Deep Learning Homework 3
Generative Models

due on April 21, 2021

In this homework, you need to implement 3 generative models to generate MNIST images. Please clone the starter code from https://github.com/IrisLi17/dl_course_hw3.git. You are only required to solve either Problem 1 or Problem 2 while Problem 3 and Problem 4 are compulsory. You will get a bonus if you solve all the 4 problems. Please submit all the code and your report to web learning.

1 Energy-Based Model

The energy-based method aims to train a parameterized model $E = -f(x;\theta)$ to model the unnormalized data distribution $p(x)$. In this problem, we instantiate $f(x;\theta)$ as a MLP (see `MlpBackbone` in `ebm/ebm.py`). Your tasks are as follows:

- Implement all the missing parts in the contrastive-divergence training pipeline in `ebm/ebm.py` and `ebm/main.py`. Basically, we want to decrease the energy of positive samples while increase the energy of negative samples. The positive samples are from the training set, and the negative samples are sampled using Langevin dynamics starting from either random noise or previously generated samples. Please see Alg. 1 for details.
- We corrupt the test images by adding noise to the pixels in even rows (see Fig. 1). Please implement an inpainting procedure to recover the original image by sampling from the learned energy-based model, then report the mean squared difference between your recovered images and the ground truth images.

(a) An image from MNIST.

(b) A corrupted image.

Figure 1: Adding noise to the pixels to generate corrupted images.

Algorithm 1: Energy-based training algorithm

Input: dataset $\mathcal{D}$; energy-based model $f(x;\theta) \to \mathbb{R}$; replay buffer $\mathcal{B}$ for storing previously generated samples

while not converged **do**

- $x^+ \sim \mathcal{D}$
- $x^0 \leftarrow$ init_negative()
- $x^- \leftarrow$ langevin_dynamics(x^0)
- `/* train_step() in ebm/ebm.py */`
- $\theta \leftarrow$ contrastive_divergence_update(x^+, x^-, θ)
- $\mathcal{B} \leftarrow \mathcal{B} \cup \{x^-\}$

end

You need to submit your code and your trained model. Make sure your model runs well with "`python main.py --play --load_dir /path/to/your/model`", and attach a note in `README.md` to specify all the command line arguments. Your implementation will be graded based on the mean squared error of inpainting.

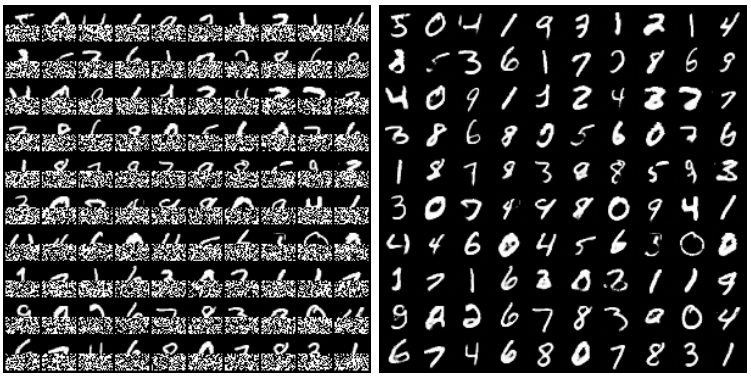
Tips:

- Training with naive contrastive-divergence algorithm will make your model diverge quickly (think about why). Therefore, you need to add a L2 regularization term $\alpha(E_\theta(x^+)^2 + E_\theta(x^-)^2)$ to stabilize training.
- Keep track of the generated samples during training to get a sense of how well your model is evolving.
- You can take a look at the paper "Implicit Generation and Generalization in Energy Based Models" to learn more about useful tricks to get your model working.

2 Normalizing Flows

In this problem, you need to train a real NVP model on MNIST images for generation and inpainting. You should

1. Complete the code in "realnvp.py".



(a) Corrupted images. (b) Recovered images.

Figure 2: An example of image inpainting using normalizing flows.

2. Run "train.py" to train your code and observe the generated images in the folder named "images/".
3. Run "inpainting.py". Under the folder "inpainting_images/", compare your generated images with "img_masked.png" and "img_xxx.png" (e.g. "img_001.png"). See Fig. 2 as an example.
4. Modify model structure and hyper-parameters to train a model for better inpainting.

Tips:

- Read "Density estimation using Real NVP" and "NICE: Non-linear Independent Components Estimation" for more useful tricks.
- We will test your model based on your performance of inpainting.
- Submit all the python files and the best model in your folder "models/", make sure you can run "inpainting.py" directly.
- You should **NOT** modify "inpainting.py".

3 Variational Autoencoder

In this problem, you need to implement a class conditioned variational autoencoder to generate MNIST images. The images x are of size 784. The labels y are one-hot vectors of size 10. The latent variables z are of size 100. The encoder $q(z|x, y; \phi)$ and the decoder $p(x|z, y; \theta)$ are both MLPs defined in `vae/vae.py`. Please do not modify the network architecture. We suppose the prior $p(z)$ is a standard Gaussian distribution $\mathcal{N}(0, I)$. Also, we assume $q(z|x, y)$ and $p(x|z, y)$ are Gaussian distributions.

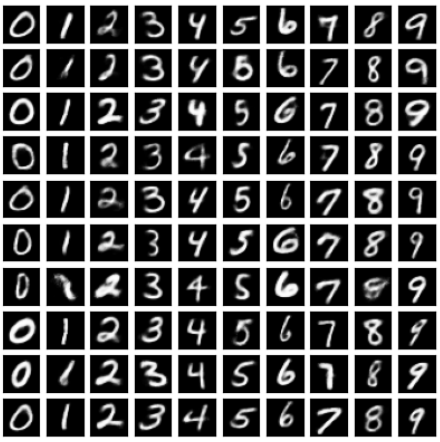


Figure 3: An example of class-conditioned generation.

You need to implement the training loop, and generate class-conditioned samples, i.e., images of digits 0-9. Please show the conditioned generation results in your report, ideally in a 10×10 plot with each column showing samples from the same class (see an example in Fig. 3). Other statistics that you believe can support your result should also be included in the report.

You need to submit all the code and your model to run "`python vae.py --eval --load_path /path/to/your/model`", and attach a note in `README.md` to specify all command line arguments. We will evaluate your implementation by classifying the generated samples of your model using a well-tuned classifier. You can get all the points for your implementation if the classifier can achieve an accuracy $\geq 98\%$. We encourage you to train a classifier (using code snippets in Homework 2) to validate your generation model before submitting it to us.

4 Generative Adversarial Network

In this problem, you need to do class-conditioned generation by training a generative adversarial network. The architecture of the generator $G : (z, y) \to x$ and the discriminator $D : x \to [0, 1]$ is already defined in the code skeleton. Please do not modify them.

You need to implement the training procedure by yourself. Again, please include the conditioned generation results and other relevant statistics in your report.

~~The grading policy is the same as that for Problem 3.~~ We will evaluate your model by running a classifier on your generated images, as what we do for Problem 3. In addition, the standard deviation of your generated images within each class must exceed a certain threshold. Suppose the generated `samples` for each class are of size `(number of samples, 28, 28)` with pixel values range

from 0 to 1, the standard deviation is given by `torch.std(samples, dim=0).mean()`. The thresholds H for different digits are listed in Table 1.

Digit	0	1	2	3	4	5	6	7	8	9
H	0.17	0.08	0.17	0.15	0.14	0.16	0.15	0.13	0.15	0.13

Table 1: Thresholds for intra-class standard deviation.