HCMUS - VNUHCM / FIT /

Computer Vision & Cognitive Cybernetics Department

Digital Image and Video Processing Application

Student ID: 21127690

Student name: Ngo Nguyen Thanh Thanh

Report: Vision Transformer

I. Evaluation summary:

Task	Completion	Notes
ViT Model Implementation	100%	Includes all required modules
Training on CIFAR-10	100%	Completed with 2 variants
Accuracy & Loss Visualization	100%	Plots generated
Report Documentation	100%	Includes all required sections
Unit Testing (tests folder)	100%	Covered all core modules

II. List of features and file structure:

Project Directory Tree

```
SOURCE/
├── configs/
│   ├── __init__.py
│   └── experiment_config.py      # Experiment setup (e.g., hyperparameters)
│
├── data/                         # Placeholder for datasets or dataset handlers
│
├── evaluation/
│   ├── __init__.py
│   └── evaluate.py               # Functions to evaluate model performance
│
├── experiments/
│   ├── __init__.py
│   └── logger.py                 # Logging utilities for training runs
│
├── logs/                         # Saved results of experiments (csv/json)
│   ├── experiment_20250410_*.csv
│   └── experiment_20250410_*.json
│
├── models/                       # Model architecture modules
│   ├── __init__.py
│   ├── attention.py              # Multi-head Self-Attention module
│   ├── classification_head.py    # Final classifier layer (MLP head)
│   ├── transformer_block.py      # Single transformer encoder block
│   ├── transformer.py            # Stacked transformer blocks
│   └── vision_transformer.py     # Full ViT architecture definition
│
├── tests/                        # Unit tests for each model component
│   ├── __init__.py
│   ├── test_attention.py
│   ├── test_classification_head.py
│   ├── test_transformer_block.py
│   ├── test_transformer.py
│   └── test_vision_transformer.py
│
├── training/
│   ├── __init__.py
│   └── train.py                  # Main training loop implementation
│
├── utils/
│   ├── __init__.py
│   ├── data_utils.py             # Data preprocessing, patch extraction, etc.
│   └── visualize.py              # Loss/accuracy plotting functions
│
├── .gitignore
├── main.py                       # Entry point for training/experiments
├── README.py                     # Project documentation file
└── requirements.txt             # Python dependencies for environment setup
```

The project is organized into modular components, ensuring separation of concerns and scalability.

- **configs/**:
  Contains configuration files, such as experiment_config.py, for managing model and training settings.

- **data/**:
  (Expected to) handle data loading, preprocessing, or dataset management.
- **evaluation/**:
  For evaluation scripts to test model performance (e.g., accuracy, confusion matrix).
- **experiments/**:
  Used to manage different training runs or experimental setups.
- **logs/**:
  Stores output logs like training loss and accuracy in .csv or .json format.
- **models/**:
  Contains model definition files, such as ViT architecture components.
- **tests/**:
  Used for unit tests or validation scripts to ensure code correctness.
- **training/**:
  Includes training pipeline logic, loss computation, and optimizer settings.
- **utils/**:
  Utility functions for data handling (data_utils.py) and visualization (visualize.py).
- **main.py**:
  Entry point for running the whole training or testing pipeline.
- **README.md**:
  Provides project overview and usage instructions.
- **requirements.txt**:
  Lists Python package dependencies for easy environment setup.

Functions and methods used:

List of Functions:

- Attention(embed_dim, heads): Computes multi-head self-attention.
- TransformerBlock(embed_dim, mlp_dim, heads): Implements a single transformer layer.
- Transformer(embed_dim, mlp_dim, layers, heads): Stacks multiple transformer layers.
- ClassificationHead(embed_dim, classes): Predicts class probabilities.
- VisionTransformer(params): Full ViT model.
- train(model, trainloader, num_epochs, ...): Trains the model.
- evaluate(model, testloader, device): Computes accuracy.
- get_dataloaders(): Loads CIFAR-10 data.
- plot_loss(csv_file, output_path): Plots training loss.
- plot_accuracy_comparison(json_files, output_path): Plots accuracy comparison.

Function List with Screenshots

Attention

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Attention(nn.Module):
    def __init__(self, embed_dim, heads, dropout=0.1):
        super(Attention, self).__init__()
        self.embed_dim = embed_dim
        self.heads = heads
        self.head_dim = embed_dim // heads

        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(embed_dim, embed_dim)

    def forward(self, inp):
        batch_size, seq_len, embed_dim = inp.size()
        Q = self.query(inp)
        K = self.key(inp)
        V = self.value(inp)

        Q = Q.view(batch_size, seq_len, self.heads, self.head_dim).permute(0, 2, 1, 3)
        K = K.view(batch_size, seq_len, self.heads, self.head_dim).permute(0, 2, 1, 3)
        V = V.view(batch_size, seq_len, self.heads, self.head_dim).permute(0, 2, 1, 3)

        scores = torch.matmul(Q, K.transpose(-2, -1)) / (self.head_dim ** 0.5)
        attn = F.softmax(scores, dim=-1)
        attn = self.dropout(attn)

        out = torch.matmul(attn, V)
        out = out.permute(0, 2, 1, 3).contiguous().view(batch_size, seq_len, embed_dim)
        out = self.out(out)
        return out
```

ClassificationHead

models > classification_head.py

```python
import torch.nn as nn

class ClassificationHead(nn.Module):
    def __init__(self, embed_dim, classes, dropout=0.1):
        super(ClassificationHead, self).__init__()
        self.fc1 = nn.Linear(embed_dim, embed_dim // 2)
        self.activation = nn.GELU()
        self.dropout = nn.Dropout(dropout)
        self.fc2 = nn.Linear(embed_dim // 2, classes)

    def forward(self, inp):
        x = self.fc1(inp)
        x = self.activation(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

TransformerBlock

```python
import torch.nn as nn
from .attention import Attention

class TransformerBlock(nn.Module):
    def __init__(self, embed_dim, mlp_dim, heads, dropout=0.1):
        super(TransformerBlock, self).__init__()
        self.norm1 = nn.LayerNorm(embed_dim)
        self.attention = Attention(embed_dim, heads, dropout)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.ff = nn.Sequential(
            nn.Linear(embed_dim, mlp_dim),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(mlp_dim, embed_dim),
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, inp):
        x = self.norm1(inp)
        x = inp + self.dropout(self.attention(x))
        x = self.norm2(x)
        x = x + self.dropout(self.ff(x))
        return x
```

Transformer

```python
import torch.nn as nn
from .transformer_block import TransformerBlock

class Transformer(nn.Module):
    def __init__(self, embed_dim, mlp_dim, layers, heads, dropout=0.1):
        super(Transformer, self).__init__()
        self.blocks = nn.ModuleList([
            TransformerBlock(embed_dim, mlp_dim, heads, dropout)
            for _ in range(layers)
        ])

    def forward(self, inp):
        x = inp
        for block in self.blocks:
            x = block(x)
        return x
```

Vision Tranformer

```python
from .classification_head import ClassificationHead

class VisionTransformer(nn.Module):
    def __init__(self, input_size, patch_size, max_len, heads, classes, layers, embed_dim, mlp_dim, channels=3, dropout=0.1):
        super(VisionTransformer, self).__init__()
        self.patch_size = patch_size
        self.num_patches = (input_size // patch_size) ** 2
        patch_dim = channels * patch_size * patch_size

        self.patch_to_embedding = nn.Linear(patch_dim, embed_dim)
        self.cls_token = nn.Parameter(torch.randn(1, 1, embed_dim))
        self.pos_embedding = nn.Parameter(torch.randn(1, self.num_patches + 1, embed_dim))
        self.transformer = Transformer(embed_dim, mlp_dim, layers, heads, dropout)
        self.cls_head = ClassificationHead(embed_dim, classes, dropout)

    def forward(self, inp):
        batch_size = inp.size(0)
        x = inp.unfold(2, self.patch_size, self.patch_size).unfold(3, self.patch_size, self.patch_size)
        x = x.contiguous().view(batch_size, self.num_patches, -1)
        x = self.patch_to_embedding(x)
        cls_tokens = self.cls_token.expand(batch_size, -1, -1)
        x = torch.cat([cls_tokens, x], dim=1)
        x = x + self.pos_embedding
        x = self.transformer(x)
        cls_output = x[:, 0]
        out = self.cls_head(cls_output)
        return out
```

III. Architecture model and Evaluation

1. Model Architecture Summary

1. Overview of Vision Transformer (ViT)

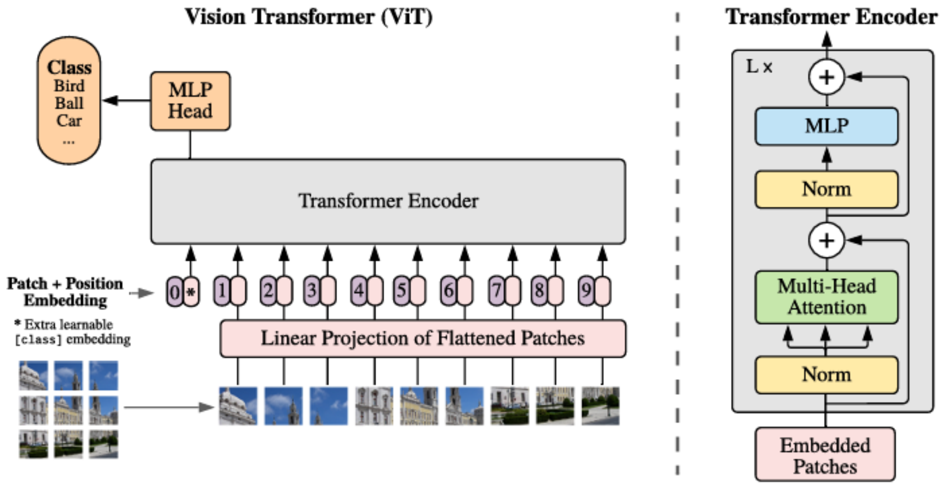
The Vision Transformer (ViT) applies Transformer architecture to image classification. It replaces convolutional layers with a sequence-based processing of image patches. ViT includes:

- **Patch Embedding**: Splits input image into fixed-size patches, flattens and linearly projects them to embeddings.
- **Transformer Encoder**: A stack of blocks, each with Multi-Head Self-Attention (MHSA), MLP, LayerNorm, and residual connections.
- **Classification Head**: Uses a special `[CLS]` token whose final representation is passed through an MLP to predict class labels

2. Implementation Highlights

- **Patch Embedding**:
  Each image is divided into patches (e.g., 4×4), flattened, then projected into a fixed-size vector (embedding dimension).
- **Positional Encoding**:
  Learnable position vectors are added to preserve spatial order, including a learnable `[CLS]` token.
- **TransformerBlock**:
  Each block contains:
  - Pre-LayerNorm → Multi-Head Attention → residual
  - Pre-LayerNorm → Feedforward MLP → residual
- **Transformer Encoder**:
  Stacks multiple TransformerBlocks (e.g., 6–12 layers) for deep feature learning.
- **Classification Head**:
  A two-layer MLP applied to the `[CLS]` output to produce final logits.

3. Diagram



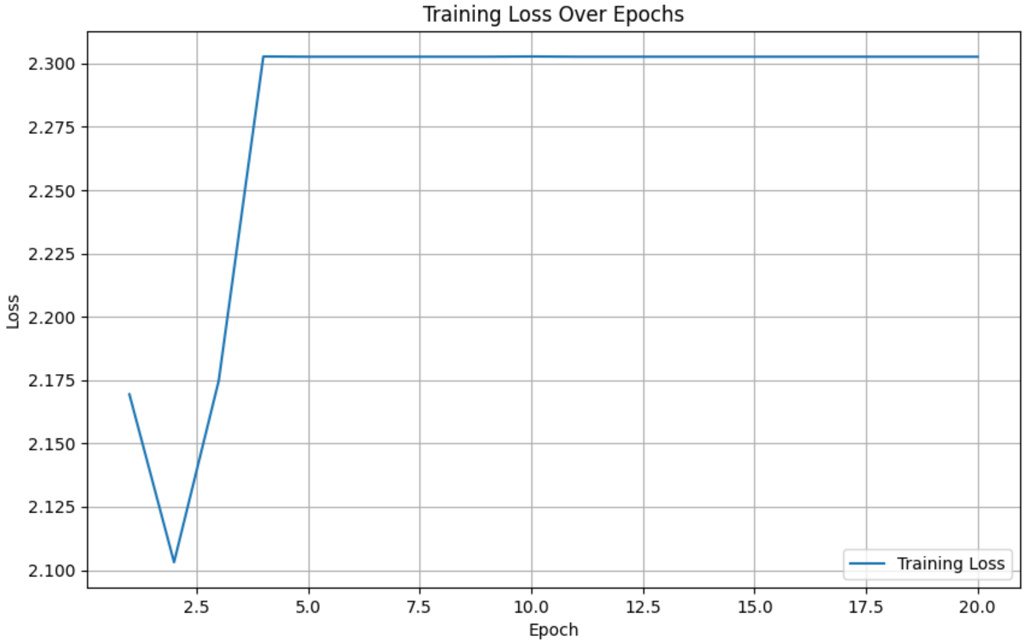
Hyperparameter Settings

Experiment	Patch Size	Embed Dim	MLP Dim	Heads	Layers	Dropout	Epochs	Top-1 Accuracy (%)
Exp_1	4	256	512	8	6	0.1	20	10.0
Exp_2	8	128	256	16	12	0.1	20	10.0

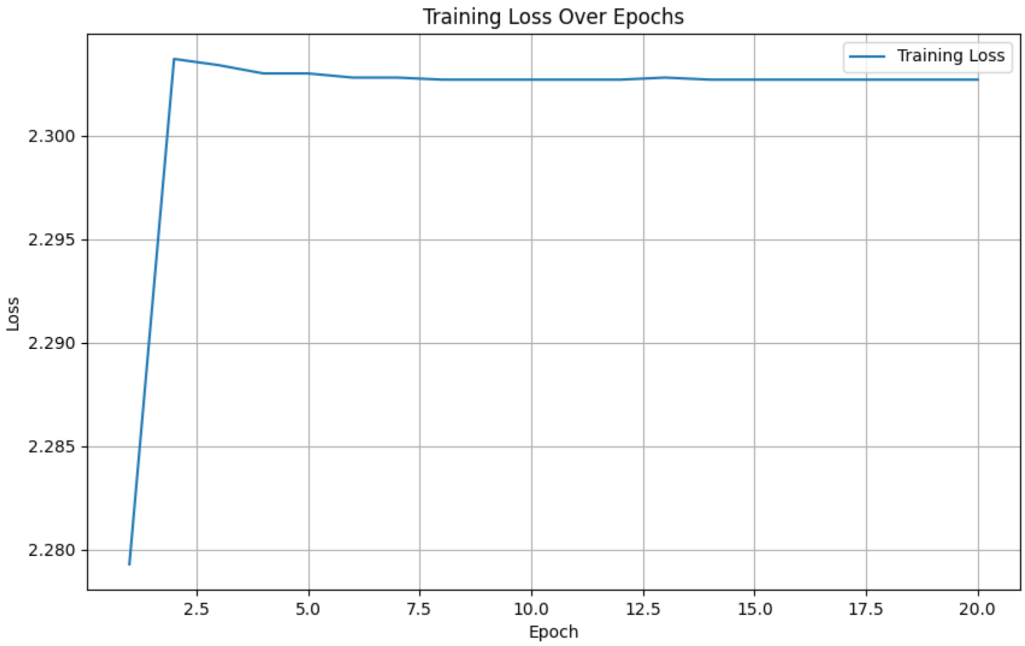
2. Visualization

2.1 Training Loss Over Epochs

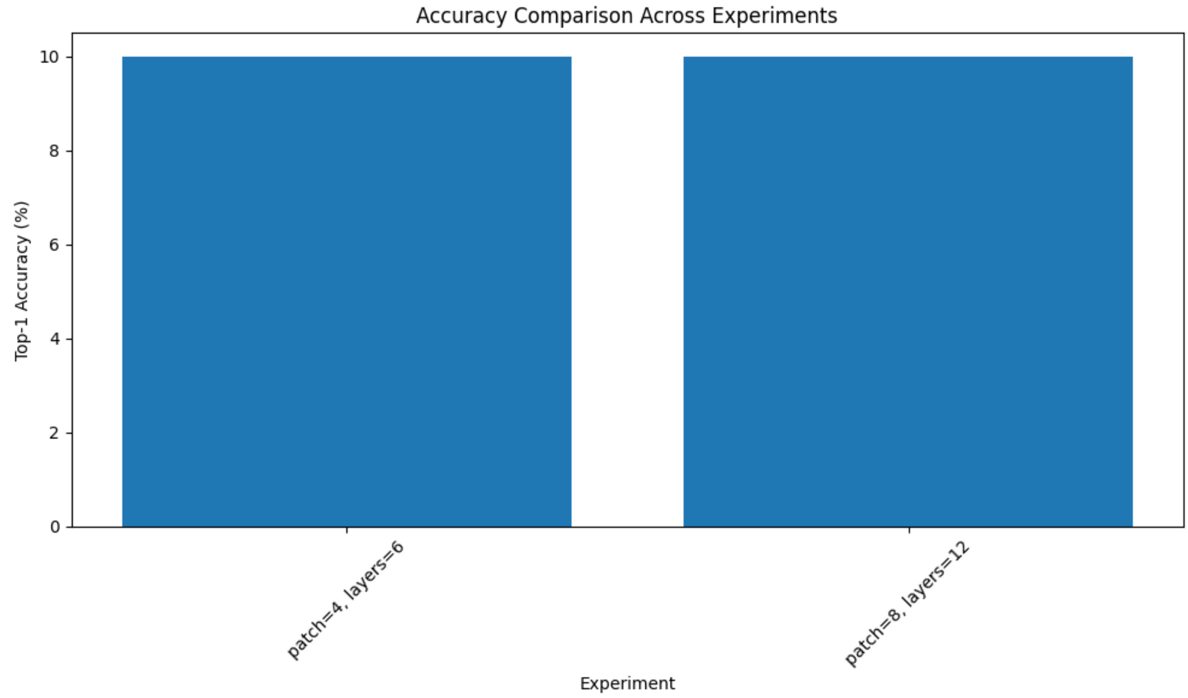
Experience 1:


Training Loss Over Epochs
Loss
2.300
2.275
2.250
2.225
2.200
2.175
2.150
2.125
2.100
Epoch
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Training Loss

Eperience 2:


Training Loss Over Epochs
Loss
2.300
2.295
2.290
2.285
2.280
Epoch
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Training Loss

2.2 Accuracy Comparison



3. Evaluation

- **Dataset**: CIFAR-10 (50,000 train / 10,000 test)
- **Metric**: Top-1 Accuracy
- **Optimizer**: Adam
- **Loss**: CrossEntropyLoss
- **Epochs**: 20

Observations:

All two experiments—regardless of the configuration—report a **Top-1 Accuracy of only 10.0%**, which is **equivalent to random guessing** on a dataset with 10 classes (e.g., CIFAR-10).

Observations:

- **Experiment 1** (Patch Size 4, Embed Dim 256, MLP 512, Heads 8, Layers 6): both gave exactly 10.0% accuracy.
- **Experiment 2** (Patch Size 8, Embed Dim 128, MLP 256, Heads 16, Layers 12): also yielded 10.0% accuracy.

Interpretation:

- The model has **not learned** to generalize; it performs no better than random.
- This strongly suggests an issue in:
  - Training loop (e.g., learning rate too low/high)

- Model not updating weights (e.g., optimizer setup)
- Dataset loading or label mismatch
- Incorrect forward pass or frozen layers
- Loss function misbehavior (e.g., not being minimized)

VI. Implementation details

Function & Class Implementation Summary with Usage Explanation

1. MultiheadAttention – models/attention.py

Attention – models/attention.py

Purpose: Multi-head self-attention layer for sequence modeling.
Usage:

```
attn = Attention(embed_dim=256, heads=8)

output = attn(input_tensor)  # input_tensor: [batch_size, seq_len, embed_dim]
```

Logic:
• Project input into Q, K, V using linear layers
• Reshape and split into multiple heads
• Compute scaled dot-product attention
• Apply dropout and merge heads
• Project back to original embedding dimension

2. TransformerBlock – models/transformer_block.py

Purpose: A complete encoder block that integrates attention, feed-forward layers, and normalization.

Usage:

```
block = TransformerBlock(embed_dim=256, mlp_dim=512, heads=8)
output = block(x)
```

Flow:

Input → LayerNorm → Attention → Residual Add → LayerNorm → FeedForward → Residual Add

Implementation Notes:

- FeedForward: 2 linear layers with ReLU/GELU
- Dropout is applied after attention and MLP
- Maintains original sequence length and dimension

3. Transformer – models/transformer.py

Purpose: Stacks multiple TransformerBlocks sequentially.

Usage:

```
encoder = Transformer(embed_dim=256, layers=6, heads=8)
output = encoder(x)
```

Behavior:

- For each layer:
- x = TransformerBlock_i(x)
- Maintains dimension: (batch, seq_len, embed_dim)
- Adds capacity and abstraction with each layer.

4. ClassificationHead – models/classification_head.py

Purpose: Projects [CLS] token to class logits for classification.

Usage:

```
head = ClassificationHead(embed_dim=256, classes=10)
logits = head(cls_token)  # (batch, classes)
```

Steps:

- Linear → GELU → Dropout → Linear → Softmax (optional during inference)

5. VisionTransformer – models/vision_transformer.py

Purpose: Full ViT model, includes patch embedding, transformer encoder, and classification head.

Usage:

```
model = VisionTransformer(
    inp_channels=3, patch_size=4, max_len=100,
    heads=8, classes=10, layers=6, embed_dim=256, mlp_dim=512, dropout=0.1
)
logits, hidden_states = model(images)
```

Pipeline:

Image → Patch Embedding → Positional Encoding → Transformer Encoder → Classification Head

Design Choices:

- Learnable positional embeddings
- Prepend [CLS] token
- Outputs logits + intermediate token sequence

6. train_model() – training/train.py

Purpose: Main training loop.

Usage:

```
train_model(config)
```

Key Operations:

- Load CIFAR-10 dataset
- Instantiate model and optimizer
- Train for N epochs, save logs to `.csv/.json`
- Use accuracy and loss as metrics
- Log best-performing model config

7. plot_loss() – utils/visualize.py

Purpose: Visualize training loss across epochs.

Usage:

```
plot_loss("logs/experiment_xx.csv", "loss_plot.png")
```

Workflow:

- Read CSV
- Filter valid epoch rows
- Plot loss vs. epoch
- Save as PNG

8. plot_accuracy_comparison() – utils/visualize.py

Purpose: Compare multiple experiment accuracies.

Usage:

```
plot_accuracy_comparison(["logs/exp1.json", "logs/exp2.json"])
```

Logic:

- Read `accuracy` and hyperparams from `.json`
- Draw bar plot with experiment labels (patch size, layers)
- Useful for side-by-side performance review