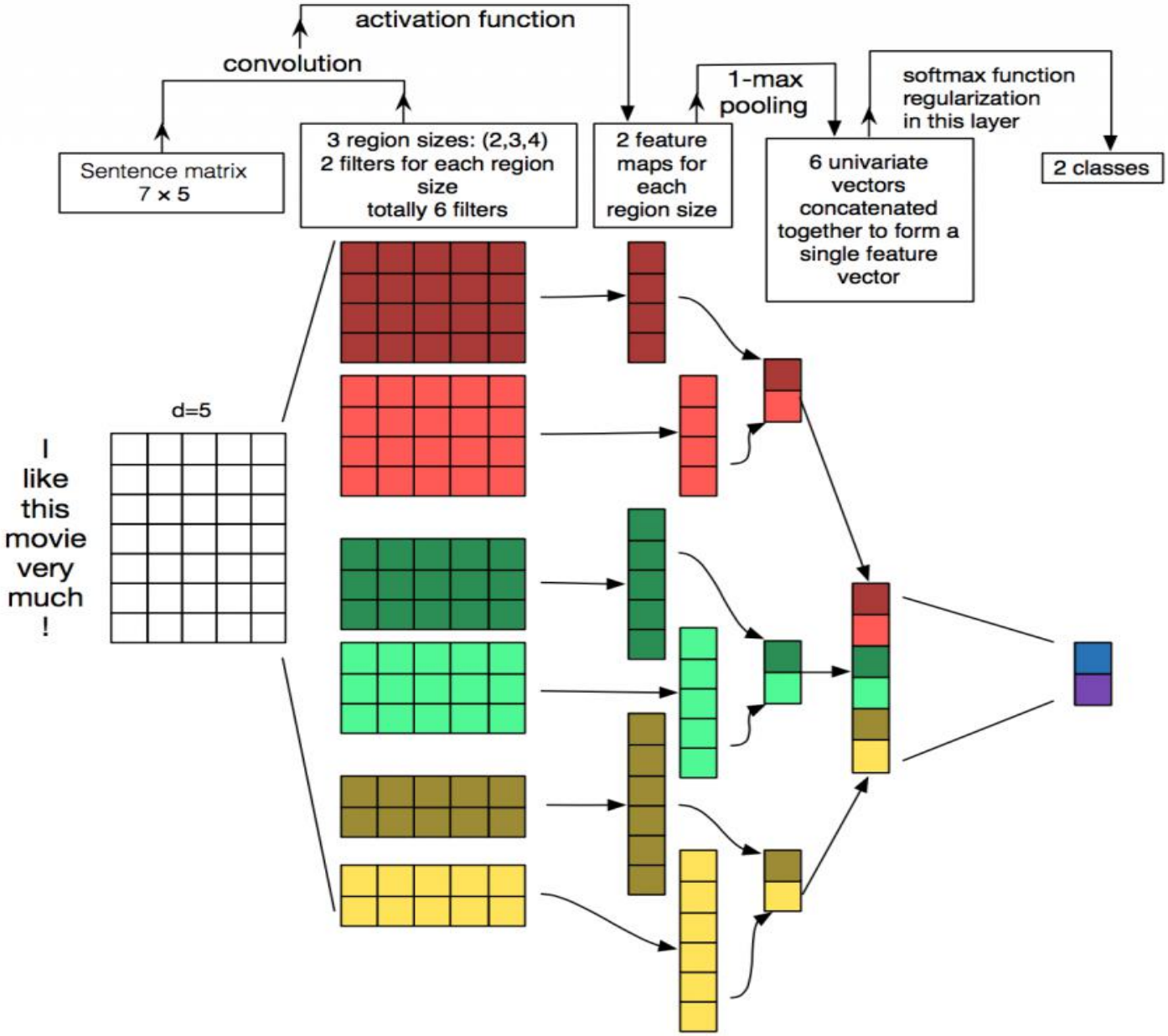
# Text-cnn

Pangkanggao

2020.07.31

- text-cnn原理
- 把词向量（word2vec）按照词顺序拼接起来，然后经过卷积层convolution1D（不同的卷积核2,3,4,5,可以视作考虑2（3,4,5）个词的关系)，然后经过池化，flatten拼接，再经过全连接层Dense，dropout（有多少个神经元不工作），之类，最后输出，输出的标签可转为one-hot格式。


activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I
like
this
movie
very
much
!

# Textcnn-embding的工作原理

以NLP词嵌入举例，Embedding层就是为了训练一个词嵌入矩阵出来，然后可以获得任意的一个词的词向量。

设：

有一个输入句子样本是‘I very happy’,词典是[0:pad_word, 1: I, 2:very, 3:happy, 4:so, 5:sad]

$$\text{词嵌入矩阵W}=\begin{bmatrix}0.12 & 0.2\\ 0.30 & 0.15\\ 1.0 & 0.69\\ 20.1 & 1.45\\ 1.29 & 2.01\\ 3.45 & 3.45\end{bmatrix}=\begin{bmatrix}\text{pad\_}word\\ I\\ \text{very}\\ happy\\ so\\ sad\end{bmatrix}$$

则：

input X=[1, 2, 3]，然后对X进行one-hot

$$\text{X\_one\_hot}=\begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}\text{,那么对输入X的词向量表示应该是X\_one\_hot}\cdot\text{ W}$$

$$\text{X的词向量表示}=\begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}*\begin{bmatrix}0.12 & 0.2\\ 0.30 & 0.15\\ 1.0 & 0.69\\ 20.1 & 1.45\\ 1.29 & 2.01\\ 3.45 & 3.45\end{bmatrix}=\begin{bmatrix}0.30 & 0.15\\ 1.0 & 0.69\\ 20.1 & 1.45\end{bmatrix}=\begin{bmatrix}I\\ very\\ happy\end{bmatrix}$$

# Textcnn-embding的工作原理

- 实现方法之keras：
- Keras.layers. Embedding(input_dim, output_dim, embeddings_initializer='uniform', embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint=None, mask_zero=False, input_length=None)
- 也就是说对于像一个句子样本X=[1,2,3] (1,2,3表示单词在词典中的索引)这样的输入可以先对它one-hot然后乘上词嵌入矩阵就可得到这个句子的词嵌入向量表示。要想得到好的词向量，我们需要训练的就是这个矩阵W(shape=(input_dim,output_dim))。Embedding层的作用就是训练这个矩阵W并进行词的嵌入为每一个词分配与它对应的词向量。这个词嵌入矩阵W可以先随机初始化，然后根据下游任务训练获得，也可以使用预训练的词嵌入矩阵来初始化它(keras中用weights来为layer初始化任意权重)，然后再训练，也可以直接用预训练的词嵌入矩阵来初始化它并冻结它，不让它变化，不让它可训练。(keras中用trainable=False)

# Textcnn-embding的工作原理

**Arguments**

- **input_dim**: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
- **output_dim**: int >= 0. Dimension of the dense embedding.
- **embeddings_initializer**: Initializer for the `embeddings` matrix (see initializers).
- **embeddings_regularizer**: Regularizer function applied to the `embeddings` matrix (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **embeddings_constraint**: Constraint function applied to the `embeddings` matrix (see constraints).
- **mask_zero**: Whether or not the input value 0 is a special "padding" value that should be masked out. This is useful when using recurrent layers which may take variable length input. If this is `True` then all subsequent layers in the model need to support masking or an exception will be raised. If mask_zero is set to True, as a consequence, index 0 cannot be used in the vocabulary (input_dim should equal size of vocabulary + 1).
- **input_length**: Length of input sequences, when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed).

# Textcnn-embding的工作原理

- 嵌入层被定义为网络的第一个隐藏层。它必须指定3个参数：
- input_dim：这是文本数据中词汇的取值可能数。例如，如果您的数据是整数编码为0-9之间的值，那么词汇的大小就是10个单词；
- output_dim：这是嵌入单词的向量空间的大小。它为每个单词定义了这个层的输出向量的大小。例如，它可能是32或100甚至更大，可以视为具体问题的超参数；
- input_length：这是输入序列的长度，就像您为Keras模型的任何输入层所定义的一样，也就是一次输入带有的词汇个数。例如，如果您的所有输入文档都由1000个字组成，那么input_length就是1000。

# Textcnn-embding的工作原理

**Example**

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Embedding(1000, 64, input_length=10))
>>> # The model will take as input an integer matrix of size (batch,
>>> # input_length), and the largest integer (i.e. word index) in the input
>>> # should be no larger than 999 (vocabulary size).
>>> # Now model.output_shape is (None, 10, 64), where `None` is the batch
>>> # dimension.
>>> input_array = np.random.randint(1000, size=(32, 10))
>>> model.compile('rmsprop', 'mse')
>>> output_array = model.predict(input_array)
>>> print(output_array.shape)
(32, 10, 64)
```

# Textcnn-卷积层的工作原理

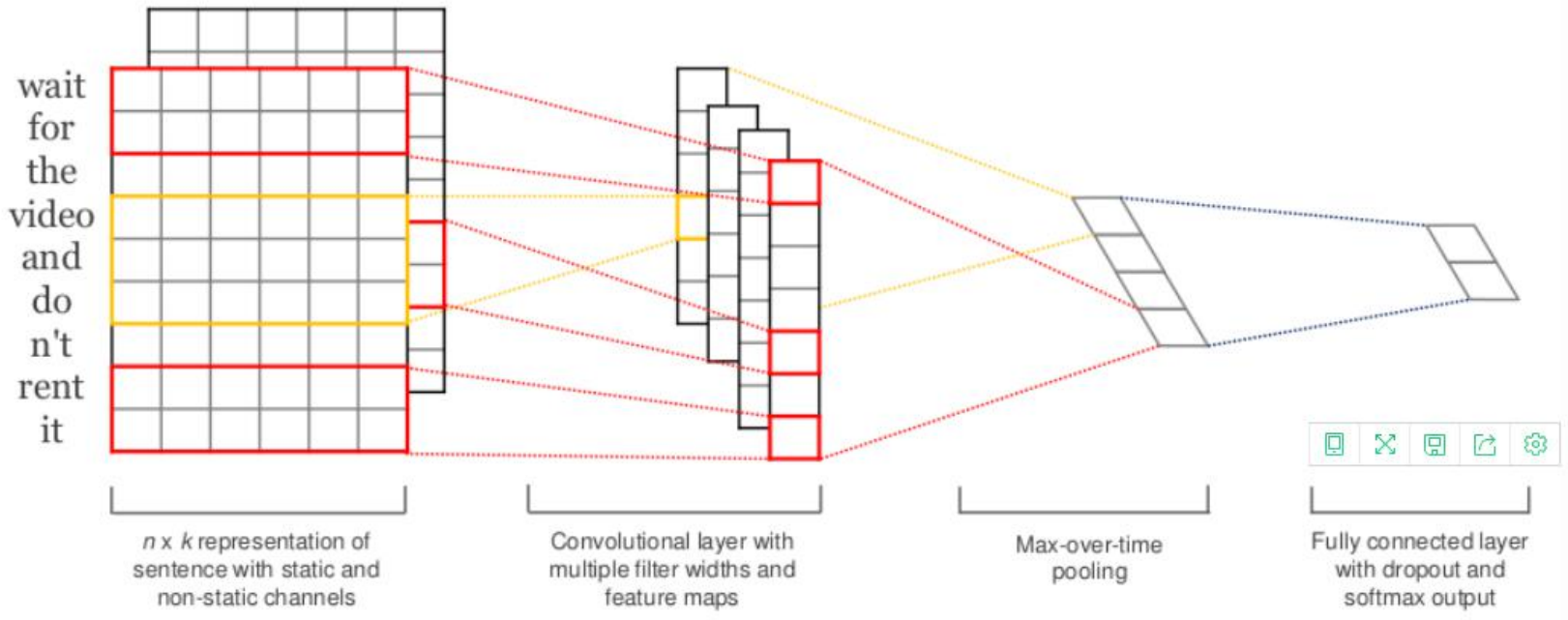
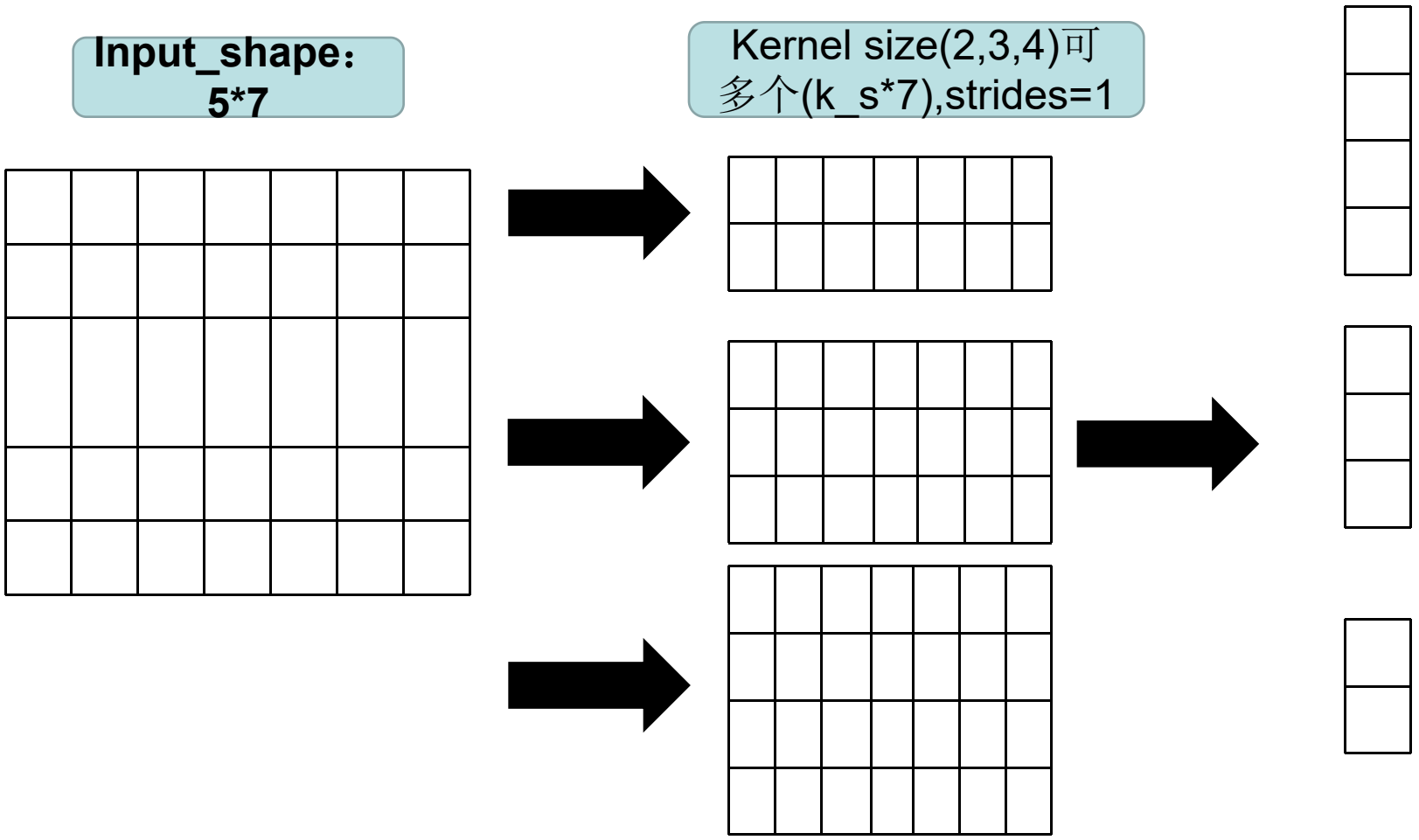



Figure 1: Model architecture with two channels for an example sentence.

# Textcnn-卷积层的工作原理



- 实现方法之keras：
- Keras.layers.conv1D()
- 参数

- filters: 整数，输出空间的维度 （即卷积中滤波器的输出数量）。
- kernel_size: 一个整数，或者单个整数表示的元组或列表， 指明 1D 卷积窗口的长度。
- strides: 一个整数，或者单个整数表示的元组或列表， 指明卷积的步长。 指定任何 stride 值 != 1 与指定 dilation_rate 值 != 1 两者不兼容。
- padding: "valid", "causal" 或 "same" 之一 (大小写敏感) "valid" 表示「不填充」。 "same" 表示填充输入以使输出具有与原始输入相同的长度。 "causal" 表示因果（膨胀）卷积， 例如，output[t] 不依赖于 input[t+1:]， 在模型不应违反时间顺序的时间数据建模时非常有用。 详见 WaveNet: A Generative Model for Raw Audio, section 2.1。
- data_format: 字符串, "channels_last" (默认) 或 "channels_first" 之一。输入的各个维度顺序。 "channels_last" 对应输入尺寸为 (batch, steps, channels) (Keras 中时序数据的默认格式) 而 "channels_first" 对应输入尺寸为 (batch, channels, steps)。
- dilation_rate: 一个整数，或者单个整数表示的元组或列表，指定用于膨胀卷积的膨胀率。 当前，指定任何 dilation_rate 值 != 1 与指定 stride 值 != 1 两者不兼容。

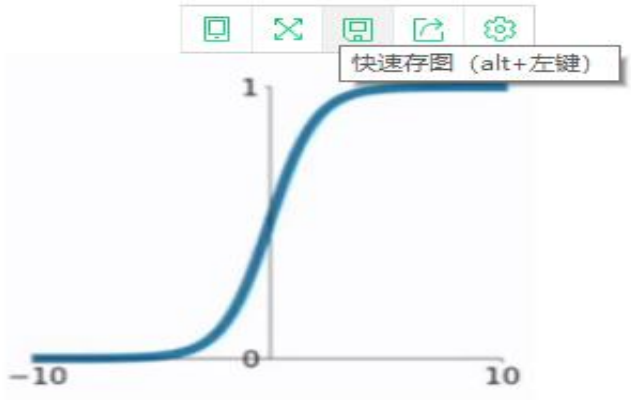
- activation: 要使用的激活函数 (详见 activations)。 如未指定，则不使用激活函数 (即线性激活： a(x) = x)。
- use_bias: 布尔值，该层是否使用偏置向量。
- kernel_initializer: kernel 权值矩阵的初始化器 (详见 initializers)。
- bias_initializer: 偏置向量的初始化器 (详见 initializers)。
- kernel_regularizer: 运用到 kernel 权值矩阵的正则化函数 (详见 regularizer)。
- bias_regularizer: 运用到偏置向量的正则化函数 (详见 regularizer)。
- activity_regularizer: 运用到层输出（它的激活值）的正则化函数 (详见 regularizer)。
- kernel_constraint: 运用到 kernel 权值矩阵的约束函数 (详见 constraints)。
- bias_constraint: 运用到偏置向量的约束函数 (详见 constraints)。

# 激活函数

**1. sigmod函数**

函数公式和图表如下图

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



# 激活函数

在sigmod函数中我们可以看到，其输出是在(0,1)这个开区间内，这点很有意思，可以联想到概率，但是严格意义上讲，不要当成概率。sigmod函数曾经是比较流行的，它可以想象成一个神经元的放电率，在中间斜率比较大的地方是神经元的敏感区，在两边斜率很平缓的地方是神经元的抑制区。

当然，流行也是曾经流行，这说明函数本身是有一定的缺陷的。

1) 当输入稍微远离了坐标原点，函数的梯度就变得很小了，几乎为零。在神经网络反向传播的过程中，我们都是通过微分的链式法则来计算各个权重w的微分的。当反向传播经过了sigmod函数，这个链条上的微分就很小很小了，况且还可能经过很多个sigmod函数，最后会导致权重w对损失函数几乎没影响，这样不利于权重的优化，这个问题叫做梯度饱和，也可以叫梯度弥散。

2) 函数输出不是以0为中心的，这样会使权重更新效率降低。对于这个缺陷，在斯坦福的课程里面有详细的解释。

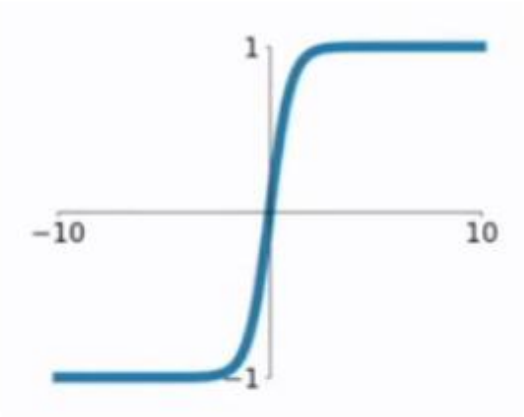
3) sigmod函数要进行指数运算，这个对于计算机来说是比较慢的。

# 激活函数

**2.tanh函数**

tanh函数公式和曲线如下

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



# 激活函数

tanh是双曲正切函数，tanh函数和sigmod函数的曲线是比较相近的，咱们来比较一下看看。首先相同的是，这两个函数在输入很大或是很小的时候，输出都几乎平滑，梯度很小，不利于权重更新；不同的是输出区间，tanh的输出区间是在(-1,1)之间，而且整个函数是以0为中心的，这个特点比sigmod的好。

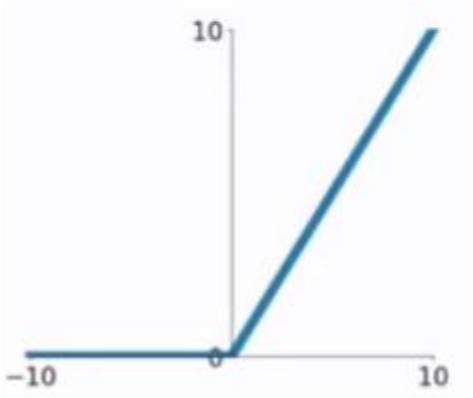
一般二分类问题中，隐藏层用tanh函数，输出层用sigmod函数。不过这些也都不是一成不变的，具体使用什么激活函数，还是要根据具体的问题来具体分析，还是要靠调试的。

# 激活函数

3.ReLU函数

ReLU函数公式和曲线如下

$$f(x) = max(0, x)$$



# 激活函数

ReLU(Rectified Linear Unit)函数是目前比较火的一个激活函数，相比于sigmod函数和tanh函数，它有以下几个优点:

1) 在输入为正数的时候，不存在梯度饱和问题。

2) 计算速度要快很多。ReLU函数只有线性关系，不管是前向传播还是反向传播，都比sigmod和tanh要快很多。（sigmod和tanh要计算指数，计算速度会比较慢)

当然，缺点也是有的:

1) 当输入是负数的时候，ReLU是完全不被激活的，这就表明一旦输入到了负数，ReLU就会死掉。这样在前向传播过程中，还不算什么问题，有的区域是敏感的，有的是不敏感的。但是到了反向传播过程中，输入负数，梯度就会完全到0，这个和sigmod函数、tanh函数有一样的问题。

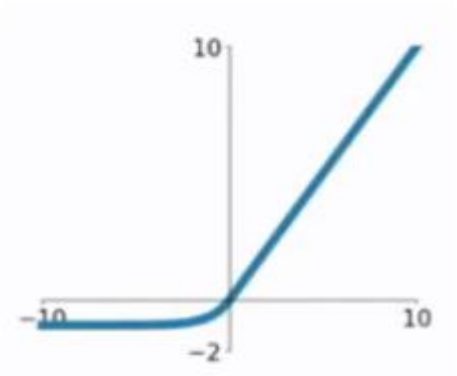
2) 我们发现ReLU函数的输出要么是0，要么是正数，这也就是说，ReLU函数也不是以0为中心的函数。

# 激活函数

**4.ELU函数**

ELU函数公式和曲线如下图

$$f(x) = \begin{cases} x & , x > 0 \\ \alpha(e^x - 1) & , x \leq 0 \end{cases}$$



# 激活函数

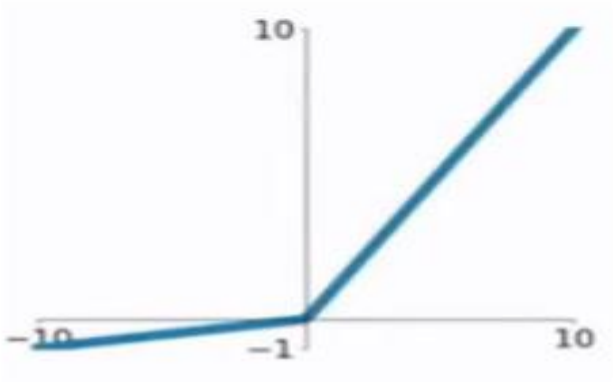
ELU函数是针对ReLU函数的一个改进型，相比于ReLU函数，在输入为负数的情况下，是有一定的输出的，而且这部分输出还具有一定的抗干扰能力。这样可以消除ReLU死掉的问题，不过还是有梯度饱和和指数运算的问题。

# 激活函数

**5.PReLU函数**

PReLU函数公式和曲线如下图

$$f(x) = max(ax, x)$$



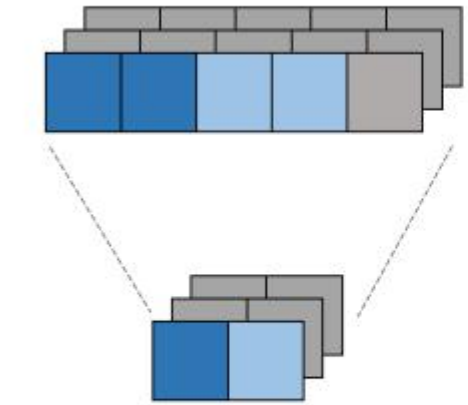
# 激活函数

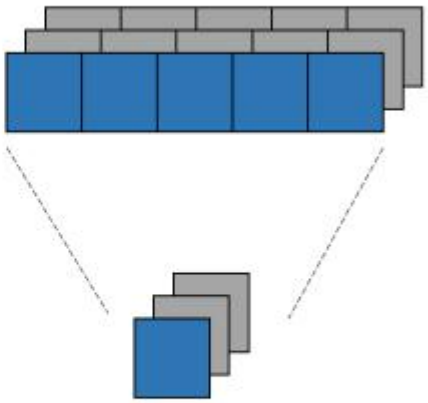
PReLU也是针对ReLU的一个改进型，在负数区域内，PReLU有一个很小的斜率，这样也可以避免ReLU死掉的问题。相比于ELU，PReLU在负数区域内是线性运算，斜率虽然小，但是不会趋于0，这算是一定的优势吧。

我们看PReLU的公式，里面的参数α一般是取0~1之间的数，而且一般还是比较小的，如零点零几。当α=0.01时，我们叫PReLU为Leaky ReLU，算是PReLU的一种特殊情况吧。

# Keras: GlobalMaxPooling vs MaxPooling

- 可以将把下面的每一条看做经过卷积层得到的



如上图所示，在`1x1x5`的网络上进行`MaxPooling1D`时，如果选择窗口大小为`2`，则面临两个问题：

1. 滑动窗口的步长选多少？（一般默认为`pool_size`）
2. 如果在上图中步长选`2`，最后一行还要不要了？（图中选择`valid`，即不做padding，放弃最后一行）

而对于`GlobalMaxPooling1D`来说，`pool_size`是固定的——就是最长那个维度（“全局”最大池化）。所以，步长不需要考虑——因为没有滑动空间了；因此，padding的问题也不存在了。

与此相应的，两种池化后结果的维度也不同：

# 参考资料

- conv1D：https://blog.csdn.net/VeritasCN/article/details/90050584
- 激活函数：https://blog.csdn.net/kangyi411/article/details/78969642
- Textcnn: https://www.cnblogs.com/bymo/p/9675654.html(里面含代码)
- maxpooling1D:
- https://blog.csdn.net/Tardigrade_/article/details/92799883?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.channel_param(不是很清晰)
- Embeding官方文档：
- https://keras.io/api/layers/core_layers/embedding/#embedding