ECE 1512 Project B Part 2

Yingshun Lu 1006029049

Minghao Ma 1010800536

1. Summary Document on VILA[1]

1.1 Key Points

Focus: VILA investigates pre-training strategies for VLMs that better align the visual and textual modalities.

Objective: To impart vision capability to the LLMs while retaining functionality typical of text-only LLMs.

Architecture: Auto-regressive architecture, taking the visual tokens as equivalent to textual tokens makes the model flexible and unified for taking multi-model inputs.

Applications: Excels in vision-language tasks like VQA, caption generation, and multi-image reasoning.

1.2 Technical Contributions

Pre-training Strategies:

That demonstrates fine-tuning in visual pre-training of LLMs as highly critical for deep embedding alignment and in-context learning.

Interleaving image-text datasets enjoy the benefits of better alignment and minimized degradation in the text-only capability compared to plain image-text pairs.

Data Blending:

It reintroduces text-only instruction data during supervised fine-tuning to regain degraded text-only capabilities and improve VLM task performance.

This paper proposes a mixture of interleaved datasets with image-text pairs to enhance the diversity and also the accuracy of downstream tasks.

Performance:

Consistently outperforms state-of-the-art models like LLaVA-1.5 on several benchmarks.

Demonstrates improved robust multiple-image reasoning and better world knowledge retention.

Efficiency:

Employs scalable approaches employing resolution adjustments and lightweight projection layers that result in improved performance with better cost trade-offs.

1.3 Areas for Improvement

Scaling:
Very limited pre-training dataset of 50M images compared to others' work in billion-scale datasets.
Further improvements may be possible by expanding the training dataset.

Instruction Dataset Quality:
Though successful, there is room for improving the diversity and quality of the prompts that constitute the instruction-tuning dataset.

Edge Deployment:
While it might be deployable with systems such as Jetson Orin, the full size of this model alone can already be a challenging prospect in environments starved for computational resources.

Generalization:
While VILA still seems to keep up with competitive text-only performance on this end, smaller models significantly degrade in performance, indicating room for improvement in preserving text-only abilities during pre-training.

2. Efficiency Optimization Report: Dynamic Token Pruning in Transformer Models

2.1. Introduction

Transformer models are very expensive due to heavy computations of self-attention and feed-forward layers at the token level. Reducing this while keeping the accuracy intact has become one of the most important factors that will help deploy efficient models into production. Thus, finding out bottlenecks is followed by optimization identification and evaluation to make up the core part of efficiency enhancement in the course of this report with the aid of Dynamic Token Pruning on the model in question.

2.2 Efficiency Bottleneck Identification

2.2.1 Profiling Results

We have profiled a baseline transformer model in order to get an idea about which are the main bottlenecks concerning efficiency. Results are summarised below:

Major Bottlenecks:

Self-Attention Layers: Dominant contributor of FLOPs and computational time.

Feedforward Layers (Linear): Responsible for secondary computational costs.

Baseline Model FLOPs and Time:

FLOPs: **12.908G**

CUDA Time: **12.490ms**

2.2.2 Conclusion

This inefficiency primarily comes from redundant computations over the input tokens that make minor contributions to the output of the whole model. In our attempt to improve it, dynamic token pruning focuses on better usage of tokens.

2.3 Optimization Method: Dynamic Token Pruning

2.3.1 Overview

Dynamic Token Pruning is based on adaptively removing irrelevant tokens computed by saliency from computation. This reduces the input sequence length dynamically during the inference process, saving considerably on FLOPs and time

2.3.2 Algorithm

1. **Compute Saliency Scores:** Each token's importance is estimated using its L2 norm.
2. **Generate Token Mask:** Saliency scores less than some threshold are pruned off the tokens.

3. **Layer-Wise Pruning:** Pass the remaining tokens to the next transformer block.
4. **Final Classification:** Pool the outputs of all active tokens.

2.3.3Pseudo Code

```
class PrunedTransformerEncoder(nn.Module):
    def forward(self, src):
        keep_tokens = torch.ones(src.shape[:2], device=src.device).bool()
        for i, layer in enumerate(self.layers):
            saliency = self.token_pruning.compute_saliency(src)
            keep_tokens = keep_tokens & (saliency > self.token_pruning.saliency_threshold)
            src = layer(src, keep_tokens=keep_tokens)
        return src
```

2.3.4 Implementation Details

Saliency Score: Computed as the L2 norm of each token vector.

Token Mask: Dynamically updated across layers, preserving only the most relevant tokens.

2.4 Results and Evaluation

2.4.1 Accuracy

The optimized model maintains the same accuracy as the baseline:

Baseline Model Accuracy: **52.5%**

Pruned Model Accuracy: **52.5%**

2.4.2 FLOPs and Time Comparison

Model	FLOPs	CUDA Time	FLOPs Reduction	Time Reduction
Baseline	12.908G	12.490ms	-	-
Pruned	8.874G	9.968ms	**31.2%**	**20.2%**

The pruning method significantly reduces FLOPs and computational time while preserving accuracy.

2.4.3 Profiling Results

Baseline Model

- FLOPs: **12.908G**
- CUDA Memory Usage: **72.02MB**
- CUDA Time: **12.490ms**

Pruned Model

- FLOPs: **8.874G**
- CUDA Memory Usage: **72.02MB**
- CUDA Time: **9.968ms**

2.4.4 Efficiency Analysis

Dynamic Token Pruning successfully reduces computation while maintaining the accuracy of the model. Profiling indicates that FLOPs and execution time reductions are most prominent in the self-attention layers.

2.5 Discussion and Considerations

2.5.1 Strengths

Efficiency Gains:

Much smaller FLOPs and execution time, over 31.2% reduction in FLOPs.

Preserves the accuracy of a binary classification task.

Scalability:

The pruning is dynamic, layer-wise for the integration of large-scale models.

2.5.2 Limitations

Static Threshold: A static setting of the saliency threshold may not generalize so well for a wide range of datasets.

Evaluation on Toy Data: While the accuracy may be fine on simulated tasks, this may need further validation on real-world datasets.

2.6. Conclusion

This report now presents the possibility of Dynamic Token Pruning in optimizing transformer efficiency. The dynamic pruning of the length of a sequence using saliency-based pruning produced:

- Significant reductions both in FLOPs and in execution time.
- Maintenance of accuracy on a binary classification task.

Future work will be conducted on the validation of this approach with larger datasets, as well as exploring adaptive thresholds for saliency computation.

Reference:

[1] Ji Lin, Hongxu Yin, Wei Ping, Yao Lu, Pavlo Molchanov, Andrew Tao, Huizi Mao, Jan Kautz, Mohammad Shoeybi, and Song Han. Vila: On pre-training for visual language models, 2023.

Appendix:

Github links:

https://github.com/ADglory/ECE1512_2024F_ProjectB_PartB_Repo_YingshunLu_Minghao-Ma

Codes:

```
1. Introduce necessary dependencies
"""

!pip install fvcore

import torch
import torch.nn as nn
import torch.nn.functional as F
from fvcore.nn import FlopCountAnalysis, flop_count_table
import torch.profiler as profiler
from torch.utils.data import DataLoader, TensorDataset
from copy import deepcopy

"""2. Token Saliency computing module"""

class TokenSaliency(nn.Module):
    """
    Compute saliency scores for visual tokens based on their contribution.
    """
    def __init__(self, method="norm"):
        super(TokenSaliency, self).__init__()
        self.method = method

    def forward(self, tokens):
        """
        Args:
            tokens: Tensor of shape (B, N, D), where
                    B = Batch size,
                    N = Number of tokens,
                    D = Dimension of each token.
        Returns:
            saliency_scores: Tensor of shape (B, N), saliency scores for each token.
        """
        if self.method == "norm":
            saliency_scores = tokens.norm(dim=-1)  # Use L2 norm
        else:
            raise ValueError(f"Unsupported method: {self.method}")

        return saliency_scores

"""3. Adaptive Token pruning module"""

class AdaptiveTokenPruning(nn.Module):
    def __init__(self, saliency_threshold=0.5):
        super(AdaptiveTokenPruning, self).__init__()
        self.saliency_threshold = saliency_threshold

    def forward(self, x):
        """
        Compute token saliency and generate a pruning mask.
        """
        saliency_scores = self.compute_saliency(x)
        keep_tokens = saliency_scores > self.saliency_threshold
        return keep_tokens

```

```python
    def compute_saliency(self, x):
        """
        Compute saliency scores (e.g., L2 norm across embedding dimensions).
        """
        saliency_scores = x.norm(p=2, dim=-1)  # Shape: (batch_size, seq_len)
        return saliency_scores

"""4. Pruned Transformer Encoder"""

class PrunedTransformerEncoder(nn.Module):
    """
    Transformer encoder with token pruning capability.
    """
    def __init__(self, encoder_layer, num_layers, saliency_threshold=0.5):
        super().__init__()
        self.layers = nn.ModuleList([deepcopy(encoder_layer) for _ in range(num_layers)])
        self.token_pruning = AdaptiveTokenPruning(saliency_threshold=saliency_threshold)

    def forward(self, src):
        """
        Forward pass with token pruning.
        Args:
            src: Input tensor of shape (batch_size, seq_len, d_model).
        Returns:
            Output tensor after pruning.
        """
        batch_size, seq_len, d_model = src.shape
        keep_tokens = torch.ones((batch_size, seq_len), device=src.device).bool()  # Initialize with all True

        for i, layer in enumerate(self.layers):
            # Calculate saliency scores
            saliency = self.token_pruning.compute_saliency(src)

            # Update keep_tokens
            new_keep_tokens = (saliency > self.token_pruning.saliency_threshold)
            keep_tokens = keep_tokens & new_keep_tokens  # Retain the accumulated crop state

            # Dynamically crop the input tensor
            pruned_src = []
            pruned_keep_tokens = []

            for batch_idx in range(batch_size):
                active_token_indices = keep_tokens[batch_idx].nonzero(as_tuple=True)[0]
                pruned_src.append(src[batch_idx, active_token_indices])
                pruned_keep_tokens.append(keep_tokens[batch_idx, active_token_indices])

            # Update src and keep_tokens with the trimmed tensor
            src = torch.nn.utils.rnn.pad_sequence(pruned_src, batch_first=True)
            keep_tokens = torch.nn.utils.rnn.pad_sequence(pruned_keep_tokens, batch_first=True)

            # Print debugging information
            print(f"Layer {i}: Active tokens per batch = {[len(t) for t in pruned_src]}")

            # Pass the clipped tensor to the next layer
            src = layer(src)
```

```
        return src

"""5. FLOPs evaluation tool"""

from fvcore.nn import FlopCountAnalysis, flop_count_table

def calculate_dynamic_flops(model, x, keep_tokens):
    """
    Calculate FLOPs dynamically based on active tokens.
    Args:
        model: The pruned Transformer model.
        x: Input tensor of shape (batch_size, seq_len, d_model).
        keep_tokens: Boolean tensor indicating active tokens for the pruned model.
    """
    #  Get the maximum number of active tokens
    active_tokens = keep_tokens.sum(dim=1).max().item()
    x = x[:, :active_tokens, :]  # Crop to active Token
    flops = FlopCountAnalysis(model, x)
    print(flop_count_table(flops))

"""6. Memory usage evaluation tool"""

def profile_memory_and_time_safe(model, input_tensor):
    """
    Profile memory and time for the given model and input.
    Args:
        model: PyTorch model to profile.
        input_tensor: Tensor input to pass through the model.
    """
    try:
        with torch.profiler.profile(
            activities=[
                torch.profiler.ProfilerActivity.CPU,
                torch.profiler.ProfilerActivity.CUDA,
            ],
            record_shapes=True,
            profile_memory=True,
            with_stack=False,  # Disable stack tracing to reduce possible conflicts
        ) as prof:
            model(input_tensor)  # Perform model forward propagation
        print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10))
    except RuntimeError as e:
        print(f"Profiler failed: {e}")

"""7. Model accuracy evaluation"""
```

```python

def evaluate_model_accuracy(model, train_data, train_labels, test_data, test_labels):
    """
    Train and evaluate model accuracy on a toy dataset.
    Args:
        model: PyTorch model to evaluate.
        train_data, train_labels, test_data, test_labels: Dataset tensors.
    """
    # Make sure the shape of the label is 1D
    train_labels = train_labels.squeeze()
    test_labels = test_labels.squeeze()

    model.train()

    # Dataset and DataLoader
    train_dataset = TensorDataset(train_data, train_labels)
    test_dataset = TensorDataset(test_data, test_labels)
    train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=16)

    # Optimizer and Loss
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    loss_fn = nn.CrossEntropyLoss()

    # Training loop
    for epoch in range(5):
        for inputs, labels in train_loader:
            inputs, labels = inputs.cuda(), labels.cuda()
            optimizer.zero_grad()
            outputs = model(inputs)
            outputs = outputs.mean(dim=1)
            loss = loss_fn(outputs, labels)
            loss.backward()
            optimizer.step()

    # Evaluation loop
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.cuda(), labels.cuda()
            outputs = model(inputs)
            outputs = outputs.mean(dim=1)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```python

    accuracy = 100 * correct / total
    print(f"Accuracy: {accuracy:.2f}%")

from fvcore.nn import FlopCountAnalysis, flop_count_table

def evaluate_pruned_model(baseline_model, pruned_model, test_data):
    """
    Compare baseline and pruned models in terms of FLOPs and active token efficiency.
    Args:
        baseline_model: The baseline Transformer model.
        pruned_model: The pruned Transformer model.
        test_data: Sample input tensor for efficiency evaluation.
    """
    print("=== Baseline Model Efficiency ===")
    flops_baseline = FlopCountAnalysis(baseline_model, test_data)
    print(flop_count_table(flops_baseline))

    print("\n=== Pruned Model Efficiency ===")
    # Assuming the PrunedTransformerEncoder dynamically prunes tokens
    with torch.no_grad():
        pruned_outputs = pruned_model[0](test_data)  # Get the intermediate result of PrunedTransformer
        active_tokens = pruned_outputs.shape[1]  # The number of valid tokens remaining
        flops_pruned = FlopCountAnalysis(pruned_model, test_data[:, :active_tokens, :])
        print(flop_count_table(flops_pruned))

"""8. Prepare the data set"""

def prepare_data():
    """
    Prepare simulated toy dataset for training and testing.
    Returns:
        train_data, train_labels, test_data, test_labels
    """
    train_data = torch.rand(1000, 128, 512).cuda()  # 1000 samples, 128 tokens, 512 dimensions
    train_labels = torch.randint(0, 2, (1000,), dtype=torch.long).cuda()  # Make sure it's a 1D long integral tensor
    test_labels = torch.randint(0, 2, (200,), dtype=torch.long).cuda()
    test_data = torch.rand(200, 128, 512).cuda()  # 200 samples for testing
    return train_data, train_labels, test_data, test_labels

train_data, train_labels, test_data, test_labels = prepare_data()
print(train_data.shape, train_labels.shape)

"""9. Define the model"""


# Keep the SimpleClassifierHead class
class SimpleClassifierHead(nn.Module):
    """
    A simple classification head for transformer output.
    """
    def __init__(self, input_dim, num_classes):
        super(SimpleClassifierHead, self).__init__()
        self.fc = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        return self.fc(x)
```

```python


# TransformerEncoderLayerWithPruning class
class TransformerEncoderLayerWithPruning(nn.TransformerEncoderLayer):
    """
    A customized TransformerEncoderLayer that supports dynamic token skipping.
    """
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def forward(self, src, src_mask=None, src_key_padding_mask=None, keep_tokens=None):
        """
        Args:
            src: Input tensor of shape (batch_size, seq_len, d_model).
            keep_tokens: Boolean tensor of shape (batch_size, seq_len).
        """
        if keep_tokens is not None:
            # Dynamically crop the tensor shape, keeping only tokens marked True
            batch_size, seq_len, d_model = src.shape
            active_indices = keep_tokens.nonzero(as_tuple=True)  # Gets the index of active tokens
            max_active_tokens = keep_tokens.sum(dim=1).max().item()  # Maximum number of active tokens
            pruned_src = torch.zeros(batch_size, max_active_tokens, d_model, device=src.device)

            for batch_idx in range(batch_size):
                active_token_indices = keep_tokens[batch_idx].nonzero(as_tuple=True)[0]
                pruned_src[batch_idx, :len(active_token_indices)] = src[batch_idx, active_token_indices]

            src = pruned_src  # Update to the clipped tensor

        # A forward method that passes the trimmed tensor to the parent class
        return super().forward(src, src_mask, src_key_padding_mask)





# create_models function
def create_models():
    """
    Create baseline and pruned Transformer models, each with a classification head.
    Returns:
        baseline_model, pruned_model
    """
    num_classes = 2  # dichotomy

    # Baseline model
    baseline_encoder = nn.TransformerEncoderLayer(d_model=512, nhead=8)
    baseline_transformer = nn.TransformerEncoder(baseline_encoder, num_layers=2).cuda()
    baseline_model = nn.Sequential(
        baseline_transformer,
        SimpleClassifierHead(input_dim=512, num_classes=num_classes).cuda()
    )
```

```python

    # Pruned model
    pruned_encoder = TransformerEncoderLayerWithPruning(d_model=512, nhead=8)
    pruned_transformer = PrunedTransformerEncoder(pruned_encoder, num_layers=2, saliency_threshold=13.0).cuda()
    pruned_model = nn.Sequential(
        pruned_transformer,
        SimpleClassifierHead(input_dim=512, num_classes=num_classes).cuda()
    )

    return baseline_model, pruned_model

baseline_model, pruned_model = create_models()
print(baseline_model)
print(pruned_model)

input_tensor = torch.rand(16, 128, 512).cuda()

pruned_outputs = pruned_model[0](input_tensor)
print(f"Output shape after pruning: {pruned_outputs.shape}")

saliency_scores = pruned_model[0].token_pruning.compute_saliency(input_tensor)
print(f"Saliency scores range: {saliency_scores.min().item()} - {saliency_scores.max().item()}")

keep_tokens = pruned_model[0].token_pruning.compute_saliency(input_tensor) > pruned_model[0].token_pruning.saliency_threshold
print(f"Keep tokens mask (sample batch): {keep_tokens[0].cpu().numpy()}")

"""10. Evaluate model accuracy"""

def compare_models_accuracy(baseline_model, pruned_model, train_data, train_labels, test_data, test_labels):
    """
    Compare accuracy of baseline and pruned models.
    """
    print("\n=== Baseline Model Accuracy ===")
    evaluate_model_accuracy(baseline_model, train_data, train_labels, test_data, test_labels)

    print("\n=== Pruned Model Accuracy ===")
    evaluate_model_accuracy(pruned_model, train_data, train_labels, test_data, test_labels)

compare_models_accuracy(baseline_model, pruned_model, train_data, train_labels, test_data, test_labels)

"""11.FLOPs versus memory performance"""

from torch.profiler import profile, ProfilerActivity
from fvcore.nn import FlopCountAnalysis, flop_count_table

def calculate_dynamic_flops_and_profile(pruned_model, input_tensor):
    """
    Calculate dynamic FLOPs and memory usage for the pruned model.
    Args:
        pruned_model: Model with dynamic token pruning.
        input_tensor: Input tensor.
    """
    # Dynamic computing FLOPs
    print("\n=== Pruned Model ===")
    flops_pruned = FlopCountAnalysis(pruned_model, input_tensor)
    print(flop_count_table(flops_pruned))

    # Dynamic profile memory and time
    with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA]) as prof:
        _ = pruned_model(input_tensor)
    print(prof.key_averages().table(sort_by="cuda_time_total"))

```

```python
def compare_efficiency(baseline_model, pruned_model):
    """
    Compare FLOPs and memory usage for baseline and pruned models.
    Args:
        baseline_model, pruned_model: Models to compare.
    """
    input_tensor = torch.rand(16, 128, 512).cuda()  # Simulated input: batch size=16, tokens=128, dim=512

    # FLOPs and performance evaluation of Baseline Model
    print("\n=== Baseline Model ===")
    flops_baseline = FlopCountAnalysis(baseline_model, input_tensor)
    print(flop_count_table(flops_baseline))
    profile_memory_and_time_safe(baseline_model, input_tensor)

    # Pruned Model dynamic FLOPs and performance evaluation
    calculate_dynamic_flops_and_profile(pruned_model, input_tensor)

compare_efficiency(baseline_model, pruned_model)
```