Advance Topics in Machine Learning

Assignment 0 - Introduction

27 August 2025

Guidelines

In this assignment, you will focus on setting up the foundation for the rest of the course by preparing a reproducible workflow, reporting your work in a professional format, and demonstrating the ability to document your findings properly. This assignment emphasizes good research practices—version control, reproducible experiments, and clear reporting.

Please note the following requirements carefully:

- Students are required to create a **public GitHub repository** and provide its link in the report as part of the submission.
- Students must write a **full report** detailing their findings in the **ICML format**[1].
- **Individual submission is mandatory** for all students.
- The deadline for submission is **8th September 2025**.

Task 1: Inner Workings of ResNet-152

1. Baseline Setup
   (a) Use a pre-trained ResNet-152 from PyTorch. [2]
   (b) Replace the final classification layer to match a smaller dataset such as CIFAR-10. [3]
   (c) Train only the classification head while freezing the rest of the backbone. [4]
   (d) Record training and validation performance for a few epochs.

   Why is it unnecessary (and impractical) to train ResNet-152 from scratch on small datasets? What does freezing most of the network tell us about the transferability of features?

2. Residual Connections in Practice
   (a) Disable skip connections in a few selected residual blocks and re-train the modified network head. [5]
   (b) Compare training dynamics and validation accuracy with the baseline.

   How do skip connections change gradient flow in very deep networks? What happens to convergence speed and performance when residuals are removed?

3. Feature Hierarchies and Representations
   (a) Collect features from early, middle, and late layers of the network. [6]
   (b) Visualize these features using dimensionality reduction (t-SNE or UMAP). [7]

   How does class separability evolve across layers? What differences can you observe between low-level and high-level representations?

4. Transfer Learning and Generalization

[1] https://www.overleaf.com/latex/templates/icml2025-template/dhxrkcgkvnkt
[2] https://pytorch.org/vision/stable/models/generated/torchvision.models.resnet152.html
[3] https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html
[4] https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.requires_grad
[5] https://arxiv.org/abs/1512.03385
[6] https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.register_forward_hook
[7] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html, https://umap-learn.readthedocs.io/en/latest/

(a) Fine-tune the model on a dataset different from ImageNet. [8]

(b) Compare performance between: (a) using ImageNet-pretrained weights, and (b) training from random initialization.

(c) Experiment with fine-tuning only the final block versus the full backbone.

(d) Compare performance between: (a) using ImageNet-pretrained weights, and (b) training from random initialization.

(e) Experiment with fine-tuning only the final block versus the full backbone.

Which setting provides the best trade-off between compute and accuracy? Which layers seem most transferable across datasets, and why?

5. Optional Experiments

   (a) Compare t-SNE vs UMAP in representing feature separability. [9]

   (b) Analyze feature similarities between classes that ResNet tends to confuse.

   (c) Compare feature quality from ResNet-152 with a shallower ResNet (e.g., ResNet-18). [10]

Task 2: Understanding Vision Transformers (ViT)

1. Using a Pre-trained ViT[11] for Image Classification: Choose a small pre-trained ViT model from PyTorch, TIMM, or HuggingFace[12], which was pre-trained on ImageNet. Use the corresponding image processor (feature extractor) to prepare input images. Select 1–3 images to test the model on – these could be from a standard dataset (e.g. an ImageNet sample) or any images of your choice (ensure they are appropriately sized, e.g. 224Ö224 pixels, or resize them with the feature extractor). Run the images through the ViT model to get predicted class labels. Record the top-1 prediction (the model's guess for the object in the image) and whether it seems reasonable.

2. Visualizing Patch Attention: One advantage of transformers in vision is the ability to visualize attention maps over the image patches, which can serve as a form of model interpretability (analogous to saliency maps in CNNs). We will create an attention-based visualization for the ViT's output. Do the following

   - Configure the model to output attention weights. In HuggingFace's ViTModel or ViTForImageClassification, you can pass output_attentions=True when calling the model (similar to the NLP case).
   - Focus on the class token's attention in the last layer. In ViTs, a special learnable "[CLS]" token is appended to the sequence of patch embeddings, and its output is used for classification. The attention weights from this CLS token to all patch tokens in the final layer indicate which patches were most influential for the classification decision. Extract the attention matrix from the last transformer layer. This will have shape (batch_size, num_heads, seq_len, seq_len) – where seq_len = N_patches + 1 (the +1 is the class token).
   - You may aggregate the heads for simplicity: for example, take the average over all heads' attention matrices in the last layer, or even just use one head if it appears to focus well. Locate the row (or column, depending on implementation) in that matrix corresponding to the CLS token's attention to other tokens. This will give a vector of size equal to the number of patches, representing how much attention the class token gave to each patch.
   - Reshape this attention vector back into the 2D spatial arrangement of patches. For instance, if the image was 224×224 with 16×16 patches, there are $(224/16)^2 = 14 \times 14 = 196$ patches. You can form a 14×14 map of the attention values.
   - Visualize the attention map: You can upsample this patch attention map to the image resolution and overlay it on the image to see which regions are highlighted. (This is similar to creating a heatmap

[8] https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
[9] https://distill.pub/2016/misread-tsne/
[10] https://pytorch.org/vision/stable/models/resnet.html
[11] https://medium.com/@nivonl/exploring-visual-attention-in-transformer-models-ab538c06083a
[12] https://pytorch.org/vision/stable/models/vision_transformer.html, https://huggingface.co/docs/transformers/main_classes/output

overlay on the image.) Use a library (like matplotlib or PIL) to create a semi-transparent red overlay where intensity corresponds to attention weight.

3. Analyze the Attention Map: Include the produced attention-map-overlaid image. Describe what you observe: Did the ViT focus on the regions of the image that correspond to the predicted class object? For example, if the image was of a dog and the model predicted "dog", do the highlighted patches outline the dog's figure? This provides insight into whether the model is looking at the correct features or if it might be focusing on background or other cues. Compare this attention-based explanation to typical CNN attention (if you are familiar, e.g. convolutional networks often use CAM or Grad-CAM – you don't need to implement those, just conceptually compare). Discuss the advantages of transformers having built-in attention for interpretability. If you notice any peculiar behavior (e.g. the model attended to an odd region), note that as well. Are different attention heads specialized? How can you tell?
4. Mask a fraction of input patches at inference and observe the effect on accuracy. How robust is ViT to missing patches? Why? Compare random masking with structured masking (e.g., masking the center). What do you observe?
5. Compare linear probes trained on the CLS token versus the mean of patch tokens. Which pooling method performs better? Why? How might this choice interact with different pretraining objectives?

Task 3: Understanding Generative Adversarial Network Dynamics

In this section, you will build and train a basic Generative Adversarial Network using the MNIST dataset.

1 Baseline Setup

1. **Dataset Setup:**
   - Begin by loading the MNIST dataset. Use `torchvision.datasets.MNIST` with appropriate transforms. We recommend normalizing the images to the range $[-1, 1]$ (because we will use a tanh activation in the generator). For example, use `transforms.ToTensor()` followed by `transforms.Normalize(mean=0.5, std=0.5)` so that pixel values of 0 map to -1 and 1 map to $+1$.
   - Set up a `DataLoader` for MNIST with a batch size (e.g., 64) that your CPU can handle. (MNIST images are 28×28 grayscale.)
2. **Model Architecture:** We will implement a straightforward fully-connected (MLP) GAN rather than a convolutional GAN, to keep it lightweight. Define the Generator and Discriminator as follows (you can adjust dimensions a bit, but stay reasonable):
   - **Generator G:** Input a noise vector z of dimension (for example) 100. Then have 2–3 linear layers with ReLU activations (and optional batch normalization) expanding to intermediate sizes, and finally a linear layer to output dimension 784 (which will be reshaped to 28×28). Use a tanh activation on the output layer to ensure the output is in $[-1, 1]$. For example, one design is:

     $$100 \to 256 \to 512 \to 784$$

     with ReLU activations on hidden layers and tanh on the output. (This matches a known stable implementation with 2 hidden layers of 256 neurons each.)
   - **Discriminator D:** Input is a 28×28 image, flattened to 784. Use 2–3 linear layers with LeakyReLU activations (e.g., slope 0.2), reducing from 784 down to 256 or 128, and ending with a single output neuron with a Sigmoid activation to estimate the probability that input is real. For example:

     $$784 \to 256 \to 256 \to 1$$

     with LeakyReLU on hidden layers and Sigmoid on output. (LeakyReLU helps avoid zero gradients by allowing small gradients even when the unit is "off.") Optionally, include Dropout (e.g., 0.3–0.5) in D's hidden layers for regularization, though this is typically not necessary for MNIST.
   - **Initialization:** Initialize weights using a normal distribution (e.g., with standard deviation ~ 0.02) if not using PyTorch's default initialization. This follows best practices from the DCGAN paper.
3. **Training Setup:**

- Use separate optimizers for D and G. We suggest the Adam optimizer for both, with learning rate $\approx 2 \times 10^{-4}$. Importantly, use $\beta_1 = 0.5$ and $\beta_2 = 0.999$ for Adam. This is a known trick to stabilize GAN training. *Explanation:* Setting $\beta_1 = 0.5$ reduces the momentum of the optimizer, which empirically stabilizes GAN training, as noted by Radford et al.
- Use binary cross-entropy (BCE) loss for both D and G objectives. In PyTorch, you can use `nn.BCELoss()`. Remember that D sees two kinds of inputs:
  - Real images with label = 1
  - Fake images generated by G with label = 0

  G's goal is to fool D, so you can compute G's loss using the non-saturating heuristic: use real label "1" for the fake images when computing G's loss, i.e., minimize $-\log(D(G(z)))$. This is preferred over the original minimax loss due to better gradient behavior.
- **Label smoothing (optional):** For added stability, you can apply one-sided label smoothing to D: when feeding real images, use labels = 0.9 instead of 1.0. This helps prevent D from becoming overconfident. *Note:* Do not smooth fake labels—only real labels should be smoothed. If used, document the effect during experiments.

4. **Training Loop:** Train the GAN for a certain number of epochs (start with, say, 20 epochs, then you can try longer if needed). In each epoch, iterate over the training data loader: for each batch, perform the following steps:
   - **Train Discriminator:** Get a batch of real images x from MNIST and a batch of random noise z. Compute $D(x)$ for real images and $D(G(z))$ for fakes. Compute D's loss as the sum of real-image BCE loss and fake-image BCE loss. Perform a D optimizer step to minimize this loss.
   - **Train Generator:** Generate a new batch of fake images $G(z)$ (use a fresh z or reuse the previous one, either is acceptable). Evaluate $D(G(z))$. Compute G's loss: we want $D(G(z))$ to be classified as real, so you can use label "1" for these fake images in the BCE loss. Backpropagate and update G's parameters.
   - Ensure you alternate updates: one update to D and one to G per batch. In this basic GAN, we use a 1:1 training ratio. After each update, you may want to zero out gradients for both optimizers using `optimizer.zero_grad()` to avoid accumulation.
5. **Print training progress:** For example, every n batches, output the current epoch, D loss, G loss, and perhaps the average $D(x)$ on real vs. $D(G(z))$ on fake (this gives a sense of how confident D is on real and fake samples). You can follow the format of the PyTorch tutorial or any reference code.
6. **Monitoring:** It's very insightful to monitor:
   - (a) the losses of D and G over time (plot them),
   - (b) sample generated images during training.

   Set aside a fixed noise vector (e.g. of size 100) at the start, and every few epochs generate images from this fixed noise and save or display them. This way you can see the progression of G's outputs from random noise to digit-like images.
7. **Results and Discussion:**
   - After training, plot the loss curves for D and G over epochs (or training iterations). Do they exhibit the dynamics you expect (e.g., D's loss initially goes down while G's goes up, etc.) or any signs of instability (e.g., oscillations)? Comment on these observations.
   - Display a set of generated images from your final generator. Ideally, generate a grid of, say, 5×5 images using different random z inputs. Comment on the quality: Do the images look like recognizable digits? Is there variety, and does the noise input seem to meaningfully affect the output? It is okay if some outputs are imperfect (e.g., blobs that resemble digits). For reference, even a simple MLP GAN can produce digit-like shapes – not perfectly, but often recognizable.
   - **Comparison with reference:** In reading list, a reference implementation of a similar GAN (see the GitHub repository `PyTorch-GAN` by Erik Linder-Norén, specifically the `gan.py` example for MNIST). Compare your results to those from the reference implementation or others' reported results. For example, did the reference GAN produce sharper or more diverse digits? If there are differences, speculate on why. Consider factors such as:

  - Differences in architecture
  - Hyperparameter choices
  - Random initialization effects

  If your results differ significantly, try to identify which design decision may have contributed.

- **Note:** Training an MLP GAN on MNIST is doable on CPU, but if you have access to a GPU, you can use it. Either way, ensure your code runs in a reasonable time. If training is too slow, you may reduce network sizes slightly or train for fewer epochs to demonstrate functionality. The key objective is to successfully train a GAN that generates somewhat digit-like images and to understand the training process.

2 Experimenting with Training Issues

One of the learning goals is to understand common training pathologies of GANs, and how to identify and mitigate them. In this part, you will perform controlled experiments with your GAN to purposefully induce some of the problems discussed (gradient vanishing, mode collapse, overfitting), and then attempt to mitigate them. Before starting, it's often useful to be able to reset your models and train from scratch under new settings, so ensure you can reinitialize your G and D and reproducibly run training.

1. **Gradient Vanishing:** Let us see how vanishing gradients can occur if the discriminator overpowers the generator, and show a technique to alleviate it.

   - **Setup / Perturbation:** Train the GAN with a much higher capacity or learning rate for D so that D learns to classify real vs. fake extremely well early on. For example, increase D's learning rate by $5\times$ or $10\times$ compared to G's, or give D extra training iterations per batch (e.g., train D 5 times for every 1 time you train G during the first few epochs). This will lead to $D(x) \to 1$ and $D(G(z)) \to 0$ quickly.
   - **Observation:** Plot the losses for G and D during this phase. What happens to G's loss after D has become very strong? You will likely observe D's loss $\to 0$ (for real/fake classification) and G's loss remaining high and nearly flat—indicating that G is not learning. This happens because G receives almost zero gradient when $D(G(z))$ is near 0 (as the gradient of $\log(1 - D(G(z)))$ approaches 0). What do you observe here? Did D's accuracy saturate near 100%? Did G's improvements stall (e.g., flat loss curve)? This demonstrates the *vanishing gradient problem* in action.
   - **Mitigation:** Apply a known trick to help G in this scenario: either one-sided label smoothing or using the non-saturating generator loss. For example:
     - Smooth the real labels to 0.9 so that D is never fully confident between real and fake.
     - Alternatively (or additionally), use the non-saturating G loss: maximize $\log(D(G(z)))$ instead of minimizing $\log(1 - D(G(z)))$.
     - Train again (you may reset to equal learning rates).
   - **Observation 2:** With these changes, does G's loss curve show better progress? Ideally, you'll observe that G's loss starts high but then decreases, indicating that G is beginning to fool D instead of remaining stuck. Plot the new loss curves alongside the original ones for comparison. You might also observe that D's loss does not immediately drop to 0 but remains at a higher value; this is desirable, as it implies D is not saturating and continues to provide useful gradients to G.
   - **Analysis:** Explain how label smoothing and/or the non-saturating loss helped mitigate the vanishing gradient issue. Connect this behavior to theory:
     - Smoothing real labels to 0.9 reduces D's confidence, keeping $D(G(z))$ away from 0 and allowing G to receive stronger gradients.
     - The non-saturating loss maximizes $\log(D(G(z)))$ directly, which yields a stronger gradient signal when $D(G(z))$ is small, compared to minimizing $\log(1 - D(G(z)))$.

2. **Mode Collapse:** Now, let us investigate mode collapse, where the generator collapses to producing only a few modes (e.g. the same digit every time). We will intentionally provoke mode collapse and then consider remedies.

- **Setup / Perturbation:** There are a few ways to induce mode collapse. One simple way in this MNIST context is to make G too powerful or too eager relative to D. For instance, try increasing G's learning rate significantly (e.g., $5\times$ or $10\times$) while keeping D's the same. Alternatively, decrease D's capacity (e.g., use fewer neurons) so it's weaker. This can lead G to exploit D's weakness by focusing on a single type of output that fools D. Another method is to train G for multiple steps per D step (the opposite of the previous setup) – this allows G to aggressively optimize against a relatively static D.
- **Observation:** During training with these settings, monitor the diversity of G's outputs. Every few epochs, generate a batch of (say 16) samples from G using different random z. Do you notice that many samples look very similar or even identical? For example, you might see that G has collapsed to always producing a "3" digit shape, ignoring other digits. Plot a grid of generated images; if mode collapse occurred, the grid might show almost all the same digit or style. Also examine D's loss: sometimes mode collapse is accompanied by D's loss stabilizing at a reasonably good value, because D finds one trick from G and G sticks to it.
- **Question:** Provide evidence of mode collapse from your experiment, e.g., "After 10 epochs with high G learning rate, all generated images were nearly the same digit 7. How can you quantitatively measure mode collapse? One indicator is low diversity: for example, very low variance between outputs despite different inputs. Another is that $D(G(z))$ becomes very high for one specific output style, and G stops exploring others.
- **Mitigation:** One strategy to combat mode collapse is to make D stronger or to encourage diversity in G. For example, revert to normal learning rates (or increase D's learning rate) and consider techniques like mini-batch discrimination or adding a small noise perturbation to G's input each time (so even the same z doesn't always produce identical outputs). Implement a simple change such as training with a more balanced approach (e.g., train D twice per G step for a while). This helps because a stronger D can punish G if it keeps generating the same outputs.
- **Observation 2:** After applying a remedy (even something simple like training D more often), do you observe the generator recovering diversity? For instance, when D is trained more aggressively, G can no longer fool it with just one mode, and may start producing more varied outputs again. See any improvements.
- **Analysis:** Explain why mode collapse is a stable but undesirable equilibrium: G might find a single output that reliably fools D, and D may get stuck if it hasn't seen enough diversity to penalize G for it. This leads to a local equilibrium rather than the desired global one. Many research works address this, such as Unrolled GANs, VAE/GAN hybrids, and more. Conclude with which intervention you found most effective in restoring diversity. You may also mention that feature matching is another effective method: instead of G maximizing D's output directly, G is trained to match statistics of real data in D's feature space. This can implicitly encourage coverage of all data modes.

3. **Discriminator Overfitting:** Lastly, let us examine what happens if the discriminator overfits the training data and how that impacts G (and generalization). While GANs are unsupervised, D can still overfit to the finite training set, especially if G isn't improving; D might memorize training examples. We'll simulate this by limiting data or over-training D.
   - **Setup / Perturbation:** One approach is to train on a reduced dataset. For example, restrict MNIST to only the first 1,000 images (or even 200 images of a single class, for a dramatic effect). Train the GAN on this limited set. Another approach is to give D a very large capacity (many layers or neurons) relative to G. In either case, monitor D's performance on the training data versus some held-out validation data (you can set aside a small portion of MNIST as "test" for D evaluation).
   - **Observation:** If D overfits, it will classify its training images as real with near 100% accuracy but may incorrectly flag real images it hasn't seen as fake. One sign is D's training loss $\to 0$ while G fails to improve (similar to earlier experiments). In extreme cases, G may also overfit by memorizing some of the real training examples—a phenomenon known as *mode replication*.
   - To detect D's overfitting, you can evaluate D on real images not seen during training. If D has memorized the training data, it might assign low $D(x)$ values to unseen real samples, treating them as fake. Another indicator is D focusing on superficial patterns in the training data (e.g., specific writers' styles in MNIST) rather than generalizable features.
   - **Question:** Describe the effects you observed. For instance: "When training on only 500 images, the discriminator quickly achieved near-zero training loss. The generator's output did not meaningfully

improve; it produced the same few digits repeatedly. When tested on new real MNIST images, the discriminator only gave ∼50% real probability, indicating it had become over-specialized to the training set." If you observed G replicating training examples, mention this as a form of overfitting.

- **Mitigation:** How can we prevent D from overfitting? In practice: use a larger dataset, regularize D (e.g., with Dropout), apply early stopping to D, or limit D's capacity. As a simple experiment, try stronger regularization:
  - Add a higher Dropout rate (e.g., Dropout(0.4)) to D's hidden layers.
  - Alternatively, stop D's training early in each epoch to avoid full convergence before G gets updated.
  - Train again on the limited dataset using these modifications.
- **Observation 2:** Does D's performance change with regularization? Ideally, Dropout should prevent D from achieving zero loss too quickly, giving G more meaningful gradients. See whether the generator's outputs looked better or showed increased variety when D was regularized.
- **Analysis:** Summarize the key insight: a discriminator that memorizes the training data provides unhelpful gradients to G and can harm generalization. Overfitting is especially problematic with small datasets. Techniques such as Dropout or reduced capacity force D to stay uncertain, making it a better teacher for G. Connect this to the broader goal: GANs should generalize to the true data distribution, not just a memorized subset. A strong but overfit D drives G to replicate examples, which is not true generation.

Task 4: Training Variational Autoencoders

1. Train the VAE
   (a) Download the FashionMNIST dataset from torchvision.
   (b) Download the provided `architecture.py` file for VAE architecture.
   (c) Define your loss function composing of MSE Loss for reconstruction and KL divergence. Without modifying the model architecture, train the VAE on FashionMNIST.
   (d) Record the training and validation losses.
2. Visualize Reconstructions and Generations
   (a) Visualize reconstructions from the encoder-decoder pipeline for several test examples.
   (b) Visualize generations by sampling $z \sim p(z)$ and decoding to image space.
   (c) Try generating samples from a different prior distribution (e.g., Laplacian) and compare.
   (d) Document these samples in your report and describe any observations or differences between reconstructions and generations.
3. Posterior Collapse Investigation
   (a) Examine your reconstructions and generations carefully. You should notice that even though your loss is decreasing, the generated samples may look very similar or form a uniform "blob."
   (b) Analyze the ELBO components: reconstruction loss and KL divergence.
   (c) Confirm whether your posterior $q_\phi(z|x)$ is collapsing (i.e., the encoder outputs are ignoring the input and matching the prior too closely).
   (d) Think about why and under what circumstances posterior collapse happens in VAEs. Consider:
      i. The relative power of the encoder and decoder.
      ii. The effect of the KL term early in training.
4. Mitigating Posterior Collapse
   (a) Design and implement a strategy to prevent posterior collapse **without modifying the model architecture**.

(b) Hint: you might want to give the encoder a chance to learn meaningful representations before the KL term is fully enforced. Consider adjusting learning rates, training the encoder more aggressively, or gradually introducing the KL term.

(c) Compare results of reconstruction and generations with your strategy and explain briefly why your strategy works.

Task 5: Modality Gap in CLIP

Contrastive Language–Image Pretraining (CLIP)[13] is a multimodal model that jointly trains a vision encoder and a text encoder to map images and natural language descriptions into the same embedding space. CLIP enables powerful zero-shot classification by leveraging human-readable labels directly as text prompts, removing the need to train a separate classifier.

1. Zero-Shot Classification on STL-10

   (a) Download the STL-10 [14] dataset from torchvision.

   (b) Load OpenAI's CLIP model [15] from the official implementation of clip.

   (c) Evaluate clip for zero-shot accuracy on STL-10 using different prompting techniques:

      i. Plain labels (e.g., "cat").

      ii. Prompted text (e.g., "a photo of a cat").

      iii. More descriptive variants of prompts.

   (d) You can experiment with the prompts on a small subset. However, you are to compare accuracies across atleast 3 different prompting strategies for the whole test set.

2. Exploring the Modality Gap

   (a) Use the vision and text model within CLIP to extract image and label embeddings from CLIP for a few (50-100) STL-10 samples.

   (b) Use dimensionality reduction techniques such as UMAP or t-SNE to project the embeddings into 2D space.

   (c) Visualize and compare the distributions of text and image embeddings.

   (d) Briefly explain your findings: How separated are the modalities? Does normalization affect the modality gap? Why does CLIP still perform well despite this gap?

3. Bridging the Modality Gap

   (a) One simple method to align modalities is the orthogonal Procrustes transform. Given two sets of embeddings X (image features) and Y (text features), the goal is to find an orthogonal matrix R that minimizes:

   $$\min_R \|XR - Y\|_F,$$

   where $\|\cdot\|_F$ is the Frobenius norm. The closed-form solution involves singular value decomposition (SVD), but a library implementation will suffice for our case.

   (b) Pair STL-10 image embeddings with their corresponding text embeddings.

   (c) Learn the optimal rotation matrix R using a library implementation of Procrustes alignment (hint: `scipy.linalg.orthogonal_procrustes` or equivalent).

   (d) Apply the rotation transform to the CLIP embeddings.

   (e) Visualize the aligned embeddings with t-SNE or UMAP. How does the alignment affect the modality gap?

   (f) Recompute classification accuracy with the aligned embeddings and compare results with Part 0.

[13] https://arxiv.org/abs/2103.00020
[14] https://docs.pytorch.org/vision/main/generated/torchvision.datasets.STL10.html
[15] https://github.com/openai/CLIP