Deep Learning 2025
Assignment 4

Simon Høpfner Hvidtfeldt — zgh166

January 18, 2026

1 Hierarchical variational autoencoder

1.1 Description of code

Dynamic binarization We find this class insider the data loader section of the notebook. Every time an image is taken from the data-set then the code converts the MNIST grayscale pixels, which has a range of 0-1 into black or white.

Inside `__getitem__` method, if `dynamic=True`, it uses `torch.bernoulli(x)`. This means that the original grayscale pixel values get treated as probabilities. A pixel value of 0.9 has 90% chance of being 1 and therefor white, or in other words, 10% chance of being black.

Warm-up We risk facing a posterior collapse when training a VAE model. This is where the decoder ignores the latent code z and the encoder just outputs the prior Gaussian noise because the penalty is too strong initially.

To fix this, the notebook multiplies the term in the loss function by a factor that starts at 0 and linearly increases to 10 over the first 50 epochs. This allows the encoder to learn meaningful representations before being forced to match the Gaussian prior.

The three defined networks The first network in the notebook is the class `MarkovTransition`. This represents the latent dynamics or the prior transition. This network defines how the latent variables evolve from layer to layer, independent of the specific image. It models the structure of the latent space itself

The next is the `Decoder` class. This is the generative network. It takes the final latent variable z and tries to reconstruct the original image pixels.

Lastly we have the `MarkovEncoder` class. This is the inference network. It looks at the input data x and tries to predict the distribution of the latent variables z. It outputs the mean and standard deviation that we use to sample z.

1.2 Latent layers and their influence

When trying to improve performance by increasing the number of latent layers, we constructed a loop, and ran trough a list of T values from 1-5. We used the same hyper-

parameters as the original notebook i.e. a learning-rate of $4 \cdot 10^{-4}$, 10 epochs and 5 warm-up epochs. The result is represented in figure 1 below.

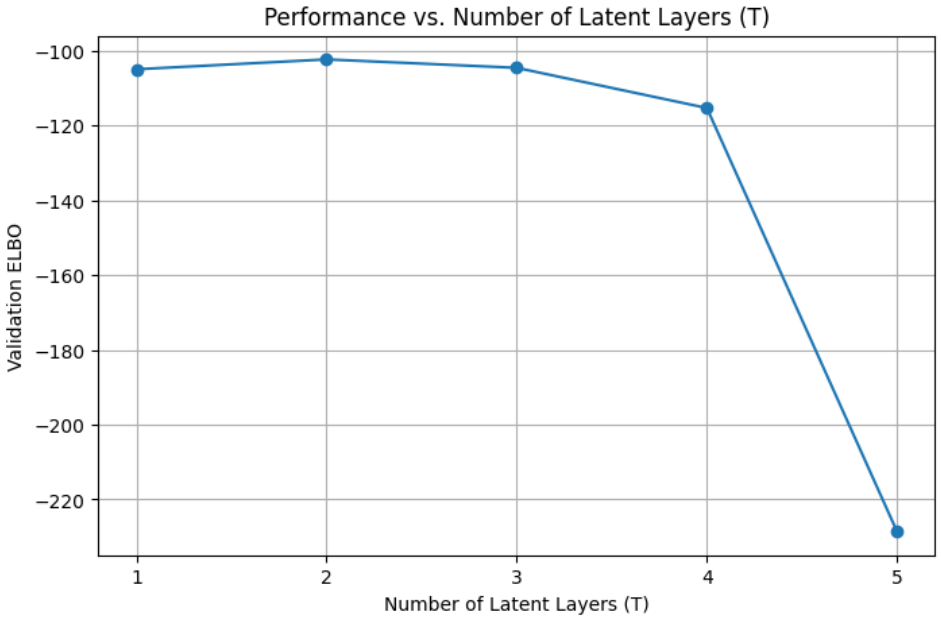


Figure 1: Performance vs. Number of Latent Layers (T)

Increasing the number of latent layers resulted in an initial performance boost, which peaked at $T = 2$. This achieved the best validation ELBO of -102.3, which outperformed the standard VAE $T = 1$, which achieved a score of -104.9. The model with $T = 3$ also performed decently with a score of -104.5, which is slightly better than the baseline $T = 1$, but still worse than $T = 2$.

However, adding layers beyond $T = 3$ caused a significant fall in performance. $T = 4$ dropped to -115.3. The deepest model $T = 5$ was by far the worst. This model failed. Already after the first epoch, the training ELBO exploded too a value of -3934847540813600256.0. This numerical instability lead to a `RuntimeError` during the sampling phase, as the model outputs became `NaN`'s making it impossible to sample from the Bernoulli distribution.

1.3 Warm-up, total epochs and validation ELBO achieved

Bases on task 1.2, we selected $T = 2$ as the best model and trained it for longer to see if the performance could be improved. We sat the model up with 2 latent layers, 100 total epochs, 10 warm-up epochs and the same learning rate of $4 \cdot 10^{-4}$. Resulting in a total runtime of 9 minutes and 15 seconds. See figure 2.

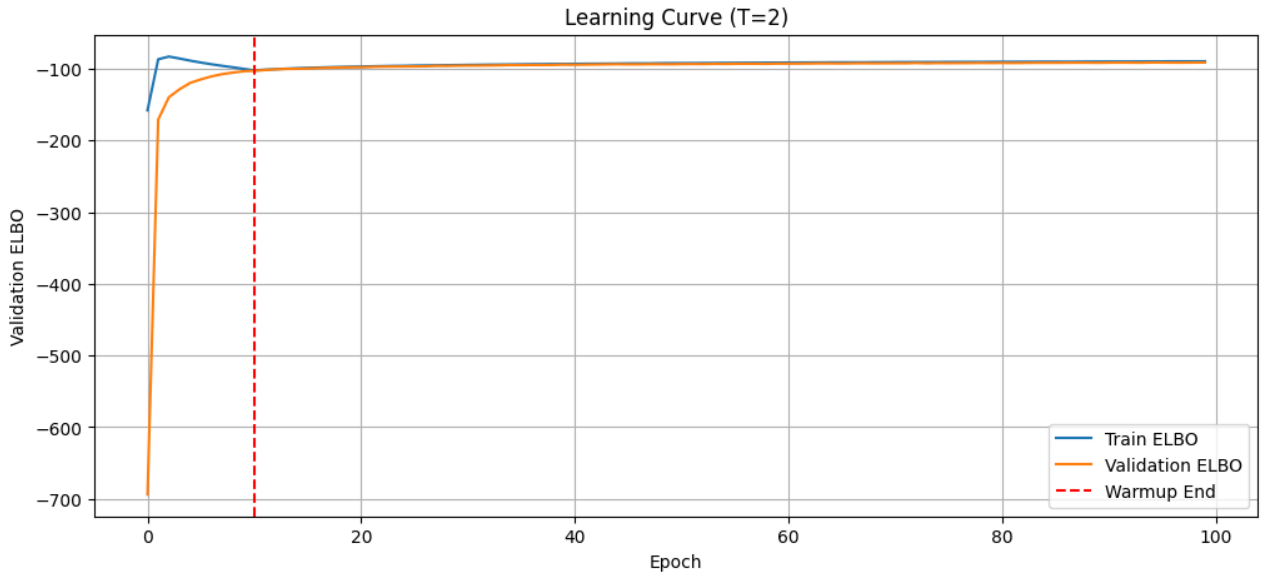
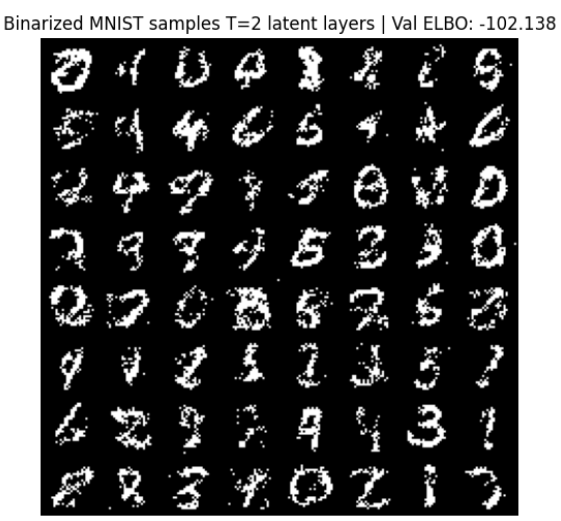


Figure 2: Learning curve for $T = 2$

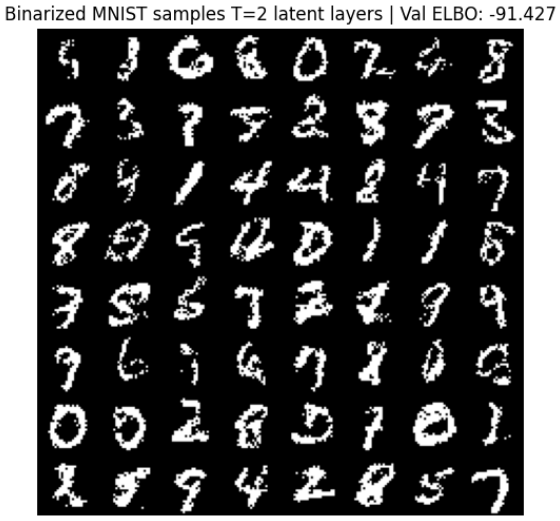
Running the model for 100 epochs gave a huge improvement in performance. The validation ElBO went from -102.3 down to -91.4. This shows that the model needs more time to converge than just the initial 10 epochs.

We also noticed that the training can be unstable because of the high learning rate of $4 \cdot 10^{-4}$. In one of our attempts, the model actually got worse and ended at an ELBO of -145. However, by removing the `torch.manual_seed(69)` that I had set up, we where consistently able to get a low validation ELBO.

Additionally, from figure 3 we can conclude that the generated images look much better compared to the model trained only for 10 epochs. The digits are clearer, and the background is completely black with almost no noise. The digits also look more correct. For example, numbers like ”8”, ”6” and ”0” have clear open holes instead of being closed blobs.



(a)
`lr = 4e-4,`
`n_epochs=10,`
`warmup_epochs=5,`
`T=2`



(b)
`lr = 4e-4,`
`n_epochs=100,`
`warmup_epochs=10,`
`T=2`

Figure 3: Comparison of the two images

1.4 IWAE bound

Since the ELBO is only a lower bound, it underestimates how well the model actually performs. To get a more precise measure of the model's performance, we calculated the IWAE bound for our best model, with `lr = 4e-4`, `n_epochs=100`, `warmup_epochs=10`, `T=2`.

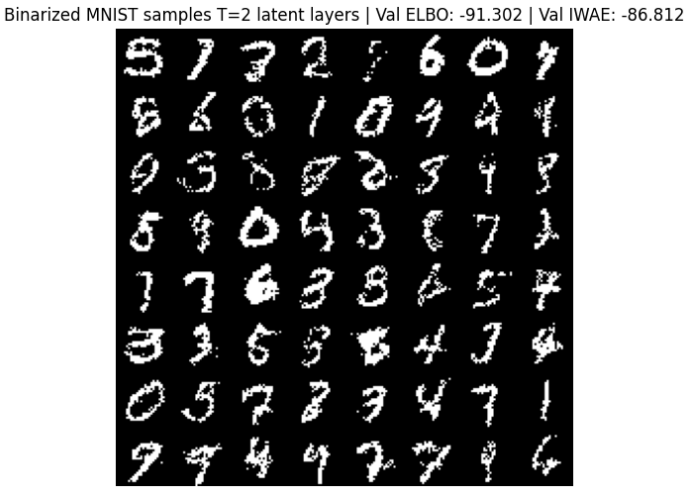


Figure 4: IWAE bound for our best model.

In figure 4, we see that the IWAE score is noticeably better than the ELBO score. The difference is about 4.5 points.

1.5 Parameterization of the hierarchical VAE

Yes, there is a change in performance. We used the same hyperparameters as previously. Based on testing, we can conclude that the SDE model performs slightly worse than the standard model. See table 1.

Model Configuration	**Latent layers**	**Validation ELBO**	**Validation IWAE**
`hierarchical_VAE`	$T = 2$	-91.302	-86.812
`SDE_hierarchical_VAE`	$T = 2$	-94.083	-88.724

Table 1: Performance comparison between `hierarchical_VAE` and `SDE_hierarchical_VAE`

This happens because the SDE parameterization is "less general". The standard model is free to learn any function. The SDE model is forced to follow the specific structure of the differential equation. Since we only use $T = 2$, this restriction makes it harder for the SDE model to fit the data compared to the free standard model.

2 ODE

2.1 Background on the methodology

The generative model equations are listed in the first markdown-cell in the notebook. Here it states that, the notebook implements generative modeling with an ordinary differential equation (ODE) for a latent variable $x_t, t \in [0,1]$ using the following three steps:

$$\begin{aligned} x_0 &\sim p(x_0) \\ dx &= f(x,t)\,dt, \qquad\qquad t \in [0,1] \\ y &= x_1 \end{aligned}$$

Additionally, from the DL_diffusion lecture slides, we have

The Variational Lower Bound (ELBO ODE):

$$\log p(\mathbf{x}) \geq \mathcal{L}_{ODE}(\mathbf{x}) := \mathbb{E}_{q_1(\mathbf{z}_1|\mathbf{x})}\left[\log \frac{p(\mathbf{x}|\mathbf{z}_1)p_1(\mathbf{z}_1)}{\mathbf{q}_1(\mathbf{z}_1|\mathbf{x})}\right]$$

The Liouville Equation (Differential Form):

$$\frac{d}{dt}\log p_t(\mathbf{z}_t) = -\nabla \cdot \mathbf{f}(\mathbf{z}_t, t)$$

The Marginal Log-Probability (Integral Form)

$$\log p_1(\mathbf{z}_1) = \log p_0(\phi_1^0(\mathbf{z}_1)) - \int_0^1 \nabla \cdot \mathbf{f}(\phi_1^t(\mathbf{z}_1), t)\,dt$$

2.2 Analyze the program

The notebook `ODE.ipynb` can be divided into these distinct parts.

`TrainingSetWithLogLikelihood` **class** This class generates 1D dataset with Gaussian, Laplace, or Mixture, and calculates the true analytical log-likelihood for these distributions. It provides the target data distribution y at $t = 1$, that the model attempts to learn, as well as a baseline to verify if the model is learning the correct density.

`Prior` **class** This defines the simple base distribution (standard Gaussian) including sampling and log-probability functions. This represents $p(x_0)$, which is the initial latent distribution at $t = 0$ which the ODE transforms into the complex data distribution over time.

`DriftNetwork` **class** This is a simple neural network that accepts position x and time t as inputs and outputs a scalar value representing velocity. This parametrizes the function $f(x,t)$ in the ordinary differential equation $\frac{dx}{dt} = f(x,t)$ serving as the learnable component that dictates how probability mass flows.

`ForwardPath` **and** `BackwardPath` **classes** These classes manage the integration of particles trajectories, moving either forward or backward in time. `ForwardPath` is used to generate new samples by transforming noise (x_0) to date (x_1), while `BackwardPath` is used during training to map data (x_1) back to the prior (x_0) to calculate likelihoods.

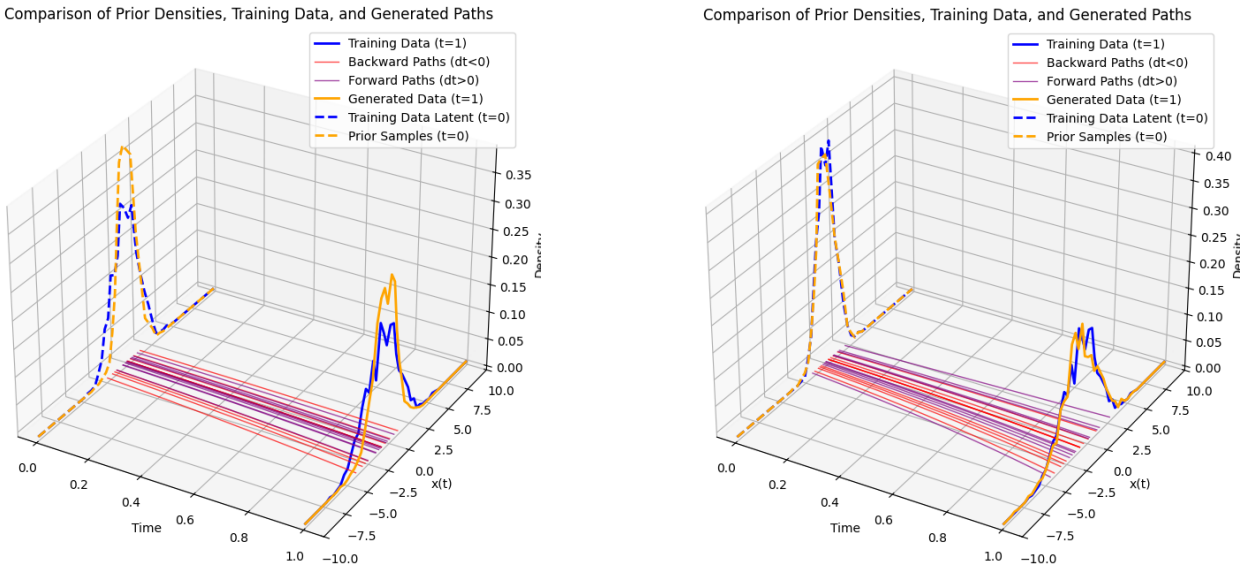
`simulate_ode` **and** `partial_drift_x` `partial_drift_x` calculates the spatial derivative of the network output while `simulate_ode` numerically integrates the combined state of particle position and log-density changes. It

`compute_log_likelihood_ode` **and** `training_loop_ode` These functions calculate the log-likelihood of the training data by integrating the ODE backward from $t = 1$ to $t = 0$ and running the optimization loop. This is the maximum likelihood estimation objective. Adjusting the neural network parameters to maximize the probability assigned to the training data.

`visualize_paths_and_marginals` This functions generates 3D plots showing the trajectories of particles over time and histograms of the density at $t = 0$ and $t = 1$. It provides a qualitative check of the "flow", effectively illustrating if the ODE successfully morphs the simple prior distribution into the shape of the training data.

2.3 Run training with Gaussian training data

The training data is generated from a gaussian distribution, `params = {'mean': torch.tensor(-1.0), 'std': torch.tensor(2.0)`, from this we can conclude that the specific parameters used are $\mu = -1.0$ and $\sigma = 2.0$.



(a) At initialization

(b) After training

Figure 5: of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

Yes, the model can learn the data. There is a significant overlap between the generated distribution (x_1) and the true distribution (y). This confirms that the model has learned to approximate the density correctly. See figure 5.

Based on 'Forward Paths', the light purple lines, we can see that the flow is systematically moving the density from $x \approx 0$ too $x \approx -1$. This shows that the $f(x,t)$ network has successfully learned the correct drift to move the distribution.

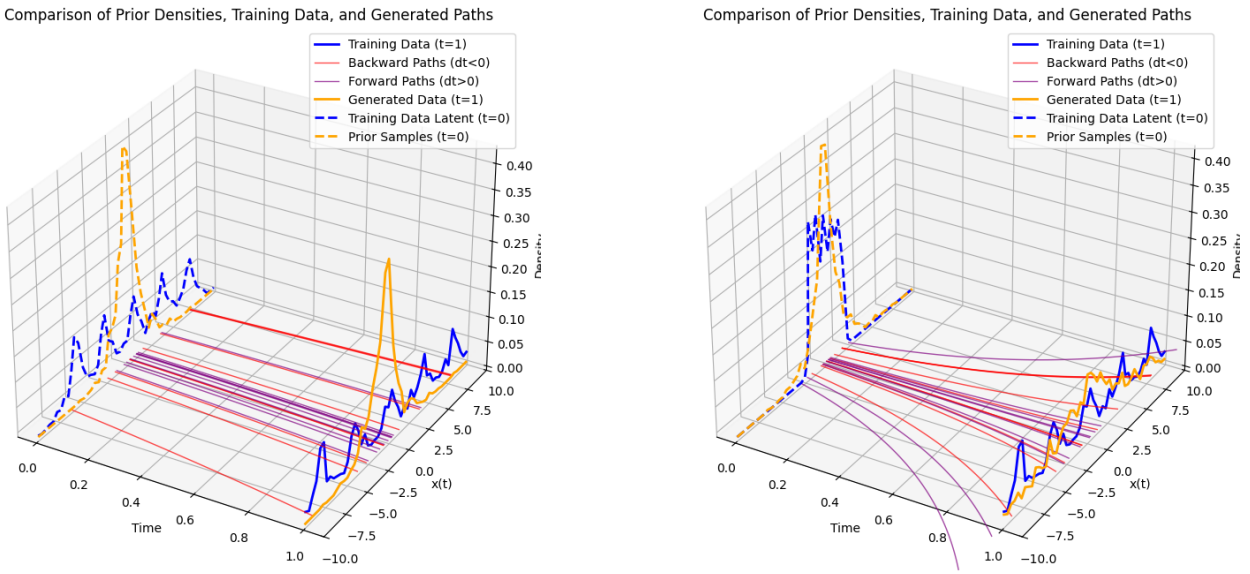
Besides moving the mean value, we also see that the paths spread out. The model has successfully learned to increase the spread from a STD too the target variance of 2.0^2.

The expected log likelihood was calculated by the dataset generator to be -2.103 for the training set. Upon completing the training, 100 epochs, the final log likelihood achieved by the model was -2.10276, rounding this number up we get -2.103. The trained model is extremely close to the ideal value. The difference between the achieved log likelihood and the true expected log likelihood is almost non-existing. This indicates that the model has converged to the optimal solution.

This numerical result matches with the visualizations of the densities. Since the log likelihoods are almost identical, the probability function learned by the model matches the true data distribution. This is confirmed by the overlap between the generated histogram (x_1) and the training data histogram (y) in the final plot.

2.4 Run training with mixture of Laplace training data

The training data is generated from a Laplace mixture distribution, `params = {'k': 5, 'spacing': 4.0, 'scale': torch.tensor(1.0)`. From this, we can conclude that the specific parameters used are $k = 5$ components (peaks), a spacing of 4.0 between the component centers, and a scale parameter $b = 1.0$.

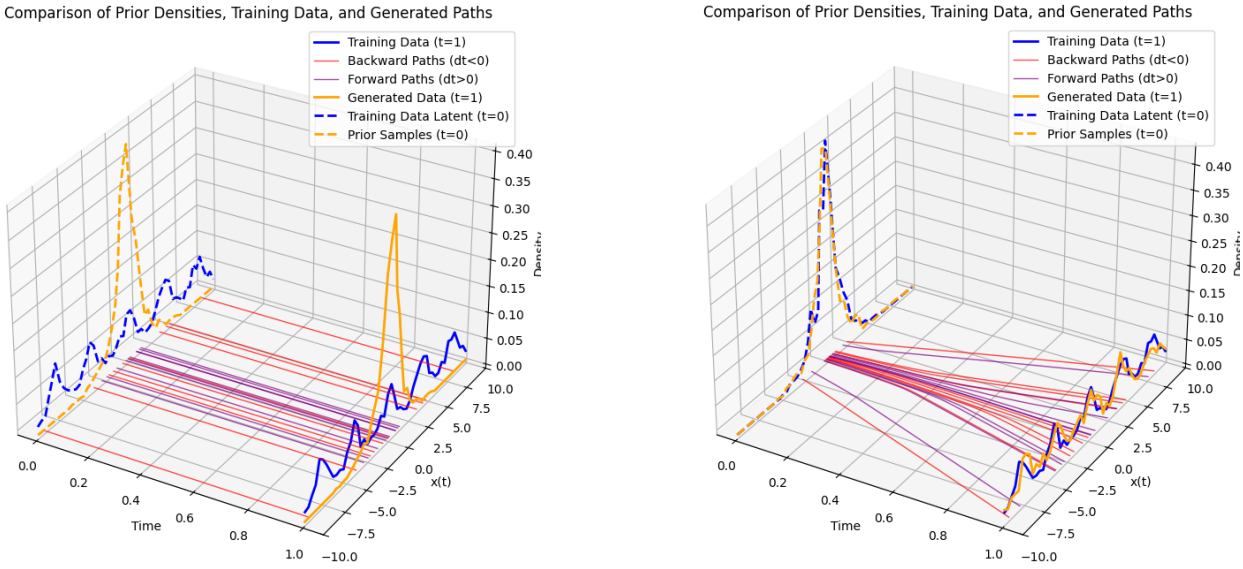


(a) At initialization

(b) After training

Figure 6: Comparison of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

Without changing the `DriftNetwork`, the model isn't really capable of learning the task. We see massive fluctuations in the 'Generated Data' that does not match the 'Training Data' (figure 6). Additionally, the expected log likelihood for the training data was calculated to be -2.975 (rounded), and the log likelihood after training for 100 epochs is calculated to be -3.313. Now, lets change the `DriftNetwork`.



(a) At initialization

(b) After training

Figure 7: Comparison of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

The model is now capable of learning the task, if we make the network capacity sufficient. To achieve this new result the `DriftNetwork` was upgraded from a linear model to one with 3 hidden layer, and one output layer.

```
class DriftNetwork(nn.Module):
    def __init__(self):
        super(DriftNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(2,200),
            nn.Tanh(),
            nn.Linear(200, 200),
            nn.Tanh(),
            nn.Linear(200, 200),
            nn.Tanh(),
            nn.Linear(200, 1)
        )
```

There is a significant overlap between the generated distribution (x_1) and the true distribution (y). This confirms that the deeper model has learned to approximate the complex density correctly.

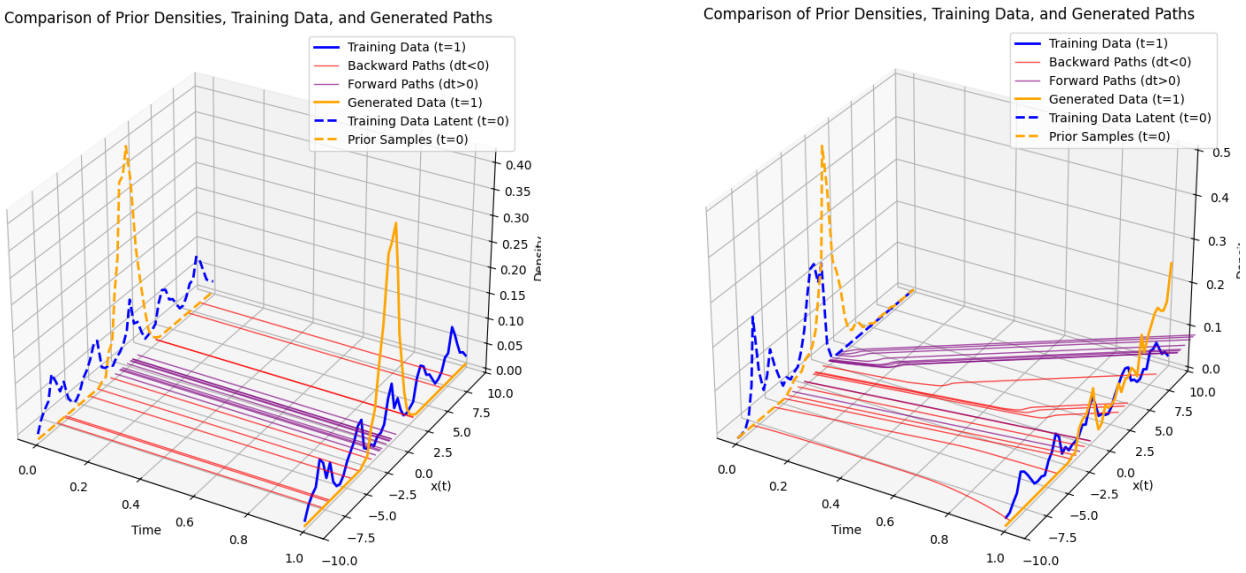
Based on 'Forward Paths', the light purple lines, we can see that the flow is splitting the density. Instead of moving towards a single mean, the paths split into five, moving towards the five distinct modes of the mixture distribution. This shows that the non-linear $f(x,t)$ network has learned the drift necessary to separate the data.

Besides moving the mean, the model successfully captures the shape of the five peaks. The expected log likelihood was calculated by the dataset generator to be -2.961 for the training set. Upon completing the training, 100 epochs, the final log likelihood achieved by the model was -2.95499. If we round this number, we get -2.955. The trained model is extremely close to the best value. This indicates that the model has converged to the optimal solution.

This numerical result matches with the visualizations of the densities. Since the log likelihoods are almost identical to the ground truth, the probability function learned by the model matches the true data distribution. This is visually confirmed by the overlap between the generated histogram (x_1) and the training histogram (y) in the final plot, where all five peaks are defined and aligned.

We will now experiment with various numerical integrator methods. We still use the upgraded version of the `DriftNetwork`.

Euler This is the simplest numerical integrator method. For each of the $T = 100$ integration steps, the drift function is evaluated only once. Using `euler` we get a runtime of 3m 34.6s.



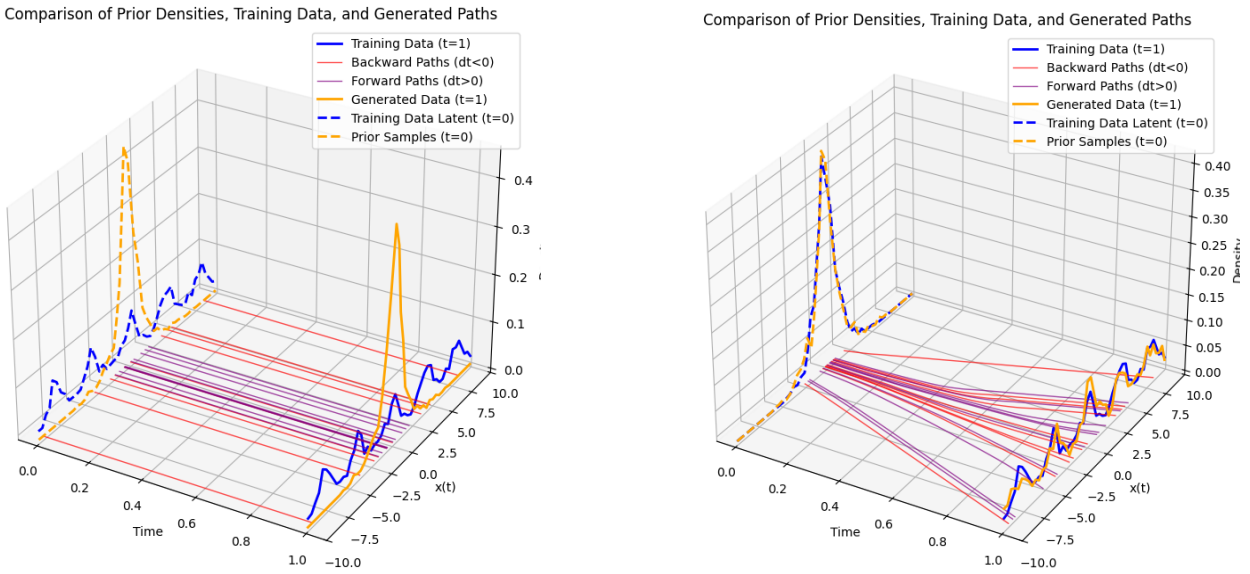
(a) At initialization

(b) After training

Figure 8: Comparison of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

rk2 This is a second-order method that requires two evaluations of the drift function per integration step. One at the start and one at the midpoint. This is the numerical integrator used to create figure 7. Using `rk2` we get a runtime of 6m 5.5s.

rk4 This is the most complex integrator implemented in the notebook (figure 9). It requires four evaluations of the drift function per step to compute a weighted average of slopes. Using `rk4` we get a runtime of 11m 20.6s.



(a) At initialization

(b) After training

Figure 9: Comparison of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

Running this experiment with mixture of Laplacians data, and the complex `DriftNetwork`, we can see that Euler is too simple to capture any meaningful densities. On the other hand, rk2 and rk4 are capable of capturing the densities.

2.5 Complexity

The calculation of the log-likelihood involves a numerical integration loop that iterates through a time grid af size T. For each of the T integration steps, the model must evaluate the function $f(x,t)$ and its partial derivative. While scaling remains linear with respect to T, the constant factor depennds on the chosen integration method. Euler requires one function evaluation per step, rk2 requires two, and rk4 requires four. Therefore doubling T will approximately double the computational time for the forward pass.

In the provided ODE notebook, parameter estimation is performed by applying back-propagation through the ODE solver. Since hte log-likelihood is the result of T sequential integration steps, the backward must go trhough the graph of the solver in reverse. Each step in the integration is a differentiable operation that the gradient must pass through.

3 SDE

3.1 Background on the methodology

The notebook implements generative modeling with an stochastic differential equation (SDE) for a latent variable x_t, $t \in [0,1]$ using the following three steps:

$$\begin{aligned} x_0 &\sim p(x_0) \\ dx &= f(x,t)\,dt + \sigma(x,t)\,dW, \qquad\qquad t \in [0,1] \\ y &\sim p(y|x_1), \end{aligned}$$

where $dW \sim N(0, dt)$ is Gaussian noise with variance dt.

In the lecture slides the log likelihood lower bound objective equation is:

$$L(x) = \mathbb{E}_q[\log p(y|x_1)] - \frac{1}{2}\int_0^1 \mathbb{E}_q\left[\sum_i \frac{(f_i(z_t,t) - g_i(x,z_t,t))^2}{\sigma_i^2(z_t,t)}\right] dt + \mathbb{E}_q\left[\log \frac{p(z_0)}{q(z_0|x)}\right]$$

The first equation follows the SDE notebook notation, where x_t is the latent variable and y is the observed data. In the second equation, the lecture slides uses z_t as the latent variable and x as the data.

3.2 Analyze the program

`DiffusionNetwork` This neural network parametrizes the diffusion coefficient $\sigma(x,t)$, which determines the amount of state dependent Gaussian noise added during the generative process.

`Likelihood` This class implements the probabilistic observation model $p(y|x_1)$, which uses a Gaussian distribution to link the final latent state produced by the SDE to the actual observed data y.

`AlphaNetwork` **and** `BetaNetwork` These networks define the time dependent mean $\alpha(y,t)$ and standard deviation $\beta(y,t)$ of the variational distribution $q_t(x_t|y)$.

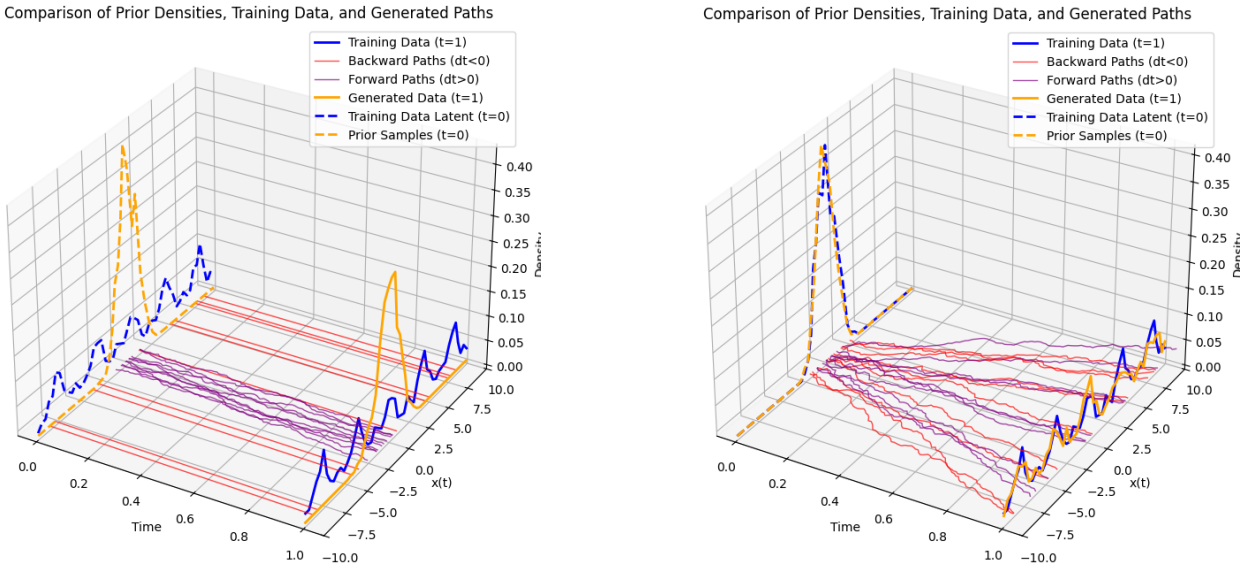
`Variational` This part manages the variational distribution and provides a method to calculate the variational drift $g(x,y,t)$ using automatic differentiation which is essential for the drift-matching term in the ELBO objective.

`Compute_elbo_sde` This function implements the "simulation-free" ELBO by combining the reconstruction term, the KL divergence to the prior, and the weighted squared difference between generative and variational drifts.

`training_loop_sde` This manages the model optimization by sampling random time points $t \in [0,1]$ to estimate the ELBO, allowing for parameter estimation without the computational cost of numerically integrating the SDE.

3.3 Run training

Without changing anything in the program, we get the following plot (figure 10).



(a) At initialization

(b) After training

Figure 10: Comparison of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

The solid blue line represents the true density of the mixture of Laplacians, while the solid orange line represents the model's generated density. In a successful run, these two distributions should overlap, with the orange line capturing the five distinct peaks of the training data, which they do.

The purple and red lines show individual sample path through time. Because the final plot uses the `mode="backward sde"` mod, these paths are not straight lines, they are wavy which reflects the stochastic noise term $\sigma(x,t)dW$ implemented in the generative process.

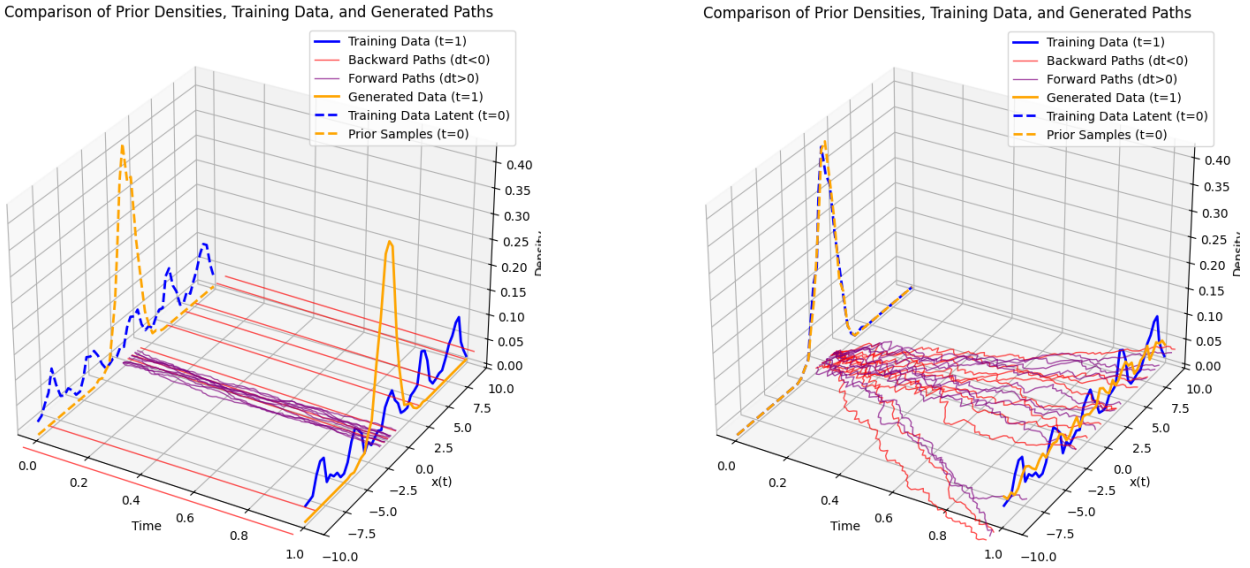
The dashed orange line shows the initial prior samples,, and the dashed blue line shows the training data 'pushed back' to the latent space. The alignment of these two dashed lines indicates that the model has successfully learned a mapping where the complex data distribution at $t = 1$ corresponds to a simple Gaussian prior at $t = 0$.

The visualization in figure 10 confirms the numerical findings. The achieved log-likelihood of -3.038 is very close to the ideal value of -3.016, which indicates that the SDE model has successfully learned to approximate the mixture of five Laplace distributions.

3.4 Change parameterization of the model components

We reduced the `DriftNetwork` capacity from 64 hidden units to just 4 hidden units in a single layer. Additionally, we swapped to `AlphaNetwork0`, which implements a strictly linear mapping between the noise and the data $t \cdot y$, removing any learned non-linearity. Lastly,

we also swapped to `BetaNetwork0`, which provides a unnaturel scaling of the variance over time. All this should in theory result in a worse model.



(a) At initialization

(b) After training

Figure 11: Comparison of Prior Densities, Training Data, and Generated Paths, at initialization and after training.

The model achieved a final validation log-likelihood of -3.3591, which is lower than the expected log-likelihood of -2.9589. This shows that the model failed to accurately capture the target distribution (figure 11).

The generative drift network $f(x,t)$ is responsible for the deterministic 'push' that transforms the prior into the complex data distribution. Because the mixture of Laplacians contains five peaks, the vector field must be highly non-linear. With only 4 neurons, the network lacks complexity to represent this mapping. This can be seen by figure 11. The solid blue line clearly shows the five peaks. In contrast, the solid orange line appears as a wavy line, that fails to connect with any of the five peaks. This is a direct result of reducing the `DriftNetwork`.

3.5 Expected log likelihood

The ELBO values achieved by the SDE model are more or less the same as the log-likelihood values obtained in the ODE case, if both models have a sufficient network. In a succesful run of mixture of Laplacians, both models converge toward the expected log-likelihood of approximately -2.95 to -2.99. In our successful SDE run, the validation log-likelihood of -3.038 was nearly identical to the expectation of -3.016, which matches the high precision seen in the ODE approach.

Visualy the SDE fit is highly accurate when using a standard parameterization. In such cases, the generated density (organge line) captures alle five peaks of the training data, which demonstrates that the SDE can effectively model complex distributions.

However, the SDE 'swings' significantly more than the ODE during training. This is due to two differences. The ODE approach computes the exact log-likelihood by performing a deterministic numerical integration over the entire time grid for every batch. The SDE approach uses a 'simulation free' objective where the time integral in the diffusion term is replaced by an expectation over t. During each training step the model samples random t-values from a uniform distribution U for each example in the mini-batch.

Because the loss is estimated using a stochastic approximation of the integral rather than a full deterministic solution, the resulting gradients have higher variance, which leads to the fluctuations we see in the training curves.

3.6 Complexity

Using ODE requires a full numerical integration over T steps to calculate the log-likelihood. This results in a computational complexity that scales linearly with T. The SDE approach uses a 'simulation-free' ELBO objective where the time integral is estimated by sampling random time points rather than integrating the entire path. This means that the complexity per training step for SDE is independent of the number of integration steps T. This makes it significantly faster and less computationally expensive than the ODE approach.