**nn.einsum()**

https://www.youtube.com/watch?v=pkVwUVEHmfI

**nn.LayerNorm()**

**nn.Embedding()**

Certainly! In PyTorch, the nn.Embedding module serves as a **lookup table** that efficiently maps discrete tokens (like words in natural language processing) into dense vector representations. These vector embeddings capture semantic relationships between tokens, allowing neural networks to process them more effectively.

Here's a deeper dive into how nn.Embedding works:

**Core Functionality:**

* It takes two main arguments:
  + num\_embeddings (int): The size of the vocabulary (total number of unique tokens).
  + embedding\_dim (int): The dimensionality of the embedding vectors (length of each vector representation).
* Internally, nn.Embedding creates a weight matrix of size (num\_embeddings, embedding\_dim). Each row in this matrix represents the embedding vector for a specific token (indexed by its position in the vocabulary).

**Usage:**

1. **Input:** You typically provide nn.Embedding with a tensor of long integers, where each element represents the index of a token within the vocabulary.
2. **Lookup:** Based on the input indices, nn.Embedding retrieves the corresponding embedding vectors from the weight matrix.
3. **Output:** The output is a tensor of size (batch\_size, sequence\_length, embedding\_dim), where:
   * batch\_size is the number of data samples being processed together.
   * sequence\_length is the length of the token sequences in the input (e.g., the number of words in a sentence).
   * embedding\_dim is the dimensionality of the embedding vectors as defined during initialization.

**Benefits of Embeddings:**

* **Semantic Relationships:** Well-trained embeddings capture semantic relationships between tokens. Words with similar meanings tend to have embedding vectors that are closer together in the embedding space. This allows neural networks to leverage these relationships during tasks like sentiment analysis or machine translation.
* **Dimensionality Reduction:** Embeddings represent discrete tokens as dense vectors in a lower-dimensional space compared to one-hot encodings. This reduces the computational cost and memory footprint, especially for large vocabularies.

**Example Code:**

Python

import torch

import torch.nn as nn

# Define an embedding layer with vocabulary size 10000 and embedding dimension 128

embedding = nn.Embedding(10000, 128)

# Example input: a tensor of indices representing a sentence

sentence\_indices = torch.tensor([23, 87, 112, 456]) # Example token indices

# Get the embedding vectors for the sentence

sentence\_embeddings = embedding(sentence\_indices)

print(sentence\_embeddings.shape) # Output: torch.Size([1, 4, 128])

**In essence, nn.Embedding provides a powerful tool for bridging the gap between discrete symbolic data (like words) and the continuous vector representations that neural networks excel at processing.**

**nn.ModuleList()**

**torch.transpose()**

Certainly! In PyTorch, `torch.transpose()` is a function that permutes the dimensions of a tensor. It returns a new tensor with the specified dimensions rearranged. Let's demonstrate with a simple example:

```python

import torch

# Create a tensor with shape (2, 3)

tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Transpose the tensor, swapping the first and second dimensions

transposed\_tensor = torch.transpose(tensor, 0, 1)

print("Original Tensor:")

print(tensor)

print("Transposed Tensor:")

print(transposed\_tensor)

```

Output:

```

Original Tensor:

tensor([[1, 2, 3],

[4, 5, 6]])

Transposed Tensor:

tensor([[1, 4],

[2, 5],

[3, 6]])

```

In this example:

- The original tensor has shape **`(2, 3)`** (2 rows, 3 columns).

- After transposing it with **`torch.transpose(tensor, 0, 1)`**, the dimensions are permuted such that the first dimension becomes the second dimension and vice versa. So, the resulting tensor has shape `**(3, 2)**` (3 rows, 2 columns).

You can use `**torch.transpose()**` to rearrange the dimensions of a tensor according to your specific needs, such as converting a batched sequence tensor from `**(batch\_size, seq\_len, input\_size)**` to `**(seq\_len, batch\_size, input\_size)**` for compatibility with certain PyTorch modules like LSTM, as we did in the previous example.

**torch.contiguous()**