Due Date: May 8th 23:00, 2023
Instructions

- *This assignment is involved – please start well ahead of time.*
- *For all questions, show your work!*
- *Submit your report (PDF) and your code electronically via the course Gradescope.*
- *TAs for this assignment are* ***Reza Bayat*** *and* ***Johan S. Obando C.***

In this assignment, you will be required to implement two different generative models that are widely popular in Machine Learning literature, namely Variational Autoencoders (VAE) and Diffusion Models and Simsiam algorithm, a self-supervised learning (SSL) model. You will try VAE and Diffusion Models on the SVHN dataset which consists of images of street-views of house numbers, and is similar to the popular MNIST dataset. For the SSL part, you will use CIFAR10 dataset.

For all the models, you have been provided with the starter codes as well as data normalizing and loading scripts. Your job would be to fill up certain missing parts in each of the model implementations (details within the code notebooks and each of the question parts) so as to enable proper training of the generative and SSL models.

Coding instructions: You will be required to use PyTorch to complete all questions. Moreover, this assignment **requires running the models on GPU** (otherwise it will take an incredibly long time); if you don't have access to your own resources (e.g. your own machine, a cluster), please use Google Colab. For most questions, unless specifically asked, only code up the logic using basic `torch` operations instead of using advanced torch libraries (eg. `torch.distributions`).

For all the models provided, you are encouraged to train for longer to see even better results.

Important: Before submitting the code to Gradescope, please remove the `!pip install ...` line from all the solution python (`.py`) files. For submitting the VAE code, rename your python file as `vae_solution.py`, for Diffusion Model / DDPM code, `ddpm_solution.py`, and for the SSL part, `q3_solution.py`.

Problem 1

The task is to implement a Variational Autoencoder on the SVHN dataset. VAEs are a class of latent-variable generative models that work on optimizing the ELBO, which is defined as

$$ELBO(\theta, \phi) = \sum_{i=1}^{N} \mathbb{E}_{q_\phi(z|x_i)}[log p_\theta(x_i|z)] + \mathbb{KL}[q_\phi(z|x_i)||p(z)]$$

where we are given a dataset $\{x_i\}_{i=1}^N$ and $p_\theta(x|z)$ is the conditional likelihood, $p(z)$ is the prior and $q_\phi(z|x)$ is the approximate variational distribution. Optimization is done by maximizing the ELBO, or minimizing the negative of it. That is,

$$\theta^*, \phi^* = \text{argmin} \;\; ELBO(\theta, \phi)$$

While there can be many choices of such distributions, in this assignment we will restrict our focus on the case where

$$\begin{aligned} q_\phi(z|x) &= \mathcal{N}(\mu_\phi(x), \Sigma_\phi(x)) \\ p_\theta(x|z) &= \mathcal{N}(\mu_\theta(z), I) \\ p(z) &= \mathcal{N}(0, I) \end{aligned}$$

where $\Sigma_\phi(x)$ is a **diagonal covariance matrix** with the off-diagonal elements as 0.

For implementing the VAE, you will have to

1. Implement the `DiagonalGaussianDistribution` class, which will form our backbone to parameterize all Gaussian distributions with diagonal covariance matrices, and will come handy when we are implementing the VAE itself.
   - (1 pts) Implement the `sample` function that samples from the given gaussian distribution using the **reparameterization** trick, so that gradients can be backpropagated through it. (*Reparameterization Trick: You can sample from a gaussian distribution with mean μ and variance σ^2 by doing a **deterministic** mapping of a sample of* $\mathcal{N}(0, 1)$)
   - (2 pts) Implement the `kl` function that computes the Kulback Leibler Divergence between the given guassian distribution with the standard normal distribution.
   - (2 pts) Implement the `nll` function that computes the negative of the log likelihood of the input sample based on the parameters of the gaussian distribution.
   - (1 pts) Implement the `mode` function that returns the mode of this gaussian distribution.
2. Implement the `VAE` class, which is the main model that takes care of encoding the input to the latent space, reconstructing it, generating samples, and computing the required ELBO-based losses and importance-sampling styled log likelihood computations.
   - (2 pts) Implement the `encode` function that takes as input a data sample and returns an object of the class `DiagonalGaussianDistribution` which parameterizes the approximate posterior with mean and log of the variance that are obtained through the `encoder` and `self.mean`, `self.logvar` Neural Networks.
   - (2 pts) Implement the `decode` function that takes as input a sample from the latent space and returns an object of the class `DiagonalGaussianDistribution` which parameterizes the conditional likelihood distribution with mean obtained from the decoder and the variance fixed as identity.
   - (2 pts) Implement the `sample` function that takes a batch size as input and outputs the mode of $p_\theta(x|z)$. To do this, first generate z from the prior, then use the `decode` function to obtain the conditional likelihood distribution, and return its mode.

- (4 pts) Implement the `log_likelihood` function which takes as input the data as well as a hyperparameter K and computes an approximation to the log-likelihood, which is a popular metric used in such generative models. To compute the approximate log-likelihood, we need to compute

$$\log p_\theta(x) \approx \log \frac{1}{K} \sum_{i=1}^{K} \underbrace{\frac{p(x_i, z_k)}{q(z_k|x_i)}}_{\Gamma}$$

  where $z_k \sim q(z|x_i)$. To compute this, we would actually compute $\log \Gamma$ and then use the log-sum-exp trick. It is also recommended to use the `DiagonalGaussianDistribution` class (that you coded above) in the solutions to this part.
- (2 pts) Implement the `forward` function that takes as input a data sample, encodes it into a variational distribution, draws a reparameterizable sample from it, decodes it and returns the mode of the decoded distribution. It should also return the conditional negative log-likelihood of the data sample under $p_\theta(x|z)$ and the KL-Divergence with standard normal.
- (2 pts) Finally, finish the code provided in `interpolate` to provide some visualization of the methodology. This is meant to interpolate (or linearly move) in the latent space between two points and see how such effects of their latent space lead to (possibly smooth?) transitions in the observed space.

3. During and after training the model, we ask you to provide the following additional results from your experiments. Please provide
   (a) (2 pts) Plot showing the wall clock time of training procedure.
   (b) (2 pts) Several generated samples during training after every 5 epochs. (It should be specified from which epoch samples were generated).
   (c) (4 pts) Some reconstructions from the test set during training after every 5 epochs.
   (d) (4 pts) Samples from the VAE model at the end of training. How do the samples look? Do you see digit patterns; are the samples blurry?
   (e) (6 pts) Show a latent variables profile for each class in the dataset by averaging the latent representations of a few samples from the training set corresponding to each class. Generate bar plots for each class to visualize the distribution of latent variables. Look for patterns in the representations and identify any spikes in certain latent variables.
   (f) (2 pts) Images of the interpolation results from the code. Is the interpolation between two points smooth? Do you see the images changing smoothly?

Problem 2

The task here is to implement the Denoising Diffusion Probabilistic Model (DDPM) on the SVHN dataset. Diffusion models are an up-and-coming class of generative models that rely on a known forward diffusion process, which progressively destroys structure from the data until it converges to unstructured noise, eg. $\mathcal{N}(0, I)$ and a learned parameterized (by a Neural Network!) backward process that iteratively removes noise until you have obtained a sample from the data distribution.

In particular, one constructs the forward diffusion process as

$$q(x_t|x_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

with β_t's defining the noise schedules, typically kept as 0.0001 at $t = 1$ and 0.02 at $t = T$ with a linear schedule in between. One can see this process as iteratively adding more noise to the sample and destroying the structure in it.

The backward process then parameterizes a distribution

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(\mu_\theta(x_t, t), \tilde{\beta}_t I)$$

where $\tilde{\beta}_t$ is the variance of the distribution $q(x_{t-1}|x_t, x_0)$, and by learning the parameter θ, one hopes to **denoise** slightly from x_t to x_{t-1}. With a bit of algebra and through some computations, this leads to parameterizing a noise model instead of the mean, that is $\epsilon_\theta(x_t, t)$, which equivalently leads to the distribution

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right), \tilde{\beta}_t I)$$

This leads to the training objective being just prediction of noise,

$$\mathbb{E}_{t\sim\mathcal{U}(1,T),x_0,\epsilon_t}\left[||\epsilon_t - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t, t)||^2\right]$$

and the sampling being iterative, where a data sample is obtained by sampling $x_T \sim \mathcal{N}(0, I)$ and then progressively sampling $x_t \sim p_\theta(x_{t-1}|x_t)$. For more details, please refer to the theory provided in the colab notebook, as well as the original DDPM paper and a popular blog post, both linked to in the notebook.

1. Implement the DDPM model and training by following the given steps:
   - (4 pts) Pre-compute the following useful coefficients / variables: β_t, α_t, $\frac{1}{\sqrt{\alpha_t}}$, $\bar{\alpha}_t$, $\sqrt{\bar{\alpha}_t}$, $\sqrt{1-\bar{\alpha}_t}$, $\bar{\alpha}_{pt}$ (which is $\bar{\alpha}_t$ right-shifted and padded with 1), and $\tilde{\beta}_t$. Details are provided in the Notebook.
   - (4 pts) Complete the `q_sample` function that implements sampling from the distribution $q(x_t|x_0)$ using the reparameterization trick.
   - (4 pts) Complete the `p_sample` function that implements sampling from the distribution $p(x_{t-1}|x_t)$ using the reparameterization trick.
   - (2 pts) Complete the `p_sample_loop` function that uses `p_sample` function and iteratively denoises completely random noise from $t = T$ to $t = 1$.
   - (4 pts) Implement the `p_losses` function that generates some random noise, uses this random noise to get a noisy sample, gets its noise prediction, and then returns the **smoothed L_1 loss** between the predicted and true noise.

- (2 pts) Finally, implement the random sampling of time-steps in the function `t_sample`.

2. During and after training the model, we ask you to provide the following additional results and insights from your experiments. Please provide
   (a) (2 pts) Plot showing the wall clock time of training procedure.
   (b) (2 pts) Several generated samples during training after every 5 epochs. (It should be specified from which epoch samples were generated).
   (c) (6 pts) Some reconstructions from the test set. In this part, add noise to samples at the timestep $t = t'$ and attempt to reconstruct the sample using the reverse diffusion process (similar idea of sampling, but not initiated from random noise.). Try different level of noise to determine the specific timestep $t = t^*$ at which the original samples can no longer be restored, may resulting in the generation of different samples. (There are some applications of this process, such as the paper on purifying adversarial examples.)
   (d) (4 pts) Some generated samples from the Diffusion model at the end of training. How do the samples look? Do you see digit patters; are the samples blurry? Compare them with VAE samples.

3. (Optional) Although these sections are not mandatory and will not be evaluated, students who are more interested in this topic are encouraged to complete this part.
   (a) (0 pts) Train two additional diffusion models with T=500 and T=1500 and study the effects of increasing the number of time steps on training/sampling time and sampling quality. What are your thoughts on this tradeoff?
   (b) (0 pts) Some studies have employed the intermediate representation of U-Net for tasks other than its original objective. What is your opinion on exploring this direction? You can refer to this paper that employs such representations for semantic segmentation. Please elaborate on your views regarding the applicability of similar concepts for other tasks.

Problem 3

Self-supervised methods learn a representation of data by solving pretext tasks to alleviate expensive supervised labelling. Contrastive learning methods, a sub-category of self-supervised learning (SSL), learn an embedding by minimizing the distance between the embedding of two different views of the same sample while maximizing the distance between the embedding of the view of two different samples. However, non-contrastive SSL methods showed comparable results without using a large number of negative samples. In this question, you will be investigating why these networks do not collapse to a trivial solution by implementing the Simsiam algorithm and experimenting on key factors of its performance.

Simsiam work surprisingly well without several strategies for preventing collapsing. This model directly maximizes the similarity of one image's two views, using neither negative pairs nor a momentum encoder. Here, you will first implement a function to estimate the negative cosine similarity. Then you will implement Simsiam loss and forward functions. In the end, you will run experiments to analyze the performance of the model in different conditions.

Dataset and dataloader for Siamese network [1] In this question, you will work on an object classification task for the **CIFAR10** dataset [2]. This dataset consist of images $\mathcal{X} \subset \mathbb{R}^{32\times32\times3}$ of 10 classes. We provide samplers to generate the different distributions that you will need for this question. In the same repository, we also provide the architecture of a neural network functions f : $\mathcal{X} \to \mathcal{Z}$ and h : $\mathcal{Z} \to \mathcal{P}$ s.t. $\mathcal{X} \subset \mathbb{R}^{32\times32\times3}, \mathcal{Z} \subset \mathbb{R}^d, \mathcal{P} \subset \mathbb{R}^d$ (d is the feature dimension, Simsiam default is 2048)).

Hyperparameters & Training Pointers We provide code for the SSL network as well as the hyperparameters you should be using. We ask you to code the training procedure to train the Simsiam as well as the qualitative exploration that you will include in your report.

1. (2 pts) Implement the forward functions of the Simsiam in '`q3_solution.py`'. This function receive two randomly augmented view x_1 and x_2 from an input sample x and compute the outputs of the network, which are as below:

$$z_1 \triangleq f(x_1), \quad p_1 \triangleq h(f(x_1)) \tag{1}$$

$$z_2 \triangleq f(x_2), \quad p_2 \triangleq h(f(x_2)) \tag{2}$$

   Note that you need to perform the grading stopping in this step as well.

2. (2 pts) Implement the function '`cosine similarity`' in '`q3_solution.py`' to compute the similarity between two inputs. The cosine similarity between two variables A and B can be defined as

$$S_c(A, B) = \frac{A}{\|A\|_2} \cdot \frac{B}{\|B\|_2} \tag{3}$$

3. (2 pts) Implement the Simsiam loss functions in '`q3_solution.py`' to compute the objective function of the Simsiam, using the cosine similarity above. Simsiam objective function is defined as:

$$L = 0.5 * \mathcal{D}(p_1, stop - grad(z_2)) + 0.5 * \mathcal{D}(p_2, stop - grad(z_1)) \tag{4}$$

   Where $\mathcal{D}$ is negative cosine similarity.

4. (8 pts) Train the model for 100 epochs with and without gradient stopping. Plot training loss and KNN accuracy against training epochs and explain results.

[1] A Siamese Neural Network is a class of neural network architectures that contain two or more identical subnetworks. (From here)

[2] The CIFAR10 dataset can be downloaded at https://www.cs.toronto.edu/ kriz/cifar.html. Please note that Pytorch CIFAR10 Dataset can download the dataset so you do not need to download it separately.

5. (9 pts) Investigate the effect of the predictor network (MLP) by experimenting the below setting(s). Plot training loss and KNN accuracy against training epochs.
   (a) Remove the predictor by replacing it with an identity mapping.
   (b) What happens when the dimensionality of the last layer of the projector network is increased? From 2048 to 4096?
   (c) According to the observation above, could you expect same behaviour for Barlow Twins[3] model? Explain your arguments.
6. (4 pts) Incorporating heavy augmentations on SimSiam results in unstable performance even collapsing. How could you alleviate this problem? Are you using data augmentation in the current SimSiam model? If so, where? Hint: Check DSSL [4].
7. (4 pts) The main challenge with joint embedding architectures is to prevent a collapse in which the two branches ignore the inputs and produce identical and constant output vectors. Could you explain how to avoid collapse issues? Hint: Check VicReg [5].
8. (5 pts) What are the main differences between BYOL and SimSiam self-supervised methods? Explain what's the role of them in each model?
9. (4 pts) Why exploiting asymmetry for Siamese representation learning is relevant? How SSL algorithms can benefit from this? [6]

[3]Barlow Twins: Self-Supervised Learning via Redundancy Reduction: https://arxiv.org/abs/2103.03230
[4]Directional Self-supervised Learning for Heavy Image Augmentations: https://arxiv.org/abs/2110.13555
[5]VICReg: Variance-Invariance-Covariance Regularization For Self-Supervised Learning: https://arxiv.org/pdf/2105.04906.pdf
[6]On the Importance of Asymmetry for Siamese Representation Learning: https://arxiv.org/abs/2204.00613