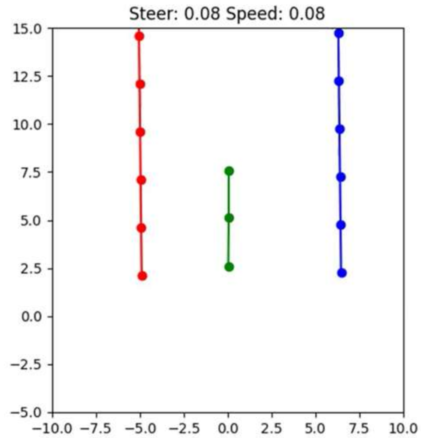
Dataset

- Dataset drive_data
- from .datasets.road_dataset import load_data

- data["track_left"]: [10, 2], float32
- data["track_right"]: [10, 2], float32
- data["waypoints"]: [3, 2], float32
- data["waypoints_mask"]: [3], bool



- track_left and track_right, each with a shape of [10, 2], store floating-point values representing coordinates of the left and right boundaries of the driving track, respectively.

- waypoints is shaped [3, 2], containing specific points that guide our vehicle's path.

- waypoints_mask is a boolean tensor of shape [3], which indicates the validity of each waypoint. A True value means the waypoint is active and should be used, while a False value indicates it should be ignored.

Dataset

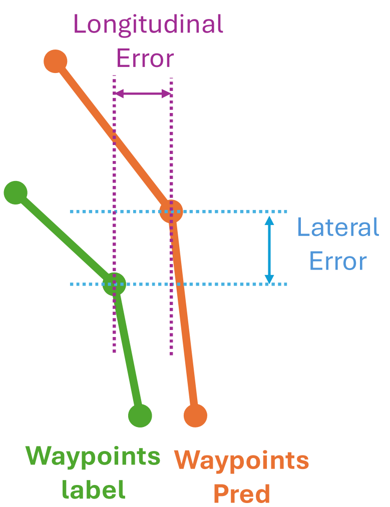
- Not all waypoint labels are valid
- Refer to PlannerMetric() in metrics.py
- Implement the same in loss function

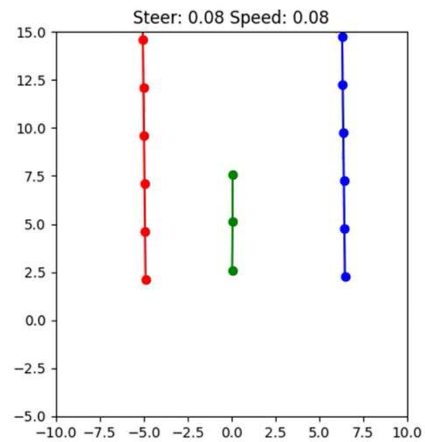
```python
class PlannerMetric:
    """
    Computes longitudinal and lateral errors for a planner
    """

    def __init__(self):
        self.l1_errors = []
        self.total = 0

    def reset(self):
        self.l1_errors = []
        self.total = 0

    @torch.no_grad()
    def add(
        self,
        preds: torch.Tensor,
        labels: torch.Tensor,
        labels_mask: torch.Tensor,
    ):
        """
        Args:
            preds (torch.Tensor): (b, n, 2) float tensor with predicted waypoints
            labels (torch.Tensor): (b, n, 2) ground truth waypoints
            labels_mask (torch.Tensor): (b, n) bool mask for valid waypoints
        """
        error = (preds - labels).abs()
        error_masked = error * labels_mask[..., None]

        # sum across batch and waypoints
        error_sum = error_masked.sum(dim=(0, 1)).cpu().numpy()

        self.l1_errors.append(error_sum)
        self.total += labels_mask.sum().item()
```

As illustrated in the code snippet, we calculate errors between predictions and labels, but critically, we multiply these errors by the labels_mask. This masking ensures that only valid waypoints contribute to our error measurement.

It's essential to incorporate the same masking approach into the loss function during training to ensure consistency between evaluation and optimization.

Metrics





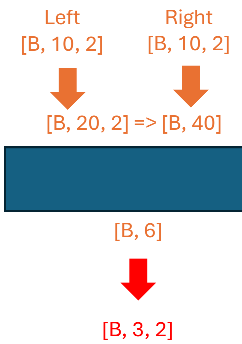
- Longitudinal error, shown here in purple, measures how far sideways, our prediction deviates from the correct waypoint. (x-axis)
  - When the track is straight, the waypoints stay in the middle of the track.
  - When the track is turning, the waypoints are closer to the sides.
  - This metrics is easier to achieve.

- Lateral error, marked in blue, represents how far our predicted waypoint is along the driving path compared to the actual labeled waypoint. (y-axis)
  - This metric reflects the speed of the kart.
  - This metrics is difficult to meet. We don't expect this metrics to be as good as the longitudinal error. The grading criteria is more relaxed on this one.

- Waypoints are the locations of the kart. The first one is the current location. The other two are future locations.
  - The track boundary does not provide direct insights about how fast the test kart ran to generate the labels.
  - If your TA is a reckless driver, the 3 waypoints would be further apart in y-axis.

- Your mission, should you choose to accept it, is to
  - Keep the longitudinal error small so that the kart does not drive off the track
  - Keep the lateral error under control so that you can finish at least finish 50% of the track in time.
  - For this mission, we provide you with a gentle driver, who only steers and accelerates if necessary. No brake and absolutely no drift. See evaluate.py.

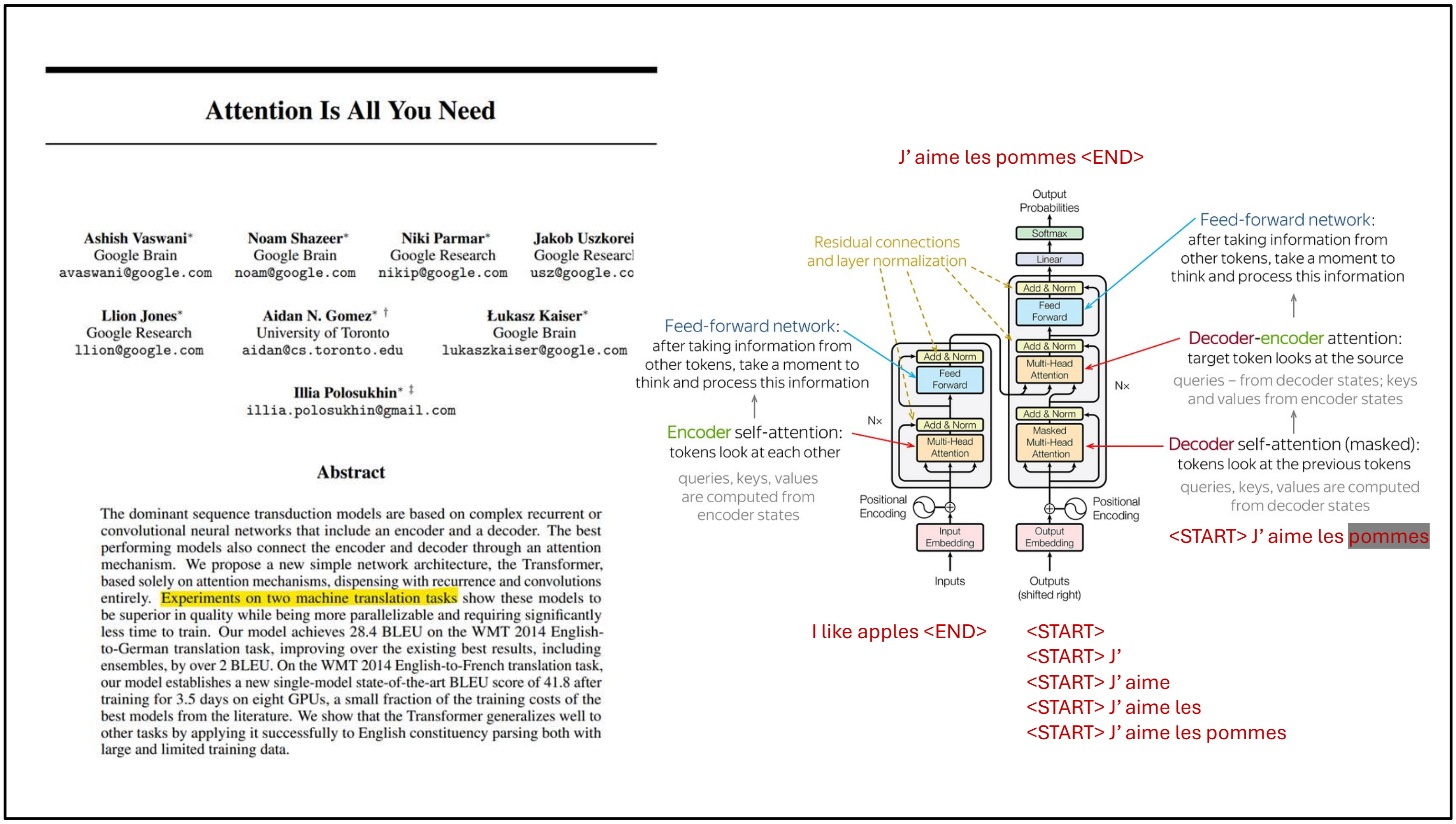
MLP Planner




```python
class MLPPlanner(nn.Module):
    def __init__(
        self,
        n_track: int = 10,
        n_waypoints: int = 3,
    ):
        """
        Args:
            n_track (int): number of points in each side of the track
            n_waypoints (int): number of waypoints to predict
        """
        super().__init__()

        self.n_track = n_track
        self.n_waypoints = n_waypoints

    def forward(
        self,
        track_left: torch.Tensor,
        track_right: torch.Tensor,
        **kwargs,
    ) -> torch.Tensor:
        """
        Predicts waypoints from the left and right boundaries of the track.

        During test time, your model will be called with
        model(track_left=..., track_right=...), so keep the function signature as is.

        Args:
            track_left (torch.Tensor): shape (b, n_track, 2)
            track_right (torch.Tensor): shape (b, n_track, 2)

        Returns:
            torch.Tensor: future waypoints with shape (b, n_waypoints, 2)
        """
        raise NotImplementedError

```

Transformer Planner

```markdown
### Part 1b: Transformer Planner (35 points)

We'll build a similar model to Part 1a, but this time we'll use a Transformer.

Compared to the MLP model, there are many more ways to design this model!
One way to do this is by using a set of `n_waypoints` learned query embeddings to attend over the set of points in lane boundaries.
More specifically, the network will consist of cross attention using the waypoint embeddings as queries, and the lane boundary features as the keys and values.

This architecture most closely resembles the [Perceiver](https://arxiv.org/pdf/2103.03206) model, where in our setting, the "latent array" corresponds to the target waypoint query embeddings (`nn.Embedding`), while the "byte array" refers to the encoded input lane boundaries.

<img src="assets/perceiver_architecture.png" width="600">

Training the transformer will likely require more tuning, so make sure to optimize your training pipeline to allow for faster experimentation.

For full credit, your model should achieve:
- < 0.2 Longitudinal error
- < 0.6 Lateral error

### Relevant Operations
- [torch.nn.Embedding](https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html)
- [torch.nn.TransformerDecoderLayer](https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoderLayer.html)
- [torch.nn.TransformerDecoder](https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html)
```



The landmark paper "Attention Is All You Need" introduced Transformers. The primary experiment was on machine translation.

In the model diagram, the encoder uses self-attention to allow tokens within the input sequence to directly relate to each other. The decoder applies masked self-attention — tokens can only look at previous positions — alongside decoder-encoder attention, where target tokens pay attention to relevant parts of the source input.

Additionally, residual connections and feed-forward networks refine these interactions, allowing the model to efficiently capture deep relationships between words.

Self Attention

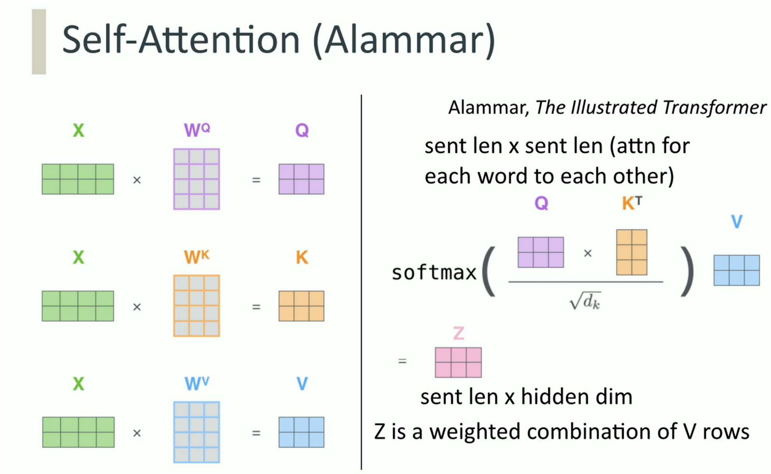
X: Input
K: Embedding of the input sequence
Q: What we want to find
V: The output

scores

$$softmax\left(\frac{\boxed{QK^T}}{\boxed{\sqrt{d_k}}}\right)V$$

Normalize
control softmax scale

$Q = XW_Q$
$K = XW_K$
$V = XW_V$



Let's look deeper into the core of the Transformer architecture: self-attention.

Here, inputs X are transformed into three key components: Queries (Q), Keys (K), and Values (V). Queries represent what we want to find; Keys are embeddings that help us locate relevant information; and Values contain the information we ultimately use to generate outputs.

In self-attention, Q, K and V have the same source of input (X). At the beginning:

- We don't quite know how to ask (query) the question. → Train the model to learn W_Q. → The model knows how to ask precise questions $Q = XW_Q$.
- We don't know how to interpret the environment input. → Train the model to learn W_K. → The model knows how to interpret the input $K = XW_K$.
- We don't know how to evaluate the value for proper output. → Train the model to learn W_V. → The model knows how to present the value $V = XW_V$.

The self-attention mechanism calculates scores by multiplying Queries (Q) with the transpose of Keys (K^T). These scores are then normalized and scaled using a softmax function to control the attention's sensitivity. This normalization ensures stability and helps the model understand how each part of the input relates to the others.

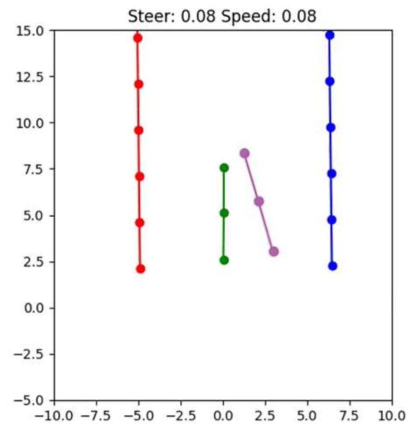
The visualization from Alammar clearly demonstrates this process: each input sequence X is transformed independently into Q, K, and V, and the final output is a weighted combination — highlighting precisely how Transformers dynamically adjust attention to capture meaningful relationships within data.

If Q, K and V have different input X, we call it cross attention.

Transformer Intuition

Find the concrete result (value) with some hints (query) from known data (key).

- Query: **Some hints** to help get to **Value**
- Key: Known data → Road boundaries (**left** and **right**)
- Value: **The correct waypoints**



- What is inside the box?

Query	**Key**	**Value**
Is it an animal or plant?	Box	Animal
Does it move by 2 limbs?	Box	It can
Does it have fur?	Box	Maybe
…		

Maybe the self-attention is still too abstract. Let’s take a look at some toy examples.

The transformer model helps us find precise answers (values) using hints (queries) based on known information (keys). Imagine driving on a road: your queries are hints guiding you, keys represent road boundaries, and values are the correct waypoints to follow.

The training process is like the “What’s inside the box?” game:

Let's say we have a box and we're trying to understand what's inside. We ask queries like: "Is it an animal or plant?", "Does it move using two limbs?", or "Does it have fur?" Each query, combined with known information (the key), helps the transformer get closer to accurately identifying the value, or what's actually inside the box.

Transformer Issues

- Scale poorly with increasing input size
  - Quadratic complexity $O(n^2)$ due to attention mechanism.
  - Struggle with long sequences or high-dimensional data (image, video, audio)
- Recall image classification in HW2 and HW3
  - MLP: Fixed input size, simple but large models
  - CNN: Variable input size, sophisticated but small models
- We need a new way of modeling to decouple input from model complexity.

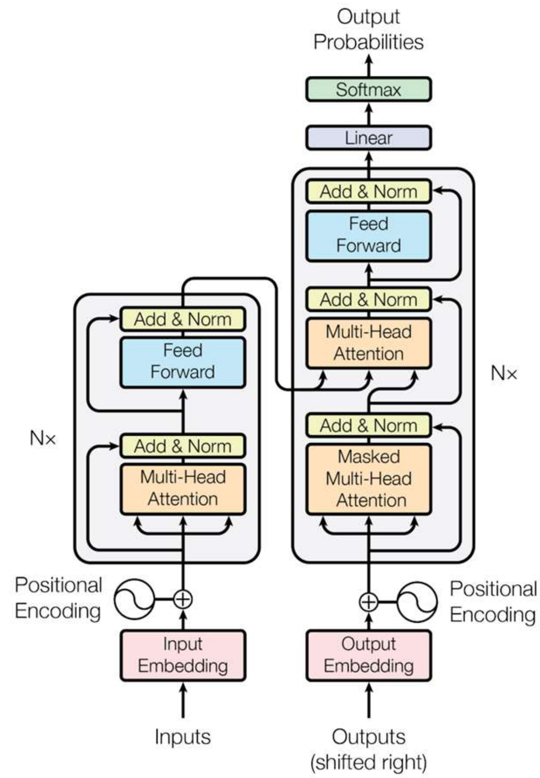


Figure 1: The Transformer - model architecture.

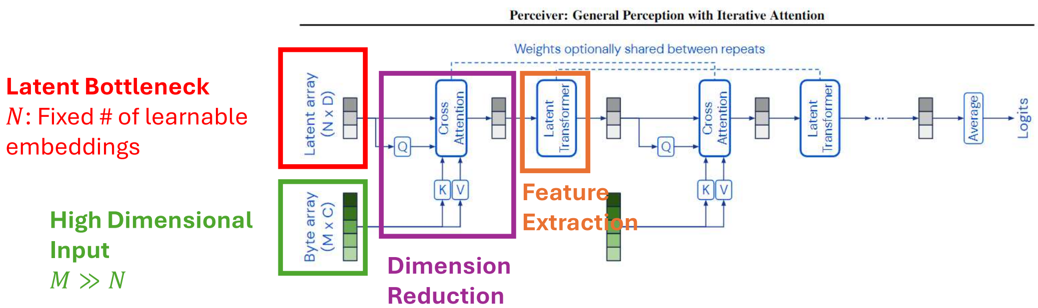
Traditional transformers, despite their powerful performance, face challenges when scaling to larger inputs. Their attention mechanism results in quadratic complexity $O(n^2)$, making them particularly inefficient when processing long sequences or high-dimensional data like images, videos, or audio signals.

To contextualize, recall our image classification tasks from previous homework assignments. Multilayer Perceptrons (MLPs) required fixed-size inputs, resulting in simpler but substantially larger models. Convolutional Neural Networks (CNNs), on the other hand, allowed variable input sizes, creating more sophisticated yet compact models.

Given these observations, it's clear we need a new modeling strategy — one that effectively separates input size from model complexity, enabling efficient and scalable performance for diverse data types.

Perceiver

- Andrew J. et. al., Perceiver: General Perception with Iterative Attention
  - Simultaneously processing high-dimensional inputs.
  - To scale, leverages an asymmetric attention to iteratively distill inputs into a tight latent bottleneck.



To address the scaling limitations of Transformers we discussed previously, let's explore the Perceiver model, introduced by Andrew Jaegle and colleagues.

The Perceiver provides a solution by simultaneously handling high-dimensional input data while maintaining computational efficiency. It accomplishes this using an asymmetric attention mechanism, which iteratively reduces the dimensionality of input data.

Specifically, it employs a fixed-size latent bottleneck (in red), which consists of a limited number of learnable embeddings. Through cross-attention (in purple), the Perceiver achieves dimensionality reduction by projecting high-dimensional input data into this smaller latent space. Subsequently, self-attention (in orange) is applied within this latent space to perform feature extraction and model complex interactions between latent embeddings. By repeatedly alternating between these cross-attention and self-attention stages, the Perceiver efficiently scales, effectively decoupling input dimensionality from model complexity.

Perceiver IO

- Andrew J. et. al., Perceiver IO: General Architecture for Structured Inputs & Outputs
  - Augments the Perceiver to enable outputs of various sizes and semantics.

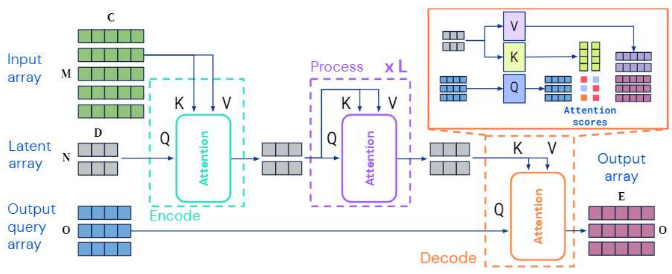
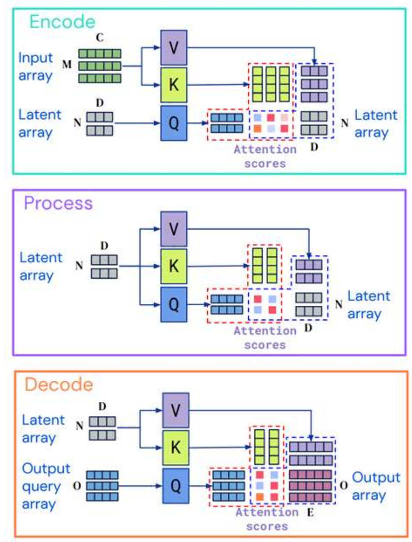


Figure 2: The Perceiver IO architecture. Perceiver IO maps arbitrary input arrays to arbitrary output arrays in a domain agnostic process. The bulk of the computation happens in a latent space whose size is typically smaller than the inputs and outputs, which makes the process computationally tractable even for very large inputs & outputs. See Fig. 5 for a more detailed look at encode, process, and decode attention.



Dimension Reduction by Cross Attention

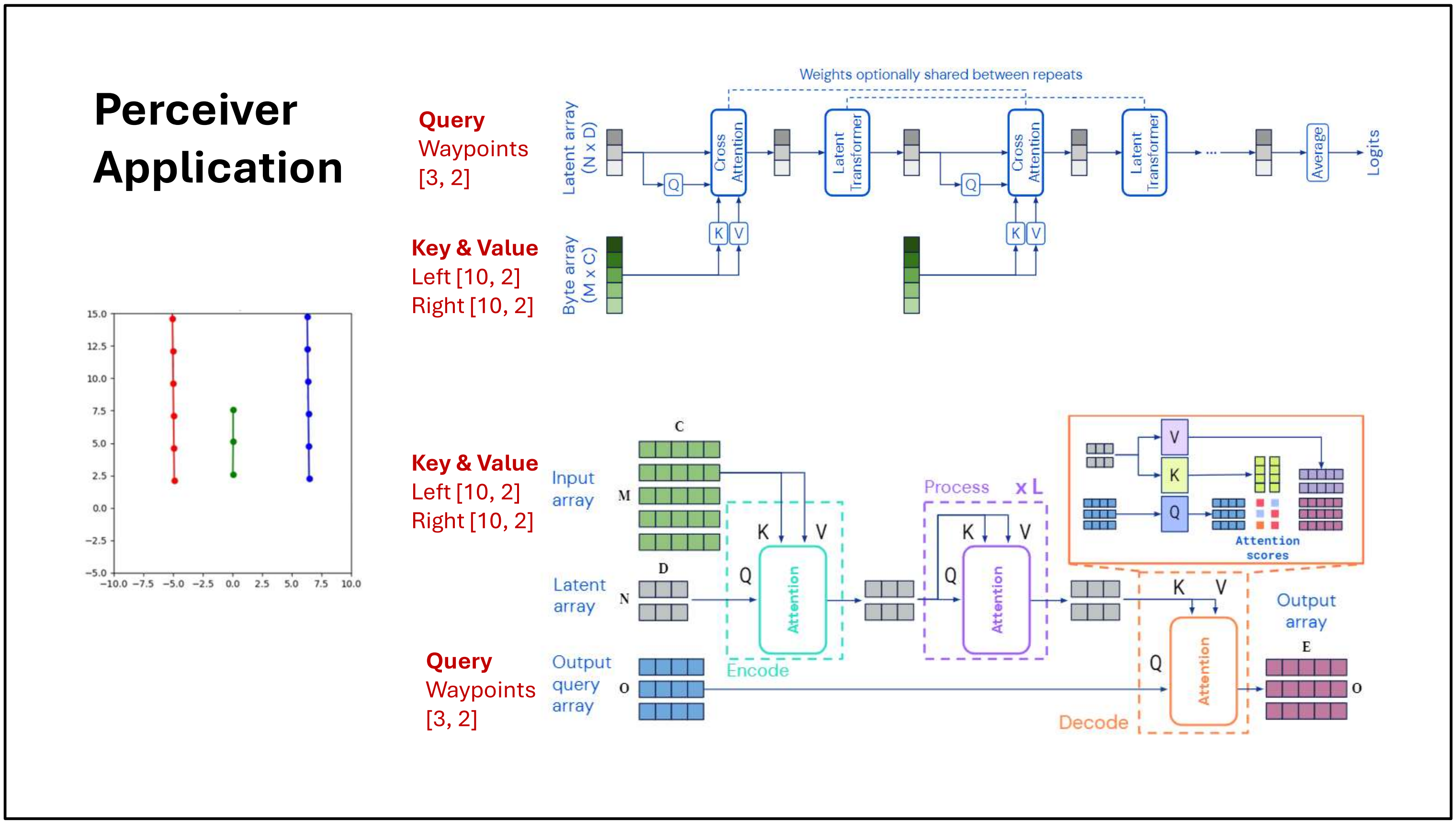
Feature Extraction by Self Attention

The original Perceiver is limited in its ability to handle varied output types. To address this limitation, Perceiver IO extends the Perceiver framework further.

Perceiver IO maintains the strengths of the original Perceiver, such as efficient dimensionality reduction through cross-attention and powerful feature extraction via self-attention. Importantly, Perceiver IO introduces a flexible query-based decoding mechanism, allowing the model to produce outputs of varying sizes and semantics, making it more general-purpose.

Looking at the diagram:

- Encoding (in green) still employs cross-attention to map high-dimensional inputs into a smaller latent space.
- Processing (in purple) uses self-attention in the latent space to perform feature extraction and interactions.
- The newly added Decoding step (in orange) leverages another cross-attention step, but this time between the latent array and a user-defined output query array. This flexible decoding allows Perceiver IO to handle arbitrary structured outputs effectively.



Here, we show how Perceiver IO can be applied to our driving dataset scenario.

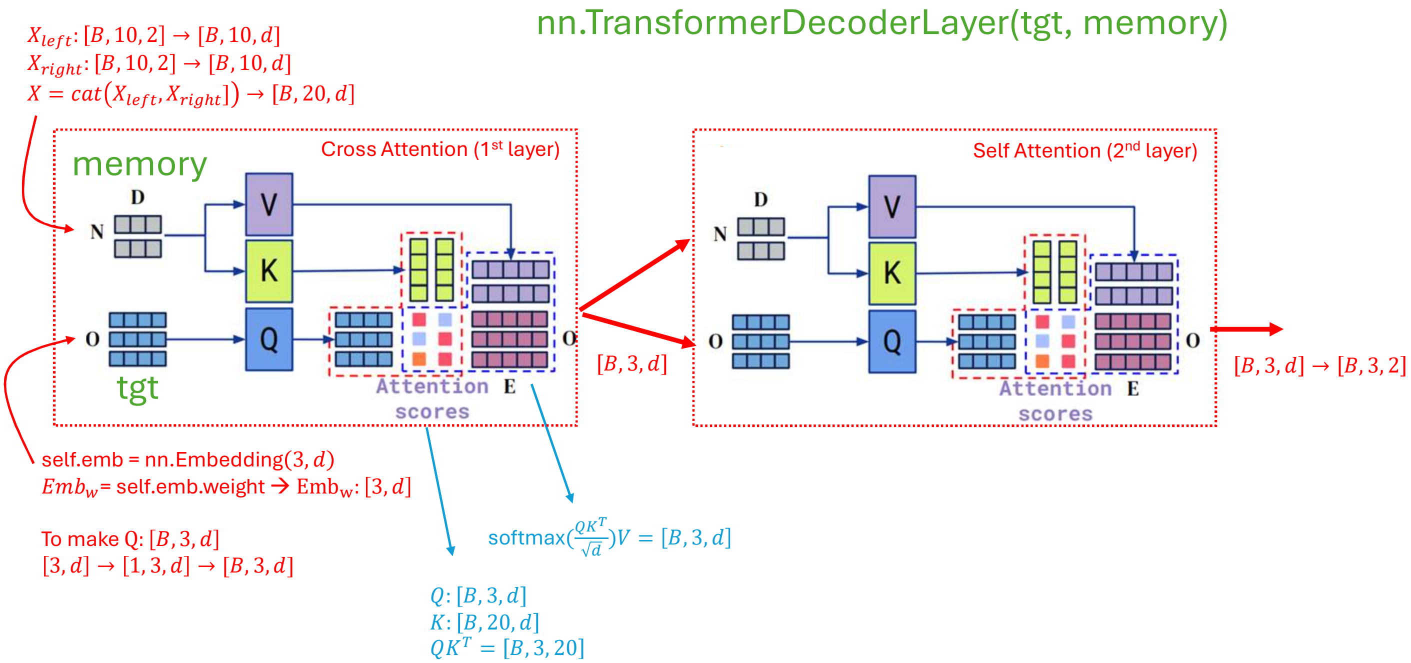
Recall our dataset consists of the left and right track boundary points (each shaped [10, 2]) shown in red and blue, respectively, and target waypoints (shown in green), which have shape [3, 2].

In the Perceiver IO framework, we can leverage these components as follows:

- Dimension Reduction (Encode): We use cross-attention to project our high-dimensional inputs (the left and right tracks) into a smaller latent array. This step reduces the dimensionality, ensuring computational efficiency even with complex, detailed input data.
- Feature Extraction (Process): We then apply self-attention within this latent array to capture rich features and interactions among embeddings. This step iteratively refines the latent representation.
- Flexible Output (Decode): Finally, we query our latent representation using the desired output waypoints. Here, another cross-attention operation occurs, using the waypoints as queries to extract structured outputs from our latent space.

This flexible querying approach allows the model to produce accurate predictions of waypoints from track information, showcasing how Perceiver IO effectively handles variable structured inputs and outputs simultaneously.

Transformer Planner



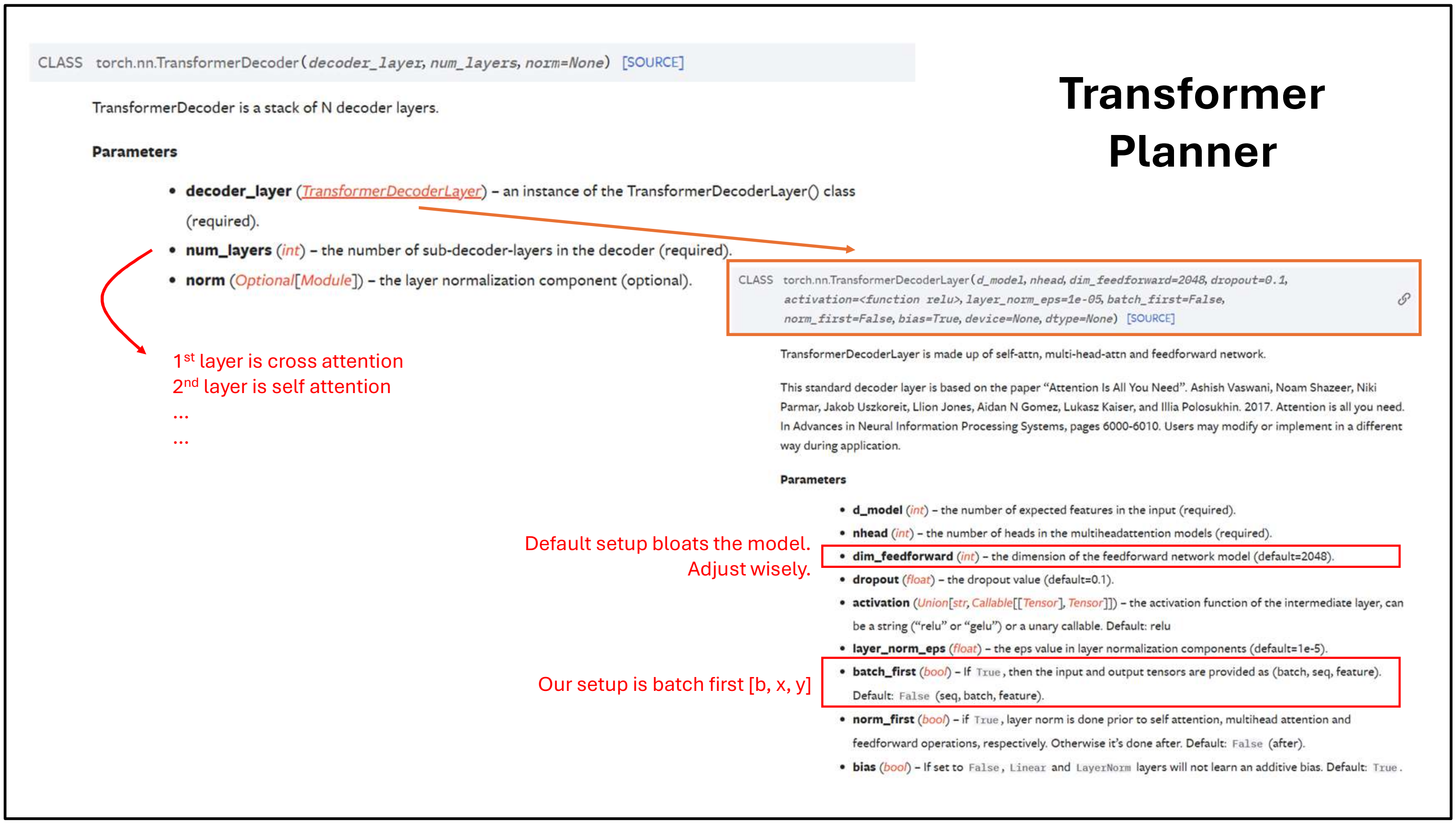
Let's walk through the implementation of our Transformer Planner, highlighting how the cross-attention mechanism from PyTorch's built-in nn.TransformerDecoderLayer aligns directly with our model's structure.

We start by preparing our inputs:
Track boundaries (X_{left} and X_{right}) initially have shapes [B, 10, 2]. We embed each track into a higher-dimensional space [B, 10, d], then concatenate them into a single input array (memory) of shape [B, 20, d].To predict exactly three waypoints, we create learnable query embeddings (tgt) using an embedding layer with shape [3, d], which expands across batches to [B, 3, d].

In the cross-attention step (left side), we feed our concatenated track embeddings as keys and values (memory), and waypoint embeddings as queries (tgt), into PyTorch's nn.TransformerDecoderLayer. The cross-attention computes attention scores, outputting [B, 3, d].

Then, in the subsequent self-attention step (right side), these [B, 3, d] embeddings further refine the relationships between the predicted waypoints themselves.

Finally, we project these refined embeddings to waypoint coordinates [B, 3, 2]. This illustrates how we utilize PyTorch's Transformer layers to implement a waypoint prediction model that leverages both cross-attention and self-attention mechanisms effectively.



Let's continue onto stitching the cross attention and the self attention layers.

The nn.TransformerDecoder class, which stacks multiple TransformerDecoderLayers. The first attention layer performs cross-attention between waypoints and tracks. The second attention layer is self-attention, refining the predictions by attending between the predicted waypoints themselves.

While setting up this decoder, it's important to pay attention to some default parameters:

- The default dimension for the feed-forward network (dim_feedforward) is 2048, which can quickly inflate model size. Adjust this parameter thoughtfully to keep your model efficient.
- Also, the default tensor shape setup is sequence-first ([seq, batch, feature]), but we use batch_first=True, making our tensors [batch, sequence, feature].

CNN Planner

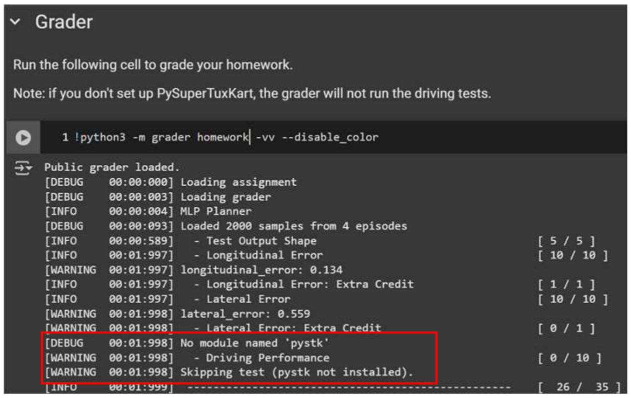
```
## Part 2: CNN Planner (30 points)

One major limitation of the previous models is that they require the ground truth lane boundaries as input.
In the previous homework, we trained a model to predict these in image space, but reprojecting the lane boundaries from image space to the vehicle's coordinate frame is non-trivial as small depth errors are magnified through the re-projection process.

Rather than going through segmentation and depth estimation, we can learn to predict the lane boundaries in the vehicle's coordinate frame directly from the image!

Implement the `CNNPlanner` model in `models.py`.

Your `forward` function receives a `(B, 3, 96, 128)` image tensor as input and should return a `(B, n_waypoints, 2)` tensor of predicted vehicle positions at the next `n_waypoints` time-steps.

The previous homeworks image backbones will be useful here, but you will need to modify the output layer to predict the desired waypoints.

Previously, we used CNNs + linear layers to predict tensors with shape
- `(B, num_classes)` for classification
- `(B, num_classes, H, W)` for segmentation
- `(B, 1, H, W)` for depth

But now we need to predict waypoints `(B, n_waypoints, 2)`.

One simple way to do this is simply produce a `(B, n_waypoints * 2)` tensor and reshape it to `(B, n_waypoints, 2)`.

For full credit, your model should achieve:
- < 0.30 Longitudinal error
- < 0.45 Lateral error
```

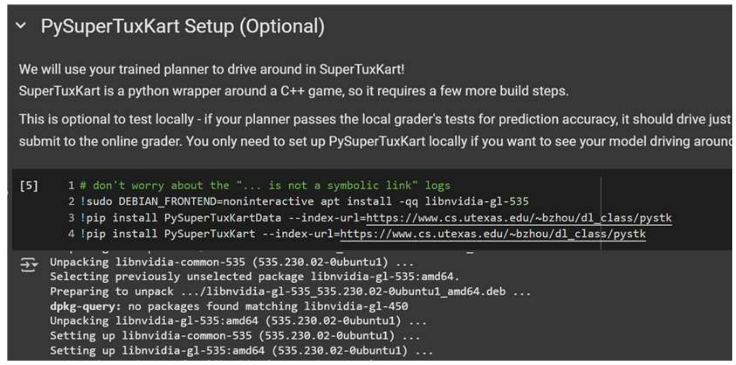
Recall that you are just given the track boundaries. The waypoints would be very different between reckless, DUI and gentle drivers. Without velocity information, you cannot tell what kind of driver generated the waypoint labels. That's why the lateral error is larger.

However, with the raw images, your model can assess the speed required to stay on the track. Do you see any off-track images in the dataset? I think our test driver was well behaved. With that, we need your CNN model to achieve a better lateral error < 0.45.

PySuperTuxKart Kit

Install PySuperTuxKart to fix “No module named pystk” error





By the way, the grader needs to simulate the track run in order to assess if your model can drive at least 50% of the track. You need to install pystk.