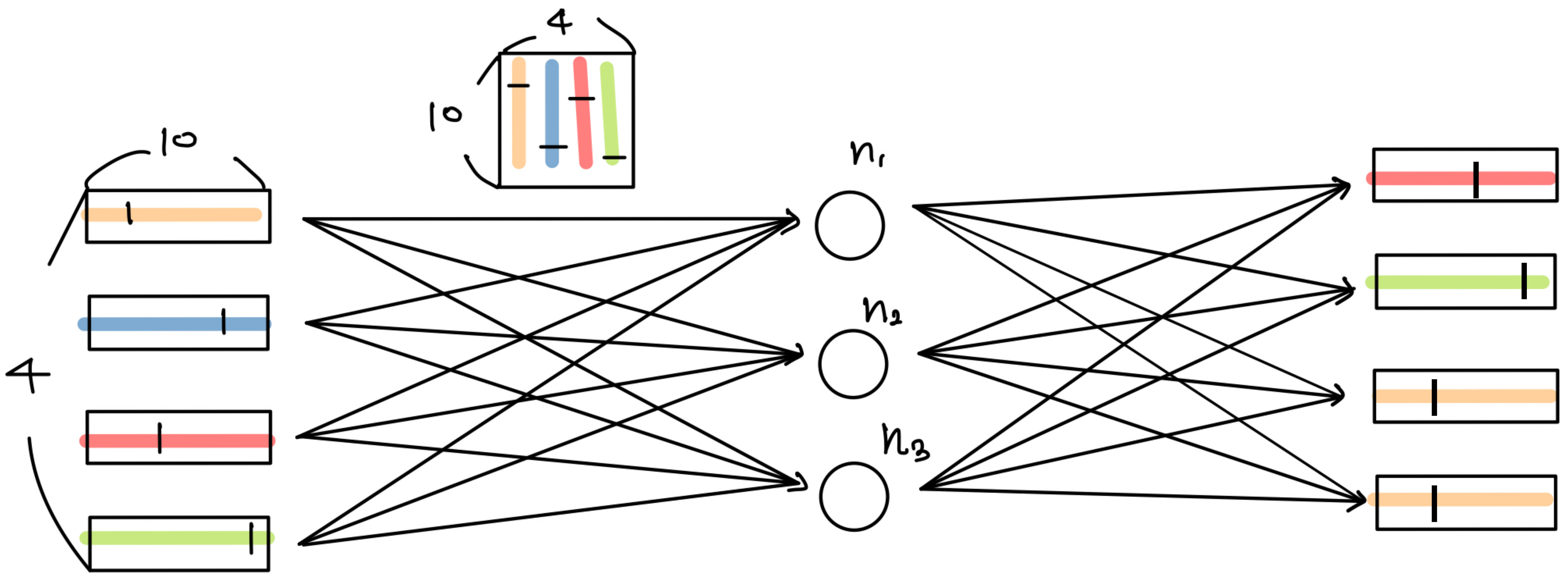
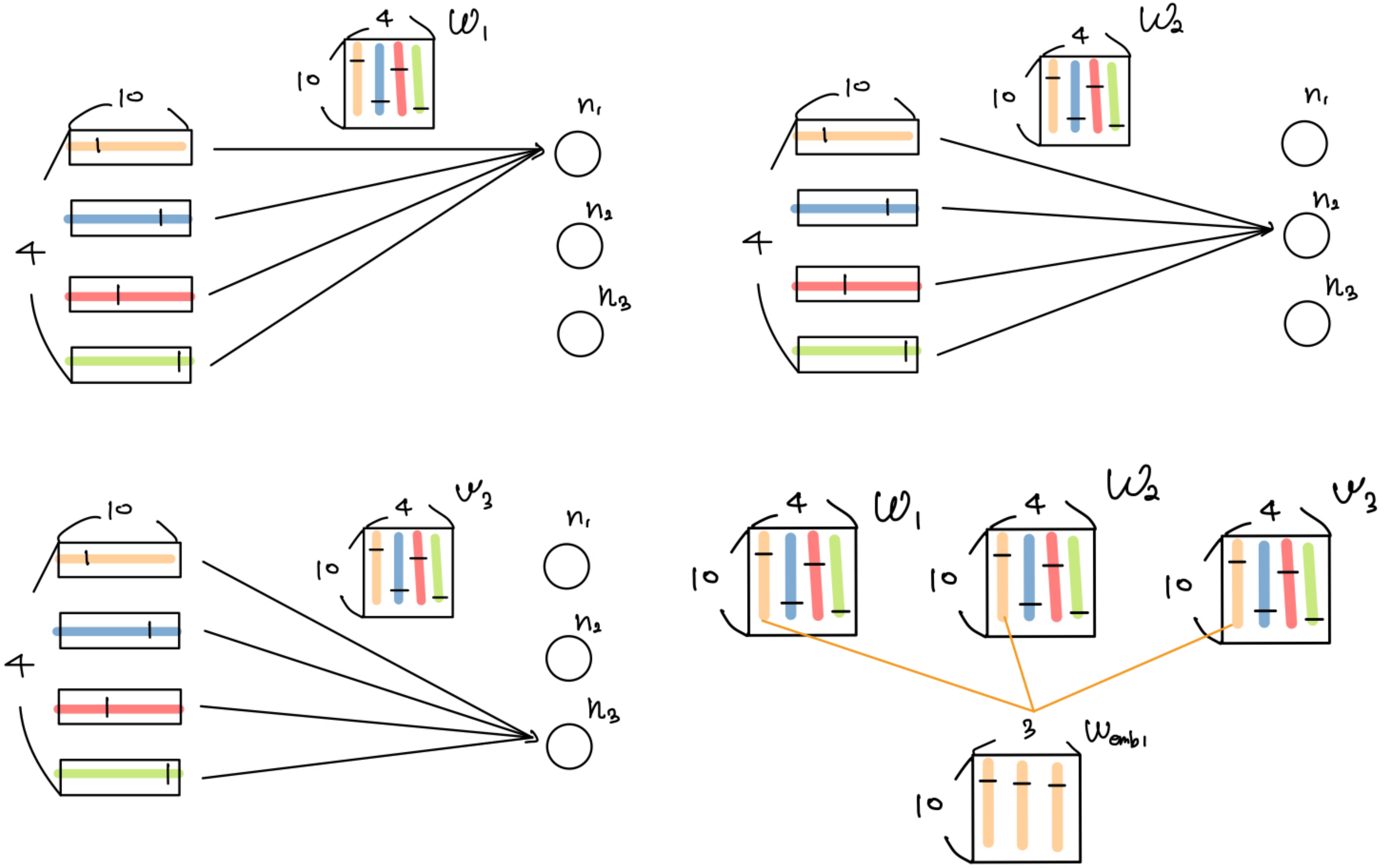
01.Word2vec. MLP와 뭐가 다른가?

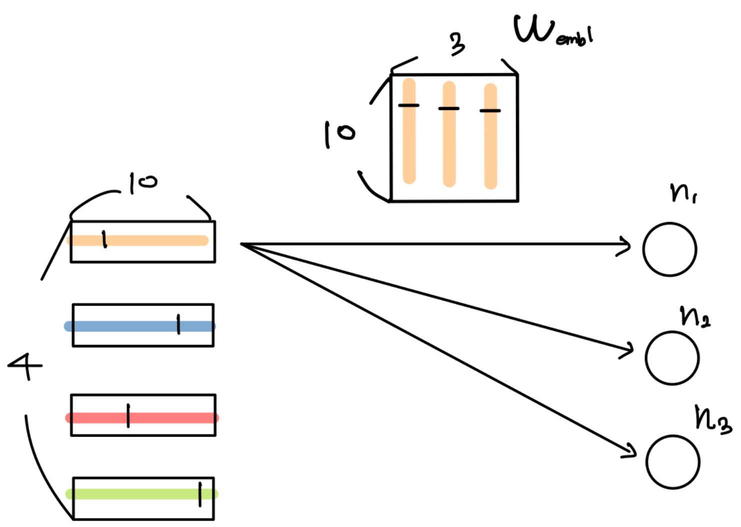
10개의 단어 ⟶ 3 차원의 embeding vector.

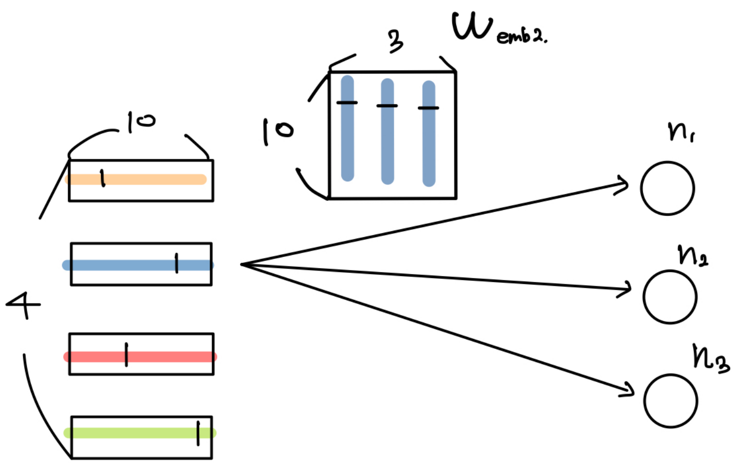


01.Word2vec. MLP와 뭐가 다른가?



01.Word2vec. MLP와 뭐가 다른가?





Linear 아닌 Embeding

```
class CBOW(nn.Module):
    def __init__(self, vocab_size, dim):
        super(CBOW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, dim, sparse=True)
        self.linear = nn.Linear(dim, vocab_size)

    def forward(self, x):
        embeddings = self.embedding(x)
        embeddings = torch.sum(embeddings, dim=1)
        output = self.linear(embeddings)
        return output
```

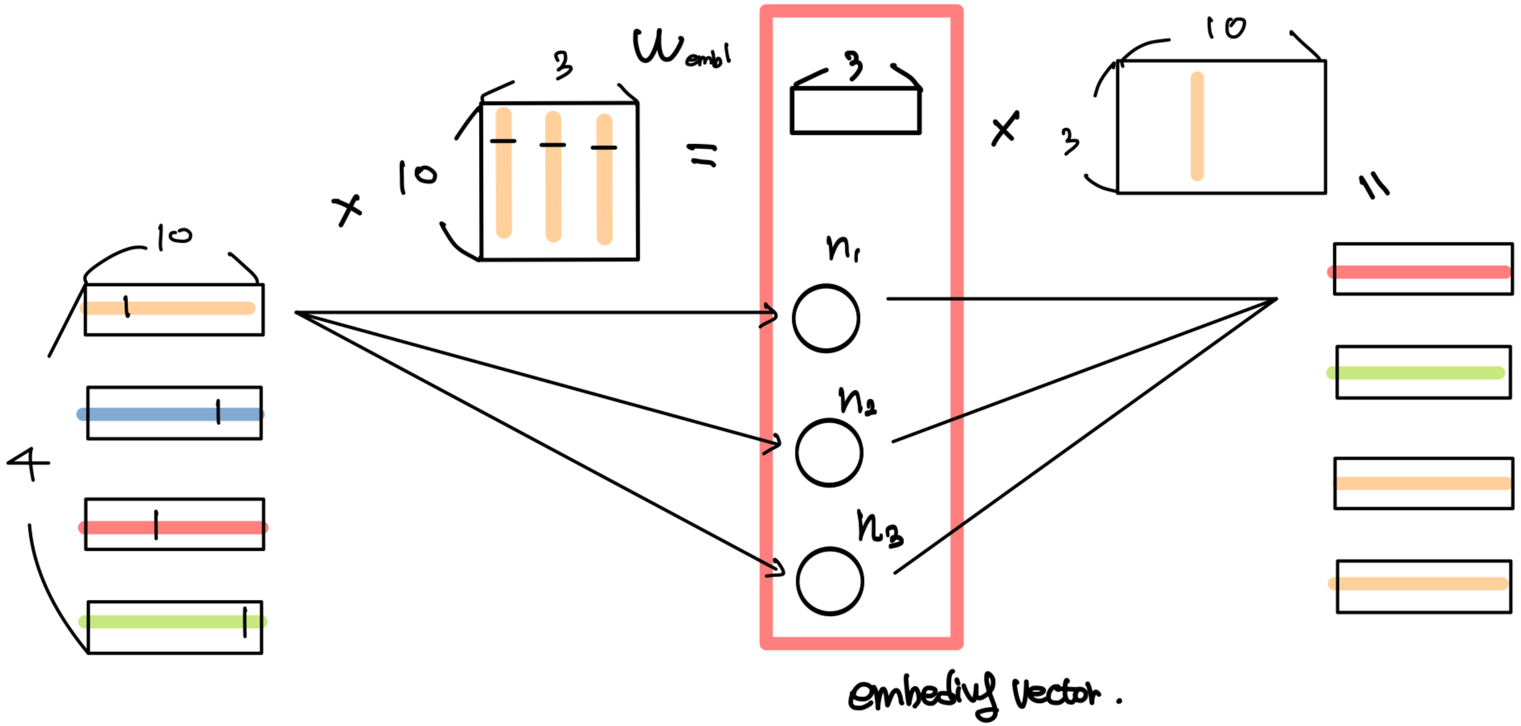
01.Word2vec. MLP와 뭐가 다른가?

```python
for word in test_words:
    input_id = torch.LongTensor([w2i[word]]).to(device)
    emb = cbow.embedding(input_id)

    print(f"Word: {word}")
    print(emb.squeeze(0))
```

```python
class CBOW(nn.Module):
    def __init__(self, vocab_size, dim):
        super(CBOW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, dim, sparse=True)
        self.linear = nn.Linear(dim, vocab_size)

    def forward(self, x):
        embeddings = self.embedding(x)
        embeddings = torch.sum(embeddings, dim=1)
        output = self.linear(embeddings)
        return output
```



입력으로 들어오는 주변 단어의 원-핫 벡터와 가중치 W 행렬의 곱이 어떻게 이루어지는지 보겠습니다. 위 그림에서는 각 주변 단어의 원-핫 벡터를 x로 표기하였습니다. 입력 벡터는 원-핫 벡터입니다. i번째 인덱스에 1이라는 값을 가지고 그 외의 0의 값을 가지는 입력 벡터와 가중치 W 행렬의 곱은 사실 W행렬의 i번째 행을 그대로 읽어오는 것과(lookup) 동일합니다. 그래서 이 작업을 룩업 테이블(lookup table)이라고 부릅니다. 앞서 CBOW의 목적은 W와 W'를 잘 훈련시키는 것이라고 언급한 적이 있는데, 사실 그 이유가 여기서 lookup해온 W의 각 행벡터가 사실 Word2Vec을 수행한 후의 각 단어의 M차원의 크기를 갖는 임베딩 벡터들이기 때문입니다.

1. Treat `nn.Embedding` as a lookup table where the key is the word index and the value is the corresponding word vector. However, before using it you should specify the size of the lookup table, and initialize the word vectors.

   Not all the weights are trained at the same time in this nn.Embedding. Weight training would depend on your training pairs. For example, lets say ('Bruce' , 'Wayne') is a training pair. Assuming that 'Bruce' and 'Wayne' words are present in your vocabulary with indices 100 and 200(just an example), nn.Embedding would allow you to pick the untrained word vectors for these two indices. This these two vectors would be brought closer to each other, resulting in their training.

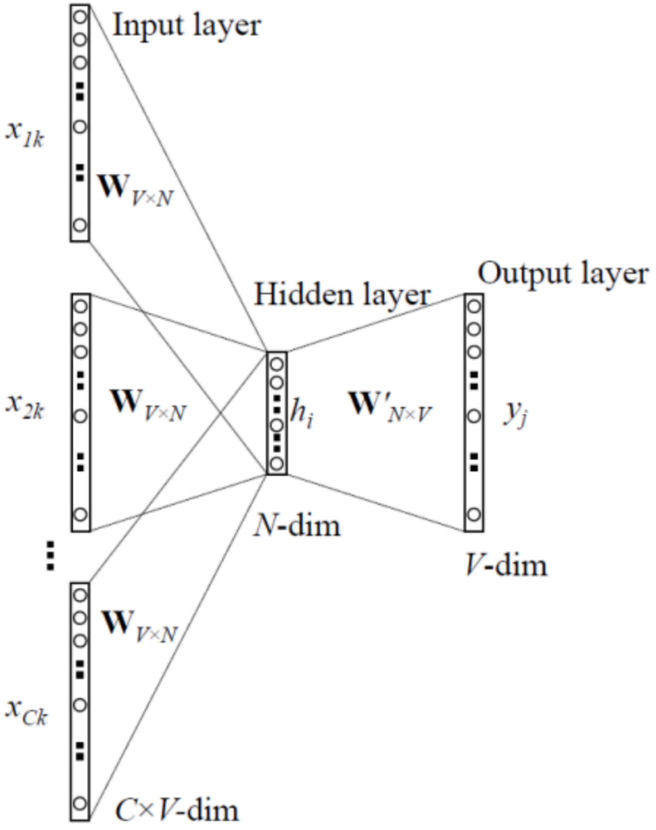
   Remember nn.Embedding is a lookup table. You just need to give in the indices of the words , and it gives you the word vectors for those words.

2. You can have a look at this pytorch implementation of Skip-Gram model

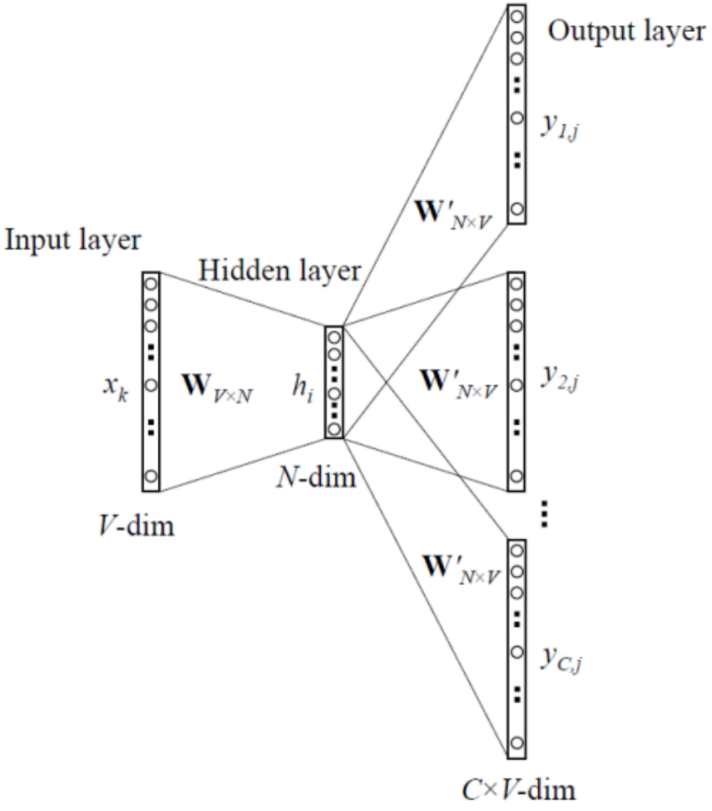
02.Dataset 만들기

발표를 준비하다 보니 벌써 새벽이다 오늘도 잠을 늦게 자겠다.

CBOW (Continuous Bag of Words)



Skip-Gram



발표를 준비하다 보니 벌써 --이다 오늘도 잠을 늦게 자겠다.

--------------- 새벽 -----------------

03.신경망의 깊이는 다를 수 있다.

수업자료

```
class CBOW(nn.Module):
    def __init__(self, vocab_size, dim):
        super(CBOW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, dim, sparse=True)
        self.linear = nn.Linear(dim, vocab_size)

    def forward(self, x):
        embeddings = self.embedding(x)
        embeddings = torch.sum(embeddings, dim=1)
        output = self.linear(embeddings)
        return output
```

Gensim

```
cbow_mean : {0, 1}, optional
    If 0, use the sum of the context word vectors. If 1, use the mean, only applies when cbow is used.
```

```
class CBOW(nn.Module):
    def __init__(self, vocab_size, embedding_size, context_size):
        super(CBOW, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        self.context_size = context_size
        self.embeddings = nn.Embedding(self.vocab_size, self.embedding_size)
        # return vector size will be context_size*2*embedding_size
        self.lin1 = nn.Linear(self.context_size * 2 * self.embedding_size, 512)
        self.lin2 = nn.Linear(512, self.vocab_size)
```

Pytorch

```
class CBOW(nn.Module):

    def __init__(self, vocab_size, embedding_dim, window_size):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, vocab_size)
        self.window_size = window_size

    def forward(self, inputs):

        embeds = torch.sum(self.embeddings(inputs), dim=1) # [200, 4, 50] => [200, 50]
        # embeds = self.embeddings(inputs).view((batch_size, -1))
        out = self.linear(embeds) # nonlinear + projection
        log_probs = F.log_softmax(out, dim=1) # softmax compute log probability

        return log_probs
```

https://wikidocs.net/60854