总结

到了现在，大家通过简单的例子以及穿插的代码对神经网络的搭建和训练有了最初步的认识。但是之前的内容仅仅是为了让大家略知一二。当大家对整体过程有了一定认识之后，我们再对每一个地方进行细节的讲解。否则，如果像传统的书籍一样，按顺序对每一个点进行深入而且严谨的讲解，会让读者不清楚这样的目的是什么。

扩展

现在，我们基于之前学习的关于机器学习的基本知识和搭建的初步神经网络对内容进行扩展。让算法和代码的每一部分都变得越来越严谨。要注意的是，这里的内容仍然只是整个机器学习的冰山一角。

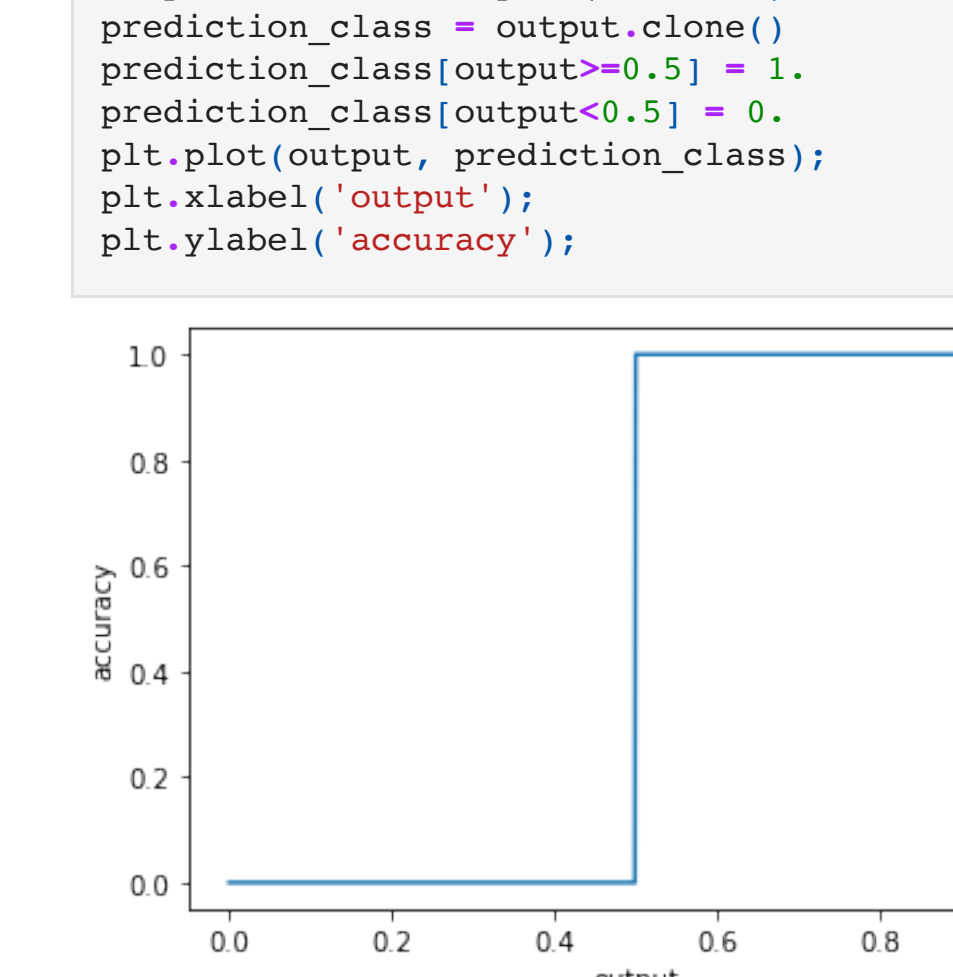
损失函数

损失函数又叫目标函数。所以显而易见，损失函数应当直接表现任务的目标。下面我们来看一下分类和回归任务中的损失函数：

分类

在一般的分类任务中，我们的最直观目标就是增加分类的准确率。但是很遗憾，准确率是一个不可导的标准。让我们以二分类问题为例子，我们把归一化之后的输出小于0.5的情况分类为0，大于0.5的情况分类为1，那么对于每一个数据，准确率和输出之间的关系就是：

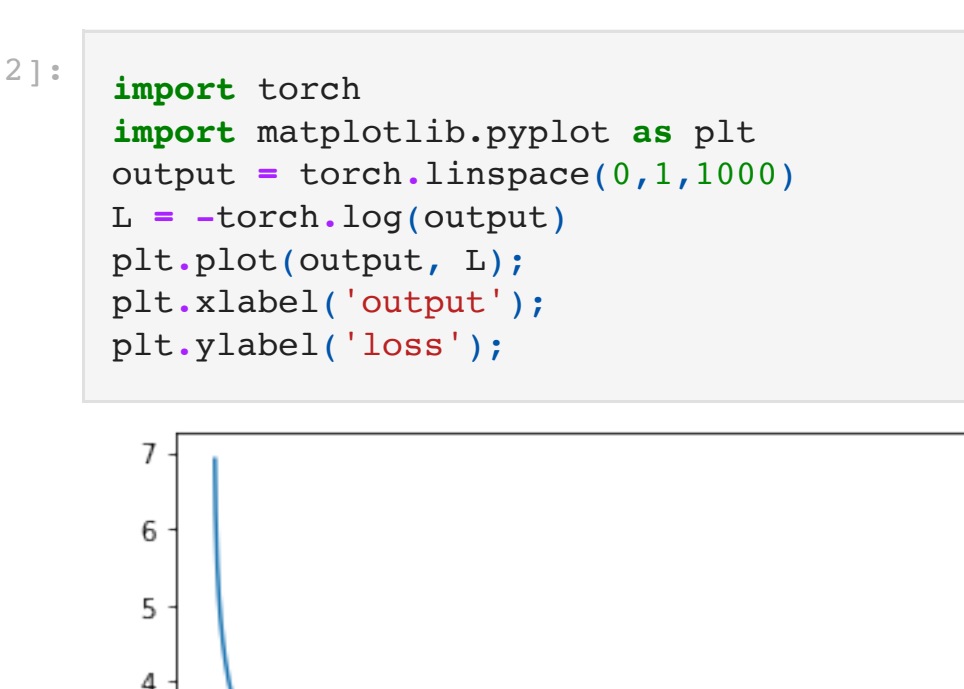
In [1]:
```
import torch
import matplotlib.pyplot as plt
output = torch.linspace(0,1,1000)
prediction_class = output.clone()
prediction_class[output>=0.5] = 1.
prediction_class[output<0.5] = 0.
plt.plot(output, prediction_class);
plt.xlabel('output');
plt.ylabel('accuracy');
```



可见accuracy对于输出output的梯度处处为0，那么梯度无法反向传播。因此我们需要一个连续的函数用于代替准确率来作为损失函数。这时候，交叉熵被摆了出来。我们先来看一个简单的函数：

$$\mathcal{L} = -\log(\hat{y})$$

In [2]:
```
import torch
import matplotlib.pyplot as plt
output = torch.linspace(0,1,1000)
L = -torch.log(output)
plt.plot(output, L);
plt.xlabel('output');
plt.ylabel('loss');
```



这样的话，当我们最小化损失函数时，就会把输出的值增加，这应该对应于分类为1的情况。对于分类应该为0的情况，我们其实应该减少输出值，这时候，我们让函数为

$$\mathcal{L} = -\log(1-\hat{y})$$

这样，损失函数就是

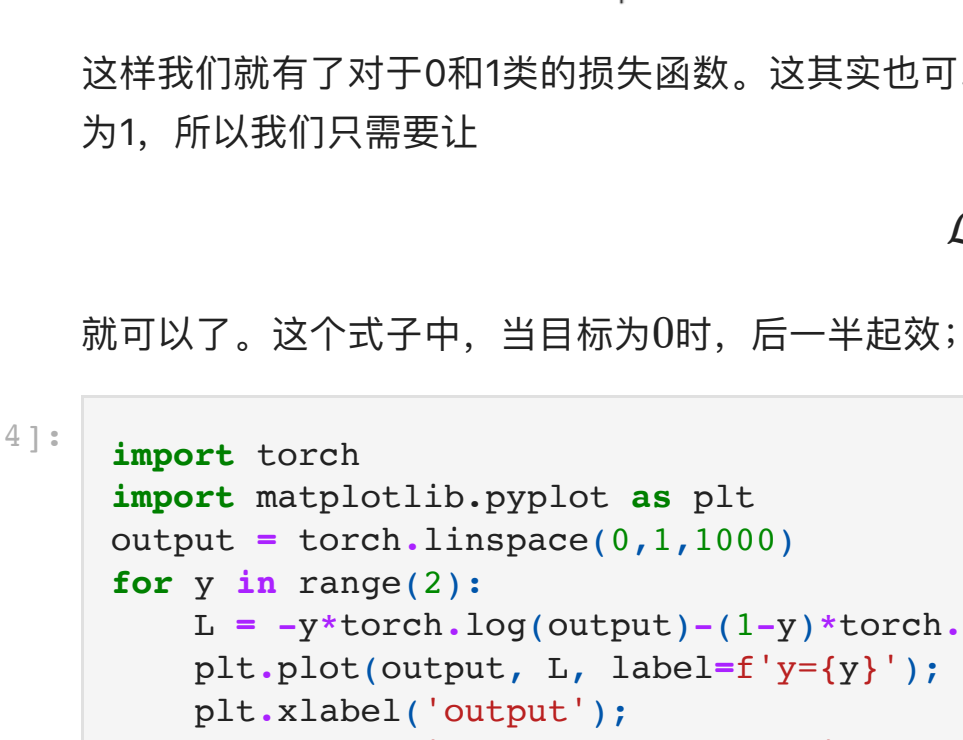
In [3]:
```
import torch
import matplotlib.pyplot as plt
output = torch.linspace(0,1,1000)
L = -torch.log(1-output)
plt.plot(output, L);
plt.xlabel('output');
plt.ylabel('loss');
```

这样我们就有了对于0和1类的损失函数。这其实也可以用于多类的分类问题：经过one-hot编码之后，错误的分类的目标值为0，正确分类的目标值为1，所以我们只需要让

$$\mathcal{L} = \sum -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

就可以了。这个式子中，当目标为0时，后一半起效；当目标为1时，前一半起效。图像如下：

In [4]:
```
import torch
import matplotlib.pyplot as plt
output = torch.linspace(0,1,1000)
for y in range(2):
    L = -y*torch.log(output)-(1-y)*torch.log(1-output)
    plt.plot(output, L, label=f'y={y}');
    plt.xlabel('output');
    plt.ylabel('cross entropy loss');
plt.legend();
```



这就是交叉熵函数(cross entropy loss)。

回归

回归（regression）问题又叫拟合（aproximation）问题，指的是根据给定的输入，让机器学习模型给出期望的函数值。对于这种问题，我们往往用输出值和目标值的差作为损失函数。例如

$$\mathcal{L} = (\hat{y}-y)^2 = \Delta^2$$

或

$$\mathcal{L} = |\hat{y}-y| = |\Delta|$$

我们可以可视化一下这两个函数

In [5]:
```
import torch
import matplotlib.pyplot as plt
D = torch.linspace(-2,2,1000)

L1 = D**2
L2 = D.abs()
plt.plot(D, L1, label=f'squared error');
plt.plot(D, L2, label=f'absolute error');
plt.xlabel('$\Delta$');
plt.ylabel('loss');
plt.legend();
```

可以看出，当我们把loss降低时，两个函数都会把Δ推向0。区别是：用平方作为损失函数时，当误差较小时，梯度较小。所以在学习过程中，当参数需要在多个训练数据间进行权衡的时候，误差大的会被着重考虑，也就是说，也就是大误差的数据会对梯度有更大的影响。而绝对值作为损失函数时，所有的误差，无论本身的大小，对于梯度的影响都一样，也就是说，他们对于学习过程的影响是一样的。所以，相比之下，前者的训练结果更倾向于消除大的误差，而后者相**比于前者**，更倾向于把小的误差消除到0。

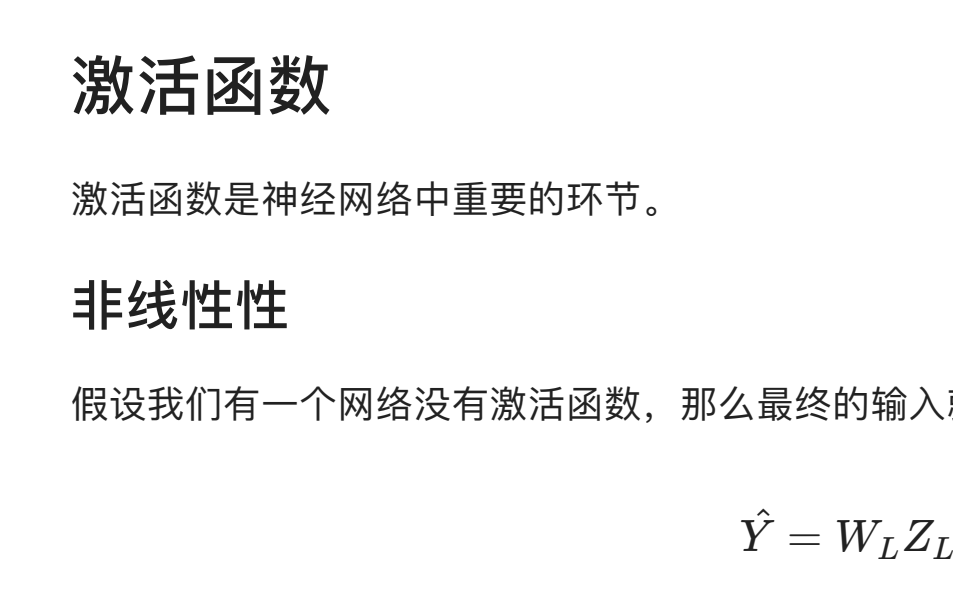
但是，我们在采集数据时，往往会出现特殊的情况，例如当我们采集身高的时候，出现了一个身高为58米的人，显然这是一个错误的数据，如果我们强行把这个数据考虑到数据集中，是愚蠢的。这种数据我们往往叫他们outlier。有的时候，由于问题的复杂，我们很难直观挑出他们，因此我们会想办法在训练过程中忽略那些误差特别离谱的数据点。这时候我们就要用 tukey loss

$$l(\Delta) = \begin{cases} K\left(1-\left(1-\left(\frac{\Delta}{\sqrt{6K}}\right)^2\right)^3\right), & |\Delta| \le \sqrt{6K} \\ K \end{cases}$$

In [6]:
```
import torch
import matplotlib.pyplot as plt
D = torch.linspace(-5,5,1000)

def tukey(d, k):
    if d.abs()>(6*k)**0.5:
        return k
    else:
        return k*(1-(1-(d/(6*k)**0.5)**2)**3)

L1 = D**2
L2 = D.abs()
plt.plot(D, L1, label=f'squared error');
plt.plot(D, L2, label=f'absolute error');
for k in range(1,4):
    L3 = [tukey(d,k) for d in D]
    plt.plot(D, L3, label=f'tukey loss k={k}');
plt.xlabel('$\Delta$');
plt.ylabel('loss');
plt.ylim([0,4])
plt.legend();
```



激活函数

激活函数是神经网络中重要的环节。

非线性性

假设我们有一个网络没有激活函数，那么最终的输入就可以表示为：

$$\hat{Y} = W_L Z_{L-1} = W_L(W_{L-1}Z_{L-2}) = \cdots = \underbrace{\prod_{l=1}^{L} W_l}_{=:W} Z_0 =: WZ_0$$

这无异于一层的神经网络。因此，（非线性的）激活函数往往是必须的。要注意的是，我们需要始终记住，加权求和+激活函数只是一种计算方式，实际中并没有对这种形式有严格的限定，例如卷积神经网络CNN中的max pooling，其实可以认为也是一种类似ReLU的变化，只不过ReLU是把小于0的变成0，大于0的不变；而max pooling是把卷积核对应区域对非最大值变成0，最大值不变，这其实也对网络引入了非线性性质。因此，要保证网络对非线性问题的解决能力，非线性的环节（往往是激活函数，但不必要）是必须的。

需要注意的是以下几点：

1. 我们对激活函数的最低要求就是非线性性，也就是$f(kx) \neq kf(x), \forall x$。任意的非线性连续函数都可以当作激活函数，甚至不需要单调性或者过零点。[1]中的可学习激活函数证明了，某些最优的激活函数不是单调的。在实际中，甚至负的激活函数都没关系，例如$-\tanh(\cdot)$。
2. 深层的线性网络的确在数学上等价于1层网络。但并不等于1层网络。很多关于神经网络的研究都可以以深度线性网络为研究对象[2]。

[1] M. Tavakoli, et al., "SPLASH: Learnable Activation Functions for Improving Accuracy and Adversarial Robustness," Neural Networks, vol. 140, pp. 1–12, 2021.

[2] MA. Saxe, et al., "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." arXiv preprint arXiv:1312.6120 (2013).

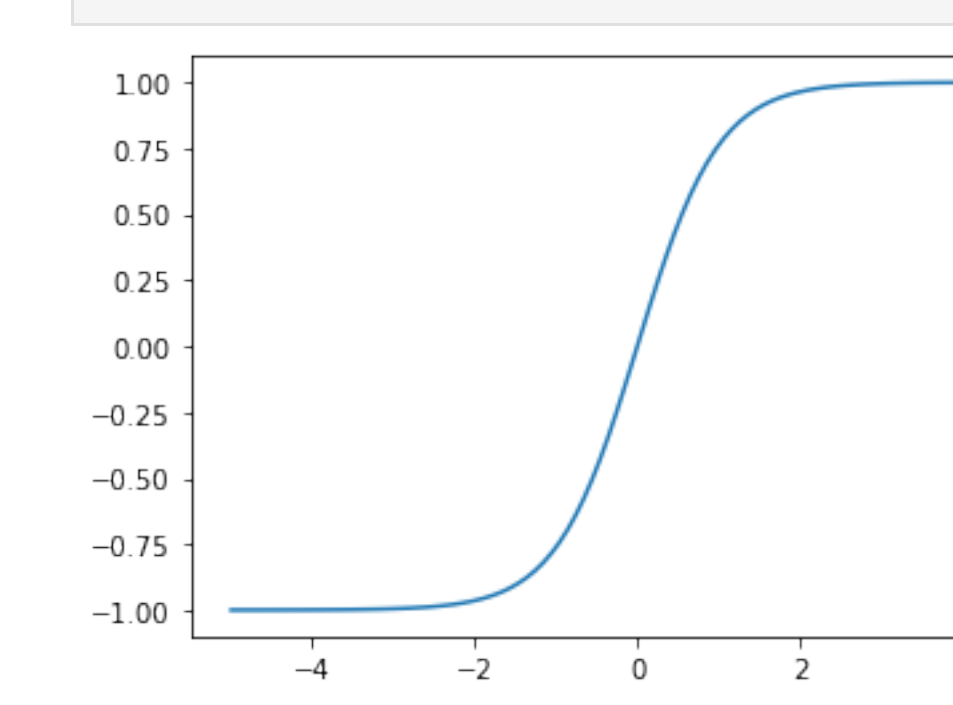
常见激活函数

正如上节所说，任意的非线性（连续）函数都可以当作激活函数，但是在实际中我们往往有常用的激活函数

1. tanh激活函数

$$A = \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}}$$

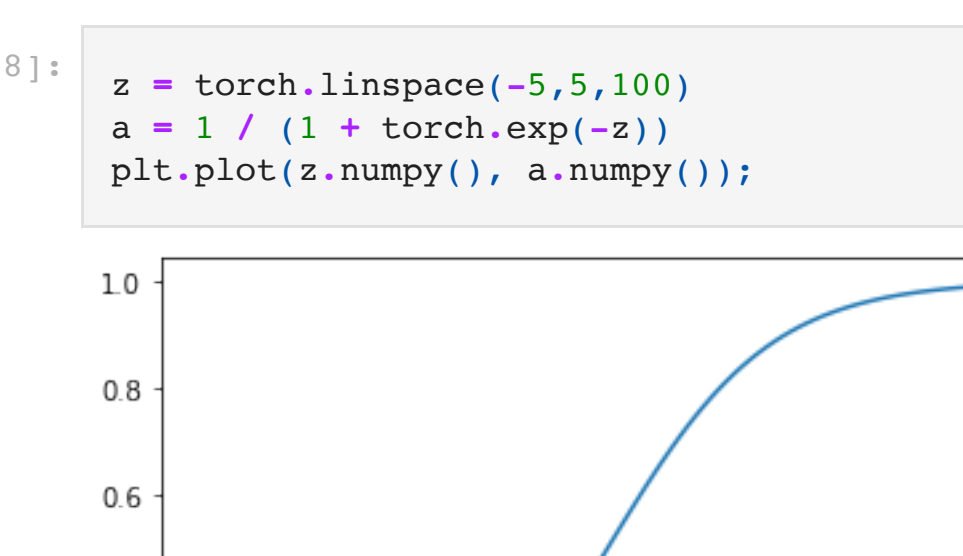
In [7]:
```
z = torch.linspace(-5,5,100)
a = (torch.exp(z) - torch.exp(-z)) / (torch.exp(z) + torch.exp(-z))
plt.plot(z.numpy(), a.numpy());
```



1. sigmoid激活函数

$$A = \frac{1}{1+e^{-z}}$$

In [8]:
```
z = torch.linspace(-5,5,100)
a = 1 / (1 + torch.exp(-z))
plt.plot(z.numpy(), a.numpy());
```



注意到上面2个函数都有值域范围，因此，如果用于回归任务，应当把输入和输出都进行正则化，否则的话，对于他们值域以外的部分，效果往往很差。

1. ReLU函数：要注意ReLU也是非线性函数，尽管它的每一段都是线性的

$$A = \begin{cases} Z, Z > 0 \\ 0, \text{otherwise.} \end{cases}$$

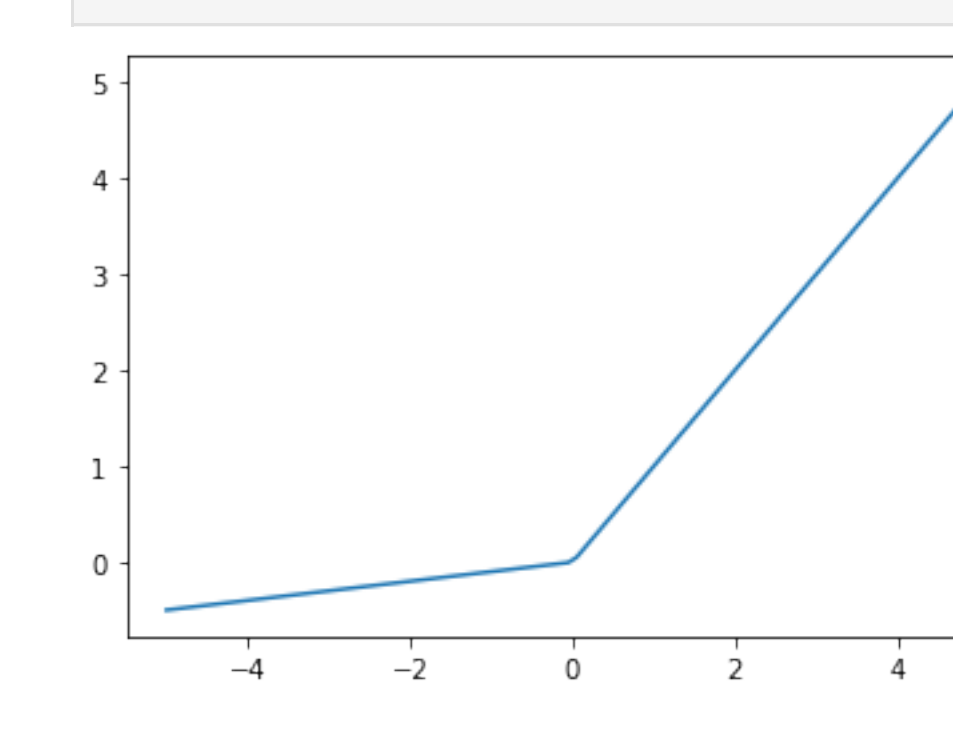
In [9]:
```
z = torch.linspace(-5,5,100)
a = z.clone()
a[a<0] = 0
plt.plot(z.numpy(), a.numpy());
```

可以看出，如果是ReLU函数的话，它的激活值是0到无穷大，而很多实际的物理值也都是正的。因此如果用ReLU作为激活函数，在合适的情况下不需要对输出正则化，也可以有较好的表现。实际上，在很多情况下，相比于tanh和sigmoid，即便ReLU是片段线性的函数，它仍然对非线性的拟合任务有更好的表现。一个很有可能的原因是ReLU是一个凸函数，而凸的问题更不容易被困在局部最优解。目前还有许多对于ReLU的改进，它们往往比ReLU更好。比如

SoftPlus函数：一个光滑的ReLU,避免了z在负半轴会被"困住"（梯度永远为0）的问题

$$A = \log(1+e^Z)$$

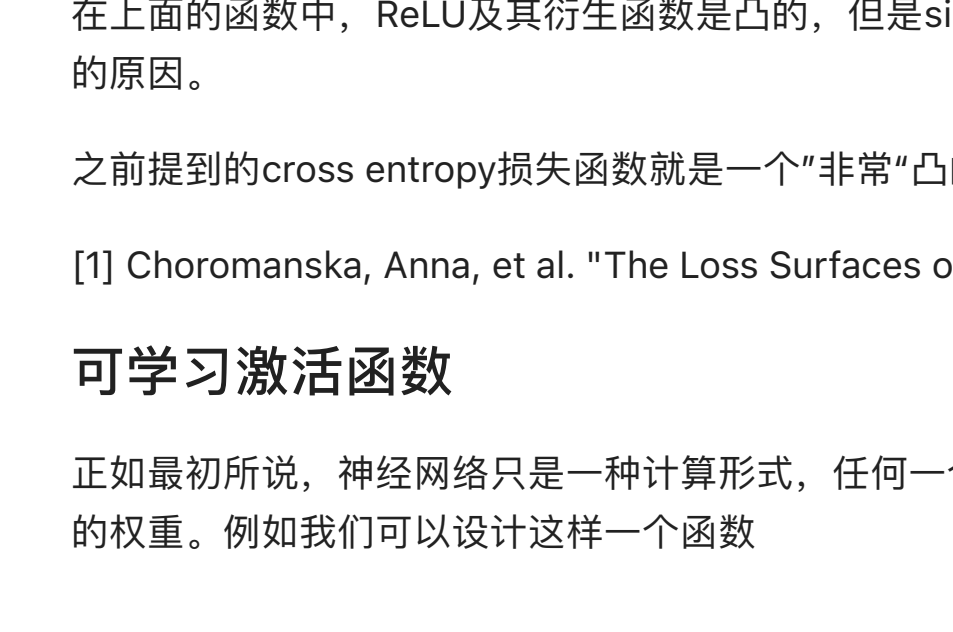
In [10]:
```
z = torch.linspace(-5,5,100)
a = torch.log(1+torch.exp(z))
plt.plot(z.numpy(), a.numpy());
```



LeakyReLU：在负半轴有一个固定斜率，避免负半轴不能学习的问题，同时还避免了梯度消失的问题（看下文）

$$A = \begin{cases} Z, & Z > 0 \\ 0.1Z, & \text{otherwise.} \end{cases}$$

In [11]:
```
z = torch.linspace(-5,5,100)
a = z.clone()
a[a<0] = a[a<0] * 0.1
plt.plot(z.numpy(), a.numpy());
```



PReLU: 斜率可以学习的激活函数（看下文）

凹凸性

一个容易被忽视的问题就是函数的凹凸性。凸函数的定义比较麻烦，但是可以简略的理解为只有1个最优解，没有局部最优解的问题。因此，如果一个问题是凸的，用梯度下降法就一定能够找到最优解。尽管神经网络的问题不是凸的[1]，但是如果神经网络里的每一个环节都尽可能是凸的，最终的目标函数的凸性就会增加，这会有助于找到问题的最优解，网络也会有更好的表现。

在上面的函数中，ReLU及其衍生函数是凸的，但是sigmoid和tanh不是。这**有可能**是ReLU系列（分段线性函数）在拟合非线性函数方面有更好表现的原因。

之前提到的cross entropy损失函数就是一个"非常"凸的函数。

[1] Choromanska, Anna, et al. "The Loss Surfaces of Multilayer Networks." Artificial intelligence and statistics. PMLR, 2015.

可学习激活函数

正如最初所说，神经网络只是一种计算形式，任何一个参数，只要能求出loss对它的梯度，它就可以被优化。因此可以被优化的参数不局限于网络中的权重。例如我们可以设计这样一个函数

$$A = \begin{cases} k_1 Z, & Z > 0 \\ k_2 Z, & \text{otherwise.} \end{cases}$$

其中k_1和k_2是可以学习的参数，他们会在训练过程中计算$\frac{\partial \mathcal{L}}{\partial k_1}$和$\frac{\partial \mathcal{L}}{\partial k_2}$并且被更新。

其实类似的函数可以任意设计。比如在[1]中，他们用了多个分段线性连续的函数，并且每一段的斜率都可以学习。一个有趣的现象（原文中没有提出）是：学习出来的函数往往都是凸的，这也从侧面印证了上一节对于凹凸性的描述。

PyTorch自带了PReLU激活函数，它具有1个可学习参数，就是在负半轴的斜率。因此在使用ReLU时，尽量使用SoftPlus或PReLU。

[1] M. Tavakoli, et al., "SPLASH: Learnable Activation Functions for Improving Accuracy and Adversarial Robustness," Neural Networks, vol. 140, pp. 1–12, 2021.

梯度消失/爆炸

ReLU最大的问题在于负半轴不可以学习，因为那里的梯度为0。同样的，如果在sigmoid或者tanh里，如果z的值过大，那么梯度也会很小。梯度就消失了。

梯度爆炸跟激活函数的关系不大，更多的是与权重相关。回忆之前的反向传播，我们发现：

$$\frac{\partial z_{l+1}}{\partial z_l} = \frac{\partial z_{l+1}}{\partial a_l}\frac{\partial a_l}{\partial z_l} = w_{l+1} \cdot a_l \cdot (1-a_l)$$

也就是说，每一层传播的时候都会乘以当前层的权重w_l，如果w过大，梯度经过多层传播就会呈指数型增长。

为了避免梯度消失/爆炸，我们不应该让权重过大或者过小。因此在就需要特别关注权重的初始化。看下一章：

参数初始化

数值优化的初始值对数值优化的结果有巨大影响。而这个问题，我们也可以从不同的角度来看待。

初始化原则

正如之前所说，初始化的目的是为了避免某些传播时产生的问题。

Xavier

对于深度神经网络，会存在这样一个问题，就是当层数过多时，输出的值以及分布会越来越不正常。例如在加权求和中$Z = XW$，如果W的数学期望大于1，那么Z就会比X大一点，那么每一层的输出值都会增加一些，而且值的分布方差也会增加一些，这样经过多层之后会导致许多问题，例如梯度爆炸或者梯度消失。Xavier初始化的目标就是通过选择合适的权重W，使得每一层输出和输入的分布一致，这样的话无论传递多少层，都不会出现极端的情况。

要注意的是，Xavier初始化假设的是激活函数经过零点，且在零点处斜率为1。同时，假设输入X和权重W都是期望为0，方差很小的变量，因此Z也是期望为0，方差很小的变量。这样的话，一个期望值为0，方差较小的随机变量经过激活函数后，分布不变。

我们把输出Z表示为$\mathbb{E}\{Z\} + \delta_Z$，其中$\mathbb{E}\{Z\} = 0$，并且$\delta_Z$是一个期望为0方差为$\text{Var}(Z)$的随机变量。那么，激活函数的输出是：

$$\text{act}(\mathbb{E}\{Z\} + \delta_z) \approx \text{act}(\mathbb{E}\{Z\}) + \frac{\partial\, \text{act}(\mathbb{E}\{Z\})}{\partial Z}\delta_Z = \delta_Z$$

所以经过激活函数之后的变量的期望和方差是：

$$\text{Var}(\text{act}(\mathbb{E}\{Z\} + \delta_z)) = \text{Var}(\delta_z) = \text{Var}(Z)$$
$$\mathbb{E}\{\text{act}(\mathbb{E}\{Z\} \pm \delta_z)\} = \mathbb{E}\{\mathbb{E}\{Z\} + \delta_z\} = 0 = \mathbb{E}\{Z\}$$

在这种假设之下，我们就可以忽略激活函数对于分布的影响了。这种情况下，我们假设输入是

$$X \in \mathbb{R}^{M \times E}$$

权重矩阵是

$$W \in \mathbb{R}^{N \times M}$$

输出是

$$Z = WX \in \mathbb{R}^{N \times E}$$

它的方差就是

$$\text{Var}(Z) = \frac{1}{E}(WX - \bar{W}\bar{X})(WX - \bar{W}\bar{X})^\top$$

根据假设，W和X的数学期望是0，所以

$$\begin{aligned} \text{Var}(Z) &= \frac{1}{E}(WX)(WX)^\top \\ &= \frac{1}{E}WXX^\top W^\top \\ &= \frac{1}{E}W \cdot E \cdot \text{Var}(X) \cdot W^\top \end{aligned}$$

由于各个特征之间相互独立，并且已经假设每个特征的方差都是一样，记为σ^2，我们简化它为$\text{Var}(Z) = \sigma^2 I$和$\text{Var}(X) = \sigma^2 I$。其中$I$是单位矩阵，维度根据所乘对象而改变。所以：

$$\begin{aligned} \sigma^2 I &= W \cdot \sigma^2 I \cdot W^\top \\ I &= WW^\top \\ &= M \cdot \text{Var}(W) \\ \text{Var}(W) &= \frac{I}{M} \end{aligned}$$

其中M是输入特征个数。

在工程上也有用输入和输出的平均值，也就是$\text{Var}(W) = \frac{2I}{M+N}$作为权重初始方差的，这没有什么数学原理，只是工程经验。

He初始化

In [ ]:

In [ ]:

随机种子对初始化的影响

优化策略和优化器

分批训练

SGD

动量

自适应长度

学习率

固定学习率

可变学习率

数据集

预处理

数据集分割

Early stop

正则化

实验的可重复性

方便自己

方便他人

In [ ]: