老师好，大家好，我今天介绍的内容是神经网络。神经网络是机器学习中非常重要的一类模型，也是深度学习的基础。掌握神经网络能够让我们在使用深度学习模型和深度学习框架时更加从容。

我今天主要是从以下6个方面来介绍神经网络，首先是神经元模型，神经元是神经网络中最基本的单元，是构建复杂模型的基础；然后是感知机模型，感知机可以看作最基础的神经网络，通过感知机模型可以了解神经网络的工作原理；接下来是多层网络，这里多层网络主要指的是多层前馈神经网络；然后是误差逆传播算法，该算法是神经网络的核心算法，其算法过程就是我们经常说的训练或者学习，使用该算法的神经网络称为BP神经网络；接下来会简单介绍西瓜书中的部分其他神经网络；最后简单说一下pytorch中的一些机制，方便我们更好地应对使用框架遇到的问题。

本次讲解的内容主要是西瓜书，并补充了一些网络上的资料。由于神经网络这些年发展得很快，有些说法当下来看可能就不是很合适了，我本身的理解也不一定完全准确，所以可能会有部分内容讲解的不到位，有问题的话欢迎提出和指正。

人工神经网络在一定程度上是受到生物神经网络启发而发展的，是一种模拟生物神经网络的数学结构。生物神经网络最基本的单元是神经元，神经元模型虽然简单，但是大量的神经元互相连接构成的生物神经网络会相当复杂，能够处理复杂的问题，有非常强的拟合能力。

具体到单独的一个神经元，它是由树突、细胞体、轴突组成。

1. 多个树突负责接收多个其他神经元的输入
2. 细胞体负责整合接收到的多个输入，并生成唯一的输出
3. 轴突负责将输出传递给其他神经元

生物神经元的典型特征是多输入，单输出，这个特点使用计算机非常容易模拟，我们平时写的大部分函数都是这样的。通常只有在神经元接收的信号强度高于一个阈值时，它才会激活，才会向相连的神经元传递信息。

ↆ当输入小于阈值时，生物神经元会抑制输出，表现为没有输出或输出为0，ↆ如动画中的绿色小球一样，数据的传导过程会被截断，信息无法传递到下一个神经元；ↆ当输入大于阈值时，生物神经元会兴奋，会正常输出数据，ↆ直观上来看如同图的小球一样，信息能够正常传递。

需要说明的是，人工神经网络只是受生物神经网络启发，真实的生物神经网络远比人工神经网络复杂，在计算机领域中谈论神经网络更多的还是要从数学的角度出发，将神经网络看作一个包含了许多参数的数学模型。

将神经元抽象成数学模型，其输入是前一层神经元的输出，考虑到不同输入的重要程度可能不同，所以实际上并不是直接将所有输入加起来，而是对多个输入求取加权和，并通过一个偏置参数进行调整。最后这个偏置可以看作是一个权重恒为-1的输入，表达起来就是-1\*seta。然后我们对求和结果应用激活函数得到输出，激活函数在神经网络中非常重要，好的激活函数能够提高模型的非线性拟合能力、加快训练过程、提高模型的性能。ↆ对于神经元模型来说，一个自然而然的问题是这里的权值和偏置参数到底取多少合适，获得较为合理的参数值是神经网络算法的核心问题是调整模型中的权重等参数来使得模型的输出符合预期的过程就是我们常说的训练。

ↆ最容易想到的确定权值的方式就是暴力破解，但实际上是几乎不可能实现的。ↆ以VGG16为例，它是一个并不算特别深的用于分类的卷积神经网络，其参数量大约为1.3亿个，而且这些参数都是连续的浮点数，而不是离散的取值。可以说，就目前的计算机来说，ↆ暴力破解几乎是不可能的实现的。

ↆ实际上我们是在原有权重的基础上进行更新来调整参数的，最开始的权重可以是随机的，一般会保证这些随机的初始参数符合一定的分布，如高斯分布。然后我们根据一些方法反复地在原有参数的基础上通过加减一个量来更新参数，如公式所示，学习率实际上是一个缩放因子，对更新的幅度做一个缩放。而后面这个量是根据一些规则计算出来的，在梯度下降算法中被称为梯度。

接下来详细地说明激活函数的问题，理想的激活函数是阶跃函数，输出0表示神经元抑制，输出1表示神经元兴奋。考虑到阶跃函数具有不连续、不光滑等不好的性质，可以使用sigmoid函数作为替代。sigmoid函数既能够将数据挤压到0到1的范围内，又处处可导。这里令f(x)是sigmoid函数，其导数是f(x)乘上1-f(x)，求导的过程这里不再给出，这个导函数公式之后还会用到。

仔细观察sigmoid激活函数，会发现其的一些不足之处，实际上现在已经不怎么使用了sigmoid作为激活函数了。sigmoid的缺点主要是在3个方面：

1. 梯度消失：当输出接近于0或1的时候sigmoid函数会变得相当平坦，或者说导数趋近于0。ↆ我们刚才那个更新公式中梯度可以直接理解为一个导数，我今天所说的导数和梯度是同一概念。最终用于更新权值的梯度实际上就是多个运算的梯度的乘积，而激活函数的梯度就是其中一项。当sigmoid的梯度趋近于0，最终的梯度也是趋近于0的。因此，该神经元的权重几乎不会得到更新。更重要的是，这个梯度还会传递给其他神经元，当神经网络中有大量这种神经元时，网络难以有效地进行更新。
2. 不以0为中心，由于sigmoid的输出是恒大于0的，这很可能导致后层神经元接收的输入不以0为中心的。由于输入为0时，sigmoid的梯度最大，那么不以0为中心的输入会使权重更新效率变低。这个问题可以通过将结果减去0.5来解决。实际上，现在的深度学习模型中常常使用BN等归一化层，其能够将输入数据拉到标准正态分布，这对于提升模型的性能是很有帮助的。
3. 计算成本高：因为sigmoid中有一个求指数的过程，其相对而言计算成本是比较高的。

除了sigmoid，目前还存在大量的激活函数，这里介绍目前常用的一些激活函数：

1. 首先是tanh，相对于sigmoid函数，他是0中心化的，不过它也有梯度消失和计算成本高的问题，目前实际上基本上不使用。
2. 然后是热炉激活函数，他是目前使用最多的激活函数，当x大于0时，其输出就等于输入，当x小于0时，其输出为0。首先，由于relu很简单，所以它在计算上非常高效；其次，relu具有单侧抑制，宽兴奋边界的性质，这在生物学上更加合理；最后，相对于sigmoid函数，relu在一定程度上缓解了梯度消失的问题。但是relu有一个死亡relu问题，指的是当学习率过大或者输入值异常时，可能导致相关的权值更新后变成一个非常大的负值。那么，在以后前向计算的过程中，由于前一层的输出被relu映射为非负值，那么乘上这个非常大的负权值有可能导致加权和一直是负的。进而使得以后每次更新的时候该神经元的梯度一直为0，即这个神经元永远也不会更新了。作为解决方案，可以减小学习率或者使用leakeyrelu或者glu等激活函数，他们在输入为负的时仍有一定的梯度。 此外，目前在深度学习中经常使用bn来对输入进行归一化处理，这一方面能够避免异常输入的影响；另一方面，即使当前神经元有个较大的负权值，但数据输入relu之前会被归一化，也就是说输入relu的数据不会一直是正的，从而避免了神经元的永久性死亡。
3. 最后是swish，他的计算公式是x乘上sigmoid（阿法x），阿法是一个调节因子。相对relu它更加平滑，输入小于0的时候也有梯度。不过实测性能也并不是普遍优于relu。

除了刚才介绍的激活函数，还有许多其他的激活函数，这些激活函数一般都有一些共性：

* 激活函数需要是连续可导的非线性函数（允许少数点不可导），这是因为误差反向传播求梯度的时候是要求导的
* 导函数尽可能简单：简单很多时候意味着较高的性能
* 值域要在合理区间：不能太大，也不能太小，否则会影响训练的效率和稳定性。现在通过在网络中应用bn等归一化层，能够在一定层度上保证值域的分布

接下来介绍是感知机，感知机由两层神经元组成，其结构如图所示，第一层神经元仅作输入用。感知机是一个二分类模型，其先是对输入求加权和，然后通过激活函数映射为0类或者1类，比如当加权和的结果大于阈值时，感知机输出1类，否则感知机输出0类。对于感知机来说，每次更新时的增量由学习率、真实值和输出值的差值、连接到权值的输入分量构成。由公式可以看出，只有当预测值与真实值不同时，感知机的参数才会被更新。感知机能够区分线性可分的两类对象，比如说图中的红色和绿色两种虫子。这两类虫子具有不同的长度和宽度特征，经过训练，感知机能够找到一条合适的分割线来对这两类虫子进行分类。这里我们假定最后输出时的阈值为0，或者说激活函数是标准阶跃函数。

具体到感知机的更新过程。首先，由感知机的权重参数w1和w2可以构成一个分界线，ↆ分界线的方程是w1x1+w2x2=0，ↆ当w2大于0时，直线上方的点会被划分为类别1；当w2小于0时，直线上方的点会被预测为0类。ↆ以图中的虚线为例，对于当前的横坐标x1’，纵坐标为绿色虚线的坐标值x2’时，ↆ按照分界线的定义w1x1’+w2x2’正好为0。而位于分界线上方的虫子的纵坐标是红色虚线对应的坐标值，ↆ此时计算该虫子的长度和宽度的加权和是在之前的基础上加上w2\*deltax2，由于w2大于零，最终的结果一定是大于0的，那么在激活函数的作用下会被预测为1类。当w2小于0时，也是相同的逻辑。ↆ

接下来我们为这两类虫子分配类别，到底哪类虫子是1类或0类都无所谓，ↆ只要把这些虫子划分成两类即可。这里将右下角的绿色虫子划分为1类。然后是当前的权重，对于感知机来说，最开始的权值一般可以设为0，不过在一般的神经网络中不会这样干。这里我们假设当前权重确定的分割线如图所示，那么w1，w2必定是一正一负，这样分割线才能够穿过1，3象限。实际上最终w2一定是负的，这样才能够保证直线上方的是0类，不过这里先假设w2当前是正的，这样能够更好地说明问题。ↆ当w2为正数时，按照刚才说的，图中所有虫子都在分割线上方，所以所有虫子都被预测为类别1，ↆ那么所有的红色虫子全部预测错误。ↆ按照感知机更新算法，各权值的更新公式如下，这会使得分割线顺时针旋转，直到w2小于0。需要注意到，分割线不是直接逆时针转动到两类之间，否则所有的红色虫子被预测为1类，绿色虫子被预测为0类，就全错了。当w2小于0时，直线上方的点被预测为0类，那么所有绿色的虫子预测错误，此时的更新公式如下ↆ。此时分割线会逆时针旋转，需要注意到，在刚才的更新过程中，ↆ每次减小权值时，错误类别是红色虫子，那么x1小于x2，每次增加权值时，错误的是绿色虫子，那么x1大于x2，这样反复迭代几次，最终一定是分割线穿过1，3象限，且w2小于0。

我们假定此时分割线的斜率较小，和最开始差不多，ↆ现在直线上方的所有点被预测为0类，分类错误的变成了绿色虫子。此时，感知机更新公式是这样的ↆ，由于w1大于0，那么加法之后其绝对值会变大ↆ；而w2由于小于0，其加上一个正数后绝对值会变小ↆ，（这里不考虑w2加上一个非常大的值之后变成一个很大的正数的情况，因为即使这样在以后的更新中也会修正回来）。刚才的更新过程使得分割线的斜率变大ↆ，表现为分割线逆时针转动ↆ。

如果分割线斜率过大，其又会顺时针旋转ↆ，这个旋转角度一定是低于刚才的角度，这里就不展开证明了。最终，分割线一定能够完美的将两类虫子分开。

从刚才的例子可以看出，感知机能够处理线性可分问题，所以它能够很好地实现逻辑上的与或非运算，因为这些运算都是线性可分的。以逻辑与为例，只有当两个输入都为1时，其结果才是1，那么这个分界线很容易找到。然而，对于异或问题，无论如何调整感知机，永远无法进行分割。解决方案也很简单，那就是多加一层神经元，比如中间这个图展示的网络结构和各层权值，其对应的分割区域是如右图所示的，完美的解决了异或问题。

为了能够处理更加复杂的任务，我们需要求助于更深的网络。经典的多层神经网络被称为前馈神经网络，每层神经元与下一层神经元完全相连，神经元之间不存在同层连接也不存在跨层连接。前馈的含义是输入层接收外界输入，隐藏层和输出层对信号进行加工，最终结果由输出层神经元输出。学习指的是根据训练数据来调整神经元之间的参数。前面也提到过，使用暴力破解是不可能的，哪怕是这个简单的模型的权重参数也有2万多个。

我们刚才介绍了神经元模型、激活函数、感知机和多层网络，这里有一个网站可以更直观看到它们的影响。这个网站左边可以选择一些二分类的任务，然后中间可以调整神经网络的深度和每层神经元的个数，上面还可以选择激活函数。输入是样本点的横纵坐标，也可以以横纵坐标的函数作为输入，比如x1的平方。首先构造一个感知机模型，可以看到，对于线性可分问题{两团}，其能够很快收敛，但是对于非线性问题{交叉}感知机模型就无能为力了。接下来，我们尝试增加神经元个数和层数{2442}。可以看到，当前模型能够很好的处理这种非线性问题。

然后我们可以试一下sigmoid函数，这里的模型压根不会收敛。换上relu试试，模型会收敛得相当快，效果也是很好的。这个网站还有一些别的有趣的地方，感兴趣的可以自己试试。

在介绍完网络的结构以及如何由输入计算输出后，接下来介绍的是参数的更新过程。为了使用误差逆传播算法训练模型，首先需要计算误差，误差的作用是量化模型预测值与真实值的不一致程度。我们使用损失函数接收模型的预测值和真实值，然后计算出误差损失。目前有各种各样的损失函数，这些损失函数适用的任务也不尽相同，比如说交叉熵非常适合于衡量分类损失。

今天我只介绍一些最基础的损失函数，首先是平均绝对误差，又称L1范数损失，实际上就是求预测值与真实值的平均绝对差值，其缺点在于不平滑。然后是平均平方误差，又称L2范数损失，它是预测值与真实值的平均平方差，其处处光滑。但是当输入偏离0较远时，l2损失可能会导致梯度爆破炸。梯度爆炸指的是当网络较深时，伴随梯度的不断累积，可能导致非常大的梯度，使得权重大幅更新，网络变得不稳定。Smooth L1损失综合了前面两种损失，当x的绝对值小于1时，使用l2范数损失；当x的绝对值大于1时，使用l1范数损失。在回归问题中，l2和smooth l1使用的都挺多的。

接下来会以一个实际的例子说明多层前馈神经网络的参数到底是如何更新的，我们采用的模型是3层，每层都是两个节点，最后的红色这个真实值，这里的损失函数采用均方误差。隐藏层输入是输入层和权重矩阵的乘积，输出层来自隐藏层和另一个权重矩阵的乘积。最后，隐藏层和输出层的神经元都对其数据应运激活函数，这里激活函数使用的是sigmoid函数。

首先，我们最终需要求的是误差对每个待更新权重的梯度，也就是导数。我们假定已经求得了当前的梯度，ↆ那么如左下的图所示，我们只知道当前的状态，也就是当前的权值、误差、梯度，别的什么也不知道。ↆ最直观的做法就是将w左移，也就是将w向着梯度反方向移动，对应的公式是这个ↆ，移动的距离由学习率和梯度共同决定。这就是我们实际上更新权值的方式，又称梯度下降法。

剩下的问题就是求梯度了，我们实际上是借助了求导的链式法则来计算梯度的ↆ，我们先求误差对第二层权值的梯度，这个任务可以分解为求误差对输出的导数，输出对最后一层神经元输入的导数和输入对连接的权值的导数。ↆ损失函数采用了均方误差，那么误差对输出的导数就是O-T。这里直接以向量和矩阵的方式来表示更加方便。然后是输出层输出对输出层输入的导数ↆ，实际上就是对激活函数求导，sigmoid激活函数的导数前面说过，那么这里导数就是O\*(1-O)ↆ。最后是输出层输入对权值求导，这个就只是线性的加权和，所以求导结果就是隐藏层的值。到此ↆ，我们将刚才算的三个值相乘就得到了误差对第二层权值的梯度。

接下来，我们来求误差对第一层权重的梯度ↆ，它可以写成误差对隐藏层输出的梯度乘上隐藏层输出对权值的梯度，ↆ前者的运算和刚才过程没什么区别，不过是最后这个求导项变了一下而已。后者运算也挺简单ↆ，实际上就是对激活函数的求导乘上对加权和的求导而已。到此，我们求得了误差对第一层权重的梯度，在求这个最终梯度的时候，很多中间梯度我们之前在求对第二层权重的梯度的时候就计算过，整理一下可以写成这样ↆ。这也就是为什么我说误差反向传播实际上是梯度的反向传播。当然，一些资料上确实有人将误差一层层的逆传播到各层，然后每层误差对其相应的权值求梯度，结果应该是一样的，我不过觉得其解释上不是很严谨性而且也不够灵活，因为随着深度学习的发展，实际上的模型并不是这种一层层的全连接网络。

刚才其实我们已经详细的说明了如何根据梯度来更新权值，下面更加全面的对梯度下降法进行介绍。函数某点的负梯度方向代表其下降最快的方向，也就是说在负梯度方向上移动参数可以减小函数值。比方说如右上角这幅图。按照梯度下降法，我们每次都是向着减小误差的方向更新权值，就类似于下山一样，一直往下走，总是能够走到比较低的地方。另一方面，同样类似于下山，有时候我们为了走到山脚，不得不在某些地方向上走，避免陷入一个山谷中。然而，梯度下降法不允许向上走，这导致了局部极小的问题，如左下角的图所示，当陷入局部极小时，按照梯度下降法，误差永远无法达到全局最小，因为从局部极小到全局最小的过程中必然经历着误差变大。针对这个问题，西瓜书上提到的解决方案主要有以下4种：

1. 以多组不同数值初始化多个神经网络，选择训练后误差最小的解。相当于尝试多个不同的出发点
2. 使用模拟退火等技术，以一定的概率接受比当前解更差的结果
3. 使用随机梯度下降，由于其不稳定性，有机会跳出局部极小（这个后面会单独说的）
4. 使用遗传算法训练网络逼近全局最小

实际上还有一种措施能够缓解局部极小值问题，如右下角所示，我们可以记录权值之前的变化情况，那么将之前的移动速度（或者说动量）作为更新权重的考量，就可以在一定程度上冲过局部极小值的地方，我们在深度学习框架中使用的adam和momentum就采用了该机制。右下角的图中红色小球冲过了左边的局部极小值，但是却没有冲过另一个局部极小值，这里表达的含义是动量只能缓解局部极小值问题，而无法彻底解决。至于绿色小球，由于其初始位置比较好，所以很容易地到达了最小值，这也体现从不同出发点训练网络地意义。

关于梯度下降算法，每次基于多少数据的误差之和来更新网络也是需要考虑的问题。因为我们最终的目标是使训练集所有样本的损失之和最小，所以以所有数据的损失之和来更新参数能够产生稳定的梯度。而且由于能够进行批次计算，所以该方法能充分发挥GPU的性能。但是其对内存容量的要求比较高。另一种策略是每个样例更新一次权值，这种算法被称作随机梯度下降法，这种方法收敛过程并不稳定，如图中紫色线条所示，且每个样本更新一次权重效率比较低。最后是mini-batch梯度下降法，其将数据集拆分成多个独立同分布的子集合，根据每个子集合的损失函数之和更新参数，该方法综合了前两种算法，在大规模数据集上应用的很普遍。需要说明的是，目前有的资料中将这三种梯度下降算法统称为随机梯度下降，查资料看到时要结合具体语境来理解。对于我们自身来说，需要理解的是各种算法的优点和缺点。

接下来是学习率，其主要就是用来调整权重更新的步幅。当学习率设置的过小时，收敛过程将变得十分缓慢。当学习率设置的过大时，梯度可能在最小值附近来回震荡，甚至可能无法收敛。实际中，我们一般可以在刚开始训练时设置相对较大的学习率，然后在训练过程中动态的调整学习率，比如说每30轮令学习率变为原来的十分之一。此外，根据经验并尝试不同的学习率也是一种训练措施。

在训练的时候，常常会遇到过拟合和欠拟合问题，过拟合指的是模型把训练样本自身特点当作所有潜在样本都具有的一般性质，导致模型泛化能力变差；欠拟合指的是对训练样本的一般性质尚未学好，一般增加网络复杂度、多训练几轮就能解决。由于强大的表示能力，神经网络经常遭遇过拟合，表现为训练误差持续降低，但测试误差却可能上升。一般是通过早停和正则化两种策略来缓解过拟合，早停指的是训练集误差降低但验证集误差上升时停止训练；正则化指的是在损失函数中增加一个描述网络复杂度的部分，让网络不再只以单一的误差为目标。例如，变分自编码器通过一个KL散度误差来对模型进行正则化。

接下来是其他神经网络，相对于前馈神经网络，这些神经网络用的并不多，但他们的思想非常值得借鉴。首先是径向基函数RBF网络，其一般是单隐层的前馈神经网络，并使用径向基函数作为隐层的激活函数。径向基函数通常定义为样本到数据中心的欧氏距离的单调函数，一般使用的是高斯径向基函数，其中Ci表示第i个隐层神经元，隐层的每个神经元代表一个数据中心，这个数据中心可以通过聚类获得，比如图中这几个红色的点。如图所示，神经元的输入离中心越远，神经元的激活程度就越低，所以RBF神经网络具有“局部映射”特性。RBF网络也是属于BP神经网络的一种，只不过在训练时需要先通过聚类等手段确定类别中心。

然后是自适应谐振理论ART网络，ART网络是竞争型学习的代表，竞争型学习指的是网络的输出神经元相互竞争，每一时刻仅有一个竞争获胜的神经元被激活，其他神经元的状态被抑制。竞争方式可以是计算输入向量与每个识别层神经元所对应的模式类的代表向量之间的距离，距离最小者获胜。ART由比较层，识别层、识别阈值和重置模块构成。比较层的主要作用就是将输入传递给识别层，识别层每个神经元对应于一个类别，类别数目在训练过程中动态增长。具体来说，如果竞争成功的类别和输入数据的相似度高于门限阈值，则将该数据归为该类，并更新网络权值，使得以后接收到相似样本时会计算出更大的相似度；否则，重置模块会在识别层增设一个神经元，该类的代表向量就是当前输入向量。

然后是级联相关网络，其是结构自适应网络的重要代表，结构自适应网络不仅仅学习权值、阈值等参数，将网络结构也当作学习目标之一。级联相关网络中级联指的是建立层次连接的层级结构，最开始网络只有输入层和输出层，训练过程中新的神经元不断加入。相关指的是通过最大化新神经元的输出与网络误差之间的相关性来训练相关的参数。与一般的前馈神经网络相比，级联相关网络无需设置网络层数、隐层神经元数目，训练速度较快，但在数据较小时易陷入过拟合。

最后是简单介绍一下深度学习。理论上，参数越多的模型复杂度越高，能够处理更加复杂的任务。但在一般情形下，复杂模型的训练效率低，易陷入过拟合。典型的深度学习模型就是很深的神经网络，但如果只是简单的提高隐层的数目，会导致权值等参数过多。另外，多隐层神经网络在应用标准的反向传播算法时，由于误差（梯度）往往会发散，从而使得网络难以收敛到稳定状态。一种节省训练开销的策略是“权共享”，即让一组神经元使用相同的权值。如图ↆ，在计算第一个输出时，我们只使用三个前一层的神经元，这就是局部连接。ↆ在计算第二个输出时，我们换了三个输入，但权值参数使用的还是之前的三个，这就是权值共享。ↆ其他的输出也是使用这种方式计算。除了大幅减少参数量，与全连接相比，局部连接+权值共享的网络结构能够更加关注数据的局部特征，这在处理很多问题上非常有优势。卷积神经网络的基本原理也是这个，这部分内容之后会有别的同学详细介绍。

最后一部分的内容是简单介绍一点点pytorch的知识，pytorch是主流的深度学习框架之一，可以理解为能够自动求导并运行在gpu上把的numpy。

tensor是pytorch的核心类，神经网络实际上就是一层层的tensor之间的运算，最终得到结果。这个过程就像是tensor在流动一般，这也是为什么谷歌的那个深度学习框架叫做TensorFlow。回到pytorch，其tensor有几个比较重要的属性，包括数据的形状、数据的步长、数据的类型、数据使用的设备，这个用于确定是在CPU上还是GPU上运算；还有就是数据所在内存中的布局，包括上面这个stride属性也是用于描述数据布局的。除了这些基本的属性外，tensor中有个很重要的属性grad，也就是梯度，其记录当前tensor上的梯度值；另一个属性requires\_grad用于控制tensor是否需要计算梯度。ↆ如果一个tensor不需要计算梯度，那么其没有梯度值。此外，所有与其相关的前面的tensor也不能计算梯度，因为梯度是要逆传播的，中间断了前面的也就不能计算了。这个属性是可以改的，我们有时候确实不需要计算梯度，不过在改的时候有一些需要注意的地方，否则经常容易报错。对于叶子节点，我们可以通requires\_grad\_()来设置是否需要记录梯度，对于非叶子节点，我们一般通过detach()方法来设置不需要梯度，有时候也能看到.data这种方式，作用其实差不多。关于叶子节点和非叶子节点马上会解释。最后，我们使用pytorch一般是用优化器来更新参数，为了能够更新一个tensor，一般需要使用Parameter类对Tensor包装一下。例如，我们平时创建了一个卷积类，那么这个卷积层的参数实际上就是paramter类的。

最后简单说一下torch中是如何实现梯度方向传播的。回到我们之前的例子，在我们一步步的前向计算的时候，torch会生成一张计算图ↆ，实际上就是一个有向无环图的结构，然后图中的节点是参数，边是运算操作，如图所示。需要注意的是，torch中的计算图是在前向计算中一步步得到的，每计算一步生成一部分，称为动态图。在生成动态图的过程中使用grad\_fn属性来记录变量是怎么来的，方便计算梯度。动态图在每一次参数更新后就会释放，这样能够节省内存；另外一种计算图被称为静态图，当我们模型确定下来的时候，静态图也就可以确定下来，这样理论上效率会高一些，但灵活性会差一些。tensorflow使用的是这种静态图，不过随着框架的发展，现在也不尽然，torch有些地方也会用静态图，tensorflow也可以弄动态图。

在前向生成计算图的时候ↆ，也会生成相应的反向图，梯度反向传播过程是以loss为输入，通过这个反向的过程来计算各个节点上的梯度。

然后看这个计算图ↆ，其最终节点，一般也就是loss被称为根节点，然后不依赖其他节点的节点被称为叶子节点，其余节点被称为子节点。之所以要这样分类，是因为绝大多数情况下只有叶子节点需要保留梯度，因为我们更新的这些权值全部都是叶子节点，所以非叶子节点上的梯度一般会被释放来提升效率。ↆ如果执行上面的代码访问中间变量y的梯度，torch会给出一个很长的警告（这里没有列出），然后打印的结果是none。当然，这个也可以通过设置让其保留，ↆ，使用retain\_grad()即可保留中间变量的梯度。