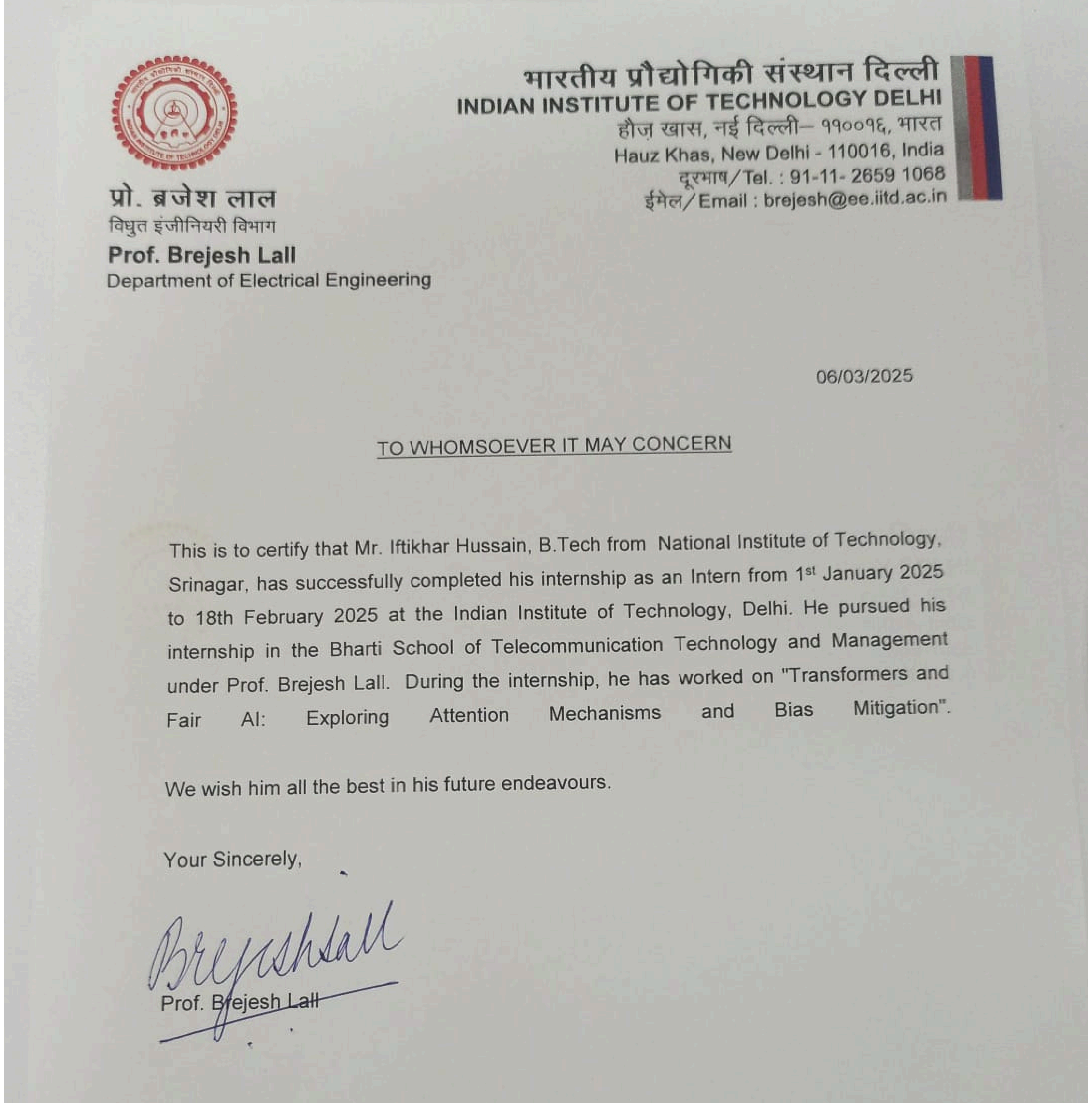
Transformer and Fair Ai: Exploring attention mechanism and bais mitigation

Name: Iftikhar Hussain
Enroll: 2022BCSE053



भारतीय प्रौद्योगिकी संस्थान दिल्ली
INDIAN INSTITUTE OF TECHNOLOGY DELHI
हौज़ खास, नई दिल्ली– ११००१६, भारत
Hauz Khas, New Delhi - 110016, India
दूरभाष/Tel. : 91-11- 2659 1068
ईमेल/Email : brejesh@ee.iitd.ac.in

प्रो. ब्रजेश लाल
विद्युत इंजीनियरी विभाग
Prof. Brejesh Lall
Department of Electrical Engineering

06/03/2025

TO WHOMSOEVER IT MAY CONCERN

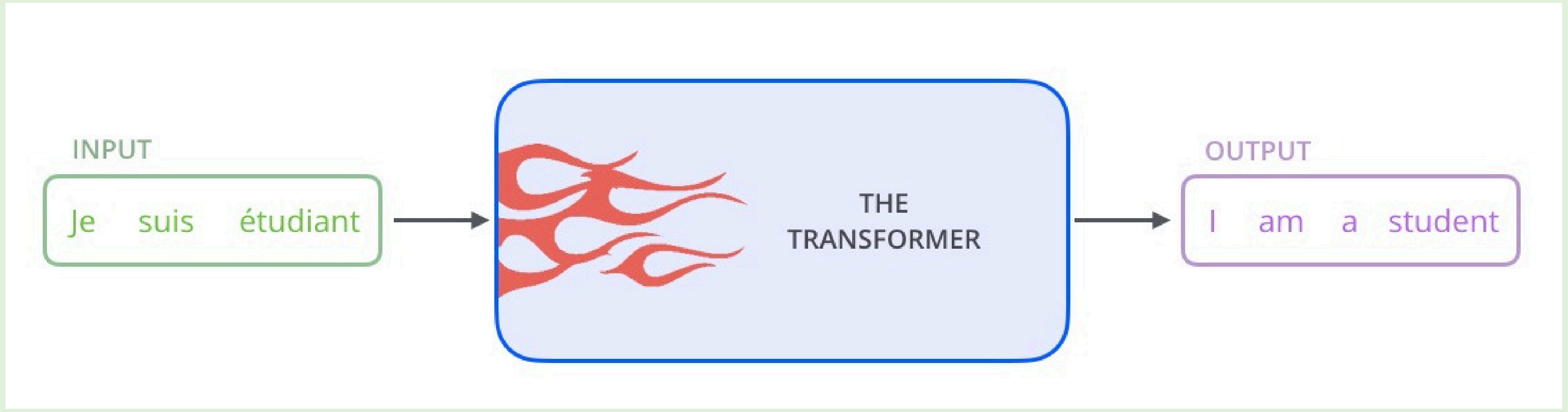
This is to certify that Mr. Iftikhar Hussain, B.Tech from National Institute of Technology, Srinagar, has successfully completed his internship as an Intern from 1st January 2025 to 18th February 2025 at the Indian Institute of Technology, Delhi. He pursued his internship in the Bharti School of Telecommunication Technology and Management under Prof. Brejesh Lall. During the internship, he has worked on "Transformers and Fair AI: Exploring Attention Mechanisms and Bias Mitigation".

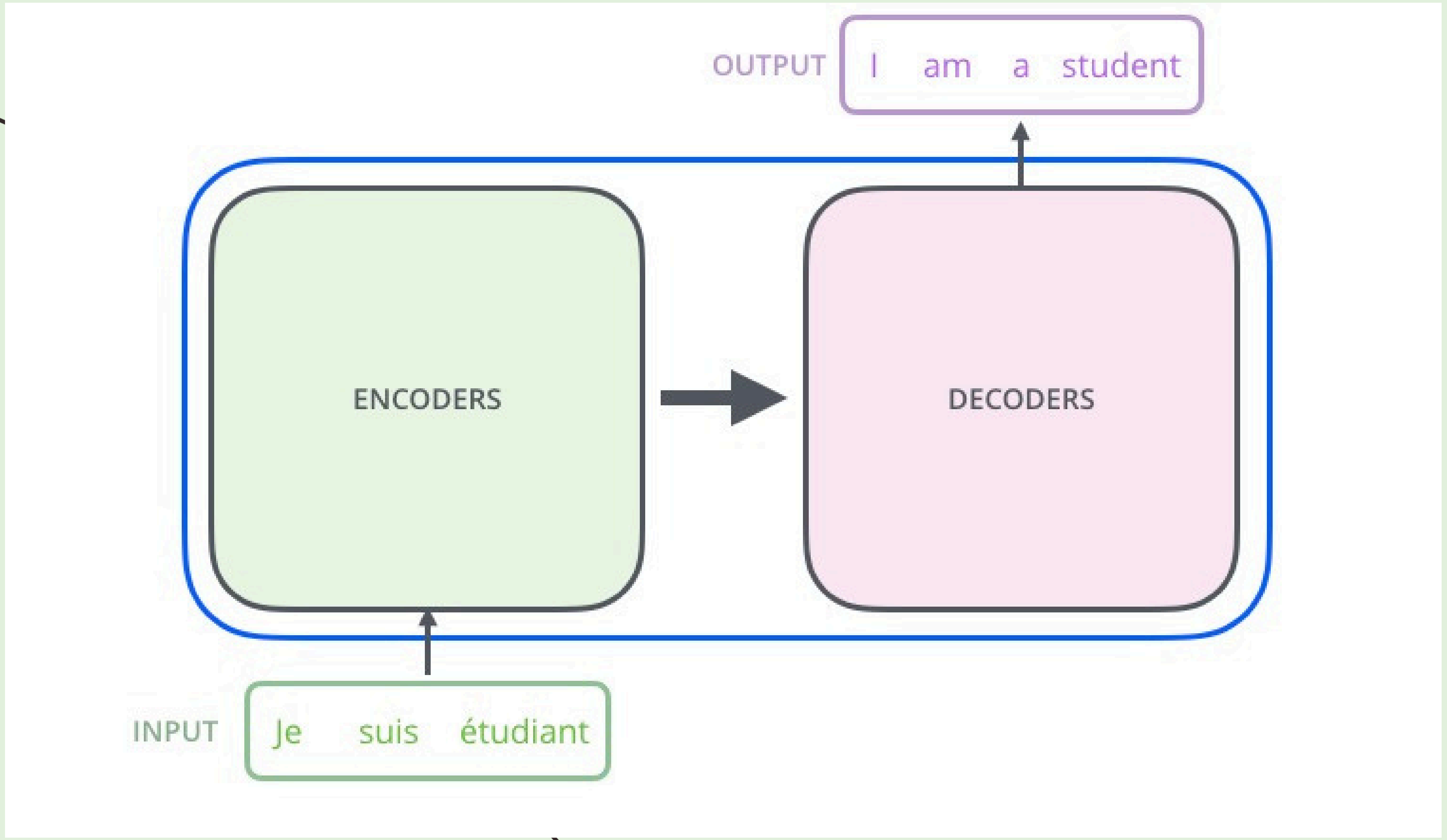
We wish him all the best in his future endeavours.

Your Sincerely,

[illegible]

Prof. Brejesh Lall

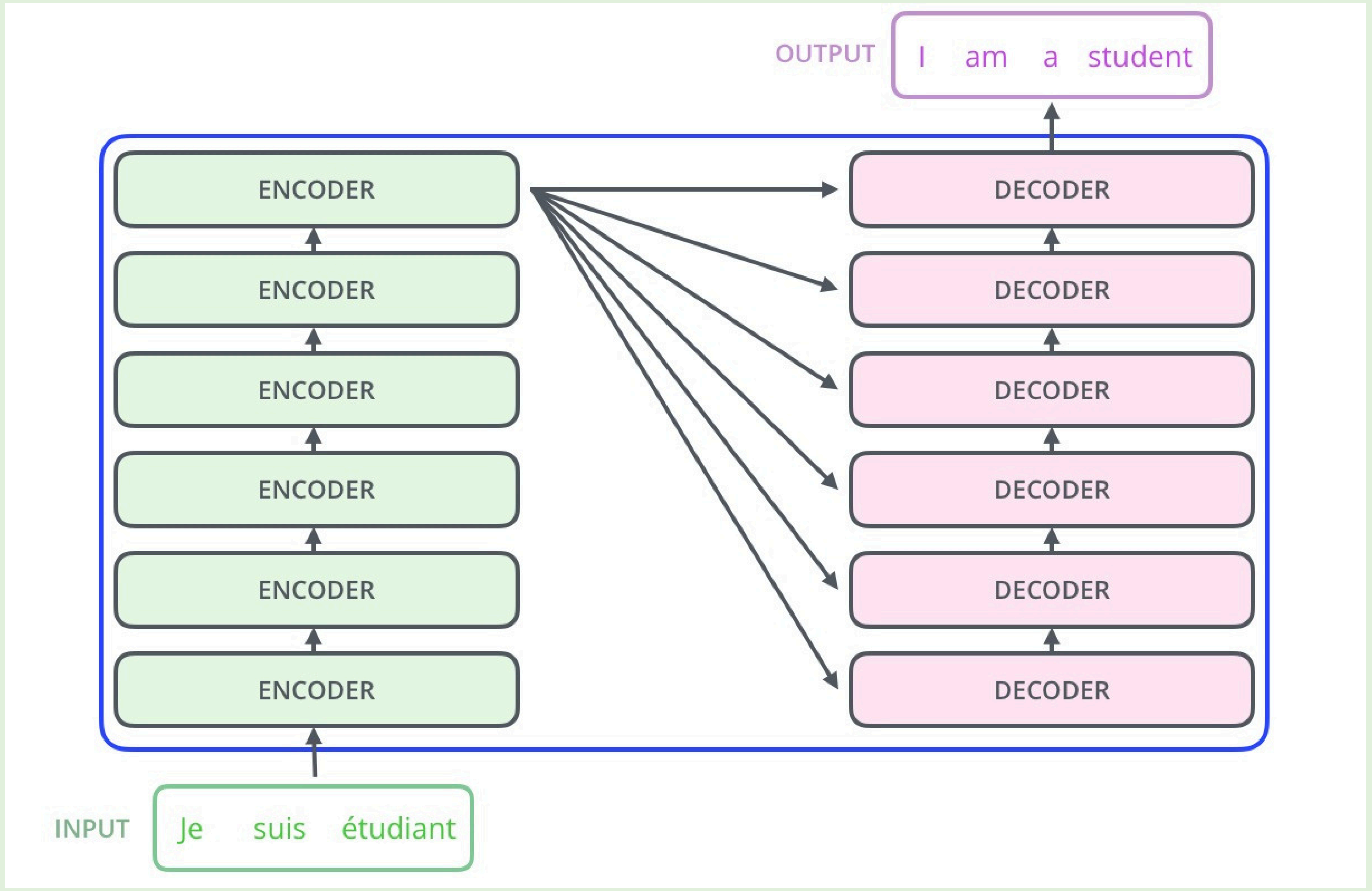


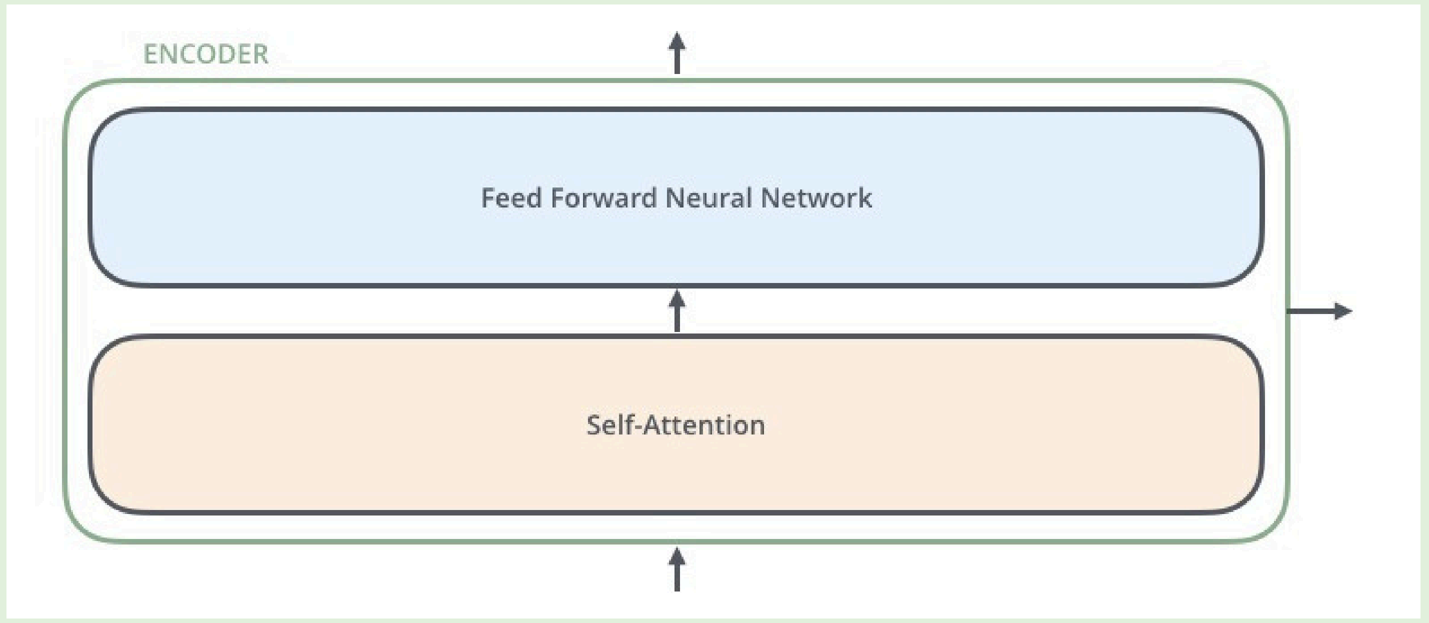


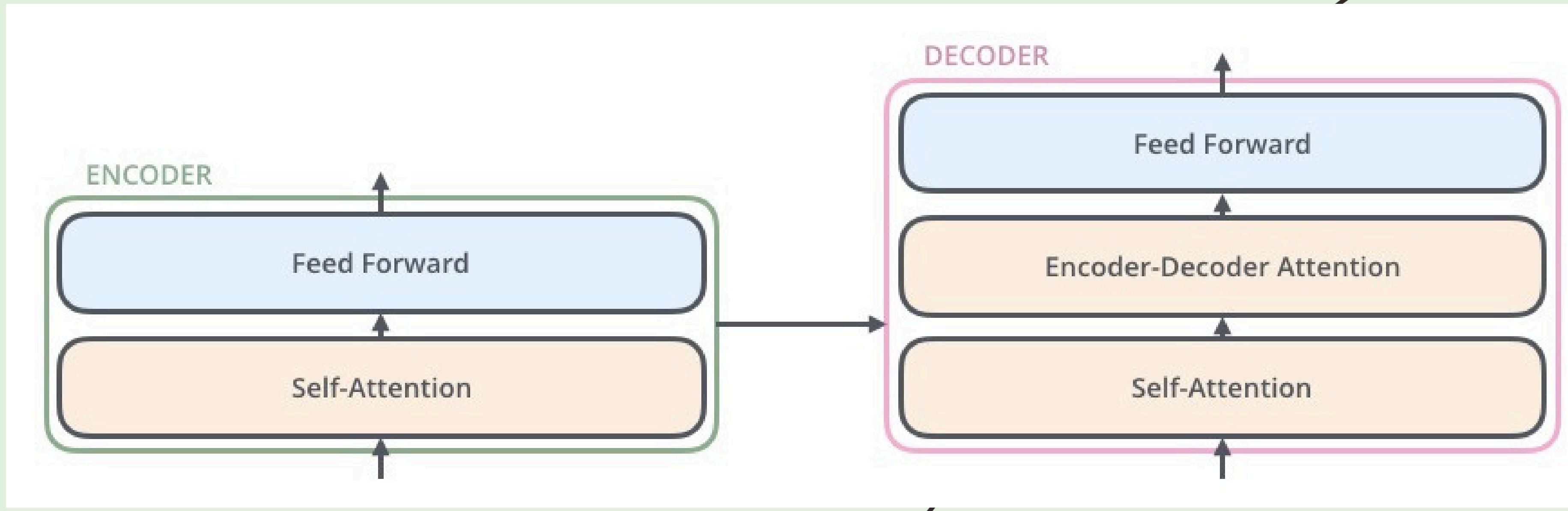
Introduction to Transformers

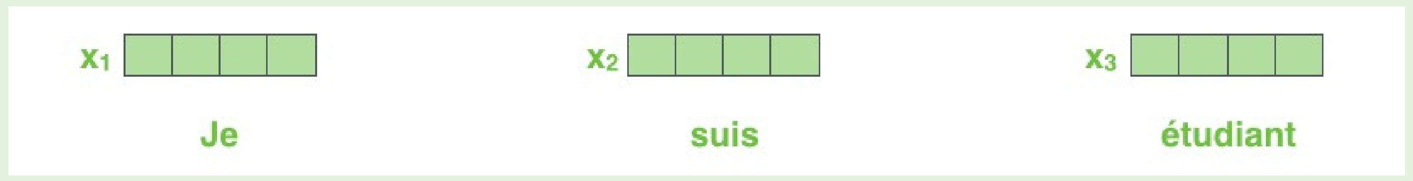
The Transformer is a neural network architecture introduced in 2017 that revolutionized NLP. Unlike previous models, it relies on self-attention mechanisms rather than recurrent layers. This allows for parallelization and improved performance on various language tasks.

The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.





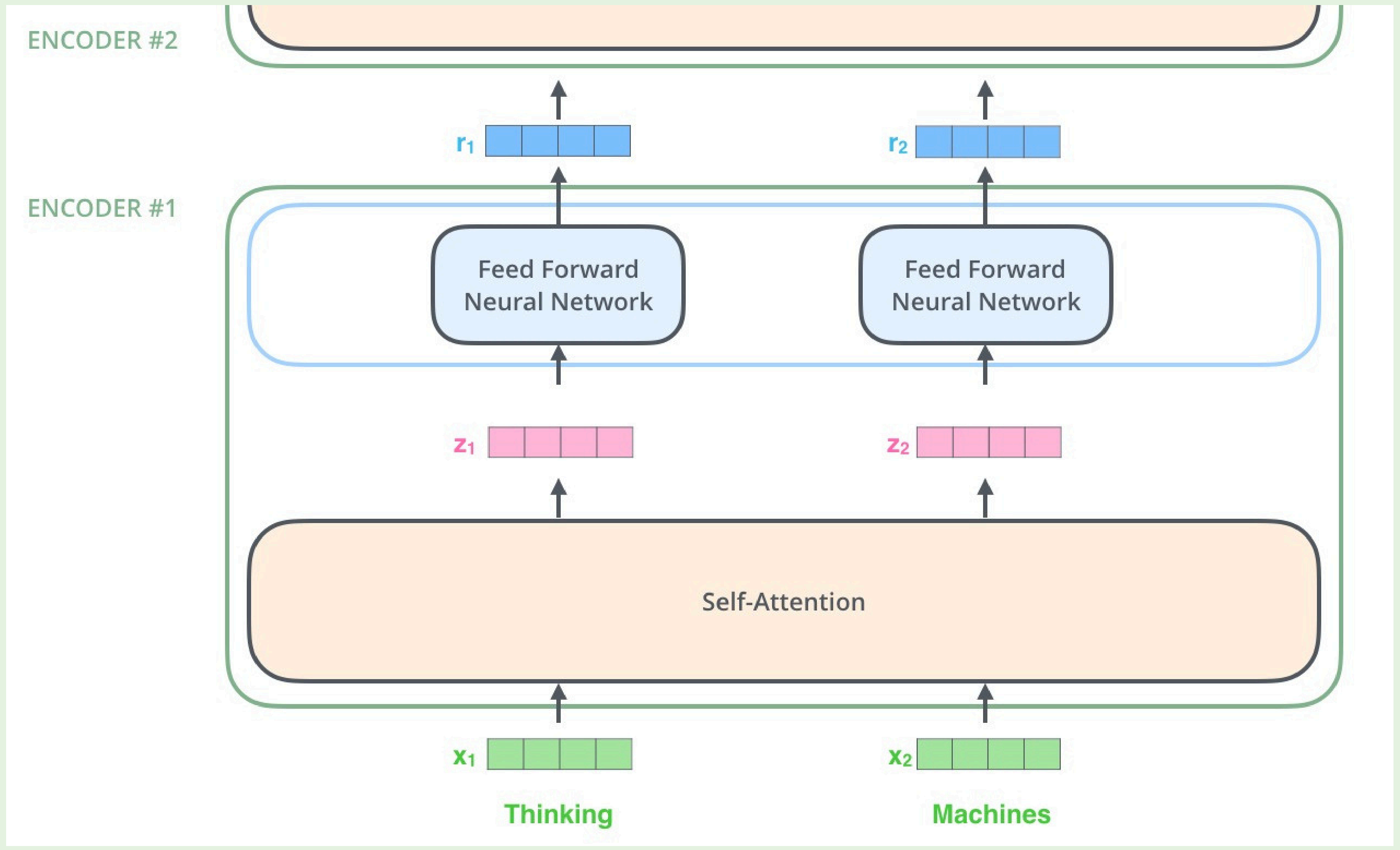




Each word is embedded into a vector of size 512.
We'll represent those vectors with these simple boxes.

Now We're Encoding!

As we've mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately

Self-Attention in Detail

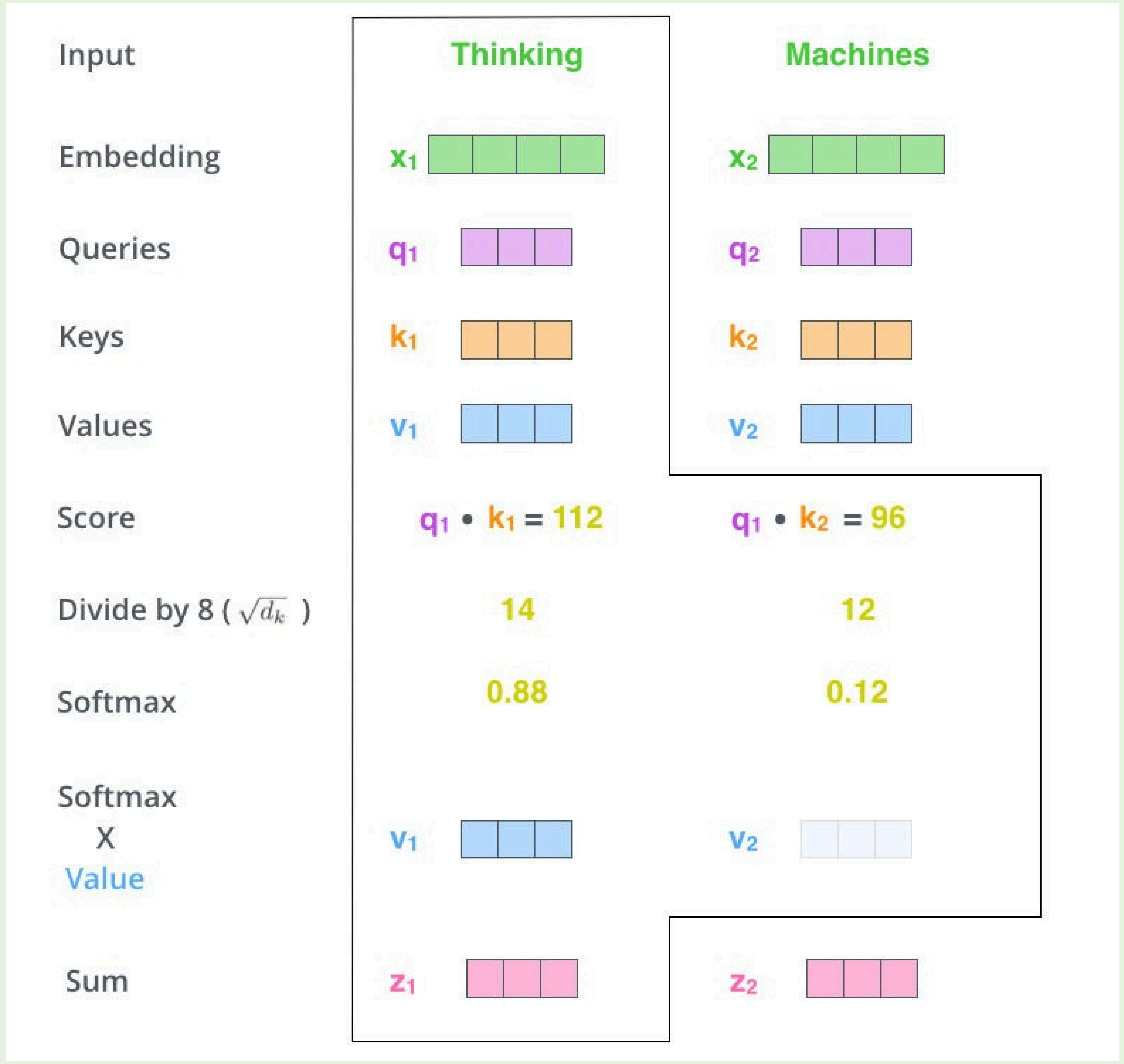
The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training proces

The second step in calculating self-attention is to calculate a score. Say we're calculating the self- attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

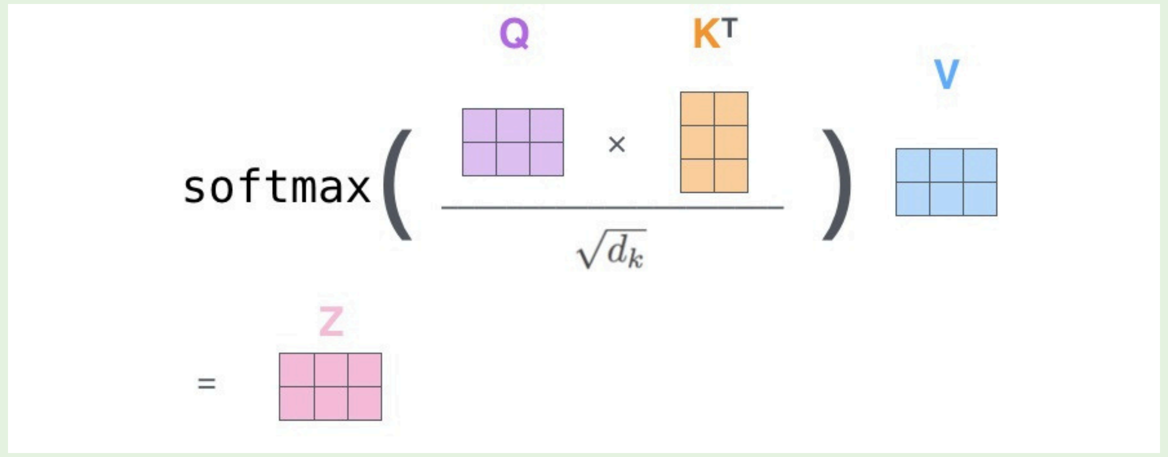
The third and fourth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words

The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

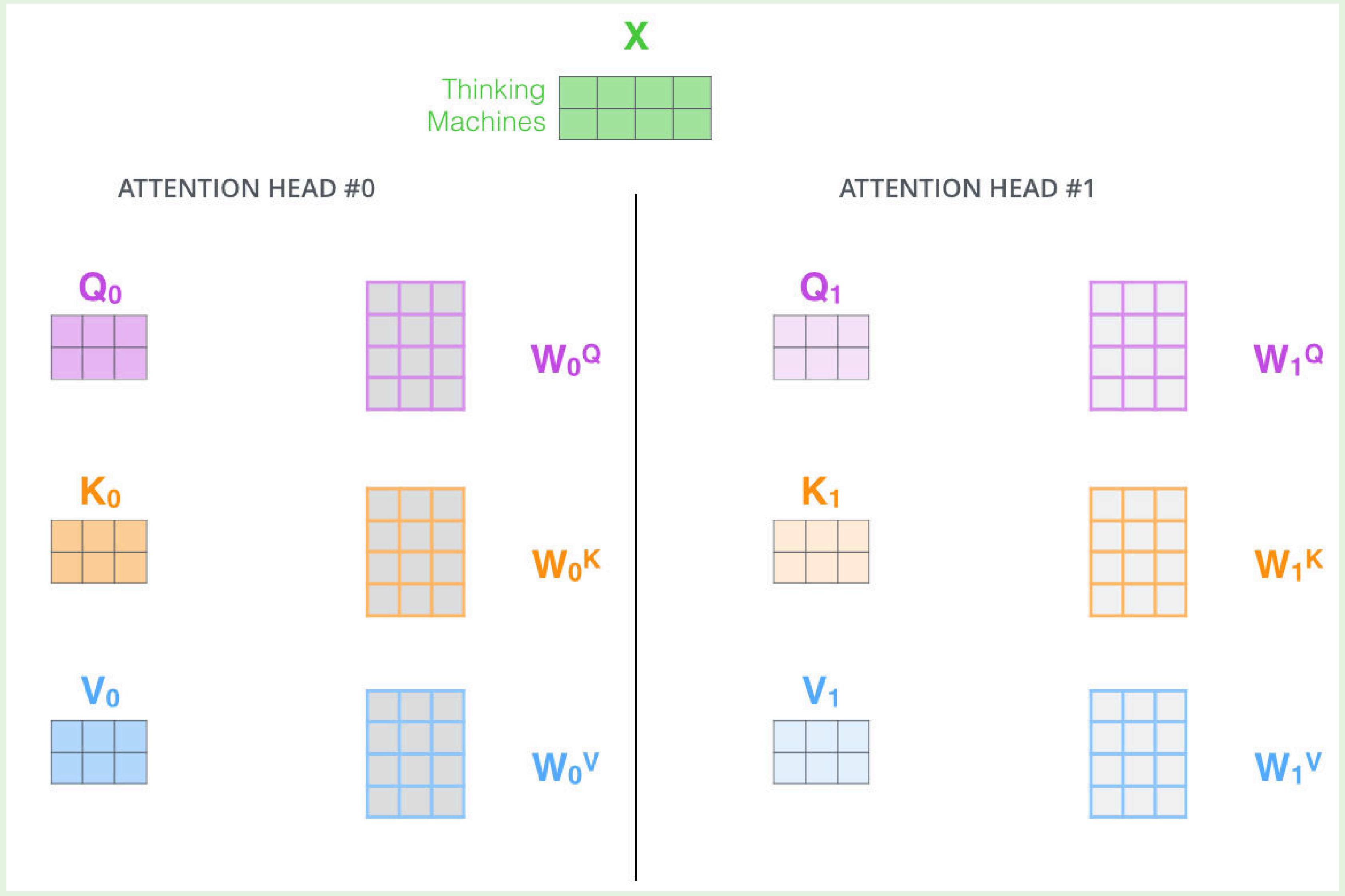


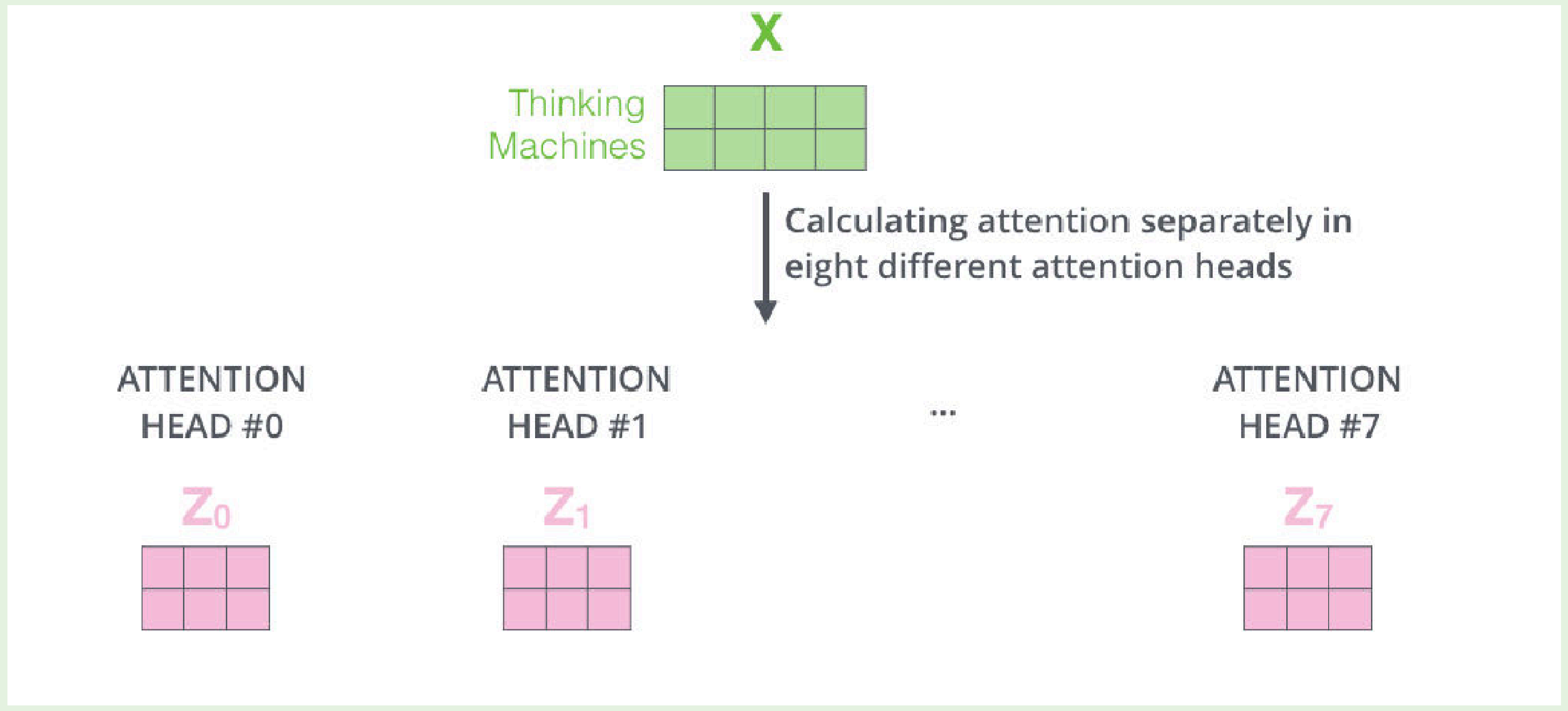
Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



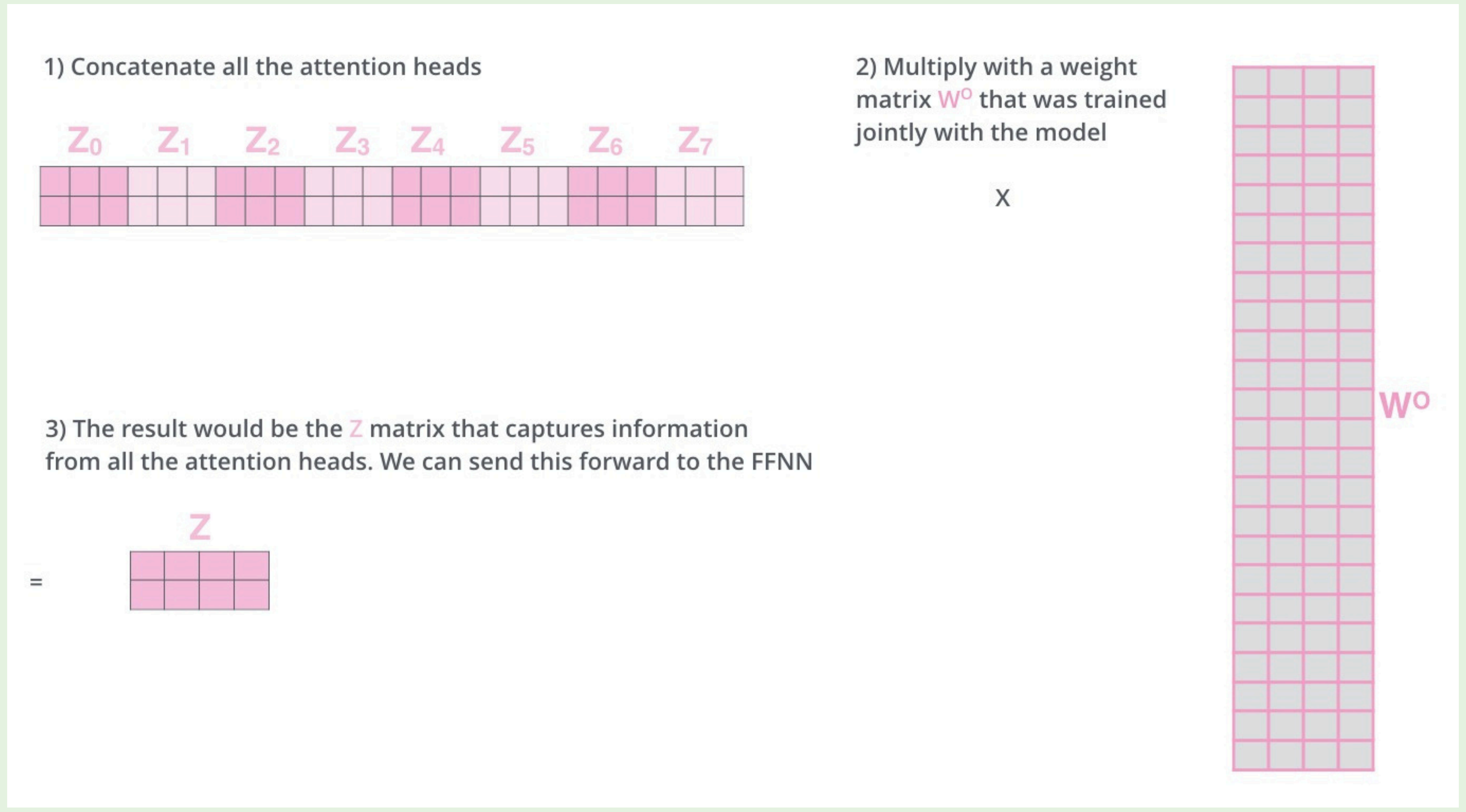
"Multi-headed" attention

- It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to. It gives the
- attention layer multiple "representation subspaces". As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

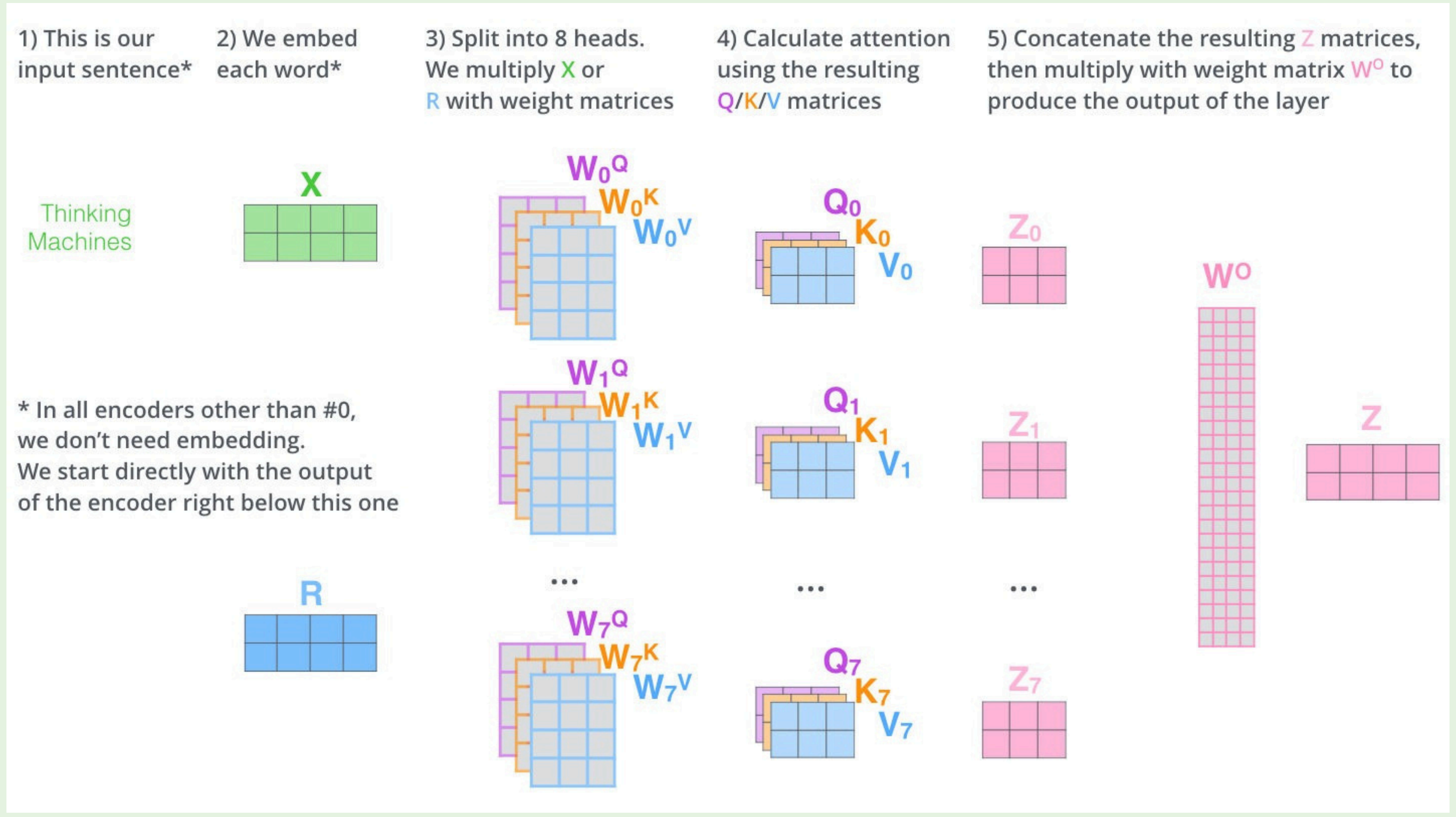


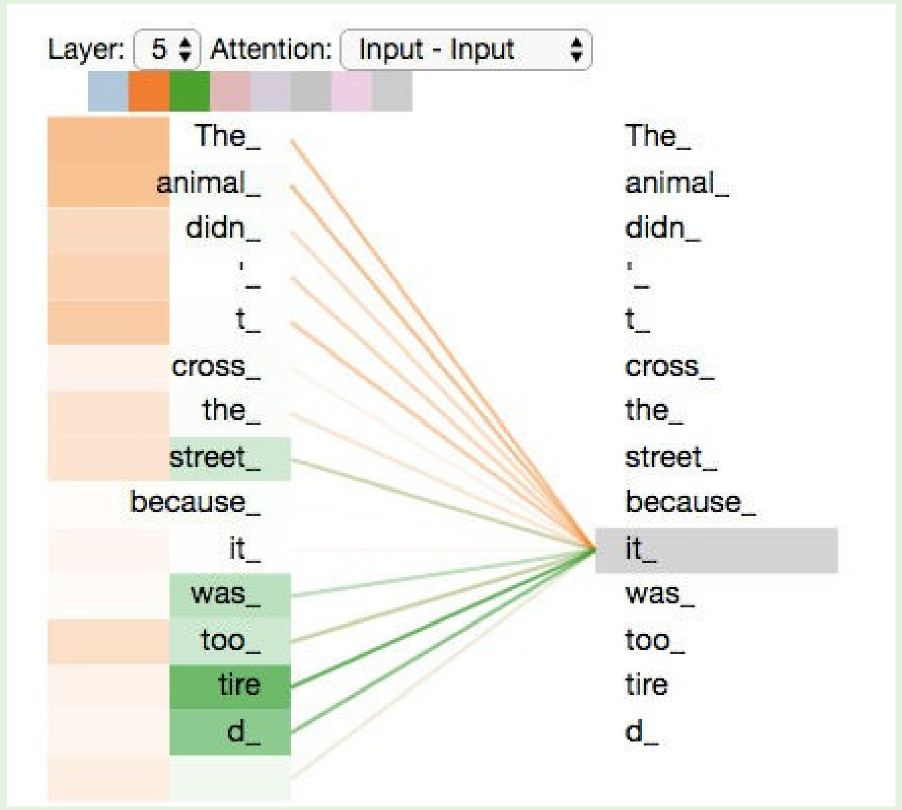


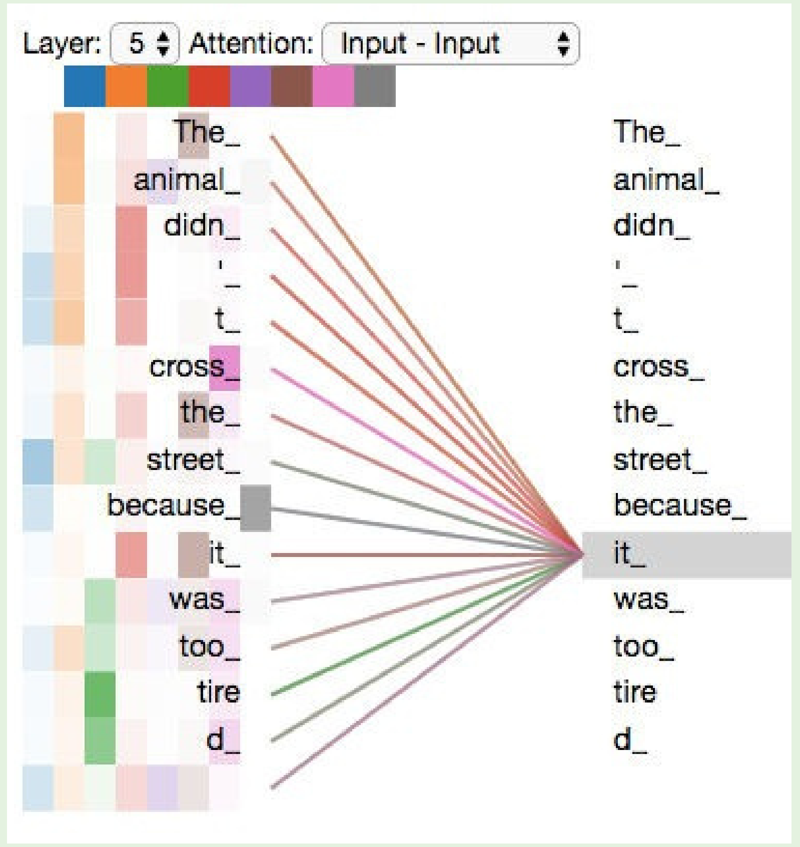
- This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.


1) Concatenate all the attention heads
Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7
2) Multiply with a weight matrix W^O that was trained jointly with the model
x
W^O
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN
=
Z

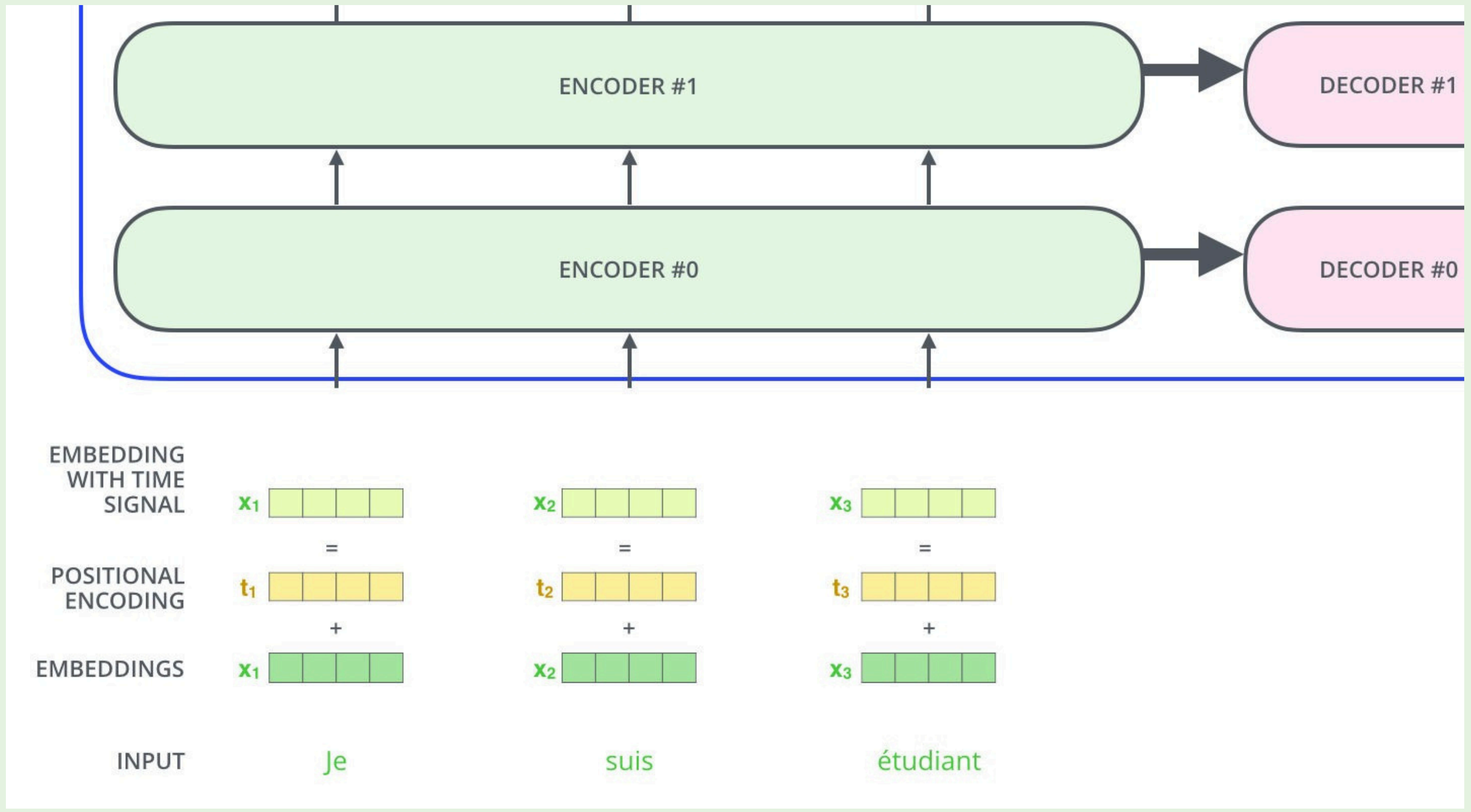
- That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place


1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0 Z1 Z7
W^O
Z
...


Layer: 5 Attention: Input - Input
The_ animal_ didn_ '_ t_ cross_ the_ street_ because_ it_ was_ too_ tire d_
The_ animal_ didn_ '_ t_ cross_ the_ street_ because_ it_ was_ too_ tire d_


Layer: 5 Attention: Input - Input
The_ animal_ didn_ '_ t_ cross_ the_ street_ because_ it_ was_ too_ tire d_
The_ animal_ didn_ '_ t_ cross_ the_ street_ because_ it_ was_ too_ tire d_

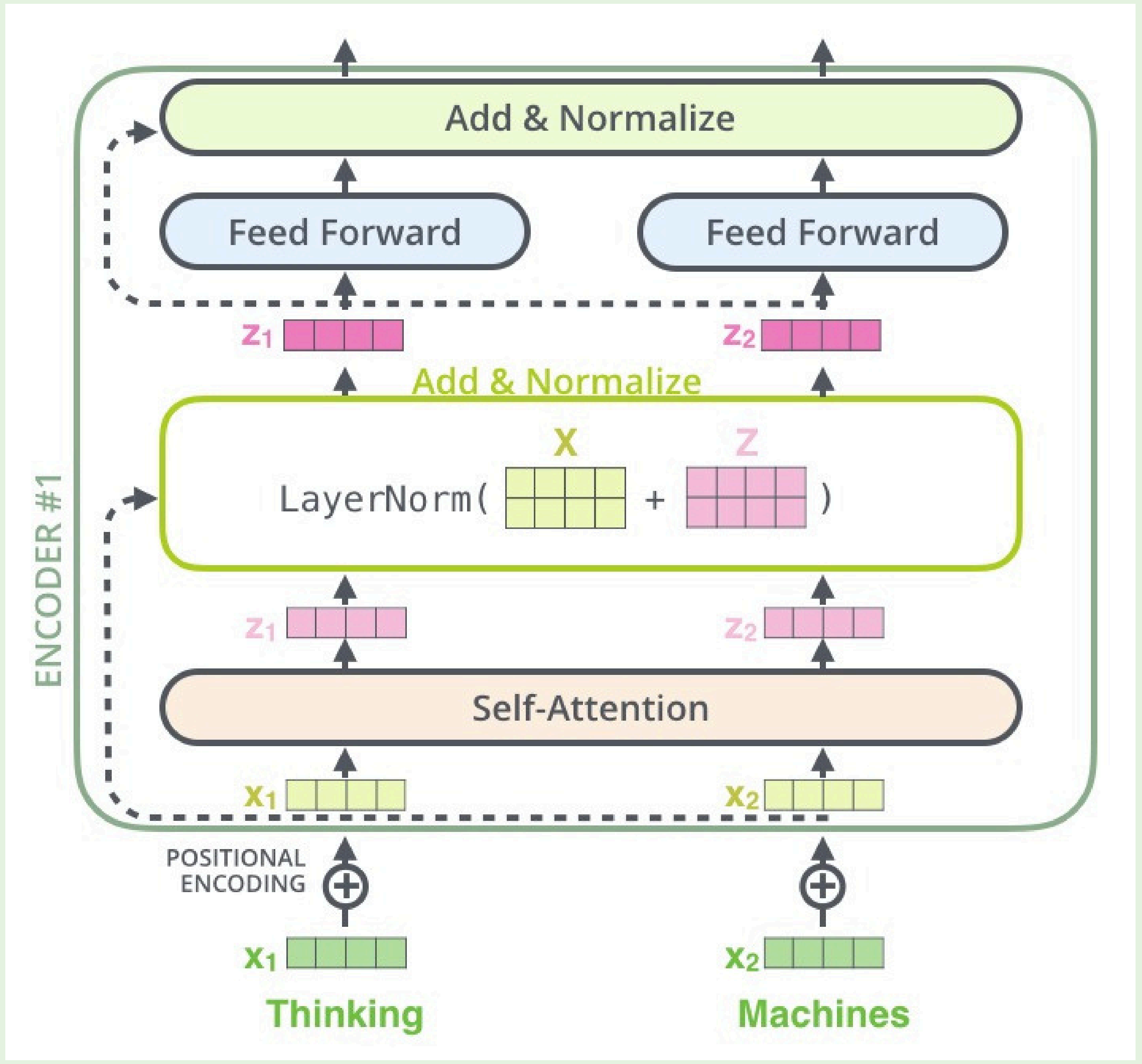
Representing The Order of The Sequence Using Positional Encoding



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

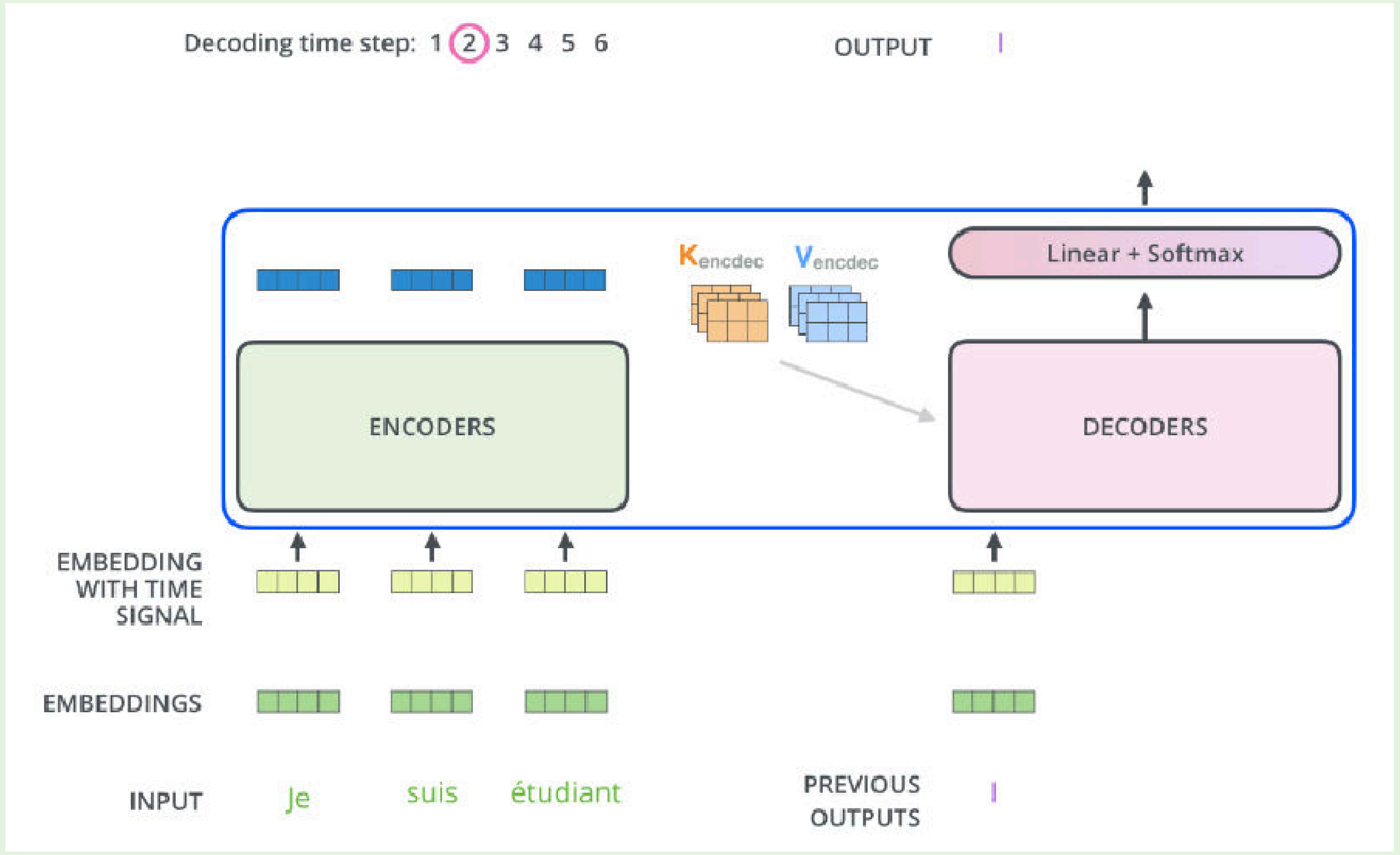
The Residuals

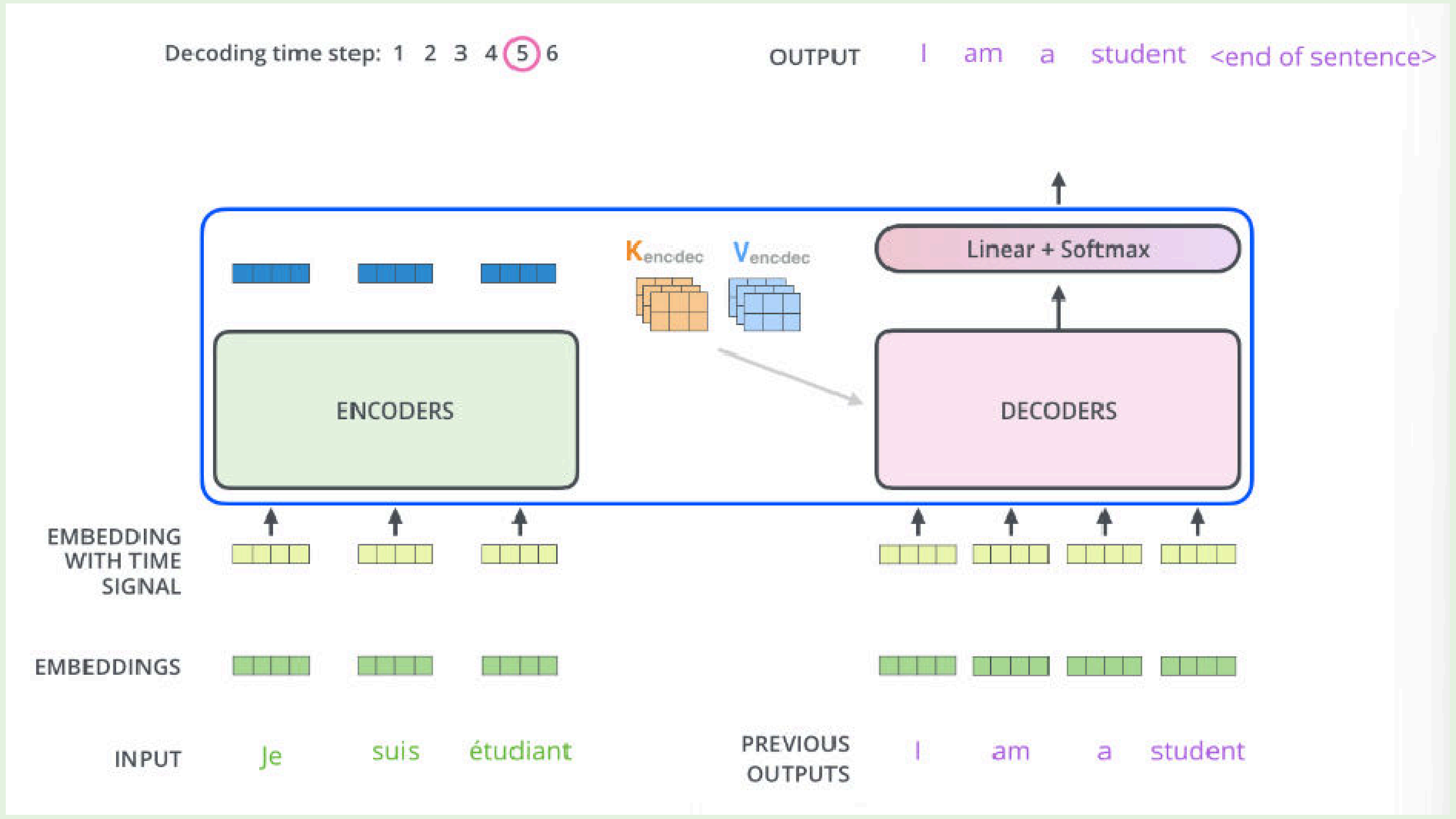
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by alayer-normalizationstep



The Decoder Side

Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.





In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to) before the softmax step in the self-attention calculation.

AN IMAGE IS WORTH 16X16 WORDS

TRANSFORMERS FOR I MAGE RECOGNITION AT SCALE

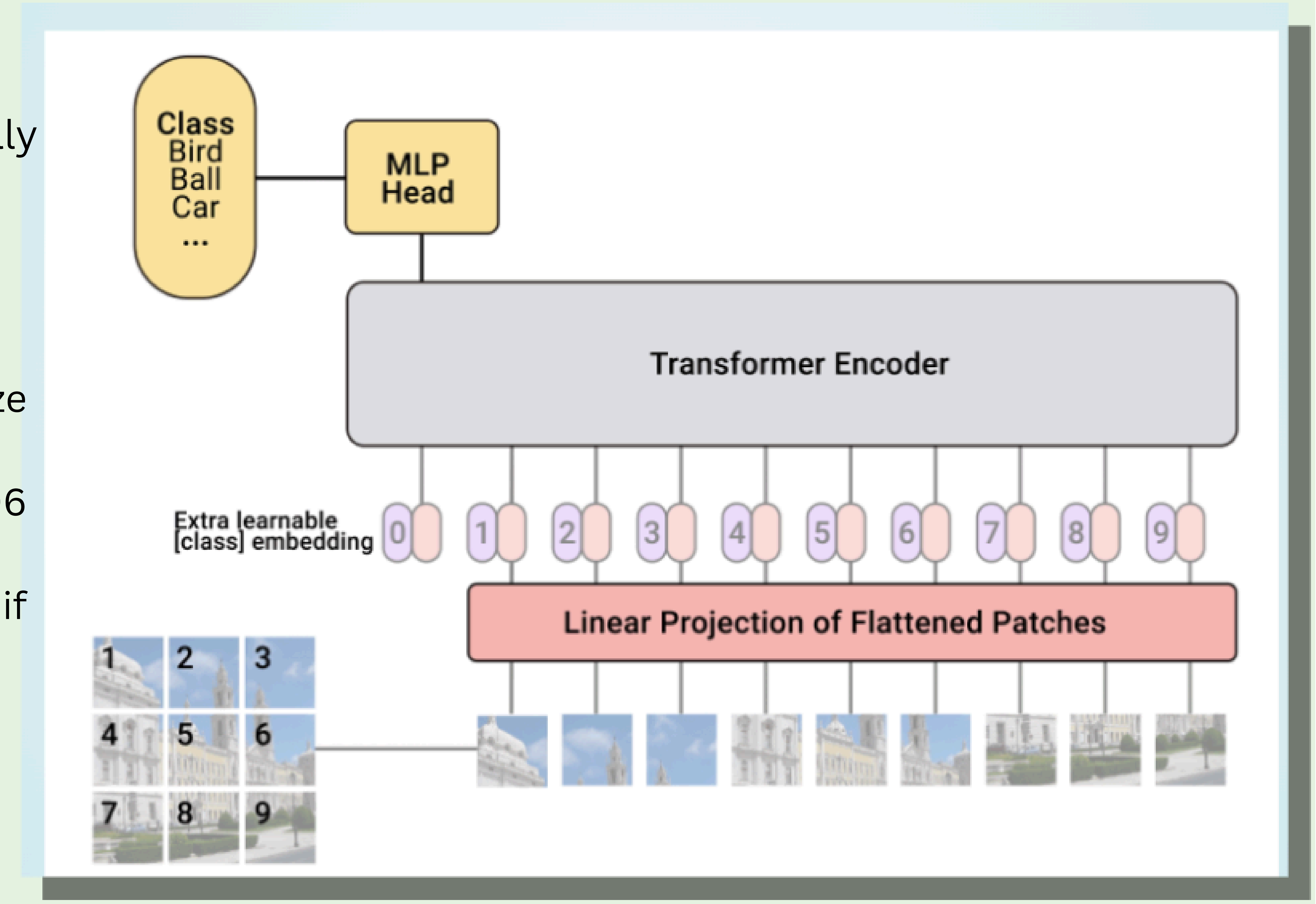
An image of size 16×16 patches is treated like a “sentence” with 256 (16×16 = 256) visual words (tokens).
Vision Transformers adapt the Transformer architecture, originally developed for NLP (e.g., BERT, GPT), for image recognition.

✅ Step 1: Image to Patches

- The input image (e.g., 224×224 pixels) is divided into fixed-size patches, such as 16×16 pixels.
- For 224×224 image with 16×16 patches → you get 14×14 = 196 patches.
- Each patch is flattened into a vector (e.g., 16×16×3 = 768 values if RGB).

✅ Step 2: Linear Projection

- Each patch vector is linearly projected into an embedding vector (like word embeddings in NLP).
- This gives a sequence of embeddings — treated like a sentence of tokens.


Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Extra learnable [class] embedding
0
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches
1
2
3
4
5
6
7
8
9

✅ Step 3: Add Positional Embeddings

- Since Transformers are position-agnostic, positional information is added so the model knows the location of each patch.

✅ Step 4: Transformer Encoder

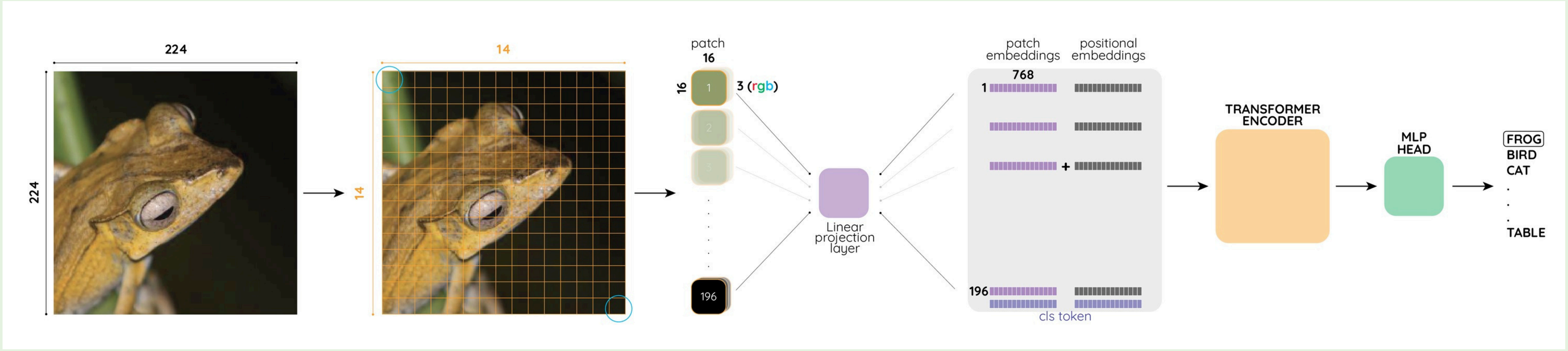
- The sequence of patch embeddings goes through standard Transformer layers:

1. Multi-head self-attention
2. Feed-forward layers

- The model learns relationships between patches — similar to how GPT learns context between words.

✅ Step 5: Classification / Output

- A special token [CLS] is prepended to the sequence.
- After processing, its final embedding is used for classification (like predicting image labels).



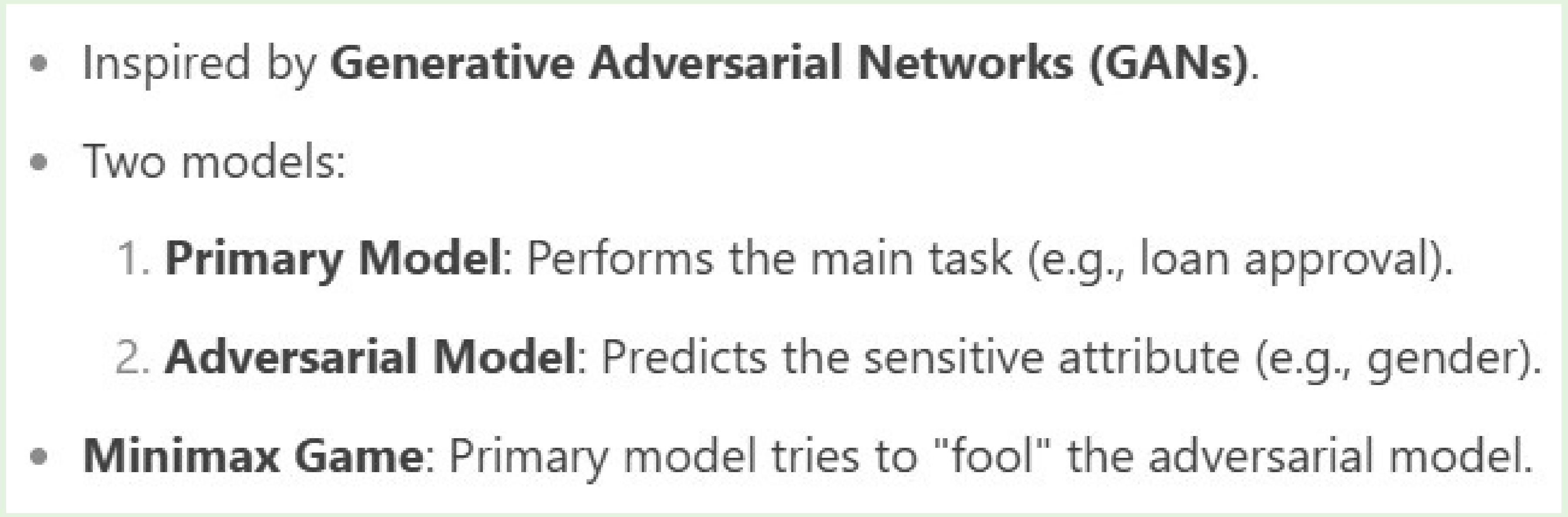
“Mitigating Unwanted Bias with Adversarial Learning”

In machine learning models — especially in NLP, vision, or tabular models — we often face bias:

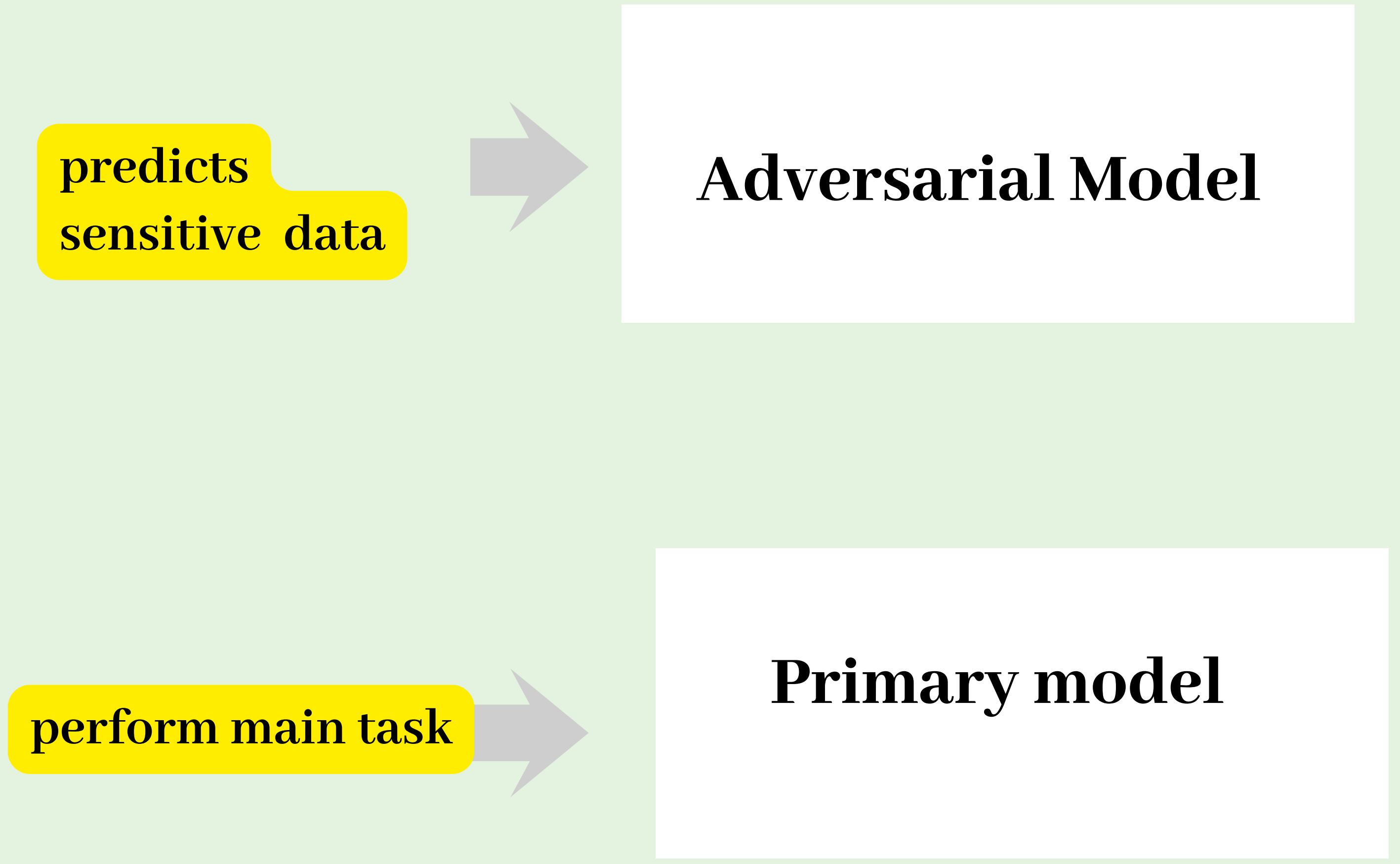
- Gender bias (e.g., associating "doctor" with male)
- Racial bias
- Age or disability bias, etc.

These biases can lead to unfair or discriminatory decisions, especially in sensitive applications (hiring, lending, law enforcement, etc.).

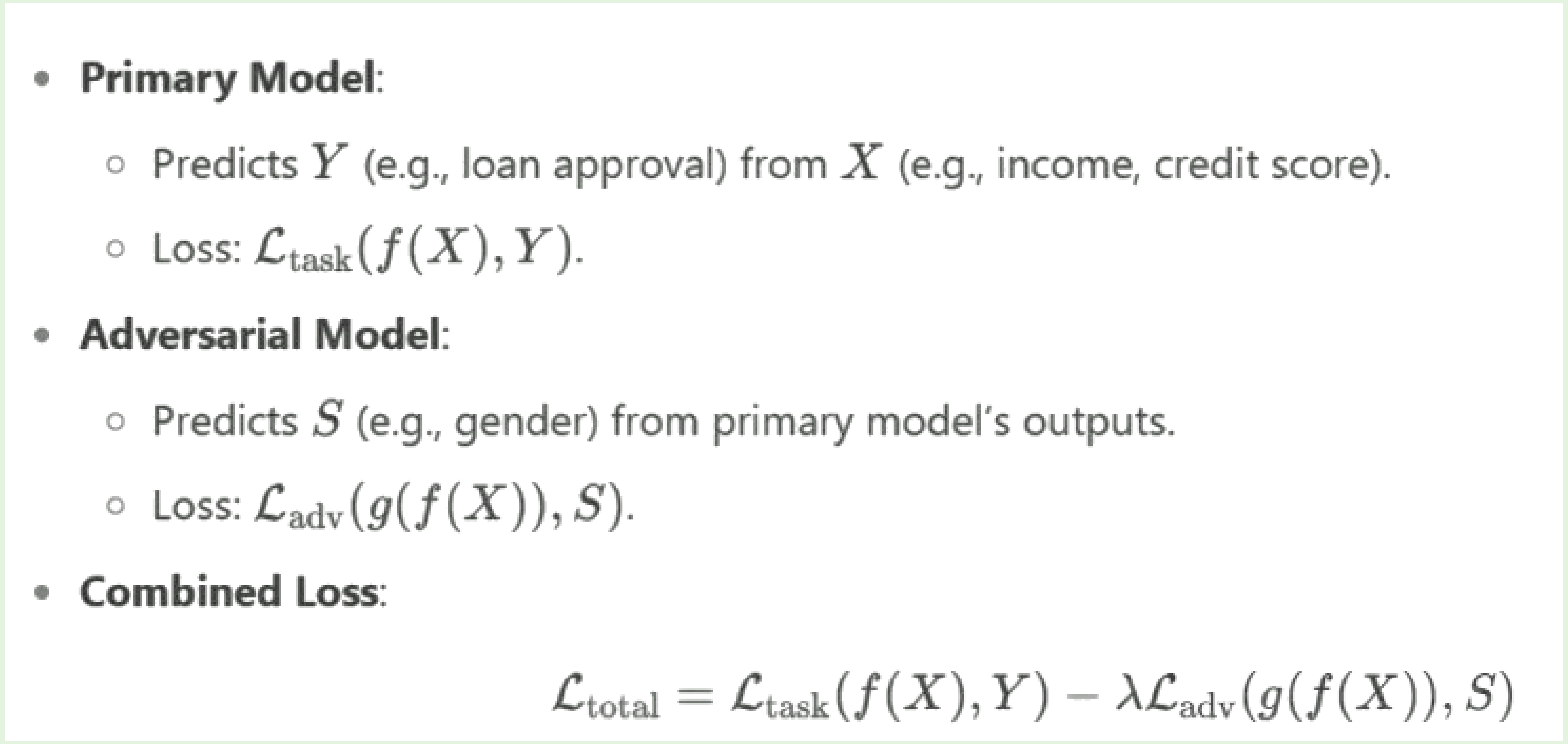
Title: Adversarial Learning Overview



- Inspired by **Generative Adversarial Networks (GANs)**.
- Two models:
  1. **Primary Model**: Performs the main task (e.g., loan approval).
  2. **Adversarial Model**: Predicts the sensitive attribute (e.g., gender).
- **Minimax Game**: Primary model tries to "fool" the adversarial model.



Methodology



- **Primary Model**:
  - Predicts Y (e.g., loan approval) from X (e.g., income, credit score).
  - Loss: $\mathcal{L}_{\text{task}}(f(X), Y)$.
- **Adversarial Model**:
  - Predicts S (e.g., gender) from primary model's outputs.
  - Loss: $\mathcal{L}_{\text{adv}}(g(f(X)), S)$.
- **Combined Loss**:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}}(f(X), Y) - \lambda \mathcal{L}_{\text{adv}}(g(f(X)), S)$$

Primary model learns to "hide" information about sensitive data. Adversarial model ensures SS is not encoded in primary model’s outputs.
Outcome: Fair predictions invariant to sensitive attributes.

FairDisco:Ai in Dermatology

🚨 Problem:

In skin disease AI (like diagnosing acne or melanoma from photos), models often work better for light skin and worse for dark skin.

This is because:

- Datasets like Fitzpatrick17k have more images of light skin tones (Types I–III).
- Fewer images of dark skin tones (Types IV–VI), so the model learns to depend on skin tone, creating bias.

Goal of FairDisCo

Make the model:

- Predict the skin disease accurately ✅
- But not be influenced by skin tone ❌

So the model should:

- Be fair to all skin tones
- Give similar accuracy across all subgroups

Step 1: Input

Each training image has:

- A photo of skin with some disease
- A label (e.g., “acne”, “eczema”)
- A skin tone type (Fitzpatrick Type I–VI)

⚙️ Step 2: Feature Extraction

- The image goes through a neural network (like ResNet).
- This creates a feature vector — a mathematical summary of the image.

Then, this feature is split into two parts:

- Content embedding → disease information 🦠
- Subgroup embedding → skin tone info 🧑🏾

🎯 Step 3: Main Classifier

- The content embedding is used to predict the disease (classification task).
- It tries to learn only disease-related features, ignoring skin tone.

🤖 Step 4: Adversarial Skin Tone Classifier

- A second model (adversary) tries to predict skin tone from the content embedding.
- We train the main model to fool this adversary:

So the content embedding does not contain any skin tone info.

🧲 Step 5: Contrastive Learning

FairDisCo also uses contrastive learning to make the model better:

- It pulls together features of images with the same disease, even if their skin tone is different.
- It pushes apart images of different diseases.

This helps the model focus on disease features, not on how the skin looks.

🧪 Final Result:

- The model performs well on all skin tones.
- The accuracy gap between light and dark skin reduces a lot.
- The model is more fair and trustworthy for all users.