AI 623 - Deep Vision Language Models

Assignment 0 - Introduction

18 January 2026

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:

- All those who have studied *CS6304: Advanced Topics in Machine Learning (ATML)* are exempted from the submission but are highly encouraged to go through this warm-up once again. Following instructions apply only on the rest of the class.
- 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**.
- The deadline for submission is **1st February 2026**. The grading will be binary.

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]

[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/

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

4. Transfer Learning and Generalization
   (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.

   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: 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 4: 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