Birkbeck, University of London

MSc Applied Statistics

Unsupervised image vectorisation

Author:
Attila Kun

Supervisor:
Dr. Swati Chandna

September 1, 2019

Declaration of Authorship

Abstract

This work investigates the application of neural networks to the problem of unsupervised image vectorisation using the MNIST dataset. Three approaches are investigated to tackle the problem. First, a traditional autoencoder with a customised rendering bottleneck layer is evaluated. The second technique uses variational inference to improve on the results of the first approach. Finally, an unsuccessful attempt at the application of likelihood-free variational inference is described. The connection between likelihood-free variational inference and generative adversarial networks is also highlighted. Qualitative results are provided for the first and second approaches.

Contents

1 Introduction

1. Dataset

The MNIST [1] database is one of the most popular datasets among computer vision researchers. The version used in present work contains 70 000 raster images of handwritten digits split between a training set of 60 000 and a test set of 10 000 images. The images were taken from American Census Bureau employees and high-school students. The collected images were originally bi-level (black and white) with anti-aliasing applied to them later on. As a result, the image pixels represent continuous-valued grayscale intensities.

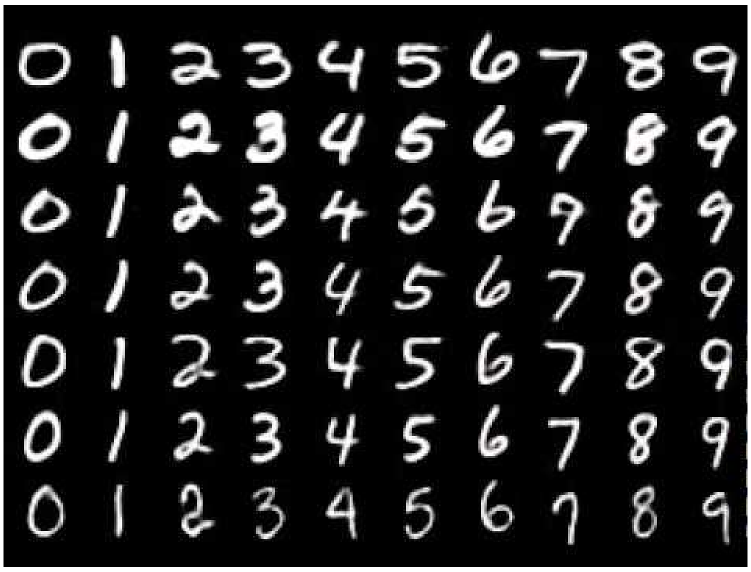


FIGURE 1.1: Samples from the MNIST database.

Present work investigates whether these raster images can be turned into vectorised images. Results are given in section 5. of chapter 2 and section 5. of chapter 3.

2. Background and motivation

In modern computers, two dimensional digital images are typically stored in the following ways:

1. Rasterised formats: These store images as a series of pixels. Some of them (like BMP) store the pixels in a straightforward way, others (like JPEG and PNG) apply compression techniques to reduce the file size. Information encoded in these formats is relatively low level as it is not straightforward to programmatically extract information with regards to the higher level content of the image.

[1]Source: http://yann.lecun.com/exdb/mnist/

These images are also of a specific resolution with a given width and height: displaying them on fewer or more pixels is only possible via various interpolation techniques which often lead to visible artefacts. The most widespread examples of rasterised images are photographs and scanned documents.

2. Vectorised formats: These store images via abstract languages. They describe the content of the image using high level abstractions. These abstractions can include geometrical shapes like lines, circles, ellipses, polygons and many others. The most widespread example of a vectorised image format on today's web is the Scalable Vector Graphics (SVG) format. Unlike rasterised images, vector images can be displayed at arbitrary resolutions with no rendering artefacts. This is because the abstract languages used by them typically describe the attributes of the various geometrical shapes in an abstract space. When the image is drawn on the screen, the shapes are eventually rasterised for the desired resolution via algorithms that are aware of efficient ways to render the shapes at any size. Typical examples of vectorised images are user interface icons in user-facing software.

Sometimes it is desirable to convert rasterised images into a vectorised format. For example:

- User-interface designers may want to design icons using a physical or digital pen. When they are finished with the design, the result is a rasterised image. It would be beneficial to find ways to automatically convert this into a vectorised format as this would allow the design to be used in a resolution independent manner in software with a graphical user interface.
- Rasterised formats are typically very high dimensional. A vectorised representation may offer a more compact format which might be beneficial for downstream tasks. For example, it should be easier to perform multiclass classification on a low dimensional vectorised representation than a high dimensional rasterised one.

Collecting rasterised images at a massive scale is relatively easy with modern technology. The availability of vectorised images however is much more scarce. Given this asymmetry, the question arises whether it is possible to leverage recent advances in machine learning in order to use the abundant pool of rasterised images to learn an automatic conversion to a vectorised format.

3. Problem statement

The goal of the present work is to create an algorithm that takes a rasterised image as its input and outputs a vectorised representation. It is acknowledged that finer details are going to be lost in the conversion process as the goal of vectorisation is to infer global structure.

One could expect that the conversion from a rasterised to a vectorised representation becomes more difficult as the rasterised inputs become more complex in terms of their contents. To keep the scope manageable and limit the computational complexity of the task, the rasterised inputs are taken from the MNIST database of handwritten digits.

There is a great degree of flexibility when deciding on the complexity of the vectorised format. For the present task of vectorising handwritten digits, it is expected that a sufficient description of the image can be achieved via outputting attributes describing the positioning of individual lines.

Mathematically, the problem can be stated as:

$$f(\text{image pixels}) = \left\{ \begin{matrix} (x_{11}, y_{11}, x_{12}, y_{12}), \\ \vdots \\ (x_{i1}, y_{i1}, x_{i2}, y_{i2}), \\ \vdots \\ (x_{n1}, y_{n1}, x_{n2}, y_{n2}) \end{matrix} \right\} \tag{1}$$

where:

- $f(\cdot)$ is the algorithm mapping the rasterised image into the vectorised representation. The goal is to train various kinds of neural networks which implement this function.
- n is the number of individual lines in the vectorised representation. Experiments in chapter chapter 2 use $n = 5$, while in chapter 3 they use $n = 6$.
- $(x_{i1}, y_{i1}, x_{i2}, y_{i2})$ is a 4-element tuple describing the ith line in the vectorised representation. The meaning of the tuple's elements is as follows:
  - x_{i1}: x coordinate of the ith line's start point.
  - y_{i1}: y coordinate of the ith line's start point.
  - x_{i2}: x coordinate of the ith line's end point.
  - y_{i2}: y coordinate of the ith line's end point.

4. Implementation plan

The MNIST dataset only contains rasterised images and no corresponding vectorised representations. Therefore, traditional supervised learning — which demands each input to be paired with the desired output — was not useful to tackle this problem.

Many algorithms exist already that address the problem of vectorising raster images. These algorithms are usually procedural in their nature, meaning that the process of finding vectors is explicitly programmed. In the present work, the goal is to investigate more declarative approaches such as designing appropriate objective functions that neural networks can learn to satisfy. This strategy should rely on fewer domain specific algorithmic steps and hopefully result in more general algorithms.

For this reason, some of the classical and recent techniques of unsupervised machine learning were utilised. The unsupervised nature of the problem differentiates present work from other attempts at neural network based vector learning such as sketch-rnn [5] that relies on pen stroke samples collected from humans. Present work is more similar in its intention to a recent paper by Ganin et al. [3] where the model is trained purely from raster images.

Three main methods are investigated. The most straightforward approach is outlined in chapter 2. This method uses a modified version of a classical autoencoder where the proposed neural network is fully differentiable. A more complicated approach outlined in chapter 3 invokes variational inference to perform the training of a neural network. It also introduces the technique of approximating the gradients of a non-differentiable renderer. The third, perhaps most technically complicated approach in chapter 4, relies on a recent paper by Tran et al. [11] which combines variational inference with advances in adversarial training.

Common to these methods is the usage of neural networks along with some kind of renderer. The job of the renderer is to produce a rasterised image of vectors, given a set of coordinates representing those vectors.

As for choice of technology, two big contenders were considered: Google's TensorFlow and Facebook's PyTorch libraries. TensorFlow is perhaps the most widely used machine learning library, however at the time present work was conducted, it suffered from some usability issues. Namely, it used to require the explicit definition of a computational graph as opposed to PyTorch's more straightforward approach of dynamically building the graph without much intervention from the user of the library. The recently introduced eager execution feature of TensorFlow addresses this issue, however PyTorch still remains a popular choice due to its conformity to widely-known Python libraries. For the reasons above, all experiments in this work were implemented in PyTorch.

Experiments were conducted on a personal computer with the following specifications:

- CPU: AMD Ryzen 7 2700X
- GPU: NVIDIA GeForce GTX 1060 6GB
- RAM: 16 GB

2 Differentiable line renderer

1. Traditional autoencoder

This approach is based on traditional autoencoders. These are feedforward neural networks whose input dimensions are the same as the output dimensions. The goal of training is to tweak the network's weights such that the input is reproduced at the output layer.

The architecture contains two main components: first, the input X flows through an encoding module which outputs an intermediate hidden code Z. Then this code is provided as input to a decoding module whose output X' has the same dimensions as X. By the end of training, any given X' should closely match X that it was generated from.

The architecture can be visualised as the diagram below[1]:

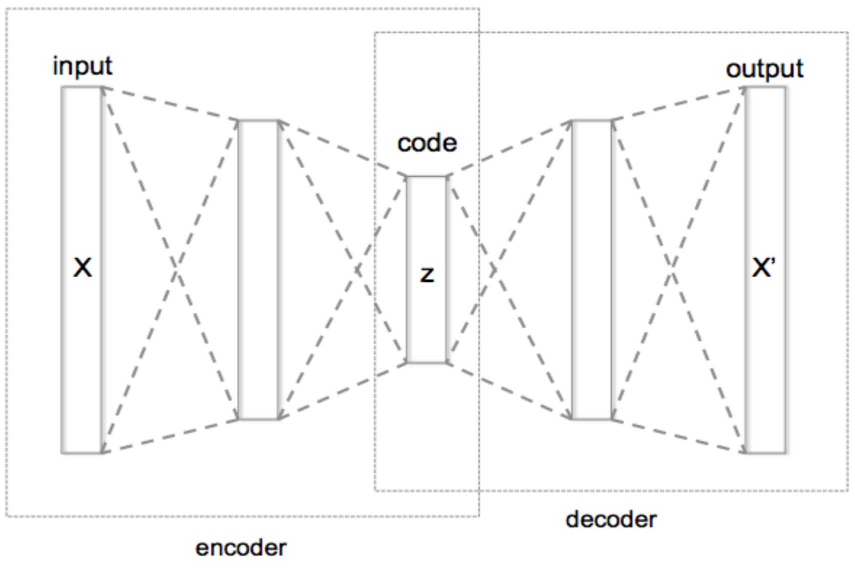


FIGURE 2.1: Schematic structure of an autoencoder.

[1]Image source: https://en.wikipedia.org/wiki/File:Autoencoder_structure.png

Mathematically, the model can be stated as:

$$\begin{aligned}
&\mathrm{enc}_\phi : \mathcal{X} \to \mathcal{Z} \\
&\mathrm{dec}_\psi : \mathcal{Z} \to \mathcal{X} \\
&Z = \mathrm{enc}_\phi(X) \\
&X' = \mathrm{dec}_\psi(Z) \\
&\phi, \psi = \underset{\phi,\psi}{\arg\min}\, \mathrm{loss}(X, X')
\end{aligned}$$

Where:

- X is the input, for example a rasterised image.
- enc_ϕ is the encoder part of the neural network parametrised by ϕ.
- dec_ψ is the decoder part of the neural network parametrised by ψ.
- Z is the hidden code produced by the encoder.
- X' is the reproduction produced by the decoder based on the hidden code Z.
- $\mathcal{X}$ is the space of possible inputs and outputs.
- $\mathcal{Z}$ is the space of possible hidden codes.
- $\mathrm{loss}(\cdot, \cdot)$ is a loss function. Squared error is a common choice, in which case $\mathrm{loss}(X, X') = \|X - X'\|^2$.

The hidden code Z is often thought of as a compressed representation of the input X. This is because the decoder solely depends on Z to produce X', so all relevant information must be present in Z. Because Z is typically smaller dimensional than X, the hidden code serves as a bottleneck layer when information flows from the input to output layers.

Note that it is not guaranteed that a small change in Z will result in a corresponding small change in X'. This is because the encoding and decoding implemented by the neural network's weights can be arbitrary, as long as the loss function is minimised. Therefore, similar inputs might be encoded to vastly different codes which means that interpolation between the codes is not necessarily meaningful. If this becomes a requirement, then the technique of variational autoencoders [8] can be employed which places a prior distribution on the hidden codes. This is discussed in more detail in chapter 3.

Training is typically performed via stochastic gradient descent, where ϕ and ψ are found via minimising the loss function. The training stops when the loss is sufficiently low. Methods such as cross validation can be used to detect overfitting.

2. Modified autoencoder

A modified version of the scheme outlined above can be used to solve the problem stated in section 3. of chapter 1. The proposal is to replace the hidden code layer with a customised renderer bottleneck. This new bottleneck is still differentiable, but it also allows the extraction of coordinates that will give the vectorised representation of the input image.

In particular, consider the following model:

$$\begin{aligned}
&\text{enc}_\phi : \mathcal{X} \to \mathcal{R}_\mathcal{I} \\
&\text{dec}_\psi : \mathcal{X} \to \mathcal{X} \\
&R_I = \text{enc}_\phi(X) \\
&R_O = \text{r}(R_I) \\
&X' = \text{dec}_\psi(R_O) \\
&\phi, \psi = \underset{\phi,\psi}{\arg\min}\, \text{loss}(X, X')
\end{aligned}$$

Where:

- X is the input.
- $\text{r}(\cdot)$ is the renderer. This is a differentiable function whose inputs are line coordinates and the output is a rendering of them. Note that while the renderer allows the propagation of the model's gradients, it does not itself have tunable parameters: for a given input, it yields the same output throughout the training process.
- R_I is the renderer's input. It represents a set of line coordinates.
- $\text{enc}_\phi(\cdot)$ is the encoder part of the neural network parametrised by ϕ. As the encoder generates R_I, the task of this component can be interpreted as inferring the line coordinates which serve as useful inputs to the renderer in reproducing the original input X.
- R_O is the renderer's output; a deterministic function of the input R_I. It is a visual rendering of the lines defined by R_I. If the training error is low, this rendering should be similar to the input image in terms of global structure. However, it lacks finer level details entirely since the renderer's inputs are not capable of encoding this information.
- $\text{dec}_\psi(\cdot)$ is the decoder part of the neural network parametrised by ψ. Its input is the renderer's output R_O. It builds on top of the global structure present in R_O, adding the finer level details. Note that these details are not sourced from the specific input image X as those are lost when generating R_I. Rather, this component helps matching the general style of the dataset. This smoothes the output pixels and makes inference using the mean squared error loss easier.

3. Renderer

As defined in the model above, the renderer is a deterministic function whose inputs are line coordinates and whose output is a rasterised image of the lines that the inputs represent. Mathematically this can be expressed as:

$$\text{r}\begin{pmatrix} (x_{11}, y_{11}, x_{12}, y_{12}), \\ \vdots \\ (x_{i1}, y_{i1}, x_{i2}, y_{i2}), \\ \vdots \\ (x_{n1}, y_{n1}, x_{n2}, y_{n2}) \end{pmatrix} = \text{image pixels} \tag{2}$$

Where:

- r$(\cdot)$ is a differentiable function. The encoder and decoder parts of the autoencoder model are differentiable by default if the idiomatic building blocks provided by PyTorch are used. Therefore, by making the r$(\cdot)$ function differentiable, it is possible to preserve the end-to-end differentiability of the model.

  End-to-end differentiability is important because it allows the application of stochastic gradient descent. It is important to note that the actual differentiation is performed via PyTorch's automatic differentiation capabilities. The user only needs to specify the differentiable model. The more complicated task of calculating gradients and performing the gradient descent step is done by PyTorch with little user intervention. This also allows for highly efficient training on the GPU.

- $(x_{i1}, y_{i1}, x_{i2}, y_{i2})$ is a 4-element tuple. The meaning of the tuple's elements is the same as in equation (1).

- The right-hand side is a two dimensional rasterised image depicting the lines defined by the inputs.

From the above, it is clear that a differentiable function implementing r$(\cdot)$ needs to be specified. A possible way to do this is to define a "heat map" whose points of higher magnitude represent the presence of one or more lines close to those points. The higher the value, the closer a line can be found to the point. Smaller magnitudes represent the lack of lines over the point.

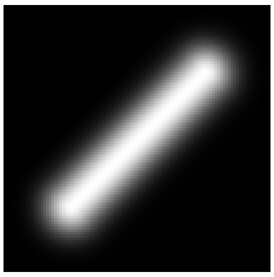


FIGURE 2.2: Heat map of a single diagonal line

A formula with this behaviour can be derived as follows. [2]

Let $s_1 = \left[s_{1x}, s_{1y}\right]^T$ and $s_2 = \left[s_{2x}, s_{2y}\right]^T$ denote the two endpoints of a line segment. The goal is to find the (squared) distance of an arbitrary point $p = \left[p_x, p_y\right]^T$ to this segment.

Let $s(t) = s_1 + t(s_2 - s_1)$ be a parameterisation of a point on the line where $t \in [0, 1]$. Also let $s_x(t) = s_{1x} + t(s_{2x} - s_{1x})$ and $s_y(t) = s_{1y} + t(s_{2y} - s_{1y})$ be the x and y components of $s(t)$.

The distance from p to $s(t)$ is $\gamma(t) = \|s(t) - p\|$. For mathematical convenience and greater numerical stability in the actual implementation, it is actually better to proceed with the squared distance $\gamma^2(t)$.

The next objective is to find the minimising $\hat{t}$, thereby giving the point on the line segment which is closest to p. First, observe that $\gamma^2(t)$ is quadratic in t:

[2]The derivation was adapted from the following StackExchange question: https://math.stackexchange.com/questions/330269/the-distance-from-a-point-to-a-line-segment

$$\begin{aligned}
\gamma^2(t) &= \|s(t) - p\|^2 = \\
&= \begin{bmatrix} s_x(t) - p_x \\ s_y(t) - p_y \end{bmatrix}^T \begin{bmatrix} s_x(t) - p_x \\ s_y(t) - p_y \end{bmatrix} \\
&= s_x^2(t) - 2s_x(t)p_x + p_x^2 + s_y^2(t) - 2s_y(t)p_y + p_y^2
\end{aligned}$$

Which is quadratic in t because $s_x^2(t)$ and $s_y^2(t)$ are quadratic in t. Now, taking the derivative of $\gamma^2(t)$ and setting to 0 gives:

$$\begin{aligned}
\frac{\partial \gamma^2(\hat{t})}{\partial \hat{t}} = \frac{\partial}{\partial \hat{t}} \left(s_x^2(\hat{t}) - 2s_x(\hat{t})p_x + p_x^2 + s_y^2(\hat{t}) - 2s_y(\hat{t})p_y + p_y^2 \right) &= 0 \\
2s_x(\hat{t})(s_{2x} - s_{1x}) - 2(s_{2x} - s_{1x})p_x + 2s_y(\hat{t})(s_{2y} - s_{1y}) - 2(s_{2y} - s_{1y})p_y &= 0 \\
s_x(\hat{t})(s_{2x} - s_{1x}) - (s_{2x} - s_{1x})p_x + s_y(\hat{t})(s_{2y} - s_{1y}) - (s_{2y} - s_{1y})p_y &= 0
\end{aligned}$$

Now, expanding $s_x(\hat{t})$ and $s_y(\hat{t})$ gives:

$$\begin{aligned}
s_{1x}(s_{2x} - s_{1x}) + \hat{t}(s_{2x} - s_{1x})^2 - p_x(s_{2x} - s_{1x}) + \\
+ s_{1y}(s_{2y} - s_{1y}) + \hat{t}(s_{2y} - s_{1y})^2 - p_y(s_{2y} - s_{1y}) = 0 \\
\hat{t} = \frac{(p_x - s_{1x})(s_{2x} - s_{1x}) + (p_y - s_{1y})(s_{2y} - s_{1y})}{(s_{2x} - s_{1x})^2 + (s_{2y} - s_{1y})^2} \\
\hat{t} = \frac{\langle p - s_1, s_2 - s_1 \rangle}{\|s_2 - s_1\|^2}
\end{aligned}$$

This expression however does not guarantee $\hat{t}$ to be between 0 and 1. When $\hat{t}$ falls outside of this range, then the point $s(\hat{t})$ falls outside of the line segment defined by s_1 and s_2. This can be fixed via applying the following constraint:

$$t^* = \min(\max(\hat{t}, 0), 1)$$

This accounts for cases where the shortest line from p to the line segment is non-orthogonal. Thus, the squared distance from p to the closest point of the line segment is given by $d_p = \|s(t^*) - p\|^2$.

Calculating the (squared) distance alone is not sufficient: ultimately the output should indicate the *presence* of a line segment at a certain point. A line is present at a point if its distance from that point is small. It is absent from that point if its distance from the point is large. If ω_p denotes the presence of a line segment at point p, then the inverse relationship between presence and distance can be modelled by:

$$\omega_p = \exp\left(-\alpha * d_p\right)$$

Where $\alpha > 0$ is a tuning parameter that decides how sensitive the algorithm is to the closeness of a line from a given point. A small value of α means that the presence of a line takes effect even if it is far away from p. A large value of α means that the line is going to be more readily ignored; a line has to be really close to p for its presence to make a difference. The experiments use $\alpha = 8$.

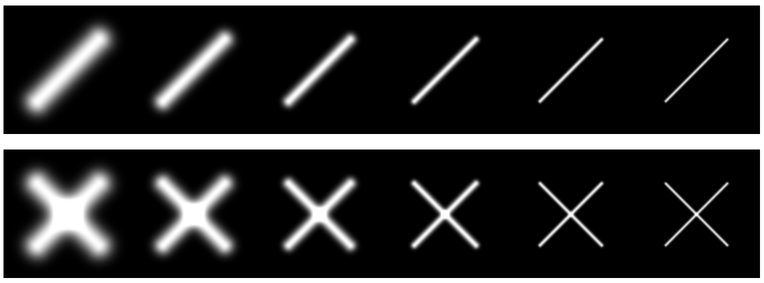


FIGURE 2.3: Heat maps with the value of α increasing from left to right

The above allows the distance to be calculated for a single point p. The right-hand side of equation (2) however is expected to be a two dimensional matrix. Therefore, the above calculation needs to be performed for every pixel of the output matrix. In practice, this is very computationally intensive if done naively on the CPU. PyTorch's tensor data structures allow for fast parallel execution on the GPU where each output pixel's distance from the line segment is calculated in the same computational step. This method can be viewed as a simplified version of more complicated 3D soft-rasterisers such as the one developed by Liu et al. [9].

4. Network implementation and training

A substantial amount of experimentation was needed to find the set of hyperparameters and network structure until some results started to appear. The final version had the following structure:

- Encoder: A three layer deep neural network was used for this component. The first two layers were convolutional ones, with channel sizes 16 and 8, the third one was a simple linear layer. The activation function for the convolutional layers was the leaky ReLU function, whereas for the third layer the sigmoid function was used. Batch normalization [6] was found to be beneficial for training. Invocation of the renderer was performed in the encoding step. Therefore, the encoder returned both the renderer input and output.
- Decoder: This was a two layer deep convolutional neural network with channel sizes 16 and 1. There were no activation functions used here, so the output was simply a linear transformation of the input.

  This component was intentionally kept simple. A large amount of computational capacity was found to be detrimental to the training process: if the component was allowed to perform complex computations, then it was able to reliably reproduce the original input image based on very little information coming from the renderer. In such case, the network did not utilise the renderer correctly; it encoded information into the renderer's inputs (and hence outputs) in a non-interpretable way. Heavily constraining the component solved this problem.

The loss function was mean squared error and the network parameters were tuned via the Adam optimiser [7]. Training was performed on the MNIST database's training set and results were taken from the test set.

5. Results

After training the network parameters on the training set of the MNIST database, the quality of vector inference can be assessed via the following process:

1. Feed an image to the network from the test set of the MNIST database.
2. Extract R_I described in section 2. which are the activations reaching the differentiable renderer's input. These are the inferred vectors.
3. Send the extracted activations through a hard rasteriser. This gives a visual representation of the inferred vectors.

The output of the above process gives the following qualitative results:

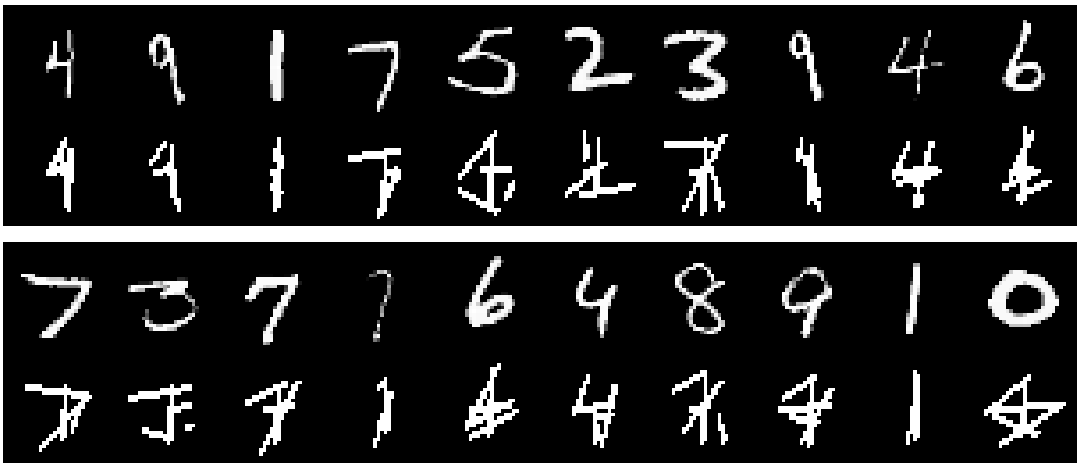


FIGURE 2.4: Qualitative results from the modified autoencoder.

The first rows show the original images from the test set of the MNIST database. The second rows show the inferred vectors associated with them.

It seems that the network manages to learn vector inference to some degree. The quality of the inference however is rather weak and also depends on the digit class. The training process is also very sensitive to the random seed it was started with. Ways to improve on these results are investigated in chapter 3.

6. Advantages and disadvantages

The biggest advantage of this method is the speed of training that full differentiability allows. On the other hand, the renderer needs to be carefully programmed to allow for fast differentiable execution on the GPU. This puts a burden on the user as the complexity of the programming task grows with the complexity of the inferred shapes.

3 Variational autoencoder

1. Probabilistic setup

Kingma and Welling [8] invoke variational inference to improve on traditional autoencoders. They consider a generative model where:

1. A hidden variable $\boldsymbol{z}_i$ is generated from some prior distribution $p_\theta(\boldsymbol{z})$.
2. An observed value $\boldsymbol{x}_i$ is generated from some conditional distribution $p_\theta(\boldsymbol{x}|\boldsymbol{z})$.

They make the following assumptions:

1. The dataset consists of N jointly independent continuous-valued samples $\{\boldsymbol{x}_i\}_i^N$.
2. $p_\theta(\boldsymbol{z})$ and $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ are parametric families of distributions whose probability density functions are differentiable almost everywhere with regards to both $\boldsymbol{\theta}$ and $\boldsymbol{z}$.
3. $\boldsymbol{\theta}$ and $\boldsymbol{z}$ are unknown.

They propose an algorithm capable of:

1. Efficient approximate maximum likelihood or maximum a posteriori estimation for the parameters $\boldsymbol{\theta}$.
2. Given the observed variable $\boldsymbol{x}$, efficient approximate posterior inference of the hidden variable $\boldsymbol{z}$. This is what present work makes use of.
3. Marginal inference of the variable $\boldsymbol{x}$. Present work does not make use of this, but in general it allows applications such as image denoising, inpainting or super-resolution.

They achieve the above via the introduction of a recognition model $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ which approximates the intractable true posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$. $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ also belongs to a parametric family and its differentiability with regards to $\boldsymbol{\phi}$ is assumed.

The goal of the algorithm is the joint inference of $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. This is achieved via the maximisation of the variational lower bound.

2. Variational lower bound derivation

In order to approximate $p_\theta(\boldsymbol{z}_i|\boldsymbol{x}_i)$ with $q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)$, an appropriate way of comparing the two distributions is necessary. This is where the Kullback–Leibler divergence becomes useful.

i) Kullback–Leibler divergence

The Kullback–Leibler divergence is a widely used statistical tool measuring the difference between two probability distributions. Given distributions $a(x)$ and $b(x)$, it can be defined as:

$$D_{KL}(a(x) \mid\mid b(x)) = -\,\mathrm{E}_{a(x)}\left[\log \frac{b(x)}{a(x)}\right]$$

For the purposes of variational inference, its most interesting property is non-negativity. The measure equals zero if and only if the probability distributions are equal almost everywhere.

ii) Variational bound

From the definition of the Kullback–Leibler divergence, consider [1]:

$$\begin{aligned} D_{KL}(q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i}) \mid\mid p_\theta(\boldsymbol{z_i}|\boldsymbol{x_i})) = -\,\mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log \frac{p_\theta(\boldsymbol{z_i}|\boldsymbol{x_i})}{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\right] &= \\ -\,\mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log p_\theta(\boldsymbol{z_i}|\boldsymbol{x_i}) - \log q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})\right] &= \\ -\,\mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log \frac{p_\theta(\boldsymbol{z_i},\boldsymbol{x_i})}{p_\theta(\boldsymbol{x_i})} - \log q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})\right] &= \\ -\,\mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log p_\theta(\boldsymbol{z_i},\boldsymbol{x_i}) - \log p_\theta(\boldsymbol{x_i}) - \log q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})\right] & \end{aligned} \tag{3}$$

However, as $\log p_\theta(\boldsymbol{x_i})$ does not depend on $\boldsymbol{z_i}$:

$$\mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log p_\theta(\boldsymbol{x_i})\right] = \log p_\theta(\boldsymbol{x_i})$$

Hence (3) becomes:

$$D_{KL}(q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i}) \mid\mid p_\theta(\boldsymbol{z_i}|\boldsymbol{x_i})) = -\,\mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log p_\theta(\boldsymbol{z_i},\boldsymbol{x_i}) - \log q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})\right] + \log p_\theta(\boldsymbol{x_i})$$

Which, after rearranging the terms can be written as:

$$\log p_\theta(\boldsymbol{x_i}) = D_{KL}(q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i}) \mid\mid p_\theta(\boldsymbol{z_i}|\boldsymbol{x_i})) + \mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log p_\theta(\boldsymbol{z_i},\boldsymbol{x_i}) - \log q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})\right] \tag{4}$$

It is known that the $D_{KL}(q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i}) \mid\mid p_\theta(\boldsymbol{z_i}|\boldsymbol{x_i}))$ term in (4) is non-negative due to the general properties of the Kullback–Leibler divergence. Hence, the following inequality can be written:

$$\log p_\theta(\boldsymbol{x_i}) \geq \mathrm{E}_{q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})}\left[\log p_\theta(\boldsymbol{z_i},\boldsymbol{x_i}) - \log q_\phi(\boldsymbol{z_i}|\boldsymbol{x_i})\right] = \mathcal{L}\left(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{x_i}\right) \tag{5}$$

This is called the variational lower bound on the marginal likelihood of the data point $\boldsymbol{x_i}$. The goal is to maximise this bound with regards to $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ for each data

[1]Derivation is adapted from: https://chrisorm.github.io/VI-ELBO.html

point $\boldsymbol{x}_i$. If this happens, then the generative model becomes more and more likely to produce the data points after drawing from the prior.

Notice that if the approximate posterior in (4) perfectly matches the real, intractable posterior (which is to say that the divergence term is 0), then the lower bound becomes tight: the marginal log-likelihood equals the variational lower bound. Or in other words, if the lower bound in (5) achieves its maximum, then the divergence term in (4) must be 0.

From (5), the following alternative form can be obtained:

$$\begin{aligned}\mathcal{L}\left(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{x}_i\right) &= \mathrm{E}_{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i) + \log p_\theta(\boldsymbol{z}_i) - \log q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)\right] \\ &= \mathrm{E}_{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\left[\log \frac{p_\theta(\boldsymbol{z}_i)}{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\right] + \mathrm{E}_{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i)\right] \\ &= \underbrace{-D_{KL}(q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i) \mid\mid p_\theta(\boldsymbol{z}_i))}_{\text{regularisation}} + \underbrace{\mathrm{E}_{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i)\right]}_{\text{reconstruction}} \qquad (6)\end{aligned}$$

The divergence term measures the difference between the approximated posterior and the prior. This is often thought to have a regularising effect: if the difference becomes large, then the bound becomes weaker. As the goal of the training algorithm is to maximise the lower bound, it tries bringing the approximated posterior closer to the prior. This ensures that the latent codes $\boldsymbol{z}_i$ (produced by sampling from $q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)$) will not be mapped to arbitrary regions of the code space, but rather will stay concentrated around the prior defined by $p_\theta(\boldsymbol{z}_i)$. This is a big difference from traditional autoencoders which lack incentives for better quality latent codes.

$\mathrm{E}_{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i)\right]$ can be interpreted as a reconstruction term. It measures the average of the following two-step process:

1. Given an input $\boldsymbol{x}_i$, a latent code $\boldsymbol{z}_i$ is sampled from the approximate posterior $q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)$.
2. The latent code $\boldsymbol{z}_i$ is put through the forward model $p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i)$ which gives rise to a distribution over the input space. The result of the operation is the log-probability of $\boldsymbol{x}_i$ under this distribution.

From the above, it is clear that the reconstruction term's value will be high if the probability of the input $\boldsymbol{x}_i$ is high given its inferred hidden code.

Conveniently, the divergence term can often be calculated analytically, for example in case of normal distributions. Present work makes use of this, therefore only the reconstruction term requires estimation via sampling.

iii) Reparameterisation trick

Differentiating $\mathrm{E}_{q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i)\right]$ with regards to $\boldsymbol{\phi}$ is not straightforward. This is because the expectation is taken with regards to $q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)$, which depends on $\boldsymbol{\phi}$.

The naive solution is to apply the technique of score function estimation[2]:

[2] Derivation is adapted from: http://blog.shakirm.com/2015/11/machine-learning-trick-of-the-day-5-log-derivative-trick/

$$\begin{aligned}\nabla_\phi \mathrm{E}_{q_\phi(z_i|x_i)}\left[\log p_\theta(x_i|z_i)\right] &= \int \nabla_\phi q_\phi(z_i|x_i) \log p_\theta(x_i|z_i) dz_i = \\ &\int \frac{q_\phi(z_i|x_i)}{q_\phi(z_i|x_i)} \nabla_\phi q_\phi(z_i|x_i) \log p_\theta(x_i|z_i) dz_i = \\ &\int q_\phi(z_i|x_i) \underbrace{\frac{\nabla_\phi q_\phi(z_i|x_i)}{q_\phi(z_i|x_i)}}_{\nabla_\phi \log q_\phi(z_i|x_i)} \log p_\theta(x_i|z_i) dz_i = \\ &\mathrm{E}_{q_\phi(z_i|x_i)}\left[\nabla_\phi \log q_\phi(z_i|x_i) \log p_\theta(x_i|z_i)\right]\end{aligned}$$

Therefore, estimation of the gradient can be achieved via sampling from the approximate posterior and differentiating the density function. However, as mentioned by Kingma and Welling [8], this estimator has high variance and is not practical for training.

An alternative approach is to make use of the specific distributional properties of $q_\phi(z_i|x_i)$. If, for example, $q_\phi(z_i|x_i)$ has the form of a normal distribution, then $\boldsymbol{\phi}$ and x_i give rise to the mean μ_i and variance σ_i of the multivariate normal distribution. Instead of sampling $z_i \sim N(\mu_i, \sigma_i)$ from this distribution directly, sampling can be achieved via the following process:

1. Let $\epsilon \sim N(\mathbf{0}, \boldsymbol{I})$ be a sample from a (multivariate) standard normal distribution.
2. Calculate $\tilde{z}_i = \epsilon * \sigma_i + \mu_i$.

This way of sampling from a distribution is often referred to as the reparameterisation trick. Because of the properties of the normal distribution, $\tilde{z}_i$ has the same distribution as z_i but now the reconstruction term can be written as:

$$\nabla_\phi \mathrm{E}_{q_\phi(z_i|x_i)}\left[\log p_\theta(x_i|z_i)\right] = \mathrm{E}_{\epsilon \sim N(\mathbf{0}, \boldsymbol{I})}\left[\nabla_\phi \log p_\theta(x_i|\tilde{z}_i)\right]$$

Here the expectation is taken with respect to ϵ which does not depend on $\boldsymbol{\phi}$. This allows estimation via sampling. The dependence on $\boldsymbol{\phi}$ only appears through $\tilde{z}_i$. This is appropriate because $p_\theta(x_i|\tilde{z}_i)$ was assumed to be differentiable with regards to $\tilde{z}_i$.

While present work only makes use of reparameterisation of the normal distribution, the authors mention a wide range of other distributions that a similar transformation can be applied to.

iv) Autoencoders

Relatively few assumptions are made with regards to the forms of $p_\theta(z_i)$, $p_\theta(x_i|z_i)$ and $q_\phi(z_i|x_i)$. As long as they satisfy the requirements outlined in section 1., they can take many forms, but perhaps the most popular choice is the usage of neural networks. In this case, the neural networks' outputs are the parameters of the desired distributions.

For example, if $p_\theta(x_i|z_i)$ is chosen to belong to the normal distribution, then the neural network's weights are parametrised by $\boldsymbol{\theta}$. The neural network's input is z_i, its outputs are μ_i and σ_i which parametrise the normal distribution. From this, sampling x_i is straightforward. $q_\phi(z_i|x_i)$ is usually handled similarly.

Models using neural networks in this fashion to represent $p_\theta(\boldsymbol{x}_i|\boldsymbol{z}_i)$ and $q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)$ are usually referred to as variational autoencoders.

The standard (multivariate) normal distribution is a common choice for the prior and this proved sufficient for present work. In this case, $\boldsymbol{\theta}$ does not contain trainable parameters with regards to the prior, so $p(\boldsymbol{z}_i)$ becomes the preferred notation. However, this assumes that the hidden codes have a single mode, which might not be appropriate for some domains. Researchers have been experimenting with using a mixture of normal distributions to represent multimodality in the hidden codes [2].

Having access to the prior is a big advantage of variational autoencoders compared to their traditional counterparts described in chapter 2. Variational autoencoders are a generative model that allow easy sampling from the prior, whereas traditional autoencoders merely reproduce their inputs. Sampling from the hidden codes of a traditional autoencoder is not trivial. A possible solution, which was inspired by the invention of generative adversarial networks, is to impose distributional constraints on the hidden codes via adversarial methods [10]. In this scenario, on top of the autoencoder, an additional neural network is employed whose goal is to discriminate between the hidden codes and samples from a predefined distribution (for example, the multivariate normal). Then, apart from reproducing its input, the autoencoder's new goal is to fool the discriminator neural network, thereby producing hidden codes that match the desired distribution. Once this happens to a sufficient degree, the hidden codes can be sampled from the target distribution.

3. Approximate renderer

The framework outlined above is not yet sufficient to solve the problem of vectorising raster images. An introduction of a renderer, similar in purpose to the one described in chapter 2, is required. The overall functionality of this renderer is essentially unchanged: given a set of line coordinates, it outputs a rasterised image depicting those lines.

In order to do this, first an appropriate place needs to be identified where the renderer can be inserted in the variational autoencoder architecture. Similarly to chapter 2, the most natural choice is to interpret the hidden code $\boldsymbol{z}_i$ drawn from $q_\phi(\boldsymbol{z}_i|\boldsymbol{x}_i)$ as a basis for the input to the line renderer. Then the output of the renderer can be interpreted as the reconstruction of the original image $\boldsymbol{x}_i$.

Differentiability was a major constraint on the renderer seen in chapter 2. In order to improve on previous results, it might be beneficial to find ways around this requirement. Not being restricted by differentiability also allows access to a much richer class of renderers: instead of ensuring that the inputs have a continuous, differentiable relationship to the output, in case of a non-differentiable renderer more complicated algorithms can be used to turn the input coordinates into a raster image output.

However, variational autoencoders require certain differentiability conditions to be met in order to perform training via stochastic gradient descent. To resolve this, the proposal is to approximate the behaviour of the non-differentiable renderer by a neural network. This network performing the approximate rendering is of course differentiable, hence it can be placed in the variational autoencoder pipeline.

4. Application to image vectorisation

Consider the following components:

- $\{x_i\}_i^N$ is the set of raster images that serve as input.
- $q_\phi(\tilde{z}_i|x_i)$ is the multivariate normal distribution induced by the neural network with parameters ϕ. The actual implementation uses a two layer deep neural network for this component. Both layers are linear and use the leaky ReLU activation function. Given an input image x_i, the network produces two outputs: a mean μ_i and a variance σ_i component. Using these outputs, the hidden code $\tilde{z}_i$ is drawn via the reparameterisation trick, as explained in subsection iii).
- $\tilde{z}_i$ is the hidden code for input image x_i. Depending on the phase of training, this hidden code is either used to generate the input to the non-differentiable renderer or its differentiable approximation.
- $r(\tilde{z}_i)$ is the non-differentiable renderer. Unlike the rest of the model's architecture which resides on the GPU, the renderer operates on the CPU instead. This is important because CPU based operation in general gives access to a wider range of possible drawing algorithms. However, writing CPU-intensive algorithms in interpreted languages like Python often comes at the cost of slower code execution. This proved a problem in practice, in fact the original CPU-based implementation has significantly slowed down training which made an iterative development workflow difficult.

  To overcome the problem of slow performance, the drawing algorithm is implemented as a native library in the Rust programming language. Rust was created by Mozilla in order to allow highly performant, concurrent and safe code execution. The Rust compiler's output is a native binary that can be invoked from the interpreted Python code. Using this library decreases training time from several hours to about 30 minutes.
- $p(z_i)$ is a multivariate standard normal distribution. This serves as the prior to the generative model.
- $p_\theta(x_i|\tilde{z}_i)$ is the neural network approximating the non-differentiable renderer's behaviour. The implementation uses a three layer deep neural network with linear layers. The first layers use the leaky ReLU activation function, while the last layer uses the sigmoid. This is because the neural network models the (multivariate) Bernoulli distribution: given the hidden code z_i, the network outputs a probability value for each pixel.

  In practice, when producing the output image, the images are *not* sampled from the Bernoulli distribution. Instead, the probability values are used as pixel intensities.

  It is worth noting that unlike the decoder described in section 2. of chapter 2, the architecture here does not have an additional neural network that adds finer level details to the renderer's output.
- Note that $\tilde{z}_i$ can not be used directly as input to the renderer or the approximator. This is because the prior is assumed to be a (multivariate) standard normal distribution and these components expect line coordinates as input.

Therefore, an additional two layer deep neural network is used to transform $\tilde{z}_i$ into a representation (line coordinates) that the renderer and its approximation can accept. The network is used internally by $r(\tilde{z}_i)$ and $p_\theta(x_i|\tilde{z}_i)$. Training of this additional network's parameters happens in algorithm 2 but for ease of notation it is omitted.

The next step is to ensure that the renderer approximator $p_\theta(x_i|\tilde{z}_i)$ matches the non-differentiable renderer's $r(\tilde{z}_i)$ behaviour. This is what the following algorithm achieves:

```
Data: Training set of MNIST database.
$\theta \leftarrow$ random initialisation or reuse parameters from previous run
$\phi \leftarrow$ reuse parameters from algorithm 2
Result: $\theta$
for all minibatches do
    loss $\leftarrow$ 0
    $\{x_i\}_i^M \leftarrow$ sample random minibatch of $M$ datapoints from MNIST
    for $x_i \in \{x_i\}_i^M$ do
        $\tilde{z}_i \leftarrow$ sample from $q_\phi(\tilde{z}_i|x_i)$ via the reparameterisation trick
        $y_i \leftarrow r(\tilde{z}_i)$
        $x'_i \leftarrow$ sample from $p_\theta(x_i|\tilde{z}_i)$
        loss $\leftarrow$ loss + MSELoss($y_i$, $x'_i$)
    end
    $g \leftarrow \nabla_\theta$loss
    $\theta \leftarrow$ Perform a stochastic gradient descent step based on gradients $g$.
end
return $\theta$
```
Algorithm 1: Approximating the non-differentiable renderer's behaviour

Note that in the algorithm above the approximate posterior parameters ϕ are kept frozen, while the approximate renderer parameters θ are being changed.

The following algorithm implements the training of the approximate posterior parameters ϕ:

```
Data: Training set of MNIST database.
Result: $\phi$
$\phi \leftarrow$ random initialisation or reuse parameters from previous run
$\theta \leftarrow$ reuse parameters from algorithm 1
for all minibatches do
    loss $\leftarrow$ 0
    $\{x_i\}_i^M \leftarrow$ sample random minibatch of $M$ datapoints from MNIST
    for $x_i \in \{x_i\}_i^M$ do
        $\tilde{z}_i \leftarrow$ sample from $q_\phi(\tilde{z}_i|x_i)$ via the reparameterisation trick
        $x'_i \leftarrow$ use the output of $p_\theta(x_i|\tilde{z}_i)$ without actual sampling
        divergence loss $\leftarrow D_{KL}(q_\phi(z_i|x_i)) \,||\, p(z_i))$
        cross entropy loss $\leftarrow$ BCELoss$(x_i, x'_i)$
        loss $\leftarrow$ loss + divergence loss + cross entropy loss
    end
    $g \leftarrow \nabla_\phi$loss
    $\phi \leftarrow$ Perform a stochastic gradient descent step based on gradients $g$.
end
return $\phi$
```
Algorithm 2: Training the approximate posterior parameters

Here the approximate renderer parameters $\boldsymbol{\theta}$ are kept frozen, while the approximate posterior parameters $\boldsymbol{\phi}$ are being changed.

Note that algorithms 1 and 2 rely on each other because of the parameter initialisation steps for $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. This poses a theoretical conundrum at the bootstrapping stage, but it was found that starting the training process with both set of parameters randomly initialised in algorithm 1 worked well in practice.

After this initial bootstrapping step, the parameters are trained via alternating invocations of algorithms 1 and 2.

5. Results

i) Vector inference

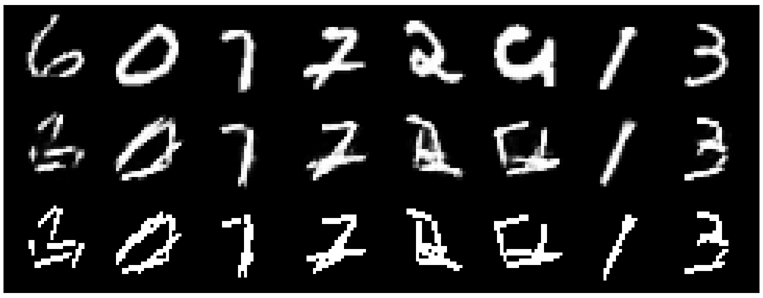


FIGURE 3.1: Vector inference with regularisation term.

The first row shows the original images from the test set of the MNIST database. The second row shows the final reproduction taken from the approximate renderer's output. The third row shows the vectorised representation taken from the non-differentiable renderer's output.

Looking at the second and third rows, it is clear that the approximate renderer managed to learn the non-differentiable renderer's behaviour to a high degree of fidelity.

Comparing the third row of Figure 3.1 to the second rows seen in Figure 2.4, the quality of the vectorisation seems to be better using the techniques outlined in this chapter.

However, Figure 3.1 shows that the quality of the inference still seems to be somewhat dependent on the digit class. Simple cases like the digit class "1" (7th column) and "7" (3rd and 4th columns) are reproduced well. Cases where part of the digit is missing (digit class "6" in first column) or the writing is less recognisable (digit class "9" in 6th column) pose a bigger difficulty to the algorithm.

Additional samples are included in appendix A.

ii) Drawing from the prior

One of the main advantages of variational autoencoders is their ability to learn a generative model of the data. It is possible to draw from the standard (multivariate) normal prior and feed these samples to the network implementing $p_\theta(x_i|z_i)$. The following are samples generated from such a process:

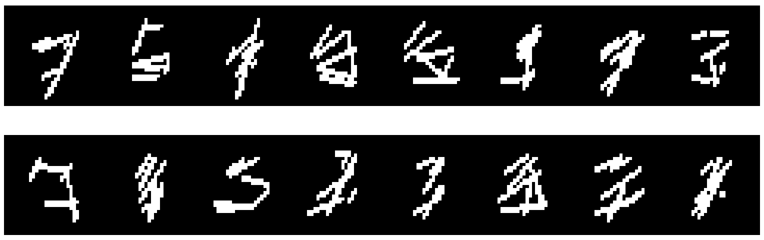


FIGURE 3.2: Samples produced by drawing from the prior after training with regularisation term.

Noticeably, the output quality is lower than the inferences made in subsection i). Still, the network visibly "tries" to at least approach the data distribution. The next section shows that this is due to the regularising effect of the Kullback–Leibler divergence term of the variational objective in equation (6).

iii) Ablation study

It is interesting to see what happens if the regularisation term is removed from equation (6). The theory of variational inference suggests that this should have a deleterious effect on the generative model. This is because the inferred hidden codes of the empirical distribution will not be forced to be similar to the prior. Therefore, $p_\theta(x_i|z_i)$ will learn to reproduce the empirical distribution from these unconstrained hidden codes. Running the experiments with the regularisation term removed from equation (6) suggests that this is indeed the case:

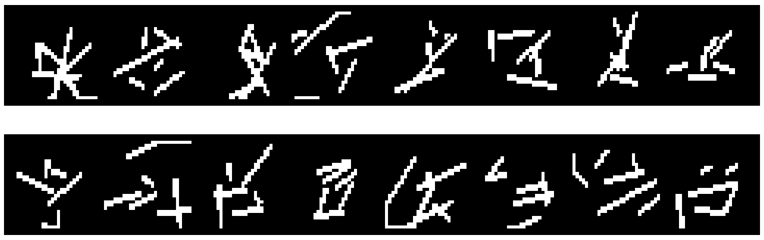


FIGURE 3.3: Samples produced by drawing from the prior after training with no regularisation term.

The obtained samples show less similarity to the empirical distribution than the ones obtained in subsection ii). This supports the claim that the regularisation term is necessary for a high quality generative model.

It is also interesting to see the effect of removing the regularisation term on the model's vector inference ability:

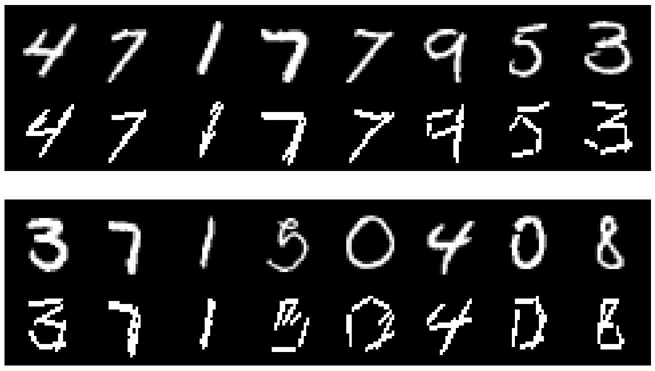


FIGURE 3.4: Vector inference with no regularisation term.

Compared to Figure 3.1, the vector inference quality is largely the same. This can be attributed to the fact that the regularisation term is only important for the purposes of sampling from the prior. Removing it still allows the model to produce hidden codes that faithfully reproduce the input, although this comes at the cost of the ability to sample from the prior.

Additional samples for the unregularised model are included in appendix B.

6. Related work

The sketch-rnn [5] architecture developed by Google Brain researchers investigates the problem of generating vector images. Unlike the present work which relies on rasterised images, the authors use a dataset consisting of pen strokes collected from humans. Every image in the dataset is represented as a sequence of these strokes.

Their model is capable of operating either in a conditional or unconditional fashion. In the conditional case, their model works as a variational autoencoder. They train both an encoder and a decoder recurrent neural network in order to produce and utilise the hidden codes. This allows them to perform latent space interpolation. For example, given the hidden codes for a cat face, pig body and pig face, their model is able to generate a picture of a cat's body.

In case the model is operating in an unconditional fashion, they only train the decoder neural network and the model reduces to a traditional autoencoder. This achieves a higher reproduction score but they lose the ability to perform latent space interpolation.

4 Likelihood-free variational inference

1. Theoretical background

In the context of variational autoencoders discussed in chapter 3, it was assumed that the likelihood functions for the generative model and the approximate posterior were fully specified: sampling and direct evaluation of their densities were possible. Likelihood-free techniques are applicable under more general circumstances. They allow inference in the presence of distributions from which only sampling is possible but not direct evaluation of the density function. Distributions with this property are called implicit distributions.

Tran et al. [11] consider the following hierarchical generative model:

- Global variables $\boldsymbol{\beta}$ are drawn from a prior distribution $p(\boldsymbol{\beta})$. These are common among all observations. $p(\boldsymbol{\beta})$ is assumed to be a fully specified likelihood function whose density can be evaluated. Differentiability with regards to $\boldsymbol{\beta}$ is also assumed.
- A set of hidden variables are drawn from a conditional distribution $p(\boldsymbol{z}|\boldsymbol{\beta})$ for each observation. Only the ability to sample is assumed.
- A set of observations are drawn from a conditional distribution $p(\boldsymbol{x}|\boldsymbol{z}, \boldsymbol{\beta})$. Only the ability to sample is assumed.

Then, the joint distribution can be written as:

$$p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\beta}) = p(\boldsymbol{\beta}) \prod_{n=1}^{N} p(\boldsymbol{x_n}|\boldsymbol{z_n}, \boldsymbol{\beta}) p(\boldsymbol{z_n}|\boldsymbol{\beta})$$

where $\boldsymbol{x_n}$ is a single observation and $\boldsymbol{z_n}$ is a hidden variable belonging to that observation.

The goal is to calculate the model's posterior: $p(\boldsymbol{\beta}, \boldsymbol{z}|\boldsymbol{x}) = p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\beta}) / p(\boldsymbol{x})$. This is difficult because the marginal distribution $p(\boldsymbol{x})$ is often intractable and $p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\beta})$ is allowed to contain implicit terms.

Similarly to the variational approach described in chapter 3, the authors posit an approximate posterior $q(\boldsymbol{\beta}, \boldsymbol{z}|\boldsymbol{x})$ which aims to approximate the true posterior $p(\boldsymbol{\beta}, \boldsymbol{z}|\boldsymbol{x})$. In order to support a rich class of distributions, $q(\boldsymbol{z_n}|\boldsymbol{x_n}, \boldsymbol{\beta})$ is also assumed to be an implicit distribution.

Finding the approximate posterior is again done via the minimisation of the Kullback-Leibler divergence. This is also equivalent to maximising the evidence lower bound:

$$\mathcal{L} = \mathrm{E}_{q(\boldsymbol{\beta}, \boldsymbol{z}|\boldsymbol{x})} \left[\log p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\beta}) - \log q(\boldsymbol{\beta}, \boldsymbol{z}|\boldsymbol{x})\right] \tag{7}$$

Due to the presence of implicit distributions in the above expression, optimising it is not directly possible. Instead, the authors suggest the following expression for maximisation:

$$\mathcal{L} \propto \mathrm{E}_{q(\boldsymbol{\beta})}\left[\log p(\boldsymbol{\beta}) - \log q(\boldsymbol{\beta})\right] + \sum_{n=1}^{N} \mathrm{E}_{q(\boldsymbol{\beta})q(\boldsymbol{z}_n|\boldsymbol{x}_n,\boldsymbol{\beta})}\left[\log \frac{p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})}{q(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})}\right] \tag{8}$$

It is worth noting that the authors define $q(\boldsymbol{x}_n)$ to be the empirical distribution on the observations. To derive the above expression, they also propose the following factorisation: $q(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta}) = q(\boldsymbol{x}_n)q(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\beta})$. This also implies conditional independence between $\boldsymbol{x}_n$ and $\boldsymbol{\beta}$ over q:

$$q(\boldsymbol{x}_n) = q(\boldsymbol{x}_n|\boldsymbol{\beta}) \tag{9}$$

The authors show that maximising (8) is equivalent to maximising (7). It is still not possible to directly evaluate the density ratio $\log \frac{p(\boldsymbol{x}_n,\boldsymbol{z}_n|\boldsymbol{\beta})}{q(\boldsymbol{x}_n,\boldsymbol{z}_n|\boldsymbol{\beta})}$ because of the assumed implicit distributions. However, it is possible to form a class probability estimator which can be used to produce a useful proxy to the ratio:

$$\mathcal{D} = \mathrm{E}_{p(\boldsymbol{x}_n,\boldsymbol{z}_n|\boldsymbol{\beta})}\left[-\log \sigma(r(\boldsymbol{x}_n, \boldsymbol{z}_n, \boldsymbol{\beta}; \boldsymbol{\theta}))\right] + \mathrm{E}_{q(\boldsymbol{x}_n,\boldsymbol{z}_n|\boldsymbol{\beta})}\left[-\log(1 - \sigma(r(\boldsymbol{x}_n, \boldsymbol{z}_n, \boldsymbol{\beta}; \boldsymbol{\theta})))\right] \tag{10}$$

where

- $\sigma(\cdot)$ is the sigmoid function.
- $r(\cdot; \boldsymbol{\theta})$ is a discriminator neural network whose weights are parametrised by $\boldsymbol{\theta}$. The inputs of the network are samples from either the generative model or the approximate posterior and the empirical observations. The output of the network is a real number. A large, positive output value indicates that the network judges the sample to come from $p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})$. A large, negative output value indicates that the network judges the sample to come from $q(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})$.

If $\mathcal{D}$ is being minimised, then it can be shown that the optimal value of $r(\cdot; \boldsymbol{\theta})$ is:

$$r^*(\boldsymbol{x}_n, \boldsymbol{z}_n, \boldsymbol{\beta}) = \log \frac{p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})}{q(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})}$$

Therefore, if $r(\cdot; \boldsymbol{\theta})$ is expressive enough to approach $r^*(\cdot)$, it can serve as a proxy for the log-ratio in equation (8).

Now, (8) can be maximised via the maximisation of:

$$\mathrm{E}_{q(\boldsymbol{\beta})}\left[\log p(\boldsymbol{\beta}) - \log q(\boldsymbol{\beta})\right] + \sum_{n=1}^{N} \mathrm{E}_{q(\boldsymbol{\beta})q(\boldsymbol{z}_n|\boldsymbol{x}_n,\boldsymbol{\beta})}\left[r(\boldsymbol{x}_n, \boldsymbol{z}_n, \boldsymbol{\beta}; \boldsymbol{\theta})\right] \tag{11}$$

2. Connection to generative adversarial networks

The framework outlined so far can be viewed as having generative and discriminative components. The generative components are: $p(\boldsymbol{\beta})$, $p(\boldsymbol{z}_n|\boldsymbol{\beta})$ and $p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\beta})$. The discriminative component is the ratio estimator $r(\cdot)$.

If certain assumptions are made, this framework can be seen as similar to generative adversarial networks (GANs). The main goal of GANs is to approximate the empirical data distribution with an implicit distribution. This is achieved via creating an adversarial scenario between a generator G and a discriminator D neural networks. The generator G is fed random noise and typically outputs a raster image. This defines an implicit distribution. The discriminator's goal is to distinguish between samples drawn from the real, empirical distribution versus samples generated by G. In turn, G's task is to confuse the discriminator D: it tries to make D less certain in its judgement about which samples are generated. Mathematically, the original paper by Goodfellow et al. [4] phrases this as the following minimax game:

$$\min_G \max_D \mathrm{E}_{x \sim p_{\text{data}}(x)} [\log D(\boldsymbol{x})] + \mathrm{E}_{z \sim p_z(z)} [\log(1 - D(G(\boldsymbol{z})))] \tag{12}$$

where

- $p_{\text{data}}(\boldsymbol{x})$ is the empirical data distribution.
- $p_z(\boldsymbol{z})$ is the noise distribution whose samples are fed to the generator.

The maximisation operator in (12) incentivises D to assign the value 1 to samples drawn from the empirical distribution and 0 to samples generated by G. In opposition to this, the minimisation operator incentivises G to generate samples that make D assign the value 1 to samples generated by G.

For the purposes of present work, it is interesting to note that the technique of likelihood-free variational inference can be made similar to GAN training under the following conditions:

1. $q(\boldsymbol{\beta})$ is a point mass distribution, so sampling from it always yields the same $\boldsymbol{\beta}$.
2. The hidden codes $\boldsymbol{z}_n$ are ignored and inference only targets $\boldsymbol{\beta}$.

This simplifies (7) to:

$$\mathrm{E}_{q(\boldsymbol{\beta}|\boldsymbol{x})} [\log p(\boldsymbol{x}, \boldsymbol{\beta}) - \log q(\boldsymbol{\beta}|\boldsymbol{x})] \tag{13}$$

Then (8) can be written as:

$$\mathrm{E}_{q(\boldsymbol{\beta})} [\log p(\boldsymbol{\beta}) - \log q(\boldsymbol{\beta})] + \sum_{n=1}^{N} \mathrm{E}_{q(\boldsymbol{\beta})} \left[\log \frac{p(\boldsymbol{x}_n|\boldsymbol{\beta})}{q(\boldsymbol{x}_n|\boldsymbol{\beta})}\right] = \\ \log p(\boldsymbol{\beta}) + \sum_{n=1}^{N} \mathrm{E}_{q(\boldsymbol{\beta})} \left[\log \frac{p(\boldsymbol{x}_n|\boldsymbol{\beta})}{q(\boldsymbol{x}_n|\boldsymbol{\beta})}\right]$$

Similarly to (11), the practical objective considered for maximisation becomes:

$$\log p(\boldsymbol{\beta}) + \sum_{n=1}^{N} \mathrm{E}_{q(\boldsymbol{\beta})}\left[r(\boldsymbol{x_n}, \boldsymbol{\beta}; \boldsymbol{\theta})\right] = \log p(\boldsymbol{\beta}) + \sum_{n=1}^{N} r(\boldsymbol{x_n}, \boldsymbol{\beta}; \boldsymbol{\theta}) \tag{14}$$

The reduced ratio objective can be written as:

$$\mathrm{E}_{p(\boldsymbol{x_n}|\boldsymbol{\beta})}\left[-\log \sigma(r(\boldsymbol{x_n}, \boldsymbol{\beta}; \boldsymbol{\theta}))\right] + \mathrm{E}_{q(\boldsymbol{x_n}|\boldsymbol{\beta})}\left[-\log(1 - \sigma(r(\boldsymbol{x_n}, \boldsymbol{\beta}; \boldsymbol{\theta})))\right] \tag{15}$$

Tran et al. [11] note that making $r(\cdot; \boldsymbol{\theta})$ a function of $\boldsymbol{\beta}$ can be costly, especially if $\boldsymbol{\beta}$ describes the weights of a neural network. They suggest that instead of backpropagating through the variational objective (14), it is possible to backpropagate through the ratio objective (15). With this in mind, (15) can be reduced to:

$$\mathrm{E}_{p(\boldsymbol{x_n}|\boldsymbol{\beta})}\left[-\log \sigma(r(\boldsymbol{x_n}; \boldsymbol{\theta}))\right] + \mathrm{E}_{q(\boldsymbol{x_n}|\boldsymbol{\beta})}\left[-\log(1 - \sigma(r(\boldsymbol{x_n}; \boldsymbol{\theta})))\right]$$

Note also that $\boldsymbol{x_n}$ and $\boldsymbol{\beta}$ are assumed to be conditionally independent over q as noted in (9). The fact that $q(\boldsymbol{x_n}) = q(\boldsymbol{x_n}|\boldsymbol{\beta})$ simplifies the second expectation:

$$\mathrm{E}_{p(\boldsymbol{x_n}|\boldsymbol{\beta})}\left[-\log \sigma(r(\boldsymbol{x_n}; \boldsymbol{\theta}))\right] + \mathrm{E}_{q(\boldsymbol{x_n})}\left[-\log(1 - \sigma(r(\boldsymbol{x_n}; \boldsymbol{\theta})))\right] \tag{16}$$

Apart from a negativity sign, (12) and (16) have the same form when $D(\cdot) = \sigma(r(\cdot; \boldsymbol{\theta}))$. Note that $p_{\text{data}}(\boldsymbol{x})$ and $G(\boldsymbol{z}) \sim p_z(\boldsymbol{z})$ in (12) play the same role as $q(\boldsymbol{x_n})$ and $p(\boldsymbol{x_n}|\boldsymbol{\beta})$ in (16) respectively. Maximising (12) with regards to D is the same as minimising (16) with regards to $\boldsymbol{\theta}$. Minimising (12) with regards to G is the same as maximising (16) with regards to $\boldsymbol{\beta}$.

This shows that, under special circumstances, GAN training is very similar to likelihood-free variational inference. The only difference is the $\log p(\boldsymbol{\beta})$ regularisation term in (14).

This observation proves useful in section 4. where it helps explaining why the full model ultimately fails to converge.

3. Application to image vectorisation

The theoretical framework outlined in section 1. can plausibly be applied to the problem of image vectorisation stated in chapter 1. Although practical experiments were not successful, a description of the experimental setup follows. The first step is to consider what roles should the various distributions play in the image vectorisation setting. The setup investigated in practice involves the following components:

- $p(\boldsymbol{\beta})$: Multivariate normal prior. Samples from this distribution are assumed to parametrise a neural network's weights. Let $\text{enc}_{\boldsymbol{\beta}}(\cdot)$ denote this neural network.

- $q(\boldsymbol{\beta})$: Point mass approximation to the prior $p(\boldsymbol{\beta})$. Using a point mass approximation here instead of a proper distribution makes the problem less computationally intensive: if a point mass approximation is used, then sampling $q(\boldsymbol{\beta})$ is tantamount to using a single $\boldsymbol{\beta}$ when estimating the expectation in (11).

- $p(\boldsymbol{z}_n|\boldsymbol{\beta})$: Implicit conditional distribution for line vector coordinates, similar to the hidden codes in chapter 2 and chapter 3. Sampling from this distribution goes as follows:

  1. Sample $\boldsymbol{\epsilon}$ from a multivariate standard normal.
  2. Calculate $\mathrm{enc}_{\boldsymbol{\beta}}(\boldsymbol{\epsilon})$ by passing $\boldsymbol{\epsilon}$ through the neural network whose weights are parametrised by the point mass approximation from $q(\boldsymbol{\beta})$. The outputs of this step are interpreted as coordinates describing the start and end coordinates of lines.

- $p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\beta})$: Implicit conditional distribution outputting raster image samples. It is a point mass distribution because passing the same $\boldsymbol{z}_n$ through the neural network implementing it always results in the same output image. Similarly to chapter 3, a differentiable approximation to a non-differentiable renderer is used in order to help propagating gradients. Sampling from this distribution is done by passing $\boldsymbol{z}_n$ through the differentiable renderer.

- $q(\boldsymbol{z}_n|\boldsymbol{x}_n)$: Implicit conditional distribution implemented by a neural network that takes $\boldsymbol{x}_n$ as parameter and outputs $\boldsymbol{z}_n$. It is a point mass distribution because passing the same $\boldsymbol{x}_n$ through it always results in the same $\boldsymbol{z}_n$. This network can be interpreted as an inference network as its outputs $\boldsymbol{z}_n$ are supposed to be the line coordinates that give rise to the observed image $\boldsymbol{x}_n$ via $p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\beta})$.

  Note that $\boldsymbol{\beta}$ is not given as an input to the neural network here, perhaps in opposition to what (11) might suggest. Omitting $\boldsymbol{\beta}$ from the input is justified because a point mass approximation is used for $q(\boldsymbol{\beta})$: if samples from $q(\boldsymbol{\beta})$ always yield the same $\boldsymbol{\beta}$, then $q(\boldsymbol{z}_n|\boldsymbol{x}_n) = q(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\beta})$, or in other words $\boldsymbol{z}_n$ and $\boldsymbol{\beta}$ are assumed to be conditionally independent.

- $r(\cdot;\boldsymbol{\theta})$: Neural network serving as a discriminator between samples from $p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})$ and $q(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\beta})$. It is trained by minimising (10).

4. Results

Experiments with this setup were not successful as the model failed to converge. As a way of debugging the problem, the discriminator $r(\cdot;\boldsymbol{\theta})$ was made to ignore $\boldsymbol{z}_n$. This reduced the setup to a generative adversarial network [4] as explained in section 2.. However, the resulting simpler model still failed to learn a generative model for the images.

The GAN setup is prone to instabilities even in its default case with relatively simple neural networks. Many attempts were made by researchers to stabilise the training process, such as the one by Arjovsky et al. [1]. In present work, the addition of an approximate renderer (for generating samples from $p(\boldsymbol{x}_n|\boldsymbol{z}_n, \boldsymbol{\beta})$) to the GAN setup plausibly makes the inference task too complicated for stochastic gradient descent to learn it. In particular, the well-known problem of mode collapse was observed during training. This means that the generator neural network kept producing the same output, no matter the input noise fed into it. By careful tuning of the hyperparameters, this problem was eventually overcome. However, this resulted in the discriminator network becoming too strong compared to the generator. When this happened, the gradients used to train the generator became too small for the training to make meaningful progress. It is reasonable to conclude that the full model did not converge because the simplified, GAN-based objective did not converge either.

5. Related work

A recent paper by Ganin et al. [3] proposes a model called SPIRAL that is capable of synthesizing instructions to drive non-differentiable graphics engines. Similarly to the simplified formulation of likelihood-free variational inference outlined in section 2., they use a GAN-based adversarial criterion to optimise their model. As in the usual GAN setup, they define a generator and a discriminator component. The generator is responsible for producing the instructions fed to the non-differentiable engine, while the discriminator judges how convincing the rendered output is. The discriminator's judgement is used as the signal when training the generator via reinforcement learning.

SPIRAL is more general than present work as it allows for the inference of arbitrary structures as opposed to inference of lines only (although extensions to present work should be possible). The generator is implemented by a recurrent neural network, therefore the generated instructions can be of arbitrary length. This means that the model is not restricted to a predefined number of shapes per image.

The model's generality comes at the cost of increased architectural and training complexity. In particular, the authors note that they use a distributed version of reinforcement learning where up to 64 actors might be used for training. As an additional complication, the original formulation of the GAN objective (12) proved to be insufficient, therefore the authors had to use the Wasserstein training criterion [1]. Training recurrent neural networks (especially in the reinforcement learning context) is also known to be more difficult than training feedforward networks.

The model is capable of operating in either conditional or unconditional modes. In the unconditional case, the model generates images from random noise. Given that the training was conducted successfully, the generated set of images is expected to match the training dataset's distribution. In the conditional case, the model tries to reproduce an input image. In a similar spirit to the present work, the intermediate instructions used for the reproduction can be interpreted as infernece of the image's constituting vectors.

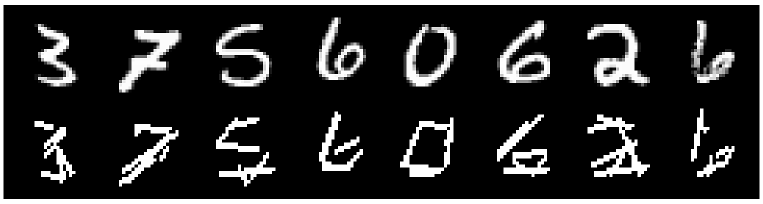
They manage to successfully train the model on real world datasets such as MNIST, OMNIGLOT and CELEBA. The reconstructions of MNIST and OMNIGLOT are of high quality. The reconstructions of CELEBA are blurry, but they still demonstrate the model's ability to infer global structures on the image. On a synthetic 3D dataset they demonstrate the model's generality by inferring object properties in 3 dimensional space too.

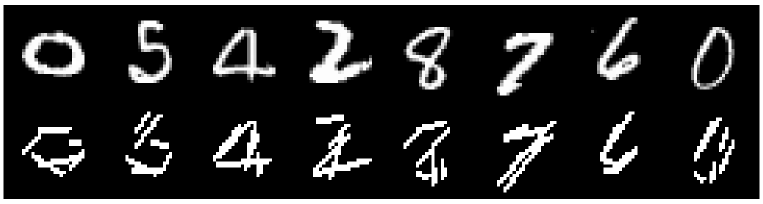
5 Conclusion

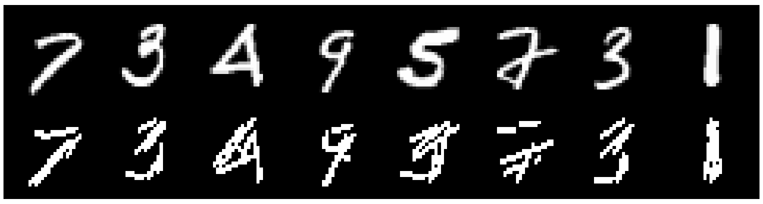
The investigated approaches vary greatly in their complexity and quality of their outputs. Appreciable results were only obtained using the techniques of traditional and variational autoencoders. Out of these two, variational autoencoders offer the better results in terms of quality but this comes at the cost of increased complexity of the variational objective and the renderer approximator. This complexity is arguably still below that of other alternative techniques, such as recurrent neural networks or reinforcement learning. Still, the promise of increased performance and the demands of richer real world datasets might plausibly justify the usage of more advanced techniques such as SPIRAL developed by Ganin et al. [3]. Given the amount of effort needed to produce high quality outputs, classical vectorisation algorithms can be a reasonable choice too.

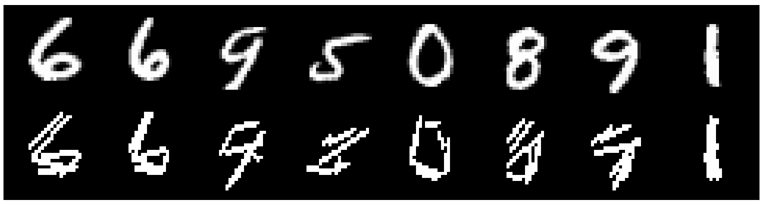
A Regularised variational autoencoder samples

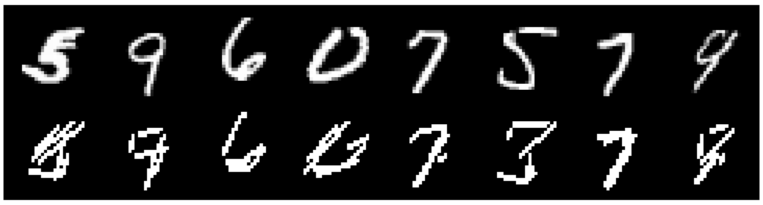
On this page, additional samples are included from the regularised variational autoencoder model. The first rows are the original images sampled from the MNIST database, the second rows show the model's vectorised reconstructions.

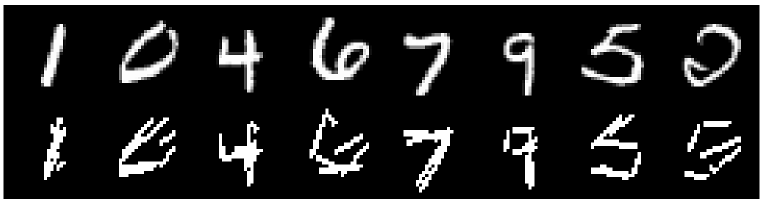






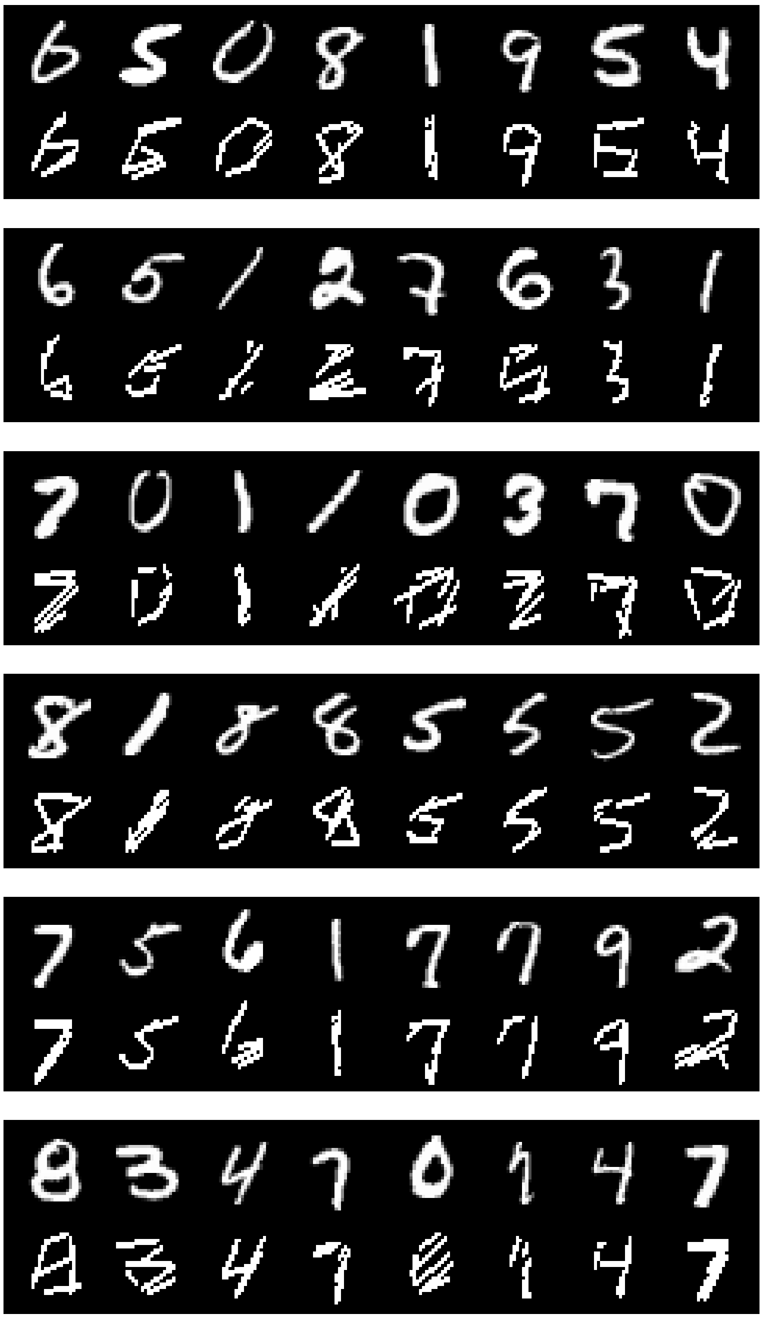






B Unregularised variational autoencoder samples

On this page, additional samples are included from the unregularised variational autoencoder model. The first rows are the original images sampled from the MNIST database, the second rows show the model's vectorised reconstructions.



Bibliography

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein GAN" (2017).

[2] Nat Dilokthanakul, Pedro A.M. Mediano, Marta Garnelo, Matthew C.H. Lee, Hugh Salimbeni, Kai Arulkumaran, and Murray Shanahan. "Deep Unsupervised Clustering with Gaussian Mixture Variational Autoencoders" (2016). URL: https://arxiv.org/abs/1611.02648.

[3] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S.M. Ali Eslami, and Oriol Vinyals. "Synthesizing Programs for Images using Reinforced Adversarial Learning" (2018). URL: https://arxiv.org/abs/1804.01118.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Networks" (2014). URL: https://arxiv.org/abs/1406.2661.

[5] David Ha and Douglas Eck. "A Neural Representation of Sketch Drawings" (2017).

[6] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" (2015). URL: https://arxiv.org/abs/1502.03167.

[7] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization" (2014). URL: https://arxiv.org/abs/1412.6980.

[8] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes" (2013). URL: https://arxiv.org/abs/1312.6114.

[9] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. "Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning" (2019). URL: https://arxiv.org/abs/1904.01786.

[10] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. "Adversarial Autoencoders" (2015). URL: https://arxiv.org/abs/1511.05644.

[11] Dustin Tran, Rajesh Ranganath, and David M. Blei. "Hierarchical Implicit Models and Likelihood-Free Variational Inference" (2017). URL: https://arxiv.org/abs/1702.08896.