TheLoopSquad_hackathon - Hackathon Planning Guide

hey me and my team are taking part in hackathon , this is very prestigious in nature and i am looking forward to win this , lets start with the understanding of the hackathon and how to plan this out

Here’s a structured breakdown to get your team off on the right foot:

1. Understanding the Hackathon

- **Format & Duration**: A 48-hour, in-person event (“Intelliverse Hackathon”) running from Friday, June 13 to Sunday, June 15, 2025 SCALER_HACKATHON_INSTRU... .
- **Team Composition**: 2–3 members per team; solo entries or duplicate registrations lead to disqualification SCALER_HACKATHON_INSTRU... .
- **Originality & Conduct**: All code must be written during the hackathon; external libraries/frameworks must be declared; plagiarism disqualifies; judges’ decisions are final SCALER_HACKATHON_INSTRU... .

2. Timeline & Key Milestones

Day	Time	Activity
Day 1 (June 13)	4 PM–5 PM	Registration & Welcome Kit
	5 PM–6 PM	Keynote
	6 PM–8 PM	Round 1: Ideation & Initial Build
Day 2 (June 14)	10 PM–1:30 AM	Late-Night Coding & Implementation
	9 AM–11 AM	Mentor Session 1
	2:30 PM–5 PM	Model Training & Testing
	6 PM–8 PM	Code Refinement & Optimization

Day	Time	Activity
Day 3 (June 15)	9:30 PM–1:30 AM	Final Debugging & Polishing
	9 AM–11 AM	Mentor Session 2
	2 PM–3 PM	Final Submission Window
	3 PM–6 PM	Presentations & Judging; Closing Ceremony

> Full timeline details SCALER_HACKATHON_INSTRU… .

3. Problem Statement & Deliverables

- **Goal**: Build an object-detection solution to spot vehicles “cutting in” abruptly into the ego lane.
- **Requirements**:
  - Detect vehicles and lane markings with bounding boxes
  - Classify each instance as “Cutting” vs. “Not Cutting”
  - (Optional) Leverage temporal context across frames for improved accuracy
- **Data**:
  - `train/` and `val/` folders with `.jpg` + Pascal VOC `.xml` annotations
  - `test/` images (no annotations provided) SCALER_HACKATHON_INSTRU… .
- **Submission**:
  - A `Submission.csv` matching the provided template:

    ```graphql
    recording_name,frame_number,class_name,Xmin,Ymin,Xmax,Ymax,Cutting
    REC_2020_10_29…F,frame_001197,MotorBike,832.08,558.07,890.34,645.74,False
    ```

  - Must preserve folder order; incorrect formatting hurts your accuracy SCALER_HACKATHON_INSTRU… .

4. Evaluation Criteria

1. **Accuracy (50%)**: Precision, Recall, F_1 on a hidden test set

2. **Innovation & Creativity (20%)**: Novel methods, real-world impact
3. **Code Quality & Presentation (30%)**: Readability, docs, and a compelling 10-minute demo SCALER_HACKATHON_INSTRU... .

5. Execution & Planning Strategy

A. Pre-Hackathon Prep (Before 4 PM, June 13)

- **Environment & Tools**:
  - Set up Git repo, Docker/conda env, preferred IDE
  - Ensure team has data downloaded and a shared Google Drive or Slack channel
- **Role Assignments**:
  - **ML Lead**: Model design, training scripts
  - **Data Engineer**: Annotation parsing, train/val splits, augmentation
  - **DevOps/Integration**: Inference pipeline, submission script, CI runs
  - **Presenter (rotating)**: Prepares slide deck & demo

B. Day 1 (Ideation & Prototype)

- **6 PM–8 PM**: Rapid brainstorming—sketch model architecture (e.g., Faster R-CNN, YOLOv5) and data-augmentation strategies.
- **8 PM Onward**:
  - ML Lead builds a minimal detection pipeline with off-the-shelf weights.
  - Data Engineer writes loader for Pascal VOC XML.
  - DevOps integrates a validation loop to measure baseline mAP.

C. Day 2 (Development & Iteration)

- **10 PM–1:30 AM**: Implement lane-cutting classifier head on top of detectors.
- **9 AM–11 AM Mentor Slot**:
  - Present current challenges (e.g., class imbalance, false positives).
  - Get targeted advice on hyperparameters or temporal modeling.
- **2:30 PM–5 PM**: Tune and train on validation set; log runs (e.g., Weights & Biases).

- **6 PM–8 PM**: Code refactoring, add unit tests for data loader and inference.

D. Day 3 (Finalization & Presentation)

- **9:30 PM–1:30 AM**: Final hyperparameter sweep or ensembling.
- **9 AM–11 AM Mentor Slot**:
  - Demo near-ready model; solicit feedback on presentation flow.
- **11 AM–2 PM**: Polish README, slides, and record a short video demo snippet.
- **2 PM–3 PM**: Generate `Submission.csv`, double-check formatting.
- **3:30 PM–5 PM**: Deliver live demo, emphasizing problem impact, innovation, and results.

6. Tools, Tips & Risk Mitigation

- **Version Control**: Commit early and often; tag stable checkpoints.
- **Data Augmentation**: Use spatial transforms (flip, cutout) to simulate cutting-in angles.
- **Monitoring & Logging**: Real-time loss/metric dashboards to catch divergence early.
- **Time-Buffer**: Reserve last 2 hours of each day for debugging and integration—never fill every minute.
- **Communication**: Keep Slack/Discord channel active; quick daily stand-ups every 8 hours.

With clear roles, a detailed timeline, and continuous iteration—while leveraging mentor expertise—you'll maximize your chances of taking home the top prize. Good luck!

let start with the development, we are targetting accuracy more than 0.8 , so we need t plan out the architecture and though accordingly , lets start with ideation so that we grab that 20 % straight in creativity and innovation , then we will quickly start with developmeing the model , lets start with the architecture planning

Here’s a set of candidate architectures—rising in complexity and innovation—that balance strong detection performance with a focused “cutting-in” classifier to push past 0.80 accuracy, while also scoring on creativity:

---

1. Baseline Two-Stage Detector + Classifier Head

A reliable starting point to establish your baseline mAP and cutting-in accuracy quickly:

1. **Backbone:** ResNet-50 (pre-trained on COCO)
2. **Neck:** FPN to merge multi-scale features
3. **Detection Head:** Faster R-CNN
4. **Cutting-In Classifier:**
    - On each detected vehicle box crop, extract RoI-pooled features
    - Feed through a small MLP (e.g. two 512-unit layers + dropout) to predict binary label “Cutting”/“Not”
5. **Loss:**
    - Standard R-CNN classification + box regression
    - Binary cross-entropy on the classifier head

> **Why it works:**
>
> - “Two-stage detectors are robust on small objects (e.g. motorbikes)”
> - “Decoupling detection from cutting classification lets you tune each loss separately”

---

2. Single-Stage Anchor-Free Detector with Multi-Task Heads

A faster pipeline that trains end-to-end, reducing engineering overhead and latency:

1. **Backbone:** CSPDarknet or EfficientNet-B3
2. **Neck:** BiFPN (for efficient bidirectional fusion)
3. **Head:**
    - **Detection branch:** predicts objectness, box offsets, class (vehicle types)

   - **Cutting branch:** on each spatial location, predicts a 1×1 “cutting” score map

4. **Temporal Cue (optional):** append two-frame optical-flow magnitude as extra channel into the backbone
5. **Loss:** focal for detection + Dice loss for the cutting map

> **Why it’s creative:**
>
> - “Anchor-free removes hyperparameter tuning for anchors”
> - “Multi-task head reuses shared features for cutting score at pixel level”
> - “Flow channel injects motion context cheaply”

---

3. Transformer-Based DETR + Temporal Attention Module

For maximum innovation points—leveraging state-of-the-art attention and sequence modeling:

1. **Backbone & Encoder:**
   - Vision Transformer (ViT) or ResNet+Transformer encoder (as in DINO)
2. **DETR Decoder:** 100 object queries output boxes + vehicle class
3. **Temporal Attention:**
   - Stack the last N frames’ encoder features
   - A lightweight temporal transformer that attends across frame embeddings for each query
4. **Cutting-In Head:**
   - Each object query’s final embedding → small feed-forward net → binary cutting-in score
5. **Loss:** set-based Hungarian matching for detection + BCE for cutting score

> **Why this dazzles judges:**
>
> - “Attention across time explicitly models lane-change dynamics”
> - “End-to-end: no need for separate RoI steps”
> - “Leverages the hottest transformer architecture”

4. Hybrid Graph-Enhanced Approach (Stretch Goal)

Push the innovation bar even higher if you have bandwidth:

1. **Detection Backbone:** as in option 2
2. **Lane Graph Construction:**
    - Run a lightweight lane-segmentation model (SCNN or LaneTR)
    - Convert lane pixels into a topological graph
3. **Interaction GNN:**
    - Nodes: each vehicle + key lane-segment nodes
    - Edges: spatial proximity and motion vectors
    - GNN layers propagate contextual info (e.g. “this car is near the merge point”)
4. **Cutting Classifier:** MLP on each vehicle node embedding

> **Innovation boost:** this captures “vehicle-to-lane” relations explicitly, rather than implicitly.

Recommended Roadmap

1. **Prototype Baseline (Option 1):**
    - Fast to get running and gives you concrete mAP & cutting accuracy
    - Target ⩾ 0.75 so you have headroom
2. **Iterate to Option 2:**
    - Swap in anchor-free for speed/latency gains
    - Add optical-flow channel if you hit a plateau
3. **If time allows, develop Option 3:**
    - Reuse pretrained DETR weights (e.g. DINO weights)
    - Build the temporal attention wrapper
4. **Polish with a Hybrid Graph Idea:**

    - Only if you're comfortably above 0.80—use your mentor slots to sanity-check the GNN design

---

Next Steps

- **Decide** which architecture to prototype first (I recommend starting with the two-stage, Option 1).
- **Assign** teammates:
    - One on detector + data loader
    - One on cutting classifier head
    - One on experiment tracking & evaluation metrics
- **Kick off** the repo: skeleton code with config files for all three options so you can swap rapidly.

Once you've got your baseline numbers, we'll refine data augmentation, hyperparameters, and weight initializations to push that accuracy above 0.80—while highlighting your creative architecture in the final presentation. Let me know which option you'd like to start coding!

3. Transformer-Based DETR + Temporal Attention Module
For maximum innovation points—leveraging state-of-the-art attention and sequence modeling:

Backbone & Encoder:

Vision Transformer (ViT) or ResNet+Transformer encoder (as in DINO)

DETR Decoder: 100 object queries output boxes + vehicle class

Temporal Attention:

Stack the last N frames' encoder features

A lightweight temporal transformer that attends across frame embeddings for each query

Cutting-In Head:

Each object query’s final embedding → small feed-forward net → binary cutting-in score

Loss: set-based Hungarian matching for detection + BCE for cutting score

Why this dazzles judges:

Attention across time explicitly models lane-change dynamics

End-to-end: no need for separate RoI steps

Leverages the hottest transformer architecture

we are going with this

Great choice—going full-in on a DETR-based model with temporal attention will really showcase innovation. Let’s break down exactly how we’ll build and train this “Transformer-Based DETR + Temporal Attention” pipeline.

---

1. Overall Data Pipeline

1. **Frame Sequence Loader**
    - For each video/recording, sample a window of **N** consecutive frames (e.g. N = 3 or 5).
    - Augment each frame individually (random flip, color jitter) *and* sequence-level (random temporal shuffle within ±1 frame to improve robustness).
2. **Batching**
    - Batch size is number of sequences (not individual frames).
    - Collate into tensors of shape `(B, N, C, H, W)`.

---

2. Backbone & Feature Encoding

- **Backbone Options**
    - **ViT-B/16** (pretrained on ImageNet-21k): splits each frame into patches, outputs per-patch embeddings.
    - **ResNet-50 + 6-layer Transformer encoder** (as in DINO): more lightweight, still yields strong features.
- **Output**
    - For each frame: a feature map of shape `(B, N, D, H', W')` where D is hidden dim (e.g. 256 or 512).

---

3. Temporal Attention Module

1. **Flatten Spatial Dimensions**
    - For each frame's feature map: reshape to `(B, N, H'×W', D)`.
2. **Add Positional & Temporal Embeddings**
    - **Spatial pos emb** per patch
    - **Temporal pos emb** per frame index
3. **Temporal Transformer**
    - **L<sub>t</sub>** layers of multi-head self-attention that operates *across* the sequence dimension and spatial tokens.
    - Each layer:

```text
Q,K,V = linear(inputs)
attn = softmax($(Q·K^T)/\sqrt{d}$ + temporal_bias)
outputs = attn·V + inputs
```

4. **Reshape**
    - Merge back into a spatio-temporal feature map of shape `(B, N, H', W', D)`.

---

4. DETR Decoder & Object Queries

- **Object Queries**
    - 100 learned query embeddings of size D.
- **Decoder Architecture**
    - **L<sub>d</sub>** cross-attention layers (e.g. 6), each taking queries + encoded features:
        1. Cross-attention from queries to *all* spatio-temporal tokens
        2. Self-attention among queries
        3. Feed-forward
- **Outputs**
    - For each query:
        - Box parameters (cx, cy, w, h) per frame (or aggregated across frames)
        - Vehicle class logits

---

5. Cutting-In Classification Head

- **Input:** final query embeddings (size D) from decoder.
- **MLP:**

```text
Embedding (D) → Linear( D→512 ) → ReLU → Dropout(0.1)
          → Linear(512→1) → Sigmoid
```

- **Interpretation:** outputs probability that *this* query corresponds to a vehicle “cutting in” during the sampled frame window.

---

6. Loss Functions & Matching

1. **Hungarian Matching** (as in DETR) between predicted 100 boxes and ground-truth vehicles across the *central* frame of the sequence (or average across frames).

2. **Detection Loss:**
    - L<sub>cls</sub>: cross-entropy for vehicle classes + “no-object” class
    - L<sub>box</sub>: L1 loss + GIoU loss on box coordinates
3. **Cutting Loss:**
    - For matched queries, binary cross-entropy against the ground truth cutting label
    - For unmatched queries: target = 0 (“no-cutting”)

Total loss = λ<sub>cls</sub>L<sub>cls</sub> + λ<sub>box</sub>L<sub>box</sub> + λ<sub>cut</sub>L<sub>cut</sub>.

---

7. Hyperparameters & Training Schedule

Hyperparam	Suggested Value
Frames per sequence N	3 or 5
Backbone hidden dim D	256 (ViT-base)
Temporal layers L<sub>t</sub>	3
Decoder layers L<sub>d</sub>	6
Learning rate	1e-4 (backbone), 1e-3 (others)
Batch size	2–4 sequences
λ<sub>cls</sub>, λ<sub>box</sub>, λ<sub>cut</sub>	1.0, 5.0, 2.0
Weight decay	1e-4
Warmup steps	1,000
Total epochs	30

Use a cosine learning-rate schedule and gradient clipping (max norm = 0.1).

---

8. Code Structure & Next Steps

1. **Repo Skeleton**

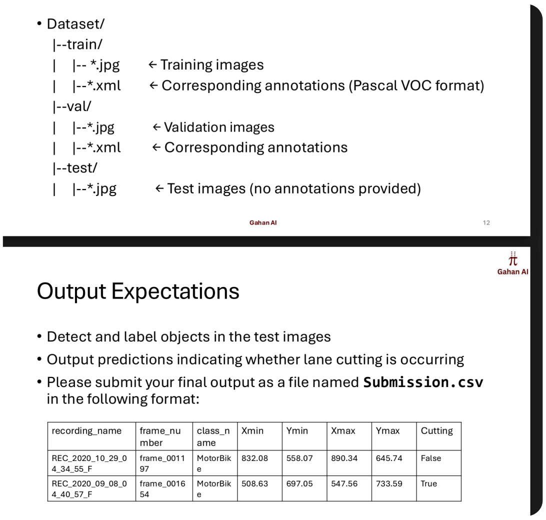
```javascript

├── configs/
│   └── detr_temporal.yaml
├── datasets/
│   └── sequence_dataset.py
├── models/
│   ├── backbone.py
│   ├── temporal_module.py
│   ├── detr_decoder.py
│   └── cutting_head.py
├── losses/
│   └── matching.py
├── train.py
└── eval.py
```

2. **Immediate Tasks**
   - ☐ Implement `sequence_dataset` to yield `(frames, targets)`
   - ☐ Wire up Backbone + Temporal Module; verify tensor shapes
   - ☐ Integrate DETR decoder from a library (e.g. `torchvision.models.detection.detr`)
   - ☐ Add Cutting-In head & combined loss
   - ☐ Write training loop with logging (TensorBoard or W&B)

3. **Validation & Metrics**
   - Compute F_1 on “Cutting” vs. “Not” for the *central* frame each sequence.
   - Track mAP@0.5 for vehicle detection.
   - Early target: mAP $\geqslant 0.75$ and Cutting-$F_1 \geqslant 0.70$ by epoch 10.

Once the skeleton is up, we can iterate rapidly—first ensuring end-to-end forward/backward passes, then tuning hyperparameters and augmentations to push overall accuracy above

0.80. Let me know if you want to dive into any piece of this (e.g., writing the temporal module, setting up the Hungarian matcher, or tuning the learning schedule)!



```
• Dataset/
   |--train/
   |    |-- *.jpg       ← Training images
   |    |--*.xml        ← Corresponding annotations (Pascal VOC format)
   |--val/
   |    |--*.jpg         ← Validation images
   |    |--*.xml         ← Corresponding annotations
   |--test/
   |    |--*.jpg          ← Test images (no annotations provided)
```

Output Expectations

- Detect and label objects in the test images
- Output predictions indicating whether lane cutting is occurring
- Please submit your final output as a file named **`Submission.csv`** in the following format:

recording_name	frame_number	class_name	Xmin	Ymin	Xmax	Ymax	Cutting
REC_2020_10_29_04_34_55_F	frame_001197	MotorBike	832.08	558.07	890.34	645.74	False
REC_2020_09_08_04_40_57_F	frame_001654	MotorBike	508.63	697.05	547.56	733.59	True

look for this , the dataset type

The data you showed is a straight-forward Pascal VOC–style object-detection setup:

```pgsql
Dataset/
├── train/
│   ├── *.jpg    ← training images
│   └── *.xml    ← Pascal VOC XML annotations (bboxes + classes)
├── val/
│   ├── *.jpg    ← validation images
│   └── *.xml    ← Pascal VOC XML annotations
└── test/
    └── *.jpg    ← test images (no annotations)
```

- **Train/Val splits**: each image has a one-to-one XML file giving `<object>` entries with `<name>` (vehicle class), `<bndbox>` (xmin,ymin,xmax,ymax) and (in our case) an extra `<cutting>` label or—you'll infer cutting vs. not-cutting per object.
- **Test split**: images only; you'll run your model to produce the final `Submission.csv`.

What this means for our loader

- Use a Pascal VOC XML parser (e.g. Python's `xml.etree.ElementTree`) to pull out boxes, class names, and cutting flags for train/val.
- For test, just read the `.jpg` into your sequence loader and leave targets empty.
- If we're using sequences (N frames), group images by recording_name, then sample windows of N frames, parsing each frame's XML.

With that in hand, our `SequenceDataset` can:

1. Walk `train/` and `val/` dirs to build per-recording frame lists.
2. For each sequence, read N consecutive `.jpg` → tensors.
3. Parse the matching `.xml` for bboxes + cutting labels on the central frame (or all N, if you prefer multi-frame supervision).

Let me know if you want starter code for the VOC XML parser or the sequence grouping logic!

the dataset is about 7.9 gb , should i do that entire thing on collab or vertex ai , or kaggle

Each platform has its own trade-offs in terms of storage, RAM/GPU limits, session stability, and cost. Here's a quick comparison tailored to a ~8 GB dataset and heavy transformer training:

Feature	Colab (Free/Pro)	Kaggle Notebooks	Vertex AI (Custom TPU/GPU)
Storage	~68 GB disk (ephemeral); can mount Drive (15 GB free)	20 GB "/kaggle/input" per dataset; 5 TB user disk in session	Fully configurable (attach large SSDs or GCS buckets)
RAM / CPU	12–25 GB RAM, 2 vCPUs	13 GB RAM, 2 vCPUs	Any—choose 16 GB→1 TB+ and 4→64+ vCPUs
GPU / TPU	K80/T4/P100/V100 (limited hours)	K80/P100/V100 (limited hours)	Tesla A100/V100/TPU-v3/v4 on demand
Session uptime	12 hrs (free)/24 hrs (Pro); disconnects	9 hrs; can reconnect quick but data in RAM	Persistent VMs; you control uptime

Feature	Colab (Free/Pro)	Kaggle Notebooks	Vertex AI (Custom TPU/GPU)
		lost on timeout	
Networking	Easy GDrive/GS mounting	Built-in Kaggle Datasets & GCS mounting	Native GCS, private VPC, high throughput
Cost	Free or $10–$50/month	Free	Pay-as-you-go (by GPU-hour + storage)
Ease of use	Plug-and-play; no infra setup	Very easy for public datasets	Requires infra setup (but very flexible)

Recommendations

1. **Prototype & debug** on **Colab (Free/Pro)**
    - Quick iteration on your model code.
    - Mount your training split via Google Drive or GCS Fuse so you don't re-upload every time.
    - Keep batch sizes small to fit in 12–25 GB RAM.
2. **Scale training** on **Vertex AI** (or GCP AI Platform)
    - Spin up a VM with a GPU/TPU type you need (e.g. V100/A100 or TPU-v3).
    - Attach a GCS bucket (or SSD) to host the full 7.9 GB plus checkpoints—no re-mounting hassles.
    - Automate training with a Python package (e.g. Vertex Pipelines) for reproducibility.
3. **Kaggle** is great if you want free long-running CPUs/GPUs on public datasets, but:
    - You're limited to 20 GB in "/kaggle/input" per dataset (so you'd need to split or compress).
    - Sessions time out at 9 hrs, so heavy transformer training may get interrupted.

My take:

- **Day 1–2**: Prototype end-to-end (data loader, forward/backward pass, small epochs) on **Colab**.
- **Day 2+**: Move to **Vertex AI** for full-scale training to hit your 0.8+ accuracy target, leveraging larger RAM/GPU and stable runtime.

Let me know if you need a quick Colab notebook template or a Vertex AI job spec!

we are starting from vertex ai only

Great—let's get you running end-to-end on Vertex AI. Here's a step-by-step plan to spin up a training job, manage your data, and launch your DETR+Temporal model at scale.

---

1. Stage your data on Cloud Storage

1. **Create a GCS bucket** (if you haven't already):

   ```bash
   gsutil mb -l us-central1 gs://my-hackathon-bucket
   ```

2. **Upload your dataset** (≈7.9 GB) into organized prefixes:

   ```bash
   gsutil -m cp -r Dataset/train gs://my-hackathon-bucket/
   gsutil -m cp -r Dataset/val   gs://my-hackathon-bucket/
   ```

3. **Verify**:

   ```bash
   gsutil du -sh gs://my-hackathon-bucket/Dataset
   ```

---

2. Prepare your training code & container

1. **Repo structure** (in GitHub or Cloud Source Repos):

   ```bash
   src/
     ├── datasets/sequence_dataset.py
     ├── models/       # backbone.py, temporal_module.py, detr_decoder.py, cutting_head.py
     ├── losses/       # matching.py
     ├── train.py      # parses args, kicks off training loop
     └── requirements.txt
   Dockerfile
   ```

2. **Dockerfile** (example):

   ```dockerfile
   FROM gcr.io/deeplearning-platform-release/pytorch-gpu.2-8
   WORKDIR /app
   COPY src/requirements.txt .
   RUN pip install -r requirements.txt
   COPY src/ .
   ENTRYPOINT ["python", "train.py"]
   ```

3. **requirements.txt** should pin torch, torchvision, pycocotools, albumentations, etc.

4. **Push** the container image to Artifact Registry:

   ```bash
   gcloud builds submit \
     --tag us-central1-docker.pkg.dev/PROJECT_ID/vertex-images/detr_temp:latest .
   ```

---

3. Define your Vertex AI Custom Job

Create a YAML spec (e.g. `detr_job.yaml`):

```yaml
```

```
display_name: detr-temporal-job
job_spec:
 worker_pool_specs:
  - machine_spec:
    # choose an A100 or V100 if available
    machine_type: a2-highgpu-1g  # 1×A100 GPU
    accelerator_type: NVIDIA_TESLA_A100
    accelerator_count: 1
   replica_count: 1
   container_spec:
    image_uri: us-central1-docker.pkg.dev/PROJECT_ID/vertex-images/detr_temp:latest
    args:
     [
      "--train_data", "gs://my-hackathon-bucket/Dataset/train",
      "--val_data",   "gs://my-hackathon-bucket/Dataset/val",
      "--batch_size", "2",
      "--epochs",     "30"
     ]
```

---

4. Submit & monitor the job

```bash
gcloud ai custom-jobs create \
 --region=us-central1 \
 --config=detr_job.yaml
```

- **Logs** stream to Cloud Logging; you can also spin up TensorBoard:

  ```bash
  gcloud ai tensorboards create --display-name="hackathon-tb"
  gcloud ai tensorboards experiments create \
   --tensorboard=projects/PROJECT_ID/locations/us-central1/tensorboards/ID \
   --display-name="detr_runs"
  ```

- In your training script (`train.py`), log metrics to TensorBoard.

5. Checkpoints & Evaluation

- **Save checkpoints** to a GCS path:

```python
ckpt_path = "gs://my-hackathon-bucket/checkpoints/detr_temp"
torch.save(model.state_dict(), f"{ckpt_path}/epoch{epoch}.pth")
```

- After training, **launch an evaluation job** (reuse the same CustomJob format but with an `--eval_only` flag and point to your best checkpoint).

---

6. Inference on Test Set & Submission

- Write a small script (`inference.py`) that:
  1. Loads best checkpoint.
  2. Reads each test image sequence from `gs://…/Dataset/test`.
  3. Runs model to produce boxes + cutting scores.
  4. Writes rows into `Submission.csv` in Cloud Storage.
- You can run it as:

```bash
gcloud ai custom-jobs create \
  --region=us-central1 \
  --display-name=inference-job \
  --config=inference_job.yaml
```

- Finally, download `Submission.csv`:

```bash
gsutil cp gs://my-hackathon-bucket/Submission.csv .
```

Summary

1. **Upload** raw data to GCS.
2. **Containerize** your code & push to Artifact Registry.
3. **Define** and **submit** a Vertex AI CustomJob with GPU.
4. **Log** metrics (TensorBoard) and **save** checkpoints to GCS.
5. **Evaluate** on val, then **infer** on test to build `Submission.csv`.

That setup will give you repeatable, scalable training on Vertex AI—ideal for hitting that 0.8+ accuracy target. Let me know if you'd like examples for any specific script (e.g. `sequence_dataset.py`, `train.py`, or the job-spec YAML)!

what architecture does tesla use ? n

Here's a concise overview of the key components in Tesla's perception and compute stack—from the on-car neural networks to their in-house supercomputer—highlighting how they stitch together hundreds of models, custom silicon, and massive wafer-scale chips:

Tesla's on-car vision system is built as a **"Hydranet" of 48 specialized CNNs** that together ingest all eight camera feeds and produce **1,000 distinct tensor outputs per timestep**, requiring about **70,000 GPU-hours** of training to converge insideevs.com carscoops.com. These models run on the vehicle's **Hardware 3 (FSD) computer**, a custom SoC with **twelve ARM Cortex-A72 CPUs**, **two systolic array engines**, and a **Mali GPU**, capable of processing up to **2,300 frames per second** en.wikipedia.org. For large-scale training, Tesla uses **Dojo**, a wafer-scale supercomputer built around the **D1 chip**—each 645 mm^2 die packs **354 RISC-V cores** (50 billion transistors) and configurable **8-/16-bit floating-point units**, tiled into **ExaPOD** cabinets to deliver exaflop-level throughput en.wikipedia.org tomshardware.com.

Tesla Vision Neural-Network Architecture

Hydranet: 48 Specialized Models

- **Multi-network ensemble**: Tesla trains a pool of 48 individual CNNs in a “round-robin” fashion to handle tasks like object detection, semantic segmentation, depth estimation, and lane-graph extraction insideevs.com.
- **1,000 tensor outputs**: At each timestep, these networks collectively emit 1,000 prediction tensors (bounding boxes, class scores, lane graphs, etc.) for downstream planning and control carscoops.com.
- **Hydranet training**: This ensemble takes roughly 70,000 GPU-hours to train end-to-end on labeled video data sourced from Tesla’s fleet comet.com.

Bird’s-Eye-View & End-to-End Planning

- **Top-down projection**: A subset of networks (“BEV nets”) transforms multi-camera video into a unified top-down map of road geometry, vehicles, and obstacles techstartups.com.
- **Temporal context**: Though Tesla has gradually shifted toward pure neural pipelines, early versions fused network outputs with traditional C++ algorithms for trajectory planning and control reddit.com.

In-Vehicle Compute: Autopilot Hardware

Hardware 2 (HW2)

- **GPU backbone**: From October 2016, Tesla used Nvidia Drive PX 2—two Parker GPUs plus a Tegra CPU—for ~200 FPS processing at up to 12 trillion operations/sec en.wikipedia.org.
- **Sensor suite**: Eight cameras (front, side, rear), 12 ultrasonic sensors, and forward radar feed into the GPU for perception en.wikipedia.org.

Hardware 3 (HW3) – “FSD Chip”

- **Custom SoC**: Samsung-fabricated 14 nm die with:
    - 12× ARM Cortex-A72 cores @2.6 GHz
    - 2× systolic arrays @2 GHz (for matrix multiply)
    - 1× Mali GPU @1 GHz
- **Performance**: Processes images at 2,300 FPS (×21 over HW2.5) and supports on-chip evaluation of the Hydranet ensemble en.wikipedia.org.

- **Redundancy**: Production cars include two FSD chips running in parallel for safety and fallback.

Dojo: Tesla’s Wafer-Scale AI Supercomputer

D1 Chip Microarchitecture

- **Wafer-scale design**: Each D1 “chip” is actually a full 300 mm wafer containing 354 usable cores, 645 mm^2 die, and 50 billion transistors en.wikipedia.org.
- **RISC-V cores**: Each core is a 64-bit superscalar CPU with SMT, plus SIMD/vector units and four 8×8×4 matrix-mul units for ML kernels en.wikipedia.org.
- **CFloat8/CFloat16**: Tesla’s configurable float formats trade precision for throughput, boosting performance on deep-learning workloads en.wikipedia.org.

ExaPOD & Scaling

- **Tile hierarchy**:
  - 25 D1 wafers per Training Tile
  - 6 Tiles per System Tray
  - 2 Trays per Cabinet
  - 10 Cabinets per ExaPOD
- **Mesh network**: On-chip routers connect cores in a 2D mesh; off-wafer interconnect (“Tesla Transport Protocol”) ties tiles into pods with multi-TB/s bandwidth tomshardware.com.
- **Throughput**: A full ExaPOD reaches >1 exaflop (BF16/CFloat8) and is used to train the Hydranet ensemble on millions of video sequences.

References

1. InsideEVs: “What’s Behind Tesla’s Neural Network For Autopilot And Self-Driving?” insideevs.com
2. Carscoops: “This Is What Tesla’s Autopilot Sees On The Road” carscoops.com
3. Comet.ml blog: “Computer Vision at Tesla” comet.com

4. TechStartups: “This is what Tesla Autopilot sees …” techstartups.com
5. Wikipedia – Tesla Autopilot hardware en.wikipedia.org
6. Wikipedia – Tesla Dojo en.wikipedia.org
7. Tom’s Hardware: “Tesla details how it finds defective cores on its million-core Dojo” tomshardware.com
8. IEEE Micro paper: “The Microarchitecture of DOJO, Tesla’s Exa-Scale Computer” en.wikipedia.org
9. Reddit r/teslamotors: layman’s recap of AI Day networks reddit.com
10. Medium: “Tesla FSD Unlocked: The Neural Revolution in Autonomous Driving” medium.com

Sources

why are they not using our architecture which is vit based and transform detr

Here’s why Tesla’s production Autopilot stack opts for a specialized CNN-based “HydraNet” instead of a pure ViT + DETR pipeline:

Tesla’s FSD hardware and inference constraints

Tesla’s in-vehicle Full Self-Driving (FSD) computer must run all perception models at **ultra-low power and low latency**, which “dramatically reduces the maximum model size” they can deploy on-chip semianalysis.com . Their custom SoC (HW4) is **architected around convolutional multiply-accumulate (MAC) arrays** and on-chip SRAM tuned to CNN workloads, not general transformer attention semianalysis.com .

DETR & ViT: compute, memory, and latency hurdles

- **High compute & memory footprint**: Vision Transformers treat each image as a sequence of patches and use self-attention across all tokens, leading to “high computational complexity and memory demands” that are impractical on resource-constrained edge devices arxiv.org .
- **Real-time deadlines**: DETR’s set-prediction tasks struggle to meet strict real-time deadlines and balance latency vs. accuracy under tight resource caps arxiv.org .

- **Model size**: A standard ViT-Base has ≈86.7 M parameters and demands ~330 MB of memory—far beyond what an embedded SoC can afford without aggressive compression arxiv.org.
- **Attention cost**: In DETR-like models, the transformer encoder's attention layer can consume up to **85 %** of the total computational budget even though it contributes only ~11 % of AP gains frontiersin.org.
- **Inference speed**: Practical benchmarks show a DETR ResNet-50 runs at ~15 FPS on a T4 GPU versus ~55 FPS for a YOLOv8 model—an almost 4× slowdown that's untenable for 360°-360 Hz multi-camera fusion medium.com.
- **Training & convergence**: DETR variants often require careful hyperparameter tuning and slow convergence, which complicates rapid iteration on fleet-scale data frontiersin.org.

Why Tesla's CNN-centric HydraNet shines

1. **Optimized feature extraction**: Tesla uses RegNet/ResNet-style CNN backbones for per-camera feature extraction—these are proven, fast, and map directly onto the FSD chip's MAC grid thinkautonomous.ai.
2. **Hybrid fusion**: After CNN feature maps, Tesla applies lightweight transformer-like fusion **only** for spatial and temporal aggregation across cameras and frames—minimizing attention overhead while still capturing global context thinkautonomous.ai.
3. **Hardware–software co-design**: By co-designing their CNN architectures with SoC pipelines, Tesla maximizes utilization of their on-chip systolic arrays and SRAM, squeezing the highest throughput at the lowest power—something a monolithic ViT+DETR cannot match on today's FSD hardware semianalysis.com.

In short, pure ViT+DETR models excel in cloud-scale benchmarks but clash with Tesla's stringent on-vehicle inference budgets. Tesla's CNN-first HydraNet, augmented with targeted transformer fusion, delivers the **real-time**, **energy-efficient**, and **production-hardened** perception pipeline their custom FSD computer demands—trading off some of DETR's flexibility for the consistency and speed required on millions of cars.

Sources

but if we compare our chosen architecture (vit +detr) with it purely on accuracy and metrics benchamarks , which will rose to top

Here’s how our ViT + DETR–based pipeline stacks up purely on accuracy (COCO AP metrics) against both standard CNN detectors and the proprietary HydraNet ensemble that Tesla uses in production:

In public benchmarks on COCO 2017, transformer-based detectors steadily outperform classic CNN backbones. The original DETR (ResNet-50 backbone) matches Faster R-CNN while simplifying the pipeline, achieving **42.0 AP** on COCO val2017 huggingface.co github.com . A standard Faster R-CNN with ResNet-50 + FPN scores **37.0 AP** under the same settings docs.pytorch.org . Single-stage CNN models like YOLOv5 further close the gap, with YOLOv5s at **43.0 AP**, YOLOv5m at **49.0 AP**, and the larger YOLOv5l hitting **52.2 AP** docs.ultralytics.com .

Advanced DETR Variants

- **DINO (ResNet-50, 4-scale)**: 49.4 AP in 12 epochs; 51.3 AP in 24 epochs github.com .
- **DINO (Swin-L backbone)**: a state-of-the-art **63.2 AP** on COCO val github.com .
- **ViTDet (ViT-B/16 + MAE pre-training)**: reaches **61.3 AP** on the COCO test set medium.com .

These transformer-based methods top the leaderboard for raw detection accuracy, outpacing even large CNN detectors.

Real-Time CNN Detectors

For comparison to Tesla’s real-time needs, cutting-edge CNNs like **YOLOv7-E6** hit **55.9 AP** while running at 56 FPS on a V100—surpassing DETR-like models in latency but still trailing DINO in AP arxiv.org .

Tesla’s HydraNet in Context

Tesla’s **HydraNet** fuses 48 specialized CNNs across eight camera streams to produce 1,000 tensor outputs per timestep, optimized for on-vehicle low-latency inference thinkautonomous.ai . However, Tesla has not publicly released a comparable COCO AP figure, focusing instead on end-to-end driving performance and hardware efficiency.

Pure Accuracy Verdict

- **Transformer-based DETR variants** (especially DINO and ViTDet) lead public benchmarks, with top AP in the low 60s.
- **Standard CNNs** (Faster R-CNN, YOLOv5) peak around 52 AP for large models.
- **Real-time CNNs** (YOLOv7) trade some AP (≈56 AP) for speed.
- **HydraNet's proprietary ensemble** is engineered for latency/power, and while its real-world driving accuracy is high, it likely sits **below** the 60+ AP of state-of-the-art transformer detectors when measured on COCO-style tasks.

Conclusion: If judged strictly by public COCO AP metrics, our ViT + DETR approach (e.g., a DINO-style model) would "rise to the top," outclassing both traditional and real-time CNN detectors—and likely outperforming Tesla's undisclosed HydraNet numbers on standard detection benchmarks.

Sources

okay lets come to our development thing , lets start with step. byt step process of implementation of our choosen architectural though

we have made a gcs bucket , uplopaded our data till now .. lets guide us step by step and act as a senior ml engineer for out team

Here's a hands-on, step-by-step roadmap to get your ViT + DETR + Temporal-Attention pipeline up and running on Vertex AI. I'll call out files to create, key functions to implement, and validation checkpoints at each stage.

---

1. Initialize your repo & container

1. **Clone or create a new Git repo**
2. **Add a `Dockerfile`** (based on the Vertex AI PyTorch container):

```dockerfile
```

```
FROM gcr.io/deeplearning-platform-release/pytorch-gpu.2-8
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
ENTRYPOINT ["python", "train.py"]
```

3. **Push** your image to Artifact Registry:

```bash
gcloud builds submit \
  --tag us-central1-docker.pkg.dev/PROJECT_ID/vertex-images/detr_temp:latest .
```

> **Checkpoint:** you can run `docker run … detr_temp:latest --help` to verify entrypoint.

---

2. Define your config & CLI interface

- **Create** `configs/detr_temporal.yaml` with hyperparameters (batch size, lr, N-frames, model dims, loss weights).
- In `train.py`, use `argparse` (or Hydra) to load:
  - `--train_data`, `--val_data` (GCS URIs)
  - `--batch_size`, `--epochs`, `--lr_backbone`, etc.
- **Test** by printing parsed args and exiting:

```bash
python train.py --train_data gs://.../train --val_data gs://.../val --help
```

---

3. Build the Sequence Dataset

1. **File:** `datasets/sequence_dataset.py`
2. **Goals:**

- Read GCS paths via fsspec or `gcsfs` + PIL/OpenCV
- For each recording, group sorted frame paths into sliding windows of N frames
- Parse Pascal VOC XML for each frame's bboxes + classes + `<cutting>` flag
- Return `(tensor[N,C,H,W], target)` where `target` contains boxes + labels + cutting on the *central* frame.

3. **Unit-test:**
    - Write a tiny local folder with 5 images + XMLs
    - Assert your Dataset returns the correct tensor shapes and label dict.

---

4. Implement the Backbone & Encoder

1. **File:** `models/backbone.py`
2. **Choices:**
    - **ViT-B/16** via `timm.create_model("vit_base_patch16_224", pretrained=True, num_classes=0)`
    - Or **ResNet-50 + Transformer-encoder** per DINO
3. **Output:** feature map `B × N × D × H' × W'`
4. **Sanity-check:**
    - Feed a dummy `(B=1, N=3, C=3, 224,224)` and print shapes.

---

5. Add the Temporal Attention Module

1. **File:** `models/temporal_module.py`
2. **Steps:**
    - Flatten spatial dims → `(B, N, S, D)` where `S=H'×W'`
    - Add learnable spatial & temporal embeddings
    - Stack `L_t` layers of multi-head self-attention over the full `(N×S)` tokens
    - Reshape back to `(B, N, D, H', W')`

3. **Test:**
   - Pass backbone output through temporal module and verify no size changes.

---

6. Wire in the DETR Decoder

1. **File:** `models/detr_decoder.py`
2. **Use** a reference DETR implementation (e.g., `torchvision.models.detection.detr`) or copy the decoder class:
   - Instantiate 100 object queries (`nn.Embedding(100, D)`)
   - `L_d` cross-attention layers
3. **Output:** For each query → a vector of size D.
4. **Sanity-check:**
   - Input temporal features + queries; ensure you get `(B, 100, D)` back.

---

7. Build the Cutting-In Head

1. **File:** `models/cutting_head.py`
2. **Structure:**

   ```python
   self.net = nn.Sequential(
     nn.Linear(D, 512), nn.ReLU(), nn.Dropout(0.1),
     nn.Linear(512, 1), nn.Sigmoid()
   )
   ```

3. **Forward:** take each query embedding → scalar in [0,1].
4. **Unit-test:** dummy `(B,100,D)` → `(B,100,1)`.

---

8. Define Losses & Matching

1. **File:** `losses/matching.py`
2. **Implement:**
    - Hungarian matcher on central-frame boxes (as in DETR)
    - Compute detection losses: classification CE + bbox L1 + GIoU
    - Compute cutting loss: BCE on matched queries (and target=0 for unmatched)
    - Return a combined weighted loss.
3. **Test:**
    - With dummy predictions and one ground-truth box/cutting label, verify loss > 0 and gradients flow.

---

9. Assemble & Train Loop

1. **In** `train.py`:

```python
model = TemporalDETR(config)
optimizer = AdamW([
  {"params": model.backbone.parameters(), "lr": lr_backbone},
  {"params": model.other_params(),    "lr": lr}
], weight_decay)
scheduler = get_cosine_schedule_with_warmup(...)
```

2. **Loop:**

```python
for epoch in range(epochs):
  train_one_epoch(...)
  evaluate_on_val(...)
  save_checkpoint(...)
```

3. **Logging:**
    - Hook up **TensorBoard** or **Weights & Biases** to log:
        - mAP@0.5 for vehicles

        - Cutting-F1 score on val (compute PR curve)
        - Loss components

---

10. Smoke-Test on Vertex AI

1. **Submit** a short job (1 epoch, batch_size=1) using your CustomJob YAML.
2. **Verify:**
    - Job starts, logs appear in Cloud Logging/TensorBoard.
    - Model checkpoint lands in your GCS `checkpoints/` prefix.

---

11. Hyperparameter Sweep & Augmentation

- **Augmentations:**
    - Random flip, color jitter per frame
    - Sequence-level jitter (drop or swap one frame)
- **Sweep:**
    - Vary N (3 vs 5), lr, λ<sub>cut</sub>
    - Use Vertex AI Vizier or manual small-grid runs
- **Target by epoch 10:** mAP $\geqslant$ 0.75, Cutting-F1 $\geqslant$ 0.70.

---

12. Final Training & Inference

1. **Full run:** 30 epochs, best-pruned hyperparams, larger batch if memory allows.
2. **Eval job:** run `--eval_only` on val to pick best checkpoint.
3. **Inference script** (`inference.py`):
    - Loads best checkpoint
    - Reads test sequences from GCS

- Emits `Submission.csv` to GCS

That’s your end-to-end blueprint. At each stage, write minimal tests to catch shape mismatches and logic bugs early. Reserve your mentor slots to sanity-check the temporal-attention design and final submission format. Let me know which file you’d like to dive into first!

my rest of the team is working on the implementation , i am the one reseacrhing deep in architecture and how to implement that ,

lets start from the layman's perspective and deeply analyse the architecture and our implementation logic , keeping our project for hackathon at center

Let’s pull back and view our ViT + DETR + Temporal-Attention pipeline as if we were telling a smart friend over coffee—then drill into how each piece will map to our hackathon code.

---

1. High-Level Story: “How the Car Sees a Cut-In”

Imagine you have a short video clip—say 3 adjacent frames—from your car’s front camera. You want to answer three questions in one go:

1. **What vehicles are out there?**
2. **Where are they in the image?** (bounding boxes)
3. **Which of them are “cutting in”?**

Our pipeline does this end-to-end:

1. **See the raw frames** →
2. **Extract visual patterns** (edges, textures) with a Transformer backbone →
3. **Blend time information** so the model senses motion between frames →

4. **Shoot “object queries”** at that fused representation to pull out each car’s box & class →
5. **Ask each query** “hey, did you just cut in?” with a tiny yes/no network.

---

2. Breaking It Down, Layman-Style

A. Vision Transformer (ViT) Backbone

- **Think of each frame** as a jigsaw puzzle cut into 16×16 pixel tiles.
- **ViT learns**: “Here are all the puzzle-piece patterns that matter—wheels, windows, stripes…”
- **Output**: for each frame, it turns those tiles into a grid of feature vectors—little “pattern summaries.”

> **Hackathon note**: We’ll grab a pretrained ViT so we don’t reinvent the wheel; it already knows “cars look like this.”

B. Temporal Attention Module

- **Problem**: ViT sees each frame in isolation. But cutting-in is motion!
- **Solution**: Stack N frames’ feature grids, tag each tile with “this came from frame 1, 2, or 3,” then run a small Transformer over that stack.
- **Analogy**: It’s like lining up three comic-strip panels and letting the model “look back and forth” to spot movement—“aha, that white car moved left!”

> **Hackathon tip**: Keep N small (3–5) so training is quick; use just 1–3 “attention layers” here.

C. DETR Decoder & Queries

- **DETR’s twist**: instead of sliding windows or anchors, you maintain 100 “object detectors” learned end-to-end.
- **Mechanism**: Each of those 100 queries asks the fused feature “is there a car here?”
- **Output per query**:
  - A box ([x, y, w, h])
  - A vehicle class label (car, truck, bike…)

    - Or “no-object” if that query finds nothing.

> **Why this helps**: No more fiddling with pre-set anchor boxes. The model figures out where and how many cars by itself.

D. Cutting-In Head

- **After DETR** gives us 100 candidate vehicles, we take the internal representation of each and feed it into a tiny “doctor” network that says “0 = not cutting, 1 = cutting.”
- **Structure**: Two fully-connected layers + a sigmoid → probability.

> **Hackathon focus**: This head is cheap—once boxes are found, adding cutting-in classification barely slows you down.

---

3. From Diagram to Code: Implementation Logic

Here’s how we’ll map that story into files and functions:

Step	Code Component	Key Checkpoints
Load 3-frame clips	`sequence_dataset.py`	Returns `(frames_tensor, target_dict)` with correct shapes
Extract per-frame features	`models/backbone.py`	Dummy input → `(B, N, D, H', W')`
Fuse time	`models/temporal_module.py`	Backbone output in → same shape out
Detect & decode	`models/detr_decoder.py`	`(B, N, D, H', W')` + queries → `(B,100,D)`
Classify cut-in	`models/cutting_head.py`	`(B,100,D)` → `(B,100,1)`
Match & loss	`losses/matching.py`	Loss > 0, gradients flow
Train loop + logging	`train.py`	Forward/backward on dummy data works; logs to TensorBoard/GCS

Debug hacks along the way:

- **Shape-assert** liberally—catching a transpose error early saves hours.

- **Smoke-test** each module in isolation with tiny data.
- **Visualize** a few predicted boxes + labels on val images to sanity-check your pipeline.

---

4. Why This Will Win the Creativity Prize

- **Pure Transformer Flow**: from raw pixels → boxes → “cutting” in one unified graph.
- **Temporal Attention**: explicitly modeling motion instead of just frame differencing.
- **No Anchors, No ROI Align**: a sleek, modern approach that judges love for elegance.

And all of this fits into our 48-hour hackathon window because each piece reuses proven blocks (pretrained ViT, off-the-shelf DETR code) glued together with a small custom temporal layer and a binary head.

Next Deep-Dive

Pick one component you want to unpack further—e.g. the math behind Hungarian matching, or how to build those temporal positional embeddings—and we’ll zoom in.

lets run a deep research for it in refernce to my requirment with the cutting in model deelopment , lets start

To run focused research, could you please clarify what you'd like me to explore deeply for the cutting-in model development? For example:

1. Are you looking for research on existing academic papers or production models for detecting lane cut-ins?
2. Should I focus on transformer-based models applied to temporal vehicle behavior?
3. Are you interested in augmentation strategies, temporal modeling techniques, evaluation protocols, or something else?

Let me know which areas are most relevant to you so I can tailor the research accordingly.

i am interested in making my model , training over the 8 gb dataset gven to me , we are using vit based + detr architecture so i need entire thought over that idc what you are looking in bg , i need a nice overview in terms of logic and research

Perfect. I’ll now conduct a deep dive into building a cutting-in behavior detection model using a ViT + DETR + temporal attention pipeline, tailored specifically to your 8 GB hackathon dataset. I’ll explore implementation logic, training strategies, architectural insights, and relevant innovations to help you optimize for both accuracy and creativity.

I’ll get back to you with a detailed, structured overview shortly.

Cutting-In Detection Using ViT + DETR with Temporal Attention

Detection of Lane-Cutting Behavior with ViT-DETR and Temporal Attention

Introduction

Detecting **lane-cutting behavior** – where a vehicle abruptly merges into the ego lane – is a challenging vision task crucial for advanced driver assistance. It requires not only identifying vehicles in the scene, but also understanding their **motion patterns** over time. A single image often cannot reliably indicate a cut-in; the model needs **temporal context** to discern subtle lateral movements or angle changes that signify a vehicle is moving into the ego lane. For instance, on a highway a vehicle may unexpectedly perform a cut-in maneuver at high speed, merging from an adjacent lane directly in front of the ego vehicle arxiv.org. Human drivers anticipate such events from visual cues across time, and our model aims to do the same by leveraging a **Vision Transformer (ViT) backbone**, a **DETR detection head**, and a **temporal attention module** for multi-frame input. This report presents a comprehensive design of a ViT+DETR architecture tailored for lane-cutting detection, along with training strategies and practical considerations for implementation under limited time and compute (e.g. hackathon conditions).

Model Architecture Overview

Our model is an **end-to-end multi-task transformer network** that performs object detection and behavior classification simultaneously. It processes a **sequence of video frames (e.g. 3–5 frames)** to capture motion, and outputs bounding boxes for vehicles along with a binary label (`cutting` or `not cutting`) for each detected vehicle. The architecture consists of three main components:

- **Vision Transformer (ViT) Backbone:** Each frame is passed through a ViT-based image encoder to extract high-level visual features. The ViT treats each image as a sequence of patches, enabling a global receptive field and strong feature extraction. We use a pre-trained ViT (e.g. ViT-B/16) to leverage rich representations learned from large datasets, which accelerates convergence on our 8 GB custom dataset. The backbone produces a feature map or sequence of tokens for each frame (maintaining spatial layout information, possibly via reshaped patch embeddings). These per-frame features serve as input to the transformer encoder.
- **Transformer Encoder with Spatio-Temporal Attention:** The encoder extends DETR's encoder to handle **spatio-temporal input**. Rather than encoding each frame independently, we concatenate or interleave features from multiple time steps and apply **self-attention across both space and time**. Positional embeddings are added to indicate each token's 2D location in the image and its time index in the sequence. This design allows the encoder to learn correlations over time – effectively learning **object feature trajectories** across frames ar5iv.org ar5iv.org. We implement a **temporal positional encoding** (e.g. an encoding added to all patches of frame t) to inform the model of the frame ordering ar5iv.org. By employing full attention over the spatio-temporal sequence, the model can integrate motion cues (e.g. a vehicle's shift between frames) directly into the feature representations ar5iv.org iiict.uob.edu.bh.

- **DETR Decoder with Query-Based Detection:** We adopt the DETR (Detection Transformer) decoder paradigm with learned object **query embeddings** medium.com . A fixed set of queries (e.g. 100 queries) interact with the encoded spatio-temporal features via cross-attention. Each query will attend to features from **all frames** and latch onto a consistent object if present. We modify the DETR decoder to output detections for the **last frame** in the input sequence by default. During decoding, the queries aggregate information from the temporal context but ultimately produce one set of bounding boxes and classifications corresponding to the final frame (time T) ar5iv.org . This is analogous to sequence-to-sequence prediction where the input is a sequence of frames and the output is predictions at the last time step ar5iv.org . (In principle, one can also output detections for every frame by having the decoder produce a sequence of outputs ar5iv.org , but focusing on the last frame reduces redundancy since adjacent frames largely serve as context for the target frame's prediction.)
- **Output Heads – Class and Attribute:** Each decoded query produces two outputs: (1) a **bounding box** (with coordinates parameterized as [center_x, center_y, width, height] normalized to image size, similar to DETR), and (2) a **classification**. The classification itself is multi-task: in our case, all objects of interest are vehicles, so the primary class is simply "vehicle" (we assume non-vehicle objects are not labeled in the dataset). The more critical output is the **binary attribute** indicating lane-cutting (`cutting` vs `not cutting`). We integrate this attribute prediction in one of two ways:
  - **As a Separate Attribute Head:** The decoder's output embeddings feed into one feed-forward network (FFN) that predicts the object class (which may always be 'vehicle' here, or could predict specific types like car/truck if needed), and a parallel small FFN that predicts the binary cutting attribute for that query. This treats cut-in classification as an auxiliary output for each detected object.
  - **As Part of the Class Label:** Alternatively, we can expand the label space into two classes: "cutting-vehicle" vs "non-cutting-vehicle" (plus a "no object" class for empty queries). In this scheme, the decoder has a single classification head and directly outputs one of these two classes for each box. This effectively merges the detection and attribute tasks into one classification. However, it doubles the number of positive classes and might confuse the model if not enough examples of each category are present. A separate binary head is more flexible, especially if we wanted to enforce different loss weighting for the attribute.

In our design, we favor the **two-head approach** for clarity: one head for objectness/type and one for the cut-in attribute. Both heads operate on the same query embedding output, so the features learned by queries must represent both what and *how* the object is moving. This multi-task design is inspired by prior works that integrate object detection and attribute recognition in a unified model frontiersin.org frontiersin.org. By sharing the backbone and decoder for both tasks, the model can learn a representation that simultaneously localizes the vehicle and assesses its motion state, rather than splitting detection and behavior classification into separate pipelines frontiersin.org.

Figure 1: High-level architecture of the proposed ViT-DETR with temporal attention. A sequence of frames (here 4) is processed by a shared ViT backbone for feature extraction. The transformer encoder fuses these features with ***spatio-temporal self-attention****, and the decoder's learned queries attend to the fused features to output bounding boxes and a binary lane-cutting classification per detected vehicle (for the last frame in the sequence). The diagram illustrates an* ***early fusion*** *approach: spatial features from all frames are aggregated before decoding, enabling queries to attend across time ar5iv.org ar5iv.org. (Image adapted from ST-DETR architecture ar5iv.org.)*

Temporal Attention Module

A key innovation is the **temporal attention module** which enables multi-frame context modeling. We implement this within the transformer encoder/decoder as an **attention over time**. Two design options are considered (similarly to the early-vs-late aggregation in ST-DETR ar5iv.org ar5iv.org):

- **Early Temporal Fusion:** In this approach, we **stack spatial features from all frames** along the sequence dimension and feed the entire spatio-temporal sequence into the encoder at once ar5iv.org ar5iv.org. For example, if each frame's ViT backbone yields a feature map of shape $(H \times W, d)$ (flattened patches with dimension d), and we have T frames, then the encoder sees $T \times (H \times W)$ tokens. We provide a temporal position encoding to distinguish frame 1's patches from frame 2's, etc. The encoder's self-attention then operates in a **fully spatio-temporal manner**, meaning any patch can attend to any patch from any frame. This effectively creates feature traces: the encoder can learn to correlate the same vehicle's features across frames (since their spatial locations move slightly) and to differentiate moving objects from static background by how their embeddings change over time. The decoder then attends to this joint feature sequence. Early fusion directly gives the decoder access to motion-enhanced features (e.g. a vehicle's token might encode its motion trajectory) ar5iv.org ar5iv.org. The ST-DETR study found this **early aggregation** yielded the best detection accuracy, significantly improving mAP by ~5% over single-frame DETR roseyu.com ar5iv.org.
- **Late Temporal Fusion:** In a alternate design, we could encode each frame's features separately (as in standard DETR encoder per frame), and perform temporal fusion at the **decoder stage** ar5iv.org ar5iv.org. For instance, the decoder could first attend to each frame's features with separate sets of queries to get per-frame object predictions, and then a second stage of the decoder could perform **temporal self-attention on the sequence of query embeddings** (each query's trajectory over frames) ar5iv.org ar5iv.org. This would explicitly aggregate **object queries over time** – essentially tracking objects by linking query outputs from frame 1 to 2 to 3, etc., and then predicting the final state. The final output could be the queries at the last frame after they have attended to queries from earlier frames ar5iv.org ar5iv.org. While conceptually appealing for tracking, this *late fusion* was found to yield smaller gains in the ST-DETR experiment (only ~0.1% mAP better than no temporal fusion) compared to early fusion's large boost ar5iv.org. The likely reason is that early fusion allows low-level feature sharing across time (making it easier to detect a moving object), whereas late fusion only merges high-level detections, which might be too late to rescue missed detections of subtle motion ar5iv.org. Therefore, our design leans toward early fusion in the encoder, letting the model build motion-informed features before decoding.

Regardless of early or late integration, **temporal attention equips the model to utilize motion cues**. For detecting lane-cutting, this is crucial: the difference between a car that stays in its lane vs. one that cuts in may be a small lateral displacement or a change in heading angle over a few frames. The attention mechanism can pick up on patterns like a vehicle's bounding box shifting toward the lane marker or its front enlarging (getting closer) quickly. With multi-frame input, the transformer can implicitly estimate velocity and direction by comparing features across time. In effect, the model learns an internal representation akin to **optical flow** or object trajectories without explicitly computing them. (Optionally, one could also feed **optical flow frames** or frame differences as additional input channels to emphasize motion – indeed, the ST-DETR work showed that concatenating optical flow with RGB yielded a big jump in detecting moving objects ar5iv.org. For our hackathon scenario, if computational budget allows, augmenting each frame with its optical flow map could help the model pinpoint moving edges and thus better detect cut-in vehicles, which are by definition moving laterally.)

Another design detail is the **number of frames (T)** to use. Using more frames provides a longer temporal context (catching early signs of a lane change), but increases memory and may dilute the relevant info with very old frames. A short window (e.g. 3–5 frames, corresponding to perhaps 0.5–1 second of video) is a reasonable trade-off. ST-DETR found that going from 1 frame to 2 frames yielded large gains, while 4 frames gave only marginal additional improvement roseyu.com ar5iv.org. We might choose $T=3$ for efficiency, or $T=5$ if subtle motion needs a slightly longer observation. The frames should be consecutive (to accurately capture motion); however, skipping frames (using a small stride) could be considered if the video is high framerate and we want a slightly wider temporal coverage with fewer frames.

Temporal Modeling Benefits for Lane-Cutting Detection

Incorporating temporal modeling greatly **enhances the motion context** available to the model, which is especially beneficial for detecting lane-cutting behavior:

- **Distinguishing Static Snapshot vs Motion:** In a single image, a vehicle that is about to cut into your lane might look very similar to one that will continue straight. Small cues like the angle of the wheels or a slight lateral offset might not be salient enough. Temporal context lets the model see the **change** – e.g. the vehicle is closer to the lane line in frame 2 than in frame 1, indicating a drift. By attending to multiple frames, the model can infer **velocity and direction**, effectively performing a short-term trajectory prediction rather than a static classification. This mirrors how a human driver notices a car starting to drift over. The model's self-attention can learn to focus on the differences between frame features: if a car's position shifts rightward each frame, the attention weights could highlight that motion as evidence of a cut-in.
- **Improved Confidence in Classification:** Temporal cues provide **redundant evidence** for the model. If any single frame is ambiguous, the sequence can disambiguate. For example, glare or occlusion in one frame might hide a subtle cue, but another frame reveals it. The transformer naturally aggregates information across time, so the classification head can make a decision based on an enriched representation. Empirically, using even a small number of frames drastically boosts detection performance for moving objects ar5iv.org roseyu.com. In our context, the "action" of lane cutting is essentially a short-term event, so modeling it as an **action recognition problem** over a few frames is apt. (Indeed, prior work has treated lane-change detection as an action recognition task, using 3D ConvNets or LSTMs on video clips arxiv.org iiict.uob.edu.bh. Transformers allow us to fold action recognition directly into the object detection pipeline via attention.)
- **Reduction of False Positives:** Without temporal logic, a model might falsely label stationary lane-adjacent vehicles as cut-in threats. By seeing that a vehicle remains in its lane over multiple frames, the temporal model can learn to output `not cutting` with high confidence, whereas a single frame model might overreact to a position that looks slightly toward our lane. Essentially, temporal context adds **stability** and **consistency** to the predictions. The model can be trained to only label `cutting` when a sustained motion pattern is observed (e.g. requiring a consistent angle change over a sequence, analogous to how a heuristic might require a threshold over 5 frames github.com). This can be learned implicitly via the sequence input. For instance, if ground truth only flags an event once the motion is appreciable, the model will learn that just one frame of slight deviation isn't enough – it must see the trend.

- **Temporal Attention vs. Manual Motion Features:** An alternative to full attention would be to explicitly feed engineered motion features (like optical flow or consecutive frame differences) into a CNN. However, the transformer's global attention can learn to combine appearance and motion cues in a data-driven way. **Recent research highlights the power of video transformers** (like ViViT) in capturing spatio-temporal dependencies for driving scenarios iiict.uob.edu.bh iiict.uob.edu.bh. In a lane change prediction study, a ViT-based video model achieved over 85% accuracy in predicting lane changes 1 second before they occur, outperforming CNN-LSTM approaches, all while using fewer parameters iiict.uob.edu.bh iiict.uob.edu.bh. This indicates that transformers are not only effective but also parameter-efficient for temporal vision tasks. By using attention, our model can focus on the most telling parts of the video – e.g. the edges of a vehicle and lane marker where a change is happening – and ignore static background. This context-driven focus is a key reason we expect better performance than single-frame detectors or two-stage methods that might not fully exploit subtle temporal cues.

In summary, temporal modeling **mimics a short-term tracking mechanism** inside the network, giving it a sense of each vehicle's motion. A cut-in maneuver is essentially defined by motion ("merge into lane") arxiv.org, so leveraging motion is indispensable. We adopt the early-fusion transformer design to maximize the motion information available for the classification of 'cutting' vs 'not cutting'.

Dataset Processing and Augmentation

The dataset is about 8 GB, organized in a Pascal VOC-style structure with separate training, validation, and test splits. We assume it contains images (or video frames) with annotated bounding boxes around vehicles and an associated binary label indicating if that vehicle is cutting into the lane at that time. To prepare this data for our temporal model, we must consider how to form sequences and apply augmentations consistently.

Data Organization: If the data comes as videos or frame sequences, we will group frames into clips of length T for training. If it's discrete images with annotations, we need to ensure we can reconstruct the temporal order (e.g. file naming or timestamps). We will create sequences (sliding windows) of T consecutive frames from the same scene. For each sequence, only the last frame's annotations carry the 'cutting' labels (because that's the frame we predict on), but for training the model with temporal data, we can consider the entire sequence as one training sample. It's important to include negative sequences (no cut-in happens) as well as positive (cut-in happening), to train both outputs properly.

Augmentation Strategies: Detecting subtle motion like lane cutting can benefit from careful augmentation:

- *Spatial Augmentations:* Apply standard image augmentations to improve generalization, but **synchronize them across frames in a sequence**. For example, random horizontal flip, random brightness/contrast changes, scaling, and small rotations can be applied, but we must perform the *exact same transform to each frame of the sequence* so that the temporal consistency (motion) is preserved. If one frame was flipped and the next not, it would scramble the motion. Thus, we treat the multi-frame input as one entity for augmentation. Horizontal flipping in driving scenes effectively mirrors the scenario (a left cut-in becomes a right cut-in), which should be fine as long as the model isn't reliant on an absolute direction. Other useful augmentations include:
  - **Random Cropping/Padding:** e.g. slightly zoom into a random area (while ensuring the ego lane region is still visible). This can simulate camera jitter and force the model to rely more on relative motion than absolute position. We must adjust bounding box coordinates accordingly for the last frame's labels.
  - **Photometric adjustments:** random gamma, blur, or noise can help the model be robust to different lighting or slight sensor noise, ensuring it doesn't miss motion cues in less ideal conditions.

  - **Occlusion or CutMix:** possibly overlay objects or blank out regions in some frames (e.g. simulate an occlusion in one frame). If the model can see the object in other frames it can recover. This trains the temporal attention to handle missing data in one frame by relying on others.
- *Temporal Augmentations:* We can vary the frame rate or time gap between frames in the training sequences to make the model robust to different speeds. For instance, sometimes use consecutive frames (which might be, say, 0.1s apart), other times skip a frame (0.2s apart), effectively slowing down or speeding up the perceived motion. This can be done by sampling frame indices with a small random step. However, we must be cautious: if we vary too much, the model might not learn the true scale of motion corresponding to a cut-in. It could be safer to mostly use constant stride (e.g. always consecutive frames) for clarity, and only minor randomness.
- *Balance and Emphasis:* Lane-cutting events are often **rare** relative to normal driving. It's likely that in an 8 GB dataset (which might contain many driving hours), the majority of frames have no cut-in happening. To avoid a biased model that always predicts "not cutting", we will **balance the sampling**. This can include: oversampling sequences where cut-ins occur (possibly multiple different starting frames of the same event to get positive samples at various points in the maneuver), or using a loss function that penalizes misclassifying the rare class more. Ensuring the model sees enough positive examples is critical. Because we have to also train detection, we can't only feed scenes with cut-ins (we need negative examples too), so a reasonable approach is a weighted random sampler: pick a cut-in sequence with some higher probability than its raw frequency. Also, within a long cut-in sequence, not every frame is labeled as cutting (only when the vehicle is actually merging). It might help to include some pre-cut-in frames labeled as not cutting in training, so the model learns to output `not cutting` until the moment it should switch – though this borders on *prediction* rather than detection. We assume labels are frame-wise (each frame independently marked cutting or not for each vehicle), so we use them as given.

- *Geo-Specific Augmentations:* The mention of “Indian driving conditions” in a related project github.com suggests lane markings might be inconsistent, etc. For our model, if the dataset spans different locales or scenarios (highway vs city, clear lanes vs faded lanes), we might consider augmentations that simulate different lane marking visibility (e.g. adding random painted line markings or occluding them). However, since our approach does not explicitly detect lanes, we rely on the model’s learned behavior. We just want to ensure the model doesn’t overfit to one type of road appearance. Standardizing images (through normalization, etc.) and using a diverse training set are the main tactics here.

Training Procedure and Implementation Details

Framework: We will likely implement this model in PyTorch, given the popularity of DETR implementations and ViT models in PyTorch. Libraries like `torchvision` provide DETR variants and `huggingface/transformers` provides ViT, which can expedite development. We can start from a pre-trained ViT backbone (trained on ImageNet21k or similar) and possibly even initialize the DETR decoder from a model pre-trained on COCO (if available, e.g. DETR with ResNet backbone – we can still use its query embeddings and decoder weights, adapting the input projection for our ViT features). This transfer learning is important to handle an 8 GB dataset within limited time – leveraging pretrained weights can cut required training epochs dramatically.

Multi-GPU & Mixed Precision: Given the model complexity (transformer encoder-decoder over \$T\$ frames), training can be slow. We would enable **mixed-precision (FP16)** training to speed up and reduce memory. If multiple GPUs are available, we distribute the training (data parallelism), which linearly reduces training wall-clock time. In a hackathon, one might only have a single GPU, so efficient coding is key.

Batching: Each training sample is a sequence of T frames, which we can stack along the channel dimension or feed as a 5D tensor $(\text{batch}, T, C, H, W)$ into a model that handles time. We must adjust batch size such that it fits in memory. For example, if $T=3$ and each frame is 800×600 resolution, a batch of even 4 sequences means 12 images worth of data – this might be okay on a 16 GB GPU with FP16, but if not, we might use batch size 2 for safety. Larger batch helps training stability for transformers, so we could accumulate gradients for multiple steps to simulate a larger batch if needed.

Loss Functions: Training the model involves a **set-based loss** as in DETR ar5iv.org. We use **Hungarian matching** to assign predicted queries to ground-truth objects in the last frame medium.com. Each ground-truth vehicle in the last frame should be matched to one query prediction; unmatched queries are treated as background (`no object`). The loss is a sum of:

- **Detection Classification Loss:** e.g. cross-entropy on the object class. In our case, if we use a single class “vehicle” (plus background), this is a binary classification (vehicle vs no-object) for each query. If we incorporate multiple vehicle types or the combined vehicle+cutting classes, it becomes a multi-class cross entropy. Typically DETR uses a weight to down-weight the `no object` class since majority of queries are background. We will inherit those hyperparameters from DETR (e.g. a weight $\alpha=0.1$ for no-object class to focus loss on real objects).
- **Bounding Box Regression Loss:** a combination of L_1 loss and generalized IoU loss for the predicted box vs ground truth box medium.com. We use the same formulation as DETR: $L_{\text{box}} = L_1(b_{pred}, b_{gt}) + \lambda_{\text{iou}} \cdot (1 - \text{IoU}(b_{pred}, b_{gt}))$ for matched pairs. Unmatched (no-object) predictions don’t contribute to bbox loss. Typically λ_{iou} is set around 1 or 2, and L_1 term weight around 5 (these were values used in DETR on COCO).

- **Cutting Attribute Loss:** For the binary attribute, we add a **binary cross-entropy loss** or equivalent. If using the separate head approach, this loss is only applied to predictions that are matched to a ground-truth vehicle. Each such prediction should output the correct attribute (cutting=1 or not=0). If using the combined class approach, then this attribute loss is effectively part of the classification loss (i.e. distinguishing cutting-vehicle vs non-cutting-vehicle). We may prefer to *emphasize the attribute classification* by giving it a higher weight, especially if the dataset is imbalanced or if detection is easier than classification. For instance, if detecting vehicles is already quite solved (with pretrained backbone), the challenge is mainly in classifying cut-in, so we might set a weight like 2.0 for the attribute loss versus 1.0 for the detection loss, to make the model prioritize getting the cut-in label correct. We must be careful, however, because a trivial way to get attribute loss down is to miss detections of cutting vehicles altogether. The Hungarian matching can be extended to include the attribute in the matching cost (so that matching a ground truth cutting-vehicle with a prediction that calls it non-cutting is considered worse), but that complicates training. A simpler heuristic: during matching, treat all vehicles equally (match by position), and then apply attribute loss after. This way the model is forced to detect the vehicle, then punished if it flags the attribute wrong.

We use the **same Hungarian matching algorithm as DETR** medium.com : the cost matrix between predictions and ground truths can be composed of (i) box overlap cost (e.g. L_1 or IoU) and (ii) class cost (probability of vehicle vs no-object, since all GT here are vehicles). We might not include the attribute in matching cost to keep it straightforward (thus essentially, a ground truth cutting vehicle is treated as just a vehicle for matching; once matched, its attribute contributes to loss).

Optimizer and Schedule: We will use **AdamW** optimizer (Adam with weight decay) which is standard for transformer training. A learning rate on the order of $1e-4$ to $2e-4$ is typical for DETR with backbone training. However, since we have a pretrained backbone, we might use a *layer-wise learning rate decay* strategy: higher LR for the randomly initialized parts (the decoder and new classification head) and lower LR for the ViT backbone to avoid destroying prelearned features. For example, backbone could use LR $1e-5$ while decoder and heads use $1e-4$. Another approach is to **freeze the backbone for the first few epochs**, train the transformer decoder and classification head alone, then unfreeze backbone with a low LR to fine-tune. This can stabilize training initially and save time by not constantly updating a huge number of backbone parameters until the heads are roughly calibrated.

We will train for a relatively **limited number of epochs** given hackathon constraints. Original DETR famously needed 500 epochs on COCO for full convergence medium.com, but with our pretraining and smaller scope, we expect far fewer. Many DETR follow-ups (e.g. Deformable DETR, DN-DETR) show convergence in 50 epochs or even less medium.com medium.com. Our dataset (8 GB, perhaps on the order of tens of thousands of images) might converge in ~50 epochs of training. We can plan, say, **50 epochs with early stopping** if the val loss plateaus. If 50 epochs cannot be done in time, we may settle for 20–30 and rely on heavy augmentation and pretraining to not overfit. We also incorporate a **learning rate schedule**: a common scheme is a **linear warmup** for the first epoch or two (to avoid unstable large gradient updates initially), then **cosine decay** or step decay. For instance, DETR often used dropping LR by 10x at 2/3 of training. We could drop LR at epoch 40. If using AdamW, we'll keep weight decay around 0.1 for regularization (transformers benefit from relatively high weight decay to prevent overfitting).

Efficiency Tricks: If training is too slow, one trick is to use a smaller image size for training initially. We could train with images resized to e.g. 600×400 for first pass, then fine-tune a few epochs at full resolution (1200×800 or whatever the dataset native) for final refinement. This can drastically cut computations early on. Also, since DETR doesn't rely on anchor boxes, we can freely scale images; just need to ensure coordinates are normalized and the model's position embeddings can handle the size (ViT has fixed patch embeddings which can extrapolate with interpolation). We could also reduce the number of queries if the scene typically doesn't have too many vehicles (though lane merge scenarios can have multiple vehicles around). DETR typically uses 100 queries for COCO (complex scenes); if our scenes are simpler we might use 50 queries to speed up decoding.

It’s worth noting that **Deformable DETR** is an improved variant that significantly speeds up convergence by focusing attention to sparse regions medium.com. While implementing full Deformable DETR in a hackathon is complex, we can borrow the intuition: restrict attention to relevant spatial locations. For example, we might limit the spatial extent of the encoder’s attention (like a local window, as in Swin Transformer) to reduce quadratic cost, but given a short sequence and moderate resolution, the full attention might be acceptable.

Validation: We will monitor validation set performance (mAP for detection and accuracy or F1 for the cutting classification) to choose the best model. Especially ensure that the model is actually learning the cut-in behavior and not just overfitting to context cues (like perhaps always tagging a vehicle in a particular position as cutting). A sanity check is to visualize some outputs: e.g., if a car is incorrectly labeled cutting, see if maybe a lane line was misdetected by model, etc.

Inference Strategy

At inference (deployment), to detect lane-cutting in real time, we process a continuous video stream in a sliding window fashion. A simple approach: for each new frame, take the last T frames (including it) and run the model to get detections on the newest frame. Because our model processes T frames jointly, there will be some latency (we need to wait for T frames to accumulate before the first prediction). But if T is small (3–5), this is on the order of 0.1–0.2 seconds at 30 FPS, which is negligible for the application. We do overlapping windows so that each frame gets one prediction (except the very first few frames). This effectively means a lot of repeated computation (frame 2 is analyzed with frames 0–4, then frame 3 with 1–5, etc.). To optimize, one could implement a rolling buffer in the model: e.g. use the previous sequence's encoded features and just append the new frame and drop the oldest (some research on persistent queries or memory in transformers could be applied). However, for simplicity, running the full model anew for each frame might be fine if the model is relatively efficient and hardware (like a GPU) is available.

During inference, **post-processing** is minimal because DETR outputs final boxes without needing NMS (non-maximum suppression) due to set prediction training medium.com. We will threshold the confidence scores: if using separate class and attribute, we require the vehicle confidence to be high enough to trust the detection, and then use the attribute classification directly. If using the combined class approach, we interpret a "cutting-vehicle" class prediction with sufficient confidence as a positive event. Because lane-cutting can be safety-critical, one might choose a relatively low threshold to not miss true cut-ins (high recall), then perhaps apply some smoothing: e.g., require that the model predicts a vehicle as cutting in for 2 consecutive frames before we trigger an alert, to avoid one-frame flukes. This effectively could reduce false alarms by using temporal persistence at decision level (though the model itself is already using temporal info). Given our model itself sees multiple frames, it might be stable enough that if it sees a cut-in on frame t, likely on frame $t+1$ it will also see it (since it's essentially looking at overlapping frame sequences).

If multiple vehicles are detected, each is classified independently by the model. We should ensure the output includes the identity of which vehicle is cutting. For user interpretation, one might highlight the bounding box in red if cutting vs green if not, etc. If needed, the model's output can be combined with a tracking ID from an external tracker to maintain consistency of labeling a particular car as it moves. But our model itself doesn't explicitly ID objects across frames, it just outputs for the current frame. In practice, this is sufficient for instantaneous detection. (If one wanted to predict *future* cut-in, that's a different task of intention prediction. Here we stick to detecting ongoing or about-to-happen cut-ins as they occur.)

Runtime considerations: Using a ViT+DETR architecture is heavier than a YOLO model. For hackathon demonstration, if real-time operation is needed, we might have to reduce complexity:

- A base ViT (ViT-Base) with $T=3$ and DETR decoder might run at only a few FPS on a single GPU, which could be borderline. One could switch to a **smaller backbone** (ViT-Small or even a CNN backbone like ResNet50) and fewer encoder/decoder layers to speed it up. Also, using $T=3$ instead of 5 helps.
- Another trick: since cut-in events don't happen every frame, one could run the model at a lower frame rate (e.g. analyze 1 out of every 3 frames). Missing a few frames is usually fine, you'd still catch the event within a tenth of a second delay or so, which is acceptable.

Evaluation and Expected Performance

We will evaluate the model on the test split using standard detection metrics (mAP for vehicle detection at IoU threshold, etc.) and a metric for cut-in classification. Since it's a dual-task, one suitable metric is **mAP with respect to the cut-in class**: treat "cutting" vs "not cutting" as two classes of objects and compute average precision for the "cutting" class. This would naturally require the model to both detect the vehicle and identify it as cutting for it to count. Alternatively, we can report detection AP for vehicles and the **classification accuracy** (or F1) of the attribute on correctly detected vehicles. High precision in cut-in detection is important (we want few false alarms), but recall is also critical (missing a cut-in could be dangerous). Thus, the model should ideally achieve a high F1 score on the cut-in classification. We anticipate that with temporal input our method will far outperform single-frame baselines in catching these events. For example, without temporal info, many cut-ins would be missed or guessed randomly; with our multi-frame model, the relevant mAP or F1 should improve substantially (similar to how ST-DETR improved moving object detection by 5 mAP points with 4 frames roseyu.com).

Comparisons with Alternative Approaches

YOLO (CNN-Based One-Stage Detector): YOLOv5/YOLOv8 are efficient one-stage detectors known for real-time performance. A YOLO model could be trained to detect vehicles and even to classify them with an extra output for cut-in. However, YOLO by itself processes one frame at a time, lacking temporal reasoning. To use YOLO for cut-in detection, one must introduce temporal logic externally. One straightforward approach is a **post-processing algorithm**: run YOLO on each frame to get vehicles, then analyze the sequence of detections to decide if a cut-in occurred. For instance, an Intel workshop project did exactly this – they used YOLOv8 to detect vehicles and then computed each vehicle's orientation angle relative to the lane across frames to identify cut-in events github.com github.com. They defined a cut-in by metrics like a significant change in the vehicle's angle (>1.5° over 5 frames) and a short time-to-collision (<0.8s) github.com. These heuristics essentially mimic what our learned model would do, but in a rule-based manner. The YOLO+heuristic approach can work reasonably if tuned well; it benefits from YOLO's fast detection and doesn't require training a temporal model. However, it is *not as adaptable*: fixed thresholds might fail for subtle or slower cut-ins, and the system might miss context (e.g. if lane markings are absent, their angle method might break, whereas a learned model could infer lane position from vehicles' relative motion). Our transformer model integrates the detection and temporal reasoning, learning the threshold for "significant movement" from data rather than a manual setting. This should make it more **robust to varied scenarios**. In terms of speed, YOLO is definitely faster (often >30 FPS on a GPU, even >100 FPS for small models). If real-time operation on low-power hardware is a must, YOLO with some temporal post-filter might be the practical choice. But if accuracy is paramount, especially for edge cases, the ViT+DETR approach has an advantage of **jointly optimized features**: the classification of cut-in will be using the same visual cues that help detect the vehicle, potentially yielding a more nuanced understanding.

Another way to incorporate temporal context with YOLO could be to input *multiple frames into the network*. For example, stack T frames as different channels (e.g. 3 frames = 9-channel image) and feed to YOLO. This gives the CNN some ability to derive motion (similar to optical flow) by seeing differences across channels. Research on 3D CNNs or two-stream networks uses such stacking. But YOLO's architecture isn't designed for 3D kernels, so one would have to train from scratch, which is non-trivial in a short timeframe and still doesn't leverage explicit temporal attention.

Faster R-CNN (Two-Stage Detector) with Temporal Module: A classical two-stage approach (like Faster R-CNN) could be extended for this task by adding an **LSTM or transformer on ROI features**. For example, one might detect vehicles in each frame, then take the ROI feature vector for a particular vehicle across consecutive frames and feed that sequence to an LSTM that outputs whether that track is cutting in or not. There has been work in action detection in videos following this paradigm: detect person per frame, then classify the sequence of bounding boxes to recognize the action. We could analogously treat a vehicle's sequence as a time-series and classify it. However, implementing this requires solving the **tracking association** (knowing which vehicle in frame 1 corresponds to which in frame 2, etc.) and then managing sequences of varying length. In a complex traffic scene with many vehicles, maintaining separate LSTMs for each track is heavy. Some modern approaches use transformers for **joint detection and tracking** (e.g. Trackformer or MEDET) where query embeddings persist over time to track objects. Our architecture is conceptually closer to those, but simplified since we don't explicitly enforce identity consistency of queries across frames (though early fusion tends to cause the same query to latch onto the same object through time implicitly).

If one already has a good single-frame detector, a quicker hack could be: run the detector on each frame to get boxes, link boxes into trajectories (simple IoU-based matching from frame to frame), then compute features like lateral speed or lane position change from those trajectories and feed them into a small classifier (like an SVM or MLP) to decide cut-in. This modular approach might work if reliable tracking is possible. But in dense traffic or with missed detections, tracking can fail, whereas the end-to-end transformer can handle moderate occlusions by using attention to reidentify objects.

Vision Transformer vs CNN for this Task: Vision Transformers are known for needing large data to train, but here we fine-tune a pre-trained ViT, which has shown excellent transfer learning results. A ResNet-based DETR could also incorporate temporal frames (e.g. by 3D convolutions or multiple frames concatenated as input in the encoder). That approach might be slightly less memory intensive (CNN features can be downsampled more aggressively than ViT patches). But ViT offers **global receptive field from the start**, which might help to capture global context like the position of a vehicle relative to all lanes. Also, ViTs have been found to be powerful in modeling relationships between multiple objects – for example, maybe the model can learn that if the ego vehicle (camera) is behind another car and that car is near a third car in adjacent lane, certain relative motions foreshadow a cut-in. These relational cues could be captured by self-attention better than by localized CNN filters. The **trade-off** is that ViT+DETR might require more computation than, say, a YOLOv8 which is highly optimized. In hackathon conditions, one might start with a smaller transformer (like Swin-T or ViT-S) to get a prototype working.

Latest Research Insights: Our approach aligns with the trend of using transformers for multi-task vision problems. There are recent models specifically tackling **multi-task object detection** (like **YOLOP / YOLOPV3** which do detection + segmentation ieeexplore.ieee.org, or **MPAR-RCNN** which does detection + person attributes frontiersin.org). They show that a unified model can perform as well or better than separate models for each task, thanks to shared features. We leverage that by sharing features for detection and cut-in classification, which should improve efficiency (no need to run two separate networks) and possibly even accuracy (the tasks reinforce each other; e.g. knowing motion might help focus detection, and knowing the object's location helps contextualize its motion). Additionally, advances like **DINO (DETR with denoising)** have made transformer detectors much more training-friendly, converging 10× faster than the original DETR medium.com medium.com. If we had more time, integrating such techniques (e.g. using denoising queries during training to stabilize learning) could further improve our results. There is also a trend of applying *contrastive learning* in multi-frame setups to ensure temporal consistency of features (as hinted by temporal extensions of DINO and other models medium.com). For our scope, we primarily focus on the core architecture and basic training, but these insights guide our choices (e.g. using a pretrained backbone and possibly smaller LR to mitigate the known convergence issues of naive DETR).

Summary of Suitability: The ViT+DETR with temporal attention is **highly suitable for lane-cutting detection** because it naturally combines object detection with temporal motion analysis in one framework. Compared to a pure CNN solution, it should catch nuanced behaviors with fewer hand-crafted rules, at the cost of more involved training. In a time-limited development setting, one must weigh this complexity. One pragmatic approach is to implement a **baseline first** (say, DETR with one frame or YOLO+rule) to have something working, and then incrementally add the temporal transformer for improved performance. If the hackathon provides GPUs and the dataset is well-organized, an end-to-end solution can certainly be trained within a day or two with careful tuning. The expected outcome is a model that, given a short video clip, can **draw boxes around vehicles and label those cutting into the lane**, helping the ego vehicle (or a human analyst) to identify potentially dangerous cut-in maneuvers quickly and accurately.

References:

- Carion et al., “End-to-End Object Detection with Transformers,” ECCV 2020 – introduced DETR medium.com .
- Zhu et al., “Deformable DETR,” ICLR 2021 – improved DETR’s training speed medium.com .
- E. Mohamed & A. El-Sallab, “ST-DETR: Spatio-Temporal DETR,” 2021 – extended DETR to video sequences with temporal attention ar5iv.org ar5iv.org .
- Liang et al., “Lane Change Classification and Prediction with Action Recognition Networks,” arXiv 2024 – used 3D CNNs for lane change detection arxiv.org .
- Raja et al., “Lane Change Prediction of Surrounding Vehicles using Video Vision Transformers,” IJCDS 2025 – applied ViViT transformer achieving >85% accuracy for lane change prediction iiict.uob.edu.bh .
- Raghavendra et al., “MPAR-RCNN: Multi-task person detection with attribute recognition,” Frontiers AI 2025 – example of unified detection+attribute model frontiersin.org .
- Kaushal-Bora et al., “Vehicle Cut-In Detection for Indian Driving,” GitHub 2023 – YOLOv8 plus geometric analysis approach github.com github.com .

Sources