降维

- 简化数据

降维

- 许多机器学习问题涉及训练实例的几千甚至上百万个特征。后面我们将会看到，这不仅导致训练非常缓慢，也让我们更加难以找到好的解决方案。这个问题通常被称为维度的诅咒。
- 我们一般可以大量减少特征的数量，将棘手的问题转化成容易解决的问题。例如，MNIST图像：图像边框的像素位上几乎全是白色，所以我们完全可以在训练集中抛弃这些像素位，也不会丢失太多信息。

降维

- 本节将探讨维度的诅咒，大致了解高维空间中发生的事情。然后，我们将介绍两种主要的数据降维方法（投影和流形学习），并学习现在最流行的三种数据降维技术：PCA、Kernal PCA以及LLE 。

维度的诅咒

- 我们太习惯三维空间的生活，所以当我们试图去想象一个高维空间时，我们的直觉思维很难成功。即使是一个基本的四维超立方体，我们也很难在脑海中想象出来，更不用说在一个千维空间中弯曲的200维椭圆体。

维度的诅咒

- 事实证明，在高维空间中，许多事物的行为都迥然不同。例如，如果你在一个单位平面（1×1的正方形）内随机选择一个点，那么这个点离边界的距离小于0.001的概率只有约0.4%（也就是说，一个随机的点不大可能刚好位于某个维度的“极端”）。但是，在一个10000维的单位超立方体（1×1…×1立方体，一万个1）中，这个概率大于99.999999%。

维度的诅咒

- 还有一个更麻烦的区别：如果你在单位平面中随机挑两个点，这两个点之间的平均距离大约为0.52。如果在三维的单位立方体中随机挑两个点，两点之间的平均距离大约为0.66。但是，如果在一个100万维的超立方体中随机挑两个点呢？
- 平均距离大约为408.25 (约等于sqrt(1000,000/6))这个事实说明大多数训练实例可能彼此之间相距很远。这也意味着新的实例很可能远离任何一个训练实例，导致预测跟低维度相比，更加不可靠。简而言之，训练集的维度越高，过度拟合的风险就越大。

维度的诅咒

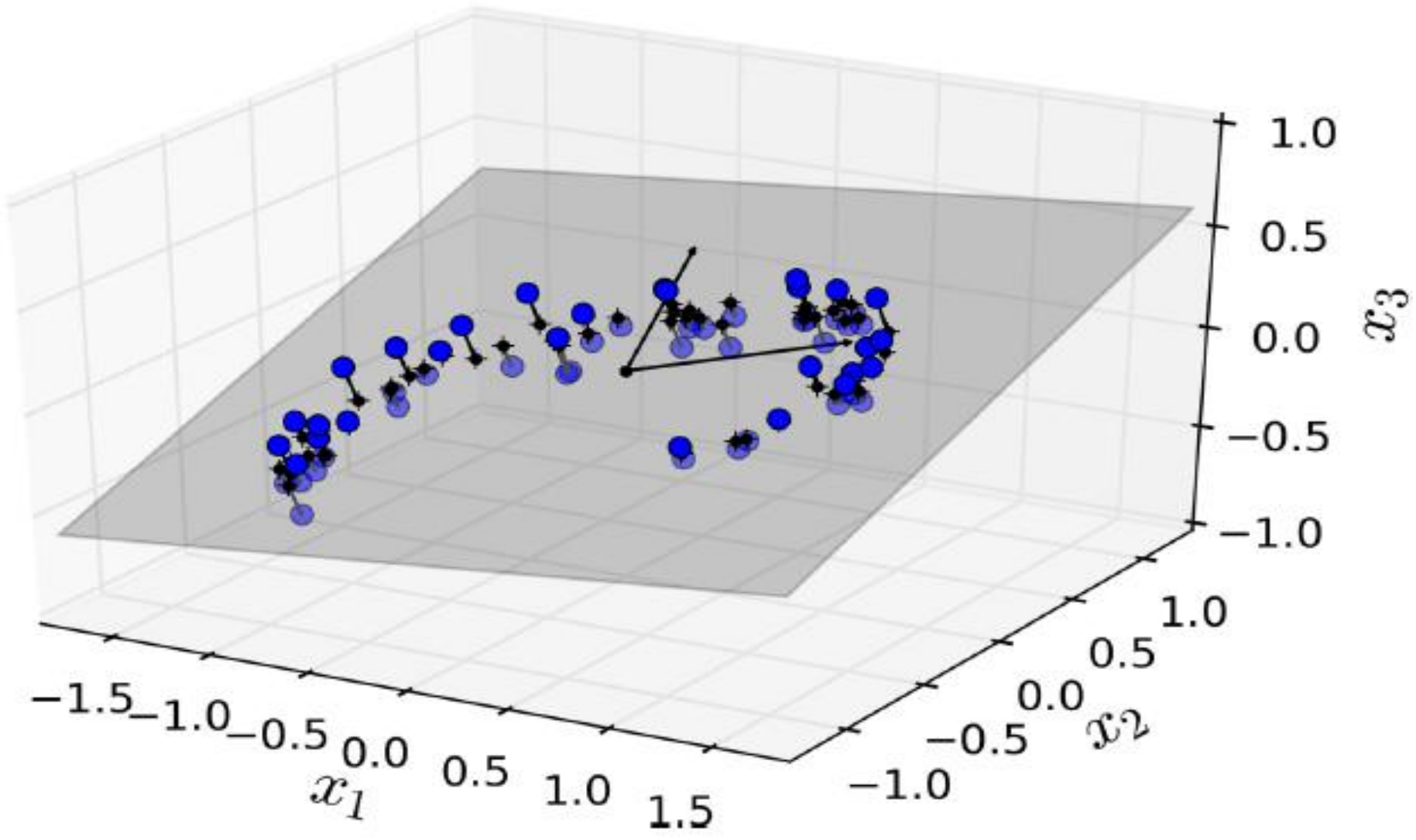
- 理论上来说，通过增大训练集，使训练实例达到足够的密度，是可以解开维度的诅咒的。然而不幸的是，实践中，要达到给定密度所需要的训练实例数量随着维度增加呈指数式上升。仅仅100个特征下（远小于MNIST问题），要让所有训练实例（假设在所有维度上平均分布）之间的平均距离小于0.1，你需要的训练实例数量就比可观察宇宙中的原子数量还要多。

数据降维的主要方法

- 投影
- 在大多数现实世界的问题里，训练实例在所有维度上并不是均匀分布的。许多特征几乎是不变的，也有许多特征是高度相关联的（如前面讨论的MNIST数据集）。因此，高维空间的所有训练实例实际上（或近似于）受一个低得多的低维子空间所影响。这听起来很抽象，所以我们来看一个例子。在图8-2中，你可以看到一个由圆圈表示的3D数据集。

数据降维的主要方法

- 投影

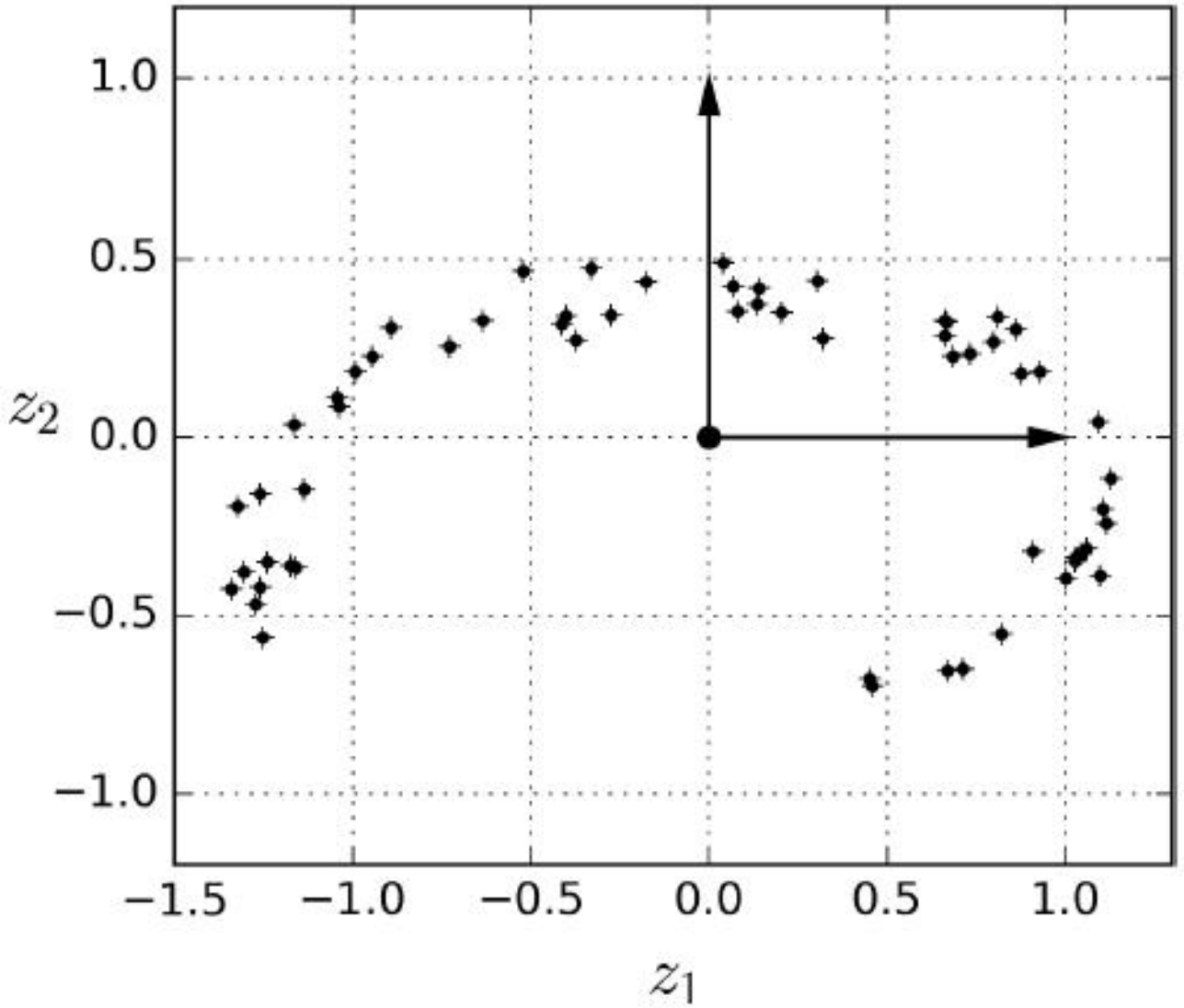


数据降维的主要方法

- 投影
- 注意看，所有的训练实例都紧挨着一个平面：这就是高维（3D）空间的低维（2D）子空间。现在，如果我们将每个训练实例垂直投影到这个子空间（如图中实例到平面之间的短线所示），我们将得到如图8-3所示的新2D数据集。我们已经将数据集维度从三维降到了二维。注意，图中的轴对应的是新特征z_1和z_2（平面上投影的坐标）。

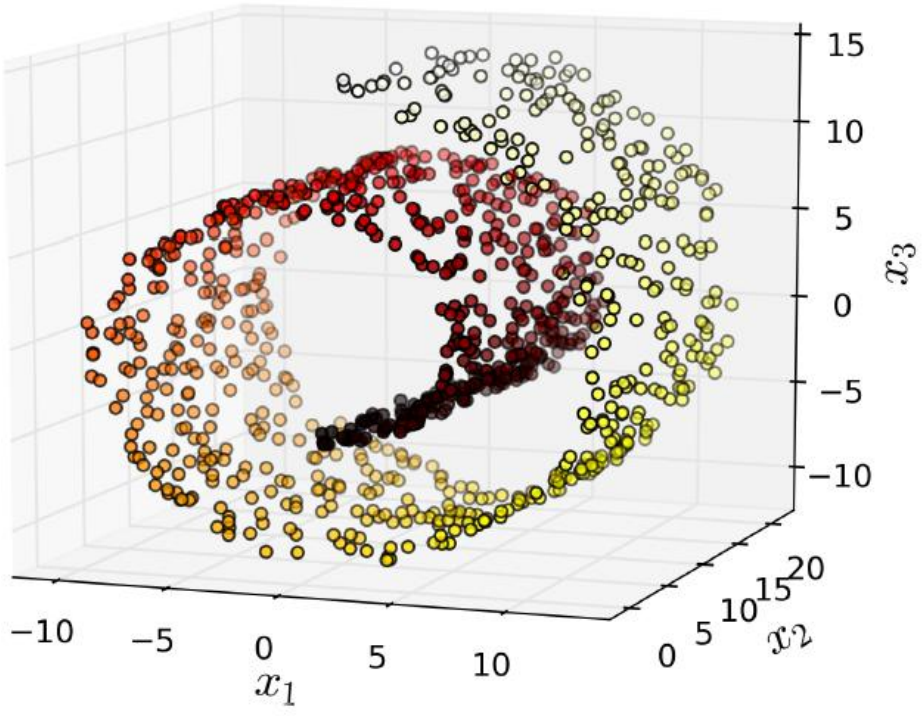
数据降维的主要方法

- 投影



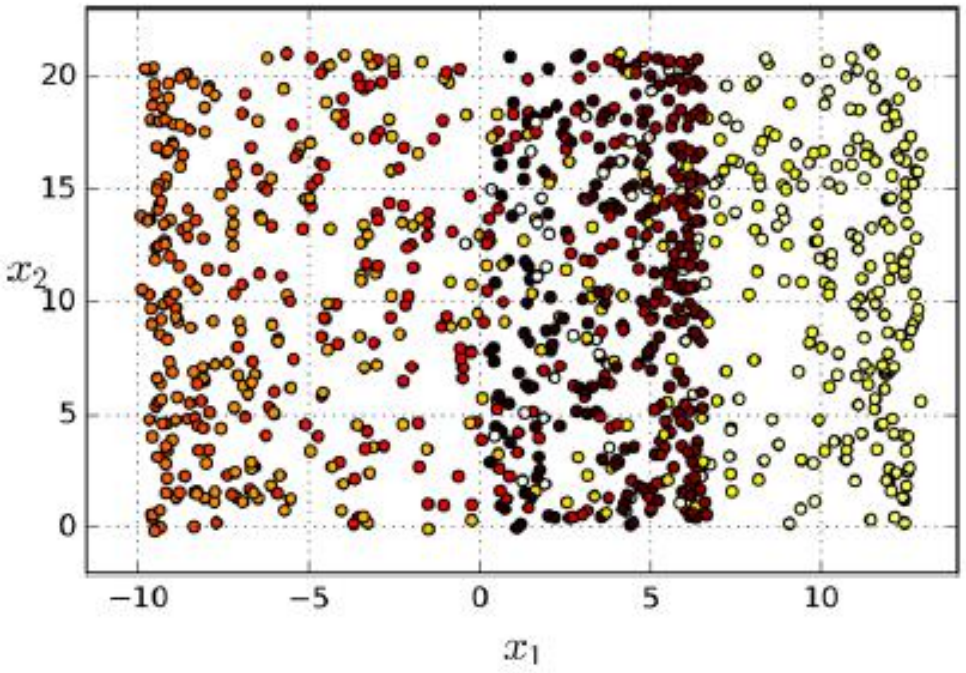
数据降维的主要方法

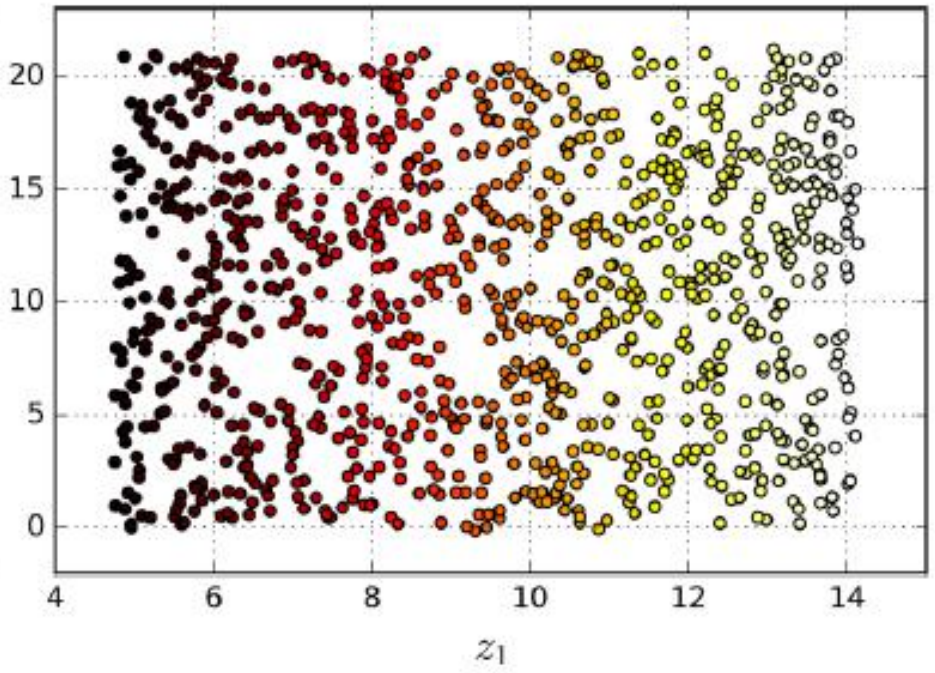
- 投影
- 不过投影并不总是降维的最佳方法。在许多情况下，子空间可能会弯曲或转动，比如图8-4所示的著名的瑞士卷玩具数据集。



数据降维的主要方法

- 投影





数据降维的主要方法

- 流形学习
- 瑞士卷就是二维流形的一个例子。简单地说，2D流形就是一个能够在更高维空间里面弯曲和扭转的2D形状。更概括地说，d维流形就是n（其中，d＜n）维空间的一部分，局部类似于一个d维超平面。在瑞士卷的例子中，d=2，n=3：它局部类似于一个2D平面，但是在第三个维度上卷起。

数据降维的主要方法

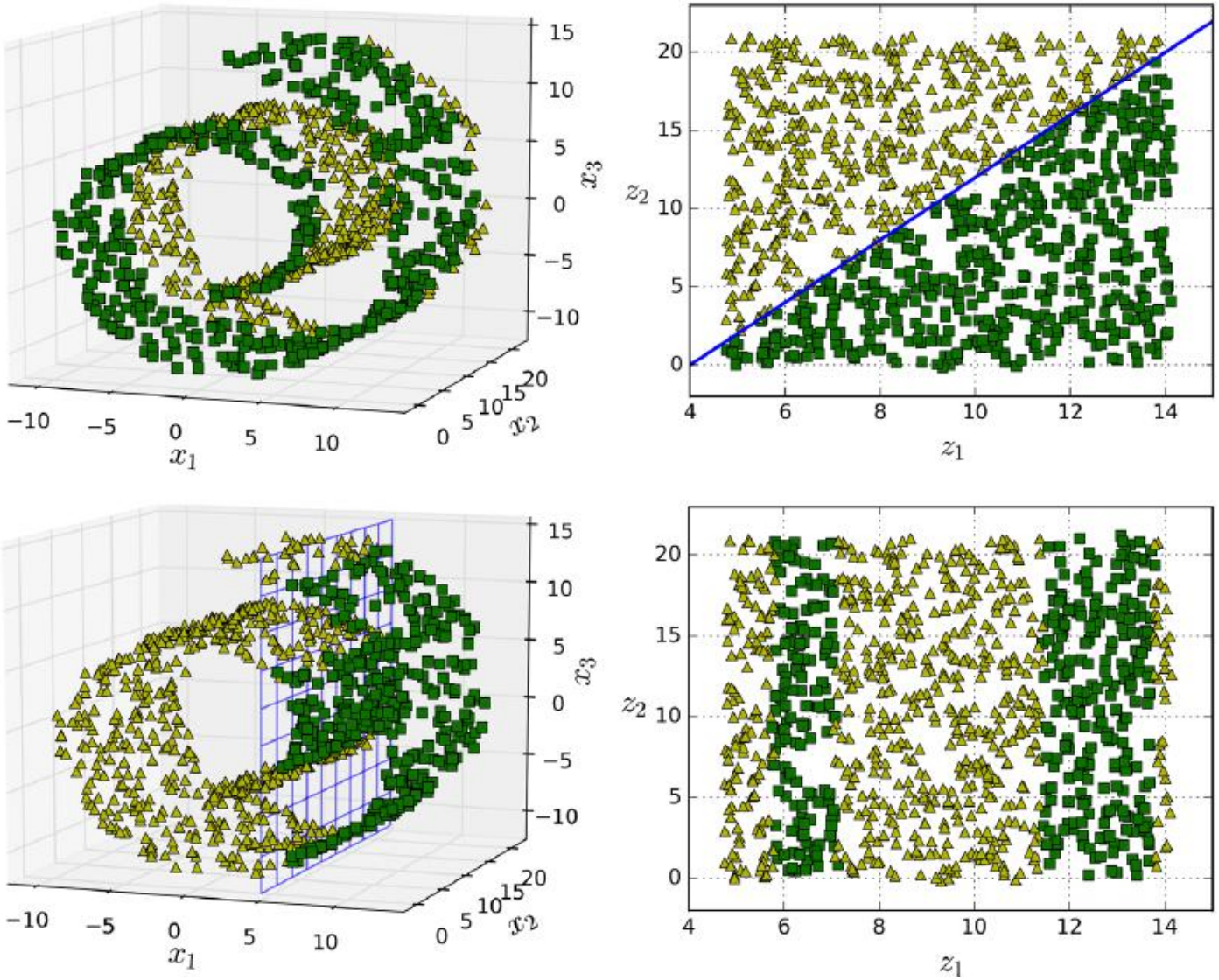
- 流形学习
- 许多降维算法是通过对训练实例进行流形建模来实现的，这被称为流形学习。它依赖于流形假设，也称为流形假说，认为大多数现实世界的高维度数据集存在一个低维度的流形来重新表示。这个假设通常是凭经验观察的。

数据降维的主要方法

- 流形学习
- 流形假设通常还伴随着一个隐含的假设：如果能用低维空间的流形表示，手头的任务（例如分类或者回归）将变得更简单。例如，图8-6的上面一行，瑞士卷被分为两类：3D空间中（左上）决策边界将会相当复杂，但是在展开的2D流形空间（右上），决策边界是一条简单的直线。

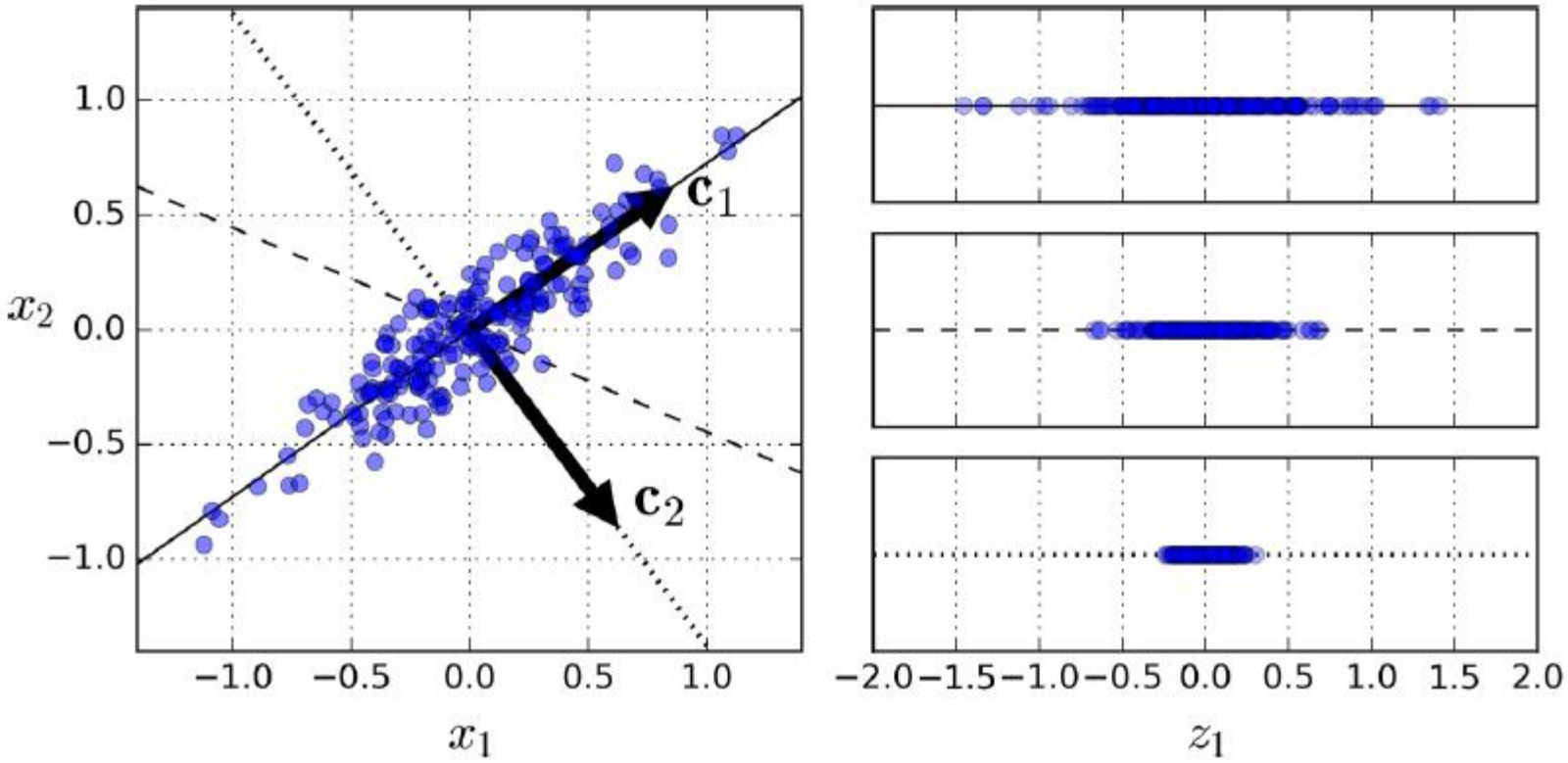
数据降维的主要方法

- 流形学习



PCA

- 主成分分析（PCA）是迄今为止最流行的降维算法。它先是识别出最接近数据的超平面，然后将数据投影其上。



主成分

- 主成分分析（PCA）可以在训练集中识别出哪条轴对差异性的贡献度最高。在图8-7中，即是由实线表示的轴。同时它也找出了第二条轴，它对剩余差异性的贡献度最高，与第一条轴垂直。因为这个例子是二维的，所以除了这条点线再没有其他。如果是在更高维数据集中，PCA还会找到与前两条都正交的第三条轴，以及第四条、第五条，等等—轴的数量与数据集维度数量相同。

主成分

- 所以怎么找到训练集的主成分呢？还好有一种标准矩阵分解技术，叫作奇异值分解（SVD）。它可以将训练集矩阵X分解成三个矩阵的点积$\mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^T$, 其中的 $\mathbf{V}^T$ 正包含我们想要的所有主成分，如公式8-1所示。

Equation 8-1. Principal components matrix

$$\mathbf{V}^T = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

主成分

```
X_centered = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
```

低维度投影

- 一旦确定了所有主成分，就可以将数据集投影到由前d个主成分定义的超平面上，从而将数据集的维度降到d维。这个超平面的选择，能确保投影保留尽可能多的差异性。例如，在图8-2中，3D数据集投影到由前两个主成分定义的2D平面上，就保留了原始数据集的大部分差异。因此，2D投影看起来非常像原始的3D数据集。

低维度投影

- 要将训练集投影到超平面上，简单地计算训练集矩阵 **X** 和矩阵 $\mathbf{W}_d$,的点积即可。 $\mathbf{W}_d$ 是是包含前d个主成分的矩阵（即由矩阵 $\mathbf{V}^T$ 的前d列组成的矩阵），参见公式8-2。

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X} \cdot \mathbf{W}_d$$

低维度投影

- Scikit-Learn的PCA类也使用SVD分解来实现主成分分析。以下代码应用PCA将数据集的维度降到二维（注意它会自动处理数据集中）：

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

- 第一个主成分即等于 pca.components_.T[:, 0]).

方差解释率

- 另一个非常有用的信息是每个主成分的方差解释率，它可以通过变量explained_variance_ratio_获得。它表示每个主成分轴对整个数据集的方差的贡献度。例如，我们看图8-2所示的3D数据集中前两个主成分的方差解释率：

```
>>> print(pca.explained_variance_ratio_)
array([ 0.84248607, 0.14631839])
```

选择正确数量的维度

- 除了武断地选择要降至的维度数量，通常来说更好的办法是将靠前的主成分方差解释率依次相加，直到得到足够大比例的方差（例如95%），这时的维度数量就是很好的选择。

```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

选择正确数量的维度

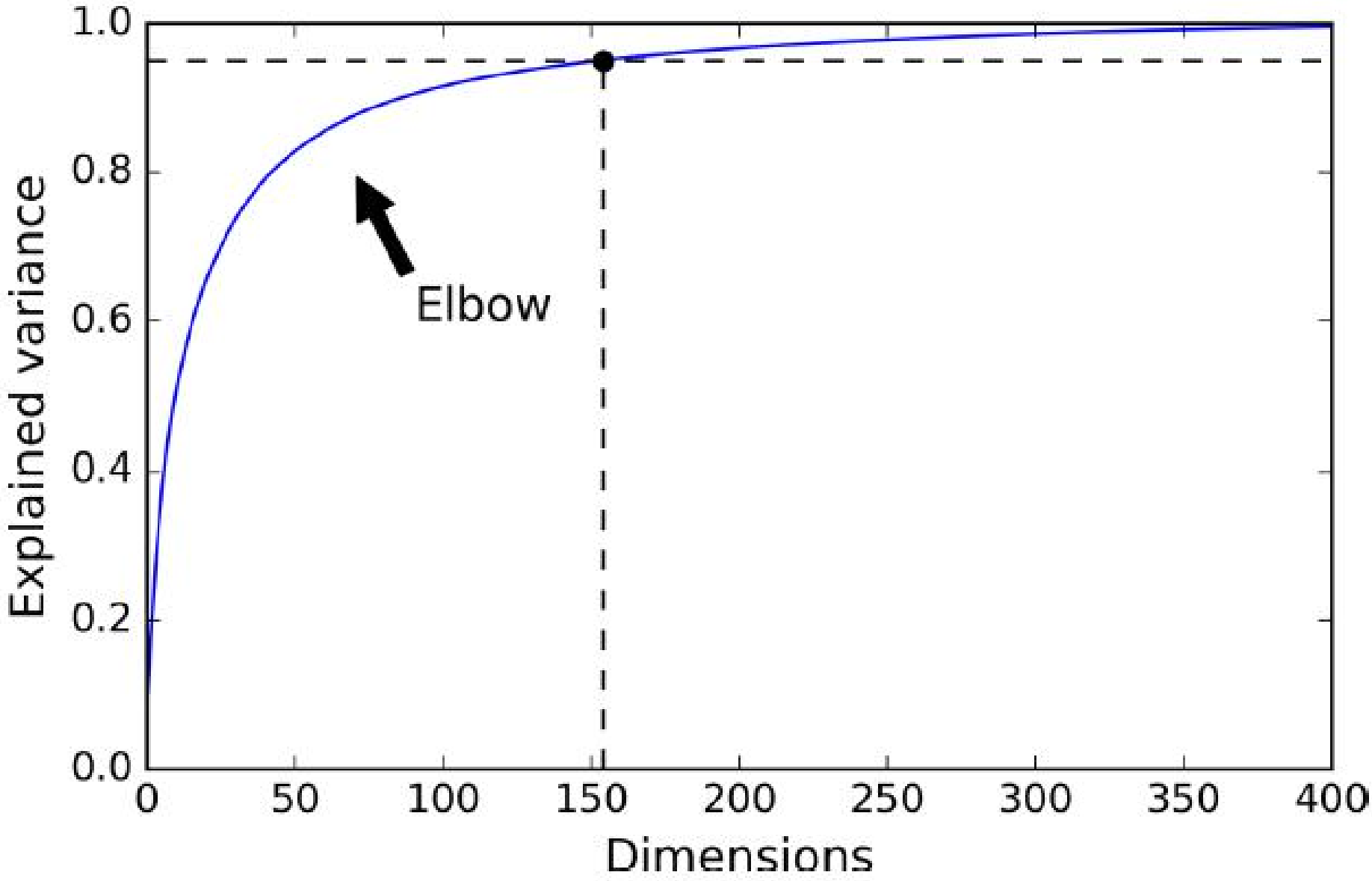
- 还有一个更好的方法：不需要指定保留主成分的数量，你可以直接将n_components设置为0.0到1.0之间的浮点数，表示希望保留的方差比：

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)
```

选择正确数量的维度

- 另外，还可以将解释方差绘制成关于维度数量的函数（绘制cumsum即可，见图8-8）。曲线通常都会有一个拐点，说明方差停止快速增长。你可以将其视为数据集的本征维数。从本例中可以看出，将维度数量降低至100维，不会损失太多的解释方差。

选择正确数量的维度


1.0
0.8
0.6
0.4
0.2
0.0
Explained variance
Elbow
0
50
100
150
200
250
300
350
400
Dimensions

PCA压缩

- 显然，降维之后训练集占用的空间要小得多。例如，对MNIST数据集应用主成分分析，然后保留其方差的95%。你会发现，原来每个实例的784个特征变得只有150多个特征。所以这保留了绝大部分差异性的同时，数据集的大小变为不到原始的20%！这是一个合理的压缩比，你可以看看它如何极大提升分类算法（例如SVM分类器）的速度。

PCA压缩

- 在PCA投影上运行投影的逆转换，也可以将缩小的数据集解压缩回784维数据集。当然，你得到的并非原始的数据，因为投影时损失了一部分信息（5%被丢弃的方差），但是它很大可能非常接近于原始数据。原始数据和重建数据（压缩之后解压缩）之间的均方距离，被称为**重建误差**。

MNIST compression preserving 95% of the variance

```
pca = PCA(n_components = 154)
X_mnist_reduced = pca.fit_transform(X_mnist)
X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)
```

Original

Compressed

增量PCA

- 前面关于主成分分析的种种实现，问题在于，它需要整个训练集都进入内存，才能运行SVD算法。幸运的是，我们有增量主成分分析（IPCA）算法：你可以将训练集分成一个个小批量，一次给IPCA算法喂一个。对于大型训练集来说，这个方法很有用，并且还可以在线应用PCA（也就是新实例产生时，算法开始运行）。

增量PCA

```
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_mnist, n_batches):
    inc_pca.partial_fit(X_batch)
X_mnist_reduced = inc_pca.transform(X_mnist)
```

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))
batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

随机PCA

- Scikit-Learn还提供了另一种实施PCA的选项，称为随机PCA。这是一个随机算法，可以快速找到前d个主成分的近似值。它的计算复杂度是 $O(m \times d^2) + O(d^3)$,而不是 $O(m \times n^2) + O(n^3)$, 所以当d远小于n时，它比前面提到的算法要快得多。

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_mnist)
```

核主成分分析

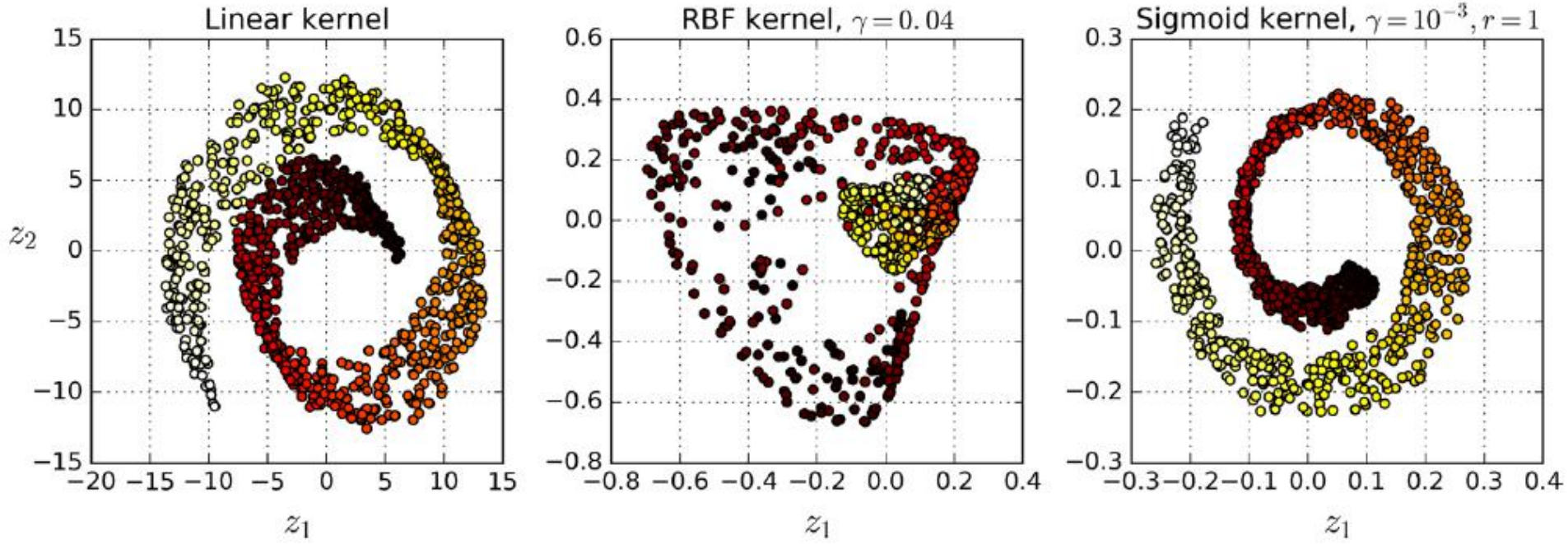
- 第5章讨论了核技巧，它是一种数学技巧，隐性地将实例映射到非常高维的空间（称为特征空间），从而使支持向量机能够进行非线性分类和回归。回想一下，高维特征空间的线性决策边界如何对应于原始空间中复杂的非线性决策边界。
- 事实证明，同样的技巧也可应用于PCA，使复杂的非线性投影降维成为可能。这就是所谓的**核主成分分析（kPCA）**。它擅长在投影后保留实例的集群，有时甚至也能展开近似于一个扭曲流形的数据集。

核主成分分析

from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)

X_reduced = rbf_pca.fit_transform(X)

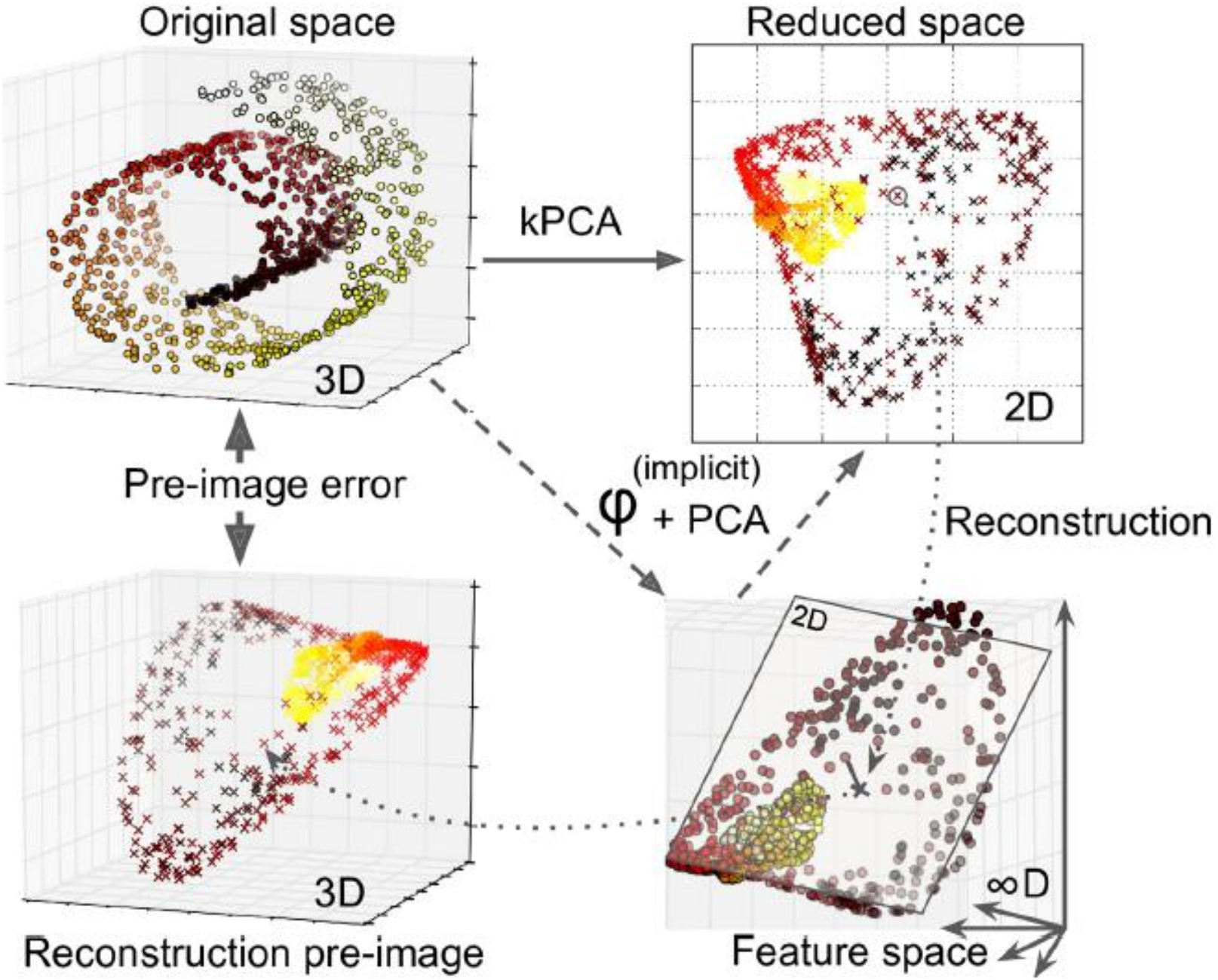


选择核函数和调整超参数

- 由于kPCA是一种无监督的学习算法，因此没有明显的性能指标来帮你选择最佳的核函数和超参数值。而降维通常是监督式学习任务（例如分类）的准备步骤，所以可以使用网格搜索，来找到使任务性能最佳的核和超参数。例如，下面的代码创建了一个两步流水线，首先使用kPCA将维度降至二维，然后应用逻辑回归进行分类。接下来使用GridSearchCV为kPCA找到最佳的核和gamma值，从而在流水线最后获得最准确的分类：

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])
param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.04333333333333335, 'kpca__kernel': 'rbf'}
```

Kernel PCA和重建原像误差



Kernel PCA和重建原像误差

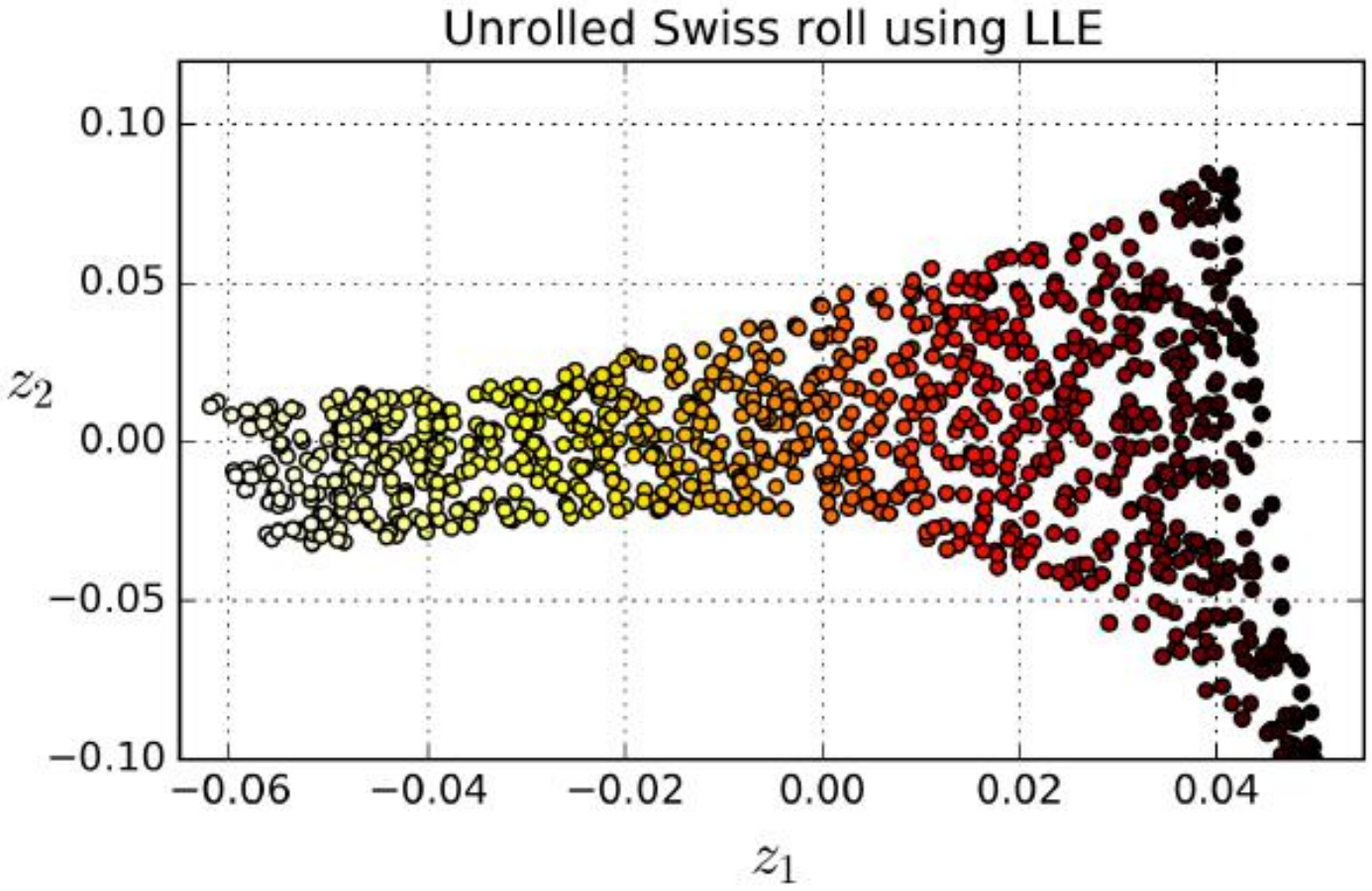
```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)

>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

局部线性嵌入 LLE

- 局部线性嵌入（LLE）是另一种非常强大的非线性降维（NLDR）技术。不像之前的算法依赖于投影，它是一种流形学习技术。简单来说，LLE首先测量每个算法如何与其最近的邻居（c.n.）线性相关，然后为训练集寻找一个能最大程度保留这些局部关系的低维表示（细节稍后解释）。这使得它特别擅长展开弯曲的流形，特别是没有太多噪声时。

局部线性嵌入 LLE

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```



局部线性嵌入 LLE

- 首先，对于每个训练实例$x^{(i)}$，算法会识别出离它最近的k个邻居（k=10），将$x^{(i)}$重建为这些邻居的线性函数。也就是要找到权重$w_{i,j}$使实例$x^{(i)}$和$\sum_{j=1}^{m} w_{i,j}x^{(j)}$之间的距离平方最小，如果实例$x^{(j)}$不是实例$x^{(i)}$的k个最近的邻居之一，$w_{i,j}=0$。因此，LLE的第一步就是公式8-4所示的约束优化问题，其中W是包含所有权重$w_{i,j}$的权重矩阵，第二个约束则是简单地对每个训练实例$x^{(i)}$的权重进行归一。 *Equation 8-4. LLE step 1: linearly modeling local relationships*

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \| \mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)} \|^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$

局部线性嵌入 LLE

- 这一步完成后，权重矩阵**W**对训练实例之间的局部线性关系进行编码。现在，第二步就是要将训练实例映射到一个d维空间（d＜n），同时尽可能保留这些局部关系。如果$\mathbf{z}^{(i)}$是实例$\mathbf{x}^{(i)}$在这个d维空间的映像，那么我们希望从$\mathbf{z}^{(i)}$到$\sum_{j=1}^{m} \hat{w}_{i,j}\mathbf{z}^{(j)}$之间的平方距离尽可能小。这个想法产生了如公式8-5描述的一个无约束优化问题。

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

$$\hat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left\| \mathbf{z}^{(i)} - \sum_{j=1}^{m} \hat{w}_{i,j}\mathbf{z}^{(j)} \right\|^2$$

其他降维技巧

- 多维缩放（MDS）算法，保持实例间距离，降低维度。
- 等度量映射（Isomap）算法，将每个实例与其最近的邻居连接起来，创建连接图形，然后保留实例之间的这个测地距离，降低维度。
- t-分布随机近邻嵌入（t-SNE）算法在降低维度时，试图让相似的实例彼此靠近，不相似的实例彼此远离。它主要用于可视化，尤其是将高维空间中的实例集群可视化（例如，对MNIST图像进行二维可视化）。
- 线性判别（LDA）实际上是一种分类算法，但是在训练过程中，它会学习类别之间最有区别的轴，而这个轴正好可以用来定义投影数据的超平面。这样做的好处在于投影上的类别之间会尽可能的分开。

人工神经网络简介

- 人工神经网络是深度学习方法的核心。它们通用、强大、可扩展，使得它成为解决大型和高度复杂的机器学习任务的理想选择。比如将数以亿计的图片分类（如Google Images），支撑语音识别服务（如Apple的Siri），为数以千万计的用户每天推荐最佳视频（如YouTube），通过研究之前的数百万次的比赛并不断地和自己比赛，在围棋和星际比赛中击败世界级人类选手（DeepMind的AlphaGo和AlphaStar）。

人工神经网络简介

- 本节将通过第一个ANN架构的快速教程来介绍人工神经网络。然后会展示多层感知器（MLP）并用TensorFlow来实现一个MLP，并用其来解决MNIST数字分类问题。

从生物神经元到人工神经元

- ANN已经存在了好长时间了：在1943年由神经学家Warren McCulloch和数学家Walter Pitts提出。
- 在他们著名的论文“A Logical Calculus of Ideas Immanent in Nervous Activity”中，McCulloch和Pitts展示了一个简化过的计算模型来描述，在动物的大脑中神经元如何通过命题逻辑来实现复杂的计算。这是第一个人工神经网络架构。

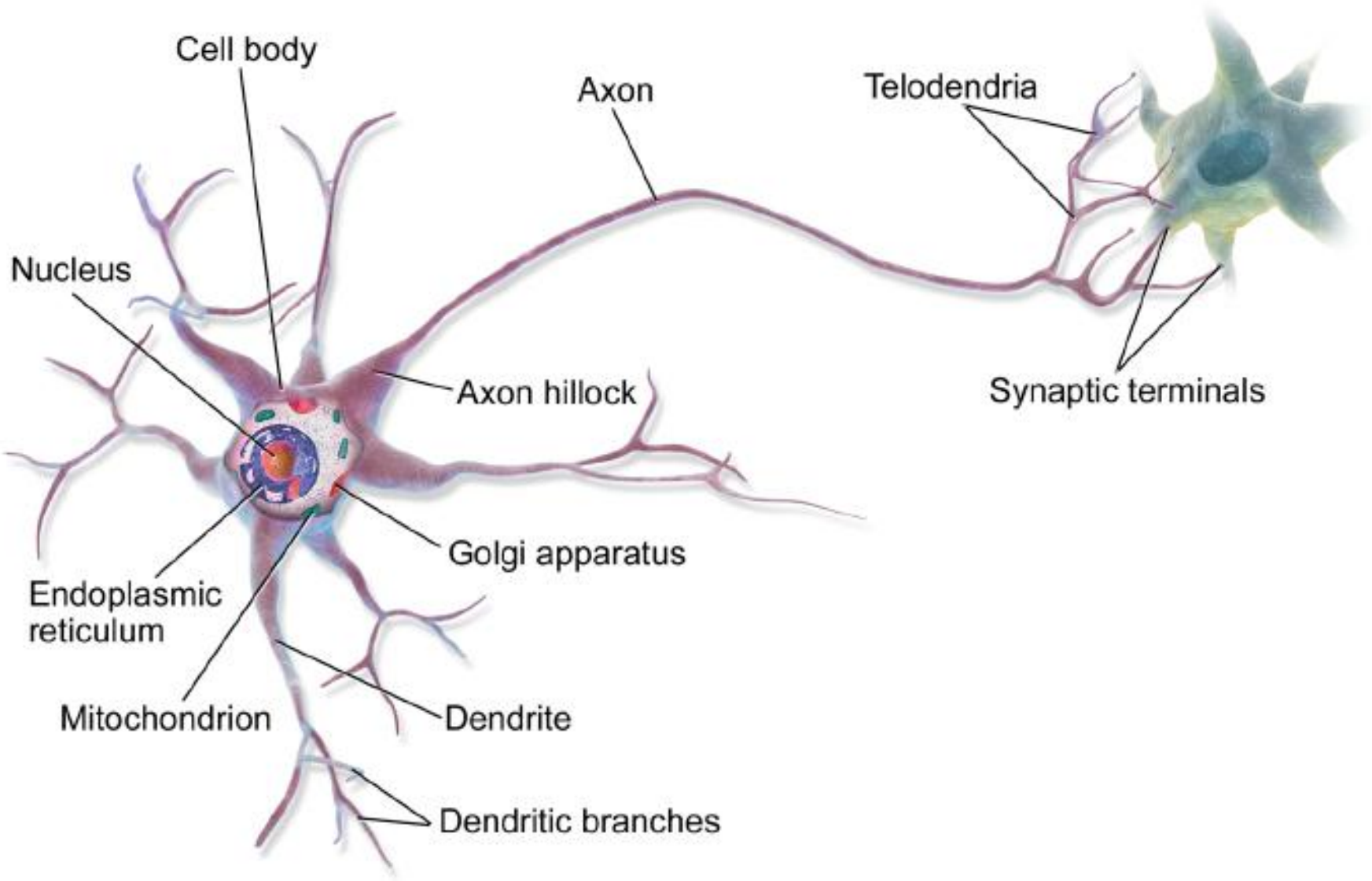
从生物神经元到人工神经元

- 直到20世纪60年代，ANN的早期成功让人们普遍认为，我们很快将会与真正智能的机器对话。当明确表示这一承诺（至少在一段时间内）将不会实现时，资金就投向了其他地方，ANN进入了漫长的黑暗时期。在20世纪80年代初，随着新网络架构的发明和更好的培训技术的发展，人们对ANN的兴趣又重新变得浓厚。不过到了20世纪90年代，更强大的机器学习技术如支持向量机（见第5章）成为大部分研究者的新宠，因为它们似乎提供了更好的结果和更强大的理论基础。

从生物神经元到人工神经元

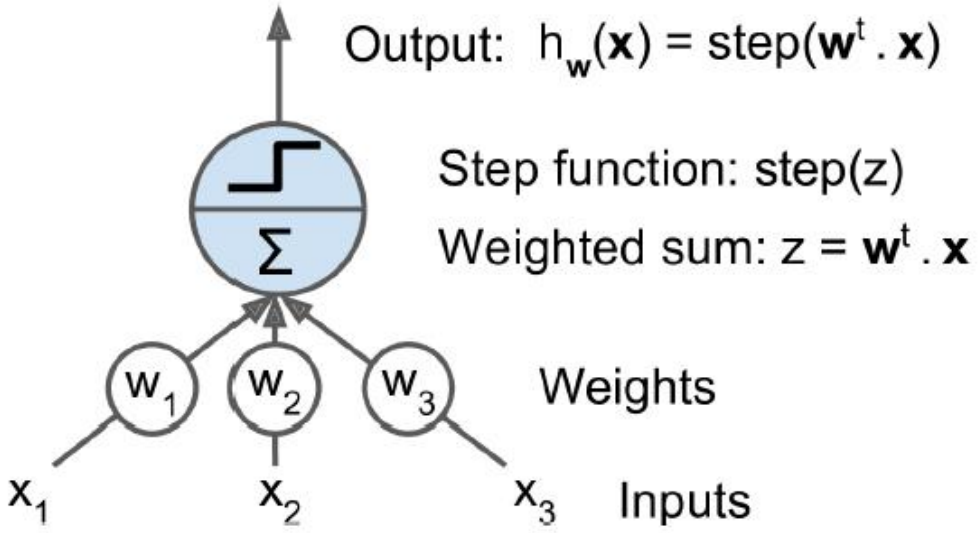
- 最终，我们见证了另一波对ANN兴趣的高潮。这次高潮会像上一次那样归于沉寂吗？有很多的原因可以相信这一次会不同，而且会给我们的生活带来很多的影响：
  - 现在有了海量的可用数据来训练神经网络，而且在超大超复杂问题上ANN比其他的ML技术性能更佳。
  - 自20世纪90年代以来，飞速增长的计算能力使得在合理时间内训练大型神经网络成为可能。部分原因是摩尔定律在生效，也要感谢游戏产业，它们制造了数以百万计的强大的GPU。
  - 训练算法也得到了很大的提升。坦白说与20世纪90年代相比，算法只有一点点不同，但是这些小的调整产生了巨大的影响。

生物神经元


Cell body
Axon
Telodendria
Nucleus
Axon hillock
Synaptic terminals
Golgi apparatus
Endoplasmic reticulum
Mitochondrion
Dendrite
Dendritic branches

感知器

- 感知器是最简单的ANN架构之一，于1957年由Frank Rosenblatt发明。它基于一个稍微不同的被称为线性阈值单元（LTU）的人工神经元：输入和输出都是数字（而不是二进制的开关状态），每个输入的连接都有一个对应的权重。LTU会加权求和所有的输入($z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = w^T \cdot x$)，然后对求值结果应用一个阶跃函数（step funciton）并产生最后的输出： $h_w(x) = step(z) = step(w^T \cdot x)$.

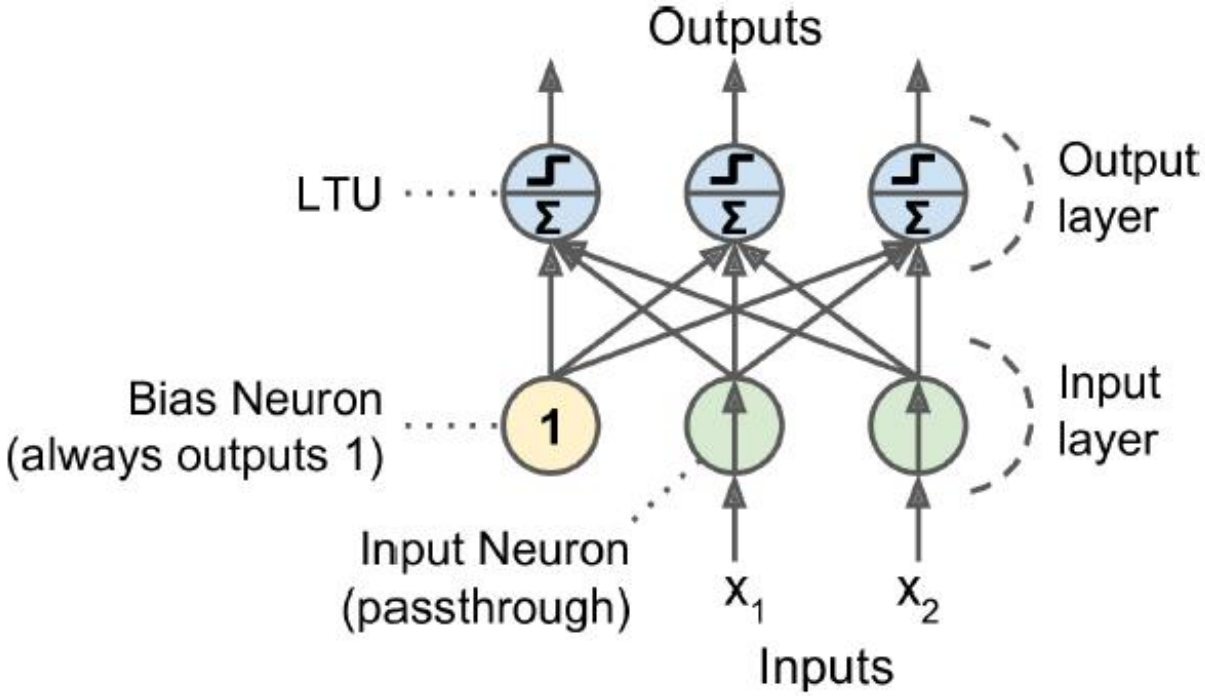


感知器

- 单个LTU可以用来做简单的线性二值分类。它计算输入的线性组合，如果结果超出了阈值，输出就是正否则为负（与逻辑回归分类器或者线性支持向量机一样）。举个例子，你可以用一个LTU来根据花瓣的长度和宽度分类鸢尾花。训练LTU的意思是寻找合适的w值（w_0, w_1, w_2）

感知器

- 感知器就是个单层的LTU，每个神经元都与所有输入相连。这些连接通常使用称为输入神经元的特殊传递神经元来表示：输入什么就输出什么。此外，还会加上一个额外的偏差特征（$x_0 = 1$）。偏差特征通常用偏差神经元来表示，它永远都只输出1。



感知器

- 感知器一次供给一个训练实例，并且对于每个实例它都会进行预测。对于产生错误预测的每个输出神经元，它加强了来自输入的连接权重，这将对正确的预测做出贡献。规则见公式10-2。

Equation 10-2. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\hat{y}_j - y_j\right)x_i$$

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

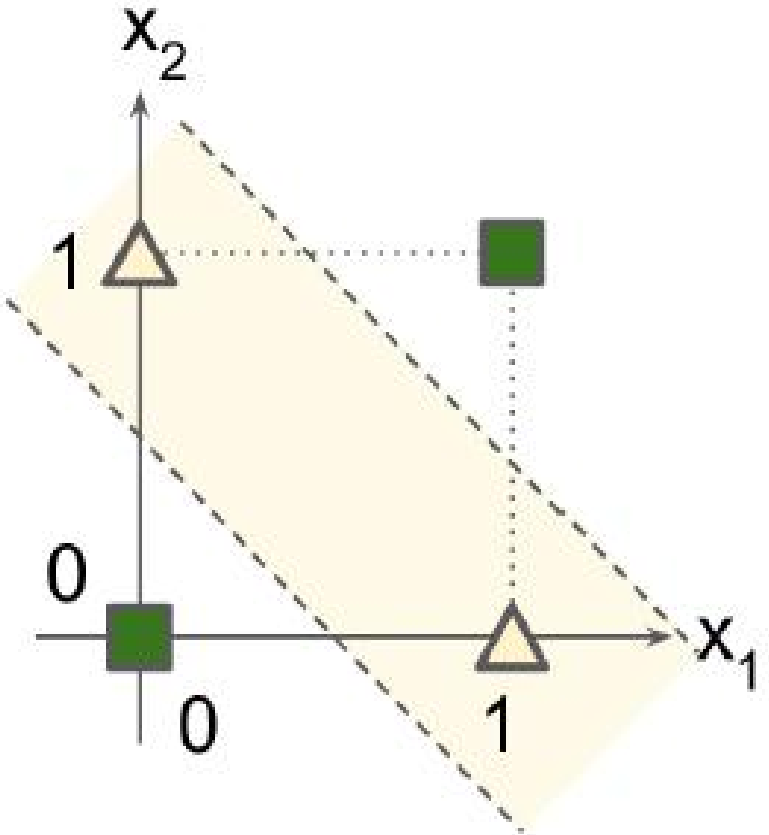
感知器

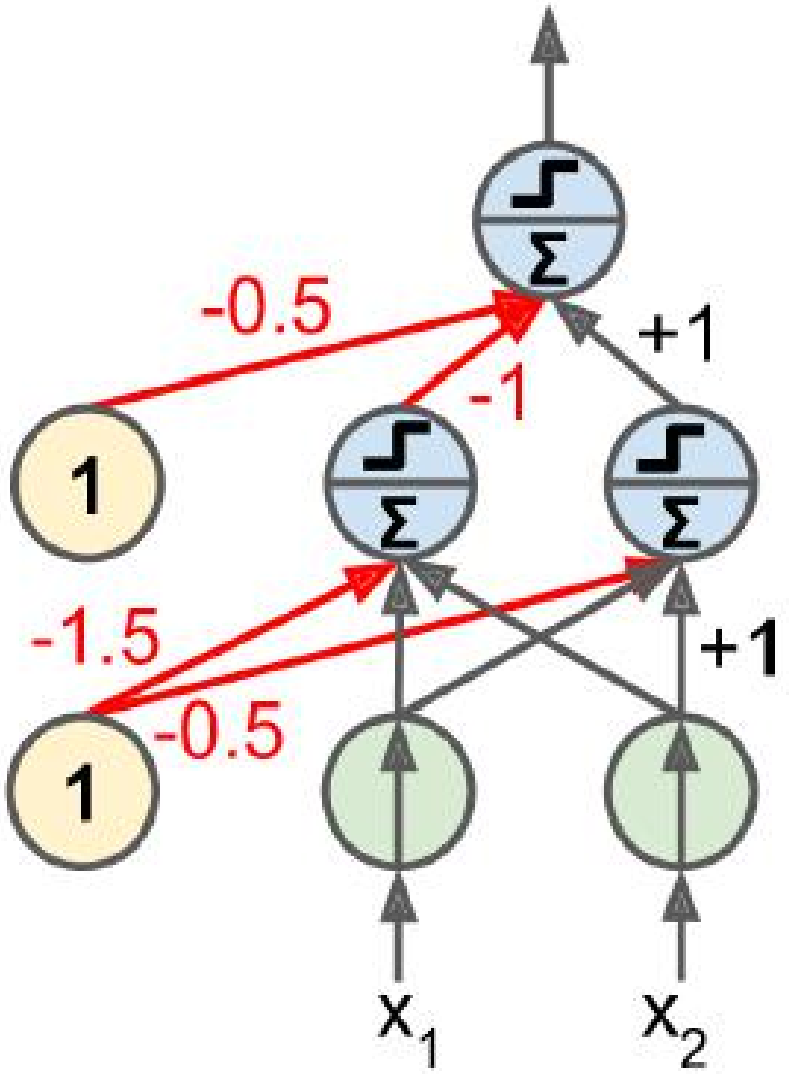
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```

感知器

- 在1969年的名为《感知器》的专著中，Marvin Minsky和Seymour Papert强调了感知器的一系列缺点，特别是它无法处理的一些很微小的问题，比如异或分类问题（XOR），见图10-6左侧。当然这个问题在其他任何的线性分类模型中一样存在，只不过研究者对感知器的期望太高，因此失望也更大：结果就是，很多研究者完全放弃连接机制（connectionism，即神经网络的研究），而倾向于更高层次的问题，如逻辑、问题解决和搜索。
- 事实证明感知器的一些限制可以通过将多个感知器堆叠起来的方式来消除，这种形式的ANN就是多层感知器（Multi-Layer Perceptron）。

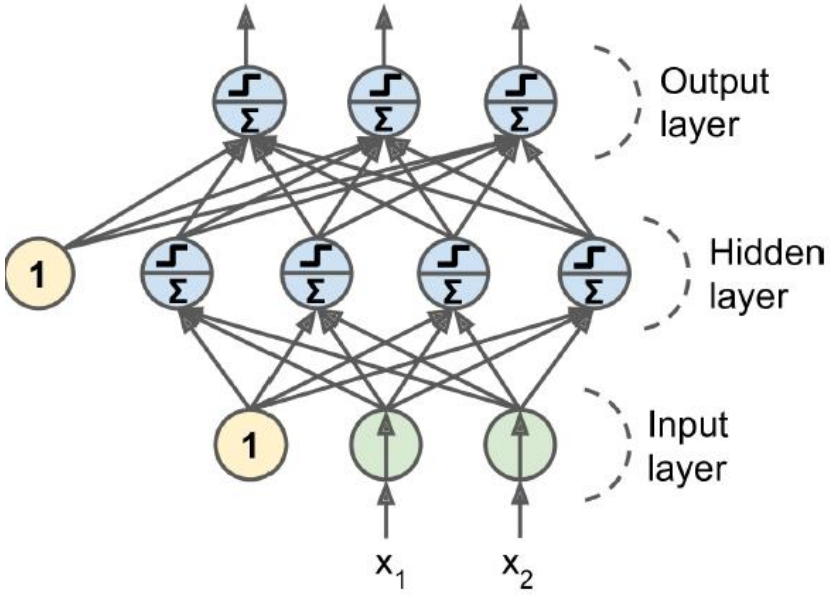
感知器





多层感知器和反向传播

- 一个MLP包含一个输入层，一个或者多个被称为隐藏层的LTU层，以及一个被称为输出层的LTU组成的最终层。除了输出层之外，每层都包含了一个偏移神经元，并且与下一层完全相连。如果一个ANN有2个以及2个以上的隐藏层，则被称为深度神经网络（DNN）。



多层感知器和反向传播

- 多年来，研究者都为如何训练MLP而头疼不已，一直没有进展。直到1986年，D.E.Rumelhart发表了一篇介绍反向传播训练算法。
- 对于每个训练实例，反向传播算法先做一次预测（正向过程），度量误差，然后反向的遍历每个层次来度量每个连接的误差贡献度（反向过程），最后再微调每个连接的权重来降低误差（梯度下降）。

多层感知器和反向传播

- 为了让这个算法正常工作，作者对MLP架构做了一个关键的调整：把阶跃函数改成了逻辑函数：$\sigma(z) = 1 / (1 + \exp(-z))$。这是非常关键的一步，因为阶跃函数只包含平面，所以没有梯度（梯度下降在平面上无法移动），但是逻辑函数则有着定义良好的偏导，梯度下降可以在每一步都做调整。

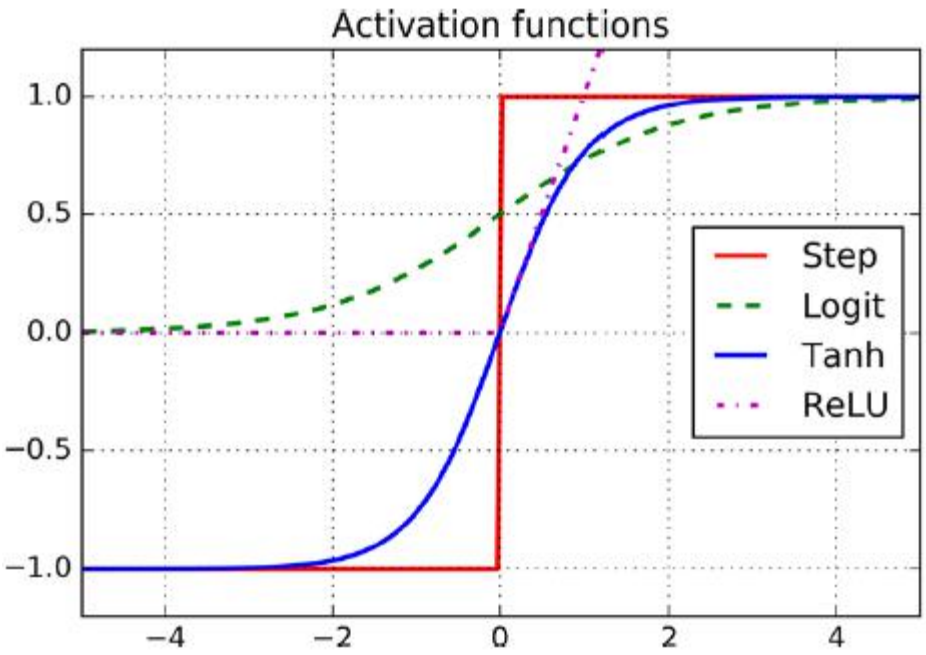
多层感知器和反向传播

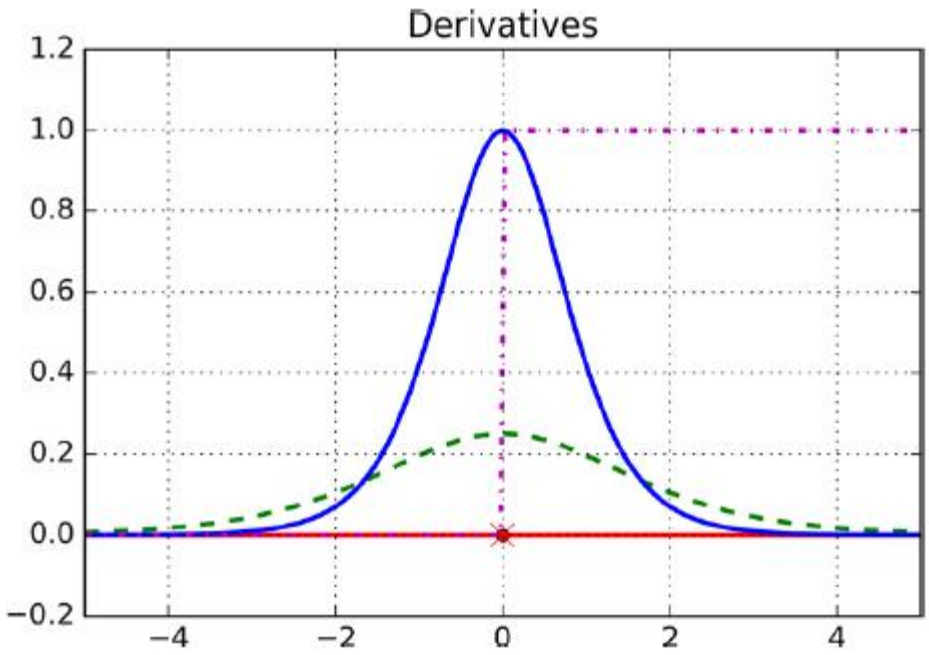
另外两个最流行的激活函数是：

双曲正切函数 $\tanh(z) = 2\sigma(2z) - 1$　与逻辑函数类似，它是一个S形曲线，连续且可微分，不过它的输出是-1到1之间的值（逻辑是0到1之间的值），这会让每层的输出在训练开始时或多或少地标准化（以0为中心）。这通常有助于快速收敛。

ReLU函数 $\text{ReLU}(z) = \max(0, z)$ 这个函数也是连续的，不过在z=0时不可微分。不过实践中它工作良好，而且计算速度很快。最重要的是，它对于消除梯度下降的一些问题很有帮助。

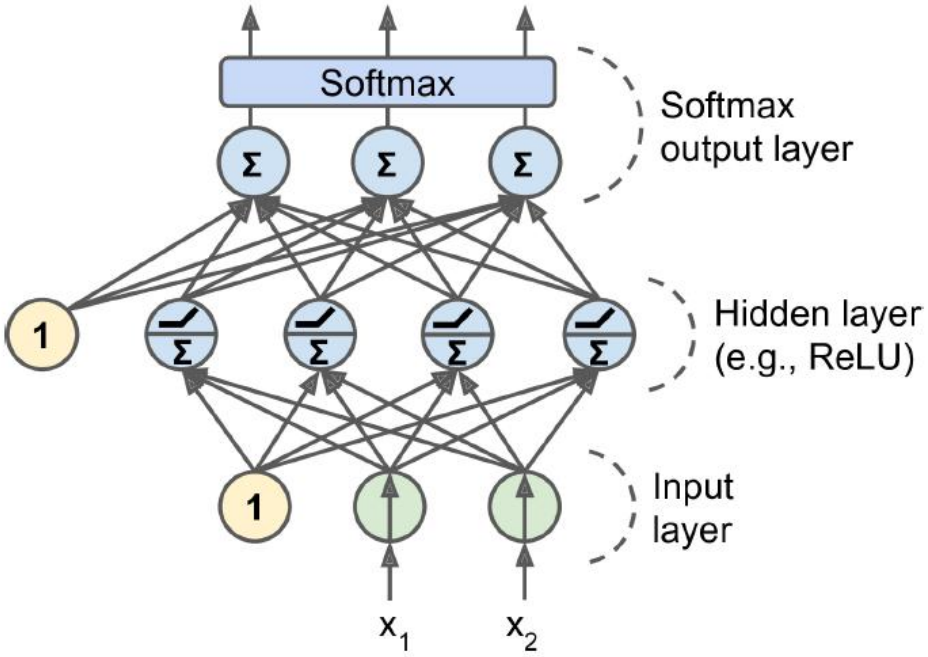
激活函数和它们的导数


Activation functions
1.0
0.5
0.0
−0.5
−1.0
−4
−2
0
2
4
Step
Logit
Tanh
ReLU


Derivatives
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−4
−2
0
2
4

多层感知器和反向传播

- MLP常常被用来做分类，每个输出对应一个不同的二进制分类（比如，垃圾邮件/正常邮件、紧急/非紧急，等等）。当每个分类是互斥的情况下（比如将图片分类为数字0～9的场景），输出层通常被修改成一个共享的soft-max函数。softmax函数可以输出对应于相应分类的估计概率。



使用纯TensorFlow训练DNN

- 在本节我们会用TensorFlow底层API构建一个模型，实现一个小批次梯度下降算法来训练MNIST数据集。
- 首先是构建阶段，建立TensorFlow的计算图，第二步是执行阶段，具体运行这个图来训练模型。

构建阶段

```
import tensorflow as tf

n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

构建阶段

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation=="relu":
            return tf.nn.relu(z)
        else:
            return z
```

构建阶段

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

```
from tensorflow.contrib.layers import fully_connected
with tf.name_scope("dnn"):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs", activation_fn=None)
```

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

learning_rate = 0.01
with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

执行阶段

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
n_epochs = 400
batch_size = 50
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
save_path = saver.save(sess, "./my_model_final.ckpt")
```

使用神经网络

- 现在神经网络已经被训练好了，可以用它来做预测了。保留构建器的代码，修改执行期的代码如下所示：

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = [...] # some new images (scaled from 0 to 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

微调神经网络的超参数

- 神经网络的灵活性也恰好是它的一个主要的短板：有太多的超参数需要调整。不仅仅是可以使用任何的网络拓扑（神经元是如何彼此连接的），即使是简单的MLP，也有很多可以调整的参数：你可以修改层数，每层的神经元数，每层用的激活函数类型，初始化逻辑的权重，等等。怎么才能知道超参数的何种组合适合你呢？

隐藏层的个数

- 对很多问题，你可以从单一的隐藏层开始，而且通常可以获得很好的效果。事实上人们发现只要神经元足够多，仅有一个隐藏层的MLP都可以建模大部分复杂的函数。很长一段时间里，研究者们都认为无须进一步研究更深的神经网络。不过他们忽视了深层网络比浅层网络有更高的参数效率：深层网络可以用非常少的神经元来建模复杂函数，因此训练起来更加快速。

隐藏层的个数

- 对于大多数问题来说，你都只需要一个或者两个隐藏层来处理（对于MINST数据集，一个拥有数百个神经元的隐藏层就可以达到97%的精度，而用同样数量神经元构建的两层隐藏层就可以获得超过98%的精度，而且训练时间基本相同）。对于更复杂的问题，你可以逐渐增减隐藏层的层次，直到在训练集上产生过度拟合。非常复杂的问题，比如大图片的分类，或者语音识别，通常需要数十层的隐藏层（甚至数百层，非全连接的层），当然它们需要超大的训练数据集。

每个隐藏层中的神经元数

- 显然，输入输出层中的神经元数由任务要求的输入输出类型决定。比如，MNIST任务需要28×28=784个输入神经元和10个输出神经元。对于隐藏层来说，一个常用的实践是以漏斗型来定义其尺寸，每层的神经元数依次减少：原因是许多低级功能可以合并成数量更少的高级功能。比如，一个典型的MINST的神经网络有两个隐藏层，第一层有300个神经元，而第二层有100个神经元。
- 一个更简单的做法是使用（比实际所需）更多的层次和神经元，然后提前结束训练（early stopping）来避免过度拟合（以及其他的正则化技术，特别是dropout）。

激活函数

- 大多数情况下，你可以在隐藏层中使用ReLU激活函数。它比其他激活函数要快一些，因为梯度下降对于大输入值没有上限，会让梯度信号始终有效（与逻辑函数或者双曲正切函数刚好相反，它们会在 1 处饱和）
- 对于输出层，softmax激活函数对于分类任务（如果分类是互斥的）来说是一个很不错的选择。对于回归任务，完全可以不使用激活函数。