ChatGPT

Token Embedding in PyTorch Transformers

A **token embedding** layer converts discrete token IDs into dense vectors. In PyTorch, this is typically done with `nn.Embedding` (and often wrapped by a `TokenEmbedding` subclass). For example, in a Transformer model the first step is to tokenize text into integer IDs, then pass those IDs through an embedding layer of shape `(vocab_size, d_model)` to get vectors of size `d_model` for each token. The `TokenEmbedding` class's constructor only needs two arguments – `vocab_size` and `d_model` – because these define the size of the internal lookup table (the embedding matrix). The `vocab_size` determines how many rows (one per token in the vocabulary) and `d_model` determines how many columns (the embedding dimension). The result is a weight matrix of shape `(vocab_size, d_model)` [1] [2]. During the forward pass, you **don't** pass raw text to this layer; instead you pass a PyTorch tensor of token indices (dtype `long`). The embedding layer then performs a table lookup: each input index selects the corresponding row of the weight matrix. The output is a dense vector (of length `d_model`) for each token index [3] [2].

Constructor: `vocab_size` and `d_model`

The `TokenEmbedding` constructor only takes `vocab_size` (number of distinct tokens) and `d_model` (vector size) because these fully specify the size of the embedding weights. In PyTorch's `nn.Embedding(num_embeddings, embedding_dim)`, `num_embeddings=vocab_size` and `embedding_dim=d_model` [4] [5]. For example, `nn.Embedding(10000, 512)` creates a `(10000×512)` weight matrix. Internally this matrix is initialized (often with small random values) and treated like a lookup table. No text or other data is needed in the constructor – the layer doesn't process actual tokens until the forward pass. The constructor only defines the shape of the weight tensor (and optionally a `padding_idx`). In summary:

- `vocab_size` (num_embeddings): number of rows in the embedding table (size of the vocabulary).
- `d_model` (embedding_dim): number of columns in each embedding vector (model dimension) [1] [5].

This matches the Transformer design, where the embedding dimension equals the model's hidden size (e.g. 512 or 768).

Forward Pass Input: Token IDs (not raw text)

During a forward pass, the **input** to `TokenEmbedding` is a tensor of token indices, **not raw text strings or images**. For text models, you first use a tokenizer (or vocabulary mapping) to convert words or subwords into integer IDs in the range `[0, vocab_size-1]`. These IDs form a PyTorch `LongTensor` of shape `[batch_size, sequence_length]`. When you call `TokenEmbedding(x)`, PyTorch treats each element of `x` as an index into the embedding table [3] [6]. Concretely, if

```
token_ids = torch.tensor([[12, 25,  7],
                          [ 4,  2, 11]])  # shape [2, 3], dtype=torch.long
```

and the embedding is `nn.Embedding(vocab_size=1000, embedding_dim=512)`, then `embedded = embedding(token_ids)` produces a tensor of shape `[2, 3, 512]`. Each `embedded[i,j]` is the 512-dimensional vector corresponding to the input index `token_ids[i,j]` [6]. The embedding layer does not understand raw text (strings); it only knows how to map integer IDs to vectors. Similarly, it does not accept image pixels – image data is typically handled by convolutional or linear layers, not `nn.Embedding`.

Key points about the forward input:

- Input is a **LongTensor of token indices**, shape `(batch_size, seq_length)`, e.g. `[[1,5,3, ...], [4,2,9, ...], ...]`.
- Each index must be an integer in `[0, vocab_size-1]` (or equal to `padding_idx` if used).
- The embedding layer performs a lookup: it replaces each index with the corresponding row vector from its weight matrix [3] [2].
- The output is a float tensor of shape `(batch_size, seq_length, d_model)`.

For example, the PyTorch docs show:

```
>>> embedding = nn.Embedding(10, 3)
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)
tensor([[[ 0.5532, -1.0127,  1.3929],
         [ 0.0723, -0.3750, -0.0773],
         [-0.3842, -1.3079, -0.9690],
         [-1.8088, -0.3024, -1.9127]],
        [[-0.3842, -1.3079, -0.9690],
         [ 0.2600, -0.3750,  0.8686],
         [ 0.0723, -0.3750, -0.0773],
         [ 0.4828, -0.8169,  0.7782]]])  # shape [2, 4, 3]
```

Here `embedding.weight` is shape `[10, 3]`, and each input index directly selects one row of that matrix [7] [2].

Internal Weight Matrix and `vocab_size`

When you construct `nn.Embedding(vocab_size, d_model)`, PyTorch allocates a weight matrix of shape `(vocab_size, d_model)` [1] [2]. Each of the `vocab_size` rows is a learnable parameter vector of length `d_model`. Intuitively, row *i* of this matrix is the embedding (vector representation) for token ID *i*. This is why the constructor only needs `vocab_size` and `d_model`: they determine the table size. The number of parameters in the embedding is `vocab_size * d_model`. During training, these vectors are learned (unless frozen or preloaded) so that tokens with similar meaning end up with similar vectors.

For example, if `vocab_size = 5000` and `d_model = 256`, then `nn.Embedding` creates a `(5000×256)` matrix of weights (often initialized from a normal distribution [1]). Conceptually, you can think of `nn.Embedding` as a special kind of linear layer with a one-hot input: an input index *k* is equivalent to a one-hot vector of length `vocab_size` (all zeros except a 1 at position *k*), and the output is that linear layer's weight matrix multiplied by the one-hot vector. In practice, `nn.Embedding` just does an efficient lookup of the *k*-th row, rather than multiplying by a one-hot vector [2].

From Token IDs to Dense Vectors

Putting it all together, during the forward pass the embedding layer converts token IDs into vectors with a simple lookup. Here's how it works step by step:

1. **Tokenization → Indices:** Convert raw text into a sequence of token IDs (e.g. using a vocabulary or tokenizer).
2. **Form Tensor of IDs:** Create a PyTorch tensor `x` (dtype `torch.long`) of shape `(batch_size, seq_length)` holding these IDs.
3. **Lookup:** Call `output = token_embedding(x)`. Internally, for each position `(i,j)`, the layer reads the ID `k = x[i,j]` and retrieves row `k` of its weight matrix.
4. **Output Shape:** The result is a tensor of shape `(batch_size, seq_length, d_model)`. Each slice `output[i,j,:]` is a dense `d_model`-dimensional vector representing the token at position `(i,j)`.

For example, with `vocab_size=10000` and `d_model=512`, an embedding layer will map an input tensor of shape `[2, 3]` (2 sequences of length 3) to an output tensor of shape `[2, 3, 512]` [6]:

```
embedding_layer = nn.Embedding(num_embeddings=10000, embedding_dim=512)
input_tokens = torch.tensor([[101, 24,  58],
                             [  4, 299, 11]])  # shape [2, 3]
embedded = embedding_layer(input_tokens)
print(embedded.shape)  # torch.Size([2, 3, 512])
```

Here, each entry in `input_tokens` is a token ID; `embedded[i,j]` is the 512-dimensional embedding for that token. In a Transformer model, these embeddings would then typically be scaled (e.g. multiplied by $\sqrt{\text{d\_model}}$) and have positional encodings added before being fed into the encoder or decoder layers.

Why Not Pass Raw Text or Images

It's important to note that **you never feed raw text (strings) or raw images into an `nn.Embedding` layer**. Embedding layers only accept numerical indices of type `LongTensor`. The raw text must first be tokenized into IDs. This is because `nn.Embedding` is essentially a lookup table keyed by integer IDs [3] [2]. Likewise, embedding layers are meant for categorical tokens (words, subwords, etc.), not continuous data like pixel values. If you are working with images in a Transformer (e.g. Vision Transformer), you first turn each image into a sequence of patch embeddings via a convolution or linear layer — again, not by feeding image pixels as indices to an `nn.Embedding`.

In summary, the embedding layer bridges between discrete token IDs and continuous model input: it takes an integer ID (obtained from text) and returns a float vector. Text or other data must be preprocessed into token IDs before this layer. There is no need – and indeed it would not work – to pass raw text or raw image data directly into an `nn.Embedding` layer [3] [2].

References: Official PyTorch docs and tutorials explain that `nn.Embedding(num_embeddings, embedding_dim)` builds a `(num_embeddings×embedding_dim)` weight matrix and performs a simple index lookup on its forward pass [3] [1]. For example, an index tensor `[[1,2,3],[4,5,6]]` into an embedding returns a tensor of shape `(2,3,embedding_dim)` [6]. Informal discussions likewise note that an embedding is just a M×N matrix (M=vocab size, N=embedding dim) and each input index picks the corresponding row [2]. These principles explain why `TokenEmbedding` only needs `vocab_size` and `d_model` to set up the layer, and why its inputs must already be numeric token IDs rather than raw text or images.

---

[1] [3] [4] Embedding — PyTorch 2.7 documentation
https://docs.pytorch.org/docs/stable/generated/torch.nn.Embedding.html

[2] How does nn.Embedding work? - PyTorch Forums
https://discuss.pytorch.org/t/how-does-nn-embedding-work/88518

[5] [6] How to Build and Train a PyTorch Transformer Encoder | Built In
https://builtin.com/artificial-intelligence/pytorch-transformer-encoder

[7] Implementation of Transformer using PyTorch (detailed explanations) | Longxiang He
https://say-hello2y.github.io/2022-08-18/transformer