world datasets? Smaller studies of specific tuning parameters and architecture choices were also done to answer research question 3: How does the use of different training objectives and neural network architectures impact the training process and the capability of CGANs to approximate distributions?

Each section of this chapter describes one experiment. Experiment description, results and discussion have been grouped together. An overarching discussion of the results can also be found in section 5.1. Small experiments related to noise distributions, noise dimensionality and activation functions can be found in appendix A. These were mainly used to motivate hyperparameter choices for other experiments. They offer some general insights on how the CGAN model should be used in practice. In tables with numerical results the best score is made bold and the second best score underlined. If there could be any ambiguity, arrows denote whether higher values (↑) or lower values (↓) are better. For the most commonly used metrics of log-likelihood and $\hat{D}_{\text{LLS}}$ higher values are better.

Because of the inherent differences in the training method, results for GMMN are reported separately from other CGANs. This should help to assess whether MMD-based CGANs can be competitive with full discriminator training.

For baselines models the true log-likelihood is given in (parentheses). In the DCTD baseline model the log-likelihood is computed through importance sampling. This is an estimate, and therefore not treated as true log-likelihood. The importance sampling estimate is given in [square brackets]. KDE-estimates are given for the DCTD model as well.

Numerical results are reported up to one or a few decimal places. Further precision was determined not to be useful due to the approximations involved in the different metrics. Numbers at later decimal places would be mostly a result of randomness involved in the sampling processes underlying the metric computations. They would offer very little insight about the performance of the actual models.
The computer used for training and evaluation was equipped with an Nvidia GTX1070 GPU and a 6-core 3.7 GHz CPU. The GPU was used for all neural network training and other large-scale numerical computation, such as KDE.

4.1 Small Synthetic Datasets

CGANs using different training objectives, as well as baseline models, were trained and evaluated on the small synthetic datasets.

4.1.1 Experiment Setup

For the const_noise and laplace datasets the small double-input CGAN network architecture was used with a noise dimensionality of 1. The CGANs were trained for 1000 epochs. Note that early stopping was always used, so the final models had not been trained the full number of epochs. Learning rate was determined by trying each value in \{0.01, 0.001, 0.0001\}.

On the datasets exponential, butterfly, heteroskedastic, bimodal and complex the medium noise-injection network was used. The noise dimensionality was set to 5. For these datasets the models were trained for 2000 epochs and learning rates in \{0.001, 0.0005, 0.00001\} were considered (for GMMN \{0.01, 0.001, 0.0001\} was used). On the exponential, heteroskedastic and butterfly datasets the scale parameter in the KDE kernel was restricted to be above 0.01. This was necessary to prevent overfitting the parameter value to the validation set, numerically resulting in a log-likelihood of negative infinity.

On the swirls dataset the large noise-injection network was used with a noise dimensionality of 5. Training was done for 2000 epochs and the same learning rates as for medium networks were tried.

The baseline models were also trained and evaluated on the small synthetic datasets. Neural networks used for the baseline models matched the sizes used for the CGANs. The same numbers of training epochs were also used. Hyperparameters for the neural network baselines were determined using grid search as described in section 4.2.4. The values considered were learning rates in \{0.01, 0.001, 0.0001\} and \(L_2\)-regularization coefficients (\(\lambda\) in eq. 2.11) in \{0, 0.001, 0.0001\}. For the MDN model the grid search also included \(K\), the number of mixture components, where the values \{2, 5, 10, 20\} were considered.

For some of the synthetic datasets additional evaluation metrics were used. These would give further insight into how well the different models estimate specific properties of the distributions. For the heteroskedastic dataset the mean difference in standard deviation was measured. The metric was computed as

\[
E_{\sigma} \overset{\text{def}}{=} \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} |\sigma(x^{(i)}) - \hat{\sigma}(x^{(i)})|, \tag{4.1}
\]

for \(N_{\text{test}} = 1000\) samples of \(x\) in the test set. Here \(\sigma(x^{(i)})\) is the true standard deviation of the synthetic data distribution \(p_d(y|x^{(i)})\) (which for heteroskedastic is gaussian) at \(x^{(i)}\). For the model distribution the standard deviation estimate \(\hat{\sigma}(x^{(i)})\) was computed by sampling 1000 \(y\):s conditioned on \(x^{(i)}\) and computing the standard deviation of the sample set.

On the bimodal dataset the additional metric \(\Psi_{\text{Support}}\) was used, denoting the mean fraction of generated samples that lie within the support of the data distribution. Equivalently, the value \(1 - \Psi_{\text{Support}}\) describes how likely a model is to generate samples that could never come from the true data distribution. The value of \(\Psi_{\text{Support}}\) was computed by sampling \(N_{\text{eval}} = 1000\) \(y\):s from the model for each \(x\) in the test set, determining the fraction of these that lie within the support of the data distribution and finally taking the mean across all \(x\):s in the test set.

\(^{1}\)The support of a distribution \(p\) is here the set of all the points \(y\) such that \(p(y) > 0\).
More formally:

$$\Psi_{\text{Support}} \overset{\text{def}}{=} \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \left| \left\{ y^{(i,j)} \mid p_d(y^{(i,j)} | x^{(i)}) > 0, j \in \{1, \ldots, N_{\text{eval}}\} \right\} \right|$$ (4.2)

$$y^{(i,j)} \sim q(y | x^{(i)}) \quad \forall j \in \{1, \ldots, N_{\text{eval}}\}, \forall i \in \{1, \ldots, N_{\text{test}}\},$$ (4.3)

where $q(y | x)$ is the model distribution being evaluated. Since the dataset-specific metrics also feature random sampling, the computation was repeated for 10 different random seeds and the mean result reported.

### 4.1.2 Results

Tables 4.1-4.5 list results on the test set for the small synthetic datasets. Standard deviations of the log-likelihood estimates were less than 0.01 in all but a few cases. Figure D.1-D.7 in appendix D show plots of samples from the trained models. Table 4.6 describes training times for the experiment, grouped by network size.

To allow for comparisons, results are reported for all versions of CGANs used. If the choice of divergence, and thereby loss functions, is considered a hyperparameter it should be determined through model selection using the validation data. The CGAN version achieving the highest log-likelihood on the validation data is denoted with the symbol $\triangle$ for each dataset. This would be the resulting CGAN model if model selection was performed over divergences. Comparisons between CGAN as a general model and the baselines should therefore consider the results from the $\triangle$ model.

<table>
<thead>
<tr>
<th>Baselines</th>
<th>const_noise</th>
<th></th>
<th>laplace</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log-like.</td>
<td>$D_{\text{LS}}$</td>
<td>Log-like.</td>
<td>$D_{\text{LS}}$</td>
</tr>
<tr>
<td>Gaussian Process</td>
<td>-0.72 (-0.72)</td>
<td>24.9</td>
<td>-1.08 (-1.06)</td>
<td>24.8</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>-0.72 (-0.71)</td>
<td><strong>25.0</strong></td>
<td>-1.07 (-1.06)</td>
<td>24.7</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>-0.72 (-0.71)</td>
<td><strong>25.0</strong></td>
<td>-1.08 (-1.06)</td>
<td>24.7</td>
</tr>
<tr>
<td>MDN</td>
<td>-0.72 (-0.71)</td>
<td><strong>25.0</strong></td>
<td>-1.02 (-1.01)</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>DCTD</td>
<td>-0.73 [-0.73]</td>
<td><strong>25.0</strong></td>
<td>-1.02 [-1.00]</td>
<td><strong>25.0</strong></td>
</tr>
</tbody>
</table>

CGANs

<table>
<thead>
<tr>
<th>Baselines</th>
<th>const_noise</th>
<th></th>
<th>laplace</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log-like.</td>
<td>$D_{\text{LS}}$</td>
<td>Log-like.</td>
<td>$D_{\text{LS}}$</td>
</tr>
<tr>
<td>Standard</td>
<td>-0.73 $\triangle$</td>
<td>24.2</td>
<td>-1.02 $\diamond$</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>KL</td>
<td>-0.73</td>
<td>22.0</td>
<td>-1.04</td>
<td>24.2</td>
</tr>
<tr>
<td>Reverse KL</td>
<td>-0.72</td>
<td>24.5</td>
<td>-1.03</td>
<td>24.8</td>
</tr>
<tr>
<td>Neyman $\chi^2$</td>
<td>-0.72</td>
<td>21.4</td>
<td>-1.04</td>
<td>24.3</td>
</tr>
<tr>
<td>Pearson $\chi^2$</td>
<td>-0.73</td>
<td>24.6</td>
<td>-1.02</td>
<td>24.7</td>
</tr>
<tr>
<td>Squared Hellinger</td>
<td>-0.73</td>
<td>23.7</td>
<td>-1.01</td>
<td>24.9</td>
</tr>
<tr>
<td>Jensen-Shannon</td>
<td>-0.73</td>
<td><strong>25.0</strong></td>
<td>-1.03</td>
<td>24.1</td>
</tr>
<tr>
<td>Least Squares</td>
<td>-0.72</td>
<td>24.4</td>
<td>-1.02 $\diamond$</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>GMMN</td>
<td>-0.73</td>
<td><strong>25.0</strong></td>
<td>-1.03</td>
<td>24.6</td>
</tr>
</tbody>
</table>

Table 4.1: Results from experiments on const_noise and laplace synthetic datasets.

### 4.1.3 Discussion

For the simplest datasets, const_noise and laplace in table 4.1, all models perform quite well. The baselines that only model a single Gaussian reach somewhat worse scores on the laplace distribution, as can be expected. This trend continues over all datasets with non-Gaussian distributions. Looking at slightly more complex datasets, such as exponential and butterfly in table 4.2, the MDN and DCTD baselines clearly stand out from the other ones. Some of the CGANs perform similarly or even better (on exponential). Although all CGAN
### 4.1. Small Synthetic Datasets

#### Table 4.2: Results from experiments on exponential and butterfly synthetic datasets.

<table>
<thead>
<tr>
<th></th>
<th>exponential Log-like.</th>
<th>$D_{LS}$</th>
<th>butterfly Log-like.</th>
<th>$D_{LS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baselines</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussian Process</td>
<td>0.19 (0.25)</td>
<td>18.9</td>
<td>-0.93 (-0.93)</td>
<td>19.9</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>0.20 (0.24)</td>
<td>19.1</td>
<td>-0.93 (-0.89)</td>
<td>19.7</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>0.42 (0.69)</td>
<td>22.5</td>
<td>-0.58 (-0.46)</td>
<td>23.5</td>
</tr>
<tr>
<td>MDN</td>
<td>0.62 (1.08)</td>
<td><strong>25.0</strong></td>
<td><strong>-0.40 (-0.33)</strong></td>
<td>24.9</td>
</tr>
<tr>
<td>DCTD</td>
<td>0.61 [1.10]</td>
<td><strong>25.0</strong></td>
<td><strong>-0.40 [-0.32]</strong></td>
<td>25.0</td>
</tr>
<tr>
<td><strong>CGANs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>0.53</td>
<td>23.7</td>
<td>-0.44</td>
<td>25.0</td>
</tr>
<tr>
<td>KL</td>
<td><strong>0.63</strong></td>
<td><strong>25.0</strong></td>
<td>-0.45</td>
<td>25.0</td>
</tr>
<tr>
<td>Reverse KL</td>
<td>0.61</td>
<td><strong>25.0</strong></td>
<td>-0.41</td>
<td>25.0</td>
</tr>
<tr>
<td>Neyman $\chi^2$</td>
<td>0.53</td>
<td>22.1</td>
<td>-0.42</td>
<td>25.0</td>
</tr>
<tr>
<td>Pearson $\chi^2$</td>
<td>0.62 $\Diamond$</td>
<td><strong>25.0</strong></td>
<td>-0.45</td>
<td>25.0</td>
</tr>
<tr>
<td>Squared Hellinger</td>
<td><strong>0.63</strong></td>
<td><strong>25.0</strong></td>
<td>-0.41</td>
<td>24.9</td>
</tr>
<tr>
<td>Jensen-Shannon</td>
<td>0.62</td>
<td>23.7</td>
<td>-0.45</td>
<td>24.9</td>
</tr>
<tr>
<td>Least Squares</td>
<td>0.62</td>
<td>24.7</td>
<td>-0.41 $\Diamond$</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>GMMN</td>
<td>0.61</td>
<td>22.9</td>
<td>-0.43</td>
<td>24.5</td>
</tr>
</tbody>
</table>

#### Table 4.3: Results from experiments on heteroskedastic synthetic datasets. $E_\sigma$ measures the mean difference in standard deviation (see eq. 4.1).

<table>
<thead>
<tr>
<th></th>
<th>heteroskedastic Log-like.</th>
<th>$D_{LS}$</th>
<th>$E_\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baselines</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussian Process</td>
<td>-0.99 (-0.97)</td>
<td>19.5</td>
<td>0.324</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>-0.97 (-0.96)</td>
<td>19.7</td>
<td>0.324</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td><strong>-0.41 (-0.24)</strong></td>
<td><strong>25.0</strong></td>
<td>0.027</td>
</tr>
<tr>
<td>MDN</td>
<td><strong>-0.41 (-0.24)</strong></td>
<td><strong>25.0</strong></td>
<td>0.028</td>
</tr>
<tr>
<td>DCTD</td>
<td><strong>-0.42 [-0.27]</strong></td>
<td><strong>25.0</strong></td>
<td>0.036</td>
</tr>
<tr>
<td><strong>CGANs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>-0.48</td>
<td><strong>25.0</strong></td>
<td>0.038</td>
</tr>
<tr>
<td>KL</td>
<td>-0.47 $\Diamond$</td>
<td>24.8</td>
<td>0.101</td>
</tr>
<tr>
<td>Reverse KL</td>
<td>-0.47</td>
<td>24.8</td>
<td>0.057</td>
</tr>
<tr>
<td>Neyman $\chi^2$</td>
<td>-0.91</td>
<td>19.9</td>
<td>0.280</td>
</tr>
<tr>
<td>Pearson $\chi^2$</td>
<td>-0.43</td>
<td>24.9</td>
<td>0.049</td>
</tr>
<tr>
<td>Squared Hellinger</td>
<td>-0.46</td>
<td>24.4</td>
<td>0.037</td>
</tr>
<tr>
<td>Jensen-Shannon</td>
<td>-0.53</td>
<td>24.0</td>
<td>0.070</td>
</tr>
<tr>
<td>Least Squares</td>
<td>-0.44</td>
<td><strong>25.0</strong></td>
<td><strong>0.024</strong></td>
</tr>
<tr>
<td>GMMN</td>
<td>-0.51</td>
<td><strong>25.0</strong></td>
<td>0.036</td>
</tr>
</tbody>
</table>
4.1. Small Synthetic Datasets

<table>
<thead>
<tr>
<th>bimodal</th>
<th>Log-like.</th>
<th>$D_{LS}$</th>
<th>$\Psi_{\text{Support}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Process</td>
<td>-1.69 (-1.69)</td>
<td>14.7</td>
<td>0.385</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>-1.69 (-1.69)</td>
<td>14.4</td>
<td>0.385</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>-1.65 (-1.65)</td>
<td>14.9</td>
<td>0.409</td>
</tr>
<tr>
<td>MDN</td>
<td>-0.77 (-0.76)</td>
<td><strong>25.0</strong></td>
<td>0.971</td>
</tr>
<tr>
<td>DCTD</td>
<td><strong>-0.75 [-0.71]</strong></td>
<td><strong>25.0</strong></td>
<td>0.981</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGANs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
</tr>
<tr>
<td>KL</td>
</tr>
<tr>
<td>Reverse KL</td>
</tr>
<tr>
<td>Neyman $\chi^2$</td>
</tr>
<tr>
<td>Pearson $\chi^2$</td>
</tr>
<tr>
<td>Squared Hellinger</td>
</tr>
<tr>
<td>Jensen-Shannon</td>
</tr>
<tr>
<td>Least Squares</td>
</tr>
<tr>
<td>GMMN</td>
</tr>
</tbody>
</table>

Table 4.4: Results from experiments on bimodal synthetic datasets. $\Psi_{\text{Support}}$ measures the mean fraction of samples in the data distribution support (see eq. 4.2).

<table>
<thead>
<tr>
<th>complex</th>
<th>swirls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log-like.</td>
<td>$D_{LS}$</td>
</tr>
<tr>
<td>Baselines</td>
<td></td>
</tr>
<tr>
<td>Gaussian Process</td>
<td>-1.00 (-0.99)</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>-1.00 (-0.99)</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>-0.70 (-0.69)</td>
</tr>
<tr>
<td>MDN</td>
<td><strong>-0.31 [-0.28]</strong></td>
</tr>
<tr>
<td>DCTD</td>
<td><strong>-0.31 [-0.28]</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGANs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
</tr>
<tr>
<td>KL</td>
</tr>
<tr>
<td>Reverse KL</td>
</tr>
<tr>
<td>Neyman $\chi^2$</td>
</tr>
<tr>
<td>Pearson $\chi^2$</td>
</tr>
<tr>
<td>Squared Hellinger</td>
</tr>
<tr>
<td>Jensen-Shannon</td>
</tr>
<tr>
<td>Least Squares</td>
</tr>
<tr>
<td>GMMN</td>
</tr>
</tbody>
</table>

Table 4.5: Results from experiments on complex and swirls synthetic datasets.


<table>
<thead>
<tr>
<th>Model</th>
<th>time/epoch (s)</th>
<th>ES epoch</th>
<th>Training time to ES (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>-</td>
<td>-</td>
<td>1:33±0:00</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>0.26</td>
<td>214±45</td>
<td>0:56±0:13</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>0.27</td>
<td>89±30</td>
<td>0:24±0:08</td>
</tr>
<tr>
<td>Small</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDN</td>
<td>0.26</td>
<td>34±5</td>
<td>0:09±0:01</td>
</tr>
<tr>
<td>DCTD</td>
<td>0.45</td>
<td>59±40</td>
<td>0:27±0:18</td>
</tr>
<tr>
<td>CGAN</td>
<td>0.28</td>
<td>159±120</td>
<td>0:41±0:30</td>
</tr>
<tr>
<td>GMMN</td>
<td>0.62</td>
<td>339±220</td>
<td>3:29±2:15</td>
</tr>
<tr>
<td>Medium</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GP</td>
<td>-</td>
<td>-</td>
<td>1:27±0:31</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>0.33</td>
<td>183±155</td>
<td>1:00±0:51</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>0.33</td>
<td>1127±458</td>
<td>6:20±2:36</td>
</tr>
<tr>
<td>MDN</td>
<td>0.32</td>
<td>651±565</td>
<td>3:13±2:31</td>
</tr>
<tr>
<td>DCTD</td>
<td>0.51</td>
<td>1079±496</td>
<td>8:16±2:40</td>
</tr>
<tr>
<td>CGAN</td>
<td>0.39</td>
<td>931±625</td>
<td>5:54±3:35</td>
</tr>
<tr>
<td>GMMN</td>
<td>0.69</td>
<td>1602±318</td>
<td>17:23±4:07</td>
</tr>
<tr>
<td>Large</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GP</td>
<td>-</td>
<td>-</td>
<td>2:32</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>0.38</td>
<td>59</td>
<td>0:21</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>0.38</td>
<td>1179</td>
<td>7:21</td>
</tr>
<tr>
<td>MDN</td>
<td>0.41</td>
<td>1119</td>
<td>7:39</td>
</tr>
<tr>
<td>DCTD</td>
<td>1.50</td>
<td>119</td>
<td>2:58</td>
</tr>
<tr>
<td>CGAN</td>
<td>0.51</td>
<td>1019</td>
<td>8:52</td>
</tr>
<tr>
<td>GMMN</td>
<td>1.20</td>
<td>699</td>
<td>12:18</td>
</tr>
</tbody>
</table>

Table 4.6: Training times for different models with different network sizes. Early stopping (ES) epoch is the epoch where the highest validation log-likelihood was achieved, where early stopping would be applied. Training time to ES is the total time until that epoch. These values are averaged across the best versions (choice of hyperparameters) of the models for each dataset. Mean ± standard deviation is reported. Since the GP model is not trained in epochs and does not use early stopping only the complete training time (for kernel parameter optimization) is reported.

versions pick up on the general distribution of the data, there is some difference in how well they seem to capture the exponential noise. This can be observed from looking at the plots in figure D.3 of appendix D.

The heteroskedastic dataset tests whether the models can learn to estimate a distribution where the variance varies with $x$. Heteroskedastic neural network regression and MDN acts as the gold standars here, since the true distribution is in their model family. As expected they perform the best, but multiple CGANs are not far behind in terms of KDE log-likelihood. Looking at $E_0$, the Least Squares CGAN achieves the best score. There is however quite a disconnect between this metric and the log-likelihood. For the KL CGAN, which performed best in validation, the $E_0$-metric is substantially worse.

With the bimodal dataset, the capabilities of modelling multimodal distributions is evaluated. The models additionally have to learn to estimate uniformly distributed noise around the modes. This is accomplished with varying success across the CGAN models. The differences become most clear from the scatter plots in figure D.6. KL, Pearson $\chi^2$ and Least Squares CGANs all perform on par with the better baselines. Other CGANs either fail to pick up on the hard edges of the uniform distribution (Reverse KL, GMMN) or does not detect the two components of the distribution properly (Neyman $\chi^2$, Jensen-Shannon). For the well-performing CGANs very few samples are outside the support of the true distribution, as indicated by high $\Psi_{\text{Support}}$-values. This shows that the CGANs can learn distributions with finite support. Very few values end up in the region between the two high probability-density areas of the distribution. For some choices of hyperparameters tried, CGAN models ended
4.2 CGAN Network Architecture

up only generating samples from one of the components in the bimodal data distribution. This is a local optimum that the training struggled to escape from. None of these cases are however present in testing, since other hyperparameters led to better results for those models. The best performing baseline on bimodal is DCTD. For an energy-based model, learning this distribution becomes similar to a binary classification problem. It is enough for it to learn to classify if an \((x,y)\)-pair is in the support of the data distribution (and output a large value) or not (output a very small value).

For the complex and swirls datasets in table 4.5 the MDN and DCTD baselines continue to perform the best. Again, a subset of the CGANs achieve competitive scores while some fail to capture aspects of the data distributions. The DCTD baseline performs slightly worse than could be expected on the swirls dataset. This could be because of the two-dimensional \(y\) in this dataset, likely leading to worse estimates in the importance sampling. The MDN baseline performs well for both complex and swirls. The true distributions for these datasets are however in the MDN model class, so good performance should be expected.

As can be seen in table 4.6, none of the models considered took very long to train. The ones that stand out are DCTD and GMMN. DCTD likely takes longer to train due to the importance sampling procedure. For the GMMN model the timing is most likely explained by the quadratic complexity in the MMD computation. The computation was implemented to utilize fast GPU matrix operations, but there might still exist possible optimizations. This was not investigated further. In table 4.6 it should be noted that the CGAN models did not take much longer to train than some of the neural network baselines. This is surprising, considering that the CGANs require two networks to be trained. The slowdown from training a secondary network does however seem to increase somewhat as the network size is increased.

Across all the small synthetic datasets, the GMMN model performed decently. It never failed to pick up on any key property in the data distributions and often achieved scores similar to other CGANs. This shows that it is possible to learn complex conditional distributions even without a discriminator network.

From all of the results in this experiment it is hard to point to a single CGAN version that is the most suitable to use. There is however clearly a subset that consistently performs better, including Standard, Pearson \(\chi^2\) and Least Squares CGAN.

4.2 CGAN Network Architecture

In the CGAN network architecture experiment different generator architectures were trained and evaluated on the same datasets. The goal of the experiment was to better understand what parts of the generator network are important.

4.2.1 Experiment Setup

The set of CGAN architectures used included the small, medium and large versions of the double-input networks. Also the medium noise-injection architecture was included.

In the double-input architecture a deep representation is learned for each of \(x\) and \(z\). To further understand which parts of this setup are important, three experimental architectures were also included in the experiment. These three were all based on a generator network with \(l_{x;G} = l_{z;G} = l_{y;G} = 3\) layers, all consisting of 64 units. The discriminator used had the same size, with 3 layers of 64 units in each part of the network. In the first of the experimental architectures the deep representation of \(x\) was removed by setting \(l_{x;G} = 0\). The same was done for \(z\) in the second one, setting \(l_{z;G} = 0\). The third experimental architecture uses 3 layers to achieve deep representations for both \(x\) and \(z\), but does not feature any hidden layers after that. Instead, with \(l_{y;G} = 0\), \(y\) is just predicted by a single linear mapping from the concatenated deep representations. The discriminator network was not modified.

These, in total seven, different architectures were trained and evaluated on the bimodal and complex synthetic datasets. This was repeated for standard CGAN, least squares CGAN and
The noise dimensionality was set to 5 in all cases. Each network was trained for 2000 epochs and learning rates were determined separately by grid search over \{0.01, 0.001, 0.0001\}.

### 4.2.2 Results

Resulting log-likelihoods for the CGAN network architecture experiment are reported in Table 4.7.

<table>
<thead>
<tr>
<th>CGAN</th>
<th>Network Architecture</th>
<th>Dataset</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(t_{x;G}) (t_{z;G}) (t_{y;G})</td>
<td>bimodal</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td><strong>Standard</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 3 3</td>
<td>-0.79</td>
<td>-0.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 0 3</td>
<td><strong>-0.77</strong></td>
<td>-0.33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 3 0</td>
<td>-1.24</td>
<td>-1.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>small</td>
<td>-0.82</td>
<td>-0.34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>-0.81</td>
<td>-0.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>-0.82</td>
<td>-0.37</td>
<td></td>
</tr>
<tr>
<td></td>
<td>noise_injection</td>
<td>-0.79</td>
<td>-0.31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 3 3</td>
<td>-0.79</td>
<td>-0.31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 0 3</td>
<td><strong>-0.78</strong></td>
<td>-0.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 3 0</td>
<td>-1.33</td>
<td>-1.01</td>
<td></td>
</tr>
<tr>
<td><strong>Least Squares</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>small</td>
<td>-0.82</td>
<td>-0.41</td>
<td></td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>-0.79</td>
<td>-0.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>-0.81</td>
<td>-0.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>noise_injection</td>
<td>-0.79</td>
<td>-0.33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 3 3</td>
<td><strong>-0.82</strong></td>
<td>-0.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 0 3</td>
<td>-0.84</td>
<td>-0.48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 3 0</td>
<td>-1.23</td>
<td>-0.96</td>
<td></td>
</tr>
<tr>
<td><strong>GMMN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>small</td>
<td>-0.82</td>
<td>-0.41</td>
<td></td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>-0.83</td>
<td>-0.44</td>
<td></td>
</tr>
<tr>
<td></td>
<td>large</td>
<td><strong>-0.82</strong></td>
<td>-0.38</td>
<td></td>
</tr>
<tr>
<td></td>
<td>noise_injection</td>
<td>-0.87</td>
<td>-0.36</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Results (log-likelihoods) for the network architecture experiment

### 4.2.3 Discussion

Table 4.7 shows that many different architectures achieved similar results. Exact structure or size of the generator network does not seem to be key to the CGAN performance. If any one architecture should be recommended for future applications, noise_injection would be a solid choice. In this experiment it has achieved top scores for almost all datasets and CGANs.

The case where \(t_{y;G} = 0\) stands out because of its poor performance. In this architecture there are no hidden layers that depend on both \(x\) and \(z\). A complex noise distribution can still be learned, since \(z\) is passed through multiple hidden layers on its own. The distribution can however not vary with \(x\), since the deep representations are only combined at the generator output. The poor performance of this type of architecture is therefore to be expected.

### 4.3 trajectories experiment

CGANs and baseline models were trained and evaluated on trajectories datasets with different dimensionality for \(y\). The experiment gives some insight into how the models capture conditional distributions over more than one dimension.
4.3.1 Experiment Setup

The CGANs were trained for 4000 epochs with a learning rate of 0.0001 and batch size 100. The large noise-injection network architecture was used. Noise dimensionality was fixed to 30 and the CGAN variants used were Standard, Least Squares, Squared Hellinger and Pearson $\chi^2$. For this experiment the divergence used for CGAN training was considered a hyperparameter. Only the version that achieved the highest log-likelihood on the validation set was evaluated on the test data. To get better estimates for the log-likelihood on the validation data, 500 samples were used in KDE.

The neural network baseline models were trained for 2000 epochs with the large network size. It was observed that this number of epochs was enough for convergence and further training would only result in overfitting. Learning rates $\{0.001, 0.0001\}$ and $L_2$-regularization coefficients $\{0, 0.0001\}$ were considered in model selection. The MDN model used 20 mixture components. For this, and upcoming experiments with large datasets, the Gaussian Process baseline was left out. The reason for this was that the standard GP model computationally scales poorly to large datasets \cite[p. 171]{9}, \cite[p. 306-310]{47}. Initial results had also shown that its performance is very similar to the neural network regression model.

For evaluating the models on the test sets, 2000 samples were used in KDE. The scale parameter was constrained to be above 0.005.

It should be clarified that the point of the experiment on the trajectories datasets was not to consider time-series data specifically. Although the datasets were generated from a random process, this was not the focus of the experiment. The underlying random process of the dataset was simply a way to achieve a synthetic dataset with a complex distribution for $y$. It is likely that better performance could be achieved for all models by taking into account the temporal structure of the data. For this experiment that structure was however assumed to be unknown.

4.3.2 Results

Results for trajectories datasets with different dimensionality of $y$, denoted $d_y$, are reported in table \ref{tab:results}. The standard deviation of log-likelihood across KDE-runs was less than 0.03. The evaluated CGAN variants, as determined through model selection, was Pearson $\chi^2$ for $d_y = 4$ and Least Squares for $d_y = 10$ and 20.

<table>
<thead>
<tr>
<th></th>
<th>$d_y = 4$</th>
<th></th>
<th>$d_y = 10$</th>
<th></th>
<th>$d_y = 20$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log-like.</td>
<td>$D_{LS}$</td>
<td>Log-like.</td>
<td>$D_{LS}$</td>
<td>Log-like.</td>
<td>$D_{LS}$</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>0.55 (0.69)</td>
<td>2.8</td>
<td>2.83 (4.26)</td>
<td>0.2</td>
<td>5.41 (10.93)</td>
<td>0.1</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>1.46 (1.59)</td>
<td>3.9</td>
<td>4.25 (6.44)</td>
<td>0.4</td>
<td>7.73 (15.17)</td>
<td>0.1</td>
</tr>
<tr>
<td>MDN</td>
<td>4.48 (4.96)</td>
<td>8.0</td>
<td>11.46 (15.57)</td>
<td>3.6</td>
<td>22.95 (30.60)</td>
<td>0.6</td>
</tr>
<tr>
<td>CGAN</td>
<td>5.27</td>
<td>24.9</td>
<td>12.38</td>
<td>15.1</td>
<td>27.03</td>
<td>9.1</td>
</tr>
</tbody>
</table>

Table 4.8: Results from experiments on trajectories datasets with different $y$-dimensionality $d_y$.

It was attempted to also include the DCTD baseline model in the experiment, but it was not possible to make the model learn anything useful. Attempts were made to improve performance by substantially increasing the number of samples used in importance sampling. This did however not lead to any major improvement, but made the training time unreasonably long. Also the GMMN model failed to learn from the data, so it was excluded from the experiment.

The MDN model used 20 mixture components for all trajectories datasets. To verify that this choice of hyperparameter did not cause a major disadvantage, an MDN model with 50 mixture components was also trained on the dataset with 20-dimensional $y$. This led to an increase of 0.86 in KDE log-likelihood, but no difference in $D_{LS}$.  

54
Figures 4.1 and 4.2 show scatter plots for trajectories datasets with $d_y = 2$ and 4 respectively. These were drawn from the ground truth distribution, the trained MDN model and the trained CGAN model. Note that each set of scatter plots is conditioned on one specific value of $x$, so it can not be assumed to be representative of model performance on the entire dataset. The high-dimensional $y$‘s are plotted in two dimensions by treating pairs of entries in the $y$-vector as points in two dimensions. This means that such a point corresponds to the position at some step $i$ in the trajectory. This is $w_i$ from the dataset generation process, defined in eq. 3.23. The points for different step index $i$ are then given different colours in the plot.

Figure 4.3 contains trajectories sampled from the CGAN and MDN models trained on trajectories with $d_y = 20$. Samples have been plotted for different values of the maximum turning parameter $t_{\max}$. This indicates how the models have learned to adapt their distributions to this entry in the conditioning variable $x$.

More plots can be found in appendix E. This includes both scatter plots and sampled trajectories for all models and all trajectories datasets.

The baseline models took 30-45 minutes to train each (1.0-1.5 seconds per epoch), with the MDN being the slowest. The CGAN model took 3 hours to train (2.6 seconds per epoch).

![Figure 4.1: Scatter plots of 2000 samples for the trajectories dataset with $d_y = 4$, conditioned on $x = [0.5, 0.5, 0.5, -1, 0.4]^\top$.](image)

![Figure 4.2: Scatter plots of 2000 samples for the trajectories dataset with $d_y = 10$, conditioned on $x = [0.5, 0.5, 0.2, -1, 0.4]^\top$.](image)

### 4.3.3 Discussion

The quantitative results in table 4.8 clearly show how the models compare to each other. As expected, the simpler neural network regression models achieve quite poor scores. It can be noted from the plots in appendix E that they only capture the most general properties of the distributions, such as mean and variance in different dimensions. The MDN and CGAN models manage to better capture the general shape of the distributions. The CGAN model best estimates the distributions, learning not just the general shape but also differences in
4.3. trajectories experiment

Figure 4.3: Trajectories sampled from the true distribution, MDN and CGAN when conditioned on different maximum turning parameters, $t_{\text{max}}$. Models were trained on the trajectories dataset with $d_y = 20$. Samples were specifically conditioned on $x = [0, 0, 0.05, 1, 571, t_{\text{max}}]^T$.

The CGAN model did perform better on the trajectories datasets, but also required a longer training time. It both took longer to train per epoch and required more epochs to properly converge. A low learning rate of 0.0001 was used for the CGAN model because this showed the best results in initial hyperparameter experiments. This likely contributed to why the model needed longer to converge. It might be possible to train CGANs in fewer epochs but with similar results, by scheduling the learning rate or tuning optimizer parameters.
The MDN baseline showed strong performance on the one-dimensional datasets, but struggled to properly model the distributions in this experiment. It was initially believed that the number of mixture components used limited the model. The additional 50-component experiment did however indicate that there are other limitations to the model in this setting. Scaling up the number of mixture components could improve the performance of the MDN model slightly, but it is unlikely that it would reach the results of the CGAN. It should also be noted that the number of weights in the MDN neural network grows with the component-count. For a fair comparison an MDN with many mixture components should therefore be compared to other upscaled models.

The failure of the DCTD baseline could be expected due to the shortcomings of importance sampling in higher dimensions. The proposal distribution used is likely too simple for the more complex trajectories datasets, even when many samples are drawn. A more sophisticated approach where the proposal distribution is learned with the rest of the model would be necessary. The other neural network baselines could be utilized in the DCTD model, either directly acting as proposal distributions or as initialization for a more refined procedure.

It is not as clear why the GMMN model failed to learn the data distributions. There is nothing inherent in the model formulation that is limited by y-dimensionality. In the trajectories datasets the relationship between x and y is complex and different entries in y have vastly different distributions. An hypothesis is that the MMD with a Gaussian kernel in this case offered too little guidance for the generator to learn. Performance could maybe be improved by further utilizing kernel learning to achieve more powerful kernel representations.

4.4 \texttt{wmix} experiment

Experiments with the \texttt{wmix} datasets test the ability of the models to learn complex conditional distributions for higher-dimensional x.

4.4.1 Experiment Setup

CGANs and baseline models were trained on \texttt{wmix} datasets with x-dimensionality $d_x = 3, 6, 9$ and 15. CGANs with noise dimensionality 30 were trained for 2000 epochs. The large noise-injection architecture was used. Standard, Least Squares, Squared Hellinger and Pearson $\chi^2$ CGANs were each trained with a learning rate of 0.0001. For GMMN and neural network baselines, learning rates in $\{0.001, 0.0001\}$ and $L_2$-regularization coefficients in $\{0, 0.0001\}$ were tried in a grid search. Baselines models used corresponding large network sizes. The number of mixture components in MDN was set to 20. Also for this experiment the GP baseline was not used, due to the large sizes of the datasets.

The DCTD model was only trained for 1000 epochs, which was enough for the model to converge or begin to overfit. For DCTD the learning rate was fixed to 0.001, but the same regularization was considered as for the other baselines.

4.4.2 Results

Results for \texttt{wmix} datasets with different $d_x$ are listed in table 4.9 and 4.10. Standard deviations for KDE log-likelihoods were all less than 0.01. The CGAN versions selected from validation results were Least Squares for the $d_x = 3$ dataset and Pearson $\chi^2$ for all other datasets.

Each CGAN model took around 4 hours to train. GMMN was slightly faster at 3 hours. DCTD was only trained for 1000 epochs (half that of other models), but still took 6 hours. The other baseline models completed 2000 epochs in 1.5 hours.

Figure 4.4 shows examples of conditional probability density functions learned by the different models. More such plots, for all values of $d_x$, can be found in appendix F.
4.4. \textbf{wmix} experiment

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
          & \(d_x = 3\) & \(d_x = 6\) & \\
\hline
NN Reg.    & -1.02 (-0.98) & 23.1 & -1.57 (-1.55) & 20.8 \\
Het. NN Reg. & -0.97 (-0.94) & 23.4 & -1.44 (-1.41) & 22.5 \\
MDN        & \textbf{-0.81} (-0.79) & \textbf{25.0} & -1.24 (-1.19) & 24.9 \\
DCTD      & -0.87 (-0.79) & \textbf{25.0} & -1.24 [-1.17] & \textbf{25.0} \\
CGAN      & \textbf{-0.81} & 24.8 & \textbf{-1.22} & 24.8 \\
GMMN      & -0.90 & 23.9 & -1.41 & 23.0 \\
\hline
\end{tabular}
\caption{Results from experiments on \textbf{wmix} datasets with \(d_x = 3\) and \(d_x = 6\).}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
          & \(d_x = 9\) & \(d_x = 15\) & \\
\hline
NN Reg.    & -1.68 (-1.68) & 21.8 & -1.75 (-1.75) & 23.5 \\
Het. NN Reg. & -1.59 (-1.58) & 22.9 & -1.71 (-1.71) & 24.0 \\
MDN        & -1.43 (-1.41) & \textbf{25.0} & -1.64 (-1.64) & \textbf{25.0} \\
DCTD      & \textbf{-1.40} [-1.36] & \textbf{25.0} & -1.62 [-1.61] & \textbf{25.0} \\
CGAN      & -1.41 & 24.9 & \textbf{-1.61} & 24.9 \\
GMMN      & -1.72 & 21.3 & -1.76 & 23.4 \\
\hline
\end{tabular}
\caption{Results from experiments on \textbf{wmix} datasets with \(d_x = 9\) and \(d_x = 15\).}
\end{table}

4.4.3 Discussion

Numerical results show that the CGAN model achieved top scores across all values of \(d_x\). The MDN and DCTD baselines performed on a similar level, with MDN achieving slightly worse log-likelihoods for higher \(d_x\). These results show no indication that the CGAN model should struggle when \(x\)-dimensionality grows.

Because of how the \textbf{wmix} datasets are created, the relationship between \(x\) and \(y\) is very complex. No model captures all parts of the distributions, despite the large amounts of data. The conditional distributions are multimodal, with the modes described by entries in \(x\). Not all modes might actually be present, since \(x\) also contains the weights over mixture components. This forces the models to learn a form of feature extraction. The mixture weight part of \(x\) indicates what other entries actually describe the conditional distribution. If one mixture weight is very low, the corresponding offset and scale modifier (that are also entries in \(x\)) do not affect the distribution and the model should learn to ignore them.

This feature extraction aspect of this experiment can explain the poor performance of the GMMN model. In MMD the kernel is applied directly to \(x\) and there is no explicit feature extraction stage (only a learned rescaling is applied). In other CGANs the discriminator neural network can learn such a feature extraction. MMD thus makes it harder for the GMMN generator to properly learn the conditional distributions. In appendix \ref{kn} it can be seen that GMMN collapses to only modelling a single mode for each distribution. This again motivates using kernel learning together with MMD.

These large datasets give useful indications for the training times of the different models. Training of two neural networks in CGANs here results in twice the training time compared to baselines like MDN. It can be noted that the two best performing models, CGAN and DCTD, both take a long time to train. The MDN model does not perform much worse, but takes a quarter of the time of DCTD to train.
4.5. Real datasets

Moving on from synthetic data, the models were also evaluated on data related to real world systems. The datasets used for this were microwave, housing and power.

4.5.1 Experiment Setup

For the small microwave dataset, the same experiment setup as described in section 4.1.1 for the exponential dataset was used. The KDE kernel scale was also restricted to be above 0.1 at test time.

power and housing are far larger datasets, so a different setup was used for them. All models used the large network sizes and were trained for 2000 epochs. CGANs used the noise-injection architecture, noise dimensionality 20 and learning rate 0.0001. The Standard, Least Squares, Squared Hellinger and Pearson $\chi^2$ variants were considered. To get some better approximations for early stopping, the CGANs used 500 y-samples (instead of 200) for KDE.

For DCTD the learning rate was fixed to 0.001 and $L^2$-regularization coefficients in \{0, 0.0001\} tried. Other neural network baselines used learning rates in \{0.001, 0.0001\} and regularization in \{0, 0.0001\}. The MDN was set to 20 mixture components.

Figure 4.4: Examples of conditional probability density functions for models trained on the \( \text{wmix} \) dataset with \( d_x = 3 \) (left) and \( d_x = 15 \) (right). Each distribution is conditioned on an \( x \)-value from the test set. KDE was used for CGAN and GMMN.
For **power** and **housing**, the GMMN model did not use the learned feature transform before the kernel (\( \xi_x \) in section 3.1.3). Substantially better performance was achieved by just tuning the kernel scale parameter \( \sigma_k^2 \) using grid search. The complete set of hyperparameters considered for GMMN were: learning rates in \( \{0.001, 0.0001\} \), regularization coefficients in \( \{0, 0.0001\} \), kernel scales for \( x \) in \( \{0.1, 1, 10\} \) and kernel scales for \( y \) in \( \{0.1, 1, 10\} \). Note that separate kernel scales were used for \( x \) and \( y \), meaning that all combinations of these were tried in the grid search.

### 4.5.2 Results

Results for the **microwave** dataset are reported in table 4.11 and plots of samples can be seen in figure 4.5. The metrics are here reported for all variants of CGANs, with ◇ denoting the version that achieved highest log-likelihood on the validation data. There were considerable differences between evaluation runs on the **microwave** dataset. For some models the standard deviation across 10 KDE-based log-likelihood estimates was above 0.05. Training times on **microwave** were similar to those described for medium-sized networks in table 4.6.

<table>
<thead>
<tr>
<th></th>
<th>Log-like.</th>
<th>( \hat{D}_{LS} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baselines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussian Process</td>
<td>-1.58 (-1.59)</td>
<td>16.0</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>-1.55 (-1.56)</td>
<td>14.6</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>-0.49 (-0.16)</td>
<td>24.8</td>
</tr>
<tr>
<td>MDN</td>
<td>-0.50 (-0.20)</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>DCTD</td>
<td>-0.54 [-0.36]</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>CGANs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>-0.65 ◇</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>KL</td>
<td>-0.66</td>
<td>24.8</td>
</tr>
<tr>
<td>Reverse KL</td>
<td>-0.97</td>
<td>17.7</td>
</tr>
<tr>
<td>Neyman ( \chi^2 )</td>
<td>-1.57</td>
<td>14.9</td>
</tr>
<tr>
<td>Pearson ( \chi^2 )</td>
<td><strong>-0.47</strong></td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>Squared Hellinger</td>
<td>-0.70</td>
<td><strong>24.9</strong></td>
</tr>
<tr>
<td>Jensen-Shannon</td>
<td>-1.64</td>
<td>16.3</td>
</tr>
<tr>
<td>Least Squares</td>
<td>-0.73</td>
<td><strong>25.0</strong></td>
</tr>
<tr>
<td>GMMN</td>
<td>-0.72</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Table 4.11: Results from experiment on the **microwave** dataset

<table>
<thead>
<tr>
<th></th>
<th>Log-like.</th>
<th>( \hat{D}_{LS} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>power</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log-like.</td>
<td></td>
<td>( \hat{D}_{LS} )</td>
</tr>
<tr>
<td>NN Reg.</td>
<td>0.06 (0.09)</td>
<td>24.5</td>
</tr>
<tr>
<td>Het. NN Reg.</td>
<td>0.14 (0.15)</td>
<td>24.6</td>
</tr>
<tr>
<td>MDN</td>
<td><strong>0.25</strong> (0.23)</td>
<td><strong>24.9</strong></td>
</tr>
<tr>
<td>DCTD</td>
<td><strong>0.25</strong> [0.25]</td>
<td><strong>24.9</strong></td>
</tr>
<tr>
<td>CGAN</td>
<td>0.11</td>
<td>24.6</td>
</tr>
<tr>
<td>GMMN</td>
<td>0.16</td>
<td>21.9</td>
</tr>
</tbody>
</table>

Table 4.12: Results from experiments on **power** and **housing** datasets

Table 4.12 lists results for the **power** and **housing** datasets. Standard deviations for the log-likelihoods were here below 0.02. The best performing CGAN version was Pearson \( \chi^2 \) for **power** and Least Squares for **housing**.
4.5. Real datasets

Figure 4.5: Samples from models trained on the microwave dataset
4.5. Real datasets

The training time on the housing dataset for CGANs, including GMMN, was 1 hour. DCTD took 4 hours and other neural network baselines took 30 minutes. Training on the power dataset required about half as much time for all models.

4.5.3 Discussion

A number of CGANs and baseline models do a good job in learning the distribution of the microwave dataset. Because of the high standard deviation in log-likelihoods, the results are best assessed by looking at figure 4.5. Based on the plots, the models can be divided into two groups. Some models have failed to capture the heteroskedasticity in the data distribution, making similar predictions for all $x$-values. Remaining models manage to capture the data distribution nicely. It is hard and not particularly meaningful to say which one performs the best among these.

The MDN baseline shows the most solid performance across the power and housing datasets. For these the CGAN model performs slightly worse than has been observed in earlier experiments. This could be traced back to the CGAN discriminator struggling in training. As a result the generator was not provided with very useful gradients. CGAN still outperforms the simplest baselines, but does not keep up with MDN or DCTD.

With larger real world datasets, like power and housing, very little is known about the underlying distribution. For example, it is not known if the conditional distributions are unimodal or contain multiple modes. The good performance of MDN in table 4.12 might indicate that the conditional distributions are well described by Gaussian mixtures, but no clear conclusions can be drawn. In applications, it is desirable to not have to reason about these properties of distributions. A versatile probabilistic regression model should adapt to general distributions without any manual modelling. In this experiment, the MDN and DCTD models seem to do a better job at this than CGAN. The shortcomings of CGAN are however more related to the training process than any inherent problem with the implicit generative approach. This is also exemplified by the GMMN performing surprisingly well in terms of log-likelihood, even surpassing the other CGAN on power.

A number that stands out in table 4.12 is the log-likelihood for DCTD on housing. There is a large discrepancy between the KDE-estimate and the estimate from importance sampling. It is not clear why this happened, but it can be noted that the KDE-estimate had a high standard deviation (above previously noted 0.02). No conclusions about the model itself should be drawn from this, but it does demonstrate issues of having to rely on estimates for quantitative evaluation metrics.
This chapter features an overarching discussion of the thesis work. Results of individual experiments have been discussed in corresponding sections in the Experiments chapter. A more general discussion around the implications of these results is presented in section 5.1. The suitability of the method used is critically assessed in section 5.2. In section 5.3 societal aspects are explored, putting the work in a wider context.

5.1 Results

The conducted experiments provide insight into the CGAN model and its applicability for regression. The usefulness of the model and practical considerations are discussed here.

5.1.1 CGAN Training Objectives

A number of different ways to train CGANs were introduced in section 2.3 and many of these also evaluated in the experiments. The results of these evaluations indicate which training objectives are useful in practice. This can hopefully offer some guidance in future applications of CGANs to regression problems.

As has been noted many times in earlier works \[38\] \[4\] \[64\], the standard GAN formulation of Goodfellow et al. \[18\] comes with some practical issues. In experiments, the standard CGAN performed well on small synthetic datasets. It was also included in later experiments, but never performed good enough on validation data to be selected in model selection. Although the standard CGAN can be used as a starting point, it is likely that better performance can be achieved with other variants. For most types of CGANs, their implementation is as simple as the standard version. This was demonstrated in section 3.1.2, where the differences only came down to the functions \(g_f\) and \((f^* \circ g_f)\). These are very simple to implement for many types of CGANs.

Through the \(f\)-GAN framework, a number of different ways to train CGANs using \(f\)-divergences have been considered. These have shown varying performance in practice. Some divergences, including Neyman \(\chi^2\) and Reverse KL, have shown poor results and should not be used in CGAN training in their current form. Squared Hellinger, Pearson \(\chi^2\) and KL did stand out as the most useful divergences. A question that arises from these results is to what extent the behaviour of each \(f\)-GAN is connected to any theoretical properties of correspond-
5.1. Results

5.1.1 f-divergence. As an example, KL and Reverse KL are divergences that represent different ways for the model distribution to match the data distribution [19, p. 73-76]. These properties are however not reflected in the results for corresponding CGANs. The differences between CGANs seem more connected to the gradients produced by the exact loss function than any underlying f-divergence. While the f-GAN framework offers a nice theoretical foundation, its usefulness for deriving good CGAN training objectives is limited.

For many of the divergences, a problem observed in training was sudden spikes in the loss function. This was believed to be caused by a combination of the CGAN training dynamics and the specific loss functions for these divergences. Such spikes would lead to a setback in training, but the networks would usually recover after some epochs. The issue could possibly be diminished by the use of gradient clipping [42]. Gradient clipping restricts the magnitude of gradients to prevent this type of training setbacks [19, p. 413-415].

Based on all experiments, it would be useful to determine the best CGAN training objective that should be used in future applications. Although there is not a clear single choice, both Pearson χ² and Least Squares CGAN have performed solidly overall. Especially for x and y of more than one dimension, these are good stable choices. It is worth investigating what these two might have in common. The motivation for each is very different, with Pearson χ² coming from the f-GAN framework and Least Squares being more practically motivated. One thing that can be noted is the common use of second order polynomials in the discriminator loss. Least Squares uses \((f^* \circ g_f)(v) = \frac{1}{2}v^2\) and Pearson χ² uses \((f^* \circ g_f)(v) = v(\frac{1}{4}v + 1)\). The \((f^* \circ g_f)\)-function is applied to the discriminator output for generated samples. Mao et al. [38] point out that this form of loss function is not flat anywhere and hence never produces near-zero gradients. The LS-CGAN also has similar, squared loss formulations for real samples and for the generator training, whereas Pearson χ² uses \(g_f(v) = v\) for these. The fact that Pearson χ² often outperformed Least Squares indicates that there is more to the story than just using squared values in the loss functions. Further investigation into these loss formulations and how they affect gradients could be useful for improving CGAN training.

5.1.2 MMD and GMMN

Results for the GMMN model have been presented separately from other CGANs due to the different training setup used. This helps to shed some light on MMD as a training objective for CGANs. The performance of GMMN is largely underwhelming. Results on the small synthetic datasets are acceptable, but on wmix the metric values are not close to other CGANs. The model also struggled to learn anything useful from trajectories datasets.

The GMMN model used is one of the more simple approaches to MMD in CGANs. It is likely that better results could have been achieved using a more refined model. One straightforward extension is to utilize the deep kernel learning discussed in section 2.4.2 to create more powerful kernel representations. A nice property of GMMN is its closed form generator loss. Introducing kernel learning changes the training to a minmax problem, like other CGANs. This is already the case with the learned scaling transformation \(\xi_p\), but this is a very simple change to the kernel. Recall however that GMMN worked better on real datasets without learning \(\xi_p\). This indicates that only adding deep kernel learning to the model might not lead to improvements. There are also possibilities to use GMMNs together with autoencoders [34]. This type of extension allows for matching distributions in transformed feature spaces instead of directly in the data space.

In section 2.4 two different approaches were mentioned for extending the unconditional GMMN model to the conditional case: CGMMN and joint GMMN. Initially in the thesis project the CGMMN model was implemented and used for some experiments. That model was however very sensitive to hyperparameter values. In particular, the model only performed well when the value \(\lambda\) (which regularizes matrix inverses) was set very high. This should only be a small regularization, so having to set this to a high value fundamentally changes the model. The original authors imply that \(\lambda\) should be kept small [48]. A quick numerical
experiment can show that when $\lambda$ takes a large value, $K^{-1} = (K + \lambda I)^{-1} \approx cI$ for some constant $c$. Inserting this into the conditional MMD in eq. 2.65 makes the CGMMN model reduce to joint GMMN (up to a constant scaling of the loss). This does explain the changed behaviour for an ill-chosen regularization parameter.

The fact remains that it was not possible to use the CGMMN model in its original form. Compared to joint GMMN, it also features a more complex loss function with matrix inverses. The computational burden of these can be reduced by reusing the inverses between epochs, but this relies on the gram matrices not changing. This severely complicates any kernel learning extensions. Judging by the better performance and more straightforward loss function of joint GMMN it is hard to see why the CGMMN approach should be necessary. The model does build on some nice theoretical ideas, but the joint GMMN approach is also completely consistent with underlying theory. Overall this warrants further investigation and comparisons. In particular, it would be useful to consider the joint GMMN approach for experiments where CGMMN has shown good results [48]. The possibility that the poor performance of CGMMN observed in this work was caused by some implementation detail should also not be ruled out.

5.1.3 CGAN Regression Compared to Baseline Models

The body of results presented in this thesis allows for drawing some conclusions about CGAN as a probabilistic regression model. Taking all experiments into account, CGAN is competitive with the most advanced of the baseline models. The more simple baselines of GP and the neural network regression model are consistently outperformed in terms of KDE-estimated log-likelihood. Poor performance on some of the real datasets does however show that consistent top results should not be expected across all datasets. The motivation for using CGANs for regression was to capture complex noise distributions. When such noise is present the difference to simple baselines is most apparent. While the conducted experiments give major insights into CGAN regression, it is not possible to draw general conclusions from only a few datasets. More applications of CGANs on regression datasets, in particular from the real world, is necessary to fully judge the usefulness of the model.

Since all baseline models are not consistently outperformed, it might not be clear if CGAN regression offers much of an improvement over current methods. It should however be noted that the best baselines are complex probabilistic models in themselves. The typical application of neural networks to regression problems follows the homoskedastic Gaussian model (NN Reg.). When data uncertainty should be quantified, the heteroskedastic version might see some use. In this probabilistic setting CGAN consistently outperforms these. An argument can be made that not just CGAN, but also MDN and DCTD, are strong probabilistic regression models that should be considered by practitioners.

Some things should be noted about how CGAN compare to the GP baseline. On the most simple synthetic datasets the performance is similar. For more complex distributions the GP falls behind. This should be expected, as the standard GP model used is restricted to modelling a homoskedastic Gaussian. There is no comparison available for larger datasets as the GP baseline was left out. It is hard to compare CGAN with GP as they fill somewhat different roles. CGANs can learn to capture data uncertainty using complex conditional distributions. GPs can learn complex relationships between $x$ and $y$, but only represent Gaussian distributions. Being Bayesian models, GPs also capture model uncertainty. There exists extensions to the standard GP model that represent a wider range of distributions [47], p. 191 [17] [46] and that scales to larger datasets [47], p. 171-184 [57]. Comparing these to CGAN regression would be of great interest.

5.1.4 Using CGAN as a Regression Model

CGAN shows some viability as a regression model, making it interesting to consider possible use cases. There are at least two general settings where a CGAN regression model could be
useful. The first is as a complete predictive model, where predictions would be presented to a user. A trained CGAN could generate a whole set of samples from the predictive distribution $p_g(y|x)$. It would then be up to the user to utilize the prediction and interpret the uncertainty. Presenting the samples in a useful way would be important, but also highly application-specific.

The second setting is using CGAN regression as part of a bigger system. In particular, a CGAN can be used for approximating a conditional expectation of some general function $t$:

$$
E_{y \sim p_d(y|x)}[t(y)] = E_{y \sim p_g(y|x)}[t(y)] \approx \frac{1}{N} \sum_{i=1}^{N} t(y^{(i)})
$$

(5.1)

$$
y^{(i)} \sim p_g(y|x) \quad \forall i \in \{1, \ldots, N\}.
$$

(5.2)

Here $p_g$ is the generator distribution of a CGAN trained to estimate the data distribution $p_d$. Each $y^{(i)}$ is easily generated by just propagating noise and $x$ through the generator. This type of Monte Carlo estimate could be useful in intermediate steps of larger systems. Any function $t$ could be used, it does not even have to be known when the CGAN itself is trained. For example, statistics like mean ($t(y) = y$) and higher order moments ($t(y) = y^n$) can be approximated this way. Note that the same type of estimate is possible for any model that can be sampled from. The benefit of using CGANs comes from the unrestricted expressiveness of implicit distributions.

Both of the described use cases only rely on the generative property of CGANs. An explicit probability density function is not required. With this in mind, it makes sense to always compare true model distributions, even when some might be implicit. Put another way, the CGAN distribution of interest is truly the implicit $p_g$ and not the explicit distribution resulting from KDE. Still, there might be some use case where probability density explicitly needs to be computed. In these cases a CGAN model would have to rely on a KDE-estimated probability density function. Using a CGAN in this setting would however be a poor choice of model. An explicit-likelihood model like MDN would be a far better option.

5.2 Method

The method used for this study is critically evaluated in the following sections. This should help to assess whether the results and conclusions are valid. Methodological improvements for future work on CGANs will also be discussed.

5.2.1 Experiment Design

The synthetic datasets have been created to evaluate specific properties of the CGAN regression model. These have nicely covered many interesting aspects. The trajectories and wmix datasets were specifically created to evaluate CGANs on different dimensionalities of $x$ and $y$. While these gave interesting insights, only one type of distribution was considered in each experiment. To draw general conclusions about the models and higher-dimensional $x$ and $y$, a whole range of diverse datasets should be used. Doing so is however outside the scope of this thesis.

In machine learning experiments, hyperparameters and the methods used to tune these are of great importance. Hyperparameters need to be set to suitable values for all models in order to get fair comparisons. A somewhat more refined approach, which has been used in this work, is to design a principled process for tuning these. When this method is used, it is important that similar attention is put into the hyperparameter tuning of all models. It is sometimes possible to substantially improve the results of a model by very fine-grained hyperparameter adjustments. If the same is not done for other models, the obtained results do not indicate any advantages of the model itself, but of the hyperparameter tuning process. The hyperparameters used throughout this thesis are believed to be suitable for the considered models. The grid search used for key parameters is a principled approach that should assure
that no model is severely impeded by poor hyperparameter choices. Observed sensitivity of
different models to specific parameters has also been taken into account. It is believed that no
model has received disproportionate attention in hyperparameter tuning and performed better
because of it.

For many of the models considered, especially the ones based on neural networks, the
space of possible hyperparameters is vast. It includes both parameters innate to the models
themselves, but also choices impacting the training process. Exhaustively exploring this hy-
perparameter space is infeasible, since it would be a high-dimensional optimization problem
in itself. Therefore choices have to be made about which parameters and values to consider.
These choices have been based both on existing literature and small initial experiments. Often
the learning rate and regularization coefficient were the key hyperparameters identified. Pa-
rameters related to optimizers were not explored in detail in this thesis. Doing so could lead
to improved results and faster convergence for some models.

5.2.2 Evaluation

As explained in section 2.6.3 the evaluation of generative models is complicated and there
is often not one correct approach. In this work, KDE-based log-likelihood has been used.
The KDE approach has been shown to be problematic for high dimensional \( y \) [52]. This was
however not deemed to be much of a problem in the low-dimensional regression setting. Even
if KDE is accepted as a method, there are details to it that can vastly change the log-likelihood
estimates. These are mainly connected to the scale parameter \( \sigma_k^2 \) and the number of samples
drawn from the model. It has been noted that the scale parameter is crucial for getting accurate
KDE-estimates. Choosing it too high leads to log-likelihoods far from the true values. The
approach used, where \( \sigma_k^2 \) is determined from validation data, largely gets around this problem.
This solution has been widely used in literature [18][42][34]. More refined methods exists,
where \( \sigma_k^2 \) is chosen separately for each sample [58]. Such adaptive methods have however not
seen much use in the GAN-literature.

The number of samples used in KDE also impacts the approximation noticeably. A very
large number of samples can easily be used in the unconditional case, but for conditional
distributions this can become problematic. Each \( x \) induces a different conditional distribution,
meaning that new samples have to be generated for KDE. The entire KDE-based log-likelihood
computation thus needs to be repeated for each data point in the test or validation set. This
can take a long time and require large amounts of memory if many samples are used. It should
however be noted that the procedure can be sped up substantially by clever implementation,
making use of fast matrix operations on the GPU. Aggarwal et al. [2] use 100 samples for
KDE-estimates of conditional distributions. In this thesis at least 1000 is used for evaluating
on the test set. There are indications that even more samples would lead to somewhat better
estimates. This is particularly true for the trajectories experiment, where \( y \) is not one-
dimensional. It has however been pointed out that for a fixed kernel scale the improvements
from drawing more samples saturate [59]. Drawing enough samples to get KDE-estimates
arbitrarily close to the true log-likelihood values is therefore infeasible. Future works should
determine when the improvements start to saturate or fix the number of samples to a very
high value. Memory restrictions might in that case become a factor. Drawing conclusions
from KDE-estimates with as few as 100 samples should be done carefully.

A greater issue with the KDE-estimates were the cases where the kernel scale \( \sigma_k^2 \) had to
be restricted above some value. The reasoning for this was initial log-likelihood results of
\( -\infty \) for multiple models. It seems unintuitive that the same \( \sigma_k^2 \) that maximized log-likelihood
for validation data should produce such a bad value on the test set. The explanation for this is
single outlier data points in the test data. Such data points could be faulty out-of-distribution
values. They could also be from areas of the sample space with small-enough probability mass
to not be represented in the validation data. It should be noted that it is enough for a single
data point to numerically results in a log-likelihood of \( -\infty \) for the entire metric to take the
value $-\infty$. The situation can be viewed as $\sigma^2_k$ overfitting to the validation set. Solving this by explicitly removing outliers from test data would introduce biases in evaluation. Setting a lower limit to the kernel scale was seen as a better option. This could be interpreted as a form of regularization for the $\sigma^2_k$-parameter. High kernel scales does however lead to poor KDE-estimates and high standard-deviations across repeated evaluations.

The situation described above ultimately stems from a mismatch in support between the data distribution and the model distribution. There are areas from which data samples are produced, but where the model distribution has zero or near-zero probability mass. In these cases the log-likelihood correctly takes the value $-\infty$. It is not obvious how the difference between two distributions with different support should be measured. Also many $f$-divergences are undefined for this case, since the ratio $p(x)/q(x)$ in eq. 2.31 can result in division by zero. This is however not a problem in practice, if the $f$-GAN framework is considered. Training a discriminator for estimating an $f$-divergence is still possible when there is a mismatch in support. This can be a useful property of the estimation procedure, but it also means that the theoretical connection to the divergence is lost. It can be noted that Integral Probability Metrics do not suffer from these issues when there is a mismatch in support. Estimated MMD or Wasserstein distance could therefore be useful as evaluation metrics in these cases.

Throughout the experiments all models have been compared mainly based on KDE-estimated log-likelihood. This despite the fact that true log-likelihoods exist for many baseline models. The KDE-estimated values are overall lower than the true ones. Comparing the KDE-estimates of CGANs to the true log-likelihoods of baseline models would therefore not be particularly useful. Comparing KDE-based values for both is more fair, but using only approximate values makes it hard to draw any concrete conclusions. It should also be noted that the discrepancy between KDE-estimated and true log-likelihood could be different between models. For some type of model KDE might get very close to the true distribution, whereas the discrepancy could be large for some other type. This would introduce a bias in the evaluation. It is therefore important to use additional evaluation methods.

The secondary metric $\hat{D}_{LS}$ was also used for all experiments. This offered a secondary evaluation score to the log-likelihood. The usefulness of the metric was widely varying, often with many models sharing values around 25.0. This represents the best possible regression model, where the trained discriminator completely fails to distinguish between real and generated samples. The values of $\hat{D}_{LS}$ were still useful in some cases, like for trajectories datasets. The choice of using the Least Squares CGAN formulation for the metric can be seen as quite arbitrary. This was mainly motivated by a typically stable training process. The question arises whether this makes the metric biased towards the LS-CGAN model. The experiments show no clear evidence of this, which agrees with the results of Im et al. [28], who introduced the metric. A more in-depth study of evaluation metrics should consider divergence estimates using each type of CGAN formulation.

It would be desirable to draw some general conclusions about which evaluation metric is most useful for evaluating CGANs. Log-likelihood gives values that can easily be compared between models. It does however suffer from the fact that KDE-estimates have to be used. For CGANs there is also a mismatch between the objective used in training and how the model is evaluated. The $\hat{D}_{LS}$-metric is more similar to the CGAN training, but instead creates such a mismatch for the baseline models. The dataset-specific metrics $E_\sigma$ and $\Psi_{Support}$ were designed to target specific properties. These both gave useful insights. Echoing the sentiment of Theis et al. [59], the best choice of evaluation metric depends on the application. Designing specific metrics for the problem at hand can be a good approach.

5.2.3 Replicability and Reliability

Models and algorithms considered in this thesis have strict, mathematical definitions. This strongly supports the replicability of the study. The exact behaviour of each model will also
depend on implementation details. While such details are not described in this thesis, they can be found in the accompanying code repository.

There is substantial amounts of randomness involved in many of the methods considered. The most obvious case of this is the CGAN model itself, that relies on generated random noise. The impact of randomness is to a large extent negated by the scale of the experiments, making the sampling distribution a far more important factor. This choice is always well-described where applicable.

Neural network architectures and hyperparameter choices have been described in great detail throughout the thesis. This should both act as a useful starting point for further studies, but also allow for easily reproducing the experiments. The first step of hyperparameter tuning, as described in section 3.2.6, is not described in detail. The goal of this step was to determine useful values to consider for model selection. These initial experiments were not as structured as the later grid search, so describing them in detail would likely not be particularly useful. Exact values of hyperparameters used in the grid search should be enough for reproducing the experiments.

The procedure used to generate the synthetic data is described in detail in section 3.3 and appendix C. The exact datasets are also made available together with the code. The real world datasets used are all openly available on the web. The simple preprocessing applied to these is also presented in section 3.3.

Some things should be noted about the reported training times. Firstly, these clearly depend heavily on the hardware used. The computer used for training was described in the beginning of chapter 4. Timing of the training runs was not done in an isolated, controlled computing environment. There were other processes running and sometimes multiple models were trained in parallel. The training processes were however not starved for CPU or GPU time. Exact training times are also heavily dependent on implementation details and the frequency of auxiliary tasks like logging and saving network parameters. The presented training times should therefore be seen as rough estimates of what can be expected. They are mostly useful for comparing the exact implementations of the models in this work.

### 5.2.4 Source Criticism

A majority of the references used are papers published in peer-reviewed journals and conference proceedings. Published books by well-known researchers are also cited. These are all seen as trustworthy sources.

A number of unpublished pre-prints have been cited. Relying on pre-prints has been necessary to cover certain areas of the theory and to present current research. These papers have been more critically scrutinized before being cited. When open reviews have been available, the presented criticism has been taken into account. It is not feasible to personally verify all claims of the pre-print papers. Most of these have been judged as reliable due to being highly cited, also from published, peer-reviewed papers. Some referenced pre-prints warrant some additional justification: The survey does not present any new contributions, but offers only an overview of GAN research. The paper proposing the DCTD model and the initial work on CGAN as a regression model are viewed from a critical perspective. Claims from these have been assessed based on other reviewed literature and results from experiments in this thesis.

A couple of sources from the web have also been referenced. These are both authored by distinguished researchers with established knowledge within the concerned area.

5.3 Societal Context

The results of this thesis have no direct connection to any specific application area. Still, there is value in considering also the indirect ways that the work can affect society. Such societal aspects are discussed in this section. With the topic of this thesis not being very applied, it is hard to focus in on concrete societal effects or ethical dilemmas. Any future applications of CGAN regression should therefore thoroughly consider these aspects also in the context of the application.

5.3.1 Uncertainty in Machine Learning Systems

As machine learning systems see use in many real world scenarios, there is an increased need for more detailed predictions. In many cases it is not enough to get a single predicted value and trust it blindly. It is important to also consider questions like: How uncertain is this prediction? What are other likely outcomes? This is especially true in many safety-critical applications, including autonomous driving and other areas of robotics. Systems in these areas should be designed with these questions in mind. This requires powerful tools for probabilistic modelling. The work on CGAN regression is an attempt to create such tools.

There are cases where poor predictions from machine learning systems have caused bad accidents [30]. Better modelling of uncertainty could prevent this from happening in the future. Recall from the introduction of this thesis the two kinds of uncertainty, data uncertainty and model uncertainty. A system acting based on model predictions needs to take both of these into account. Model uncertainty can additionally be used to detect situations that the model has not been sufficiently trained on. While this thesis has mainly been concerned with data uncertainty, a good model for one can lead to benefits for both types of uncertainty. There are indications that when only model uncertainty is properly modelled, it starts to compensate for the ignored data uncertainty [30]. This in turn leads to inaccuracies in model uncertainty. There is therefore great value also in properly modelling the data uncertainty. The combination of powerful models for both types of uncertainty is thus integral for building more safe and trustable machine learning systems.

5.3.2 Energy Usage and Environmental Impact

Deep neural networks have become a central part of many modern machine learning models. This has allowed for building large models that require powerful hardware and long training times. The energy usage of training these deep networks is not negligible. In most cases the electricity used is not derived from carbon-neutral sources. There is thereby an indirect environmental effect of training the models. While training a single network might not require enormous amounts of energy, developing a model often requires retraining many times. This is necessary for finding useful neural network architectures and hyperparameter values. Strubell et al. [55] have estimated the total CO$_2$ emissions of training some of the largest models used in natural language processing. With extensive experimentation and hyperparameter tuning taken into account, the total emissions are estimated to be comparable to the amount of CO$_2$ emitted by a car throughout its lifetime. While the emission values depend heavily on the composition of energy sources used for electricity generation, this exemplifies that the environmental impact should not be ignored.

The models considered in this thesis are small and training times relatively short. Substantial amounts of experimentation and hyperparameter tuning has however resulted in many networks being trained. None of these have any use outside of this thesis project. Still, the total emissions of the networks trained here are probably far from those of a car. The environmental aspect becomes increasingly important as the methods explored in this thesis are scaled up to larger models and larger datasets.
The negative environmental impact of machine learning is however not the entire story. As large parts of society move to become more sustainable, machine learning systems have the potential to enable and accelerate this change \[49\]. There are many opportunities in areas such as transportation, electricity distribution, space heating and agriculture to use smart solutions to reduce the environmental impact. These opportunities include the use of machine learning both in empowering research and in practical applications. Predicting the energy output of a power plant, as was done for the power dataset in this thesis, is an example of how machine learning can assist in optimizing energy systems. In many of these cases, the potential positive environmental effects greatly overshadow the emissions from training large machine learning models.
Conclusions

This thesis offers meaningful insights into the CGAN model and its use for regression problems. The work has been grounded in theory through a detailed presentation of CGANs and other relevant concepts. Different versions of GANs have been extended to the conditional setting. This includes CGANs trained using general $f$-divergences and MMD, both of which have got little attention in existing literature. CGANs have been compared to a number of different baseline models on both synthetic datasets and data from the real world. These experiments have shown that CGANs possess the ability to estimate many types of complex conditional distributions. Results concerning hyperparameters and network architectures also offer some guidance for future applications and research. The stated aim of the thesis has thereby largely been achieved. Detailed answers to each research question will be presented in section 6.1.

The conclusions drawn from this thesis should support practitioners in future applications of CGANs. Experimental results show which CGAN versions are more likely to be useful. The methodology used also offers a good starting point for designing training and evaluation procedures.

This thesis has raised multiple new questions that could fuel further research within the area. On the more theoretical side this includes further exploration of the CGAN model itself. Perhaps even more interesting would be more applied research, using CGANs for regression problems within application areas. Details on some specific directions for future work will be given in section 6.2.

6.1 Answers to Research Questions

Concrete answers to each research question are presented below.

How well can CGANs approximate simple and complex data distributions?

CGANs can approximate many different kinds of distributions. Experiments on synthetic datasets have shown that distributions from known families, like Gaussian, Laplacian and exponential distributions, can be learned by the model. The same has been demonstrated for more complex distributions without a known probability density function. Also properties like heteroskedasticity and multimodality can be well modelled by CGANs without any
extensions to the model. Comparing these capabilities of CGANs to other probabilistic regression models, performance is on par with the advanced baselines. Simpler baseline models like Gaussian processes and neural network regression are consistently outperformed on more complex distributions.

Nothing indicates that the capabilities of CGANs deteriorate as the dimensionality of $x$ and $y$ increases. In the conducted experiments CGANs clearly outperformed baseline models for $y$ above one dimension. More research is however needed to conclude if this holds in general.

How does the use of different training objectives and neural network architectures impact the training process and the capability of CGANs to approximate distributions?

The noise-injection neural network architecture has shown to work well on many different datasets. The exact architecture is not of huge importance, as the double-input variant also has proven to be useful. It is however necessary that the conditioning variable $x$ and noise $z$ are not combined too late in the generator architecture. Having multiple layers that depend on both inputs has shown to be important.

The training objectives corresponding to Pearson $\chi^2$ and Least Squares CGAN have empirically performed the best. These typically also have a relatively stable training process. When less suitable training objectives are used the data distribution can be poorly approximated.

Training CGANs using MMD has led to acceptable results. The resulting GMMN model is however rarely competitive with the previously mentioned CGAN versions. Some extensions do exist that could bridge this gap.

How does CGANs compare to alternative probabilistic regression models on real world datasets?

CGAN regression has achieved varying results on the considered real world datasets. On the smaller microwave dataset CGAN was competitive with the best baselines, but on the larger power and housing datasets the model performed worse. Even on these datasets the CGAN model did outperform simple neural network regression. Other baseline models did however achieve substantially better results. While this shows a negative result for these datasets in particular, it is hard to draw any general conclusions from it. More work is needed to better evaluate the usefulness of CGAN regression on real world data.

6.2 Future Work

While this thesis has covered many aspects of using CGANs for regression, there are still a few questions that have gone unexplored. Different neural network architectures have been briefly covered, but there is still a large space of possible architectural choices that have not been considered. This includes special network types for structured data, such as convolutional neural networks [19, p. 330-334] and recurrent neural networks [19, p. 378-394]. Another unexplored question is how the CGAN performance changes with the amount of training data. The question was briefly touched upon by Aggarwal et al. [2], but it would be of interest to further investigate this on more complex datasets. In addition to these unexplored aspects, it would also be valuable to repeat experiments from this thesis on more datasets. Considering additional real world datasets should then be prioritized.

An exciting direction for future work is to further explore how conditional generative models can be trained using MMD. The implemented GMMN model is a simple approach with many possible extensions, as discussed in section 5.1.2. Using joint kernels that decompose into one kernel for $x$ and one kernel for $y$ allows for treating these differently. This could open up for exciting applications, where $x$ is some complex type of data while $y$ is still low-dimensional.
If $x$ for example is an image, a pretrained feature extractor network could be utilized as the transform $\xi_{\theta}$ for $x$ in deep kernel learning.

The CGAN regression approach explored in this thesis has been entirely focused on data uncertainty. A useful continuation would be to combine this with methods for quantifying model uncertainty. Such a complete approach would allow for reasoning about whether predictions are uncertain due to the inherent randomness of a task or due to a lack of relevant training examples. A possible way to tackle this problem would be through the use of Bayesian neural networks. Bayesian methods have successfully been applied to GANs in earlier works [50]. The perspective of separating and quantifying different types of uncertainty in CGANs has however not received much attention.
Bibliography


[57] Jinxuan Sun, Guoqiang Zhong, Yang Chen, Yongbin Liu, Tao Li, and Kaizhu Huang. “Generative Adversarial Networks with Mixture of t-Distributions noise for diverse image generation”. In: *Neural Networks* 122 (2020).


A Additional Experiments

This appendix includes some small-scale experiments related to the CGAN model. These are not central to the research questions of the thesis, but offer some guidance for practical applications of CGANs. Section A.1 describes an experiment exploring different noise distributions for the CGAN generator. In section A.2 the dimensionality of the noise is instead considered. Different choices for activation functions in the CGAN networks are explored in section A.3.

A.1 Noise Distribution

Different distributions can be used for the noise input of the CGAN generator. To better understand how this choice impacts performance, multiple CGANs were trained using different noise distributions.

A.1.1 Experiment Setup

The different noise distributions considered were:

- A standard Gaussian distribution
- A uniform distribution on [0,1]
- An exponential distribution with rate parameter 1

For each of these distributions both a standard and least squares CGAN model were trained. This was repeated for the exponential, heteroskedastic and bimodal datasets. The medium noise-injection CGAN architecture was used and training was done for 2000 epochs with a learning rate of 0.0001. This learning rate had previously been determined to be suitable for that network architecture across different loss functions and datasets. The trained models were evaluated using KDE-estimated log-likelihoods.

A.1.2 Results

Resulting log-likelihoods for different noise distributions are reported in table A.1. The poor results in the last column for some of the models were due to mode collapse, where only one
of the modes of the bimodal distribution was properly modelled by the CGAN. This fact was determined by plotting samples from the models.

<table>
<thead>
<tr>
<th>CGAN</th>
<th>Noise Distribution</th>
<th>exponential</th>
<th>heteroskedastic</th>
<th>bimodal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Gaussian</td>
<td>0.59</td>
<td>-0.48</td>
<td>-1.71</td>
</tr>
<tr>
<td></td>
<td>Uniform</td>
<td>0.59</td>
<td>-0.47</td>
<td>-0.78</td>
</tr>
<tr>
<td></td>
<td>Exponential</td>
<td>0.60</td>
<td>-0.46</td>
<td>-1.69</td>
</tr>
<tr>
<td>Least Squares</td>
<td>Gaussian</td>
<td>0.62</td>
<td>-0.45</td>
<td>-0.77</td>
</tr>
<tr>
<td></td>
<td>Uniform</td>
<td>0.61</td>
<td>-0.45</td>
<td>-0.78</td>
</tr>
<tr>
<td></td>
<td>Exponential</td>
<td>0.60</td>
<td>-0.46</td>
<td>-1.76</td>
</tr>
</tbody>
</table>

Table A.1: Test log-likelihoods for CGANs using different distributions for the noise fed to the generator.

A.1.3 Discussion

As can be seen in table A.1, there are almost no differences in performance between the considered distributions. The CGAN generator seems to always be able to transform the noise to fit the distribution of the dataset. The exceptions to this are the cases of mode-collapse on the bimodal dataset. It is however unclear if this is related to the noise distribution or other hyperparameters. Note for example that the standard CGAN with Gaussian noise has this issue, but for the Least Squares CGAN the Gaussian distribution results in the highest log-likelihood.

A.2 Noise Dimensionality

An experiment was done to test how the dimensionality of the noise vector $z$ impacts CGAN performance.

A.2.1 Experiment Setup

The noise in this experiment was sampled from a standard Gaussian. A Standard CGAN and a Least Squares CGAN were trained and evaluated on the exponential and bimodal synthetic datasets. The CGANs were trained once for each noise dimensionality in $\{1, 2, 3, 5, 10, 20, 50, 100\}$. Training followed the same setup as the noise distribution experiment in section A.1, but with the large double-input CGAN architecture.

A.2.2 Results

Figure A.1 shows the impact of noise dimensionality for the exponential and bimodal datasets.

A.2.3 Discussion

Figure A.1a shows a weak trend of higher log-likelihood for higher-dimensional noise on the exponential dataset. The improvement is however very small. A different behaviour can be seen in figure A.1b for the bimodal dataset. For this dataset the performance is very poor when the noise is one-dimensional. Better results are achieved with two-dimensional noise. Log-likelihoods are then somewhat constant as the dimensionality is increased further. This result for the bimodal dataset is quite intuitive. It is possible to think of the data distribution as having two independent noise components. One component selects between the two modes and one acts as the added uniformly distributed noise. Having at least two independent noise dimensions can therefore be helpful for the CGAN generator estimating this distribution.
A conclusions to draw from the experiment is that the noise dimensionality should be set sufficiently high, taking into account the complexity of the dataset. When very little is known about the data distribution, as is often the case with real world data, there is no direct drawback to setting the dimensionality too high. Better performance of CGAN models should however not be expected just from increasing the noise dimensionality.

A.3 Activation Functions

This experiment tested which network activation functions are suitable to use in CGANs.

A.3.1 Experiment Setup

The same experiment setup was used as for the noise dimensionality experiment, but the noise dimensionality was fixed to 5 and the bimodal and complex datasets considered. The activation functions ReLU, ELU and Leaky ReLU were tried. Only one function was used at a time for all layers in the networks. Activation functions were changed in both the generator and discriminator. Results were mainly evaluated visually, by plotting samples from the trained CGANs.
A.3.2 Results

Resulting scatter plots are displayed in figure A.2 and A.3. Log-likelihoods on the test set were similar for all activation functions and similar to those reported in the small synthetic datasets experiment (section 4.1).

![Figure A.2: Resulting scatter plots for CGANs trained on the bimodal dataset with different activation functions](image)

(a) Leaky ReLU, Standard CGAN
(b) ReLU, Standard CGAN
(c) ELU, Standard CGAN
(d) Leaky ReLU, Least Squares CGAN
(e) ReLU, Least Squares CGAN
(f) ELU, Least Squares CGAN

Figure A.2: Resulting scatter plots for CGANs trained on the bimodal dataset with different activation functions

A.3.3 Discussion

All the activation functions used resulted in CGAN models that captured the general distribution of the data. There are however some notable differences in the learned model distributions. For the Leaky ReLU case in figure A.2a, the fact that the noise is uniform is not very well captured. Also for the complex dataset the Leaky ReLU CGANs seem to struggle with learning the bimodal part of the distribution correctly. Models with ELU activations does for some $x$ only output a few values, instead of properly modelling the full conditional distribution. The same issue in the bimodal part of complex can be seen here. The distributions of ReLU CGANs have some small flaws for the bimodal dataset. All parts of complex are however learned very nicely, making ReLU stand out as a good choice for activation function in CGANs.
Figure A.3: Resulting scatter plots for CGANs trained on the complex dataset with different activation functions
Deriving the Loss Function for Heteroskedastic Regression

This appendix shows the derivation of the loss function used when training the heteroskedastic neural network regression model. The model distribution is a Gaussian with diagonal covariance matrix:

$$p(y|x) = \mathcal{N}(y|\mu(x), \text{diag}(\sigma_1^2(x), \sigma_2^2(x), \ldots, \sigma_{d_y}^2(x)))$$ (B.1)

and $$\{(x^{(i)}, y^{(i)})\}_{i=1}^{N_{\text{train}}}$$ is the training dataset. Recall that $$d_y$$ is the dimensionality of $$y$$ and to avoid clutter let

$$\mu^{(i)} \overset{\text{def}}{=} \mu(x^{(i)}) \quad (B.2)$$

$$\sigma_{j}^{(i)} \overset{\text{def}}{=} \sigma_{j}^2(x^{(i)}) \quad (B.3)$$

$$\Sigma^{(i)} \overset{\text{def}}{=} \text{diag}(\sigma_1^{2(i)}, \sigma_2^{2(i)}, \ldots, \sigma_{d_y}^{2(i)}) \quad (B.4)$$
where the dependence on the model parameters
regression loss function:
loss invariant to the size of the training set. All of this allows for defining the heteroskedastic
function. By dividing with the constant
Since this quantity should be maximized the sign has to be changed to turn it into a loss
where constants with respect to
The log-likelihood for the training data is then:
\[
\log \left( \prod_{i=1}^{N_{\text{train}}} p \left( y^{(i)} | x^{(i)} \right) \right) = N_{\text{train}} \log \left( \frac{1}{\sqrt{2\pi d_y}} \exp \left\{ -\frac{1}{2} (\mu^{(i)} - y^{(i)})^\top \Sigma^{(i)}^{-1} (\mu^{(i)} - y^{(i)}) \right\} \right) 
\]  
(B.5)
\[
= \sum_{i=1}^{N_{\text{train}}} \log \left( \frac{1}{\sqrt{2\pi d_y}} \exp \left\{ -\frac{1}{2} (\mu^{(i)} - y^{(i)})^\top \Sigma^{(i)}^{-1} (\mu^{(i)} - y^{(i)}) \right\} \right) 
\]  
(B.6)
\[
= \sum_{i=1}^{N_{\text{train}}} \left[ -\frac{1}{2} \log ((2\pi)^{d_y} | \Sigma^{(i)}) \right] - \frac{1}{2} (\mu^{(i)} - y^{(i)})^\top \Sigma^{(i)}^{-1} (\mu^{(i)} - y^{(i)}) 
\]  
(B.7)
\[
= -\frac{1}{2} \sum_{i=1}^{N_{\text{train}}} \left( d_y \log(2\pi) + \log \left( \prod_{j=1}^{d_y} \sigma_{y, j}^{(i)} \right) \right) 
\]  
(B.8)
\[
+ (\mu^{(i)} - y^{(i)})^\top \text{diag} \left( \frac{1}{\sigma_{y, 1}^{(i)}}, \frac{1}{\sigma_{y, 2}^{(i)}}, \ldots, \frac{1}{\sigma_{y, d_y}^{(i)}} \right) (\mu^{(i)} - y^{(i)}) 
\]  
(B.9)
\[
= -\frac{d_y N_{\text{train}}}{2} \log(2\pi) - \frac{1}{2} \sum_{i=1}^{N_{\text{train}}} \left[ d_y \log \left( \sigma_{y, j}^{(i)} \right) + \sum_{j=1}^{d_y} \frac{1}{\sigma_{y, j}^{(i)}} \left( \mu_{y, j}^{(i)} - y_{y, j}^{(i)} \right)^2 \right] 
\]  
(B.10)
\[
= \text{Const.} - \frac{1}{2} \sum_{i=1}^{N_{\text{train}}} \sum_{j=1}^{d_y} \left[ \log \left( \sigma_{y, j}^{(i)} \right) + \frac{1}{\sigma_{y, j}^{(i)}} \left( \mu_{y, j}^{(i)} - y_{y, j}^{(i)} \right)^2 \right], 
\]  
(B.11)
where constants with respect to \( \mu^{(i)} \) and \( \Sigma^{(i)} \) can be ignored since they do not impact gradients.
In the neural network case the reparameterization \( \alpha_{y, j}^{(i)} \overset{\text{def}}{=} \log \left( \sigma_{y, j}^{(i)} \right) \) is utilized. Using this the log-likelihood can be further rewritten as
\[
\log \left( \prod_{i=1}^{N_{\text{train}}} p \left( y^{(i)} | x^{(i)} \right) \right) = \text{Const.} - \frac{1}{2} \sum_{i=1}^{N_{\text{train}}} \sum_{j=1}^{d_y} \left[ \alpha_{y, j}^{(i)} + \exp \left( -\alpha_{y, j}^{(i)} \right) \left( \mu_{y, j}^{(i)} - y_{y, j}^{(i)} \right)^2 \right]. 
\]  
(B.12)
Since this quantity should be maximized the sign has to be changed to turn it into a loss function. By dividing with the constant \( N_{\text{train}} \) it is also possible to make the magnitude of the loss invariant to the size of the training set. All of this allows for defining the heteroskedastic regression loss function:
\[
L_{\text{Het}}(\theta) \overset{\text{def}}{=} \frac{1}{2N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \sum_{j=1}^{d_y} \left[ \alpha_{y, j}^{(i)} + \exp \left( -\alpha_{y, j}^{(i)} \right) \left( \mu_{y, j}^{(i)} - y_{y, j}^{(i)} \right)^2 \right], 
\]  
(B.13)
where the dependence on the model parameters \( \theta \) comes from all \( \alpha_{y, j}^{(i)} \) and \( \mu_{y, j}^{(i)} \).
This appendix describes how samples were generated for all the synthetic datasets used. Each dataset is completely specified by the process to generate samples from it. This process is here described for a single pair \((x, y)\). For examples of samples from each dataset see figure 3.3 and 3.4 in section 3.3.

The notation \(\sim\) is here used to denote sampling. \(U(a, b)\) is the continuous uniform distribution over the interval \([a, b]\). \(N(m, v)\) is the normal distribution with mean \(m\) and variance or covariance matrix \(v\). Exponential\((r)\) is the exponential distribution with rate parameter \(r\).

**const_noise**

\[
x \sim U(-1, 1) \\
\epsilon \sim N(0, 0.25) \\
y = x + \epsilon
\]  
\[\text{(C.1)}\]  
\[\text{(C.2)}\]  
\[\text{(C.3)}\]

**laplace**

\[
x \sim N(0, 1) \\
\epsilon \sim \text{Laplace}(0, 0.5) \\
y = 0.5x + \epsilon
\]  
\[\text{(C.4)}\]  
\[\text{(C.5)}\]  
\[\text{(C.6)}\]

**exponential**

\[
x \sim U(-1, 1) \\
r = \frac{1}{0.3(1 - |x|)} \\
\epsilon \sim \text{Exponential}(r) \\
y = -0.5x^2 + \epsilon
\]  
\[\text{(C.7)}\]  
\[\text{(C.8)}\]  
\[\text{(C.9)}\]  
\[\text{(C.10)}\]
butterfly

\[ x \sim \mathcal{U}(-2, 2) \]  
\[ \epsilon \sim \mathcal{N}(0, 1) \]  
\[ y = \log(1 + \exp(-x\epsilon)) \]  

\[ \text{(C.11)} \]
\[ \text{(C.12)} \]
\[ \text{(C.13)} \]

heteroskedastic

\[ x \sim \mathcal{U}(0, 1) \]  
\[ \sigma = \sin^2(\pi x) + 0.01 \]  
\[ \epsilon \sim \mathcal{N}(0, \sigma^2) \]  
\[ y = 0.5 + x^2 + \epsilon \]  

\[ \text{(C.14)} \]
\[ \text{(C.15)} \]
\[ \text{(C.16)} \]
\[ \text{(C.17)} \]

bimodal

\[ x \sim \mathcal{U}(-1, 1) \]  
\[ \epsilon \sim \mathcal{U}(-0.5, 0.5) \]  
\[ m_0 = 0.5x + 2.5 \]  
\[ m_1 = -0.5x \]  
\[ c \sim \text{Bernoulli}(0.5) \]  
\[ y = (1 - c)m_0 + cm_1 + \epsilon \]  

\[ \text{(C.18)} \]
\[ \text{(C.19)} \]
\[ \text{(C.20)} \]
\[ \text{(C.21)} \]
\[ \text{(C.22)} \]
\[ \text{(C.23)} \]

complex

\[ x \sim \mathcal{U}(-2, 2) \]  
\[ s = 1_{\{x \geq 0\}} \]  
\[ y_n \sim \mathcal{N}(0, -0.5x + 0.1) \]  
\[ c_p \sim \text{Bernoulli}(0.5) \]  
\[ \epsilon_p \sim \mathcal{N}(0, 0.1) \]  
\[ y_p = c_p(0.5x) + (1 - c_p)(-0.5x) + \epsilon_p \]  
\[ y = sy_p + (1 - s)y_n \]  

\[ \text{(C.24)} \]
\[ \text{(C.25)} \]
\[ \text{(C.26)} \]
\[ \text{(C.27)} \]
\[ \text{(C.28)} \]
\[ \text{(C.29)} \]
\[ \text{(C.30)} \]

swirls

\[ x \sim \mathcal{U}(0, 2) \]  
\[ c \sim \text{Bernoulli}(0.5) \]  
\[ \phi = \pi c + 0.5\pi x \]  
\[ \mu = [\cos(\phi), \sin(\phi)]^T \]  
\[ v = 0.3 - 0.2|x - 1| \]  
\[ y \sim \mathcal{N}(\mu, vI) \]  

\[ \text{(C.31)} \]
\[ \text{(C.32)} \]
\[ \text{(C.33)} \]
\[ \text{(C.34)} \]
\[ \text{(C.35)} \]
\[ \text{(C.36)} \]

88
Samples from Models Trained on Small Synthetic Datasets

This appendix contains scatter plots of samples from all different models used in the experiment on small synthetic datasets. The test set is also plotted for each dataset.
Figure D.1: Samples from models trained on the const_noise dataset
Figure D.2: Samples from models trained on the laplace dataset
Figure D.3: Samples from models trained on the exponential dataset
Figure D.4: Samples from models trained on the butterfly dataset
Figure D.5: Samples from models trained on the heteroskedastic dataset
Figure D.6: Samples from models trained on the bimodal dataset
Figure D.7: Samples from models trained on the complex dataset

(a) Test set  (b) Gaussian Process  (c) NN Regression

(d) Heteroskedastic NN  (e) Mixture Density Network  (f) DCTD

(g) Standard CGAN  (h) KL CGAN  (i) Reverse KL CGAN

(j) Neyman $\chi^2$ CGAN  (k) Pearson $\chi^2$ CGAN  (l) Squared Hellinger CGAN

(m) Jensen-Shannon CGAN  (n) Least Squares CGAN  (o) GMMN
Plots for trajectories datasets

Figures E.1, E.4 and E.7 contain example trajectories sampled from models trained on trajectories datasets with \( y \)-dimensionality \( d_y = 2, 5 \) and 10. Figure E.2, E.5 and E.8 show how sample trajectories change for the MDN and CGAN models when the maximum turning parameter \( t_{\text{max}} \) is changed. For all models, figure E.3, E.6 and E.9 show scatter plots of samples. See section 4.3.2 for a description of how scatter plots were generated.

Figure E.1: Sampled trajectories for the dataset with \( d_y = 4 \), all conditioned on \( x = [-0.16, -0.07, 0.34, 0.85, 0.27]^T \).
Figure E.2: Trajectories sampled from MDN and CGAN when conditioned on different maximum turning parameters, $t_{\text{max}}$. Models were trained on the trajectories dataset with $d_y = 4$. Samples were specifically conditioned on $x = [0, 0, 0.3, 1.571, t_{\text{max}}]^\top$. 
(a) True Distribution

(b) NN Regression

(c) Heteroskedastic NN Regression
Figure E.3: Scatter plots of 2000 samples for the trajectories dataset with $d_y = 4$. Conditioned on $x = [0, 0, 0.2, 0.25, 0.25]^T$ (left) and $x = [0.5, 0.5, 0.5, -1, 0.4]^T$ (right).

Figure E.4: Sampled trajectories for the dataset with $d_y = 10$, all conditioned on $x = [0.98, -0.3, 0.16, -0.04, 0.37]^T$. 
Figure E.5: Trajectories sampled from MDN and CGAN when conditioned on different maximum turning parameters, $t_{\text{max}}$. Models were trained on the trajectories dataset with $d_y = 10$. Samples were specifically conditioned on $x = [0, 0, 0.1, 1.571, t_{\text{max}}]$. 

(a) True Distribution

(b) Mixture Density Network

(c) CGAN
(a) True Distribution

(b) NN Regression

(c) Heteroskedastic NN Regression
Figure E.6: Scatter plots of 2000 samples for the *trajectories* dataset with $d_y = 10$. Conditioned on $x = [0, 0, 0, 0.25, 0.25]^\top$ (left) and $x = [0.5, 0.5, 0.2, -1, 0.4]^\top$ (right).

Figure E.7: Sampled trajectories for the dataset with $d_y = 20$, all conditioned on $x = [0.51, -0.87, 0.07, -1.5, 0.3]^\top$. 

103
Figure E.8: Trajectories sampled from MDN and CGAN when conditioned on different maximum turning parameters, $t_{\text{max}}$. Models were trained on the trajectories dataset with $d_y = 20$. Samples were specifically conditioned on $x = [0, 0, 0.05, 1.571, t_{\text{max}}]^\top$. 

(a) True Distribution

(b) Mixture Density Network

(c) CGAN
(a) True Distribution

(b) NN Regression

(c) Heteroskedastic NN Regression
Figure E.9: Scatter plots of 2000 samples for the trajectories dataset with $d_y = 20$. Conditioned on $x = [0, 0, 0.03, 0.25, 0.25]^T$ (left) and $x = [0.5, 0.5, 0.1, -1, 0.4]^T$ (right).
Plots for \texttt{wmix} datasets

Figures F.1-F.4 display conditional probability density functions for \texttt{wmix} datasets with different x-dimensionality, $d_x$. Plots are generated for 2 different x-values in the test set. The model distribution (pink, solid line) is compared to the true distribution (green, dashed line). KDE was used for CGAN and GMMN to estimate an explicit density function.
Figure F.1: Examples of conditional probability density functions for models trained on the \texttt{wmix} dataset with $d_x = 3$
Figure F.2: Examples of conditional probability density functions for models trained on the \texttt{wmix} dataset with $d_x = 6$
Figure F.3: Examples of conditional probability density functions for models trained on the \texttt{wmix} dataset with $d_x = 9$
(a) NN Regression

(b) Heteroskedastic NN Regression

(c) Mixture Density Network

(d) DCTD

(e) CGAN

(f) GMMN

Figure F.4: Examples of conditional probability density functions for models trained on the \texttt{wmix} dataset with $d_x = 15$