Teaching Machines to Paint

Miguel Jaques

Master of Science
Artificial Intelligence
University of Edinburgh
2016

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Miguel Jaques*)

Abstract

The success of neural network-based methods has enabled the development of systems that emulate many human activities. In this work we approach the act of painting, and create a model that is able to reconstruct images by means of brushes or strokes, using a limited set of tools. To this end, the *Painter network* is developed. This network architecture is shown to be able to handle a variety of tasks, from simple images containing alphabet characters to larger and more complex images, like faces or natural scenes. The network is made end-to-end differentiable by using differentiable attention mechanisms like spatial transformers [1] and creating a technique to allow discrete actions to be taken in a differentiable way. Additionally, we show that it is possible to create an occlusion-aware system that improves the model's ability to understand scenes containing occlusions. We also dedicate a substantial portion of this work to showing that while sequential variational models like DRAW [2] and AIR [3] achieve good performance in their original tasks, they do not form an appropriate basis for a neural network-based painting model.

Acknowledgements

I want to thank my supervisor, Amos Storkey, for the guidance and insight provided throughout this project, especially during the first days when there was great uncertainty surrounding it. He always encouraged me to pursue the problems that interested me the most, while keeping me on a path towards tackling academically relevant issues.

I also thank Harri Edwards for all the discussions and help provided. He was instrumental in allowing me to go from the models' concepts to their implementation, and this project would not have gotten so far without his help during critical times.

Last but not least, I thank my family for their never-ending love and support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and goals

In this work we aim to develop a model that is able to reconstruct (and possibly generate) images given a limited set of painting tools/actions. Essentially, the goal is to create a network that emulates the human painting procedure, i.e., that understands the structure of an image and replicates it in terms of a sequential composition of shapes and colors.

This problem can be seen as unsupervised learning of image representations (Chapter 2), an area that has seen great advances in the last couple of years. Some notable models include DRAW [2], AIR [3] (generative and variational), GAN [4] (generative and adversarial), Ladder networks [5] (denoising and skip-connections) and DC-IGN [6] (vision as inverse graphics). What all of these models have in common is that the visual scope of the canvas updates is unconstrained, i.e., the patch that is written onto the canvas at each step can be the result of any neural network output. This is not at all similar to human painting, since we do not have the ability to manipulate an image at the pixel level in such an unconstrained manner. Instead, what we paint is constrained by the tools we have available: physical tools like pencils/brushes in physical painting or software tools like shapes/brushes/lines in software painting. Although at first this may seem a purely aesthetically interesting setting, there are several reasons why this is an interesting problem from a computer vision and representation learning research perspective:

- A system that aims to emulate a human painter needs to have an understanding of occlusions, a major topic in computer vision. For example, if we have two shapes of different colors, with one partially occluding the other, and we allow the network to replicate the image using only two moves, it must understand that there is an underlying order to the placement of these circles: bottom one first, top one second, and that this is the only correct order.
- How do the painting policies learned vary depending on the set of painting actions avail-

able? For example, how does the policy the network learns in order to reproduce MNIST digits differ if it is given a circular shape or a rectangular shape? What tools provide the most representational power?

- In typical unsupervised learning of images, the 'what' and the 'where' of image components are both learnable and unconstrained. What kind of learning difficulties arise on the 'where' side from the fact that in this case the 'what' is extremely constrained?

The main contributions of this work are the following:

- Showing that it is possible to create a scalable painting network system that is able to understand a variety of image settings, and that current sequential variational models that achieve state-of-the-art results in image modeling are not able to deal with this problem efficiently due to severe convergence problems.
- Showing that this painting network can be made end-to-end differentiable even in the presence of discrete quantities (through appropriate techniques), allowing for very efficient training.
- Showing that using dedicated layers to handle occlusion in autoencoder models significantly improves their ability to model scenes with occluded parts.

1.2 Structure of this work

Chapter 2 is an overview of unsupervised learning and representation learning, and of the most relevant approaches within the context of vision.

Chapter 3 describes the problem in more detail, including the fundamental settings that need to be in place before a painting network can be designed.

Chapter 4 explores sequential variational models as a base architecture for building a painting network, with experiments and analysis to explain why this kind of model, that is very successful in the unconstrained case, performs so poorly in a constrained setting

Chapter 5 describes a successful painting network architecture, the *Painter network*, along with extensive experimentation in a variety of settings in order to test the limits of such models and identify shortcomings.

1.3 Experimental details

The code for this work was written in Python using the TensorFlow framework [7], and additional libraries used for other tasks include Numpy, Scipy, Matplotlib, PIL and OpenCV.

All experiments were run using the Adam optimizer algorithm [8] with a learning rate of 1×10^{-3} (or 1×10^{-4}, where explicitly stated), the default parameters $\beta_1 = 0.999$ and $\beta_2 = 0.9$,

and batch size 100. Throughout this work we always refer to the training process in terms of the number of batch iterations instead of epochs (one pass over all the batches), because some visualizations contain important information at the sub-epoch level, like those of the training progress (e.g. Fig 4.6).

Regarding the plots, the shaded area surrounding the main plot lines represents the standard deviation of each 100 batch iteration samples. This is made to smooth the plot and make visualizations less cluttered.

Chapter 2

Unsupervised learning of image representations

When looking at a picture, we do not think of it as simply a mesh of retinal gland red, blue or green light excitations: through intricate neural processing, the visual cortex is able to turn these raw inputs into concepts that it can then use to interpret what it sees. When looking at the image of a person, we think of the person in terms of the features that compose it, both high level (two arms, two legs, one head, etc.) and low level (curly hair, pointy nose, blue eyes, etc). In principle, given enough of these features we can perfectly describe any scene. These features are usually called *representations*, and a model that is able to construct the representations needed to provide machines with the same visual ability as humans constitutes the holy grail of computer vision. However seamless this process may be for human brains, it has proven to be a tremendous challenge to understand and model. Ever since the inception of computer vision as a research area, researchers have tried to construct hand-crafted feature extractors (e.g. Gabor filter, SIFT [9]) that could provide the features to machine learning methods in order to tackle the problems of object recognition, detection and tracking. Even though these methods provided reasonable performance on small context-specific problems, they never got close to being able to provide a framework able to rival the human visual system.

Concurrently, researchers in machine learning and probabilistic modeling developed more general models that could learn the optimal features necessary to solve a given task, such as Restricted Boltzmann Machines, Bayesian Networks (stochastic) and Neural Networks (deterministic). These (especially the latter, with the advent of the backpropagation algorithm [10]) would come to provide much more powerful and general feature extraction mechanisms.

As a field of machine learning, unsupervised learning refers to the task of constructing useful representations from unlabeled data. For example, when looking at Fig. 2.1 (left) a human is able to immediately identify 3 clusters in the point cloud (Fig. 2.1 (right)), even though no labels were given. In doing so we are creating an alternative representation for each datapoint

(the cluster to which it belongs) which might make learning more efficient in a supervised setting.

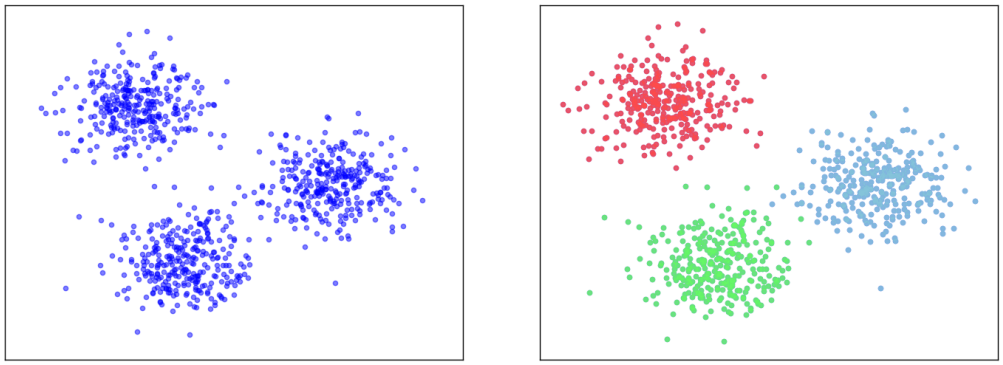


Figure 2.1: Point cloud (left) with a natural division into clusters (right).

This is a well known problem in machine learning: the data representation chosen is one of the most influential factors in the learning ability of a model. Let us take logistic regression, which can achieve 100% accuracy as long as the dataset is linearly separable by class. If we use the original representation of some data that is radially distributed (Fig. 2.2 (left)), hence not linearly separable, logistic regression would only achieve around 50% accuracy. However, if we change from Cartesian to polar coordinates (Fig. 2.2 (right)) the classes become linearly separable, and logistic regression is able to correctly predict every point in the training data. This very simple example shows the importance of the representations used. Of course, the power of the model is also an important factor. The data representation becomes even more important when handling high-dimensional data: the higher the dimensionality, the harder it is for a machine learning model to fit the training data manifold while generalizing correctly to unseen data.

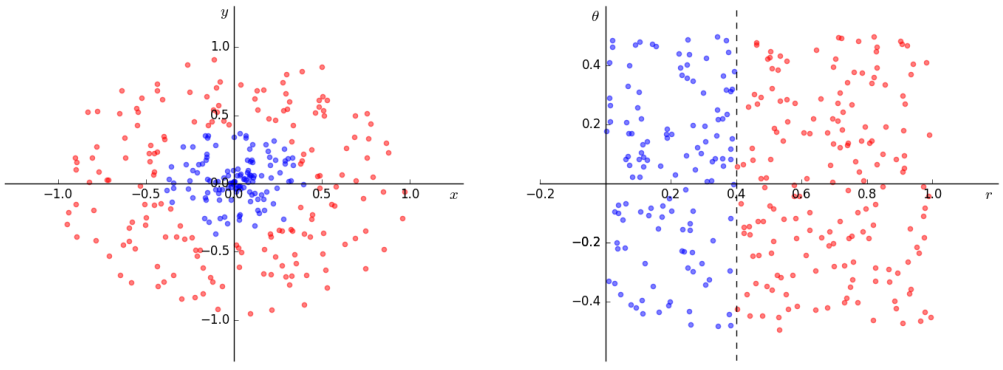


Figure 2.2: Points belonging to two classes (red and blue) distributed in a radial manner. In Cartesian coordinates (left) the data is not linearly separable, but it becomes linearly separable when changing to polar coordinates (right).

Neural networks (or multi-layer perceptrons) [11, 12] are, at this point, the method that provides the most scalable yet flexible function approximations. Unlike other methods, the general training procedure for neural networks is quite straightforward: in order to minimize an error objective, we can use gradient descent techniques by efficiently computing the derivatives with respect to network's parameters via the backpropagation algorithm [10]. Deep neural networks work even for high-dimensional data, such as images or speech, because they create a hierarchy of representations, with lower layers capturing finer details and higher layers constructing more abstract concepts, which can be used by a classifier in the top layer. Neural networks excel at this task because they can learn much more complex feature representations than those provided by methods like Principal Component Analysis (PCA) or Gaussan Mixture Models (GMM) for clustering, which are too limited to be useful in complex high-dimensional data like images. Representation learning can be seen as a field spanning supervised, unsupervised and reinforcement learning, and has been the subject of active research for several decades now. For a comprehensive overview of this subject see [13].

Although neural networks have existed for some decades, it was only in 2012, with the use of GPU accelerated hardware to allow faster training of deeper architectures [14], that they were shown to be able to tackle challenging large-scale machine learning and pattern recognition problems. Since then we have witnessed a true revolution in AI, with deep learning systems achieving unprecedented (and even super human) results in several tasks such as image recognition [15, 16], speech recognition [17], machine translation [18] and game-playing [19, 20]. Such results are obtained by training networks on extremely large labeled datasets where pairs (input, target output) are available. Despite the good results obtained, this approach has two problems:

1. Building such datasets is costly. In order to construct large datasets every single datapoint has to be labeled by a human annotator. It is easy to see that when the task at hand is, for example, object recognition with thousands of possible labels or image segmentation where each pixel has to be assigned to a class, building a large dataset becomes an incredibly time-consuming endeavor.

2. This is not how humans learn. In order to learn the concept of 'car' we do not need to be given every single variation of a car in shape, color, etc. and be told 'Look, this is a car!' In fact, with just a few examples (usually not more than a handful) we can understand a concept well enough to extrapolate/generalize correctly to unseen examples. This is because we have, just by observing the world around us created a good representation of what objects look like and their characteristics and variations, even if we do not know their name. A neural network trained on a handful of labeled examples would overfit the training data, rendering it unable to generalize correctly.

Given the abundance of unlabeled data available today, it is easy to see why unsupervised

learning is important: being able to build robust representations from unlabeled data would allow us to create machine learning systems capable of greatly surpassing current systems even if just provided a small amount of labeled data, since they would be able harness these representations and efficiently use them to solve the task at hand.

In the following sections we present an overview of the most relevant approaches in unsupervised learning of images. These approaches will form the basis to the models developed in later sections.

2.1 Autoencoders and Denoising Autoencoders

One of the simplest ways to learn unsupervised representations is by training a neural network to reconstruct its input. Such a network is usually composed of an *encoder* that constructs a representation (or *code*) $\mathbf{z} = f_{enc}(\mathbf{x})$ and a *decoder* that uses that representation to reconstruct the original input, $\hat{\mathbf{x}} = f_{dec}(\mathbf{z})$ (Fig. 2.3, left), with $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{z} \in \mathbb{R}^n$. This encoder-decoder structure is referred to as an *autoencoder* (AE). Since the network structure is deterministic, it can easily be trained end-to-end using backpropagation by minimizing the reconstruction error, $E(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$.

When $n < m$ the autoencoder performs dimensionality reduction (e.g. when using a linear encoder with squared error the encoder learns the same representation as PCA), which can be seen as compression, as $\mathbf{z}$ has to contain all the information necessary for the decoder to perform the reconstruction of the higher-dimensional $\mathbf{x}$. However, instead of constructing a *dense* representation $\mathbf{z}$, one can use $n > m$ in order to construct *overcomplete* representations. In this case it is necessary to impose some regularization term on $\mathbf{z}$ in order to prevent the autoencoder from learning a useless identity transformation. This usually involves applying some sparsity regularization (such as L1) that causes the model to learn sparse features [21]. This is naturally related to Sparse Coding, which takes inspiration from the way the brain processes and stores information (e.g. [22]).

A more interesting structure is that of *denoising autoencoders* (DAEs) [23]. Here the encoder is not given the original input $\mathbf{x}$ but a noisy version of it, $\tilde{\mathbf{x}}$, so the autoencoder has to reconstruct data although it only ever sees its corrupted version (see Fig. 2.3 (right)). One attractive property of denoising autoencoders is that although a probabilistic model is not explicitly defined, it can be shown that the denoising objective using Gaussian noise and squared error is equivalent to the score matching objective [24, 25]. Furthermore, the representations learned with DAEs are more robust than those of normal autoencoders because the DAE has to capture the underlying structure of the data in order to undo the noise corruptions.

A major advantage of the autoencoder/denoising autoencoder framework is that the error objective is easy to minimize, making training more efficient and less prone to local minima than its variational counterpart, covered in Section 2.2.1. Unlike the latter, however, it does

not have an explicit probabilistic formulation, making operations like sampling from a prior not straightforward. Nonetheless, there is recent research [26, 27] that shows that denoising autoencoders can be interpreted probabilistically and generative sampling can in fact be done (although it requires several passes through the network). The denoising framework is also the base for recent models like the Ladder networks [5, 28], which have achieved competitive results in semi-supervised tasks, highlighting the quality of the unsupervised features learned in this setting.

From the description in this section, it is possible to realize that a painting system has all the characteristics of an autoencoder, so it should be developed under this framework: given an input image, the goal of the model is to obtain the best possible reconstruction using a limited number of moves and a fixed set of tools. However, by using a more formal probabilistic formulation it may be possible not only to reconstruct an image but also to produce new unseen images. This would be called a generative model of images, which we cover in the following section.

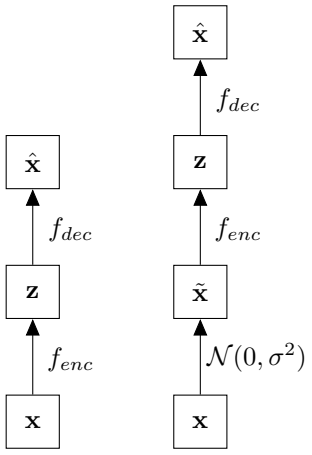


Figure 2.3: Autoencoder (left) and Denoising autoencoder (right).

2.2 Generative latent variable models

In generative latent variable models we want to model the joint distribution $p(\mathbf{x}, \mathbf{z})$, where $\mathbf{x}$ are the observations and $\mathbf{z}$ are the corresponding latent representations, in order to maximize the likelihood $p(\mathcal{D})$ of the visible data $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$ (although in some cases the quantity $\mathbb{E}_{p(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{x}, \mathbf{z})]$ is maximized instead). It is well known that in order to perform learning it is necessary to be able to perform inference. For simple models like Hidden Markov Models or Gaussian Mixture Models, the EM algorithm [29] can be used efficiently because, for both cases, the E-step has closed form, i.e., the posterior is computable using Bayes' rule:

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{\int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\ d\mathbf{z}} \tag{2.1}$$

Unfortunately, the integral in (2.1) is computable only for a limited family of classes with limited expressive power. Therefore, for large-scale and high-dimensional problems we have to resort to approximate inference methods. For example, for more general directed (Deep Belief Networks) and undirected (Deep Boltzmann Machines [30]) graphical models, $p(\mathbf{x})$ cannot be computed exactly, so it was necessary to develop special purpose algorithms to address this difficulty (e.g. [31, 32, 33]). Even then, these methods scaled poorly since they usually involve sampling within each inference/learning cycle.

Below we describe some of the most recent approximate inference methods that have been at the center of the progress made in generative modeling in the last couple of years due to their ability to handle inference and learning in large datasets and high dimensional data efficiently.

2.2.1 Variational Autoencoder

One of the most popular and influential approximate inference methods is variational inference [34]. Say we want to model the joint probability $p_\theta(\mathbf{x}, \mathbf{z})$, with parameters θ. In variational inference, we approximate the exact posterior $p_\theta(\mathbf{z}|\mathbf{x})$ directly with a function $q_\phi(\mathbf{z}|\mathbf{x})$ with parameters ϕ. The goal is to minimize the Kullback-Leibler (KL) divergence between the approximate and the true posterior, $KL(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x}))$. This term can be manipulated in the following way:

$$\begin{aligned} KL(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x})) &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x})}\right] \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{x}, \mathbf{z})}\right] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x})\right] \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{x}, \mathbf{z})}\right] + \log p_\theta(\mathbf{x}) \end{aligned}$$

where we used $p_\theta(\mathbf{z}|\mathbf{x}) = p_\theta(\mathbf{x}, \mathbf{z})/p_\theta(\mathbf{x})$. Using the fact that $KL(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x})) \geq 0$ and $p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z})/p_\theta(\mathbf{z})$ we can obtain a variational lower bound $\mathcal{L}$ on the marginal likelihood:

$$\log p_\theta(\mathbf{x}) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})}\right] = \mathcal{L} \tag{2.2}$$

where the equivalent expression for an i.i.d. dataset $\mathcal{D}$ is straightforward. This way we can train this model by maximizing the lower bound $\mathcal{L}$ with respect to the parameters θ and ϕ.

However, the gradients $\nabla_\theta \mathcal{L}$ and $\nabla_\phi \mathcal{L}$ are hard to compute exactly because the expectations do not always have a closed form. A technique that allows end-to-end training of such networks was introduced by [35, 36], called *stochastic backpropagation*, and was one of the main contributors to the recent popularity and success of generative models. If $\mathbf{z}$ is continuous, the authors realized that for many families of distributions it is possible to write a sample $\mathbf{z}^{(i,j)} \sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$ as a deterministic function of the input $\mathbf{x}^{(i)}$ and an independent random variable $\epsilon^{(j)}$, i.e., $\mathbf{z}^{(i,j)} = g_\phi(\mathbf{x}^{(i)}, \epsilon^{(j)})$. This means that it is possible to write the expectation in (2.2) with respect to $q_\phi(\mathbf{z}|\mathbf{x})$ such that it is possible to differentiate this expectation with respect to ϕ. This is called the *reparametrization trick*.

For example, in the most common form $q_\phi(\mathbf{z}|\mathbf{x})$ is modeled as Gaussian, with the mean and variance being outputs of MLPs with parameters ϕ, i.e., $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_\phi(\mathbf{x}), \text{diag}(\boldsymbol{\sigma}^2_\phi(\mathbf{x})))$; this is the encoder part of the network. In this case the reparametrization is $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \cdot \boldsymbol{\epsilon}$, with $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$. The prior is taken to be a standard Gaussian, $p_\theta(\mathbf{z}) = \mathcal{N}(0, \mathbf{I})$ (so

$p_\theta(\mathbf{z}) \equiv p(\mathbf{z})$), and the likelihood term $p_\theta(\mathbf{x}|\mathbf{z})$ is a cross-entropy (CE) or Gaussian loss (for discrete or continuous data respectively), where the reconstruction $\hat{\mathbf{x}}$ given $\mathbf{z}$ is the output of an MLP decoder with parameters θ, i.e., $p_\theta(\mathbf{x}|\mathbf{z}) = CE(\mathbf{x}|\hat{\mathbf{x}})$, $\hat{\mathbf{x}} = f_{dec}(\mathbf{z})$. The setting described in this paragraph is referred to as a *Variational Autoencoder* (VAE). Here the lower bound $\mathcal{L}$ can be written as:

$$\begin{aligned}
\mathcal{L}(\mathbf{x}^{(i)}) &= \mathcal{L}^z(\mathbf{x}^{(i)}) + \mathcal{L}^x(\mathbf{x}^{(i)}) \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log \frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\right] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})\right] \qquad (2.3) \\
&\approx \frac{1}{2}\sum_{d=1}^{D}\left(1 + 2\log(\boldsymbol{\sigma}_d^{(i)}) - (\boldsymbol{\mu}_d^{(i)})^2 - (\boldsymbol{\sigma}_d^{(i)})^2\right) + \frac{1}{J}\sum_{j=1}^{J}\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,j)}) \qquad (2.4)
\end{aligned}$$

where D is the dimensionality of $\mathbf{z}$, $\mathbf{z}^{(i,j)} \sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$, and we used $\boldsymbol{\mu}_d^{(i)} \equiv \boldsymbol{\mu}_{\phi,d}(\mathbf{x}^{(i)})$ and $\boldsymbol{\sigma}_d^{(i)} \equiv \boldsymbol{\sigma}_{\phi,d}(\mathbf{x}^{(i)})$ for ease of notation. The first expectation, known as *latent cost*, has a closed form for this particular case of a Gaussian prior and posterior but the second, known as *reconstruction cost*, must be approximated by sampling $\mathbf{z}^{(i,j)}$ from the posterior as described above. In the general form, the gradients of (2.3) w.r.t θ and ϕ are:

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\mathbf{x}^{(i)}) &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\left[\nabla_\theta \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})\right] \\
\nabla_\phi \mathcal{L}(\mathbf{x}^{(i)}) &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log \frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\nabla_\phi q_\phi(\mathbf{z}|\mathbf{x}^{(i)})\right]
\end{aligned}$$

and for the particular case of the VAE equations (2.4) we get:

$$\nabla_\theta \mathcal{L}(\mathbf{x}^{(i)}) \approx \frac{1}{J}\sum_{j=1}^{J}\nabla_\theta \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,j)}) \qquad (2.5)$$

$$\nabla_\phi \mathcal{L}(\mathbf{x}^{(i)}) = \frac{1}{2}\sum_{d=1}^{D}\left(1 + 2\frac{\nabla_\phi \boldsymbol{\sigma}_d^{(i)}}{\boldsymbol{\sigma}_d^{(i)}} - 2\,\boldsymbol{\mu}_d^{(i)}\,\nabla_\phi \boldsymbol{\mu}_d^{(i)} - 2\,\boldsymbol{\sigma}_d^{(i)}\,\nabla_\phi \boldsymbol{\sigma}_d^{(i)}\right) \qquad (2.6)$$

where the gradients $\nabla_\theta \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,j)})$, $\nabla_\phi \boldsymbol{\mu}_d^{(i)}$ and $\nabla_\phi \boldsymbol{\sigma}_d^{(i)}$ can be readily computed using backpropagation.

In later work, [37] proposed a generalization of this stochastic gradient mechanism to variational models involving discrete variables, although in that case the error gradients are more akin to those of a policy gradient method, like REINFORCE [38] (see Section 2.3.1 for an overview). However, the gradients still cannot be passed from the decoder to the encoder in a straightforward manner as with continuous variable, due to the discrete nature of the variables.

The reparametrization trick is important because it allows stochastic gradients to be backpropagated directly from the decoder to the encoder, and the use of MLPs in the parametrization together with the ability to do stochastic backpropagation allows these variational models to be much more efficient and scalable. Since its publication, many extensions and improvements

to its formulation have been explored, such as [39, 40, 41].

The VAE architecture is the base to some of the most successful sequential generative models of images, two of which serve as the base to half of the models and experiments developed in this work: DRAW [2] and AIR [3], which are described in more detail in Sections 2.2.2 and 2.2.3, respectively.

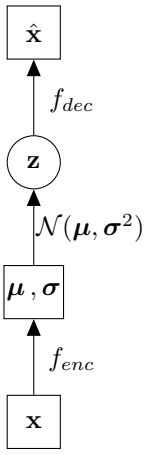


Figure 2.4: Variational Autoencoder.

2.2.2 DRAW

The *Deep Recurrent Attentive Writer* (DRAW) [2], was the first model that successfully employed VAEs in a sequential manner in order to greatly improve the performance of generative models. Here 'sequential' is the keyword: instead of trying to model the whole picture at once, DRAW iteratively reads from the image given and writes to the canvas, such that the encoder-decoder at each time step can represent small fractions of the image, improving its representational capabilities. This is in line with the discussion on visual attention in Section 2.3.

The full computational graph is shown in Fig. 2.5. In summary, at each time step t, a portion $\mathbf{r}_t$ of the image $\mathbf{x}$ is obtained using a *read* operation that uses soft Gaussian attention (see details in Section 2.3.2). This $\mathbf{r}_t$ is then passed as input to the encoder RNN, RNN^{enc}, and its output $\mathbf{h}_t^{enc}$ is passed to the functions $\boldsymbol{\mu}_\phi(\cdot)$ and $\boldsymbol{\sigma}_\phi(\cdot)$ that parametrize the latent distribution from which $\mathbf{z}_t$ is sampled, $\mathbf{z}_t \sim q_\phi(\mathbf{z}|\mathbf{h}_t^{enc})$. This sample is passed to the decoder RNN, RNN^{dec}, whose output $\mathbf{h}_t^{dec}$ is passed to a *write* operator that uses the same attention mechanism to write a patch onto the canvas $\mathbf{c}_{t-1}$. The canvas is then updated according to an additive rule $\mathbf{c}_t = \mathbf{c}_{t-1} + write(\mathbf{h}_t^{dec})$. This canvas has unbounded values, so the sigmoid function[1] is applied in order to obtain values between 0 and 1.

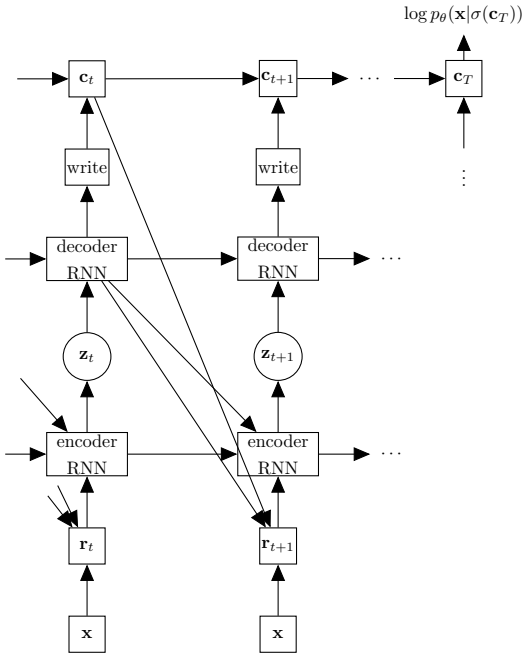


Figure 2.5: DRAW model.

[1] We use σ to symbolize the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$; not to be confused with a standard deviation.

The list of operations with all the dependencies shown in the graphical model is:

$$\hat{\mathbf{x}}_t = \mathbf{x}_t - \sigma(\mathbf{c}_{t-1}) \tag{2.7}$$

$$\mathbf{r}_t = read(\mathbf{x}_t, \hat{\mathbf{x}}_t, \mathbf{h}_{t-1}^{dec}) \tag{2.8}$$

$$\mathbf{h}_t^{enc} = RNN^{enc}(\mathbf{h}_{t-1}^{enc}, \mathbf{r}_t, \mathbf{h}_{t-1}^{dec}) \tag{2.9}$$

$$\mathbf{z}_t \sim q_\phi(\mathbf{z}|\mathbf{h}_t^{enc}) \tag{2.10}$$

$$\mathbf{h}_t^{dec} = RNN^{dec}(\mathbf{h}_{t-1}^{dec}, \mathbf{z}_t) \tag{2.11}$$

$$\mathbf{c}_t = \mathbf{c}_{t-1} + write(\mathbf{h}_t^{dec}) \tag{2.12}$$

where $\hat{x}_t$ is the error canvas at each time step. Both the encoder and decoder RNNs are *Long Short Term Memories* (LSTM) [42].

Substituting the posterior sampling of (2.10) for sampling from the prior $\mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I})$ and running steps (2.11) and (2.12) makes it possible to generate samples from the model.

One of the advantages of the DRAW model is that even though its structure is quite complex, it is fully differentiable, since the posterior, prior and likelihood have the same form as the VAE and the attention mechanism is differentiable. In fact, one of the main contributions of this work was the formulation of this soft attention mechanism for images and showing that the use of attention within a recurrent structure improves the lower bound of the marginal likelihood significantly. However, throughout this work all DRAW models and derivations thereof will use spatial transformers (see Section 2.3.3) instead of the original attention mechanism, since the former are more general and flexible.

The lower bound here is very similar to that of the VAE:

$$\begin{aligned}\mathcal{L}(\mathbf{x}^{(i)}) &= \mathcal{L}^z(\mathbf{x}^{(i)}) + \mathcal{L}^x(\mathbf{x}^{(i)}) \\ &\approx \frac{1}{2}\sum_{t=1}^{T}\sum_{d=1}^{D}\left(1 + 2\log(\boldsymbol{\sigma}_{d,t}^{(i)}) - (\boldsymbol{\mu}_{d,t}^{(i)})^2 - (\boldsymbol{\sigma}_{d,t}^{(i)})^2\right) - \frac{T}{2} + \frac{1}{J}\sum_{j=1}^{J}\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,j)})\end{aligned}$$

where $p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,j)}) = CE(\mathbf{x}^{(i)}|\sigma(\mathbf{c}_T^{(i)}))$, in the case of Bernoulli data. The gradients are equivalent to those in (2.5) and (2.6), and full backpropagation through time [43] is used in the LSTMs.

2.2.3 AIR and the problem of *what* and *where*

Although DRAW obtained great results in image modeling, it was not built to explicitly model the image as a composition of its parts. Humans naturally decompose an image into its parts, usually in a layered way: instead of identifying an image as a continuum of patterns, we have a statistical model of the objects (or layers) that compose it, such that any image can be interpreted as a superposition of known objects (within the respective statistical models). This

natural decomposition is usually referred to as the problem of *what* and *where*. It is easy to see why this is not a straightforward problem: in order to know the identity of an object we need to know its location, but in order to know the location of an object we need to be able to identify it (in fact, it is still not well known exactly how the cortex processes these two components; this is known as the *binding problem*). This is a particularly important problem in the context of building a network that can paint, since at each move it must decide what (brush, color, etc) to paint and where (position, orientation) to paint it.

Probably the most notable early attempt to try to provide the ability to model *what* and *where* in an unsupervised setting is the work on *capsules* [44, 45]. More recently, and using the more powerful VAE framework, [46] takes a step in the direction of modeling images as a composition of layers, by separating the content of a layer from its location, modeling them as interacting but separate stochastic variables, spatial transformers to provide flexibility and differentiability to the location modeling. AIR (*Attend, Infer, Repeat*) [3] takes this idea one step further by allowing a variable number of objects to be modeled in each image and an overall simpler encoder-decoder structure. In this work we only consider the fixed sequence length AIR, both for simplicity and because it suffices our needs here - refer to the original paper for the complete formulation.

In AIR the variational formulation is similar to DRAW, although here the stochastic variables $\mathbf{z}$ are split into a *what* and *where* component, $\mathbf{z} = (\mathbf{z}^{what}, \mathbf{z}^{where})$. $\mathbf{z}^{where}$ defines the location of a particular instance, like for example a digit on Multi-MNIST, and $\mathbf{z}^{what}$ defines the content of that instance, i.e., the autoencoding model of that digit. This way it is possible to explicitly encode an image in terms of a set of instances.

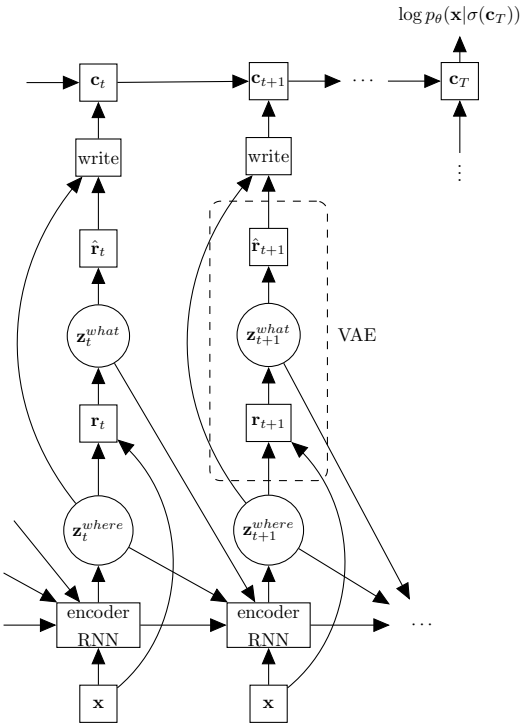


Figure 2.6: AIR model.

The full computational graph of the version used here is shown in Fig. 2.6. In summary, at each time step a recurrent state $\mathbf{h}_t^{enc}$ is computed by an encoder RNN, RNN^{enc}, which is used to compute the $\mathbf{z}_t^{where}$ stochastic variable. This variable defines the spatial transformer parameters that will be used to read a portion r_t of the input $\mathbf{x}$. A variational autoencoder is then applied to this portion yielding a reconstruction $\hat{\mathbf{r}}_t$. The stochastic variable $\mathbf{z}_t^{what}$ is equivalent to the z variable used in a standard VAE. This reconstructed portion is then placed back onto the canvas by inverting

the attention parameters defined by $\mathbf{z}_t^{where}$, once again using an additive canvas.

The list of operations, with all the dependencies shown in the graphical model is:

$$\mathbf{h}_t^{enc} = RNN^{enc}(\mathbf{h}_{t-1}^{enc}, \mathbf{z}_{t-1}^{where}, \mathbf{z}_{t-1}^{what}) \tag{2.13}$$

$$\mathbf{z}_t^{where} \sim q_\phi(\mathbf{z}^{where}|\mathbf{h}_t^{enc}) \tag{2.14}$$

$$\mathbf{r}_t = read(\mathbf{x}_t; \mathbf{z}_t^{where}) \tag{2.15}$$

$$\mathbf{z}_t^{what} \sim q_\phi(\mathbf{z}^{what}|\mathbf{r}_t) \tag{2.16}$$

$$\hat{r}_t = VAE^{dec}(\mathbf{z}_t^{what}) \tag{2.17}$$

$$\mathbf{c}_t = \mathbf{c}_{t-1} + write(\mathbf{r}_t; \mathbf{z}_t^{where}) \tag{2.18}$$

where $\hat{\mathbf{x}}_t$ is the error canvas at each time step. Once again, the encoder RNN is an LSTM. The model is fully differentiable and the approximate lower bound is exactly the same as the one used in DRAW, (2.13), with the stochastic sample here being $\mathbf{z}^{(i,j)} = (\mathbf{z}_{what}^{(i,j)}, \mathbf{z}_{where}^{(i,j)})$.

2.2.4 Other approaches

Although the line of research followed in this work is that of VAEs, there are several other approaches to image modeling, such as adversarial learning [4, 47, 48, 49] and density estimation [50, 51].

During the writing of this dissertation the authors of DRAW and AIR published follow-up papers [52, 53, 54] containing improvements and application to more complex settings. Although these models were not followed directly, they provided important additional insight into the behavior of this class of models.

2.3 Visual Attention

When looking at this text, the reader is probably not looking at the whole page at once in order to extract meaning from the text in it - each individual word is attended to, sequentially, as meaning is built for the sentence read. In fact, the images received by the retina are never interpreted all at once. When we look at a scene, our eyes naturally scan specific parts of the image in order to extract the relevant information and build a solid representation of it. Any process that takes all the information available and selectively processes just some part of it, such as these foveal glimpses in human vision, is a form of *attention*.

A visual system that does not use attention (such as a standard convolutional neural network) is required to densely process every single pixel in the image at its lowest scale, with much of this possibly useless information being carried to higher layers, potentially interfering with the information extracted from useful parts of the image. This problem can be fixed by building deeper and deeper neural networks, but with greater depth comes poorer scalability

(the number of operations scales quadratically with the image size). Furthermore, these structures are inherently not scale invariant, so it is common that many versions of the same image at different resolutions have to be passed to the network in order to capture visual cues at any levels (ref here). Attention mechanisms can solve both of these issues. Instead of processing the image in its original size, attention allows the system to process a low resolution version of the full image and use that to select the next locations on which to focus, and at what scale. This can be seen as a coarse-to-fine processing of images, which humans naturally do as well. Furthermore, by using a fixed-size glimpse smaller than the original image size, the number of operations scales only linearly with the number of glimpses, allowing the system to extract fine details only from locations that are relevant.

Although the concept of attention in AI is not new (going as far back as [55]), many advances have been made in recent years, to the point that there is now a wealth of results showing that attention is an essential mechanism in allowing a greater breadth of problems to be tackled, with some of its most notable uses including image caption and comprehension [56], machine translation [57], image generation [2, 52] and memory addressing [58]. Below we describe the three most common types of attention used, along with their advantages and disadvantages. Due to the abundance of literature in this topic we will restrict this analysis to vision-related tasks.

2.3.1 Hard attention

The first type of attention is *hard* or *stochastic* attention. Here the sequence of glimpses $\{\mathbf{x}_t\}_{t=1}^T$ is seen as the result of a sequence of actions $\{\mathbf{a}_t\}_{t=1}^T$, so finding the optimal sequence can be framed as a reinforcement learning problem: by defining an attention policy $\pi_\theta(\mathbf{a}_t|\mathbf{x}_{1:t-1}, \mathbf{a}_{1:t-1})$ from which actions are sampled, and a reward r_t at each step, it is possible to maximize the expected reward $J(\theta) = \mathbb{E}_{\pi_\theta}[R]$, where $R = \sum_{t=1}^T \gamma^{t-1} r_t$ is the discounted reward, by using the REINFORCE [38] policy gradient approximation

$$\nabla_\theta J(\theta) \approx \frac{1}{M} \sum_{m=1}^{M} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t^m|\mathbf{x}_{1:t-1}^m, \mathbf{a}_{1:t-1}^m)(R_t^m - b_t) \tag{2.19}$$

for M episodes/samples, and b_t are the so-called *baselines* [59, 37], which are used to reduce the variance of this gradient estimator. The gradient of the log-policy is simply a maximum likelihood gradient so if the policy is parametrized by an MLP, the gradients w.r.t. θ can be computed by backpropagation. This attention scheme has been used extensively for image recognition tasks such as in [60, 61, 56]. For instance, an action can correspond to 'read a square of side length s at location (x, y)', where the policy π_θ is a Gaussian distribution. This kind of stochastic policy can be incorporated into a more formal variational model as shown by [37].

An advantage of hard attention is that it can be applied to essentially any kind of task in

any form of data. For instance, this is the only possible approach for tasks where the actions are discrete, since backpropagation through these stochastic variables is not possible. However, there are known difficulties in training with these methods, such as the high variance of the gradient (2.19), which can lead to poor learning and the need to use the baselines mentioned to prevent such problem, which in turn add complexity to the system.

2.3.2 Soft attention

Even though hard attention is applicable in most cases, is it preferred to use a fully differentiable attention mechanism when possible, since learning is much more efficient in this case. Such differentiable mechanisms are called *deterministic* or *soft* attention.

The simplest example of soft attention is a convex combination of context vectors. Let us say we have a set of vectors $\mathbf{c} \equiv \{\mathbf{c}_m\}_{m=1}^M$, where each vector corresponds to a region of an image (say, squared regions in a grid). By taking a convex combination of this set of vectors, $\sum_{m=1}^M w_m \mathbf{c}_m$, where $\sum_{m=1}^M w_m = 1$ the model can choose to focus its attention on specific parts of the image, as done in [56]. Since the weights sum to 1, this convex combination is equivalent to an expectation over the weight distribution $\mathbb{E}_{p_\theta(w|\mathbf{c})}[\mathbf{c}]$. By parametrizing this distribution via some parameters θ (by, say, an MLP), the output of the expectation can be plugged anywhere in a neural network architecture and the gradients w.r.t. θ can be computed via backpropagation.

This particular method is not ideal for visual attention since it requires the image to be divided into a fixed grid. This also means that it cannot be used to perform writing attention, which is a fundamental part of any attention-based unsupervised learning model. A more interesting approach is that used in the DRAW model, which takes inspiration from [58]. Since this mechanism is not used or even considered in any way in this work, we do not cover it here in detail. Please refer to the original paper [2] for the full mathematical formulation.

In a soft setting every vector or component that can be attended to will be attended to, because (in practice) the weights will never be a one-of-k vector. This is a disadvantage relative to hard attention since it is not always desirable to attend to everything, even if softly. When this is acceptable, however, soft attention is much more efficient since training is not subject to the difficulties of stochastic variables and can easily be incorporated as part of an end-to-end differentiable system.

2.3.3 Spatial transformers

As we have seen, hard and soft attention both have some advantages and disadvantages. If we restrict ourselves to visual 2D attention we can use a third approach: *spatial transformers* [1]. Spatial transformers have the advantages of both hard and soft attention: like hard attention, the attended region can have hard boundaries (i.e., not every location has to be softly attended

to) and like soft attention, it is differentiable.

Spatial transformers achieve this in the following way. Given the pixel coordinates (x_i^U, y_i^U) of an input image $U \in \mathbb{R}^{W \times H}$, with height H and width W, the corresponding pixel coordinates (x_i^V, y_i^V) of the resulting image $V \in \mathbb{R}^{W' \times H'}$ are obtained through an affine transformation Φ:

$$\begin{pmatrix} x_i^U \\ y_i^U \end{pmatrix} = \Phi \begin{pmatrix} x_i^V \\ y_i^V \end{pmatrix} = \begin{pmatrix} \Phi_{11} & \Phi_{12} & \Phi_{13} \\ \Phi_{21} & \Phi_{22} & \Phi_{23} \end{pmatrix} \begin{pmatrix} x_i^V \\ y_i^V \end{pmatrix} \tag{2.20}$$

This means that the pixel values V_i of the output image at location (x_i^V, y_i^V) correspond to the pixel value U_i of the input image at location (x_i^U, y_i^U). This can be seen more clearly in Fig. 2.7. The parameters Φ_{11} and Φ_{22} control the scale, Φ_{12} and Φ_{21} the rotation/skewness, and Φ_{13} and Φ_{23} the shift of the attention window. Naturally, this mechanism is not restricted to affine transformations. For other types of transformations see in [1, 54].

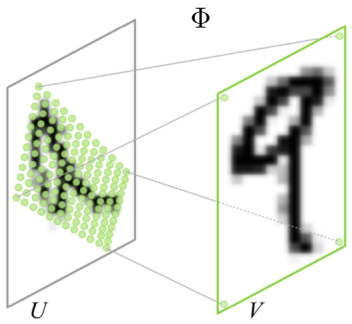


Figure 2.7: Sampling a portion of a source image U to target a target image V according to the affine parameters Φ. Image taken from [1].

Since the pixel locations are integers but the results of the transformation may yield real values, bilinear interpolation is used to obtain the output pixel values:

$$V_i = \sum_n^H \sum_m^W U_{nm} \max(0, 1 - |x_i^U - m|) \max(0, 1 - |y_i^U - n|) \tag{2.21}$$

where U_{nm} is the value at location (n, m) of the input and V_i is the output value for the pixel at location (x_i^V, y_i^V).

The derivatives of (2.21) with respect to the input values U_{nm} and the pixel locations (x_i^U, y_i^U) are:

$$\frac{\partial V_i}{\partial U_{nm}} = \sum_n^H \sum_m^W \max(0, 1 - |x_i^U - m|) \max(0, 1 - |y_i^U - n|)$$

$$\frac{\partial V_i}{\partial x_i^U} = \sum_n^H \sum_m^W \max(0, 1 - |y_i^U - n|) \begin{cases} 0 & \text{if } |m - x_i^U| \geq 1 \\ 1 & \text{if } m \geq x_i^U \\ -1 & \text{if } m < x_i^U \end{cases}$$

Since the parameters Φ are usually computed by some MLP (called the *localization network*), the equations above allow us to backpropagate an error signal from the output U to the input V, making the spatial transformer a sub-differentiable (i.e., differentiable with derivatives 0 almost everywhere) module that can be placed at any stage as part of a larger architecture. Additionally, this spatial transformers can be used both for reading attention (i.e. selecting a portion of an image) and writing attention (i.e. taking a patch and place it on the canvas according to some transformation).

Chapter 3

Building a template-based painter

As stated before, the main goal of this work is to design an autoencoding model of images where the writing (or painting) actions are constrained in its visual scope: unlike DRAW's *write* block (cf. [2]), where a visual patch is generated and then placed at some location and scale, here the network may only choose among a set of predefined shapes, e.g., circle, rectangle or line. These shapes will be called the *templates*, and they are defined more precisely (including the way they may be transformed and placed onto the canvas) throughout the rest of this chapter.

In order to perform a painting action, there are three components that need to be defined: the *where*, the *what*, and the *which*.

3.1 The *where* of templates

In order to efficiently perform painting actions and learning thereof the network needs to have a representation that makes it possible to place a template onto the canvas in any position/orientation in a differentiable way.

The most immediate and intuitive representation is to use the position and size of the template, represented by, for example, a tuple of the form $(x_{center}, y_{center}, \text{height}, \text{width})$, and render the desired shape using a shape-specific function (like those provided by some painting software, etc.). However, such a renderer makes this approach non-differentiable[1], hence only trainable through reinforcement learning, which is not efficient. This is a form of hard attention for writing, and suffers from the problems discussed in Section 2.3.1.

Instead, a much more general and convenient approach is used: the template is not defined by its categorical class (circle, rectangle, etc.) but by an $S \times S$ binary mask, $\mathbf{T}^{mask}$, with value 1

[1] There is some recent work done in differentiable graphics renderers. See [62].

in the shape's area and 0 elsewhere. This mask is then transformed into $\mathbf{p}^{mask}$ according to some affine parameters Φ using a spatial transformer (see Section 2.3.3) and written onto the canvas using some canvas update rule - in DRAW's case, additive. The parameters Φ will be encoded by the variable $\mathbf{h}^{where} \in \mathbb{R}^6$, which can correspond to the output of an MLP. The advantages of this approach are two-fold: firsty, this allows templates to be fully customizable, needing only to provide the respective mask, which can be created by hand in a simple way by using any image editing tool; secondly, and more importantly, this makes the process differentiable, allowing this template writing mechanism to be used as part of any autoencoding network and to be trained using backpropagation. This corresponds in essence to a differentiable 2D renderer. Throughout this work only affine transformations were used, since using a more flexible transformation would only increase complexity without providing additional insight. Since the template masks are binary, when they are transformed they produce a transformed

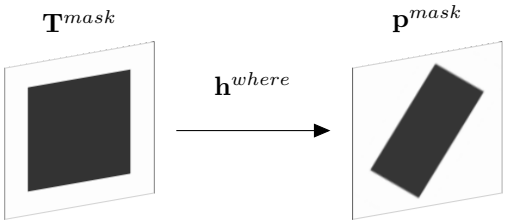


Figure 3.1: A square template mask $\mathbf{T}^{mask}$ is rescaled and rotated according to the affine parameters $\mathbf{h}^{where}$.

mask that is mostly binary (due to bilinear sampling used by the spatial transformer there is some blurring on the edges of the shape), which means that only the transformed template's area will be altered during the canvas overwriting, under any method.

3.2 The *what* of templates

While the above mechanism describes how to take a template and write it according to some affine parametrization Φ, it is not enough to fully describe the writing process: the appearance of the template also has to be defined.

The simplest appearance aspect (and the only one covered here) is color, which can be encoded by a normalized one/three-dimensional variable $h^{what} \in [0, 1]$ for grayscale/RGB images. Once again, h^{what} can be computed using an MLP with sigmoid activation. In order to obtain a colored patch from a transformed mask $\mathbf{p}^{mask}$ we just need to multiply the mask by the color, $\mathbf{p}^{color} = h^{what} \cdot \mathbf{p}^{mask}$, since this will set the non-zero values of the $\mathbf{p}^{mask}$ to the correct grayscale/RGB value.

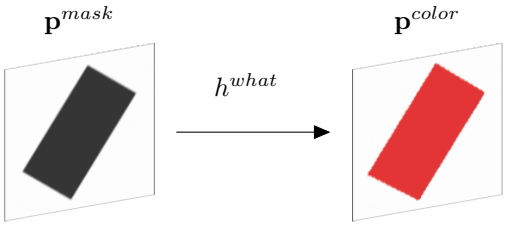


Figure 3.2: A transformed template mask $\mathbf{p}^{mask}$ is colored according to the value of h^{what}.

3.3 The *which* of templates

In the general case one wants to provide several templates that the agent can choose from. When only one template is given, there is no need for a template choice variable. However, if we have a collection of K templates (which we will refer to as the *template bank*, $\mathbf{T}^{bank} = \{\mathbf{T}_k\}_{k=1}^{K}$), it is necessary to use a discrete variable that represents the template choice. Below we discuss the challenges with using discrete variables and the alternative approach used in this work.

3.3.1 Making discrete variables differentiable

The use of discrete variables poses a problem to any system that aims to be fully differentiable and trained using end-to-end backpropagation. Although discrete variables are common in reinforcement learning and generative models (e.g. Sigmoid Belief Networks, Boltzmann Machines), they are inherently non-differentiable. In order to obtain a gradient signal in such models, methods like the REINFORCE algorithm [38], likelihood ratio gradients [37] or the wake-sleep algorithm [32] have to be used. However, since discrete variables are inherently non-differentiable, this prevents the gradients to be obtained directly by differentiating a loss function as happens with stochastic continuous variables [35, 36]. Furthermore, these methods suffer from some of the problems mentioned in Section 2.3.1.

While there are several approaches that try to solve this problem (see [63] for a recent overview), a simple yet efficient solution that suits the problem of choosing templates is the *weight-sharpening* method from [58]. In this method, instead of using purely discrete variables to choose templates, a linear (convex) combination of the K templates, $\sum_{k=1}^{K} w_k \mathbf{T}_k$, where $\sum_{k=1}^{K} w_k = 1$, is used (this can be considered a form of soft attention over templates). This combination is also a template, which means that the network can choose to paint a template that is e.g. $0.35\ \mathbf{T}_{circle} + 0.65\ \mathbf{T}_{square}$. This is possible to do because of the convenient fact that convex combinations of the discrete variables (templates) form valid options in this particular setting. This would not be possible in many other settings, for example, if the options were "Take a cab" and "Take a bus", since we cannot choose to "Take a 0.35 Cab + 0.65 Bus". The trick here is how to set the coefficients vector w such that it tends to a one-hot vector, hence a

single template choice. Weight sharpening works by using w as:

$$w_i = \frac{{w'}_i^{\gamma}}{\sum_j^K {w'}_j^{\gamma}} \tag{3.1}$$

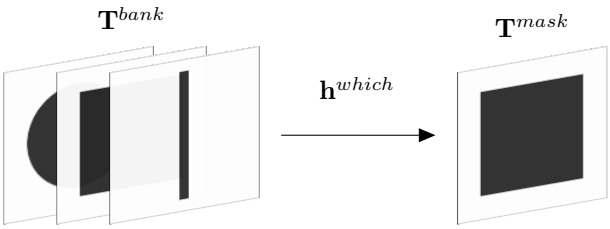


Figure 3.3: A square template is chosen from a template bank using the $\mathbf{h}^{which}$ variable.

In order to further encourage exploration, Gaussian noise can be added to the weights prior to sharpening. [64] suggests adding noise according to:

$$w_i = \frac{(w'_i + \mathcal{N}(0, \eta^2))^{\gamma}}{\sum_j^K {w'_j}^{\gamma}} \tag{3.2}$$

where the standard deviation η controls the extent of the exploration. However, this expression does not yield normalized weights. Since in practice the weights $\mathbf{w}'$ come from applying the softmax function to a set or unnormalized weights $\tilde{\mathbf{w}}'$, we add the Gaussian noise to these instead, in order to preserve normalization as well as positiveness. The final weights $\mathbf{w}$ will be represented by the variable $\mathbf{h}_{which} \in \mathbb{R}^K$, which may be computed as the output of an MLP followed by the noise addition, normalization and weight sharpening.

3.4 Occlusion and the canvas overwriting scheme

The final piece needed to perform realistic painting actions is conceiving a canvas overwriting scheme, or canvas update rule. As seen in Section 2, most of the models use an additive canvas, i.e., at each step (for sequential models) a hidden canvas $\mathbf{c}_t^h$ is updated using new patch $\mathbf{p}_t^{color}$ according to the additive formula:

$$\mathbf{c}_{t+1}^h = \mathbf{c}_t^h + \mathbf{p}_t^{color} \ , \ i \in \{1..T\} \tag{3.3}$$

Both the hidden canvas and the patch have unbounded range. In the end, the canvas image is created by taking the sigmoid of the hidden canvas, $\mathbf{c} = \sigma(\mathbf{c}_T^h)$, as in [2, 3], or by applying a further non-linear operation, $\mathbf{c} = \sigma(f_O(\mathbf{c}_T^h))$, as in [52]. Although this approach works successfully in the aforementioned models, it is not very realistic if we want to emulate the act of painting.

One of the main goals of this work is to provide the network with the ability to understand occlusion. When using an additive canvas with unconstrained writing actions, the model can

simply replicate visual input without having to understand the underlying geometrical causes of the scene. While a layered approach using the *over* operator was proposed in [46], the authors did not use it in any occlusion task: they did show results on overlapping MNIST digits but since the digits were the same color, no occlusion existed. In [3] the authors experiment with overlapping colored shapes, but once again occlusion is not present. In the same work, the authors experiment with a 3D rendering scene of solid objects where the network has to infer the identity of the objects present and their position/orientation in space given a 2D projection of the scene. However, since a 3D space is used, the objects can be placed interchangeably on the scene in order to produce the same result - this does not apply to painting, where a scene with occlusion has a well-defined ordering (for elements with overlapping locations).

Take as an example Fig. 3.4. If a human was given a rectangular shape and had to replicate this image in 2 moves, the correct procedure is immediately perceived: first draw the blue rectangle, then the red one. An architecture using an additive canvas on a constrained action space is (mathematically) unable to reproduce such an image, which can be visualized in Fig. 3.5 (top). This points to the fact that when humans reason about a scene, they tend to think of images as a superposition of shapes, not a sum.

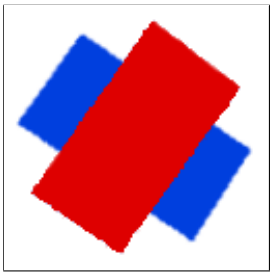


Figure 3.4: Simple image containing occlusion between two colored rectangles.

Be it in a physical canvas or software, painting is always superpositional, i.e., newer painting actions will always overwrite older ones (locally). With this in mind, a *superpositional canvas* is introduced. This type of canvas uses the following overwriting formula:

$$\mathbf{c}_{t+1} = \mathbf{c}_t \cdot (1 - \mathbf{p}_t^{mask}(\mathbf{T})) + h_t^{what} \cdot \mathbf{p}_t^{mask} \tag{3.4}$$

$$= \mathbf{c}_t \cdot (1 - \mathbf{p}_t^{mask}(\mathbf{T})) + \mathbf{p}_t^{color} \tag{3.5}$$

where $\mathbf{p}_t^{mask}(\mathbf{T})$ is the transformed template mask (binary) and $\mathbf{p}_t^{color}$ is the colored mask. This rule is equivalent to the *over* operation used in *alpha composition* in the field of computer graphics, where $\mathbf{p}^{mask}$ acts as the opacity component and $\mathbf{p}^{color}$ acts as the RGB components [65]. The difference between the additive and superpositional canvas is made clear in Fig. 3.5.

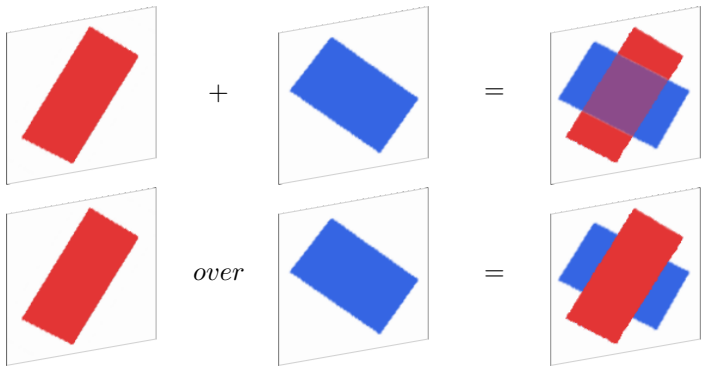


Figure 3.5: Difference between additive (top) and superpositional (bottom) canvas in their abilities to handle occlusion. Adding the blue and red rectangles results in a purple color in their intersection area. The *over* operator allows the red rectangle to be placed above the blue one, occluding it.

This type of canvas is used throughout the experiments that use templates, unless explicitly stated otherwise. Its advantages and shortcomings are discussed in later sections.

3.5 The *template writer* and inverse graphics

With the *what*, *where*, *which* and canvas overwriting scheme put together, we can visualize the final graphical model in Fig. 3.6, and the respective simplified pseudocode in Algorithm 1. There is a block comprised of the *what*, *where* and *which* components, which will be referred to as the *template writer*. This architecture has two main advantages: the modularity of the template writer allows it to be used as the writing mechanism in any existing network that contains a write block; the input to the writer may be stochastic or deterministic, allowing for the construction of various kinds of models, as we will see in later sections. Although in the figure $\mathbf{h}^{which}$, $\mathbf{h}^{where}$ and h^{what} are computed from a shared h (usually via MLPs), this need not be the case.

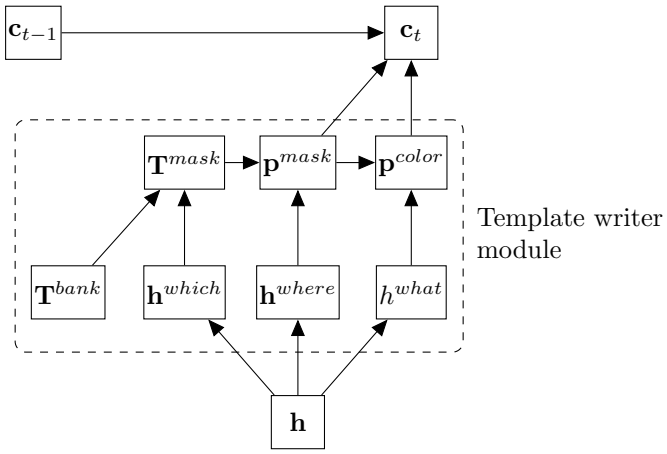


Figure 3.6: Template writing mechanism.

By this point it has become evident that this problem is related not only to unsupervised

Algorithm 1 Template writer algorithm

```
function WriteTemplate(h_where, h_which, h_what, template_bank, current_canvas)
    T_mask ← choose_template(h_which, template_bank)
    p_mask ← transform_mask(h_where, T_mask)
    p_color ← colorize_mask(h_what, p_mask)
    new_canvas ← update_canvas(p_mask, p_color, current_canvas)
    return new_canvas
```

learning of images in general but to computer graphics as well. In fact, the network described above can be seen as an instance of *autoencoders with domain-specific decoders* [45] in a *vision as inverse graphics* context [6, 64]. The template writer module is a differentiable encoder-decoder graphics network: the encoder is the part that computes the components h^{what}, $\mathbf{h}^{where}$ and $\mathbf{h}^{which}$, which are the graphics codes, and the decoder is comprised of the remaining parts that pick the template according to $\mathbf{h}^{which}$, transform it according to $\mathbf{h}^{where}$ and color it according to h^{what}, such that it becomes ready to be placed onto the canvas.

Chapter 4

Sequential variational models: failure case

Considering the success of DRAW and AIR, both in terms of image reconstruction and generation, it is only natural to explore their suitability for the task at hand. The versions of these models using templates will be referred to as T-DRAW and T-AIR, respectively. If successful these models would allow us to obtain not only a reconstruction of the images given using templates, but also generate new images, due to its variational nature. Unfortunately, they proved to be unsuitable for the task of creating a painter network, as will be discussed in this chapter.

The two models are tested on two datasets: MNIST [66], which is the baseline for any unsupervised image learning task, and a custom n-Rectangles dataset. This n-Rectangles dataset is composed of images that contain n rectangles placed at random locations in the image. Many variations can be included, such as the use of rectangles with varying grayscale values (for occlusion experiments) or varying deformation. See Appendix A for samples from this dataset. The importance of this dataset will become clear during the rest of this work, as it proves to be much more challenging than it seems.

4.1 T-DRAW: DRAW with templates

Since the template writer was constructed in a way such that it is completely modular, obtaining T-DRAW from DRAW is straightforward: all that is needed is to replace the default *write* operation (see Fig. 2.5) with the template writer in Fig. 3.6, where $\mathbf{h}$ in the figure corresponds to the output $\mathbf{h}_{dec}$ of the decoder RNN in DRAW (Fig. 2.5). The canvas overwriting scheme must also be changed accordingly, from additive to superpositional. Results for MNIST and the 2-Rectangles dataset are shown below.

MNIST

In order to test the basic ability of T-DRAW to reconstruct MNIST digits using painting actions, we use a simple setting where only a circle template (see Appendix A) is provided (so there is no *which* component), and the template writer just has to choose the transformation to be applied (*where*) and the grayscale color to be used (*what*). This circle template provides the ability to generate both large circular strokes and long strokes with curved edges, which are enough to paint any digit in this dataset.

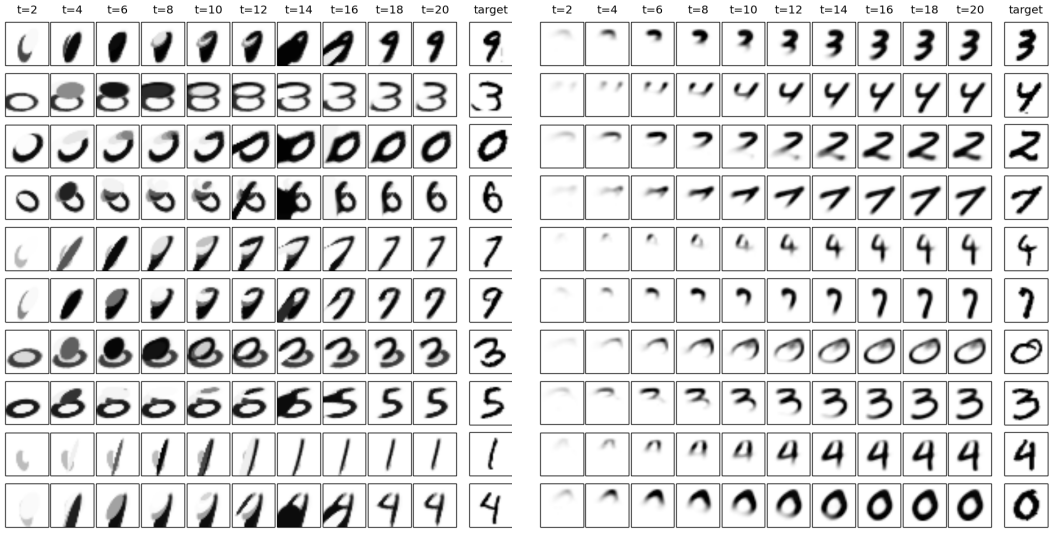


Figure 4.1: Painting process for the 20-move T-DRAW (left) and DRAW (right) models, trained on the MNIST dataset.

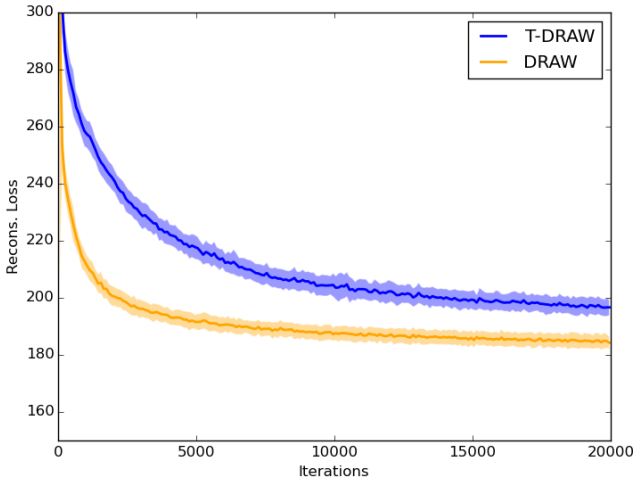


Figure 4.2: Negative lower bound of training set for T-DRAW and DRAW, trained on the MNIST dataset.

We run the model for 20000 batch iterations using 20 moves and the result can be seen in Fig. 4.1 (left). Please refer to Section 1.3 for full experimentation details. This image shows

the current canvas progress during the 20-move painting process - the final canvas is on the $t = 20$ column, and the image given to the network as target/input is seen on the last column. This type of plot will be used extensively throughout this work, since it not only allows us to inspect the final canvas but also to visualize the painting process. On Fig. 4.1 (right) is the equivalent drawing process for a DRAW network[1] trained for the same number of iterations and the same number of moves. It can be seen (Fig. 4.2) that although the lower bound obtained by T-DRAW is not as good as DRAW's, the reconstructions using templates are still quite good. However, looking at the painting process, it seems like the moves are quite erratic, with large regions being painted only to be erased (i.e., painted in white) by later moves.

The T-DRAW model seems to be able to handle this dataset with relative ease. Problems start arising in next dataset.

2-Rectangles

After successfully being able to model MNIST digits, we move on to trying to solve the basic task of identifying and painting two black rectangles in a small white-background image, using a 2-move network. The results are shown in Fig. 4.3.

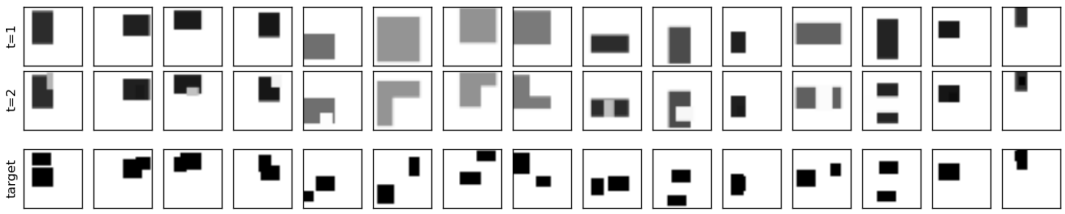


Figure 4.3: Painting process for 2-move T-DRAW model trained on the 2-Rectangles dataset.

As we can see, T-DRAW is not able to solve this task. It is possible to notice that while some reconstructions are approximately correct, the painting process is not: in all images, the network paints a large rectangle covering both rectangles in the first move (t=1) and then tries to correct the white spaces using the second move (t=2), with disappointing results. Many parameter settings were tested, always with the same outcome. This behavior shows that T-DRAW permanently gets stuck in a local minimum: after the network has used the first move to place a rectangle over the two target rectangles, any deviation incurs an increase of reconstruction loss, so the network cannot escape the minimum. It is not clear, though, the reason for this behavior, considering T-DRAW was able to 'solve' MNIST.

The generative performance of T-DRAW was not tested here since appropriate reconstruction was not achieved. Without being able to correctly reconstruct the training images, it does not make sense to look at the generative capabilities.

[1] We used spatial transformers as the reading and writing attention instead of the original Gaussian attention in order to provide more accurate comparisons.

4.2 T-AIR: AIR with templates

Since T-DRAW is not able to find the solution for the 2-Rectangles dataset, we decide to use the template adaptation of AIR, T-AIR, due to its explicit modeling of individual image components. Given that the use cases of the AIR model in the original paper ([3]) were the most similar to the ones being tested here, it seems like a good architecture to try to tackle this issue.

In order to construct T-AIR from AIR, we use the $\mathbf{z}^{what}$ variable as a stochastic h^{what} in the template writer, and $\mathbf{z}_t^{where}$ corresponds to the stochastic $\mathbf{h}_t^{where}$. This is different from T-DRAW, since here the writing parameters are being used as the latent variables, while in T-DRAW the writing parameters were deterministically computed as part of the decoder.

Using the same experimental settings as those used for T-DRAW, we run AIR on the 2-Rectangles dataset, with disappointing results. From Fig. 4.4 we can see that AIR learned a strategy similar to that learned by T-DRAW: fill the area of both rectangles in the first move and try to make up for it in the 2nd move.

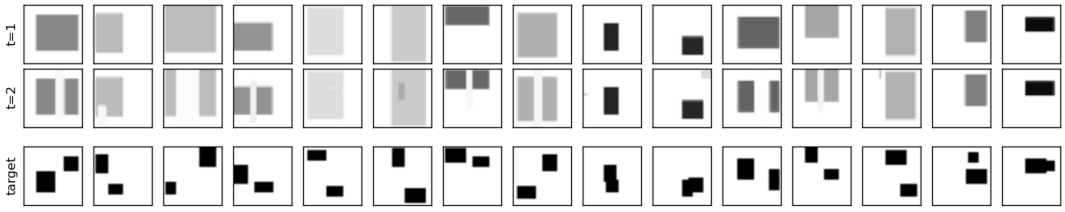


Figure 4.4: Painting process for 2-move T-AIR model trained on the 2-Rectangles dataset.

In fact, unlike T-DRAW, T-AIR was not even able to learn the MNIST datasets, with some results shown in Fig. 4.5. This shows that T-AIR is definitely not an appropriate architecture to create a painting network. But what are the reasons for this, considering that AIR was successful in tasks similar to these?

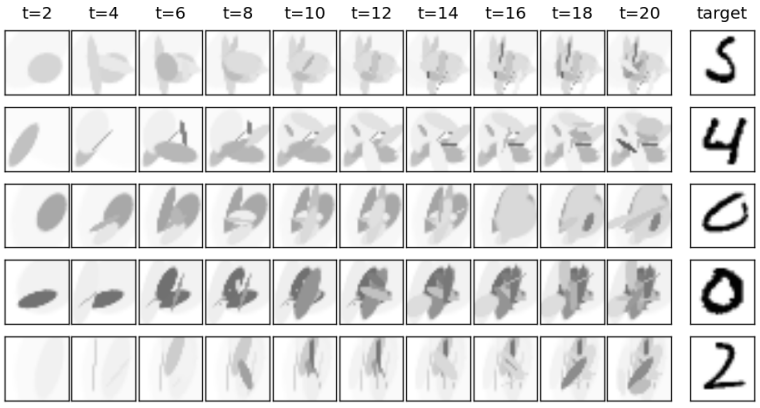


Figure 4.5: Painting process for 2-move T-AIR model trained on the MNIST dataset.

Looking more carefully at the AIR architecture, one clear limitation is that the writing has

to be performed in the same location as the reading. This works in the standard AIR setting where the autoencoding mechanism is an unconstrained variational autoencoder. However, given a location, the only sources of variability for templates are the template choices and the color. This means that if the reading location is not perfectly aligned with a template-like shape on the scene, the writer will fail to provide an accurate reconstruction of the patch read. We can also conclude that using the writers' outputs as the latent variables is not ideal since they will be subjected to the sampling process, which can make the forward passes very variable and the training very unstable.

Another major issue with this architecture is that the reading attention windows have to be constrained to be roughly the size of the elements of the image, otherwise the network will not learn to represent the individual elements on its own. T-AIR was able to identify the individual rectangles in the 2-Rectangles dataset when we set the reading attention windows to be about the size of the rectangles (in fact, the authors of AIR used the same trick to obtain the results reported in the original paper), but a good model should not have to be subject to such strong constrains in order to learn appropriately.

4.3 Why more complex $\neq$ harder

It is not immediately obvious why the templated counterpart of a powerful model like DRAW is not able to solve the 2-Rectangles dataset, which is almost the simplest one can conceive in this context, but is able to solve MNIST rather easily, considering the former uses two moves and the latter can use tens of moves. Given the simplicity of 2 rectangles in an image compared to that of handwritten digits, *a priori* one would think that the systems above would have a harder time dealing with MNIST than with 2-Rectangles. However, in this case, as experience shows, this complexity perceived by humans does not correlate directly with the complexity perceived by the models. It is usually not a good sign when AI's behavior differs so much from human behavior.

If we are to obtain a human-like painting model, it is important that both tasks are handled with ease, and preferably that 2-Rectangles is easier than MNIST, as it should. In order to do this, it is paramount to understand the reasons for such behavior in T-DRAW and T-AIR. While the reasons will not become fully clear until a more in-depth analysis is done in Section 5.3, it is possible to visualize the data available at this point to get a glimpse into the reasons.

From the learning process in Fig. 4.6 it is possible to see that besides the fact that learning is very slow, the digits are learned by a nearly continuous deformation of the previous iteration. Furthermore, it is possible to see that not all the moves are being updated, only some of them, and only after these last are sufficiently correct do the remaining moves get updated. This is a problem because it means that in cases where the structure of the image is not continuous there is just a couple of moves that are receiving all the gradient signal. This causes the behavior

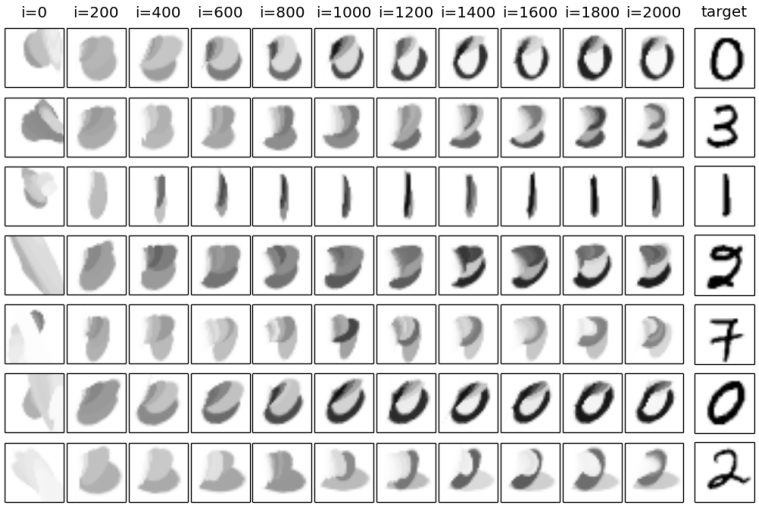


Figure 4.6: Learning progress of T-DRAW on the MNIST dataset in the first 2000 iterations, using the circle template

observed in Fig. 4.7, where one rectangle tries to cover both rectangles, while the other one remains idle. Ideally, both moves should learn at approximately the same rate. Although the templates' opacity, which kills the gradient passing through its mask region, is a factor contributing to this issue because it prevents gradient signal from reaching earlier moves, it is not the most important factor. In fact, the learning process by continuous deformation might hint to the recurrent nature of the models as being a problem, since RNNs are known to optimize slowly and not always stably through time, but this is not actually the case here. As it will be seen in depth in Section 5.3, a recurrent structure can solve the 2-Rectangles problem, and the main problem lies in the variational objective.

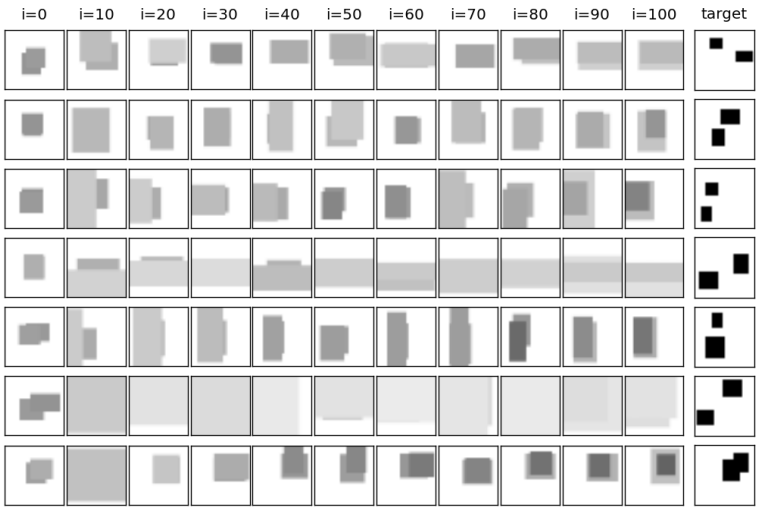


Figure 4.7: Training progress for 2-move T-DRAW model trained on the 2-Rectangles dataset in the first 200 iterations, using the square template

Chapter 5

The *Painter* network: success case

After the unexpected failure (and its extent) of the T-DRAW and T-AIR to solve the simple 2-Rectangle dataset, it was necessary to abandon the complex sequential variational models in order to find a model that would be able to reconstruct a large variety of image types, not just those whose structures can be obtained through nearly continuous movements.

5.1 A simple architecture

Instead of using the sequential variational models mentioned, we should go back-to-basics and ask "What is the simplest type of autoencoder architecture that one can use for this task?". The answer here is an autoencoder containing T writers that are computed at once, each of which is responsible for a move, without recurrence or variational objective, with the reconstruction cost being the MSE of the image reconstruction. It turns out that this very simple architecture achieves a surprising performance in the datasets previously mentioned and many others that are discussed throughout this chapter.

The structure proposed is indeed very simple, and is depicted in Fig. 5.1 (left). As we can see, each move $t \in \{1..T\}$ is picked by a separate template writer (see Fig. 3.6), writer_t, and the painting actions are taken sequentially once the moves have been computed. It is worth noting a few points about this architecture. Firstly, there is no temporal dependency between writers, i.e., writer_2 does not depend neither on the current canvas $\mathbf{c}_1$ nor any parameter of the previous writer, writer_1, which means that the different writers' moves are all computed at once. In fact, there need not be any shared lower level features h, in which case all the moves are completely independent. Furthermore, not only can the moves be independent, but the components within each move can be independent as well, with small loss of performance. Secondly, the move sequence is always the same: writer_1 always corresponds to the first move,

writer_2 to the second move, etc. This architecture will be referred to as *simple Painter*, and it will form the basis to all the remaining architectures developed. It is interesting to note that its structure is quite similar to that of capsule-based autoencoders (cf. [44]).

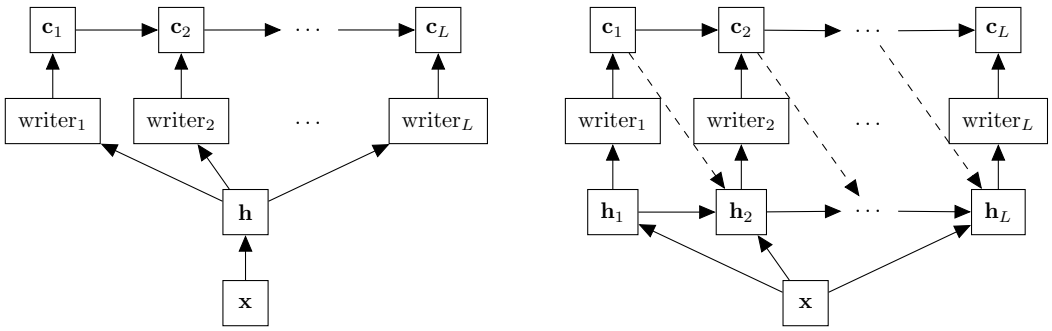


Figure 5.1: The simple (left), recurrent (right, no dashed connections) and canvas-recurrent (right, dashed connections) Painter network architectures

From the simple Painter network we can easily construct its recurrent and stochastic variations. Since T-DRAW and T-AIR are recurrent and variational, if the simple Painter network works (which will be shown to be the case), then the problem with the former models must lie in the recurrence, the variational objective and stochasticity, or both.

On Fig. 5.1 (right) is a recurrent version that will be considered in order to investigate the influence that a recurrent structure might have had in the poor performance of the aforementioned models. When excluding the dashed connections, we have a simple RNN (referred to as *recurrent Painter*) on the feature extraction part, and when including the dashed connections the recurrent state $\mathbf{h}_t$ depends not only on the previous state $\mathbf{h}_{t-1}$ but also on the canvas drawn so far, $\mathbf{c}_{t-1}$ (referred to as *canvas-recurrent Painter*). For all these cases, the error function to minimize during training is the mean squared reconstruction error of the final canvas, $\|\mathbf{x}-\mathbf{c}_T\|^2$.

It is straightforward to convert these deterministic (recurrent and non-recurrent) models into variational or denoising ones: in the simple Painter by turning the deterministic hidden variable $\mathbf{h}$ into stochastic variable $\mathbf{z}$, and in the recurrent Painters by turning the deterministic hidden variable at each step $\mathbf{h}_t$ into stochastic features $\mathbf{z}_t$. If the error objective is still the reconstruction MSE and the stochastic variables apply noise at fixed σ we have a denoising autoencoder (the noise can be applied at the hidden variable, at the input, or both); if the error objective is the negative variational lower bound from (2.2) and $\mathbf{z}$ is a Gaussian stochastic variable we have a model analogous to a VAE (or DRAW, if recurrent). As we have seen in Section 4.2, it is not a good idea to have stochasticity in the *where*/*what* variables themselves, so we will not consider an AIR-like case here.

5.2 Escaping the local minimum

As previously mentioned, if a model is not able to solve both MNIST and the 2-Rectangles dataset it will not go very far in the context of painting with templates. Here we show the performance of the Painter architectures of Fig. 5.1 in these datasets and variations thereof. The following experiments were ran for 10000 training batch iterations using the square template, with the spatial transformer's affine parameters $\Phi_{12} = \Phi_{21} = 0$, in order to prevent skewing of the templates. This constraint is more for ease of visualization, since the same conclusions are reached if these two parameters were unconstrained. The layers used in the networks were:

- simple Painter: architecture like 5.1 (left), with $\mathbf{h}$ being a 256-dimensional layer with ReLU non-linearity[1] and the writers' outputs being obtained by a further layer with the respective dimensions and non-linearities.
- recurrent and canvas-recurrent Painter: architecture like 5.1 (right), with 256 LSTM units ($\mathbf{h}_t$) layer and the writers' outputs being obtained by a further layer with the respective dimensions and non-linearities.

The networks were found to work best by randomly initializing the weights using a Gaussian with mean 0 and standard deviation 0.05.

Starting with the 2-Rectangles dataset, which was the problematic one in the previous chapter, in Fig. 5.2 it is possible to see that all the Painter networks correctly identify both of the rectangles present without getting stuck in the local minimum observed before. The realization that they work is one of the most important milestones in this work, since it hints towards the fact that the problem proposed is attainable and that the root of the problems with T-DRAW and T-AIR may have to do with the variational nature of the models. This will be discussed in later sections.

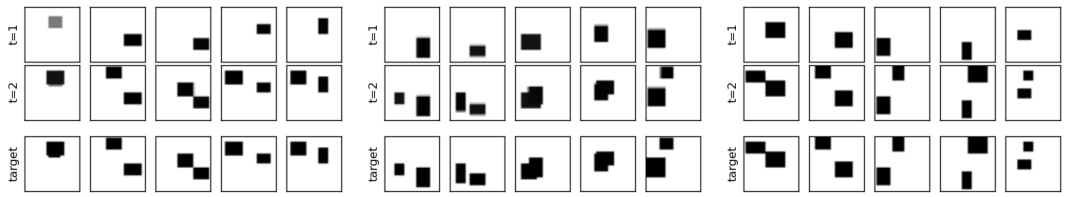
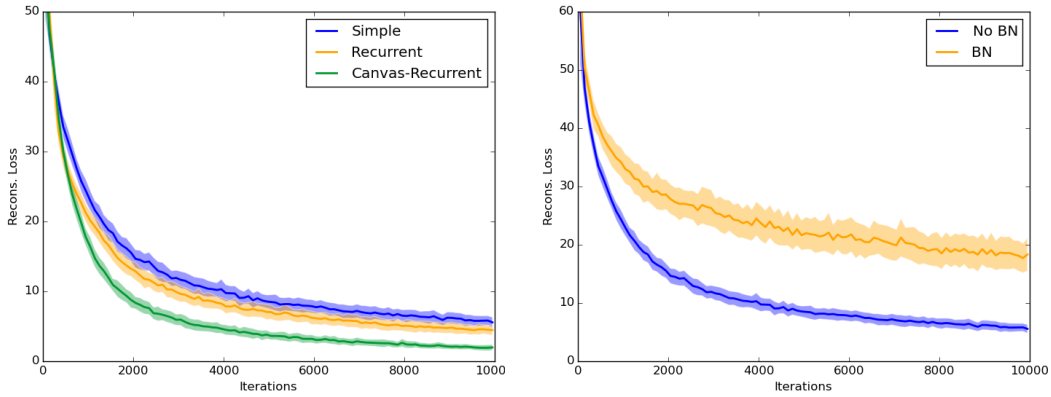


Figure 5.2: Painting process in a 2-move simple (left), recurrent (middle) and canvas-recurrent (right) Painter network, trained on the 2-Rectangles dataset.

In Fig. 5.3a we can see the comparative learning speed of these three architectures: the recurrent performs slightly better than the simple Painter and the canvas-recurrent performs slightly better than the recurrent Painter. The use of batch normalization [67] after the hidden layer was tested for the simple Painter in the 2-Rectangles dataset (5.3b), to check if it would play an important role in escaping the local minimum, but it was found not to improve the

[1] $\text{ReLU}(x) = max(0, x)$.

learning performance (in fact, learning becomes significantly slower, but the correct strategy is found nonetheless).



(a) Comparison between a simple, recurrent and canvas-recurrent Painter network on the 2-Rectangles dataset.

(b) Comparison between a simple Painter network with and without batch normalization on the 2-Rectangles dataset.

Figure 5.3: Experiments involving the Painter networks.

At this point it is important to emphasize that the goal of this chapter is not so much to try to squeeze the maximum performance through fine-tuning, but rather to obtain an architecture that is able to tackle a wide variety of datasets without getting stuck in the prevalent local minima found before and test the conditions under which particular architectures fail. This is the main reason why extensive architecture comparison is not made here: optimization is always possible, and the most important difference in this case is not whether a model performs marginally better than the other, but if it solves the task at hand at all. Only comparison between a standard feedforward network (simple Painter), a basic recurrent network (recurrent Painter) and a more complex recurrent network (canvas-recurrent Painter) was done since they represent all the cases of interest that can not only solve the tasks but also provide insight into the causes of failure of T-DRAW and T-AIR.

Although these three architectures perform somewhat similarly in a simple setting like the 2-Rectangles dataset, the recurrent versions tend to get stuck in poor local minima when scaling to more complex datasets, which is shown in Section 5.7. For this reason, the remainder of the experiments in the next sections will be made using the simple Painter network[2], unless stated otherwise.

Before moving on to testing the Painter network on the MNIST dataset, we test it in a couple more interesting variations of the n-Rectangles setting, in order to make sure that the 2-Rectangles dataset is not just an isolated case that the network can solve. The first is the 4-Rectangles dataset and the second is the 2-Rectangles-Mixed dataset (see Appendix A), which is a version of 2-Rectangles containing a mix of black rectangles on white background and vice

[2]The name *Painter network* will be used to refer to the simple Painter network, for ease of notation.

versa. These two tasks are important to further assess the capabilities of the Painter network.

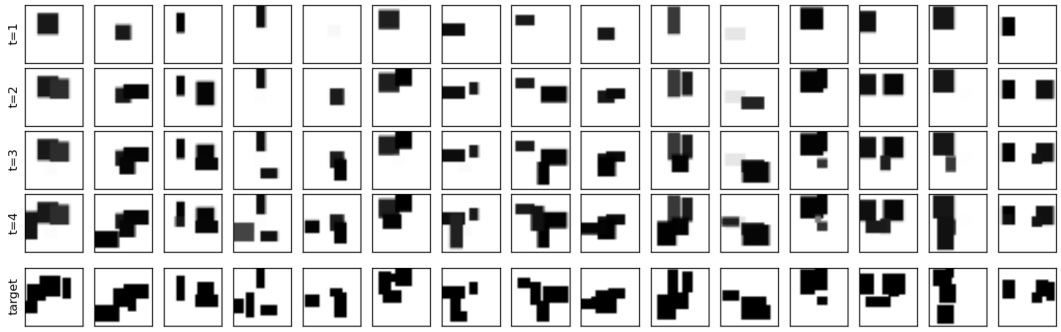


Figure 5.4: Painting process in a 4-move Painter network, trained on the 4-Rectangles dataset.

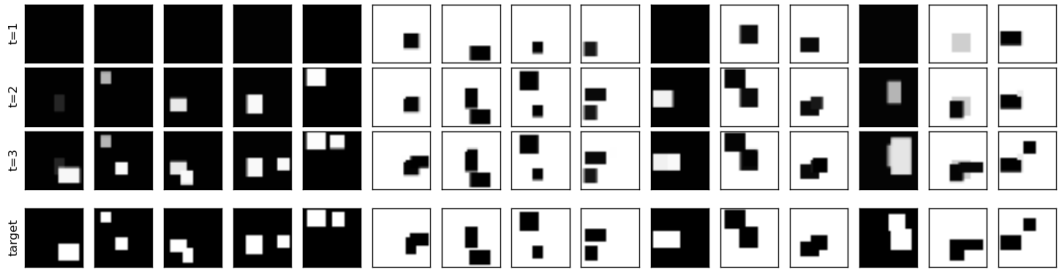


Figure 5.5: Painting process in a 3-move Painter network, trained on the 2-Rectangles-Mixed dataset.

In Fig. 5.4 is the result on the 4-Rectangle dataset using a 4-move network. While the network is able to roughly work out the location of the rectangles, it has some trouble correctly perceiving the smaller details, which causes some moves to not be used to the best or their ability. This hints to a behavior that is explored further in later sections: the Painter has trouble capturing and reproducing fine details in the pictures.

In Fig. 5.5 is the result on the 2-Rectangle dataset using a 3-move network. A 3-move network was used since a white-on-black image requires at least three moves to be correctly drawn - one move to paint the background and two moves to paint the white rectangles. From the Figure we can see that the network does find the optimal painting strategy for this situation: for white-on-black images, it learns to fully paint the canvas black in the first move and use the two following moves to paint the white rectangles. For the black-on-white images it is equivalent to the simple 2-Rectangles case but it learns to neglect one of the moves. The results of both experiments are encouraging in terms of this being an appropriate Painter architecture.

Regarding MNIST, it is possible to see in Fig. 5.6 that with the same 20 moves that were used in T-DRAW, the reconstructions are equivalent, but with two important differences: the drawing process is much less erratic (looks more human-like) and the Painter achieves visually plausible samples after just a few hundred iterations, while with T-DRAW these take several thousand iterations.

Figure 5.6: Painting process in a 20-move Painter network, trained on the MNIST dataset.

5.3 Why variational models are not suitable for a painter

After testing the recurrent Painters and concluding that they can solve the 2-Rectangles dataset, we test the variational version of the Painter network. Using the usual Gaussian stochastic variables with variational objective (cross-entropy reconstruction cost instead of MSE) we get the results in Fig. 5.8. As with T-DRAW and T-AIR, the network gets stuck in the same local minimum. This means that the problems in sequential variational models are mostly caused by their variational nature. We now have to investigate further why this is so. Is it the CE reconstruction loss? Is it the latent cost? Is it the standard deviation of the prior? Is it the stochasticity of the latent variables?

In order to answer these questions we run a series of experiments, involving several variations relative to the variational objective (2.2): using fixed $\boldsymbol{\sigma}$ (latent cost only contains the term in $\boldsymbol{\mu}$), using deterministic $\mathbf{z}$ (instead of stochastic) without any latent cost, or using deterministic $\mathbf{z}$ while applying the latent cost in $\boldsymbol{\mu}$ (for deterministic $\mathbf{z}$, $\mathbf{z} = \boldsymbol{\mu}$). All of these experiments can further be done for MSE loss instead of CE. Not all of these variations are mathematically correct, but they allow us to test each and every component of the variational framework that may cause the convergence problems. Using a deterministic $\mathbf{z}$ with CE rather than MSE loss allows us to see if the former causes bad gradients to be propagated due to the large contribution of incorrect locations in log terms. Using a stochastic $\mathbf{z}$ with latent cost in $\boldsymbol{\mu}$ but fixed $\boldsymbol{\sigma}$ will allow us to see if the latent cost term in σ is responsible for the divergence and to see the

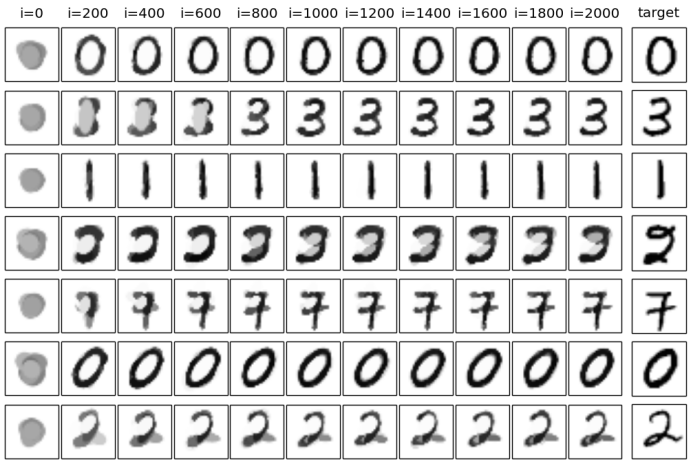


Figure 5.7: Training progress in a 20-move Painter network, trained on the MNIST dataset.

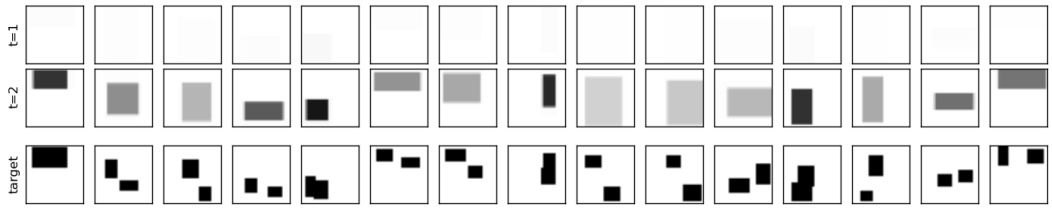


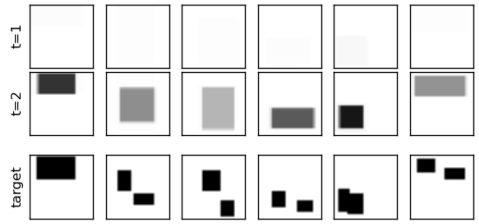
Figure 5.8: Painting process in a 20-move variational Painter network, trained on the 2-Rectangles dataset.

influence of the term $\boldsymbol{\mu}^2$, which is just a form of L2 regularization. Using stochastic $\mathbf{z}$ without any latent cost (with fixed $\boldsymbol{\sigma}$) will allow us to see if the network is able to perform denoising.

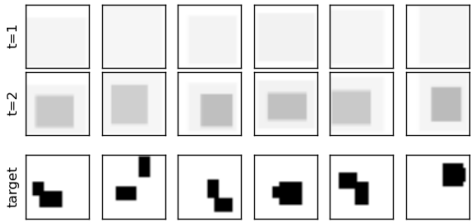
The results of several experiments are shown in Fig. 5.9. As we can see, only 3 of the 10 settings tested were able to not get stuck and successfully solve the 2-Rectangles task: deterministic networks with no latent cost (for both MSE and CE reconstruction cost; Figs. 5.9j and 5.9e, respectively) and the stochastic network with no latent cost (for MSE reconstruction cost only; Fig. 5.9i). The remaining failed with varying degrees of failure. The important realization here is that whenever a latent cost term was included (either on $\boldsymbol{\mu}$ only or on $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$), the network got stuck in some non-optimal strategy, from which it could not leave. It is not immediately clear why this would be the case.

The intuition here seems to be that the gradients of the latent cost tend to dominate over the reconstruction cost, hence these have a much stronger effect on the learning process. This causes the network's parameters to move to some region in parameter space where it becomes unable to solve the reconstruction task. We tried multiplying the latent cost by some small constant (like done in usual regularizations), in order to try to reduce the dominance of the latent term. Unfortunately, this proved unsuccessful, as low values resulted in the latent term being neglected and high values resulted in the reconstruction term being neglected. No equilibrium value was found. For the lack of a better explanation, we have to leave this question unanswered and we

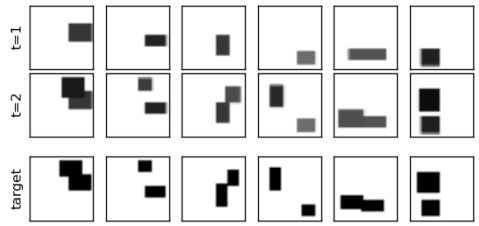
proceed with the realization that variational autoencoders may not be suitable for the painting task.



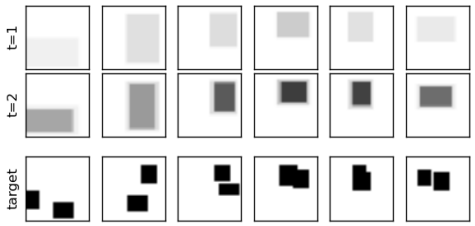
(a) Stochastic; latent cost in $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$; CE reconst. loss.



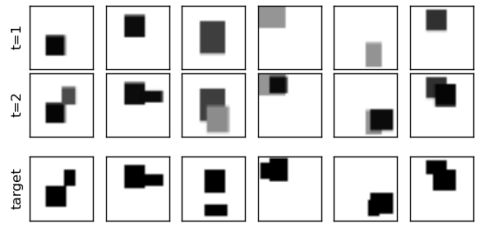
(b) Stochastic; latent cost in $\boldsymbol{\mu}$; fixed $\boldsymbol{\sigma} = 0.3$; CE reconst. loss.



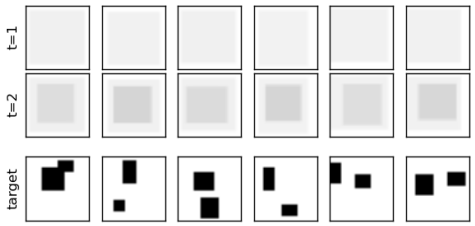
(c) Stochastic; no latent cost; fixed $\boldsymbol{\sigma} = 0.3$; CE reconst. loss.



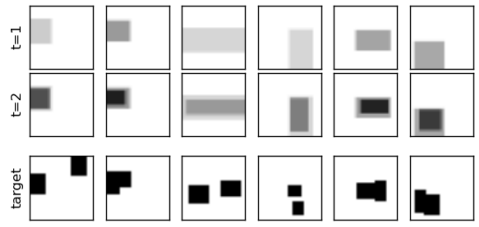
(d) Deterministic; latent cost in $\boldsymbol{\mu}$; CE reconst. loss.



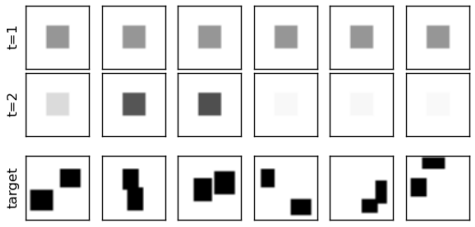
(e) Deterministic; no latent cost; CE reconst. loss.



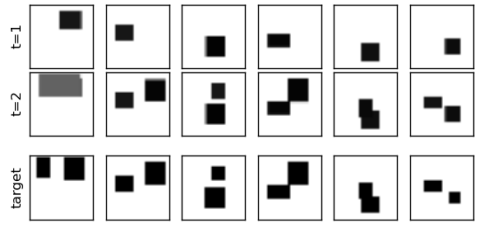
(f) Stochastic; latent cost on $\boldsymbol{\mu}$; fixed $\boldsymbol{\sigma} = 0.3$; MSE reconst. loss.



(g) Stochastic; latent cost on $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$; MSE reconst. loss.



(h) Deterministic; latent cost on $\boldsymbol{\mu}$; MSE reconst. loss.



(i) Stochastic; no latent cost; fixed $\boldsymbol{\sigma} = 0.3$; MSE reconst. loss.

(j) Deterministic; MSE reconst. loss.

Figure 5.9: Experiments using several variations of the variational and simple Painter networks.

5.4 The problem of varying length

One important aspect of the AIR model is the introduction of variable sequence length in generative models. By using no variable length in the Painter network we are forfeiting the ability to use this feature. However, it is interesting to see how a fixed network architecture deals with images that have a varying number of elements, say, rectangles. To test this, a variation of the 2-Rectangles dataset is used, which we call 0to2-Rectangles: a dataset where each image randomly contains between 0 and 2 rectangles. The resulting move sequences can be seen in Fig. 5.10.

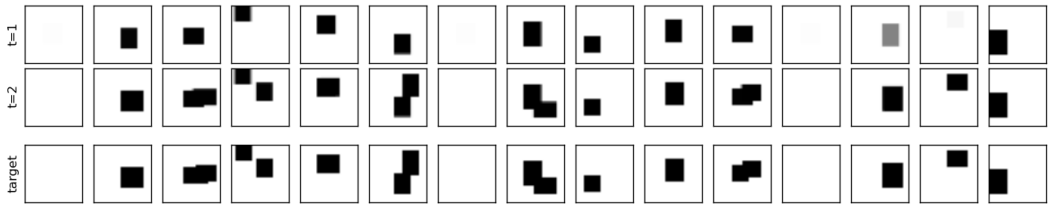


Figure 5.10: Painting process in a 2-move Painter network, trained on the 0to2-Rectangles dataset.

We can see that the network is able to cope with this variability quite well. By inspecting the numerical output data and the initial learning process (Fig. 5.11), it is possible to understand the way it does so: in images with less than two rectangles, it simulates a null move by painting a white rectangle over the white background (it is possible to notice this in some moves where the color was not fully set to white) or drawing a rectangle that will then be overwritten.

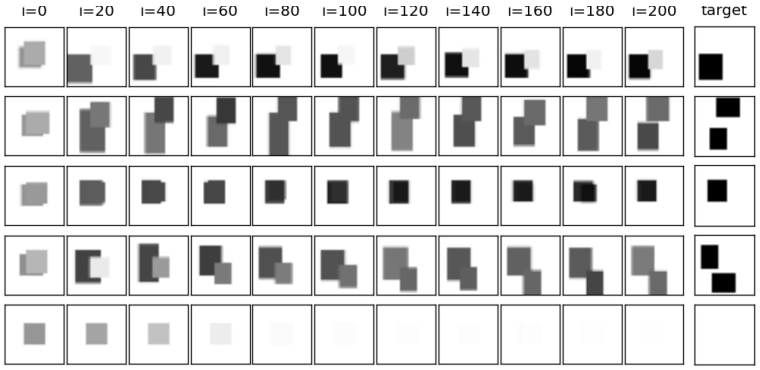


Figure 5.11: Training progress of the Painter network in the 0to2-Rectangles dataset in the first 200 iterations.

It is clear that the network still identifies individual rectangles correctly, instead of, for example, in the images with one rectangle, identifying this rectangle as two contiguous rectangles, hence needing to do two moves. This is a sign that, in the context of a painter, variable sequence length may not be necessary. However, there is an important downside to this approach: if we were to scale this model to images that can contain anywhere from a handful to dozens of moves, the network must always process the maximum number of moves, even though in many

cases most of them will be rendered useless.

Another way that could be used to simulate null moves would be to use a transparent template, i.e., a template that when picked would always leave the image unchanged. As we will see in Section 5.5, when this template is allowed, the network usually ends up using the other template (like rectangle) in a redundant way, due to the numerical properties of the implementation of multiple template choice.

5.5 Multiple templates

In the previous sections we have seen how the Painter network is able to solve tasks with varying degrees of difficulty. However, on all of those tasks we used only a single template, so the *which* component was not part of the computation. As noted in Section 3.3, in order to use multiple templates while keeping the network end-to-end differentiable it is necessary to use some workaround, which must be properly tested before it can be claimed to be an appropriate one. In this case, the workaround used was weight-sharpening, which turns discrete actions into continuous actions that are discrete in the limit of the sharpening factor $\gamma \to \infty$. In this section this mechanism will be tested in a set of basic but representative datasets.

The first case will be with the 2-Shapes dataset, a multi-shape equivalent of 2-Rectangles: instead of each image having two rectangles, it has two different possible shapes. Rectangles and circles will be used, so each image can have two rectangles, one rectangle and one circle, or two circles.

The Painter network was trained for 20000 iterations using $\gamma = 1$ throughout, i.e., no sharpening. At test time we used $\gamma = 30$ since this value was found to be enough to make the moves visually discrete. While the `argmax` could be used to provide completely discrete actions, using large γ provides enough discretization in most cases and allows us to observe the cases in which the template choice is so close to uniform that even with this sharpening factor the moves are still a blend of shapes. The results are in Fig. 5.12. It is possible to see that in most cases the shape is correctly chosen, which means that the template combination coefficients leaned towards the correct shape, prior to the sharpening at test time. The accuracy was found to be 95.2% on the training set and it was computed in the following way: an example was considered "correct" if a move was chosen within a 4 pixel radius of a move made with the same template (we have this information because the dataset is artificially created), i.e., the largest weight w'_i corresponded to the correct template at approximately the correct location.

Though encouraging, this is a very simple example, since the squares and circles here have only isomorphic scale deformation, no rotation or anisomorphic deformations, and as such the square and circle shapes are already very similar to each other.

Since no γ annealing was used here, it is interesting to see how different annealing rates would influence the training speed. We also use this opportunity to test two additional settings

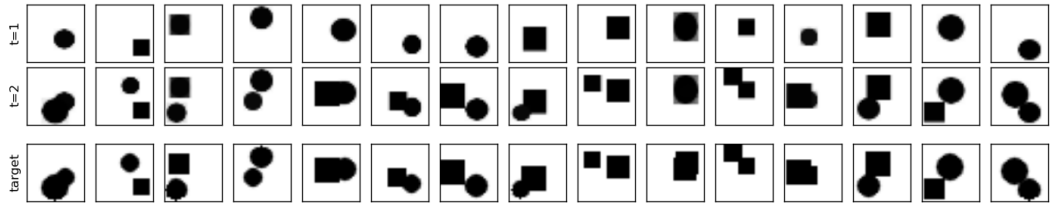


Figure 5.12: Painting process in a 2-move Painter network, trained on the 2-Shapes dataset.

simultaneously: the suggestion made in the end of Section 5.4 of using a transparency template to simulate null moves, and applying this multiple template mechanism to a dataset that has complex structures and is not composed exactly by the templates' shapes, like the n-Shapes datasets. For these experiments, we use the Omniglot dataset [68], since it contains a wide variety of alphabet characters with intricate patterns (see Appendix A). The templates used were the circle, line, curve and transparency templates (see Appendix A), and a 10-move Painter network was used.

Regarding the use of several γ annealing rates, the training errors for the 2-Shapes and Omniglot datasets can be seen in 5.13. The annealing rates used were 0.0 (equivalent to no annealing), 0.005, 0.01, 0.05 and 0.1 per iteration. For both cases the sharpening factor alone does not seem to provide any significant improvement to the training performance on either of the datasets.

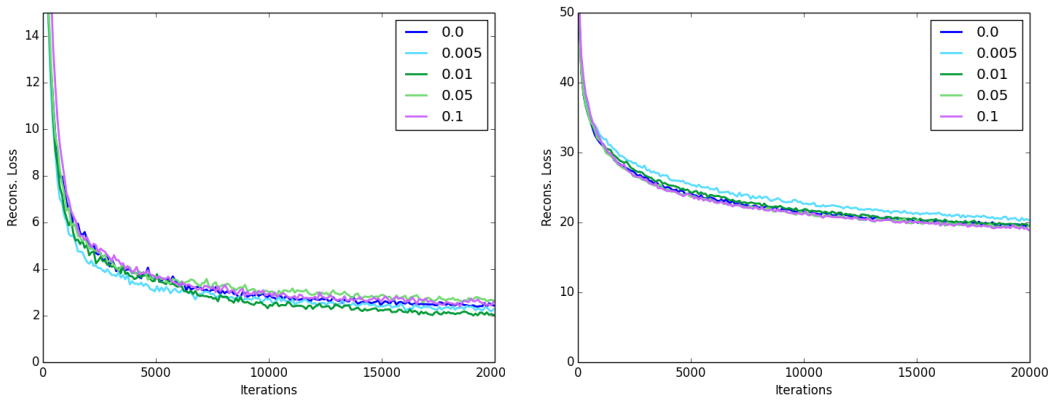


Figure 5.13: Comparison of training cost for varying γ annealing rates in a Painter network, trained on the 2-Shapes (left) and Omniglot (right) datasets, using 2 and 10 moves, respectively. The standard deviation area around the smoothed plot lines were omitted to prevent excessive cluttering.

Examining eq. (3.1), it is possible to understand why the sharpening factor would not make any difference in the learning process. For any value of γ, we can find equivalent weights with values $w_i'^{\gamma}$ that produce the same result. For this reason, the network can always adjust the magnitude of the weights $w_i'^{\gamma}$ if it wants to keep a soft combination of templates, meaning that the γ values do not actually enforce the network to strongly pick one of the templates.

Unfortunately, adding noise to the pre-sharpened weights does not yield any improvement either, as can be seen in 5.14. In fact, larger noise values cause learning to be significantly slower.

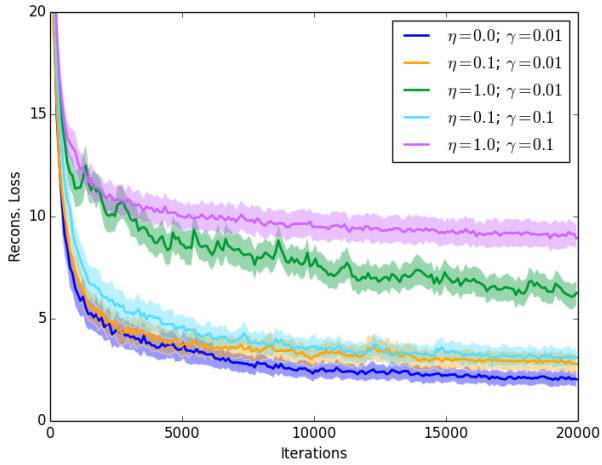


Figure 5.14: Comparison of reconstruction loss for varying γ and σ in a 2-move Painter network, trained on the 2-Shapes dataset.

With this analysis we can conclude that using a convex combination of templates is a good idea, and in fact the use of additional weight-sharpening is not necessary since the network will converge to a discrete template choice on its own.

There is one additional insight we can obtain from these experiments. Regarding the use of a transparency template to simulate null moves in the Omniglot dataset, inspecting the template choice data we noticed that this template was never chosen by the trained network. This means that the network is favoring the behavior described in 5.4, where needless moves either end up painting over the background or being overwritten by later moves. Because of this, we will not make use of this transparency template in the rest of the experiments.

5.6 Handling occlusion

As mentioned in Chapter 1, a correct painter system must be able to understand occlusion and perceive the layered structure of images. Since neither T-DRAW nor T-AIR were able to solve the order-invariant 2-Rectangles dataset, which has no color, hence no occlusions, they were naturally not able to handle an ordered task (this was tested, for sanity check, and it failed indeed). Although this is an important property of the layers that compose any natural scene, it is one that has been avoided by all of the recent papers in image modeling, even those that explicitly try to model images as composition of layers. As mentioned in Section 3.4, both [3] and [46] had the opportunity to tackle this issue but chose not to. As noted in that section, the use of additive canvases in the former makes this task unattainable, but in the latter it would

have been possible to test.

In this section we take an important step in showing that picking the moves' order after they have been chosen (instead of having a fixed architecture) increases the network's ability to cope with occlusion.

In general, the way to tackle this issue is the following. Using any model that outputs a set of T moves, $\mathbf{M} = \{\mathbf{M}_t\}_{t=1}^{T}$ (such as the Painter network, Fig. 5.1), we use a sorting module that takes all the moves and picks an ordering in which they will be placed onto the canvas. This allows the writer components to be freed from the constraint of having to identify the correct object in the correct order, being only required to identify the correct object instead, since the sorter takes care of the ordering. When applied to the Painter network this general sorting mechanism can be seen in Fig. 5.15.

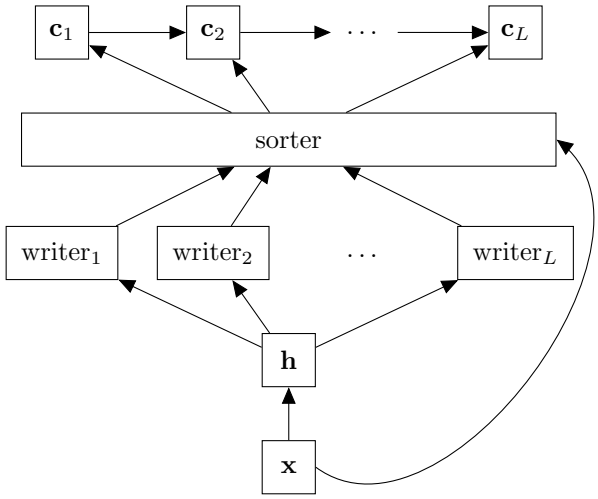


Figure 5.15: The Painter network architecture with sorting mechanism.

However, implementing a neural sorter is a non-trivial task, especially if trying to make it fully differentiable. In a classical computer science setting, a sorter is a function $s : \mathbb{R}^T \to \mathbb{R}^T$ that takes a list of T elements and returns a new list with those entities in the correct indexes, according to the some sorting criterion. In our case the elements are the T moves $\mathbf{M}$ and the criterion is trying to minimize the reconstruction cost. Since we want the sorter to be a neural network-based module, the move sequence can be computed in a single forward pass through the sorter, making it faster than regular sorting but, obviously, non-exact. Some implementation ideas are the following. Given the unordered moves $\mathbf{M}$, the sorter function s:

- stochastically picks one of the $T!$ possible output sequences. This approach scales very poorly with the number of moves and furthermore it is not differential, so policy gradient would be necessary to train it.

- at each time step t picks one among the $T - t$ moves remaining, and that move gets

removed from the list of moves that can be picked later in the sequence. This requires an MLP to be ran T times, which is good, but if the move-picking is stochastic it once again requires reinforcement learning to be used. However, just like in the multiple template case (Section 3.3), each move can be picked as a convex combination of the $\mathbf{M}$ moves provided and an additional RNN can be used to handle the sequential process. Although this approach has the advantage of being fully differentiable, it is actually inadequate since successively painting a convex combination of all the moves would cause the only move to be shown in the final canvas to be the last one (due to the overwriting property of the canvas used), causing all the previous moves to be useless and not receive any gradient.

Since both implementations have their disadvantages, neither is ideal, so an efficient and scalable implementation remains to be developed. A possible way to solve this problem is to make use of Neural Turing Machines [58] to handle the sorting mechanism, but due to the implementation difficulty and the time constraints in this project it was not possible to explore such a method.

Instead, a simple 2-object occlusion case was tested by using a simplified approach: after the 2 moves are computed and placed onto the canvas using their original fixed order, the intersection of the masks $\mathbf{p}^{mask}$ is computed and a convex combination of their colors, $\sum_t w_i \cdot h_i^{what}$, is used to color the intersection region. This yields a method equivalent to alpha blending used in computer graphics. The dataset used is the 2-Rectangles-Occlusion (see Appendix A), which a version of 2-Rectangles where each rectangle may have one of two grayscale tones (gray or black), such that occlusion is present unless the rectangles have the same color.

The results are shown in the figures below. Fig. 5.16 shows the painting process of a simple Painter without sorting module, and Fig. 5.17 shows the painting of a Painter with the sorting process described in the previous paragraph. Comparing the results from both images, we can see that when we include a sorting mechanism, all the occlusion situations were solved correctly (Fig. 5.17, while a simple Painter has significant more trouble and it is possible to see that not only were many occlusions wrongly identified, but the fact that they did caused some moves to have an incorrect color in order to correct for the incorrect placement.

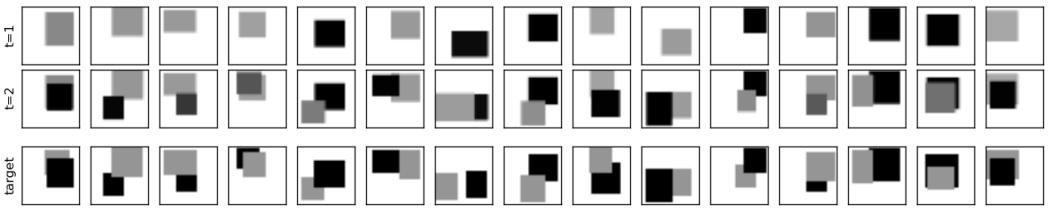


Figure 5.16: Painting process of a 2-move Painter network without sorter layer in the 2-Rectangles-Occlusion dataset.

With this simple but insightful experiment we can conclude that explicitly sorting the moves

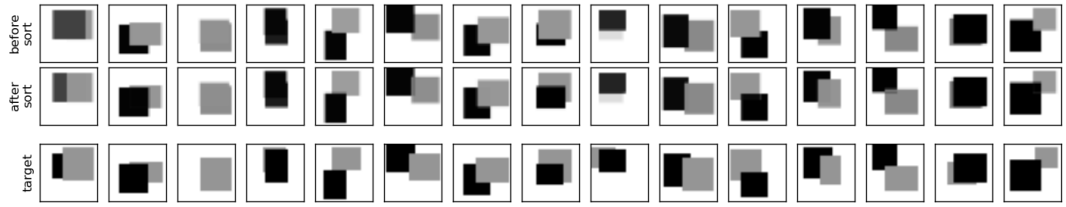


Figure 5.17: Painting process of a 2-move Painter network with sorter layer in the 2-Rectangles-Occlusion dataset. Top row shows the moves chosen before the sorting step is applied, and middle row shows the final canvas, after the sorting is applied.

after they have been chosen may provide substantial improvements to the ability to model scenes with occlusion. From Fig. 5.18 it is possible to understand why the standard Painter network is not able to appropriately cope with occlusion: after the network has approximately identified the image's constituting rectangles, it is stuck in move order that it has found so far, so it resorts to adjusting the colors to find a local minimum or making the rectangles mutually exclusive in the canvas, which defeats the occlusion modeling. These are very encouraging results but, as mentioned, this method does not scale to more than two objects and a good scalable method was not found during the course of this project.

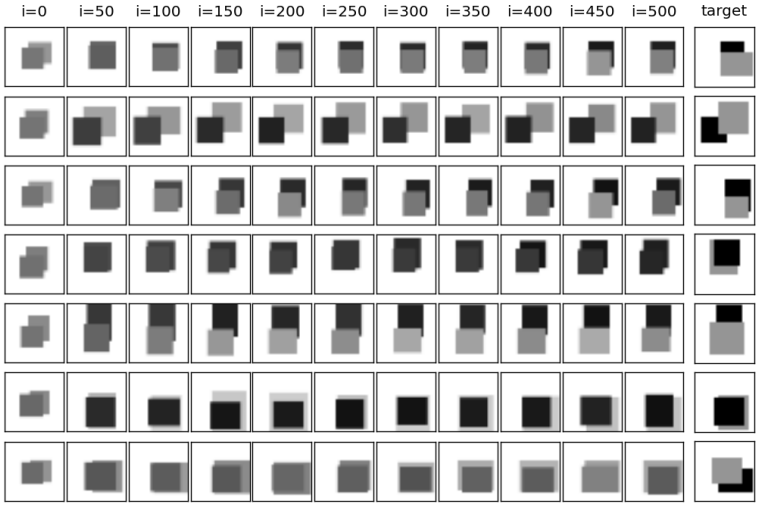


Figure 5.18: Training progress of a 2-move Painter network without sorter layer in the 2-Rectangles-Occlusion dataset, for the first 500 training iterations.

5.7 Scaling up

After having conceived a Painter network that is shown to be able to handle a wide variety of simple tasks, it is necessary to show that this same architecture is able to tackle more challenging tasks: larger images and more complex datasets. All the datasets that have originally had RGB images were converted to grayscale for simplicity and ease of experimentation, since this does not affect the difficulty of the task.

The main difference between the simple Painter network and the one used here is that convolutional layers were used instead of fully connected ones, since these are known to provide better feature extraction. For an overview of convolutional networks see [69].

Faces

The first dataset to be tested was the Faces dataset (see Appendix A). This dataset is composed of 100x100 face images taken from the BioID facial keypoint dataset [70]. Faces have a complex yet geometrically balanced structure: we can identify the face shape as a clear blob, eyes as dark blobs, mouth as another blob, etc, and they are in general symmetric. This makes it great to test a Painter , because it is easy to check if the network has a basic understanding of these blob-like structures.

This dataset contrasts with the ones used so far in the sense that all of the image has to be painted, unlike digits or rectangles where much of the background remains unchanged. Because of this aspect, we use this dataset to compare 3 important settings: the simple Painter, the recurrent Painter and a convolutional Painter. The simple and recurrent Painters use the same layer structure used so far, with the addition of a 256-dimensional layer to provide better feature extraction for these larger images, and the convolutional Painter uses two convolutional layers, with 10 and 5 feature maps respectively, the output of which corresponds to the hidden features $\mathbf{h}$. The writers' components, h^{what}, $\mathbf{h}^{where}$ and $\mathbf{h}^{which}$, are then computed using fully-connected layers as before.

The training process can be seen in Fig. 5.19. There are several interesting aspects to note here. The first is that the recurrent Painter never achieves a performance even close to that of the simple and convolutional Painters. In fact, looking at the recurrent Painter's samples in Fig. 5.20, it is clear that the network got stuck in some non-optimal minimum, resembling the behavior observed in the variational models. This means that although recurrent Painters can model the simple 2-Rectangles and MNIST datasets, they have problems when scaling to more complex settings. This shows that the variational objective is not the only source of error in sequential variational models: the recurrent structure is also plays its part in impairing appropriate convergence of the models. From the plot it is also visible that although a convolutional Painter achieves better performance than the simple Painter, the training process is not as stable. This is made clear by the fact that when changing the learning rate from the usual 10^{-3} to 10^{-4} in the convolutional Painter, the network immediately stabilized and found a good minimum.

In Fig. 5.21 we show we best results obtained in the Faces dataset with the convolutional Painter. We consider the latter reconstructions to be quite impressive, considering the amount of fine details that exists in faces. However, there is one scalability issue that must be addressed. During the experiments, we noticed that while performance improved very slowly with the increase in number of moves, and after some point (around 40 moves) it did not improve

anymore. This is likely due to the fact that above a certain number, many of the moves will end up being completely overwritten during the painting process, which means that they will never receive error gradients. If they do not receive error gradients they are never able to contribute to a better reconstruction, so the network tends to make use of approximately the same number of moves.

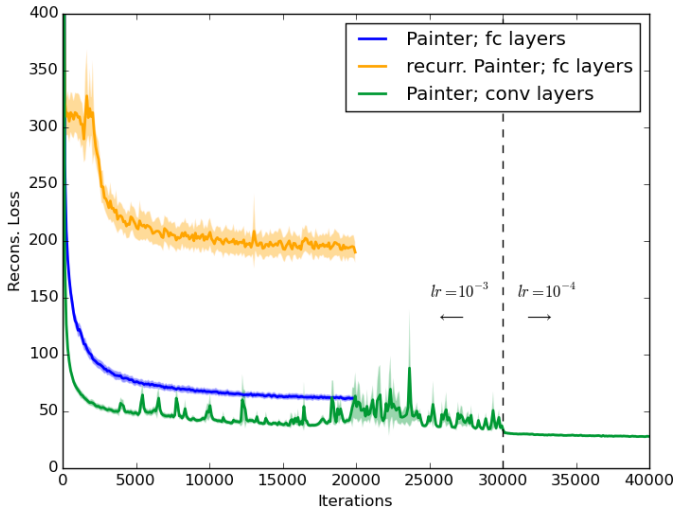


Figure 5.19: Comparison between 40-move simple Painter with fully-connected layers, 30-move recurrent Painter with fully-connected layers and a 40-move convolutional Painter, in the Faces dataset. The learning rate is changed from 10^{-3} to 10^{-4} after 30000 training iterations.

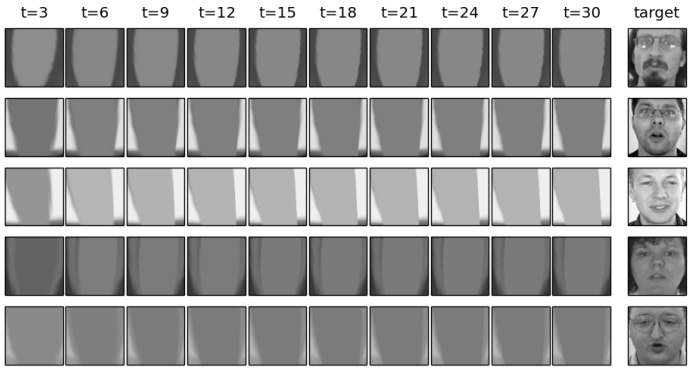


Figure 5.20: Painting process of a 30-move recurrent Painter network in the Faces dataset.

Omniglot

The Omniglot dataset [68] (see Appendix A) is a particularly interesting case in which to test a Painter network because it is essentially a more complex version of MNIST, and the images are 105x105 instead of 28x28, so there is a much greater level of detail. This dataset was used in 5.5 to test the performance of our multi-template implementation, but in this section we use the convolutional Painter using just the circle template and 40 moves to obtain the best

possible results.

Many reconstructions are shown in Fig. 5.22. Once again the results show that the network is able to reconstruct the images given with great precision[3], even under the strong painting constraints.

Cifar

The final and most challenging dataset tackled during this scalability testing section is the Cifar-10 dataset [71] (see Appendix A). This dataset is composed of 32x32 images of natural scenes with vast visual variety, which make it a very hard challenge. We run a 60-move convolutional Painter to try to obtain the best possible reconstructions. The results are shown in Fig. 5.23. We can see that although the images are not very sharp (actually, the target images are not very sharp to start with), the network makes a good effort in trying to capture most of the global visual patterns.

After the Faces and Cifar datasets it becomes clear that the Painter network is not able to capture the finer details of images. This is likely due to the use of mean squared error, since it promotes correct color on average (locally) instead of corners and edges.

[3] If you look at them from a distance it is nearly impossible to tell which are painted and which are the original.

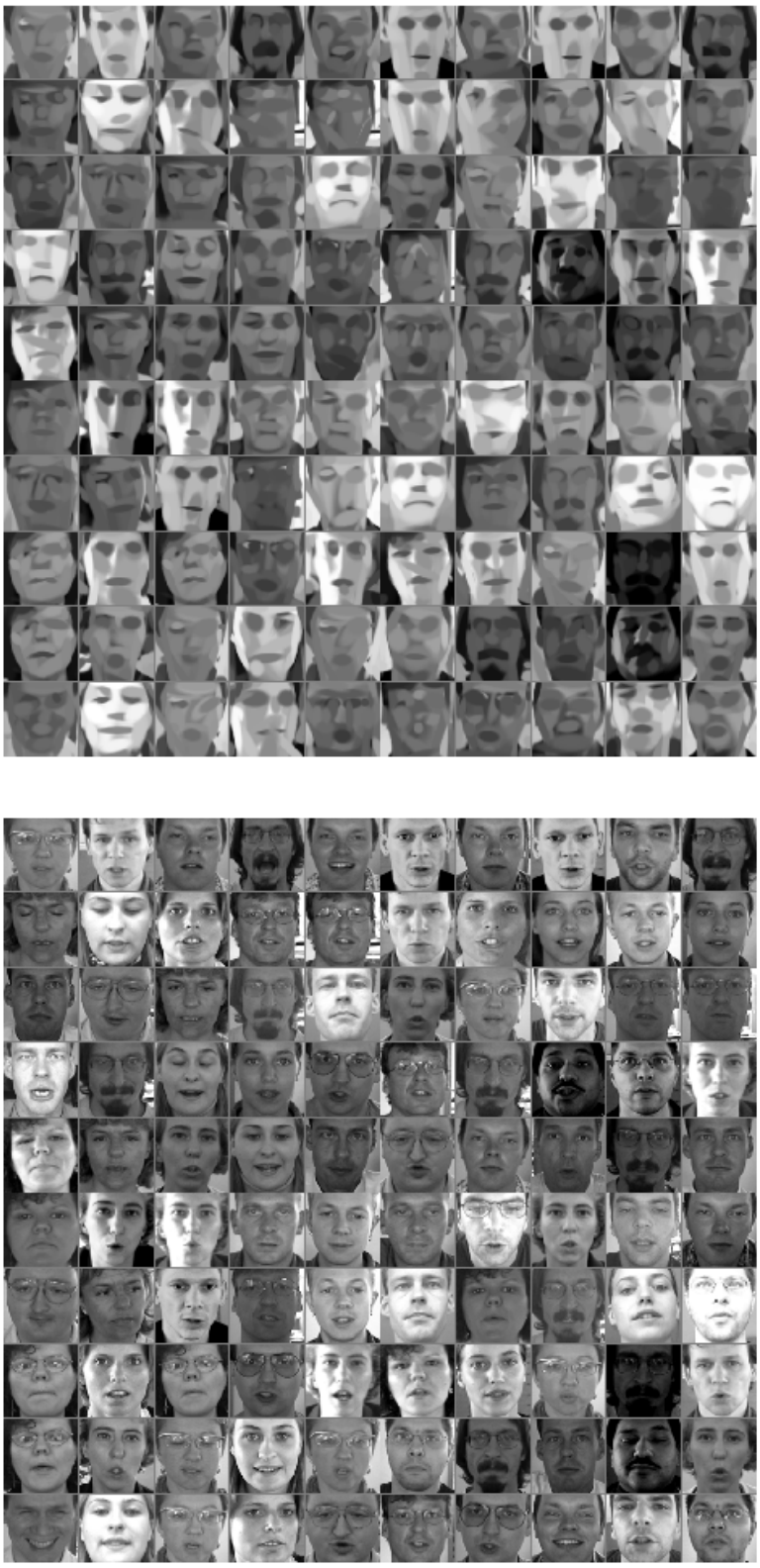


Figure 5.21: Painting reconstructions (top) of 100 target images (bottom) using a 40-move convolutional Painter network in the Faces dataset.

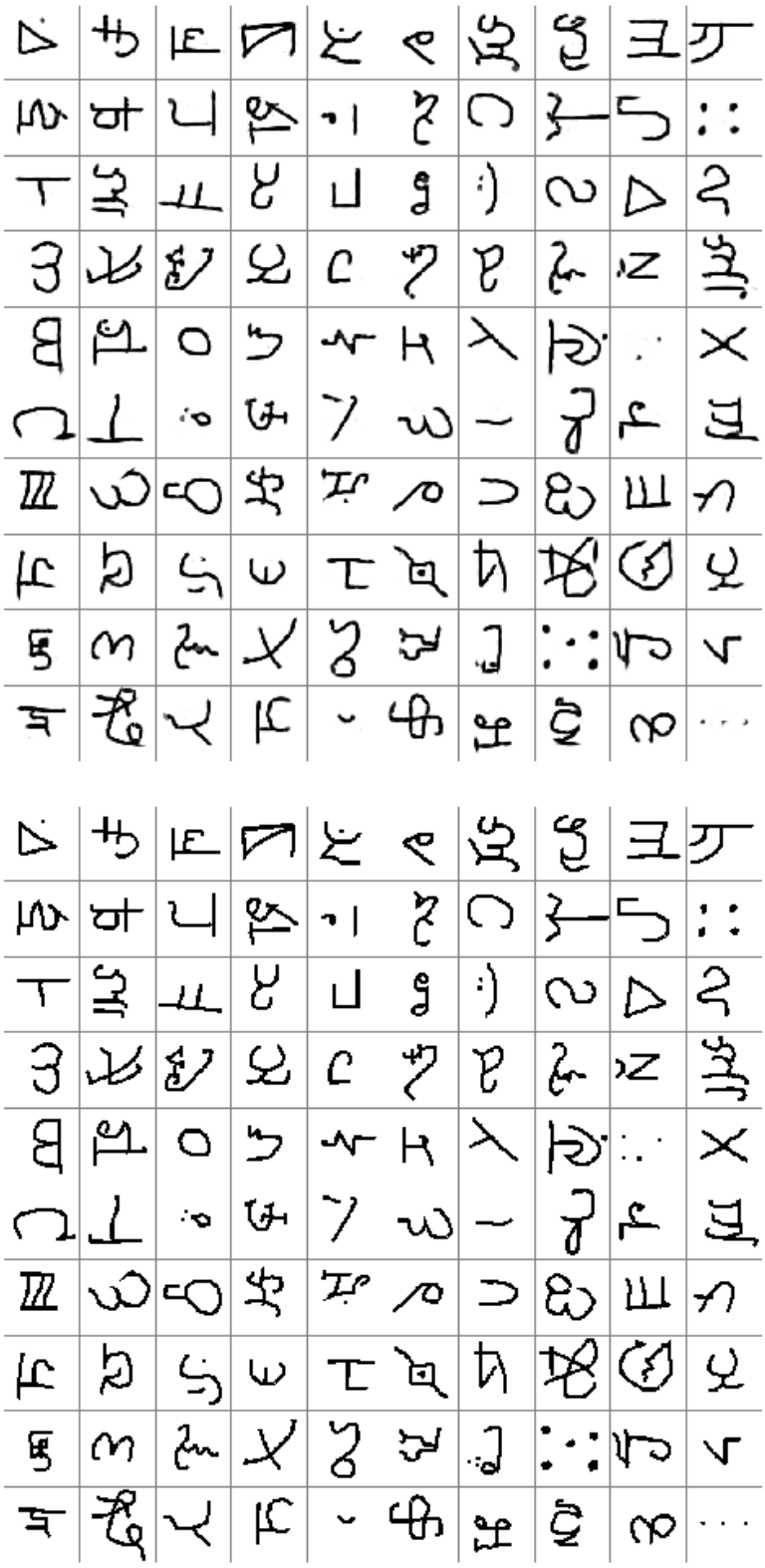


Figure 5.22: Painting reconstructions (top) of 100 target images (bottom) using a 40-move convolutional Painter network in the Omniglot dataset.

Figure 5.23: Painting reconstructions (top) of 100 target images (bottom) using a 60-move convolutional Painter network in the Cifar10 dataset.

Chapter 6

Discussion

6.1 Conclusion

In this work we showed that it is possible to create a neural network-based painter system that is fully differentiable and can scale to images containing intricate patterns. However, the elements of the template writer needed to perform painting moves, such as binary masks and spatial transformers, cause the convergence of a painter network to be very unstable for most settings other than the simplest one (deterministic or denoising autoencoder with mean squared reconstruction loss). Variational, but also recurrent, models suffered greatly from this instability, rendering them untrainable for the majority of tasks. The fact that any attempt to obtain a generative painting model through variational inference, either by using complex models like DRAW [2] and AIR [3] or simple variational versions of the Painter network proved unsuccessful was a great setback faced during this project, since it prevented us from going further and analyzing performance not only in terms of its reconstruction ability but also in terms of its generative abilities (by evaluating the variational lower bound obtained against current state-of-the-art models and doing generative sampling). Furthermore, the failures of T-DRAW and T-AIR required us to go back-to-basics in Section 5, and explore a wide range of basic tasks, which prevented us from focusing on building a more complex large-scale system. This was somewhat done in Section 5.7, but not to the extent that was desired at the start of the project. Nonetheless, the need to test a wide range of settings was paramount in laying a solid ground for the Painter network in terms of its basic abilities, with the correct architecture needed to tackle more challenging tasks in the future.

Despite the good results that were obtained by the Painter network architecture in the end, there are some criticisms to be made to the methods used. First and foremost, much of the analysis and model testing in this work was made through visual inspection of samples, and not necessarily numerical (other than cost terms) or theoretical exploration. The weak gradient signals passing through the template writer (due to the conjugation of spatial transformers

and templates) made it very hard to accurately determine what the model's source of failure is from numerical data and even more from the neural network's equations, despite numerous attempts. Therefore, extensive trial-and-error was necessary throughout the experiments in order to empirically obtain insight into a specific model's behavior and possible improvements. In fact, some other architectures were tested in the process of going from T-DRAW and T-AIR to the simple Painter network, although these were not mentioned because they did not provide additional insight into the root of the problem, which was found to be mostly the variational objective.

Also due to the great amount of time spent testing basic settings it was not possible to study the generalization abilities (on validation/test sets) of the network, which is a major part of the evaluation of any machine learning system. Most of the results here were reported on the training set since it was necessary that the models were able to even learn the training set before showing good generalization ability.

Throughout this work, a big effort was made in trying to keep the model fully differentiable. This forced us to employ some tricks, like weight sharpening, to simulate discrete variable choice in a continuous manner. However, these were shown not to be very effective, and perhaps using L2 regularization on the template combination weights could have been provided better results. Since discrete variable training using NVIL [37] was not tested, it remains to be seen if there are performance advantages to be gained from a non-fully differentiable system. For example, when using multiple templates, if the images are not explicitly composed of these templates (like the n-Shapes datasets), they will not be used appropriately, as the most versatile template will tend to be used (generally the circle template). This points to a sub-optimal use of the templates provided.

One additional problem noted during Section 5.7 was the fact that the Painter tends to generate low resolution versions of the images, when these contain complex visual patterns. We can identify two main reasons for this. Firstly, using an MSE reconstruction loss encourages the colors to be correct on average (locally), instead of correctly drawing corners and edges, which are the features humans use the when identifying pictures. Secondly, since natural images contain many color gradients, it is a hard task for the network to replicate these since it can only paint single-color shapes.

6.2 Future work

Due to time constraints, there were many interesting avenues of work that couldn't be addressed during the course of this project. One of the most important would be using the features learned (either using the hidden layer or the *where*, *what*, etc. outputs themselves) by the Painter network to train a supervised system on a dataset like MNIST, Omniglot or even Cifar10. Since the painter learns a very geometrical model of the images given, testing

such supervised or semi-supervised tasks would provide further insight into the usefulness of the features learned.

It would be of interest to analyze the causes of failure of variational models even further, since creating a generative model is an attractive goal in any unsupervised learning task. However, since the Painter network was shown to be able to perform denoising, it would be possible to generate samples using DAE-based sampling as purposed in [26], although such a generative process is not ideal.

Perhaps one of the most interesting problems that can be investigated in future work is the occlusion modeling, introduced in Section 5.6. The fact that the very simple sorter that was used provides such significant improvement in the tests with the occluded setting suggests that a more complete and scalable type of neural sorter could increase the ability of current state-of-the-art systems to model images as a composition of layers.

One additional task that can be conceived in the context of a painter is the following: what happens if the template bank is not fixed, but learnable? What kind of templates would the network construct in order to optimize for a given dataset? This setting was tested briefly, and preliminary results showed that the template remains nearly constant throughout the training, and the model makes use of whatever initialization was used for the template. If the template is randomly initialized, it will remain a square-like pattern; if we set it to a circle, it will remain a circle. This is likely due to the fact that the training of the template is slower than the rest of the parameters, so the variable template never gets a chance to adapt to the dataset. This setting should be explored further, though, since only a brief experiment was made.

Despite the extensive use of writing attention, reading attention was only used in the T-DRAW and T-AIR models, not on the Painter. As mentioned in Section 2.3, reading attention can help not only with scalability but also with performance. Using spatial transformers for reading attention can be incorporated into the Painter network by adding a *read* step before each move, such that a writer's input would only be conditioned on a particular portion of the image. This is an interesting option to consider in future work if we intend to scale to larger natural images and complex problems.

Bibliography

[1] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial Transformer Networks," in *Advances in Neural Information Processing Systems*, pp. 1–14, 2015.

[2] K. Gregor, I. Danihelka, A. Graves, D. Jimenez Rezende, and D. Wierstra, "DRAW: A Recurrent Neural Network For Image Generation," in *International Conference on Machine Learning*, pp. 1462–1471, 2015.

[3] S. M. A. Eslami, N. Heess, T. Weber, Y. Tassa, K. Kavukcuoglu, and G. E. Hinton, "Attend, Infer, Repeat: Fast Scene Understanding with Generative Models," *arXiv preprint arXiv:1603.08575*, 2016.

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems 27*, pp. 2672–2680, 2014.

[5] H. Valpola, "From Neural PCA to Deep Unsupervised Learning," in *Adv in Independent Component Analysis and Learning Machines*, pp. 143–171, 2015.

[6] T. D. Kulkarni, W. Whitney, P. Kohli, and J. B. Tenenbaum, "Deep Convolutional Inverse Graphics Network," in *Advances in Neural Information Processing Systems*, 2015.

[7] M. Abadi, A. Agarwal, P. Barham, and E. Al, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *None*, p. 19, 2015.

[8] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[9] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, oct 1986.

[11] M. A. Nielsen, "Neural Networks and Deep Learning," 2015.

[12] L. Y., B. Y., and H. G., "Deep learning," *Nature*, vol. 521, pp. 436–444, may 2015.

[13] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition,"

[17] D. Amodei, R. Anubhai, E. Battenberg, and E. Al, "Deep-speech 2: End-to-end speech recognition in English and Mandarin," *arXiv preprint arXiv:1512.02595*, 2015.

[18] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN EncoderDecoder for Statistical Machine Translation," *arXiv preprint arXiv:1406.1078*, 2014.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. a. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[20] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, jan 2016.

[21] M. Ranzato, Y.-l. Boureau, and Y. L. Cun, "Sparse Feature Learning for Deep Belief Networks," in *Advances in Neural Information Processing Systems*, pp. 1185–1192, 2008.

[22] P. Foldiak and P. Földiak, "Sparse coding in the primate cortex," *The Handbook of Brain Theory and Neural Networks*, 2002.

[23] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *International conference on Machine learning*, pp. 1096–1103, 2008.

[24] P. Vincent, "A Connection Between Score Matching and Denoising Autoencoders," *Neural Computation*, vol. 23, no. 7, pp. 1661–1674, 2010.

[25] A. Hyvärinen, "Estimation of non-normalized statistical models by score matching," *Journal of Machine Learning Research*, vol. 6, pp. 695–708, 2006.

[26] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," *Advances in Neural Information Processing Systems*, 2013.

[27] Y. Bengio, E. Thibodeau-Laufer, G. Alain, and J. Yosinski, "Deep Generative Stochastic Networks Trainable by Backprop," in *International Conference on Machine Learning*, pp. 226–234, 2014.

[28] A. Rasmus, H. Valpola, and M. Berglund, "Semi-Supervised Learning with Ladder Network," in *Advances in Neural Information Processing Systems*, pp. 3532–3540, 2015.

[29] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.

[30] R. Salakhutdinov and G. E. Hinton, "Deep Boltzmann Machines," in *International Conference on Artificial Intelligence and Statics*, no. 3, pp. 448–455, 2009.

[31] G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal, "The "wake-sleep" algorithm for unsupervised neural networks.," *Science*, vol. 268, no. 5214, pp. 1158–1161, 1995.

[32] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets,"

[33] R. Salakhutdinov and H. Larochelle, "Efficient learning of deep boltzmann machines," *AISTATs*, vol. 9, pp. 693–700, 2010.

[34] M. I. Jordan, T. S. Jaakkola, and L. K. Saul, "An Introduction to Variational Methods for Graphical Models," *Machine Learning*, vol. 37, pp. 183–233, 1999.

[35] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *International Conference on Learning Representations*, 2014.

[36] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models," *arXiv preprint arXiv:1401.4082*, 2014.

[37] A. Mnih and K. Gregor, "Neural Variational Inference and Learning in Belief Networks," *arXiv:1402.0030*, 2014.

[38] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

[39] D. J. Rezende and S. Mohamed, "Variational Inference with Normalizing Flows," *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1530–1538, 2015.

[40] Y. Burda, R. Grosse, and R. Salakhutdinov, "Importance Weighted Autoencoders," in *International Conference on Learning Representations*, pp. 1–12, 2015.

[41] C. K. Sønderby, T. Raiko, L. Maaløe, S. K. Sønderby, and O. Winther, "Ladder Variational Autoencoders," *arXiv preprint arXiv:1602.02282*, 2016.

[42] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] P. J. Werbos, "Backpropagation Through Time: What It Does and How to Do It," in *Proceedings of the IEEE*, vol. 78, pp. 1550–1560, 1990.

[44] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *International Conference on Artificial Neural Networks*, pp. 44–51, 2011.

[45] T. Tieleman, "Optimizing Neural Networks that Generate Images," *PhD thesis*, 2014.

[46] J. Huang and K. Murphy, "Efficient Inference in Occlusion-Aware Generative Models of Images," *arXiv preprint arXiv:1511.06362*, 2015.

[47] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv preprint arXiv:1511.06434*, 2015.

[48] V. Dumoulin, I. Belghazi, B. Poole, A. Lamb, M. Arjovsky, O. Mastropietro, and A. Courville, "Adversarially Learned Inference," *arXiv preprint arXiv:1606.00704*, 2016.

[49] D. J. Im, C. D. Kim, H. Jiang, and R. Memisevic, "Generating images with recurrent adversarial networks," 2016.

[50] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel Recurrent Neural Networks," in *International Conference on Machine Learning*, 2016.

[51] H. Larochelle and I. Murray, "The Neural Autoregressive Distribution Estimator," in *International Conference on Machine Learning*, vol. 15, pp. 29–37, 2011.

[52] D. J. Rezende, S. Mohamed, I. Danihelka, K. Gregor, and D. Wierstra, "One-Shot Generalization in Deep Generative Models," *arXiv preprint arXiv:1603.05106*, 2016.

[53] K. Gregor, F. Besse, D. Jimenez Rezende, I. Danihelka, and D. Wierstra, "Towards Conceptual Compression," *arXiv preprint arXiv:1604.08772*, 2016.

[54] D. J. Rezende, S. M. A. Eslami, S. Mohamed, P. Battaglia, M. Jaderberg, and N. Heess, "Unsupervised Learning of 3D Structure from Images," *arXiv preprint arXiv:1607.00662*, 2016.

[55] J. Schmidhuber and R. Huber, "Learning To Generate Artificial Fovea Trajectories for Target Detection," *International Journal of Neural Systems*, vol. 02, pp. 125–134, 1991.

[56] K. Xu, A. Courville, R. S. Zemel, and Y. Bengio, "Show , Attend and Tell : Neural Image Caption Generation with Visual Attention," *arXiv preprint arXiv:1502.03044*, 2015.

[57] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference on Learning Representations*, 2015.

[58] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines," *arXiv preprint arXiv:1410.5401*, 2014.

[59] J. Peters and S. Schaal, "Policy Gradient Methods for Robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.

[60] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent Models of Visual Attention," in *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[61] J. L. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple Object Recognition With Visual Attention," in *International Conference on Learning Representations*, 2015.

[62] M. M. Loper and M. J. Black, "OpenDR: An Approximate Differentiable Renderer," in *European Conference on Computer Vision*, 2014.

[63] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation.," *arXiv preprint arXiv:1308.3432*, 2013.

[64] W. Whitney, *Disentangled Representations in Neural Models*. PhD thesis, Massachusetts Institute of Technology, 2016.

[65] T. Porter and T. Duff, "Compositing digital images," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 253–259, 1984.

[66] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[67] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv:1502.03167*, pp. 1–11, 2015.

[68] B. M. Lake, R. Salakhutdinov, and J. B. Tnenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[69] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pp. 253–256, 2010.

[70] "BioID Faces Database." `https://ftp.fau.de/pub/facedb/readme.html`.

[71] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.

Appendix A

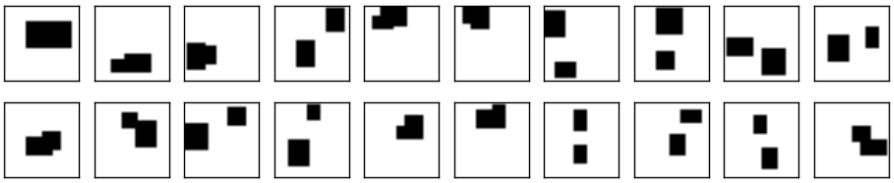
Dataset samples and templates used

Here are shown data samples and short descriptions of all the datasets used in this work, along with a list of templates used. Every dataset (except Faces and Cifar10) and template is shown in inverted grayscale color, here and throughout this work, for ease of visualization.

A.1 Datasets

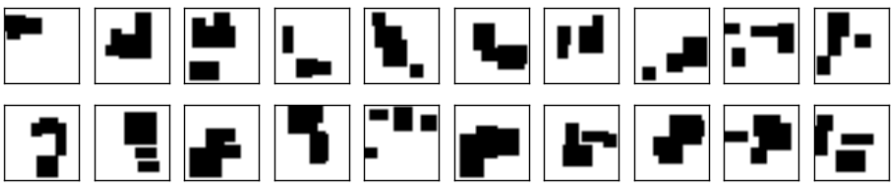
2-Rectangles

Artificial dataset where each image contains two randomly placed rectangles within the image's boundary. The rectangles have vertical and horizontal deformation, but no rotation. All rectangles are black and the images are 28x28. Dataset size: 10000 images.



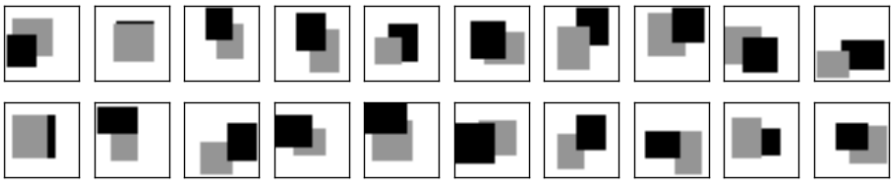
4-Rectangles

Artificial dataset similar to the 2-Rectangles, but with four randomly placed rectangles. Dataset size: 10000 images.



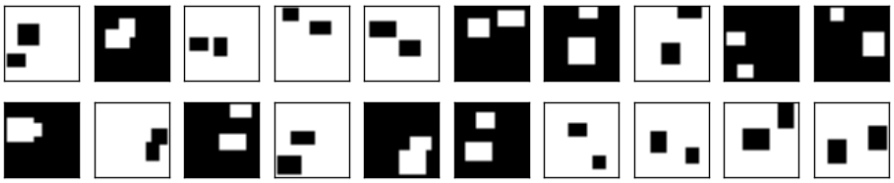
2-Rectangles-Occlusion

Artificial dataset similar to the 2-Rectangles, but one of the rectangles is black and the other one is gray, placed at random locations in a random order. The rectangles can occlude each other. Dataset size: 10000 images.



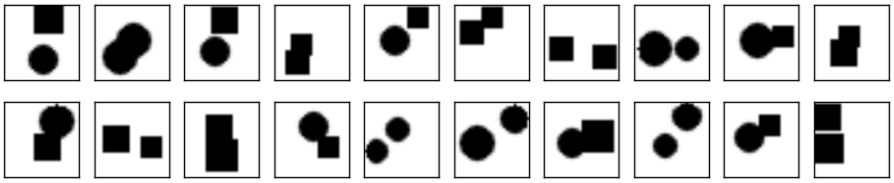
2-Rectangles-Mixed

Artificial dataset similar to the 2-Rectangles, but each image may contain black rectangles in white background or white rectangles in black background. Dataset size: 10000 images.



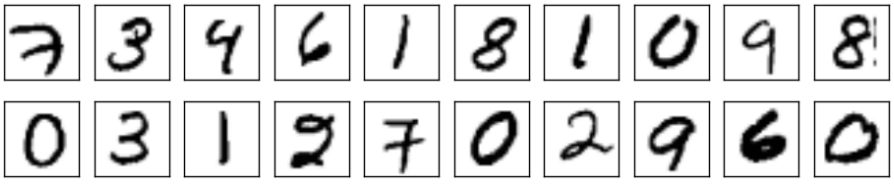
2-Shapes

Artificial dataset where each image contains two randomly placed shapes (rectangles or circles). Each image may have zero, one or two of the same shape. The shapes only contain isomorphic scale deformation. Dataset size: 10000 images.



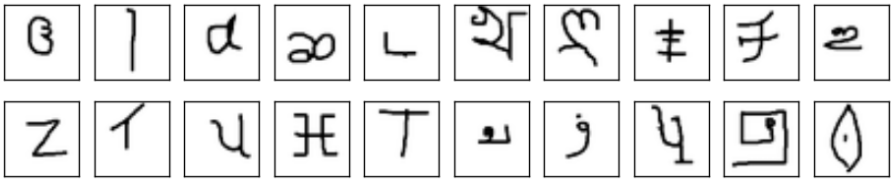
MNIST

In the MNIST dataset [66] each image contains a handwritten digit between 0 and 9. The images are 28x28. Dataset size: 60000 images.



Omniglot

In the Omniglot dataset [68] each image contains a handwritten character from 30 different alphabets. The images are 105x105. Dataset size: 19280 images.



Faces

In the Faces dataset each image contains a face from the BioID Face Database [70]. To create this dataset we cropped just the face region from each original image (since these contain more than just the face) in order to obtain a 100x100 image. The images are originally in grayscale. Dataset size: 1521 images.

Cifar10

In the Cifar10 dataset [71] each image contains an natural scene from 10 different object classes. The images are 32x32. The dataset is originally RGB but it was converted to grayscale. Dataset size: 50000 images.

A.2 Templates