In the context of input embedding in a Transformer, a vector representation means that each word (or subword/token) from the input sequence is mapped to a fixed-length numerical vector—typically a high-dimensional dense vector.

Here's a breakdown:

What is a Vector Representation?

A vector is simply an array of numbers (like `[0.1, -0.3, 0.7, ..., 0.05]`) that represents the semantic meaning of a token in a form that a neural network can understand.

- For example, if the embedding size is 512, each token is mapped to a 512-dimensional vector.
- This vector captures various linguistic properties (like syntax and semantics) learned during training.

Why Use It?

- Machine learning models can't work with raw text (e.g., "apple"). They need numerical input.
- Embedding layers learn to assign similar vectors to semantically similar words, so "cat" and "dog" might have vectors that are closer to each other than to "car" or

“computer”.

How Is It Done?

- A lookup table (called the embedding matrix) is initialized with random values and learned during training.
- Each token’s index is used to fetch its corresponding vector from this table.

Example:

Suppose the sentence is:

```
Input: ["I", "am", "a", "student"]
```

After tokenization and embedding:

```
"I"       → [0.12, -0.45, 0.63, ..., 0.08]
"am"      → [0.03, 0.91, -0.25, ..., -0.12]
...
```

These are then passed into the Transformer layers.

Q. From this page, embedding size is 512, means ?

When we say embedding size is 512, it means that each input token is represented by a vector of 512 numbers (dimensions) after the embedding layer.

Let’s break this down:

 What it means

- You have a vocabulary (say 50,000 tokens) → words, subwords, or characters.
- The embedding layer maps each token in that vocabulary to a 512-dimensional dense vector.

So:

- “cat” → [0.12, -0.33, 0.78, …, 0.05] → vector length = 512
- “dog” → [-0.41, 0.29, 0.55, …, -0.27] → vector length = 512

Even though “cat” and “dog” are very different words, after embedding they become points in a 512-dimensional space, and their relative positions (distance, angle) can capture meaningful relationships.

 Why 512?

- It’s a design choice—large enough to capture complex patterns, but small enough to train efficiently.
- Vaswani et al. (2017) chose 512 as a balance between performance and computation.
- Other models like BERT or GPT often use 768, 1024, or even 4096 depending on their scale.

 Summary

> *Embedding size = 512 → every token gets represented as a vector with 512 numerical values.*

Q. Is, each token has its own separate 512-dimensional vector ?

 Yes — each token has its own separate 512-dimensional vector.

Let me explain clearly:

What happens in the embedding layer?

- You have a vocabulary of tokens, say 50,000 tokens.
- You set embedding size = 512.

The embedding layer creates a lookup table (embedding matrix) of size:
50,000 tokens×512 dimensions

So, it's essentially a matrix of shape (50000, 512):

- Row 1 → embedding vector for token 1 (512 numbers)
- Row 2 → embedding vector for token 2 (512 numbers)
- …
- Row 50000 → embedding vector for token 50000 (512 numbers)

When you pass in a sequence

For example:

```
Input sequence: ["I", "am", "a", "student"]
```

Let’s say:

- “I” → token ID 12
- “am” → token ID 45
- “a” → token ID 78
- “student” → token ID 3000

The embedding layer:

- Looks up row 12 → 512-dimensional vector
- Looks up row 45 → 512-dimensional vector
- Looks up row 78 → 512-dimensional vector
- Looks up row 3000 → 512-dimensional vector

So, the sequence becomes a matrix of shape (4 tokens, 512 dimensions).

Important point:

Each token has its own unique 512-dim vector,
These vectors are learned and adjusted during training,
Similar tokens may end up with similar vectors.

 Example in simple form

Token	512-dimensional vector (example values)
“I”	[0.1, -0.3, 0.7, ..., 0.05]
“am”	[0.03, 0.91, -0.25, ..., -0.12]
“a”	[-0.15, 0.44, 0.62, ..., 0.09]
“student”	[0.55, -0.67, 0.11, ..., 0.33]

Q. How embeddings are initialized and updated,

Great! Here’s a clear explanation on how embeddings are initialized and updated during model training:

---

1. How embeddings are initialized

At the start of training, the embedding layer is typically initialized with random numbers.

Specifically:

- The embedding matrix (shape: `vocab_size × embedding_dim`) is filled with small random values, often drawn from:
  - Uniform distribution (e.g., `U(-0.1, 0.1)`), or
  - Normal distribution (e.g., `N(0, 0.01)`).

Why random?

- Because we have no prior knowledge about the meaning of tokens before training.
- The model will learn the best values by itself during training.

---

2. How embeddings are updated

The embeddings are updated during backpropagation like any other model parameters.

Here's the process:

1 Forward pass

- Input tokens → embedding layer → downstream layers → compute predictions → compute loss.

2 Backward pass

- Compute gradient of the loss with respect to each embedding vector.
- Example: If “student” appears in the input and contributes to the prediction error, the gradient of the loss will flow back to its embedding vector.

3 Parameter update

- The optimizer (e.g., Adam, SGD) adjusts the embedding matrix using the gradients: new embedding=old embedding−learning rate×gradient\text{new embedding} = \text{old embedding} – \text{learning rate} \times \text{gradient}

This happens for only the tokens present in the batch, not for all tokens.

---

3. Summary

Step	What happens
Initialization	Random small values assigned to each embedding vector.
Forward pass	Lookup embeddings → pass through model → compute output.
Backward pass	Compute gradients of loss w.r.t. embeddings.
Update	Adjust only the embeddings used in the batch, via optimizer.

Bonus: Pretrained embeddings

Sometimes we don’t start from random — we initialize with pretrained embeddings like:

- Word2Vec
- GloVe
- FastText

These are loaded into the embedding layer as initial values, and can be:

- Frozen → fixed during training.
- Fine-tuned → further updated with gradients.

Example in PyTorch

```
embedding = nn.Embedding(vocab_size, embedding_dim)
# Initialized randomly by default

optimizer = torch.optim.Adam(embedding.parameters(), lr=0.001)
# During training, optimizer updates embedding weights via backprop
```