CSCD84: Artificial Intelligence
Assignment 4: Techniques on Generative Modelling
Due Monday 11:59PM, Apr 7, 2025

1 Written Component

1.1 A Benefit of Layer Normalization

In lecture, we have seen that layer normalization (LN) is commonly applied to modern architectures such as the original transformer (Vaswani et al., 2017), GPT-like architecture (Radford et al., 2019), ViT (Dosovitskiy et al., 2021), and many more. However, it is interesting to see that various models apply LN at different steps of the architecture, e.g. Pre-LN and Post-LN (Xiong et al., 2020). Xiong et al. (2020) have noticed that Pre-LN suffers from *representation collapse*, a scenario where hidden representations near the output are similar, while Post-LN suffers from *vanishing gradient*. Here we will investigate the benefit of applying LN before the activation function, which is known to help alleviate the *plasticity loss* in neural networks (Lyle et al., 2024). Plasticity loss means that the neural network can no longer update its own parameters to account for new information, a common problem in reinforcement learning.

For simplicity, let's consider a simplified layer normalization that normalizes the vector $\mathbf{h} \in \mathbb{R}^d$: $\text{SLN}(\mathbf{h}) = \frac{\mathbf{h}}{\|\mathbf{h}\|}$. We can apply SLN either before or after an activation σ:

$$\text{PreSLN}(\mathbf{h}) = \sigma(\text{SLN}(\mathbf{h})), \qquad \text{PostSLN}(\mathbf{h}) = \text{SLN}(\sigma(\mathbf{h})).$$

We can show that when σ is a ReLU function, using PreSLN can potentially provide dead units a second chance at life, in other words, its gradient is non-zero. On the other hand, using PostSLN fails to achieve this. Concretely, let $i, j \in \{1, \ldots, d\}, i \neq j$, and let $z_j = \text{SLN}(\mathbf{h})_j$ be the j'th entry after applying SLN on $\mathbf{h}$. Assume that σ is an element-wise operation, answer the following questions:

1. Show that with PreSLN, we have that
$$\frac{\partial \text{PreSLN}(\mathbf{h})_j}{\partial h_i} = -\frac{\partial \sigma(z_j)}{\partial z_j} \frac{h_i h_j}{\|\mathbf{h}\|^3}.$$
2. Show that with PostSLN, we have that
$$\frac{\partial \text{PostSLN}(\mathbf{h})_j}{\partial h_i} = -\frac{\partial \sigma(h_i)}{\partial h_i} \frac{\sigma(h_i)\sigma(h_j)}{\|\sigma(\mathbf{h})\|^3}.$$
3. Assume that σ is the ReLU activation. Provide a concrete example where PreSLN results in non-zero gradient w.r.t. h_i while PostSLN results in zero gradient w.r.t. h_i.

1.2 Generative Adversarial Networks

Recall that generative adversarial network (GAN) consists of two components: a discriminator $D_\phi(\mathbf{x})$ and a generator $G_\theta(\mathbf{z})$. Suppose that we have a fixed generator $G_\theta(\mathbf{z})$, the discriminator is trained to optimize the following:

$$\max_\phi \mathbb{E}_{\mathbf{x} \sim p_x(\mathbf{x})} \left[\log(D_\phi(\mathbf{x}))\right] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} \left[\log(1 - D_\phi(G_\theta(\mathbf{z}))\right],$$

where p_x is the true data distribution and p_z is some noise distribution. Since the generator is a deterministic function of $\mathbf{z}$, the second term $\mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}\left[\log(1 - D_\phi(G_\theta(\mathbf{z}))\right]$, can be rewritten as

$$\mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}\left[\log(1 - D_\phi(G_\theta(\mathbf{z}))\right] = \mathbb{E}_{\mathbf{x}\sim p_\theta}\left[\log(1 - D_\phi(\mathbf{x}))\right]$$

Answer the following questions:

1. What is the solution to the above optimization problem? Express the solution in terms of p_x and p_θ.
2. Suppose we fix $p_\theta = p_x$, what is the optimal discriminator in this case?
3. Finally, what is the solution for the complete GAN objective

$$\min_\theta \max_\phi \mathbb{E}_{\mathbf{x}\sim p_x(\mathbf{x})}\left[\log(D_\phi(\mathbf{x}))\right] + \mathbb{E}_{\mathbf{x}\sim p_\theta}\left[\log(1 - D_\phi(\mathbf{x}))\right]?$$

1.3 Variational Autoencoder

From lecture we learned that variational autoencoder (VAE) optimizes variational lower bound objective, specifically:

$$\mathcal{L}_{VAE}(\theta, \psi) = -\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] + KL(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z})),$$

where the first term is the reconstruction loss and the second term is the prior loss. Show that when $p_\theta(\mathbf{x}|\mathbf{z})$ is assumed to be an isotropic Gaussian with mean $D_\theta(\mathbf{z})$ and variance 1, i.e., $\mathcal{N}(D_\theta(\mathbf{z}), I)$, minimizing the reconstruction loss is equivalent to minimizing the average Euclidean distance:

$$\arg\min_\theta -\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] = \arg\min_\theta \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\|\mathbf{x} - D_\theta(\mathbf{z})\|_2^2\right].$$

This tells us that we are minimizing the pixel distance between the image from the dataset and the reconstructed image from the decoder D_θ. Naturally, the more "similar" the images are, the closer they should be! However, using Euclidean distance can be undesirable when dealing with high-dimensional data like images—this is known as the "curse of dimensionality".

1.4 Curse of Dimensionality

In Question 1.3 we introduced the curse of dimensionality, a phenomenon where high-dimensional data can be more difficult to learn from. Let's investigate the curse of dimensionality using a simple example—here we will consider inspecting data in a hypersphere. A hypersphere is a generalization of the concept of a sphere (not just 3 dimensional). Consider a d dimensional hypersphere with radius r. Its (hyper)volume is

$$V_d(r) = \begin{cases} \frac{\pi^k}{k!} r^d, & d = 2k \\ \frac{2(k!)(4\pi^k)}{(2k+1)!} r^d, & d = 2k+1 \end{cases}.$$

The fraction of its hypervolume lying between values $r - c$ and r, where $0 < c < r$ is given by

$$f = 1 - \left(1 - \frac{c}{r}\right)^d.$$

1. For any fixed c, f tends to 1 as $d \to \infty$. Show this numerically with $\frac{c}{r} = 0.01$, for $d = 2, 10, 500$.
2. For any fixed r, evaluate the fraction of the hypervolume which lies inside the radius $\frac{r}{2}$ for $d = 2, 10, 500$.
3. Suppose we sample points according to a uniform distribution inside a very high-dimensional hypersphere centered at the origin, where in the hypersphere would we most likely find the sampled points? Why is this a potential problem for VAE with the Gaussian assumption? Does this argument apply to generative adversarial network (GAN)?

2 Programming Component

2.1 Python Packages and Imports

You will need to install:

```
$ pip install numpy
$ pip install torch
$ pip install torchvision
$ pip install matplotlib
```

We have already included all the necessary Python packages for this assignment—you are not allowed to import any other packages.

2.2 Tasks

In this assignment you have two tasks: implement a better objective for adversarial learning and implement parts of a vision transformer (ViT). The former will investigate a common problem in training GANs using the original objective, whereas the latter will provide you hands-on experience on implementing modern architectures.

Files you will edit:

- `train.py`: The training pipeline—this includes two adversarial learning objectives.
- `np_attention.py`: A multi-head attention layer implemented with NumPy.
- `vit.py`: A GAN model with a ViT.

Files you should read but can ignore:

- `main.py`: The entrypoint for training deep-learning models.
- `mlp.py`: A GAN model with only MLPs.
- `test_train.py`: A simple test for Wasserstein GAN.
- `test_vit.py`: A simple test for ViT.

To run the code, you may simply execute `python main.py`. Here, for every 1000 gradient steps, the code will generate a PDF that displays 64 GAN-generated images.

Implementing Wasserstein GAN with Gradient Penalty. Models trained with the original GAN objective are known to suffer from **mode collapse**, where the trained model fails to capture the full diversity of the data distribution. This is because the generator can solely generate one specific class—for example we can see in Figure 1, left, that GAN generates the digit "1" more often.

Your goal is to resolve this issue by implementing Wasserstein GAN (Arjovsky, Chintala, and Bottou, 2017) with gradient penalty. Let's reconsider Question 1.2 in the written component. What we showed there implies that optimizing the original GAN objective is equivalent to optimizing the Jensen-Shannon (JS) divergence between the real-data distribution p_x and the data distribution p_G induced by generator G:

$$JS(p_x \| p_G) = \frac{1}{2} KL\left(p_x \middle\| \frac{p_x + p_G}{2}\right) + \frac{1}{2} KL\left(p_G \middle\| \frac{p_x + p_G}{2}\right).$$

We can replace the JS divergence with another distance, e.g., the Wasserstein distance, which can be written as

$$\mathcal{W}(p_x, p_G) = \sup_{\|D\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p_x}\left[D(\mathbf{x})\right] - \mathbb{E}_{\mathbf{x}' \sim p_G}\left[D(\mathbf{x}')\right],$$

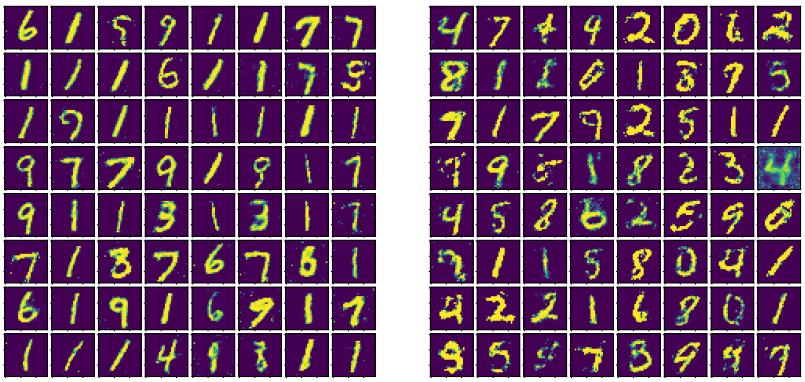

Figure 1: MNIST images generated by GAN **(left)** and Wasserstein GAN **(right)**. Both models are trained with 40000 gradient steps. You should be able to reproduce these results after implementing the Wasserstein GAN objective with gradient penalty, setting the penalty coefficient $\lambda = 10$.

where $\mathbf{x} \sim p_x$ is real data and $\mathbf{x}' \sim G(\mathbf{z})$ is generated data through mapping a noise vector $\mathbf{z}$ using generator G, $\|D\|_L \leq 1$ means that the function D is 1-Lipschitz continuous, i.e., $|D(\mathbf{u}) - D(\mathbf{v})| \leq \|\mathbf{u} - \mathbf{v}\|$. This results in Wasserstein GAN (WGAN) that alleviates mode collapse. In practice, the objectives for training the discriminator (called critic in WGAN) and the generator are

$$\mathcal{L}_D(\phi) = \frac{1}{N}\sum_{n=1}^{N} D_\phi(\mathbf{x}_n) - \frac{1}{N}\sum_{n=1}^{N} D_\phi(G_\theta(\mathbf{z}_n)),$$
$$\mathcal{L}_G(\theta) = \frac{1}{N}\sum_{n=1}^{N} D_\phi(G_\theta(\mathbf{z}_n)),$$

both of which are aimed to be minimized here. In `train.py`, finish the `train_step_wgan` method by implementing the two objectives.

Now, you might be asking how to ensure that D is 1-Lipschitz—we will achieve this through a technique called **gradient penalty**:

$$\mathcal{L}_{GP}(\phi) = \mathbb{E}_{\hat{\mathbf{x}}}\left[(\|\nabla_{\hat{\mathbf{x}}} D_\phi(\hat{\mathbf{x}})\|_2 - 1)^2\right] \approx \frac{1}{N}\sum_{n=1}^{N}(\|\nabla_{\hat{\mathbf{x}}} D_\phi(\hat{\mathbf{x}})\|_2 - 1)^2,$$

where $\hat{\mathbf{x}} = \varepsilon \mathbf{x}_{real} + (1-\varepsilon)\mathbf{x}_{fake}$ is sampled from $\mathbf{x}_{real} \sim p_x$ and $\mathbf{x}_{fake} \sim p_G$, with $\varepsilon \sim \mathcal{U}[0,1]$. Then, the complete training objective for the critic is $\mathcal{L}_D(\phi) + \lambda \mathcal{L}_{GP}(\phi)$, where $\lambda > 0$ controls how strongly D has to be 1-Lipschitz. Complete the `compute_gradient_penalty` method in `train.py` by implementing the gradient penalty objective. You may use the `--gradient_penalty` flag to adjust λ.

Implementing Vision Transformer (ViT) for GAN. You might find the results from MLP GAN to be quite lacking. We know from tutorial that an MLP for classifying MNIST is actually quite good, so the problem is likely the MLP generator (you can verify this yourself)! To this end, we will implement a more modern architecture for the generator, in particular a vision transformer (ViT). Figure 2 provides the high-level architecture of ViT.

Before we implement ViT, let's get a better understanding of the multi-head attention mechanism. Recall that from Vaswani et al. (2017), attention is defined as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where the queries Q and keys K are of dimension d_k, and the values V is of dimension d_v. Here, note in general that the number of keys and values should match while the number of queries can vary as the names suggest. Intuitively,

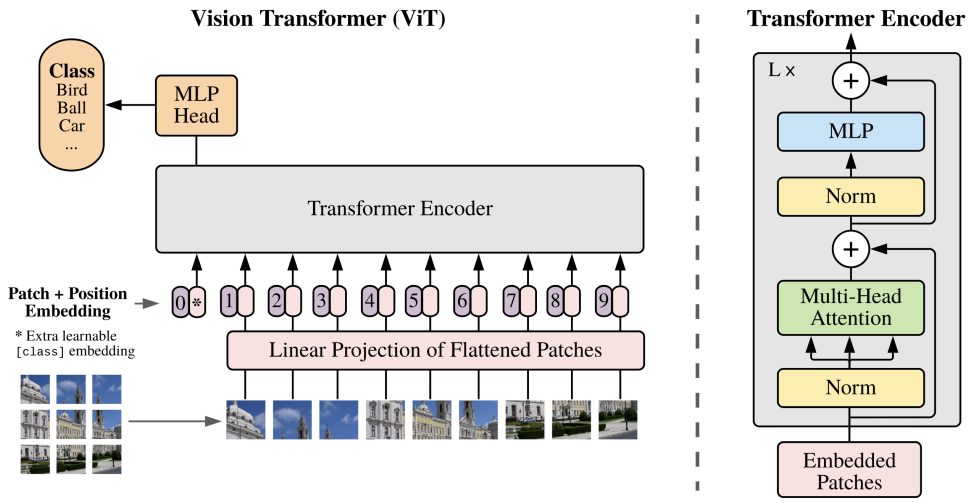

Figure 2: Vision transformer (ViT) architecture from Dosovitskiy et al. (2021).

the inner product $QK^\top$ finds the similarity between the queries and keys that is then weighted based on a softmax function. Finally, we take the combination of the values weighted by the softmax output. This particular attention is also known as a **softmax attention** the weights are generated by a softmax—we can in fact replace this with other non-linearities, a common alternative is using a ReLU function. One can show that using softmax attention in decoder-architecture encourages position bias while ReLU attention does not (Wu et al., 2025).

Vaswani et al. (2017) have also introduced the multi-head attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W^O,$$

where $\text{head}_h = \text{Attention}(QW_h^Q, KW_h^K, VW_h^V)$ is the h'th attention head with query, key, and value weight matrices W_h^Q, W_h^K, W_h^V respectively, and W^O is the output weight matrix. Here, suppose that the model takes in a d-dimensional query, key, and value. W_h^Q and W_h^K are $\mathbb{R}^{d\times d_k}$ weight matrices, W_h^V is a $\mathbb{R}^{d\times d_v}$ weight matrix, and W^O is a $\mathbb{R}^{Hd_v\times d}$ weight matrix, where d is divisible by d_k. This mechanism enables multiple representations, so that each attention head can focus on different positions. This is done by having separate query, key, and value weight matrices for each attention head, projecting the queries, keys, and values into a different subspace. Complete `np_attention.py` by implementing your own multi-head attention. For simplicity, you can assume that $d_k = d_v = d/H$ and H divides d.

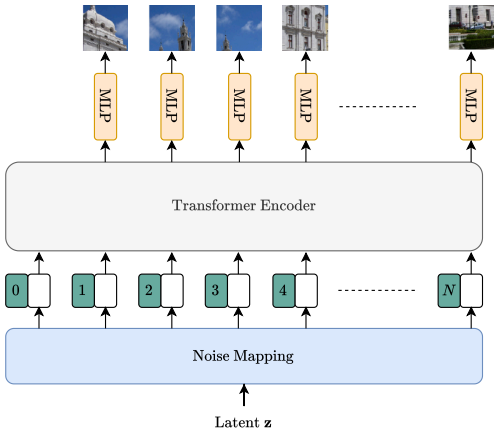

Figure 3: ViT architecture for generating images from latent $\mathbf{z}$.

Now that you are familiar with the attention mechanism, let's move onto implementing ViT. As we can see in Figure 2, ViT is originally presented for classification but this is not a problem as we can just modify the inputs and outputs. Figure 3 is a modification of ViT to generate images from a latent vector $\mathbf{z}$. Here, we sample a latent vector from a standard Gaussian, i.e., $\mathbf{z} \sim \mathcal{N}(0, I)$, and pass it through a noise mapping which linearly transforms the latent vector into an embedding. We encode positional information to the embedding and pass it through a transformer encoder. Finally, to obtain the images, we project the embedding back into an image patch using a MLP. In `vit.py`, complete the `forward` methods in both `TransformerEncoder` and `VisionTransformer` classes, and train a ViT-based model using the `--model` argument. Hopefully your ViT generates better-quality images!

Grading. We will place one point for each of the method implementations, two from `train.py` and two from `vit.py`, as well as two points for `np_attention.py`, a total of six points. You may test your code using the provided tests.

3 Submission Instructions

For the written component, please submit `A4.pdf`. For the programming component, please submit only `train.py`, `np_attention.py`, and `vit.py`.

References

Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." International conference on machine learning. PMLR, 2017.

Radford, Alec, et al. "Language models are unsupervised multitask learners." OpenAI blog 1.8 (2019): 9.

Dosovitskiy, Alexey, et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." International Conference on Learning Representations.

Xiong, Ruibin, et al. "On layer normalization in the transformer architecture." International conference on machine learning. PMLR, 2020.

Lyle, Clare, et al. "Disentangling the causes of plasticity loss in neural networks." arXiv preprint arXiv:2402.18762 (2024).

Wu, Xinyi, et al. "On the Emergence of Position Bias in Transformers." arXiv preprint arXiv:2502.01951 (2025).