第5章　神经网络的神经单元

目前基于BP反向传播算法训练而来的神经网络都会面临着一个重要的问题，那就是梯度消失，梯度消失会让神经网络无法得到充分的训练，在性能上无法达到期望。无法得到有效训练的神经网络就好像一个无法发挥潜力的天才，为此，我们有一系列的方案去解决深度网络的训练问题。

5.1　梯度消失和梯度爆炸

虽然我们已经在第2章中的误差反向传播算法的本质中提到了梯度消失，并将其原因归结为激活函数的导数太小，以至于误差在层与层的传递之间逐渐减小。但是，为了更直观地看到激活函数对于权重更新的影响，我们再次举一个简单例子。可以假设某个神经网络由 l 层构成，但每一层都只有一个神经元，不使用偏置，前向传播过程为：

$$O=\omega_1\cdots\omega_3\omega_2\omega_l X \tag{5.1}$$

在上式中加入激活函数sigmoid，其实就是将每一层嵌套了起来，表现为多层的复合函数：

$$O=\sigma_l(w_l,\sigma_{l-1}(\cdots\sigma_1(\omega_1,x))) \tag{5.2}$$

同样，为了简单起见，假设我们的损失函数为 L，学习率为1，使用梯度下降算法，第 l 层的权重系数变化为：

$$\Delta w_1=-\frac{\partial L}{\partial O}\frac{\partial O}{\partial \omega_1}=-\delta^O\sigma'_l\sigma_{l-1} \tag{5.3}$$

在此基础上，利用式(2.27)，我们所更新的第 $l-1$ 层权重系数变化为：

$$\Delta w_{l-1}=-\frac{\partial L}{\partial \sigma_{l-1}}\frac{\partial \sigma_{l-1}}{\partial \omega_{l-1}}=-\delta^O\omega_l\sigma'_{l-1}\sigma_{l-2} \tag{5.4}$$

在这里我们可以看到，层的参数更新的幅值不仅取决于激活函数的导数 ω'_{l-1}，也取决于 w_1 和 δ^O。当 ω'_{l-1} 趋于零时，$\Delta\omega_{l-1}$ 也趋于零，表示

这层的参数几乎不会更新，这就是梯度消失，而当 ω_l 变得非常大的时候，$\Delta\omega_{l-1}$ 就会变得非常大，这层的参数会发生巨变，这就是梯度爆炸。

所以梯度消失和梯度爆炸名字虽然非常对应，但是内在的原因并不一样。梯度消失往往是因为激活函数，而梯度爆炸往往是因为权重，尤其是在初始化的过程中，如果将权重设置得过大，那么就很可能发生梯度爆炸。

5.2 隐藏单元设计原则和 sigmoid 的非零中心

首先，我们有必要强调 sigmoid 隐藏单元更多意义在于其生物角度上，因为它将大范围的输入挤压到[0,1]，对应着生物学神经元抑制和激活两种状态，但却在神经网络的优化过程中非常容易出现迭代缓慢和梯度消失的问题。但根据目前的主流观点，激活函数最重要的作用并非为了与生物学上状态严格对应，而是作为一个非线性函数来满足万能近似定理，从而保证神经网络可以近似任意的函数。

我们可以很快地排除两个简单的隐藏单元，即阶跃函数和线性函数，如图 5.1，阶跃函数在变量两端的导数均为零，在反向传播时，梯度流将会恒为零，左导数和右导数均为无穷大，不可计算。而线性函数在优化上没有什么困难，而且梯度流在逆向传播时会很稳定，但不符合万能近似定理，无法近似非线性函数。

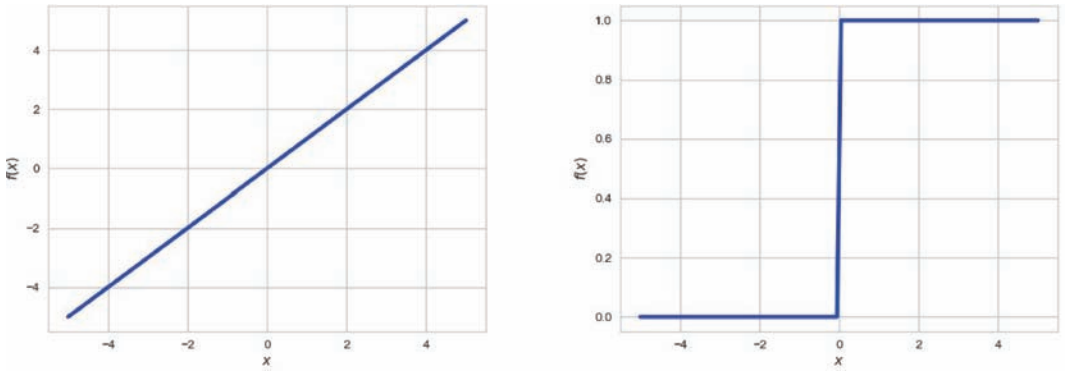


图 5.1 作图为线性函数，右图为阶跃函数

对隐藏单元的理解有两条思路，其一，根据一些基本原则去自行设计，其二，基于阶跃函数和线性函数分别作出改进。首先我们根据理论知识来对隐藏单元的基本原则作出总结，也就是隐藏单元必须或者最好能拥有的特性：

（1）近似线性。为了避免大的梯度和小的梯度在深层网络中传播时对优化的影响，若梯度为 1，则可以避免一些麻烦，但是，完全的线性又会违反万能近似定理，所以我们希望激活函数在某个区间内保持线性。

（2）连续性。我们希望对于每一个输入，都有输出。当某个可能的数值在激活函数上没有定义时，这个激活函数将没有输出，这是不被允许的。注意，输出为零与没有输出是两

种截然不同的情况。

（3）几乎可微。完全可微性经常被人误以为是激活函数的必要条件，因为总是要计算梯度。实际上，我们不需要保证对所有的点都存在导数，有限的几个点不存在导数仍然是可以接受的，我们会用左导数或者右导数来替代导数没有定义的点。

（4）在上述三条的基础上，我们还希望它可以满足单调性。这一点存在着一些争议，因为有些激活函数并不是单调性的，但效果却很好。一般认为，单调性的激活函数会带来更好的收敛效果。

（5）小的饱和区域，甚至不饱和。我们已经在上文中看到，大量的饱和区域会使得深层的网络参数很难得到大幅度的更新，我们希望激活函数的梯度在很小的区域内为零。除此之外，函数本身等于零的区域也要尽量少。

（6）更小的计算量。一方面，我们希望激活函数计算起来较为简单，另一方面，我们希望激活函数的参数尽量少，因为激活函数每增加一个参数，整个神经网络要相应增加隐层包含神经元数量之和的参数数量，这对于优化仍然是不利的。

虽然我们称其为隐藏单元的设计原则，但这些原则都不是本质意义上的，但可以说，根据目前我们对神经网络的了解，这些原则是适用的。接下来，我们遵循第二条思路，从逐步改进的思路去理解隐藏单元的设计。根据阶跃函数不可微的缺点，我们可以选取喜闻乐见的 sigmoid 函数，这一函数曾经被大量使用，是很多人入门深度学习见到的第一个函数，如图 5.2。

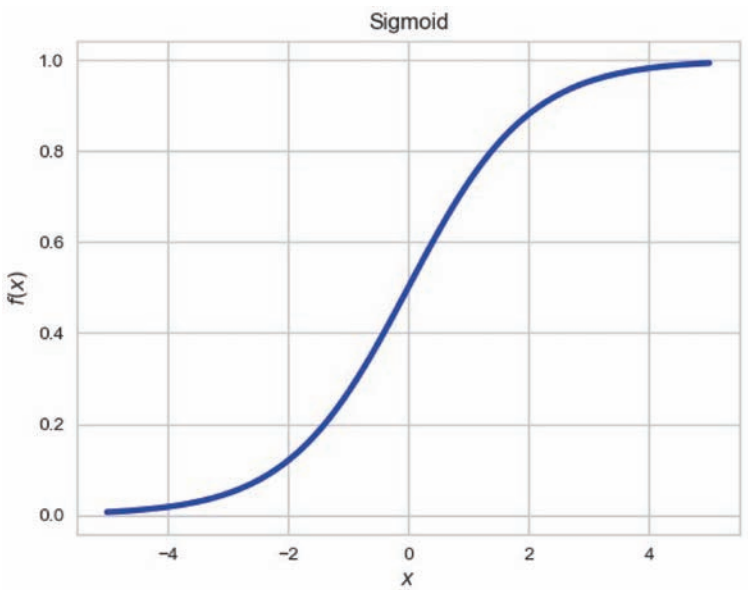


图 5.2　sigmoid 函数

在这里，根据式(5.3)，我们将会面对非零中心带来的问题，如图 5.3，sigmoid 函数和其导数均为正，σ_l' 永远是正的，而 σ_{l-1} 也永远是正的。

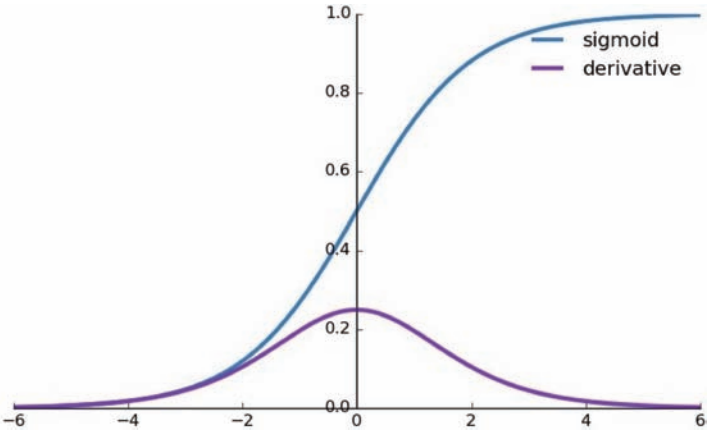


图 5.3 sigmoid 函数

唯一决定其更新方向只有损失对输出的偏导，但对于某一确定的数据，这一项也是常数。也就是说，在 sigmoid 作为激活函数的情况下，存在多个参数的情况下，权重系数的更新会全部沿着同一个方向，我们假设存在两条权重边连入到其中，另一个参数的更新也与上述情况类似。此时，我们进行两个参数的更新：

$$\Delta w_1 = -\frac{\partial L}{\partial O}\sigma'\sigma_1 \tag{5.5}$$

$$\Delta w_0 = -\frac{\partial L}{\partial O}\sigma'\sigma_0 \tag{5.6}$$

因为 $\sigma_0\sigma_1$ 均为正，其余的条件均一样，ω_1 和 ω_0 的更新沿着同一个方向，当我们遇到需要减小一个参数，而增大另一个参数的时候，此种性质会造成迭代的冗余。如图 5.4，当我们需要增大 ω_0，减小 ω_1 时，由于上一步 sigmoid 函数的输出永远为正，参数只能沿同一个方向更新，要么全部增大，要么全部减小，就形成了 Z 形折线。

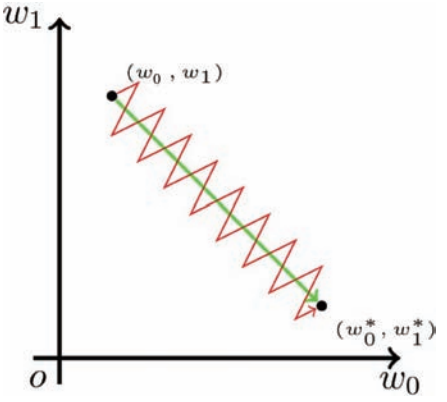
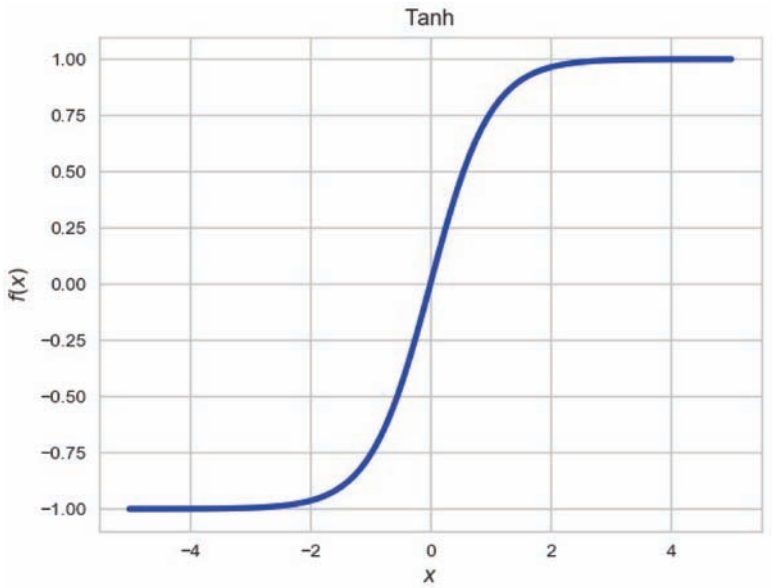


图 5.4 sigmoid 函数作为激活函数，两个参数的可能会出现的迭代过程

sigmoid 的导数和函数值对于所有权重的更新永远沿着一个方向，所以迭代必然很缓慢。一个自然的思路就是，将其中之一的输出区间扩展到以零为中心，基于这样的思路，我们可以选取 tanh 函数，如图 5.5，tanh 函数是奇函数，而其导数均为正，使得连接到同一神经元的连接权重不再同时增大或者减小，而且在图中的[－2，2]上，tanh 函数的输出值与线性函数类似，满足近似线性化的原则。

$$\tanh(z)=2\sigma(2x)-1$$



■图 5.5　tanh 函数

同时为了发挥线性的巨大优势，我们可以将中间区间的全部强制变成线性，这样就成为了 hardtanh 函数，如图 5.6。

$$y=\begin{cases}-1, & -x\leqslant -1\\ x, & -1<x\leqslant 1\\ 1, & x>1\end{cases}$$

需要注意的是，这样的拓展方法本质是将原来的平滑曲线，强行分段，也被称为“硬饱和”，对噪声就比较敏感，我们对输入加上一个极小的偏移，梯度仍然不变，但在训练过程中，我们希望在梯度流中，加了噪声和不加噪声具有不同的梯度。同时，tanh 和 hardtanh 并没有缓解梯度消失问题，反而加剧了它，所以我们需要一个在输入值太小和太大的时候，梯度变化仍需要显著的函数，我们尝试使用反正切函数，如图 5.7。

$$y=\arctan(x)$$

类似地，我们还可以使用 softsign 函数，如图 5.8。

$$\frac{x}{1+|x|}$$

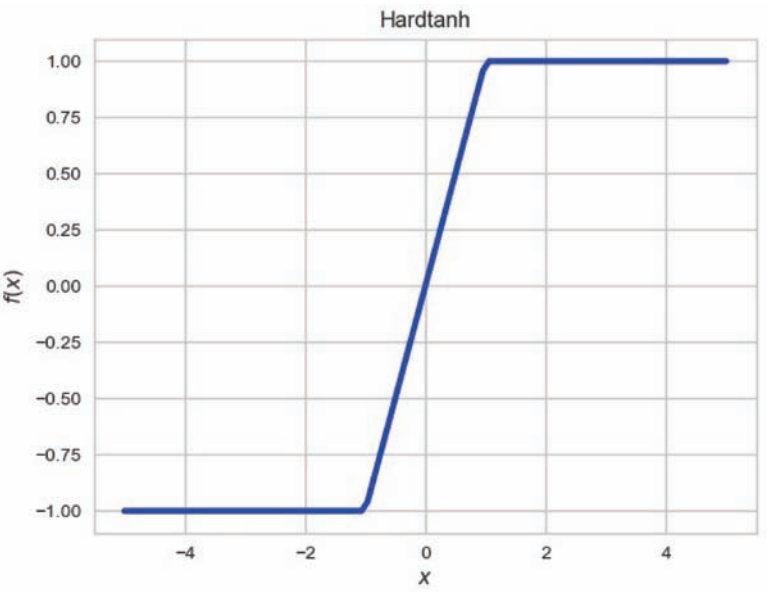


图 5.6 hardtanh 函数

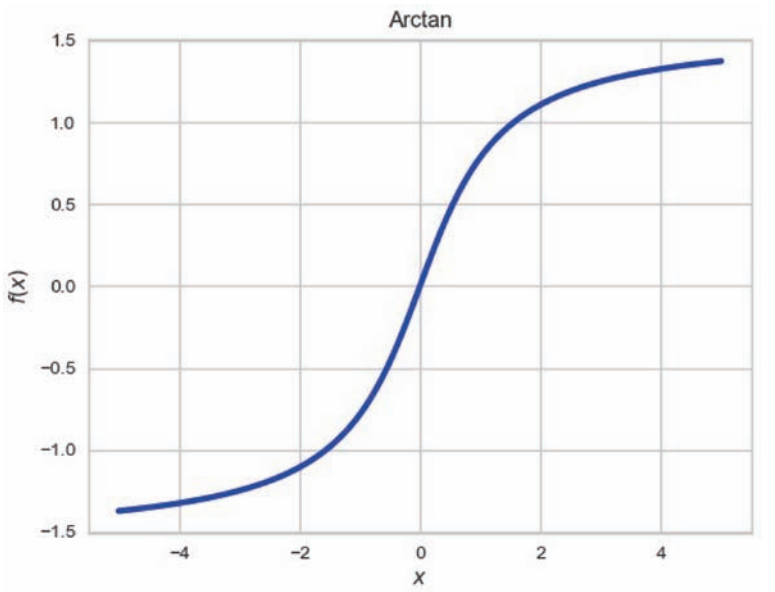


图 5.7 arctan 函数

目前对于反正切函数和 softsign 函数都介绍很少，因为它们的梯度计算起来都是比较困难的。事实证明，在能够使用 sigmoid 函数作为激活函数的深层网络中，将其替换为 tanh 等类似零中心的激活函数可以改善学习效果。

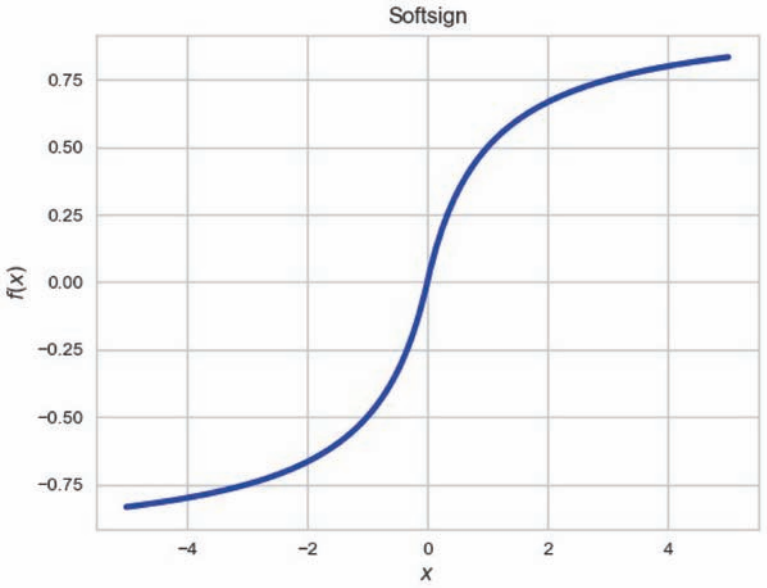


图 5.8　softsign 函数

5.3　基于线性函数的改进和 maxout 单元

对阶跃函数的陆续改进中，一方面，仍然延续了“挤压”的基本性质，事实上我们已经了解到要达到非线性的目标，“挤压”并不是必要的，另一方面，在梯度计算简单、近似线性化和缓解梯度消失和爆炸，这三者中我们最多只能拥有其中的两个。所以，我们有必要思考另一条思路，就是基于线性函数的改进。对线性函数的改进首先要解决的问题是，如何将一个线性函数变为非线性的，同时又要继承近似线性化的优点。我们寻找近似线性的激活函数，比如 Bent identity，它的定义是：

$$\frac{\sqrt{x^2+1}-1}{2}+x$$

如图 5.9，Bent identity 是一个具有近似线性性质的函数，只是导函数比起 ReLU 较为复杂。ReLU 的设计是来源于另一个简单的想法，就是将激活函数变为一个分段线性函数，ReLU(Rectified Linear Unit)它被定义为：

$$\max\{0,x\}$$

如图 5.10，ReLU 表现为一个分段的线性函数来实现非线性。为什么分段线性函数是非线性的呢？回顾我们在统计学习中讲解的线性的定义，从数学上，我们可以说 ReLU 满足齐次性，但不满足可加性；直观来说，一个线性函数会将空间分割为平直的两部分，而非线性函数则不会，分段线性函数并不平滑，按照微积分的基本原理(黎曼可积)，一个任意复

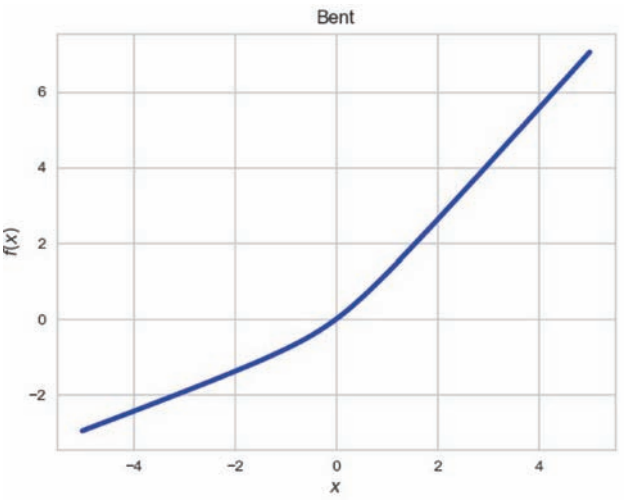


图 5.9 Bent identity 函数

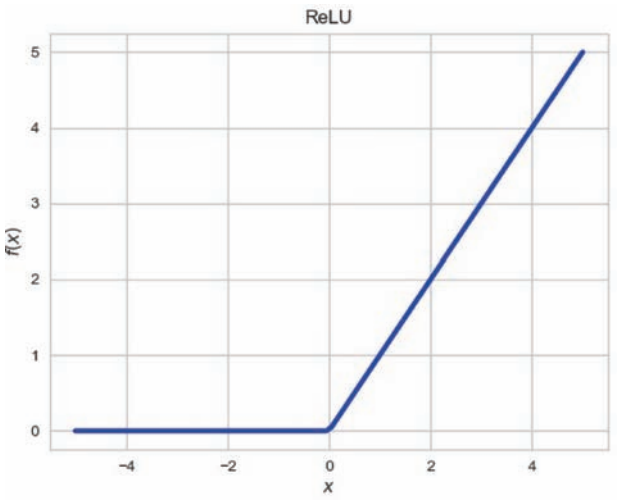


图 5.10 ReLU 函数

杂的凸函数都可以通过线性函数的多次分段来逼近。

ReLU 在很大的区间内近似线性，避免了梯度消失和梯度爆炸问题。所谓的“单侧抑制”还会带来一个巨大的好处，当神经元的输入小于零时，输出为零，神经元未被激活，只有当神经元被激活时才会有信息被传入下一级的神经元，当神经元关闭时，与之相连的权重边就不再重要，就减少了参数的数量，因为参数的数量与模型容量密切相关，这种对网络的稀疏化减小了过拟合的可能。

同时，也会带来风险，我们已经在上一节知道，反向传播更新参数时，可以看作梯度的流动。而 ReLU 输入小于零的时候，梯度也为零，当神经元一旦关闭，就很难再次激活，当我

们神经元的参数初始化为零，或者更新幅度太大时就会发生某些关闭的神经元在整个训练中就不再激活，这就是所谓的神经元“死亡”。

为了解决 ReLU 带来的潜在的优化风险，势必要破坏 ReLU 的稀疏化优势。我们可以尝试在输入小于零的时候，让其梯度不为零，比如 LeakyReLU，它被定义为：

$$y=\begin{cases}\alpha x, & x\leqslant 0\\ x, & x>0\end{cases}$$

其中，α 是一个非常小的参数，就是希望能够避免神经元的死亡，又同时尽可能地保持原本 ReLU 的优点，当输入小于零时，神经元的激活也就很微弱，绝对值也不会特别大，这与我们的 L_2 正则化有相似之处，如图 5.11，同时我们可以注意到 LeakyReLU 仍然是硬饱和的。

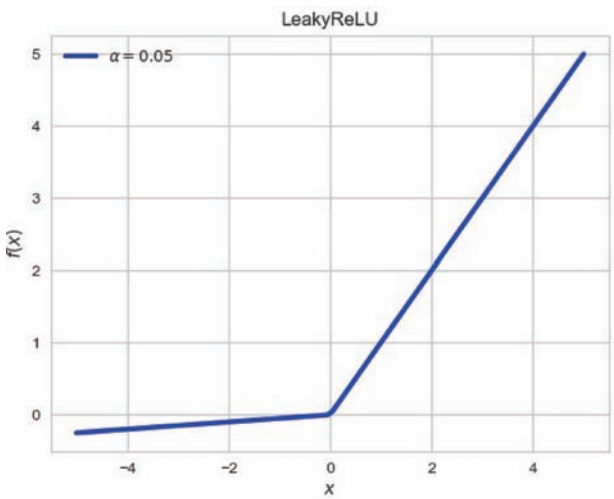


图 5.11　LeakyReLU 函数

ELU 单元则改进 ReLU 的硬饱和特点，它被定义为：

$$y=\begin{cases}\alpha(\varepsilon^{x}-1), & x\leqslant 0\\ x, & x>0\end{cases}$$

如图 5.12，ELU 函数是平滑的。注意到我们使用 ELU 或者 LeakyReLU 时，引入了一个需要提前设定的参数，这个参数随着神经网络的结构和数据的不同，可能会存在差异，大多数情况下，我们会引入参数共享，即会对于所有的单元采取相同的 α，但将其确定好，仍然需要花费不少工夫。把如果这个参数可以内嵌到神经网络的学习过程中，使其变为一个可以被训练的参数，那么性能就会更加灵活，这就是所谓的 PReLU(Parametric ReLU)，具体的形式与 ELU 和 LeakyReLU 相同，只是在训练过程中，我们需要多计算一个参数的梯度：

$$\Delta\alpha=\frac{\partial L}{\partial\alpha}=\frac{\alpha L}{\partial f}\frac{\partial f}{\partial\alpha}$$

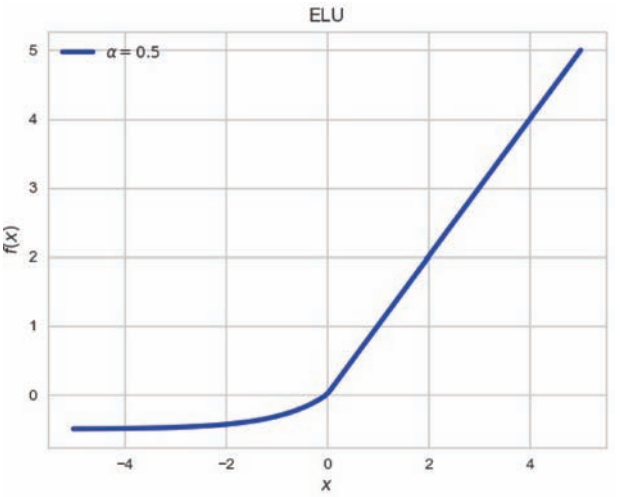


图 5.12 ELU 函数

在 PReLU 中引入的可训练的参数是为了激活函数具备一定的灵活性，灵活性的另一种实现方法则可以基于门控(gated)加强对激活函数的参数化，比如 Swish 函数，它被定义为：

$$y = x\sigma(\beta x) \tag{5.7}$$

它使用 sigmoid 函数来控制近似行为，当 sigmoid 函数接近激活时，Swish 函数接近 x，当 sigmoid 函数接近关闭时，Swish 函数接近 0。sigmoid 函数内部的参数 β 来控制上述渐进行为的快慢，参数越大，表示随着 x 的变化，越快的接近激活或者关闭。

如图 5.13，随着 β 的增大，Swish 函数越来越接近 ReLU 的表现，当 $\beta=0$，函数就成为了一个完全线性函数。

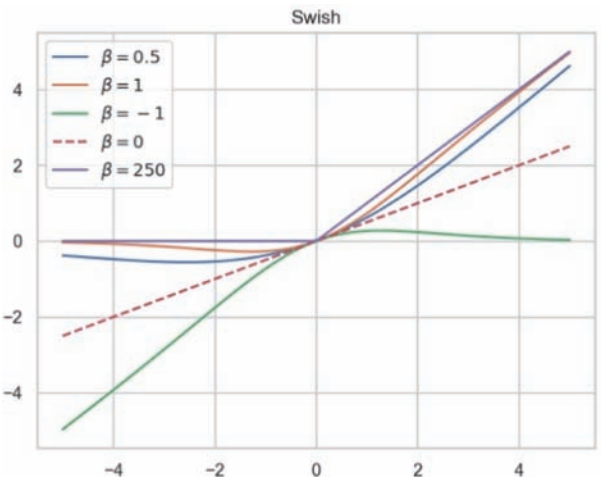


图 5.13 Swish 函数

灵活性的最极端方式就是把激活函数本身当作学习的对象，比如 maxout 单元，它接受从上一层神经元的全部输出，而非传统激活函数只接受上一层神经元输出的和。考虑不加偏置的全连接网络，上一级有 i 个节点，那么这一层第 j 个 ReLU 接收的输入就是 $z_j = X_i^{\mathrm{T}} w_{ij}$，进入激活后，产生的输出就是 $\max\{0, z_j\}$。

而每一个 maxout 单元接受上一层神经元的全部输出，是一个向量，$[x_1, x_2, \cdots, x_i]$。如图 5.14，假设有两个输入，标准的 ReLU 神经元产生的是 $\max\{0, w_1 x_1 + w_2 x_2\}$，而添加 3 个神经元的 maxout 神经元产生的输出为：

$$\max\{v_{11}x_1 + v_{12}x_2, v_{12}x_1 + v_{22}x_2, v_{13}x_1 + v_{23}x_2\} \tag{5.8}$$

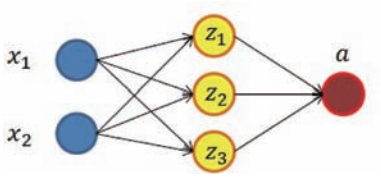


图 5.14　maxout 激活函数

注意图中最后三条边并非权重边，在这个例子中，maxout 包含 6 个参数，若含有 k 个神经元，则参数的数量会变成原来的 k 倍。事实上，maxout 通过取最大值的方式可以逼近任意的凸函数，随着 k 的增加，拟合能力就会越来越强，同时参数的增多使得它比起一般的激活函数具有更多过拟合风险，一般需要加上更为有力的正则化机制。

5.4　使用 keras

经过上述的学习，我们都已经知道神经单元的类型会对网络的训练产生重要的影响，梯度消失是最为典型的情况。神经元影响梯度消失程度分为两种，一种是损失函数的选择，均方误差比起交叉熵更容易产生梯度消失；另一种是隐藏单元的选择，sigmoid 函数比起 ReLU 更容易产生梯度消失。我们会对这两种情况作出详细验证。

在这里，我们采用 MNIST 数据集，这是手写数字识别的著名数据。我们将数据导入，并定义了一个非常简单的模型：

```
import numpy as np
from keras.datasets import mnist
from sklearn.model_selection import KFold
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers
from keras.utils import to_categorical
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# 导入数据
(X_train,y_train),(X_test,y_test) = mnist.load_data()

train_labels = to_categorical(y_train)
test_labels = to_categorical(y_test)

X_train_normal = X_train.reshape(60000,28 * 28)
X_train_normal = X_train_normal.astype('float32') / 255
X_test_normal = X_test.reshape(10000, 28 * 28) X_test_normal = X_test_normal.astype('
float32') / 255

# 定义模型
def full_model(act, loss):
    model = Sequential()
    model.add(Dense(512,activation = act,input_shape = (28 * 28,)))
    model.add(Dense(256,activation = act))
    model.add(Dense(128,activation = act))
    model.add(Dense(64,activation = act))
    model.add(Dense(10,activation = 'softmax'))
    model.compile(optimizer = optimizers.Adam(),loss = loss ,\
                metrics = ['accuracy'])
    return(model)

def train_plot_loss(model, epochs, batch_size, losses):
    losses_his = {}
    for loss in losses:
        full_model = model(act = 'sigmoid',loss = loss)
        his = full_model.fit(X_train_normal, train_labels,
            batch_size = batch_size,\validation_data = (X_test_normal,test_labels), \
                        verbose = 1,epochs = epochs)
        losses_his[loss] = his.history
    sns.set(style = 'white')
    for key_loss in losses_his.keys():
        plt.plot(range(epochs), losses_his[key_loss]['loss'],label =
            key_loss)
        plt.legend()
        plt.show()
    return losses_his

losses_his = train_plot_loss(model = full_model, epochs = 10, batch_size
    = 256,\losses = ['mean_squared_error','categorical_crossentropy'])
```

在上段代码中，我们搭建了一个四个隐层的模型，并且定义了训练模型的函数，其中使用了均方误差和交叉熵分别训练模型。得到图 5.15，可以看出均方误差作为损失函数在训练集上几乎没有变化，而交叉熵却在缓慢下降，这说明使用均方误差可能会出现参数更新缓慢的情形。

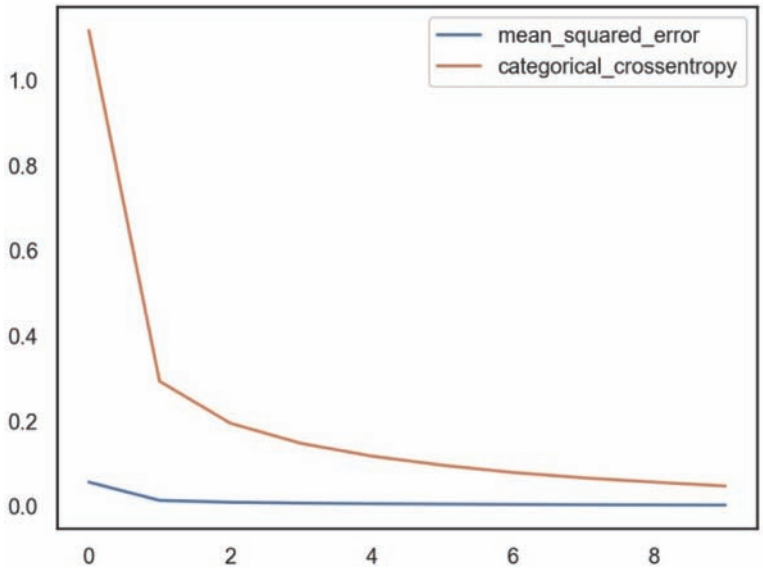


图 5.15　均方误差和交叉熵随着迭代的变化

单纯的看训练集的损失并不能真的确定均方误差会让模型难以训练，更何况交叉熵和均方误差本来就是两个不同的函数，放在一起看没有太多的意义，所以我们还需要观察准确率和测试集的表现，添加如下代码：

```
....
sns.set(style = 'white')
for key_loss in losses_his.keys():
    plt.plot(range(10),losses_his[key_loss]['accuracy'],label = key_loss)
    # plt.plot(range(10), losses_his[key_loss]['val_accuracy'],label =
        key_loss)
plt.legend()
plt.show()
....
```

如图 5.16，无论是训练集还是测试集，使用均方误差在迭代的整个过程的准确率都小于使用交叉熵。这说明交叉熵能归模型进行更有效的训练。

经过上述的验证，我们对于分类问题采用交叉熵会让模型得到更好的训练，接着我们来探究隐藏单元对模型的影响，添加代码如下：

```
.....
def train_plot_act(model, epochs, batch_size, acts):
    acts_his = {}
    for act in acts:
        full_model = model(act = act, loss = 'categorical_crossentropy') his = full_model.
        fit(X_train_normal, train_labels,
```

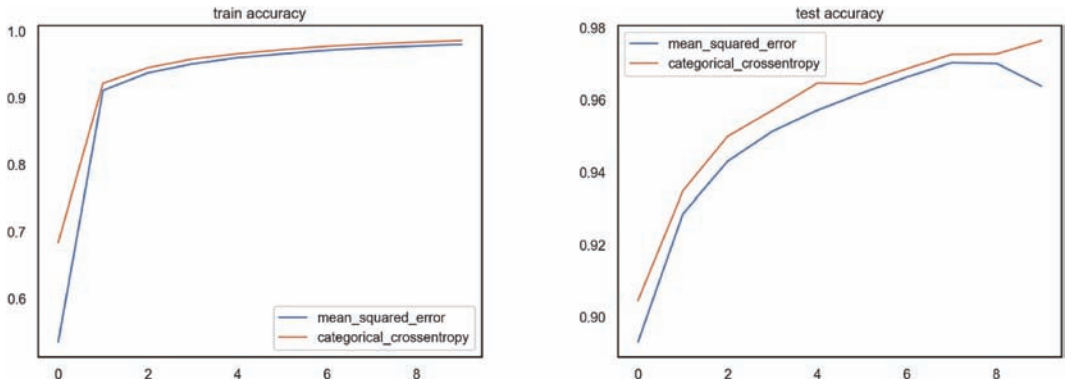


图 5.16 均方误差和交叉熵作为损失，训练集和测试集上的准确率变化

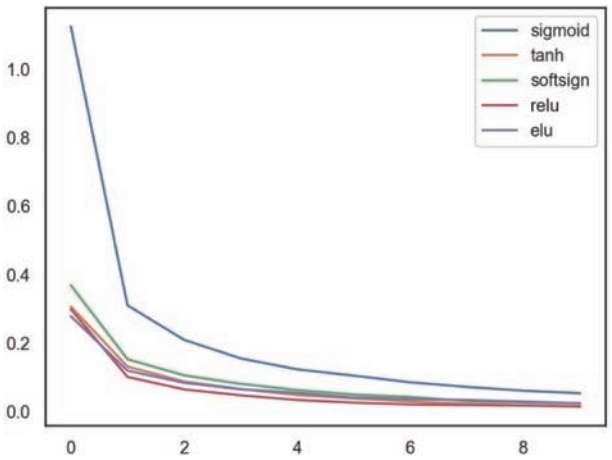
```
            batch_size = batch_size, validation_data = (X_test_normal, test_labels), \
                        verbose = 1, epochs = epochs)
        acts_his[act] = his.history
    sns.set(style = 'white')
    for key_act in acts_his.keys():
            plt.plot(range(epochs), acts_his[key_act]['loss'], label = key_act)
        plt.legend()
            plt.show()
    return acts_his

acts_his = train_plot_act(model = full_model, epochs = 10, batch_size =
    256,\acts = ['sigmoid','tanh','softsign','relu','elu'])
```

在上述代码中，我们分别设置了激活函数采用 sigmoid，tanh，softsign，relu 和 elu，并分别对同一个模型进行训练，结果如图 5.17，sigmoid 函数训练损失一直处于其他激活函数的上方，这正是因为 sigmoid 的饱和区较大，在误差反向传播中加重了梯度消失。并且因为非零中心的缘故，损失下降到和其他激活函数一样的水平需要更多次的迭代。

同时，我们也可以对使用不同激活函数时测试集和训练集上的准确率作出评估，添加代码如下：

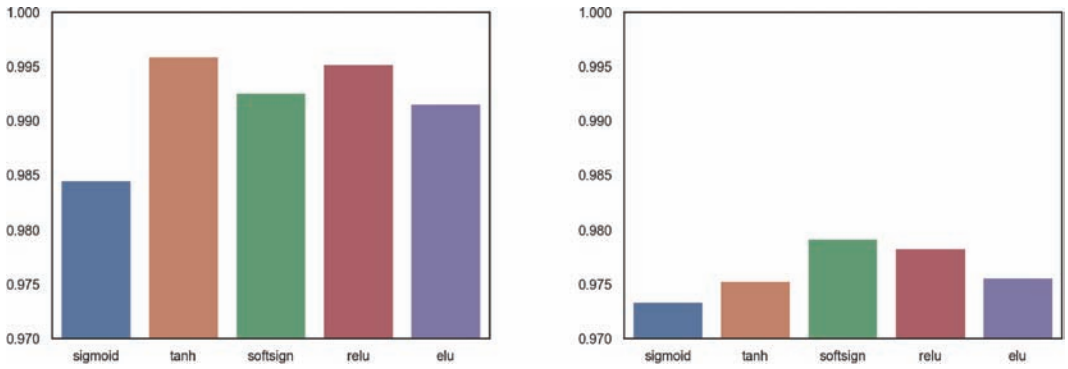
```
act_train_acc = [acts_his[i]['accuracy'][-1] for i in acts_his]
act_val_acc = [acts_his[i]['val_accuracy'][-1] for i in acts_his]
acts = [i for i in acts_his]
sns.set(style = 'white')
plt.ylim(0.97, 1)
sns.barplot(acts, act_train_acc)
#sns.barplot(acts, act_val_acc)
plt.show()
```

如图 5.18，我们发现不同的激活函数在训练集和测试集上准确率的区别很小，这是因为神经网络的激活函数对于表示学习可能只起到非线性的作用，而 sigmoid 函数比起其他



■图 5.17 不同类型的隐藏单元的训练损失随着迭代的变化

激活函数得到的准确率在训练集和测试集上都要小，也是因为网络没有得到有效训练的缘故。



■图 5.18 10 个 epochs 后，不同类型的隐藏单元在训练集和测试集上的准确率