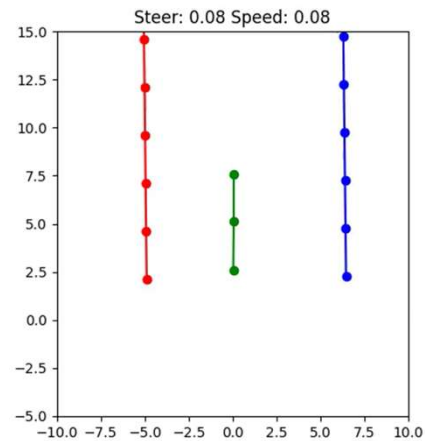


# Dataset

- 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

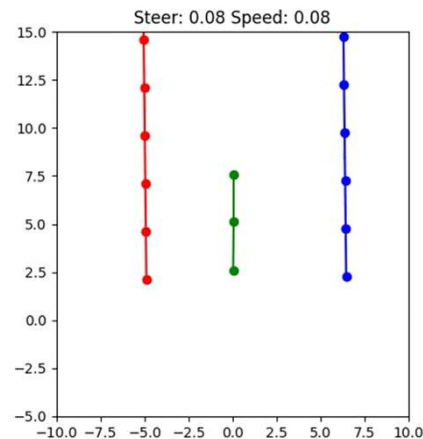
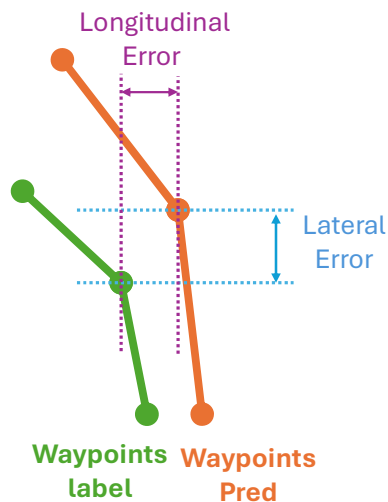
- Not all waypoint labels are valid
- Refer to `PlannerMetric()` in `metrics.py`
- Implement the same in loss function

```
5 class PlannerMetric:
6     """
7     Computes longitudinal and lateral errors for a planner
8     """
9
10    def __init__(self):
11        self.l1_errors = []
12        self.total = 0
13
14    def reset(self):
15        self.l1_errors = []
16        self.total = 0
17
18    @torch.no_grad()
19    def add(
20        self,
21        preds: torch.Tensor,
22        labels: torch.Tensor,
23        labels_mask: torch.Tensor,
24    ):
25        """
26        Args:
27            preds (torch.Tensor): (b, n, 2) float tensor with predicted waypoints
28            labels (torch.Tensor): (b, n, 2) ground truth waypoints
29            labels_mask (torch.Tensor): (b, n) bool mask for valid waypoints
30        """
31        error = (preds - labels).abs()
32        error_masked = error * labels_mask[..., None]
33
34        # sum across batch and waypoints
35        error_sum = error_masked.sum(dim=(0, 1)).cpu().numpy()
36
37        self.l1_errors.append(error_sum)
38        self.total += labels_mask.sum().item()
39
```

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.

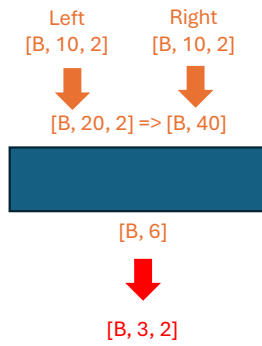
It's essential to incorporate the same masking approach into the loss function during training to ensure consistency between evaluation and optimization.

# 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



```
11 class MLPPlanner(nn.Module):
12     def __init__(
13         self,
14         n_track: int = 10,
15         n_waypoints: int = 3,
16     ):
17         """
18         Args:
19             n_track (int): number of points in each side of the track
20             n_waypoints (int): number of waypoints to predict
21         """
22         super().__init__()
23
24         self.n_track = n_track
25         self.n_waypoints = n_waypoints
26
27     def forward(
28         self,
29         track_left: torch.Tensor,
30         track_right: torch.Tensor,
31         **kwargs,
32     ) -> torch.Tensor:
33         """
34         Predicts waypoints from the left and right boundaries of the track.
35
36         During test time, your model will be called with
37         model(track_left=..., track_right=...), so keep the function signature as is.
38
39         Args:
40             track_left (torch.Tensor): shape (b, n_track, 2)
41             track_right (torch.Tensor): shape (b, n_track, 2)
42
43         Returns:
44             torch.Tensor: future waypoints with shape (b, n_waypoints, 2)
45         """
46         raise NotImplementedError
47
```

# Transformer Planner

```
125  ### Part 1b: Transformer Planner (35 points)
126
127  We'll build a similar model to Part 1a, but this time we'll use a Transformer.
128
129  Compared to the MLP model, there are many more ways to design this model!
130  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.
131  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.
132
133  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.
134
135  
136
137  Training the transformer will likely require more tuning, so make sure to optimize your training pipeline to allow for faster
    experimentation.
138
139  For full credit, your model should achieve:
140  - < 0.2 Longitudinal error
141  - < 0.6 Lateral error
142
143  ### Relevant Operations
144  - [torch.nn.Embedding](https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html)
145  - [torch.nn.TransformerDecoderLayer](https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoderLayer.html)
146  - [torch.nn.TransformerDecoder](https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html)
147
```

# Attention Is All You Need

Ashish Vaswani\*  
Google Brain  
avaswani@google.com

Noam Shazeer\*  
Google Brain  
noam@google.com

Niki Parmar\*  
Google Research  
nikip@google.com

Jakob Uszkorei  
Google Research  
usz@google.com

Llion Jones\*  
Google Research  
llion@google.com

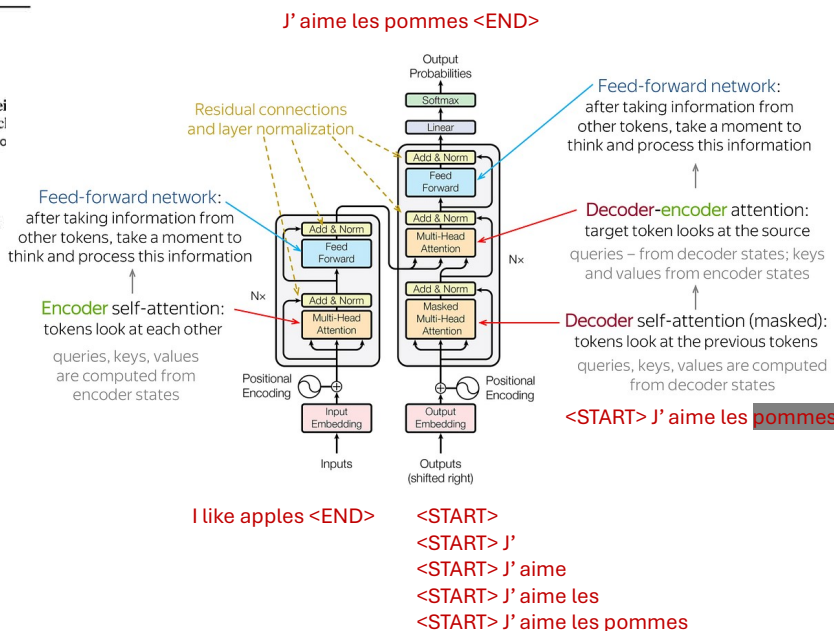
Aidan N. Gomez\*  
University of Toronto  
aidan@cs.toronto.edu

Lukasz Kaiser\*  
Google Brain  
lukaszkaier@google.com

Illia Polosukhin\*  
illia.polosukhin@gmail.com

## Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.



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

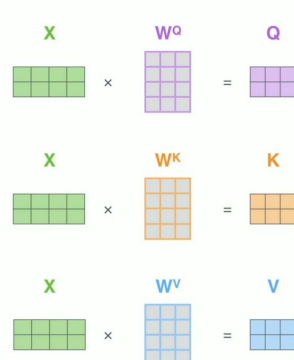
$$\text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

scores

Normalize  
control softmax scale

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

## Self-Attention (Alammar)



Alammar, *The Illustrated Transformer*

sent len x sent len (attn for each word to each other)

$\text{softmax} \left( \frac{Q \times K^T}{\sqrt{d_k}} \right)$

$Z$

sent len x hidden dim

$Z$  is a weighted combination of  $V$  rows

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.

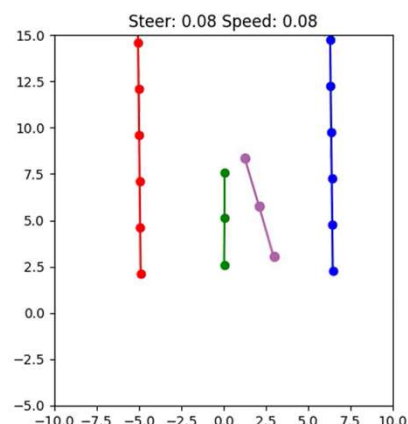
# Transformer Intuition

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.

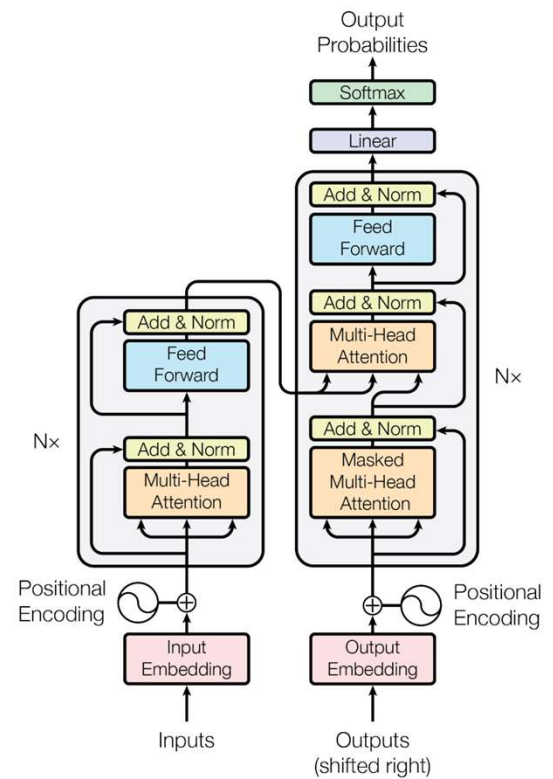


Figure 1: The Transformer - model architecture.

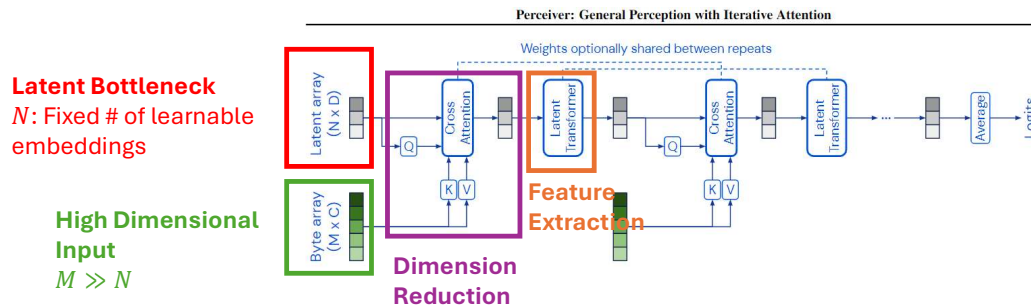
Traditional transformers, despite their powerful performance, face challenges when scaling to larger inputs. Their attention mechanism results in quadratic complexity  $O(n^2)$ , making them particularly inefficient when processing long sequences or high-dimensional data like images, videos, or audio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



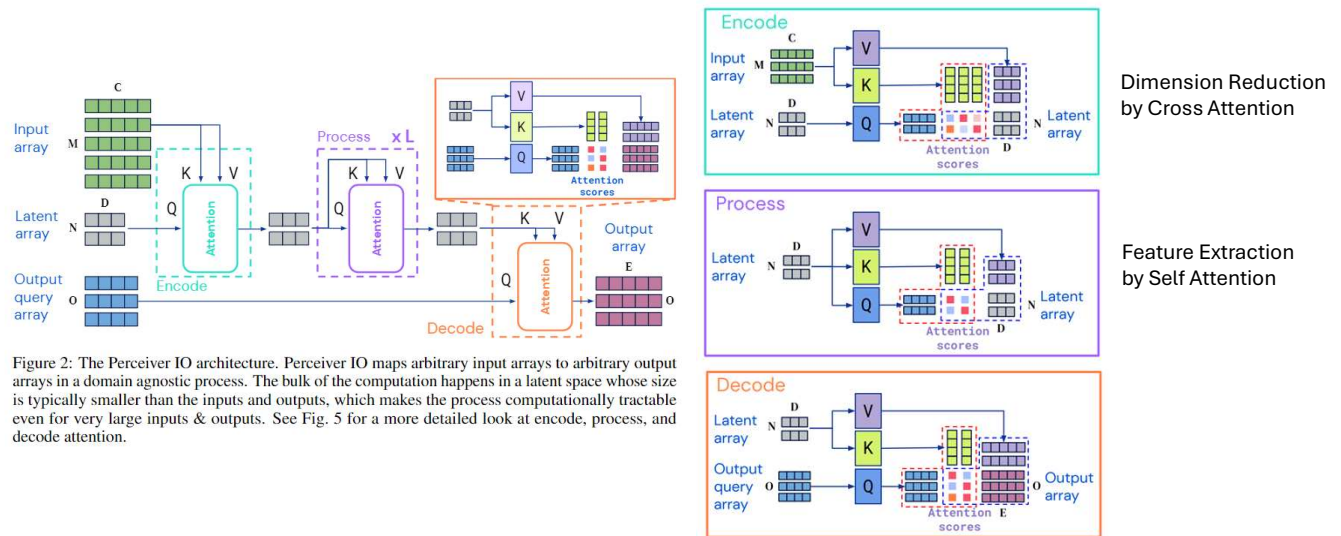
To address the scaling limitations of Transformers we discussed previously, let's explore the Perceiver model, introduced by Andrew Jaegle and colleagues.

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.



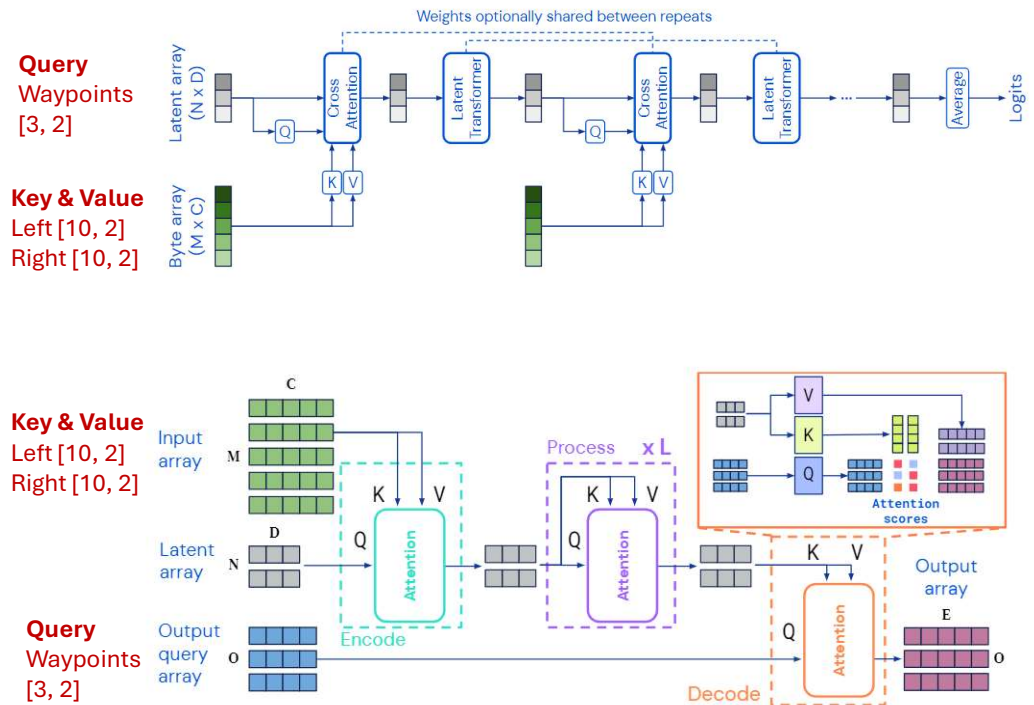
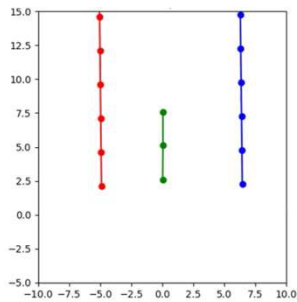
The original Perceiver is limited in its ability to handle varied output types. To address this limitation, Perceiver IO extends the Perceiver framework further.

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.

# Perceiver Application



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:

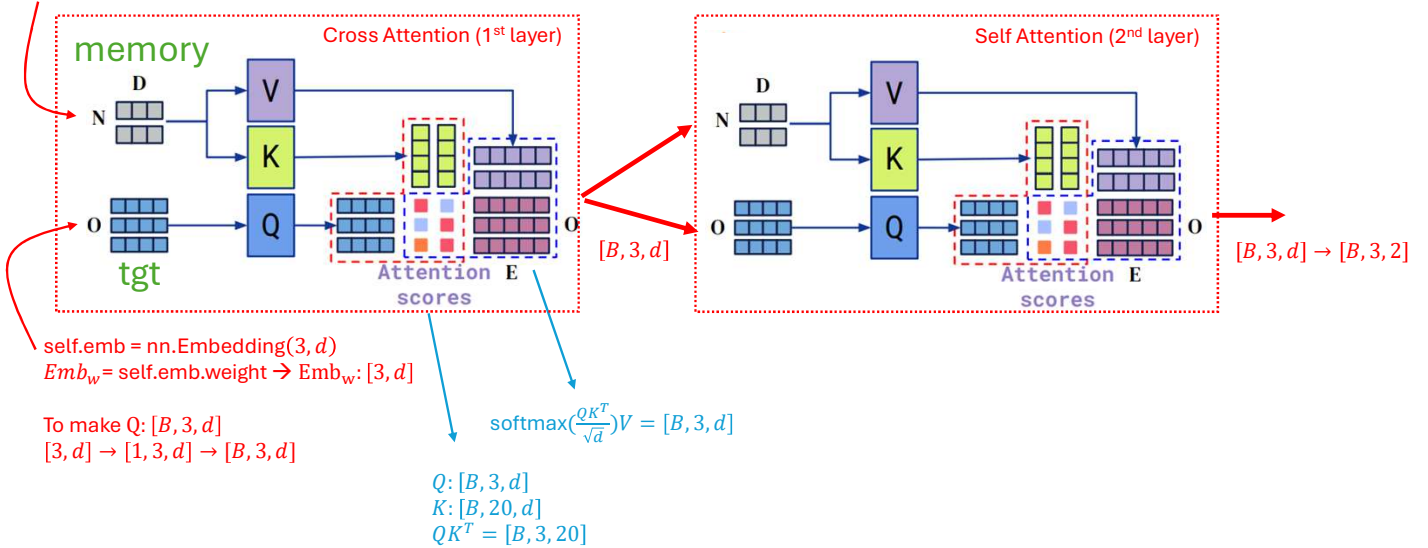
- **Dimension Reduction (Encode):** We use cross-attention to project our high-dimensional inputs (the left and right tracks) into a smaller latent array. This step reduces the dimensionality, ensuring computational efficiency even with complex, detailed input data.
- **Feature Extraction (Process):** 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

`nn.TransformerDecoderLayer(tgt, memory)`

$X_{left}: [B, 10, 2] \rightarrow [B, 10, d]$   
 $X_{right}: [B, 10, 2] \rightarrow [B, 10, d]$   
 $X = \text{cat}(X_{left}, X_{right}) \rightarrow [B, 20, d]$



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.

# Transformer Planner

CLASS `torch.nn.TransformerDecoder(decoder_layer, num_layers, norm=None)` [SOURCE]

TransformerDecoder is a stack of N decoder layers.

## Parameters

- **decoder\_layer** (`TransformerDecoderLayer`) – an instance of the `TransformerDecoderLayer()` class (required).
- **num\_layers** (`int`) – the number of sub-decoder-layers in the decoder (required).
- **norm** (`Optional[Module]`) – the layer normalization component (optional).

1<sup>st</sup> layer is cross attention  
2<sup>nd</sup> layer is self attention

...

CLASS `torch.nn.TransformerDecoderLayer(d_model, nhead, dim_feedforward=2048, dropout=0.1, activation=<function relu>, layer_norm_eps=1e-05, batch_first=False, norm_first=False, bias=True, device=None, dtype=None)` [SOURCE]

TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network.

This standard decoder layer is based on the paper “Attention Is All You Need”. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010. Users may modify or implement in a different way during application.

## Parameters

- **d\_model** (`int`) – the number of expected features in the input (required).
- **nhead** (`int`) – the number of heads in the multiheadattention models (required).
- **dim\_feedforward** (`int`) – the dimension of the feedforward network model (default=2048).
- **dropout** (`float`) – the dropout value (default=0.1).
- **activation** (`Union[str, Callable[[Tensor], Tensor]]`) – the activation function of the intermediate layer, can be a string (“relu” or “gelu”) or a unary callable. Default: relu
- **layer\_norm\_eps** (`float`) – the eps value in layer normalization components (default=1e-5).
- **batch\_first** (`bool`) – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False` (seq, batch, feature).
- **norm\_first** (`bool`) – If `True`, layer norm is done prior to self attention, multihead attention and feedforward operations, respectively. Otherwise it's done after. Default: `False` (after).
- **bias** (`bool`) – If set to `False`, `Linear` and `LayerNorm` layers will not learn an additive bias. Default: `True`.

Default setup bloats the model.  
Adjust wisely.

Our setup is batch first [b, x, y]

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

```
148 ## Part 2: CNN Planner (30 points)
149
150 One major limitation of the previous models is that they require the ground truth lane boundaries as input.
151 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.
152
153 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!
154
155 Implement the 'CNNPlanner' model in 'models.py'.
156
157 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.
158
159 The previous homeworks image backbones will be useful here, but you will need to modify the output layer to predict the desired
    waypoints.
160
161 Previously, we used CNNs + linear layers to predict tensors with shape
162 - '(B, num_classes)' for classification
163 - '(B, num_classes, H, W)' for segmentation
164 - '(B, 1, H, W)' for depth
165
166 But now we need to predict waypoints '(B, n_waypoints, 2)'.
167
168 One simple way to do this is simply produce a '(B, n_waypoints * 2)' tensor and reshape it to '(B, n_waypoints, 2)'.
169
170 For full credit, your model should achieve:
171 - < 0.30 Longitudinal error
172 - < 0.45 Lateral error
173
```

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

Grader

Run the following cell to grade your homework.  
Note: if you don't set up PySuperTuxKart, the grader will not run the driving tests.

```
1 |python3 -m grader homework| -vv --disable_color
```

```
Public grader loaded.
[DEBUG 00:00:00] Loading assignment
[DEBUG 00:00:00] Loading grader
[INFO 00:00:00] MLP Planner
[DEBUG 00:00:00] Loaded 2000 samples from 4 episodes
[INFO 00:00:58] - Test Output Shape [ 5 / 5 ]
[INFO 00:01:57] - Longitudinal Error [ 10 / 10 ]
[WARNING 00:01:57] longitudinal_error: 0.134
[INFO 00:01:57] - Longitudinal Error: Extra Credit [ 1 / 1 ]
[INFO 00:01:57] - Lateral Error [ 10 / 10 ]
[WARNING 00:01:58] lateral_error: 0.559
[WARNING 00:01:58] - Lateral Error: Extra Credit [ 0 / 1 ]
[DEBUG 00:01:58] No module named 'pystk'
[WARNING 00:01:58] - Driving Performance [ 0 / 10 ]
[WARNING 00:01:58] Skipping test (pystk not installed).
[INFO 00:01:59] ----- [ 26 / 35 ]
```

PySuperTuxKart Setup (Optional)

We will use your trained planner to drive around in SuperTuxKart!  
SuperTuxKart is a python wrapper around a C++ game, so it requires a few more build steps.  
This is optional to test locally - if your planner passes the local grader's tests for prediction accuracy, it should drive just submit to the online grader. You only need to set up PySuperTuxKart locally if you want to see your model driving around

```
[5] 1 # don't worry about the "... is not a symbolic link" logs
    2 !sudo DEBIAN_FRONTEND=noninteractive apt install -qq libnvidia-gl-535
    3 !pip install PySuperTuxKartData --index-url=https://www.cs.utexas.edu/~bzhou/dl_class/pystk
    4 !pip install PySuperTuxKart --index-url=https://www.cs.utexas.edu/~bzhou/dl_class/pystk
```

```
Unpacking libnvidia-common-535 (535.230.02-0ubuntu1) ...
Selecting previously unselected package libnvidia-gl-535:amd64.
Preparing to unpack .../libnvidia-gl-535_535.230.02-0ubuntu1_amd64.deb ...
dpkg-query: no packages found matching libnvidia-gl-450
Unpacking libnvidia-gl-535:amd64 (535.230.02-0ubuntu1) ...
Setting up libnvidia-common-535 (535.230.02-0ubuntu1) ...
Setting up libnvidia-gl-535:amd64 (535.230.02-0ubuntu1) ...
```

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.