02460 - MINI PROJECT 1 - GENERATIVE MODELS

Klaus Jupiter Bentzen (s204123), Andreas Krogsgaard Holme (s204160) & Kasper Jørgensen (s204231)

In this project, we aim to train and assess a variety of deep generative models on a binarized and a standard MNIST dataset. We split the data into a training set (n=60,000) and a test set (n=10,000). Our project has two parts. In the first part, we look at the impact of different priors on a Variational Autoencoder (VAE). In the second part, we train a flow based model, and a DDPM. We compare the quality of the samples these models produce to each other and a VAE model.

Architectures We train three architectures: a Variational Autoencoder (VAE) with a 2D latent space, a Normalizing Flow model, and a Denoising Diffusion Probabilistic Model (DDPM). The VAE is a symmetric encoder-decoder network with linear layers (two hidden layers, 512 units each) and ReLU activations. The VAE employs a Gaussian encoder and a Bernoulli decoder for binary data, and a continuous Bernoulli decoder for continuous data. The Flow model has 20 affine coupling layers with alternating checkerboard masking. In the affine coupling layers the scale and translation networks have one hidden layer (256 units) and ReLU activations. The DDPM is based of a U-Net (Ho et al., 2020; Ronneberger et al., 2015) with four down/upscaling layers and a 1000-step discrete denoising process using a linear variance schedule. See the code in the zip file for exact implementations.

Priors for Variational Autoencoders We train VAEs with different priors on binarized MNIST: A standard Gaussian, a Mixture of Gaussians (MoG) with 16 learnable components, a Flow Prior with 8 units and 10 transformation layers, and a VampPrior with 16 learnable pseudo-input components and learnable importance weights. For more accurate test statistics we train and evaluate 40 VAEs, 10 for each prior (epochs: 20, batch size: 128, learning rate: 0.001). We evaluate two key test statistics: the Evidence Lower Bound (ELBO) and the Importance-Weighted Autoencoder (IWAE). These statistics approximate the data's log likelihood. It's worth noting that the IWAE is considered a more accurate estimator. Details of these metrics are presented in Table 1. We wish to compare the aggregate posterior to the prior. The aggregate posterior can be described as the data projected to the latent space. We expect that effective priors will align well with the aggregate posterior. We observe this alignment in all priors, except the Gaussian prior1. The Gaussian prior has more gaps as well as more points in lower density regions compared to the other priors. We expected to see this as the Gaussian prior isn't as flexible as the other priors.

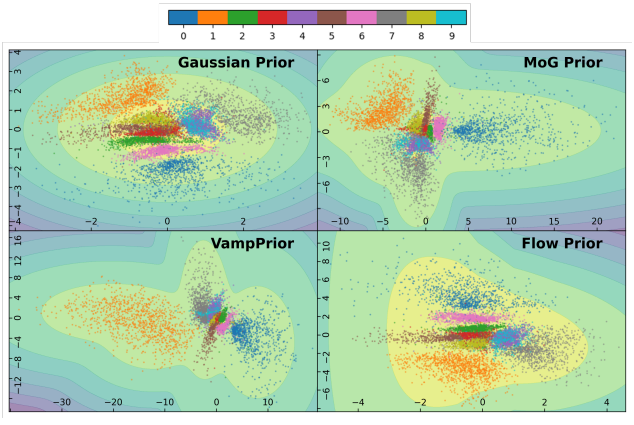

Figure 1: Density plots of VAE priors with test data, color-coded by label, after encoding to latent space. In the plots the density is proportional to the brightness of the region.

We expected that a more flexible prior would improve ELBO/IWAE metrics. However, our analysis showed no significant impact from changing the priors on these metrics. We hypothesize that the simplicity of binarized MNIST leads to the models' indifference towards the priors.

Table 1: Test set metrics for VAEs using different priors

Prior	ELBO↑	IWAE↑
Gaussian Prior	-135.00±0.40	-29.40±2.95
MoG Prior ($k = 16$)	-134.07±0.41	-29.12±2.10
VampPrior ($k = 16$)	**-134.00±0.29**	-28.83±1.76
Flow Prior	-134.88±0.66	**-27.27±2.71**

Sampling Quality We assess the sampling quality of a VAE with a flow prior, a Flow model, and a DDPM. All the models are trained on non-binarized MNIST data. We dequantize the data for training a Flow and DPPM model. In addition, we shift the data to a range of [-1, 1] for the DDPM. The DDPM converges after 100 epochs. This is in contrast to the VAE and Flow which converge after 20 epochs. We evaluate the models quantitatively by measuring the Fréchet Inception Distance (FID) (Heusel et al., 2017) and Kernel Inception Distance (KID) (Bińkowski et al., 2021). The results are presented in Table 2. We evaluate the models qualitatively based on the samples they generate. The samples are in Figure 2. The DDPM yields the clearest samples. The digits in Flow model's outputs are blurry and difficult to discern, but the black background of MNIST images is captured perfectly. We speculate that optimizing scale and translation networks could improve the clarity of the digits. The VAE produces fuzzy images. The visual and data reconstruction sample quality of the models is reflected in the FID and KID. However, we think the digits in the VAE pictures are more readable than Flow pictures.

Table 2: Quality metrics of different generative models.

Model	FID↓*	KID↓*
VAE	4.168	0.189±0.008
Flow	0.452	0.158±0.010
DDPM	**0.018**	**0.012±0.004**

*Calculations based on 256 test vs. 256 generated samples

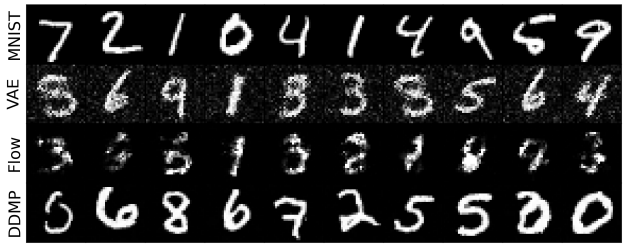

Figure 2: MNIST images generated by each of the three models, with real MNIST images for comparison.

CODE SNIPPETS

The code below is our implementation of the ELBO loss for the VAE. We require a prior to implement a log probability in order to evaluate non standard distributions. For the Flow model, we have implemented this and for the MoG prior the mixture same family automatically derives this. The Kl divergence is estimated using Monte Carlo if no analytical solution is implemented in torch.distributions.

```
def elbo(self, x):
    q = self.encoder(x); z=q.rsample()
    return torch.mean(self.decoder(z).log_prob(x) -
            self.kl_divergence_with_fallback(q, self.prior()), dim=0)
def kl_divergence_with_fallback(self, q, prior, num_samples=256):
        try:
            kl_div = td.kl_divergence(q, prior) # works for e.g. gaussian prior
        except NotImplementedError: # Fallback to Monte Carlo estimation
            samples = q.rsample((num_samples,))
            log_q_x = q.log_prob(samples)
            log_prior_x = prior.log_prob(samples)
            return (log_q_x - log_prior_x).mean(0)
```

The Gaussian Mixture

```
class GaussianMixturePrior(nn.Module):
    def __init__(self, dim, **kwargs):
        self.dim = dim
        self.num_comps = 16
        self.means = nn.Parameter(torch.randn(self.num_comps, dim), requires_grad=True)
        self.log_vars = nn.Parameter(torch.zeros(self.num_comps, dim), requires_grad=True
    ) # Use log variance for numerical stability
        self.logits = nn.Parameter(torch.zeros(self.num_comps), requires_grad=True)
    def forward(self):
        variances = torch.exp(self.log_vars)
        mixture_dist = td.Categorical(logits=self.logits)
        component_dist = td.Independent(td.Normal(self.means, variances.sqrt()), 1)
        return td.MixtureSameFamily(mixture_dist, component_dist)
```

Main parts of Flow prior implementation:

```
class FlowPrior(nn.Module):
    def __init__(self,dim,**kwargs):
        self.masking = kwargs.get('masking', 'checkerboard')
        self.base_dist = GaussianBase(dim)
        self.num_transforms, self.num_hidden = 10, 8
        self.transforms = []
        mask = torch.Tensor([1 if (i) % 2 == 0 else 0 for i in range(dim)])
        <Initialize masking layers>
        scale_net = <SEQ of nn and relu >
        translation_net = <SEQ of nn and relu>
        self.flow = Flow(self.base_dist, self.transforms)
        self.transforms.append(MaskedCouplingLayer(scale_net, translation_net, mask))
    def forward(self):
        return self.flow
```

The negative DDPM ELBO.

```
    def negative_elbo(self, x):
        t = torch.randint(0, self.T, (x.shape[0], 1)).to(x.device)
        eps = torch.randn(x.shape).to(x.device)
        sqrt_alpha_bar_t = torch.sqrt(self.alpha_cumprod[t])
        sqrt_one_minus_alpha_bar_t = torch.sqrt(1-self.alpha_cumprod[t])
        x_t = sqrt_alpha_bar_t*x+sqrt_one_minus_alpha_bar_t*eps
        eps_pred = self.network(x_t, t/self.T)
        return torch.norm(eps - eps_pred,p=2,dim=1)**2
```

DDPM Sampling

```
    def sample(self, shape):
        # Sample x_t for t=T (i.e., Gaussian noise)
        x_t = torch.randn(shape).to(self.alpha.device)
        for t in range(self.T-1, -1, -1):
            z = torch.randn(shape).to(self.alpha.device) if t > 1 else 0
            C1 = 1 / torch.sqrt(self.alpha[t])
            C2 = (1 - self.alpha[t])/torch.sqrt(1-self.alpha_cumprod[t])
            t_ = t*torch.ones(shape[0], 1).to(self.alpha.device)
            x_t = C1 *(x_t - C2 * self.network(x_t, t_/self.T)) + z*torch.sqrt(self.beta[
    t])
        return x_t
```

REFERENCES

Mikołaj Bińkowski, Danica J. Sutherland, Michael Arbel, and Arthur Gretton. Demystifying mmd gans, 2021.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.