Word2Vec

Word2Vec 是一种将单词转换为具有语义信息的稠密数字向量方法，通过 Word2Vec 可以将具有相同语义信息的单词转换成"距离"接近的特征向量空间的向量；

One-Hot 的缺陷：

1. 高维稀疏：One-Hot 最终形成的向量维度是词汇表大小，当词汇表特别大的时候，向量维度非常大，并且仅对应位置为 1，其它位置均为 0；
2. 向量之间没有语义相似度信息：所有词向量之间的相似度距离完全一样，无法体现 token 之间的相关性；

Word2Vec 核心思想：

朴素假设：一个词的语义信息取决于它周围出现的词，或者说经常在相同上下文出现的词对应的词向量应该具有比较高的语义相似性；比如：

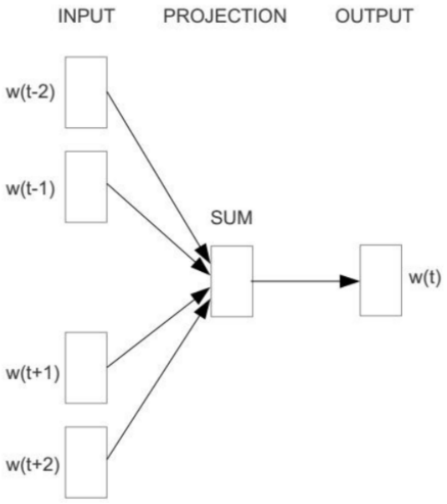
从人的认知来看，"猫"和"狗"两个词具有比较高的相似性，都具有宠物的特性。而这两个词语可能经常在下列类型的上下文中出现，可以非常明显的看到两个词语出现的上下文具有比较高的相似性：

1. 当我家的猫来到我的生活中后，我发现我的生活变得更加多彩了；
2. 当我家的狗来到我家后，我必须每天带它去散步；

两种结构：

CBOW：使用上下文词预测中心词；比如："我最喜欢的动物是小猫和松鼠"就存在这样一组训练样本：**[动物, 是，和，松鼠] → 小猫**

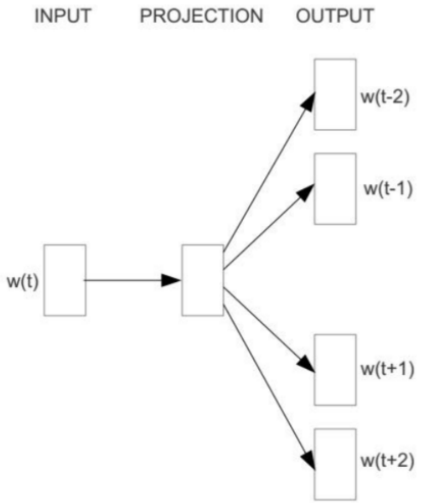
CBOW 结构相对来讲更适合大规模数据上的训练，比较擅长处理高频词，在常见词上的表现更加稳定，相对来讲结构更加简单一些，训练速度更快一些。



CBOW

Skip-Gram：使用中心词预测上下文单词；比如："我最喜欢的动物是小猫和松鼠"就存在这样一组训练样本: **小猫 → [动物, 是，和，松鼠]**

Skip-Gram 在小数据集或者低频词上的处理效果更好，直接使用中心词预测上下文，强制对稀有词汇的语义信息的学习更新，通常来讲效果优于 CBOW。



Skip-gram

为什么 Word2Vec 的训练就可以捕获词语的语义信息呢?

当词语 A 和词语 B 经常在相同或者类似的上下文中出现后，此时词语 A 和词语 B 的对应词向量就是类似的，结合 Skip-Gram 结构来理解，此时中心词就是 A 或者 B，但是此时对应的上下文单词是相同的，也就是对模型来讲：输入不同的单词 X，但是最终得到的预测结果 Y 是一样的，那要求损失最小化，那就只能将 A 和 B 对应的词向量映射为两个接近的向量。

两种优化：

1. 霍夫曼树：层级 Softmax
2. 负采样

两种结构的 PyTorch 结构：

Word2Vec CBOW Python

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class CBOWWord2Vec(nn.Module):
    def __init__(self, vocab_size, dim_size):
        super().__init__()
        self.vocab_size = vocab_size
        self.dim_size = dim_size
        # 这个w就是需要训练学习的参数列表，也是最终期望输出的词向量矩阵(包含了每个单词对应的词向量)
        self.w = nn.Parameter(torch.randn(self.vocab_size, self.dim_size))
        # 输出层使用不同参数，也可以直接使用w(相当于复用)
        self.v = nn.Parameter(torch.randn(self.dim_size, self.vocab_size))

        # self.emb_layer = nn.Embedding(num_embeddings=vocab_size, embedding_dim=dim_size)

    def forward(self, input_tokens):
        """
        :param input_tokens: [bs,m] bs个样本，每个样本输入m个token id
        :return:
        """
        # 1 针对 input_token id做哑编码, [bs,m] -> [bs,m,v]
        x = F.one_hot(input_tokens, self.vocab_size)
        x = x.to(dtype=self.w.dtype)
        # 2 获取每个单词对应的词向量 [bs,m,v]*[v,e] -> [bs,m,e]
        input_token_embs = torch.matmul(x, self.w)

        # 3. 求和 将m个单词的特征向量合并到一起 [bs,m,e] -> [bs,e]
        input_ctx_embs = torch.sum(input_token_embs, dim=1)

        # 4. 基于上下文求解属于各个单词的置信度 [bs,e] * [e,v] -> [bs,v]
        # z = torch.matmul(input_ctx_embs, self.w.T)
        z = torch.matmul(input_ctx_embs, self.v)
        return z


if __name__ == '__main__':
    net = CBOWWord2Vec(vocab_size=100, dim_size=4)
```

```
    loss_fn = nn.CrossEntropyLoss() # 交叉熵损失函数
    opt = optim.SGD(params=net.parameters(), lr=0.01)

    y = torch.tensor([5, 13])
    r = net(
        torch.tensor([
            [1, 8, 10, 13, 14, 16],  # 中心词是5
            [8, 10, 5, 14, 16, 18]  # 中心词是13
        ])
    )
    print(r.shape)
    loss = loss_fn(r, y)
    print(loss)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

Word2Vec Skip-Gram Python

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class SkipGramWord2Vec(nn.Module):
    def __init__(self, vocab_size, hidden_size):
        super().__init__()
        self.emb_table = nn.Embedding(num_embeddings=vocab_size, embedding_dim=hidde
n_size)

        self.fc_layer = nn.Linear(hidden_size, vocab_size, bias=False)

    def forward(self, token_ids):
        """
        SkipGram
        :param token_ids: 中心词 [bs,1]
        :return:
        """
        # 1. 获取输入token的特征向量 [bs,1] -> [bs,1,hidden_size]
        x = self.emb_table(token_ids)

        # 2. 将输入token特征向量合并为上下文特征向量
        x = torch.mean(x, dim=1)  # [bs,1,hidden_size] -> [bs,hidden_size]

        # 3. 基于合并的特征向量预测其它token属于各个类别的置信度
        score = self.fc_layer(x)  # [bs,vocab_size]
```

```
        return score

def tt01():
    vocab_size = 100
    net = SkipGramWord2Vec(vocab_size=vocab_size, hidden_size=4)

    # loss_fn = nn.CrossEntropyLoss() # softmax交叉熵损失函数
    loss_fn = nn.BCEWithLogitsLoss()  # sigmoid交叉熵损失函数
    opt = optim.SGD(params=net.parameters(), lr=0.01)

    x = torch.tensor([[5], [13]])
    y = torch.tensor([
        [1, 8, 10, 8, 14, 16],  # 中心词是5
        [8, 10, 5, 14, 16, 18]  # 中心词是13
    ])  # [bs,t]
    y_onehot = F.one_hot(y, num_classes=vocab_size)  # [bs,t] -> [bs,t,vocab_size]
    y_onehot = torch.max(y_onehot, dim=1).values  # [bs,t,vocab_size] -> [bs,vocab_s
ize]
    r = net(x)  # [bs,vocab_size]
    print(r.shape)
    loss = loss_fn(r, y_onehot.to(dtype=r.dtype))
    print(loss)
    opt.zero_grad()
    loss.backward()
    opt.step()

if __name__ == '__main__':
    tt01()
```

参考：

https://zhuanlan.zhihu.com/p/371147732