**basic NN**

**tensor，叶子节点**

每一个tensor都具有3个主要信息：数据域，梯度和计算图

tensor

数据域

梯度

计算图buffer

我们考虑一个简单的神经网络：

如果表示为tensor的计算图，则为：

点乘

点乘

不更新，不需要计算梯度的

称为叶子节点

需要计算梯度的，但由人为创建的

称为叶子节点

**梯度，反向传播**

由计算图我们可以得到：

的偏导数为：

由链式法则可知，在前向传播的过程中（也就是变量赋值过程），我们把每一层的偏导数相乘就可以得到叶子节点对于**目标变量**的偏导数。在前向传播中会生成计算图的buffer。当执行backward()时，默认情况下只有require\_grad的叶子节点的梯度会被记录，计算图会被销毁。

使用**目标变量**.backward()可以得到叶子tensor的梯度，即

如果我们把tensor 的每个分量的偏导数都按照顺序排好，可以得到一个同型的梯度tensor，它被记录在tensor的梯度之中。（把两组权重的雅克比矩阵排列起来）

以此类推。

中间变量的backward

从链式法则可知，如果我们已知中间变量tensor 关于目标的偏导数，就可以通过它乘以叶子节点关于的偏导数得到叶子节点关于目标变量的梯度。



对于中的每一组权重，我们首先可以得到其关于中间变量的雅克比矩阵，

我们可以看到，叶子节点的梯度就是这个中间变量的雅克比矩阵左乘中间变量的梯度矩阵。



**梯度累积**

每个tensor的梯度域，会根据每次的反向传播结果进行累计。可以通过zero\_grad清空。

**参数更新**

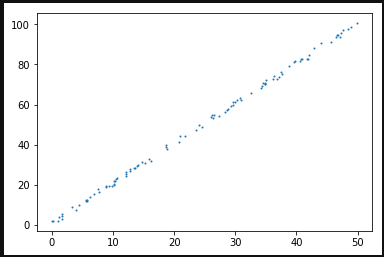
实现此简单网络的权重更新。

**线性回归模型**

参照《动手学深度学习》里的线性回归模型

设直线为，是符合正态分布的噪声。

首先生成数据集，先生成100个x坐标，然后根据直线计算y坐标，再加上噪声。



mini-batch

batch代表的是每一次参数更新时的数据量，损失函数

**再谈反向传播**

假设一个叶子tensor最终导向了多个目标变量，则对其中一个目标变量的反向传播不会影响另一个，如果两个都进行了反向传播，则梯度会被累积。

tensor

.backward ()

tensor

tensor

.backward ()

**tensor的in-place操作**

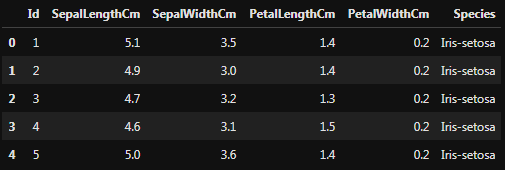
pytorch中的规定，tensor本身不能采取in-place操作，比如自加：t += 1。但是，我们可以写t=t+1这样的写法。这是因为，编译器不认为它是一个in-place操作，等号左边的t和右边的t是不同的。这样的写法会产生的一个问题就是如果t原来是一个叶子节点，通过t=t+1后，t指向了新的tensor，这样反向传播后就会丢失t的梯度，因为pytorch只保存叶子节点的梯度，而该节点的指针已经丢失。

**.data**

它是旧版本中的函数，返回和Variable同型的tensor，并且共享数据域。新版本中不建议使用。

**用pytorch内置函数实现softmax回归**

Iris数据集：



这个数据集有4个特征，标签有3个。我们可以使用如下的神经网络：

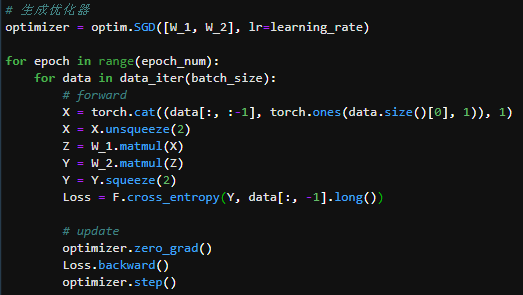
**关于矩阵相乘matmul()**

Pytorch中，向量和一维矩阵不能等价，如果说类似，向量可以看作是竖着的一维矩阵。

Matmul()是一个自动的矩阵相乘函数，根据输入数值的维数而变化。

当两个输入都小于等于2维时，该函数等价于向量-向量，矩阵-矩阵，或矩阵-向量乘法。

当其中一个维数大于2维时，则将前几维视为batch的维数。



这么写的原因是，数据是一维的，加上batch变成了二维数据，而参数也是二维数据，使用matmul()会将它们当做矩阵相乘，不会有广播机制。所以我需要先将batch的一维数据先变成三维数据，这样它就会把第一位识别为batch，后两位识别为矩阵。这里原本是，通过unsqueeze扩张维度变为，然后再与参数矩阵相乘，放在右侧。