UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

A look into generative query networks

Süleyman Güney
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Nick MICHIELS

BEGELEIDER :
De heer Bram VANHERLE

2022
2023

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

A look into generative query networks

Süleyman Güney

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Nick MICHIELS

BEGELEIDER :

De heer Bram VANHERLE

Universiteit Hasselt

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

A look into Generative Query Networks

Author:

Güney Süleyman

Promotor:

prof. dr. Michiels Nick

Supervisor:

Vanherle Bram

Academiejaar 2022-2023

Contents

Chapter 1

Introduction

Humans have certain skills that are naturally developed to some degree just from experience. Some of these skills are the ability to form thoughts, to commit sensory information to memory and form a foreknowledge, and to use this foreknowledge to make inferences about certain things.

The problem domain which will be the focus of this thesis is "Neural Scene Representation and Rendering" (NSRR). This is the process where the visual information in a scene gets compressed into a concise description, i.e. a representation vector, which will then be used for predicting the view of the scene from previously unobserved viewpoints.

This concept can be closely related to the human ability to commit information to memory and make new inferences and imaginings from this foreknowledge. Each human being who has a home will know the layout and contents of their home inside-out. But in what way is this information stored in our minds? It is not every image we have perceived of our homes until now, but some abstract knowledge that has been derived from our sensory inputs. We convert the information we perceive to some abstract form which gets stored in our memory. [Tak19] So when we are thinking about the layout of our house we may remember the most recent or nostalgic images of it that we have perceived with our eyes. But we could also form the floor plan of the house through imagining and making inferences from our foreknowledge. We could go even further and form a detailed image of the living room. Some may be blessed with photographic memory which would enable them to imagine the exact image they perceived, but most humans will form an image from abstract information which has been stored in memory. With this abstract information, we would remember the rough shape of the room, the positions of the objects, the colours of the walls, etc. Once the rough details are remembered we can retrieve even more information and fill in the details, like the design of the TV, the pattern on the carpet, the paintings on the walls, etc. [LH08] [BEN45] [Bra+08]

This is exactly what a model would do that solves the NCRR problem, like the Generative Query Network (GQN) [Esl+18]. This model is composed of two sub-models, i.e. the representation network and the generation network. As their names imply, the representation network is responsible for creating and updating a representation vector of the scene. This vector is the "abstract knowledge" that has been gathered about the scene so far. The generator network will then take this representation vector as input along with a viewpoint and will generate an image. The generated image is the "imagined" image that the GQN believes to be perceived had the scene been witnessed from the given viewpoint. The generator network converts the representation vector - the "abstract knowledge" - into an image with all necessary details, this makes the parameters of the generator network the foreknowledge. So making an inference with the generator network corresponds with the human ability to apply the foreknowledge of perceived information to make visual inferences.

To understand the importance of unsupervised representation learning, we must look at the drawbacks of supervised alternatives. These supervised alternatives are systems that provide various hand-made representations on given scenes, such as pose estimation [Zhe+23], detecting object bounding boxes and labelling them [RF16][Liu+16], semantic segmentation [RFB15a] [RFB15a], etc. These systems are developed by training neural networks on large human-labelled datasets [Kri09][Gar+17][Lin+15][GB19][Rus+15]. It is important to remove the dependence on humans for labelling data due to various drawbacks. First, it takes a lot of time and effort to provide labels for large datasets and since there is a never-ending

demand for more data, manual labelling does not look like a viable option in the long term. Secondly, humans are prone to make errors, so it is unavoidable to introduce faulty samples in large datasets. Finally, the information encoded in hand-made scene representations is fixed and limited. Fixed because models will only learn the representations that are given, if a visual feature is not described by the given representation then it cannot be learned. Limited because larger representations will take more time to create and are more likely to contain errors.

Unsupervised representation learning does not suffer from these drawbacks since there is no need for labels. Instead of labels, the representations are learned from the statistical properties of the training dataset. This has the benefit of learning the optimal representation for the task at hand. The disadvantage is that the learned representations may not be interpretable by humans, due to representation vectors often being entangled, i.e. the lack of a clear relation between scene properties and elements of the representation vector. However, advancements have been made in recent years to learn disentangled representation vectors which are summarized in [Wan+22].

As noted, the GQN solves the problem of NCRR to some extent, albeit in a virtualized environment. The focus of this thesis will be to study and analyze the capabilities and limitations of the GQN. During this analysis, the goal will be to find answers to the following questions:

1. How can a multi-layer GQN be implemented and can it perform better than the standard one-layer GQN as proposed by [Esl+18]?
2. To what extent can the GQN represent and generate images of unseen objects and scenes? How dissimilar can the new scenes get before the GQN starts showing noticeable artefacts and is unable to properly represent the scene and generate images of it?
3. What are the effects on the generated images when modifying the representation vector?

The structure of the thesis has roughly the following form:

- Chapters about the foreknowledge required to understand the GQN model:
  - Chapter 2 Recurrent Neural Networks: explains the fundamental method that allows the GQN to generate images in multiple steps, instead of being constrained to generate whole images in a single step.
  - Chapter 3 Generative Models: gives an overview of the different types of generative models and how they work. This chapter also includes the Deep Recurrent Attentive Writer model (DRAW), the model on which the GQN's generator is based.
- Chapter 4 Generative Query Networks: gives a detailed explanation of the GQN model along with our contribution where we attempt to chain multiple generator layers of different resolutions.
- Chapter 5 Experiments: contains the experiments which were conducted in an attempt of answering the aforementioned research questions.
- Chapter 6 Conclusion: contains a critical reflection of the conducted research and self-reflection of the author.

Chapter 2

Recurrent Neural Networks

2.1 Standard Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of neural network that has the capability of handling sequential data of variable length, like sentences and audio sequences.

Some of the applications of RNNs are:

- Speech recognition
- Machine translation
- Video tagging
- Forecasting time series

To retain information about previous steps, a *hidden state* h_t is updated and passed onto the next *state* t. It is important to make a distinction between a hidden *layer* and a hidden *state*. A hidden layer represents the layer of nodes that is hidden from view, just as in a Multi-Layer Perceptron (MLP). A hidden state, however, refers to the particular *state* of a hidden layer. The state of a hidden layer at step t can essentially be seen as the inputs it receives at that step.

An obvious question that arises at this point is: "If the state of a layer is the input it receives, don't MLPs also have states? By feeding-forwarding data through the MLP, can the inputs that are passed to each consecutive layer be considered as the next hidden state?". This is not true however, the inputs of the hidden layers of an MLP can be considered to have a single "state", whose lifespan ends as soon as they are passed onto the *next* layer. The next time data is passed through the network for inferencing, those "hidden states" will be lost and will not be considered in the computation of the new input data. Hence, MLPs don't have hidden states.

The difference between MLPs and RNNs is that each recurrent layer in an RNN passes its output (hidden state) back to itself as input, which will be used when processing the *next* input. These hidden states persist until they are explicitly reset making them suitable for processing sequential data of arbitrary length. However, processing very long sequences causes problems in computing the gradients (e.g. vanishing/exploding gradient problem).

2.1.1 Definition

Let H_t and O_t be the hidden state and the output of the output layer at time step t respectively. These can be computed as follows:

$$\mathbf{H}_t = \phi_h(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h) \tag{2.1}$$

$$\mathbf{O}_t = \phi_q(\mathbf{H}_t\mathbf{W}_{hq} + \mathbf{b}_q) \tag{2.2}$$

- ϕ_h, ϕ_q are the activation functions of the hidden and output layer respectively

- $\mathbf{H}_t, \mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ are the matrices representing the hidden states at time step t and $t-1$ respectively. Their computation (the right-hand-side of equation 1) represent the hidden layer (see "Rec" in figure 1).
- $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ is the input mini-batch, with n the batch-size and d the dimension of the samples
- $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ is the weight matrix for mapping d-dimensional data samples to h-dimensional hidden state
- $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ is the weight matrix to manipulate H_{t-1}
- $\mathbf{b}_h \in \mathbb{R}^{1 \times h}, \mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the bias vectors of the hidden and output layers respectively. Note that they need to be broadcasted (copied) along the rows to size $n \times h$ or $n \times q$ to make the addition possible.

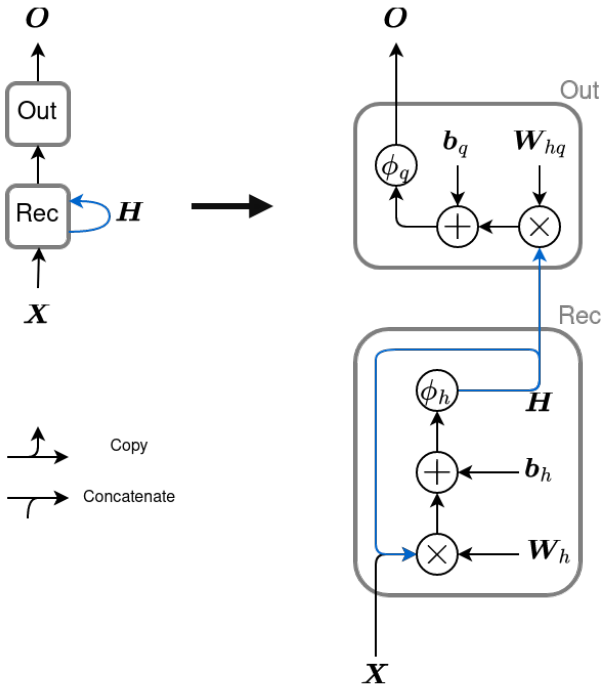


Figure 2.1: Simple RNN structure. Left: simplified schema. Right: expanded schema. The gray areas Rec (recurrent) and Out (output) are fully connected layers with an activation function. $\mathbf{W}_h$ is the concatenation of the weight matrices $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$. $\mathbf{X}$ is the input and $\mathbf{O}$ the output. The blue arrows represent the flow of hidden states. The hidden state at the first time step H_1 must be initialized appropriately. Source: Author.

In figure 2.1, we can see a schematic overview of the equations above: the "Rec" area corresponds with equation 1 and "Out" corresponds with equation 2. This graph contains cycles and is the rolled version of the RNN schema, the unrolled version can be seen in figure 2.2. Another thing to note is that $\mathbf{X}$ and $\mathbf{H}_t$ are concatenated at each step and are then multiplied with the concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$. This produces the same result as $\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh}$.

One way to increase the capacity of an RNN is to increase the amount of nodes in the hidden layers. Another way to increase its capacity is to chain multiple recurrent layers as shown in figure 2.3.

2.2 Vanishing/exploding gradients

RNNs suffer from the vanishing or exploding gradients problem where the gradients tend to vanish ($\to 0$) or explode ($\to \infty$) respectively. This is due to the long chain of multiplications that are required to compute the gradients for updating the model parameters during training.

Consider the following simplified notation for calculating the hidden state h and output o at step t:

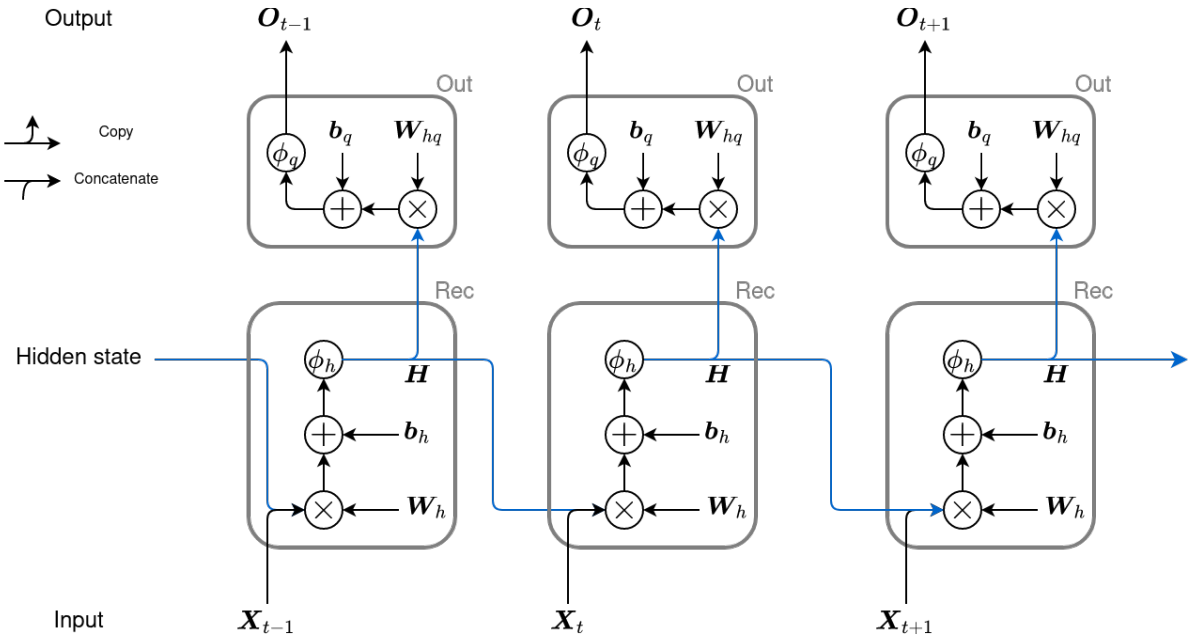


Figure 2.2: Simple RNN structure with unrolled schema. Each column represents a state of the recurrent layer along with its output layer, at three consecutive time steps $t-1$, t and $t+1$. The blue arrows represent the flow of hidden states. As in figure 2.1: $\mathbf{W}_h = [\mathbf{W}_{xh}\mathbf{W}_{hh}]$. Source: Author.

$$h_t = f(x_t, h_{t-1}, w_h) \tag{2.3}$$
$$o_t = h(h_t, w_o) \tag{2.4}$$

- With x_t the input at step t
- With w_h and w_o the weights of the hidden and output layer respectively
- With f and g some kind of transformation functions of the hidden and output layer respectively

Consider the following generic loss function for calculating the difference between labels y_t and predictions o_t:

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t) \tag{2.5}$$

The challenging part of backpropagation for RNNs is calculating the partial derivative of w_h:

$$\begin{aligned}\frac{\partial L}{\partial w_h} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial w_h} \\ &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial g(h_t, w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}\end{aligned} \tag{2.6}$$

- With T the total number of steps

The third factor $\frac{\partial h_t}{\partial w_h}$ is the challenging one to calculate as h_t is computed recurrently and depends on h_{t-1}, which on its turn depends on h_{t-2}, and so on:

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h}. \tag{2.7}$$

Consider the sequences $\{a_t\}, \{b_t\}, \{c_t\}$ with $t \in \{1, ..., T\}$ which satisfy the following:

$$\forall t : a_0 = 0 \wedge a_t = b_t + c_t a_{t-1} \tag{2.8}$$

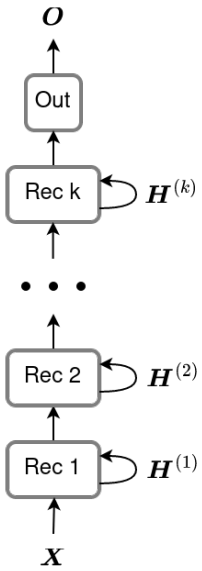


Figure 2.3: Deep RNN with k chained recurrent layers, each with its own hidden state. Source: Author.

Then for $t \geq 1$, it can be shown that ([Zha+] chapter 9.7):

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} c_j \right) b_i \tag{2.9}$$

The gradient computation in equation 2.7 satisfies $a_t = b_t + c_t a_{t-1}$ with the following substitutions:

$$\begin{aligned} a_t &= \frac{\partial h_t}{\partial w_h} \\ b_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} \\ c_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \end{aligned} \tag{2.10}$$

Carrying this substitution out yields the following gradient computation:

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1} \underbrace{\left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}} \right)}_{\text{problematic}} \frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h} \tag{2.11}$$

Thus, the longer the sequence length, the longer the chain of multiplications. Take, for example, an audio sequence of four seconds sampled at the standard rate of 44.1kHz which would yield a sequence length of 176400 samples. If the gradients being multiplied in the chain are less than one, then the final result would become vanishingly small. Conversely, if the gradients in the chain are larger than one, then the final result would become very large.

2.3 Long Short-Term Memory (LSTM)

As described Recurrent Neural Network (RNN) is a type of neural network that has the capability of handling sequential data of variable length and remembering past outputs by passing the output of a hidden state back to itself.

The major drawback of this method is that gradients tend to vanish ($\to 0$) or explode ($\to \infty$) rather quickly due to the recurrent connections and the way that gradients are calculated. Long Short-Term

Memory (LSTM) and Gated Recurrent Unit (GRU) are improvements of the RNN that mitigate this problem.

2.3.1 LSTM intuition

An LSTM model is essentially an RNN with each recurrent node replaced by a *Gated Memory Cell*. Each cell consists of an internal state (cell state), a hidden state (i.e. cell output) and three gates. The gates regulate how the cell state gets updated and how the cell state affects the cell output. The three gates are as follows:

- Forget gate: determines what will be forgotten of the previous cell state.
- Input gate: determines how the cell state will be updated with the cell input.
- Output gate: determines how the cell state will affect the cell output.

2.3.2 LSTM architecture

The Input, Forget and Output gates

Each gate can be represented by a fully-connected layer followed by a sigmoid activation function so that the outputs are matrices with elements in the range of $(0, 1)$:

$$\begin{aligned} \text{Input gate } \mathbf{I}_t \in \mathbb{R}^{n\times h}: &\quad \mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i) \\ \text{Forget gate } \mathbf{F}_t \in \mathbb{R}^{n\times h}: &\quad \mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f) \\ \text{Output gate } \mathbf{O}_t \in \mathbb{R}^{n\times h}: &\quad \mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o) \end{aligned} \tag{2.12}$$

- t is the current step
- n is the batch size
- h is the dimension of the cell state and hidden state
- $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ is the d-dimensional input
- $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$ is the hidden state of the previous step
- $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ are the weight matrices of the gates
- $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ are the biases of the gates
- σ is the sigmoid function that maps real values to the range of $(0, 1)$

Note that the biases need to be broadcasted to size $h \times h$ in order to make the addition possible. Broadcasting from size $1 \times h$ to $h \times h$ is simply done by copying and stacking the single row h times.

The Input node

The input node determines *how much* the input gate will be added to the cell state. Like the three gates, it is also defined by a fully-connected layer, but followed by a tanh function instead of a sigmoid function. The tanh function has a range of $(-1, 1)$, unlike the $(0, 1)$ range of the sigmoid function which enables the addition of negative values to the cell state.

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c), \tag{2.13}$$

Where

- $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$ is the input node
- $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ are the weights of the input node
- $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ is the bias

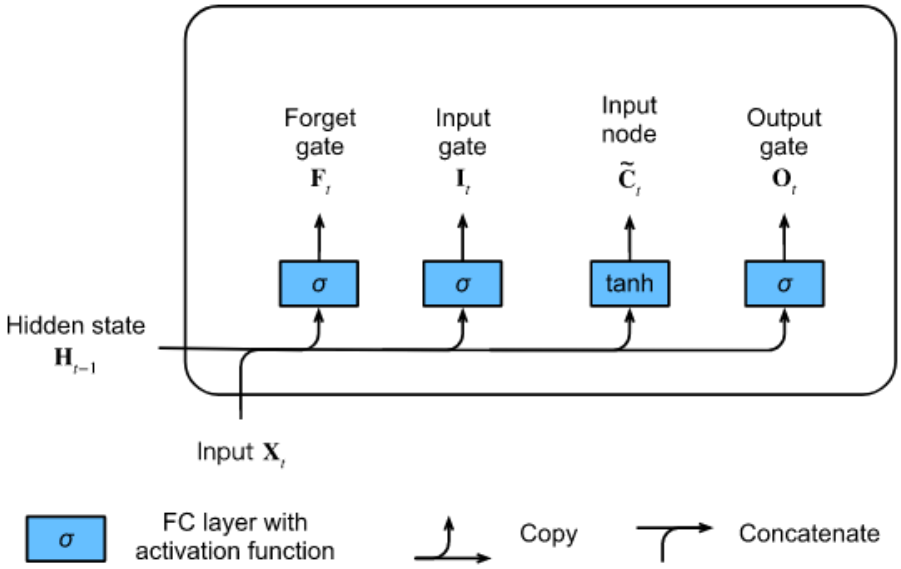


Figure 2.4: The Forget, Input and Output gates of the LSTM along with the input node. FC layer means fully-connected layer. Source: [Zha+]

Updating the cell state

The cell state is updated by the forget gate and the Input gate as follows:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \tag{2.14}$$

The previous cell state $\mathbf{C}_{t-1}$ is first manipulated by the forget gate F_t to determine what fraction of the previous cell state $\mathbf{C}_{t-1}$ will be involved in the calculation of the current cell state $\mathbf{C}_t$. This can simply be done with the elementwise (Hadamard) multiplication of both matrices since all elements of F_t are in the range of $(0, 1)$.

Next, $\mathbf{I}_t \odot \tilde{\mathbf{C}}_t$ is added to attain the new cell state $\mathbf{C}_t$. The input gate $\mathbf{I}_t$ is a transformation of the cell input and the input node $\tilde{\mathbf{C}}_t$ determines what fraction of $\mathbf{I}_t$ will be added to attain the new cell state. The elements of the input node have a range of $(-1, 1)$, meaning that the input gate can also be subtracted if it becomes negative.

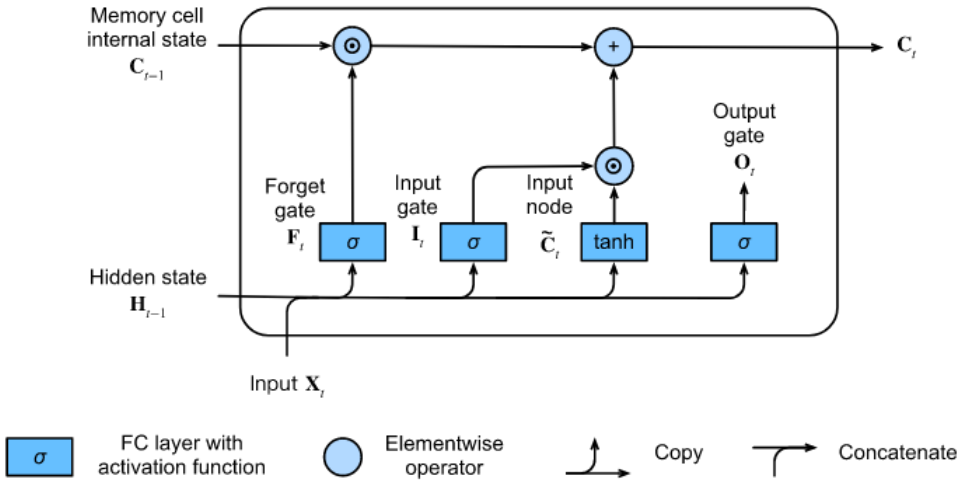


Figure 2.5: Computing the cell state $\mathbf{C}_t$ of the LSTM model. Source: [Zha+]

These operations are illustrated in figure 2.5 with the circular nodes representing elementwise operations. Note that figure 2.5 only represents a single state t of one LSTM cell. Each cell state $\mathbf{C}$ is internal and

are not passed onto other LSTM cells.

Computing the cell output

The final step is to compute the cell output, i.e. the hidden state of the current step $\mathbf{H}_t \in \mathbb{R}^{n \times h}$:

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t) \tag{2.15}$$

The output gate $\mathbf{O}_t$ is also a transformation of the cell input. Multiplying it elementwise with the tanh of the cell state assures that the cell output $\mathbf{H}_t$ has elements in the range of $(-1, 1)$. Like the cell state $\mathbf{C}_t$, the cell output $\mathbf{H}_t$ is also passed back to the same cell recurrently, but is also passed to other cells/layers as input. Figure 2.7 gives a better view of the flow of hidden and cell states when multiple LSTM cells are chained.

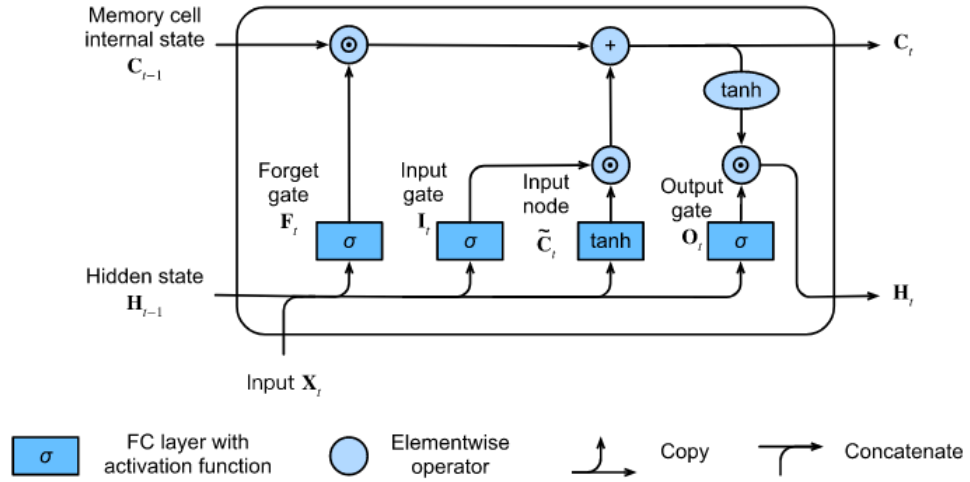


Figure 2.6: The complete form of a single LSTM cell. Note that this only represents a single state - t - of one cell. Source: [Zha+]

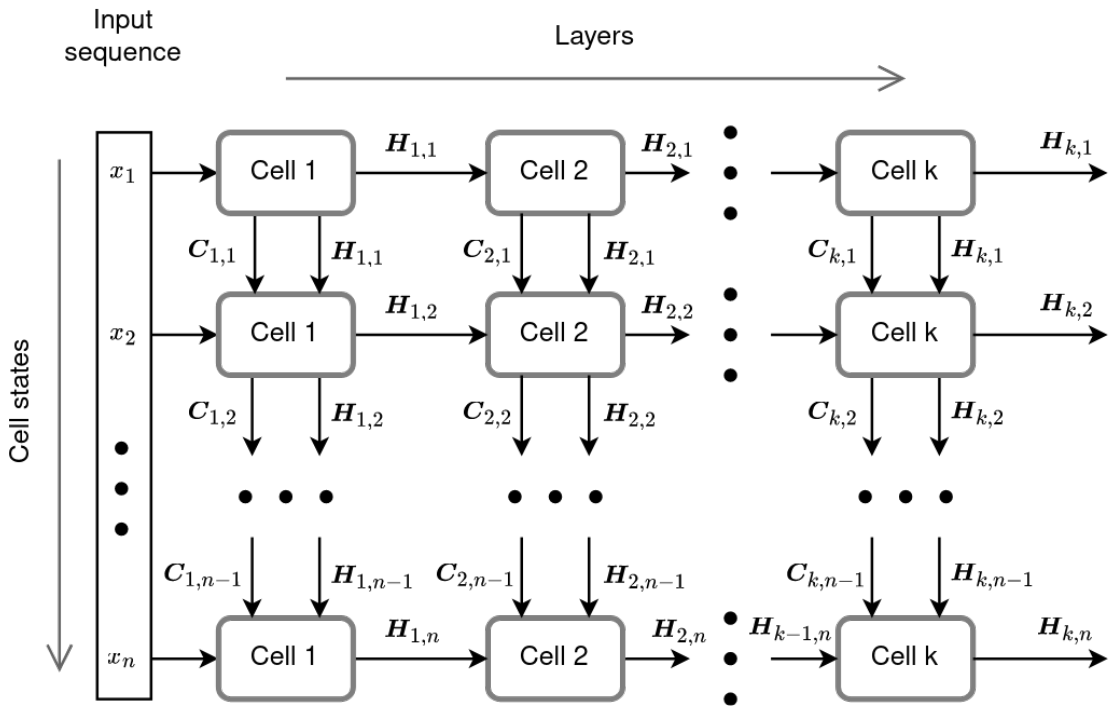


Figure 2.7: Flow of hidden and cell states across steps and layers (LSTM cells). The hidden and cell state suffixes are the cell number and step number respectively. k is the number of LSTM cells that are chained together. Input variables $x_1, ..., x_n$ are the elements of the input sequence, e.g. the letters of a sentence. Source: Author.

Chapter 3

Generative Models

3.1 Inuition

In essence, generative models see data samples as samples from a probability distribution p_{data}. So the goal is, given an observed dataset $\mathcal{D}$, approximate its distribution p_{data}. With this approximation, new samples can be generated.

Let θ be the parameters of the generative model, then we want to tune these parameters such that the model distribution p_θ (the approximate distribution) is as close as possible to the data distribution p_{data}.

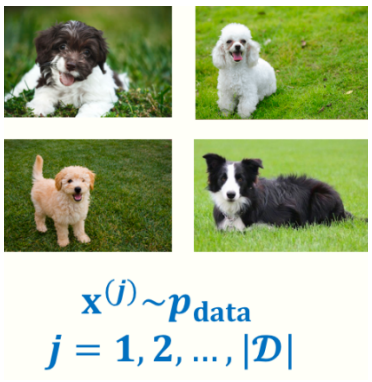


Figure 3.1: Samples from the dataset $\mathcal{D}$. Some unknown complex distribution that describes the dataset is assumed to exist such that its samples are the training samples. Source: [Gro]

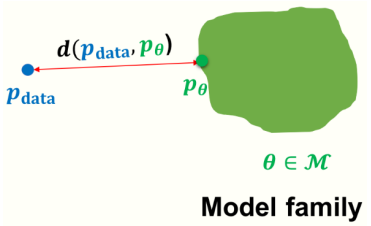


Figure 3.2: Statistical distance between the dataset distribution p_{data} and the approximate distribution p_θ Source: [Gro]

Suppose we have a dataset of dog images, each image being a sample from p_{data}, then we want to find the parameters θ such that the distance between the model distribution p_θ and p_{data} is minimal. Let d be a distance function between probability distributions, then the objective is to solve:

$$\underset{\theta \in \mathcal{M}}{\operatorname{argmin}}\, d(p_{\text{data}}, p_\theta) \tag{3.1}$$

One of the main challenges is that the problem can become intractable due to the huge amount of possible data samples. Considering images with a shape of 700x1400x3, then the amount of all possible pixel combinations is $256^{700\times1400\times3} \approx 10^{800000}$. But thanks to the structured nature of reality, only a fraction of that can occur, meaning a large portion of the 10^{800000} possible combination will just look like noise or abstract art. So the goal is to make the generative model learn these structural properties of the scenes it observes.

Some main questions regarding this problem are:

- What is the representation for the model family $\mathcal{M}$?
- What is the objective function d?
- What is the optimization procedure for minimizing d?

Three fundamental questions for evaluating a generative model are:

1. *Density estimation:* given a training sample $\mathbf{x}$, what is the probability given by the model, e.g. $p_\theta(\mathbf{x})$? In other words, has the generative model learned the training sample $\mathbf{x}$?
    - E.g. the probabilities for dog images must be high, and low for others.
2. *Sampling:* How can we generate novel samples using the model distribution, e.g. $\mathbf{x}_{\text{new}} \sim p_\theta$?
    - E.g. how to generate new dog images?
3. *Unsupervised representation learning:* how can we learn meaningful concise representations for a data point $\mathbf{x}$?
    - E.g. useful features that can be represented can be amount of fur, fur color, dog breed, ...

3.2 Autoregressive models

Finding the dataset's distribution p_{data} entails finding the joint distribution of all random variables x_i. For example, in the case of images, each pixel value would be a random variable. The distribution of the entire image would be the joint distribution of all its pixels.

Let $\mathbf{x}$ be an n-dimensional vector $\mathbf{x} = [x_1, ..., x_n]$ with each element x_i being a random variable. The joint distribution can be expressed with the chain rule of probability as follows:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<i}) \tag{3.2}$$

- With $\mathbf{x}_{<i} = [x_1, ..., x_{i-1}]$

As the name suggest, in the context of autoregressive models, the joint distribution is defined in an autoregressive manner. This essentially means that no assumptions are made that some of the random variables are conditionally independent. In other words, all values from the previous steps are used in predicting the value of the current step. This is shown in the equation above where the probability of sample $\mathbf{x}$ is calculated with the product of the probability of all values conditioned on all previous values.

It is important to note that the order of random variables is fixed, once and ordering is specified it must not change. If we take an image as an example, we can pick the top left pixel as the first random variable x_1 and choose a row-by-row and left-to-right ordering for the remaining random variables (pixels).

In autoregressive generative models, it is assumed that the conditional distributions $p(x_i|\mathbf{x}_{<i})$ correspond with a Bernoulli random variable:

$$p_{\theta_i}(x_i|\mathbf{x}_{<i}) = \text{Bernoulli}(f_i(x_1, ..., x_{i-1})) \tag{3.3}$$

- With $f_i : \{a_1, \ldots, a_{i-1}\} \to [0, 1]$

The function f_i maps the parameters $x_1, ..., x_{i-1}$ to the mean of the Bernoulli distribution. θ_i is the set of parameters used by f_i.

The drawback is that the conditional distributions are constrained to Bernoulli distributions, meaning the data values need to be discrete.

3.2.1 Mapping functions

Here we will mention two ways to implement the function f_i.

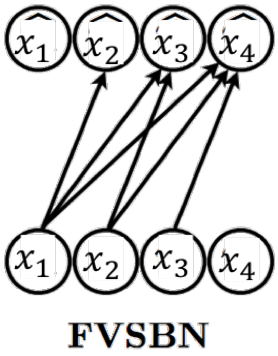


Figure 3.3: A fully visible sigmoid belief network. The variables with the hats are the conditionals. Source: [Gro]

Fully visible sigmoid belief network (FVSBN) The variables with the hats are the conditionals.

The mapping function f_i is simply a linear combination of its inputs:

$$f_i(x_1, ..., x_{i-1}) = \sigma(\alpha_0^{(i)} + \alpha_1^{(i)} x_1 + ... + \alpha_{i-1}^{(i)} x_{i-1}) \tag{3.4}$$

- With $\theta_i = \{\alpha_0^{(i)}, ..., \alpha_{i-1}^{(i)}\}$ be the set of parameters for the i'th conditional

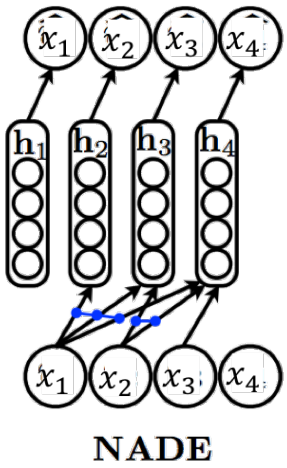


Figure 3.4: A Neural autoregressive density estimator. The variables with the hats are the conditionals. The blue connections denote the tied weights $W[., i]$ used for computing the hidden layer activations. Source: [Gro]

Neural autoregressive density estimator Using a more powerful mapping function will increase the expressive capacity of the generator. NADE uses multi-layer perceptrons (MLP) to map the previous random variables to the mean of the Bernoulli distribution. The mapping function is defined as follows:

$$f_i(x_1, ..., x_{i-1}) = \sigma(\boldsymbol{\alpha}^{(i)}\mathbf{h}_i + b_i) \tag{3.5}$$

$$\mathbf{h}_i = \sigma(\mathbf{W}_{.,<i}\mathbf{x}_{<i} + \mathbf{c}) \tag{3.6}$$

- With $\mathbf{h}_i$ the hidden layer of the i'th conditional
- With $\boldsymbol{\alpha}^{(i)} \in \mathbb{R}^d$ the weights for the activations of the hidden layer
- With $b_i \in \mathbb{R}$ the bias for the activations of the hidden layer
- With $\mathbf{W} \in \mathbb{R}^{d \times n}$ the weight matrix. $\mathbf{W}_{.,<i}$ selects all rows and the first $i-1$ columns
  - d is the amount of perceptrons in the hidden layer
  - n is the number of random variables x
- With $\mathbf{c} \in \mathbb{R}^d$ the bias for the perceptrons in the hidden layer
- With $\sigma(\cdot)$ the sigmoid function for mapping values to the range of $[0, 1]$

All learnable parameters for all mapping functions is the following set:

$$\theta = \{\mathbf{W}, \mathbf{c}, \{\boldsymbol{\alpha}^{(i)}\}_{i=1}^n, \{\mathbf{b}_i\}_{i=1}^n\} \tag{3.7}$$

An improvement of the NADE is RNADE which is the real-valued alternative of NADE. Instead of learning a Bernoulli distribution for each conditional, RNADE models K Gaussian distributions for each conditional distribution. For each conditional p_{θ_i} there is a mapping function $g_i : \mathbb{R}^{i-1} \to \mathbb{R}^{2K}$ which outputs the means and variances for each of the K Gaussians given the previous $i-1$ random variables.

3.2.2 Optimization

A known distance measure between two distributions is the Kullback–Leibler (KL) divergence. The KL divergence is a statistical distance that denotes how different a probability distribution Q is from a reference distribution P. Usually P represents the observed data and Q an approximation to P.

The KL divergence is defined as follows:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)} \tag{3.8}$$

- With $\mathcal{X}$ being the sample space

There are two important things to note:

- The KL-divergence is not symmetrical, meaning $D_{KL}(P||Q) \neq D_{KL}(Q||P)$.
- The range of the KL-divergence is $[0, \infty]$: it is 0 if both distributions are identical, and ∞ in the case that $Q(x) = 0 \wedge P(x) > 0$. In other words: KL-divergence is infinity if the approximated distribution gives a probability of 0 for a sample from the observed dataset.

The optimization objective for a generative model can be expressed with KL-divergence as follows:

$$\underset{\theta}{\operatorname{argmin}} \left[D_{KL}(p_{\mathcal{D}}, p_\theta) = \mathbb{E}_{\mathbf{X} \sim p_{\mathcal{D}}}[\log p_{\mathcal{D}}(\mathbf{x}) - \log p_\theta(\mathbf{x})]\right] \tag{3.9}$$

- With $\mathbb{E}_{\mathbf{X} \sim p_{\mathcal{D}}}[...]$ the expectation with respect to the samples $\mathbf{x}$ from p_{data}

We know that p_{data} is independent from θ, so we can simplify the optimization objective to the following:

$$\underset{\theta}{\operatorname{argmax}} \, \mathbb{E}_{\mathbf{X} \sim p_{\mathcal{D}}}[\log p_\theta(\mathbf{x})] \tag{3.10}$$

We defined p_{data} to be the underlying distribution of the observed dataset $\mathcal{D}$, so we assume that $\mathcal{D}$ is sampled *independent and identically distributed* from p_{data}. This enabled us to obtain an estimate of equation (1) as follows:

$$\underset{\theta}{\text{argmax}} \left[\frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \log p_\theta(\mathbf{x}) = \mathcal{L}(\theta|\mathcal{D}) \right] \tag{3.11}$$

This translates into: pick model parameters θ such that the log probability of samples $\mathbf{x}$ with respect to the model distribution p_θ is maximized.

The optimization can be done with mini-batch gradient ascent. Say we train the model for t steps and at each step t we optimize on a mini-batch B_t, with $B_t \subset \mathcal{D}$. The model parameters at step $t+1$ can then be expressed as follows:

$$\theta^{(t+1)} = \theta^{(t)} + r_t \nabla \mathcal{L}(\theta^{(t)}|B_t) \tag{3.12}$$

- With r_t the learning rate at step t
- Other optimizers such as Adam can also be used for optimizing θ

Finally, we modify the optimization objective to optimize the parameters of the conditionals of autoregressive generative models:

$$\underset{\theta}{\text{argmax}} \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i=1}^{n} \log p_{\theta_i}(x_i|\mathbf{x}_{<i}) \tag{3.13}$$

- With each θ_i being the set of parameters for the i'th conditional.

3.2.3 Inference

Estimating the density of some datasample $\mathbf{x}$ with respect to the model, or in other words the log likelihood assigned by the model to sample $\mathbf{x}$, we simply calculate the log probabilities for all conditionals and add them up:

$$\sum_{i=1}^{n} \log p_{\theta_i}(x_i|\mathbf{x}_{<i}) \tag{3.14}$$

Since $\mathbf{x}$ is known, this can be done in parallel.

Sampling new data from the model must happen sequentially. Lets call the new data sample $\mathbf{y} = \{y_1, ..., y_n\}$. Each value y_i is sampled from the corresponding conditional distribution in an autoregressive manner, meaning y_1 is sampled first followed by y_2 until y_n, sequentially:

$$\begin{aligned} y_1 &\sim p_{\theta_1}(y_1) \\ y_2 &\sim p_{\theta_2}(y_2|y_1) \\ &... \\ y_n &\sim p_{\theta_n}(y_n|\mathbf{y}_{<n}) \end{aligned} \tag{3.15}$$

3.3 Variational Autoencoders

A variational autoencoder (VAE) [KW22] is a modification of the autoencoder where input data is encoded into a distribution rather than a vector. To understand how a VAE works, we will first briefly explain what autoencoders are.

3.3.1 Autoencoders

Autoencoders are a type of neural network with two important components: the encoder and decoder. The encoder serves to reduce the amount of dimensions of the input data. In other words it encodes the input data to a vector of lower dimensions, this vector is commonly called the latent vector or code.

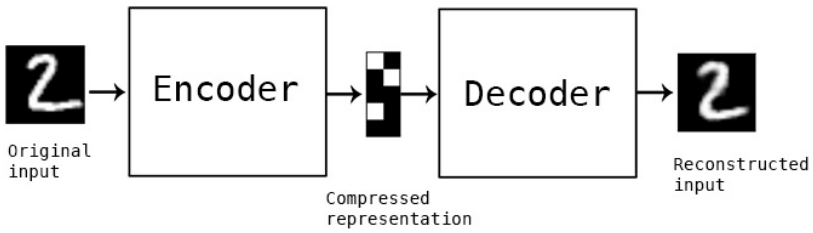


Figure 3.5: An autoencoder network. Source: [Cho]

The decoder on the other hand, serves to do the opposite: reconstruct the original input data, given the latent vector.

Convolutional neural networks are suitable architectures for the encoder and decoder. In the case of the decoder, the network layers would be composed of transposed convolutions to produce outputs that are larger than the inputs. Optimizing an autoencoder is pretty straightforward and happens by minimizing the difference between the decoded output and the original input data (that was passed to the encoder), e.g.:

$$\underset{\theta,\phi}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} ||\mathbf{x}_i - D_\theta(E_\phi(\mathbf{x}_i))||_2^2 \tag{3.16}$$

- With $E(.)$ being the encoder network and ϕ its parameters
- With $D(.)$ the decoder network and θ its parameters
- $||...||_2^2$ denotes the squared 2-norm, e.g. $||x||_2^2 = x_1^2 + x_2^2 + ... + x_k^2$ with k the dimensions of x

Autoencoders are not suitable for generating new data. The reason is that the goal when optimizing an autoencoder is to reconstruct the training data as well as possible from the latent vector. No goal is specified to learn an encoding that produces general latent vectors. This results in input features being encoded into latent vectors which are entangled, meaning there is no clear relation between the elements of the latent vectors and properties of the input feature. Hence, decoding the interpolated versions of these latent vectors may not necessarily produce something resembling the inputs. Variational autoencoders however achieve the required generality in the latent space to generate new data, even when interpolating the latent vector.

3.3.2 VAE intuition

The VAE is similar to a standard autoencoder in that it also has an encoder and decoder network, but instead of encoding input data to a latent vector, it encodes it to a set of distributions. Generating output data from a VAE happens by decoding the a latent vector that is sampled from the latent distributions.

Encoding data to a distribution is not the only factor in learning meaningful representations that can be used for generating new data. A second factor is the regularization of the latent distributions, which happens by adding the KL-divergence of the latent distributions w.r.t. the standard Gaussian distribution. This enforces the encoder to learn encoding data to distributions that are close to the standard Gaussian distribution. This has two benefits which are also visualized by figure 3.6:

- The means of the latent distributions are close to zero, preventing the centers of the distributions to be too far apart from each other;
- The variances of the latent distributions are close to one, preventing data to be encoded into point-like distributions (very low variance) so that interpolating latent vectors is possible to generate meaningful data.

(a) Without regularization **(b)** With regularization

Figure 3.6: The colored areas are the spots where the latent distributions (gaussians) peak and have a non-zero value. The distributions to the left have not been regularized and are thus further apart, this causes meaningless interpolations. The distributions to the right are spread over a wider area and are closer to each other thanks to regularization. Source: [Roc]

3.3.3 VAE definition and optimization

The equations in this section are based on the explanation in **??**. Let $\mathbf{x}$ be the data, and $\mathbf{z} \in \mathbb{R}^k$ the latent variable sampled from the latent distribution. The latent distribution is a multivariate distribution from which high dimensional vectors can be sampled. An example of such a distribution is the multivariate normal distribution which will be used extensively in defining the VAE model.

$$\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{3.17}$$

- $\mathbf{z} \in \mathbb{R}^k$ is the random vector sampled from the multivariate normal distribution
- $\boldsymbol{\mu} \in \mathbb{R}^k$ is the vector of means, one mean for each element in $\mathbf{z}$
- $\boldsymbol{\Sigma} \in \mathbb{R}^{k \times k}$ is the covariance matrix representing the covariance between each pair of elements (which are in turn random variables) of $\mathbf{z}$. If $\boldsymbol{\Sigma}$ is a diagonal matrix, then all random variables of $\mathbf{z}$ are uncorrelated.

As a generative model, the VAE represents the two following steps for the observed data $\mathbf{x}$:

- $\mathbf{z} \sim p(\mathbf{z})$
- $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$

With this we can redefine the deterministic encoder and decoder as probabilistic models. The encoder is the distribution $p(\mathbf{z}|\mathbf{x})$, i.e. the posterior distribution. Conversely, the decoder is the distribution $p(\mathbf{x}|\mathbf{z})$, i.e. the likelihood distribution.

The goal is to find such a latent distribution $p(\mathbf{z})$ - also called the prior distribution - that represents the observed dataset well and which can be used to sample new useful data. In a sense, we want to update the latent distribution with each observation we make, making this a Bayesian inference problem.

The distribution we want to compute is the probability of a latent variable $\mathbf{z}$ given data $\mathbf{x}$. The posterior distribution can be calculated with Bayes' theorem as follows:

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})} = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{\int p(\mathbf{x}|\mathbf{u})p(\mathbf{u})d\mathbf{u}} \tag{3.18}$$

The challenge with this is that the integral in the denominator often becomes intractable due to the curse of dimensionality. A possible solution instead is to compute an approximation with methods such as Markov Chain Monte Carlo (MCMC) [Spe20] and Variational Inference (VI) [JJ00].

Before continuing on to approximate the posterior with VI, the prior $p(\mathbf{z})$ and likelihood $p(\mathbf{x}|\mathbf{z})$ need to be defined. These are defined as normal distributions as follows:

$$\begin{aligned} p(\mathbf{z}) &\equiv \mathcal{N}(0, I) \\ p(\mathbf{x}|\mathbf{z}) &\equiv \mathcal{N}(f(\mathbf{z}), cI), \quad f \in F, \quad c > 0 \end{aligned} \tag{3.19}$$

- With f a well-defined and fixed function from the family of functions F (which is left unspecified for now).
- And cI being the constant c multiplied with the identity matrix.

Let $q_x(\mathbf{z})$ be a normal distribution which will be used to approximate the posterior $p(\mathbf{z}|\mathbf{x})$:

$$q_{\mathbf{x}}(\mathbf{z}) \equiv \mathcal{N}(g(\mathbf{x}), h(\mathbf{x})), \quad g \in G, \quad h \in H \tag{3.20}$$

- With g and h are functions mapping $\mathbf{x}$ to the mean and variance of the normal distribution respectively.
- G and H are also families of functions which are also left unspecified for now.

We approximate the posterior by finding such functions g and h that the KL-divergence of the approximation $q_{\mathbf{x}}(\mathbf{z})$ is minimal from the posterior:

$$(g^*, h^*) = \underset{(g,h)\in G\times H}{\operatorname{argmin}} KL(q_{\mathbf{x}}(\mathbf{z}), p(\mathbf{z}|\mathbf{x})) \tag{3.21}$$

$$= \underset{(g,h)\in G\times H}{\operatorname{argmin}} \left(\mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}(\log q_{\mathbf{x}}(\mathbf{z})) - \mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}\left(\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})}\right)\right) \tag{3.22}$$

$$= \underset{(g,h)\in G\times H}{\operatorname{argmin}} (\mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}(\log q_{\mathbf{x}}(\mathbf{z})) - \mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}(\log p(\mathbf{z})) - \mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}(\log p(\mathbf{x}|\mathbf{z})) + \mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}(\log p(\mathbf{x}))) \tag{3.23}$$

$$= \underset{(g,h)\in G\times H}{\operatorname{argmax}} (\mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}(\log p(\mathbf{x}|\mathbf{z})) - KL(q_{\mathbf{x}}(\mathbf{z}), p(\mathbf{z}))) \tag{3.24}$$

$$= \underset{(g,h)\in G\times H}{\operatorname{argmax}} \left(\mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}\left(-\frac{||\mathbf{x} - f(\mathbf{z})||^2}{2c}\right) - KL(q_{\mathbf{x}}(\mathbf{z}), p(\mathbf{z}))\right) \tag{3.25}$$

The second last equation clearly shows the objectives of this optimization. On the left-hand-side we have the positive expectation of the likelihood, which means the likelihood of generating data $\mathbf{x}$ given latent $\mathbf{z}$ will be maximized. On the right-hand-side we have the KL-divergence of the approximation $q_{\mathbf{x}}(\mathbf{z})$ from the prior $p(\mathbf{z})$, subtracting this from the likelihood means that we want to minimize the divergence. So a trade-off needs to be found during the optimization to maximize the likelihood while keeping the KL-divergence as low as possible. Since the prior $p(\mathbf{z})$ was defined as the standard normal distribution, the approximated distribution will be similar to it adding the benefit of regularization. This means that the means of the latent distributions will be close to 0, making them all close to each other, and their variances close to 1, allowing the interpolation of latent vectors to generate meaningful new data.

Up until this point, the function f was assumed to be known and fixed. This is not the case in reality and must be chosen as well. Since the prior definition is fixed, the only two parameters influencing the model's encoding-decoding performance are the function f and the constant c of the likelihood $p(\mathbf{x}|\mathbf{z})$.

Equation 3.25 gives us the optimal parameters (g^*, h^*) for approximating the posterior $p(\mathbf{z}|\mathbf{x})$ which is the encoder of the VAE. So, the optimal function f^* is one that results in the optimal decoder (remember that f is a parameter of the likelihood/decoder, therefore it directly influences the decoding performance). The optimal decoder is one that can reconstruct the observed data as well as possible from a given encoder (e.g. the posterior $p(\mathbf{z}|\mathbf{x})$).

Concretely, the optimal function f^* is a function $f \in F$ which yields the highest likelihood:

$$\begin{aligned} f^* &= \underset{f\in F}{\operatorname{argmax}} \mathbb{E}_{\mathbf{z}\sim q^*_{\mathbf{x}}}(\log p(\mathbf{x}|\mathbf{z})) \\ &= \underset{f\in F}{\operatorname{argmax}} \mathbb{E}_{\mathbf{z}\sim q^*_{\mathbf{x}}}\left(-\frac{||\mathbf{x} - f(\mathbf{z})||^2}{2c}\right) \end{aligned} \tag{3.26}$$

- With $q^*_{\mathbf{x}}$ being the approximation of the posterior given a function f (see equation 3.21)

The objective of this optimization is, in other words, maximizing the probability that the observed data (training data) will be generated by the decoder. More specifically, maximizing the probability that $\mathbf{x}' = \mathbf{x}$ where $\mathbf{x}$ is training data and $\mathbf{x}'$ is the decoder's attempt to reconstruct $\mathbf{x}$ with the given encoder (approximation) $q^*_{\mathbf{x}}$.

- With $\mathbf{x}' \sim p(\mathbf{x}|\mathbf{z}')$ and $\mathbf{z}' \sim q^*_{\mathbf{x}}$

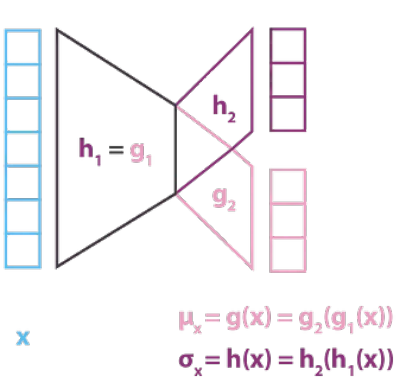
Notice that the final form of equation 3.26 is present in equation 3.25. This allows us to modify equation 3.25 to also compute the optimal f:

$$(f^*, g^*, h^*) = \underset{(f,g,h)\in F\times G\times H}{\operatorname{argmax}} \mathbb{E}_{\mathbf{z}\sim q_{\mathbf{x}}}\left(-\frac{||\mathbf{x} - f(\mathbf{z})||^2}{2c}\right) - KL(q_{\mathbf{x}}(\mathbf{z}), p(\mathbf{z})) \tag{3.27}$$

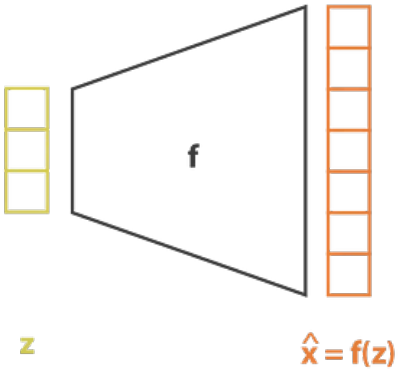
An important question that is left unanswered is what the function families F, G and H are and how the VAE can be expressed as a neural network. These will be explained in the next section.

3.3.4 VAE architecture

Since it would be unpractical to optimize over the whole range of the functions f, g and h, these are expressed as neural networks and optimized accordingly. Thus, F, G and H represent the families of functions defined by neural networks. The encoder part of the VAE is shown in figure 3.7a:



(a) The encoder of the VAE. Source: [Roc]



(b) The decoder of the VAE. Source: [Roc]

Figure 3.7: The VAE encoder and decoder schemas.

As we can see, h and g share the first half of the encoder then split up to give the means and correlations of the latent distribution. We can see that g and h have an equal number of output features, while h is representing the covariance matrix. If k is the dimension of the latent variable z then the amount of output features of h should be k^2 (and k for g). The amount of output features of h can be reduced to k if the distribution approximating the encoder (posterior) is constrained to be defined by a diagonal covariance matrix, so the only relevant elements would the k elements on the diagonal. This speeds up training and reduces the amount of features with the drawback of reducing the range of distributions that can be used in approximating the posterior resulting in less accurate posterior estimations.

The decoder that models the likelihood of $\mathbf{x}$ has a fixed covariance matrix cI as defined in the likelihood equation and it's only learnable part is the neural network f whose weights can be optimized with gradient-based approaches. An important thing to note is the input $\mathbf{z}$, it is not a distribution but a *vector* that has been *sampled* from the latent distribution as defined by the output features of the encoder.

To get the full VAE network, we have to concatenate the encoder and decoder networks. However, concatenating the networks as is will cause problems for backpropagation due to the "sampling" layer between the encoder and decoder. In order for backpropagation to work, we need the layers to represent deterministic functions whose gradients can be calculated. A sampling layer is stochastic and there is no suitable notion of gradients for it. The *reparametrization trick* can be used to remove the stochastic layer from the path along which gradients flow. This trick is illustrated clearly in figure 3.8.

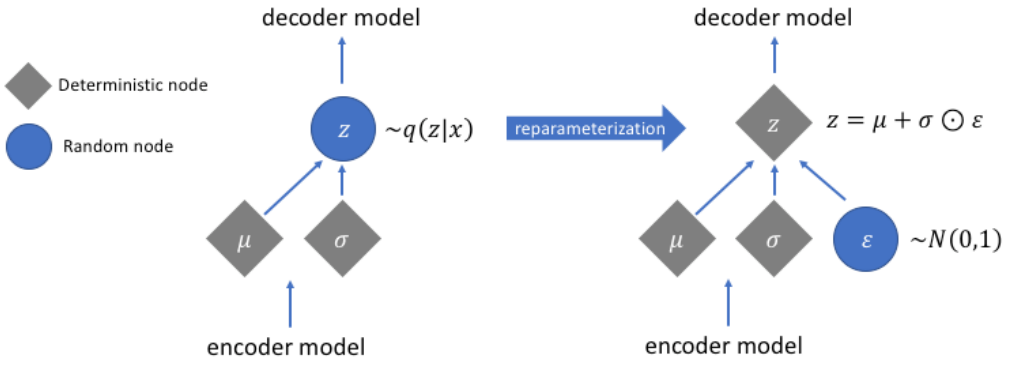


Figure 3.8: The reparametrization trick. Source: [Hos]

Instead of sampling the latent vector z in a layer between the encoder and decoder, it is computed with the formula $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, with $\boldsymbol{\epsilon} \in \mathbb{R}^k$ being sampled separately from the standard normal distribution and multiplied element-wise with $\boldsymbol{\sigma}$. It's clearly seen that the stochastic layer is now outside of the path along which gradients will flow.

The final version of the VAE network can be seen in figure 3.9:

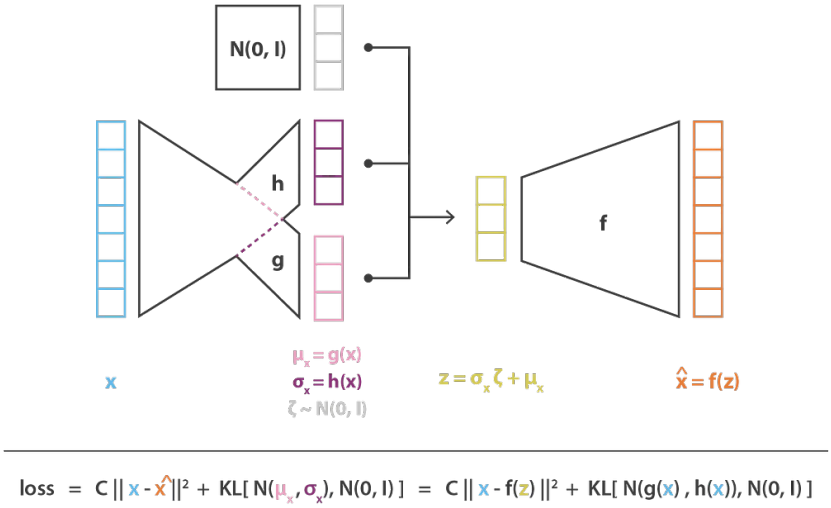


Figure 3.9: The full VAE network. Source: [Roc]

3.4 Deep Recurrent Attentive Writer

A Deep Recurrent Attentive Writer (DRAW) model [Gre+15] is a generative model capable of generating images in multiple steps instead of whole images in a single step. At each step, the image is refined and details are added until the last step. The initial steps focus on general visuals (e.g. rough shapes of the dominant objects in the scene) while the later steps focus on adding details (e.g. lines, shading, ...).

The DRAW model is a type of VAE, but with a recurrent encoder and decoder to make it possible to generate the image in multiple steps. Also, an attention mechanism is used to restrict which portion of the input image the model can read and on which portion of the output canvas the model can write.

3.4.1 DRAW definition

Let RNN_{enc} and RNN_{dec} be the encoder and decoder RNNs with h_t^e and h_t^d their hidden states at time step t respectively. Let $Q(Z_t|h_t^e)$ be the posterior distribution. Let $t \in \{1, ..., T\}$ with t the current time

step and T the total number of recurrent steps which is a hyperparameter of the DRAW model. Notice that the letter superscripts are used for identification and not as powers for simplicity unless stated otherwise. On the other hand, numerical superscripts are interpreted as powers.

Intuitively, the DRAW model will execute a fixed set of operations to generate an update for each step t. Each update will be aggregated in a tensor c which we will call the "canvas", with c_t being the canvas state at step t. Since the DRAW model is a type of variational autoencoder, the encoder model will be used during the training phase to compute the approximate posterior, which in turn will be used to sample a latent vector z. The latent vector z_t can be interpreted as the summary of what kind of update should be generated at step t by the decoder.

Hence, a single time-step t of the generation process during the training phase can be expressed with the following equations:

$$\hat{x} = x - \boldsymbol{\sigma}(c_{t-1}) \tag{3.28}$$
$$r_t = read(x_t, \hat{x}_t, h^d_{t-1}) \tag{3.29}$$
$$h^e_t = RNN_{enc}(h^e_{t-1}, [r_t, h^d_{t-1}]) \tag{3.30}$$
$$z_t \sim Q(Z_t|h^e_t) \tag{3.31}$$
$$h^d_t = RNN_{dec}(h^d_{t-1}, z_t) \tag{3.32}$$
$$c_t = c_{t-1} + write(h^d_t) \tag{3.33}$$

After the recurrent updates are done, the canvas is used as the parameter of a probability distribution to sample the generated image. $\boldsymbol{\sigma}$ is the sigmoid function which is used to compute $\hat{x}_t$, the error of the generated image so far. The *read* and *write* functions control which portion of the inputs will be considered and which portion of the canvas will be updated in the current time step respectively. There is no fixed model for the encoder and decoder RNNs, various architectures can be used such as the LSTM. A suitable candidate for the posterior is a Gaussian distribution, i.e. $Q(Z_t|h^e_t) = \mathcal{N}(Z_t|\mu_t, \sigma_t)$ with $\mu_t = W(h^e_t)$ and $\sigma_t = \exp(W(h^e_t))$ where W is a weight matrix (convolutional layers could be used as alternatives if the shape of the tensor h^e_t allows it). The number of trainable parameters can be reduced if the covariance matrix of the Gaussian is constrained to be a diagonal matrix.

As stated earlier, the canvas matrix of the final step c_T is used to condition a distribution $D(X|c_T)$ for the to-be-generated image. If the input data is binary then a natural choice is the Bernoulli distribution with its means computed with the sigmoid function $\boldsymbol{\sigma}(c_T)$. If the input data is an image scaled down to the range of $[0, 1]$, then a continuous distribution like a Gaussian is fitting with its parameters calculated by passing the canvas through a fully-connected layer like that of the posterior distribution:

1. $\mu_T = W_1(c_T)$
2. $\sigma_T = \boldsymbol{\sigma}(W_2(c_T))$

3.4.2 Optimization

The network is optimized by minimizing the total loss $\mathcal{L}$, which is defined as the sum of the reconstruction loss $\mathcal{L}^x$ and the latent loss $\mathcal{L}^z$:

$$\mathcal{L}^x = -\log D(x|c_T) \tag{3.34}$$
$$\mathcal{L}^z = \sum_{t=1}^{T} KL(Q(Z_t|h^e_t)||P(Z_t)) \tag{3.35}$$

The reconstruction loss $\mathcal{L}^x$ is the negative log-likelihood of generating x given c_T. The goal is to maximize the log-likelihood of generating x which is equivalent to minimizing the negative log-likelihood. Adding both losses yields us the total loss $\mathcal{L}$:

$$\mathcal{L} = -\log D(x|c_T) + \sum_{t=1}^{T} KL(Q(Z_t|h^e_t)||P(Z_t)) \tag{3.36}$$

This loss function has been previously derived in section 3.3 about variational autoencoders at equation 3.24. Minimizing $\mathcal{L}$ is equivalent to maximizing $-\mathcal{L}$ which is directly derived from the KL-divergence between the approximate posterior and the true posterior (see equations 3.21-3.25).

If the latent distribution $Q(Z_t|h_t^e)$ (i.e. approximate posterior) has a diagonal covariance matrix, then the prior $P(Z_t)$ can be a standard Gaussian with a mean of zero and a standard deviation of one. This has the benefit of learning a posterior that is close to the standard Gaussian distribution which makes it possible to generate useful data when interpolating in the latent space as mentioned in section 3.3.2.

The optimization is carried out with stochastic gradient descent using one sample z for each training step:

$$\mathcal{L} = \langle \mathcal{L}^x + \mathcal{L}^z \rangle_{z \sim Q} \tag{3.37}$$

3.4.3 Generating new images

Just like in the training phase, the data generation process is carried out in T steps with $t = 1, \ldots, T$ the current step. The encoder is left out of the equation and the latent vector $\tilde{z}_t$ is sampled directly from the prior:

$$\tilde{z}_t \sim P(Z_t) \tag{3.38}$$
$$\tilde{h}_t^d = RNN_{dec}(\tilde{h}_{t-1}^d, \tilde{z}_t) \tag{3.39}$$
$$\tilde{c}_t = \tilde{c}_{t-1} + write(\tilde{h}_t^d) \tag{3.40}$$
$$\tilde{x} \sim D(X|\tilde{c}_T) \tag{3.41}$$

After all T steps have been carried out, the final image is sampled from a distribution conditioned on the canvas matrix $\tilde{c}_T$. Naturally, the distribution D must be the same type of distribution as the one used during the training phase.

3.4.4 The attention mechanism

The attention mechanism defines the *read* and *write* operations in the equations. Attention determines which part of some canvas will be accessible, i.e. *read* attention for determining which part of the input will be read and *write* attention for determining which part of the output canvas will be manipulated. Two different definitions for the *read* and *write* operations, the first without attention and the second with attention.

Reading and writing without attention

Using no attention means that the entire input image will be read by the encoder and the entire canvas will be affected by the decoder at each step. In this case, the read and write operations can be defined as follows:

$$read(x, \hat{x}_t, h_{t-1}^d) = [x, \hat{x}_t] \tag{3.42}$$
$$write(h_t^d) = W(h_t^d) \tag{3.43}$$

Selective Attention

The attention mechanism selects an $N \times N$ patch with a varying degree of range and sharpness. To extract an $N \times N$ patch, an $N \times N$ grid of Gaussian filters is superimposed on the image, which is shown in figure 3.10a. The stride between the filters, δ, determines the range of the image that will be captured in the patch. Meaning, the more widespread the filters are, the larger the area that is captured in the $N \times N$ patch. The variance of the Gaussians determines the sharpness of the visuals that are captured in the patch.

There are three pairs of images on figure 3.10b. The left image of the pair shows the input image along with a green rectangle representing the attention window. The attention window is just another visualization of the grid of filters (figure 3.10a) displayed for convenience. The size of the rectangle corresponds with the the area that will be captured on the patch while its thickness corresponds with the variance of the Gaussians. The higher the variance, the more blurry the captured visuals will look.

(a) (b)

Figure 3.10: DRAW attention mechanism. **(a)** 3x3 grid of filters on the input image. δ is the stride and (g_X, g_Y) is the center of the grid. **(b)** Images on the left show the original image along with the attention window that is formed with the grid of filters. Images on the right show the actual patch that has been captured by the attention window. Thick lines mean high variance which causes blurry patches. Conversely, thin lines mean low variance causing crisp patches. Source images: [Gre+15]

The effects of applying Gaussians with low variance versus high variance can be seen by comparing the second and third pairs of images. The second one has low variance filters causing the captured patch to look sharp while the third one has high variance, blurring the visuals substantially.

Let (g_X, g_Y) be the centre of the grid and δ the stride between filters. The dots in figure 3.10a represent the position of the Gaussians which can be specified with their means. The mean location (μ_X^i, μ_Y^j) of the filter at row i and column j in the $N \times N$ patch is defined as follows:

$$\mu_X^i = g_X + (i - N/2 - 0.5)\delta \tag{3.44}$$

$$\mu_Y^j = g_Y + (j - N/2 - 0.5)\delta \tag{3.45}$$

The attention parameters g_X, g_Y and δ along with the variance of the Gaussians σ^2 and filter response strength γ are computed at each time step by a linear transformation W of the decoder output h^d:

$$(\tilde{g}_X, \tilde{g}_Y, \log\sigma^2, \log\tilde{\delta}, \log\gamma) = W(h^d) \tag{3.46}$$

$$g_X = \frac{A+1}{2}(\tilde{g}_X + 1) \tag{3.47}$$

$$g_Y = \frac{B+1}{2}(\tilde{g}_Y + 1) \tag{3.48}$$

$$\delta = \frac{\max(A,B)-1}{N-1}\tilde{\delta} \tag{3.49}$$

With these parameters, the horizontal and vertical filterbank matrices $F_X \in \mathbb{R}^{N\times A}$ and $F_Y \in \mathbb{R}^{N\times B}$ can be defined as follows:

$$F_X[i,a] = \frac{1}{Z_X}\exp\left(-\frac{(a-\mu_X^i)^2}{2\sigma^2}\right) \tag{3.50}$$

$$F_Y[j,b] = \frac{1}{Z_Y}\exp\left(-\frac{(b-\mu_Y^j)^2}{2\sigma^2}\right) \tag{3.51}$$

Where:

- $F_X[i,a]$ is the element of F_X at row i and column a
- (i,j) is a point in the attention patch at row i and column j
- (a,b) is a point in the input image at row a and column b
- Z_x and Z_y are normalization constants such that each row in F_X and F_Y add up to one

It is worth noting that each row of the filterbank matrices F_X and F_Y corresponds with the evaluation of a Gaussian function with mean μ_X^i and μ_Y^i respectively.

We can now define the *read* and *write* operations as follows:

$$read(x, \hat{x}_t, h_{t-1}^d) = \gamma[F_X x F_Y^T, F_X \hat{x} F_Y^T] \tag{3.52}$$

$$w_t = W(h_t^d) \tag{3.53}$$

$$write(h_t^d) = \frac{1}{\hat{\gamma}} \hat{F}_X^T w_t \hat{F}_Y \tag{3.54}$$

Where

- F_X, F_Y and γ are extracted from h_{t-1}^d
- $\hat{F}_X$, $\hat{F}_Y$ and $\hat{\gamma}$ are a second set of parameters extracted from h_t^d
- w_t is an $N \times N$ writing patch

In the case of RGB images, the same read and write filters are used for the three channels.

3.4.5 Convolutional DRAW

The Convolutional DRAW model is a modification of the standard DRAW model proposed by [Gre+16]. The difference is that it doesn't use an attention mechanism and the encoder and decoder LSTMs are implemented with convolutional layers instead of linear layers. It has been reported to perform better than standard DRAW.

3.5 Generative Adversarial Networks

Generative Adversarial Networks (GAN) consist of two main components, i.e. the discriminator D and the generator G. The discriminator's job is to tell whether a given sample x is sampled from the given dataset. For the ideal discriminator $D(x) = 1$ would hold if x is a real sample from the dataset and $D(x) = 0$ if x is generated by G. The goal is to find a generator that produces samples that are like those in the training dataset. To achieve this, the discriminator and generator are pitted against each other where the generator tries to fool the discriminator into thinking that the generated samples are real ones sampled from the training dataset.

The loss function for optimizing the GAN looks as follows:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

To make the explanation simpler, we can divide the loss function's equation into two separate parts:

$$\max_D V(D) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{3.55}$$

$$\min_G V(G) = \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{3.56}$$

Eq. 1 is the loss function for optimizing the discriminator where the left-hand side of the addition serves to make the discriminator better at recognizing real samples and the right-hand side for recognizing "fake", generated images better. Eq. 2 is the loss function for optimizing the generator, it tells the generator how to become better at fooling the discriminator.

The training of a GAN is carried out by switching between the discriminator and generator at each step. At one step, the parameters of G are locked and the parameters of D are optimized given a real sample from the dataset and one generated by the fixed G. And at the other step, the parameters of D are fixed and G is optimized.

Some of the applications of GANs are:

- Synthesizing novel images (StyleGAN [Kar+20])

- Transforming the style of an image (CycleGAN [Zhu+20]), this could be the style of an object in the scene (e.g. converting horses to zebras) or the whole scene itself (converting a real-life picture to a Van Gogh style painting)
- Image super-resolution (SRGAN [Led+17]) to upscale the resolution of images.
- Converting semantic maps to realistic images (GauGAN [Par+19])
- Image inpainting (Context encoder [Pat+16])
- Text-to-image synthesis (GigaGAN [Kan+23])
- And more...

The advantages of GANs are:

- Ability to generate high-resolution images (StyleGAN)
- Ability to generate a wide variety of data (if mode collapse can be prevented)
- Suitable for unsupervised learning where no labels are available for training samples
- GANs can be made conditional (e.g. GigaGAN, [MO14])

The disadvantages of GANs are [Dew+21]:

- GANs are hard to train due to the competitive nature of the discriminator and generator which may cause the following difficulties [Hui23]:
  - Oscillation of gradients preventing them from converging;
  - The generators gradients may diminish if the discriminator becomes too successful and prevent the generator from optimizing
  - Mode collapse: this occurs when the generator tends to stick with generating a small set of samples instead of generating a wide variety;
- A large dataset required to properly train a GAN
- Highly sensitive to hyperparameters: we don't want either the generator or discriminator to overpower the other affecting their gradients, their competence needs to be balanced so both can be trained effectively. [PAB]

We note that new methods for training GANs have been developed to address the training instability issues such as the Wasserstein GAN [ACB17] in trade for training speed.

3.6 Diffusion Models

Diffusion models [HJA20] are generative models that iteratively remove noise from a Gaussian sample to obtain an image resembling those in the training dataset. The diffusion process is composed of two processes, i.e. the forward process where noise is gradually added in T steps to an image $\mathbf{x}$ and the reverse diffusion process where the model learns to revert the noise that was added to the image.

The equations will be based on the explanations provided in [Ser22]. Let $\mathbf{x}_0$ be the original image and $\mathbf{x}_T$ the same image with Gaussian noise added to it T times, which should ideally look like a sample from the Gaussian distribution. The forward process can be defined as a Markov chain of T steps where each state of the image $\mathbf{x}_t$ depends on the previous state $\mathbf{x}_{t-1}$:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{3.57}$$

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{3.58}$$

With β_t the variance of the Gaussian noise is added to the image at step t. The variance β_t could be fixed or scheduled over T steps. The authors of Denoising Diffusion Probabilistic Models (DDPM) [HJA20] have chosen a linear schedule increasing from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$ while others have shown that the diffusion performance can be increased by using a cosine schedule [ND21].

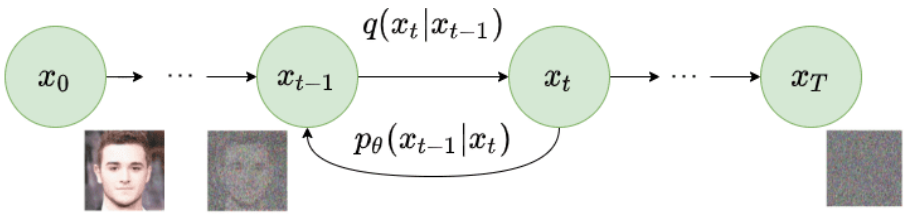


Figure 3.11: The forward and reverse diffusion processes. Source: [Ser22]

The difficult part is reversing the forward process, i.e. learning a distribution $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$. If this is achieved, images can be sampled by starting from a random sample $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and iteratively removing the noise until $\mathbf{x}_0$ is obtained, which is the "generated" image.

The distribution for the denoising process $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is approximated by another Gaussian p_θ whose parameters (mean and variance) are learned by neural networks:

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{3.59}$$

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)) \tag{3.60}$$

The forward process is a fixed process, there are no parameters that need to be optimized for it. It is the reverse diffusion process that needs to be approximated using neural networks. The derivation of the loss function is out of the scope of this thesis, we refer interested readers to [Wen21] for a detailed explanation. The loss function is defined as follows ([Wen21]):

$$\begin{aligned} L_{\text{VLB}} &= L_T + L_{T-1} + \cdots + L_0 \\ \text{where } L_T &= D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p_\theta(\mathbf{x}_T)) \\ L_t &= D_{\text{KL}}(q(\mathbf{x}_t|\mathbf{x}_{t+1}, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_t|\mathbf{x}_{t+1})) \text{ for } 1 \leq t \leq T-1 \\ L_0 &= -\log p_\theta(\mathbf{x}_0|\mathbf{x}_1) \end{aligned} \tag{3.61}$$

Where

- L_{VLB} is the total loss, i.e. the variational lower bound of the likelihood for $\mathbf{x}_0$ to be generated, i.e. $\log p_\theta(\mathbf{x}_0)$
- L_T has no trainable parameters and $\mathbf{x}_T$ is Gaussian noise, so it can be ignored during training
- L_0 can be seen as the reconstruction loss, similar to the likelihood term in the loss function of a VAE model
- L_t indicates the difference between the desired reverse diffusion steps $q(\mathbf{x}_t|\mathbf{x}_{t+1}, \mathbf{x}_0)$ and the approximated ones $p_\theta(\mathbf{x}_t|\mathbf{x}_{t+1})$

The type of neural network used in DDPM [HJA20] is a U-Net [RFB15b] which got its name from its U-shaped diagram which can be seen in figure 3.12. The layers in the first half of the U-Net reduce the resolution of the input to a small size and the layers in the second half scale it back up to the desired output resolution. Residual connections (gray arrows in figure 3.12) have been added to encourage the reuse of the features that were computed by the first half of the network and to prevent gradients from vanishing.

Some of the applications of diffusion models are:

- Unconditional Image generation (Stable Diffusion [Rom+22])
- Text-to-image (DALL-E2 [Ram+22], [Rom+22])
- Inpainting ([Lug+22])
- Image super-resolution ([Niu+23], [Rom+22])
- Video generation ([Ho+22])

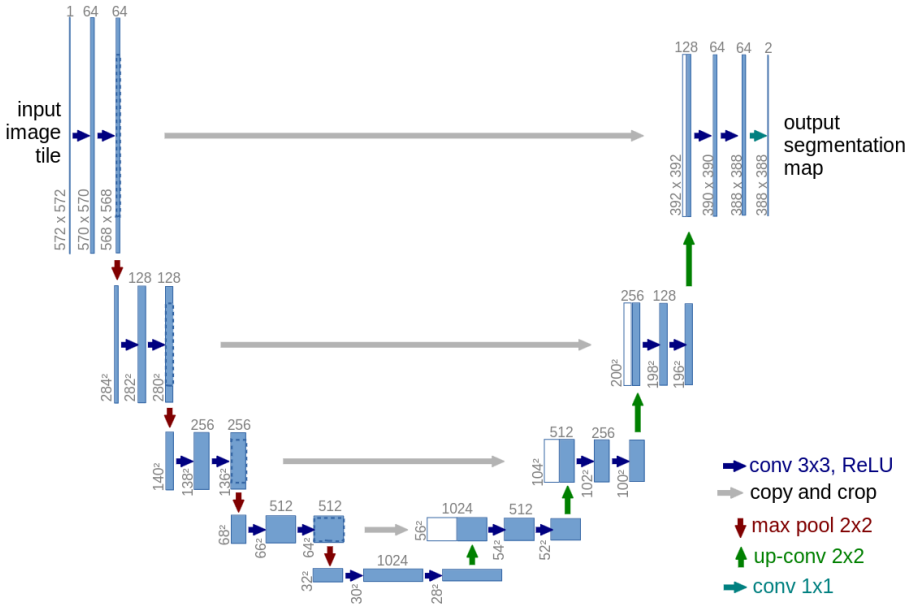


Figure 3.12: The U-Net architecture. Source: [RFB15b]

- Denoising ([HJA20])

The advantages of diffusion models are:

- Ability to generate high-resolution images better than GANs, as claimed by [DN21]
- Is able to learn a wide range of the dataset enabling it to generate a wide variety of images, it doesn't have the mode-collapse problem like GANs
- Easy to train thanks to the tractable loss function ([Wen21])

The disadvantages of diffusion models are:

- Slow sampling: "It takes around 20 hours to sample 50k images of size 32×32 from a DDPM, but less than a minute to do so from a GAN on an Nvidia 2080 Ti GPU." as stated by [SME22]
- The well-known diffusion models such as DDPM [HJA20] have a latent vector whose size is fixed and is as large as the input images. Alternative diffusion models have been developed which can have a latent vector with a different size than the input images, such as Vector Quantized diffusion models [Gu+22].

A lot of research has been made in recent years to improve diffusion models, which are summarized in the following survey: [Cao+23].

Chapter 4

Generative Query Networks

A Generative Query Network (GQN) is a model that can learn to represent a scene with just the observed images and their viewpoints. The goal is to create an artificial system that learns to model data (scene observations) to scene representations without domain knowledge, like object classes, their locations, scene types and so on.

A GQN is composed of two parts: a representation network and a generation network. The representation network converts the scene observations to a representation vector. Each observation consists of an observed image $\mathbf{x}$ and the extrinsic camera parameters such as camera position and orientation $\mathbf{v}$ through which the image has been observed. Let o_i be the set of observations of the i'th scene with K the amount of observations:

$$o_i = \{(\mathbf{x}_i^k, \mathbf{v}_i^k)\}_{k=i...K}$$

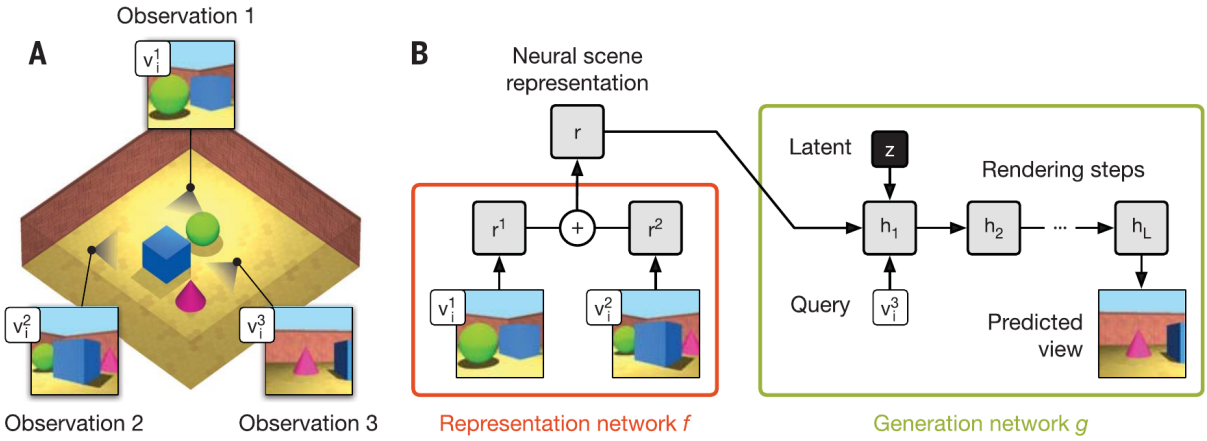


Figure 4.1: High-level overview of the GQN. A) Scene observations from viewpoints (v_i^k) B) First and second observations are passed as context observations to the representation network (RN). The RN computes representation vectors for individual observations. The scene representation r is the sum of individual observation representations. r is passed along with a query viewpoint to the generator network to generate an image predicting how the scene will look from the query viewpoint. Source: [Esl+18]

For each observation the agent makes, the representation network f generates an observation representation vector $\mathbf{r}^k$. The scene representation vector $\mathbf{r}$ is the sum of all observation representation vectors:

$$\mathbf{r}_i = \sum_{k=0}^{K} \mathbf{r}_i^k$$

The idea is that with each observation, $\mathbf{r}$ is updated with new information about the scene and gets more accurate with each observation.

The Generator Network g takes $\mathbf{r}$ and a query viewpoint $\mathbf{v}^q$ as input to generate an image predicting how the scene will look from $\mathbf{v}^q$. Stochastic latent variables $\mathbf{z}$ are used to generate variability where necessary, like spots in the scene that were occluded in the observations.

Both networks f and g are trained in an end-to-end fashion to maximize the likelihood of generating ground-truth images from arbitrary viewpoints. The representation network f is unaware of the query viewpoints that are asked of the generator network g. So when optimizing the networks, f learns to encode as much information as necessary for g to generate accurate images. This enables the GQN to learn by itself what information to encode and how to encode it in the representation vector.

Statistical regularities of the training scenes (typical colours of the sky, shapes, symmetries, patterns, textures, ...) are learned and internalized by the generator network. This allows for the capacity of the scene representation to be reserved for general descriptions of the scene. The generator network will know what the bits of information mean in the scene representation and will generate the necessary details in the resulting image. Consider a robot arm for example. Instead of the object's surface details, the representation network might encode information about the joints and the dominant colours. The generator network will then know how to generate the full robot arm from these encoded joint and colour descriptions.

Filling Occlusions If the GQN receives a set of scene observations where one spot of the scene is occluded and is asked to generate an image of the occluded spot, the GQN will be able to fill in the occluded part of the scene with reasonable visuals. In figure 4.2 we can see the four images on the right that have been sampled from the GQN's generator given the observation on the left. For each of the four samples, the GQN has filled the occluded part of the scene with objects that it has seen during the training phase. The knowledge it has gathered during the training phase is encoded in the prior distribution of the GQN's generator. When generating an image, the generator will sample a latent vector $\mathbf{z}$ from the prior which will then directly influence what gets rendered in the occluded spot. So the actual visuals that get rendered depend on the values that have been sampled from the prior distribution, hence the reason why different visuals are rendered each time an image is generated. See section 4.1.2 for more details about the generator.

It must be noted that each time the same viewpoint is sampled, a different image will be generated. A modification has been proposed in [Kum+19] to enable sampling images consistently from the GQN, i.e. when sampling a viewpoint with uncertainties (e.g. occlusions) the generated image will consistently be the same. This is especially useful for generating frames for a video sequence. The consistent GQN constructs the representation of a set of frames (that don't necessarily occur in the given order) and generates images for arbitrary points in time. For example, it is capable of generating a frame that occurs much later in the video without having to generate the preceding frames, as reported in [Kum+19].

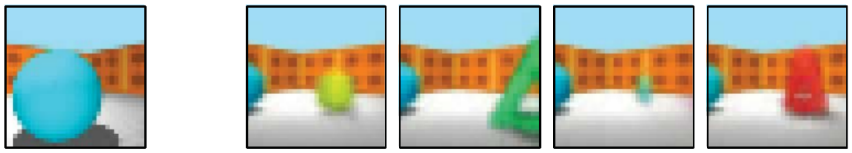


Figure 4.2: The GQN generates reasonable images of the occluded spot. The image on the left is the observation, the four images on the right are different samples. Source: [Esl+18]

Compositionality The GQN is able to learn compositional representations to some extend. This means that it is able to differentiate between scene properties (such as object types, colors and positions) and encodes them separately in the representation vector. By extension, it is also able to apply these properties to objects that haven't been observed with those properties before. Figure 4.3 shows the results of an experiment where the GQN was trained on spheres of various colors, excluding red. It was Also trained with different shapes that were all red. The GQN was asked to generate an image of a red sphere and was able to do so 50% to 70% of the time. In other cases it generated a red cylinder as reported in [Esl+18].

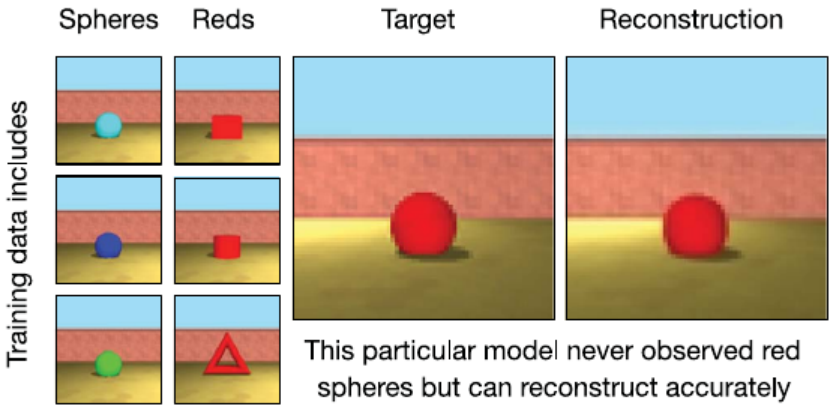


Figure 4.3: GQN compositionality experiment. Source: [Esl+18]

Scene Algebra Scene algebra can be performed on the representation vectors to manipulate the generated images. Figure 4.4 shows how object properties of one scene can be removed by subtracting its representation with another scene that has a similar object with similar properties. This shows that properties can be removed with subtraction and added with addition.

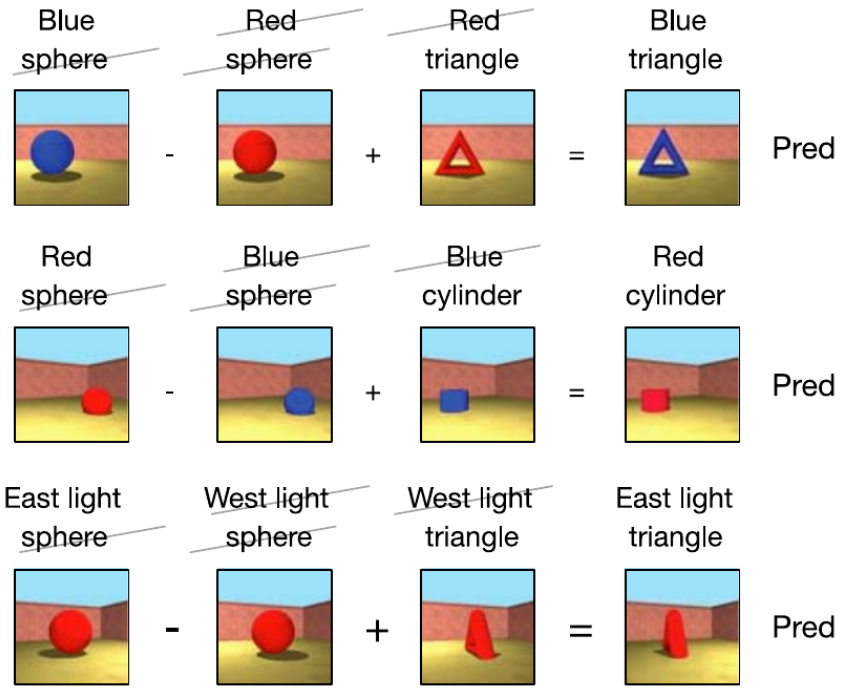


Figure 4.4: Scene algebra: scene properties can be controlled by adding and subtracting representation vectors. The properties that have been crossed indicate that they have been cancelled out. For example, in the first row, one representation is subtracted while the other is added, cancelling out all common properties: "red" in this case. Source: [Esl+18]

Similarity of Representation Vectors Representation vectors of observations of the same scene seem to have a high t-SNE similarity. t-SNE, otherwise known as t-distributed stochastic neighbour embedding, is a nonlinear dimensionality reduction technique with the aim of mapping similar high-dimensional objects to nearby points and dissimilar ones to distant points with high probability.

In figure 4.5, we see t-SNE embeddings of representation vectors produced by the representation network of the GQN and by a VAE model. Each color represents a scene and each dot corresponds with the representation vector of an observation. By observing GQN's t-SNE values, we can see that the embeddings of observations belonging to the same scene are clustered together. The images with the dagger and

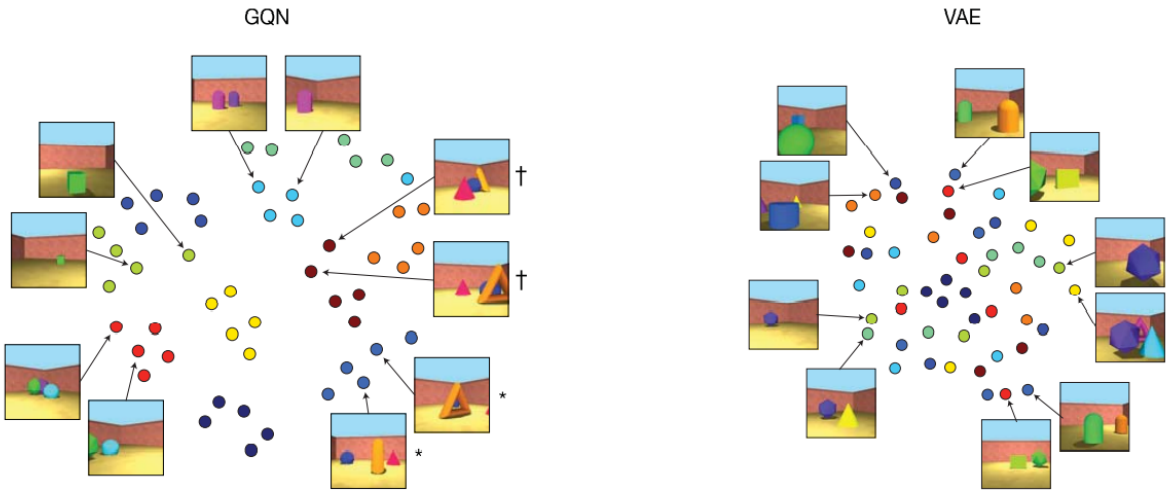


Figure 4.5: Left: t-SNE embeddings of GQN representation vectors. Right: t-SNE embeddings of VAE representation vectors. Source: [Esl+18]

asterisk symbols belong to two different scenes that contain the same objects but with different positions, yet they're clearly separated.

4.1 GQN Model Architecture

4.1.1 Representation Network

The conditioning variables that are required to generate an image from a queried viewpoint are all observed images $\mathbf{x}_i^{1,\dots,K}$, their viewpoints $\mathbf{v}_i^{1,\dots,K}$ and the queried viewpoint $\mathbf{v}_i^q$. The representation network is used to compute a scene representation vector $\mathbf{r}$ which is effectively a summary of the scene observations:

$$\mathbf{v}^k \equiv (\mathbf{w}^k, \mathbf{y}^k, \mathbf{p}^k) \tag{4.1}$$

$$\hat{\mathbf{v}}^k = (\mathbf{w}^k, \cos(\mathbf{y}^k), \sin(\mathbf{y}^k), \cos(\mathbf{p}^k), \sin(\mathbf{p}^k)) \tag{4.2}$$

$$\mathbf{r}^k = \psi(\mathbf{x}^k, \hat{\mathbf{v}}^k) \tag{4.3}$$

$$\mathbf{r} = \sum_{k=1}^{K} \mathbf{r}^k \tag{4.4}$$

Where $\psi(\mathbf{x}^k, \hat{\mathbf{v}}^k)$ is the representation network and $\mathbf{w}^k$, $\mathbf{y}^k$ and $\mathbf{p}^k$ are the position, yaw and pitch of the viewpoint (of the k'th observation) respectively. We take the cosine and sine of the yaw and pitch so that they can be represented by numbers within the range of $[0, 1]$ instead of $[0, 2\pi]$ for numerical stability. Attaining the scene representation vector by summing the representations of individual observations is simple but effective. During an agent's observation of a scene, adding new evidence about the scene is as simple as adding the representation vector of the latest observation to the scene representation. Another benefit of the additive aggregation method is that it is permutation invariant, it does not matter in what order the representation vectors are aggregated, the result will be the same. However, adding representation vectors element-wise means that each element - a floating-point number - might potentially hold information about multiple components of the scene. This means that adding the representations of new observations to the scene representation beyond a certain point might not add new information but may interfere due to the entangled nature of the scene representation vector.

Representation architectures

Three possible representation network architectures were proposed by the authors of the paper Neural Scene Representation and Rendering: Pyramid, Tower and Pool.

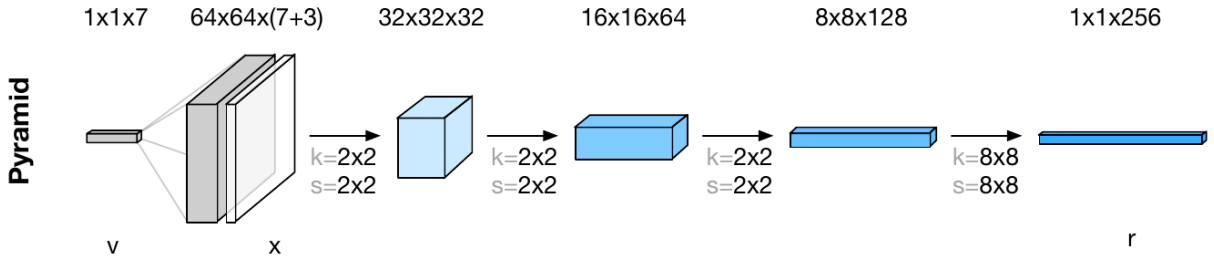


Figure 4.6: Pyramid representation network. Source:[Esl+18]

The pyramid network is the simplest of the three but has the most trainable parameters nonetheless due to the large kernel size of the last layer. The observed viewpoint is included in the input by copying it for each pixel in the observed image. The idea is to keep a certain number of features in the end which will become the representation vector, in this case there are 256 features.

All black arrows represent convolutional layers followed by ReLU activations. The kernel sizes and strides are denoted by k and s respectively.

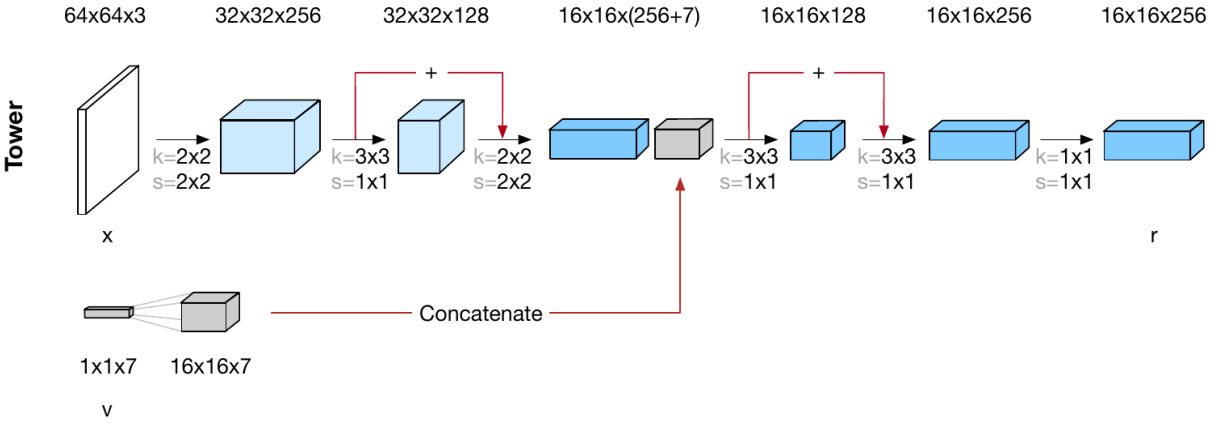


Figure 4.7: Tower representation network. Source: [Esl+18]

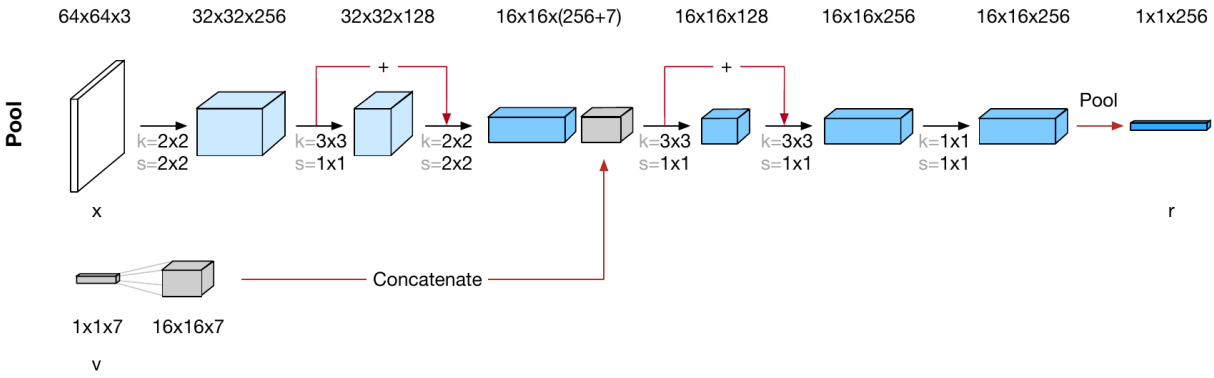


Figure 4.8: Pyramid representation network. Source: [Esl+18]

The only difference between the tower and pool networks is that the pool network has an average pooling layer at the end to reduce the representation size from 16x16x256 to 1x1x256. The viewpoint is concatenated to the output of the first residual block instead of the input image. The red arrows with the + symbol indicate residual connections, e.g. the input of the residual block is added to its output.

Residual connections, also called *skip connections* [He+15], help mitigate the vanishing gradient problem. The GQN will be trained with the back-propagation algorithm. This means that the weights of the representation network will be updated by a small fraction of the loss function's gradients which is calculated with the chain rule. If the network is deep, the chain of multiplications will get long and if a lot of terms smaller than one are multiplied together then the gradients of the earlier layers will become

vanishingly small. It is possible for the gradients to become zero after a certain amount of layers, causing the earlier layers to stay unchanged.

Skip connections essentially connect two non-adjacent layers by *skipping* some layers. This provides an extra, shorter path for the gradients to flow through to the earlier layers without getting too small.

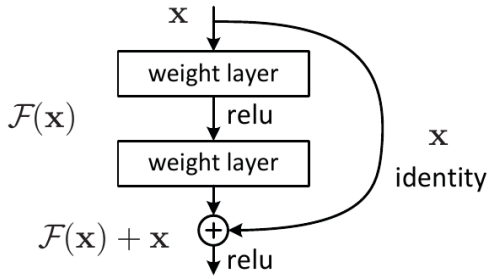


Figure 4.9: An additive residual (skip) connection introduced by the original ResNet paper. Source: [He+15]

4.1.2 Generator Network

The generator of the GQN model is the conditional version of the Convolutional DRAW (CDRAW) model [Gre+16] and has additional conditioning variables besides the latent variable $\mathbf{z}$ for the prior and posterior distributions. These additional conditioning variables are the scene representation vector $\mathbf{r}$ and the query viewpoint $\mathbf{v}^q$. Like the standard CDRAW model, the GQN generator has an encoder and a decoder which are both implemented as convolutional LSTMs.

The generator described in the supplementary materials of [Esl+18] consist of a single pair of encoder and decoder. We will call such a pair a "DRAW layer" which is illustrated in figure 4.10. The number of recurrent steps that the generator takes to generate an image is a fixed hyperparameter which we will call T.

Since this model is a type of Variational Autoencoder, the encoder and decoder have the same exact purpose as described in section 3.3, i.e. the encoder is used during the training phase to approximate the distribution of the training dataset's latent space and a KL-term is added to the loss function to learn a prior distribution that is close to the approximate posterior. Conceptually speaking, this encodes the knowledge that is present in the training dataset to the prior distribution, with the condition that the KL divergence could be converged to zero. This way, new images that look like images from the dataset can be sampled from the generator using the prior. Adding extra conditionals to the prior that describe the details of the scene ($\mathbf{r}$ and $\mathbf{v}^q$) will constrain the generator to produce images that resemble the observed scene.

The prior, posterior and likelihood distributions are all modelled with Gaussian distributions. The parameters of the prior, posterior and likelihood (i.e. the means and variances for the Gaussian) are computed by passing the hidden state of the decoder $\mathbf{h}_t^g$, hidden state of the encoder $\mathbf{h}_t^e$ and the final state of the canvas $\mathbf{u}_T$ through a convolutional layer respectively.

Firstly, we will describe the sampling process where the prior is used to generate an image, this process is done after training to use the model in practice. Secondly, we will describe the "inferencing" process where the target posterior is approximated using the encoder, this is done during training to learn a good prior. Finally, we will describe the details of how the model is optimized, i.e. trained.

It is worth noting that the term "inferencing" should not be confused with "Machine Learning Inference" (ML inference). ML inference is the process of using the trained model in practice which is equivalent to the "sampling process" we'll describe next. The inferencing process here corresponds with Variational Inference [JJ00].

The sampling process

The sampling process is for generating an image after the model has been trained. To generate an image, we must sample a latent vector $\mathbf{z}$ for the likelihood distribution after which we can sample an image from it: $\mathbf{x}^q \sim g_\theta(\mathbf{x}|\mathbf{z}, \mathbf{v}^q, \mathbf{r})$. Since the image is generated in T steps, the latent vector $\mathbf{z}$ is split into T groups

of variables $\mathbf{z}_t$ with $t \in \{1, ..., T\}$. Each $\mathbf{z}_t$ is sampled sequentially from the prior distribution. This means that a latent vector $\mathbf{z}_t$ depends on all latent vectors that have been sampled before it, yielding an auto-regressive prior distribution:

$$\pi_\theta(\mathbf{z}|\mathbf{v}^q, \mathbf{r}) = \prod_{t=1}^{T} \pi_{\theta_t}(\mathbf{z}_t|\mathbf{v}^q, \mathbf{r}, \mathbf{z}_{<t}) \tag{4.5}$$

With θ_t the model parameters involved in recurrent step t. With the prior defined, we can express a single recurrent time step in the generation process with the following set of equations:

$$\pi_{\theta_t}(\mathbf{z}_t|\mathbf{v}^q, \mathbf{r}, \mathbf{z}_{<t}) = \mathcal{N}(\mathbf{z}_t|\eta^\pi_\theta(\mathbf{h}^g_t)) \tag{4.6}$$

$$\mathbf{z}_t \sim \pi_{\theta_t}(\mathbf{z}_t|\mathbf{v}^q, \mathbf{r}, \mathbf{z}_{<t}) \tag{4.7}$$

$$(\mathbf{c}^g_{t+1}, \mathbf{h}^g_{t+1}) = \text{ConvLSTM}^g_\theta(\mathbf{c}^g_t, \mathbf{h}^g_t, [\mathbf{v}^q, \mathbf{r}, \mathbf{z}_t]) \tag{4.8}$$

$$\mathbf{u}_{t+1} = \mathbf{u}_t + \text{UpConv}(\mathbf{h}^g_{t+1}) \tag{4.9}$$

Where

- $\mathbf{c}^g_t$ and $\mathbf{h}^g_t$ are the cell and hidden states of the decoder LSTM at recurrent step t
- $\mathbf{u}_t$ is the *canvas* tensor which gets updated at each step and will be used to compute the parameters of the likelihood distribution for sampling $\mathbf{x}$
- $\eta^\pi_\theta(\mathbf{h}^g_t))$ is the convolutional layer for computing the means and variances of the prior distribution
- ConvLSTM^g_θ is the decoder LSTM
- $\text{UpConv}(\mathbf{h}^g_{t+1})$ is a transposed convolutional layer for upscaling $\mathbf{h}^g_{t+1}$ to the same resolution as $\mathbf{u}_{t+1}$
- The cell state, hidden state and canvas are all initialized with zero: $(\mathbf{c}^g_0, \mathbf{h}^g_0, \mathbf{u}_0) = (\mathbf{0}, \mathbf{0}, \mathbf{0})$

After T recurrent steps have been executed, the generated image can be sampled from a likelihood distribution whose mean is computed by passing the final canvas $\mathbf{u}_T$ through a convolutional layer η^g_θ:

$$\mathbf{x} \sim \mathcal{N}(\mathbf{x}^q|\mu = \eta^g_\theta(\mathbf{u}_T), \sigma = \sigma_t) \tag{4.10}$$

The variance of the likelihood distribution is annealed during the training process, starting from a relatively large value and getting smaller as the training advances. A larger variance forces the model to focus on the high-level details while a small variance forces it to focus on smaller details.

The inferencing process

The goal of the inferencing process is to approximate the distribution of the latent space conditioned on the observations, i.e. the posterior distribution. Compared with the prior, the approximate posterior q_ϕ gets the to-be-generated image $\mathbf{x}^q$ as an additional conditioning variable, which makes it possible to approximate the true posterior. When training the model, we use q_ϕ to sample $\mathbf{z}$ instead of the prior. The approximate posterior can therefore be defined as an auto-regressive distribution analogous to the prior:

$$q_\phi(\mathbf{z}|\mathbf{x}^q, \mathbf{v}^q, \mathbf{r}) = \prod_{t=1}^{T} q_{\phi_t}(\mathbf{z}_t|\mathbf{x}^q, \mathbf{v}^q, \mathbf{r}, \mathbf{z}_{<t}) \tag{4.11}$$

During the inferencing process, both the encoder and decoder will be used, so ϕ is the union of the encoder and decoder parameters. This means that θ is a subset of ϕ because the encoder is not used during the sampling process.

With the approximate posterior q_ϕ defined, we can define a single recurrent time step during the inferencing process with the following equations:

$$(\mathbf{c}^e_{t+1}, \mathbf{h}^e_{t+1}) = \text{ConvLSTM}^e_\phi(\mathbf{c}^e_t, \mathbf{h}^e_t, [\mathbf{u}_t, \mathbf{h}^g_t, \mathbf{x}^q, \mathbf{v}^q, \mathbf{r}]) \tag{4.12}$$

$$q_{\phi_t}(\mathbf{z}_t|\mathbf{x}^q, \mathbf{v}^q, \mathbf{r}, \mathbf{z}_{<t}) = \mathcal{N}(\mathbf{z}_t|\eta^q_\phi(\mathbf{h}^e_t)) \tag{4.13}$$

$$\mathbf{z}_t \sim q_{\phi_t}(\mathbf{z}_t|\mathbf{x}^q, \mathbf{v}^q, \mathbf{r}, \mathbf{z}_{<t}) \tag{4.14}$$

$$(\mathbf{c}^g_{t+1}, \mathbf{h}^g_{t+1}) = \text{ConvLSTM}^g_\theta(\mathbf{c}^g_t, \mathbf{h}^g_t, [\mathbf{v}^q, \mathbf{r}, \mathbf{z}_t]) \tag{4.15}$$

$$\mathbf{u}_{t+1} = \mathbf{u}_t + \text{UpConv}(\mathbf{h}^g_{t+1}) \tag{4.16}$$

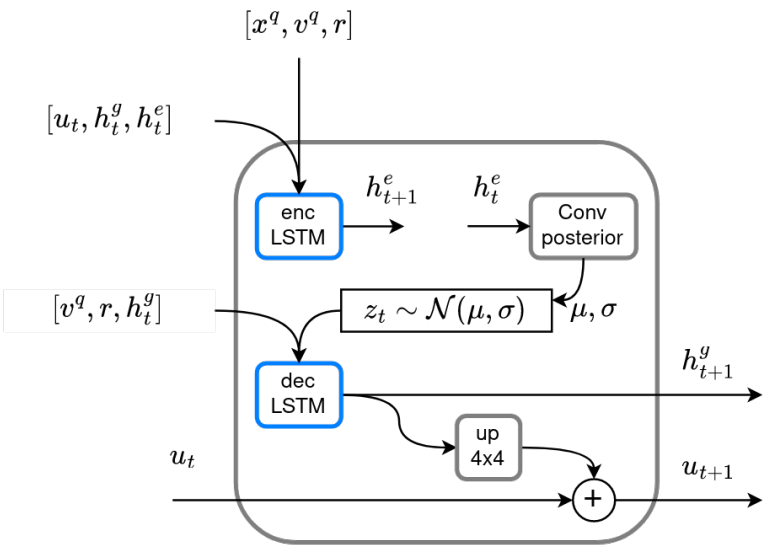


Figure 4.10: A single layer of the GQN generator architecture, i.e. the "DRAW layer". The generator described in the supplementary materials of [Esl+18] consists of a single layer. Notice that this diagram corresponds with the generator operations that are done during the inferencing process (section 4.1.2). The encoder LSTM is not used during the sampling process (section 4.1.2), so $\mathbf{z}$ would be sampled from the prior which is computed using the decoder hidden state $\mathbf{h}^g$. For backpropagation to work with distributions, the reparametrization trick [KW22] is applied for sampling $\mathbf{z}$. Source: Author

Where

- $\mathbf{c}_t^e$ and $\mathbf{h}_t^e$ are the cell and hidden states of the encoder LSTM respectively
- $\eta_\phi^q(\mathbf{h}_t^e))$ is the convolutional layer for computing the means and variances of the approximate posterior distribution

With the equations for computing the prior 4.6, the posterior 4.13 and the likelihood 4.10, we can now move on to compute the loss function for optimizing the model.

4.1.3 Optimization

As mentioned in section 3.3, the loss function has two major components, i.e. the reconstruction likelihood and the KL divergence between the approximate posterior and the prior. The reconstruction likelihood tells us how likely it is that the queried image $\mathbf{x}^q$ will be sampled from the likelihood $\mathbf{x}^q \sim g_\theta(\mathbf{x}|\mathbf{z}, \mathbf{v}^q, \mathbf{r})$. We want the model to generate the queried images, so the goal is to maximize the reconstruction likelihood. The KL divergence between the approximate posterior and the prior serves to minimize the divergence (statistical distance) between the two distributions so that the information captured by the posterior is also stored in the prior. The loss function is defined as the following:

$$\mathcal{F}(\phi, \theta) = \mathbb{E}_{(\mathbf{x},\mathbf{v})\sim D, \mathbf{z}\sim q_\phi} \left(\underbrace{-\ln \mathcal{N}(\mathbf{x}^q|\eta_\theta^g(\mathbf{u}_L))}_{\text{likelihood}} + \underbrace{\sum_{t=1}^{T} \mathrm{KL}[\mathcal{N}(\cdot|\eta_\phi^q(\mathbf{h}_t^e))||\mathcal{N}(\cdot|\eta_\theta^\pi(\mathbf{h}_t^g))]}_{\text{KL div posterior||prior}} \right) \tag{4.17}$$

The training for a single-layer GQN is carried out in n steps, where each training step looks as follows:

1. Sample a mini-batch of size B from the training dataset D where each scene has K observations, which is picked randomly to be a number between 0 and $N-1$, with N the total number of available observations per scene in the dataset. The batch consists of B scenes with K observations along with a query viewpoint and image.

2. Compute the scene representation vector $\mathbf{r}$ (eq. 4.4);
3. Do the following for T recurrent time-steps:
   (a) Update the encoder LSTM state (eq. 4.12);
   (b) Compute the approximate posterior parameters and sample $\mathbf{z}$ from it (eqs. 4.13 & 4.14);
   (c) Update the decoder LSTM state and update the canvas (eqs. 4.15 & 4.16);
   (d) Compute the prior (eq. 4.6);
   (e) Compute the KL divergence between the approximate posterior and the prior, and add it to the total KL divergence of the current training step (see eq. 4.17);
4. Compute the log-likelihood of the queried image (see eq. 4.17);
5. Compute the loss by aggregating the log-likelihood and the total KL divergence (eq. 4.17);
6. Compute gradients and do backpropagation to update the model parameters.

4.1.4 Multi-scale generators

The generator "layers" that are mentioned in the GQN paper [Esl+18] refer to the time steps of an RNN. They possibly refer to stacking an RNN *horizontally* which means adding more recurrent time steps, or in other words, layers that share the same parameters. Vertical stacking in RNN terminology refers to stacking different layers with their own parameters. To avoid confusion, we refer to horizontally stacked layers as "time steps" and vertically stacked layers simply as "layers".

The original GQN paper [Esl+18] did not describe any form of multi-layer generator model and as of writing this, *to our knowledge*, no research has been published describing a multi-layer generator for the GQN. To this end, as our own contribution, we introduce a couple of different ways to link multiple generator layers. We will refer to the layer of a GQN generator as a "DRAW layer" which is illustrated in figure 4.11. As described in section 4.1.2, DRAW layers are a type of variational autoencoder consisting of an encoder and a decoder LSTM. The inputs are the concatenation of the queried image $\mathbf{x}^q$ along with its viewpoint $\mathbf{v}^q$, the scene representation $\mathbf{r}$ and the canvas $\mathbf{u}$. $\mathbf{h}^g$ is the hidden state of the decoder LSTM which is the first output along with the updated canvas $\mathbf{u}_{t+1}$. We will refer to the decoder hidden state simply as $\mathbf{h}$.

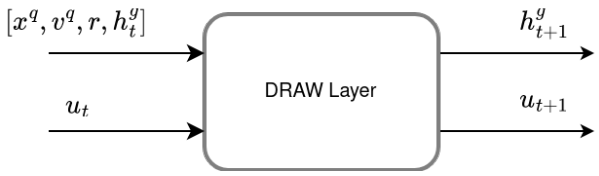


Figure 4.11: Simplified version of the DRAW layer, see figure 4.10 for a schema of the inner workings. Source: Author

To connect multiple DRAW layers together, we need to pass the outputs $\mathbf{h}$ and $\mathbf{u}$ in some way to the next layer. We will not constrain the layers to have the same amount of time steps meaning only the outputs of the layer's final time step will be sent to the next layer. Let L be the number of layers and let $T_i, i \in \{1, ..., L\}$ be the number of time steps of layer i, then the outputs passed onto the next layer are $\mathbf{h}^i_{T_i}$ and $\mathbf{u}^i_{T_i}$. The superfix i denotes the layer that the variable belongs to.

The rough details in an image can be stored in a lower resolution, which is easier to learn due to the reduced amount of parameters. So when stacking multiple DRAW layers, each consecutive layer will have an increasing resolution, hence the name "multi-scale" generators. The first layer will receive images in the lowest resolution, the second layer will receive the upscaled results of the first layer and add further details to it before passing its results onto the third layer. For our experiments, we chose a scaling factor of two between layers. More concretely, let $\downarrow_s (x)$ and $\uparrow^s (x)$ be the downscaling and upscaling operations respectively that scale the resolution of an image x by a factor of s, then the image that layer i will receive is $x_i = \downarrow_{j(L-i)} (x)$ with j the scaling factor between layers. To keep it simple, we assume that the edge case $s = 0$ does not scale the input just like $s = 1$. The layer inputs are always downscaled with bilinear interpolation.

We define two types of multi-scale generators:

- **Type 1**: the canvas is shared between the layers. When a layer is done updating it, the canvas is upscaled either with bilinear interpolation or with a transposed convolutional layer. The hidden state $\mathbf{h}_T^i$ is also upscaled to match the resolution of the next layer's hidden state and is passed to the next layer by concatenating it with the inputs. The hidden state is only upscaled using transposed convolutional layers. Type 1 is illustrated in figure 4.12;
- **Type 2**: the canvas is not shared between layers, but is instead passed as input to the next layer along with the hidden state. The inputs for the layers starting from the second layer could either be the downscaled original images x_i or their high-frequency details $y_i = x_i - \uparrow^2 (x_{i-1})$. The upscaling and downscaling operations used for obtaining y_i are done with bilinear interpolation. Type 2 is illustrated in figure 4.13.

For both types, there is also the option to add the negative log-likelihood (NLL) of each layer to the total loss function or only that of the last layer. We illustrate both methods schematically, adding only the NLL of the last layer in figure 4.13 and the NLL of all layers in figure 4.12. The advantage of including the NLL of each layer is that we can force the layers to generate the inputs they receive. If we apply this to the layers that receive the high-frequency details y_i of the images, we could force those layers to become Image Super-Resolution models. The disadvantage is that the layers will be constrained to learn their inputs which may not be the optimal solution in terms of how efficiently the model parameters are utilized.

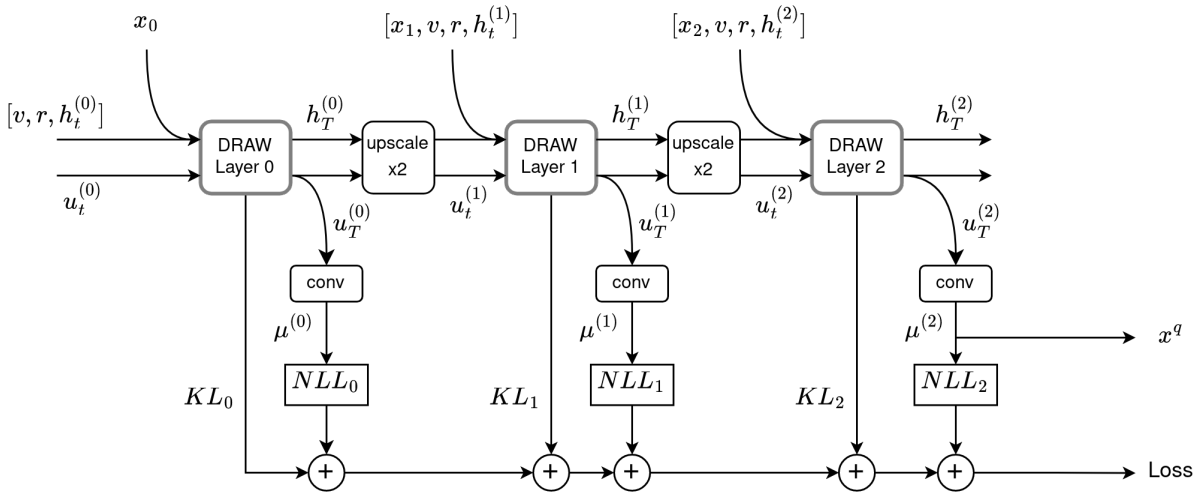


Figure 4.12: Type 1 multi-scale GQN generator where the canvas is reused between layers. Source: Author

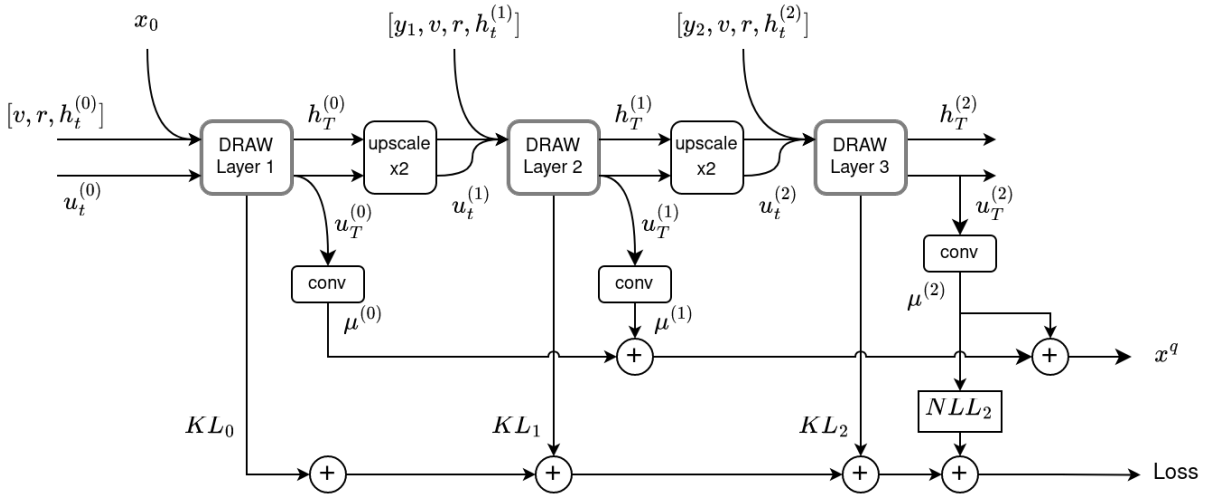


Figure 4.13: Type 2 multi-scale GQN generator where the canvas is passed as a secondary input to the next layer. y_i only contains the details of x_i which is attained through the difference of Gaussians [23]. Source: Author

Chapter 5

Experiments

We used the PyTorch [Pas+19] library to implement and train the GQN model. All experiments have been carried out in Python with the help of tools provided by PyTorch. All line charts have been extracted from WandB, an online platform for managing and tracking the progress of training models. The mentioned datasets are always split into the training and testing set with a ratio of 9 training samples per 1 testing sample. For all models, the "pool" representation network is used as introduced by [Esl+18].

5.1 Choice of performance metric

To measure the performance of the generator network consistently and robustly, we need an objective Image Quality Assessment method (IQA). The difference between subjective and objective IQA is that the former is done by humans and the latter with mathematical methods. Subjective IQA is impractical and prone to error, whereas objective IQA is consistent and its robustness depends on the technique being used. The property of the IQA method we're interested in is that it must do well in measuring the structural similarity independently of luminance and contrast similarity. To this end, we will use the Structural Similarity (SSIM) [HZ10] method as the main metric and PSNR [Far+16] as a secondary extra metric to compute the similarity between the ground truth and generated images.

We refer readers to appendix A for a description of the metrics that have been considered.

5.2 GQN with multi-scale generators

In section 4.1.4 we introduced our own method for chaining multiple generator layers, instead of just using one DRAW layer. Our goal in this section is to define different configurations of the multi-scale generator, train these, get some insights on the contributions of the individual layers and compare them with the standard GQN which has a one-layer generator.

5.2.1 Training configuration

In this section, we describe the configurations of the models that were trained. All multi-scale generators have a layer configuration of $(2, 4, 6)$, meaning they consist of three layers where the first, second and third layers have two, four and six time-steps respectively. The latent space $\mathbf{z}$ has 16 channels and the scene representation $\mathbf{r}$ has 256 channels. The hidden state $\mathbf{h}$, cell state $\mathbf{c}$ and canvas $\mathbf{u}$ have all 128 channels. The resolution scaling between the layers is two, meaning that the resolution of the input and output doubles at each consecutive layer where the last layer has the same resolution as the images in the dataset. The resolution of the hidden states is always four times less than that of the input, e.g. hidden state tensors have a height and with of 16 if the input images have a height and width of 64 pixels.

All models are trained on the "shepard_metzler_5_parts" (SM5) dataset [Dee] for 300000 steps. The dataset roughly contains 800000 scenes of 5-block Shepard Metzler (SM) objects, with image resolutions of 64x64. The type 1 models are trained with a batch size of 24 while the type 2 models are trained with a batch size of 20. The difference in batch size is due to a typing error, but should not affect the results significantly since the loss is averaged over all data samples in the batch.

The optimizer hyperparameters are the same as in the supplementary materials of the GQN paper [Esl+18]. The models are trained with the Adam [KB17] optimizer with the following hyperparameters:

- Learning rate: 0.0005
- ϵ: 1e-08
- β_1: 0.9
- β_2: 0.999

A learning rate scheduler is also used to gradually reduce Adam's learning rate from 0.0005 to 0.00005 over 1.6 million steps. The variance of the negative log-likelihood 4.10 in the loss function is annealed from 2.0 to 0.7 in the first 200000 training steps.

We have trained seven models in total with the following distinct configurations:

- **Model 0**: The original one-layer generator as introduced by [Esl+18]. We will sometimes refer to this model as the "one-layer model" for clarity.
- Type 1 multi-scale generators: the layers share the canvas, and the initial canvas state of a layer is the final canvas state of the previous layer:
  - **Model 1 (interpolation)**: the canvas is passed onto the next layers by upscaling it with bilinear interpolation.
  - **Model 2 (upconvolution)**: the canvas is upscaled with a transposed convolutional layer.
  - Note: for both models, only the log-likelihood of the last layer is added to the total loss.
  - Note: the last hidden state $\mathbf{h}_{T_i}$ of a layer is passed as an additional input to the next layer along with the viewpoints $\mathbf{v}^q$, the representation vector $\mathbf{r}$, etc.
- Type 2 multi-scale generators: no shared canvas between layers, the final result is computed by summing the outputs of each layer:
  - **Model 3 (normal)**: the input of each layer is just the images downscaled to the layer's resolution ($\mathbf{x}_l$) and not the high-frequency details ($\mathbf{y}_l$). The Log-likelihood of each layer is added to the total loss.
  - **Model 4 (normal, last LL)**: same as model 3 but only the log-likelihood of the last layer is added to the total loss.
  - **Model 5 (high-pass)**: the layers starting from the second receive the high-frequency details (i.e. high-pass) of the images as input. The Log-likelihood of each layer is added to the total loss.
  - **Model 6 (high-pass, last LL)**: same as model 5 but only the log-likelihood of the last layer is added to the total loss.
  - Note: each layer only sends the canvas state as input to the next layer by upscaling it with a transposed convolutional layer.

For the type 2 models, we only send the final canvas state $\mathbf{u}_{T_i}$ to the next layer as it contains the accumulated results of all time steps, while the final hidden state $\mathbf{h}_{T_i}$ only contains information about the last time step. Note that only model 0 is the original model that was proposed by the GQN paper [Esl+18], all the rest are multi-layer generators. All models execute 12 recurrent time steps in total.

5.2.2 Training results

The KL divergence and negative log-likelihood (NLL) losses in figure 5.1 shows us that there is no noticeable performance gain when upscaling the canvas with a transposed convolutional layer instead of simple bilinear interpolation. We also do not see a noticeable difference when looking at the generation quality metrics, SSIM and PSNR, in figure 5.3. However, when looking at some of the generated samples in figure 5.6, we can see that model 1 renders some blocks in the wrong colour. From these observations, we can conclude that it is sufficient to upscale the canvas with bilinear interpolation when only

structural accuracy is needed, but transposed convolutional layers are necessary if colour accuracy is a requirement.

When looking at the type 2 models, we see from the KL divergence plot in figure 5.2b that model 3 performs very poorly compared with the others. It is much slower in encoding the information in the posterior into the prior. Although, we can also see that it has a downward trend, meaning it could potentially catch up to the other models after training for an additional couple hundred thousand steps. We can see how this affects the reconstruction quality from the reconstruction losses in figure 5.4 and the errors in the generated images shown in figure 5.7. A possible explanation could be that the added NLL terms of each layer make the training objective more difficult. Though it is interesting to see that model 5, the high-pass counterpart of model 3, performs reasonably well. This may mean that it is easier to learn high-frequency details instead of the images themselves. The remaining two models that only include the NLL of the last layer to the total loss perform very similarly to each other. We can see from the SSIM and PSNR plots in figure 5.4 that model 6 (high-frequency inputs) runs just behind model 4 (normal inputs) but catches up towards the end.

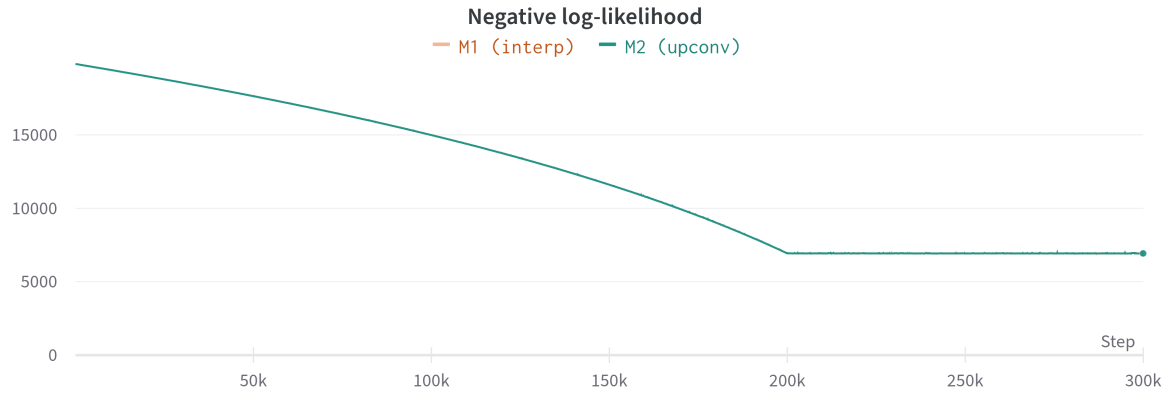
From our observations of type 2 generators, we can conclude that it is easier to converge the KL divergence of multi-layer generators when the NLL losses of the intermediate layers are excluded from the total loss. This may be due to the constraints that the NLL losses of the intermediate layers impose. When adding the NLL loss of an intermediate layer to the total loss, the corresponding layer must learn to generate its inputs in order for the total loss to go down. Although it is natural to make each layer learn to generate the inputs they receive, this may not be the most optimal method when aggregating the contributions of each layer, e.g. the capacity of the generator may be too high and not all layers may be needed to generate the desired images as shown by the results in section 5.2.3 (see figures 5.10, 5.11, 5.13 and 5.15).

For the following experiments, we will use model 2 because it is the better performing type 1 model and model 5 because it has a different training target than other models, i.e. the intermediate layers are forced to learn high-frequency details. See section 5.2.3 for a visualization of the contribution of each layer of models 1 to 6. The performance of model 4 is decent, but we will exclude it in the experiments because it resembles model 2. The layers of both model 4 and 2 receive normal images as inputs and they both generate an image with roughly the same operations, the only difference being *when* information is added to the canvas. Model 2 has a slightly better reconstruction quality than model 6 and the standard single-layer GQN with 12 time steps, as seen in figure 5.5. The single-layer GQN has been trained on the same dataset with a batch size of 24 (same as model 2). The other hyperparameters are exactly the same except for the layer configuration, i.e. 1 layer with 12 time steps.

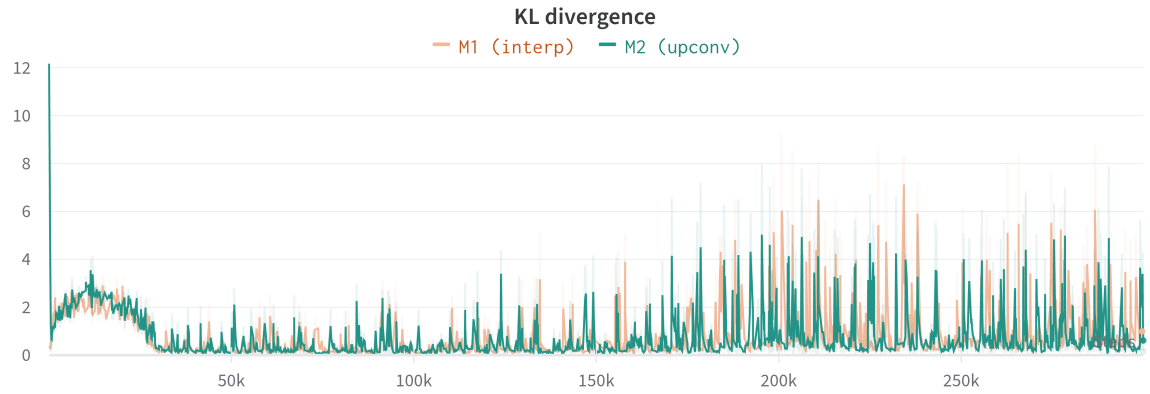
The negative log-likelihood and KL divergence plot do not give us much insight into the reconstruction quality, so we only included the SSIM and PSNR plots in figure 5.5 for comparing the multi-layer generators with the single layer GQN (model 0). We can see that the SSIM and PSNR curves of model 2 (shared canvas) are both consistently above those of the one-layer GQN after around step 55000 while model 6 is slightly worse than the single-layer GQN. Although we can state that model 2 performs slightly better than the single-layer GQN, the difference in quality is almost impossible to see with the naked eye, aside from the chromatic errors. Due to issues with the model implementations and training, the models have not been properly trained on a higher-resolution dataset, although the potential capacity of the multi-scale generators lies in generating higher-resolution images.

However, we did a preliminary training of the GQN with an early version of the multi-scale generator on a dataset with 128x128 images of three-block and four-block Shepard Metzler objects, where the results were showing potential. The generated images can be seen in figure 5.9 along with a plot of the unstable KL divergence. Each group of three images correspond with the ground-truth queried image of a scene, generated image using the inferencing process and the generated image using the sampling process. We can see from the second column of the three scenes that the model is able to capture the necessary information in the approximate posterior and generate images that are structurally close to the queried image. However, due to an inadequate training configuration, the model failed to reduce the divergence between the posterior and the prior, causing it to fail in the sampling process (third column). The cause for this is unclear and is investigated in more detail in section 5.3.4.

Aside from performing slightly better, the multi-layer GQNs also require less time to be trained. Model 2 has a process time of 24.97 hours, model 2 has a process time of 21.76 while the single layer GQN with 12 time steps has a process time of 38 hours. Model 2 and 6 have been trained on a computer with an



(a)



(b)

Figure 5.1: Test losses of type 1 models. M1 and M2 stand for Model 1 and 2.

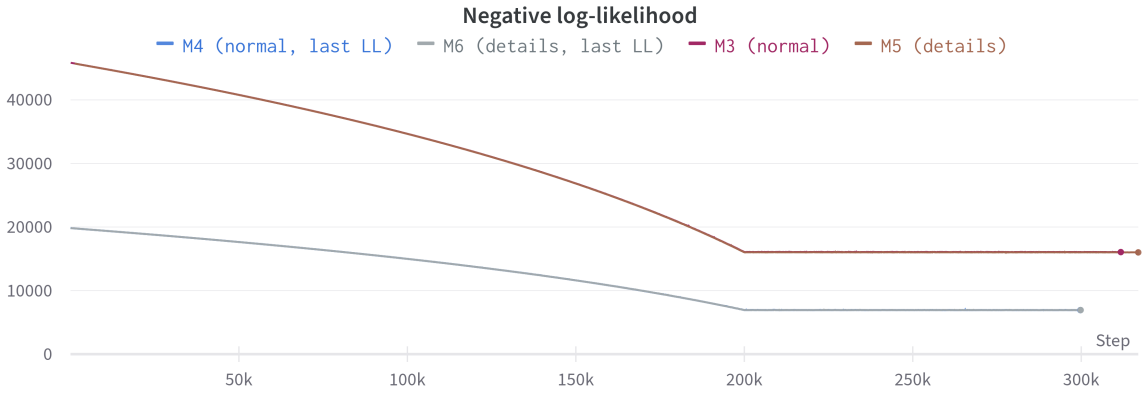
RTX 3070 GPU (8GB) and an Intel i9 9900k CPU while the single-layer GQN has been trained on a different computer with an RTX 2080 GPU (8GB) and an Intel i7 4700 CPU. Although the CPU is of an older generation, the only thing that happens on the CPU side is preparing the data for the GPU while the forward and backpropagation happen on the GPU. In figure 5.8 we can see that the GPU utilization is consistently being kept above 90%. Even though the multi-layer and single-layer GQNs have been trained on different GPUs, with a process time difference of around 13 hours it is safe to say that the multi-layer models are trained faster. This is not unexpected since the single-layer GQN executes 12 recurrent time steps on full resolution, while the multi-layer versions scale the data down for the earlier layers, effectively reducing the number of parameters (gradients) that need to be kept track of during the 12 recurrent time steps.

5.2.3 Generated images of intermediate layers

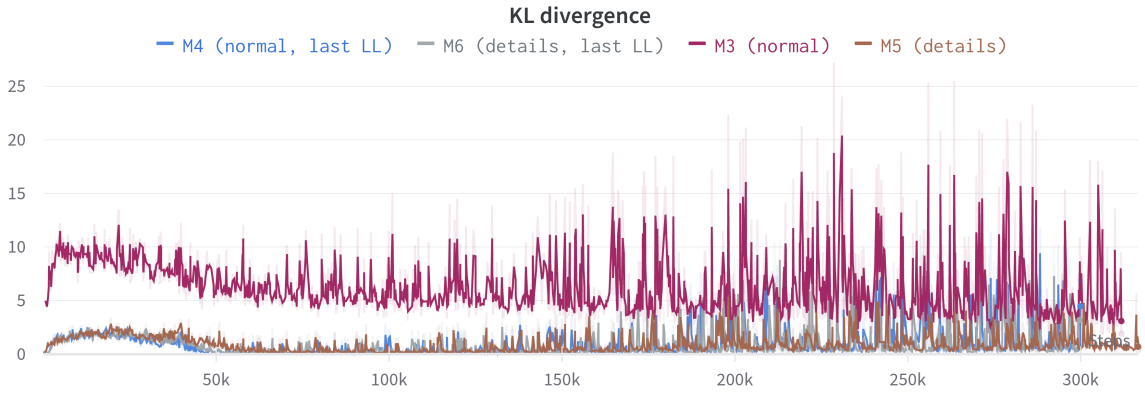
In this section, we will look at what each layer of the multi-layer generators produce. Models 1 through 6 are visualized in figures 5.10 through 5.15 in increasing order. The rows correspond with the layer outputs and the columns correspond with queried images from different scenes.

We can see that the first two layers of models 1, 2, 4 and 6 are barely used. The generated images of the first two layers of those models are practically black (notice that the colors have been scaled to range [0,1] causing each image to have a stark color). The only contribution of the first two layers may be the hidden state they pass onto the next layer.

Figure 5.11 shows the images that have been generated by the layers of model 2. We can see that the first two rows almost have a uniform color which means that the first two layers practically don't generate anything, their only contribution may be the hidden state they pass onto the next layer. Although these results are surprising, they are not entirely unexpected. When defined the model configurations, we

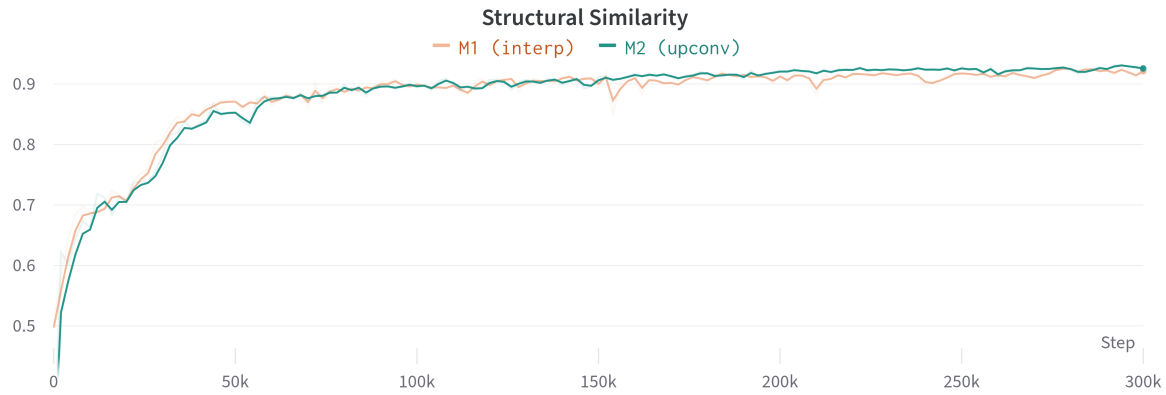


(a)

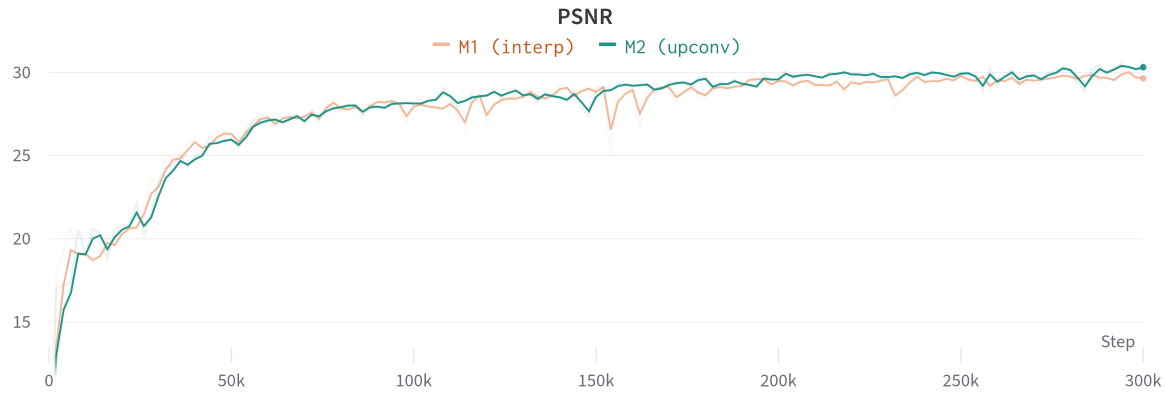


(b)

Figure 5.2: Test losses of type 2 models. The KL divergence plot is smoothed by 15 percent. In the NLL plot, we see that model 3 and model 5 are much higher than model 4 and model 6. This is expected as the NLL of each layer is added to the total loss. In the KL divergence plot we can see that model 3 failed to reduce the divergence between the prior and posterior, meaning the sampled images will likely look incorrect.

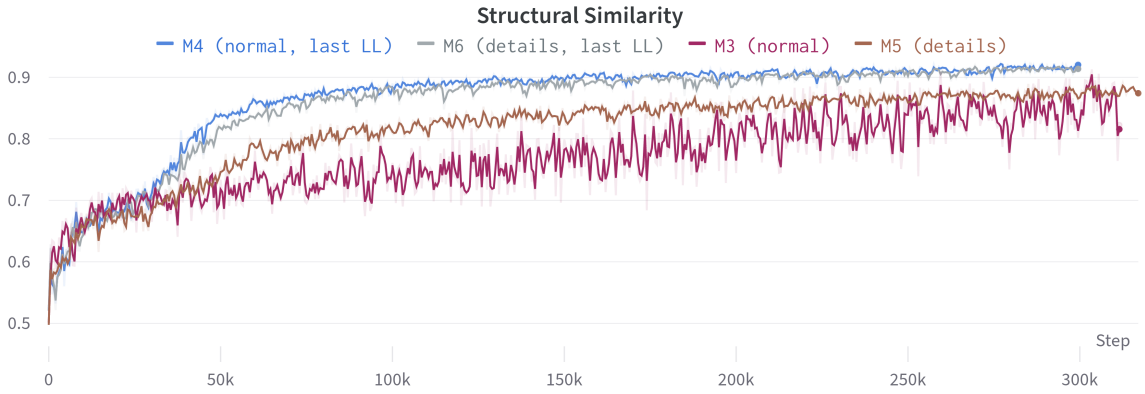


(a)

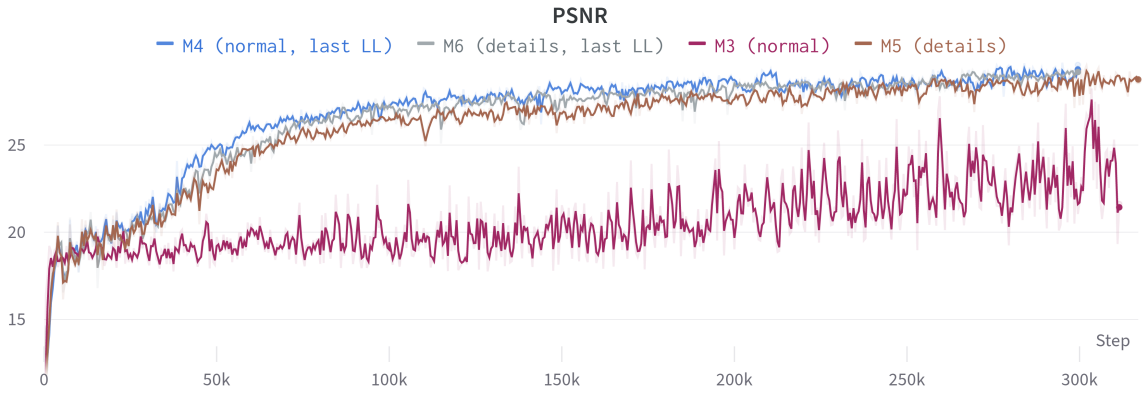


(b)

Figure 5.3: Image sampling (generation) quality of type 1 models on a held-out test-batch. SSIM is computed with a Gaussian kernel of size 11 and $\sigma = 1.5$.

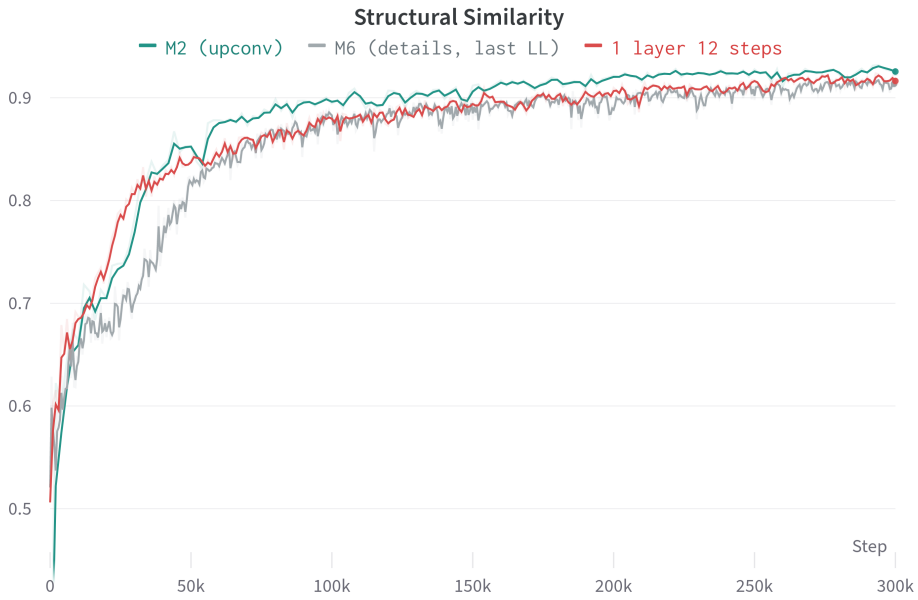


(a)

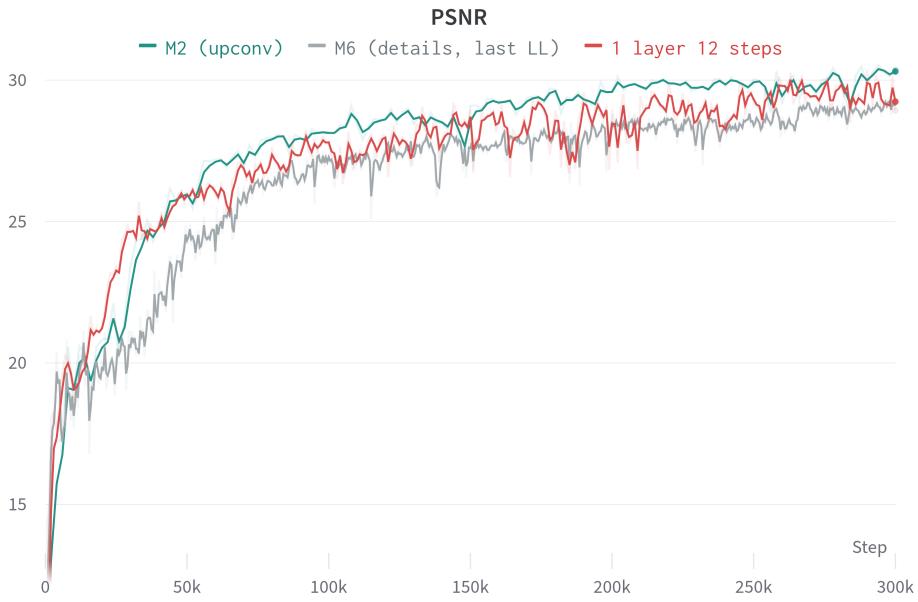


(b)

Figure 5.4: Image sampling (generation) quality of type 2 models on a held-out test-batch. Both plots are smoothed by 15 percent. We see that model 3 performs poorly compared with the other ones, which aligns with our observation in the KL divergence plot in figure 5.2b. Looking at the PSNR plot, we can barely see that model 5 performs worse than models 4 and 6. This quality difference is emphasized better in the SSIM plot.



(a)



(b)

Figure 5.5: Image sampling (generation) quality of model 2 and 6 along with a one-layer generator with 12 time steps on a held-out test-batch. Both plots are smoothed by 15 percent.

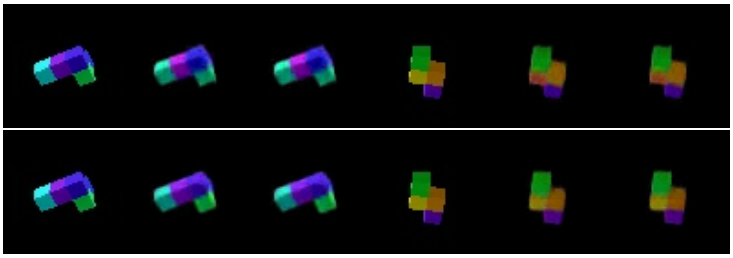


Figure 5.6: Generated images with type 1 models on a held-out test batch. Each row corresponds with a model and each group of 3 columns corresponds with a scene. The first of the 3 images is the ground-truth image that is queried, the second image is generated using the inferencing process (see section 4.1.2) and the third image is sampled from the model using the generation process (see section 4.1.2). These are generated in the "evaluation" mode where no tensor gradients are tracked during forward propagation and no gradients are updated. So, even though the second image is generated through a method used for training the model, the model doesn't learn anything by generating these images. Row 1 corresponds with model 1 and row 2 with model 2. We can see that model 1 is a bit color-inaccurate.

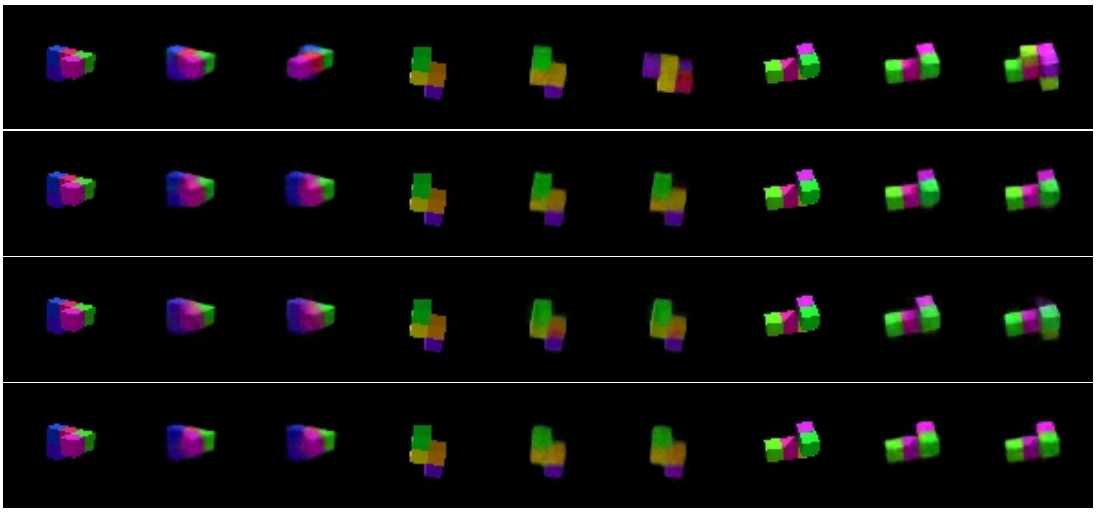


Figure 5.7: Generated images with type 2 models on a held-out test batch. Each row corresponds with a model in chronological from model 3 until 6. Each group of 3 images correspond with a scene. The first of the 3 images is the ground-truth image that is queried, the second image is generated using the inferencing process (see section 4.1.2) and the third image is sampled from the model using the generation process (see section 4.1.2) We see that model 3 (row 1) performs noticeably worse than the other models, it is able to generate something that looks like a Shepard Metzler object but the structure and colours are completely off. Model 5 (row 3) has an error in the third sample: the purple block on the top seems to be missing and it filled a space just below it that should have been empty.

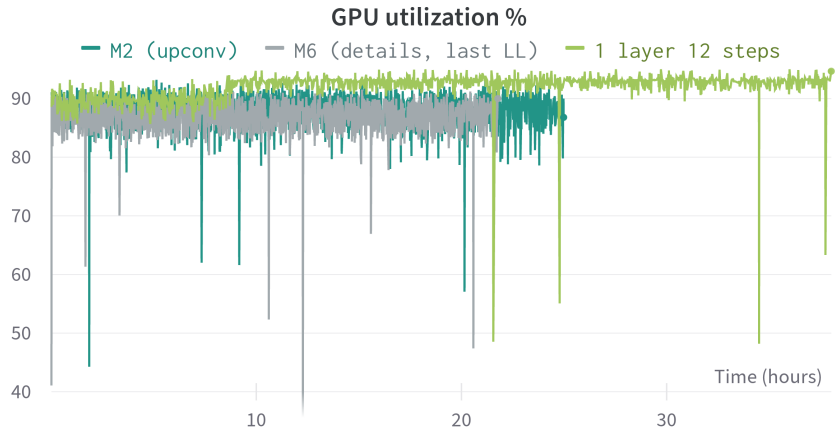


Figure 5.8: GPU utilization in percents w.r.t. process time when training model 2, 6 and the single-layer GQN.

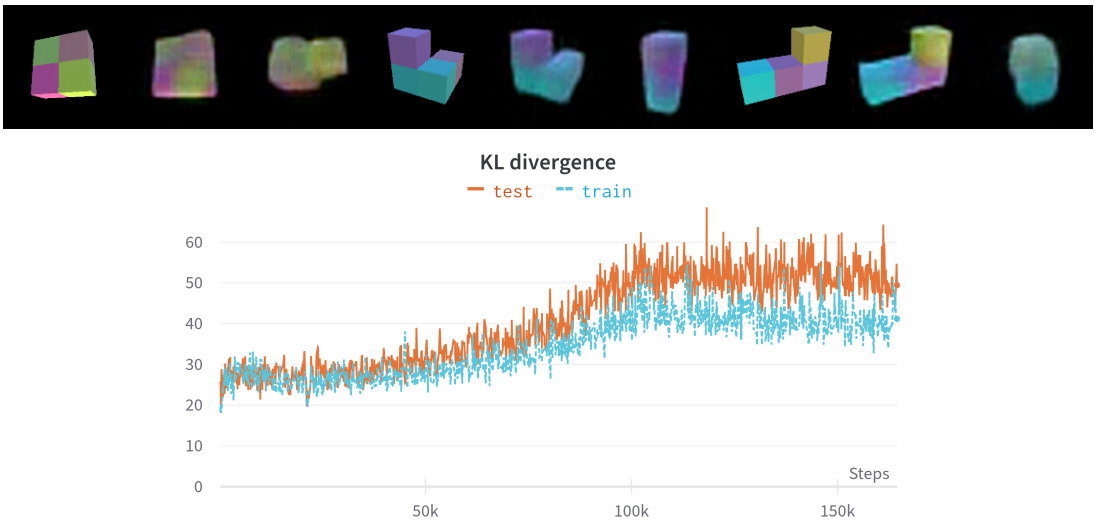


Figure 5.9: Samples of the GQN with an early version of the multi-scale generator which was similar to model 1. This early version of the model was trained on a dataset consisting of 128x128 images of 3-block and 4-block Shepard Metzler images. Like in figure 5.6 each group of three images corresponds with the ground truth (column 1), the image generated using the inferencing process (column 2) and the image generated using the sampling process (column 3). We can see that the generator had the capacity to encode the 128x128 images into the posterior and generate them (column 2) but it was unable to encode the information in the posterior into the prior (as shown by the rising KL divergence), causing it to fail in the sampling process (third column).

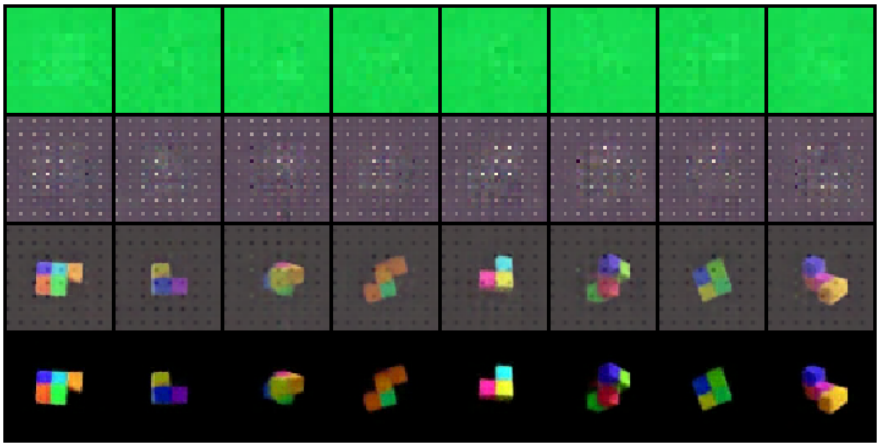


Figure 5.10: The images generated by the layers of model 1 (shared canvas, upscaled with bilinear interpolation). Each row corresponds with a layer and each column with the queried image of a scene. The images in the final row are an aggregation of the images generated by each row, which is the image returned as the final result. The images in all rows except the last row are rescaled to the range of [0, 1] to make the details of the first two layers more clearly visible. We can see that an aliasing-like artifact is introduced in the generated images of the second layer, which may be due to bilinear interpolation. The third layer has learned to revert these artifacts to produce a clean image reconstruction. The average minimum and maximum pixel values for each row are the following increasing order, respectively: $(-0.0687, 0.0415)$, $(-0.3069, 0.6072)$, $(-0.4203, 1.1196)$, $(-0.2120, 1.1275)$.

had predicted that including the NLL of each layer to the total loss will constrain each layer to learn to generate the inputs they receive. Not including the NLL of the intermediate layers removes that constraint and allows the model to look for more efficient solutions, in this case it is only using the last layer to generate the whole image and leaving the earlier layers blank. This theory is further complemented by the results of models 3 and 5 where the NLL of all layers have been included in the total loss, forcing all layers to be utilized to generate their respective inputs. The layers of model 3 had received the original images which are downscaled to the layer's resolution, while model 5 layers received the high-frequency details. An interesting thing to note is that the first layer of model 3 generates grayscale values, while it is not constrained to do so.

Models where the NLL of all layers are included in the total loss, like model 5, may be more useful than models where only the last layer's NLL is included (like model 2 & 6). One reason is that it is possible to just attach an additional layer to the former type, and train all layers end-to-end to generate higher resolution images. This will work because the targets of each layer will remain the same. We cannot make the same claim for the latter model type, since only the last layer is used to generate the final image, which may or may not depend on the hidden state it received from the previous layer. So it is not clear how the latter kind of model will behave during training when a new layer is added later on. A clear disadvantage of including the NLL of each layer is that it becomes more difficult to train these models due to the increased complexity in the loss function. However, the training could be improved and stabilized using normalization techniques for VAE-like loss functions. One method, among others, is the β-VAE method [Hig+16] where the KL divergence is scaled with a factor β where the scaling factor is used to favor either a better reconstruction quality or learning a better disentangled representation.

5.3 GQN generality

With the generality of a GQN model, we mean how well the model can handle scenes that are different from the scenes in the training dataset. This is an important quality because we would want the model to learn general concepts rather than memorizing every training scene it has seen. This way the model does not have to be trained on every possible scene configuration that exists to work well in practice.

To test the generality of the GQN, we will look at two different characteristics. The first is how well it handles colours it hasn't seen before and the second is how well it handles shapes it hasn't seen

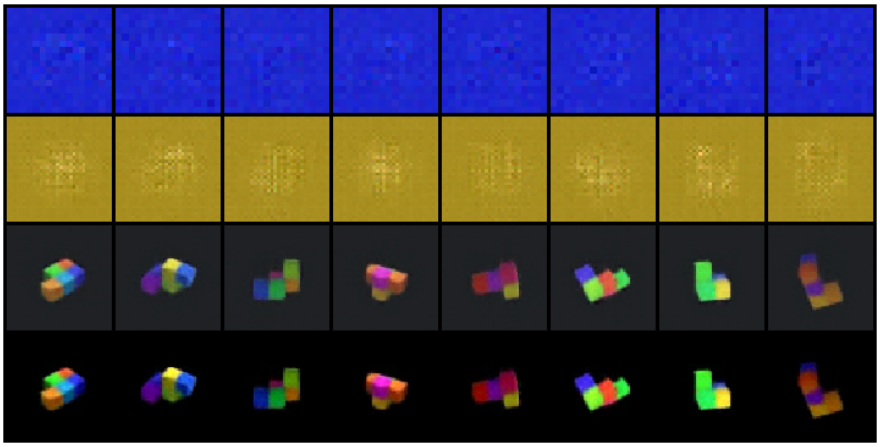


Figure 5.11: The images generated by the layers of model 2 (shared canvas, upscaled with transposed convolution). The structure of the images are the same as in figure 5.10 and the images are also rescaled to the range of [0, 1]. The average minimum and maximum pixel values for each row are the following increasing order, respectively: $(-0.0375, 0.0434)$, $(-0.0895, 0.0621)$, $(-0.1397, 1.1399)$, $(-0.1641, 1.1340)$.

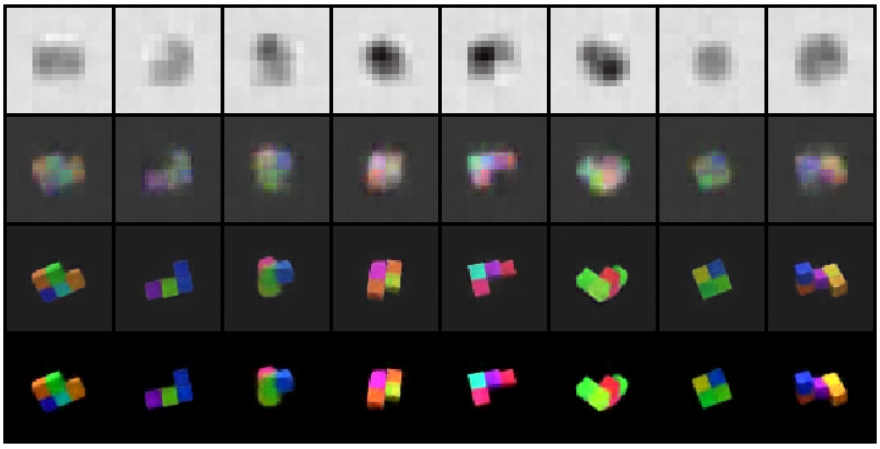


Figure 5.12: The images generated by the layers of model 3 (normal inputs, no shared canvas). The structure of the images are the same as in figure 5.10 and the images are also rescaled to the range of [0, 1]. The average minimum and maximum pixel values for each row are the following increasing order, respectively: $(-0.3184, 0.0440)$, $(-0.1554, 0.5737)$, $(-0.1566, 1.0931)$, $(-0.1472, 1.2848)$.

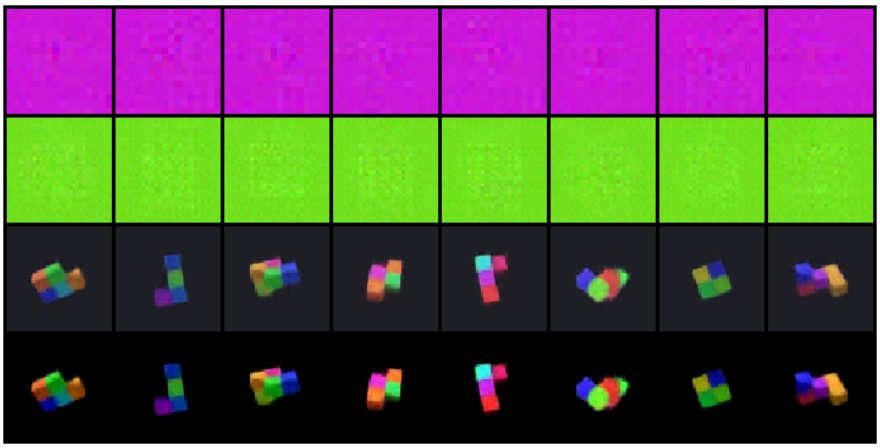


Figure 5.13: The images generated by the layers of model 4 (normal inputs, no shared canvas, only last layer NLL). The structure of the images are the same as in figure 5.10 and the images are also rescaled to the range of [0, 1]. The average minimum and maximum pixel values for each row are the following increasing order, respectively: $(-0.0026, 0.0739)$, $(-0.0980, 0.0176)$, $(-0.1701, 1.1257)$, $(-0.1582, 1.1402)$.

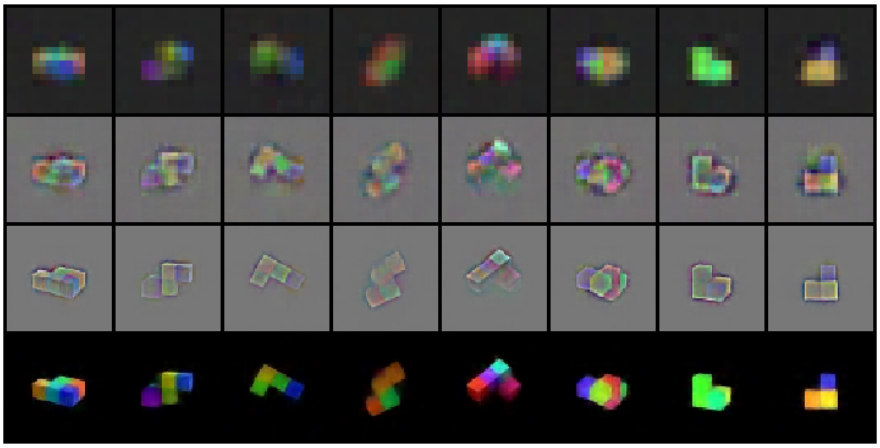


Figure 5.14: The images generated by the layers of model 5 (high frequency detail inputs). The average minimum and maximum pixel values for each row are the following increasing order, respectively: $(-0.2071, 1.2364)$, $(-0.3679, 0.4619)$, $(-0.4086, 0.4573)$, $(-0.2053, 1.1118)$.

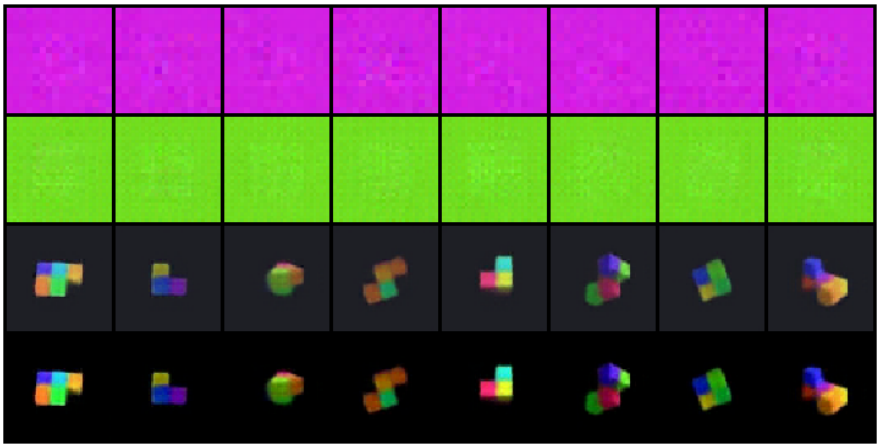


Figure 5.15: The images generated by the layers of model 6 (high frequency detail inputs, only last layer NLL). The average minimum and maximum pixel values for each row are the following increasing order, respectively: $(-0.0050, 0.0715)$, $(-0.0990, 0.0201)$, $(-0.1691, 1.1382)$, $(-0.1532, 1.1509)$.

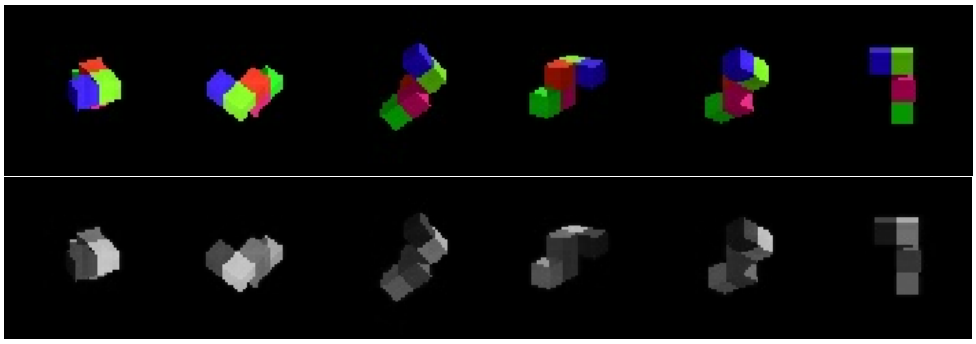


Figure 5.16: The original and gray-scale test scene (old model).

before.

5.3.1 Testing Shepard Metzler 5 models on grayscale Shepard Metzler 5 scenes

We test in this section whether the GQN model can work with grayscale scenes. We have two sets of results. The first one is the result of an old version of the single-layer GQN, which can be seen in figures 5.16 and 5.17. We trained the single-layer GQN again because the old version was trained with a wrong configuration because of a misunderstanding. It actually consists of 12 distinct layers, each having a single recurrent time step. Because this was not the intended design of the generator, i.e. it is a recurrent network whose layers need multiple time-steps to generate images in multiple steps, we have trained another GQN with 1 layer and 12 recurrent time steps which became model 0.

For clarity: the single-layer GQN we mentioned until now is model 0, the 1 layer GQN. The 12 layer GQN is only mentioned in this section.

We can see from figure 5.17 that the old model with 12 layers does pretty well for having observed a scene with unknown colours. The model does not know how to encode colours it has not seen before, causing it to render the colours incorrectly. However, the shape of the sampled image is close to the shape of the ground-truth query image, but it is missing the front-left block. This may be due to the darkness of the colour, causing the model to see it as part of the background.

Figure 5.19 shows the samples that have been made from the one-layer GQN, model 2 (shared canvas) and model 5 (high-pass). We can clearly see that model 2 is the best performing model. Although it is missing a block and the colours are incorrect, we can see that the colours it has rendered are consistent with the observed gray colours, i.e. it picks a shade of green for dark gray and purple for light gray. Model 5 is the second best out of the three, which has a similar shape to the sample of model 2 and misses the same block. Model 5 clearly has problems with colours though, as the borders between the blocks almost completely blended, while there are clear borders in the observed gray scale images. The single-layer GQN on the other hand performs very poorly, whose sample is not even close to resembling the ground truth structurally and chromatically.

By also taking the results of the 12 layer old GQN into account, these results give us the idea that generators with multiple layers can handle unseen colors better. Although, we can see by comparing model 5 (who actively uses all layers) and model 2 (who almost only uses the last layer, see figure 5.29) that the amount of layers which are actively used might not necessarily improve chromatic generality. The way the layers are connected and the final form of the generator loss function (i.e. only including NLL of the last layer or those of all layers) are clearly factors in how well the model can learn to generalize during training.

5.3.2 Testing Shepard Metzler 5 models on Shepard Metzler 5 scenes with uniformly coloured objects

This is a complementary experiment to the experiment where we tested the models on gray-scale scenes. In this experiment, we test the models on scenes containing objects of a uniform solid colour, removing details such as block borders and shades.

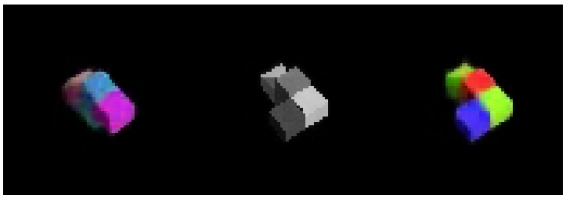


Figure 5.17: Image sampled from a gray-scale scene with the single-layer gqn (old version) trained on colored data. Left: sampled image from gray-scale scene. Center: gray-scale ground truth query image. Right: sampled image from the colored scene.

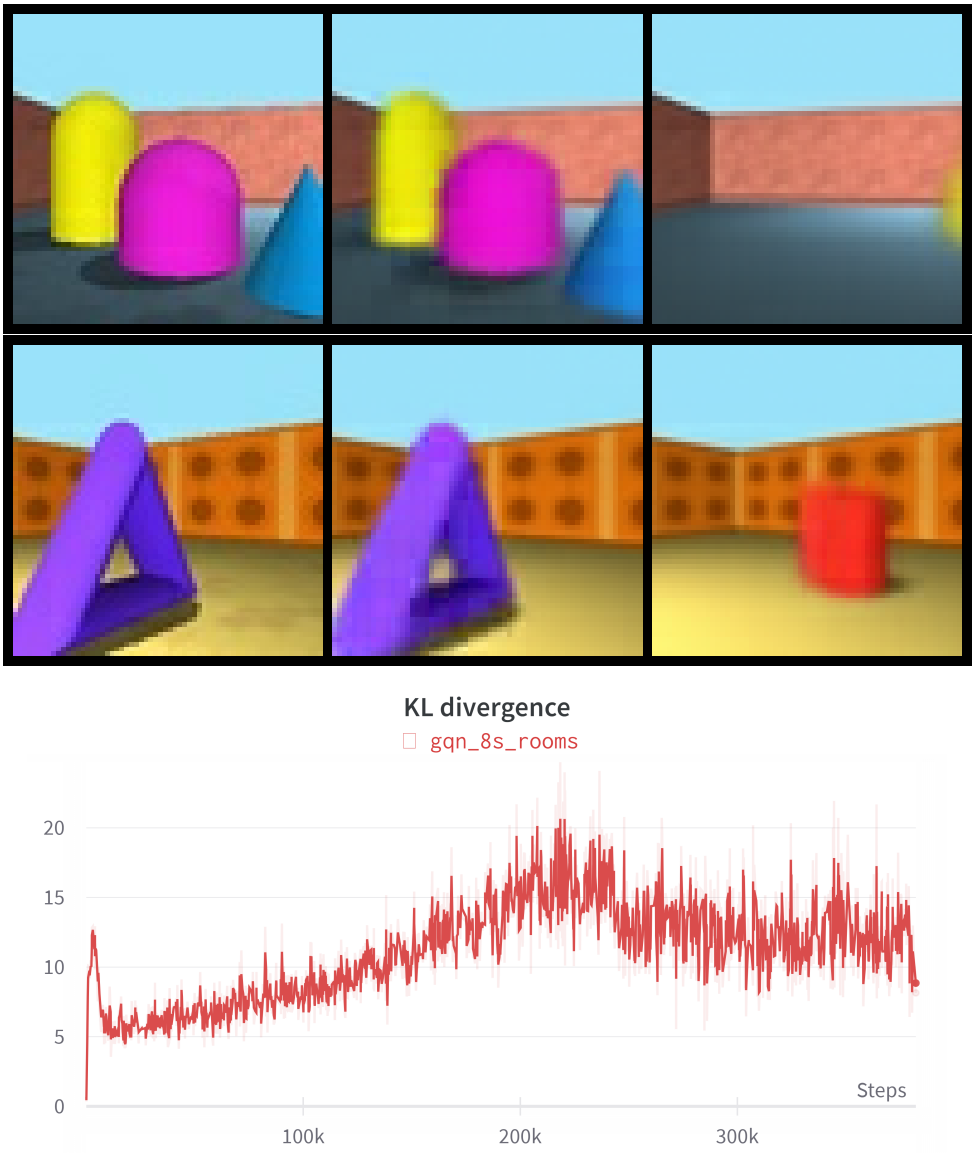
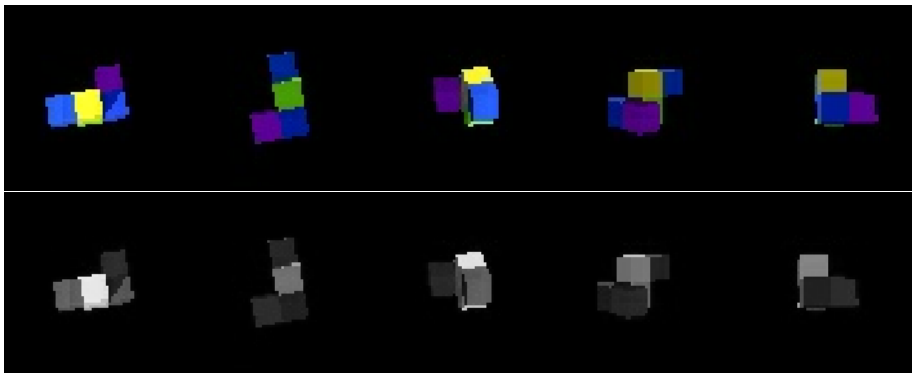
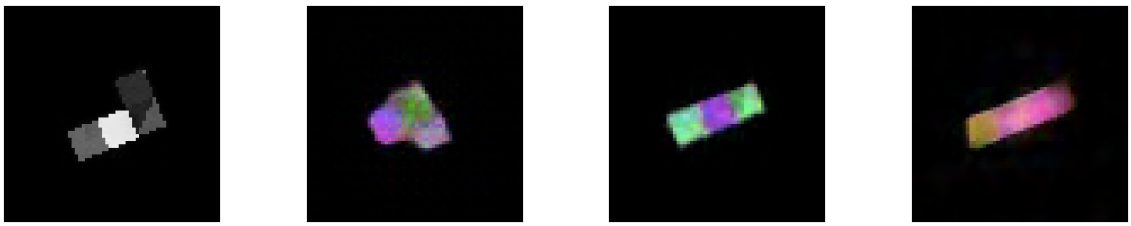


Figure 5.18: The old GQN with shared core set to false and has 8 layers



(a) The original and gray-scale observation of a test scene.



(b) From left to right: the ground-truth image, the image generated by model 0, followed by model 2 (shared canvas) then 5 (high-pass).

Figure 5.19: Testing models 0 (one layer), 2 and 5 on gray-scale scenes.

We can see the results in figure 5.20 where we can see that the single-layer GQN perform better on the green scene than model 2 (shared canvas) which was the best model with the gray-scale scene. This time, model 5 (high-pass) performs the best and generated images that are rather close to the ground-truth images, which shows that the GQN model is able to infer the shape of an object in a scene that lacks lighting details. We think that model 5 succeeded in this experiment with the help of the context viewpoints by inferring depth information from the camera position and direction.

Seeing the single-layer GQN and model 5 outperform model 2, who was the best in the gray-scale experiment, leads us to believe that the amount of generator layers do not have a direct influence on chromatic generality.

5.3.3 Testing Shepard Metzler 5 models on Shepard Metzler 7 scenes

In figure 5.22 we can see the images that were sampled from models that were trained on the 5-block Shepard Metzler dataset fail to accurately sample images on 7-block objects. Figure 5.21 shows the context images of the first scene, there are 10 observations per scene. Even though the model shapes are inaccurate, they resemble the ground truth and are not totally random, meaning that the models are able to extract and encode useful information in the representation vector. It is as though the models try to represent the 7-block objects using 5 blocks only.

5.3.4 Difficulties of training the GQN on custom datasets

To conduct rigorous tests about the generality of the models, we needed more control over the scenes. To render custom scenes, we made a program in Unity to automatically generate Shepard Metzler objects, whose user interface can be seen in figure 5.23. We first made a dataset of 32x32 images consisting of 3-block Shepard Metzler objects. When training the single-layer GQN on this dataset, we noticed that the model was unable to converge the KL divergence, just like in figure 5.9. Since we cannot know the reason behind such a behaviour directly, we set out to experiment by tuning hyperparameters one by one and observed the results.

We started out with the following configurations:

- Scene representation **r** size: 128 (since we were dealing with a smaller image size than the usual 64x64)

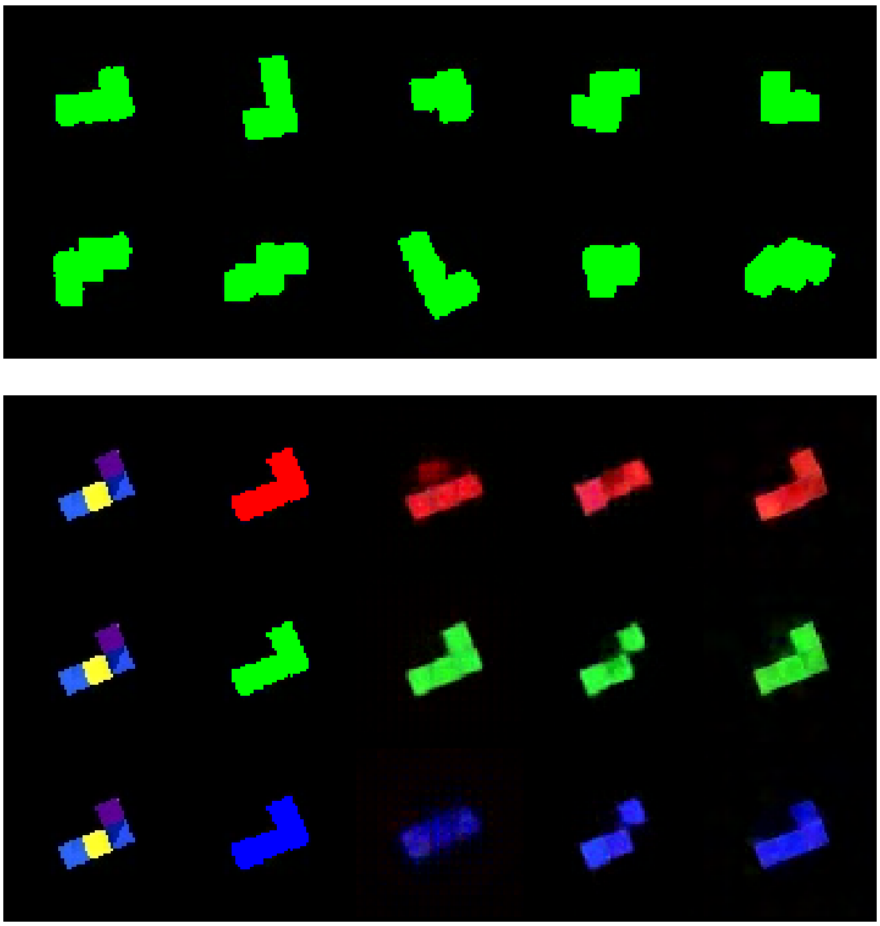


Figure 5.20: On the top we see some of the context images of the green scene, the red and blue scenes have identical shapes. On the bottom we see 5 columns which correspond with the following in order: the ground-truth original image, the ground-truth painted image, samples from the single-layer GQN, samples from model 2 and samples from model 5. For each row, the models have observed the scenes with the corresponding colors.

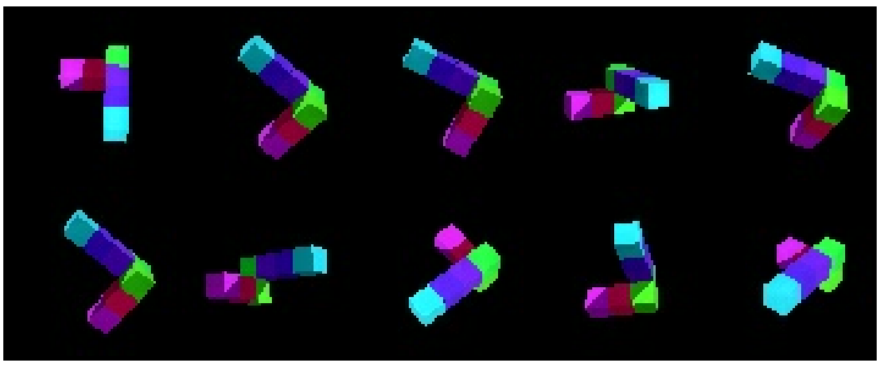


Figure 5.21: Context images of the first scene.

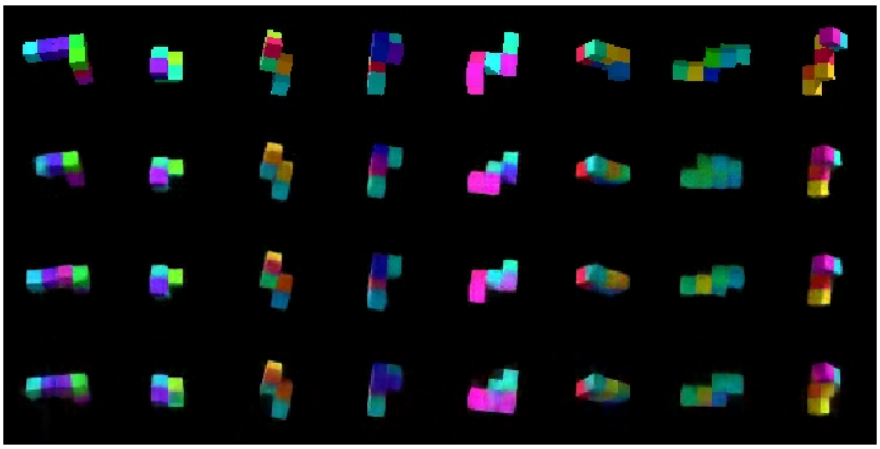


Figure 5.22: The first row contains the ground-truth query images that need to be generated. The second, third and fourth row contain the sampled images from the following models in this order: single-layer GQN, model 2 (shared canvas, upconv) and model 5 (high-pass inputs, NLL all layers). The SSIM scores of the three models averaged over 32 scenes are as follows: 1-layer GQN = 0.74183, model2 = 0.76289 and model5 = 0.68610.

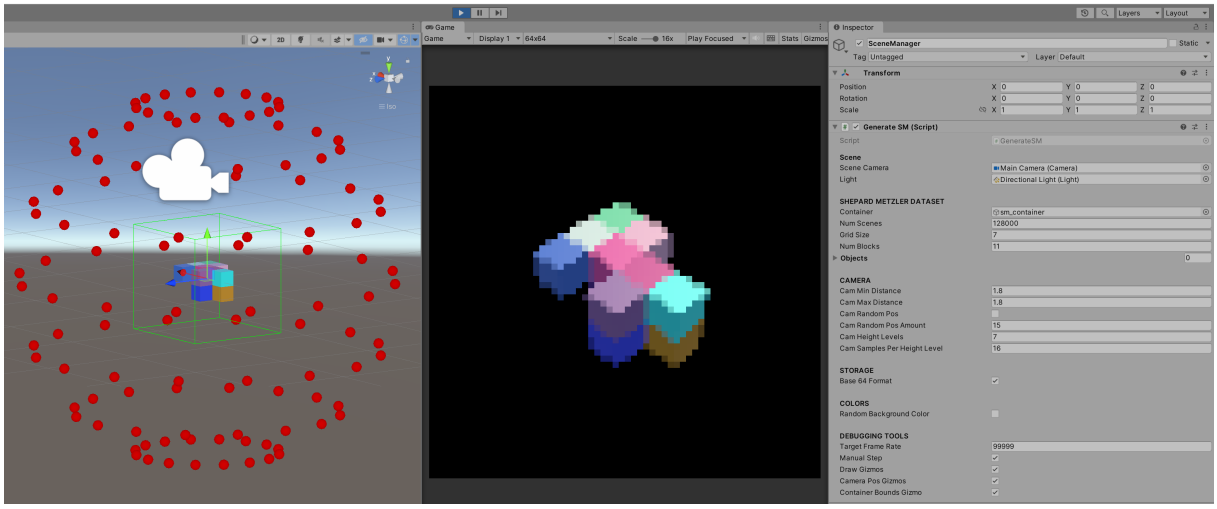


Figure 5.23: A screenshot of the Unity program for automatically generating scenes with Shepard Metzler objects. A random object can be generated whose features (number of blocks and size) can be controlled, or a custom object can be loaded in the scene to make a dataset out of. The camera positions are spherically around the observed object (as displayed by the red dots). It is also possible to sample random camera positions spherically around the object. To add further variability, we added the feature to set a minimum and maximum camera distance, the distance per camera position will be chosen uniformly random.

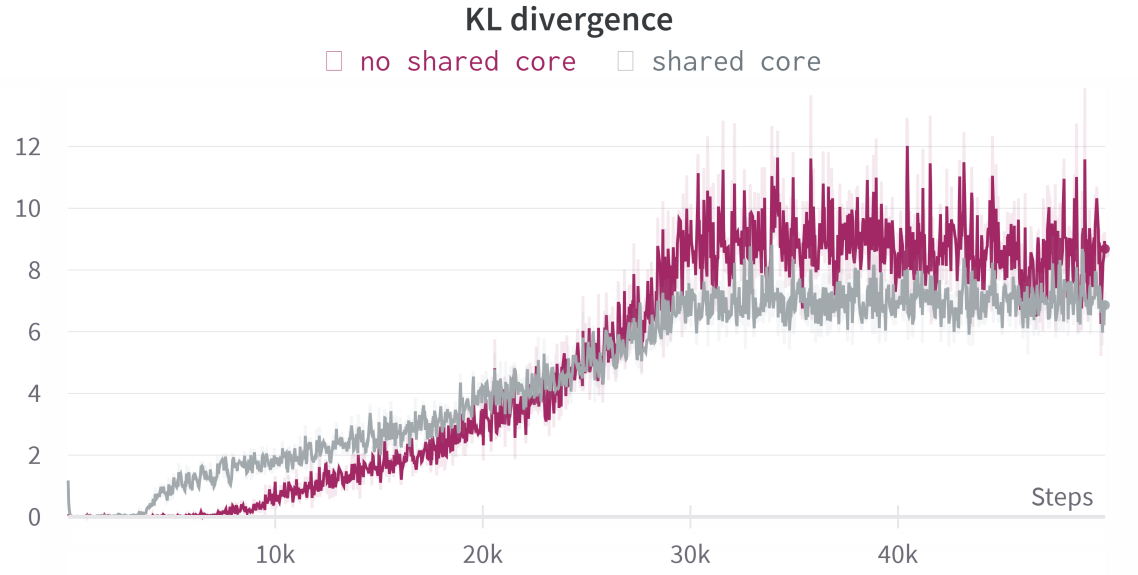


Figure 5.24: Two GQN models trained on a dummy dataset consisting of 15000 scenes of 3-block Shepard Metzler objects. The "shared core" model is a GQN with one DRAW layer with 6 time steps, while the other has 6 DRAW layers with 1 time step. The dataset was created using the tool in figure 5.23. We can see that the recurrent version is slightly more stable.

- 6 recurrent time steps
- Batch size of 24
- Annealing likelihood standard deviation (see eq. 4.10) in 30000 steps instead of 200000 steps

The first thing we had tried was to compare the model with "shared core" turned on and off. This was a boolean flag passed onto the constructor of the GQN object for controlling whether weights should be shared across time-steps. This meant that when the number of layers were set to 6 and "shared core" was set to true, the resulting GQN model had a generator consisting of a single DRAW layer with 6 recurrent time steps. Setting "shared core" to false resulted in a generator with 6 DRAW layers each having 1 recurrent time step, effectively becoming a convolutional auto-encoder.

This "shared core" parameter was present in all open-source implementations of the GQN we had access to. Since we did not want to waste time on implementing the model, not much attention was paid to the inner workings of the GQN. This caused us to think that the "shared core" parameter was a legitimate part of the GQN and that the model was intended to be trained with "shared core" set to True. However, this was not the case, we later found out that this was not the intended design for the GQN since it removed recurrence from the generator's DRAW layers by setting their max time-steps to one. This realization did not come to us until much later. Hence, this parameter is not present any more in the most recent version of our codebase, but we have conducted experiments using it (which is also how the model in figure 5.17 came to be).

We tested one model with "shared core" set to True, which had 1 DRAW layer of 6 time steps, while the other model with "shared core" set to false had 6 DRAW layers each with 1 time step. From the corresponding plot of the KL divergence of the first, 50000 training steps in figure 5.24, we can see that the recurrent version has a slightly more stable KL divergence curve. However, both KL divergences are far off from 0 and don't look like they will be converging any time soon. We excluded the sampled images because they look very much like the samples in figure 5.9, but at a lower resolution.

Next, we tried training the recurrent version on the original 5-block Shepard Metzler (SM5) dataset [Dee], which were provided by the authors of the GQN [Esl+18]. This time the model performed worse, the KL divergence reached 25 at around training step 30000 and consistently stayed there until the end of the training (step 50000). We did an additional test where we doubled the size of the representation vector from 128 to 256 and trained the model again on the SM5 dataset, this time we same an improvement

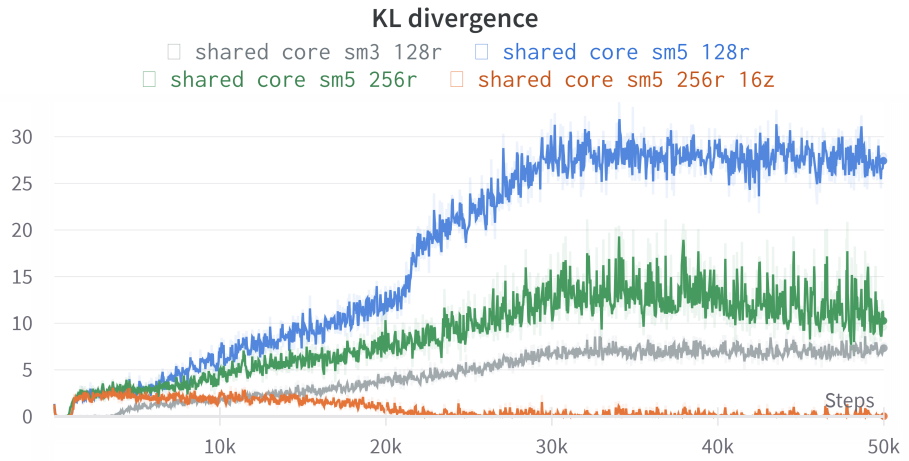


Figure 5.25: Trained the shared core version of the GQN model (gray) on the SM5 dataset with the representation size of 128 (blue) and one with double the representation size of 256 (green) which showed an improvement. The last test (orange) was done by increasing the number of channels in the latent tensor **z** from three to 16 and setting the number of steps for annealing the likelihood standard deviation back to the default of 200000 steps instead of 30000.

over the previous test. The resulting KL divergence plots can be seen in figure 5.25. We can clearly see that the model struggles with learning a proper prior distribution. If it was succeeding in doing that, the KL divergences would converge close to zero, as in figure 5.2.

These observations led us to think that the capacity of the model just was not enough to handle the variety and complexity of the SM5 dataset. For this we increased the amount of channels of the latent tensor **z** from three to 16, and we also increased the number of annealing steps for the likelihood standard deviation (see eq. 4.10) from 30000 steps to 200000 steps which was the default value recommended by the authors of the GQN [Esl+18]. This time the model succeeded on converging the KL divergence to zero, and we continued training the same model until training step, 300000 with 12 recurrent time steps. This became model 0, the single layer GQN that we have compared against the multi-layer GQNs in the previous experiment sections.

Now that the model got successfully trained on the given SM5 dataset, we tried it again on the custom SM3 dataset consisting of 15000 scenes of 32x32 images. The only difference was that we reduced the batch size from 24 to 16 for updating the gradients more frequently and the annealing steps to 100000 thinking that simpler scenes and a smaller dataset should require less training steps for convergence since each scene would be seen again by the model each 938 training steps. The results were unsatisfactory due to the KL divergence not converging again, affecting the sampling performance. We adjusted the hyperparameters again step by step until we managed to converge the model on the custom SM3 dataset. These were the steps that were taken to solve the issue:

1. Adjusted the custom dataset such that the centre of the observed object is always close to the origin and the means of the camera positions is close to zero. Besides the difference in the number of scenes, the difference between the custom SM3 dataset and the given SM5 dataset was that the means of the camera positions of SM5 were close to zero and all camera distances were 3.333. This did not improve the results.

2. We trained on a subset of the SM5 dataset consisting of 15000 scenes, because we thought that 15000 scenes should be enough to produce somewhat acceptable results and that the problem was potentially with how the custom scenes were captured (bad randomness of colours, difference camera parameters, absence of noise, etc.). This did not improve the results also.

3. Next we tried to train the model back on the SM5 dataset, which surprisingly did not work, the KL divergence was misbehaving again.

4. Next we conducted two more tests to change back the only differences that were left: first the amount of annealing steps were increased to 200000 again followed by a second test where we

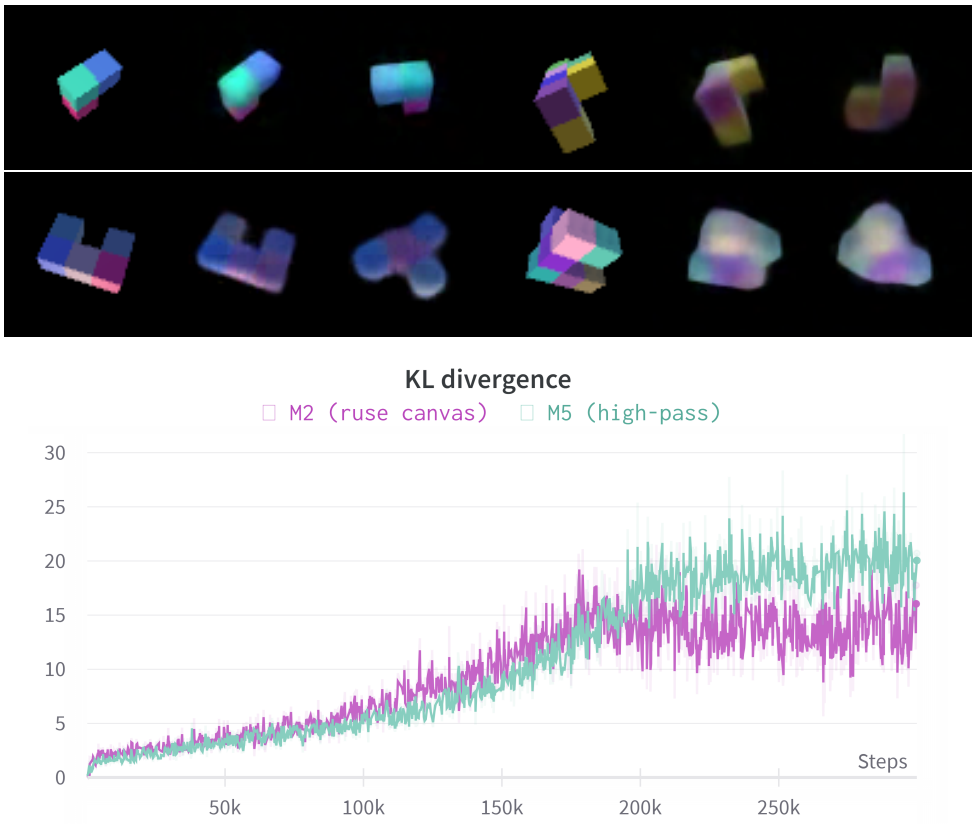


Figure 5.26: KL divergences and some samples during the training of models 2 (shared canvas) and 5 (high-pass) trained on the mixed SM dataset. As mentioned before: the first of the three images is the ground-truth, the second one is sampled using the posterior (inference process) and the third one using the prior (sampling process).

increased the batch size back from 16 to 24. After the second test, the model started converging again.

5. The next obvious test would be to test the model again on the custom SM3 dataset of 15000 scenes, which did not converge. This left one last obvious option as to why the model still did not converge: 15000 scenes were not enough.

6. Finally, we created a new 32x32 SM3 dataset, but this time consisting of 300000 scenes. The model finally succeeded on converging.

From these steps, we can conclude that multiple hyperparameters need to be set to proper values and a sufficiently large dataset needs to be provided in order for GQN models to converge properly.

Since the models trained on the given SM5 datasets could not handle the scenes generated with the scene generation tool we developed, we tried to make our own dataset to train model for better GQN generality experiments. We made a mixed dataset with a resolution of 64x64 which consists of 124000 scenes of 1-block, 124000 scenes of 3-block, 300000 scenes of 5-block and 300000 scenes of 7-block Shepard Metzler objects, with a total of 848000 scenes. We trained model 2 (reuse canvas) and model 5 (high-pass) on this dataset with the same training configuration we used for training the models on the given SM5 dataset. Unfortunately, the same problem with the KL divergence occurred again where it kept increasing as illustrated in figure 5.26, probably caused due to the amount of scenes not being enough.

The next obvious steps were to generate different kinds of datasets (e.g. combination of SM4 and SM5 instead of SM1, 3, 5, and 7), hyperparameter tuning techniques to explore different sets of potentially better hyperparameters for training the models and normalization techniques to ensure that the inputs were always within a certain range. But due to time running out, the experiments had to be halted at this point.

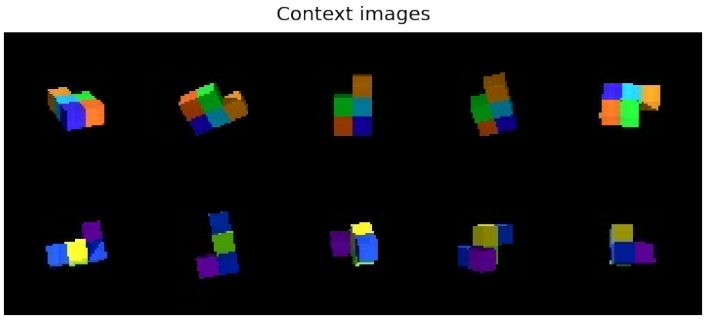


Figure 5.27: The first five context images of the two scenes, row 1 = scene 1, row 2 = scene 2.

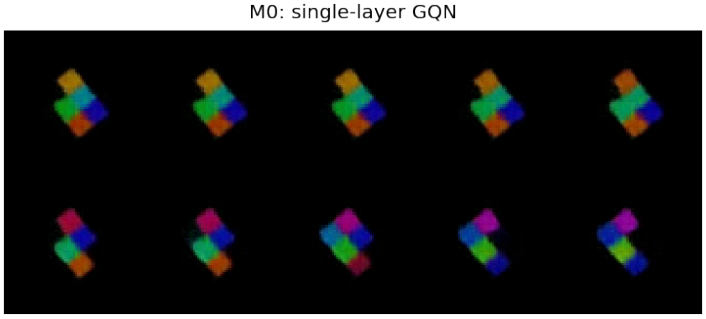


Figure 5.28: Samples generated from interpolated scene representation vectors with the single-layer GQN. The top-left image is sampled using the unchanged representation of scene 1, while the image to the bottom-right is sampled from the representation of scene 2. The images in between are sampled from an interpolation of the two representations.

5.4 Interpolating representation vectors

An important characteristic of variational autoencoder is the latent space that they learn tends to be continuous, making operations such as interpolating latent vectors to obtain useful outputs possible. If a latent space is learned with a proper structure, then it can be further used to facilitate other tasks by reducing the dimensionality of the feature space for other problems, effectively reducing the impact of the curse of dimensionality. Two of the most obvious applications, among others, of such a property could be training classifiers on the latent space instead of images and generating images given some condition (what the GQN does). To this end, we will look at how the scene representation, one of the conditioning variables for the GQN generator, affects the generated images.

Here, we pick two scenes from the training dataset and sample images from representation vectors that are attained by linearly interpolating between the representation vectors of the two scenes. We can see the results generated by the single-layer GQN, model 2 (shared canvas) and model 5 (high-pass) in figures 5.28, 5.29 and 5.30 respectively. The top-left image is sampled using the unchanged representation of scene 1, while the image to the bottom-right is sampled from the representation of scene 2. The images in between are sampled from an interpolation of the two representations.

An interesting observation is that all images that are sampled from the interpolated representation are valid 5-block Shepard Metzler objects. The sampled images of two consecutive interpolated representations are either exactly the same or have a clear difference, i.e. either the colour of a block changes, either a block disappears or a new one appears. The transition from the first scene to the second scene almost seems logical. We can clearly see in figure 5.28 that there is almost no difference between the first six samples, suggesting that each block configuration corresponds with a continuous range of values in the representation space, which is a desirable property of learned representation spaces.

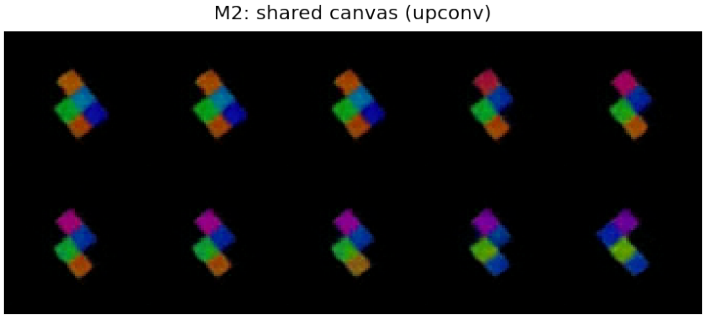


Figure 5.29: Samples generated from interpolated scene representation vectors with model 2 (shared canvas, upconv).

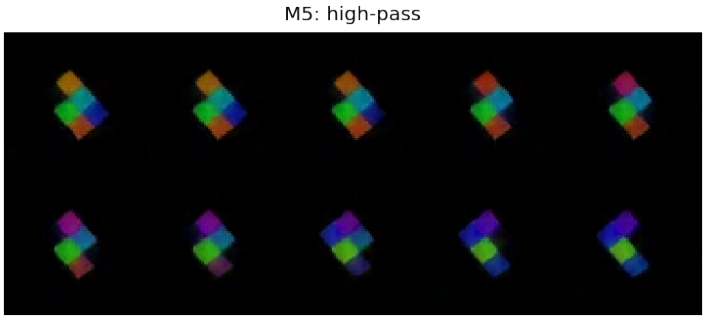


Figure 5.30: Samples generated from interpolated scene representation vectors with model 5 (high-pass).

Chapter 6

Conclusion

The goal of this thesis was to look into Generative Query Networks (GQN) [Esl+18] and get a better understanding of how these models work and what their capabilities are. Through the course of this thesis, we have described some fundamental concepts in Deep Learning which form the basis of the GQN. We started with Recurrent neural networks, a basic yet powerful set of tools which enabled the GQN to generate images in an iterative manner. Next, we described different kinds of generative models like the Variational Autoencoders [KW22] and Deep Recurrent Attentive Writers [Gre+15] which were the basis for the GQN's generator. After we covered the necessary foreknowledge, we gave a detailed description of how the GQN model works and even introduced our own extension of the GQN's generator: we proposed different methods to chain the DRAW layers together to obtain a working multi-layer generator for the GQN.

We made use of the fact that rough, structural details in images can also be stored in lower resolutions, so the multi-layer generator we proposed consisted of a chain of DRAW layers that increased in resolution the further along they were on the chain. For this reason, we have named this configuration multi-scale generators. This way, in theory, the earlier layers could focus on generating rough details while the later layers could improve on the results of the previous layers and add more details until an image at the desired resolution was generated. However, we have seen in our experiments that this is not always the case. We observed that multi-scale generators tend to not use some of their layers if they don't need to (see section 5.2.3). If the capacity of a single layer is enough to generate the desired image, the visual contribution of the previous layers will be next to nothing, letting the last layer generate the whole image. We also trained two additional models (models 3 and 5, defined in section 5.2.1) where we included the negative log-likelihood of the output of each generator layer with respect to their inputs to the total loss function to force each layer to learn the inputs they received. The layers of model 3 received normal RGB images as inputs that were downscaled to the right resolution, while the layers of model 5 received the high-frequency details of those input images. Model 5 was the more interesting one of the two because its layers effectively became reverse low-pass filters, i.e. they learned to compute missing details of the inputs they received.

We have looked into the GQN's ability to generalize what it has learned during the training process to handle out-of-distribution scenes in practice. We looked at its ability to handle unseen colours and shapes separately. We tested the single-layer version of the model that was proposed by the GQN's authors and two GQN models with multi-scale generators, i.e. models 2 (whose canvas was shared among its layers) and 5 (the model that learned the reverse low-pass filter). We observed that the single layer GQN was the weakest in handling gray-scale images, but did not come last when handling scenes single-color objects. We also tested the same models that were trained on 5-block Shepard Metzler objects on scenes that contained 7-block Shepard Metzler objects. Although none of the models could produce fully accurate results because they were trying to render 7-block objects with 5 blocks, their results were still strikingly similar to the ground-truth images. This shows that GQN models were able to generalize structural information to some degree, enabling them to encode and generate structural information that looks very similar to that of the ground-truth image.

Finally, we looked at whether the representation space of the GQN was continuous, like the latent space of variational autoencoders. We did this by interpolating a representation vector and observed that

the images generated from the interpolated values were not random. In fact, the only visible changes were either blocks appearing or disappearing from the rendered images, or some of the block colours changing gradually. Although this experiment alone cannot serve as a proof, it is an indication that the representation space is continuous, where each value in that space causes the generation of a valid scene (a scene that conforms to the structures observed in the training dataset).

While working towards solutions for finding answers to the research questions, many problems have been faced. The severity of these problems ranged from trivial to critical. Some trivial problems were dealing to very large dataset sizes that spanned from a dozen to a few hundred gigabytes, writing scripts that could create and process data into training datasets in a reasonable amount of time or finding a reliable way to keep track of the training experiments conducted so far. Some more difficult and pressing problems were the creation of a future-proof pipeline, the training speeds and the memory requirements of the models, which meant we were limited in how much we could scale the models up if we wanted to, or being sure of the correctness of the available GQN implementations. A critical problem that occurred was the indeterministic bug in the GQN code, which caused the model to sometimes work and sometimes fail after the training phase, even if the same exact configuration was used to train the model. Running the same exact training script back to back would sometimes yield a model that worked and sometimes one that just produced a black image.

Even though the difficulty of each faced problem differed greatly, we noticed something that was common to each of the problems: they all took much more time than anticipated. Many of the problems or tasks that need to be solved were related to problems and tasks in some way. Often, when trying to tackle one problem, multiple different problems would emerge. For example, when a model is trained, we would want to test how well it performs and conduct experiments on it. These experiments are controlled easier in case custom scenes could be rendered. One could set out to create a quick, minimal Unity project to just render one specific scene to do a quick experiment. While developing a program to generate datasets, other problems occur like what the structure of this dataset will be, how it will be converted to a training dataset, in what format the generated scenes would be stored on the disk before being converted into a training dataset. Trying to tackle these problems without clear short-term and long-term goals would cause one to get lost in trying to solve all these problems without any order. Clear goals were not set during the early months of working on this thesis, which caused stagnation in the implementation of a proper pipeline early on. This problem was exacerbated due to the fact of learning the details and inner workings of the GQN model were avoided until the second semester, due to "not wanting to waste time on understanding finer details about a model since there are multiple online implementations available". This statement may have been valid if the implementations were well-tested and the model at hand was well known with a lot of resources available on how to train the model successfully. Unfortunately, this was not the case and was decided, very late into the second semester, to learn how the model works and implement it from scratch. After having implemented the model, a lot more things became clear, the conceptual error of the "shared core" flag was spotted which led to the idea of multi-scale generators.

The "shared core" flag was a boolean value used to change the hierarchy of the generator. It was meant for controlling whether weights should be shared across time-steps in the GQN's generator. This meant that when the number of layers were set to 6 and "shared core" was set to true, the resulting GQN model had a generator consisting of a single DRAW layer with 6 recurrent time steps. Setting "shared core" to false resulted in a generator with 6 DRAW layers each having 1 recurrent time step, effectively making it a convolutional auto-encoder. Since the generator of the GQN is based on the DRAW architecture, which is explicitly designed to work with recurrent time steps, setting this flag to false would go against the concept of the DRAW model. This flag was present in multiple open-source implementations of the GQN, which we had based our code off of [iSh] [Mwu] [Woh], so we thought it was a legitimate part of the GQN model. We trained this model on the Shepard Metzler dataset with 5-block objects, where it performed really well, but did less well with the medium "rooms" dataset [Dee] (see figure 5.18).

In short: when working on a deep learning project, it is worth it to understand the details and inner workings of the model and the methods involved in training it. Since this was not done in this thesis, a lot of time has been wasted which affected the quality and quantity of the experiments that have been conducted due to the less amount of remaining time, which is evidence from the aforementioned conclusions about the experiments.

Had this conclusion been implemented from the start, the direction of the research in this thesis might have been gone in a different direction: the comparison between different conditional generative models

within the context of the problem of neural scene representation and rendering.

Chapter 7

Conclusie

Het doel van deze thesis was om Generative Query Networks (GQN) [Esl+18] te onderzoeken en een beter begrip te krijgen van hoe deze modellen werken en wat hun mogelijkheden zijn. Gedurende het verloop van deze thesis hebben we enkele fundamentele concepten in Deep Learning beschreven die de basis vormen van de GQN. We begonnen met Recurrent neural networks, een eenvoudige maar krachtige reeks tools die de GQN in staat stelden om afbeeldingen iteratief te genereren. Vervolgens hebben we verschillende soorten generatieve modellen beschreven, zoals de Variational Autoencoders [KW22] en Deep Recurrent Attentive Writers [Gre+15], die de basis vormden voor de generator van de GQN. Nadat we de benodigde voorkennis hadden behandeld, hebben we een gedetailleerde beschrijving gegeven van hoe het GQN-model werkt en hebben we zelfs onze eigen uitbreiding van de generator van de GQN geïntroduceerd: we hebben verschillende methoden voorgesteld om de DRAW-lagen aan elkaar te schakelen om een werkende multi-layer generator voor de GQN te verkrijgen.

We hebben gebruik gemaakt van het feit dat ruwe, structurele details in afbeeldingen ook op lagere resoluties kunnen worden opgeslagen. Daarom bestond de door ons voorgestelde multi-layer generator uit een reeks DRAW-layers die in resolutie toenamen naarmate ze verder waren op de keten. Om deze reden hebben we deze configuratie multi-scale generators genoemd. Op deze manier zouden de vroegere lagen zich theoretisch kunnen concentreren op het genereren van ruwe details, terwijl de latere lagen de resultaten van de vorige lagen konden verbeteren en meer details konden toevoegen totdat een afbeelding op de gewenste resolutie was gegenereerd. We hebben echter in onze experimenten gezien dat dit niet altijd het geval is. We hebben waargenomen dat multi-scale generators soms bepaalde lagen niet gebruiken als dat niet nodig is (zie sectie 5.2.3). Als de capaciteit van een enkele laag voldoende is om de gewenste afbeelding te genereren, zal de visuele bijdrage van de vorige lagen bijna niet te waarnemen zijn, waardoor de laatste laag de hele afbeelding genereert. We hebben ook twee aanvullende modellen getraind (modellen 3 en 5, gedefinieerd in sectie 5.2.1), waarbij we de negatieve log-likelihood van de output van elke generatorlaag ten opzichte van hun inputs aan de totale verliesfunctie hebben toegevoegd om elke laag te dwingen de inputs te leren die ze ontvingen. De lagen van model 3 ontvingen normale RGB-afbeeldingen als inputs die werden verkleind naar de juiste resolutie, terwijl de lagen van model 5 de hoogfrequente details van die invoerafbeeldingen ontvingen. Model 5 was het interessantere van de twee, omdat de lagen ervan effectief omgekeerde low-pass filters werden, d.w.z. ze hebben geleerd om ontbrekende details van de ontvangen inputs te berekenen.

We hebben onderzocht in hoeverre de GQN in staat was om hetgeen het tijdens het trainingsproces heeft geleerd, te generaliseren om ook met out-of-distribution scènes in de praktijk om te gaan. We hebben gekeken naar zijn vermogen om om te gaan met ongeziene kleuren en vormen afzonderlijk. We hebben het enkelvoudige laag model getest dat werd voorgesteld door de auteurs van de GQN en twee GQN-modellen met multi-scale generators, namelijk modellen 2 (waarvan het canvas werd gedeeld tussen de lagen) en 5 (het model dat de omgekeerde low-pass filter heeft geleerd). We hebben waargenomen dat de enkelvoudige laag GQN het zwakst was in het omgaan met grijswaardenafbeeldingen, maar niet als laatste eindigde bij het verwerken van scènes met éénkleurige objecten. We hebben ook dezelfde modellen getest die waren getraind op Shepard Metzler-objecten met 5 blokken op scènes die 7-block Shepard Metzler-objecten bevatten. Hoewel geen van de modellen volledig nauwkeurige resultaten kon produceren omdat ze probeerden 7-block objecten te renderen met 5 blokken, waren hun resultaten toch opvallend vergelijkbaar met de grondwaarheidafbeeldingen. Dit toont aan dat GQN-modellen structurele

informatie tot op zekere hoogte kunnen generaliseren, waardoor ze in staat zijn om structurele informatie te coderen en genereren die zeer vergelijkbaar is met die van de grondwaarheidafbeelding.

Ten slotte hebben we gekeken of de representatieruimte van de GQN continu is, zoals de latente ruimte van variational autoencoders. We hebben dit gedaan door een representatievector te interpoleren en waargenomen dat de afbeeldingen die gegenereerd zijn uit de geïnterpoleerde waarden niet willekeurig waren. In feite waren de enige zichtbare veranderingen het verschijnen of verdwijnen van blokken in de gegenereerde afbeeldingen, of sommige van de blokkleuren die geleidelijk veranderden. Hoewel dit experiment op zichzelf niet als bewijs kan dienen, is het een aanwijzing dat de representatieruimte continu is, waarbij elke waarde in die ruimte de generatie van een geldige scène veroorzaakt (een scène die overeenkomt met de structuren die zijn waargenomen in de trainingsdataset).

Tijdens het werken aan oplossingen om antwoorden te vinden op de onderzoeksvragen, zijn veel problemen opgetreden. De ernst van deze problemen varieerde van triviaal tot kritiek. Sommige triviale problemen waren het omgaan met zeer grote datasetgroottes die varieerden van enkele tientallen gigabytes tot enkele honderden gigabytes, het schrijven van scripts die gegevens konden creëren en verwerken tot trainingsdatasets in een redelijke hoeveelheid tijd, of het vinden van een betrouwbare manier om de tot nu toe uitgevoerde trainingsexperimenten bij te houden. Sommige moeilijkere en dringendere problemen waren de creatie van een toekomstbestendige pipeline, de trainingsnelheden en de geheugenvereisten van de modellen, wat betekende dat we beperkt waren in hoeverre we de modellen konden opschalen als we dat wilden, of zeker te zijn van de juistheid van de beschikbare GQN-implementaties. Een kritisch probleem dat zich voordeed, was de onvoorspelbare bug in de GQN-code, die ertoe leidde dat het model soms werkte en soms faalde na de trainingsfase, zelfs als dezelfde exacte configuratie werd gebruikt om het model te trainen. Dezelfde trainingsopdracht exact achter elkaar uitvoeren zou soms een model opleveren dat werkte en soms een model dat alleen een zwarte afbeelding produceerde.

Hoewel de moeilijkheidsgraad van elk problemen sterk verschilde, merkten we iets op dat gemeenschappelijk was voor al deze problemen: ze kostten allemaal veel meer tijd dan verwacht. Veel van de problemen of taken die moesten worden opgelost, hadden op de een of andere manier verband met elkaar. Vaak zouden bij het proberen van één probleem meerdere verschillende problemen ontstaan. Bijvoorbeeld, bij het trainen van een model, zouden we willen testen hoe goed het presteert en experimenten ermee uitvoeren. Deze experimenten zijn gemakkelijker te controleren als aangepaste scènes kunnen worden weergegeven. Men zou kunnen besluiten om een snel, minimaal Unity-project te maken om gewoon één specifieke scène weer te geven om een snel experiment uit te voeren. Bij het ontwikkelen van een programma om datasets te genereren, ontstaan er andere problemen zoals wat de structuur van deze dataset zal zijn, hoe het zal worden omgezet naar een trainingsdataset, in welk formaat de gegenereerde scènes op de schijf zouden worden opgeslagen voordat ze worden omgezet naar een trainingsdataset. Proberen deze problemen aan te pakken zonder duidelijke korte- en langetermijndoelen zou ertoe leiden dat men verdwaalt in het proberen van al deze problemen op te lossen zonder enige ordening. Er werden geen duidelijke doelen gesteld tijdens de eerste maanden van het werken aan deze scriptie, wat leidde tot stagnatie bij het implementeren van een degelijke pipeline in een vroeg stadium. Dit probleem werd verergerd door het vermijden van het leren van de details en de innerlijke werking van het GQN-model tot het tweede semester, vanwege het "niet willen verspillen van tijd aan het begrijpen van fijnere details over een model omdat er meerdere online implementaties beschikbaar zijn." Deze verklaring zou geldig kunnen zijn geweest als de implementaties goed getest waren en het model zelf goed bekend was met veel beschikbare bronnen over hoe het model met succes getraind kon worden. Helaas was dit niet het geval en werd pas zeer laat in het tweede semester besloten om te leren hoe het model werkt en het vanaf nul te implementeren. Nadat het model was geïmplementeerd, werd veel duidelijker, werd de conceptuele fout van de "gedeelde kern" vlag opgemerkt, wat leidde tot het idee van multi-scale generators.

De "gedeelde kern" vlag was een booleaanse waarde die werd gebruikt om de hiërarchie van de generator te veranderen. Het was bedoeld om te bepalen of gewichten over tijdstappen heen in de generator van de GQN gedeeld moesten worden. Dit betekende dat wanneer het aantal lagen was ingesteld op 6 en "gedeelde kern" was ingesteld op waar, het resulterende GQN-model een generator had die bestond uit één DRAW-laag met 6 recurrente tijdstappen. Het instellen van "gedeelde kern" op onwaar resulteerde in een generator met 6 DRAW-lagen, elk met 1 recurrent tijdstap, waardoor het effectief een convolutioneel auto-encoder werd. Aangezien de generator van de GQN gebaseerd is op de DRAW-architectuur, die expliciet is ontworpen om te werken met recurrente tijdstappen, zou het instellen van deze vlag op onwaar ingaan tegen het concept van het DRAW-model. Deze vlag was aanwezig in meerdere open-source implementaties van de GQN, waarop we onze code hadden gebaseerd [iSh] [Mwu] [Woh], dus we

dachten dat het een legitiem onderdeel was van het GQN-model. We hebben dit model getraind op de Shepard Metzler-dataset met 5-block objecten, waar het heel goed presteerde, maar minder goed met de medium "kamers" dataset [Dee] (zie figuur 5.18).

Kortom: bij het werken aan een deep learning-project is het de moeite waard om de details en innerlijke werking van het model en de gebruikte trainingsmethoden te begrijpen. Omdat dit in deze thesis niet is gedaan, is er veel tijd verspild die de kwaliteit en kwantiteit van de uitgevoerde experimenten heeft beïnvloed vanwege de beperkte hoeveelheid resterende tijd, wat blijkt uit de eerdergenoemde conclusies over de experimenten.

Als deze conclusie vanaf het begin was geïmplementeerd, zou de richting van het onderzoek in deze thesis mogelijk in een andere richting zijn gegaan: de vergelijking tussen verschillende conditionele generatieve modellen in het kader van het probleem van neurale scènerepresentatie en weergave.

Bibliography

[23] July 2023. URL: https://en.wikipedia.org/wiki/Difference_of_Gaussians (visited on 08/23/2023).

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. arXiv: 1701.07875 [stat.ML].

[BEN45] ARTHUR L. BENTON. "A VISUAL RETENTION TEST FOR CLINICAL USE". In: *Archives of Neurology and Psychiatry* 54.3 (Sept. 1945), pp. 212–216. ISSN: 0096-6754. DOI: 10.1001/archneurpsyc.1945.02300090051008. eprint: https://jamanetwork.com/journals/archneurpsyc/articlepdf/650143/archneurpsyc_54_3_008.pdf. URL: https://doi.org/10.1001/archneurpsyc.1945.02300090051008.

[Bra+08] Timothy F. Brady et al. "Visual long-term memory has a massive storage capacity for object details". In: *Proceedings of the National Academy of Sciences* 105.38 (2008), pp. 14325–14329. DOI: 10.1073/pnas.0803390105. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.0803390105. URL: https://www.pnas.org/doi/abs/10.1073/pnas.0803390105.

[Cao+23] Hanqun Cao et al. *A Survey on Generative Diffusion Model*. 2023. arXiv: 2209.02646 [cs.AI].

[Cho] Francois Chollet. *Building Autoencoders in Keras — blog.keras.io*. https://blog.keras.io/building-autoencoders-in-keras.html. [Accessed 28-Jun-2023].

[Dee] Deepmind. *Deepmind/gqn-datasets: Datasets used to train generative query networks (gqns) in the "neural scene representation and rendering" paper*. URL: https://github.com/deepmind/gqn-datasets.

[Dew+21] Christine Dewi et al. "Various Generative Adversarial Networks Model for Synthetic Prohibitory Sign Image Generation". In: *Applied Sciences* 11 (Mar. 2021), p. 2913. DOI: 10.3390/app11072913.

[DN21] Prafulla Dhariwal and Alex Nichol. *Diffusion Models Beat GANs on Image Synthesis*. 2021. arXiv: 2105.05233 [cs.LG].

[Esl+18] S. M. Ali Eslami et al. "Neural scene representation and rendering". In: *Science* 360.6394 (2018), pp. 1204–1210. DOI: 10.1126/science.aar6170. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6170. URL: https://www.science.org/doi/abs/10.1126/science.aar6170.

[Far+16] Fernando A. Fardo et al. *A Formal Evaluation of PSNR as Quality Measurement Parameter for Image Segmentation Algorithms*. 2016. arXiv: 1605.07116 [cs.CV].

[Gar+17] Alberto Garcia-Garcia et al. *A Review on Deep Learning Techniques Applied to Semantic Segmentation*. 2017. arXiv: 1704.06857 [cs.CV].

[Gre+15] Karol Gregor et al. *DRAW: A Recurrent Neural Network For Image Generation*. 2015. arXiv: 1502.04623 [cs.CV].

[Gre+16] Karol Gregor et al. *Towards Conceptual Compression*. 2016. arXiv: 1604.08772 [stat.ML].

[GB19] David Griffiths and Jan Boehm. "A Review on Deep Learning Techniques for 3D Sensed Data Classification". In: *Remote Sensing* 11.12 (June 2019), p. 1499. DOI: 10.3390/rs11121499. URL: https://doi.org/10.3390%2Frs11121499.

[Gro] Aditya Grover. *Contents — deepgenerativemodels.github.io*. https://deepgenerativemodels.github.io/notes/. [Accessed 28-Jun-2023].

[Gu+22] Shuyang Gu et al. *Vector Quantized Diffusion Model for Text-to-Image Synthesis*. 2022. arXiv: 2111.14822 [cs.CV].

[He+15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[Hig+16] Irina Higgins et al. *Early Visual Concept Learning with Unsupervised Deep Learning*. 2016. arXiv: 1606.05579 [stat.ML].

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. *Denoising Diffusion Probabilistic Models*. 2020. arXiv: 2006.11239 [cs.LG].

[Ho+22] Jonathan Ho et al. *Video Diffusion Models*. 2022. arXiv: 2204.03458 [cs.CV].

[HZ10] Alain Hore and Djemel Ziou. "Image quality metrics: PSNR vs. SSIM". In: *2010 20th International Conference on Pattern Recognition* (2010). DOI: 10.1109/icpr.2010.579.

[Hos] Hossein. *Why is reparameterization trick necessary for variational autoencoders? — stats.stackexchange.com*. https://stats.stackexchange.com/questions/429315/why-is-reparameterization-trick-necessary-for-variational-autoencoders. [Accessed 29-Jun-2023].

[Hui23] Jonathan Hui. *Gan - why it is so hard to train generative adversarial networks!* Apr. 2023. URL: https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b (visited on 08/23/2023).

[iSh] iShohei220. *ISHOHEI220/torch-gqn: Pytorch implementation of Generative Query Network*. URL: https://github.com/iShohei220/torch-gqn/tree/master (visited on 08/21/2023).

[JJ00] Tommi S. Jaakkola and Michael I. Jordan. In: *Statistics and Computing* 10.1 (2000), pp. 25–37. DOI: 10.1023/a:1008932416310. URL: https://doi.org/10.1023/a:1008932416310.

[Kan+23] Minguk Kang et al. *Scaling up GANs for Text-to-Image Synthesis*. 2023. arXiv: 2303.05511 [cs.CV].

[Kar+20] Tero Karras et al. *Analyzing and Improving the Image Quality of StyleGAN*. 2020. arXiv: 1912.04958 [cs.CV].

[KW22] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML].

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[Kri09] KAlex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. MaRS Centre, West Tower, 661 University Avenue, Suite 505: Canadian Institute for Advanced Research, Apr. 2009.

[Kum+19] Ananya Kumar et al. *Consistent Generative Query Networks*. 2019. arXiv: 1807.02033 [cs.CV].

[Led+17] Christian Ledig et al. *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*. 2017. arXiv: 1609.04802 [cs.CV].

[Lin+15] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[Liu+16] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2. URL: https://doi.org/10.1007%2F978-3-319-46448-0_2.

[LH08] Steven J. Luck and Andrew Hollingworth. *Visual Memory*. Oxford University Press, Aug. 2008. ISBN: 9780195305487. DOI: 10.1093/acprof:oso/9780195305487.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780195305487.001.0001.

[Lug+22] Andreas Lugmayr et al. *RePaint: Inpainting using Denoising Diffusion Probabilistic Models*. 2022. arXiv: 2201.09865 [cs.CV].

[MO14] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. arXiv: 1411.1784 [cs.LG].

[Mwu] Mwufi. *MWUFI/Pytorch-gqn: Neural scene rendering! with helpful comments and a video -gt;* URL: https://github.com/mwufi/pytorch-gqn/tree/master (visited on 08/21/2023).

[ND21] Alex Nichol and Prafulla Dhariwal. *Improved Denoising Diffusion Probabilistic Models*. 2021. arXiv: 2102.09672 [cs.LG].

[Niu+23] Axi Niu et al. *ACDMSR: Accelerated Conditional Diffusion Models for Single Image Super-Resolution*. 2023. arXiv: 2307.00781 [cs.CV].

[PAB] Claire Pang, Kyle Astroth, and AJ Bethel. URL: https://cpang4.github.io/gan/ (visited on 08/23/2023).

[Par+19] Taesung Park et al. *Semantic Image Synthesis with Spatially-Adaptive Normalization*. 2019. arXiv: 1903.07291 [cs.CV].

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Pat+16] Deepak Pathak et al. "Context Encoders: Feature Learning by Inpainting". In: 2016.

[Ram+22] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. arXiv: 2204.06125 [cs.CV].

[RF16] Joseph Redmon and Ali Farhadi. *YOLO9000: Better, Faster, Stronger*. 2016. arXiv: 1612.08242 [cs.CV].

[Roc] Joseph Rocca. *Understanding Variational Autoencoders (VAEs)*. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73. [Accessed 28-Jun-2023].

[Rom+22] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2022. arXiv: 2112.10752 [cs.CV].

[RFB15a] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[RFB15b] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[Rus+15] Olga Russakovsky et al. *ImageNet Large Scale Visual Recognition Challenge*. 2015. arXiv: 1409.0575 [cs.CV].

[Ser22] Nikolas Adaloglou Sergios Karagiannakos. *How diffusion models work: The math from scratch*. Sept. 2022. URL: https://theaisummer.com/diffusion-models/ (visited on 08/21/2023).

[SME22] Jiaming Song, Chenlin Meng, and Stefano Ermon. *Denoising Diffusion Implicit Models*. 2022. arXiv: 2010.02502 [cs.LG].

[Spe20] Joshua S. Speagle. *A Conceptual Introduction to Markov Chain Monte Carlo Methods*. 2020. arXiv: 1909.12313 [stat.OT].

[Tak19] Masaki Takeda. "Brain mechanisms of visual long-term memory retrieval in primates". In: *Neuroscience Research* 142 (2019), pp. 7–15. ISSN: 0168-0102. DOI: https://doi.org/10.1016/j.neures.2018.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0168010218300142.

[Wan+22] Xin Wang et al. *Disentangled Representation Learning*. 2022. arXiv: 2211.11695 [cs.LG].

[Wan+04] Z. Wang et al. "Image quality assessment: From error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/tip.2003.819861.

[WSB03] Zhou Wang, Eero P. Simoncelli, and Alan C. Bovik. *Multi-scale structural similarity for image quality assessment , Eero P ...* 2003. URL: https://www.cns.nyu.edu/pub/eero/wang03b.pdf.

[Wen21] Lilian Weng. *What are diffusion models?* July 2021. URL: https://lilianweng.github.io/posts/2021-07-11-diffusion-models/ (visited on 08/21/2023).

[Woh] Wohlert. *Wohlert/generative-query-network-pytorch: Generative Query Network (GQN) in Pytorch as described in "neural scene representation and rendering"*. URL: https://github.com/wohlert/generative-query-network-pytorch/tree/master (visited on 08/21/2023).

[Zha+] Aston Zhang et al. *Dive into Deep Learning — d2l.ai*. https://d2l.ai/index.html. Accessed 29-Jun-2023, chapter 10.1. Long Short-Term Memory (LSTM).

[Zhe+23] Ce Zheng et al. *Deep Learning-Based Human Pose Estimation: A Survey*. 2023. arXiv: 2012.13392 [cs.CV].

[Zhu+20] Jun-Yan Zhu et al. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. 2020. arXiv: 1703.10593 [cs.CV].

Appendix A

Performance metrics

To measure the performance of the generator network consistently and robustly, we need an objective Image Quality Assessment method (IQA). The difference between subjective and objective IQA is that the former is done by humans and the latter with mathematical methods. Subjective IQA is impractical and prone to error whereas objective IQA is consistent and its robustness depends on the technique being used.

Numerous methods have been proposed, each with their own pros and cons: some are more sensitive to noise (PSNR [Far+16]) while others are more suitable for measuring structural similarity (SSIM) [HZ10]. The property of the IQA method we're interested in is that it must do well in measuring the structural similarity independently from luminance and contrast similarity. To this end, we have chosen Structural Similarity (SSIM) as the main metric and PSNR as a secondary extra metric to compute the similarity between the ground truth and generated images.

In the rest of this section, we will give an overview of the other objective IQA methods that have been considered.

A.1 Root Mean Squared Error

The most common metric for IQA is Mean Squared Error (MSE) or Root Mean Squared Error (RMSE). There is not a real difference between MSE and RMSE other than that RMSE has a smaller value range which makes it more suitable to interpret the similarity score. Consider $\mathbf{x}$ to be the ground-truth image, $\mathbf{y}$ the reconstructed image and N the number of pixels, RMSE can then be defined as follows:

$$\begin{aligned} \text{MSE}(\mathbf{x}, \mathbf{y}) &= \frac{1}{N} \sum_{i=0}^{N} (\mathbf{x}_i - \mathbf{y}_i)^2 \\ \text{RMSE}(\mathbf{x}, \mathbf{y}) &= \sqrt{\text{MSE}(\mathbf{x}, \mathbf{y})} \end{aligned} \tag{A.1}$$

This function has the range of $[0, \alpha]$, with alpha being the largest possible pixel value. For example, $\alpha = 255$ in case the pixels are represented by 8-bit values.

Although RMSE is very easy and convenient to compute, it is not a suitable metric to assess the quality of the generated images. This is because we are more interested in the structural accuracy of the generated image than its chromatic accuracy. RMSE is not suitable for measuring structure because it is sensitive to noise.

A.2 Peak Signal To Noise Ratio

The Peak Signal to Noise Ratio of two images $\mathbf{x}$ and $\mathbf{y}$ is the logarithm of the ratio between the largest possible pixel value (K) and the MSE of $\mathbf{x}$ and $\mathbf{y}$:

$$\text{PSNR}(\mathbf{x}, \mathbf{y}) = 10 \log_{10} \left(\frac{K_x^2}{\text{MSE}(\mathbf{x}, \mathbf{y})} \right) \tag{A.2}$$

If the pixel values are represented by an unsigned 8-bit integer then $K_x = 255$ because the pixel values would be between 0 and 255. $K_x = 1$ if the pixel values are represented by a floating point number and have a range of $[0, 1]$. Since this is a logarithmic function, the result is interpreted on the logarithmic scale, the decibel scale in this case. The PSNR function gives a small value if the two images are dissimilar and a large value converging to ∞ if they are similar. If the two images are identical, then the MSE becomes 0 causing the PSNR to converge to ∞.

The range of the PSNR function $[0, \infty)$ seems inconvenient at first, but it's very difficult to get a PSNR value larger than 100. The PSNR of an image with itself yields ∞ as expected. However, reducing the pixel values of the original image with just 0.01% yields a PSNR of 96.53 (see figure A.1). Therefore, if the PSNR of two images is 90 or more, they can be considered to be virtually identical. Due to the fact that PSNR is directly influenced by MSE, it is not suitable to be used as the main quality metric but will be used nonetheless in some cases as an extra metric for general image reconstruction quality.

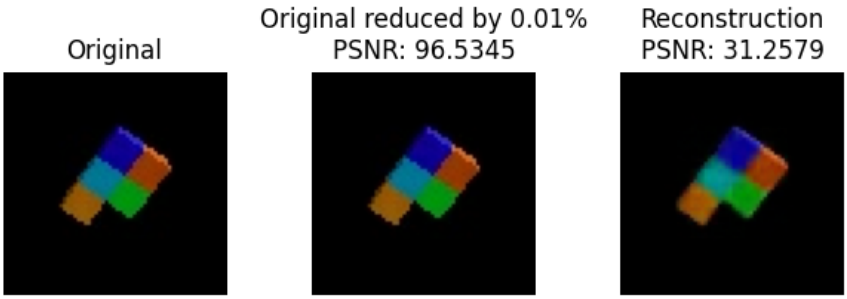


Figure A.1: Left: the original image, Middle: original image with pixel values reduced by 0.01%, Right: reconstruction of the original image with the GQN model.

A.3 Structural Similarity

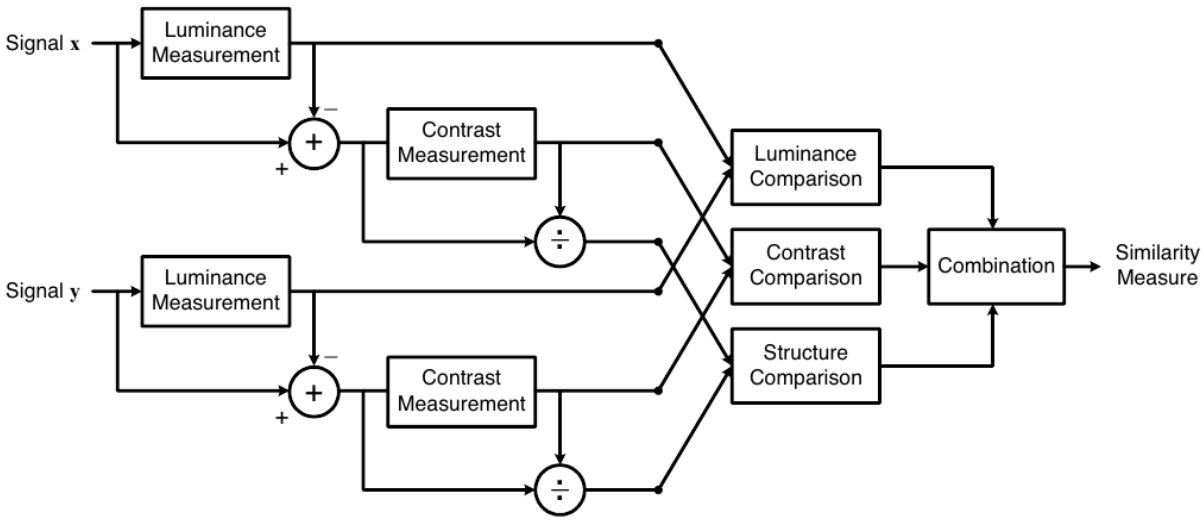


Figure A.2: Diagram of the structural similarity (SSIM) measurement method. Source: [Wan+04]

SSIM [Wan+04] is the aggregation of 3 different similarity scores of the compared images, these scores correspond with the luminance, contrast and structure similarity. Each of these scores can be weighed during aggregation to give more bias to some components over the others. The diagram in figure A.2 shows an overview of how the Structural Similarity (SSIM) of two images $\mathbf{x}$ and $\mathbf{y}$ is computed. First, the luminance similarity is measured from the raw input images. Second, the contrast similarity is calculated from the mean-shifted images, whose luminance has been subtracted from them. Finally, the structure similarity is calculated from the *standardized* image data, obtained by mean-shifting the images and dividing them by their standard deviations.

The SSIM function is defined as follows:

$$S(\mathbf{x}, \mathbf{y}) = [l(\mathbf{x}, \mathbf{y})]^{\alpha}.[c(\mathbf{x}, \mathbf{y})]^{\beta}.[s(\mathbf{x}, \mathbf{y})]^{\gamma} \tag{A.3}$$

The functions $l(\mathbf{x}, \mathbf{y})$, $c(\mathbf{x}, \mathbf{y})$ and $s(\mathbf{x}, \mathbf{y})$ are the luminance, contrast and structure similarity functions respectively. Their powers α, β and γ are positive numbers serving to control the bias of the similarity functions.

Each similarity function is designed to meet the following conditions:

- Symmetry: $S(\mathbf{x}, \mathbf{y}) = S(\mathbf{y}, \mathbf{x})$
- Boundedness: $S(\mathbf{x}, \mathbf{y}) \leq 1$
- Unique maximum: $S(\mathbf{x}, \mathbf{y}) = 1 \iff \mathbf{x} = \mathbf{y}$

Before defining the similarity function we first define the means (μ_x), standard deviations (σ_x) and the correlation (σ_{xy}) of the images:

$$\mu_x = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i$$

$$\sigma_x = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (\mathbf{x}_i - \mu_x)^2}$$

$$\sigma_{xy} = \frac{1}{N-1} \sum_{i=1}^{N} (\mathbf{x}_i - \mu_x)(\mathbf{y}_i - \mu_y)$$

Now, we can define the similarity functions:

$$l(\mathbf{x}, \mathbf{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{A.4}$$

$$c(\mathbf{x}, \mathbf{y}) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{A.5}$$

$$s(\mathbf{x}, \mathbf{y}) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{A.6}$$

The constants C_1, C_2 and C_3 are needed to avoid instability in case the denominators approach zero. The following value for each constant is recommended by the SSIM paper [Wan+04]:

$$C_i = (K_i L)^2 \tag{A.7}$$

With L being the dynamic range of the images and each K_i a small constant such that $K_i < 1$.

SSIM is a suitable IQA method for the generator of the GQN because its structure similarity score is relatively unaffected by changes in luminance or contrast. There is also an improved version of this, e.g. the Multi-Scale SSIM which will be described next.

A.4 Multi Scale Structural Similarity

An extension of the standard SSIM method is the multi-scale SSIM (MS-SSIM [WSB03]) which is illustrated in figure A.3. This method combines the structural and contrast similarities of the input images $\mathbf{x}$ and $\mathbf{y}$ at multiple levels of detail. The MS-SSIM is calculated in M steps. At each step i, the contrast $c_i(\mathbf{x_i}$ and $\mathbf{y_i})$ and structural $s_i(\mathbf{x_i}$ and $\mathbf{y_i})$ similarities are calculated with the inputs received from the previous step. Afterward, a low-pass filter is applied to both images, after which they are down-scaled to half their size (this is illustrated by L and $2 \downarrow$ in figure A.3) and passed onto the next step. Finally, at step M, the luminance similarity $l_M(\mathbf{x_M}, \mathbf{y_M})$ is also calculated along with the contrast and structural similarities, and all results are aggregated as follows:

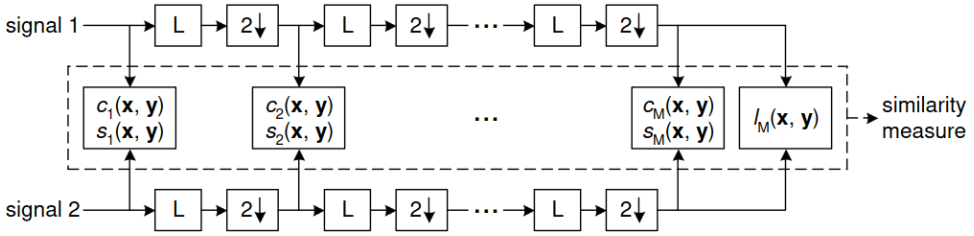


Figure A.3: Diagram of the Multi-Scale Structural Similarity measurement method. L is a low pass filter and 2 with the arrow pointing downwards means down-sampling the input image to half its resolution. Source: [WSB03]

$$\text{MS-SSIM}(\mathbf{x}, \mathbf{y}) = [l_M(\mathbf{x}_M, \mathbf{y}_M)]^{\alpha_M} \cdot \prod_{j=1}^{M} [c_j(\mathbf{x}_j, \mathbf{y}_j)]^{\beta_j} [s_j(\mathbf{x}_j, \mathbf{y}_j)]^{\gamma_j} \tag{A.8}$$

$$\mathbf{x}_i = L(D(\mathbf{x}_{i-1})), i \in \{2, ..., M\} \tag{A.9}$$

With L being the low-pass filter and D the function for down-scaling the image by a factor of two.

As mentioned in the previous section (A.3), SSIM is a fitting IQA method for the GQN's generator, even more so its improvement MS-SSIM which aligns even better with the subjective quality measurements conducted by humans [WSB03]. On top of this, it has the same qualities and properties as SSIM, e.g. its ability to measure the structural similarity independently from luminance and contrast similarity. One drawback of the multi-scale version is that it is not practical on images with a low resolution, like 64 by 64 pixels. SSIM will be used when doing quality assessment for images with a low resolution and MS-SSIM for images with a sufficiently high resolution.