GROUP 61

ViT-tiny Inference Pipeline in C

> ***Problem Statement:*** *Implement an end-to-end ViT-tiny inference pipeline in pure C that ingests a single RGB image, splits it into patches, performs linear patch embedding with positional encoding, processes the tokens through Transformer encoder layer (multi-head self-attention, MLP, residuals, and LayerNorm), and applies a classification head to produce the predicted class using pretrained ViT-tiny weights for all model parameters.*

PIPELINE STEP 1

Image Preprocessing

The pipeline begins by ingesting a standard RGB image. To match the model's training requirements, the image undergoes a series of transformations:

1. **Load:** A *.ppm* image file is read into memory.
2. **Resize:** The image is resized to 256 pixels along shorter side, maintaining aspect ratio, using bilinear interpolation.
3. **Crop:** A 224×224 center crop is extracted from the resized image.
4. **Normalize:** Pixel values are normalized using the standard ImageNet mean and standard deviation to center the data.

PIPELINE STEP 2

Patch Embedding (Conv2D)

This is the key adaptation of the Transformer for vision. The 224 × 224 × 3 image is converted into a sequence of tokens, similar to words in a sentence. This is achieved with a 2D convolution using a kernel having spatial size 16×16, 3 input channels, 192 output channels and a 16×16 stride.

This operation divides the image into a 14 × 14 grid of 196 patches. Each patch is then linearly projected from 16 × 16 × 3 = 768 dimensions down to the model's embedding dimension of D=192.

```
Input: [1, 3, 224, 224] → Output: [1, 196, 192]
```

PIPELINE STEP 3

Token Preparation

Two components are added to the patch sequence before it enters the encoder:

1. **[CLS] Token:** A special, learnable token (from *cls_token.bin*) is prepended to the sequence. This token acts as a global aggregator that will collect information from all other tokens to make the final classification.

2. **Positional Encoding:** Transformers are permutation-invariant (order-blind). To provide spatial information, a learnable positional embedding (from *pos_embed.bin*) is added element-wise to each token in the sequence.

PIPELINE STEP 4

The Transformer Encoder (L=12)

This is the core of the model. The token sequence (now $1 \times 197 \times 192$) is processed by a stack of L=12 identical Encoder Layers. Each layer consists of two main sub-layers: Multi-Head Self-Attention and a Multi-Layer Perceptron (MLP).

This implementation uses a **Pre-Norm** architecture: the *LayerNorm* is applied *before* each sub-layer to stabilize the activations, followed by a residual connection *after* each sub-layer.

ENCODER SUB-LAYER 1A

Layer Normalization (1)

This component stabilizes the network by normalizing the activations. It operates position-wise, meaning each of the 197 token vectors is normalized independently across its D=192 dimensions.

This implementation uses **Welford's algorithm**, a one-pass method to compute the mean (μ) and variance (σ^2) simultaneously, which is more memory-efficient than a standard two-pass calculation.

$$\mu_x = (1/D)\ \Sigma\ x_i \quad ; \quad \sigma^2_x = (1/D)\ \Sigma\ (x_i - \mu_x)^2$$

$$\mathrm{LayerNorm}(x) = \gamma\ (\ (x - \mu_x)\ /\ \sqrt{(\sigma^2_x + \varepsilon)}\ ) + \beta$$

LayerNorm: A Comparison of Methods

Standard Two-Pass Method

This is the straightforward implementation. It is highly inefficient, requiring two full passes over the token's data.

```
// Loop 1: Calculate Mean
μ = 0
for d in 0..D:
    μ += data[d]
μ /= D

// Loop 2: Calculate Variance
σ² = 0
for d in 0..D:
    σ² += (data[d] - μ)²
σ² /= D
```

Welford's Algorithm (One-Pass)

This is a more efficient and numerically stable implementation. It reads the token's data only once and computes variance on the fly.

```
// Calculate Mean & Variance at once
μ = 0; M2 = 0
for d in 0..D:
    delta = data[d] - μ
    μ += delta / (d + 1)
    delta2 = data[d] - μ
    M2 += delta · delta2

σ² = M2 / D
```

ENCODER SUB-LAYER 1B

Multi-Head Attention (MHA)

Self-Attention allows every token to look at and score its relevance against every other token in the sequence. Multi-Head (H=3) means this is done in parallel, allowing the model to learn different relationships in different subspaces. The mathematical steps are:

1. **Project:** Extract Q, K, V into single matrix of size $1 \times 197 \times 576$
2. **Split:** Q_h, K_h, V_h for $h \in \{1..H\}$ (where H=3)
3. **Score:** Scores = $Q_h \cdot K_h^T$
4. **Scale:** Scores = Scores / $\sqrt{d_k}$ (where $d_k = 64$)
5. **Softmax:** A = softmax(Scores)
6. **Attend:** $Z_h = A \cdot V_h$
7. **Merge/Project:** Out = Concat(Z_1..Z_H)

ENCODER COMPONENT

Residual Connection (1)

This is the Add part in the Add & Norm architecture. It's a skip connection that adds the input of the sub-layer (the original *PreprocessedInputs*) to its output (the *MHAOutput*).

This is one of the most important concepts in deep learning. It allows the network to bypass a layer if it's not useful (learning an identity function) and, more critically, it prevents the **vanishing gradient problem**.

$$X_{out} = X_{in} + \mathrm{Sublayer}(\mathrm{LayerNorm}(X_{in}))$$

ENCODER SUB-LAYER 2A

Layer Normalization (2)

A second *LayerNorm* is applied, this time to the output of the first residual connection. This stabilizes the input that will be fed into the MLP.

This Pre-Norm approach (normalizing *before* the transform) is a key part of the ViT architecture that improves training stability.

ENCODER SUB-LAYER 2B

MLP: Linear Layer 1

The MLP (Multi-Layer Perceptron) begins. Each token is processed independently.

The first linear layer (GEMM) expands the dimension of each token from **192 to 768**. This is the *mlp_forward* function in the code, which handles the matrix multiplication (*gemm*) and bias addition (*add_bias*).

$$X_{hidden} = X_{normed} \cdot W_1 + b_1$$

ENCODER SUB-LAYER 2C

GELU Activation

The **GELU** (Gaussian Error Linear Unit) activation function is applied element-wise to the output of the first linear layer.

This is the critical non-linear "thinking" part of the MLP block, allowing it to learn complex patterns. This is a separate call to *gelu(FC1out)* in the main loop.

$$X_{activated} = \mathrm{GELU}(X_{hidden})$$

Comparison of GELU Implementations

Standard GELU

This is the true mathematical definition of the Gaussian Error Linear Unit. It relies on the **Gaussian Error Function (erf)**, which is accurate but computationally expensive.

```
// The true mathematical definition
GELU(x) = 0.5 · x · (1 + erf(x / √2))
```

Optimized GELU (Tanh Approx.)

This is a fast, high-quality approximation that replaces the slow erf function with the much faster tanh function. This is the implementation we use in our C pipeline.

```
// The fast & good enough approximation
GELU(x) =
0.5·x·(1+tanh(√(2/π)·(x+0.044715·x³)))
```

ENCODER SUB-LAYER 2D

MLP: Linear Layer 2 & Residual

1. **Linear Layer 2:** The second linear layer (a call to *mlp_forward*) contracts the dimension of each token from **768** to **192**.

2. **Residual Connection:** The output of this second linear layer is added to the output of the *first* residual connection (the one saved before this sub-layer began).

This final tensor is the output of the entire encoder layer.

$$X_{out} = (X_{activated} \cdot W_2 + b_2) + X_{resid2}$$

Data Flow & Dimensions

A trace of the tensor dimensions as a single image passes through the pipeline.

1. Preprocessing

Input Image	(1 × 3 × 333 × 500)
Resized Image	(1 × 3 × 256 × 384)
Cropped Image	(1 × 3 × 224 × 224)
Normalized Image	(1 × 3 × 224 × 224)
Patch Embeddings	(1 × 196 × 192)
Token Sequence	(1 × 197 × 192)

2. Encoder Block (L=12)

Block Input	(1 × 197 × 192)
Norm1 Output	(1 × 197 × 192)
MHA Output	(1 × 197 × 192)
(Residual Add 1)	
Norm2 Output	(1 × 197 × 192)
MLP (fc1) Output	(1 × 197 × 768)
GELU Output	(1 × 197 × 768)
MLP (fc2) Output	(1 × 197 × 192)
(Residual Add 2)	
Block Output	(1 × 197 × 192)

3. Classification Head

Final Norm	(1 × 197 × 192)
Final CLS Token	(1 × 1 × 192)
Final Logits	(1 × 1 × 1000)

PIPELINE STEP 5

Classification Head

After the token sequence passes through all 12 encoder layers, we **discard** all 196 patch tokens. We only keep the very first token: the **[CLS]** token.

This token, having aggregated information from the entire image, is passed through one final *LayerNorm* and a single linear layer (the head). This layer projects the D=192 dimension vector into a 1000-dimension vector, one for each possible ImageNet class.

$$\text{Logits} = \text{LayerNorm}(X_{final}[0]) \cdot W_{head} + b_{head}$$

PIPELINE STEP 6

Final Output (ArgMax)

The 1 × 1000 vector of logits represents the model's raw, unnormalized score for each class.

To get the final prediction, an *argmax* operation is performed. This simple function scans the entire vector and finds the index (from 0 to 999) that has the single highest score. This index is then used to look up the human-readable class name from the loaded labels file.

```
Prediction = Labels[ argmax(Logits) ]
```

Project Group 61

Vivaan Malik

B25PH1020

Preprocessing, Conv2D, MHA & Classification Head

Devavrat Khandekar

B25MT1012

Softmax, MHA & README

Shivansh Sahu

B25BB1032

LayerNorm, MLP & Presentation