# embedding_layer

January 23, 2023

[ ]:
```
#necessary imports
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
```

## 0.1 Defination of Embedding Layer:

```
class torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_
```

- num_embeddings (int) - size of the dictionary of embeddings

- embedding_dim (int) - the size of each embedding vector

### 0.1.1 Now let's initialize the embedding layer with random weights:

```
import torch
import torch.nn as nn

emb = nn.Embedding(10, 3)
print(emb.weight)
```

[ ]:
```
emb = nn.Embedding(10, 3)
print(emb.weight)
```

```
Parameter containing:
tensor([[ 0.0813, -0.2706, -0.4212],
        [ 1.7065,  0.1108,  0.4840],
        [-0.0782,  0.1840,  0.8727],
        [-0.3051,  0.8612, -1.3237],
        [-1.0023,  0.5371, -0.2529],
        [ 1.1306, -0.8332,  0.3118],
        [ 1.4076, -0.1370, -0.1875],
        [ 2.0790,  0.9954, -0.0522],
        [ 1.2692, -1.1039,  1.6065],
        [-0.0756,  0.1147,  0.6017]], requires_grad=True)
```

So, what it means, is that we have a dictionary of 10 words or anything that you might want to encode, and each word is represented by a vector of size 3. So, the size of the embedding layer is 10x3.

Now, let's see how the embedding layer works. We will create a tensor of indices and pass it to

```
[ ]: X = torch.randint(0, 10, (10, ))
     print(X.shape)
```

torch.Size([10])

So, we have a tensor of size 10, which is 10 rows. Now, each of the number in X will be represe

```
[ ]: out = emb(X)
     print(out.shape)
```

torch.Size([10, 3])

3 is important there, as this will pertain no matter that the input size is. If the input size

```
[ ]: X = torch.randint(0, 10, (5, ))
     print(X.shape)

     out = emb(X)
     print(out.shape)
```

torch.Size([5])
torch.Size([5, 3])

```
[ ]: print(X)
```

tensor([5, 5, 3, 9, 3])

Which means, in our embedding, we can encode 10 numbers at most, but we are encoding 5 in this
And all of these 5 numbers are in the range of 0 and 9. Our embedding also can fit at most 10 n

Now, let's see how the weights look like

```
[ ]: print(emb.weight)
```

```
Parameter containing:
tensor([[ 0.0813, -0.2706, -0.4212],
        [ 1.7065,  0.1108,  0.4840],
        [-0.0782,  0.1840,  0.8727],
        [-0.3051,  0.8612, -1.3237],
        [-1.0023,  0.5371, -0.2529],
        [ 1.1306, -0.8332,  0.3118],
        [ 1.4076, -0.1370, -0.1875],
        [ 2.0790,  0.9954, -0.0522],
        [ 1.2692, -1.1039,  1.6065],
        [-0.0756,  0.1147,  0.6017]], requires_grad=True)
```

[ ]:
```python
print(emb(torch.tensor([0,1])))
print(emb(torch.tensor([2, 3, 4])))
```

```
tensor([[ 0.0813, -0.2706, -0.4212],
        [ 1.7065,  0.1108,  0.4840]], grad_fn=<EmbeddingBackward0>)
tensor([[-0.0782,  0.1840,  0.8727],
        [-0.3051,  0.8612, -1.3237],
        [-1.0023,  0.5371, -0.2529]], grad_fn=<EmbeddingBackward0>)
```

So, what if we want to encode a number bigger than 9? Can our embedding layer do that?
As we can see from above prints, each element of in rows are assigned to numbers from 0 to 9. S

print(emb(torch.tensor([10])))
this will throw an error: IndexError: index out of range in self

### 0.1.2 How does it act in a neural network?

Lets say we have 10 numbers to encode, and each number will have a vector of size 3.

[ ]:
```python
emb = nn.Embedding(10, 3)
```

We need a batch number of encoding for number 0-9

[ ]:
```python
labels = torch.randint(0, 10, (64, ))
print(labels.shape)
```

```
torch.Size([64])
```

We want an input to the network. Let's say the input noise size is 10. So, out input will be ba

[ ]:
```python
X = torch.randn(64, 10)
print(X.shape)
```

```
torch.Size([64, 10])
```

As we are feeding batch number of labels to the embedding layer, we will have batch number of e
Output will be batch size x embedding size.

[ ]:
```python
emb_out = emb(labels)
print(emb_out.shape)
```

```
torch.Size([64, 3])
```

Now we can concatenate the embedding vectors to the end of the input noise.
So, the output will be batch size x (input noise size + embedding size)

[ ]:
```python
cat = torch.cat((X, emb_out), dim=1)
print(cat.shape)
```

```
torch.Size([64, 13])
```

Let's see what happens

[ ]:
```
print(X[0])
print(emb.weight[0])
```

```
tensor([-0.5269,  1.0298, -0.6908,  1.8272, -0.9981,  0.0789, -2.8797,  1.0594,
         1.1529, -1.0838])
tensor([-0.8949,  0.3877,  0.5457], grad_fn=<SelectBackward0>)
```

These two will be concatenated

[ ]:
```
print(cat[0])
```

```
tensor([[-0.5269],
        [ 1.0298],
        [-0.6908],
        [ 1.8272],
        [-0.9981],
        [ 0.0789],
        [-2.8797],
        [ 1.0594],
        [ 1.1529],
        [-1.0838],
        [-1.8254],
        [ 0.1329],
        [ 0.0273]], grad_fn=<SelectBackward0>)
```

For time series, we can unsqueeze and add one dimension at the end.
So, we will have 1 feaure column.

[ ]:
```
#unsqueezing
cat = cat.unsqueeze(-1)
print(cat.shape)
```

```
torch.Size([64, 13, 1])
```

[ ]:
```
cat[0]
```

[ ]:
```
tensor([[-0.5269],
        [ 1.0298],
        [-0.6908],
        [ 1.8272],
        [-0.9981],
        [ 0.0789],
        [-2.8797],
        [ 1.0594],
        [ 1.1529],
        [-1.0838],
        [-1.8254],
        [ 0.1329],
        [ 0.0273]], grad_fn=<SelectBackward0>)
```