EMBEDDING

`CLASS torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None, _freeze=False, device=None, dtype=None`

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Parameters:

> **num_embeddings** (int) — size of the dictionary of embeddings
>
> **embedding_dim** (int) — the size of each embedding vector
>
> **padding_idx** (int, optional) — If specified, the entries at padding_idx do not contribute to the gradient; therefore, the embedding vector at padding_idx is not updated during training, i.e. it remains as a fixed “pad”. For a newly constructed Embedding, the embedding vector at padding_idx will default to all zeros, but can be updated to another value to be used as the padding vector.
>
> **max_norm** (float, optional) — If given, each embedding vector with norm larger than max_norm is renormalized to have norm max_norm.
>
> **norm_type** (float, optional) — The p of the p–norm to compute for the max_norm option. Default 2.
>
> **scale_grad_by_freq** (bool, optional) — If given, this will scale gradients by the inverse of frequency of the words in the mini–batch. Default False.
>
> **sparse** (bool, optional) — If True, gradient w.r.t. weight matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

Share weights in nn.Embedding

```
import torch
import torch.nn as nn

# Define the size of the embedding vector
embedding_size = 100

# Define the total number of unique input indices
num_indices = 10

# Create the weight tensor
weights = nn.Parameter(torch.randn(num_indices, embedding_size))

# Create two instances of the embedding layer and share the weights
embedding1 = nn.Embedding.from_pretrained(weights)
embedding2 = nn.Embedding.from_pretrained(weights)

# Define two input tensors
input1 = torch.LongTensor([1, 3, 5])
input2 = torch.LongTensor([3, 7, 9])

# Apply the embedding layers to the inputs
output1 = embedding1(input1)
output2 = embedding2(input2)

# Check if the weights are shared
print(embedding1.weight.data_ptr() == embedding2.weight.data_ptr())  # True
```

Sharing method 2

```
import torch
import torch.nn as nn
import torch.optim as optim

class testModule(nn.Module):

    def __init__(self):
        super(testModule, self).__init__()
        self.fc1 = nn.Linear(5, 10, bias=True)
        self.fc2 = nn.Linear(10, 10, bias=False)

        # Remove the weights as we override them in the forward
        # so that they don't show up when calling .parameters()
        del self.fc1.weight
        del self.fc2.weight

        self.fc2_base_weights = nn.Parameter(torch.randn(10, 10))
        self.shared_weights = nn.Parameter(torch.randn(10, 5))

    def forward(self, x):
        # Update the weights
        index = [1, 3, 5, 7, 9]
        self.fc1.weight = self.shared_weights
        self.fc2.weight = self.fc2_base_weights.clone()
        self.fc2.weight[:, index] = self.shared_weights

        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

Method 3

```
class MyModel(nn.Module):
    def __init__(self):
        self.base = ...
        self.head_A = ...
        self.head_B = ...

    def forward(self, input1, input2):
        return self.head_A(self.base(input1)), self.head_B(self.base(input2))
```

Method 4