Cornell University
CS 5670: Introduction to Computer Vision
Fall 2025. Instructor: Andrew Owens

Problem Set 3: Transformers

Posted: Monday, September 29, 2025	**Due:** Friday, October 17, 2025

Please submit your Colab notebook to Gradescope as a `.pdf` file. Please convert your Colab notebook to PDF. For your convenience, we have included the PDF conversion script at the end of the notebook.

The starter code can be found at:
https://colab.research.google.com/drive/1DEqd3sC5xU31FUMQNtvFG6mhLxGzbofp

Please see **GPU Tips** at the end of this document to better understand how to efficiently use GPUs for this homework set (and all future homeworks that require GPUs). **tldr:** your code will run *very* slow if you run out of GPU hours early and you have to train your models on CPUs. Please follow the tips we provide to avoid this! We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

Problem 3.1 *Vision Transformer*

In the previous problem set, we implemented MLPs, CNNs, and networks with residual connections. In this assignment, we'll take a step further by implementing the Vision Transformer (ViT) [1]. Inspired by advances in NLP, ViTs treat an image as a sequence of fixed-size patch tokens and process them with a transformer. Unlike CNNs, which introduce strong spatial biases through convolution, ViTs rely more on data to learn these structures. This makes them more data hungry but also arguably more flexible and scalable. However, as a consequence, the ViT will not perform quite as well as the CNNs from PS2, due to the small dataset and model size.

In the first half of this problem set, we will build a ViT incrementally, starting with the patch embedding and the positional encoding, then implementing multi-head self-attention and transformer blocks. Finally, you will assemble these components into a tiny ViT model and train it for image classification (Fig. 1).

As in the previous problem set, we will continue to use CIFAR-10 [3]. Since CIFAR images are only 32×32 pixels, we will use a patch size of 4 (instead of 16 in standard ViTs).

We will train the model using the AdamW optimizer [4], which is standard for transformers. While we didn't discuss this optimizer in class, it is quite closely related to stochastic gradient descent with momentum. You can find more information about it here.

Important: For each subproblem in Problem 3.1, we provide a test network to verify the

correctness of your implementation. Be sure to train each test network and confirm it reaches the expected accuracy before proceeding. We will also take these training results into account when grading your work. Please do not modify the architecture or training hyperparameters of these networks, as they have been carefully calibrated.

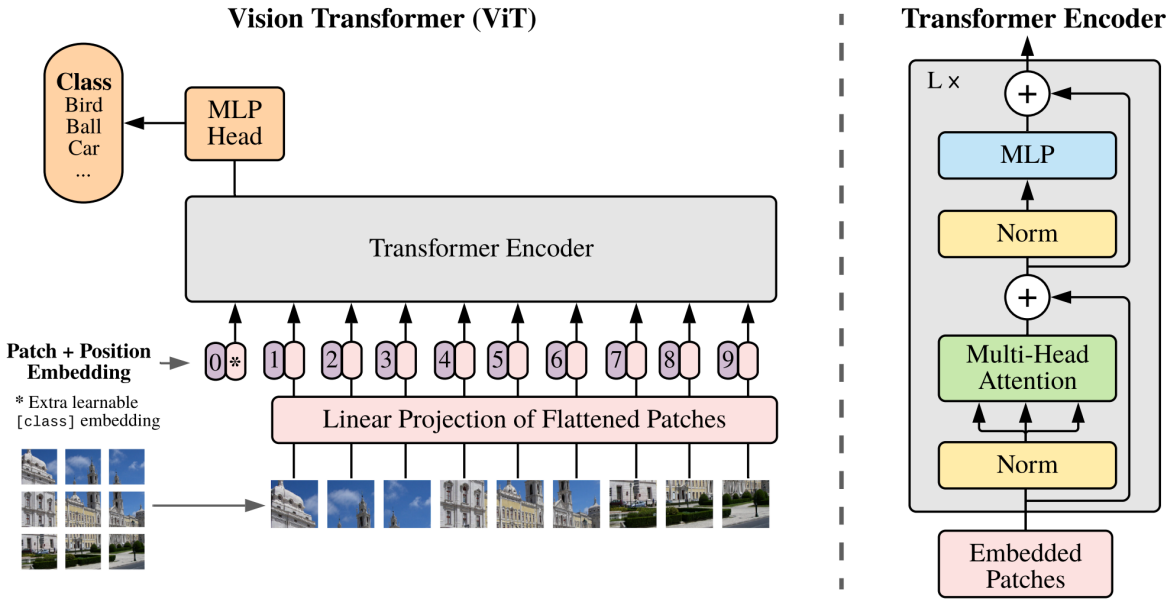

Figure 1: **Vision transformer overview**. (Left) We will create a transformer whose tokens come from image patches. (Right) Transformer architecture diagram. Figure source: Dosovitskiy et al. [1]

(a) **(1 point) Patch Embedding.** We will obtain tokens from image patches. The patch embedding module splits the image into non-overlapping patches (e.g., (16×16) pixels), flattens each patch into a vector, and passes it through a linear projection to a common embedding dimension (C). If the input image has shape ((B, 3, H, W)) and the patch size is (P), the output becomes a sequence of $(H \cdot W/P^2)$ tokens, each of feature dimension (C).

i. **Patchify.** Implement the `patchify` function following the instructions. You should not use any for loop.

ii. **Patch Embed.** Complete the implementation of the `PatchEmbed` module following the instructions.

(b) **(1 point) Positional Embedding.** ViT (and Transformer in general) is order-agnostic. They treat input tokens as a set instead of a sequence. To inject information about the spatial layout of image patches, we need to add a positional embedding to each token. The positional vectors have the same dimension as the patch embeddings and are simply added to them before feeding the sequence into the Transformer.

i. **Sinusoidal embedding** is a non-learnable encoding based on sine and cosine functions (of different frequencies) on the token position. Run the visualization code we provided.

ii. **Learnable embedding** is a trainable parameter that learns an embedding for each token position. Implement the `LearnablePosEmbed` module following the instructions.

(c) **(1 point) Layer Normalization.** Layer normalization stabilizes and accelerates training by normalizing activations across the feature dimension of each input token. Given an input vector $x \in \mathbb{R}^d$, LayerNorm will normalize it to zero mean and unit variance, then rescale and shift the result using learnable parameters γ (scale) and β (bias):

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

Implement the `LayerNorm` module following the instructions.

(d) **(3 points) Transformer Block.** A Transformer block is the main building unit of Vision Transformers. Each block contains two key components: **multi-head self-attention** (MHA), which allows tokens to exchange information globally, and a **feed-forward network** (FFN), which refines each token's representation.

i. **FFN.** is a two-layer MLP applied independently to each token. We usually expand the hidden dimension by a factor (e.g., 4) inside the two linear layers. Implement the `MLP` module following the instructions.

ii. **MHA.** computes the *scaled dot-product attention* by projecting tokens into query, key, and value vectors, which allows each token to aggregate information from the entire sequence:

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

Using multiple heads allows the attention block to attend to different types of relationships or spatial patterns in parallel. Implement the `Attention` module following the instructions.

iii. **TransformerBlock.** combines MHA and FFN. It applies LayerNorm to both MHA and FFN and adds skip connections after each of them. Modern architectures commonly use the **pre-norm** design, which applies normalization before each subblock:

$$x = x + \mathrm{MHA}\big(\mathrm{LN}(x)\big),$$
$$x = x + \mathrm{FFN}\big(\mathrm{LN}(x)\big).$$

In contrast, **post-norm** applies Layer Normalization after each subblock's residual addition, which modifies the identity branch and makes training harder. Implement the `TransformerBlock` module following the instructions.

(e) **(2 points) Vision Transformer.** Let's put everything together into a ViT. Since CIFAR is a small dataset, we set our ViT with {feature dimension: 192, number of heads: 3, patch size: 4}. We also use a smaller learning rate (1e-4) to stablize training.

i. Implement the `VisionTransformer` module following the instructions. ViT introduces a learnable **[CLS] token** (classification token) that is prepended to the token sequence. This token works as a summary that attends to all other tokens through the attention layers. After passing through all Transformer blocks, we will take the [CLS] token as a global representation of the image and feed it into the linear classification head.

ii. Notice that our ViT overfits due to insufficient data. One way to mitigate this is data augmentation. Implement `data augmentation` following the instructions and re-train our model.

Problem 3.2 *Masked Autoencoder*

In this problem, we will implement and train the Masked Autoencoder (MAE) [2], a self-supervised learning framework designed to pretrain ViTs without labeled data. Instead of predicting class labels, the model learns to reconstruct the original image from a heavily masked version. It consists of a ViT encoder and a lightweight decoder (Fig. 2):

Encoder: The encoder processes the unmasked patches. We will first randomly *shuffle* and *mask* a large portion of image patches (often 75% or more). We feed only the remaining visible patches into the ViT encoder. This encoder produces latent representations for the visible tokens.

Decoder: The decoder reconstructs the image, given the output of the encoder. It takes the encoder outputs and inserts a set of special [*mask tokens*] to represent the missing patches. We will *reorder* the sequence to the original patch order using an *index map* (saved when we shuffle image patches). We then add *decoder positional embeddings* to the tokens so the model knows where each token is spatially. This full sequence (visible + mask tokens) is fed through a lightweight Transformer decoder. Finally, a linear projection head maps each token to the pixel space to predict the pixel values of all patches.

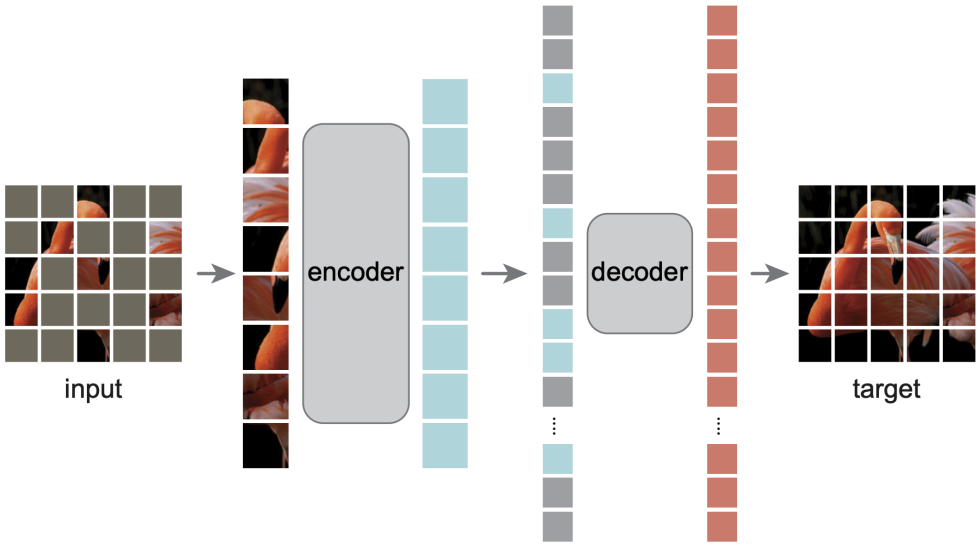

Figure 2: **Masked autoencoder**. We randomly mask (i.e., drop) image patches from an input image. We then train a transformer to recover the original image. From this process, the network learns a feature representation that we can transfer to other tasks. Figure source: He et al. [2]

In order to reconstruct the image, the encoder needs to capture the relationship between different patches. Through this process, it learns a representation that conveys information about objects and visual structures in the scene. After pretraining, the decoder is discarded, and the encoder can be fine-tuned for downstream tasks such as classification or detection.

(a) **(2 points)** Implement the functions `forward_encoder` and `forward_decoder` following the instructions. Pretrain the MAE. **Note:** training may take roughly an hour, so please plan ahead. Make sure you save the weights properly. We will use it in the next problem.

(b) **(1 point)** We can now fine-tune the encoder for classification, starting from the pretrained weights. Implement the function `forward` following the instructions and train the model.

Acknowledgements. This problem set was created by Xuanchen Lu.

1 GPU Tips

For the remainder of the class, we will be using GPUs with Google Colab. We use Colab, since we believe that it is the best possible option (given the fact that high quality GPUs are very expensive). However, there are a number of challenges due to the limits that Colab puts on GPU usage for free accounts. To help you get some information on this and make the most use of the free limits we have compiled a list of best practices to follow.

Colaboratory or simply Colab is part of the Google suite of products. Put simply, Colab is a free Jupyter notebook environment that runs entirely in the cloud. You can read more about it here: https://colab.research.google.com/. We use Colab with Python environment and to speed up model training. We will be using a very important feature of Colab which is GPU access.

Colab usually provides T4 GPU for free accounts. Very rarely you'll get a A100 or L4, which are much faster. Colab has time and usage limits for GPU access and the rules for these limits are vague so we have to be careful using them. The general observation is that you start off with a fixed (4 to 8hr, could be lower) bank. Once you exhaust that, you will not longer be able to use GPU for upto 24hrs. Once the counter resets, you get lower hours of usage each time. So how do we maximize our free GPU resources?

(1) **Use CPU for debugging**

The majority of the implementation in most of the assignments can be completed in CPU; you should only switch to GPU once you are confident about your code and ready to train your finals models. The notebook we give you might connect to a GPU runtime by default. You can change between CPU and GPU instances using Runtime → Change Runtime Type; this resets the notebook backend, so you'll need to rerun all cells after switching instance types. You will also need to change `device = torch.device('cuda')` to `device = torch.device('cpu')` Please make sure you don't have any `.cuda()` statements but rather `.to(device)`

Also, please note that **you do not need the GPU for problems 2.1 and 2.2, so make sure to run in CPU-only mode for these problems.**

(2) **Use other Google accounts**

If you have multiple Google accounts, please note that each one has a separate limit.

Enable cell execution notifications: Colab can notify you of completed executions even if you switch to another tab, window, or application. You can enable it via Tools → Settings → Site → Show desktop notifications (and allow browser notifications once prompted) to check it out. This is helpful when training models so you don't leave GPU connected and idle for long times.

(3) **Debugging tips for GPU**

```
RuntimeError: CUDA device-side assert triggered
```

Do not rely on the line where this error is raised. Even the error traceback says this:

RuntimeError: CUDA error: device-side assert triggered CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect. For debugging consider passing `CUDA_LAUNCH_BLOCKING=1`

The stack trace might be incorrect. For better debugging, re-run your code as

```
CUDA_LAUNCH_BLOCKING=1 python script.py
```

or add

```
import os
os.environ["CUDA_LAUNCH_BLOCKING"] = 1
```

at the top of your code (temporarily) for debugging. The traceback you get now will point you to the actual line where the error occurs.

(4) **Non-Colab options**

We highly recommend using Colab. However, there are several other options.

If you have a GPU in your local machine and can set it up to use within Jupyter notebooks, you can use it to run the notebooks. In our experience, this approach is also less reliable, so we recommend using Colab when possible. Also, since the problem sets are designed for Colab, running them here will require modifications. You can also connect Colab to a local runtime with GPU. Instructions here: https://research.google.com/colaboratory/local-runtimes.html. Please note that the instructors will not be able to help debug environment setup and code issues on non-Colab systems.

(5) **Colab Pro**

Google offers paid premium upgrades: Colab Pro and Pro Plus which are $10/month and $50/month respectively. These upgrades have access to faster GPUs and for longer. If you need more access, you can consider these options. All assignments will be thoroughly tested to ensure that it can completed within the limits of free Colab resources.

References

[1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[2] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16000–16009, 2022.

[3] A. Krizhevsky, V. Nair, , and G. Hinton. Cifar, 2009. URL https://www.cs.toronto.edu/~kriz/cifar.html.

[4] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.