反向传播算法

1. 前向传播 `forward propagation` 过程： 当前馈神经网络接收输入 $\vec{\mathbf{x}}$ 并产生输出 $\mathbf{y}$ 时，信息前向流动。

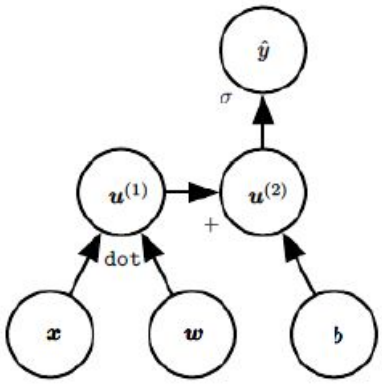
   输入 $\vec{\mathbf{x}}$ 提供初始信息，然后信息传播到每一层的隐单元，最终产生输出 $\mathbf{y}$ 。

2. 反向传播算法 `back propagation` 允许来自代价函数的信息通过网络反向流动以便计算梯度。
   - 反向传播并不是用于学习整个神经网络的算法，而是仅用于计算梯度的算法。

     神经网络的学习算法是随机梯度下降这类基于梯度的算法。
   - 反向传播不仅仅适用于神经网络，原则上它适用于计算任何函数的导数。

3. 计算图 `computational graph` ：
   - 图中的每个节点代表一个变量（可以是标量、向量、矩阵或者张量）。
   - `操作：operation` 为一个或者多个变量的简单函数。
     - 多个操作组合在一起可以描述一个更复杂的函数。
     - 一个操作仅返回单个输出变量（可以是标量、向量、矩阵或者张量）。
   - 如果变量 $\mathbf{y}$ 是变量 $\mathbf{x}$ 通过一个操作计算得到，则在图中绘制一条从 $\mathbf{x}$ 到 $\mathbf{y}$ 的有向边。

   如： $\hat{y} = \sigma(\vec{\mathbf{x}}^T \vec{\mathbf{w}} + b)$ 的计算图：



一、链式法则

1. 反向传播算法是一种利用链式法则计算微分的算法。
2. 在一维的情况下，链式法则为： $\frac{dz}{dx} = \frac{dz}{dy} \times \frac{dy}{dx}$ 。
3. 在多维情况下，设： $\vec{\mathbf{x}} \in \mathbb{R}^m, \vec{\mathbf{y}} \in \mathbb{R}^n$， g 为 $\mathbb{R}^m$ 到 $\mathbb{R}^n$ 的映射且满足 $\vec{\mathbf{y}} = g(\vec{\mathbf{x}})$， f 为 $\mathbb{R}^n$ 到 $\mathbb{R}$ 的映射且满足 $z = f(\vec{\mathbf{y}})$。则有：

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}, \quad i = 1, 2, \cdots, m$$

   使用向量记法，可以等价地写作：

$$\nabla_{\vec{\mathbf{x}}} z = \left(\frac{\partial \vec{\mathbf{y}}}{\partial \vec{\mathbf{x}}} \right)^T \nabla_{\vec{\mathbf{y}}} z$$

其中：$\frac{\partial \vec{\mathbf{y}}}{\partial \vec{\mathbf{x}}}$ 为 g 的 $n \times m$ 阶雅可比矩阵，$\nabla_{\vec{\mathbf{x}}} z$ 为 z 对 $\vec{\mathbf{x}}$ 的梯度，$\nabla_{\vec{\mathbf{y}}} z$ 为 z 对 $\vec{\mathbf{y}}$ 的梯度：

$$\nabla_{\vec{\mathbf{x}}} z = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix} \quad \nabla_{\vec{\mathbf{y}}} z = \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{bmatrix} \quad \frac{\partial \vec{\mathbf{y}}}{\partial \vec{\mathbf{x}}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

反向传播算法由很多这样的雅可比矩阵与梯度的乘积操作组成。

1.1 张量链式法则

1. 链式法则不仅可以作用于向量，也可以应用于张量：
   - 首先将张量展平为一维向量。
   - 然后计算该向量的梯度。
   - 然后将该梯度重新构造为张量。
2. 记 $\nabla_{\mathbf{X}} z$ 为 z 对张量 $\mathbf{X}$ 的梯度。$\mathbf{X}$ 现在有多个索引（如：二维张量有两个索引），可以使用单个变量 i 来表示 $\mathbf{X}$ 的索引元组（如 $i = 1 \sim 9$ 表示：一个二维张量的索引，每个维度三个元素）。

   这就与向量中的索引方式完全一致：$(\nabla_{\mathbf{X}} z)_i = \frac{\partial z}{\partial x_i}$ 。

   如：

   $$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}$$

   则有：

   $$\nabla_{\mathbf{X}} z = \begin{bmatrix} \frac{\partial z}{\partial x_1} & \frac{\partial z}{\partial x_2} & \frac{\partial z}{\partial x_3} \\ \frac{\partial z}{\partial x_4} & \frac{\partial z}{\partial x_5} & \frac{\partial z}{\partial x_6} \\ \frac{\partial z}{\partial x_7} & \frac{\partial z}{\partial x_8} & \frac{\partial z}{\partial x_9} \end{bmatrix}$$

3. 设 $\mathbf{Y} = g(\mathbf{X}), z = f(\mathbf{Y})$，用单个变量 j 来表示 $\mathbf{Y}$ 的索引元组。则张量的链式法则为：

   $$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \Rightarrow \nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} y_j) \frac{\partial z}{\partial y_j}$$

   如：

   $$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}$$

   则有：

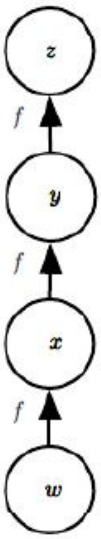
   $$\nabla_{\mathbf{X}} z = \begin{bmatrix} \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_1} & \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_2} & \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_3} \\ \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_4} & \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_5} & \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_6} \\ \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_7} & \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_8} & \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_9} \end{bmatrix}$$

.

1.2 重复子表达式

1. 给定计算图以及计算图中的某个标量 z ，根据链式法则可以很容易地写出 z 对于产生 z 的任意节点的梯度的数学表达式。

   但是在计算该表达式的时候，许多子表达式可能在计算整个梯度表达式的过程中重复很多次。



   如图中：

$$
\begin{aligned}
x &= f(w) \\
y &= f(x) \\
z &= f(y) \\
\Rightarrow \frac{dz}{dw} &= \frac{dz}{dy}\frac{dy}{dx}\frac{dx}{dw} \\
&= f'(y)f'(x)f'(w) \\
&= f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

   可以看到 $f(w)$ 被计算多次。

   - 在复杂的计算图中，可能存在指数量级的重复子表达式，这使得原始的链式法则几乎不可实现。
   - 一个解决方案是：计算 $f(w)$ 一次并将它存储在 x 中，然后采用 $f'(y)f'(x)f'(w)$ 来计算梯度。

     这也是反向传播算法采用的方案：在前向传播时，将节点的中间计算结果全部存储在当前节点上。其代价是更高的内存开销。

2. 有时候必须重复计算子表达式。这是以较高的运行时间为代价，来换取较少的内存开销。

二、反向传播

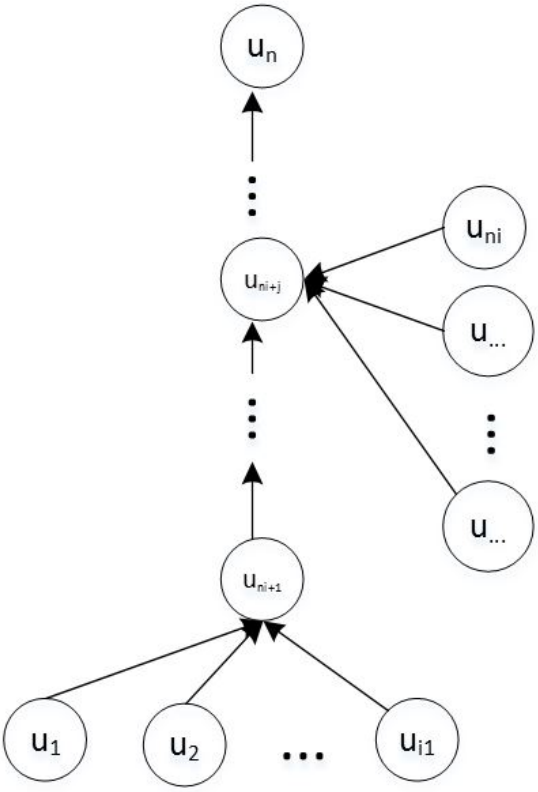
2.1 前向传播

1. 考虑计算单个标量 u_n 的计算图：

   - 假设有 n_i 个输入节点： $u_1, u_2, \cdots, u_{n_i}$ 。它们对应的是模型的参数和输入。
   - 假设 $u_{n_i+1}, \cdots$ 为中间节点。
   - 假设 u_n 为输出节点，它对应的是模型的代价函数。

- 对于每个非输入节点 u_i，定义其双亲节点的集合为 $\mathbb{A}_i$。
- 假设每个非输入节点 u_i ，操作 f_i 与其关联，并且通过对该函数求值得到：$u_i = f_i(\mathbb{A}_i)$ 。

通过仔细排序（有向无环图的拓扑排序算法），使得可以依次计算 $u_{n_i+1},\cdots,u_n$ 。

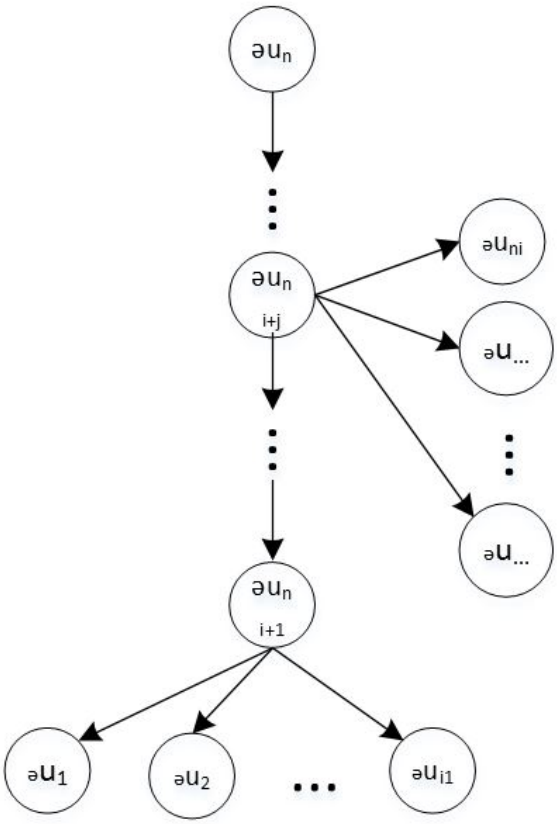


2. 前向传播算法：
   - 输入：
     - 计算图 $\mathcal{G}$
     - 初始化向量 $\vec{\mathbf{u}}^*$
   - 输出：u_n 的值
   - 算法步骤：
     - 初始化输入节点： $u_i = u_i^*, i = 1,2,\cdots,n_i$ 。
     - 根据计算图，从前到后计算 $u_{n_i+1},\cdots,u_n$。对于 $u_j, j = n_i+1,\cdots,n$ 计算过程为：
       - 计算 u_j 的双亲节点集合 $\mathbb{A}_j$。
       - 计算 u_j： $u_j = f_j(\mathbb{A}_j)$ 。
     - 输出 u_n 。

2.2 反向传播

1. 计算 $\frac{\partial u_n}{\partial u_j}, j = 1,2,\cdots,n_i$ 时需要构造另一张计算图 $\mathcal{B}$： 它的节点与 $\mathcal{G}$ 中完全相同，但是计算顺序完全相反。

   计算图 $\mathcal{B}$ 如下图所示：



2. 对于图中的任意一非输出节点 u_j （非 u_n），根据链式法则：

$$\frac{\partial u_n}{\partial u_j} = \sum_{(\partial u_i, \partial u_j) \in \mathcal{B}} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

其中 $(\partial u_i, \partial u_j) \in \mathcal{B}$ 表示图 $\mathcal{B}$ 中的边 $\partial u_i \to \partial u_j$ 。

- 若图 $\mathcal{B}$ 中存在边 $\partial u_i \to \partial u_j$，则在图 $\mathcal{G}$ 中存在边 $u_j \to u_i$，则 u_i 为 u_j 的子节点。
- 设图 $\mathcal{G}$ 中 u_j 的子节点的集合为 $\mathbb{C}_j$，则上式改写作：

$$\frac{\partial u_n}{\partial u_j} = \sum_{u_i \in \mathbb{C}_j} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

3. 反向传播算法：

- 输入：
  - 计算图 $\mathcal{G}$
  - 初始化参数向量 $\vec{\mathbf{u}}^*$
- 输出： $\frac{\partial u_n}{\partial u_j}, j = 1, 2, \cdots, n_i$
- 算法步骤：
  - 运行计算 u_n 的前向算法，获取每个节点的值。
  - 给出一个 `grad_table` 表，它存储的是已经计算出来的偏导数。

    u_i 对应的表项存储的是偏导数 $\frac{\partial u_n}{\partial u_i}$ 。

- 初始化 $\text{grad\_table}[u_n]=1$ 。
- 沿着计算图 $\mathcal{B}$ 计算偏导数。遍历 j 从 $n-1$ 到 1：
  - 计算 $\frac{\partial u_n}{\partial u_j}=\sum_{u_i\in\mathbb{C}_j}\frac{\partial u_n}{\partial u_i}\frac{\partial u_i}{\partial u_j}$ 。其中：$\frac{\partial u_n}{\partial u_i}$ 是已经存储的 $\text{grad\_table}[u_i]$ ，$\frac{\partial u_i}{\partial u_j}$ 为实时计算的。

    图 $\mathcal{G}$ 中的边 $u_j\rightarrow u_i$ 定义了一个操作，而该操作的偏导只依赖于这两个变量，因此可以实时求解 $\frac{\partial u_i}{\partial u_j}$ 。
  - 存储 $\text{grad\_table}[u_j]$ 。
- 返回 $\text{grad\_table}[u_j],j=1,2,\cdots,n_i$ 。

4. 反向传播算法计算所有的偏导数，计算量与 $\mathcal{G}$ 中的边的数量成正比。

   其中每条边的计算包括计算偏导数，以及执行一次向量点积。

5. 上述反向传播算法为了减少公共子表达式的计算量 ， 并没有考虑存储的开销。这避免了重复子表达式的指数级的增长。
   - 某些算法可以通过对计算图进行简化从而避免更多的子表达式。
   - 有些算法会重新计算这些子表达式而不是存储它们，从而节省内存。

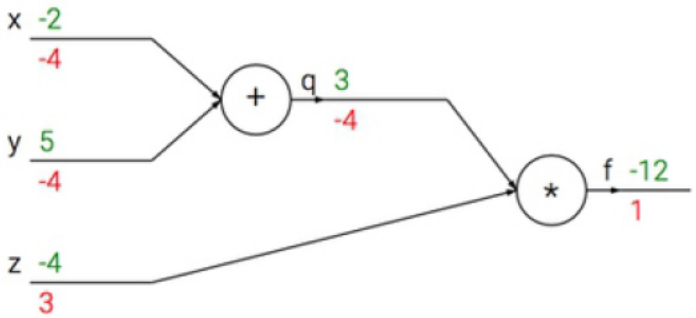
2.3 反向传播示例

1. 对于 $f(x,y,z)=(x+y)z$，将公式拆分成 $q=x+y$ 和 $f=qz$，则有：

$$\frac{\partial q}{\partial x}=1,\quad \frac{\partial q}{\partial y}=1,\quad \frac{\partial f}{\partial q}=z,\quad \frac{\partial f}{\partial z}=q$$

   根据链式法则，有 $\frac{\partial f}{\partial x}=\frac{\partial f}{\partial q}*\frac{\partial q}{\partial x}$ 。

   假设 $x=-2,y=5,z=-4$，则计算图如下。其中：绿色为前向传播的值，红色为反向传播的结果。



   - 前向传播，计算从输入到输出（绿色）；反向传播，计算从尾部开始到输入（红色）。
   - 在整个计算图中，每个单元的操作类型，以及输入是已知的。通过这两个条件可以计算出两个结果：
     - 这个单元的输出值。
     - 这个单元的输出值关于输入值的局部梯度比如 $\frac{\partial q}{\partial x}$ 和 $\frac{\partial q}{\partial y}$ 。

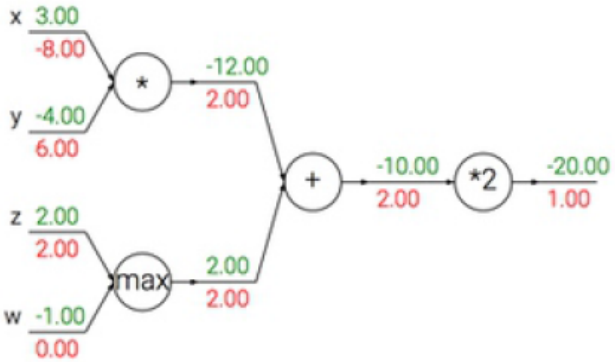
     每个单元计算这两个结果是独立完成的，它不需要计算图中其他单元的任何细节。

     但是在反向传播过程中，单元将获取整个网络的最终输出值（这里是 f ）在单元的输出值上的梯度，即回传的梯度。

     链式法则指出：**单元应该将回传的梯度乘以它对其输入的局部梯度，从而得到整个网络的输出对于该单元每个输入值的梯度。**如：$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial q}*\frac{\partial q}{\partial x}$。

2. 在多数情况下，反向传播中的梯度可以被直观地解释。如：加法单元、乘法单元、最大值单元。

   假设： $f=2*(x*y+\max(z,w))$，前向传播的计算从输入到输出（绿色），反向传播的计算从尾部开始到输入（红色）。



   - 加法单元 $q=x+y$，则 $\frac{\partial q}{\partial x}=1,\frac{\partial q}{\partial y}=1$。如果 $\frac{\partial f}{\partial q}=m$，则有：

$$\frac{\partial f}{\partial x}=m,\quad \frac{\partial f}{\partial y}=m$$

     这表明：加法单元将回传的梯度相等的分发给它的输入。

   - 乘法单元 $q=x*y$，则 $\frac{\partial q}{\partial x}=y,\frac{\partial q}{\partial y}=x$。如果 $\frac{\partial f}{\partial q}=m$，则有：

$$\frac{\partial f}{\partial x}=my,\quad \frac{\partial f}{\partial y}=mx$$

     这表明：乘法单元交换了输入数据，然后乘以回传的梯度作为每个输入的梯度。

   - 取最大值单元 $q=max(x,y)$，则：

$$\frac{\partial q}{\partial x}=\begin{cases}1, & x>=y\\ 0, & x<y\end{cases},\quad \frac{\partial q}{\partial y}=\begin{cases}1, & y>=x\\ 0, & y<x\end{cases}$$

     如果 $\frac{\partial f}{\partial q}=m$，则有：

$$\frac{\partial f}{\partial x}=\begin{cases}m, & x>=y\\ 0, & x<y\end{cases},\quad \frac{\partial f}{\partial y}=\begin{cases}m, & y>=x\\ 0, & y<x\end{cases}$$

     这表明：取最大值单元将回传的梯度分发给最大的输入。

3. 通常如果函数 $f(x,y)$ 的表达式非常复杂，则当对 x,y 进行微分运算，运算结束后会得到一个巨大而复杂的表达式。

   - 实际上并不需要一个明确的函数来计算梯度，只需要如何使用反向传播算法计算梯度即可。
   - 可以把复杂的表达式拆解成很多个简单的表达式（这些表达式的局部梯度是简单的、已知的），然后利用链式法则来求取梯度。
   - 在计算反向传播时，前向传播过程中得到的一些中间变量非常有用。实际操作中，最好对这些中间变量缓存。

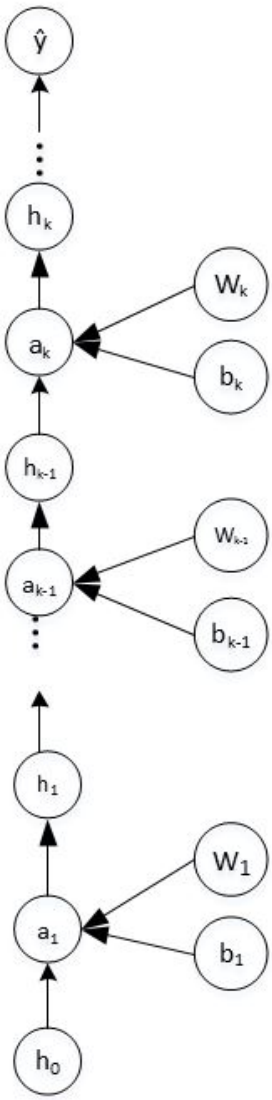
2.4 深度前馈神经网络应用

1. 给定一个样本，其定义代价函数为 $L(\hat{y},y)$，其中 $\hat{y}$ 为神经网络的预测值。

   考虑到正则化项，定义损失函数为：$J(\vec{\theta})=L(\hat{y},y)+\Omega(\vec{\theta})$。其中 $\Omega(\vec{\theta})$ 为正则化项，而 $\vec{\theta}$ 包含了所有的参数（包括每一层的权重 $\mathbf{W}_i$ 和每一层的偏置 $\vec{\mathbf{b}}_i$）。

这里给出的是单个样本的损失函数，而不是训练集的损失函数。

2. 计算 $\hat{y}$ 的计算图为：



3. 前向传播用于计算深度前馈神经网络的损失函数。算法为：
   - 输入：
     - 网络层数 l
     - 每一层的权重矩阵 $\mathbf{W}_i, i = 1, 2, \cdots, l$
     - 每一层的偏置向量 $\vec{\mathbf{b}}_i, i = 1, 2, \cdots, l$
     - 每一层的激活函数 $f_i(\cdot), i = 1, 2, \cdots, l$

       > 也可以对所有的层使用同一个激活函数

     - 输入 $\vec{\mathbf{x}}$ 和对应的标记 y 。
     - 隐层到输出的函数 $\mathrm{func}(\cdot)$ 。
   - 输出：损失函数 $J(\vec{\theta})$
   - 算法步骤：

- 初始化 $\vec{\mathbf{h}}_0 = \vec{\mathbf{x}}$
- 迭代：$k = 1, 2, \cdots, l$，计算：
  - $\vec{\mathbf{a}}_k = \vec{\mathbf{b}}_k + \mathbf{W}_k \vec{\mathbf{h}}_{k-1}$
  - $\vec{\mathbf{h}}_k = f_k(\vec{\mathbf{b}}_k)$
- 计算 $\hat{y} = \text{func}(\vec{\mathbf{h}}_l)$， $J(\vec{\theta}) = L(\hat{y}, y) + \Omega(\vec{\theta})$ 。

4. 反向传播用于计算深度前馈网络的损失函数对于参数的梯度。

   梯度下降算法需要更新模型参数，因此只关注损失函数对于模型参数的梯度，不关心损失函数对于输入的梯度 $\nabla_{\vec{\mathbf{x}}} J$ 。

   - 根据链式法则有： $\nabla_{\vec{\mathbf{a}}_k} J = (\frac{\partial \vec{\mathbf{h}}_k}{\partial \vec{\mathbf{a}}_k})^T \nabla_{\vec{\mathbf{h}}_k} J$ 。

     考虑到 $\vec{\mathbf{h}}_k = f_k(\vec{\mathbf{a}_\mathbf{k}})$ ，因此雅可比矩阵 $\frac{\partial \vec{\mathbf{h}}_k}{\partial \vec{\mathbf{a}}_k}$ 为对角矩阵，对角线元素 $[\frac{\partial \vec{\mathbf{h}}_k}{\partial \vec{\mathbf{a}}_k}]_{i,i} = f_k'(a_{k,i})$。$a_{k,i}$ 表示 $\vec{\mathbf{a}}_k$ 的第 i 个元素。

     因此 $\nabla_{\vec{\mathbf{a}}_k} J = (\frac{\partial \vec{\mathbf{h}}_k}{\partial \vec{\mathbf{a}}_k})^T \nabla_{\vec{\mathbf{h}}_k} J = (\nabla_{\vec{\mathbf{h}}_k} J) \odot f_k'(\vec{\mathbf{a}}_k)$ ，其中 $\odot$ 表示 `Hadamard` 积。

   - 因为 $\vec{\mathbf{a}}_k = \vec{\mathbf{b}}_k + \mathbf{W}_k \vec{\mathbf{h}}_{k-1}$，因此：

     $$\nabla_{\vec{\mathbf{b}}_k} J = \nabla_{\vec{\mathbf{a}}_k} J, \quad \nabla_{\mathbf{W}_k} J = (\nabla_{\vec{\mathbf{a}}_k} J) \vec{\mathbf{h}}_{k-1}^T$$

     上式仅仅考虑从 $L(\hat{y}, y)$ 传播到 $\mathbf{W}_k, \vec{\mathbf{b}}_k$ 中的梯度。 考虑到损失函数中的正则化项 $\Omega(\vec{\theta})$ 包含了权重和偏置，因此需要增加正则化项的梯度。则有：

     $$\nabla_{\vec{\mathbf{b}}_k} J = \nabla_{\vec{\mathbf{a}}_k} J + \lambda \nabla_{\vec{\mathbf{b}}_k} \Omega(\vec{\theta})$$
     $$\nabla_{\mathbf{W}_k} J = (\nabla_{\vec{\mathbf{a}}_k} J) \vec{\mathbf{h}}_{k-1}^T + \lambda \nabla_{\mathbf{W}_k} \Omega(\vec{\theta})$$

   - 因为 $\vec{\mathbf{a}}_k = \vec{\mathbf{b}}_k + \mathbf{W}_k \vec{\mathbf{h}}_{k-1}$，因此： $\nabla_{\vec{\mathbf{h}}_{k-1}} J = \mathbf{W}_k^T \nabla_{\vec{\mathbf{a}}_k} J$ 。

5. 反向传播算法：

   - 输入：
     - 网络层数 l
     - 每一层的权重矩阵 $\mathbf{W}_i, i = 1, 2, \cdots, l$
     - 每一层的偏置向量 $\vec{\mathbf{b}}_i, i = 1, 2, \cdots, l$
     - 每一层的激活函数 $f_i(\cdot), i = 1, 2, \cdots, l$
     - 输入 $\vec{\mathbf{x}}$ 和对应的标记 y
   - 输出：梯度 $\nabla_{\vec{\mathbf{b}}_k} J, \nabla_{\mathbf{W}_k} J, \quad , k = 1, 2, \cdots, l$
   - 算法步骤：
     - 通过前向传播计算损失函数 $J(\vec{\theta})$ 以及网络的输出 $\hat{y}$ 。
     - 计算输出层的导数： $\nabla_{\hat{y}} J = \frac{\partial J}{\partial \hat{y}} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}}$ 。

       这里等式成立的原因是：正则化项 $\Omega(\vec{\theta})$ 与模型输出 $\hat{y}$ 无关。
     - 计算最后一层隐单元的梯度： $\vec{\mathbf{g}} \leftarrow \nabla_{\vec{\mathbf{h}}_l} J = \nabla_{\hat{y}} J \times \nabla_{\vec{\mathbf{h}}_l} \hat{y}$ 。
     - 迭代： $k = l, l-1, \cdots, 1$，迭代步骤如下：

       每一轮迭代开始之前，维持不变式： $\vec{\mathbf{g}} = \nabla_{\vec{\mathbf{h}}_k} J$ 。
       - 计算 $\nabla_{\vec{\mathbf{a}}_k} J$ ： $\nabla_{\vec{\mathbf{a}}_k} J = (\nabla_{\vec{\mathbf{h}}_k} J) \odot f_k'(\vec{\mathbf{a}}_k) = \vec{\mathbf{g}} \odot f_k'(\vec{\mathbf{a}}_k)$ 。

- 令：$\vec{\mathbf{g}} \leftarrow \nabla_{\vec{\mathbf{a}}_k} J = \vec{\mathbf{g}} \odot f_k'(\vec{\mathbf{a}}_k)$ 。
- 计算对权重和偏置的偏导数：

$$\nabla_{\vec{\mathbf{b}}_k} J = \vec{\mathbf{g}} + \lambda \nabla_{\vec{\mathbf{b}}_k} \Omega(\vec{\theta})$$

$$\nabla_{\mathbf{W}_k} J = \vec{\mathbf{g}} \vec{\mathbf{h}}_{k-1}^T + \lambda \nabla_{\mathbf{W}_k} \Omega(\vec{\theta})$$

- 计算 $\nabla_{\vec{\mathbf{h}}_{k-1}} J$ ： $\nabla_{\vec{\mathbf{h}}_{k-1}} J = \mathbf{W}_k^T \nabla_{\vec{\mathbf{a}}_k} J = \mathbf{W}_k^T \vec{\mathbf{g}}$ 。
- 令：$\vec{\mathbf{g}} \leftarrow \nabla_{\vec{\mathbf{h}}_{k-1}} J$ 。

三、算法实现

3.1 符号-数值 / 符号-符号方法

1. 代数表达式和计算图都是对符号 `symbol` 进行操作，这些基于代数的表达式或者基于图的表达式称作符号表达式。

   当训练神经网络时，必须给这些符号赋值。如：对于符号 $\vec{\mathbf{x}}$ 赋予一个实际的数值，如 $(1.1, 1.2, -0.3)^T$ 。

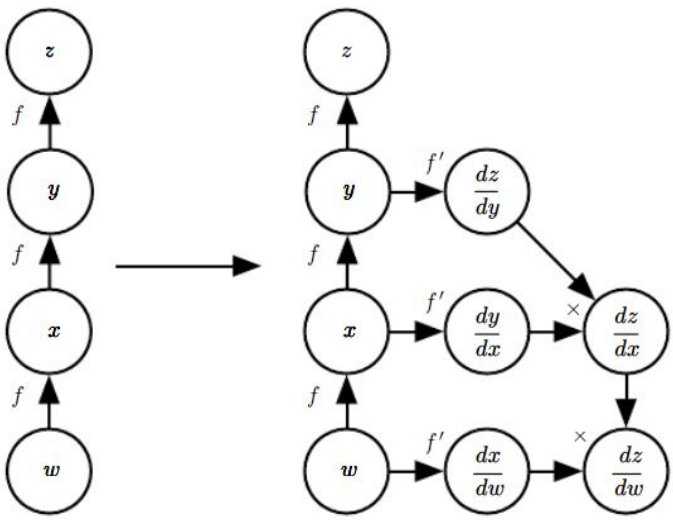
2. 符号到数值的方法：给定计算图，以及图的一组输入的数值，然后返回在这些输入值处的梯度。

   这种方法用于 `Torch` 和 `Caffe` 之类的库中。

3. 符号到符号的方法：给定计算图，算法会添加额外的一些节点到计算图中，这些额外的节点提供了所需的导数的符号描述。

   这种方法用于 `Theano` 和 `TensorFlow` 之类的库中。

   下图左侧为 $z = f(f(f(w)))$ 的计算图，右侧添加了若干节点从而给出了计算 $\frac{dz}{dw}$ 的计算图。



4. 符号到符号的方法的优点：导数可以使用与原始表达式相同的编程语言来描述。

   导数只是另外一张计算图，因此可以再次运行反向传播算法对导数再进行求导，从而获取更高阶的导数。

5. 推荐使用符号到符号的方法来求导数。一旦构造出了添加导数后的计算图，那么随后如果给出了输入的数值，可以对图中的任意子节点求值。

   目前通用的计算图求解引擎的做法是：任意时刻，一旦一个节点的父节点都求值完毕，那么该节点将能够立即求值。

6. 事实上符号到数值的方法与符号到符号的方法执行了同样的计算过程，区别在于：

- 符号到数值的方法并没有暴露内部的计算过程。
- 符号到符号的方法将各种求导运算暴露出来，添加到计算图中成为了节点。

3.2 算法框架

1. 假设计算图 $\mathcal{G}$ 中的每个节点对应一个变量。这里将变量描述为一个张量 $\mathbf{V}$，它可以具有任意维度并且可能是标量、向量、矩阵。

   根据前面介绍的张量的链式法则， $\mathbf{Y}=g(\mathbf{X}), z=f(\mathbf{Y})$，则张量的链式法则为：

   $$\nabla_{\mathbf{X}} z=\sum_{j}(\nabla_{\mathbf{X}} y_j)\frac{\partial z}{\partial y_j}$$

   其中 j 为张量 $\mathbf{Y}$ 展平为一维向量后的索引， y_j 为张量 $\mathbf{Y}$ 展平为一维向量之后的第 j 个元素。

3.2.1 三个子过程

1. $\text{get\_operation}(\mathbf{V})$ ：返回用于计算 $\mathbf{V}$ 的操作 `operation` 。它就是 `tensorflow` 中的 `Operation` 对象。

   该函数通常返回一个操作对象 op：

   - 该对象有个 `f` 方法，该方法给出了父节点到 $\mathbf{V}$ 的函数：$\mathbf{V}=\text{op}.f(\text{inputs})$ 。

     其中 inputs 为 $\mathbf{V}$ 的父节点集合： $\mathbb{A}_{\mathbf{V}}^{(\mathcal{G})}$

   - 该操作对象有个 `bprop` 方法。给定 $\mathbf{V}$ 的某个子节点 $\mathbf{C}$，该方法用于已知 $\mathbf{C}$ 的梯度 $\mathbf{G}_C=\nabla_{\mathbf{C}} z$ ，求解 $\mathbf{C}$ 对于 $\mathbf{V}$ 的梯度的贡献：$\text{op.bprop}(\mathbf{C},\mathbf{V},\mathbf{G}_C)=(\nabla_{\mathbf{V}}\mathbf{C})\mathbf{G}_C$ 。

     如果考虑 $\mathbf{V}$ 的所有子节点集合 $\mathbb{C}_{\mathbf{V}}^{(\mathcal{G})}$ ，则 它们的梯度贡献之和就是总的梯度：

     $$\nabla_{\mathbf{V}} z=\sum_{\mathbf{C}\in\mathbb{C}_{\mathbf{V}}^{(\mathcal{G})}}(\nabla_{\mathbf{V}}\mathbf{C})\mathbf{G}_C$$

2. $\text{get\_consumers}(\mathbf{V},\mathcal{G})$：返回图 $\mathcal{G}$ 中节点 $\mathbf{V}$ 的子节点列表，也就是节点 $\mathbf{V}$ 的子节点集合： $\mathbb{C}_{\mathbf{V}}^{(\mathcal{G})}$ 。
3. $\text{get\_inputs}(\mathbf{V},\mathcal{G})$：返回图 $\mathcal{G}$ 中节点 $\mathbf{V}$ 的父节点列表，也就是 $\mathbf{V}$ 的父节点集合： $\mathbb{A}_{\mathbf{V}}^{(\mathcal{G})}$ 。
4. `op.bprop` 方法总是假定其输入节点各不相同。

   如果定义了一个乘法操作，而且每条输入节点都是 `x` ，则 `op.bprop` 方法也会认为它们是不同的：

   `op.bprop` 会认为其输入分别为 `y` 和 `z` ，然后求出表达式之后再代入 `y=x,z=x` 。
5. 大多数反向传播算法的实现都提供了 `operation` 对象以及它的 `bprop` 方法。

   如果希望添加自己的反向传播过程，则只需要派生出 `op.bprop` 方法即可。

3.2.2 反向传播过程

1. `build_grad` 过程采用 `符号-符号方法` ，用于求解单个结点 $\mathbf{V}$ 的梯度 $\nabla_{\mathbf{V}} z$ 。
2. `build_grad` 在求解过程中会用到裁剪的计算图 $\mathcal{G}'$ ， $\mathcal{G}'$ 会剔除所有与 $\mathbb{V}$ 梯度无关的节点，保留与 $\mathbb{V}$ 梯度有关的节点。
3. `build_grad` 过程：
   - 输入：
     - 待求梯度的节点 $\mathbf{V}$
     - 计算图 $\mathcal{G}$
     - 被裁减的计算图 $\mathcal{G}'$

        - 梯度表 grad_table
    - 输出： $\nabla_{\mathbf{V}} z$
    - 算法步骤：
        - 如果 $\mathbf{V}$ 已经就在 grad_table 中，则直接返回 $\text{grad\_table}[\mathbf{V}]$ 。

          > 这样可以节省大量的重复计算

        - 初始化 $\mathbf{G} = \mathbf{0}$ 。
        - 在图 $\mathcal{G}'$ 中， 迭代遍历 $\mathbf{V}$ 的子节点的集合 $\mathbb{C}_{\mathbf{V}}^{(\mathcal{G}')}$ ： for $\mathbf{C}$ in $\text{get\_consumers}(\mathbf{V}, \mathcal{G}')$:
            - 获取计算 $\mathbf{C}$ 的操作： $\text{op} = \text{get\_operation}(\mathbf{C})$
            - 获取该子节点的梯度，这是通过递归来实现的： $\mathbf{D} = \text{build\_grad}(\mathbf{C}, \mathcal{G}, \mathcal{G}', \text{grad\_table})$ 。

              > 因为子节点更靠近输出端，因此子节点 $\mathbf{C}$ 的梯度一定是先于 $\mathbf{V}$ 的梯度被计算。

            - 计算子节点 $\mathbf{C}$ 对于 $\nabla_{\mathbf{V}} z$ 的贡献： $\mathbf{G}_C = \text{op.bprop}(\mathbf{C}, \mathbf{V}, \mathbf{D})$ 。
            - 累加子节点 $\mathbf{C}$ 对于 $\nabla_{\mathbf{V}} z$ 的贡献： $\mathbf{G} \leftarrow \mathbf{G} + \mathbf{G}_C$ 。
        - 存储梯度来更新梯度表： $\text{grad\_table}[\mathbf{V}] = \mathbf{G}$ 。
        - 在 $\mathcal{G}$ 中插入节点 $\mathbf{G}$ 来更新计算图 $\mathcal{G}$ 。插入过程不仅增加了节点 $\mathbf{G}$ ， 还增加了 $\mathbf{G}$ 的父节点到 $\mathbf{G}$ 的边。
        - 返回 $\mathbf{G}$ 。

4. 反向传播过程：
    - 输入：
        - 计算图 $\mathcal{G}$
        - 目标变量 z
        - 待计算梯度的变量的集合 $\mathbb{T}$
    - 输出： $\frac{\partial z}{\partial \mathbf{V}}, \mathbf{V} \in \mathbb{T}$
    - 算法步骤：
        - 裁剪 $\mathcal{G}$ 为 $\mathcal{G}'$ ，使得 $\mathcal{G}'$ 仅包含 z 的祖先之中，那些同时也是 $\mathbb{T}$ 的后代的节点。

          > 因为这里只关心 $\mathbb{T}$ 中节点的梯度

        - 初始化 grad_table，它是一个表，各表项存储的是 z 对于对应节点的偏导数。
        - 初始化 $\text{grad\_table}[z] = 1$（因为 $\frac{\partial z}{\partial z} = 1$）。
        - 迭代：对每个 $\mathbf{V} \in \mathbb{T}$，执行 $\text{build\_grad}(\mathbf{V}, \mathcal{G}, \mathcal{G}', \text{grad\_table})$ 。
        - 返回 grad_table 。

3.3.3 算法复杂度

1. 算法复杂度分析过程中，我们假设每个操作的执行都有大概相同的时间开销。

   实际上每个操作可能包含多个算术运算，如：将矩阵乘法视为单个操作的话，就包含了很多乘法和加法。因此每个操作的运行时间实际上相差非常大。

2. 在具有 n 个节点的计算图中计算梯度，不会执行超过 $O(n^2)$ 的操作，也不会执行超过 $O(n^2)$ 个存储。

   因为最坏的情况下前向传播将遍历执行图中的全部 n 个节点，每两个节点之间定义了一个梯度。

3. 大多数神经网络的代价函数的计算图是链式结构，因此不会执行超过 $O(n)$ 的操作。

从 $O(n^2)$ 降低到 $O(n)$ 是因为：并不是所有的两个节点之间都有数据通路。

4. 如果直接用梯度计算公式来求解则会产生大量的重复子表达式，导致指数级的运行时间。

   反向传播过程是一种表填充算法，利用存储中间结果（存储子节点的梯度） 来对表进行填充。计算图中的每个节点对应了表中的一个位置，该位置存储的就是该节点的梯度。

   通过顺序填充这些表的条目，反向传播算法避免了重复计算公共表达式。这种策略也称作动态规划。

3.4、应用

1. 考虑只有单个隐层的最简单的深度前馈网络，使用小批量（`minibatch`）随机梯度下降法训练模型。反向传播算法用于计算单个`minibatch`上的代价函数的梯度。

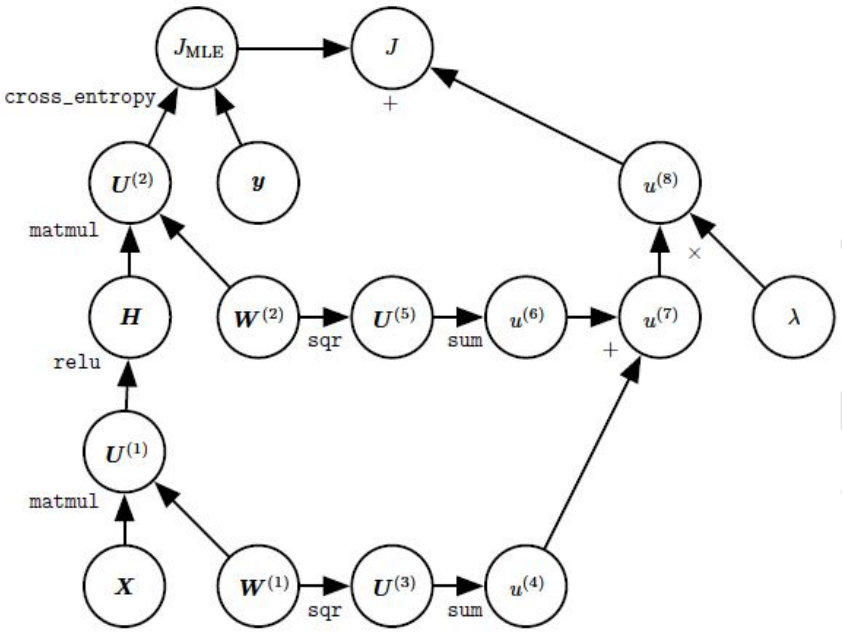
2. 取训练集上的一组`minibatch`实例，记做输入矩阵 $\mathbf{X}\in\mathbb{R}^{m\times n}$ ，矩阵的每一行就是一个实例，其中 m 为样本数量， n 为特征数量。同时给出标记 $\vec{\mathbf{y}}\in\mathbb{R}^{m\times 1}$ ，它是每个样本的真实标记。

   设激活函数为 `ReLU` 激活函数，设模型不包含偏置。设输入层到隐层的权重矩阵为 $\mathbf{W}_1$，则隐层的输出为：$\mathbf{H}=\max\{0,\mathbf{X}\mathbf{W}_1\}$ 。设隐层到输出层的权重矩阵为 $\mathbf{W}_2$ ，则分类的非归一化对数概率为 $\mathbf{H}\mathbf{W}_2$ 。

   假设程序包含了`cross_entropy`操作，用于计算未归一化对数概率分布定义的交叉熵 ，该交叉熵作为代价函数 J_{MLE} 。引入正则化项，总的代价函数为：$J=J_{MLE}+\lambda\left(\sum_{i,j}(\mathbf{W}_1[i,j])^2+\mathbf{W}_2[i,j])^2\right)$ 。

   > 交叉熵为 $J(\vec{\theta})=-\mathbb{E}_{\vec{\mathbf{x}},y\sim\hat{p}_{data}}\log p_{model}(y\mid\vec{\mathbf{x}};\vec{\theta})$ 。最小化交叉熵就是最大化似然估计

   其计算图如下所示：



3. 目标是通过小批量随机梯度下降法求解代价函数的最小值，因此需要计算 $\nabla_{\mathbf{W}_1}J,\nabla_{\mathbf{W}_2}J$。

   从图中看出有两种不同的路径从 J 回退到 $\mathbf{W}_1,\mathbf{W}_2$：

   - 一条路径是通过正则化项。

     这条路径对于梯度的贡献相对简单，它对于 $\mathbf{W}_i$ 的梯度贡献为 $2\lambda\mathbf{W}_i$ 。

   - 一条路径是通过交叉熵。

     - 对于 $\mathbf{W}_2$ ，这条分支其梯度的贡献为 $\mathbf{H}^T\mathbf{G}$，其中 $\mathbf{G}$ 为 $\frac{\partial J_{MLE}(\mathbf{Z})}{\partial\mathbf{Z}}$，将 $\mathbf{Z}$ 替换为 $\mathbf{H}\mathbf{W}_2$
     - 对于 $\mathbf{W}_1$，这条分支对于梯度的贡献计算为：

        - 首先计算 $\nabla_{\mathbf{H}}J=\mathbf{G}\mathbf{W}_2^T$ 。
        - 然后根据 `relu` 操作的反向传播规则：根据 $\mathbf{U}_1$ 中小于零的部分，对应地将 $\nabla_{\mathbf{H}}J$ 对应位置清零，记清零后的结果为 $\mathbf{G}'$ 。
        - 分支的梯度贡献为： $\mathbf{X}^T\mathbf{G}'$。
4. 该算法的计算成本主要来源于矩阵乘法：
    - 前向传播阶段（为了计算对各节点求值）：乘-加运算的数量为 $O(w)$，其中 w 为权重的数量。
    - 在反向传播阶段：具有相同的计算成本。
5. 算法的主要存储成本是：需要存储隐层非线性函数的输入。因此存储成本是 $O(mn_h)$，其中 m 为 `minibatch` 中样例的数量，n_h 是隐单元的数量。
6. 这里描述的反向传播算法要比现实中实际使用的实现更简单。
    - 这里定义的 `operation` 限制为返回单个张量的函数，大多数软件实现支持返回多个张量的 `operation` 。
    - 这里未指定如何控制反向传播的内存消耗。反向传播经常涉及将许多张量加在一起。
        - 朴素算法将分别计算这些张量，然后第二步中将所有张量求和，内存需求过高。
        - 可以通过维持一个 `buffer` ，并且在计算时将每个值加到 `buffer` 中来避免该瓶颈。
    - 反向传播的具体实现还需要处理各种数据类型，如32位浮点数、整数等。
    - 一些 `operation` 具有未定义的梯度，需要跟踪这些情况并向用户报告。

四、自动微分

1. 自动微分研究如何数值计算导数，反向传播算法只是自动微分的一种方法，它是反向模式累加算法的特殊情况。

   不同的自动微分算法以不同的顺序计算链式法则的子表达式，找到计算梯度的最优操作序列是 `NP` 完全问题。
2. 事实上，反向传播算法不能化简梯度计算公式，它只会通过计算图来传播梯度。如：

$$J=-\sum_i p_i\log\left(\frac{\exp(z_i)}{\sum_j\exp(z_j)}\right)\longrightarrow\frac{\partial J}{\partial z_i}=p_i\left(\frac{\exp(z_i)}{\sum_j\exp(z_j)}-1\right)$$

   而反向传播算法并不知道可以这样简化。

   有些软件如 `Theano` 和 `TenserFlow` 能够尝试执行某些类型的代数替换来部分地简化梯度计算公式。
3. 若前向图 $\mathcal{G}$ 具有单个输出节点，并且每个偏导数 $\frac{\partial u_i}{\partial u_j}$ 都能够以恒定的计算量来计算时，反向传播保证梯度的计算数量和前向计算的计算数量都在同一个量级。计算量都是 $O(l)$，l 为边的数量。
4. 反向传播算法可以扩展到计算 `Jacobi` 矩阵。矩阵来源可能是图中的 k 个不同标量节点；也可能来源于某个节点，但是该节点是个张量，张量内部具有 k 个内部标量。
5. 当图的输出数目大于输入数目时，有时倾向于使用另一种形式的自动微分，称作前向模式累加。
    - 前向模式累加可以用于循环神经网络梯度的实时计算。
    - 前向模式累加避免了存储整个图的值和梯度，是计算效率和内存消耗的折中。
    - 前向模式和后向模式（反向传播算法就是后向模式的一种）的关系类似于：左乘、右乘一系列矩阵。如： $\mathbf{ABC}\vec{\mathbf{d}}$ 。
        - 如果从右向左进行，则只需要计算矩阵-向量乘积。这对应着反向模式。
        - 如果从左向右进行，则需要计算矩阵-矩阵乘积（只有最后一次计算是矩阵-向量乘积）。这对应着前向模式，这时总的计算量和存储空间更大。

6. 在机器学习以外的领域，传统的编程语言比如`C/Python/C++`直接实现了求导数的库。
    - 在深度学习中无法使用这些库，而是必须使用专用库创建明确的数据结构来表示求导。
    - 深度学习需要深度学习库的开发人员为每个操作定义`bprop`方法，并且限制库的用户只能使用该方法来求导。
7. 一些软件框架支持使用高阶导数（如`Theano/TensorFlow`），这些库需要使用一种数据结构来描述要被微分的原始函数。

    它们使用相同的数据结构来描述原始函数的导数（导数也是一个函数对象），这意味着符号微分机制可以产生高阶导数。
8. 在深度学习领域很少关注标量函数的单个二阶导数，而是关注海森矩阵的性质。

    假设函数 $f:\mathbb{R}^n\rightarrow\mathbb{R}$，则海森矩阵大小为 $n\times n$。在某些深度学习任务中 n 为模型参数数量，很可能达到十亿，因此完整的海森矩阵无法在内存中表示。
    - 一个解决方案是使用`Krylov`方法。该方法使用矩阵-向量乘法来迭代求解矩阵的逆、矩阵特征值、矩阵特征向量等问题。
    - 为了使用`Krylov`方法，需要计算海森矩阵和任意向量 $\vec{\mathbf{v}}$ 的乘积：$\mathbf{H}\vec{\mathbf{v}}=\nabla_{\vec{\mathbf{x}}}[(\nabla_{\vec{\mathbf{x}}}f(\vec{\mathbf{x}}))^T\vec{\mathbf{v}}]$。

        > 因为 $\vec{\mathbf{v}}$ 与 $\vec{\mathbf{x}}$ 无关，因此 $\vec{\mathbf{v}}$ 在`[]`里面或者外面没有区别。

        上式中的两个梯度计算都可以由软件库自动完成。

        注意：如果 $\vec{\mathbf{v}}$ 本身如果已经是计算图产生的一个向量，则需要告诉软件库不要对产生 $\vec{\mathbf{v}}$ 的节点进行微分。