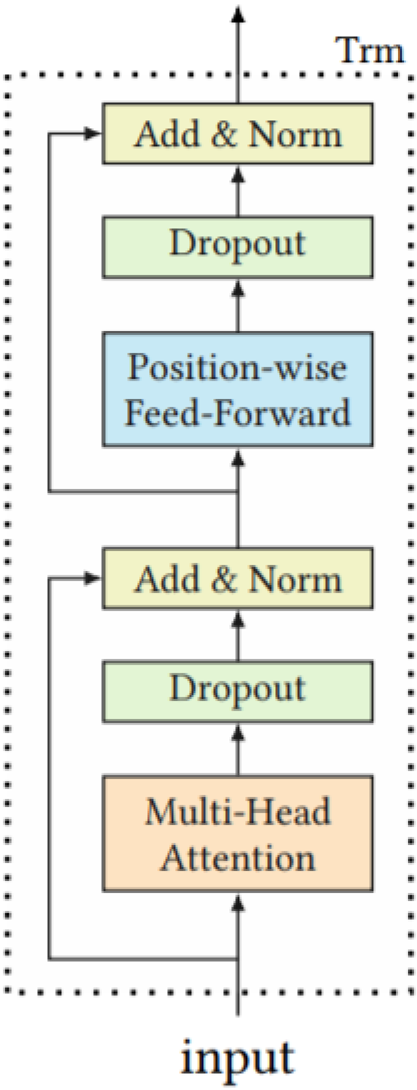
1. Transformer的整体结构

当我们对每个位置的token都有一个hidden_size大小的embedding之后，把这个序列输入到Transformer中。由以下的代码可以知道，每层Transformer都先计算multi-head self-attention，之后的结果再经过dropout, residual connection, layernorm, 还有feed-forward network。



(a) Transformer Layer.

```
def transformer_model(input_tensor,  ##[batch_size,seq_length,hidden_size]
                      attention_mask=None, ##[batch_size, seq_length,seq_length],1表示这个位置可以被关注到,0表示不可以
                      hidden_size=768,
                      num_hidden_layers=12,
                      num_attention_heads=12,
                      intermediate_size=3072,
```

```
                        intermediate_act_fn=gelu, ##the output of the intermediate/feed-forward layer激活函数
                        hidden_dropout_prob=0.1, ##FFN的dropout
                        attention_probs_dropout_prob=0.1, ##self-attention算QKV的dropout
                        initializer_range=0.02,
                        do_return_all_layers=False):

    if hidden_size % num_attention_heads != 0: ##hidden state需要按attention head的数量进行拆分，所以必须要能够整除
        raise ValueError(
            "The hidden size (%d) is not a multiple of the number of attention "
            "heads (%d)" % (hidden_size, num_attention_heads))

    attention_head_size = int(hidden_size / num_attention_heads) ## 拆分attention_head，每个attention head的大小
    input_shape = get_shape_list(input_tensor, expected_rank=3)
    batch_size = input_shape[0] ## 128, batchsize
    seq_length = input_shape[1] ## 20,sequence length
    input_width = input_shape[2] ## 32, hidden size

    # We keep the representation as a 2D tensor to avoid re-shaping it back and
    # forth from a 3D tensor to a 2D tensor. Re-shapes are normally free on
    # the GPU/CPU but may not be free on the TPU, so we want to minimize them to
    # help the optimizer.
    prev_output = reshape_to_matrix(input_tensor) ##(128*20,32), 记录上一层输出

    all_layer_outputs = []
    for layer_idx in range(num_hidden_layers): ## 多层Transformer
        with tf.variable_scope("layer_%d" % layer_idx):
            layer_input = prev_output ##此层的输入是上一层的输出
            with tf.variable_scope("attention"):
                attention_heads = []
                with tf.variable_scope("self"):
                    attention_head = attention_layer(  ### multi-head self-attention层
                        from_tensor=layer_input,
                        to_tensor=layer_input,
                        attention_mask=attention_mask, ##attention mask 只有decoder会用
                        num_attention_heads=num_attention_heads,
                        size_per_head=attention_head_size,
                        attention_probs_dropout_prob=
                        attention_probs_dropout_prob,
                        initializer_range=initializer_range,
                        do_return_2d_tensor=True,
                        batch_size=batch_size,
                        from_seq_length=seq_length,
                        to_seq_length=seq_length)
                    attention_heads.append(attention_head)

                attention_output = None
                assert len(attention_heads) == 1
                attention_output = attention_heads[0]  ##(20*128,32), 经过self-attention层的结果

                # Add & Norm层
```

```
                # Run a linear projection of `hidden_size` then add a residual
                # with `layer_input`.
                with tf.variable_scope("output"):
                    attention_output = tf.layers.dense(
                        attention_output,
                        hidden_size, ##和输入input的hidden size一样，没有升维
                        kernel_initializer=create_initializer(
                            initializer_range))
                    attention_output = dropout(attention_output,
                                               hidden_dropout_prob) ##隐藏层dropout
                    attention_output = layer_norm(attention_output + ## 残差连接
+layernorm
                                                  layer_input)
            # FFN层
            # The activation is only applied to the "intermediate" hidden layer.
            with tf.variable_scope("intermediate"):
                intermediate_output = tf.layers.dense(
                    attention_output,
                    intermediate_size, ##一般取4*hiddensize
                    activation=intermediate_act_fn, ###"gelu"
                    kernel_initializer=create_initializer(initializer_range))

            # Down-project back to `hidden_size` then add the residual.
            with tf.variable_scope("output"):
                layer_output = tf.layers.dense(
                    intermediate_output, ##4*hidden size
                    hidden_size, ##降维到hiddensize
                    kernel_initializer=create_initializer(initializer_range))
                layer_output = dropout(layer_output, hidden_dropout_prob)
                layer_output = layer_norm(layer_output + attention_output)
                prev_output = layer_output ##这一层的输出已经得到了!
                all_layer_outputs.append(layer_output)

    if do_return_all_layers:
        final_outputs = []
        for layer_output in all_layer_outputs:
            final_output = reshape_from_matrix(layer_output, input_shape)
            final_outputs.append(final_output)
        return final_outputs
    else:
        final_output = reshape_from_matrix(prev_output, input_shape) ##reshape得到三维输
出(128,20,32)
        return final_output
```

常见面试题:

- **Feed forward network (FFN)的作用?**

答：Transformer在抛弃了 LSTM 结构后，FFN 中的激活函数成为了一个主要的提供**非线性**变换的单元。

- **GELU原理？相比RELU的优点?**

答：ReLU会确定性的将输入乘上一个0或者1(当x<0时乘上0，否则乘上1)，Dropout则是随机乘上0。而GELU虽然也是将输入乘上0或1，但是输入到底是乘以0还是1，是在取决于输入自身的情况下随机选择的。

什么意思呢？具体来说：

我们将神经元的输入 x乘上一个服从伯努利分布的 m 。而该伯努利分布又是依赖于 x的：

$$m \sim Bernoulli(\Phi(x)$$

)

其中， $X \sim N(0,1)$，那么 $\Phi(x)$就是标准正态分布的累积分布函数。这么做的原因是因为神经元的输入x往往遵循正态分布，尤其是深度网络中普遍存在Batch Normalization的情况下。当x减小时，$\Phi(x)$的值也会减小，此时x被“丢弃”的可能性更高。所以说这是随机依赖于输入的方式。

现在，给出GELU函数的形式：

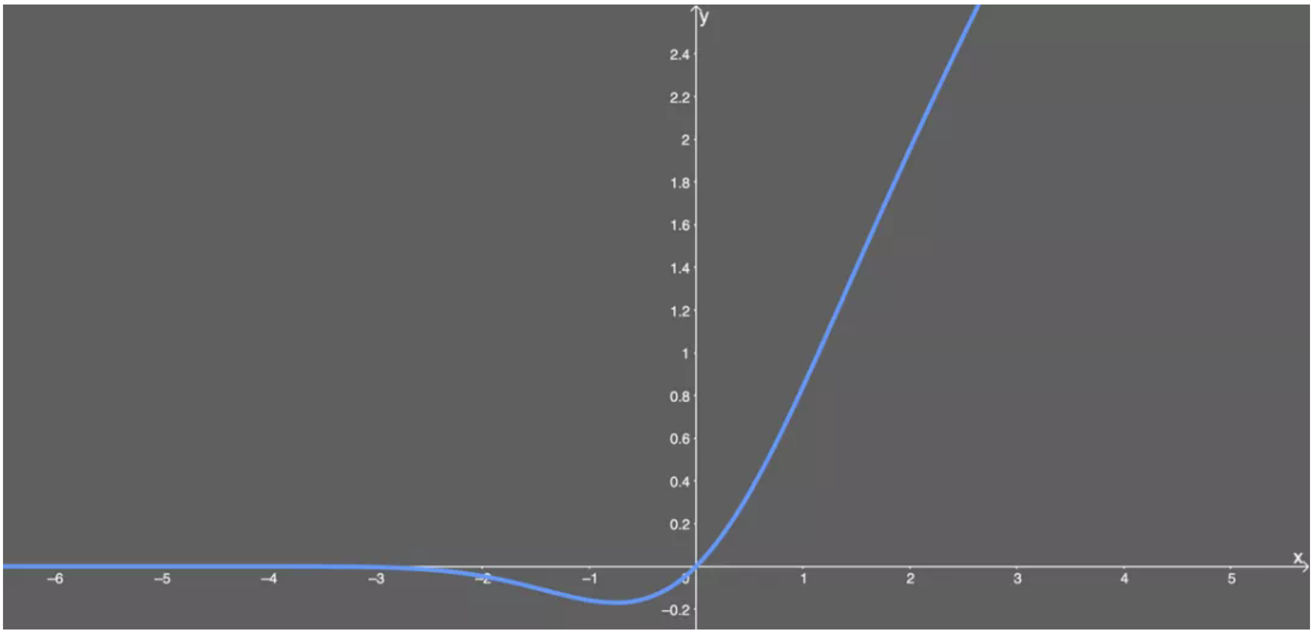
$$GELU(x) = x\Phi(x)$$

其中

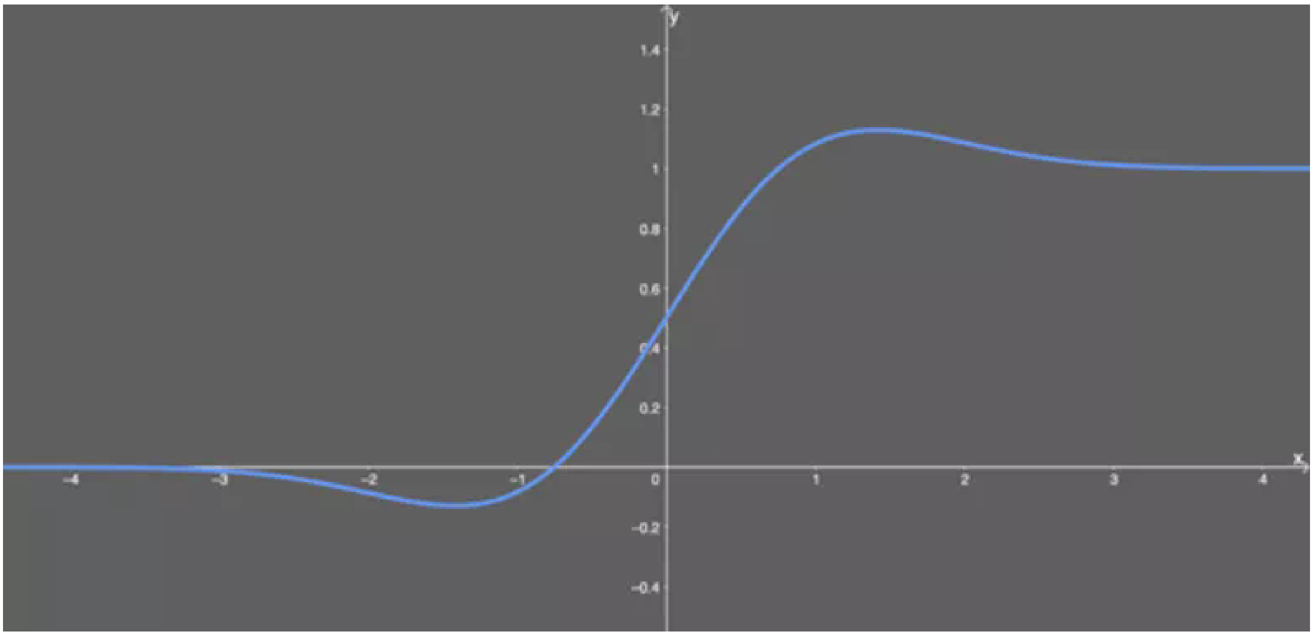
$$\Phi(x)$$

\Phi(x) 是上文提到的标准正态分布的累积分布函数。因为这个函数没有解析解，所以要用近似函数来表示。

图像：



导数形式：



和RELU一样，可以解决梯度消失

所以，GELU的优点就是在ReLU上增加随机因素，x越小越容易被mask掉。

- **为什么用layernorm不用batchnorm?**

答：对于RNN来说，sequence的长度是不一致的，所以用很多padding来表示无意义的信息。如果BN会导致有意义的embedding损失信息。所以，BN一般用于CNN，而LN用于RNN。

layernorm是在hidden size的维度进行的，跟batch和seq_len无关。每个hidden state都计算自己的均值和方差，这是因为不同hidden state的量纲不一样。beta和gamma的维度都是(hidden_size,)，经过白化的hidden state * beta + gamma得到最后的结果。

LN在BERT中主要起到白化的作用，增强模型稳定性（如果删除则无法收敛）

2. Multi-head Self-Attention

```
def attention_layer(from_tensor,  ##[128*20,32]
                    to_tensor,  ##[128*20,32]
                    attention_mask=None,
                    num_attention_heads=1,
                    size_per_head=512,
                    query_act=None,
                    key_act=None,
                    value_act=None,
                    attention_probs_dropout_prob=0.0,
                    initializer_range=0.02,
                    do_return_2d_tensor=False,
                    batch_size=None,
                    from_seq_length=None,
                    to_seq_length=None):

    def transpose_for_scores(input_tensor, batch_size, num_attention_heads,
                             seq_length, width):
```

```
    output_tensor = tf.reshape(
        input_tensor, [batch_size, seq_length, num_attention_heads, width])

    output_tensor = tf.transpose(output_tensor, [0, 2, 1, 3])
    return output_tensor

from_shape = get_shape_list(from_tensor, expected_rank=[2, 3])
to_shape = get_shape_list(to_tensor, expected_rank=[2, 3])

if len(from_shape) != len(to_shape):
    raise ValueError(
        "The rank of `from_tensor` must match the rank of `to_tensor`.")

if len(from_shape) == 3:
    batch_size = from_shape[0]  ##128
    from_seq_length = from_shape[1] ##20
    to_seq_length = to_shape[1] ##20
elif len(from_shape) == 2:
    if (batch_size is None or from_seq_length is None
            or to_seq_length is None):
        raise ValueError(
            "When passing in rank 2 tensors to attention_layer, the values "
            "for `batch_size`, `from_seq_length`, and `to_seq_length` "
            "must all be specified.")

# Scalar dimensions referenced here:
#   B = batch size (number of sequences)
#   F = `from_tensor` sequence length
#   T = `to_tensor` sequence length
#   N = `num_attention_heads`
#   H = `size_per_head`

from_tensor_2d = reshape_to_matrix(from_tensor) ##(128*20,32)
to_tensor_2d = reshape_to_matrix(to_tensor)##(128*20,32)

# `query_layer` = [B*F, N*H]
query_layer = tf.layers.dense( ##用不带激活函数的dense来模拟矩阵相乘，得到Query
    from_tensor_2d,
    num_attention_heads * size_per_head,
    activation=query_act, ##None
    name="query",
    kernel_initializer=create_initializer(initializer_range)) ####(128*20,32)


# `key_layer` = [B*T, N*H]
key_layer = tf.layers.dense(##用不带激活函数的dense来模拟矩阵相乘，得到Key
    to_tensor_2d,
    num_attention_heads * size_per_head,
    activation=key_act, #None
    name="key",
    kernel_initializer=create_initializer(initializer_range))##(128*20,32)

# `value_layer` = [B*T, N*H]
```

```
    value_layer = tf.layers.dense(##用不带激活函数的dense来模拟矩阵相乘，得到value
        to_tensor_2d,
        num_attention_heads * size_per_head,
        activation=value_act, #None
        name="value",
        kernel_initializer=create_initializer(initializer_range))##(128*20,32)

    # `query_layer` = [B, N, F, H]
    query_layer = transpose_for_scores(query_layer, batch_size,
                                       num_attention_heads, from_seq_length,
                                       size_per_head)  ##[128,2,20,16]
    # `key_layer` = [B, N, T, H]
    key_layer = transpose_for_scores(key_layer, batch_size,
                                     num_attention_heads, to_seq_length,
                                     size_per_head)##[128,2,20,16]

    # Take the dot product between "query" and "key" to get the raw
    # attention scores.
    # `attention_scores` = [B, N, F, T]
    attention_scores = tf.matmul(query_layer, key_layer, transpose_b=True) ##
[128,2,20,20]
    attention_scores = tf.multiply(attention_scores,
                                   1.0 / math.sqrt(float(size_per_head)))

    if attention_mask is not None:
        # `attention_mask` = [B, 1, F, T]
        attention_mask = tf.expand_dims(attention_mask, axis=[1])

        # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
        # masked positions, this operation will create a tensor which is 0.0 for
        # positions we want to attend and -10000.0 for masked positions.
        adder = (1.0 - tf.cast(attention_mask, tf.float32)) * -10000.0  ##-infty

        # Since we are adding it to the raw scores before the softmax, this is
        # effectively the same as removing these entirely.
        attention_scores += adder

    # Normalize the attention scores to probabilities.
    # `attention_probs` = [B, N, F, T]
    attention_probs = tf.nn.softmax(attention_scores) ##[128,2,20,20]

    # This is actually dropping out entire tokens to attend to, which might
    # seem a bit unusual, but is taken from the original Transformer paper.
    attention_probs = dropout(attention_probs, attention_probs_dropout_prob)

    # `value_layer` = [B, T, N, H]
    value_layer = tf.reshape(
        value_layer,
        [batch_size, to_seq_length, num_attention_heads, size_per_head])

    # `value_layer` = [B, N, T, H]
    value_layer = tf.transpose(value_layer, [0, 2, 1, 3]) ##(128,2,20,16)
```

```
    # `context_layer` = [B, N, F, H]
    context_layer = tf.matmul(attention_probs, value_layer)

    # `context_layer` = [B, F, N, H]
    context_layer = tf.transpose(context_layer, [0, 2, 1, 3])

    if do_return_2d_tensor:
        # `context_layer` = [B*F, N*V]
        context_layer = tf.reshape(context_layer, [
            batch_size * from_seq_length, num_attention_heads * size_per_head
        ])
    else:
        # `context_layer` = [B, F, N*V]
        context_layer = tf.reshape(
            context_layer,
            [batch_size, from_seq_length, num_attention_heads * size_per_head])
    return context_layer  ##[128*20,32]
```

如果是单头注意力，就是每个位置的embedding对应 Q, K, V三个向量，这三个向量分别是embedding点乘 W_Q, W_K, W_V 矩阵得来的。每个位置的Q向量去乘上所有位置的K向量，其结果经过softmax变成attention score，以此作为权重对所有V向量做加权求和即可。

用公式表示为：

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

其中， d_k 为 Q, K 向量的hidden size。除以 d_k 叫做**scaled dot product.**

那么多头注意力是怎样的呢？

Transformer中先通过切头（**spilt**）再分别进行Scaled Dot-Product Attention。

step1：一个768维的hidden向量，被映射成Q，K，V。 然后三个向量分别切分成12(head_num)个小的64维的向量，每一组小向量之间做self-attention。不妨假设batch_size为32，seqlen为512，隐层维度为768，12个head。

hidden(32 x 512 x 768) -> **Q**(32 x 512 x 768) -> 32 x 12 x 512 x 64

hidden(32 x 512 x 768) -> **K**(32 x 512 x 768) -> 32 x 12 x 512 x 64

hidden(32 x 512 x 768) -> **V**(32 x 512 x 768) -> 32 x 12 x 512 x 64

step2：然后Q和K之间做attention，得到一个32 x 12 x 512 x 512的权重矩阵（时间复杂度$O(n^2d)$），然后根据这个权重矩阵加权V中切分好的向量，得到一个32 x 12 x 512 x 64 的向量，拉平输出为768向量。

32 x 12 x 512 x 64(query_hidden) * 32 x 12 x 64 x 512(key_hidden) -> 32 x 12 x 512 x 512

32 x 12 x 64 x 512(value_hidden) * 32 x 12 x 512 x 512 (权重矩阵) -> 32 x 12 x 512 x 64

然后再还原成 -> 32 x 512 x 768

简言之是12个头，每个头都是一个64维度，分别去与其他的所有位置的hidden embedding做attention然后再合并还原。

常见面试题：

- **为什么要做scaled dot product?**

答：当输入信息的维度 d 比较高，会导致 softmax 函数接近饱和区，梯度会比较小。因此，缩放点积模型可以较好地解决这一问题。

- **为什么用双线性点积模型（即Q，K两个向量）**

双线性点积模型使用Q，K两个向量，而不是只用一个Q向量，这样引入非对称性，更具健壮性（Attention对角元素值不一定是最大的，也就是说当前位置对自身的注意力得分不一定最高）。

- **多头机制为什么有效?**

类似于CNN中通过多通道机制进行特征选择。Transformer中使用切头(split)的方法，是为了在不增加复杂度（

$$O(n^2d)$$

）的前提下享受类似CNN中"不同卷积核"的优势。

- **Transformer的非线性来自于哪里?**

FFN的gelu激活函数和self-attention，注意self-attention是非线性的（因为有相乘和softmax）。

Transformer复杂度分析

1. self-attention复杂度

记：序列长度为n，一个位置的embedding大小为d。例如，(32,512,768)的序列，n = 512, d = 768.

首先，得到的Q,K,V都是大小为n*d的。

- 相似度计算 QK^T： $n \times d$ 与 $d \times n$ 运算，得到 $n \times n$ 矩阵，复杂度为 $\mathcal{O}(n^2d)$
- softmax计算：对每行做softmax，复杂度为 $\mathcal{O}(n)$，则n行的复杂度为 $\mathcal{O}(n^2)$
- 乘上V加权： $n \times n$ 与 $n \times d$ 运算，得到 $n \times d$ 矩阵，复杂度为 $\mathcal{O}(n^2d)$

2. 多头self-attention复杂度

- Attention操作复杂度：首先经过"切头"，把输出变成(batchsize, n, d/h)长度， QK^T 就是(n,d/h)和(n,d/h)的运算，由于h为常数，复杂度为 $\mathcal{O}(n^2d)$
- 之后的softmax和乘V加权同上。
- 之后，还需要把这些头拼接起来，经过一层线性映射之后输出。concat操作拼起来形成 $n \times d$ 的矩阵，然后经过输出线性映射，保证输出也是 $n \times d$ 的，所以是 $n \times d$ 与 $d \times d$ 计算，复杂度为

故最后的复杂度为：

$$\mathcal{O}(n^2d + nd^2)$$

RNN 复杂度分析

$$h_t = f(Ux_t + Wh_{t-1})$$

- Ux_t ： $d \times m$ 与 $m \times 1$ 运算，复杂度为 $\mathcal{O}(md)$ ， m 为输入x_t的embedding size，d为hidden state的embedding size
- Wh_{t-1} ： $d \times d$ 与 $d \times 1$ 运算，复杂度为 $\mathcal{O}(d^2)$

故一次操作的时间复杂度为 $\mathcal{O}(d^2)$ ， n 次序列操作后的总时间复杂度为

CNN复杂度分析

> 使用conv1d，这里保证输入输出都是一样的，即均是
> $$n \times d$$

- 大小为 $k \times d$ 的卷积核一次运算的复杂度为： $\mathcal{O}(kd)$ ，一共做了 n 次，故复杂度为 $\mathcal{O}(nkd)$
- 为了保证第二个维度在第二个维度都相同，故需要 d 个卷积核，所以卷积操作总的时间复杂度为 $\mathcal{O}(nkd^2)$