EE660 Homework 5

Assigned: 4/3/24, Due: 4/24/24

Instructions: While you are free to discuss and debug your implementations with others, we expect every student to submit their **own** solution.

Solutions are due by 11:59pm Pacific Time on the due date, and are only to be submitted on DEN. Do not email the course staff with your assignment.

Problem Statement

This last homework is a programming assignment. Your task is to implement two of the generative models we discussed in class, and provide both a quantitative and qualitative assessment comparing the performance of the models you implemented. This assignment has two deliverables: (a) the code you wrote to complete the assignment (notebooks, modules, etc.), and (b) a short ($\sim$ 3 page), write-up which will be described shortly.

To jog your memory, here is the list of models we discussed in this class which you may choose from to implement:

(a) Energy-based models (EBM),

(b) Denoising score matching (DSM),

(c) Variational autoencoder (VAE),

(d) Denoising diffusion probabilistic model (DDPM),

(e) Continuous normalizing flow (CNF),

(f) Generative adversarial network (GAN).

(Note that score matching (SM) is excluded from this list above, because we will be providing you with an example notebook containing an implementation of SM.)

Dataset wise, the following two-dimensional dataset generators are provided for you in the notebook `datset_generators.ipynb`:

(a) Spiral,

(b) Pinwheel,

(c) Checkerboard,

(d) Gaussian mixtures.

Here is a detailed breakdown of what is expected in this assignment, along with a grading breakdown (note that this assignment is out of 100 points):

Implement two generative models (50 points): Choose two generative models from the list above, and implement both the model's *training* procedure, in addition to the model's *sampling* procedure.

In all of these generative models, there is a point where a general purpose function approximator is needed. For example, in energy models, where $p_\theta(x) \propto \exp(f_\theta(x))$, the function approximator is $f_\theta(x)$. As another example, in DDPM, we need to learn a denoiser $f_\theta(x, t)$ using a general function approximator. In this assignment, you are free to use any general function approximator you want (e.g., kernel machine, neural networks, random features, etc.). However, if you do not have any particular affinity for a specific approximator, we recommend using a simple fully connected multi-layer perceptron (MLP).

Furthermore, you are also free to use any software stack you wish. But again if you have no particular affinity, we recommend that you use `jax`,[1] in particular the `flax` library[2] to define neural networks, and the `optax` library[3] to implement gradient-based optimization. This is the same software ecosystem that the notebook we provide you is written in.

Regarding code reuse: the reuse of code snippets that you find online is permissible given proper attribution, **for code not directly related to the generative model**. For example, if you find a snippet online for defining a 3 layer neural network in `flax`, this is an acceptable piece of code reuse. However, copying an existing implementation of a VAE off the internet and submitting it as yours is **strictly prohibited**. You are of course allowed to look at somebody else's implementation to get an idea of how they e.g., defined the training loss, but you need to write your own original implementation. Another thing which is prohibited is directly calling a library which implements a generative model. For example, using `diffusers.DDPMPipeline` from huggingface's API is **not allowed**. Please exercise common sense in this manner, and if there is a piece of code reuse you are unsure about, feel free to come to office hours and discuss it with the course staff.

Full credit here involves turning in original, working code which trains on the datasets from the dataset generators and generates new samples. The easiest way to demonstrate this to perform the required analysis (described subsequently in the document) in a Jupyter notebook which clearly imports your model/train code (we recommend keeping the model/train code separate from the analysis notebook), and submit this notebook (with the cell outputs kept) along with the rest of your code. Note that the *quality* of the training/sampling is assessed later: this step is simply about producing original implementations which actually run.

Hints: we recommend selecting DSM as one of your model implementations, since it is very closely related to the SM example provided in the notebook. But this is not a requirement. Also, if you choose to implement a continuous normalizing flow, we recommend using `diffrax`[4] to define the log-probability ODE, since `diffrax` takes care of auto-differentiation through ODEs.

Train both models on two datasets listed above (20 points): Choose two datasets listed above and train both your generative models on $n = 2000$ training samples from each dataset (that is, you should have *four* model parameters in total: the cross product of two models and two dataset). In your writeup, include the training curves, showing the progress of training for each model/dataset. These training curves should typically be of the form where a loss function is plotted

[1] `https://jax.readthedocs.io/en/latest/`

[2] `https://github.com/google/flax`

[3] `https://github.com/google-deepmind/optax`

[4] https://docs.kidger.site/diffrax/

on the y-axis, and the number of gradient steps (if you are using gradient based optimization) is plotted on the x-axis. The precise value of the loss function plotted will be different for each model. For example, for VAE the loss would be the empirical ELBO, for DSM the loss would be the denoising score matching loss, etc. Also, in your write-up, plot the value of the loss function on $n = 500$ sample hold-out set, alongside with your training loss values, to show that your model is not overfitting.

Regarding hyperparameter values (e.g., the step sizes for gradient based optimization, width of the neural networks used, dimensionality of the latent variable in a VAE, etc.), we do *not* expect you to conduct an exhaustive hyperparameter sweep. Simply document in your write-up the various hyperparameter values that you played with, if necessary to improve the training performance and/or reduce overfitting; you only need to plot the train/hold-out curves for your best hyperparameter values.

Full credit for this step involves demonstrating that both your model implementations are able to stably decrease their training loss (or increase its objective depending on how you define it) on both datasets. In other words, the training curves should trend in the right direction (they certainly do not need to be monotonic, but the trend should be clear). There are two exceptions to this: (1) the training process of energy based model is quite difficult to stabilize, and (2) for GANs, since a min/max game is being optimized, the best we can hope for in the training curves is that the value stabilizes to a number (ideally this number is $-2 \log 2$, but it is OK if your implementation does not converge precisely to $-2 \log 2$). Therefore, for EBM and GAN training curves, we will be fairly lenient on the grading; feel free to check with the course staff in OH if you are wondering whether your implementation is behaving reasonably.

Qualitative comparison of model sample quality (15 points): Qualitatively compare the quality of the learned model samplers by drawing samples from each of your learned models, and comparing it to the samples from the true distribution. Specifically, for each (model, dataset) pair, draw N samples (N is a free parameter for you to decide) from both the model and the dataset generator, and visualize all the $2N$ cumulative samples on the same 2D plot (using one color for the model and another color for the dataset generator). Include this comparison in your write-up, and make an assessment based on your plots about which model generates samples with higher fidelity. Comment on any abnormalities in the samples generated by your models.

Note that there is no right answer here: depending on a combination of various factors including learning rates, function approximator capacity, etc., since the datasets we consider are relative simple, we do not expect there to be a clear-cut winner in every situation.

Quantitative comparison of model sample quality (15 points): The last part we ask is to give a *quantitative* comparison of the model sample quality. In general, this is actually quite difficult to do. But the fact that our datasets are all two dimensional, and the fact that we have a dataset generator available to us (not just a fixed training set) affords us some opportunities.

Here are a few ideas. For instance, imagine we are comparing a normalizing flow model to a VAE. For the normalizing flow, we could simple evaluate the average log-likelihood under our trained model of a hold out dataset, and that would give us an unbiased estimate of the true log-likelihood. On the other hand, for the VAE, we could evaluate a Monte-Carlo estimate of the ELBO on a holdout dataset, and use that as a lower bound of the true log-likelihood. Of course this methodology is not 100% satisfying since we not exactly comparing apples to apples by ignoring

the gap between the ELBO and the log-likelihood, but for our purposes it will suffice. What if, on the other hand, our model does not have a likelihood function? Since our distributions are two dimensional, we can turn to kernel density estimation (KDE). We can fit a KDE to samples from our model, using say `scipy.stats.gaussian_kde`,[5] and use the KDE model to compute the log-likelihood (see the `logpdf` function). This can be done across models to get an apples-to-apples numerical comparison. See Section 7.2 of `https://arxiv.org/pdf/2309.05803.pdf` for an example of both a qualitative comparison, and a quantitative comparison between different generative models on different datasets.[6]

Of course, there are other methods. Again, there is no right method here. A full credit answer can use any method (including the ones described in the previous paragraph) that reasonably allows us to compare models numerically, as long as it is explained clearly what you are doing. If you come up with another method and want to discuss it with the course staff, again feel free to come to OH. Also, if you come across another evaluation methodology in a research paper that seems sound, feel free to cite it and use it (with proper explanation).

Deliverables

Here is a list of the items to submit on DEN:

(a) Code (notebooks, modules, etc.).

(b) A writeup (in PDF format) describing which models you implemented, which datasets you trained your models on, the training curves (along with the hyperparameters you used for your model and training), the qualitative comparisons (figures and explanations), and finally the quantitative comparisons (tables containing the quantitative metrics and explanations). There is no page limit, but we encourage you to write concisely (a longer writeup is not necessarily better). There is also no style file requirement (so you can format the writeup however you like).

Extra Credit Opportunities

We will provide the opportunity to earn up to 70 points of extra credit on this assignment (so the maximum score is 170/100 points): 70 points on one homework assignment represents about $\sim 5\%$ of the total grade in the course.

Recall in the problem description, you are required to implement two different models and train your models on two different datasets. You can get extra credit by running your models on more datasets, and also by implementing more models. Specifically, training your model on 3 datasets will receive an extra 10 points, and training on 4 datasets an extra 20 points. Furthermore, implementing 3 models will receive an extra 25 points, and implementing 4 models will receive an extra 50 points. Note that to receive the full extra credit, you need to propagate the extra models/datasets throughout your writeup. For example, in your qualitative comparison, if you implement 3 models and 4 datasets, then you should have 12 total plots (organized via say a 4×3

[5] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html`

[6] Note that the quantitative comparison here is a bit more complex than described in the main problem description, since it uses a KDE to approximate the integral defining the Bhattacharyya distance (`https://en.wikipedia.org/wiki/Bhattacharyya_distance`) between the true data distribution and the model's sampling distribution.

table of subplots, please do not submit pages and pages of figures without making an effort to consolidate the figures in a logical way).

If you decide to go for the extra credit, please indicate very clearly at the beginning of your writeup which extra credit options you decided to implement.

Questions and Answers

Can I implement another type of generative model not on this list? If you are interested in doing so, please email the course staff and we will discuss it.

What if I want to train my generative models on image datasets (e.g., MNIST/CIFAR10) instead of these 2D datasets? We chose to do 2D datasets for this assignment to ease computation requirements; none of the models trained for this assignment should require access to GPUs in order to train in a reasonable amount of time. However, if you have access to GPUs and are interested in working with image data, come speak with the course staff. We will allow you to substitute the extra credit portion of this assignment with extra credit based on implementing image-based generative models. Note that the total amount of extra credit points will be the same, and furthermore we expect implementing the image-based models to be *strictly more work than the current extra credit opportunities*. But, if this is something you are passionate about pursuing, we will be happy to accommodate your request.

Hints

Denoising score matching. We found the following implementation details helped our DSM model train better:

(a) *Rewrite the loss:* The original DSM loss is

$$\hat{L}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \| f_\theta(\tilde{x}_i) - (x_i - \tilde{x}_i)/\sigma^2 \|^2.$$

One issue is that for small σ, the RHS can become quite large. Thus, we found it beneficial to use the following identity:

$$f_\theta(\tilde{x}_i) - (x_i - \tilde{x}_i)/\sigma^2 = \frac{1}{\sigma^2}\left[\sigma^2 f_\theta(\tilde{x}_i) - (x_i - \tilde{x}_i)\right],$$

which makes our loss:

$$\hat{L}_n(\theta) = \frac{1}{n\sigma^4}\sum_{i=1}^{n} \| \tilde{x}_i + \sigma^2 f_\theta(\tilde{x}_i) - x_i \|^2.$$

Furthermore, when implementing a batch gradient, we remove the scaling factor $1/(n\sigma^4)$.

(b) *Use multiple noise realizations for every x_i:* Fix a positive integer k. We let $\tilde{x}_i^j = x_i + \sigma w_i^j$, where $\{w_i^j\}_{i,j=1}^{n,k}$ are iid $N(0, I)$ noise vectors. With this notation, we consider the following

variance-reduced loss (building on top of our previous modification):

$$\hat{L}_n(\theta) = \frac{1}{nk\sigma^4} \sum_{i=1}^{n} \sum_{j=1}^{k} \|\tilde{x}_i^j + \sigma^2 f_\theta(\tilde{x}_i^j) - x_i\|^2.$$

Furthermore, in forming a stochastic gradient, while we dropped the $1/(n\sigma^4)$ scaling factor, we kept the $1/k$ scaling factor (so that the scale of the batch gradient remains the same regardless of our choice of k).

(c) *Noisy training curve:* Due to the added noise to the loss function, the training curves for DSM will be substantially noiser than the training curves for regular score matching. This is to be expected. You may also plot (on top of the original training curve) a filtered version of the training loss running through an exponential moving average to smooth out the noise (this is not required, but may help in understanding if your model is actually training).