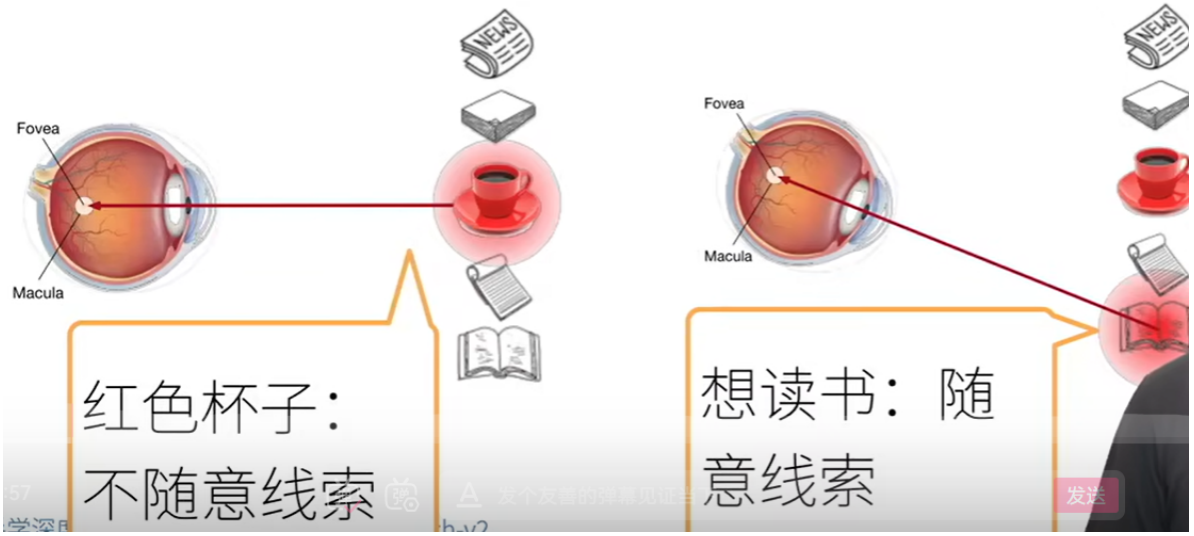
笔记：自然语言处理

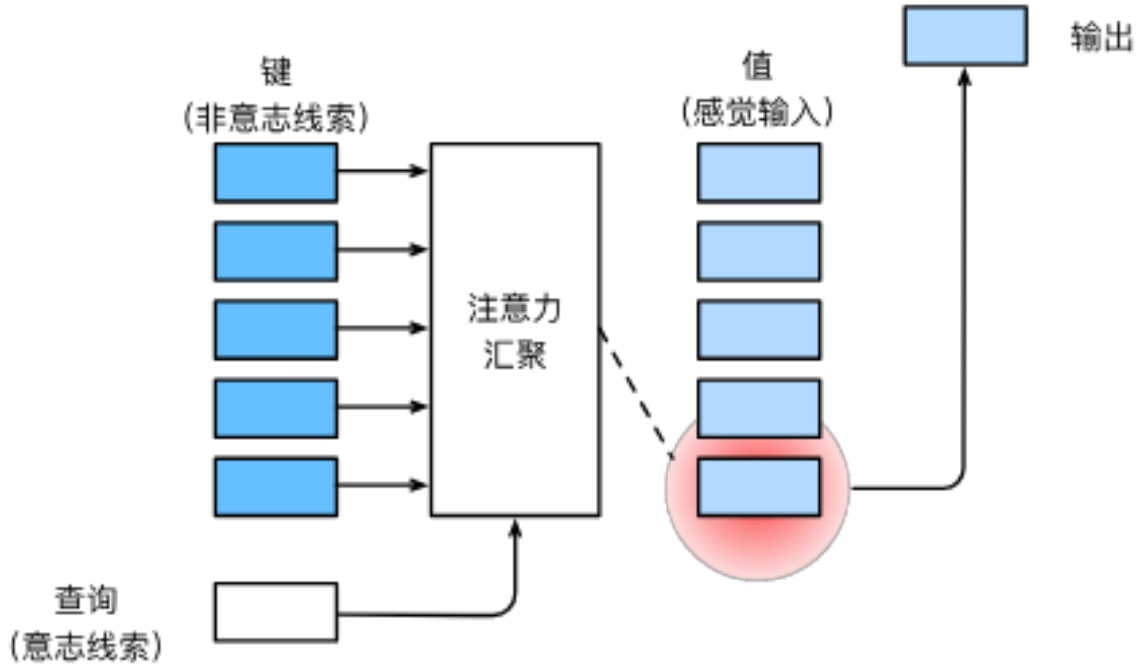
第十章 注意力机制

64 注意力机制

1.

- 卷积、全连接、池化层都只考虑不随意线索
- 

- 注意力机制考虑随意线索
  - 随意线索被称之为查询query
  - 每个输入是一个 值value和不随意线索key 的对
  - 通过注意力池化层来有偏向性地选择某些输入
  -



2.非参注意力池化层

- Nadaraya-Watson核回归
- $f(x)=\sum_{i=1}^{n}\frac{K(x-x_i)}{\sum_{j=1}^{n}K(x-x_i)}y_i$
- 其中x是查询，(xi,yi)是键值对
- K是核函数，用来计算距离，分数表示概率，即每个的相对重要性，离x越近的值越大，也就是说对于要拟合的x，通过这个系数找到离他比较近的点，给他们的y值比较大的权重，（获得了更多的注意力）最终得到一个加权平均
- 简单来说，根据输入的位置对输出yi进行加权
- 使用高斯核$K(u)=\frac{1}{\sqrt{2\pi}}exp(-\frac{u^2}{2})$
- $f(x)=\sum_{i=1}^{n}\frac{exp(-\frac{(x-x_i)^2}{2})}{\sum_{j=1}^{n}exp(-\frac{(x-x_j)^2}{2})}y_i$
- $=\sum_{i=1}^{n}softmax(-\frac{(x-x_i)^2}{2})y_i$

3.参数化的注意力机制

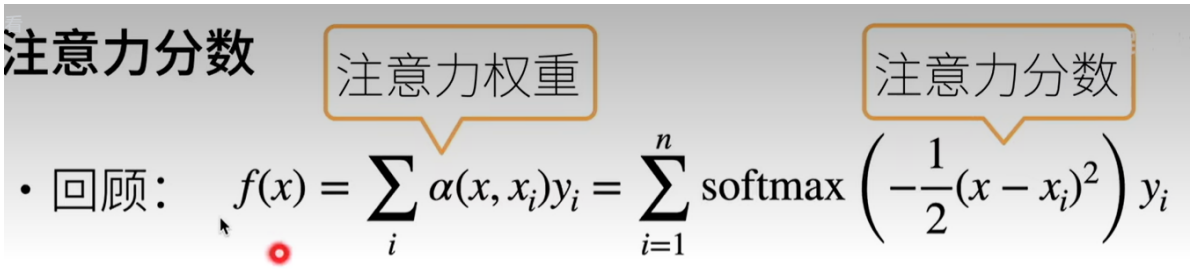
- 在之前的基础上引入可以学习的w
- $f(x)=\sum_{i=1}^{n}softmax(-\frac{((x-x_i)w)^2}{2})y_i$

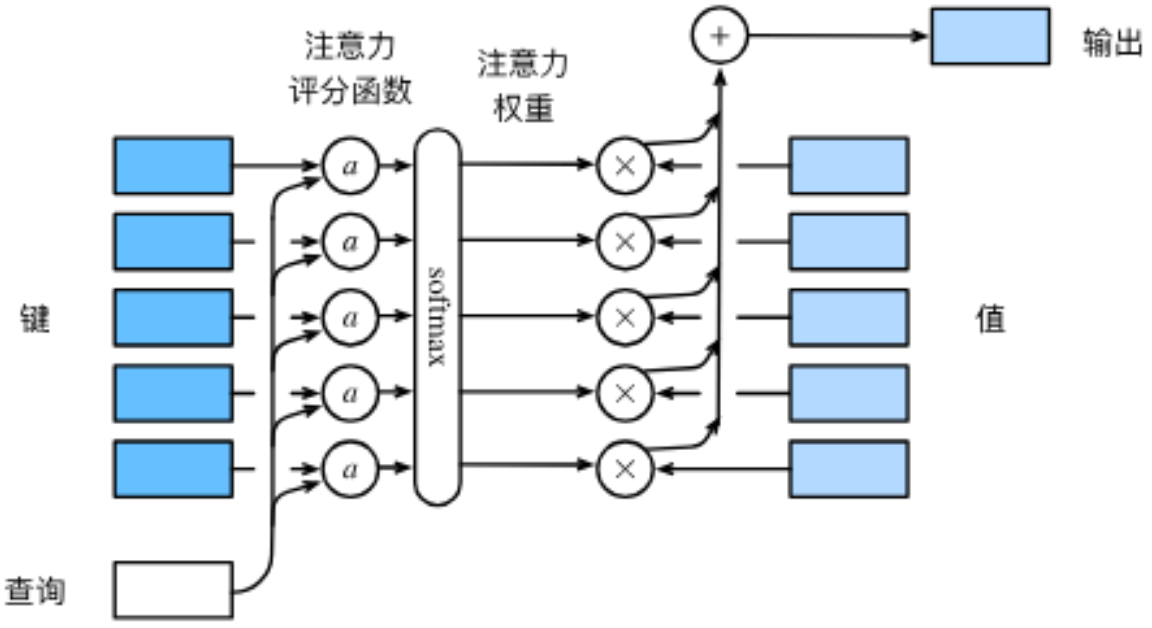
4.总结

- 受试者使用非自主性和自主性提示有选择性地引导注意力。前者基于突出性，后者则依赖于意识
- 注意力机制通过注意力汇聚使选择偏向于值（感官输入），其中包含查询（自主性提示）和键（非自主性提示）。键和值是成对的

65 注意力分数

1.注意力分数

- 

- 

2.Additive Attention

- 可学参数：$W_k \in \mathbb{R}^{h \times k}, W_q \in \mathbb{R}^{h \times q}, v \in \mathbb{R}^h$
- $a(k, q) = v^T tanh(W_k k + W_q q)$
- 等价于将key和query合并起来后放入到一个隐藏大小为h输出大小为1的单隐藏层MLP

3.Scaled Dot-Product Attention

- 如果query和key都是同样的长度d，$a(ki, q) =< q, ki > /\sqrt{d}$
- 分母保证对d没那么敏感

4.总结

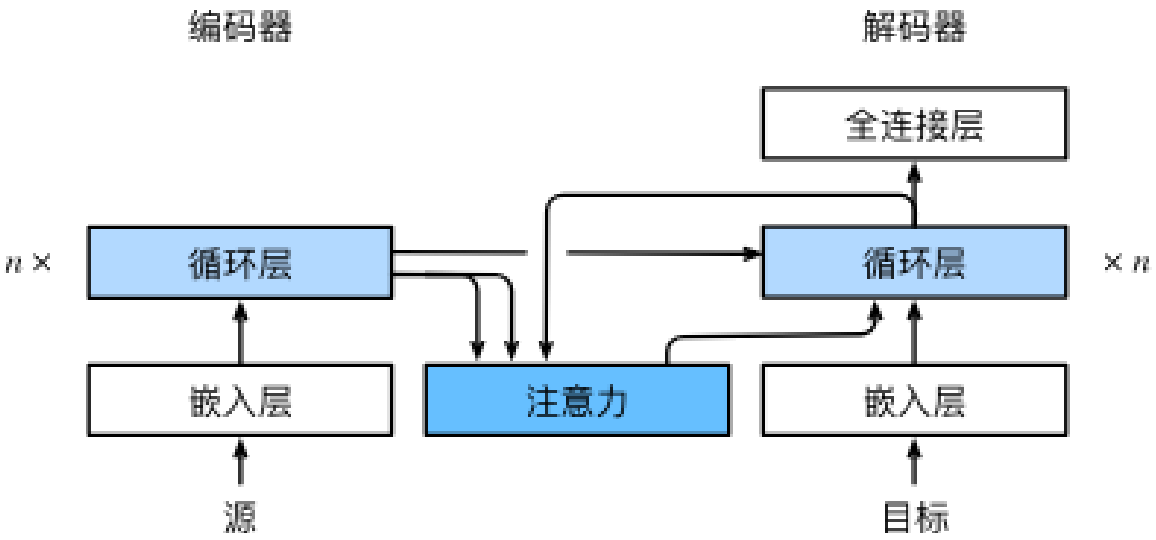
- 注意力分数是query和key的相似度，注意力权重是分数的softmax结果

66. 使用注意力机制的seq2seq

1.动机

- 机器翻译中，每个生成的词可能相关于源句子中不同的词，在seq2seq中源句子所有信息都被压缩在最后一个隐藏状态，引入注意力可以去关注需要的部分

  2.加入注意力



- 编码器对每次词的输出作为key和value
- 解码器RNN对上一个词的输出是query
- 注意力的输出和下一个词的词嵌入合并进入

  3.总结

- 注意力机制可以根据解码器RNN的输出来匹配到合适的编码器的RNN的输出来更有效地传递信息

  4.Bahdanau注意力

带有注意力机制的解码器基本接口

```
class AttentionDecoder(d2l.Decoder):
    """带有注意力机制的解码器基本接口"""
    def __init__(self, **kwargs):
        super(AttentionDecoder, self).__init__(**kwargs)

    @property
    def attention_weights(self):
        raise NotImplementedError
```

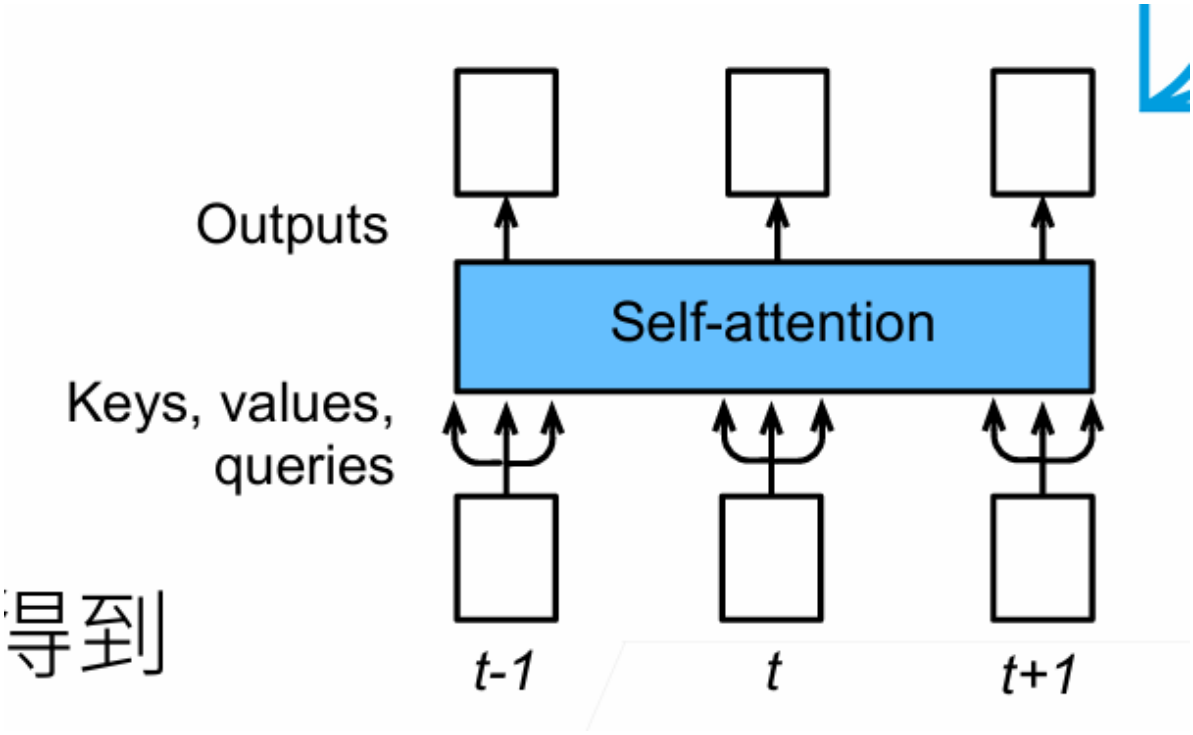
实现带有Bahdanau注意力的循环神经网络解码器

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
```

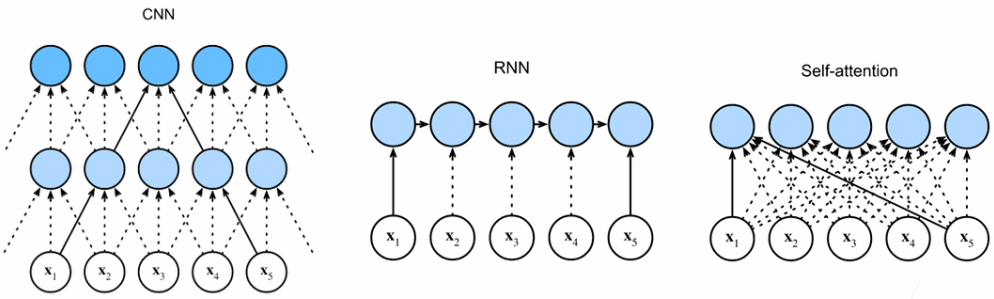
```
                 dropout=0, **kwargs):
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
        self.attention = d2l.AdditiveAttention(
            num_hiddens, num_hiddens, num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        # outputs的形状为(batch_size，num_steps，num_hiddens).
        # hidden_state的形状为(num_layers，batch_size，num_hiddens)
        outputs, hidden_state = enc_outputs
        return (outputs.permute(1, 0, 2), hidden_state, enc_valid_lens)

    def forward(self, X, state):
        # enc_outputs的形状为(batch_size,num_steps,num_hiddens).
        # hidden_state的形状为(num_layers,batch_size,
        # num_hiddens)
        enc_outputs, hidden_state, enc_valid_lens = state
        # 输出X的形状为(num_steps,batch_size,embed_size)
        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:
            # query的形状为(batch_size,1,num_hiddens)
            query = torch.unsqueeze(hidden_state[-1], dim=1)
            #解码器RNN对上一个词的输出
            # context的形状为(batch_size,1,num_hiddens)
            context = self.attention(
                query, enc_outputs, enc_outputs, enc_valid_lens)
            # 在特征维度上连结
            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
            # 将x变形为(1,batch_size,embed_size+num_hiddens)
            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)
        # 全连接层变换后，outputs的形状为
        # (num_steps,batch_size,vocab_size)
        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
                                          enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

67.自注意力

1.自注意力

- 给定序列x1,...,xn,xi作为key，value，query得到y1,...,yn
- yi=f(xi,(x1,x1),...(xn,xn))
- 

2.对比



	CNN	RNN	自注意力
计算复杂度	O(knd^2)	O(nd^2)	O(n^2d)
并行度	O(n)	O(1)	O(n)
最长路径	O(n/k)	O(n)	O(1)

k是窗口的大小，n是序列长度，d是词向量特征维度，d^2是矩阵乘法。最长路径指信息传递。并行度是每个计算是否依赖其他的输出。n^ 2d是每次q要和n组做乘法，每个长度为d。由此可以看出，自注意力最长路径短说明他擅长看很长的序列，但反映在计算复杂度上他需要很大的计算量（n方）

3.位置编码

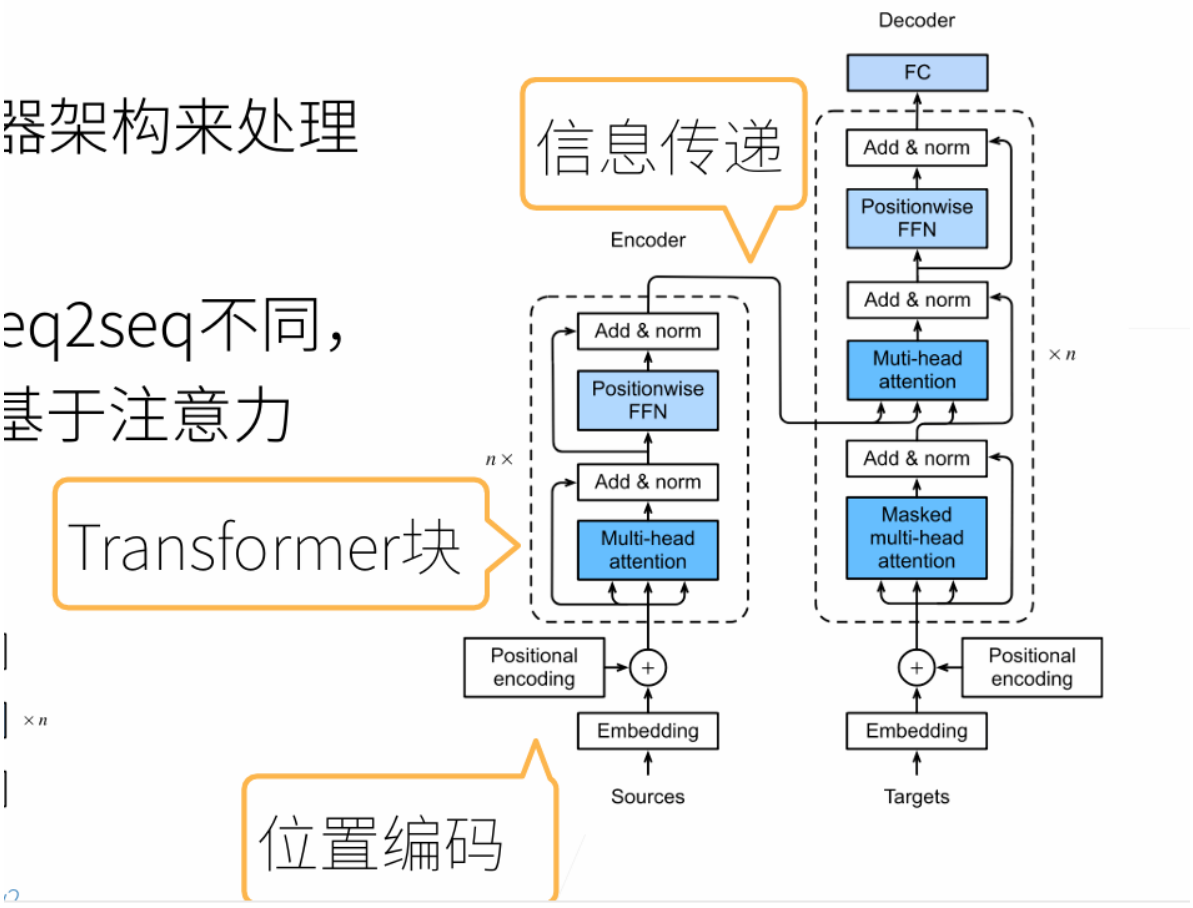
- 跟CNN/RNN不同，自注意力并没有记录位置信息
- 位置编码将位置信息注入到输入里
  - 假设长度为n的序列是：$X \in \mathbb{R}^{n\times d}$,那么使用位置编码矩阵$P \in \mathbb{R}^{n\times d}$来输出X+P作为自编码输入
- $p_{i,2j} = sin(\frac{i}{10000^{2j/d}}), p_{i,2j+1} = cos(\frac{i}{10000^{2j/d}})$

68.Transformer

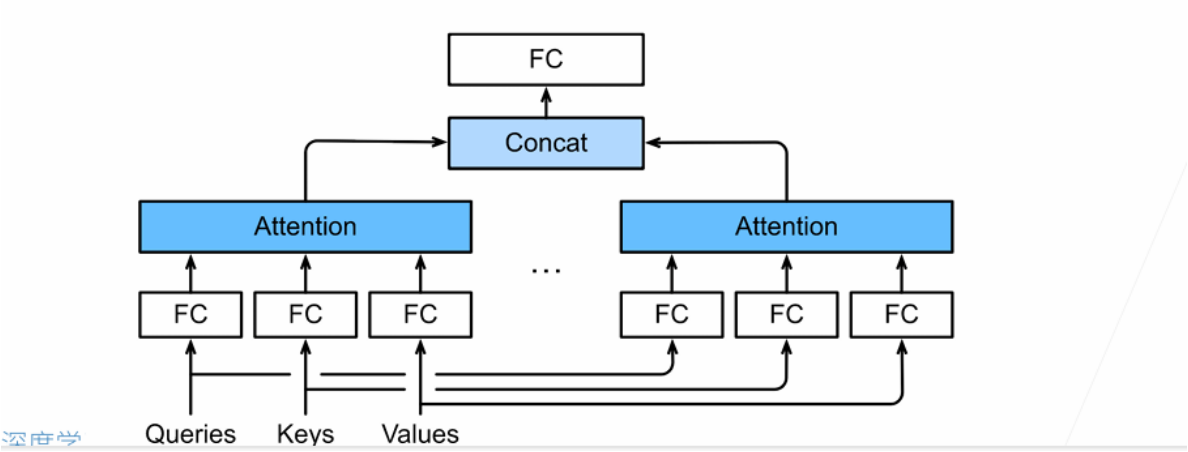
1.架构

纯基于注意力

- 编码器：多头自注意力
- 解码器：解码器自注意力，编码器-解码器注意力



2.多头注意力



对同一key，value，query，希望抽取不同的信息（例如短距离关系和长距离关系）

->多头注意力使用h个独立的注意力池化，合并各个头输出得到最终输出

3.基于位置的前馈网络

- 输入形状由（b,n,d）变换成（bn，d）
- 作用于两个全连接层
- 输出形状由（bn，d）变化回（b,n,d）
- 等价于两层核窗口为1的一维卷积层

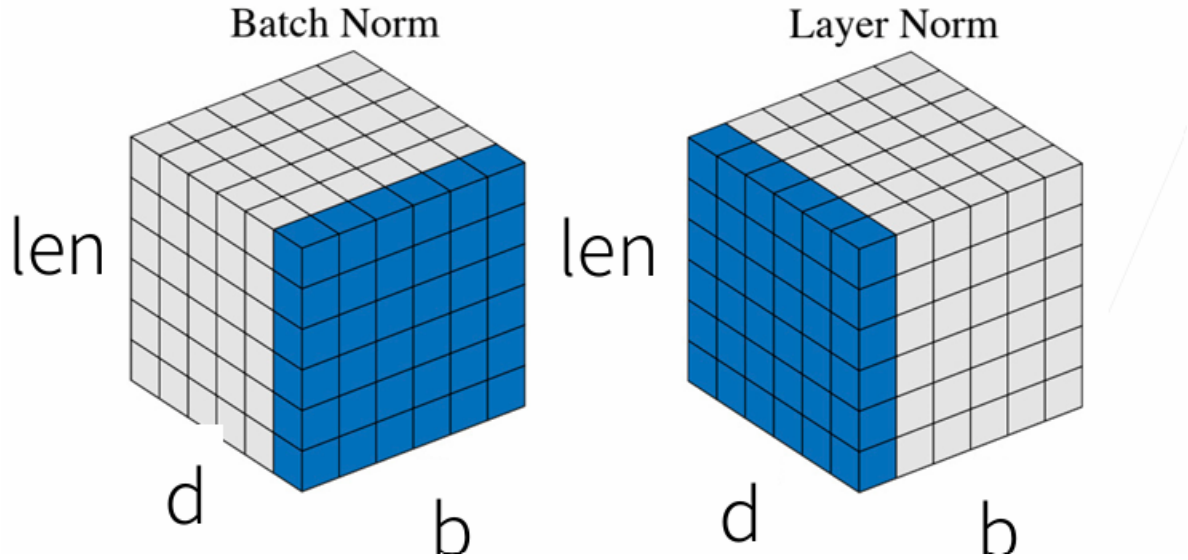
4.层归一化

- Add：ResNet
- Norm：层归一化

  - 批量归一化对每个特征/通道里元素进行归一化，不适合序列长度会变的NLP

  - 层归一化对每个样本里的元素进行归一化

  -

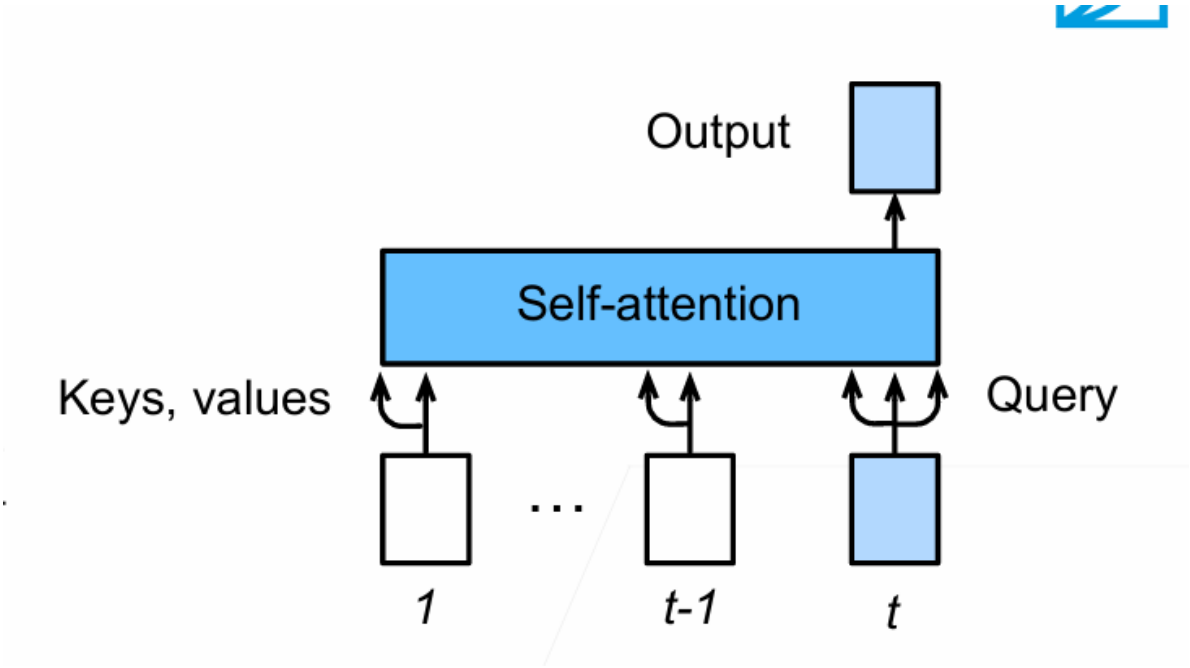


5.信息传递

- 编码器中的输出y1，...yn

- 将其作为解码器中第i个Transformer块中多头注意力的key和value，它的query来自目标序列
- 意味着编码器和解码器中块的个数和输出维度都是一样的

  6.预测
- 预测第t+1个输出时，解码器中输入前t个预测值。在自注意力中，前t个预测值作为key和value，第t个值还作为query



  7.代码

基于位置的前馈网络

```
class PositionWiseFFN(nn.Module):
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs,
                 **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

使用残差连接和层归一化

```
class AddNorm(nn.Module):
    def __init__(self, normalized_shape, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
```

```
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

实现编码器中的一个层

```
class EncoderBlock(nn.Module):
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(key_size, query_size,
                                                value_size, num_hiddens,
                                                num_heads, dropout,
use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
                                   num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        #这里q，k，v都是自己
        return self.addnorm2(Y, self.ffn(Y))
```

Transformer编码器

```
class TransformerEncoder(d2l.Encoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module(
                "block" + str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, use_bias))

    def forward(self, X, valid_lens, *args):
```

```
        X = self.pos_encoding(self.embedding(X) *
math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

Transformer解码器也是由多个相同的层组成

```
class DecoderBlock(nn.Module):
    """解码器中第i个块"""
    '''类比之前使用注意力机制的seq2seq，第一个attention层相当于原来的rnn层，第二
个attention层就是attention层'''
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads,
dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads,
dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
                                   num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # 训练阶段，输出序列的所有词元都在同一时间处理，q，k，v都是X
        # 因此state[2][self.i]初始化为None。
        # 预测阶段，输出序列是通过词元一个接着一个解码的，合并X和前面的输出
        # 因此state[2][self.i]包含着直到当前时间步第i个块解码的输出表示
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # dec_valid_lens的开头:(batch_size,num_steps),
            # 其中每一行是[1,2,...,num_steps]
            dec_valid_lens = torch.arange(
```

```
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None

        # 自注意力
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # 编码器－解码器注意力。
        # enc_outputs的开头:(batch_size,num_steps,num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

Transformer解码器

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                DecoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, i))
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        return [enc_outputs, enc_valid_lens, [None] * self.num_layers]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) *
math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range
(2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # 解码器自注意力权重
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # “编码器－解码器”自注意力权重
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
```

```
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

第十四章 自然语言处理：预训练

69 BERT预训练

1.NLP里的迁移学习

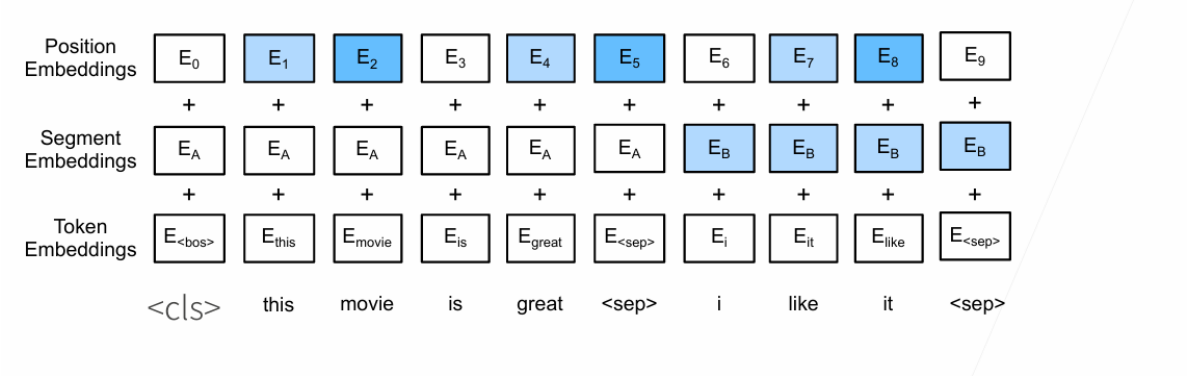
- 使用预训练好的模型来抽取词、句子的特征
  - 例如word2vec或语言模型
- 不更新预训练好的模型
- 需要构建新的网络来抓取新任务需要的信息

  - word2vec忽略了时序信息，语言模型只看了一个方向

  2.BERT的动机
- 基于微调的NLP模型
- 预训练的模型抽取了足够多的信息
- 新的任务只需要增加一个简单的输出层

  3.BERT架构
- 只有编码器的Transformer
- 两个版本：
  - Base：blocks=12，hidden size=768，heads=12，parameters=110M
  - Large：blocks=24，hidden size=1024，heads=16，parameters=340M
- 在大规模数据上训练>3B词

  4.对输入的修改
- 因为只有编码器，所以source和target都得给编码器
- 每个样本是一个句子对
- 加入额外的片段嵌入，用segment embedding分隔
- 位置编码可学习，每个token有一个position embedding



5.预训练任务1：带掩码的语言模型

- Transformer的编码器是双向的，标准语言模型要求单向
- 带掩码的语言模型每次随机（15%概率）将一些词元换成mask，这样Transformer的编码器仍然可以做双向，相当于完形填空
- 因为微调任务中不会出现mask，所以预训练中
  - 80%概率，将选中的词元变成mask
  - 10%概率，换成一个随机词元
  - 10%概率，保持原有的词元

6.预训练任务2：下一个句子预测

- 预测一个句子对中两个句子是否相邻
- 训练样本中：50%概率选择相邻句子对，50%概率选择随机句子对
- 将cls对应输出放到一个全连接层预测

7.代码

Input Representation

```
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """Get tokens of the BERT input sequence and their segment IDs."""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

BERTEncoder class

```
class BERTEncoder(nn.Module):
    """BERT encoder."""
```

```
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module(f"{i}", d2l.EncoderBlock(
                key_size, query_size, value_size, num_hiddens, norm_shape,
                ffn_num_input, ffn_num_hiddens, num_heads, dropout, True))
        self.pos_embedding = nn.Parameter(torch.randn(1, max_len,
                                                      num_hiddens))

    def forward(self, tokens, segments, valid_lens):
        X = self.token_embedding(tokens) + self.segment_embedding(segments)
        X = X + self.pos_embedding.data[:, :X.shape[1], :]
        for blk in self.blks:
            X = blk(X, valid_lens)
        return X
```

Masked Language Modeling

把要预测的位置的词所对应的encoder的输出拿到这里来预测位置的值

```
class MaskLM(nn.Module):
    """The masked language model task of BERT."""
    def __init__(self, vocab_size, num_hiddens, num_inputs=768, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential(nn.Linear(num_inputs, num_hiddens),
                                 nn.ReLU(),
                                 nn.LayerNorm(num_hiddens),
                                 nn.Linear(num_hiddens, vocab_size))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
        batch_size = X.shape[0]
        batch_idx = torch.arange(0, batch_size)
        batch_idx = torch.repeat_interleave(batch_idx, num_pred_positions)
        masked_X = X[batch_idx, pred_positions]
        masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
        mlm_Y_hat = self.mlp(masked_X)
        return mlm_Y_hat
```

Next Sentence Prediction

```
class NextSentencePred(nn.Module):
    """The next sentence prediction task of BERT."""
    def __init__(self, num_inputs, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Linear(num_inputs, 2)

    def forward(self, X):
        return self.output(X)
```

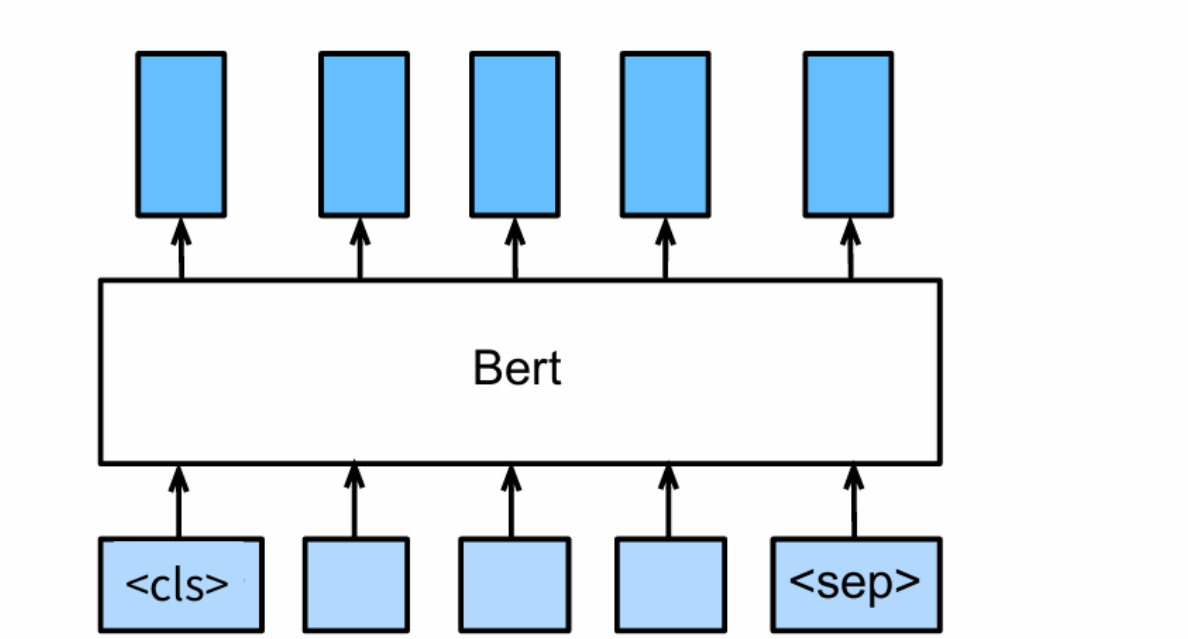
Putting All Things Together

```
class BERTModel(nn.Module):
    """The BERT model."""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 hid_in_features=768, mlm_in_features=768,
                 nsp_in_features=768):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape,
                    ffn_num_input, ffn_num_hiddens, num_heads, num_layers,
                    dropout, max_len=max_len, key_size=key_size,
                    query_size=query_size, value_size=value_size)
        self.hidden = nn.Sequential(nn.Linear(hid_in_features, num_hiddens),
                                    nn.Tanh())
        self.mlm = MaskLM(vocab_size, num_hiddens, mlm_in_features)
        self.nsp = NextSentencePred(nsp_in_features)

    def forward(self, tokens, segments, valid_lens=None,
pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

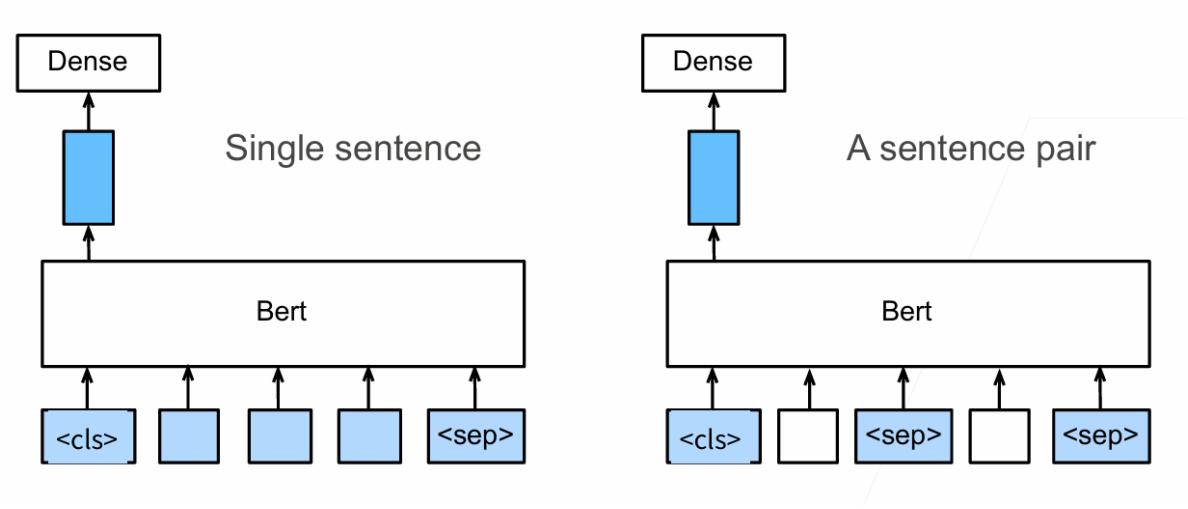
第十五章 自然语言处理：应用

70 BERT 微调

1.

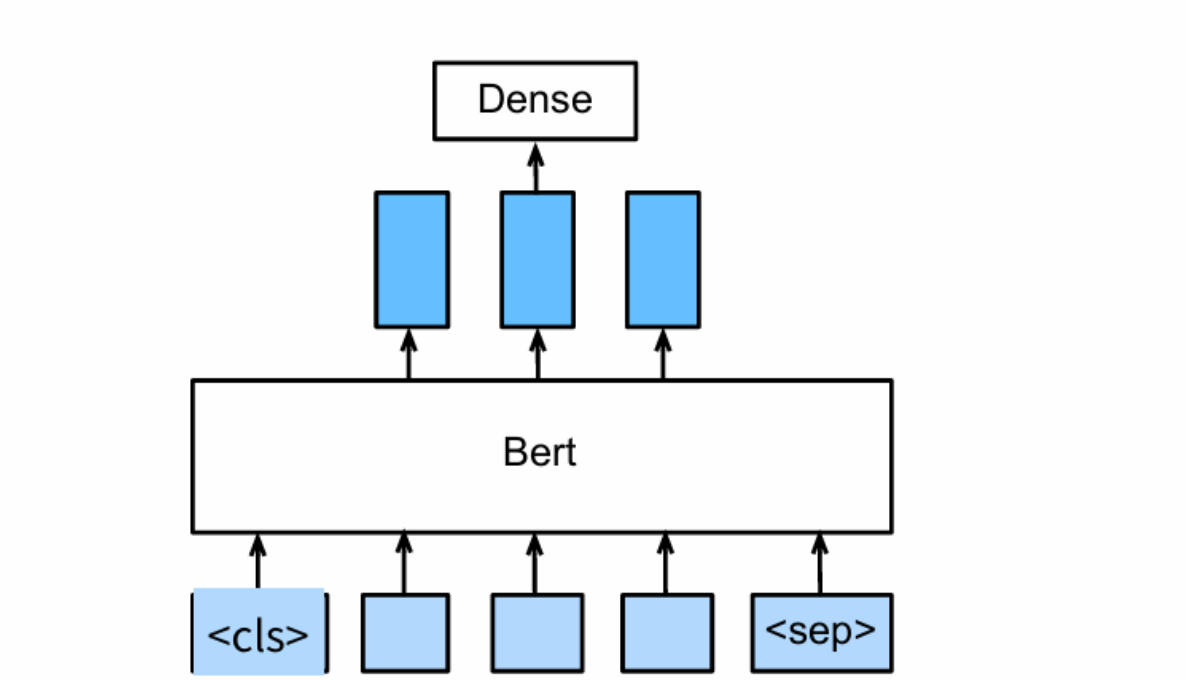
- BERT对每一个词元返回抽取了上下文信息的特征向量
- 不同的任务使用不同的特征
- 

2.句子分类

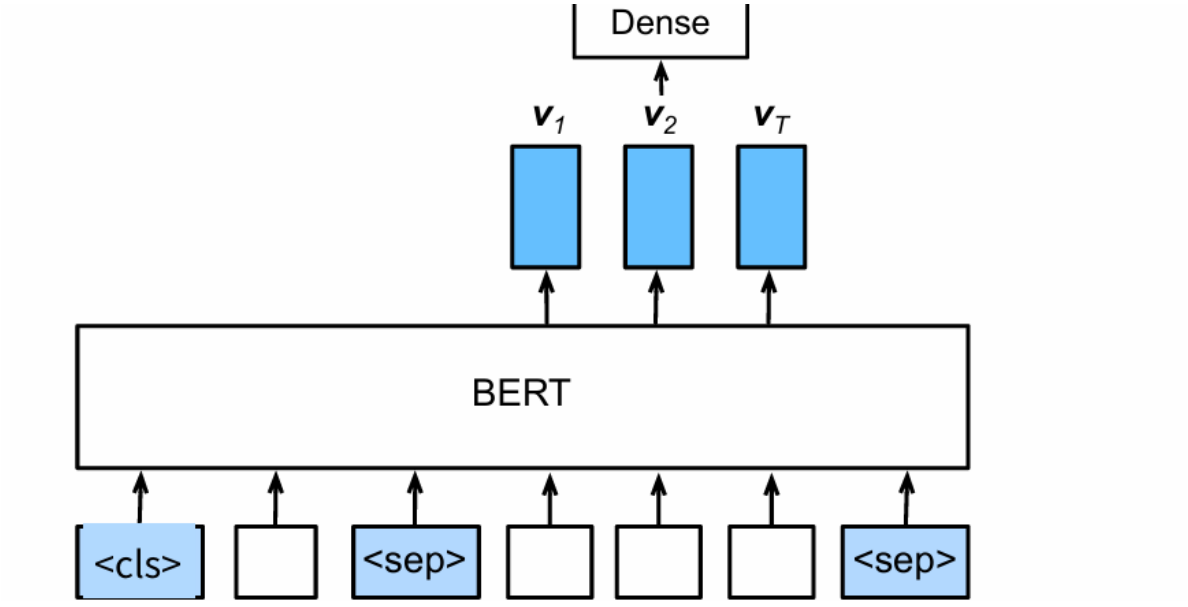
- 将cls对应的向量输入到全连接层分类
- 

3.命名实体识别

- 识别一个词元是不是命名实体，例如人名、机构、位置
- 将非特殊词元放进全连接层分类

- 

4.问题回答

- 给定一个问题，和描述文字，找出一个片段作为回答
- 对片段中的每个词元预测它是不是回答的开头和结尾（三分类问题）
- 

- 前面是问题，后面是描述

第十一章 优化算法

1.优化问题

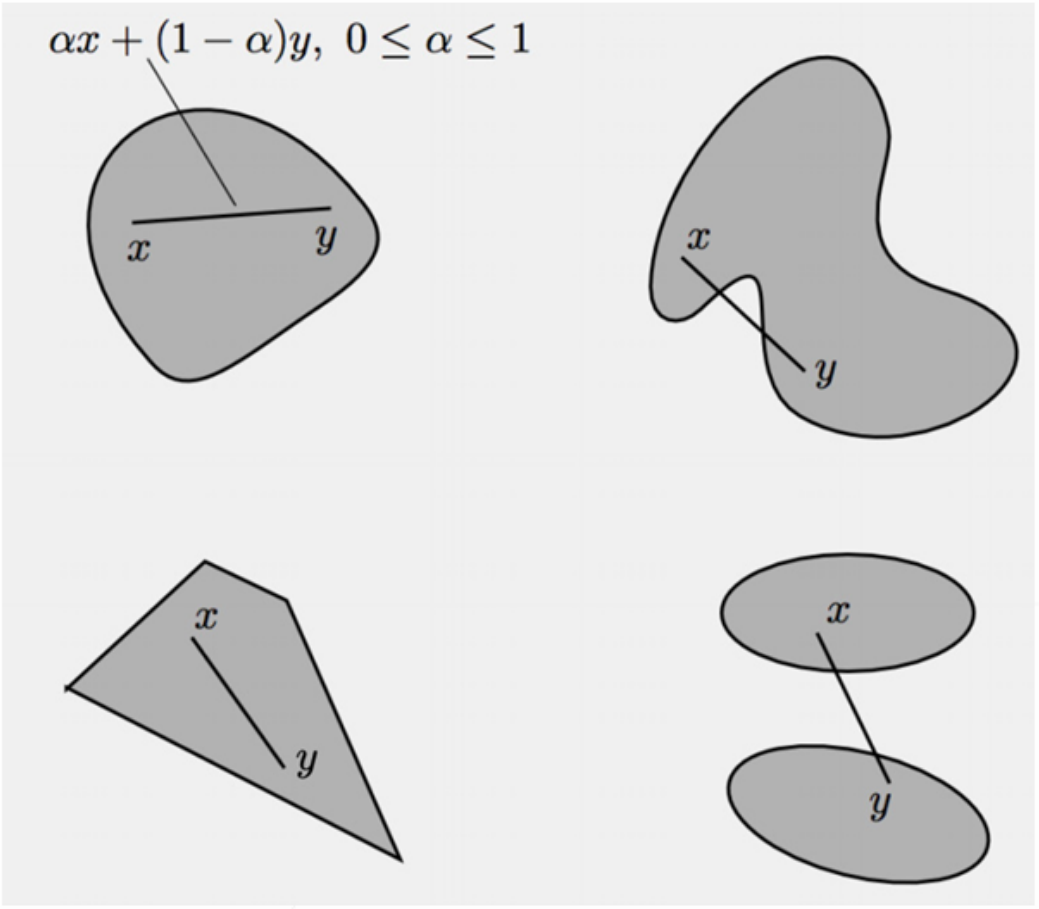
- 一般形式：minimize f（x） subject to x ∈C

2.局部最小vs全局最小

- 迭代算法一般只能找到局部最小

  3.凸集

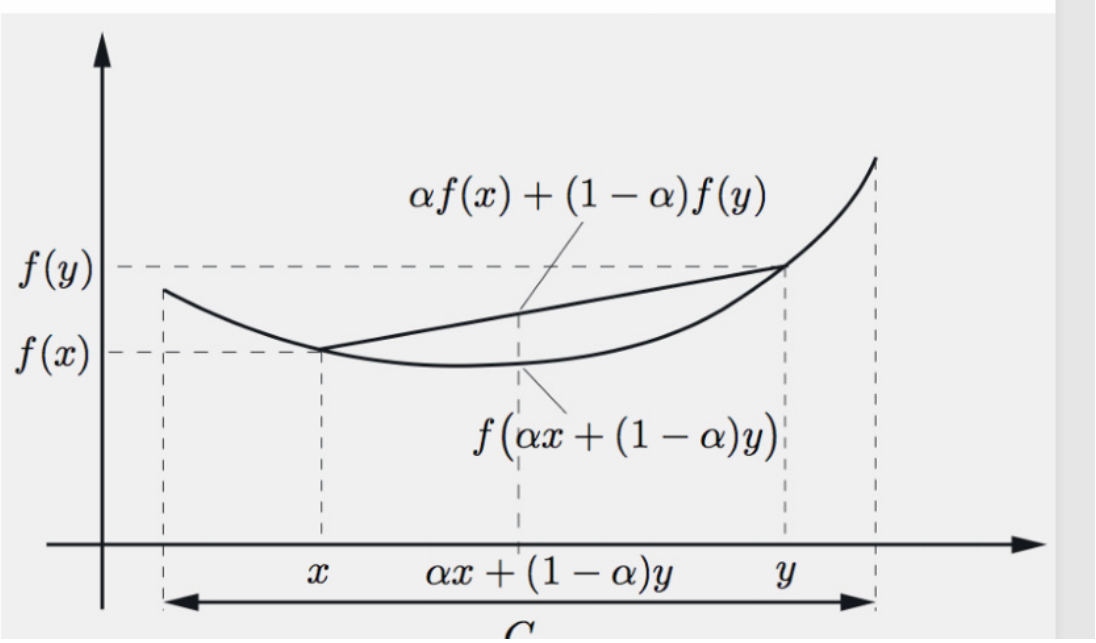
- $\alpha x+(1-\alpha)y\in C\forall\alpha\in[0,1]\forall x,y\in C$



-

  4.凸函数

- $f(\alpha x+(1-\alpha)y)\leqslant\alpha f(x)+(1-\alpha)f(y)\in C\forall\alpha\in[0,1]\forall x,y\in C$

- 

5.凸函数优化

- 如果代价函数f是凸的，且限制集合c是凸的，那么就是凸优化问题，那么局部最小就是全局最小
- 严格凸优化问题有唯一的全局最小

6.例子

- 凸：线性回归，sofmax
- 非凸：其他：MLP,CNN,RNN,attention

7.冲量法

- 不会一下很猛地改变很多，因为每次改变的方向不仅取决于当前的梯度，还却决于之前的梯度

冲量法

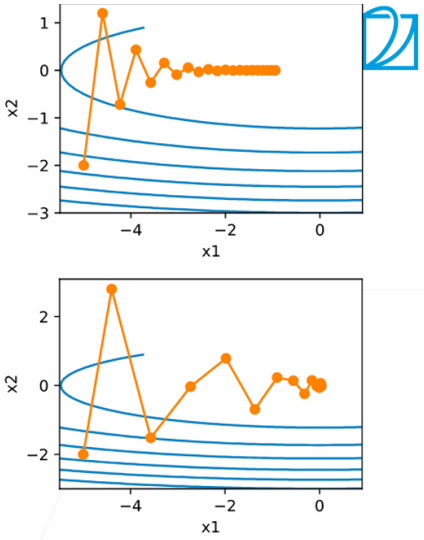
- 冲量法使用平滑过的梯度对权重更新

$$\mathbf{g}_t = \frac{1}{b} \sum_{i \in I_t} \nabla \ell_i(\mathbf{x}_{t-1})$$

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_t \quad \mathbf{w}_t = \mathbf{w}_{t-1} - \eta \mathbf{v}_t$$

梯度平滑：$\mathbf{v}_t = \mathbf{g}_t + \beta \mathbf{g}_{t-1} + \beta^2 \mathbf{g}_{t-2} + \beta^3 \mathbf{g}_{t-3} + \ldots$

- β 常见取值 [0.5, 0.9, 0.95, 0.99]



8.Adam

- 对学习率不那么敏感，非常平滑
- $v_t = \beta_1 v_{t-1} + (1-\beta_1) g_t, \beta_1 = 0.9$
- $v_t = (1-\beta_1)(g_t + \beta_1 g_{t-1} + \beta_1^2 g_{t-2} + \beta_1^3 g_{t-3} + \dots)$
- 无穷等比求和（betan次方）->权重和为（1-beta）1/（1-beta）=1
- $\sum_{i=0}^{t} \beta_1^i = \frac{1-\beta_1^t}{1-\beta_1}$->在t比较小时修正$\hat{v_t} = \frac{v_t}{1-\beta_1^t}$
- $s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2, \beta_2 = 0.999$
- 在t比较小时修正$\hat{v_t} = \frac{v_t}{1-\beta_2^t}$
- 计算重新调整后的梯度$g_t' = \frac{\hat{v_t}}{\sqrt{\hat{s_t}+\epsilon}}$，分子是使得变化比较平滑，分母是和分子制衡，使得不同维度梯度变化差不多，不会让太大的影响比较小的
- 最后用这个梯度更新