HOMEWORK 4
MULTI-MODAL FOUNDATION MODELS *

10-423/623/723 GENERATIVE AI
https://www.mlcourse.org/10423

OUT: October 24, 2026
DUE: October 31, 2026
TAs: Joshua, Alan, Raj

Instructions

- **Collaboration Policy**: Please read the collaboration policy in the syllabus.
- **Late Submission Policy:** See the late submission policy in the syllabus.
- **Submitting your work:** You will use PhiGrade to submit answers to all questions and code.
  - **Written:** You will submit your completed homework as a PDF to PhiGrade. Please use the provided template. Submissions can be handwritten, but must be clearly legible; otherwise, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each answer should be within the box provided. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader and there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score).
  - **Programming:** You will submit your code for programming questions by running the PhiGrade autograder on your own machine. We will examine your code by hand and may award marks for its submission.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Question	Points
LaTeX Template Alignment	0
Latent Diffusion Model (LDM)	7
VQ-VAEs	8
CLIP	4
Vision-Language Models	14
Programming: Text-to-Image Generation	48
Code Upload	0
Collaboration Questions	2
Total:	83

The **autograded code adds 28 points** not shown above, for a total of 111, so the 50% threshold is 56 points.

*Compiled on Saturday 24th October, 2026 at 22:46

1 LaTeX Template Alignment (0 points)

1.1. (0 points) **Select one:** Did you use LaTeX for the entire written portion of this homework?

- ◯ Yes
- ◯ No

1.2. (0 points) **Select one:** I have ensured that my final submission is aligned with the original template given to me in the handout file and that I haven't deleted or resized any items or made any other modifications which will result in a misaligned template. I understand that incorrectly responding yes to this question will result in a penalty equivalent to 2% of the points on this assignment.
Note: Failing to answer this question will not exempt you from the 2% misalignment penalty.

- ◯ Yes

2 Latent Diffusion Model (LDM) (7 points)

2.1. (1 point) **Math:** A latent diffusion model uses a compression factor of $f = 8$ to downsample. If an input image is 512×512 pixels, what are the dimensions of the latent representation of this image?

2.2. (2 points) **Short answer:** Why does a latent diffusion model run diffusion in a latent space instead of pixel space?

2.3. **Short answer:** Standard cross-attention for a diffusion-based text-to-image model defines the queries $\mathbf{Q}$ as a function of the pixels (or latent space) $\mathbf{Y} \in \mathbb{R}^{m \times d_y}$, and the keys $\mathbf{K}$ and values $\mathbf{V}$ as a function of the text encoder output $\mathbf{X} \in \mathbb{R}^{n \times d_x}$.

$$\mathbf{Q} = \mathbf{Y}\mathbf{W}_q, \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_k, \qquad \mathbf{V} = \mathbf{X}\mathbf{W}_v$$

(where $\mathbf{W}_q \in \mathbb{R}^{d_y \times d}$ and $\mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{d_x \times d}$) and then applies standard attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^T/\sqrt{d})\mathbf{V}$$

Now, suppose you instead defined a new formulation where the values are a function of the pixels (or latent space): $\mathbf{V} = \mathbf{Y}\mathbf{W}_v$ where $\mathbf{W}_v \in \mathbb{R}^{d_y \times d}$.

2.3.a. (2 points) What goes wrong mathematically in the new formulation?

2.3.b. (2 points) Intuitively, why doesn't the new formulation make sense? Briefly begin with an explanation of what the original formulation of cross-attention is trying to accomplish for a single query vector, and why this new formulation fails to accomplish that.

3 VQ-VAEs (8 points)

3.1. The objective function for a VQ-VAE contains two terms in addition to the reconstruction loss term:

- The vector-quantization loss, $\|\text{sg}\left[z_e(x)\right] - e\|_2^2$ and
- The "commitment" loss, $\beta\|z_e(x) - \text{sg}\left[e\right]\|_2^2$

where sg is the stopgradient operator and e is the latent embedding vector closest to the output of the encoder $z_e(x)$.[1]

In this question, you will examine the impact of these terms and the stopgradient operator. Let

$$\mathcal{L} = \|\text{sg}\left[z_e(x)\right] - e\|_2^2 + \beta\|z_e(x) - \text{sg}\left[e\right]\|_2^2 \tag{1}$$

$$\tilde{\mathcal{L}} = \|z_e(x) - e\|_2^2 + \beta\|z_e(x) - e\|_2^2 \tag{2}$$

3.1.a. (1 point) **Math:** What is the gradient of $\mathcal{L}$ with respect to $z_e(x)$?

3.1.b. (1 point) **Math:** What is the gradient of $\mathcal{L}$ with respect to e?

3.1.c. (1 point) **Math:** What is the gradient of $\tilde{\mathcal{L}}$ with respect to $z_e(x)$?

3.1.d. (1 point) **Math:** What is the gradient of $\tilde{\mathcal{L}}$ with respect to e?

3.1.e. (2 points) **Short answer:** Given your findings from the previous parts, how would you describe the impact of the stopgradient operator on the optimization of these two terms? How do the gradients with and without the stopgradient operator differ?

[1] For the purposes of this question, we are still assuming that the encoder outputs a single vector; in practice, a true VQ-VAE encoder would output multiple vectors and these terms in the objective function would be sums over these vectors.

3.2. (2 points) In a VQ-VAE, the codebook embeddings e need to be learned during training. Recall that the VQ-VAE objective includes the vector-quantization loss $\|\text{sg}\left[z_e(x)\right] - e\|_2^2$, where $z_e(x)$ is the encoder output, e is the nearest codebook embedding, and $\text{sg}\left[\cdot\right]$ is the stop gradient operator. Using your gradient with respect to e from part (a), describe in words what the codebook embeddings are being optimized to do.

4 CLIP (4 points)

4.1. (4 points) **Short answer:** In Radford et al. (2021), the original CLIP paper, they pre-trained the image and text encoders using a *symmetric cross-entropy loss*. Formally, given a mini-batch of N (image, caption) pairs, let $e_i^{(n)}$ be the image embedding of the nth image normalized to have unit norm and let $e_t^{(n)}$ be the text embedding of the nth caption, also normalized to have unit norm.

The symmetric cross-entropy loss consists of two terms for each (image, caption) pair, a term that compares the image embedding to all N caption embeddings and a term that compares the caption embedding to all N image embeddings. Formally, let

$$\ell_{i\to t}^{(n)} = -\log\left(\frac{\exp\left(e_i^{(n)}\cdot e_t^{(n)}\right)}{\sum_{m=1}^{N}\exp\left(e_i^{(n)}\cdot e_t^{(m)}\right)}\right) \tag{3}$$

$$\ell_{t\to i}^{(n)} = -\log\left(\frac{\exp\left(e_i^{(n)}\cdot e_t^{(n)}\right)}{\sum_{m=1}^{N}\exp\left(e_i^{(m)}\cdot e_t^{(n)}\right)}\right) \tag{4}$$

where $\cdot$ is the dot-product operation between two (same-length) vectors. The CLIP objective can then be expressed as

$$\ell = \frac{1}{N}\sum_{n=1}^{N}\frac{\ell_{i\to t}^{(n)} + \ell_{t\to i}^{(n)}}{2} \tag{5}$$

Show that minimizing this objective function is equivalent to jointly maximizing the cosine similarity of an image embedding and its corresponding caption embedding while minimizing the cosine similarity of all other pairs of image and caption embeddings.

Hint: *It may be helpful to start with* $\min(\ell)$*, and follow through with the mathematical logic until you show that it evaluates to* $\max_n\{e_i^{(n)}\cdot e_t^{(n)}\}$ *and* $\min_{n\neq m}\{e_i^{(n)}\cdot e_t^{(m)}, e_i^{(m)}\cdot e_t^{(n)}\}$.

5 Vision-Language Models (14 points)

This section examines the architecture of a modern open vision-language model (VLM), **Qwen2.5-VL**. We encourage you to read the Qwen2.5-VL Technical Report (Bai et al., 2025), in particular Section 2.1 on the model architecture. Like the PaliGemma family discussed in lecture, Qwen2.5-VL turns an image into a sequence of "visual tokens" that a decoder-only language model reads alongside text tokens. The questions below introduce each of its components, and how they differ from PaliGemma 2, in turn.

5.1. (2 points) **Architecture overview.** Figure 1 shows the high-level architecture of Qwen2.5-VL for a single $H \times W$ input image and a text prompt.

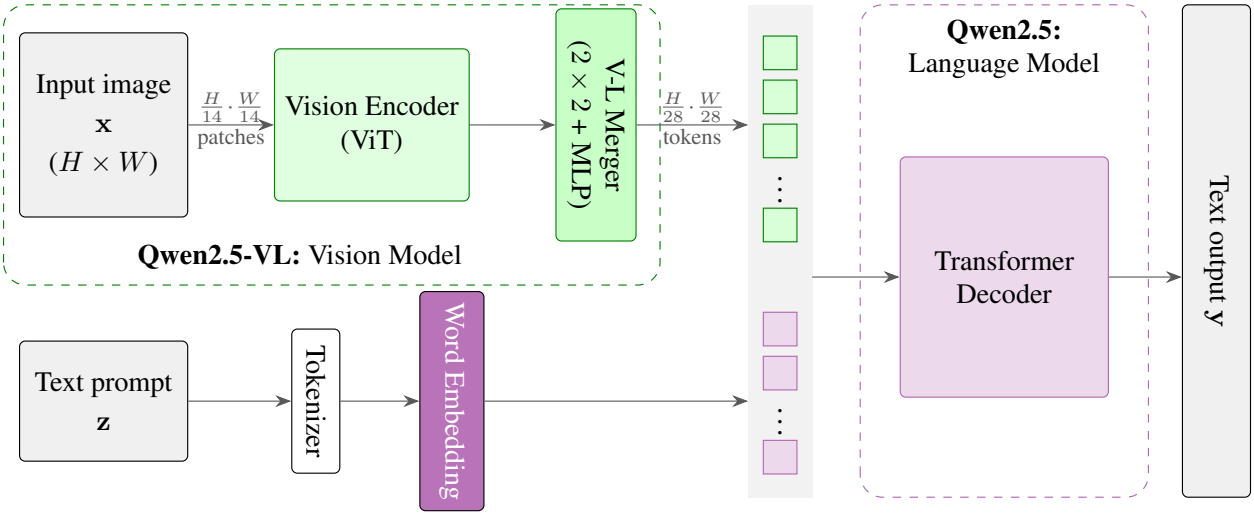

Figure 1: High-level architecture of Qwen2.5-VL. The visual tokens (green) and the text prompt tokens (purple) form one input sequence to the language model, which generates the text output autoregressively.

Short answer: Which component in Figure 1 is the most computationally expensive at inference time? Justify your answer.

5.2. **Vision-language merger.** Qwen2.5-VL's vision encoder outputs one feature vector per 14×14 patch. The merger then concatenates the features of each group of 2×2 spatially adjacent patches and passes them through a two-layer MLP, producing *one* visual token (in the language model's embedding space) per group.

5.2.a. (1 point) **Short answer:** In 1–2 sentences, describe the role of the vision-language merger.

5.2.b. (1 point) **Short answer:** In 1–2 sentences, explain why the merger combines 2×2 groups of patches into a single token instead of passing every patch to the language model as its own token.

5.2.c. (2 points) **Short answer:** A 28×28-pixel patch covers the same area as a 2×2 group of 14×14-pixel patches, so using 28×28 patches in the ViT would give the language model the same number of visual tokens. Why might it be better to use 14×14 patches and merge *after* the ViT, rather than using 28×28 patches from the start?

5.2.d. (3 points) **Math:** How many visual tokens does Qwen2.5-VL feed to its language model for (i) a 448×448 image, and (ii) a 1344×896 image? Assume neither image is resized (both side lengths are already multiples of 28). Compare this against PaliGemma 2 at 448×448 resolution (patch size 14×14, no token merging), which produces 1024 visual tokens. Show your work.

5.3. **Vision encoder: Qwen2.5-VL vs. PaliGemma 2.** The two models turn pixels into patch features in different ways (see Section 2.1 and Table 1 of the Qwen2.5-VL report):

- **PaliGemma 2** uses a SigLIP ViT with 14×14 patches. Every image is first *resized to a fixed square resolution* chosen when the model is trained (224^2, 448^2, or 896^2), so a given checkpoint always sees the same patch grid and always produces the same number of visual tokens (256, 1024, or 4096, one per patch). Patch positions come from a *learned position-embedding table* sized for that grid, and *every* ViT layer uses full self-attention over all patches.
- **Qwen2.5-VL** also uses a Vision Transformer (ViT) that splits the image into 14×14-pixel patches and outputs one feature vector per patch, but it processes images at (approximately) their *native resolution and aspect ratio*: the height and width are only rounded to multiples of 28 pixels, so the size of the patch grid, and therefore the number of visual tokens, depends on the image.

5.3.a. (1 point) **Short answer:** Give one advantage of processing images at their native resolution and aspect ratio, compared to resizing every image to a fixed square resolution.

5.3.b. (2 points) **Short answer:** At native resolution, the number of patches (and visual tokens) grows linearly with the image's area, i.e. its number of pixels. What is the disadvantage of this for the vision encoder, and what architectural choice does Qwen2.5-VL use to combat it?

5.4. In its language model, Qwen2.5-VL uses *Multimodal RoPE* (M-RoPE), which gives every token a (temporal, height, width) position instead of a single integer position.

5.4.a. (1 point) **Short answer:** Why can't Qwen2.5-VL use a learned position-embedding table sized for the patch grid, as PaliGemma 2 does?

5.4.b. (1 point) **Short answer:** What are the benefits of using M-RoPE over traditional 1D RoPE?

6 Programming: Text-to-Image Generation (48 points)

Introduction

Text-to-image generation has revolutionized how we create visual content, with models like DALL-E, Stable Diffusion, and Midjourney demonstrating remarkable capabilities. These large-scale models require extensive computational resources and massive datasets. Such models have moved us closer to the objective of being able to combine the knowledge of an LLM with the visual quality of a image generation model for a chat-like generation experience. In this assignment, you'll explore the fundamental principles of LLM-conditioned image generation by combining a pretrained LLM and class-conditional diffusion model into a single text-to-image generation model.

The core challenge in text-to-image generation is bridging the semantic gap between language and vision. While diffusion models excel at generating high-quality images and language models capture rich semantic information, connecting these modalities efficiently remains non-trivial. In this assignment, you will train a **Q-Former** (Query Former) architecture proposed by Perciever IO, and used to connect text and image features in BLIP-2 (Li et al., 2023) and MetaQueries. The Q-Former uses a fixed number of learnable query tokens to extract relevant visual features from arbitrary length language model representations.

Your will:

- Train a Q-Former to use language model text embeddings from GPT-2 as conditioning for a pretrained diffusion model with a Diffusion Transformer (DiT) backbone, while keeping both GPT-2 and the DiT model weights frozen.
- Generate 32×32 images from simple text prompts like "fluffy cat" or "red airplane".
- Examine cross-attention maps to determine what information the Q-Former is extracting from the pretrained LLM.

Note: CIFAR-10 images are only 32×32 pixels, so generated images may appear blurry or lack fine details. This is expected due to the low resolution, not a sign of poor model performance. We highly recommend that you take a look at CIFAR-10 for reference.

Background

The Text-to-Image Pipeline:

Our goal is to generate images from text descriptions. We have three components:

1. **GPT-2 (Frozen)**: A pretrained language model that converts text into semantic feature vectors
2. **DiT Diffusion Model (Frozen)**: A pretrained class-conditional diffusion transformer trained on the CIFAR dataset
3. **Q-Former (Trained)**: A lightweight adapter that translates language features into visual conditioning, allowing us to provide precise captions for cifar-like images, and extrapolate beyond class-based generation

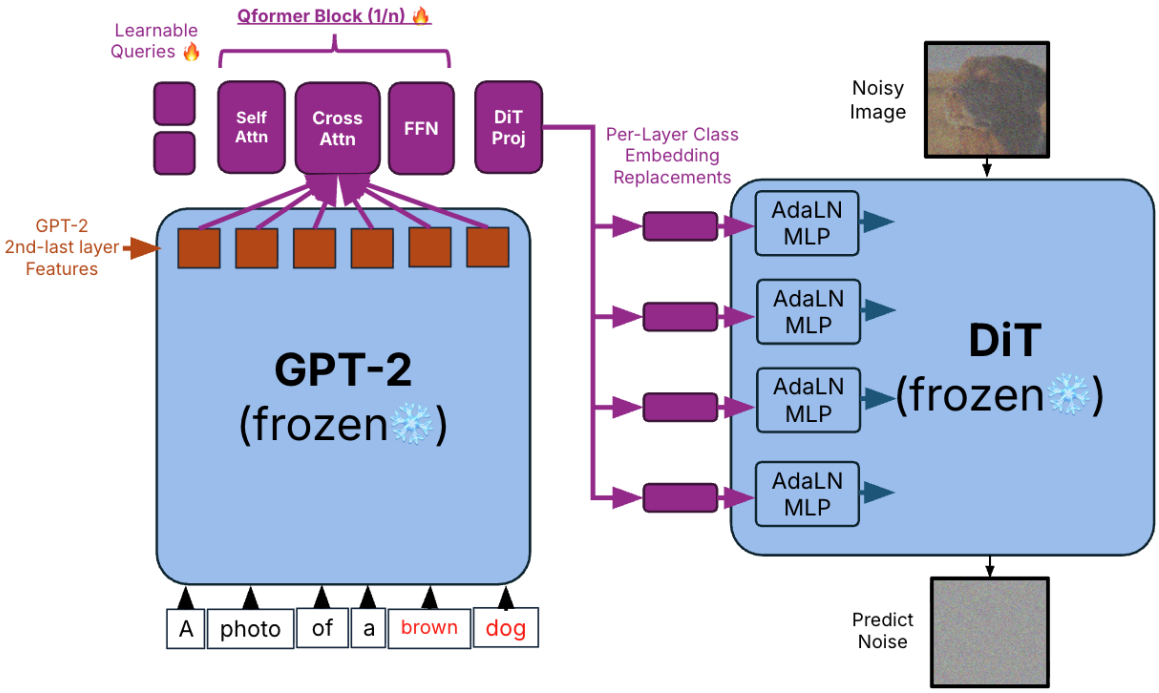

Figure 2: Our Q-Former architecture for text-to-image generation.

First, descriptive text is tokenized and passed through GPT-2 to produce **per-token text features** (bottom-left, white boxes up to orange layer). Note that, we actually use the **2nd-to-last** layer's embedding instead of the final layer embeddings for performance reasons. Next, we initialize **learnable query embeddings**, and transform them with transformer blocks that perform self-attention, cross-attention with the **per-token text features**, and a per-token feed-forward network (FFN) (top-left, orange layer to purple blocks). Then, a DiT projector outputs a set of **class embedding features** (middle, purple block). During the forward pass of the Diffusion Transformer (DiT), the class embedding vector for the current image class is replaced with the **per-layer class embedding features** at every layer (from purple middle and top-right noisy image to bottom-right predicted noise).

During training, only the purple layers are trained, while all other weights are frozen.

Why Q-Former?

Why don't we use GPT-2 features directly as conditioning for the diffusion model? Here are some challenges:

- GPT-2 outputs a variable length sequence of 768-dimensional vectors, but DiT expects a single 256-dimensional conditioning vector per layer.
- GPT-2 features live in "language space" while DiT needs "visual conditioning space," and we would not expect these representations to be aligned in any sense.

The Q-Former solves this by learning to extract relevant information from language features and projecting it into the format DiT expects.

Diffusion Models for Image Generation:

Denoising Diffusion Probabilistic Models (DDPMs) (Ho et al., 2020) generate images by iteratively denoising random Gaussian noise. The model learns to predict and remove noise step-by-step:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t, c)\right) + \sigma_t z$$

where ϵ_θ is the noise prediction network (DiT), c is the conditioning information, and α_t are noise schedule parameters.

For this assignment, we use a pretrained Diffusion Transformer (DiT) trained on CIFAR-10 with class label conditioning. The sampling process is already implemented for you. To perform the conditioning, the pretrained model has a learned class embedding for each of the 10 classes, which is used to modulate each transformer layer via an Adaptive LayerNorm. This takes the class embedding as input and outputs LayerNorm scale and shift parameters:

$$x = (\text{AdALN}(\mathbf{F}_{\text{class embedding}}))(x)$$

For more details, see the Adaptive LayerNorm section of the original Diffusion Transformer paper, or the `adaLN_modulation` function in `dit.py`.

Classifier-Free Guidance:

To improve prompt adherence, the provided code uses classifier-free guidance (Ho & Salimans, 2022), which amplifies the effect of conditioning:

$$\tilde{\epsilon}_\theta(x_t, t, c) = \epsilon_\theta(x_t, t, \emptyset) + w \cdot (\epsilon_\theta(x_t, t, c) - \epsilon_\theta(x_t, t, \emptyset))$$

where w is the guidance scale. During training, we randomly drop text conditioning with probability 0.1 to enable this technique.

Q-Former Architecture:

The Q-Former uses learnable query tokens that extract information from text features through attention mechanisms:

- **Learnable Queries**: A small set (2-8) of learned embedding vectors that act as "questions" to ask the language model
- **Self-Attention**: Queries interact with each other to build coherent representations
- **Cross-Attention**: Queries attend to GPT-2 hidden states to extract semantic information relevant for image generation
- **Feed-Forward Networks**: Process the attended features
- **Output Projection**: Project to the conditioning dimension expected by DiT

The Q-Former processes text features $\mathbf{F}_{\text{text}} \in \mathbb{R}^{B\times S\times 768}$ through multiple layers:

$$\mathbf{Q}^{(0)} = \text{LearnedEmbedding}(\text{query_ids})$$
$$\mathbf{Q}^{(l)} = \mathbf{Q}^{(l-1)} + \text{SelfAttn}(\text{LN}(\mathbf{Q}^{(l-1)}))$$

$$\mathbf{Q}^{(l)} = \mathbf{Q}^{(l)} + \text{CrossAttn}(\text{LN}(\mathbf{Q}^{(l)}), \mathbf{F}_{\text{text}})$$
$$\mathbf{Q}^{(l)} = \mathbf{Q}^{(l)} + \text{FFN}(\text{LN}(\mathbf{Q}^{(l)}))$$

where LN denotes LayerNorm. The final queries are projected to produce per-layer conditioning vectors for the DiT model. See Figure 2 for details.

Starter Code

The files are organized as follows:

```
handout/
    requirements.txt
    image_caption_data.py
    train_qformer.py
    eval_qformer.py
    inference.yaml
    run_in_cloud.ipynb
    dit.py
    ddpm.py
    attention_visualization.py
    download_data.sh
    test_all.py
    phigrade.yaml
    pytest.ini
```

The codebase is partially built off of this open source diffusion implementation. We will use DDPM and the CIFAR Dataset for our implementation.

Below is a description of what you will find in each file. Functions are already implemented and provided for you, unless specified otherwise with TO DO.

1. `image_caption_data.py`: Functions for the image/caption data and building/caching LLM embeddings
    - `get_cifar_names`: Gets CIFAR class names
    - `get_synonyms`: Gets synonyms for CIFAR class names, for data augmentation
    - `get_text_captions`: Retrieves text captions from cached text caption path
    - `class PromptVariantDataset`: Dataset for Stage 1 training using CIFAR to select a single prompt variant per sample index. Class methods include`_maybe_apply_synonym` which possibly swaps out class names for synonyms.
    - `class HiddenStateDataset`: Dataset for Stage 2 training using an instance of the `PromptVariantDataset`.
    - `class HiddenStateCache`: Cache of GPT-2 hidden states of prompt text embeddings for instance of `HiddenStateDataset`. The cache is built in memory when the `HiddenStateDataset` is created, by calling your `prompts_to_padded_hidden_states` on every training prompt.

- `prompts_to_padded_hidden_states`: **TODO:** Function to turn text into GPT-2 embeddings

2. `train_qformer.py`: Training script
   - `setup_optimizer_and_scheduler`: **TODO:** Given a model and a set of arguments, set up the optimizer, scheduler, and loss criterion
   - `maybe_load_resume_states`: Allows for restarting from a given optimizer and weight state in the case of disconnection
   - `run_qformer_inference`: General inference function
   - `train_cifar10_qformer_dense_captions`: Main training loop
3. `eval_qformer.py`: Evaluation script
   - `eval_cifar10_qformer`: Main inference code
4. `run_in_cloud.ipynb`: Cloud based ipynb task that has scripts to run training/inference
5. `dit.py`: Model architecture
   - `QueryEmbedder`: **TODO:** Q-Former implementation (init is provided for you)
   - `DiT_Llama`: Pretrained diffusion transformer, and associated classes used within
   - `Attention`: Multi-head attention with cross-attention support
   - `FeedForward`: FFN with SwiGLU activation
6. `ddpm.py`: DDPM pipeline
   - `DDPM`: forward diffusion and sampling of DDPM
   - `ddpm_schedules`: schedules for DDPM sampling
7. `attention_visualization.py`: Visualization of cross-attention maps
8. `download_data.sh`: Script to download data files
   TODO: You should download the two items below by running this shell script. There is a cell in `run_in_cloud.ipynb` that completes this for you.
   - Pretrained DiT weights: A DiT with our architecture, trained on the class-conditional CIFAR dataset task
   - Dense CIFAR text captions: A dataset of dense CIFAR captions for enhancing text to image training
9. `requirements.txt`: A list of packages that need to be installed for this homework, including `pytest` and the `phigrade` client that you need in order to submit. Install them by running `pip install -r requirements.txt` from the handout directory.
10. `test_all.py`: The unit tests that make up the autograder. Run them by executing `pytest test_all.py` from the handout directory. Doing so both scores your code and submits it.
11. `phigrade.yaml` and `pytest.ini`: Configuration for the autograder.

(We also included some IDE specific settings files {`.noai`, `.vscode`, `.zed`} to help you avoid AI use during Slot A.)

Implementation

You will implement three key functions that enable Q-Former training and help you understand what it learns:

1. `prompts_to_padded_hidden_states`: Extract features from a pretrained LLM
2. `setup_optimizer_and_scheduler`: Configure which parameters to train and how
3. `QueryEmbedder.forward`: Implement Q-Former to process text features

All other components (cross attention, training loop, inference) are provided. Your focus is on understanding how efficient fine-tuning works and analyzing the learned attention patterns.

Part 1: Extract LLM Features from any layer of GPT-2

Implement `prompts_to_padded_hidden_states` in `image_caption_data.py`.

```
def prompts_to_padded_hidden_states(
    prompts: List[str],
    gpt2,
    tokenizer,
    gpt2_layer_index: int,
    device: torch.device,
):
    """
    Convert a list of text prompts to a single padded tensor
    of GPT-2 hidden states.

    Returns the hidden states from that specific layer along
    with masks as a tuple (hidden_states, masks)
    """
```

Your task:

- For all prompts, do the following:
  - Call the tokenizer with `return_tensors="pt"` to get valid inputs for GPT-2
  - Call GPT-2 on these inputs, with `output_hidden_states` set to True. You can call `outputs.hidden_states` after this to get the hidden states
  - Given these hidden states, use the `gpt2_layer_index` to determine exactly which layer of the hidden states to keep for this prompt. **NOTE:** the layer index is the layer *after* the current hidden states, so layer index 0 returns the states before the layer 0 weights, and layer index N for an N layer model is the embeddings after the last layer, which is allowed.
- After you have a list of the kept hidden states for all prompts, perform max-length padding of the hidden states, so that the final hidden state size is

  `(Batch_size, max_token_length_in_batch, gpt2_hidden_dimension)`.

- **NOTE:** Also return a boolean padding tensor of size `(Batch_size, max_seq)` where True indicates that a token is not a padded token
- **NOTE:There are 12 GPT-2 layers, and there is a hidden state before the first layer and after the last layer, so in total there are 13 possible hidden layers. Be sure to accommodate any of these indices in the code.**

Part 2: Setup Optimizer and Scheduler

Implement `setup_optimizer_and_scheduler(model, args)` in `train_qformer.py`.

```
def setup_optimizer_and_scheduler(model, args):
    """
    Configure training: freeze/unfreeze parameters, initialize,
    create optimizer/scheduler.

    Returns: optimizer, scheduler, criterion
    """
```

Step 1: Freeze and Unfreeze Parameters (Parameter-Efficient Training)

The key insight: We want to train only the Q-Former while keeping DiT ($\sim$13M) and GPT-2 ($\sim$124M) frozen. This makes training fast and prevents catastrophic forgetting of pretrained capabilities.

Your task:

- Set `requires_grad = False` for all model parameters, except for those in the query embedder

Step 2: Create Optimizer

Your task:

Create `torch.optim.AdamW` with:

- Only parameters where `requires_grad=True`
- `lr=args.lr`, `weight_decay=args.weight_decay`, `eps=args.adam_epsilon`
- `betas=(0.9, 0.95)`

Step 3: Create Learning Rate Scheduler

We might expect the first few steps of training to be very unstable, as we're replacing the class embedding from a pretrained model with random noise from a randomly initialized Q-Former. We can add warmup steps to smoothly begin training and determine good momentum and RMS parameters for AdamW.

Your task:

If `args.warmup_steps > 0`, create a linear warmup scheduler:

- Use `torch.optim.lr_scheduler.LambdaLR`
- The lambda function should return: `min(1.0, (step + 1) / warmup_steps)`

- This gradually increases learning rate (LR) from 0 to `args.lr` over the first `warmup_steps`
- If `warmup_steps <= 0`, set `scheduler = None`

Why warmup? Warmup gradually ramps up the learning rate to prevent unstable updates before model weights and optimizer statistics are well-initialized.

Step 4: Create Loss and Return

Your task:

- Set criterion to MSE loss
- Return designated parameters.

Part 3: Q-Former Forward

Implement `forward` in `dit.py`.

```
    def forward(self, cross_attention_states, batch_size=1,
    cross_attention_mask=None):
    """
        Forward pass of the Q-Former query embedder.

        Returns: Final per-layer features
        (batch_size, num_dit_layers, conditioning_hidden_size)
    """
```

Your task:

- Get the set of learnable query embeddings from the query table using indices `0` to `num_queries-1`.
- Expand the queries to shape `(batch_size, num_queries, transformer_hidden_size)` and put the queries on the same device as the `cross_attention_states`.
- For each block in `self.blocks`, do the following:
  - Apply the self-attention block (i.e. `x -> x + self_attn(attn_norm(x))`).
  - Apply the cross-attention block (i.e. `x -> x + cross_attn(cross_norm(x), cross_attention_states, cross_attention_mask)`).
  - Apply the feed-forward block (i.e. `x -> x + ffn(ffn_norm(x))`).
  - You are implementing the equations under Q-Former Architecture.
- Use `output_proj` to project the resulting shape to the `conditioning_hidden_size`.
- Map the queries to the per-layer features using the `query_to_layer` linear layer to get a tensor with a shape of `(batch_size, num_dit_layers, conditioning_hidden_size)`.
- Returns per-layer features.

Hint: If you are stuck, look in `__init__()` to see what you have to work with.

Part 4: Training

We train using cached embeddings from GPT-2[2] with the following text prompts:

- simple text prompts with class name: "a photo of a dog"
- simple text prompts with a synonym for class name: "a photo of a four legged animal that barks"
- dense text prompts with class name: "a photo of a black dog in a grassy field"
- dense text prompts with a synonym for class name: "a photo of a four legged animal that barks in a grassy field"

These different captions allow the Qformer to learn how to transform semantic text information into dense conditioning information to the pretrained and frozen DiT. The training loop is provided but you should understand how it works:

Algorithm 1 Q-Former Training (Provided)

Load pretrained DiT: `model.load_state_dict(torch.load(pretrained_path))`
Set up the optimizer, scheduler, and criterion
for each epoch **do**
  for each batch $(x_{\text{img}}, y_{\text{caption}})$ from densely labeled, randomly captioned CIFAR-10 **do**
    Sample timestep: $t \sim \text{Uniform}(1, T)$
    Sample noise: $\epsilon \sim \mathcal{N}(0, I)$
    Add noise: $x_t = \sqrt{\bar{\alpha}_t} x_{\text{img}} + \sqrt{1 - \bar{\alpha}_t}\epsilon$
    Load pre-computed text embedding y_{emb} for text y_{caption})
    or calculate $y_{\text{emb}} = \text{extract_features(gpt2, } y_{\text{caption}})$
    Predict noise: $\epsilon_{\text{pred}} = \text{DiT}(x_t, t, y_{\text{emb}}, \text{text})$
    Compute loss: $\mathcal{L} = \|\epsilon - \epsilon_{\text{pred}}\|^2$
    Backpropagate and update only Q-Former parameters
  Generate sample images to monitor training progress
  Save Q-Former weights

[2] we could have had you generate these, but it would add unnecessary extra time to the already long assignment

Running Training:
See the `run_in_cloud.ipynb` file for the training script. Keep all hyperparameters from these files, but feel free to inspect `train_qformer.py` to see all the arguments that you can use and how they work.

NOTE: If your training gets killed after 5 or more epochs, the script will automatically save both your weights and optimizer states every 5 epochs. Use `--resume_optimizer_path`, `--resume_model_path`, and `--start_epoch` to restart from an epoch that is a multiple of 5.

Recommended: Sign in to Google Colab to claim Colab Pro, which is available when you verify with your CMU email. We strongly advise using the A100 GPU when it's available, but if not, using a L4 or T4 should work as well.

Running Inference:
Using the `run_in_cloud.ipynb` file, run `eval_qformer.py` in Colab to run inference.

NOTE: `inference.yaml` already contains one entry per empirical question (6.6 to 6.8); each entry writes its grid under `./outputs/`. You can add entries of your own to try other prompts.

Understanding the Big Picture

Your Implementation	What It Enables
Freeze/unfreeze parameters	Preservation of pretrained model weights and fast training
Optimizer & scheduler	Stable gradient updates
QFormer forward	Understanding how the QFormer model works

After implementing these functions, the training loop (provided) will:

1. Compute and cache the text embeddings with your `prompts_to_padded_hidden_states`
2. Train only the Q-Former using your optimizer
3. Save samples each epoch for monitoring

Key Insight: You're implementing the "glue code" that makes efficient multi-modal fine-tuning work. The architecture is provided; your job is to configure *how* it learns.

Empirical Questions

6.1. Code Comprehension (Part 1): GPT-2

One of the parameters for the function,`prompts_to_padded_hidden_states`, is a variable called `gpt2`. This is the model we will use to encode the given `prompts`.

6.1.a. (2 points) Let's say you pass in one prompt with N tokens into the `gpt2` language model. Let L be the number of transformer layers in `gpt2`, and h be the length of its hidden vectors.

What would be the dimensions of `output.hidden_states`, with `output` being what `gpt2` returns?

Please be clear what each mode represents.

6.2. Code Comprehension (Part 3): Q-Former Init Function

Take a look at the architecture defined in the `QueryEmbedder.__init__` function, and answer the following questions:

6.2.a. (3 points) Take a look at this line:

```
self.query_table = nn.Embedding(num_queries, transformer_hidden_size)
```

and answer **all** the following questions in the box below.

- What is the shape of the embedding table `query_table.weight`?
- What is the meaning of `query_table`? More specifically, what does each row correspond to?
- What are the expected inputs to this module, and what would it output in return? A description suffices.

6.2.b. Now, take a look at the declaration of `self.blocks`.

Answer the following questions.

6.2.b.i. (3 points) For each of the following modules, describe its function in one sentence.

- `self_attn`
- `cross_attn`
- `ffn_norm`

6.2.b.ii. (2 points) Which input tensors do the keys, values, and queries come from in the self-attention layer? What about in the cross-attention layer?

Hint: It might be helpful to take a look at the `Attention` *class defined in the code file.*

6.2.b.iii. (2 points) In this architecture, normalization is applied before each operation (e.g. `self_attn(attn_norm(x))`). This is known as a "pre-norm" transformer block.

What dimension of the tensor is being normalized in the LayerNorm blocks? What are the advantages of normalizing over this particular dimension?

6.2.b.iv. (2 points) What advantages does a "pre-norm" block bring during training and evaluation (as opposed to not having one)?

Select all that apply.

- ◯ It improves gradient flow through the network, which helps stabilize training for deeper models.
- ◯ It adjusts the initialization of the model parameters and leads to faster convergence.
- ◯ It makes training more stable by preventing activations from growing too large.
- ◯ It adds additional parameters that give the model new capabilities by making it more expressive.
- ◯ None of the above.

6.3. **Q-Former Architecture Understanding + Dataset Dive**

6.3.a. (2 points) Looking at the code, we observe that each of the Q-Former transformer block has 9.4M parameters, with 2 such blocks. Combining that with the number of parameters of the **query embeddings** and **output projections** how many trainable parameters does the Q-Former have given the config in the training script? (round to nearest million).

Suppose the GPT-2 has 124M parameters, and DiT has 13M parameters, what percentage of the weights in the entire process are trained (not frozen) in our method?

6.3.b. (2 points) Paste 3 text-image pairs from the densely captioned CIFAR dataset, with their CIFAR-10 training-set indices. The "Caption-image pairs" cell in `run_in_cloud.ipynb` shows how to look them up.

6.4. Q-Former Training Understanding

Consider our Q-Former setup with the following specifications:

- the DiT is pretrained on a large corpus of high quality images I
- the LLM is pretrained on a large corpus of text T
- the Q-Former training set is on a small set of medium quality images I' with corresponding text T'

We will consider two scenarios:

- Scenario A: Same as in the homework; we only train the Q-Former and keep the LLM and DiT fixed.
- Scenario B: We will unfreeze parameters of the DiT and LLM along with the Q-Former while training.

6.4.a. (2 points) Compare and contrast scenario A and B in terms of final asymptotic loss value when trained continuously. Would you expect one scenario to have a lower asymptotic loss, and if so, which one?

6.4.b. (2 points) Consider running a general text generation validation set (i.e. for a token prediction task) on the LLM used at the end of training in scenarios A and B. Which would you expect to have higher mean token accuracy on this general validation set, and why?

6.5. Q-Former Training

As indicated in `run_in_cloud.ipynb`, train your Q-Former for 25 epochs with `--num_query_tokens 4`. [Expected runtime on T4: 3-4 hours] [Expected runtime on A100: 1 hour]

Note: Since we are working with limited runtimes, we do not expect the quality of the images generated to be very high-resolution.

6.5.a. (2 points) Paste your training Exponential Moving Average (EMA) loss curve for the first 1000 steps.

6.5.b. (3 points) Paste the sample grids logged to wandb for epochs 5, 12, and 24 (wandb numbers epochs from 0, so epoch 24 is the final one). How does image quality/prompt adherance change during training?

6.6. **Classifier-Free Guidance**

Experiment with different CFG scales with the original pretrained model: $w \in \{1.0, 2.0, 4.0, 8.0\}$ for the class "deer" (class 4).

6.6.a. (2 points) Paste the 10 images generated for each of the different guidance scales. How does the appearance change as you increase w? How does the diversity of the set of 10 images change as you increase w? Why does this occur?

6.6.b. (1 point) Based on your results, what seems to be a reasonable guidance scale for this model?

6.7. **Text-Conditioned Generation**

After training, generate images from custom text prompts.

6.7.a. (2 points) Generate 40 images of cars under two conditions with cfg 3.0:

- 20 images with the pretrained class-conditional DiT with class label for automobile
- 20 images with the Qformer DiT with text “a photo of a blue automobile with a bright blue sky”

Paste all 40 images. Is there any noticeable difference between the two sets of images?

6.7.b. (2 points) Compare generation of OOD underspecified prompts. Generate 10 photos with caption:“a photo of a vehicle that goes on grass” and compare with 10 photos with caption: “a photo of a vehicle that goes in water” with cfg 3.0.

Paste the 20 images: How do these images differ?

6.8. **Cross-Attention Maps**

Finally, we will examine the cross-attention maps between the LLM text features and the Q-Former. The Q-Former uses cross-attention to query the text embeddings. By saving these attention patterns, we can visualize which words the model focuses on when generating specific image features.

6.8.a. (6 points) Using the text prompt "a photo of a red airplane with a green field in the background", generate the cross-attention maps of both Q-Former layers (the provided `inference.yaml` entry does this). Paste the two maps and the generated airplane image.

Interpreting the attention maps, what do you notice about what tokens are getting attended to and what are getting completely ignored? Why might this be the case?

6.9. Flow Matching

6.9.a. (2 points) Understanding Flow Matching

As a warmup, conceptually, what does a flow-matching model learn to predict instead of noise? How does this prediction help transform Gaussian noise into an image?

6.9.b. (4 points) Modifying the Training Algorithm

Suppose we replace the DDPM backbone in Algorithm 1 with a pretrained flow-matching DiT, while still training only the Q-Former.

We use the straight-line path

$$x_t = (1-t)\epsilon + t x_{\text{img}}, \quad \epsilon \sim \mathcal{N}(0, I).$$

How would you modify **Lines 5, 7, 10, and 11** of Algorithm 1 to train with flow matching instead of DDPM?

6.9.c. (2 points) Flow Matching Inference

How would you generate an image using a trained flow-matching model? How does this differ from DDPM sampling?

7 Code Upload (0 points)

You submit your code with the PhiGrade autograder, which you run from your own machine. There is no separate "submit" step and no file upload: running the autograder *is* how you submit.

One-time setup The `phigrade` client is included in `requirements.txt`. Create an API key on your account page at `https://phigrade.mlcourse.org`, then run the command below **from the handout directory** (the one containing `phigrade.yaml`) and paste the key when prompted. Do not share your key or check it into version control.

```
phigrade auth login
```

Submitting From the handout directory, run:

```
pytest test_all.py
```

This scores your code and uploads `train_qformer.py`, `dit.py`, and `image_caption_data.py`, the only files you submit for this homework. You will be asked to choose a submission slot: *Slot A* (for human work only) or *Slot B* (for human or AI work). Choose the slot that reflects how you actually completed the assignment; see the syllabus for what is permitted in each.

Note: Every run of `pytest test_all.py` is recorded as a submission, and your most recent run before the deadline is the one that counts.

Note on points: The 0-point question below only records that you submitted. Your code is scored separately by the PhiGrade autograder: the 10 tests in `test_all.py` are worth 28 points in total, and those points count toward your homework grade alongside the written questions. The point values printed in this PDF cover only the written questions (83 points), so the full homework is worth 83 + 28 points, and any threshold expressed as a percentage of the homework (for example the Slot A bonus) is computed on that full total.

7.1. (0 points) Did you upload your code to the appropriate programming slot on PhiGrade?
Hint: The correct answer is 'yes'.

- ◯ Yes
- ◯ No

8 Collaboration Questions (2 points)

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found in the syllabus.

8.1. (1 point) Did you collaborate with anyone on this assignment? If so, list their name or Andrew ID and which problems you worked together on.

8.2. (1 point) Did you find or come across code that implements any part of this assignment? If so, include full details.