Assignment 2 — Modular Image Classification Training + Deployment (ONNX + PTQ)

Goal

Build a **reusable training + evaluation pipeline** for **image classification** in PyTorch where:

- backbones are **swappable**
- datasets are **swappable**
- “advanced training features” can be toggled on/off
- evaluation produces **class-wise + global metrics** + **confusion matrix plots**
- deployment includes **ONNX export + PTQ (dynamic + static)** and a comparison report

Deliver as a **public GitHub repo** with required result folders and artifacts.

Part 1 — Training + Evaluation Framework (Modular)

1) Model requirement: Swappable backbones

You must support at least these backbone options (classification only):

Required model options

1. **Custom backbone** (your own tiny CNN class)
2. **ConvNeXt** (from `torchvision` or `timm`)

3. **ViT** (from `torchvision`, `timm`, or Hugging Face)

4. **“Faster R-CNN backbone”**: **use only the backbone feature extractor**, then attach a classification head

    - For example: extract the ResNet+FPN backbone from a detection model and pool features to classify.

Important constraint: final system is still **classification**, not detection.
So you must convert any “detection backbone” output into a fixed feature vector (pooling) → linear head.

Expected outcome: a simple “model factory” like:

```
model = make_model(arch="convnext_tiny", num_classes=..., pretrained=True)
```

---

2) Dataset requirement: Swappable torchvision datasets

Support easy switching among at least **2–3 torchvision datasets**, for example:

- CIFAR10
- CIFAR100
- STL10

Use a dataset factory like:

```
train_loader, val_loader, num_classes = make_dataloaders(name="cifar10", ...)
```

---

3) Training pipeline requirement: Features + toggles

Your training loop must include the “baseline” loop and support turning these features on/off via a config:

Required toggles

- `use_amp`: Mixed Precision AMP
- `use_scheduler`: LR scheduler (e.g., cosine)
- `use_grad_clip`: gradient clipping
- `use_weight_decay`: optimizer regularization (AdamW)
- `freeze_backbone`: optional transfer-learning mode
- `seed`: reproducibility toggle (or always on)

Must be controlled via config (argparse or a config dict).

---

4) Evaluation requirement: Metrics + confusion matrix

For each experiment (model + dataset):

Compute and report:

- **Per-class**: accuracy, precision, recall, F1
- **Global**: macro avg, micro avg, weighted avg
- Confusion matrix plot

You may use:

- pure PyTorch + numpy
- OR `sklearn.metrics` for reliability and less bugs

Confusion matrix file naming

Save confusion matrix as:

`part_1_results/cn_{model_arch}_{dataset}.png`

---

5) Results folder structure: `part_1_results/`

Inside repo root:

Required files

1. `part_1_results/summary.txt`
2. multiple confusion matrix images:
   `cn_{model_arch}_{dataset}.png`

summary.txt format

For **each experiment**, append:

- A short header line: `Experiment: {model_arch} | Dataset: {dataset}`
- A pandas-style table for **class-wise metrics**
- A pandas-style table for **global aggregates**
- Then **one empty line** between experiments

You can generate tables using pandas and convert to string:

```
summary += df.to_string()
```

Part 2 — Deployment: ONNX + PTQ comparison

Goal

Pick **one model + one dataset** (your choice), train with a configuration you choose, then compare:

1. **Original model (PyTorch FP32)** metrics
2. **ONNX dynamic INT8 PTQ** metrics
3. **ONNX static INT8 PTQ (calibrated)** metrics

Requirements

- Export ONNX FP32
- Create `dynamic` quantized ONNX
- Create `static` quantized ONNX using calibration data

What to compare

At minimum, report:

- overall accuracy
- macro F1
- weighted F1
- (optional) per-class F1 if you want extra credit

Results folder structure: `part_2_results/`

Required:

- `part_2_results/summary.txt`
  Include a small comparison table like:

Model Artifact	Accuracy	Macro F1	Weighted F1
PyTorch FP32	...	...	...

ONNX INT8 Dynamic	...	...	...
ONNX INT8 Static	...	...	...

(Optional) also include confusion matrices for the three variants.

Repo expectations (engineering quality)

Your repo should be runnable like:

- `python train.py --arch convnext_tiny --dataset cifar10 --use_amp 1 --use_scheduler 1 ...`
- `python eval.py --ckpt ...`
- `python export_onnx.py ...`
- `python quantize.py --mode dynamic|static ...`

Minimum nice-to-have:

- `README.md` with setup + commands
- `requirements.txt` or `pyproject.toml`
- clear code layout:
  - `datasets.py`
  - `models.py`
  - `train.py`
  - `eval.py`
  - `export_onnx.py`

    - `quantize.py`
    - `utils/metrics.py`

Hard but fair “challenges” learners will hit (intentionally)

- making backbones output a consistent feature shape (especially the “Faster R-CNN backbone” option)
- getting confusion matrix + per-class metrics correct
- keeping configs clean without spaghetti toggles
- keeping preprocessing consistent between PyTorch and ONNX evaluation for Part 2