Model Pruning

Why We Use Pruning Before Knowledge Distillation

Knowledge Distillation (KD)

KD is when a large model (teacher) transfers its "knowledge" to a smaller model (student) by training the student on:

- The original dataset
- The soft predictions (logits) from the teacher model

Model Purning is the process of removing redundant or less important parameters (like weights or channels) in a trained deep learning model. The goal is to:

- Reduce model size
- Speed up inference
- Reduce memory usage
- Retain as much accuracy as possible

In YOLO, pruning can mean removing unnecessary filters in the convolution layers, which reduces complexity without harming performance much.

Step 1: Load the Trained YOLO Model

```
from ultralytics import YOLO

# Load the YOLO model

model = YOLO('D:/prodigal-4/yolov12_quant_api/models/yolo12n.pt')
```

This loads the full model (yolo12n.pt) which has all original weights.

Step 2: Apply Pruning to Reduce Weights

We manually removed unimportant filters by thresholding based on their **L1 norm** (importance).

```
# Access the PyTorch model inside
```

```
model_torch = model.model

# Apply global unstructured pruning

parameters_to_prune = []

for module in model_torch.modules():

    if isinstance(module, nn.Conv2d):

        parameters_to_prune.append((module, 'weight'))

# Apply L1 unstructured pruning globally with 30% sparsity

prune.global_unstructured(

    parameters_to_prune,

    pruning_method=prune.L1Unstructured,

    amount=0.3,  # 30% of weights pruned globally

)
```

This prunes 30% of weights in each Conv2D layer.

Step 3: Remove Reparametrization and Save Pruned Weights

```
# Optional: Remove pruning re-param so weights are fixed

for module, _ in parameters_to_prune:

    prune.remove(module, 'weight')

# Save the pruned model

torch.save(model_torch.state_dict(), 'yolo12n_pruned.pth')

print("Pruned model saved as 'yolo12n_pruned.pth'")
```

- prune.ewmove() removes hooks and makes pruning permanent.
- The pruned weights are saved as yolo12n_pruned.pth

How Pruning Helps KD

- Pruning gives us a **leaner teacher** model (smaller, faster).
- This model can guide the student with distilled knowledge but with less computational cost.
- It removes unnecessary complexity, so KD learns only the **essential features**.

Knowledge Distillation for YOLOv12.

Overview of Knowledge Distillation

Knowledge distillation is a model compression technique where a smaller, less complex model (the student) is trained to replicate the behavior of a larger, more powerful model (the teacher). The objective is to retain much of the performance of the teacher while benefiting from the efficiency of the student model. This is especially useful for deploying models on resource-constrained devices.

In this project, knowledge distillation is applied to the YOLOv12n object detection model. The teacher model is a full-sized version of YOLOv12n, while the student is a pruned variant. The distillation is supervised using the Mean Squared Error (MSE) loss between the student and teacher outputs, in combination with the student's original loss.

KD in Object Detection with YOLO Models

Applying KD to object detection frameworks like YOLO requires specialized methods due to the unique characteristics of detection tasks, such as handling both classification and localization outputs. Researchers have developed KD techniques tailored for YOLO models to effectively transfer knowledge from teacher to student.

🔍 How It Works:

- The **teacher model** is typically a high-performing, deep neural network trained on a dataset.
- The **student model** is usually smaller and more efficient, intended for faster inference or deployment on edge devices.

- During training, the student not only learns from the actual labels (hard targets) but also tries to mimic the **soft outputs** (probabilistic predictions) of the teacher.

💡 Why Use It?

- To **compress** large models while maintaining performance.
- To **speed up** inference for real-time applications.
- To **reduce memory and power** requirements—important for mobile and embedded systems

🧠 Loss Function

The total loss typically combines:

1. **Hard loss**: between the student's output and ground truth.
2. **Soft loss**: between the student's and teacher's logits (often using KL divergence or MSE).

Formula:

$$\text{Total Loss} = \alpha * \text{Hard Loss} + \beta * \text{Soft Loss}$$

Coding Knowledge Distillation in Python

Step 1: Import Required Libraries and Define KD Trainer

```python
from ultralytics import YOLO
import torch
import torch.nn.functional as F


# --- Custom Knowledge Distillation Trainer ---
def build_kd_trainer_class(base_trainer_cls):
    class KDTrainer(base_trainer_cls):
        def __init__(self, overrides=None, _callbacks=None, teacher_path=None):
            self.teacher_path = teacher_path
            super().__init__(overrides=overrides, _callbacks=_callbacks)

        def setup_model(self):
            super().setup_model()

```

This step sets up the environment by importing necessary libraries. The function build_kd_trainer_class() dynamically creates a subclass called KDTrainer, which inherits from Ultralytics' base trainer. This allows the use of custom training behavior specific to knowledge distillation.

Step 2: Load Teacher Model and Freeze Weights

```python
def setup_model(self):
    super().setup_model()

    # Load teacher model
    assert self.teacher_path is not None, "teacher_path must be set!"
    self.teacher = YOLO(self.teacher_path).model
    self.teacher.eval()
    for p in self.teacher.parameters():
        p.requires_grad = False
```

The setup_model() method loads the teacher model from the specified path, sets it to evaluation mode (so that it does not update during training), and

freezes all parameters to avoid gradient computation. This ensures the teacher model serves purely as a reference for training the student.

Step 3: Compute Total Loss with KD

```
    def train_batch(self, batch):
        # Student forward pass
        student_out = self.model(batch['img'])
        loss, loss_items = self.criterion(student_out, batch)

        # Teacher forward pass (no grad)
        with torch.no_grad():
            teacher_out = self.teacher(batch['img'])

        # Distillation loss (L2)
        kd_loss = F.mse_loss(student_out[0], teacher_out[0])

        # Final loss
        total_loss = loss + 0.3 * kd_loss
        return total_loss, loss_items

return KDTrainer
```

During training, the student model's output is first passed through the original YOLO loss function to compute the standard detection loss. At the same time, the teacher model generates its predictions in evaluation mode, ensuring that no gradients are calculated for it. Then, a knowledge distillation loss is computed using the Mean Squared Error (MSE) between the outputs of the student and the teacher, specifically comparing their feature maps or logits. Finally, the total loss used to update the student model is a weighted combination of the original YOLO loss and the distillation loss, where the distillation loss is scaled by a factor of 0.3 to balance its influence during training.

Step 4: Initialize and Train the Model with KD

```python
# --- Train with KD ---
if __name__ == "__main__":
    # Paths
    pruned_student_pth = r"C:\Users\mohas\Downloads\yolov12_quant_api-main\yolov12_quant_api-main\models\yolo12n_pruned.pt"
    teacher_pt = r"C:\Users\mohas\Downloads\yolov12_quant_api-main\yolov12_quant_api-main\models\yolo12n.pt"
    data_yaml = r"C:\Users\mohas\Downloads\yolov12_quant_api-main\yolov12_quant_api-main\coco128\coco128\coco128.yaml"
    output_dir = r"C:\Users\mohas\Downloads\yolov12_quant_api-main\yolov12_quant_api-main\models"

    # Load pruned student model
    student_model = YOLO(pruned_student_pth)

    # Init KD Trainer
    overrides = {
        'model': pruned_student_pth,
        'data': data_yaml,
        'epochs': 5,
        'imgsz': 640,
        'batch': 8,
        'project': output_dir,
        'name': 'distilled_pruned_model',
        'pretrained': False,
    }

    # Train with KD
    # Load model to determine base trainer
    dummy_model = YOLO(pruned_student_pth)
    BaseTrainerClass = dummy_model._smart_load("trainer")

    # Create KDTrainer subclass dynamically
    KDTrainer = build_kd_trainer_class(BaseTrainerClass)

    # Train with KD
    trainer = KDTrainer(overrides=overrides, teacher_path=teacher_pt)
    trainer.train()
```

The script begins by defining the paths for the student model, teacher model, dataset configuration file, and the output directory. It then specifies training parameters using an overrides dictionary, which includes settings such as the number of epochs, image size, batch size, and model name—customized for the YOLO training process. Next, the appropriate base trainer class is dynamically retrieved from the YOLO model using the _smart_load("trainer") method. Finally, an instance of the KDTrainer class is created with the defined parameters and the teacher model path, and the training is initiated using the .train() method, which integrates both standard YOLO training and knowledge distillation.

Output :

Upload a Video

Choose File | No file chosen | Upload

Live Classification:

Output video saved to: `test(out_put)/output_...mp4`

The output from your knowledge distillation code is a distilled pruned YOLO model that combines the efficiency of model pruning with the performance retention of knowledge distillation. This resulting model is saved to the "distilled_pruned_model" directory within your specified output path and demonstrates effective person detection

Summary:

The provided code demonstrates a **Knowledge Distillation (KD)** process applied to a YOLOv12 model. Knowledge distillation involves training a smaller "student" model to mimic the behavior of a larger "teacher" model. This code implements a custom training procedure where the student model learns not only from the ground truth labels but also from the soft predictions (logits) of the teacher model. By combining the standard YOLO loss and a distillation loss computed as the Mean Squared Error (MSE) between the student and teacher outputs, the student model is able to retain much of the teacher model's performance while being more lightweight and efficient.

References for Further Reading

1. **Ultralytics YOLO Documentation**: Provides comprehensive guides on model deployment practices, including knowledge distillation techniques for YOLO models.
    - Ultralytics Knowledge Distillation Guide
2. **YOLOv5 Knowledge Distillation Implementation**: A GitHub repository demonstrating the implementation of distilling object detectors with fine-grained feature imitation on YOLOv5.
    - YOLOv5 Knowledge Distillation Repository
3. **Research on Knowledge Distillation for YOLO Models**: An academic paper discussing the application of knowledge distillation in optimizing YOLOv5 models.
    - Research Paper on Knowledge Distillation for YOLO

4. **Best Practices for Model Deployment**: Ultralytics' guide on deploying YOLO models, including insights into model optimization techniques like knowledge distillation.
    - Ultralytics Model Deployment Practices
5. **Comprehensive Guide to YOLOv5**: Official documentation providing detailed information on training, deployment, and optimization of YOLOv5 models.
    - Ultralytics YOLOv5 Documentation

Benchmarking of YOLOv12 Variants

Code

- All models were evaluated using a common set of pre-recorded videos(send by the team) located in a specified directory – test(out_put).
- The **PyTorch-based models** (the original YOLOv12 model trained in PyTorch and a lightweight version of the base model obtained through knowledge distillation and model pruning) were evaluated using the ultralytics.YOLO wrapper.
- The **ONNX model** (the previous model converted into ONNX format and further dynamically quantized for inference optimization) was evaluated using onnxruntime, with necessary preprocessing steps like resizing, normalization, and transposition.
- Metrics were logged and saved to a results file in a formatted table.

```
video_dir = r"C:\Users\user\Desktop\ADVANCED_MODEL_OPTIMISATION\test(out_put)"
results_file = r"C:\Users\user\Desktop\ADVANCED_MODEL_OPTIMISATION\benchmarking\results.txt"

models = {
    "Base Model": r"C:\Users\user\Desktop\ADVANCED_MODEL_OPTIMISATION\models\yolo12n.pt",
    "Distilled + Pruned": r"C:\Users\user\Desktop\ADVANCED_MODEL_OPTIMISATION\models\distilled_pruned_model\weights\best.pt",
    "Quantized ONNX": r"C:\Users\user\Desktop\ADVANCED_MODEL_OPTIMISATION\models\yolo12n.quantized.onnx"
}

metrics = defaultdict(dict)

def evaluate_pytorch_model(model_path, video_paths):
    model = YOLO(model_path)
    total_time, frame_count = 0, 0

    for video in tqdm(video_paths, desc=f"Evaluating PyTorch: {os.path.basename(model_path)}"):
        cap = cv2.VideoCapture(str(video))
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            start = time.time()
            _ = model(frame, verbose=False)
            total_time += time.time() - start
            frame_count += 1
        cap.release()

    latency = total_time / frame_count
    fps = frame_count / total_time
    return round(latency, 4), round(fps, 2), frame_count, round(total_time, 2)

def evaluate_onnx_model(onnx_path, video_paths):
    ort_session = ort.InferenceSession(onnx_path)
    input_name = ort_session.get_inputs()[0].name
    total_time, frame_count = 0, 0
```

```
def evaluate_onnx_model(onnx_path, video_paths):

    for video in tqdm(video_paths, desc=f"Evaluating ONNX: {os.path.basename(onnx_path)}"):
        cap = cv2.VideoCapture(str(video))
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            img = cv2.resize(frame, (640, 640))
            img = img.transpose(2, 0, 1)  # HWC to CHW
            img = img / 255.0
            img = img.astype('float32')
            img = img.reshape(1, 3, 640, 640)

            start = time.time()
            _ = ort_session.run(None, {input_name: img})
            total_time += time.time() - start
            frame_count += 1
        cap.release()

    latency = total_time / frame_count
    fps = frame_count / total_time
    return round(latency, 4), round(fps, 2), frame_count, round(total_time, 2)

def get_model_size(path):
    return round(os.path.getsize(path) / (1024 * 1024), 2)  # MB

def main():
    video_paths = list(Path(video_dir).glob("*.mp4"))
    if not video_paths:
        print("No test videos found.")
        return

    for name, path in models.items():
        if path.endswith(".onnx"):
            latency, fps, frames, total_time = evaluate_onnx_model(path, video_paths)
        else:
            latency, fps, frames, total_time = evaluate_pytorch_model(path, video_paths)
        metrics[name]["Latency (s/frame)"] = latency
        metrics[name]["FPS"] = fps
        metrics[name]["Frames"] = frames
        metrics[name]["Total Time (s)"] = total_time
        metrics[name]["Model Size (MB)"] = get_model_size(path)
```

```
    # Format Output
    headers = ["Model", "Latency (s/frame)", "FPS", "Frames", "Total Time (s)", "Model Size (MB)"]
    col_widths = [25, 18, 10, 10, 17, 17]

    with open(results_file, 'w') as f:
        # Header
        header_line = " | ".join(h.ljust(w) for h, w in zip(headers, col_widths))
        f.write(header_line + "\n")
        f.write("-" * len(header_line) + "\n")

        # Rows
        for model_name, result in metrics.items():
            row = [
                model_name.ljust(col_widths[0]),
                str(result["Latency (s/frame)"]).ljust(col_widths[1]),
                str(result["FPS"]).ljust(col_widths[2]),
                str(result["Frames"]).ljust(col_widths[3]),
                str(result["Total Time (s)"]).ljust(col_widths[4]),
                str(result["Model Size (MB)"]).ljust(col_widths[5])
            ]
            f.write(" | ".join(row) + "\n")

    print(f"\n✅ Benchmark saved to: {results_file}")

if __name__ == "__main__":
    main()
```

Evaluation Metrics

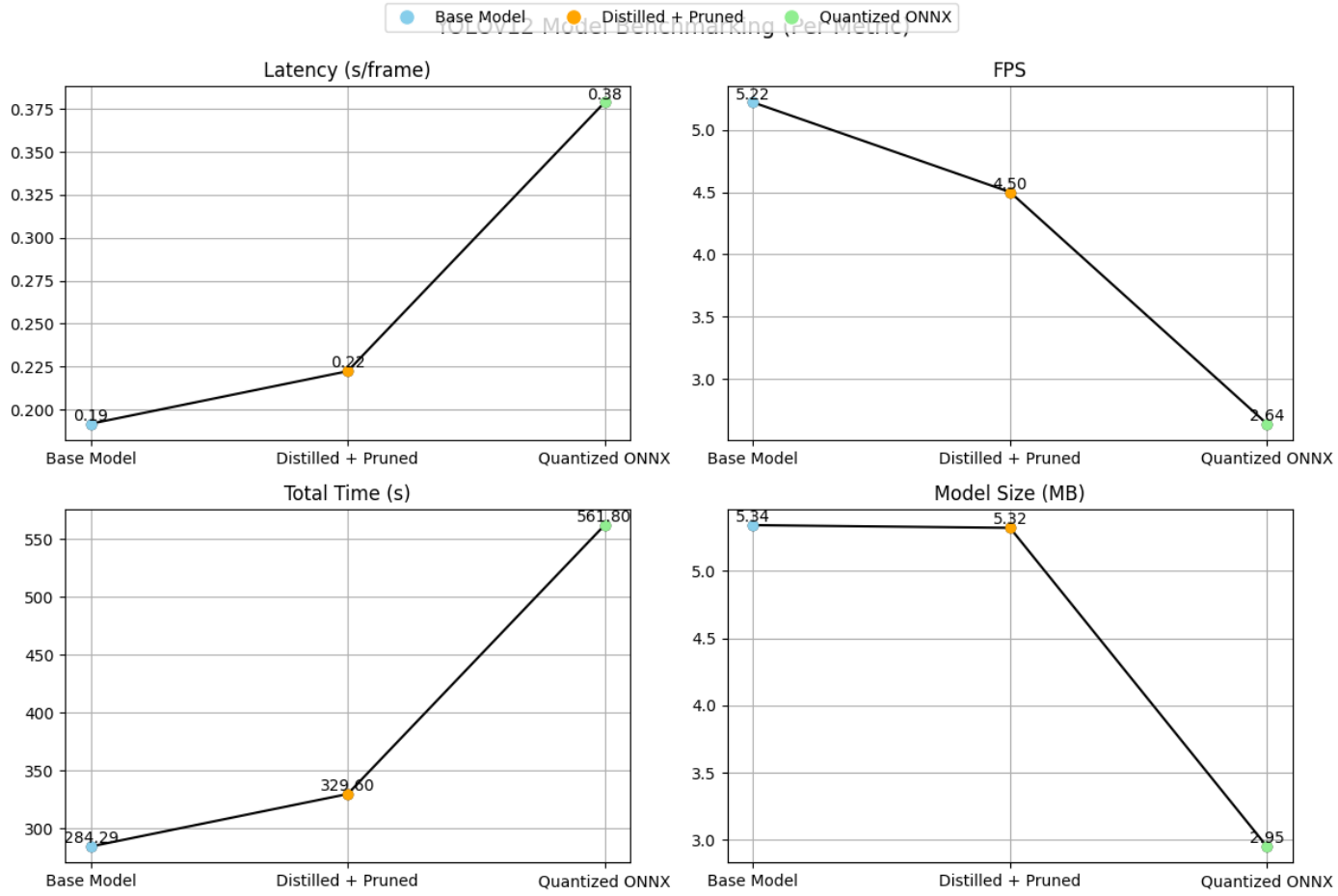
- **Latency (s/frame):** Time taken by the model to process a single frame. Lower is better.
- **FPS (Frames Per Second):** Number of frames the model can process in one second. Higher is better.
- **Total Time (s):** Total time taken to process all frames.
- **Frames:** Number of video frames processed (to ensure consistency across models).
- **Model Size (MB):** Storage footprint of the model on disk.

Results

Model	Latency (s/frame)	FPS	Frames	Total Time (s)	Model Size (MB)
Base Model	0.1917	5.22	1483	284.29	5.34
Distilled + Pruned	0.2223	4.5	1483	329.6	5.32
Quantized ONNX	0.3788	2.64	1483	561.8	2.95

Insights

1. **Base Model (YOLOv12 - PyTorch)**
   This demonstrates the lowest latency (0.1917 s/frame) and highest throughput (5.22 FPS) but consumes more memory compared to the quantized version. It's size is reasonable at 5.34 MB.
2. **Distilled + Pruned Model**
   This model is slightly smaller (5.32 MB) and has slightly increased latency (0.2223 s/frame). This could be that while the model is smaller, the structure may have lost some hardware-optimization characteristics due to pruning.
3. **Quantized ONNX Model**
   Despite being the smallest, it has the highest latency (0.3788 s/frame) and lowest FPS (2.64).



Conclusion & Improvements

- **Base Model – Best for Speed**
  If you need fast results and have access to powerful devices like a good GPU or high-end CPU, then the base model is the best choice. It processes videos quickly but needs more memory and computing power.
- **Distilled + Pruned Model – Good Balance**
  This version is a smart middle-ground. It runs a bit slower than the base model but takes up slightly less space. It's a great option when you're working on a normal laptop or computer that doesn't have top-level hardware.
- **Quantized ONNX Model – Best for Low-Memory Devices**
  This model is very small in size, so it’s perfect for devices like mobile phones or other low-memory systems. But it’s a bit slower when it runs.