NLP 发展及其实现

许振雷

摘　　要

本文描述了自然语言的发展及其基本的论文实现。

关键词： NLP

1 A Neural Probabilistic Language Model

统计语言模型一个核心思想是学习每一个单词的联合概率分布。但是由于单词量的大，在维度空间上是非常困难的，传统一个有效的方法就是"n-gram" 模型。该文章提出了一种单词的分布，给定一个句子，预测最后一个单词是什么？该文章提出了使用 Embedding 进行单词的表示。图 1可

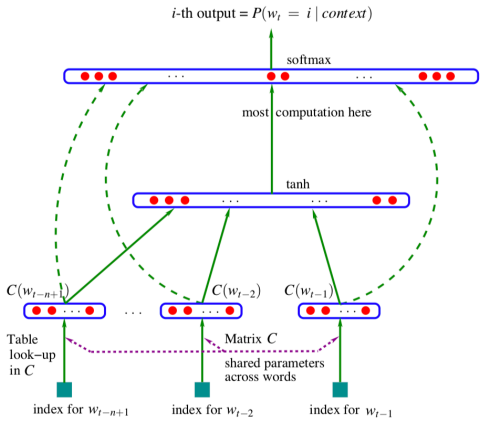


Figure 1: Neural architecture: $f(i,w_{t-1},\cdots,w_{t-n+1}) = g(i,C(w_{t-1}),\cdots,C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

图 1: NNLM 网络图

以发现，使用了共享参数也就是 Embedding 来表示每一个单词，同时进行激活层，以及矩阵操作，叠加以后进行 softmax。而对应的代码实现图 2非常简单的网络，要知道的是，该论文是 2003 年由 Bengio 提出来的，也就是今年的图灵奖的获得者之一，在 03 年 SVM 兴起的时代，仍然支持神经网络，我对 Bengio 老爷子的佩服犹如滔滔江水连绵不绝。

```python
m = 2
n_step = 2
n_hidden = 2

class NNLM(nn.Module):
    def __init__(self):
        super(NNLM, self).__init__()
        self.C = nn.Embedding(n_class, m)
        self.H = nn.Parameter(torch.randn(n_step*m, n_hidden))
        self.W = nn.Parameter(torch.randn(n_step*m, n_class))
        self.d = nn.Parameter(torch.randn(n_hidden))
        self.U = nn.Parameter(torch.randn(n_hidden, n_class))
        self.b = nn.Parameter(torch.randn(n_class))
    def forward(self, x):
        X = self.C(x)
        X = X.view(-1, n_step * m)
        tanh = torch.tanh(self.d + torch.mm(X, self.H))
        out = self.b + torch.mm(X,self.W) + torch.mm(tanh, self.U)
        return out
```

图 2: NNLM 网络图

2 word2vec

13 年提出的"word2vec" 是 NLP 领域的一个重点，该文章的出现，引起了自然语音处理的复兴，其核心思想是采用周围单词来预测中心词或者用中心词来预测周围单词。一种全新的方式来表达词的表达，相比 03 年的《A Neural Probabilistic Language Model》，使用周围来进行预测，更加能够学习到上下文特征。同时进行负采样

$$\log \sigma(v'^{T}_{wO} v_{wI}) + \sum_{i=1}^{k} \mathbb{E}_{wi \sim P_n(w)}[\log \sigma(-v'^{T}_{wi} v_{wI})]$$

，该目标函数是越大越好，对应的为前一项越大，也就是正样本，同时使得负样本越小越好，也就是给定一个中心词，正样本出现概率大，负样本出现概率小，Google 凭借着这篇文章掀起了自然语音处理的热潮。图 3，通过这个网络学习的两个矩阵就是 Embedding 向量，其中前者是我们需要的

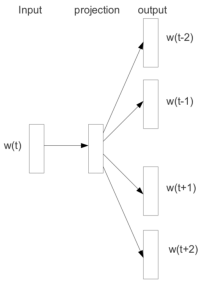


Figure 1: The Skip-gram model architecture. The training objective is to learn word vector representations that are good at predicting the nearby words.

图 3: word2vec

Embedding 向量，后者不需要，通过 Embedding 向量，我们能够获得每一个单词的向量表达，从而能够进行其他任务。该文章是一个先河，以后的工作基本都是如何准确的训练出 Embedding 向量。图 4，而该向量是可以用图像来表达，我们获得的效果如下图 5，一般来说 Embedding 向量只有 2

```python
class Word2Vec(nn.Module):
    def __init__(self):
        super(Word2Vec, self).__init__()
        self.W = nn.Parameter(-2*torch.rand(voc_size, embedding_size) + 1)
        self.WT = nn.Parameter(-2*torch.rand(embedding_size, voc_size) + 1)

    def forward(self, X):
        h = torch.matmul(X, self.W)
        o = torch.matmul(h, self.WT)
        return o
```

图 **4:** word2vec-code

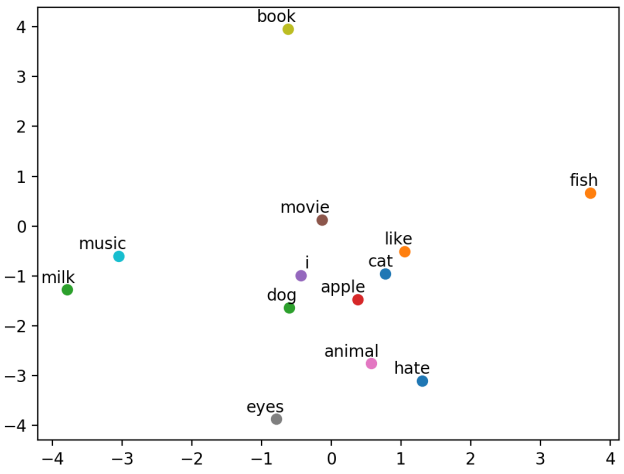


图 **5:** word2vec-res

维是远远不够的，这里为了快速展示出效果，同时该网络非常粗糙，训练也非常粗糙，对于大型语料来说，需要使用负样本采样方法进行加速网络的训练。

3 FastText

2016 年，Facebook 提出了针对文本提出了 FastText，该文提出了各种各样的训练技巧，使得训练速度加快很多，但是核心思想都是使用之前的 Embedding 方法，通过图 6来获得每一个 N-gram

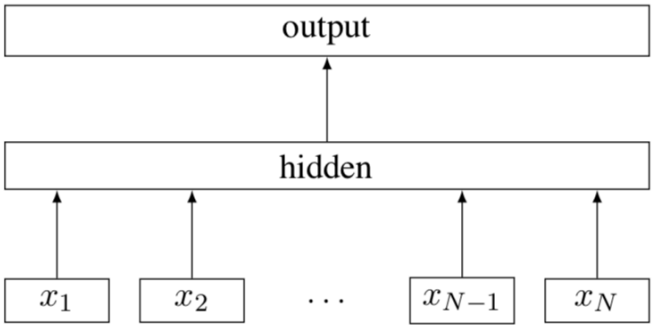


Figure 1: Model architecture of `fastText` for a sentence with N ngram features $x_1, \ldots, x_N$. The features are embedded and averaged to form the hidden variable.

图 6: fastText

的表示再进行训练。

4 TextCNN

该文章提出了一种使用 CNN(卷积) 的方式进行抽取文本特征进行分类，卷积操作在图像用的是非常多的，用于文本却不是很流行。该文章提出了文本层面的卷积操作。实验证明少量的参数也能够获得很好的效果。本质上对每一个词向量进行卷积，后面取最大化 Pooling，最后经过全连接层进行分类。图 7, 其对应简单代码为图 8

5 RNN

在处理自然语言，一个非常重要的网络就是循环递归网络，该网络的参数是共享的，同时上一层的输出同时作为下一层的部分输入，一直这样循环下去，直到序列全部喂进去。最后输出为每一个隐藏层的状态以及每一个序列的输出。图 9，代码图 10，代码实现的是最简单的循环递归网络。

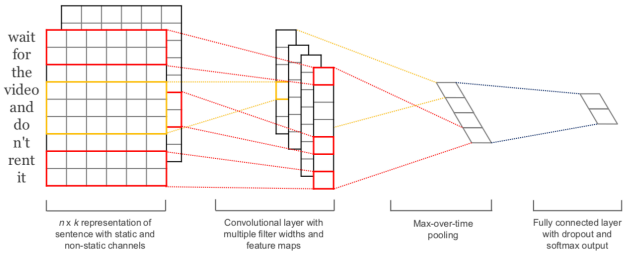


Figure 1: Model architecture with two channels for an example sentence.

图 **7:** textcnn

```python
class TextCNN(nn.Module):
    def __init__(self):
        super(TextCNN, self).__init__()

        self.num_filters_total = num_filters * len(filter_sizes)
        self.W = nn.Parameter(torch.empty(vocab_size, embedding_size).uniform_(-1, 1))
        self.Weight = nn.Parameter(torch.empty(self.num_filters_total, num_classes).uniform_(-1, 1))
        self.Bias = nn.Parameter(0.1 * torch.ones([num_classes]))

    def forward(self, x):
        embedded_chars = self.W[x] # bs,len,emb_size

        embedded_chars = embedded_chars.unsqueeze(1) # bs, 1, len, emb_size

        pooled_outputs = []

        for filter_size in filter_sizes:
            conv = nn.Conv2d(1, num_filters, (filter_size, embedding_size), bias=True)(embedded_chars)


            h = F.relu(conv)

            mp = nn.MaxPool2d((sequence_length - filter_size + 1, 1))

            pooled = mp(h)
            pooled = pooled.permute(0, 3, 2, 1)

            pooled_outputs.append(pooled)

        h_pool = torch.cat(pooled_outputs, dim=len(filter_sizes))

        h_pool_flat = torch.reshape(h_pool, [-1, self.num_filters_total])

        out = torch.mm(h_pool_flat, self.Weight) + self.Bias

        return out
```

图 **8:** textcnn-code

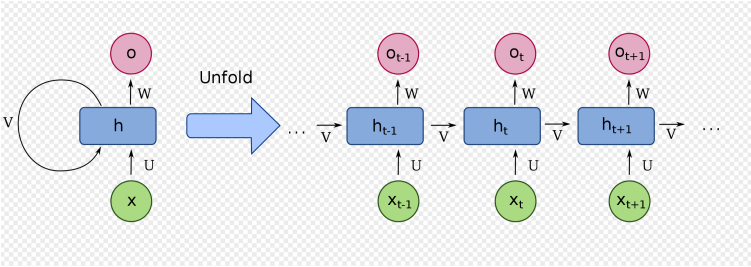


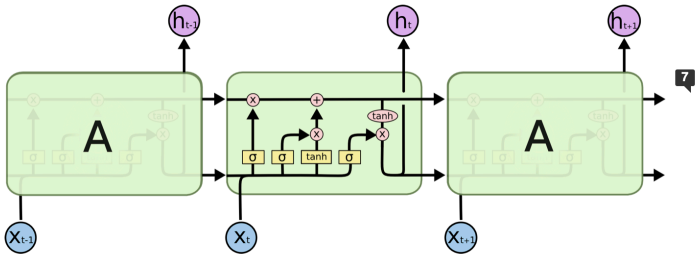
图 **9:** rnn

```python
class TextRNN(nn.Module):
    def __init__(self):
        super(TextRNN, self).__init__()

        self.rnn = nn.RNN(input_size=n_class, hidden_size=n_hidden)
        self.W = nn.Parameter(torch.randn((n_hidden, n_class)))
        self.b = nn.Parameter(torch.randn(n_class))

    def forward(self, hidden, x):
        # bs, len, v
        X = x.transpose(0, 1)
        outputs, hidden = self.rnn(X, hidden)
        outputs = outputs[-1]
        model = torch.mm(outputs, self.W) + self.b
        return model
```

图 **10:** rnn 代码

6 LSTM

由于 RNN 容易出现梯度消失，也就是当序列长度非常长的时候，梯度累加要么消失要么爆炸，因此 RNN 适合短序列任务，当序列非常长的时候，使用 LSTM 进行计算。计算与 RNN 是一致的，但是内部却多了几个门，能够很好的控制梯度的流传。图 11而bi-LSTM则是 LSTM 的升级版，普通



The repeating module in an LSTM contains four interacting layers.

图 11: lstm 示例图

的 LSTM 只能够前向传播，也就是只能捕捉历史信息，而不能捕捉未来信息，而 bi-LSTM 能够同时捕捉双向信息，表面上看是没有没有意义的，因为大部分任务是预测未来啊，现在把未来信息添加进来不是信息泄露了么？其实自然语言任务中，远远不止预测未来词这一个任务，还有很多，比如命名体识别，分词、情感分类都是给定一个句子，那么预测中间某一个词的词性，那么双向信息都学习到更加能够准确预测，因此双向 LSTM 是非常具有前景的。图 12

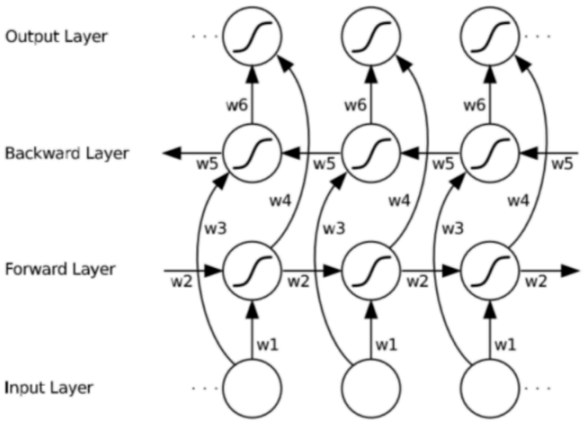


Fig. 3. Bidirectional LSTM

图 12: bi-lstm 示例图

7 seq2seq

seq2seq 是自然语言发展的一个里程碑，该技术的提出，使得翻译质量进一步提升，而不需要很多的人工处理。之前机器翻译极其依赖平行语料，需要大量的平行语料才能进行翻译，而大量的

平行语料需要人工去收集。而 seq2seq 的提出，使得句子翻译更加简单，比如英语到德语的翻译过程，先进行英语的编码，然后解码，从而输出翻译后的结果。图 13所示，将 seq 进行编码压缩，接

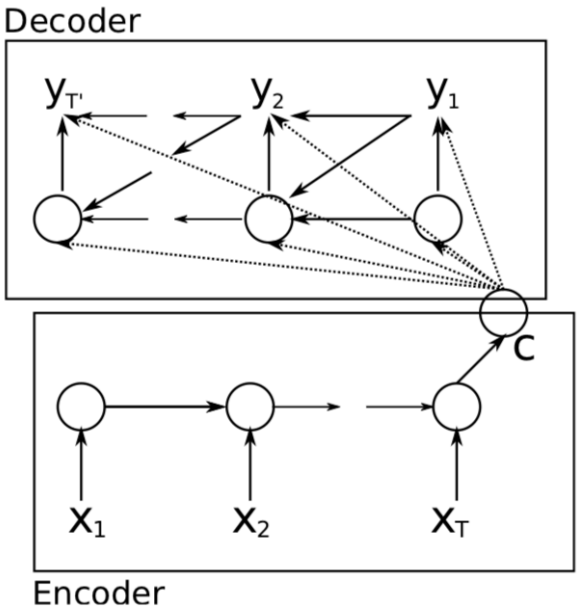


Figure 1: An illustration of the proposed RNN Encoder–Decoder.

图 13: seq2seq 示例图

下来进行解码。计算 LOSS 是解码的输出与真实的标签进行计算 loss。图 14

```
class Seq2Seq(nn.Module):
    def __init__(self):
        super(Seq2Seq, self).__init__()

        self.enc_cell = nn.RNN(input_size=n_class, hidden_size=n_hidden, dropout=0.5)
        self.dec_cell = nn.RNN(input_size=n_class, hidden_size=n_hidden, dropout=0.5)
        self.fc = nn.Linear(n_hidden, n_class)


    def forward(self, enc_input, enc_hidden, dec_input):
        enc_input = enc_input.transpose(0, 1)
        dec_input = dec_input.transpose(0, 1)

        _, enc_states = self.enc_cell(enc_input, enc_hidden)

        outputs, _ = self.dec_cell(dec_input, enc_states)

        out = self.fc(outputs)

        return out
```

图 14: seq2seq-code

8 Attention

Attention 机制的提出，使得自然语言处理的翻译质量进一步提升，从此真正进入了机器翻译时代，该机制采用了注意力，想象一下，从小我们学习英语，有一个重要的法则就是**就近原则**，也就是

后面的内容与前面的内容不是非常相关的，也就是对前面的注意力下降，而对附近的信息需要加大注意力。该论文提出了 **Transformer**，一种基于注意力机制的框架，同时该框架没有循环网络以及卷积操作，因此训练速度非常快，但是由于自然语言天生具有时序机制，因此论文提出了 **Position Encoding**。将每一个词进行编码，使得其具有位置信息。

$$PE_{(pos,2i)} = \sin\left(\text{pos}/10000^{2i/d_{\text{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d_{\text{modd}}}\right)$$

，这就是位置信息。整体的 Transformer 模型为图 15，左边为 **Encoder**，同时右边为 **Decoder**，而其

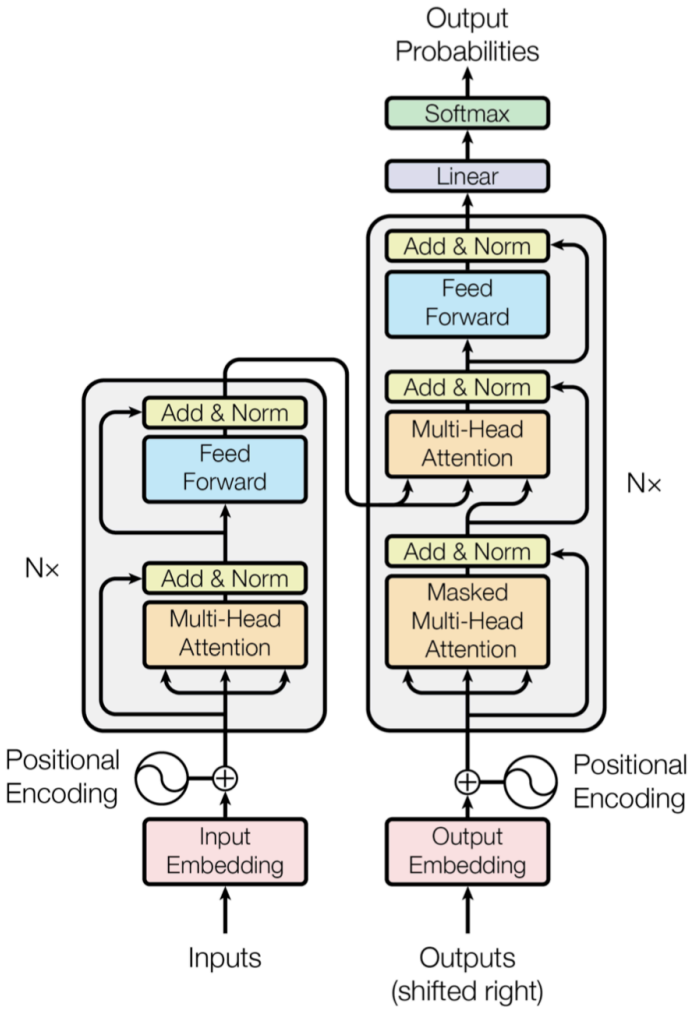


图 15: Transformer 示例图

中的 Encoder 以及 Decoder 使用的 **Multi-Head Attention** 图 16，而其中的 **Feed Forward** 网络可以使用卷积来代替，也就是卷积核为 1 的卷积，来代替全连接层。文章表示，该框架效果非常棒，但是训练非常慢。

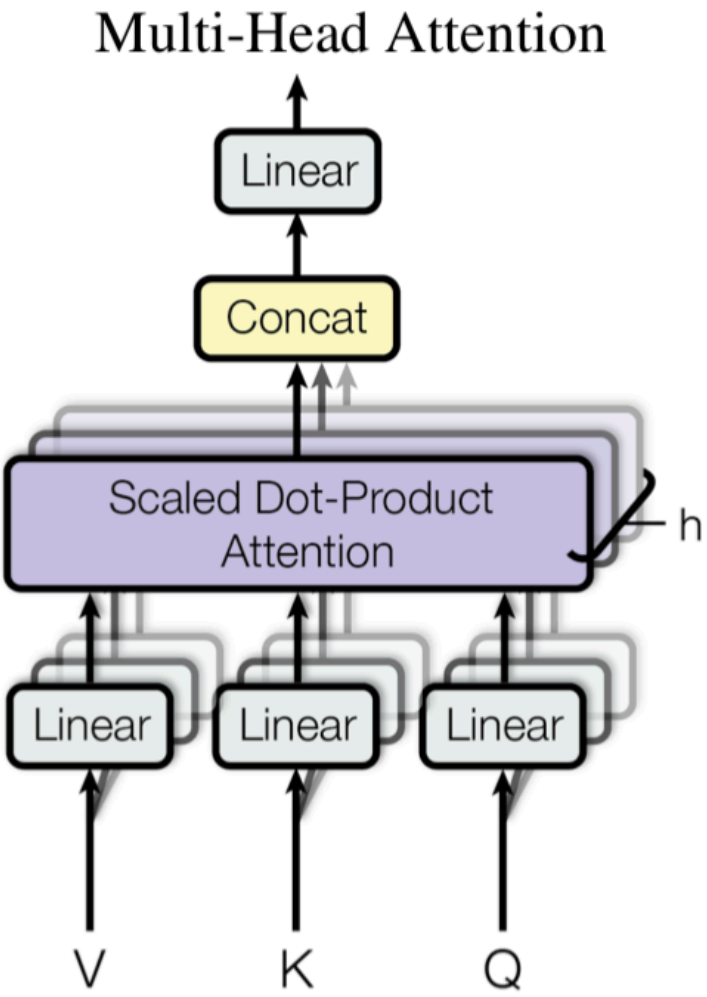


图 16: Multi-Head 示例图

9 BERT

2018 年底最棒的文章，刷新了 11 个自然语言处理的任务记录，而 BERT 全称为 **Bidirectional Encoder Representations from Transformers**，而该模型是基于 Transformers 的，与前面工作不同，BERT 采用了连接左右上下层所有的信息，因此参数更多，同时也更加难以训练。但是由于此模型的强大，当 BERT 训练完成，也就是预训练模型完成，对于各种各样的任务只需要最后一层就能够进行微调，从而能够应用于各种各样的自然语言处理任务。

9.1 Data

在 BERT 中，数据的准备是非常有意思的。使用了"[CLS],[SEP],[PAD]" 来准备数据。其中 CLS 为分类嵌入，表示了该句子的最终隐藏状态，用作分类任务，如果不是分类任务，则可以忽略。随机选择两个句子，拼在一起成为单独的序列。而拼的过程为 **[CLS] + A + [SEP] + B + [SEP]**，同时使用 **segment** 来表示句子的属性 (属于第一个还是属于第二个)，图 **17**, 通过这种方式产生一半正样

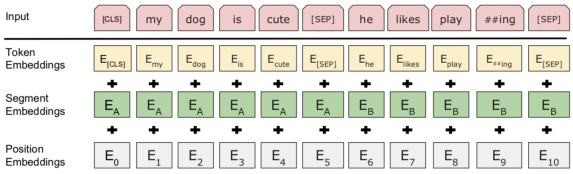


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

图 17: bert-data 示例图

本，一半负样本 (如果两个句子是上下文关系就是正样本，反之负样本)。同时生成数据的同时，随机替换 MASK 词。

- 80% 的情况下: 用 [MASK] 替换被选择的单词，例如，my dog is hairy → to my dog is [MASK]
- 10% 的情况下: 用一个随机单词替换被选择的单词，例如，my dog is hairy → my dog is apple
- 10% 的情况下: 保持被选择的单词不变，例如，my dog is hairy → my dog is hairy。这样做的目的是使表示偏向于实际观察到的词。

那么任务就是给定一个句子，第一预测上下文关系是否正确，第二预测被 mask 掉的词能否被正确预测。训练的模型为图 **18**

```python
if __name__ == '__main__':
    model = BERT()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    batch = make_batch()

    input_ids, segment_ids, masked_tokens, masked_pos, isNext = map(torch.LongTensor, zip(*batch))

    for epoch in range(100):
        optimizer.zero_grad()
        logits_lm, logits_clsf = model(input_ids, segment_ids, masked_pos)
        loss_lm = criterion(logits_lm.transpose(1, 2), masked_tokens)
        loss_lm = (loss_lm.float()).mean()
        loss_clsf = criterion(logits_clsf, isNext)
        loss = loss_lm + loss_clsf # 上下文有没有预测对 以及 maks的词有没有预测对

        print(f'epoch:{epoch} -- loss:{loss.item()}')

        loss.backward()
        optimizer.step()


    input_ids, segment_ids, masked_tokens, masked_pos, isNext = batch[0]
    print(text)
    print([number_dict[w] for w in input_ids if number_dict[w] != '[PAD]'])

    logits_lm, logits_clsf = model(torch.LongTensor([input_ids]), \
                                   torch.LongTensor([segment_ids]), torch.LongTensor([masked_pos]))
    logits_lm = logits_lm.data.max(2)[1][0].data.numpy()
    print('masked tokens list : ',[pos for pos in masked_tokens if pos != 0])
    print('predict masked tokens list : ',[pos for pos in logits_lm if pos != 0])

    logits_clsf = logits_clsf.data.max(1)[1].data.numpy()[0]
    print('isNext : ', True if isNext else False)
    print('predict isNext : ',True if logits_clsf else False)
```

图 **18:** bert-code 示例图