Mini-project 2 in Advanced Machine Learning (02460), Group 50

by Kasper J. (s204231), Andreas K. H. (s204160) & Klaus J. B. (s204123) on 3 April 2024

Part A: Fisher-Rao geodesics

We train a VAE on a 3-class subset of MNIST with a 2-dimensional latent space. The main goal is to explore geodesics as a distance measure between latent variables, ensuring they stay invariant to potential deformations in the encoded space. We calculate geodesics under the Fisher-Rao metric by minimizing the Fisher-Rao energy of parameterized third-order polynomial curves. These curves start with a linear initialization (Figure 1, top left) and are optimized using an LBFGS optimizer with Wolfe line search. Figure 1 (bottom left) shows 50 computed geodesics between randomly selected pairs of test data. The geodesics' paths prioritize denser data regions compared to the linear initialization. This aligns with our intuition regarding the Fisher-Rao energy, as similar latent variables should result in similar distributions, thereby minimizing local KL divergence. However, we observe certain shortcuts occurring over less dense regions.

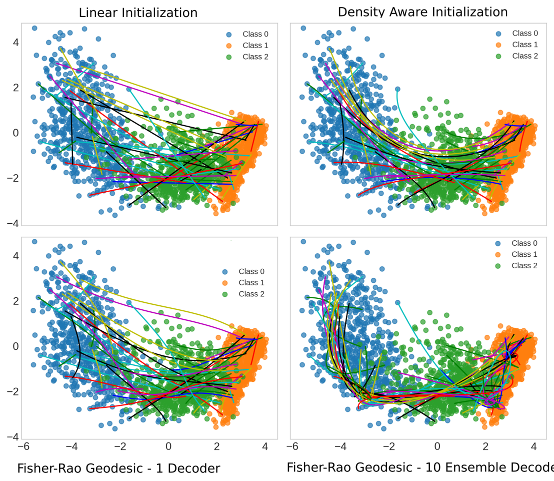

Figure 1: Geodesics in latent space. Top: Different initialization strategies - Linear and density-aware. Bottom left: Geodesics minimizing Fisher-Rao energy with our VAE. Bottom right: Geodesics with our 10-decoder ensemble VAE.

We assess the reliability of computed geodesics across two VAE training runs with different weight initializations. We compute the Mean Relative Absolute Error (MRAE) for the geodesic curve lengths across runs for identical data pairs. Table 1 shows that the geodesic lengths are only slightly more consistent across runs than Euclidean distances. A paired t-test shows a significant difference between geodesics between two training runs. We hypothesize that geodesics on the manifold leverage the smooth structure to 'cut across' less dense regions. The geodesics thus fail to capture the data's full structure.

Part B: Ensemble VAE geometry

To address reliability issues with computed geodesics across training runs, we repeat the experiments now using an ensemble decoder. Geodesics are now fitted by minimizing the model-average Fisher-Rao curve energy through Monte Carlo approximation. The ensemble decoder comprises ten decoders, uniformly selected at each training call. Figure 1 (bottom right) shows the geodesics obtained using this ensemble decoder. The geodesics now appear to follow more similar paths to each other compared to the single decoder case and avoid taking shortcuts. However, some geodesics display small loops that we cannot entirely explain. We hypothesize that this could be mitigated by refining the optimization algorithm or employing another curve parameterization. Training two ensemble models, we find that the geodesic lengths are not statistically significantly different, see Table 1. Thus our distance measure is reliable across different training seeds. We also measured the adherence of computed geodesics with the latent test data using proximity. We assess the proximity of the curves as a function of ensemble member count, see Figure 2. We see that proximity decreases until an ensemble size of 3, where it plateaus. This suggests that reliable geodesics might be achievable using fewer than 10 ensembles.

Table 1: Comparison of the reliability of distance measures between two differently seeded training runs.

	p-val	**MRAE**
Euclidean distance	0.008	22.40 %
Geodesic length	0.014	20.86 %
Geodesic length (10 ensembles)	0.415	4.32 %

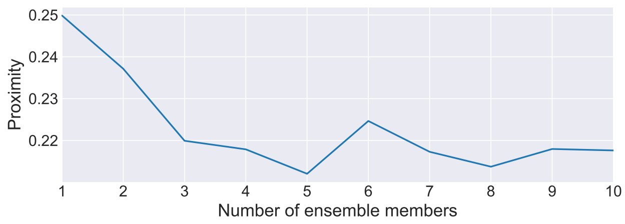

Figure 2: Average proximity as a function of the number of ensemble members in the decoder.

Part C: Impact of Initialization

Instead of our current linear initialization, we could use the geodesics under the Abstract Density Metric (ADM) from (Tomczak, 2022), as shown in Figure 1 (top right). We expect this initialization to be good because dense areas likely contain similar distributions. The mean model-average Fisher-Rao energy for the ADM geodesics is **1241**, which is smaller than **1420** for linear initialization. A lower initial energy can thus be achieved relatively inexpensively with ADM, as it is fast to compute. This can help reduce the number of expensive optimization steps required to find the Fisher-Rao geodesics. We noticed a one-third decrease in optimization steps with Adam, but LBFGS did not benefit much from improved initialization due to its ability to take large initial steps with line search.

Code snippets

A code snippet of the algorithm for computing the Fisher-Rao energy of a curve.

```python
class FisherRao(nn.Module):
    def __init__(self, curve: nn.Module, decoder: nn.Module, n: int = 1024):
        # definitions of stuff
    def energy(self) -> torch.tensor:
        fct = self.decoder(self.curve(self.t))
        probs = fct.base_dist.probs  # Access probs of ContinuousBernoulli dist
        dist_a = td.Independent(td.ContinuousBernoulli(probs=probs[:-1]), 3)
        dist_b = td.Independent(td.ContinuousBernoulli(probs=probs[1:]), 3)
        return td.kl_divergence(dist_a, dist_b).sum()
    def optimize_curve(self, n_iter: int = 1000, lr: float = 1e-1, gamma: float = 0.8):
        #lbfgs optimizer
        optimizer = torch.optim.LBFGS(self.curve.parameters(), line_search_fn='strong_wolfe')
        def closure():
                optimizer.zero_grad(); loss = self.energy() loss.backward(); return loss
        for i in range(n_iter):
            optimizer.step(closure)
            #... code to calculate the stop criteria
            if max_param_change < tolerance:
                break
```

Init and forward function for the third order curve parameterization (Cubic Bezier curves (Mortenson, 1999)).

```python
def __init__(self, x0: torch.tensor, x1: torch.tensor):
    super(ThirdOrderPolynomialCurve, self).__init__()
    self.x0, self.x1 = x0, x1 #end points
    # Initialize control points P1 and P2 as torch parameters to be optimized
    # Initially setting them to be linearly spaced between x0 and x3
    self.P1 = nn.Parameter((2*x0 + x1) / 3, requires_grad=True)
    self.P2 = nn.Parameter((x0 + 2*x1) / 3, requires_grad=True)
def forward(self, t: torch.tensor) -> torch.tensor:
    return ((1-t)**3)*self.x0+3*((1-t)**2)*t*self.P1+3*(1-t)*(t**2)*self.P2+(t**3)*self.x1
```

Relevant code for ensemble model. Inherits most code but energy calculation is updated.

```python
class FisherRaoEnsemble(FisherRao):
    def __init__(self, curve: nn.Module, decoder_list: nn.Module, n: int = 1024):
        # Other self.__ definitions and super
        self.decoder = decoder_list
        #having one decoder reduce the energy to normal fisher rao energy.
        self.mc_samples = 1 if len(decoder_list) == 1 else 5
    def energy(self) -> torch.tensor:
        #code is vectorized for speed. However it makes it a bit more difficult to read
        fct_list = [decoder(self.curve(self.t)) for decoder in self.decoder]
        probs_all = torch.stack([fct.base_dist.probs for fct in fct_list])
        monte_carlo_samples = torch.randint(0, len(fct_list), (self.n, self.mc_samples, 2))
        probs_a, probs_b = [], []
        for i in range(self.n-1):
            probs_a.append(probs_all[monte_carlo_samples[i,:, 0], i])
            probs_b.append(probs_all[monte_carlo_samples[i,:, 1], i+1])
        dist_a = td.Independent(td.ContinuousBernoulli(torch.stack(probs_a)), 3)
        dist_b = td.Independent(td.ContinuousBernoulli(torch.stack(probs_b)), 3)
        return td.kl_divergence(dist_a, dist_b).sum()/self.mc_samples
```

Class for optimizing geodesic under the abstract density metric, for improved initialization technique:

```python
class AbstractDensityMetricGeodesic(nn.Module):
    def __init__(self, curve: nn.Module, data: torch.tensor, n: int = 1024):
        #initialize stuff
    def riemannian_metric(self, x: torch.tensor) -> torch.tensor:
        #evaluate the density of a batch of points x under the kernel density estimator
        px, n = self.evaluate_p_x_batch(x), x.shape[1]
        G = torch.zeros((x.shape[0], n, n))
        G[:, torch.arange(n), torch.arange(n)] = 1/(px+1e-4)[:,None]
        return G
    def energy(self) -> torch.tensor:
        # code is vectorized for speed . However it makes it a bit more difficult to rea
        curve_points = self.curve(self.t)
        curve_grad = self.curve.gradient(self.t).unsqueeze(-1) # shape -> [n, d, 1]
        G = self.riemannian_metric(curve_points) #shape -> [n, d, d]
        Gx = torch.matmul(G, curve_grad)  # Shape: [n, d, 1]
        curve_grad = curve_grad.transpose(1, 2)  # Shape: [n, 1, d]
        integrand = torch.matmul(curve_grad, Gx).squeeze(-1).squeeze(-1)  # Shape: [1024]
        return torch.trapz(integrand , self.t)
    def optimize_curve(self, n_iter: int = 1000, lr: float = 1e-1, gamma: float = 0.8):
        #lbgfs optimizer
```

References

M. E. Mortenson. *Mathematics for Computer Graphics Applications.* Industrial Press Inc., 1999. ISBN 9780831131111.

J. M. Tomczak. *Deep Generative Modeling.* Springer, 2022.