ECOLE
POLYTECHNIQUE
DE BRUXELLES

Building a foundation model for trajectory from scratch

Presented by Mahmoud Sakr, Gilles Dejaegere, and Gaspard Merten

Objectives of the tutorial:

- Introduce the main vocabulary/concepts of FMs/LLMs
- Adapt existing LLM architecture to trajectory FM
- Provide a mock-up FM for trajectory to the community

Structure of the tutorial

1. Deconstructing GPT-2
2. Adapting GPT-2 to a trajectory FM
3. State of the art trajectory FMs

Modalities

- All code available in notebooks
- Can be run locally or in Google-Colab

Resources and notebooks:

Part 1
Deconstructing GPT-2

Part 1 - Resources

Code & Visualizations : Raschka, Sebastian. *Build A Large Language Model (From Scratch)*. Manning, 2024. ISBN: 978-1633437166.

Visualizations : Grant Sanderson, https://www.3blue1brown.com.

What Is GPT-2

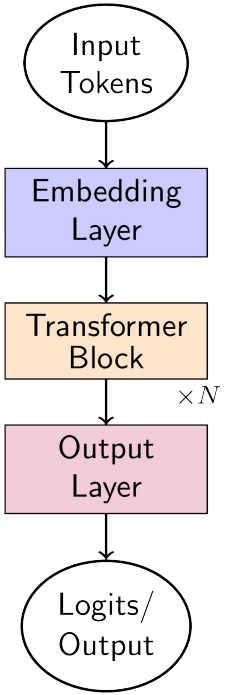
- **Generative**: it creates text
- **Pre-trained**: trained on large amount of text data
- **Transformer**: neural network architecture

Pre-training task: autoregressive next token generator:

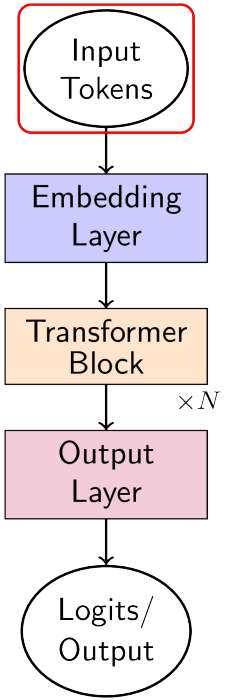
The cat that chased →the

The cat that chased the→mouse

GPT-2 High Level Architecture



Creating Input Tokens



1. **Text is split into tokens:**

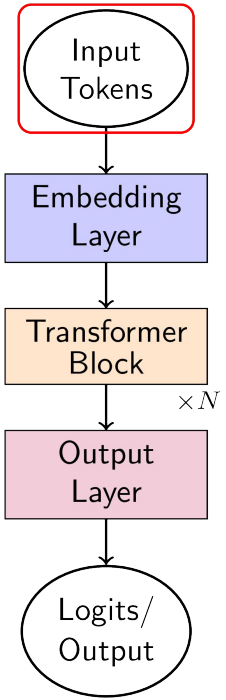
   the blue cat chased the green mouse ← Context

2. **Tokens are mapped to token-IDs**

   the → [262]
   blue → [4171]
   cat → [3797]
   chased → [26172]
   the → [262]
   …

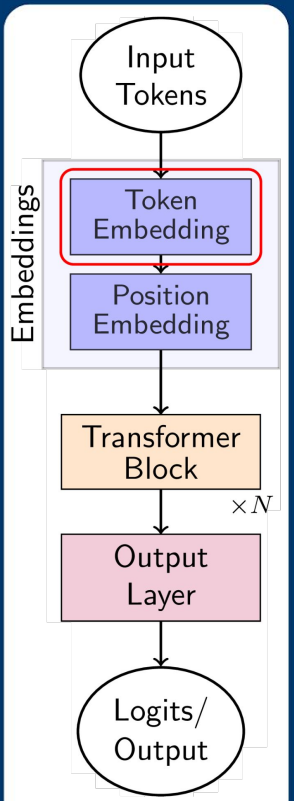
BytePaire Encoding: Unexpectedly, globalization transformed industries rapidly

Creating Input Tokens



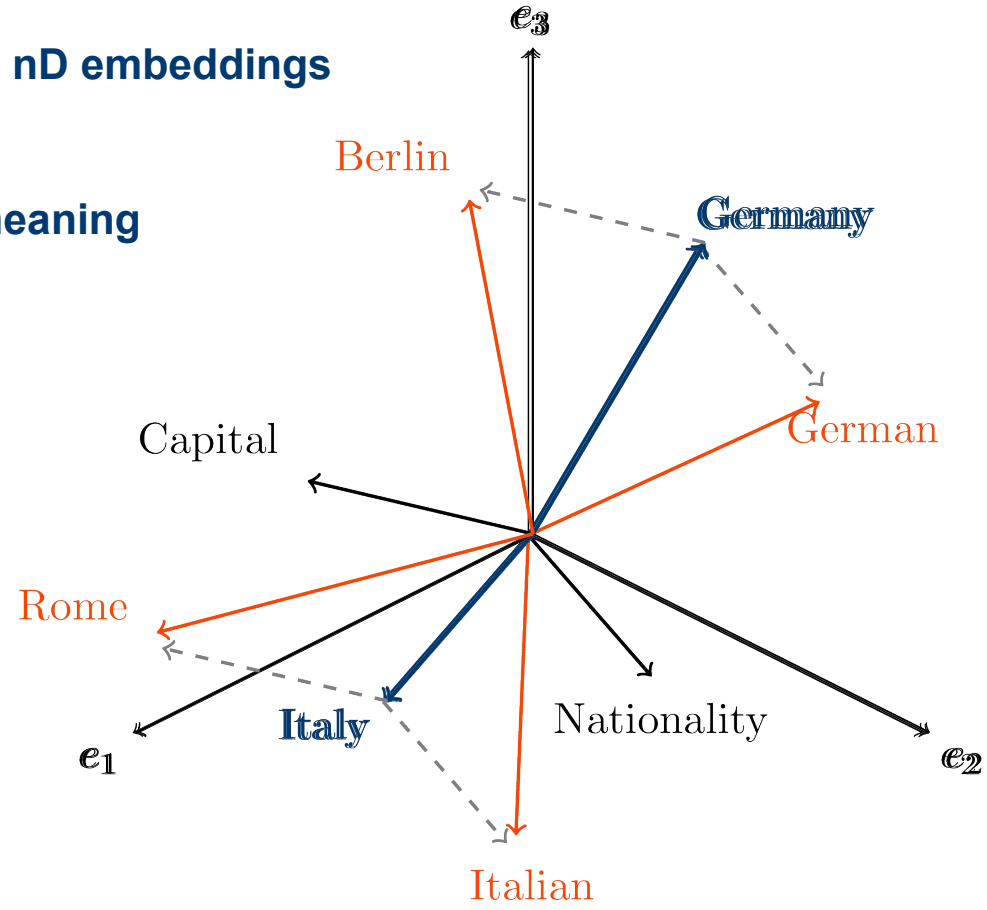

```python
import tiktoken


tokenizer = tiktoken.get_encoding("gpt2")

text = " the blue cat chased the"
token_ids = tokenizer.encode(text)
# [262, 4171, 3797, 26172, 262]

text = "Unexpectedly, globalization transformed industries rapidly"
token_ids = tokenizer.encode(text2)
# [52, 42072, 306, 11, 39155, 14434, 11798, 8902]
```

Embedding Space/Latent Space

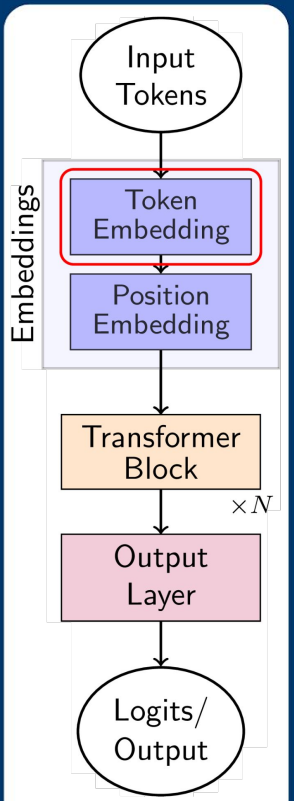


1D TokenIDs ➔ nD embeddings

Directions ➔ meaning



Token Embeddings



TokenIDs

the [262] | blue [4171] | cat [3797] | chased [26172] | the [262]

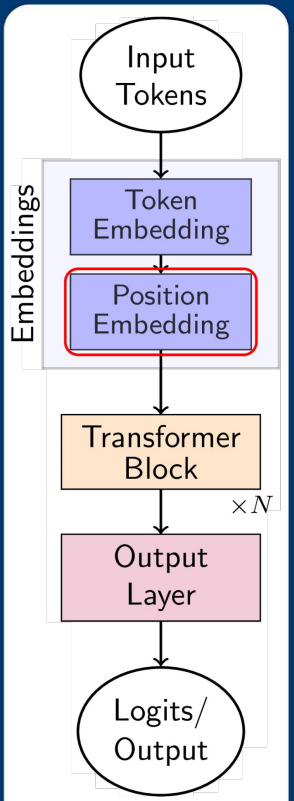
Embedding Matrix

$$emb\_d = 768$$

! Random until training

$$\begin{array}{c|cccccc} & c_0 & c_{262} & c_{3797} & c_{4171} & c_{26172} & c_{50,256} \\ \hline & 0.49 & 0.12 & 0.56 & 0.87 & 0.33 & 0.17 \\ & 0.73 & 0.44 & 0.91 & 0.15 & 0.72 & 0.92 \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & 0.26 & 0.35 & 0.63 & 0.40 & 0.20 & 0.85 \end{array}$$

$$\underbrace{\vec{E}_0/\vec{E}_4 \quad \vec{E}_2 \quad \vec{E}_1 \quad \vec{E}_3}_{vocab\_size = 50{,}257}$$



Token Embeddings

the [262] blue [4171] cat [3797] chased [26172] the [464]

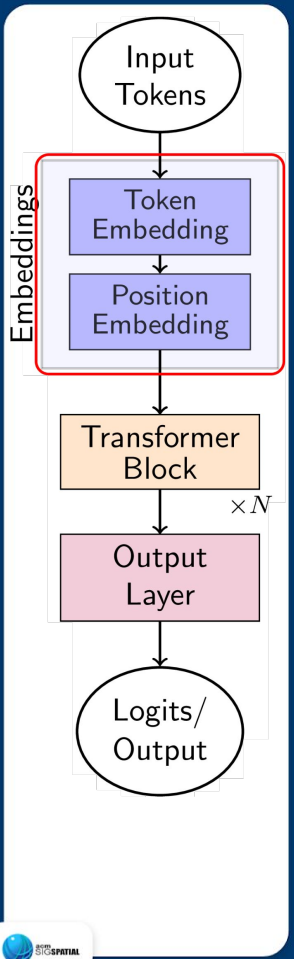
$$emb\_d = 768 \quad \underbrace{\begin{matrix} \vec{E}_{t0} & \vec{E}_{t1} & \vec{E}_{t2} & \vec{E}_{t3} & \vec{E}_{t4} & \\ 0.12 & 0.87 & 0.56 & 0.33 & 0.12 & \cdots \\ 0.44 & 0.15 & 0.91 & 0.72 & 0.44 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0.35 & 0.40 & 0.63 & 0.20 & 0.35 & \cdots \end{matrix}}_{context\_size = 1024}$$

Position Embeddings

$$\vec{E}_0 = \vec{E}_{t0} + \vec{E}_{p0}$$

$$\begin{bmatrix} 0.94 \\ 0.95 \\ \vdots \\ 0.47 \end{bmatrix}$$

Positional Embedding

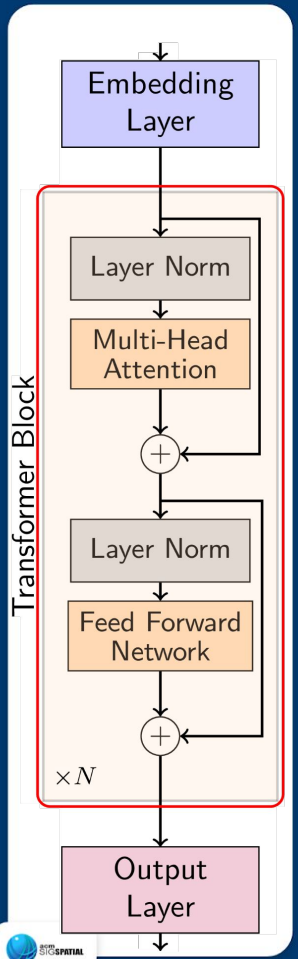


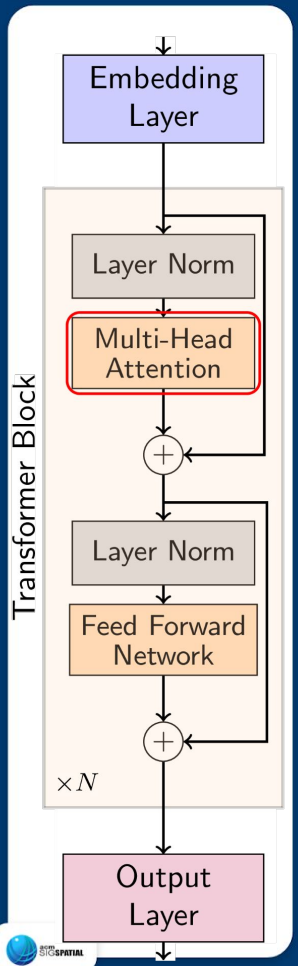

```python
import torch
import torch.nn as nn

class EmbeddingLayer(nn.Module):
  def __init__(self, d_emb=768, context_size=1024):
    super().__init__()
    self.vocab_size = 50257

    self.token_emb_layer = torch.nn.Embedding(self.vocab_size, d_emb)
    self.pos_emb_layer = torch.nn.Embedding(context_size, d_emb)

  def forward (self, input_ids):
    num_tokens = input_ids.shape[0]
    token_embeddings = self.token_emb_layer(input_ids)
    pos_embeddings = self.pos_emb_layer(torch.arange(num_tokens))
    return token_embeddings + pos_embeddings

embedding_layer = EmbeddingLayer(context_size=len(token_ids))
res = embedding_layer(torch.tensor(token_ids))
```

← Design Choices

← Random until training

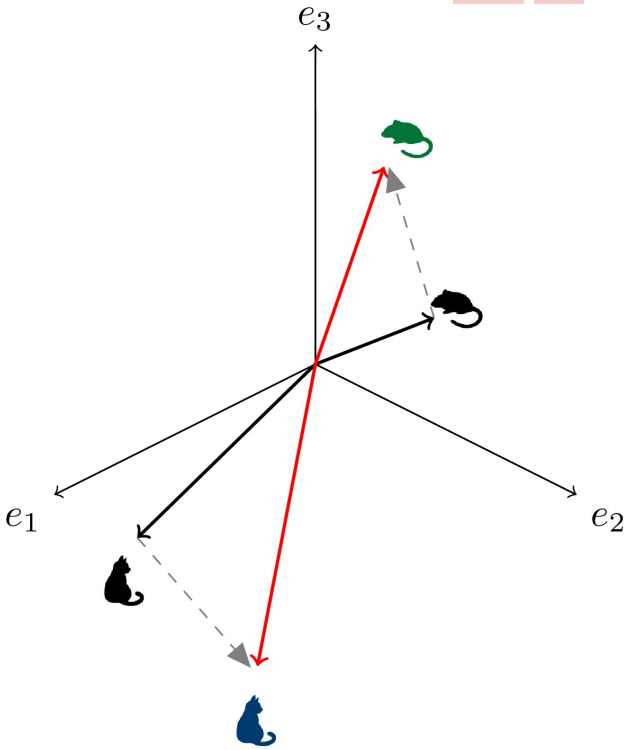
Transformer - Attention





Capturing relationship between tokens

the blue cat chased the green mouse

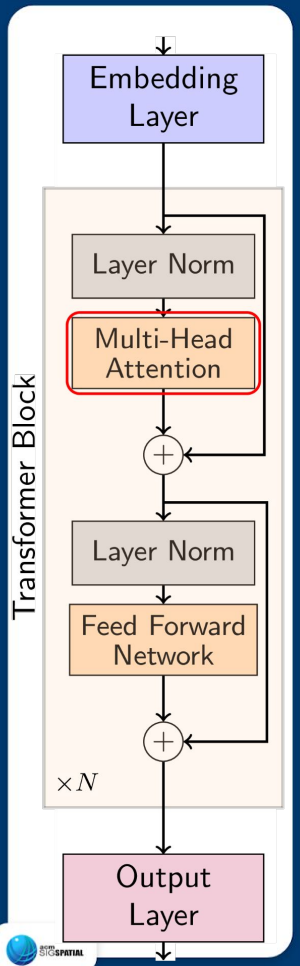


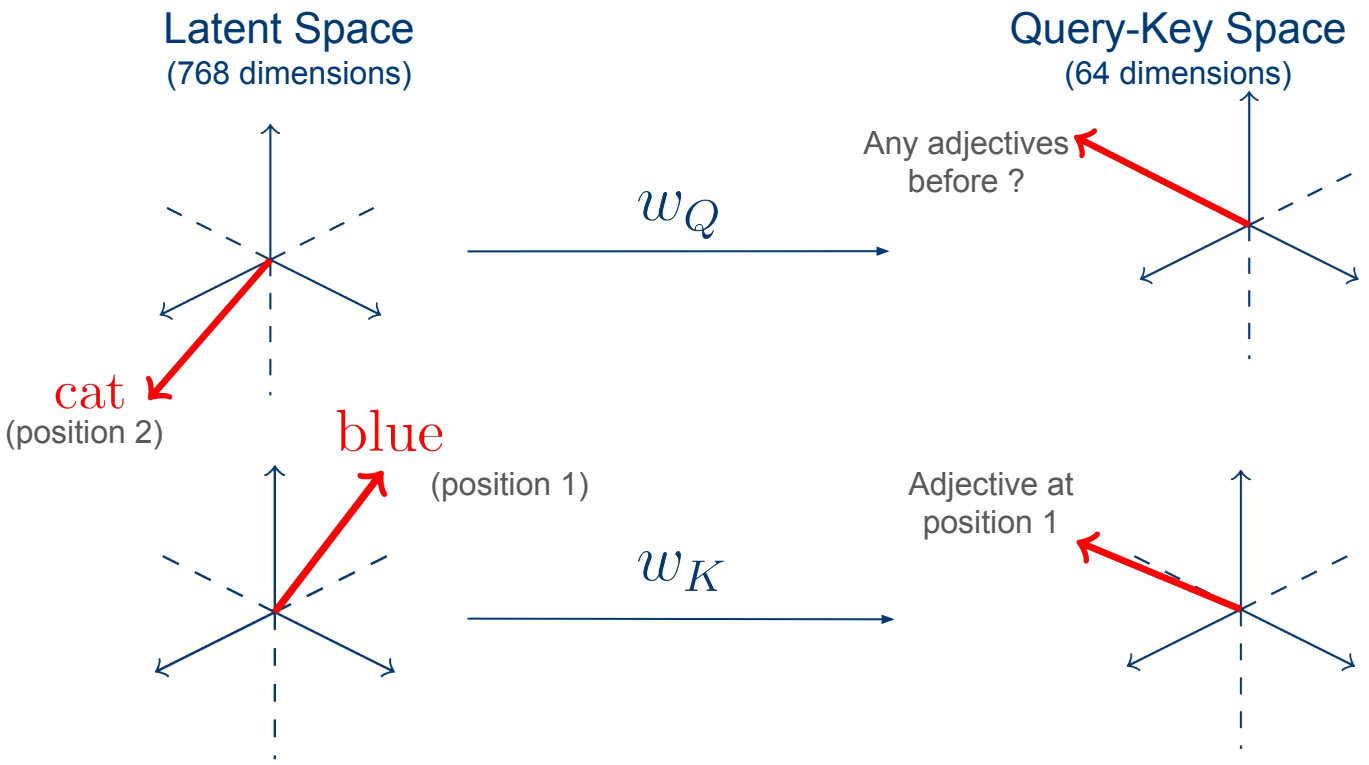
1. Which tokens should pay attention to each other?

2. How should a token update its embedding using the others?

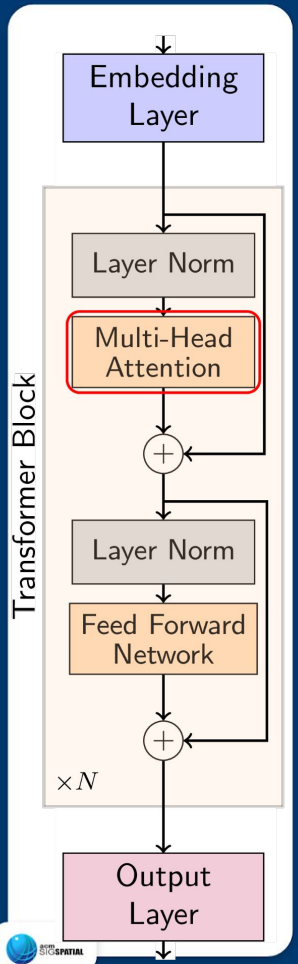
the cat chased the mouse

Attention - 1. Which tokens should pay attention to each other ?





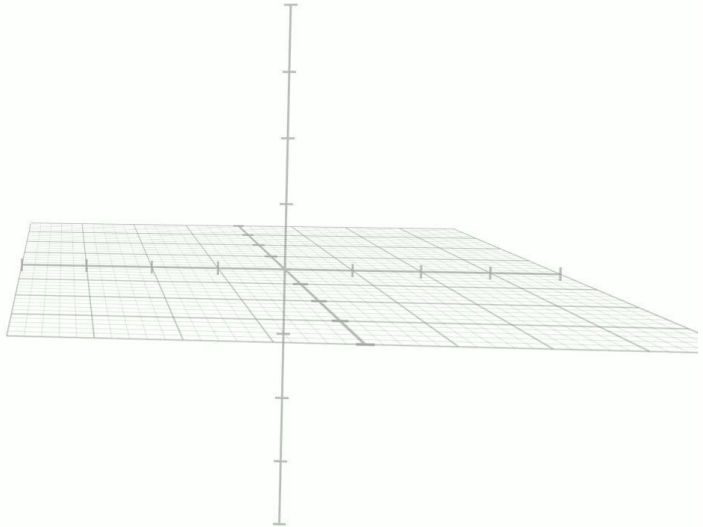
Attention - 2. How should a token update its embedding using the others?

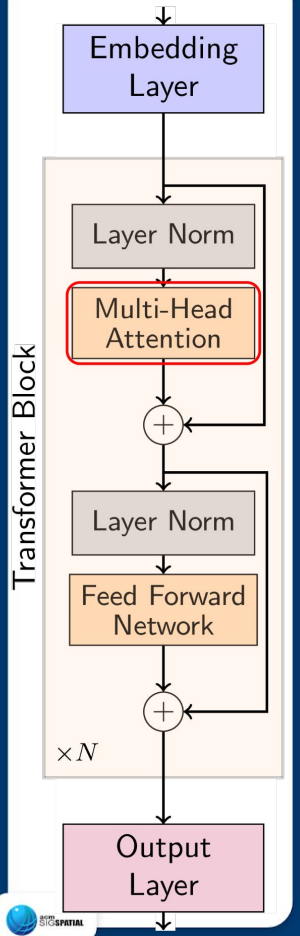


blue cat

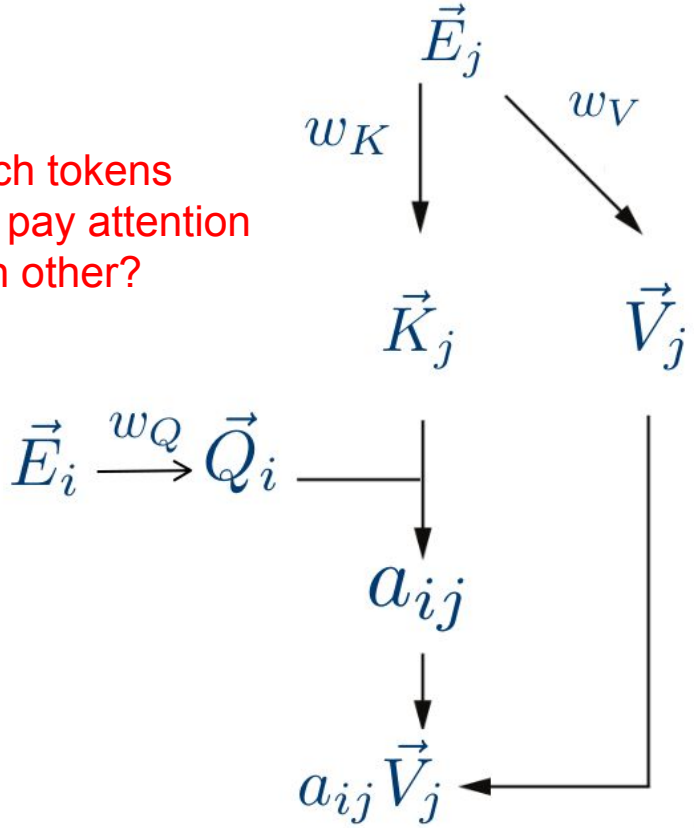
$\vec{E}_b$ $\vec{E}_c$

+9.2	-9.5
-2.3	+6.6
+5.8	+5.5
+0.6	+7.3
+1.3	+9.5
+8.4	+5.9
⋮	⋮
-8.2	+5.6

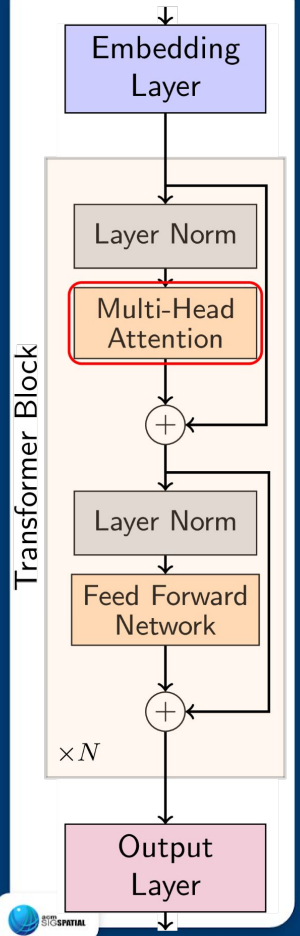




1. Which tokens should pay attention to each other?



2. How should a token update its embedding using the others?

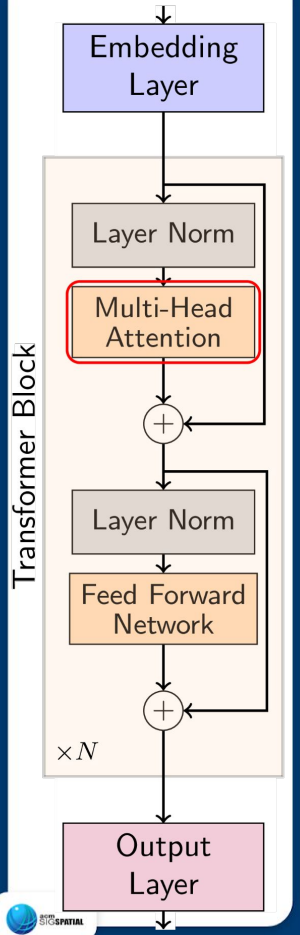


the blue cat chased the green mouse

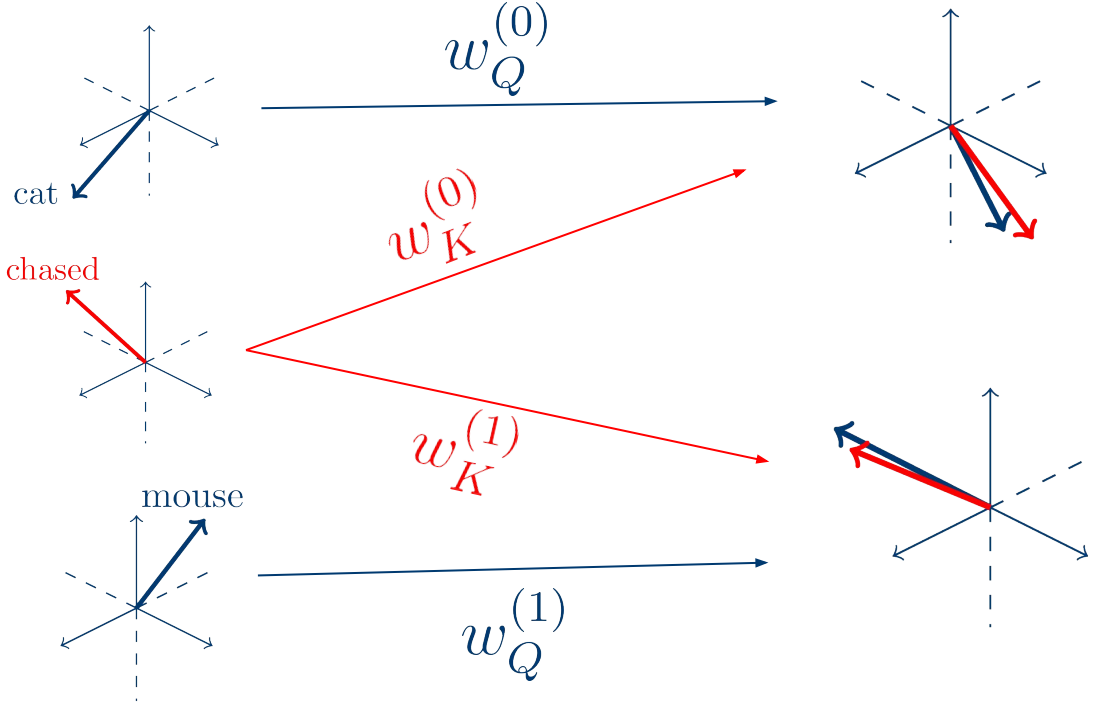
Rel 1: blue → cat (noun-adjective)
Rel 2: chased → the mouse (action-object)
Rel 3: cat → chased (subject-action)

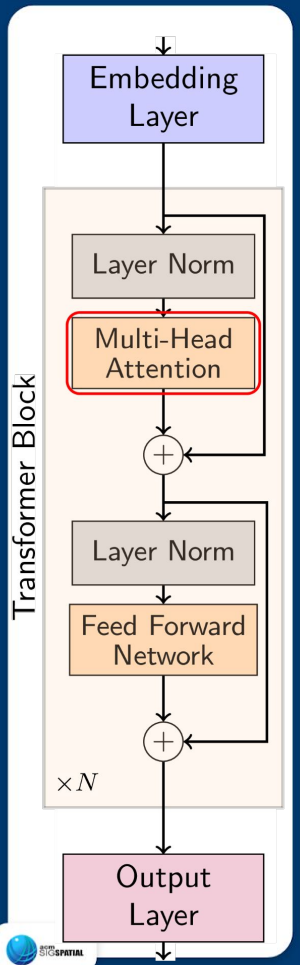
➔ One big head can capture different relationships

➔ But … several small heads are more efficient!



the blue cat chased the green mouse





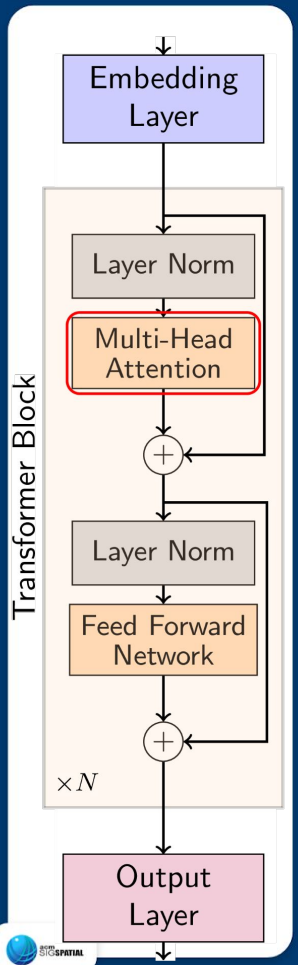
➔ Embeddings vectors are passing through multiple heads in parallel

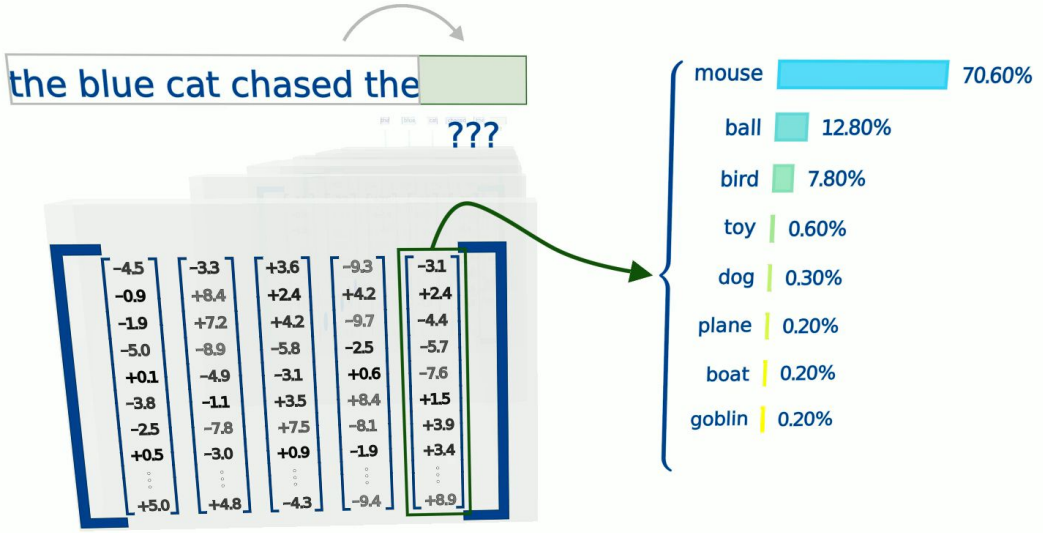
$$\vec{E}_i$$

$$W_Q^{(1)}, W_K^{(1)}, W_V^{(1)} \quad W_Q^{(2)}, W_K^{(2)}, W_V^{(2)} \quad \cdots \quad W_Q^{(h)}, W_K^{(h)}, W_V^{(h)}$$

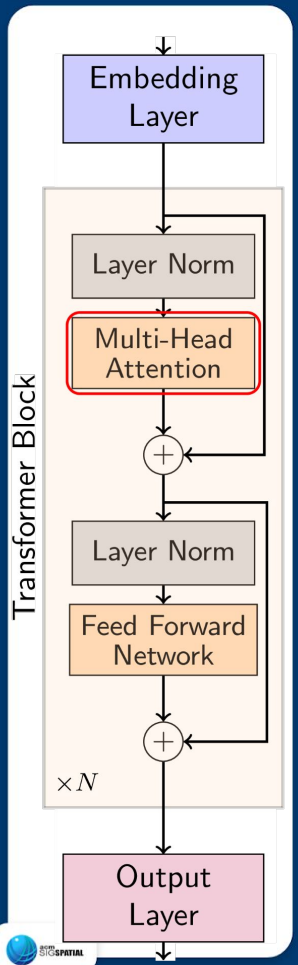
$$\vec{E}_i' = \vec{E}_i + \text{Concat}(\Delta\vec{E}_i^{(1)}, \Delta\vec{E}_i^{(2)}, \ldots, \Delta\vec{E}_i^{(h)})$$

$$\begin{bmatrix} | \\ \vec{E}_i' \\ | \end{bmatrix} = \begin{bmatrix} | \\ \vec{E}_i \\ | \end{bmatrix} + \begin{bmatrix} | \\ \Delta\vec{E}_i^{(1)} \\ \Delta\vec{E}_i^{(2)} \\ \Delta\vec{E}_i^{(3)} \\ \Delta\vec{E}_i^{(4)} \\ \vdots \end{bmatrix} + \begin{bmatrix} | \\ \Delta\vec{E}_i^{(2)} \\ | \end{bmatrix} \cdots$$

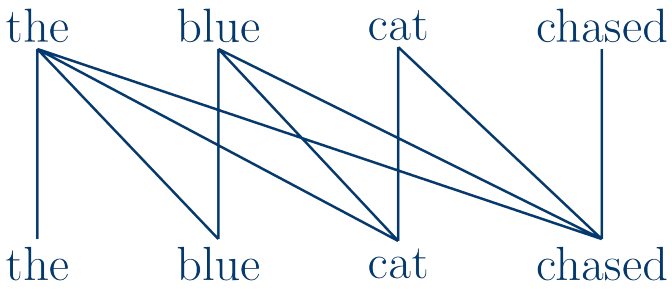




A token can not update a previous one ➔ this would be leaking the answer

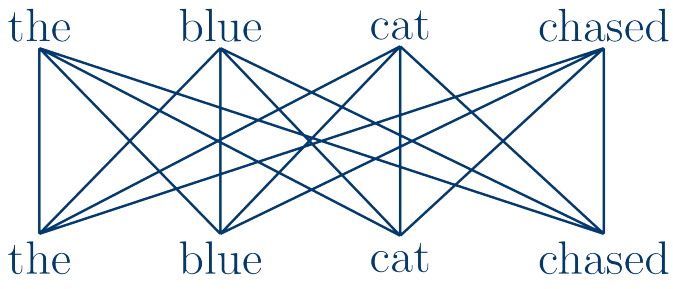


Causal Attention
(GPT-Like)



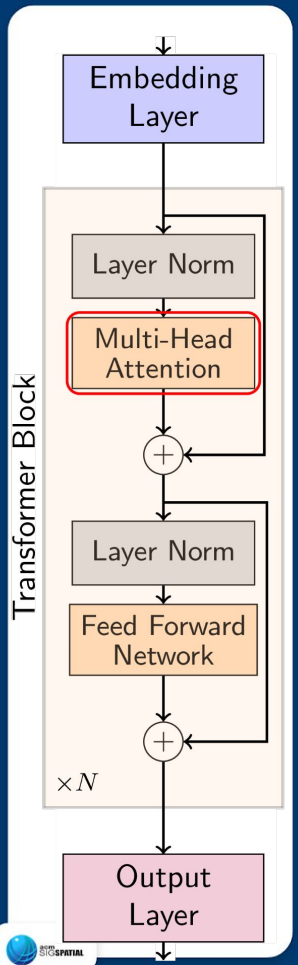
	the	*blue*	*cat*	*chased*
the	0.49			
blue	0.73	0.44		
cat	0.26	0.35	0.63	
chased	0.16	0.52	0.23	0.80

Cross Attention
(Bert-Like)



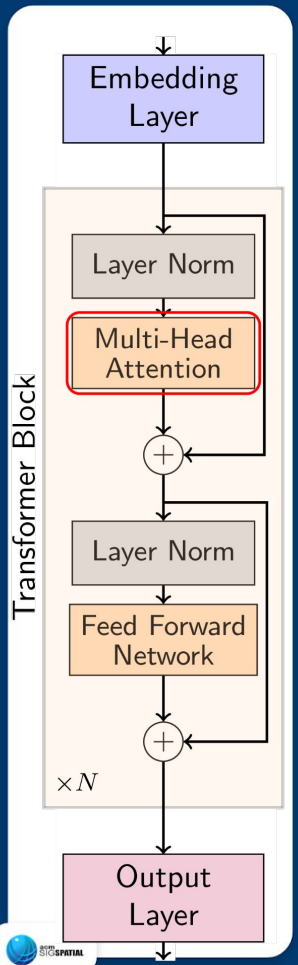
	the	*blue*	*cat*	*chased*
the	0.49	0.12	0.56	0.87
blue	0.73	0.44	0.91	0.15
cat	0.26	0.35	0.63	0.40
chased	0.16	0.52	0.23	0.80

Transformer block - self attention code 1/2



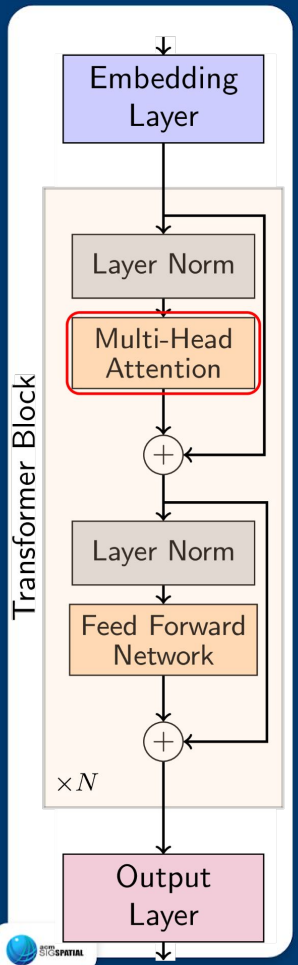

```python
import torch
import torch.nn as nn


class CausalAttention(nn.Module):
  def __init__(self, d_emb, d_QK, d_head, context_length):
    super().__init__()


    self.W_query = nn.Linear(d_emb, d_QK, bias=False)
    self.W_key = nn.Linear(d_emb, d_QK, bias=False)
    self.W_value = nn.Linear(d_emb, d_head, bias=False)
    self.register_buffer("mask", torch.triu(torch.ones(context_length,
                                                       context_length),
                                            diagonal=1)
    )
```

Transformer block - self attention code 2/2



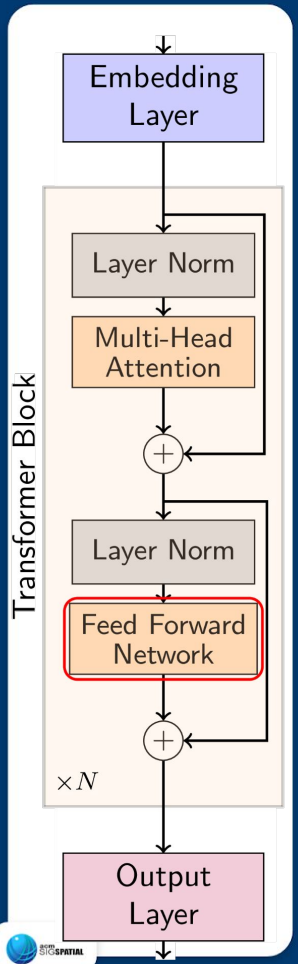

```python
class CausalAttention(nn.Module):
   ...
  def forward(self, x):
    b, num_tokens, d_emb = x.shape # [1, context_size, d_emb]
    keys = self.W_key(x)           # [1, context_size, d_QK]
    queries = self.W_query(x)
    values = self.W_value(x)

    attn_scores = queries @ keys.transpose(1, 2)
    attn_scores.masked_fill_(self.mask.bool()[:num_tokens, :num_tokens],
                             -torch.inf)

    attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)

    context_vec = attn_weights @ values
    return context_vec
```

Transformer block - Multi-Head attention - Code




```python
class MultiHeadAttention(nn.Module):
  def __init__(self, d_embedding, d_QK, context_length, num_heads):
    super().__init__()
    assert d_embedding % num_heads == 0,
    d_head = d_embedding // num_heads

    self.heads = nn.ModuleList(
      [CausalAttention(d_embedding, d_QK, d_head, context_length)
       for _ in range(num_heads)]
    )

  def forward(self, x):
    return torch.cat([head(x) for head in self.heads], dim=-1)
```

Transformer block - Feed Forward Network (FFN)



➔ **The attention layer mixes information across tokens**

The friend of Harry Potter is Ron

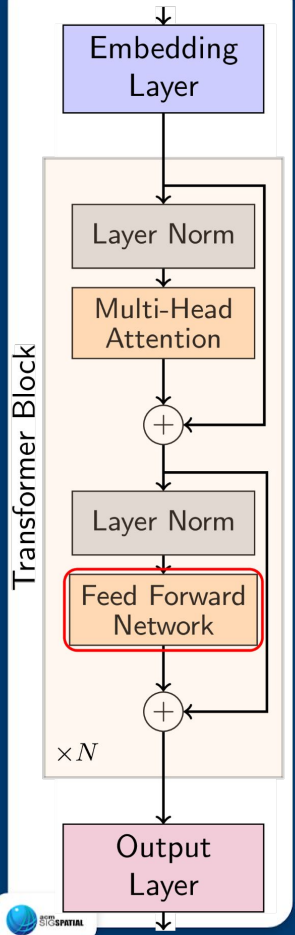
➔ **FFN then applies a nonlinear transformation per token**

- ➔ reinterpret the attended information into a richer space
- ➔ Memory of the Transformer

FFN: Harry Potter == Wizard

➔ Attention: Ron == Wizard

Simplification, knowledge is stored in all weights



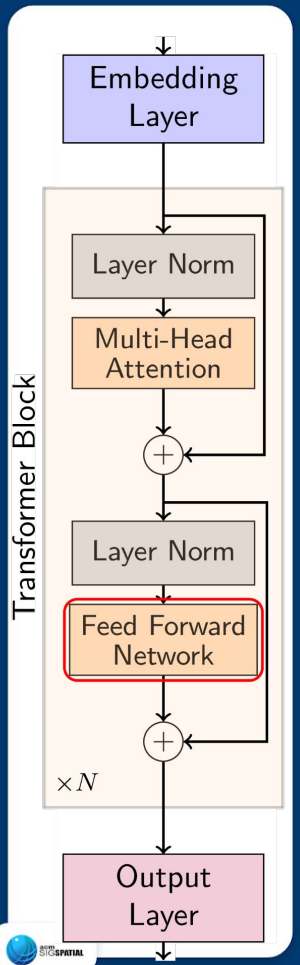
Multi-Layer Perceptron (MLP) applied independently on all "Embedding"

$$FFN(\vec{E}_i) = w_2 \times \sigma(w_1 \times \vec{E}_i + b_1) + b_2$$

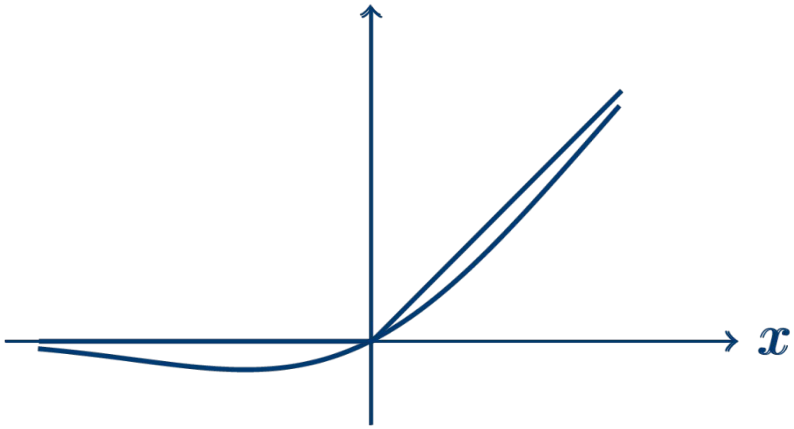
FFN behave as key-value associative memory:

- ➔ 1st layer projects into a "key space" $(w_1,\ b_1)$ ➔ Questions
- ➔ Nonlinearity activates only some neurons (σ) ➔ Filter
- ➔ 2nd layer maps selected activations to "values" $(w_2,\ b_2)$ ➔ Answers

Feed Forward Network - Activation function



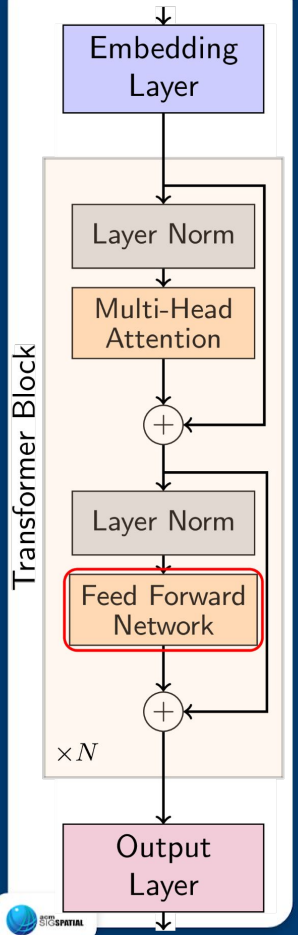
$$\begin{bmatrix} -2 \\ -0.5 \\ 1 \\ 3 \end{bmatrix}$$



ReLU

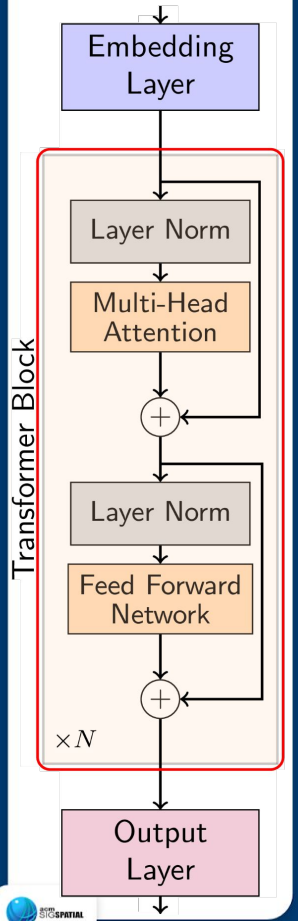
$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 3 \end{bmatrix}$$

Feed Forward Network - Code



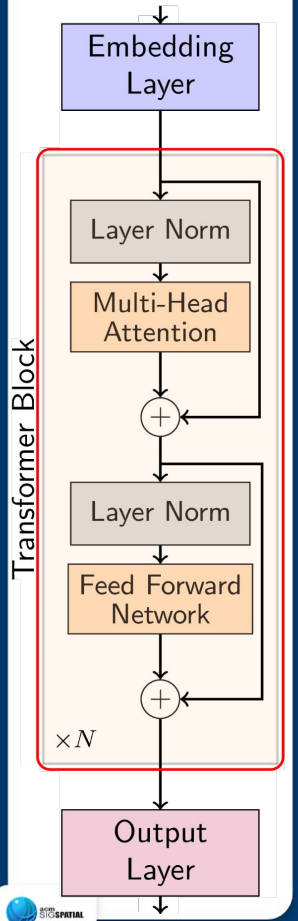

```python
class FeedForward(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(emb_dim, 4 * emb_dim),
            GELU(),
            nn.Linear(4 * emb_dim, emb_dim),
        )

    def forward(self, x):
        return self.layers(x)
```

Transformer - Code

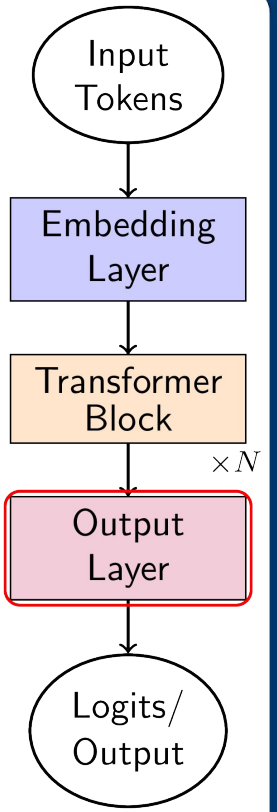



```python
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_embedding=cfg["emb_dim"],
            d_QK=cfg["QK_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"])
        self.ff = FeedForward(cfg["emb_dim"])
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
```

Transformer - Code

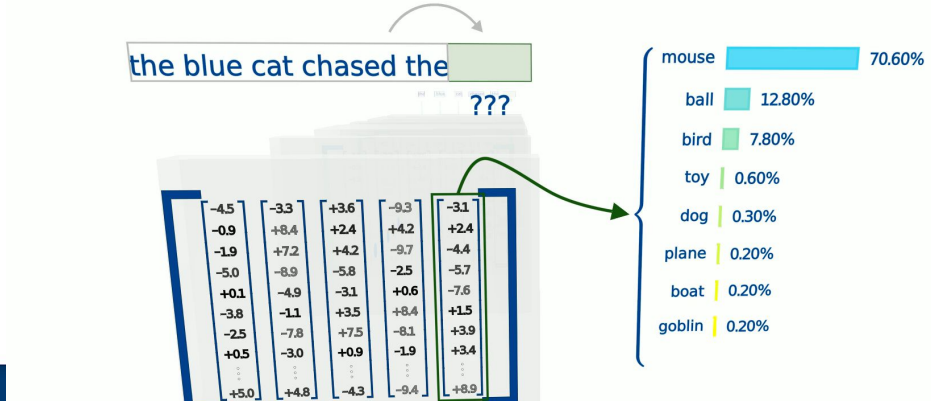



```
class TransformerBlock(nn.Module):
  ...
  def forward(self, x):
    shortcut = x
    x = self.norm1(x)
    x = self.att(x)
    x = x + shortcut

    shortcut = x
    x = self.norm2(x)
    x = self.ff(x)
    x = x + shortcut

    return
```

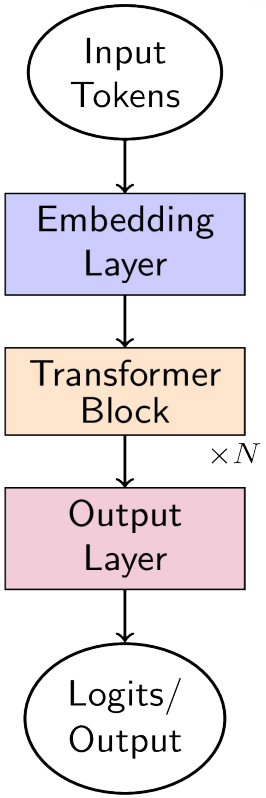


Linear transformation from embedding space to vocabulary space

$$\underbrace{\begin{bmatrix} \\ \\ \\ \vdots \end{bmatrix}}_{d_{emb}} \underbrace{\begin{bmatrix} \vec{E}_0 & \vec{E}_1 & \cdots & \vec{E}_n \end{bmatrix}}_{context} = \underbrace{\begin{bmatrix} \vec{L}_0 & \vec{L}_1 & \cdots & \vec{L}_n \end{bmatrix}}_{context} \Big\} vocab\_size$$

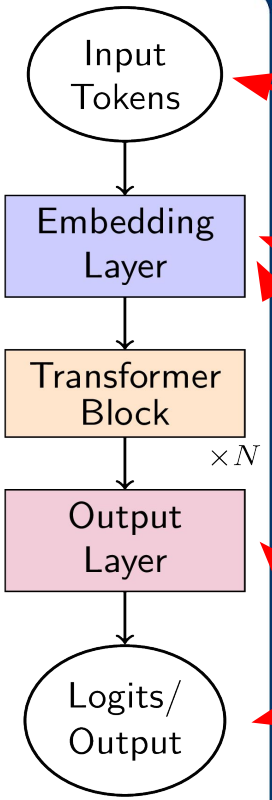


GPT2 - Main Class - Code



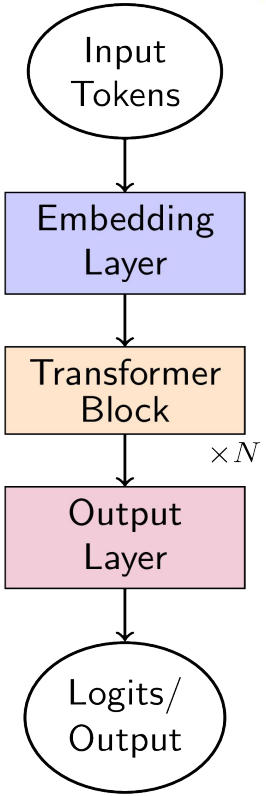

```python
class GPTModel(nn.Module):
  def __init__(self, cfg):
    super().__init__()
    self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
    self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])

    self.trf_blocks = nn.Sequential(
        *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

    self.final_norm = LayerNorm(cfg["emb_dim"])
    self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)
```

From GPT-2 to a trajectory foundation model

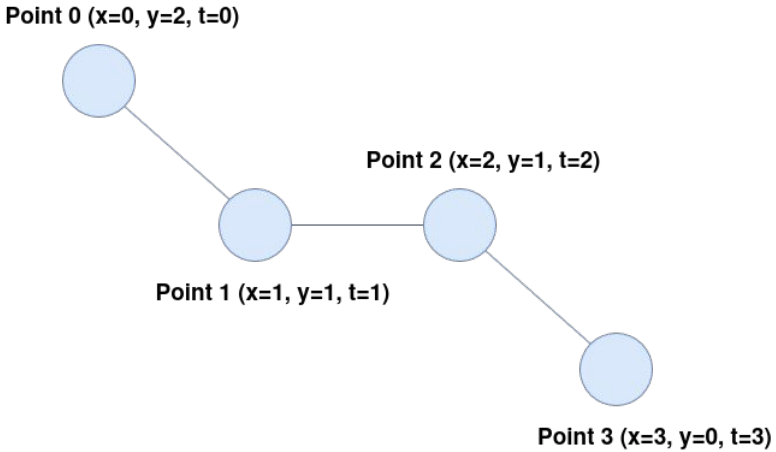


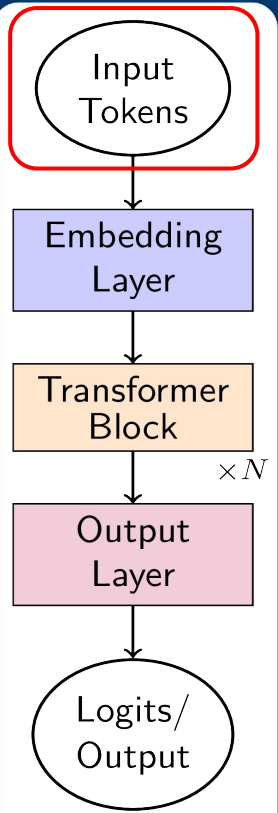
- Sentences use **discrete tokens** (words/subwords) while trajectories use **continuous spatiotemporal points (st points)**.
- Sentence representations come from having an embedding for each token, while there are an **infinity of possible**
- Words in a sentence do not hold **positional information**, while st points do due to their **time dimension**.
- Next-word prediction is **classification** while next-point prediction is **regression.**

Definition of a trajectory for the purpose of this presentation

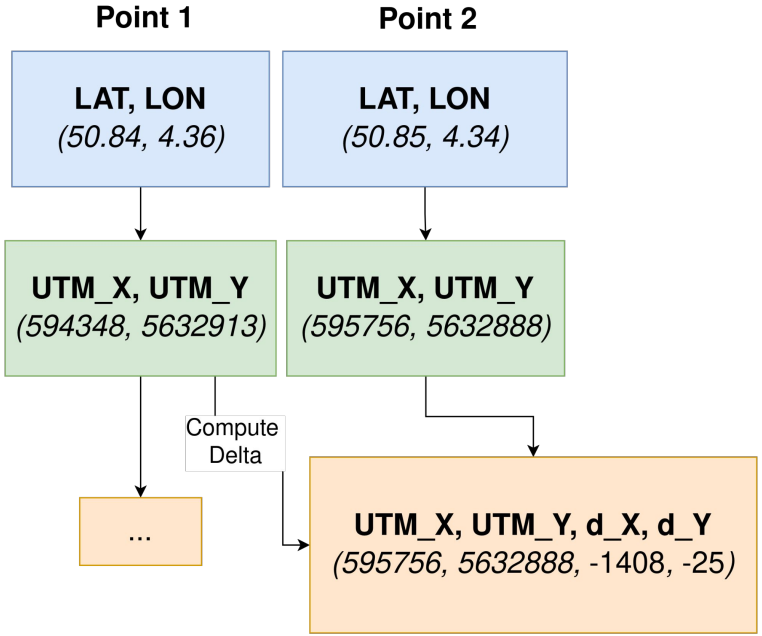


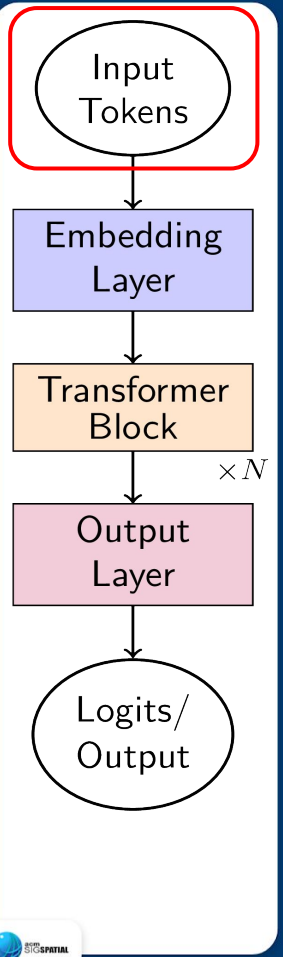
- A trajectory is a list of **tuples**
- Each tuple is composed of three values; position (x, y) and the time (t) at which the position



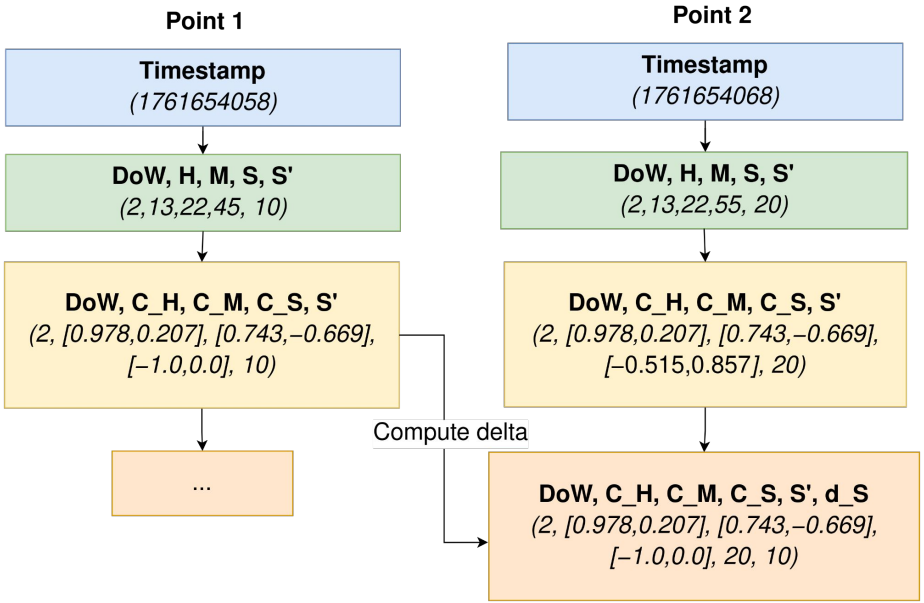


- We first convert our longitude, latitude to UTM.
- We compute **d_x** and **d_y**, the distance delta between two st points.

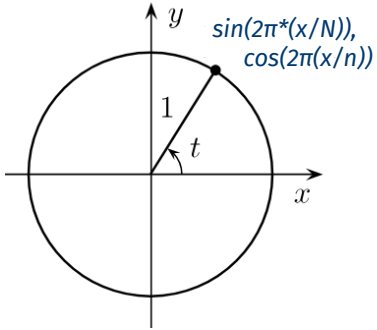




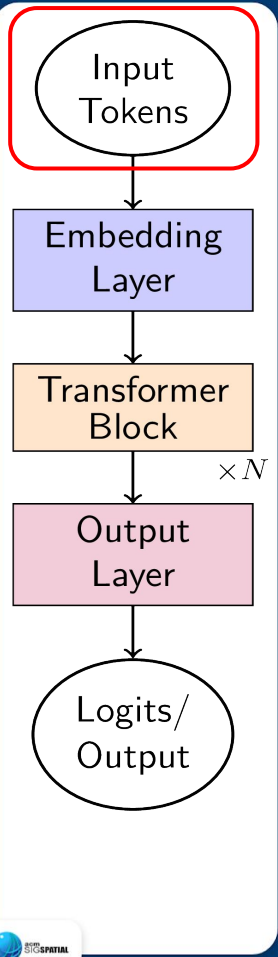
- We then split the temporal part (timestamp), into **multiple sub components**
- We apply **cyclic encoding** to all components except day of the week.



Cyclic encoding

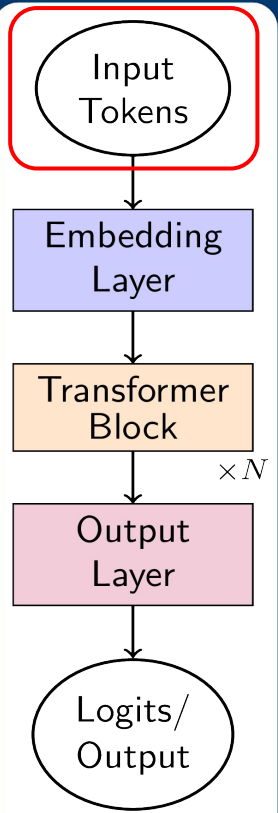


Spatiotemporal point feature extraction - The code

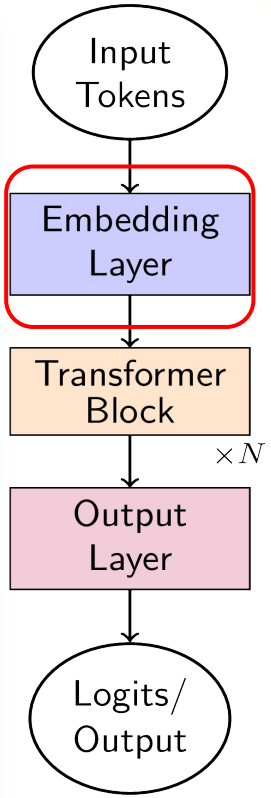



```python
def extract_spatial_features(
    coordinates: torch.Tensor,
) → torch.Tensor:

    utm_list = [
        utm.from_latlon(lat.item(), lng.item())[:2]
        for lat, lng in coordinates
    ]

    utm_tensor = torch.tensor(utm_list, dtype=torch.float32)

    deltas = torch.zeros_like(utm_tensor)
    deltas[1:] = utm_tensor[1:] - utm_tensor[:-1]

    deltas /= DELTA_SCALING_FACTOR_X
    normalized = utm_tensor / SCALING_FACTOR_X

    return torch.cat([normalized, deltas], dim=1)
```

Spatiotemporal point feature extraction - The code

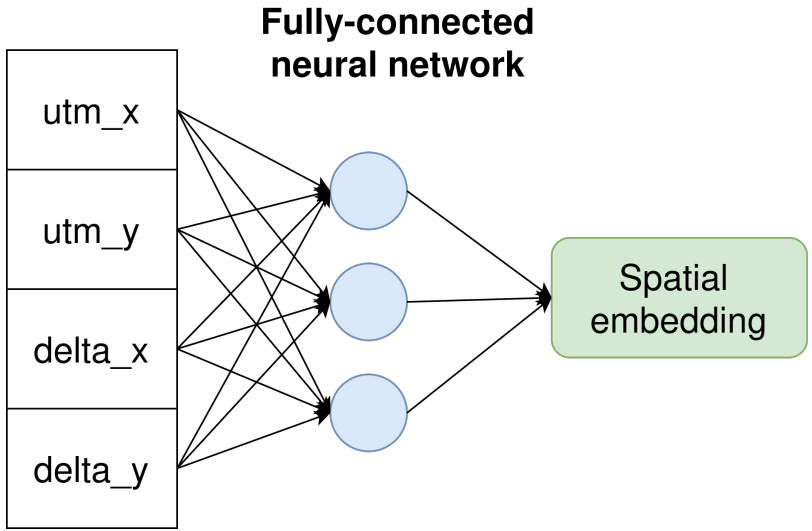



```python
def extract_temporal_features(timestamps: torch.Tensor) → torch.Tensor:
    # Convert to datetime objects
    dt_list = [
        datetime.datetime.fromtimestamp(t.item(), tz=datetime.timezone.utc)
        for t in timestamps
    ]

    # Extract calendar components
    day = torch.tensor([dt.weekday() for dt in dt_list], dtype=torch.float32)
    hour = torch.tensor([dt.hour for dt in dt_list], dtype=torch.float32)
    minute = torch.tensor([dt.minute for dt in dt_list], dtype=torch.float32)
    second = torch.tensor([dt.second for dt in dt_list], dtype=torch.float32)

    # Cyclic encodings
    hour_enc = cyclic_encoding(hour, 24)
    minute_enc = cyclic_encoding(minute, 60)
    second_enc = cyclic_encoding(second, 60)

    # Timing features
    seconds_since_start = timestamps - timestamps[0]
    delta_seconds = torch.zeros_like(timestamps)
    delta_seconds[1:] = timestamps[1:] - timestamps[:-1]

    # Concat all..
```

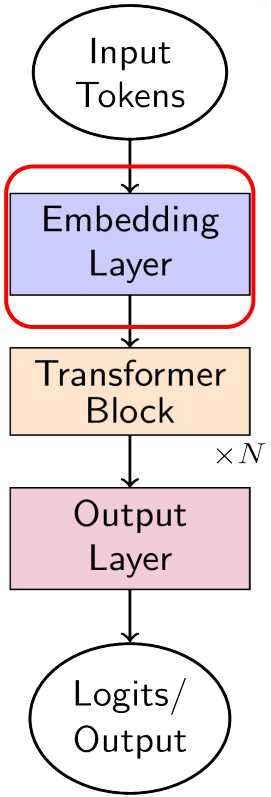


Now that we have our encoded values, we will derive their embeddings

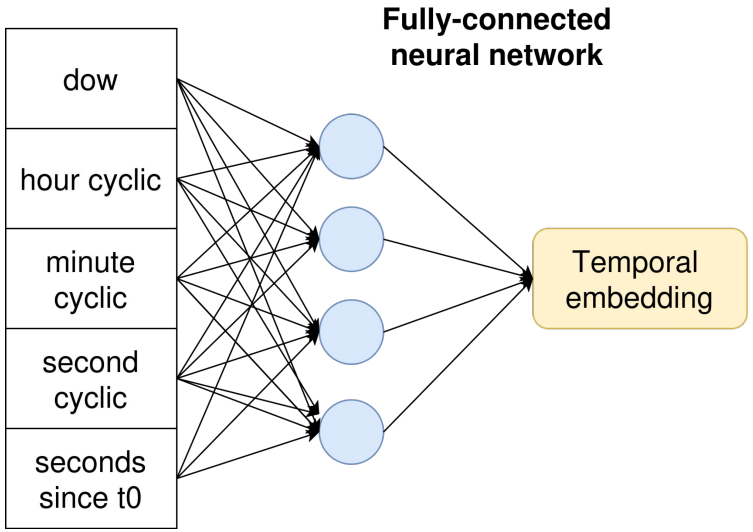
➔ We will first generate an embedding for the **spatial dimension**



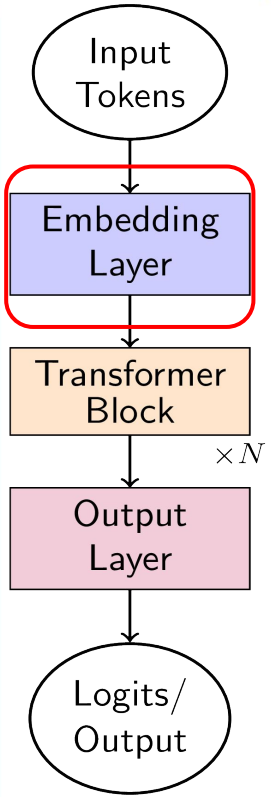
S' =SpatialEmbedding(S)



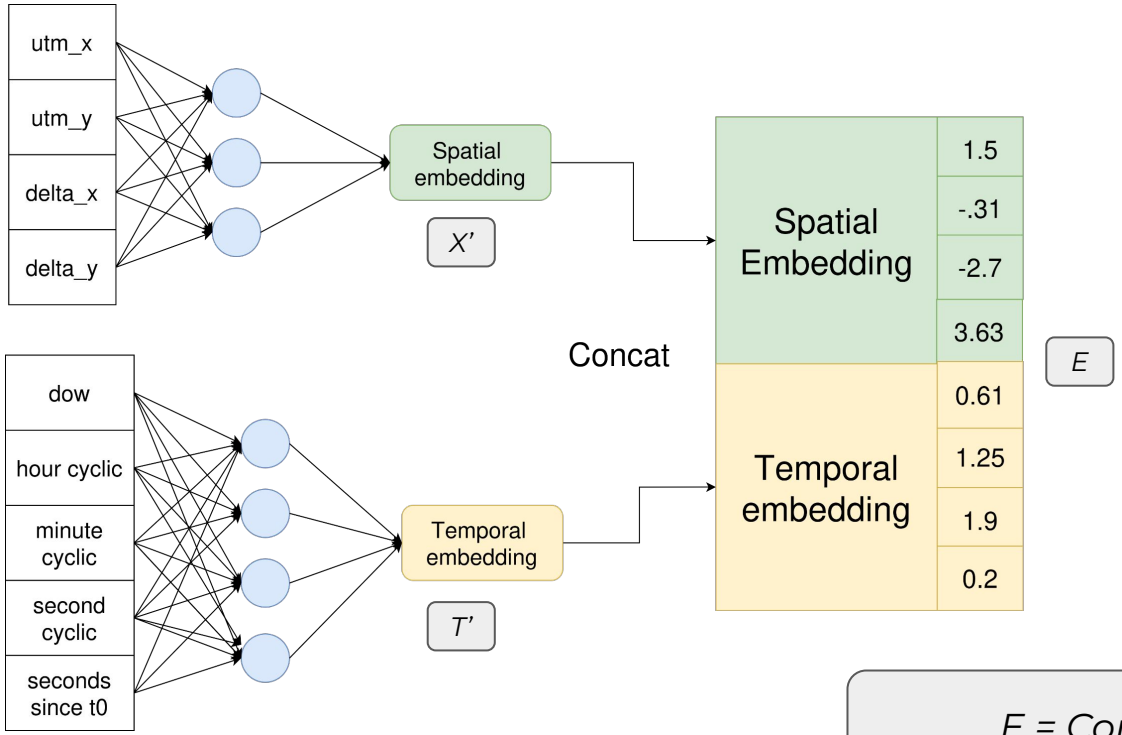
➔ And now we can generate the temporal embedding using a similar approach.



T' =TemporalEmbedding(T)

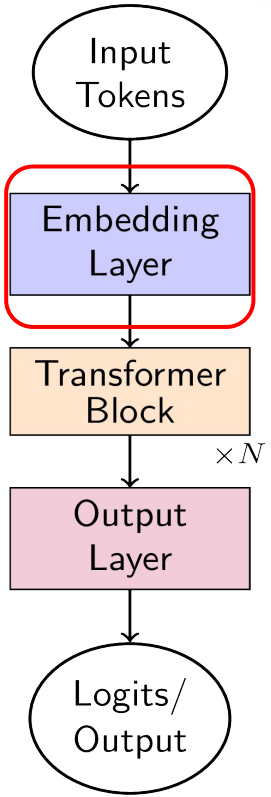


➔ Finally, we concatenate both into a single embedding



$E = Concat(T', S')$

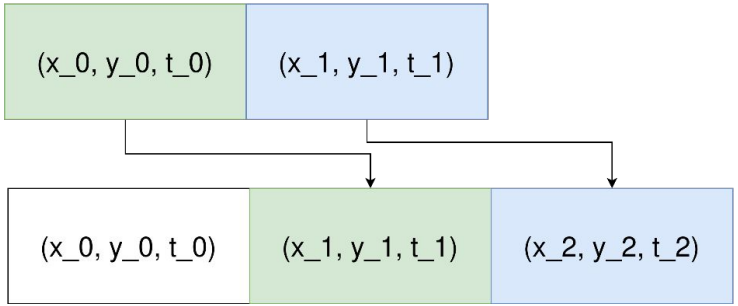
From our encoded values towards embedding - The code



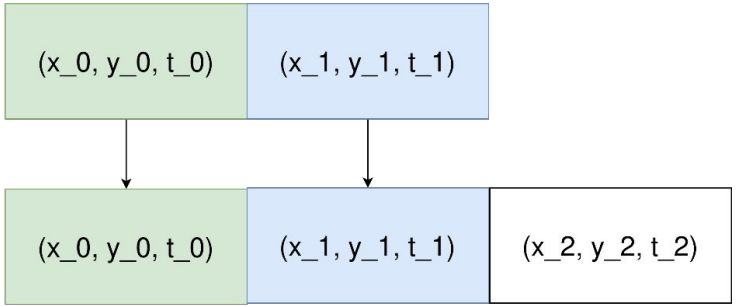

```python
class SpatioTemporalEncoder(nn.Module):
    def __init__(
        self,
        spatial_features_dimension: int = constants.SPATIAL_FEATURES_DIMENSION,
        temporal_features_dimension: int = constants.TEMPORAL_FEATURES_DIMENSION,
        spatial_embedding_dimension: int = constants.SPATIAL_EMBEDDING_DIMENSION,
        temporal_embedding_dimension: int = constants.TEMPORAL_EMBEDDING_DIMENSION,
        eps: float = 1e-6,
    ):
        super().__init__()
        self.eps = eps
        self.spatial_fc = nn.Linear(
            spatial_features_dimension, spatial_embedding_dimension, dtype=torch.float32
        )
        self.temporal_fc = nn.Linear(
            temporal_features_dimension,
            temporal_embedding_dimension,
            dtype=torch.float32,
        )

    def forward(
        self, spatial_input: torch.Tensor, temporal_input: torch.Tensor
    ) → torch.Tensor:
        spatial_embed = self.spatial_fc(spatial_input)
        temporal_embed = self.temporal_fc(temporal_input)

        return torch.cat(
            [spatial_embed, temporal_embed], dim=-1
        )  # [B, T, spatial_emb + temporal_emb]
```

Test, test, test... But how?

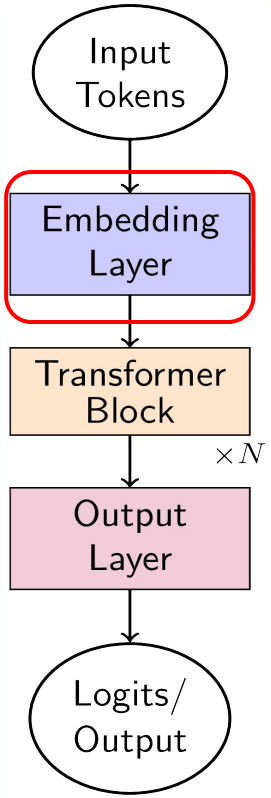
- Observe that the embedding is basically an **encoder**.
- For each step, we propose to build a mirrored decoder model, similar to an **auto-encoder.**
- We use **synthetic data (controlled data)** while in the **first stage** of building the model.



Actual training



Testing purposes

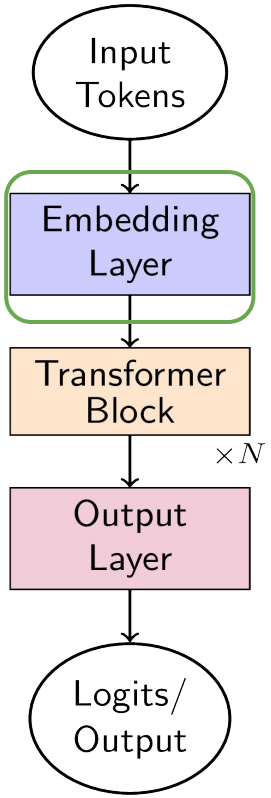


Positional encoding ➔ a modification on the vectors of the embedding matrix to include positional information.

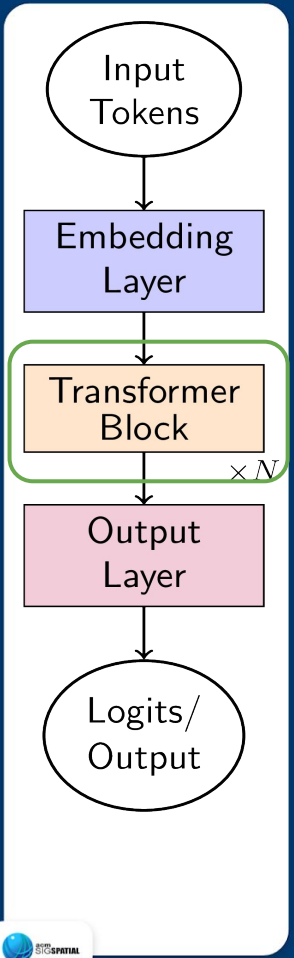
One can argue that the **time** dimension already holds this information.

➔ In our case we decided to keep the same learnable absolute embeddings as GPT-2

Positional encoding - The code



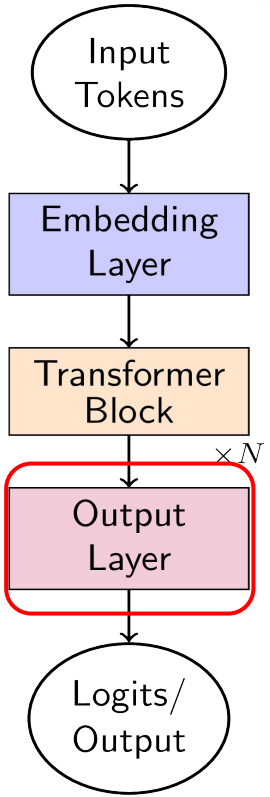

```python
import torch
from torch import nn

class LearnedPositionalEncoding(nn.Module):
    """Learned positional encoding (like GPT-2)."""

    def __init__(self, embed_dim: int, max_len: int):
        super().__init__()
        self.pos_emb = nn.Embedding(max_len, embed_dim)

    def forward(self, x: torch.Tensor) → torch.Tensor:
        seq_len = x.size(0)
        positions = torch.arange(seq_len, device=x.device)
        pos = self.pos_emb(positions)

        pos = pos.unsqueeze(0)

        return x + pos
```



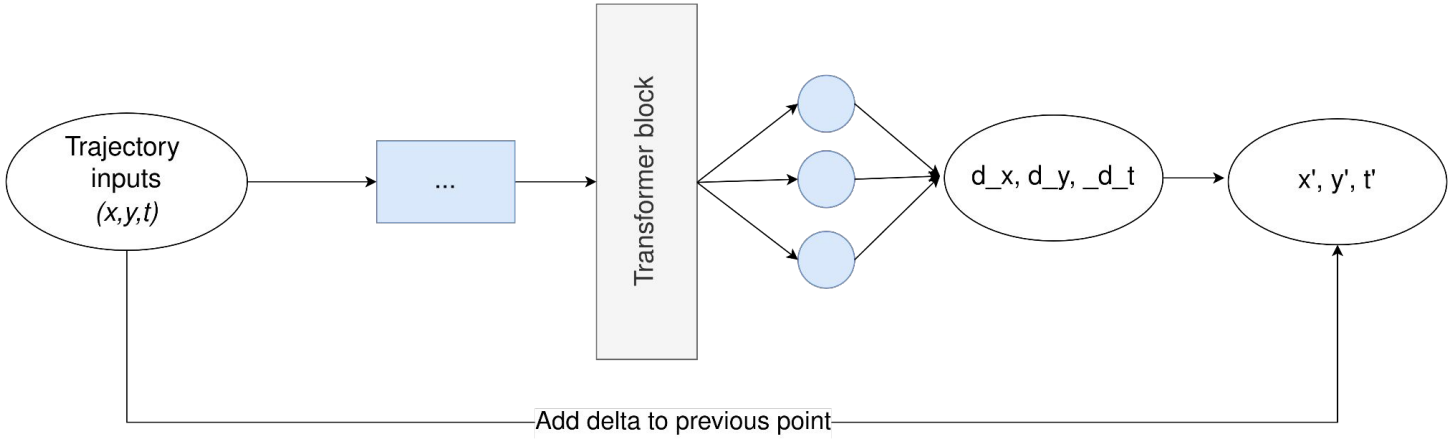
- **Ctrl+C Ctrl+V from GPT-2**
- We reuse the **transformer block as is with the same causal mask.**
- Hence we can simply plug it and feed it the matrix we built before
- We will only use a smaller number of stacked transformers *N*.

E' = TransformerBlock(E, N)

Test, test, test...



- After our sequence of transformer blocks, we want to convert each embedding back into a **st point.**

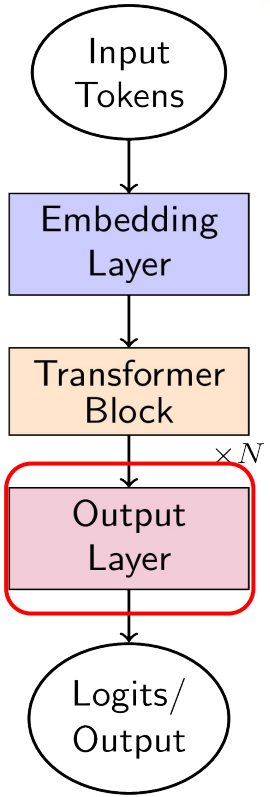


d_x, d_y, d_t = Output(E')

Test, test, test

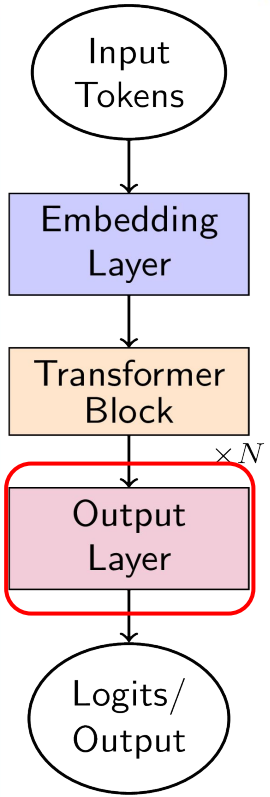
We still predict the **same input point.**

From our embedding back to a spatio-temporal point




```python
import torch
from torch import nn, Tensor
from src.modules import constants


class OutputModule(nn.Module):
    """
    Output projection module.

    Projects the transformer hidden representations into the final output space.
    """

    def __init__(self, embed_dim: int = constants.EMBEDDING_DIMENSION):
        """
        :param embed_dim: Dimensionality of transformer embeddings before projection.
        """
        super().__init__()
        self.output_proj = nn.Linear(embed_dim, constants.OUTPUT_DIMENSION,
dtype=torch.float32)

    def forward(self, x: Tensor) -> Tensor:
        """
        :param x: Input tensor of shape [T, B, E], representing hidden states
                  from the transformer encoder, where
                  T = sequence length,
                  B = batch size,
                  E = embedding dimension.
        :return: Tensor of shape [T, B, OUTPUT_DIMENSION], representing projected
outputs.
        """
        return self.output_proj(x)
```

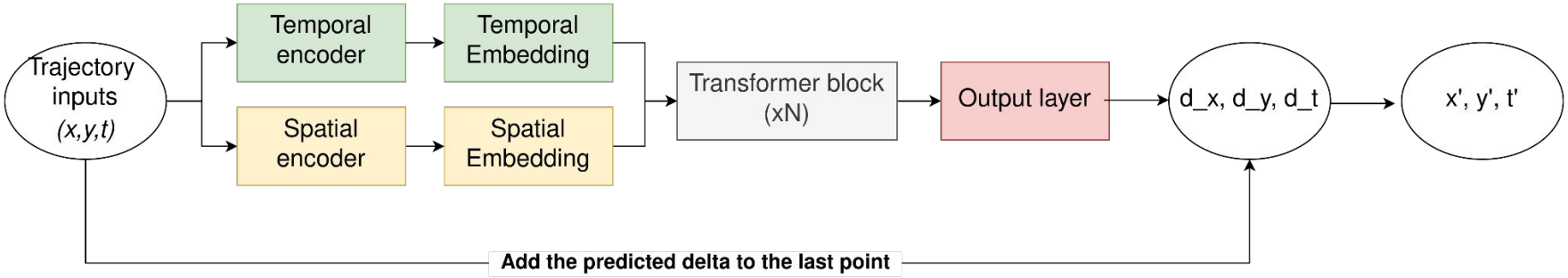
From our embedding back to a spatio-temporal point




```python
class CheminTF(nn.Module):
    def __init__(
        self,
        n_heads: int,
        num_layers: int,
        embed_dim: int = constants.EMBEDDING_DIMENSION,
        max_len: int = 50,
        dropout: float = 0.1,
    ):
        super().__init__()

        self.input_encoder = SpatioTemporalEncoder(
            spatial_features_dimension=constants.SPATIAL_FEATURES_DIMENSION,
            temporal_features_dimension=constants.TEMPORAL_FEATURES_DIMENSION,
            spatial_embedding_dimension=constants.SPATIAL_EMBEDDING_DIMENSION,
            temporal_embedding_dimension=constants.TEMPORAL_EMBEDDING_DIMENSION,
        )

        self.pos_encoder = LearnedPositionalEncoding(
            embedding_dim=constants.EMBEDDING_DIMENSION, max_seq_len=max_len
        )

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim,
            nhead=n_heads,
            dim_feedforward=4 * embed_dim,
            dropout=dropout,
            batch_first=False,
            dtype=torch.float32,
        )

        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=num_layers, )

        self.output = OutputModule(embed_dim=embed_dim)
```

Time to review it all

Let us now go back to our model schema

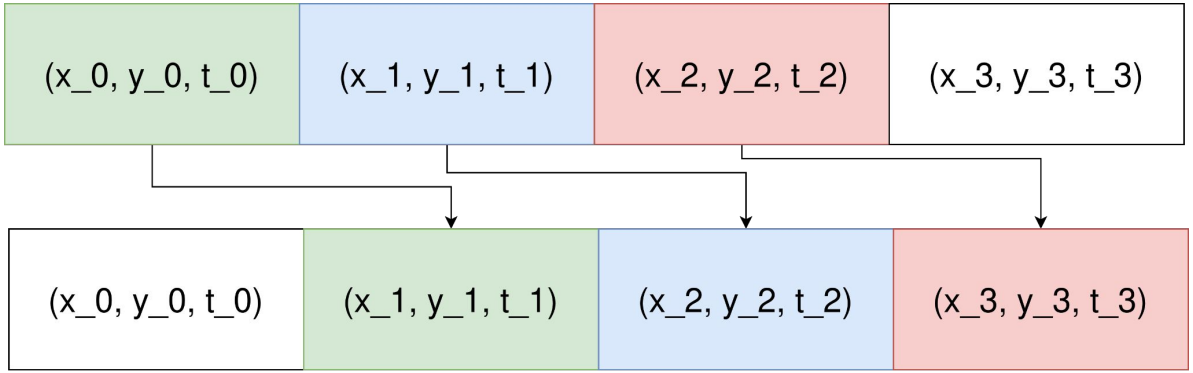



```
d_x, d_y, d_t= Output(
   TransformerBlock(
      Concat(
         SpatialEmbedding(
            SpatialEncoder(x, y)),
         TemporalEmbedding(
            TemporalEncoder(t)) )
   , N ) )
```

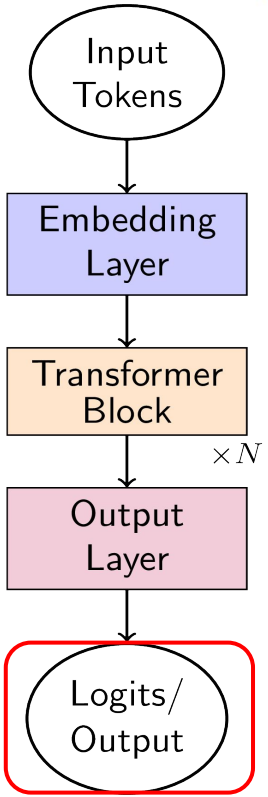
Training our model

With a synthetic dataset at first

- Until now, the only tests we did were with an auto-encoder **reconstructing back the input value.**
- We will now ask the model to predict the **next point instead of regenerating the previous one** using the same training pipeline as GPT-2.

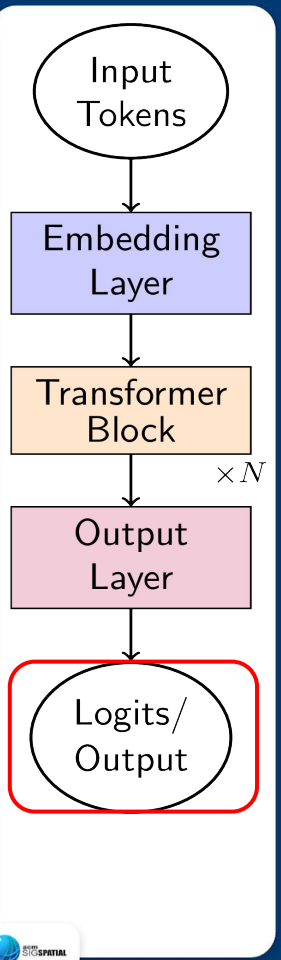


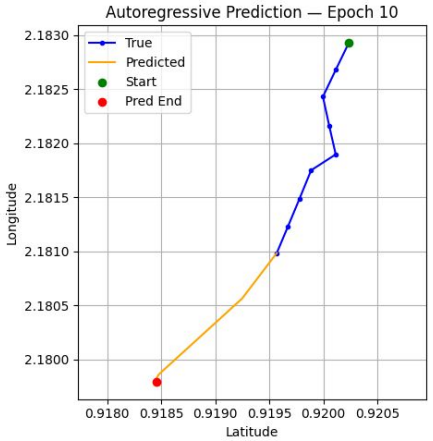
Training our model - The code

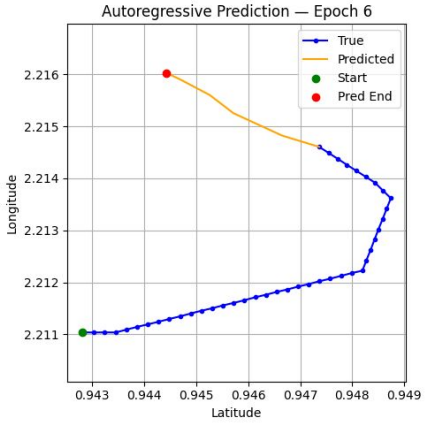


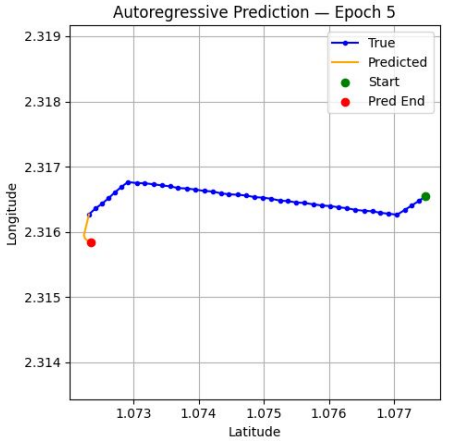

```python
def train_model(epochs=20, bs=8, lr=1e-4, device="cuda"):
    """Train CheminTF to predict spatial deltas (Δlat, Δlng, Δt)."""
    ds = SyntheticTrajectoryDataset(100)
    dl = DataLoader(ds, batch_size=bs, shuffle=True, collate_fn=collate_batch)

    model = CheminTF(n_heads=8, num_layers=2).to(device)
    opt = Adam(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()

    for e in range(epochs):
        model.train(); total = 0
        for s, t, d in tqdm(dl, desc=f"Epoch {e+1}/{epochs}"):
            opt.zero_grad();
            loss = loss_fn(model(s, t), d)
            loss.backward();
            opt.step();
            total += loss.item()

      return model
```

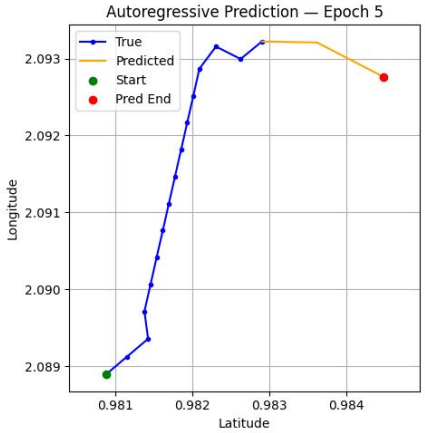
A little view of the model output

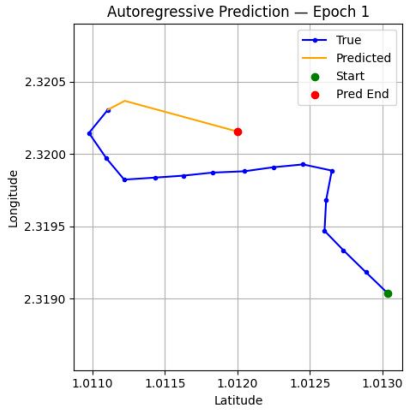


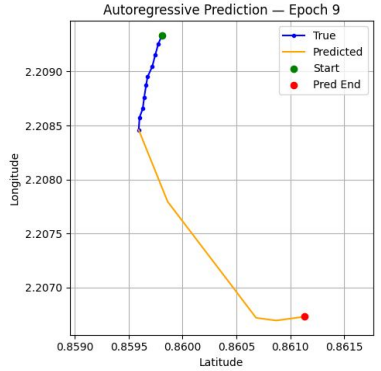










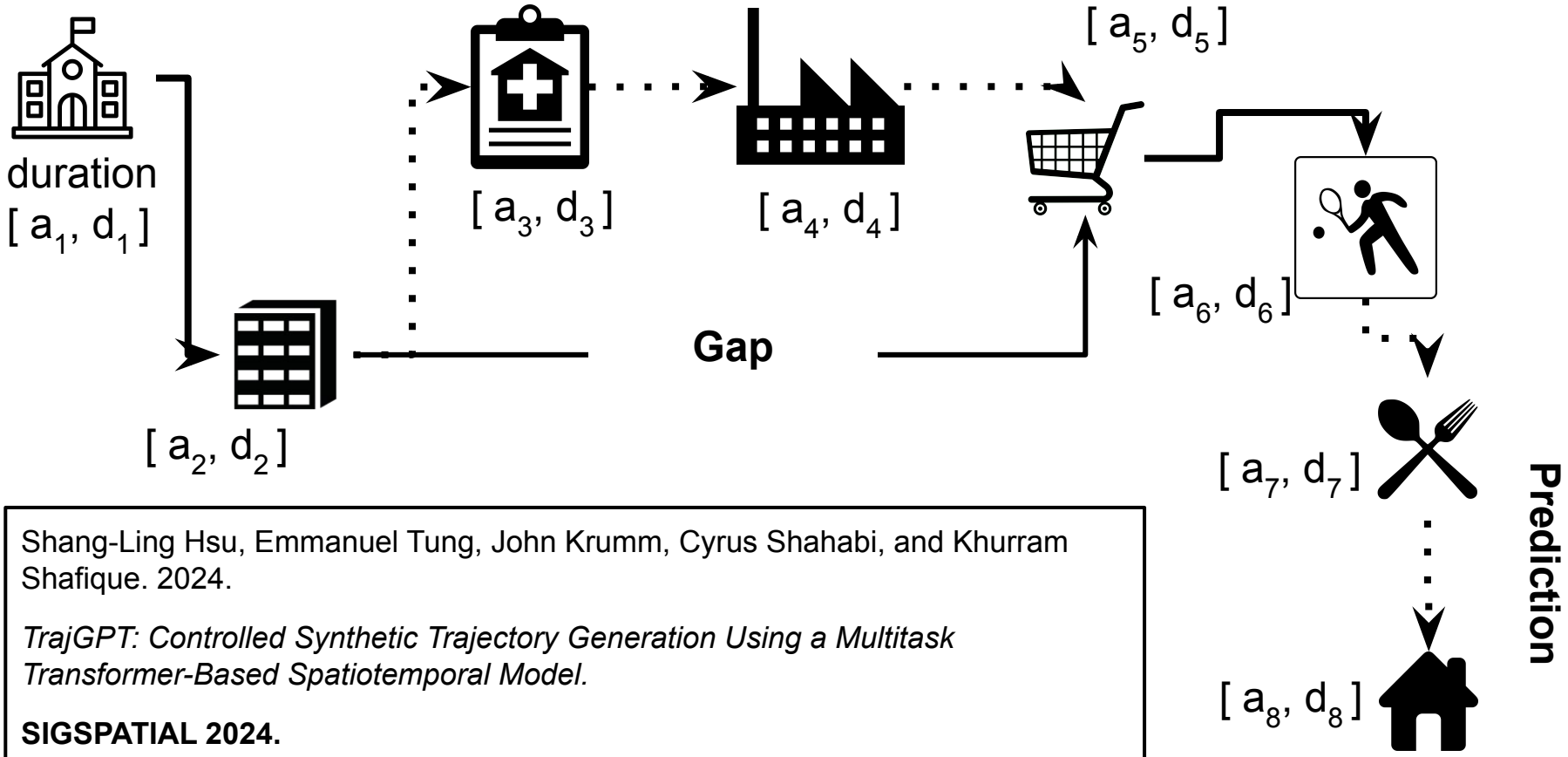


Part 3
Examples from the state of the art

Bridging our educational model with SotA

- We presented an intuitive approach to build a foundation model for trajectory starting from the GPT-2 architecture.

- We now review several related papers and highlight their specific contributions, drawing connections to the model we have just developed whenever possible

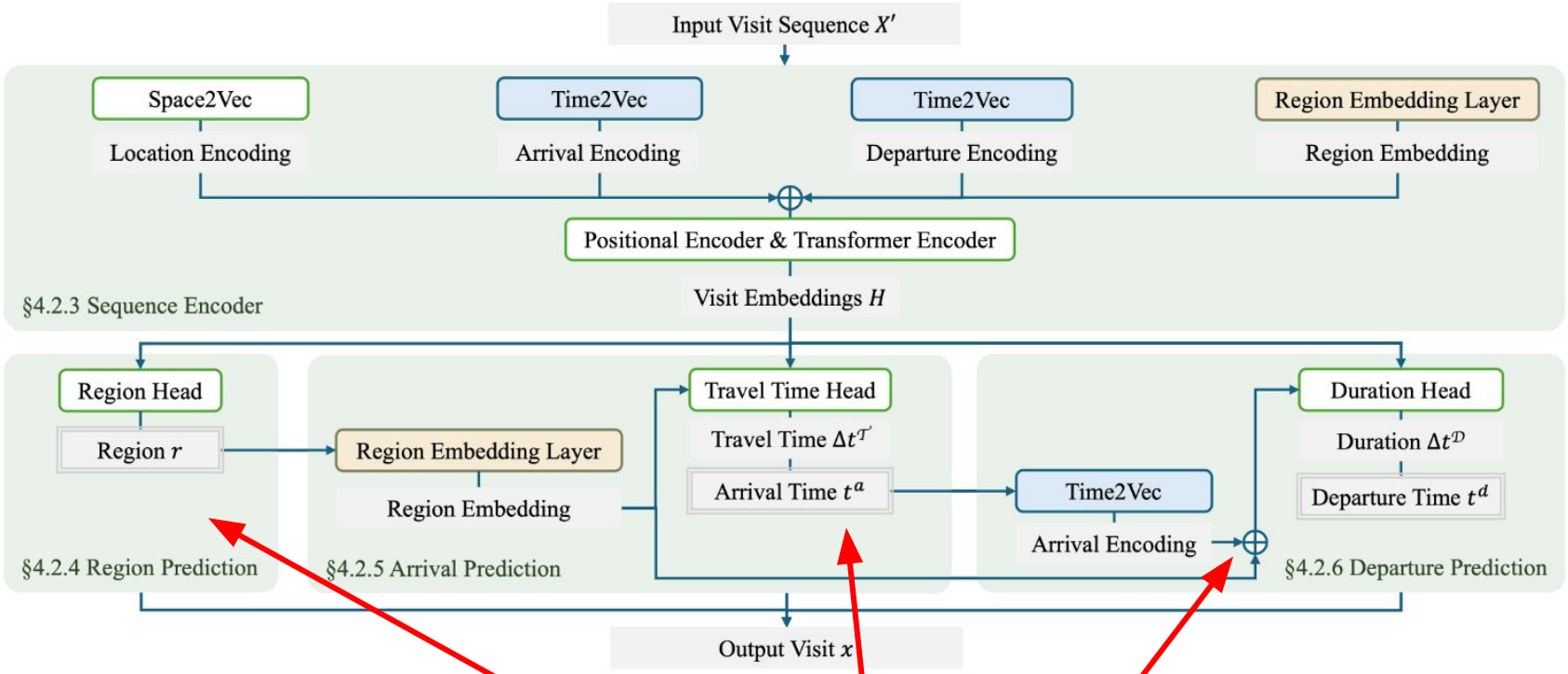


Shang-Ling Hsu, Emmanuel Tung, John Krumm, Cyrus Shahabi, and Khurram Shafique. 2024.

TrajGPT: Controlled Synthetic Trajectory Generation Using a Multitask Transformer-Based Spatiotemporal Model.

SIGSPATIAL 2024.

TrajGPT - architecture



$$P(x \mid X') = P(r,\ t^a,\ t^d \mid X') \quad = P(r \mid X')\ P(t^a \mid X',\ r)\ P(t^d \mid X',\ r,\ t^a)$$

TrajGPT - architecture code

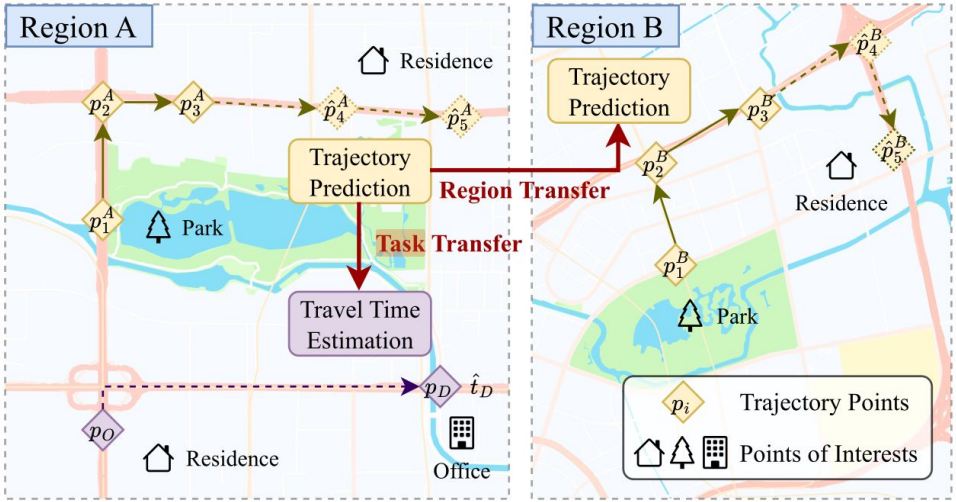
```
class TrajGPT(nn.Module):
    def __init__(self, num_regions, sequence_len, lambda_max,
                 num_heads=2, num_layers=4, num_gaussians=3, d_feedforward=32, d_embed=32, lambda_min=1e0):
        super().__init__()
        self.num_regions = num_regions
        self.d_model = d_embed * 4  # Four: location, arrival_time, departure_time, region_id

        # Sequence encoder
        self.input = SourceInput(num_regions, d_embed, lambda_min, lambda_max)
        self.encoder = CausalEncoder(self.d_model, num_heads, num_layers, sequence_len)

        # Region prediction
        self.region_id_decoder = CausalEncoder(self.d_model, num_heads, 1, sequence_len)
        self.region_id_head = nn.Linear(self.d_model, num_regions + N_SPECIAL_TOKENS)

        # Arrival (travel) prediction
        self.d_travel = d_embed * 2  # Two: region_id, location
        self.travel_decoder = CausalMemoryDecoder(self.d_travel, num_heads, sequence_len, d_feedforward)
        self.travel_head = GMM(self.d_travel, num_gaussians=num_gaussians)

        # Departure (duration) prediction
        self.d_duration = d_embed * 3  # Three: region_id, location, arrival_time
        self.duration_decoder = CausalMemoryDecoder(self.d_duration, num_heads, sequence_len, d_feedforward)
        self.duration_head = GMM(self.d_duration, num_gaussians=num_gaussians)
```

TrajGPT - input embedding

```
class SourceInput(nn.Module):
    def forward(self, region_id, x, y, arrival_time, departure_time):
        """
        Each input variable is a tensor of size (batch_size, seq_len)
        """
        locations = torch.stack([x, y], dim=-1)
        location_encoding = self.space2vec(locations)
        arrival_encoding = self.time2vec(arrival_time)
        departure_encoding = self.time2vec(departure_time)
        region_embedding = self.region_embedding(region_id)
        visit_embedding = torch.concat([
            location_encoding,
            arrival_encoding,
            departure_encoding,
            region_embedding
        ], dim=-1)  # (batch_size, seq_len, d_embed*4)
        return visit_embedding
```

TrajGPT - teacher forcing

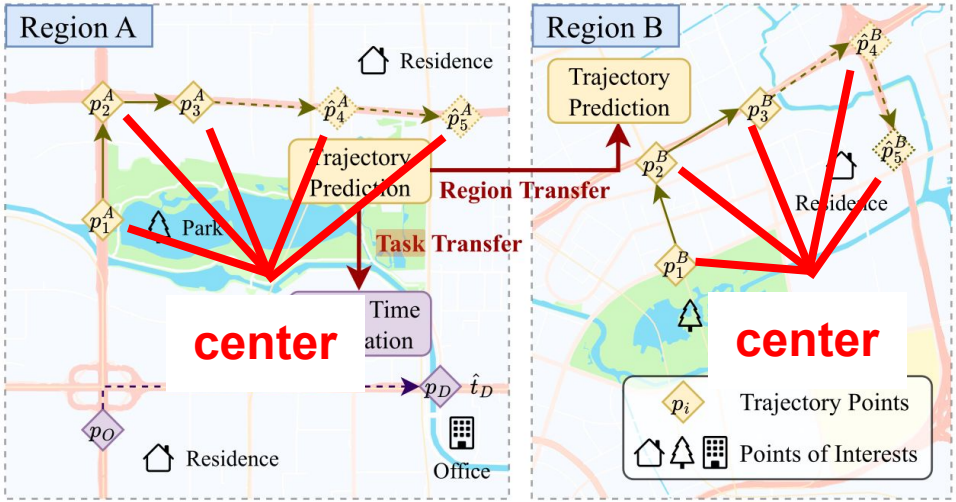
```python
class TrajGPT(nn.Module):

    def forward(self, kwargs):
        # Encode input sequence
        memory, tgt = self.encode_sequence(kwargs)
        # Predict region id
        region_id_out = self.predict_region(memory)
        # Predict travel time with teacher forcing
        travel_out = self.predict_travel_time(memory, tgt)
        # Predict duration with teacher forcing
        duration_out = self.predict_duration(memory, tgt)
        return {
            'region_id': region_id_out,
            'travel_time': travel_out,
            'duration': duration_out
        }
```



Yan Lin, Tonglong Wei, Zeyu Zhou, Haomin Wen, Jilin Hu, Shengnan Guo, Youfang Lin, Huaiyu Wan

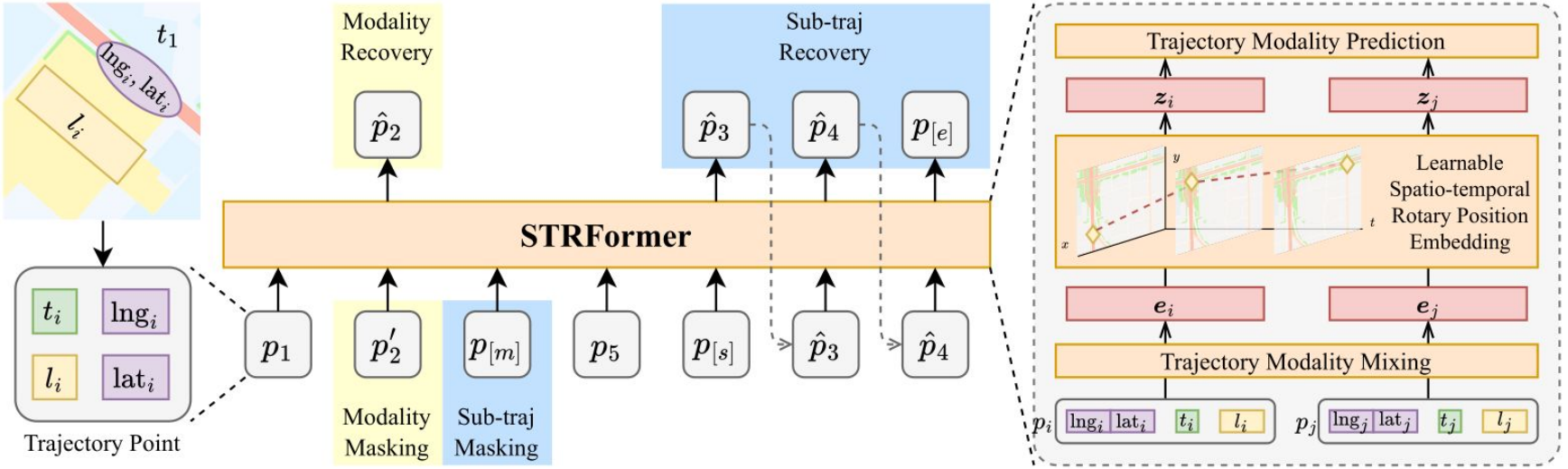
TrajFM: A Vehicle Trajectory Foundation Model for Region and Task Transferability

arXiv:2408.15251



Learnable spatio-temporal Rotary position embedding (STRPE)

$$x_i = (\mathrm{UTM}(\mathrm{lng}_i) - \mathrm{UTM}(\mathrm{lng}_{\mathrm{cen}}))/s_x$$
$$y_i = (\mathrm{UTM}(\mathrm{lat}_i) - \mathrm{UTM}(\mathrm{lat}_{\mathrm{cen}}))/s_y$$



Tasks:

- Trajectory travel time estimation
- Origin-destination travel time estimation
- Trajectory prediction

- Uni-variate time series prediction
- GPT decoder only architecture
- Different context length and prediction horizon

$$f : (\mathbf{y}_{1:L}) \longrightarrow \hat{\mathbf{y}}_{L+1:L+H}.$$

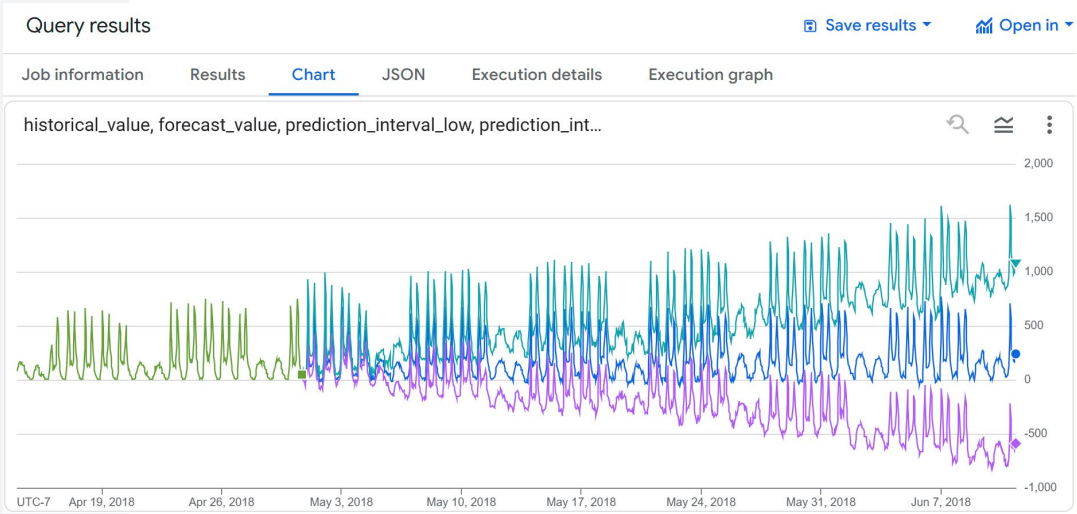
Abhimanyu Das, Weihao kong, Rajat Sen, Yichen Zhou. Google Research, 2024.

A Decoder-Only Foundation Model For Time-Series Forecasting

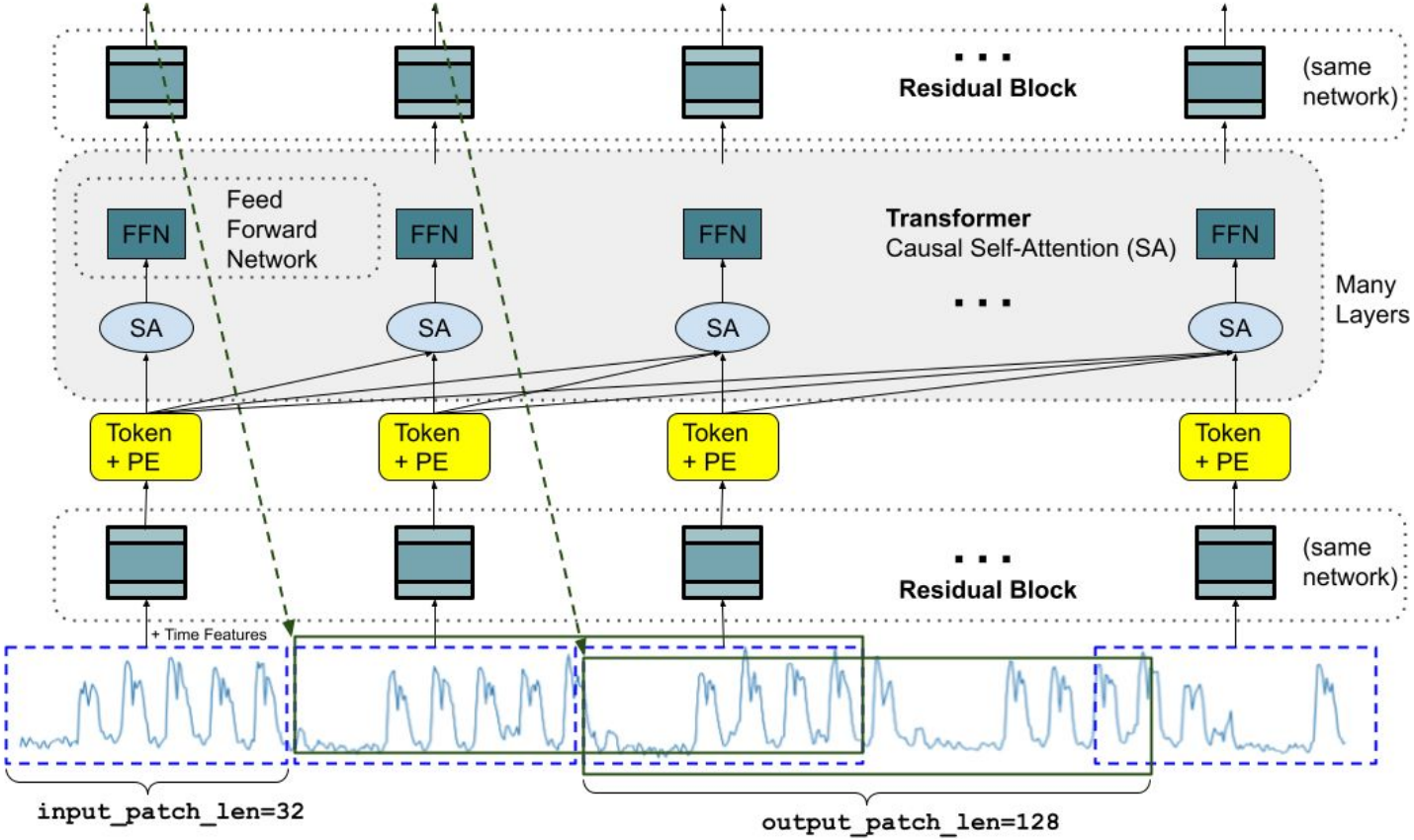
ICML 2024.

TimesFM in BigQuery

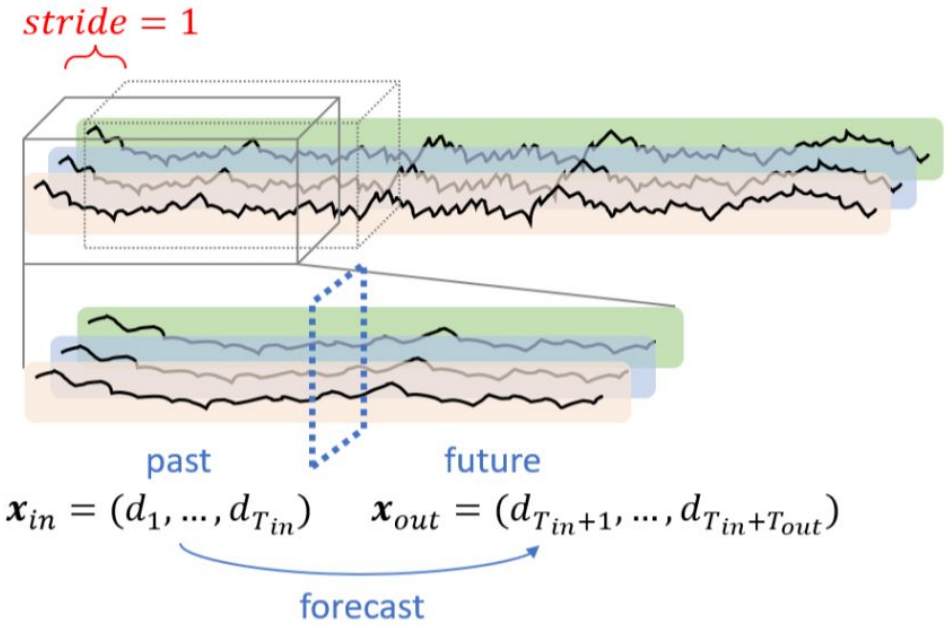
```
UNION ALL
(SELECT forecast_timestamp AS date,
        NULL as historical_value,
        forecast_value as forecast_value,
        'forecast' as type,
        prediction_interval_lower_bound,
        prediction_interval_upper_bound
FROM
    AI.FORECAST(
    (
    SELECT * FROM historical
    ),
    horizon => 720,
    confidence_level => 0.99,
    timestamp_col => 'trip_hour',
    data_col => 'num_trips')))
ORDER BY date asc;
```



TimesFM architecture



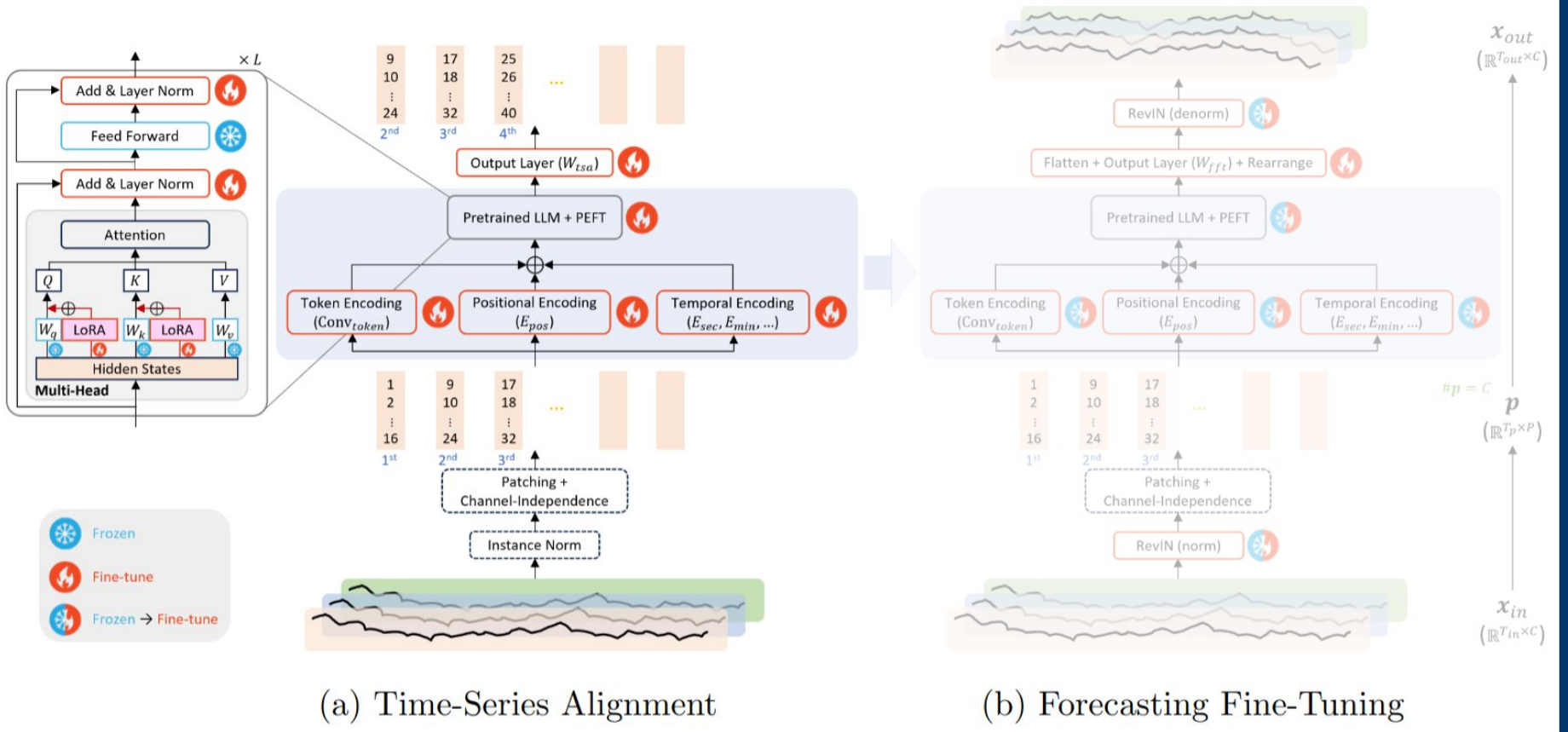
- LLM ➔Time series foundation model
- Multi-variate time series, with C channels



Ching Chang, Wei-Yao Wang, Wen-Chih Peng, and Tien-Fu Chen.

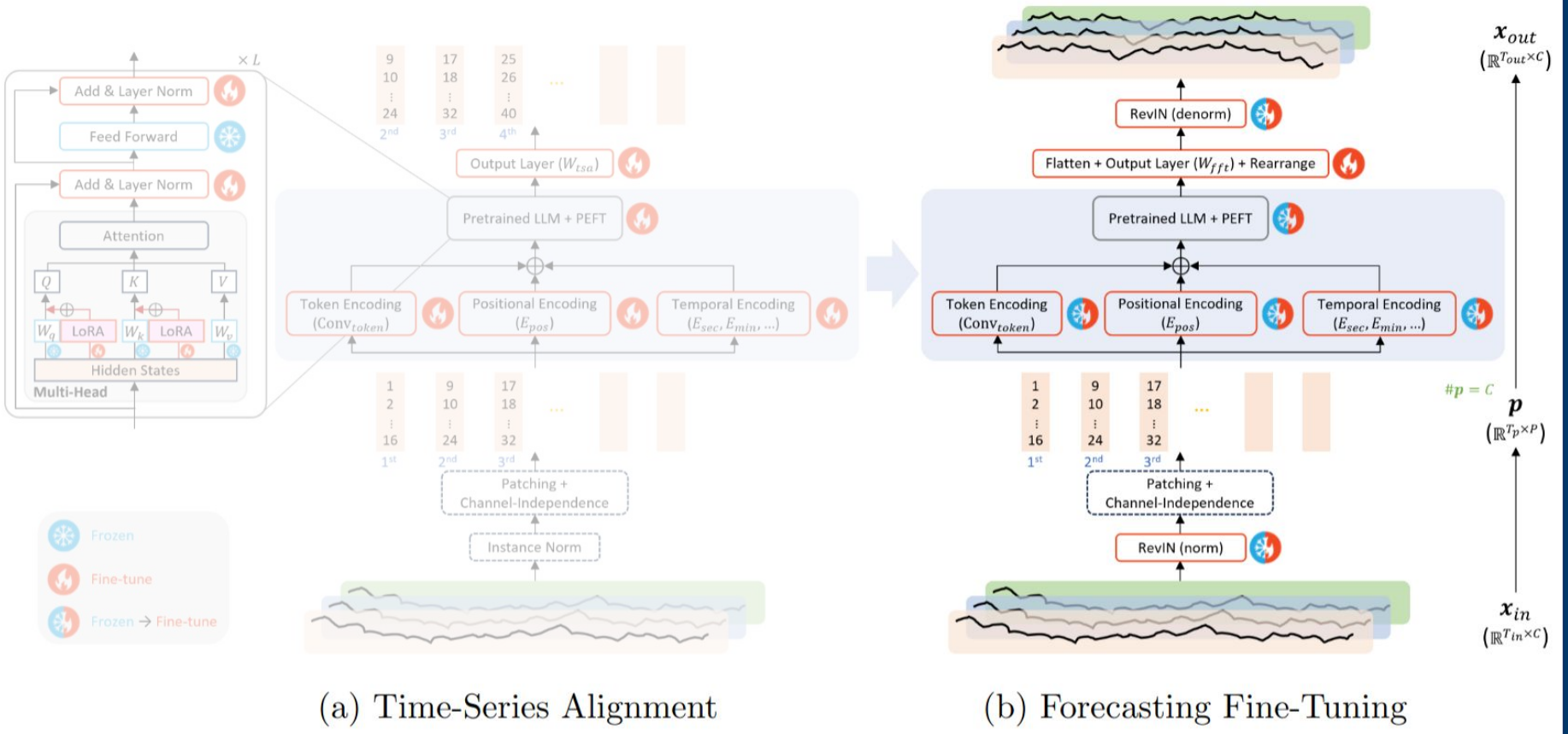
LLM4TS: Aligning Pre-Trained LLMs as Data-Efficient Time-Series Forecasters

ACM TIST 2025.



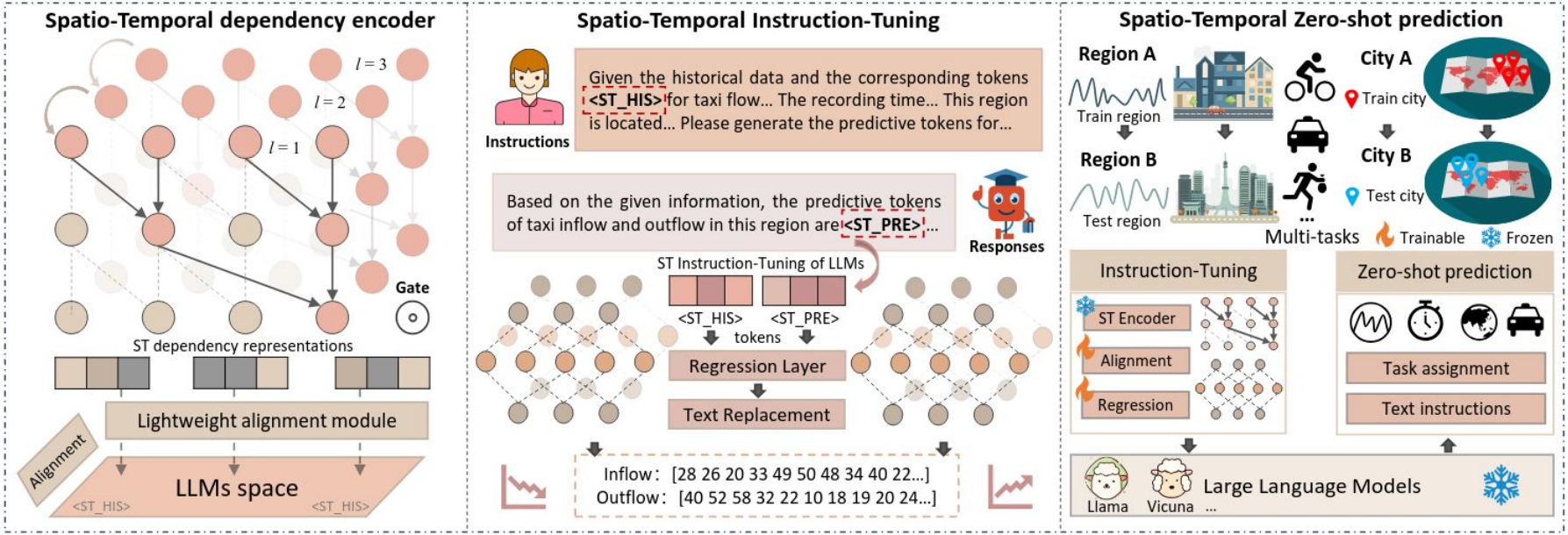
(a) Time-Series Alignment

(b) Forecasting Fine-Tuning



(a) Time-Series Alignment

(b) Forecasting Fine-Tuning



Zhonghang Li, Lianghao Xia, Jiabin Tang, Yong Xu, Lei Shi, Long Xia, Dawei Yin, and Chao Huang.

UrbanGPT: Spatio-Temporal Large Language Models.

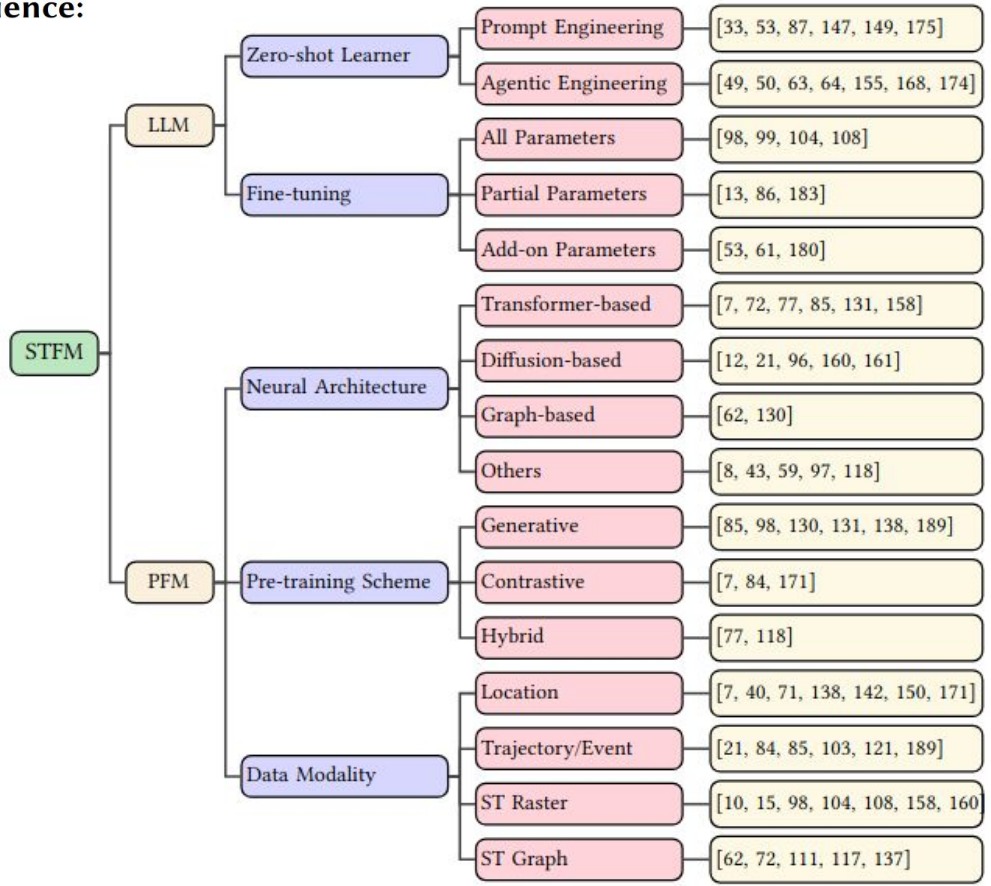
ACM SIGKDD (KDD '24)

ULB ECOLE POLYTECHNIQUE DE BRUXELLES

Foundation Models for Spatio-Temporal Data Science: A Tutorial and Survey

Yuxuan Liang[1], Haomin Wen[2,1], Yutong Xia[3], Ming Jin[4], Bin Yang[5], Flora Salim[6], Qingsong Wen[7], Shirui Pan[4], Gao Cong[8]

ACM SIGKDD 2025

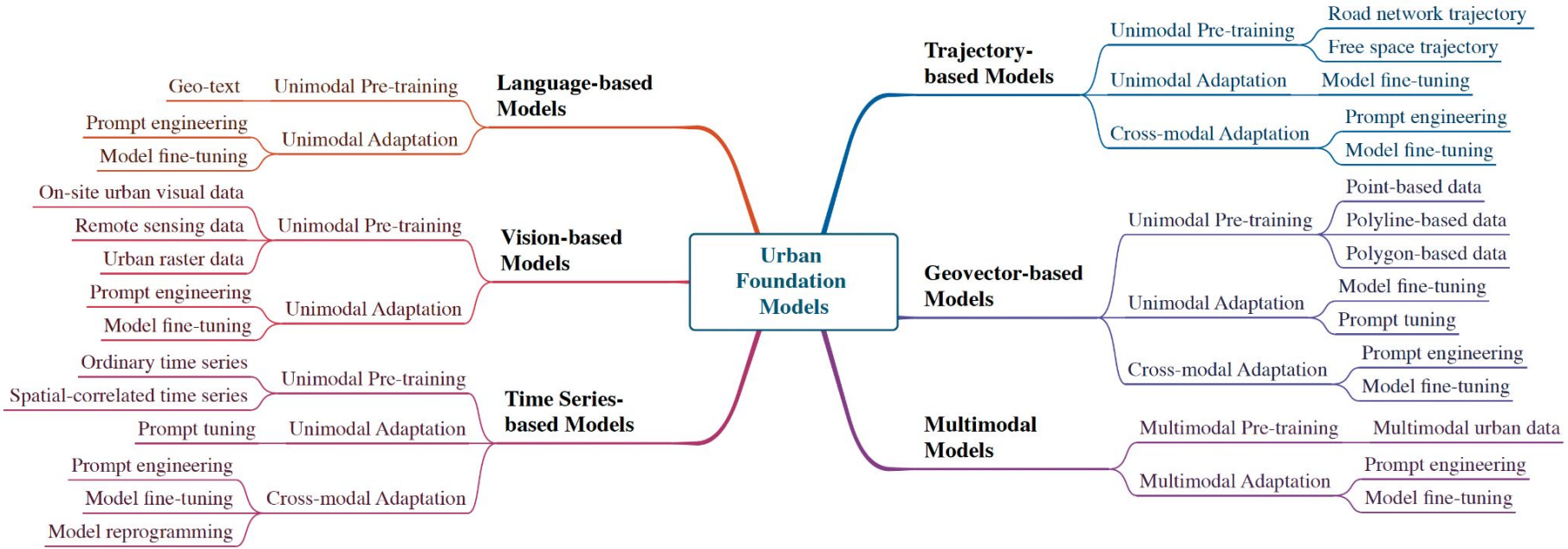


ULB ECOLE POLYTECHNIQUE DE BRUXELLES

Towards Urban General Intelligence: A Review and Outlook of Urban Foundation Models

Weijia Zhang, Jindong Han, Zhao Xu, Hang Ni, Tengfei Lyu, Hao Liu, *Senior Member, IEEE* and Hui Xiong, *Fellow, IEEE*

Arxiv 2024



Part 4
Conclusion

- Introduction to the main concepts in LLMs/FMs
- Illustrated how to adapt existing architectures to trajectories
  - Emphasis on the necessity of testing
  - Emphasis on the different possibilities in all components
- **Call for action**: the “mockup” trajectory foundation model is available on github
  - Modify/adapt combine the different components
  - Use as collaborative community project for further research

Thank You !

ECOLE
POLYTECHNIQUE
DE BRUXELLES

A super cool title for the best presentation ever to exist.

Presented by Roméo, Juliette
And all the Italian dramas.

A section separator

ULB ECOLE POLYTECHNIQUE DE BRUXELLES

A slide title which should never be too long

Some **explanation** about the science of drinking tea.
A list of important elements:

- Super
- Tea