Learning To Model A Driving Simulator

3D Hierarchical VQVAE for Modelling Driving Environments

Mattias Xu

Thesis submitted for the degree of
Master in Informatics: Robotics and Intelligent Systems
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

Learning To Model A Driving Simulator

3D Hierarchical VQVAE for Modelling Driving Environments

Mattias Xu

Abstract

When developing self-driving cars, using real roads is costly and may even be dangerous. Therefore, driving simulators are frequently used. These are mostly hand-coded by domain experts to model the driving environment. However, *learned* models within model-based reinforcement learning algorithms have recently shown great results. A learned model of driving scenarios may also be beneficial, and this thesis will attempt to model the driving environment explicitly .

More specifically, we explore using a combination of an autoencoder and an autoregressive model to model a driving simulator. The data consists of driving scenarios of 16 frames collected from a driving simulator.

The implemented autoencoder has good performance and can compress 16 frames of video to a latent space. The latent space reduces bits required to store videos by 98.8% without losing significant information. Using this latent space, we are able to predict one frame when conditioning on eight frames reliably. However, subsequent frames are usually not predicted well.

Contents

List of Figures

List of Tables

Acknowledgements

During my summer internship at Simula Research Laboraty the foundation of this project was set and the research goals were set together with my supervisors. I would like to thank Simula Research Laboraty for having me and letting me use their facilities.

To my supervisors, Shaukat Ali, Ferhat Ozgur Catak and Jim Tørresen, thank you for your support and guidance. I would also like to thank all friends and family, who has assisted me in many ways.

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Autonomous driving is one of the hottest topics within technology development. Today, the reality of self-driving cars seems to be quickly approaching, but many challenges remain. Technical challenges include mapping, perception, planning etc., but there are also regulatory and ethical challenges.

Aside from the physical controls, driving is similar to a vision based video game, where you can see what is happening and have to take the right actions. Reinforcement learning has had much success in this area, achieving superhuman abilities, mainly in the classic ATARI games [46] [47] [3], but also in more advanced games like StarCraft II [65].

A significant reason for the success is unrestricted access to the environment. These various algorithms learn by repeatedly playing the games, with no restrictions or consequences. Model-free algorithms like [47] [58] have dominated many benchmarks, but recent research on model-based algorithms [30] [21] shows competitive results. Model-based algorithms utilize a model of the environment to learn optimal behaviour. These algorithms learn to model the environment, often in a more compressed space. This approach requires far less interaction with the actual environment, as exploration can be done in the model.

It is expensive and dangerous to both train and test on real roads, not to mention slow and ineffective. Thus, less interaction with real roads is preferred and a model-based approach is advantagoues for self-driving cars. Simulators like [17] [56], which are hand-coded models by domain experts to model the environment, are frequently used and typically based on game engines like Unity or Unreal Engine. These engines are dependent on advanced computer graphics techniques and physics simulation, requiring much computation. A learned model in a compressed space may therefore be advantageous.

Inspired by the model-based reinforcement learning algorithms, this thesis will explore the approach of learning a model that can look into future states of the environment regarding self-driving cars. Ideally, a learned model will be a full-fledged replacement for a hand-coded simulator, using only a fraction of the computational power.

1.2 Problem Statement

The ultimate goal is to create a deep generative model to generate several frames into the future, given a driving video. In essence, the deep generative model has to learn to model the driving environment to predict the future states of the environment.

The methods used in the project are based on state-of-the-art research in deep learning and deep generative modelling. More specifically, the VQVAE-2 framework [54] is used. The framework splits the research goal into two objectives:

1. Design a model to compress driving videos to a lower dimensional latent space.
2. Design an autoregressive model to generate video frames in the latent space from the previous step.

With the goal in mind, we will aim to answer the following research questions:

RQ1: Can a compressed quantized latent space of driving scenarios be learned and used for modelling?

RQ2: If so, to what extent can we model a driving simulator using that latent space?

To scope the project, we will collect data through a simulator where the driving environment will be of limited complexity. The work done in this thesis is not an attempt to learn a fully-fledged model. However, it will focus on training a model that generates a few frames, given previous frames, without considering controls like steering, braking and acceleration. This is a natural first step. If successful, further research can be done to work towards a trained model that can generate new frames conditioned on controls like steering, braking and acceleration.

1.2.1 Note on Machine Learning Literature

Interest in machine learning has grown significantly over the last decade, which may be the cause of some ”troubling trends” within machine learning research [39].

Often, being first is more important than being thorough. For the sake of speed, many works of significance are not published through traditional peer-reviewed journals, but as conference papers or even just an online open-access archive like arXiv.org. An example where speed mattered was at the end of 2014 when around five research groups almost simultaneously submitted papers on arXiv about image captioning using convolutional neural networks combined with recurrent neural networks [43][15][66][32][36].

Machine learning research often utilizes comparisons using benchmark datasets, like ImageNet [12]. This may have caused research to be more focused on empirical evidence rather than solid theoretical justifications. Empirical gains are made, but the source of these gains may sometimes be unknown, making it hard to explain why something should be used. For instance, the Adam optimizer [35] showed solid empirical results, and the authors also offered a theorem regarding convergence in convex cases. However, the theorem is perhaps irrelevant as the paper focuses on non-convex optimization. Also, the theorem was later proved to be incorrect [55].

Due to this, it is often hard to explain every choice made in this project, but theoretical justification will be given where possible

Many terms found in machine learning literature can also be misleading and imprecise with regard to the original definition. For example, *convolutions* in convolutional neural networks are used to mean a *correlation*, while *deconvolution* is used to mean a *transposed convolution*. Some terms are given new names without any new meaning. For example, *Step size* is also called *learning rate*.

As this work is based mainly on machine learning work, the terms used in machine learning research will be used. When terms are interchangeable, the most suitable is used based on context, i.e. *step size* will be used in the introduction when optimization is presented, but learning rate will be used later as the context changes.

Chapter 2

Deep Learning Background

This chapter provides a brief background, from a computer vision standpoint, on machine learning and deep neural networks[1].

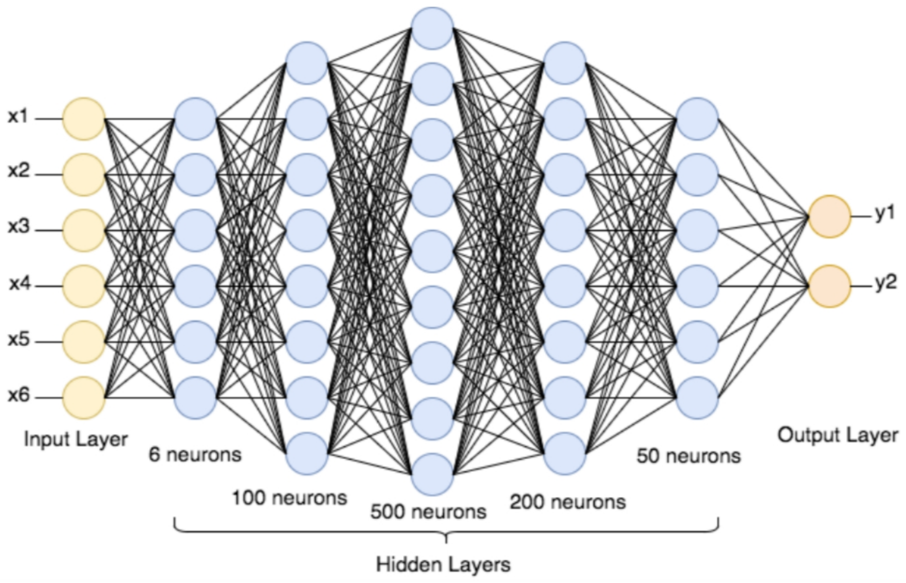

Figure 2.1: A deep neural network. Today, deep neural networks with over a trillion parameters exist [18].

2.1 Introduction

Many kinds of problems can be solved by a function performing a mapping, $f : X \rightarrow Y$, where X is an input space, and Y is an output space.

For autonomous cars, it is necessary to detect objects on the road. Here, X can be the space of images, and Y could be the probability of

[1]For a more in-depth introduction, `https://www.deeplearningbook.org/` is recommended.

an object being present. This is a trivial task for a human but specifying a function to map between millions of pixel intensities to probabilities is challenging by traditional means, and alternative approaches are sought-after. Instead of manually designing a function, a computer can *learn* the mapping $\boldsymbol{f} : X \rightarrow Y$.

In 1997, Mithcell [45] provided the following definition of machine learning:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

The task T is what we want to learn. Machine learning can be used for tasks like object classification, image generation and language translation. These are just a few examples, and machine learning has proved to perform well at an increasing number of tasks.

In deep learning, the concerning subset of machine learning, the experience E, usually comes from a training dataset of samples $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_n, \boldsymbol{y}_n)\}$, where $(\boldsymbol{x}, \boldsymbol{y})$-pairs are examples of the desired mapping.

The performance measure P is a quantitative measure that has to be designed. This is dependent on the task. For example, classification can use accuracy as a performance measure. As we would like to evaluate how well the machine learning algorithm performs on data outside of the training set, a separate test set of data is used for evaluation.

Closely related to the performance measure is the loss function. A loss function $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ gives a scalar value describing the difference between a single function output $\hat{\boldsymbol{y}}$ and the desired output $\boldsymbol{y}$. Loss functions are designed such that the lower the loss is, the better the task T is performed.

For example, a measure like accuracy is not desirable to use as a loss function. On a single sample, the accuracy is either 0 or 1. Accuracy does not differentiate between a 51% and a 100% confident prediction. However, a loss like the cross-entropy loss gives a high loss to the 51% prediction if it is correct, and a higher loss to the 100% prediction if it is wrong.

Now, we can formulate our goal of learning $\boldsymbol{f}$ as finding the function with the lowest expected loss over samples drawn from the actual distribution D of $(\boldsymbol{x}, \boldsymbol{y})$-pairs.

$$\boldsymbol{f}^* = \arg\min_{\boldsymbol{f}} \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim D}[L(\boldsymbol{f}(\boldsymbol{x}), \boldsymbol{y})] \tag{2.1}$$

Unfortunately, the true distribution D is not accessible, and it is impossible to solve the optimization problem above. Instead, the expected loss can be approximated by taking the average loss over the training dataset, which leaves the following optimization problem:

$$f^* = \arg\min_f \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i) \tag{2.2}$$

2.2 Regularization

Optimizing 2.2 as an approximation for 2.1 poses some challenges. For simplicity, consider a function that perfectly maps every x_i to y_i in the training set. This leads to a minimal loss, but if the function returns a constant k for every x not in the training set, a very high loss is expected for samples outside the training set.

There is a danger of detecting patterns in the noise in the training set, leading to overfitting. When overfit, the learned function f performs well on the training set but fails to generalize to all $(x, y) \sim D$.

Also, two different functions can achieve the same loss, but one may generalize better than the other. With this in mind, it is crucial to not evaluate on training data, but on a test set the model does not directly optimize for.

Strategies designed to combat this problem are known in machine learning as regularization. Goodfellow et al. [19] defined regularization as follows:

> Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

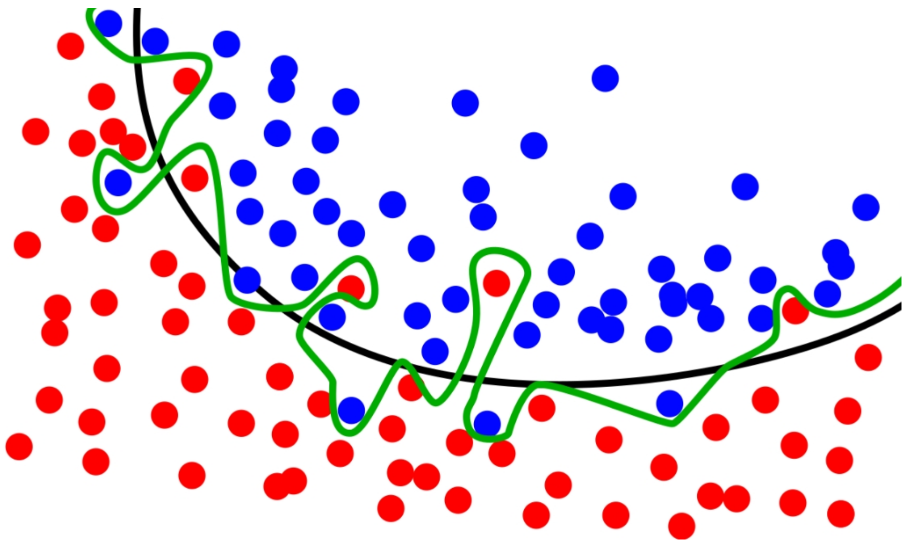

Figure 2.2: The green decision boundary is overfitted, while the black decision boundary is more sensible.

Source: link

The rest of the subsection will go through some classes and examples of regularization strategies.

2.2.1 L_2 Regularization and Weight Decay

Regularization has been used a long time prior to deep learning. Consider a linear regression problem where the input is 2-dimensional, and the target is a scalar. The linear function can look like $f(x_1, x_2) = x_1 w_1 + x_2 w_2 + b$. A common loss function for regression is the mean squared error (MSE) $L(y, \hat{y}) = (y - \hat{y})^2$. To avoid the parameters w_1 and w_2 from being too large and overly emphasizing one input feature, a regularization strategy of adding a term penalizing large weights is used. What we now want to minimize is

$$\frac{1}{n}\sum_{i=1}^{n} \underbrace{(y_i - (w_1 x_{i,1} + w_2 x_{i,2} + b)}_{\text{loss}} + \underbrace{\lambda(w_1^2 + w_2^2))}_{\text{regularization}} \tag{2.3}$$

where λ is a parameter that tunes the regularization strength. Here, the L_2 regularization is used.

This regularization technique can also be used on deep neural networks, with the only difference being the much more significant number of weights. Despite its long history predating neural networks L_2 regularization is commonly used to prevent overfitting.

More generally, if a regularization strategy uses some sort of penalty dependent on the function, the optimization problem can be written as

$$\boldsymbol{f}^* = \arg\min_{\boldsymbol{f}} \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i), \boldsymbol{y}_i) + R(\boldsymbol{f}) \tag{2.4}$$

This approach avoids large weights by penalizing the L^2 norm of the weights by adding it to the loss function, but there are other ways to avoid it.

Weight decay is another method to avoid large weights. Here, the weights are decayed directly during the weight update during optimization (see next subsection). In [22] they described the weight update as follows:

$$\boldsymbol{w}_{t+1} = \beta \boldsymbol{w}_n - \alpha(\frac{\partial L}{\partial \boldsymbol{w}})_n \tag{2.5}$$

where $\beta < 1$ and α are parameters. The term $\beta \boldsymbol{w}_n$ acts similar to the L^2 regularization, as bigger weights will be reduced more than smaller weights.[2]

[2] L^2 regularization is commonly referred to as weight decay, but this is an inaccuracy. This will be explained in chapter 6.

2.2.2 Dataset Augmentation and Noise Injection

A simple way to make a machine learning model generalize better is to add more data to the training set. Having more unique data samples gives the model more information about the underlying distribution. However, it may not be easy to collect more data in practise.

Instead of collecting more data, it is possible to generate fake data in many cases. With images already in the training set, we can perform transformations to create new images. Rotating, mirroring, scaling and translating are examples of transformations that can be used. For text-based tasks, swapping out words in sentences with synonyms may be beneficial. It is essential that the transformations do not change the correct output. For example, rotating an image 180° for a digit recognizer would change the correct output of ”6” and ”9”.

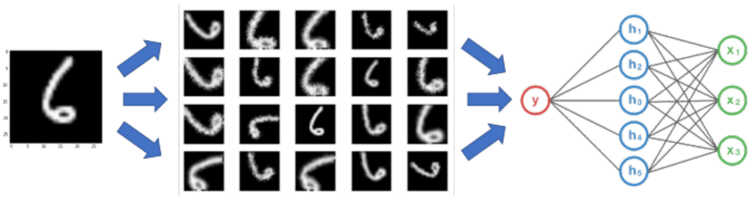

Figure 2.3: Transformations that can be used for a digit recognizer. Notice that the images are rotated a maximum of 90 degrees.
Source: link

Another form of data augmentation is injecting noise during training time. This effectively makes the size of the training dataset larger, as noise is randomly applied, making the same input different every time it is presented. Another intuitive way to explain the effectiveness is to look at applying noise as a measure to prevent the neural network from being too reliant on specific features, as they may be noisy. This is similar to weight decay.

Sietsma and Dow [60] demonstrated that injecting random noise to the input improved neural networks’ ability to generalize well and many regularization techniques is based on random noise. Simply applying Gaussian noise can be effective, but more modern forms of noise injection for computer vision tasks include randomly masking out squares of the image [14] and even randomly combining images and their labels [70] [71].

Noise is not limited to the input. It can also be effectively applied to the hidden layers, weights [29] and gradients [48]. One of the most commonly used regularization techniques is dropout [61], which stochastically drops neurons during training. Every neuron has a probability p of getting dropped.

Intuitively, you can think of this as preventing specific features from being dominant, as they can be dropped, but training with dropout can also be seen as training an ensemble of neural networks. For every training

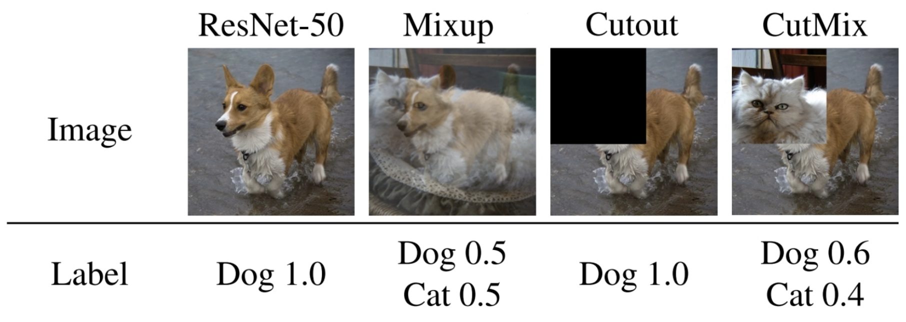

Figure 2.4: Modern forms of data augmentation.
Source: [70]

sample, parts of the neural network are dropped, and a subnetwork is trained. During test time, you approximate all subnetworks' average output by scaling every weight by the dropout probability p.

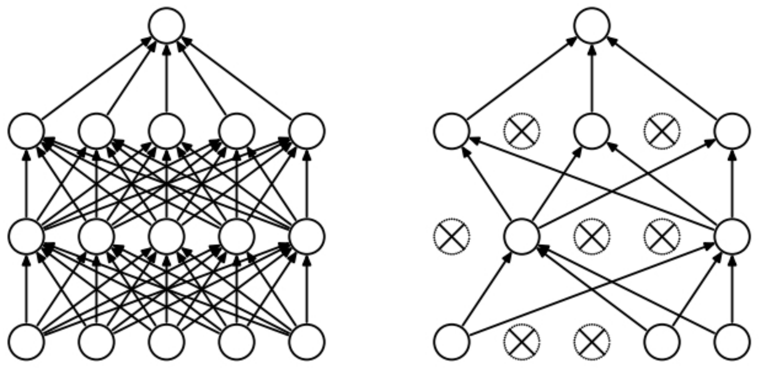

Figure 2.5: **Left:** Neural network with 2 hidden layers. **Right:** Subnetwork after applying dropout.
Source: [61]

2.3 Optimization

Looking at equation (2.4), we know that we are looking for the function that minimizes the loss over a training set, often with a regularization term. During optimization, the neural network architecture is set, and the only way the function is changed is by tweaking its parameters/weights, which will be denoted as $\boldsymbol{\theta}$. Now, the optimization problem is

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} L(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i) + R(\boldsymbol{\theta}) \tag{2.6}$$

Remember that performing the optimization in equation (2.6) is not the goal. The true goal is to find $\boldsymbol{\theta}$ that minimizes the expected loss of the true data distribution. Equation (2.6) is optimized for in the hope that it will help towards the true goal.

As we can see in equation (2.6) it is possible to evaluate the total loss (including the regularization term) over the training set for any $\boldsymbol{\theta}$. From here on out, the loss L denotes the loss over the training set, including the regularization term if used. As L can be evaluated, optimization techniques like random search or hill climbing can be used. However, the high number of parameters makes it very computationally expensive to guess and check.

2.3.1 The Learning Algorithm

To make optimization more effective, the loss is constrained to be differentiable, making gradient-based optimization techniques possible. The gradient of the loss over the training set $\nabla_{\boldsymbol{\theta}} L$ can be computed and provides the slope along every dimension of $\boldsymbol{\theta}$. This will give us the direction of the steepest ascent of L. By taking the negative of the gradient, the direction of the steepest descent is found.

Therefore, by adding a small amount of the negative gradient to $\boldsymbol{\theta}$, L will decrease, and a better $\boldsymbol{\theta}$ is found. The batch gradient descent algorithm works by alternating between computing $\nabla_{\boldsymbol{\theta}} L$ and applying a small step in the negative direction of $\nabla_{\boldsymbol{\theta}} L$.

Instead of computing $\nabla_{\boldsymbol{\theta}} L$, it can be estimated by calculating the gradient of the loss over a mini-batch of m samples drawn from the training set. This is called stochastic gradient descent (SGD).

Algorithm 1 Stochastic Gradient Descent

```
Require: Initial parameters $\boldsymbol{\theta}$
Require: Step size $\alpha$
  while not converged or stopped do                                ▷ Training loop
      Sample a mini-batch $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$
      Estimate $\nabla_{\boldsymbol{\theta}} L \approx \nabla_{\boldsymbol{\theta}}[\frac{1}{m}\sum_{i=1}^{m} L(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)), \boldsymbol{y}_i]$
      Update parameters $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L$
  end while
```

The mini-batch size m is typically set to a much lower number than the number of samples in the training set. As this introduces noise, it requires more iterations and a smaller step size than batch gradient descent, but the trade-off is worthwhile.

The computational burden and memory usage is $O(m)$. The standard error of the mean from n samples is $\sigma/\sqrt{n}$, so the accuracy of the estimated gradient is $O(\sqrt{m})$ [19], which means increasing m has a diminishing return on the accuracy of the estimation.

Remember that the whole training set estimates the true data distribution, so the batch gradient is not the "true" gradient. In practice, smaller mini-batches actually increase the model's performance, measured by its ability to generalize. Larger mini-batch sizes lead to convergence in sharper local minima, which is known to generalize poorly. It is believed that this is due to the higher noise in small mini-batch sizes [33].

Step 1 and 2 in algorithm 1 is used in almost all optimization algorithms in deep learning, but there are many variations of step 3, where the update rule is modified. These other methods incorporate momentum or adaptively change the effective step size, which can speed up convergence and make setting ϵ easier. Examples are SGD with momentum, Root Mean Squared Propagation (RSMProp) [25] and Adaptive Momentum (Adam) [35].

Algorithm 2 Adam

```
Require: Initial parameters $\boldsymbol{\theta}$
Require: Step size $\alpha$
Require: Small number to avoid division by 0 $\epsilon$
Require: Exponential decay rates for moment estimates $\beta_1, \beta_2 \in [0, 1)$
  Initialize moment vectors and time step:
    $m \leftarrow 0$
    $v \leftarrow 0$
    $t \leftarrow 0$
  while not converged or stopped do                                ▷ Training loop
    Increment time step $t \leftarrow t + 1$
    Sample a mini-batch $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$
    Estimate $\nabla_{\boldsymbol{\theta}} L \approx \nabla_{\boldsymbol{\theta}}[\frac{1}{m}\sum_{i=1}^{m} L(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i + R(\boldsymbol{\theta})]$
    Update moment vectors:
      $m \leftarrow \beta_1 \cdot m + (1 - \beta_1)\nabla_{\boldsymbol{\theta}} L$
      $m_t \leftarrow m/(1 - \beta_1^t)$
      $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot (\nabla_{\boldsymbol{\theta}} L)^2$
      $v_t \leftarrow v/(1 - \beta_2^t)$
    Update parameters $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \cdot m_t/(\sqrt{v_t} + \epsilon)$
  end while
```

Adam can be seen as a combination of momentum and RMSprop, where the m moment vector comes from momentum and the v moment vector comes from RMSprop. As the gradient can be noisy, m holds a decaying sum of current and previous gradients and stabilizes the steps taken. v holds a decaying sum of current and previous gradients squared and decreases the effective step size in "steep" directions while increasing step size in "shallow" directions.

Notice that the moment vectors are initialized and biased to 0. This is compensated for by the use of m_t and v_t.

2.3.2 Backpropagation

Both steps 1 and 3 in algorithm 1 are trivial, so what is left in the training loop is calculating the gradient over the mini-batch. This is done by applying the chain rule repeatedly.

Consider a 3-layer neural network, where $\boldsymbol{\theta}_i$ are the parameters of the i-th layer, and $\boldsymbol{x}_i$ is the output of the i-th layer. The gradient of the loss with respect to the parameters can be calculated as follows

$$\frac{dL}{d\boldsymbol{\theta}_3} = \frac{dL}{d\boldsymbol{x}_3}\frac{d\boldsymbol{x}_3}{\boldsymbol{\theta}_3} \qquad \frac{dL}{d\boldsymbol{\theta}_2} = \frac{dL}{d\boldsymbol{x}_3}\frac{d\boldsymbol{x}_3}{d\boldsymbol{x}_2}\frac{d\boldsymbol{x}_2}{d\boldsymbol{\theta}_2} \qquad \frac{dL}{d\boldsymbol{\theta}_1} = \frac{dL}{d\boldsymbol{x}_3}\frac{d\boldsymbol{x}_3}{d\boldsymbol{x}_2}\frac{d\boldsymbol{x}_2}{d\boldsymbol{x}_1}\frac{d\boldsymbol{x}_1}{d\boldsymbol{\theta}_1} \tag{2.7}$$

Notice that the multiplication happens by going backwards through the layers from the loss. The loss is first calculated by doing a forwards pass through the network and then the gradients are calculated during a backwards pass. Hence, the name backpropagation. This requires keeping the intermediate outputs in memory, but this memory cost is traded with efficiency.

Essentially, the derivation consists of multiplying Jacobian matrices. Due to the input dimension often being higher than the output dimension in deep learning, doing the differentiation in a backward pass has been the standard.

2.3.3 Hyperparameter Optimization

The hyperparameters $\boldsymbol{\lambda}$ are the parameters that are set before the training starts. This includes what optimizer to use, the step size, also known as learning rate, how deep the neural network is etc. The model parameters are optimized by training to increase the performance, but the hyperparameters can also have a significant effect. For example setting the training step too low might lead to slow convergence, while setting it too high might lead to divergence.

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \left[\arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} L(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i) + R(\boldsymbol{\theta}) \right] \tag{2.8}$$

Looking at equation (2.8), optimizing hyperparameters is much more tricky than optimizing the parameters, as it is not possible to evaluate the loss without finishing training. Still, it has to be done and is of importance.

Bergstra and Bengio [5] compared search algorithms for deciding the set of hyperparameters to use. They concluded that random search is a simple, practical and efficient way to find hyperparameters, especially when $\boldsymbol{\lambda}$ is of higher dimension.

When evaluating choices of hyperparameters, the test set cannot be used. Suppose the test set is used to evaluate hyperparameters repeatedly.

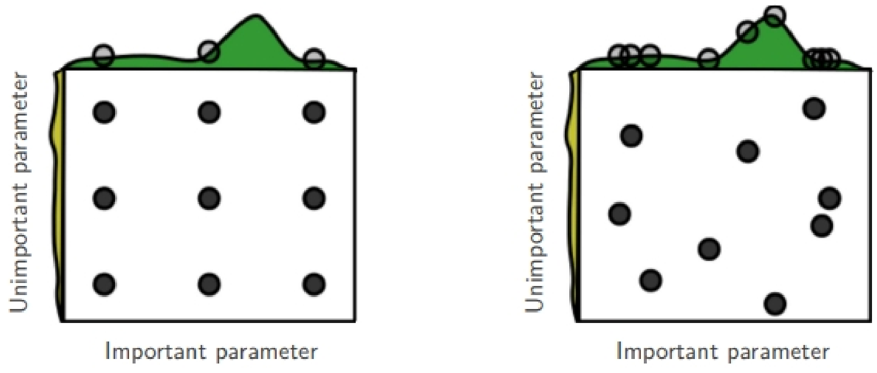

Figure 2.6: Grid search vs random search. Grid search only produces 3 different values, as the y-axis parameter is not important. In higher dimensions, this failure mode is common.
Source: [5]

In that case, the hyperparameters can overfit to the test set, and the test set no longer provides an unbiased evaluation of the final model. To evaluate the hyperparameters, an additional set of data is required, called the validation set.

Training is done on the training set, and the hyperparameters are evaluated on the validation set. Finally, the training set is only used for evaluating a final model.

An alternative way to do this is k-fold cross-validation. A separate test is put aside, but the rest of the data is split into k folds. Then, one of the k folds is selected as the validation set, and the $k - 1$ folds left over are used to train the model. Repeat this until every fold has served as the validation set and use the average performance to evaluate the hyperparameters selected. This is done to reduce bias and is especially useful when data is limited.

2.4 Deep Neural Networks

2.4.1 Feedforward Neural Networks

So far, the equations have included an arbitrary function f with parameters $\boldsymbol{\theta}$ and only a vague idea of what deep neural networks are has been given. Deep neural networks (DNN) are very powerful models and can be tuned to approximate any function to any desired accuracy [27]. They are *deep* because they consist of many layers. Despite this, neural networks are, in essence, simple.

Feedforward neural networks, also known as multilayer perceptrons (MLP), consists of layers of matrix multiplication followed by an element-wise non-linearity denoted σ. These layers are called linear, fully connected or feedforward layers. A two-layer feedforward neural network with input $\boldsymbol{x}$ is a function $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W_2}\sigma(\boldsymbol{W_1}\boldsymbol{x})$, where $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$ are matrices consisting of the weights. Note that the last layer usually is not followed up by a non-linearity. A bias, $\boldsymbol{b}$, is often included explicitly. Several outputs can be computed at once by forming an input matrix, where each column is an input. If the layer's input consists of n elements, the output consists of m elements, and the mini-batch size is 2, the linear layer will look like this.

$$\sigma(\boldsymbol{WX}) = \sigma\left(\begin{bmatrix} w_{1,1} & w_{1,\dots} & w_{1,n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ w_{m,1} & w_{m,\dots} & w_{m,n} & b_m \end{bmatrix} \begin{bmatrix} x_{1,1} & x_{2,1} \\ \vdots & \vdots \\ x_{1,n} & x_{2,n} \\ 1 & 1 \end{bmatrix}\right) \tag{2.9}$$

Without the non-linearity, the network would simply be a linear transformation. Among common non-linearities, or activation functions (see Figure 2.7), are $\tanh x$, $\text{sigmoid}(x)$ and the rectified linear unit (ReLU).

$$\tanh(x), \qquad \text{sigmoid}(x) = \frac{1}{1 - e^{-x}}, \quad \text{ReLU}(x) = \max(0, x) \tag{2.10}$$

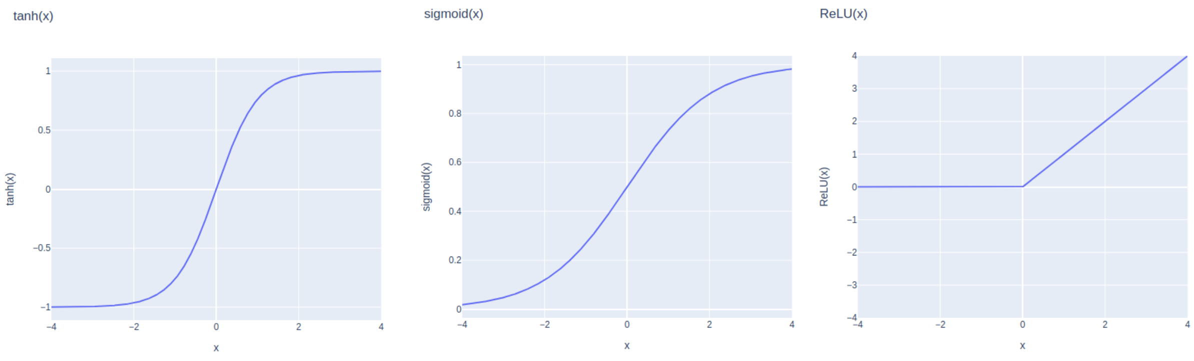

Figure 2.7: Different activation functions. From left to right: tanh(x), sigmoid(x), ReLU(x).

Looking at tanh and sigmoid, they both have a small slope at high absolute values. This causes the vanishing gradient problem, as small gradients cause small updates, which slows down training. In deep networks, this problem is compounded through backpropagation, as multiple small gradients multiplied causes even smaller gradients.

ReLU is commonly used, as it solves this problem, but has its own drawbacks, like "the dying ReLU problem" and exploding gradients. Many activation functions similar to ReLU exist, like leaky ReLU, Gaussian error linear unit [24] (GeLU), exponential linear unit [11] (ELU) etc.

Today, deep neural network architectures consist of linear layers like in feedforward neural networks, but also many different layers serving different purposes. The rest of the section will introduce residual connections, convolutional layers and self-attention, which are key building blocks in DNNs for computer vision tasks.

2.4.2 Residual Connections

Increasing the depth of a neural network can increase its performance, but it also increases the difficulty of training it. To alleviate this problem with deeper neural networks, He et al. presented the residual learning framework [23].

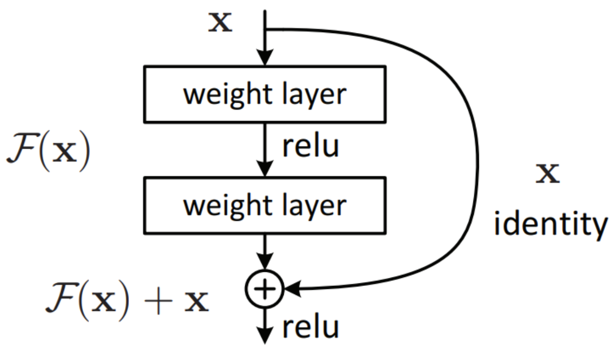
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2.8: Residual connection.
Source: [23]

A residual connection is simply a connection that adds the input of a layer to the output of a layer. This allows for significantly deeper networks and He et. al even showed no problems in training a network with over 1000 layers. Today, residual connections can be found in almost every modern neural network architecture.

2.4.3 Convolutional Neural Networks

Convolution neural networks (CNN) are neural networks designed to handle data with spatial relations, like images. The key part of CNNs is the convolutional layer (conv layer).

The conv layer works by convolution sliding a set of small learnable filters over an image to create a new hidden representation of the image, also called *activation map*. The idea is that a filter will extract useful features from a small patch of the image. These same features can also be found in other parts of the image. With only a few weights, useful local features can be extracted from the whole image.

Recall that images are of size $H \times W \times C$, where H is the height, W is the width, and C is the channel count. In an RGB image, $C = 3$, while in a grayscale image $C = 1$. The size of the activation map depends on the conv layer's 4 parameters. The number of filters, the size of the filters, the stride of the filters and the amount of zero-padding on the input.

A $n \times n \times c$ filter produces one channel of the output by starting at the upper-left most $n \times n$ patch of the input. Like in a linear layer with one output, every patch element is multiplied by a weight and summed. This produces a single value of one channel of the activation map. Next, the same filter with the same weights is moved to the right, producing a new value. As the filter was moved to the right, the new value is placed to the right of the previous one to maintain spatial correlation. When the filter cannot move further, it is moved down and starts on a new row. The same weights are used over and over over the whole input. In practice, this is done much more efficiently by calculating it all in parallel instead of sequentially.

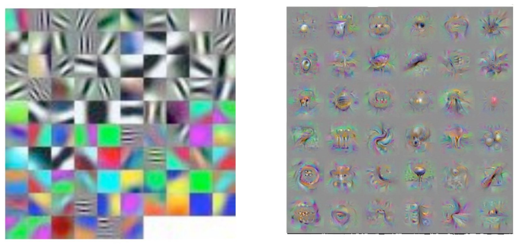

Figure 2.9: Visualized filters from the first and final layers of a trained CNN.

Source: [69]

After one layer, a single "pixel" in the activation map only detects very local spatial features, but after several layers, the *receptive field* increases and complex features are detected (see Figure 2.9).

A conv layer consisting of 64 filters with size $5 \times 5 \times 3$ has $64 \cdot (5 \cdot 5 \cdot 3 + 1) = 4864$ parameters including biases. With a stride of 1 and no zero-padding, the height and width of the output will be slightly reduced, while the channel count will be the amount of filters. Given a $256 \times 256 \times 3$ image the conv layer will transform it into an activation map of size $252 \times 252 \times 64$ using only 4864 parameters. For a linear layer to do the same, it would require $(252 \cdot 252 \cdot 64) \cdot (256 \cdot 256 \cdot 3 + 1) = 799\,069\,307\,904$ parameters.

The amount of parameters needed is drastically reduced by assuming that useful local spatial features exist in the data and is sufficient for performance.

2.4.4 Recurrent Neural Networks

A recurrent neural network (RNN) is an extension of the conventional neural network that can process sequential data using a recurrence formula $h_t = f(h_{t-1}, x_t)$, where h_t is the hidden state and x_t is the input at time t. The hidden state can be seen as a summary of all previous input and is updated using the next input by the neural network f. Then, the hidden state can be used to produce the output.

The vanilla RNN cell has the following recurrence formula:

$$h_t = \tanh(W\begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}) \tag{2.11}$$

While this is simple, vanilla RNNs have problems learning longer-term dependencies due to exploding and vanishing gradients during training [4].

Long Short-Term Memory (LSTM) [26] cells were designed to address the challenges of the vanilla RNN. The recurrence formula of the LSTM is more complex but also more powerful than the simple vanilla recurrence. In addition to the hidden state, LSTM cells also have a memory state and trainable gates that at each time step decides to what degree the cell should read from, write to and reset the cell.

A more recent RNN cell, the gated recurrent unit (GRU), was motivated by the LSTM and was proposed by Cho et al. in 2014 [9]. Similarily to the LSTM, the GRU has gates, but only a reset and update gate. It also omits the memory cell, making the GRU simpler and computationally cheaper. An empirical evaluation [10] deemed both GRUs and LSTMs superior to vanilla RNNs, but they could not decide which of the two gated units is better.

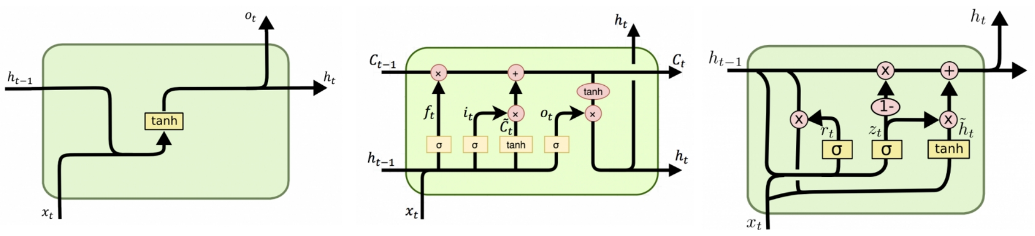

Figure 2.10: **Left:** Vanilla RNN cell. **Middle:** LSTM cell. **Right:** GRU cell. Source: link

2.4.5 Transformers

In natural language processing (NLP) and other sequence processing tasks, RNNs were the conventional approach for a long time. However, in 2017, Vaswani et al. [64] proposed the Transformer architecture.

Previously, different *attention-mechanisms* had been used in conjunction with recurrent units (e.g. [20] [68] [42]), but Vaswani et al. were able to achieve state of the art results on language translation tasks using attention without using recurrent units. The Transformer architecture solely uses self-attention mechanisms, giving the paper the fitting title "Attention is All you Need".

The concept of self-attention works by making every element in a sequence "attend" to other parts of the same sequence, which leads to a better understanding. Intuitively, a word alone has a meaning, but by looking at the whole sentence, more meaning is given to the word.

The specific type of attention they used is the scaled dot-product attention. The input consists of $\boldsymbol{Q}, \boldsymbol{K}$ and $\boldsymbol{V}$, which are the queries, keys and values of the input embeddings. d_k is the key and query dimension.

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V} \tag{2.12}$$

The matrices are obtained by multiplying the input embeddings (one embedding per element in the input sequence), $\boldsymbol{X}$, with weight matrices.

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q \qquad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K \qquad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V \tag{2.13}$$

Looking at the i-th row in the output of the attention mechanics, it will be a weighted sum of the value of all elements in the sequence. The weighted sum will tell the elements in the sequence how much it should attend to all of the elements' values, including itself. Naturally, each element will attend the most to itself, but also other elements' values, as they contain useful information.

The Transformer consists of an encoder consisting of a number of self-attention blocks followed by a decoder, also consisting of a number of self-attention blocks. The difference is that the decoder also processes previous outputs before considering the encoder output. For instance, to translate "Just do it" to Spanish, the sentence is first processed by the encoder blocks and given to the decoder. The decoder outputs "Solo". Then "Solo" is given as input to the decoder, alongside the same encoder output, producing "Solo hazlo".

Recently, Transformer-based models, like the Vision Transformer (ViT) [16], have been used in computer vision with compelling performance. As Transformers take sequences as input, images are split into a sequence of fixed-size patches when given as input.

2.4.6 Recent Research

For a long time, CNNs were dominant in computer vision tasks. More recently, transformer-based models like the ViT has shown great performance, but interestingly, very recent research has proposed architectures that differ from traditional CNNs and Transformers.

Tolstikhin et al. [63] argue that CNNs and Transformer-based models are sufficient for great performance but not necessary. With their MLP-Mixer architecture, they were able to obtain competitive scores on benchmarks without using convolutions. As with Transformer-based models, they used fixed-size patches as input representation. The authors of ”Patches are all you need?” [1] argues that representing the inputs as patches might be a key to great performance. Their very simple ConvMixer architecture achieves great results, using patches as input representation.

Chapter 3

Methods

3.1 Video Prediction

Video prediction is the task of predicting futures frames given past video frames. Approaches to this are similar to approaches for conditioned image generation. Generative modelling have different approaches, including generative adversial networks [38][44][41], variational autoencoders [2][13], autoregressive models [31][67] and flow based [37] models. These different approaches all have their strengths and weaknesses [6].

The VQVAE framework is chosen because: First, it has the ability to generate high quality and high resolution output. Second, the training progress is simple and efficient. Third, it has the ability to conditionally sample, which is required to model future frames.

The VQVAE framework includes a vector quantized autoencoder and an autoregressive model named PixelSNAIL. The background and a high-level view of these models are given in this chapter. Implementation details of the exact models that are used will be given in the next chapter.

3.2 Autoencoders

3.2.1 Vanilla Autoencoders

Autoencoders are neural networks trained to copy the input. Of course, it is possible to use an identity function, but this does not produce anything valuable. Instead, autoencoders are constrained to be unable to copy the input perfectly by forcing it to compress the input through a *bottleneck* layer. This forces autoencoders to learn useful features of the data it is trained on.

Autoencoders can be viewed as consisting of two neural networks, an encoder g and a decoder f. Both of these networks are built like other neural networks with linear layers, convolutions, dropout etc. The encoder translates the input to a lower-dimensional *latent code*, while the decoder tries to recover the original input using the latent code. The goal of the autoencoder is to reconstruct the original input as well as possible, so we want to find the parameters, $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, of the encoder and decoder that minimizes the loss.

$$\boldsymbol{\theta}^*, \boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\phi}} \frac{1}{n} \sum_{i=1}^{n} L(g_\phi(f_\theta(x)), x) \tag{3.1}$$

The loss function is a reconstruction loss, telling how well the reconstruction is. Various loss functions can be used, for example, the simple mean-squared error. Notice that only inputs are needed for this kind of training, as the desired output is just the input itself. This makes the learning process *unsupervised*. Autoencoders can learn useful features without the need of labelling of data, which in many cases is expensive.

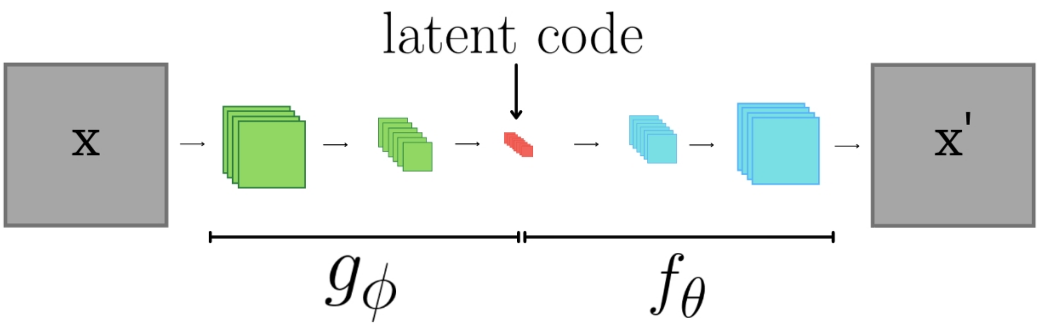

Figure 3.1: An illustration of a convolutional autoencoder.

3.2.2 Variational Autoencoder

In short, the Variational Autoencoder (VAE) [34] works by encoding the input to a probability distribution instead of encoding the input into a fixed latent code $p_{\boldsymbol{\theta}}(\boldsymbol{z})$. The VAE has roots in *Variational Bayesian methods*, so it will first be presented as a probability model. Then, a more practical deep learning point of view will be presented.

Probability Model View

Consider a dataset $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ and assume the data is generated from a process using a randomly sampled latent code $\boldsymbol{z}$. The generation process works by sampling $\boldsymbol{z}_i$ from a prior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$, and then sampling $\boldsymbol{x}_i$ from a conditional distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$. However, both the parameters $\boldsymbol{\theta}$ and the sampled latent codes are unknown.

Having a good approximation of the parameters $\boldsymbol{\theta}$ makes it possible to mimic the generation process and generate artificial data that is similar to the real data. Another interesting use case is utilizing the posterior distribution (equation 3.2) to encode that sample $\boldsymbol{x}$ back into the latent code $\boldsymbol{z}$.

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x})} \tag{3.2}$$

The optimal parameters can be found by maximizing the probability of generating real data samples.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} p_{\boldsymbol{\theta}}(\boldsymbol{x}_i) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \tag{3.3}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{z})d\boldsymbol{z} \tag{3.4}$$

However, looking at equation 3.4, the marginal probability is intractable due to the integral. The posterior in equation 3.2 is also intractable for the same reason. To combat these challenges, an approximation of the posterior, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, parameterised by $\boldsymbol{\phi}$ is introduced. This approximation should be as close as possible to the true posterior and the *Kullback-Leibler divergence* (KL-divergence, D_{KL}) is used.

$$D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) = \int q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})} d\boldsymbol{z} \tag{3.5}$$

This can be rewritten to equation 3.6.

$$\begin{aligned} &D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) \\ &\quad = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) + D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) - \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) \end{aligned} \tag{3.6}$$

Then, this is rearranged to equation 3.7.

$$\begin{aligned} &\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) - D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) \\ &\quad = \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) - D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) \end{aligned} \tag{3.7}$$

The left-hand side consists exactly of what should be maximized and is called the *variational lower bound*. As D_{KL} is non-negative, the variational lower bound is a lower bound on $p_{\boldsymbol{\theta}}(\boldsymbol{x})$. By maximizing the variational lower bound, the probability of generating real samples is maximized, and the difference between the distributions is minimized. The posterior in the left hand side is still intractable, so the right hand side can be maximized instead.

Taking a look, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ is given $\boldsymbol{x}$ and produces a distribution over values of $\boldsymbol{z}$ from which $\boldsymbol{x}$ could have been generated from. $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ is given $\boldsymbol{z}$ and produces a distribution over values of $\boldsymbol{x}$. If neural networks are used for these distributions, this structure resembles the vanilla autoencoder, as seen in Figure 3.2.

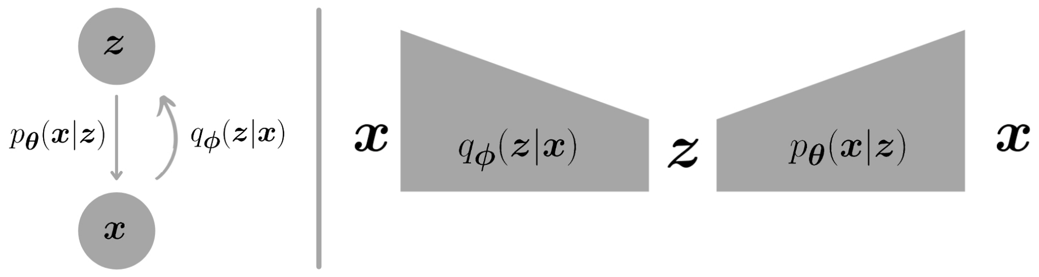

Figure 3.2: Resemblance of autoencoder

Deep Learning View

As with the vanilla autoencoder, the VAE consists of two neural networks. The encoder $q_\phi(z|x)$ is a neural network that is given an input x and produces parameters for a distribution of the latent code z.

A latent code z is sampled from the encoder distribution $q_\phi(z|x)$ and given to the other neural network, the decoder. Then, the decoder produces a conditional distribution of x, $p_\theta(x|z)$. The reconstruction of the original input will either be a sample or the expected value of $p_\theta(x|z)$

The true posterior that $q_\phi(z|x)$ tries to approximate is assumed to be a multivariate Gaussian with a diagonal covariance matrix. Therefore, the encoder $q_\phi(z|x)$ is a neural network that outputs parameters (means and standard deviations) to a similar Gaussian distribution.

The prior, $p_\theta(z)$, is set to a unit Gaussian distribution. As we want the variational lower bound to be maximized, a loss function can be the negative variational lower bound.

$$\begin{aligned} L_{\text{VAE}} &= -(\mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z) - D_{KL}(q_\phi(z|x)||p_\theta(z))) && (3.8)\\ &= -\mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z) + D_{KL}(q_\phi(z|x)||p_\theta(z)) && (3.9)\end{aligned}$$

In practice, the left term can be seen as a *reconstruction loss*. The higher the probability of the original input, the better the reconstruction. Often, a loss like MSE is used instead, and the two different losses has to be scaled, with the scaling α being a hyperparemeter.

$$L_{\text{VAE}} = (\alpha)(x - \hat{x})^2 + (1-\alpha)D_{KL}(q_\phi(z|x)||p_\theta(z)) \tag{3.10}$$

An argument for using MSE as the reconstruction loss can be that if distribution $p_\theta(x|z)$ is a Gaussian with unit covariance matrix, the log of the Gaussian will simply be a scaled MSE with an added constant. Here, μ is the mean and expected output from the decoder.

$$\log p_{\theta}(\boldsymbol{x}|\boldsymbol{z}) = \log \left(\frac{1}{\sqrt{(2\pi)^k |I|}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^{\mathrm{T}} I (\boldsymbol{x}-\boldsymbol{\mu}) \right) \right) \tag{3.11}$$

$$= c|\boldsymbol{x}-\boldsymbol{\mu}|^2 + k \tag{3.12}$$

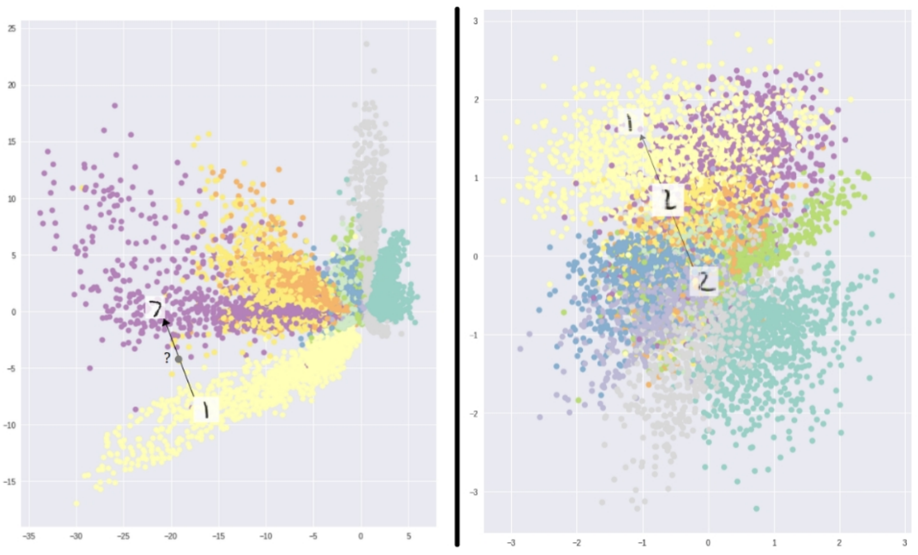

Figure 3.3: Latent space of a VAE trained on digits.
Left: Only reconstruction loss. **Right:** Reconstruction loss + D_{KL}
Source: link (edited)

The second term can be seen as a regularizer, where you force the approximation to stay relatively close to a unit Gaussian. This forces the approximation to give similar data points similar latent codes and prevent gaps in the latent space, as seen in Figure 3.3.

With the latent space being close to prior $p_{\theta}(\boldsymbol{z})$, which is set to a unit Gaussian, it is possible to sample $\boldsymbol{z}$ from a unit Gaussian instead of the encoder, to make the decoder generate samples that are not seen in the training set, making the VAE a *generative model*. Without including the D_{KL} in the loss, it would not be possible to generate new samples, as most of the latent space would probably be noise.

Reparameterization Trick

Generating the latent code $\boldsymbol{z}$ involves sampling from a distribution. The problem is that this is stochastic, and it is not possible to backpropagate through it. A clever solution is the *reparameterization trick*, which moves the source of randomness to an external node.

In the Gaussian case, z can be obtained by first sampling a random variable ϵ from a unit Gaussian. Then, z is obtained by equation 3.13, where σ and σ are obtained from $q_\phi(z|x)$ and $\odot$ is element-wise multiplication. This way, the gradient can backpropagate through the encoder as well.

$$z = \mu + \rho \odot \epsilon \tag{3.13}$$

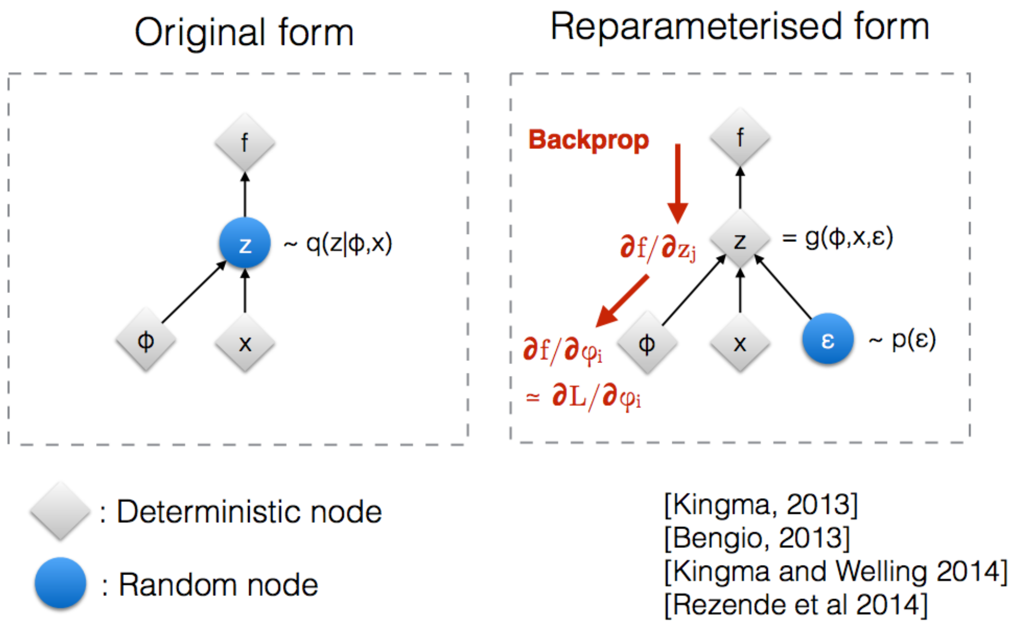

Figure 3.4: Reparameterization trick, allowing gradient to flow
Source: Kingma's NIPS 2015 workshop slides

3.2.3 Vector Quantized Variational Autoencoder

Discrete Latent Code Space

There are two key differences between Vector Quantized Variational Autoencoders (VQ-VAE) and VAEs. VQ-VAEs encode the input into a *discrete* latent code. Oord et al. [50] justify this choice by saying that discrete representations "are potentially a more natural fit for many of the modalities we are interested in. Language is inherently discrete, similarly speech is typically represented as a sequence of symbols. Images can often be described concisly by language".

The other key difference is that in VAEs, the posterior and prior are typically assumed to be Gaussian. In VQ-VAEs, the posterior and prior are categorical distributions.

The latent code space, also known as the *codebook*, is defined as $\boldsymbol{e} \in \mathbb{R}^{K \times D}$, where K is the number of latent codes and D is the dimension of the latent codes. The K latent codes will be $\boldsymbol{e}_i \in \mathbb{R}^D, i \in 1, \dots, K$. The latent code is sampled from the posterior distribution $q(\boldsymbol{z}_q|\boldsymbol{x})$ defined as

$$q(\boldsymbol{z} = \boldsymbol{e}_k|\boldsymbol{x}) = \begin{cases} 1, & \text{if } k = \arg\min_j ||\boldsymbol{z}_e(\boldsymbol{x}) - \boldsymbol{e}_j||_2. \\ 0, & \text{otherwise.} \end{cases} \tag{3.14}$$

The posterior is deterministic, and the encoder output $\boldsymbol{z}_e(\boldsymbol{x})$ is quantized into $\boldsymbol{z}_q(\boldsymbol{x})$ by nearest neighbour lookups in the codebook. Note that $\boldsymbol{z}_q$ consists of several codebook vectors. For example, a $256 \times 256 \times 3$ image may be compressed into $32 \times 32 \times 1$ codebook vectors. The prior distribution is simply a uniform distribution over the codebook.

Training

The loss function (equation 3.15) used to train VQ-VAE is similar to the VAE loss function, but there are notable differences. Because the prior is uniform and the posterior is deterministic and categorical, the KL-divergence is constant and omitted from the loss function.

The reconstruction loss is the same as for the VAE, but the codebook loss and commitment loss are two new terms.

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \underbrace{||\boldsymbol{x} - \hat{\boldsymbol{x}}||_2^2}_{\text{reconstruction loss}} + \underbrace{||\text{sg}[\boldsymbol{z}_e(\boldsymbol{x})] - \boldsymbol{e}||_2^2}_{\text{codebook loss}} + \beta \underbrace{||\boldsymbol{z}_e(\boldsymbol{x}) - \text{sg}[\boldsymbol{e}]||_2^2}_{\text{commitment loss}} \tag{3.15}$$

The codebook is learned through *Vector Quantization* (VQ). This works by using a distance measure, commonly the squared l_2 error, to move the latent code $\boldsymbol{e}_i$ closer to the encoder output $\boldsymbol{z}_e(\boldsymbol{x})$. This is the codebook loss, which is solely used for training the codebook. To only train the codebook, the stopgradient (sg) operator is used. It is defined as the identity function, but with zero partial derivatives.

The sg-operator is similarly used in the commitment loss to train the encoder. The commitment loss trains and encourages the encoder to output $\boldsymbol{z}_e$ close to the chosen codebook vector, preventing the encoder outputs from fluctuating between codebook vectors.

Notice that the nearest neighbour codebook lookup does not have a gradient. Instead, the gradients from the decoder input $\boldsymbol{z}_q$ are simply copied to the encoder output $\boldsymbol{z}_e$. The idea is that the encoder output should be close to the codebook vector due to the commitment loss and can be used to approximate $\nabla \boldsymbol{z}_e \approx \nabla \boldsymbol{z}_q$.

Hierarchical Latent Code Space

In [54] Razavi et al. proposed the VQ-VAE-2, an improvement of VQ-VAE. Instead of a single vector quantized code, a hierarchy of vector

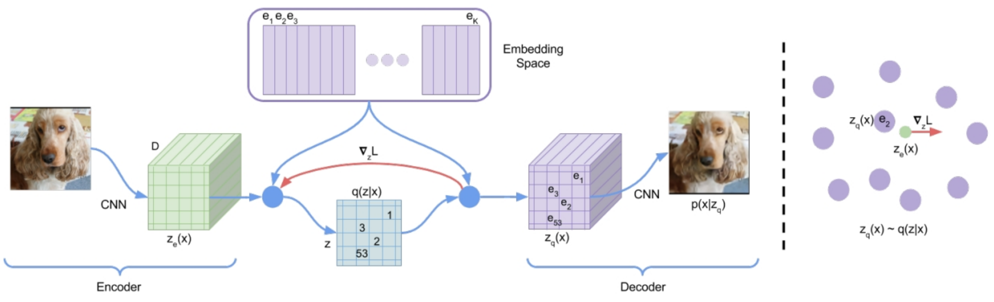

Figure 3.5: **Left:** A figure describing the VQ-VAE. **Right:** Visualization of the latent code space with a gradient estimation.
Source: [50]

quantized codes of different dimensions is used. The motivation is to capture the more local information with higher dimensional codes and the more global information with lower dimensional codes. Compared to the VQ-VAE, the reconstructions are of higher fidelity.

Another difference is that the codebook loss is removed. Instead, the codebook is updated by an exponential moving average as described in equations 5.4. This was proposed as an idea in the original VQ-VAE. The latent codes are simply an exponential moving average of decoder outputs.

$$\begin{aligned} N_i^{(t)} &= \gamma N_i^{(t-1)} + (1-\gamma) n_i^{(t)} \\ \boldsymbol{m}_i^{(i)} &= \gamma \boldsymbol{m}_i^{(t-1)} + (1-\gamma) \sum_{j=1}^{n_i^{(t)}} \boldsymbol{z}_{i,j}^{(t)} \\ \boldsymbol{e}_i^{(t)} &= \frac{\boldsymbol{m}_i^{(t)}}{N_i^{(t)}} \end{aligned} \tag{3.16}$$

Learning the Prior

The prior during training is kept at a uniform categorical distribution. However, learning prior distributions is now common practice and improves the performance of latent variable models [8] like the VQ-VAE. Thus, a prior distribution is learned in a separate process after training the codebook, encoder and decoder.

In a VAE, new outputs are generated by giving samples from the prior to the decoder. In VQ-VAE the samples will be closer to what the decoder were given during training by having a learned prior. The prior is learned

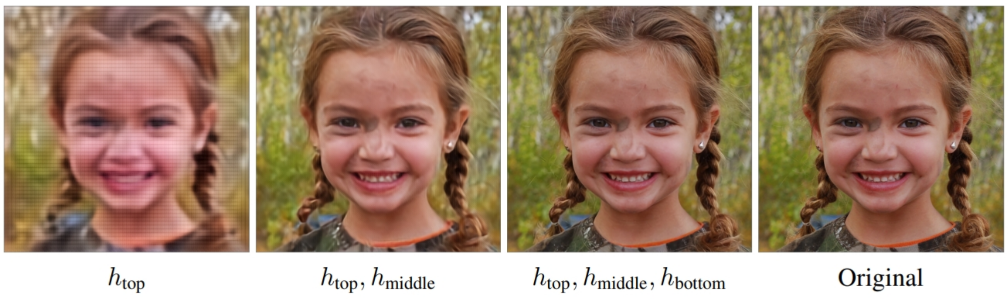
h_{top}
$h_{\text{top}}, h_{\text{middle}}$
$h_{\text{top}}, h_{\text{middle}}, h_{\text{bottom}}$
Original

Figure 3.6: VQ-VAE-2 reconstruction with 3 hierarchical latent codes. h_{middle} and h_{bottom} add details and texture to h_{top}.
Source: [54]

as an *autoregressive model*, allowing us to generate new latent codes with ancestral sampling.

3.3 Autoregressive Model

In a VQ-VAE an image of $n \times n$ pixels can be compressed into $m \times m$ codebook vectors, where $m < n$. The goal of the autoregressive prior model is to assign a probability distribution $p(\boldsymbol{z})$, giving the probability of each latent code $\boldsymbol{z}$. The latent code can be seen as a sequence of $m \times m$ codebook vectors, making $p(\boldsymbol{z})$ a joint distribution. It can be written as a product of conditional distributions over the codebook vectors:

$$p(\boldsymbol{z}) = \prod_{i=1}^{m^2} p(\boldsymbol{z}_i | \boldsymbol{z}_1, \ldots, \boldsymbol{z}_{i-1}) \tag{3.17}$$

$p(\boldsymbol{z}_i | \boldsymbol{z}_1, \ldots, \boldsymbol{z}_{i-1})$ is the probability of the i-th codebook vector given all the previous codebook vectors. A latent code can be generated by sampling a codebook vector one by one.

The sequence is usually created by raster scanning the latent code, starting at the top and going through every row one by one.

Optionally, the distribution can also be conditioned on some global information $\boldsymbol{h}$, making the conditional distribution $p(\boldsymbol{z}_i | \boldsymbol{z}_1, \ldots, \boldsymbol{z}_{i-1}, \boldsymbol{h})$. For generating images, $\boldsymbol{h}$ can, for example, be a class label.

3.3.1 Approaches to Model the Conditional Distribution

These approaches have traditionally been used directly on images, so they will be described as such. In VQ-VAEs, they are used on latent codes, but as the latent codes are $m \times m$ codebook vectors, you can think of these codebook vectors as pixels.

RNN

Using RNNs to model (this includes other cells like LSTMs and GRUs) $p(z_i|z_1, \dots, z_{i-1})$ is an obvious choice, as we can sequentially feed the pixels. RNNs, like PixelRNN [49], have been shown to perform great in autoregressive generative tasks.

A challenge to RNN-based models is that the cells propagate information by sending states from one time step to the next, making long-range dependencies challenging. Also, RNNs are slow to train, as the model has to be fed pixels sequentially.

Causal Convolutions

An alternative approach uses *causal* convolutions . Causal convolutions are a type of convolution where you mask parts of the filters to ensure that the order of the input sequence is not violated. Convolutions are fast as you can parallelize computations to a much higher degree compared to RNNs that are sequential in nature. Note that this only applies during training. During training, you have all the pixels needed, so you can parallelize the convolutions, but when generating new latent codes, you have to sequentially generate one pixel at a time.

While this approach is very fast, the receptive field is small at the start and grows with the number of convolutional layers, making the long-range dependencies challenging. The receptive field also has some blind spots, as illustrated in 3.7. These special CNNs do not perform as well as RNNs, unless some tricks are used.

To combat the blind spot problem, two different convolutions are used. One vertical stack is conditioned on rows above, and one horizontal stack is conditioned on the current row so far, as well as the output from the vertical stack.

Another trick that is used in [49] is the use of a gated activation unit instead of the rectified linear unit. The authors reason that multiplicative units in LSTM/GRU cells may help model more complex interactions.

Self-attention

Auto-regressive models based on self-attention have also shown success. The Image Transformer [52] is a model similar to the original Transformer [64]. For use in autoregressive tasks, the Image Transformer only consists of the decoder part, as there is no sequence to condition the output on, expect previous outputs.

A challenge is that self-attention has quadratic complexity with regard to the input sequence length. A 256×256 image is a sequence of 65536 pixels, a number higher than any sequence in traditional NLP-tasks.

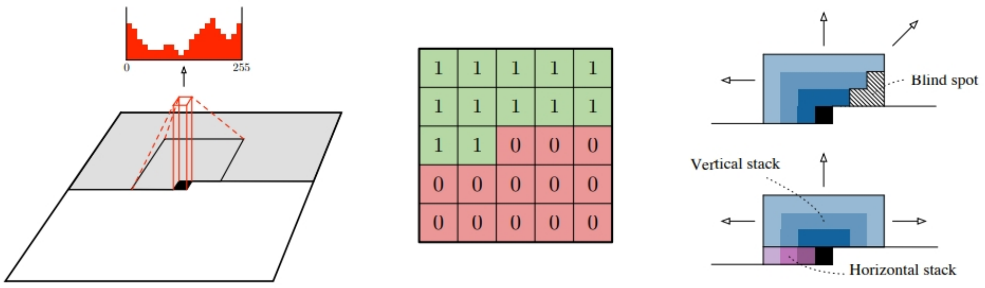

Figure 3.7: **Left and middle:** Visualization of the causal convolution with its mask. **Right:** The receptive field has blind spots, so two different convolutions are used instead.
Source: [51]

In the Image Transformer Parmar et al. introduce *local* self-attention to reduce the computational costs. Local self-attention restricts the sequence to a local neighbourhood around the query position.

3.3.2 PixelSNAIL

Chen et al. proposed PixelSNAIL [7], an architecture that combines causal convolutions with self-attention. The authors argue that causal convolutions provide high bandwidth access in a local context, while attention only provides a small amount of information, but in a global context. The different methods complement each other, with causal convolutions taking care of local features, while attention takes care of the global ones.

The authors tested this hypothesis by randomly initializing a model and checking the middle pixel's sensitivity to other pixels by finding the gradient with respect to the other input image pixels.

$$\nabla_x \log p(x_{middle}| \ldots) \tag{3.18}$$

This suggests that a randomly initialized PixelSNAIL with comparable amount of parameters is able to attain information in a larger context. However, keep in mind that dependencies with a small gradient magnitude may exist, as well as that training can have an effect on the receptive field.

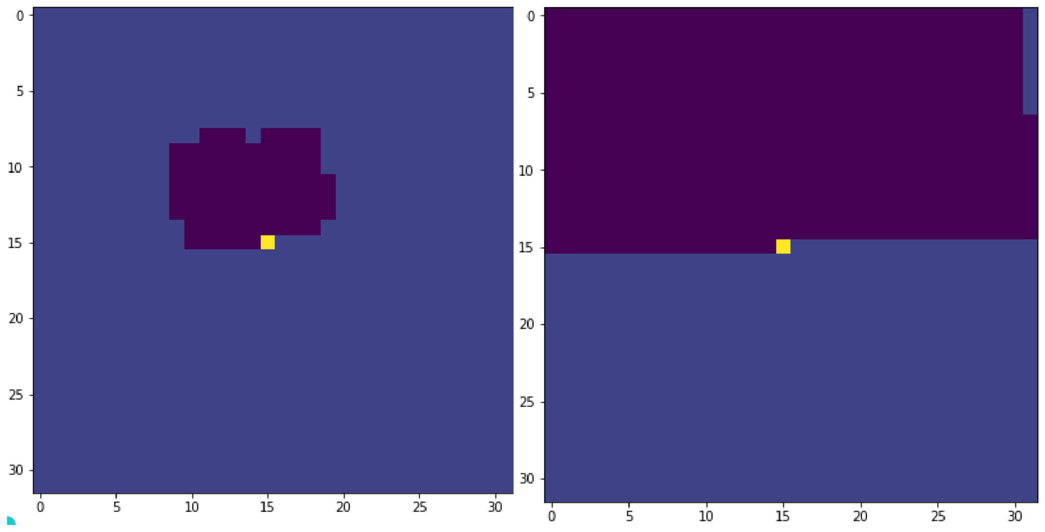

Figure 3.8: Yellow pixel indicates the pixel under inspection, purple pixels indicate pixels with derivative magnitude above 0.001. **Left:** Gated PixelCNN [51]. **Right** PixelSNAIL [7]

Part II

The project

Chapter 4

Experiment Setup

Implementations and examples are available at `https://github.com/mattiasxu/master_project_github`. The implementations are done in Python, using the PyTorch [53] deep learning framework. On top of that, the PyTorch wrapper PyTorch Lightning was used. PyTorch Lightning removes a lot of boilerplate code and simplifies changing hardware, checkpointing, logging, running distributed training and more.

4.1 Dataset

4.1.1 Collecting Data from Driving Simulator

As mentioned in the introduction, the goal of this project is to be able to model a driving environment. A real-world driving environment can be extremely complex and varied, depending on weather, other people's behaviour, location, etc. In addition, collecting data from a real-world environment can be both time consuming and expensive to collect.

For the project, data from a driving simulator was used. With a simulator, data is more easily collected, and it is possible to adjust the environment's complexity. A driving simulator is a good way to collect data, both for its simplicity and ability to scale up the problem quickly. Suppose the model can successfully learn how to model an easier environment, the simulator can quickly be tuned to produce a more complex environment, by for example having a bigger variety of weather conditions.

CARLA (Car Learning to ACT) [17], an open-source simulator, was used. It is a high fidelity simulator built on the game engine Unreal Engine 4 (UE4) and features various settings on urban layouts, car models, buildings, pedestrians and light and weather conditions. CARLA offers a flexible setup of sensors, like camera, LIDAR and radar. However, this project will solely focus on images from the car's point of view and will only use a single camera sensor collecting video from the front of the car.

Figure 4.1: CARLA [17] high fidelity simulator

The resolution of the camera was set to 256 × 256.

"Town03" will be used, featuring a varied driving environment with a roundabout, tunnels, highway etc. 11.5 hours of driving was recorded. In total, 208144 images were shot at an interval of 0.2 seconds. The car was driven by CARLA's built-in autopilot, which is hard-coded and drives randomly around the map.

4.1.2 Training, Validation and Test Set

The dataset was split into training, validation and test sets at a 80%/10%/10% split. The first 80% of the driving went to the training set, the next 10% went to the validation set, and the last 10% went to the test set.

Recall that the collected data are images, while video is what we need. The split was not done randomly, as doing so would break the continuity of the images. In addition, this split ensures that the validation and test sets are not too similar to the training set.

With a random split, there may be validation/test samples that are almost identical to training samples, where the difference is one shifted frame. Then, 15 out of 16 frames would already be seen in the training set, and there is a concern that a model will be able to perform well during validation and testing, even when overfit.

4.1.3 Preprocessing

Every pixel is zero-centred and scaled according to the channel mean and standard deviation. Mean and standard deviation per input channel are found across the training set. Every pixel in the images is normalized by substracting the mean and dividing by the standard deviation.

$$p_{\text{normalized}}[\text{channel}] = \frac{p[\text{channel}] - \mu[\text{channel}]}{\sigma[\text{channel}]} \tag{4.1}$$

As the raw data is in image form, they have to be concatenated to form a video. To take advantage of the continuity of the images, a sliding window can be used to create video samples of 16 frames. For the training set, a step size of 1 is used to create as many different video samples as possible. The images are repeatedly used as the videos contain duplicate frames. Thus, the process of creating video from images is done on the fly. Given an index i, the dataset will return image i to $i + 15$ concatenated. In total, 166500 video clips are created from the 166515 images in the training set.

For the validation and test set, the step size of the sliding window is increased to 8. The model is expected to have similar loss when given very similar samples, so validation can be sped up by increasing the step size, without losing much accuracy in the validaiton loss.

4.1.4 Latent Representation Dataset

While the input to the hierarchical VQVAE is in video space, the input to the two PixelSNAILs is in latent space. Thus, a dataset consisting of latent codes is needed.

After training the VQVAE, it will be used to create the latent representation dataset. Every video clip from the video datasets is encoded by the trained VQVAE and stored in a dataset as pairs of top and bottom latent codes.

The video dataset created samples on the fly, but this is not possible with the latent codes. However, the latent codes are of much smaller size, keeping the dataset at a tractable size. The latent codes are stored with their IDs, i. e., integers from $[0 - 511]$. When in use, these IDs are transformed into one-hot encodings.

4.2 Compressing Video with Hierarchical VQ-VAE

Autoregressive models are not typically used for generating high resolution, and they are usually benchmarked on images with a resolution of 32×32 or 64×64. With higher resolutions, these autoregressive models suffer from the curse of dimensionality. As the VQVAE encoder compresses the input to a tractable lower dimension, the autoregressive models can be used to model the prior distribution of the encodings.

The project's VQVAE is implemented with the same structure as VQVAE-2 from Razavi et al. [54], but adjusted for handling video input.

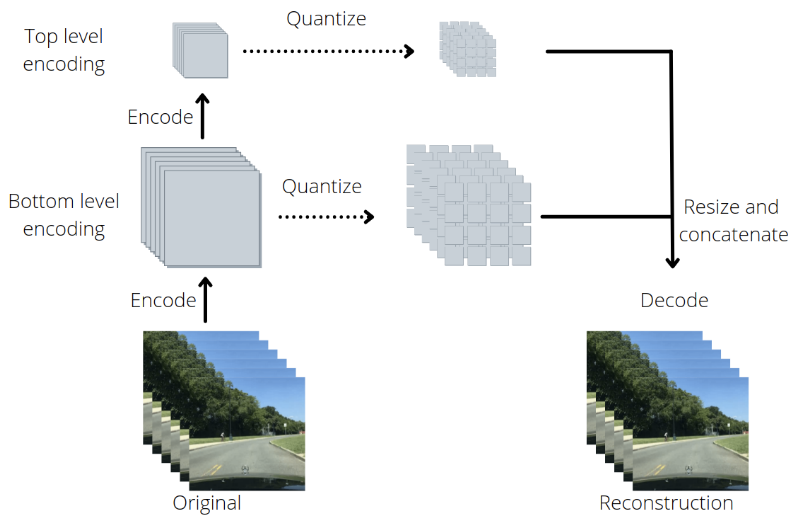

Figure 4.2: Overview of the VQVAE, with two levels

Video is essentially a stack of images, making them three-dimensional by adding an extra time dimension. To deal with this, 2D convolutions are swapped for 3D convolutions. Figure 4.2 provides a high-level overview of the model.

The video in the dataset consists of 16 frames of 256×256 resolution with three colour channels. Using 8-bit colour channels, the video consists of $3 \times 16 \times 256 \times 256 \times 8$ bits.

The bottom level encoder encodes the input by reducing the temporal dimension with a factor of 2 and the spatial dimensions with a factor of 4. The top level encoder takes this encoding and reduces it further by a factor of 2. In the end, the two encoding consists of $4 \times 32 \times 32$ and $8 \times 64 \times 64$ codebook vectors.

A single decoder takes upsampled top level encodings concatenated with bottom level encodings and creates the reconstruction.

Depending on the sizes of the codebooks, the reduction in bits required is greatly reduced. For instance, if the codebook sizes are 512, each individual code can be represented by $\log(512)$ bits. Thus, the total size of encodings is $4 \times 32 \times 32 \times \log(512) + 8 \times 64 \times 64 \times \log(512)$ bits. This is a 98.8% reduction in bits required.

4.2.1 Encoder

The bottom level encoder starts by reducing the temporal and spatial dimensions of the input using a convolutional layer, with filters of size $(4, 8, 8)$, stride of $(2, 4, 4)$ and padding of $(1, 2, 2)$. This gives the desired temporal reduction of 2 and the spatial reduction of 4.

The receptive field does not increase a lot from this single layer. Ideally, the receptive field should be sufficient to capture all needed dependencies.

As the receptive field of a convolutional network increases linearly with the number of convolutional layers, a deeper model is needed. The residual framework from [23] is used to ease the training of the deeper model.

The output of the convolutional layer is then given to a residual block. Between each residual connection, the residual layer consists of ReLU – Conv – ReLU – Conv, as seen in Figure 4.3. The first convolution outputs a chosen number of channels, while the second convolution takes the number of channels back.

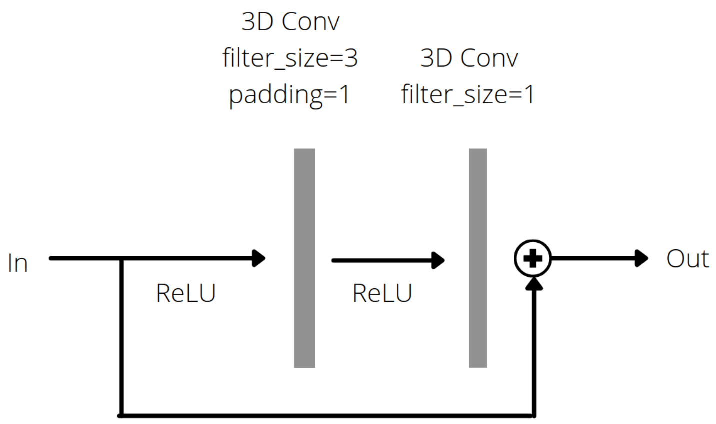

Figure 4.3: Residual layer

Finally, a last convolutional layer with filter size of 1 is applied to match the channels to the dimension of the codebook vectors.

The top level encoder is very similar, with the only difference being the first convolutional layer. The top level encoder uses a convolution with kernel size of 4, stride of 2 and padding of 1 to reduce all dimensions by 2.

4.2.2 Vector Quantizer

Vectors are quantized by a nearest neighbour lookup in the codebook. The distance used is simply the Euclidean distance. The codebook consists of a set of unique codes of a chosen dimension. Each number in the latent code is initialized by a uniform distribution between 0 and 1. The codebook acts like a lookup table, where each of the latent codes has a unique bit-encoded key.

4.2.3 Decoder

The decoder is of a very similar structure to the encoder but in reverse. The concatenated encoding is first fed into a residual block. Then, instead of downsampling using a convolution, the decoder upsamples the encodings using a transposed convolution [59]. To upsample the

$8 \times 16 \times 16$ concatenated encoding back to $16 \times 256 \times 256$, filters of size $(4, 8, 8)$, stride of $(2, 4, 4)$ and padding of $(1, 2, 2)$ is used.

4.2.4 Hyperparameters

Some parts of the VQVAE are left as hyperparameters to be tuned to explore the trade-offs of different implementation details without changing the overall architecture.

Due to the symmetric nature of the encoder and decoder, their counterparts receive the same parameters. The number of residual channels, the convolutional channels and the number of residual layers is kept the same between the two components.

For the vector quantizer, the number of codebook vectors and their dimension are hyperparameters to be tuned.

4.3 Autoregressive Prior Model

Since the video is decoded into two latent codes, two autoregressive models are used to model their prior distributions. The top level autoencoder generates the top latent code, while the bottom level autoencoder is conditioned on the generated top latent code and generates the bottom latent code.

The autoregressive models used are the same chosen by the authors of the VQVAE-2 [54], but adjusted to handle 3D data, which is also done by replacing 2D convolutions with 3D convolutions. The output is also adjusted to quantized output. With 512 latent codes, each "pixel" in the output has 512 values, containing the log probabilities of being the corresponding latent vector.

4.3.1 Top Prior Model

The top level autoencoder follows the PixelSNAIL [7] architecture and uses a combination of convolutions and attention to model the conditional distribution $p(x_i | x_1, \ldots, x_{i-1})$.

The ELU activation function is used. In contrast to ReLU, the ELU has negative values, which allows them to push mean unit activations closer to zero [11] and has an effect similar to batch normalization [28], which has shown to smooth the optimization landscape, allowing for faster and more predictable training [57].

The two main building blocks of the PixelSNAIL are the residual block and the attention block (Figure 4.6).

The residual block consists of layers where a single residual layer (see Figure 4.5) consist of ELU - Conv - ELU - Conv - Gated Activation before the residual connection. The gated activation works by feeding the output

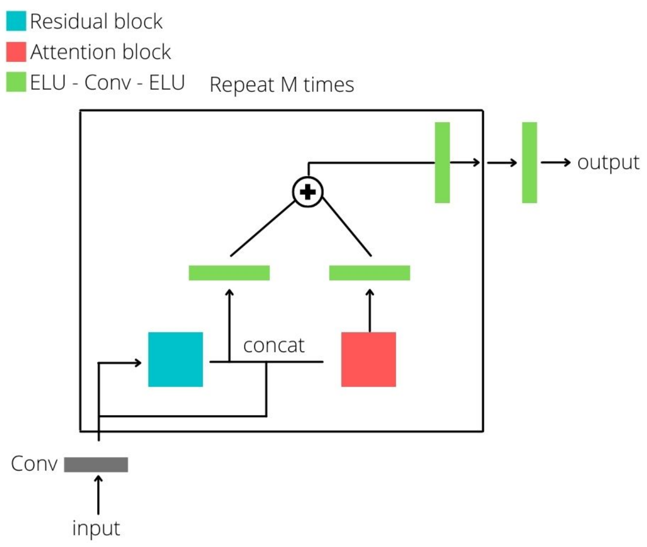

Figure 4.4: The PixelSNAIL architecture

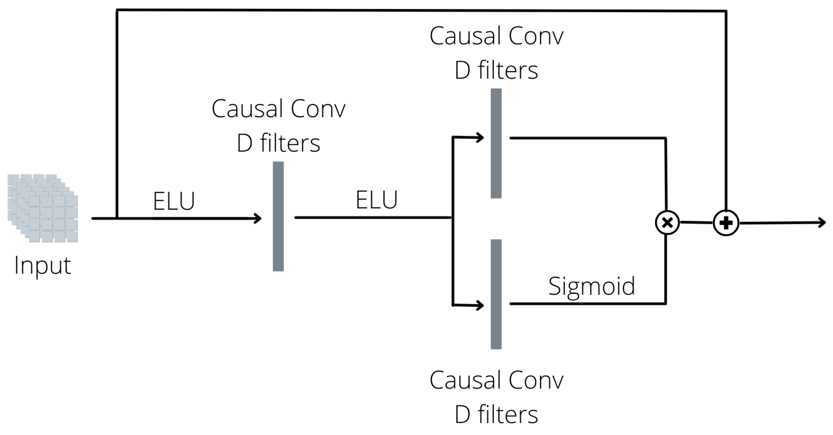

Figure 4.5: Residual layer with gated activation

of the first convolution into two separate convolutions with the same amount of filters. The output of one of the convolutions goes through a sigmoid activation. It is element-wise multiplied with the output from the other convolution, making the output a weighted average of this output. This resembles the gate mechanism found in LSTMs and GRUs.

The attention block consists of a single key-value lookup. 1×1 convolutions are used to create the query, key and value. To keep causality the attention mechanism has to be masked, so that pixels can only attend to previous pixels.

The rest of the PixelSNAIL architecture consists of more simple elements, like convolutions, additions and concatenations. The PixelSNAIL mainly consists of a repeatedly used block (see Figure 4.4), which includes the residual block and the attention block.

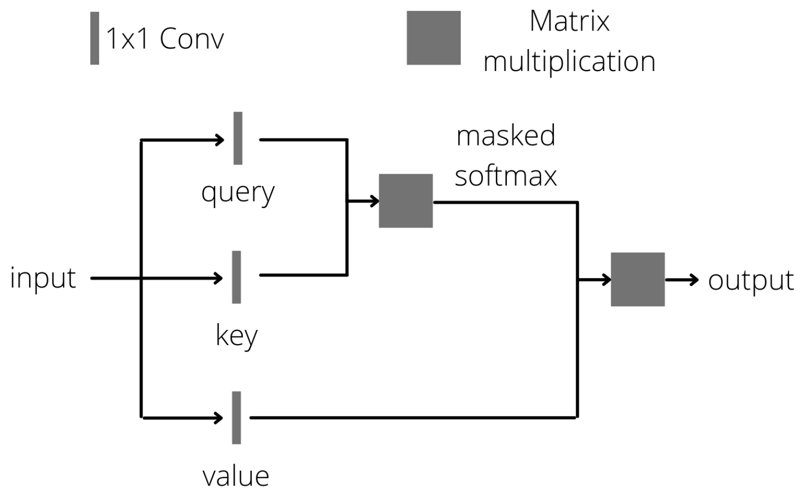

Figure 4.6: Attention Block

4.3.2 Bottom Prior Model

Recall that the top level code is supposed to capture more global features, like background colour, shapes etc., while the bottom level code is of higher dimension and adds details. Using attention makes sense for the top level code, as it is of low dimension and captures global features.

Due to the attention's $O(n^2)$ complexity with regards to the input length, using it on the bottom level encoding is computationally expensive. The bottom encoding is also supposed to only add local details. Attention's advantage is the global access to information, but this is unnecessary and not worth the computational cost. Therefore, the bottom level autoregressive model has a simpler architecture solely based on causal attention.

The bottom prior model is also conditioned on a top level code. The conditioning works by sending the top level code through a convolutional block before being added to the bottom prior model input.

4.3.3 Generative Modelling

During training, each output pixel is only dependent on previous pixels and can be calculated in parallel. However, this operation has to be done serially when generating an encoding. The first $n-1$ pixels have to be generated before the n-th pixel can be conditioned on them, so a total of $H \times W$ forward calls has to be done to generate one encoding.

The generation is implemented by using an input of all zeros and generating the pixels one by one by setting the n-th pixel in the input to the previous output. To condition on previous frames, the frames are concatenated with zeros to match the input size and given as input. Then, the model will generate the rest of the pixels one by one.

Chapter 5

Optimization

The hardware used in this project comes from Simula Research Laboratory and their partners' computing cluster, eX3[1]. Training, hyperparameter tuning, etc. was done on a single NVIDIA V100 Tensor Core GPU with 32 GB local memory.

5.1 Choice of Optimizer

The optimizer chosen is Adam, as it is known to be forgiving regarding setting the learning rate. As recommended in [35] the following parameters will be set to the default settings: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Even though Adam is adaptive and forgiving, it is beneficial to tune the learning rate.

Weight decay

For regularization purposes, weight decay will be used as it is shown to perform better than L_2 regularization [40]. Recall from chapter 2 that weight decay and L_2 regularization are very similar. In fact, weight decay is commonly mistaken for L_2 regularization. Due to this, the Adam optimizer in many deep learning libraries has an argument called weight decay that actually is L_2-regularization. The source of this may be that with SGD, L_2 regularization and weight decay can be identical.

With SGD, when the regularization term is explicitly added, the update will be:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha \nabla_{\boldsymbol{\theta}_n} (L + \lambda ||\boldsymbol{\theta}||_2^2) = \boldsymbol{\theta}_n - \alpha \nabla_{\boldsymbol{\theta}_n} L - 2\alpha\lambda\boldsymbol{\theta}_n \tag{5.1}$$

SGD with weight decay with no regularization term in the loss function will update the weights:

$$\boldsymbol{\theta}_{n+1} = \beta\boldsymbol{\theta}_n - \alpha \nabla_{\boldsymbol{\theta}_n} L \tag{5.2}$$

[1]https://www.ex3.simula.no/

If $\beta = 1 - 2\alpha\lambda$, the two updates are identical. However, with adaptive methods, weight decay and L_2 regularization will differ. The weights' magnitude will be penalized differently due to the adaptive gradients with an adaptive method like Adam. Weight decay penalizes weight magnitude separately from the steps taken regarding the loss function and therefore decays all weights by the same factor.

In PyTorch and other frameworks, Adam with weight decay is available with the AdamW optimizer, while the standard Adam optimizer only supports L_2 regularization.

5.2 Hierarchical VQVAE

5.2.1 Loss

The loss function used will be the version where the codebook is updated by a different process, so the codebook term seen in Section 3.1.3 is ommitted.

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||_2^2 + \beta||\boldsymbol{z}_e(\boldsymbol{x}) - \boldsymbol{e}||_2^2 \tag{5.3}$$

β is set to 0.25 and the codebook is updated with exponential moving averages with decay factor $\gamma = 0.99$ as done in [54].

$$\begin{aligned} N_i^{(t)} &= \gamma N_i^{(t-1)} + (1-\gamma) n_i^{(t)} \\ \boldsymbol{m}_i^{(i)} &= \gamma \boldsymbol{m}_i^{(t-1)} + (1-\gamma) \sum_{j=1}^{n_i^{(t)}} \boldsymbol{z}_{i,j}^{(t)} \\ \boldsymbol{e}_i^{(t)} &= \frac{\boldsymbol{m}_i^{(t)}}{N_i^{(t)}} \end{aligned} \tag{5.4}$$

5.2.2 Hyperparameter Tuning

The VQVAE has a few hyperparameters to be tuned. In addition to batch size and learning rate, choices regarding the number of channels in the encoder and decoder have to be made, while the codebook size and dimension of the quantizer also has to be decided.

Hypertuning was done with random search on values described in Table 5.1. 15 runs in total were completed, and each run lasted for two epochs.

All runs led to a validation loss in the same order of magnitude. Visually, even the model with the highest loss was able to produce reasonable reconstructions. This suggests that the model is relatively forgiving with regarding hyperparameters. However, one hyperparameter has a clear correlation with the validation loss. A higher codebook size (n_embed in

Hyperparameter	Values
Batch Size	8, 16, 32
Learning Rate	[0.01, 0.00001]
Encoder/Decoder Conv Channels	64, 128, 256
Encoder/Decoder Residual Blocks	[2, 12]
Encoder/Decoder Residual Channels	8, 16, 32, 64, 128
Codebook Size	64, 128, 256, 512
Codebook Vector Dimensions	32, 64, 128, 256

Table 5.1: Hyperparameters and their possible values. Learning rates are drawn from a log uniform distribution, the residual block number is drawn from a uniform distribution, and the rest are drawn from an uniform categorical distribution.

Figure 5.1) resulted in lower validation loss. This makes sense, as the codebook size puts a hard representational limit on the encodings. Other than that, it is hard to find any significant correlation between hyperparameters and validation loss.

A higher number of channels, residual blocks, and embed dimensions all increase the complexity and computational cost of the model, but they do not have a clear correlation to validation loss. This suggests that the model is capable enough even with a lower amount of parameters, as long as the codebook size is large enough.

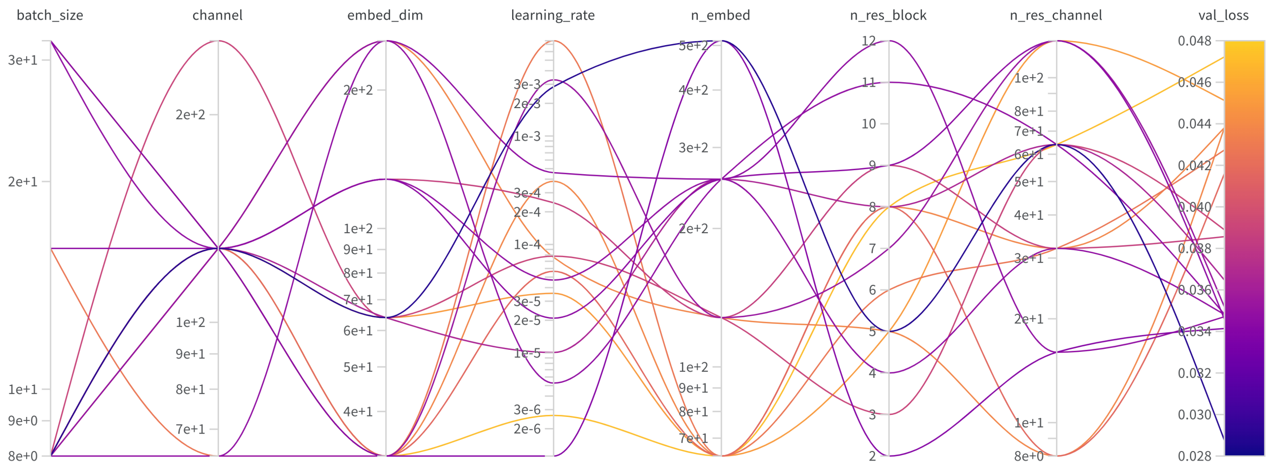

Figure 5.1: Results from 15 runs.

5.2.3 Final Training Run

Based on results from hyperparameter optimization described in the previous subsection, a set of hyperparameters were chosen. The chosen set was simply the set with the best validation loss. The learning rate was set to 0.0028 and the batch size was set to 8. The set of hyperparameters regarding the architecture can be seen in Table 5.2. The final run lasted for

Codebook Size	512
Codebook Vector Dimension	64
Residual Blocks	5
Residual Channels	64
Decoder/Encoder Channels	128

Table 5.2: Final hyperparameters for the architecture.

8 epochs which took 28 hours.

5.2.4 Loss Curves

The final loss over the training and validation set can be seen in Table 5.3 below. The training loss decreased very rapidly at the start of the first epoch and was left out of Figure 5.2, which was scaled to give a better look after the initial "hockey stick" curve.

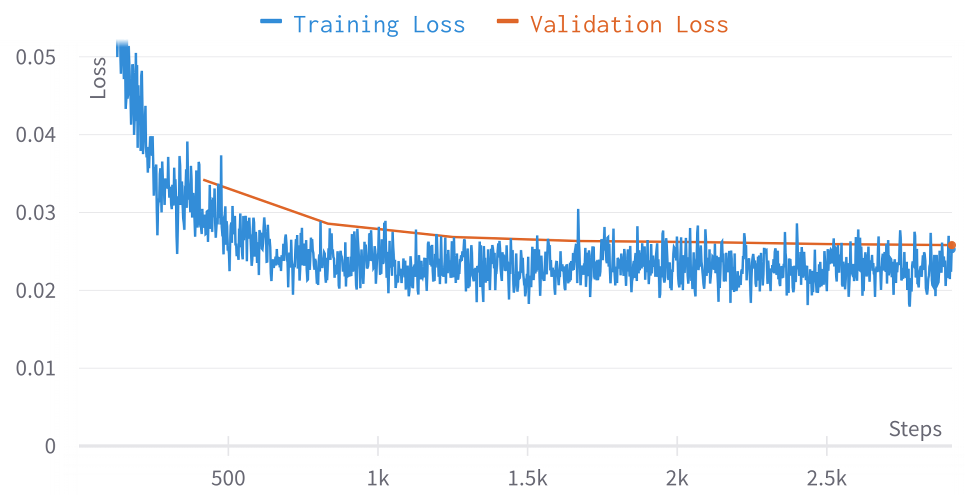

Figure 5.2: Training and validation loss from VQVAE training. The validation loss was only calculated at the end of every epoch.

Looking at the loss curves, the training loss is slightly noisy but converges. The validation loss also converges. It is stable and only slightly

Set	Training	Validation
Loss	0.025	0.026

Table 5.3: Final loss over the datasets.

higher than the training loss, suggesting that the model is regularized well. However, the validation loss was slightly decreasing for every epoch, suggesting that some additional performance could be squeezed out if given more time.

5.3 PixelSNAILs

The two PixelSNAIL models are trained separetly, with the top PixelSNAIL trained first. As these autoregressive models are quite powerful with many parameters, they take a long time to train and not much hyperparameter optimization was done.

As the PixelSNAIL tries to solve a classification problem with 512 classes for every ”pixel”, the negative log likelihood loss is used.

5.3.1 Top-level PixelSNAIL

The first training run was done with the same set of hyperparameters as described in the PixelSNAIL paper [7]. The model converged during training (see Figure 5.4), but it was unable to produce anything valuable. When conditioned on eight frames, the eight next frames were predicted to simply be a static image.

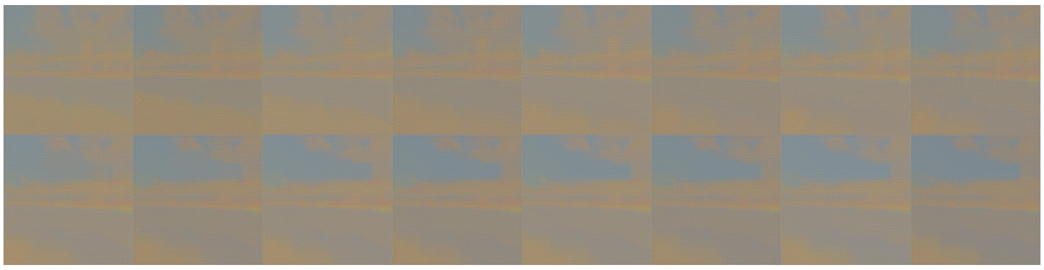

Figure 5.3: Reconstruction of eight conditioning frames and eight predicted frames. The last eight frames look static.

The original set of hyperparameters were used on 2D image data. With higher dimensional video data, the model needs to be more powerful. This can be done by making the model deeper. The two subsequent training runs were done with with more powerful models, scaled up by increasing the number of residual blocks in each SNAIL block. The model-specific hyperparameters can be seen in Table 5.4.

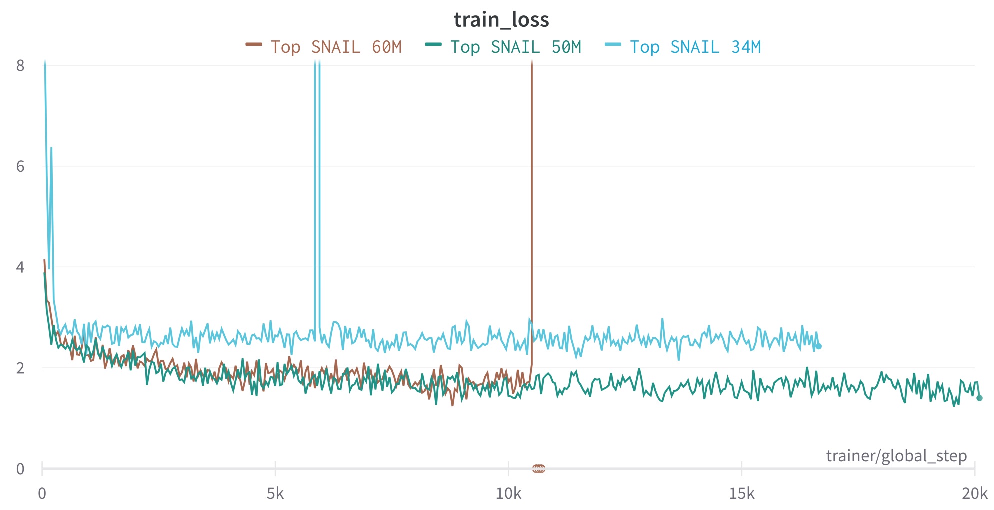

Figure 5.4: Loss curves from the three training runs.

The three models had approximately 34 million, 50 million and 60 million parameters. Looking at Figure 5.4, the scaled-up models were able to achieve a lower loss. However, going from six to eight residual blocks did not seem to help and only slowed down the training. The validation losses follows the training losses in all three cases, being slightly higher than the training loss.

The two big spikes the 34 million and 60 million parameter models had can be explained by one or more ”unlucky” minibatches containing many high loss samples in the start of an epoch.

Table 5.4: Top PixelSNAIL hyperparameters.

Conv Channels	256
Residual Blocks	4/6/8
SNAIL Blocks	5
Key Channels	16
Value Channels	128

The training of the 34 million and 50 million parameter were stopped after a long period with no improvement. The 60 million parameter model was stopped early as there was no significant difference between its loss and the 50 million parameter model loss. As the 50 million parameter model has a lower computational cost than the 60 million parameter model while having the same loss, it is selected as the final model to be used to be combined with the bottom-level PixelSNAIL.

The final run lasted for eight epochs, which took 37 hours.

5.3.2 Bottom-level PixelSNAIL

The bottom-level PixelSNAIL does not use the attention mechanism due to the higher dimension of the bottom-level encoding. Therefore, the model resembles a deep residual network. When there is no attention block in the PixelSNAIL, the number of residual blocks and SNAIL blocks decides how deep the model is. As the bottom-level encoding is more detailed and higher dimensioned than the top-level encoding, it is assumed that learning the conditional distribution over the bottom-level encoder is harder.

As the task is assumed to be difficult, the hyperparameters (see Table 5.5) were chosen to make the model as powerful as possible with a batch size of eight, given the hardware constraints. The bottom-level encoding is supposed to add details to the top-level encoding, which makes the top-level encoding critical for the bottom-level PixelSNAIL. Therefore, the residual conditioning stack is made quite deep.

In total, the model has 102 million parameters.

Table 5.5: Bottom PixelSNAIL hyperparameters.

Conv Channels	256
Residual Blocks	6
SNAIL Blocks	2
Condition Residual Blocks	8

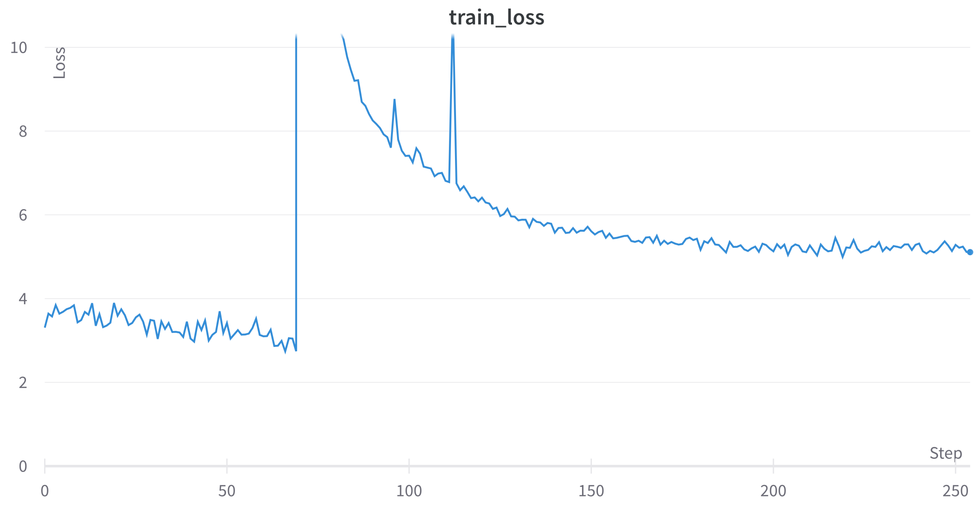

Figure 5.5: First training run of the bottom-level PixelSNAIL.

During the first training run, the loss quickly spiked (see Figure 5.5). It is suspected that this is due to exploding gradients during high-loss batches. The loss spiked to almost 4×10^{10} and a loss this high has a huge impact on the weight update. After the exploding gradients, the loss was not able to recover and got stuck in a local minima.

To handle this problem, the next training run was done with gradient clipping. All gradients with a norm higher than 0.5 were scaled down to match this norm.

From Figure 5.6 we can see that the loss was still spiking, but the spikes were limited in size. After the spikes, the loss was able to go back down to approximately the same loss before the spike.

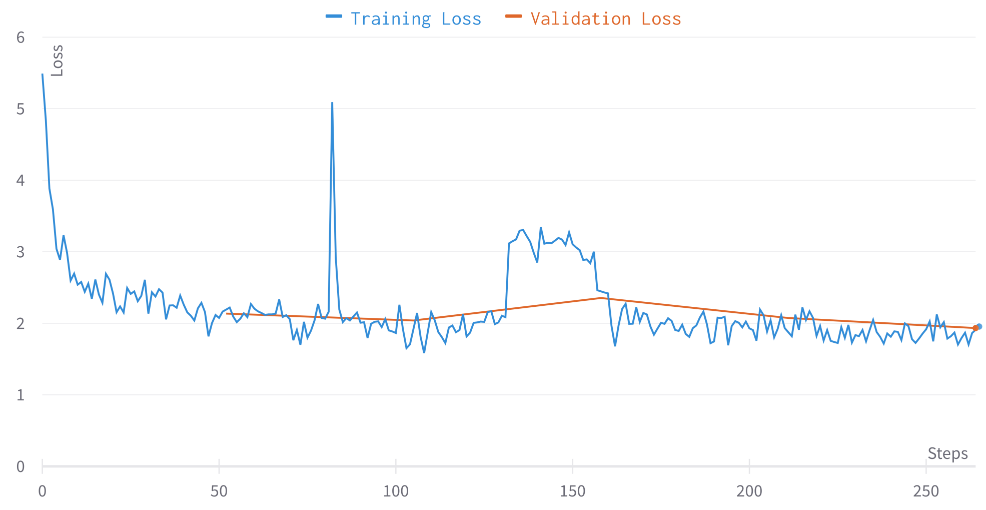

Figure 5.6: Second and final bottom-level PixelSNAIL training run.

The final training run lasted for five epochs, which took 38 hours.

Chapter 6

Results and Discussion

6.1 Hierarchical VQVAE

It is highly recommended to also look at the examples in video format. They are provided as GIFs at `https://github.com/mattiasxu/master_project_github`.

6.1.1 Quantitative Results

Image quality assessment is a challenge [62] and is a research topic in itself. This naturally extends to video quality assessment as well.

During training, the VQVAE was optimized to minimize the MSE, a commonly used metric. In Table 6.1, the MSE over the training, validation and test set is reported. This metric as a loss has its drawbacks, including that it tends to result in blurry images. This also applies to video, as will be shown later.

Note that this metric differs slightly from the training loss reported in the previous section. Alongside MSE, a commitment loss was included. The commitment loss was used solely to help the encoder commit to a codebook vector and not fluctuate between different codebook vectors, which is not directly related to the video quality. From the differences, we can see that the commitment loss caused a significant portion of the loss.

Table 6.1: Final loss for the hierarchical VQVAE.

	Training	Validation	Test
MSE	0.018	0.021	0.019

6.1.2 Qualitative Results

In Figure 6.2, three randomly picked video snippets from the test set are shown frame by frame with their reconstructions below. As the number of codebook vectors was chosen to be 512, the latent representation reduces the number of bits required to represent the image by 98.8% compared to raw RGB video assuming 8-bit colour channels.

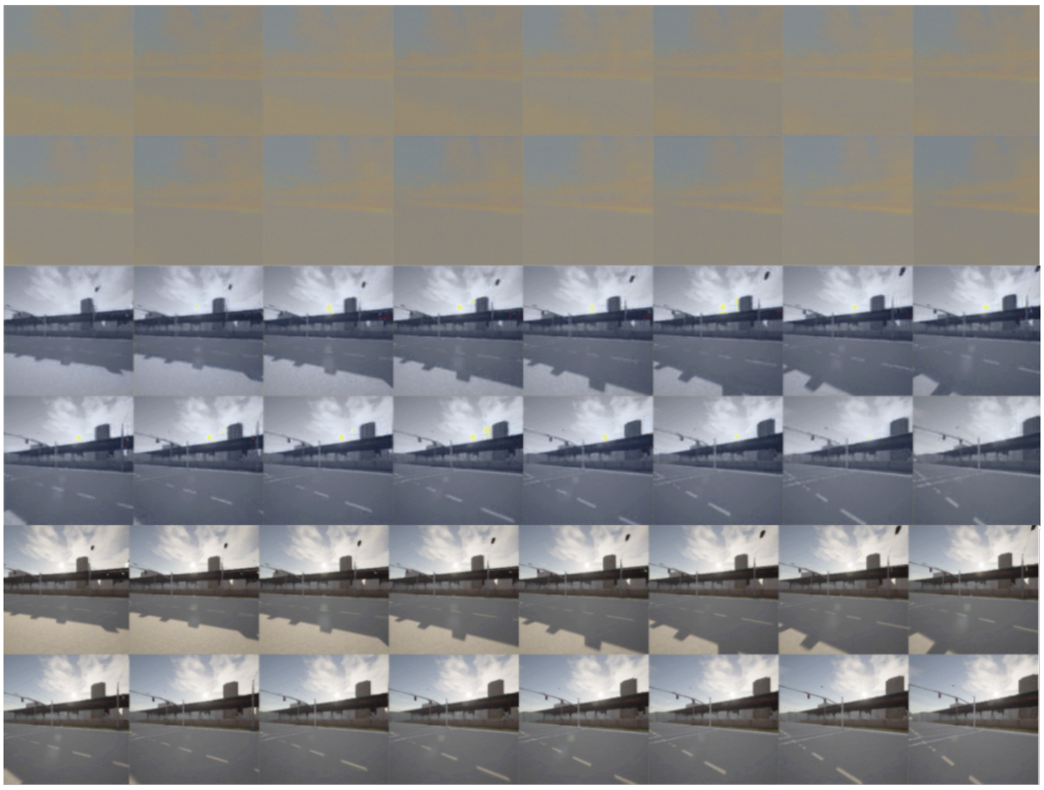

Figure 6.1: **Top:** Reconstruction given only top encoding. **Middle:** Reconstruction given only bottom encoding. **Bottom:** Original video.

The decoder reconstructs the videos from the 80x smaller than the latent representation with little distortion, but not perfectly. It is easy to tell the originals from the reconstructions. The most obvious difference is the seemingly random noise introduced in parts of the reconstruction. Notice that the noise does not appear on the road or the markings but on the background or less common objects. Looking at the two first examples, minor noise is found the in trees, the roof-like structure, the background building and the incoming car. The third example has the most noise, found in the roof of a tunnel.

By zeroing one of the encodings before decoding, the encodings' role in the reconstruction can be seen. The top encoding produces some vague shapes and colouring, as seen in figure 6.1.

The bottom encoding is much more detailed, looking like the original video, without the colors. It is also in the bottom encoding most of the noise is introduced. This makes sense, as the bottom encoding is bigger in dimension and has the hardest task.

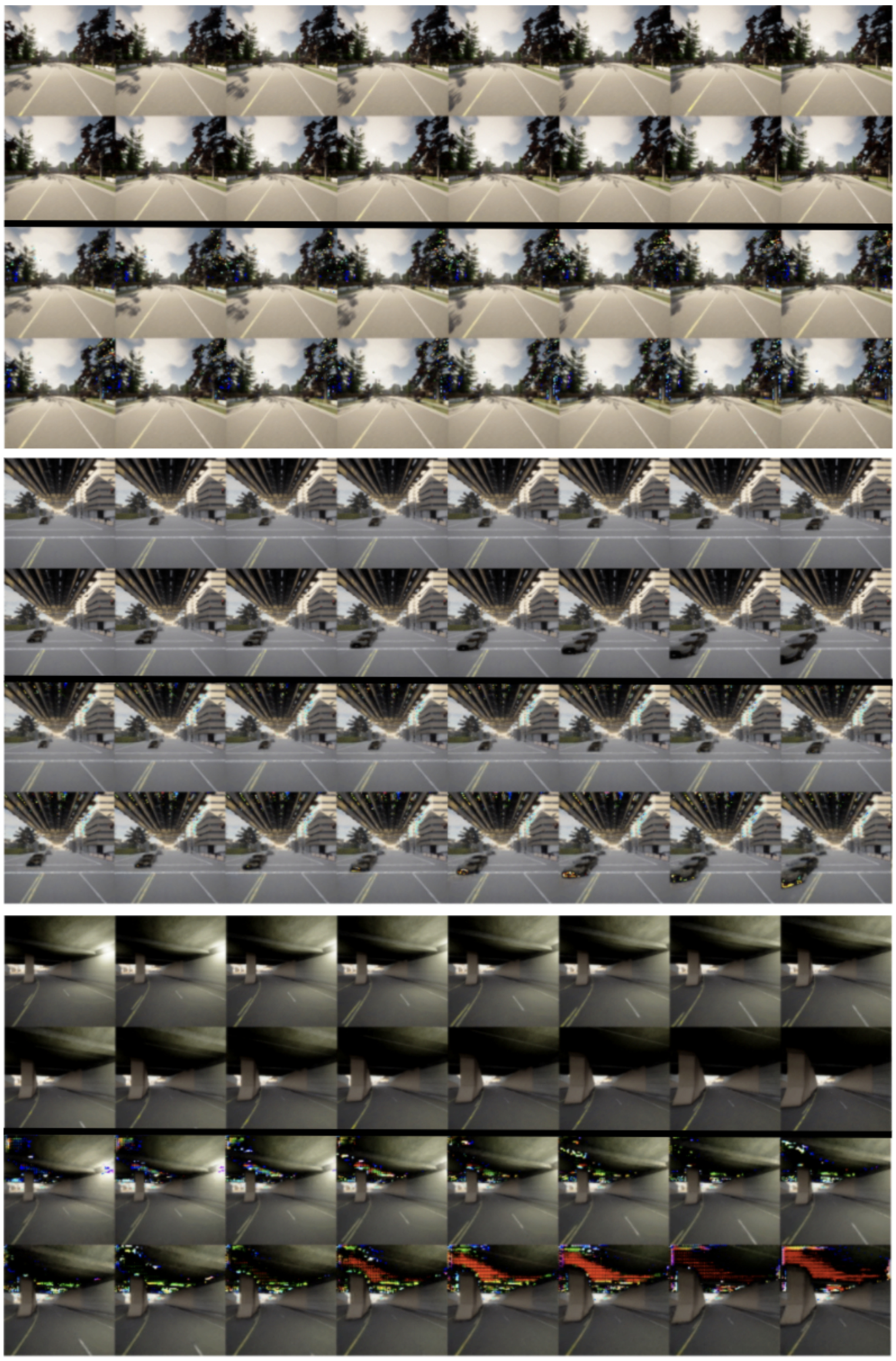

Figure 6.2: Three reconstruction examples, where original the original video is on top.

6.2 Top PixelSNAIL

6.2.1 Quantitative Results

The negative log likelihood (NLL) loss of the training, validation, and test set are reported in Table 6.2. The NLL tells us how well the model can predict the next codebook vector, given all the previous ones. However, the NLL loss does not directly report how well the model can predict many steps ahead, which is the actual objective.

Table 6.2: The mean loss over the different datasets for the Top PixelSNAIL. The loss for a simple sample is the mean of the $4 \times 32 \times 32$ 512-way classification tasks.

	Training	Validation	Test
Mean NLL	1.591	1.764	1.905

This can be done by decoding the original encoding and the generated encoding and using a video quality measure to quantify the differences. However, a single number does not give an intuitive sense of how well the generated encodings are, so this is left out, and the focus is set on looking at decoded generated encodings.

6.2.2 Qualitative Results

Samples from the test set are used as input, and the generated top encodings are decoded alone and together with the matching bottom encoding. Three examples of mixed quality are given.

In Figure 6.3, we can see the generated frames behave nicely and move a little bit frame by frame. When decoded with the bottom encoding, only small differences can be seen. There are some additional noisy spots and some noisy spots are bigger in the generated frames. Other than that, there are no clear visual differences.

Other samples struggle more. The example given in Figure 6.4 shows that the last four generated frames become more and more faded, especially in the top part. This produces only subtle differences when decoded with the bottom encoding. However, you can clearly see that the top part in the generated frames has a warmer color than the original decoding. Furthermore, some additional noise can be seen.

In the last example in Figure 6.5, all the frames look similar. However, when played as a video, the first eight frames move forwards, while the generated frames quickly become frozen. Even though the generated frames fail to capture the motion in the video, the video generated with the bottom encoding looks fine, except for some additional noise.

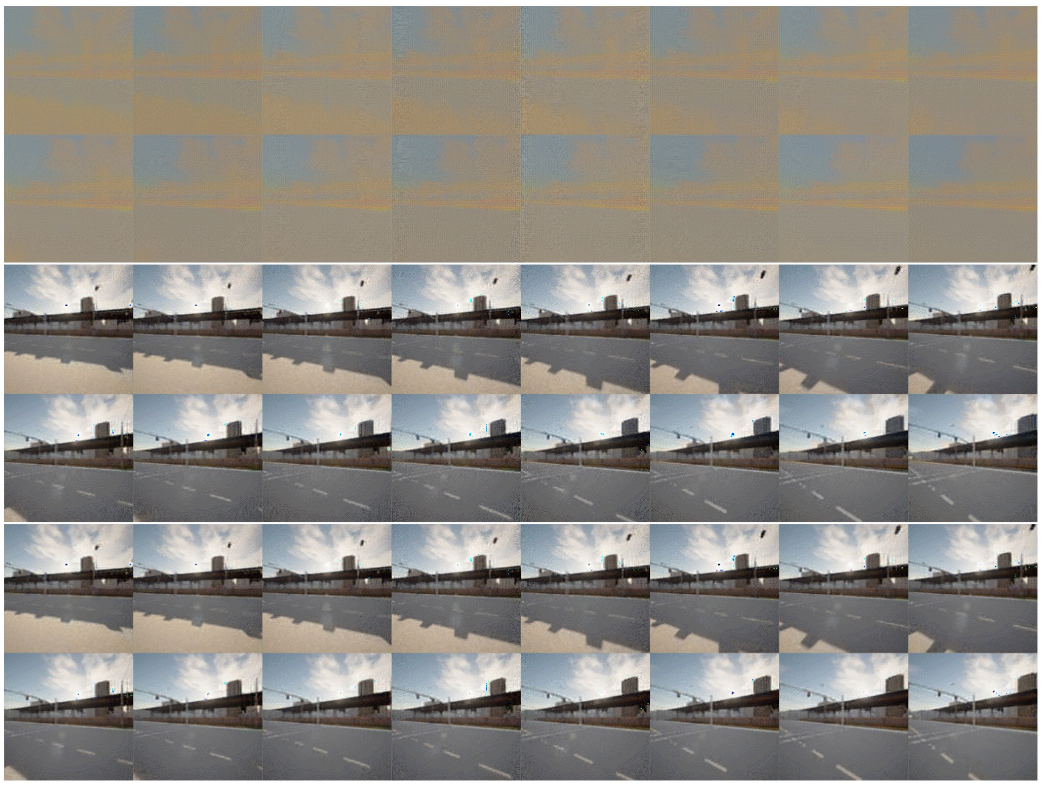

Figure 6.3: **Top:** Decoded top encoding, with eight last frames generated. **Middle:** Generated top encoding decoded with its bottom encoding. **Bottom:** Original top encoding decoded with its bottom encoding.

The generated top encodings are of mixed quality. Some generated encodings predict the movement nicely, while others fail to do so. When failing, the generated frames can be blurred out, or the same frame can be predicted repeatedly. Generally, using the generated encoding leads to more noise and some slight miscoloring. The miscoloring is expected, as the top encoding is responsible for coloring the video. However, there is no distinct noise to be seen in the decoded top encodings, yet it results in more distinct noise in the final output.

When the generated encoding is decoded with the original bottom encoding, there are no big visual differences between using the generated top encoding and the original top encoding. Even when the generated frames are freezing, the decodings look fine.

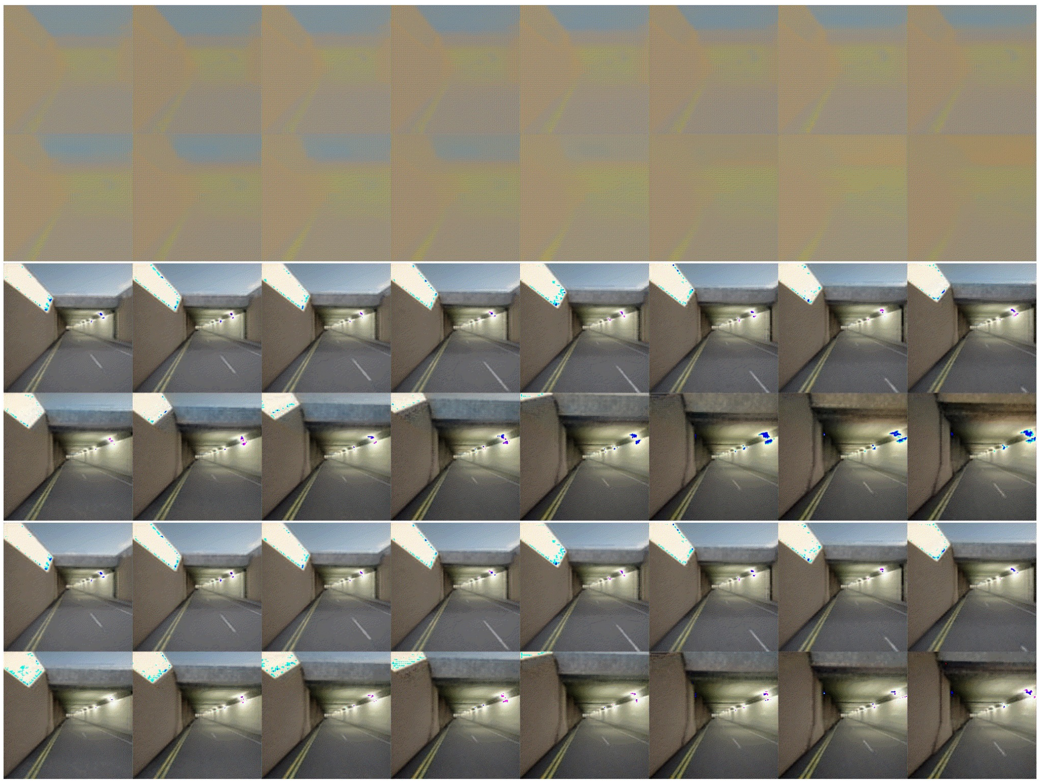
Figure 6.4: Example with differences in noise and coloring.

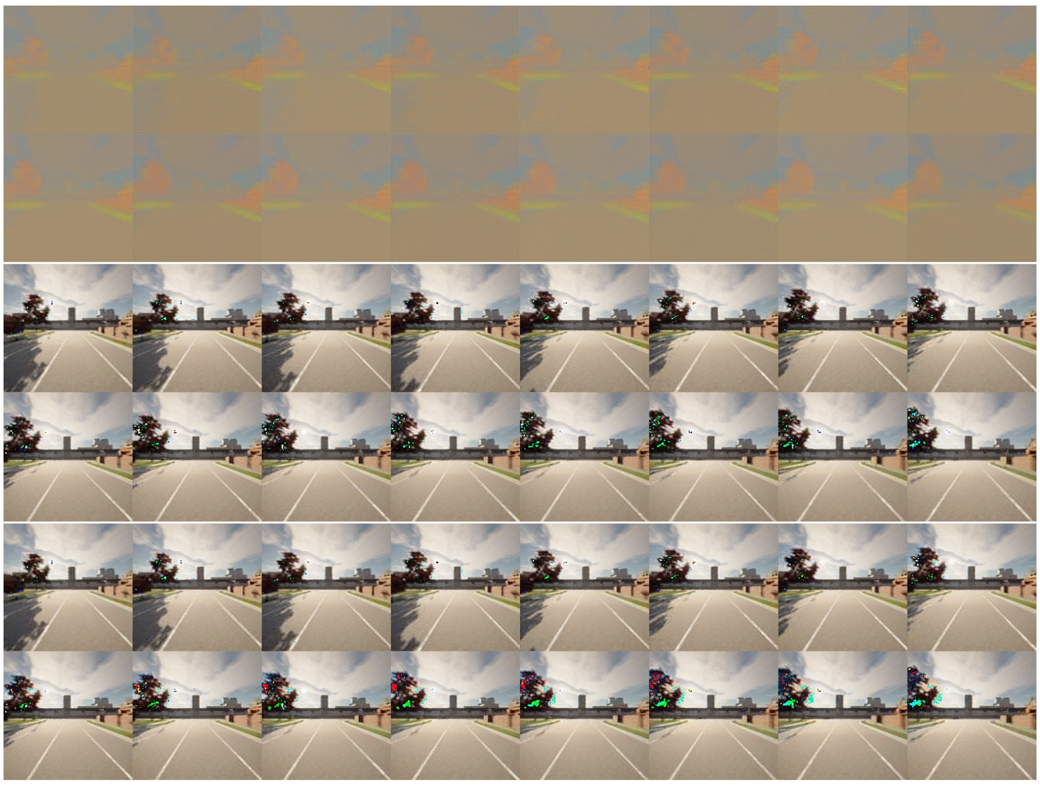
Figure 6.5: Example where generated frames are mostly static.

6.3 Bottom PixelSNAIL

6.3.1 Quantitative Results

The loss of the training, validation, and test set are reported in Table 6.3. As with the top PixelSNAIL, we will focus on looking at decoded generated encodings to visually evaluate the results.

Table 6.3: The mean loss over the different datasets for the Bottom PixelSNAIL.

	Training	Validation	Test
Mean NLL	1.864	1.933	1.987

6.3.2 Qualitative Results

The samples shown from the top PixelSNAIL are also shown here.

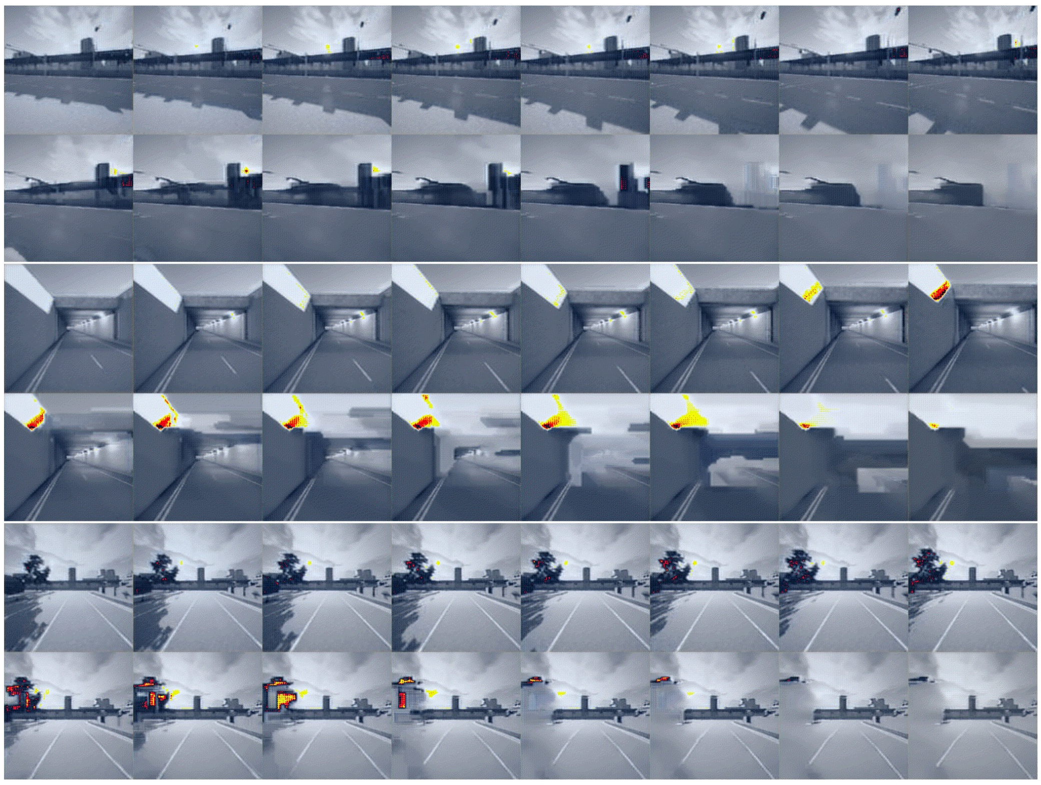

Figure 6.6: Three exaples of generated bottom encodings, conditioned on generated top encodings.

Looking at Figure 6.6, we can see that in all three examples, the bottom PixelSNAIL fails to generate coherent frames. As the PixelSNAIL is autoregressive, an error will propagate through all subsequent pixels and frames. A spot gets blurry in the generated frames and spreads to nearby pixels in the subsequent frames. Notice that it mainly spreads to the right and downwards due to the bottom PixelSNAIL's sole reliance on causal convolutions.

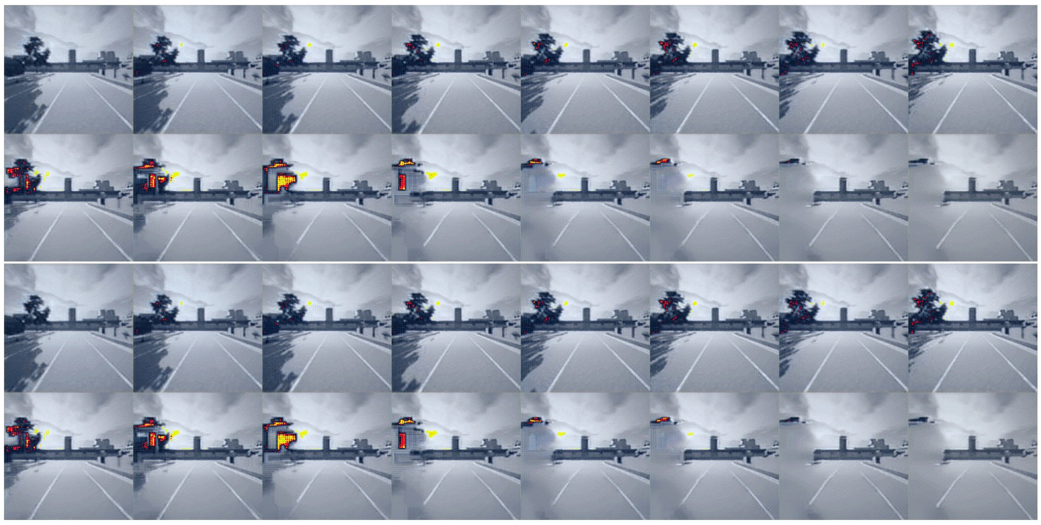

Figure 6.7: **Top:** Generated bottom encoding, conditioned on generated top encoding. **Bottom:** Generated bottom encoding, conditioned on its matching top encoding.

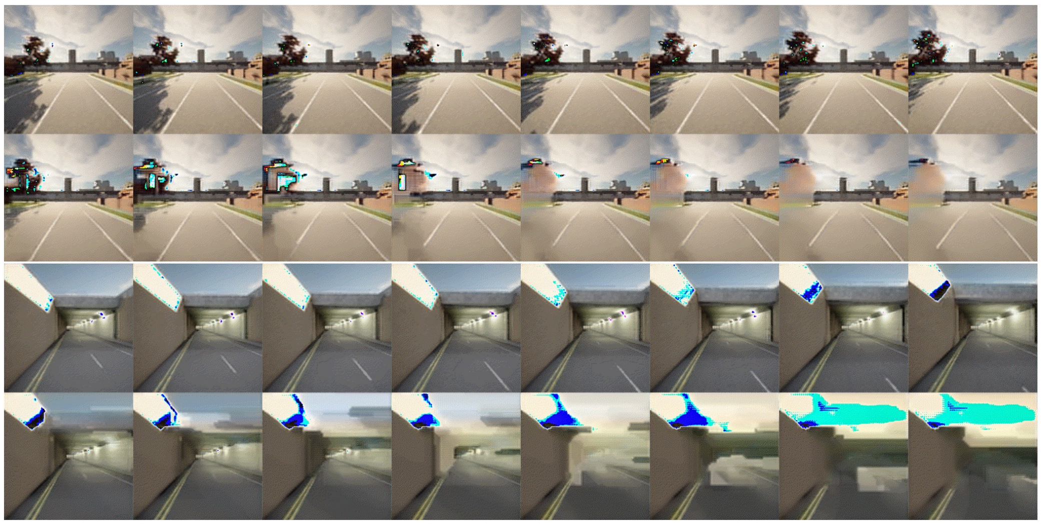

Figure 6.8: Two examples generated by a hierarchical PixelSNAIL.

Looking at Figure 6.7, there is no visible difference between conditioning the bottom PixelSNAIL on a generated top encoding or the matching top encoding.

Given the results from the bottom PixelSNAIL, the hierarchical PixelSNAIL made by the top and bottom PixelSNAILs also fails to generate several coherent frames. Figure 6.8 shows frames predicted by the hierarchical PixelSNAIL.

6.4 Discussion

RQ1: Can a compressed quantized latent space of driving scenarios be learned and used for modelling?

To use a learned quantized latent space to model driving scenarios, the latent space needs to be able to capture all relevant parts of the driving scenario. Even if the latent space introduces some noise when decoded, it is mostly in parts of the video that is not important for driving. For example, the roof of tunnels, trees, etc. may be distorted by noise. While the road the car is driving on is not distorted by noise, some other important objects may be distorted. For example, an incoming car is very important, but in the reconstructions, they are slightly distorted, but not to the extent that a human would not be able to understand what is going on.

It is promising that the VQVAE performs better when reconstructing relevant parts, but this feature was not designed explicitly. The model does not know it is reconstructing driving scenarios, and the loss function punishes distortion everywhere in the video equally. It is likely that this is a result of the dataset. Roads and markings are included in every single training sample, so this is naturally reconstructed well. Things that are less commonly found in the training set are not reconstructed that well. The latent space can only capture what is seen in the dataset well. Therefore, modelling driving scenarios in the latent space is limited to the scenarios similar to the ones in the dataset.

Another important factor is that the latent space is generalized well. With an overfit latent space, it would be difficult to use it for modelling, as a slight devation from samples in the training dataset can cause failure. The validation and test set losses suggest that the VQVAE is not overfit. However, when using the bottom and top PixelSNAILs to model the latent space, the results are not promising.

The top PixelSNAIL produces mixed results, and in some cases, it was able to model the top latent space well for several frames. The bottom PixelSNAIL collapses after very few frames, but is at least able to model the first few frames.

The VQVAE is able to reconstruct the driving scenarios well enough for the PixelSNAILs to model at least one frame. This suggests that the latent space can be used to model driving scenarios, but the errors introduced by the PixelSNAILs causes the PixelSNAILs to be unable to model many frames.

In short, the answer to RQ1 is that a compressed quantized latent space can be learned and used to model driving scenarios that are constrained to scenarios that are similar to the ones in the dataset. As the answer to RQ1 is positive, we can move on to RQ2.

RQ2: To what extent can we model a driving simulator using that latent space?

As the bottom PixelSNAIL was not able to model many frames ahead, we are not able to model a driving simulator well using the latent space. In many cases, we are only able to model one frame of sufficient quality before the subsequent frames fail. The poor results can be explained for several reasons.

The results were better when only looking at the top PixelSNAIL, but the top encodings are much simpler than the bottom encodings. The bottom encoding has more ”responsibility” and contains more complex information. The smaller top encoding is responsible for coloring and perhaps some structuring, which is not as demanding. This is one of the reasons the bottom PixelSNAIL struggles much more than the top PixelSNAIL. The bottom encodings may be too complex for the bottom PixelSNAIL to model well.

Another problem is that the bottom PixelSNAIL can not use the attention mechanism due to the bigger dimensionality of the bottom encoding. As a result, the bottom PixelSNAIL has to rely on causal convolutions that only works well in a local context. Therefore, for the bottom PixelSNAIL to work, it needs to add local details to the top encoding that it is conditioned on.

All three models, the VQVAE, the top PixelSNAIL, and the bottom PixelSNAIL has to work well together to model the driving simulator. However, the bottom PixelSNAIL does not work well with the top PixelSNAIL. In Figure 6.7 there are no visible differences between the bottom encoding conditioned on a flawed top encoding and the bottom encoding conditioned on the matching top encoding. Given the deep residual conditioning stack, the bottom PixelSNAIL has the opportunity to take the top encoding more into use. This means that the bottom PixelSNAIL learns not to take the top encoding into use, as it does not provide useful information for the bottom PixelSNAIL to use.

This problem can also stem from the fact that the bottom encoding is much more complex than the top encoding. Even though the VQVAE does a good job of encoding and decoding the driving scenarios, the bottom encoding may have too much responsibility, reducing the usefulness of the top encoding. Even though the top encoding is of lower dimension, it could be more useful.

6.5 Further Work

Further work can be done to model the driving scenarios more successfully.

As discussed, the top encoding may be too simple compared to the bottom encoding. This problem can be alleviated by forcing more responsibility on the top encoding when training the VQVAE. For example, dropout can be heavily applied to the bottom encoding during training to avoid making the reconstructions overly dependent on the bottom encoding.

The three different models are dependent on each other, but they are trained independently. The models may work more coherently if trained in an end-to-end fashion. This may improve the results as the models cooperate better. However, this requires more powerful hardware.

If further work is able to model the driving scenarios successfully, a natural continuation is to gather more diverse data from the driving simulator. This will make the latent space, and therefore the autoregressive generative models, able to handle more diverse scenarios. Different maps, weather conditions, etc. can be used, or even real-world driving data.

Chapter 7

Conclusion

The goal of this work was to model future frames in driving scenarios. This was attempted by learning a compressed latent space and autoregressively modelling the latent space. A 3D hierarchical VQVAE, 3D PixelSNAIL with attention, and a 3D PixelSNAIL without attention has been implemented and trained.

The Hierarchical VQVAE framework shows promising results for modeling a driving simulator. The 3D hierarchical VQVAE is able to efficiently compress and decompress the original video. By learning a two-level hierarchical discrete latent space, we are able to compress video of $3 \times 16 \times 256 \times 256 \times 8$ bits to two latent codes of $4 \times 32 \times 32 \times \log(512) + 8 \times 64 \times 64 \times \log(512)$ bits. This is a reduction of 98.8% bits required, and only some distortion can be seen in the reconstructions, mostly in elements not seen much in the training set.

The PixelSNAILs fail to model the learned latent space well. The top PixelSNAIL is in some cases able to model the latent space well. The bottom encoding has a much bigger effect on the final result, but the bottom PixelSNAIL is often only able to model one frame of sufficient quality. Consequently, the final result also often fails after one frame.

To be able to model several frames, further research has to be done. It is suggested to investigate making the three models cooperate better by forcing the top latent code to be more useful. This can be done by applying dropout to the bottom encoding, or training the models end-to-end.

Bibliography

[1] Anonymous. "Patches Are All You Need?" In: *Submitted to The Tenth International Conference on Learning Representations*. under review. 2022. URL: https://openreview.net/forum?id=TVHS5Y4dNvM.

[2] Mohammad Babaeizadeh et al. *Stochastic Variational Video Prediction*. 2017. DOI: 10.48550/ARXIV.1710.11252. URL: https://arxiv.org/abs/1710.11252.

[3] Adrià Puigdomènech Badia et al. *Agent57: Outperforming the Atari Human Benchmark*. 2020. arXiv: 2003.13350 [cs.LG].

[4] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[5] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[6] Sam Bond-Taylor et al. "Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. DOI: 10.1109/tpami.2021.3116668. URL: https://doi.org/10.1109%2Ftpami.2021.3116668.

[7] XI Chen et al. "PixelSNAIL: An Improved Autoregressive Generative Model". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 864–872. URL: https://proceedings.mlr.press/v80/chen18h.html.

[8] Xi Chen et al. *Variational Lossy Autoencoder*. 2017. arXiv: 1611.02731 [cs.LG].

[9] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL].

[10] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[11] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2016. arXiv: `1511.07289 [cs.LG]`.

[12] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[13] Emily Denton and Rob Fergus. *Stochastic Video Generation with a Learned Prior*. 2018. DOI: `10.48550/ARXIV.1802.07687`. URL: `https://arxiv.org/abs/1802.07687`.

[14] Terrance DeVries and Graham W. Taylor. *Improved Regularization of Convolutional Neural Networks with Cutout*. 2017. arXiv: `1708.04552 [cs.CV]`.

[15] Jeff Donahue et al. *Long-term Recurrent Convolutional Networks for Visual Recognition and Description*. 2016. arXiv: `1411.4389 [cs.CV]`.

[16] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: `2010.11929 [cs.CV]`.

[17] Alexey Dosovitskiy et al. *CARLA: An Open Urban Driving Simulator*. 2017. arXiv: `1711.03938 [cs.LG]`.

[18] William Fedus, Barret Zoph, and Noam Shazeer. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. 2021. arXiv: `2101.03961 [cs.LG]`.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[20] Alex Graves, Greg Wayne, and Ivo Danihelka. *Neural Turing Machines*. 2014. arXiv: `1410.5401 [cs.NE]`.

[21] Danijar Hafner et al. *Mastering Atari with Discrete World Models*. 2021. arXiv: `2010.02193 [cs.LG]`.

[22] Stephen Hanson and Lorien Pratt. "Comparing biases for minimal network construction with back-propagation". In: *Advances in neural information processing systems* 1 (1988).

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

[24] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs)*. 2020. arXiv: `1606.08415 [cs.LG]`.

[25] Geoff Hinton. *Overview of mini-batch gradient descent*. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[26] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667.

[27] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080.

[28] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[29] Kam-Chuen Jim, C.L. Giles, and B.G. Horne. “An analysis of noise in recurrent neural networks: convergence and generalization”. In: *IEEE Transactions on Neural Networks* 7.6 (1996), pp. 1424–1438. DOI: 10.1109/72.548170.

[30] Lukasz Kaiser et al. *Model-Based Reinforcement Learning for Atari*. 2020. arXiv: 1903.00374 [cs.LG].

[31] Nal Kalchbrenner et al. *Video Pixel Networks*. 2016. DOI: 10.48550/ARXIV.1610.00527. URL: https://arxiv.org/abs/1610.00527.

[32] Andrej Karpathy and Li Fei-Fei. *Deep Visual-Semantic Alignments for Generating Image Descriptions*. 2015. arXiv: 1412.2306 [cs.CV].

[33] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017. arXiv: 1609.04836 [cs.LG].

[34] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML].

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

[36] Ryan Kiros, Ruslan Salakhutdinov, and Richard S. Zemel. *Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models*. 2014. arXiv: 1411.2539 [cs.LG].

[37] Manoj Kumar et al. *VideoFlow: A Conditional Flow-Based Model for Stochastic Video Generation*. 2019. DOI: 10.48550/ARXIV.1903.01434. URL: https://arxiv.org/abs/1903.01434.

[38] Alex X. Lee et al. *Stochastic Adversarial Video Prediction*. 2018. DOI: 10.48550/ARXIV.1804.01523. URL: https://arxiv.org/abs/1804.01523.

[39] Zachary C. Lipton and Jacob Steinhardt. *Troubling Trends in Machine Learning Scholarship*. 2018. arXiv: 1807.03341 [stat.ML].

[40] Ilya Loshchilov and Frank Hutter. *Fixing Weight Decay Regularization in Adam*. 2017. arXiv: 1710.05101 [cs.LG].

[41] Pauline Luc et al. *Transformation-based Adversarial Video Prediction on Large-Scale Data*. 2020. DOI: 10.48550/ARXIV.2003.04035. URL: https://arxiv.org/abs/2003.04035.

[42] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. 2015. arXiv: 1508.04025 [cs.CL].

[43] Junhua Mao et al. *Explain Images with Multimodal Recurrent Neural Networks*. 2014. arXiv: 1410.1090 [cs.CV].

[44] Michael Mathieu, Camille Couprie, and Yann LeCun. *Deep multi-scale video prediction beyond mean square error*. 2015. DOI: 10.48550/ARXIV.1511.05440. URL: https://arxiv.org/abs/1511.05440.

[45] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[46] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[47] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[48] Arvind Neelakantan et al. *Adding Gradient Noise Improves Learning for Very Deep Networks*. 2015. arXiv: 1511.06807 [stat.ML].

[49] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. *Pixel Recurrent Neural Networks*. 2016. arXiv: 1601.06759 [cs.CV].

[50] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2018. arXiv: 1711.00937 [cs.LG].

[51] Aaron van den Oord et al. *Conditional Image Generation with PixelCNN Decoders*. 2016. arXiv: 1606.05328 [cs.CV].

[52] Niki Parmar et al. *Image Transformer*. 2018. arXiv: 1802.05751 [cs.CV].

[53] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[54] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. *Generating Diverse High-Fidelity Images with VQ-VAE-2*. 2019. arXiv: 1906.00446 [cs.LG].

[55] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. *On the Convergence of Adam and Beyond*. 2019. arXiv: 1904.09237 [cs.LG].

[56] Guodong Rong et al. *LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving*. 2020. arXiv: 2005.03778 [cs.RO].

[57] Shibani Santurkar et al. *How Does Batch Normalization Help Optimization?* 2019. arXiv: 1805.11604 [stat.ML].

[58] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[59] Evan Shelhamer, Jonathan Long, and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. 2016. arXiv: 1605.06211 [cs.CV].

[60] Jocelyn Sietsma and Robert J.F. Dow. "Creating artificial neural networks that generalize". In: *Neural Networks* 4.1 (1991), pp. 67–79. ISSN: 0893-6080.

[61] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[62] Lucas Theis, Aäron van den Oord, and Matthias Bethge. "A note on the evaluation of generative models". In: *arXiv preprint arXiv:1511.01844* (2015).

[63] Ilya Tolstikhin et al. *MLP-Mixer: An all-MLP Architecture for Vision*. 2021. arXiv: `2105.01601 [cs.CV]`.

[64] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: `1706.03762 [cs.CL]`.

[65] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354.

[66] Oriol Vinyals et al. *Show and Tell: A Neural Image Caption Generator*. 2015. arXiv: `1411.4555 [cs.CV]`.

[67] Dirk Weissenborn, Oscar Täckström, and Jakob Uszkoreit. *Scaling Autoregressive Video Models*. 2019. DOI: `10.48550/ARXIV.1906.02634`. URL: `https://arxiv.org/abs/1906.02634`.

[68] Kelvin Xu et al. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. 2016. arXiv: `1502.03044 [cs.LG]`.

[69] Jason Yosinski et al. *Understanding Neural Networks Through Deep Visualization*. 2015. arXiv: `1506.06579 [cs.CV]`.

[70] Sangdoo Yun et al. *CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features*. 2019. arXiv: `1905.04899 [cs.CV]`.

[71] Hongyi Zhang et al. *mixup: Beyond Empirical Risk Minimization*. 2018. arXiv: `1710.09412 [cs.LG]`.