神经网络训练

网络训练的定义

- 给定神经网络中的参数，即权重和偏置，就可通过前向传播进行预测。
- 一个神经网络通常是针对某一特定问题而言的。例如，手写数字识别的神经网络，通常只能用来识别手写数字，而不能用来识别手写字母。尽管不同问题的网络架构可能相同，但其网络的参数不尽相同。
- 如何针对某一特定问题，确定合理的网络参数，是神经网络的关键。确定合理参数的过程，通常称为神经网络训练。

如果非要给一个数学上的定义，那么可能这样表述:

神经网络训练的定义

给定一组输入样本 $\{x: x\in X\}$， 设神经网络的输出为 $\{y(x;\theta): x\in X\}$, 其中 θ 是网络参数。神经网络训练是：求参数 θ 使得

$$L(\{y(x;\theta): x\in X\}) = \min,$$

式中，L 是关于神经网络输出的某种目标函数。

L 可能依赖于其他量，这与具体问题相关。例如，

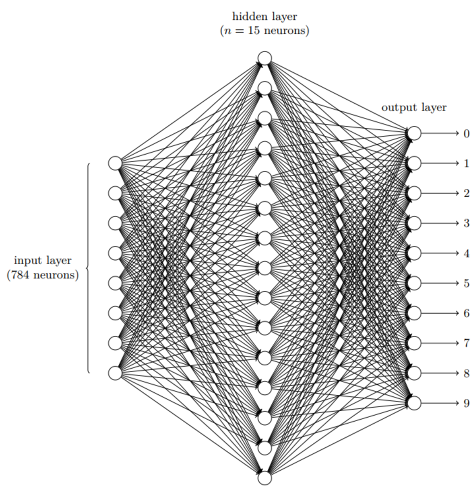
- 对手写数字识别，我们的目标是：神经网络预测的数字与真实数字相同。这样，在训练时，我们要给定每个样本对应的真实输出，保证样本总体的差距最小，即构造最小二乘。
- 对 PDE 问题，输入样本通常是定义域内的坐标点 $\boldsymbol{x}=(x_1,x_2)$，输出的是函数 $u(\boldsymbol{x};\theta)$ ，作为 PDE 的近似。由于真实解未知，因此不可能给出一组真实输出。此时，目标函数可以通过 PDE 问题本身构造。例如，对 Poisson 方程 $-\Delta u = f$, 可要求 $-\Delta u(\boldsymbol{x};\theta)\approx f$, 或

$$\|-\Delta u(\boldsymbol{x};\theta) - f\| = \min.$$

  当然，这里还要加上某种边界条件限制。

手写数字识别的优化问题

神经网络的训练归结为求解某个优化问题，下面回顾一下手写数字分类对应的优化问题。



- 给定一个手写数字的输入 x，即 784 维的向量，设其**真实输出为** $y=y(x)$。注意输出是 10 维向量，例如 $y(x)=(0,0,1,0,0,0,0,0,0,0)^T$ 表示数字 2。

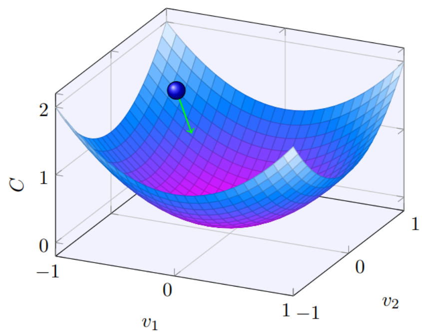
- 对训练数据 x，设神经网络给出的输出为 $a=a(x)$。
- 为了确定神经网络中的权重 w 和偏置 b，可最小化如下的损失函数：
$$C(w,b)=\frac{1}{2n}\sum_x \|a(x)-y(x)\|^2, \tag{1}$$
其中的 n 是训练时的输入数据个数。注意，$a(x):=a^L(x)$ 实际上应该为 $a(x;w,b)$，它是神经网络函数。

优化问题的求解

- 神经网络的参数确定归结为一个优化问题，即找到合适的参数，使得损失函数最小。
- 神经网络的优化问题是高维优化问题，常用梯度下降法求解。

梯度下降法

基本思想



梯度下降法的思想非常朴素，为了方便，以上图的函数 $C(v_1,v_2)$ 为例。注意它是是变量 v_1,v_2 的函数。

- 任取定义域内的一点 $v=(v_1,v_2)$，对应的函数值 $C(v)=C(v_1,v_2)$ 如图上的小球所示。
- 现在，我们让小球沿着某个方向滚动，并不断调整方向，使得它能尽快跑到图像的底部，从而获得最小值。
- 数学上可以证明，这个方向是负梯度方向 $-\nabla C$。设每次走的“步长”为 η，则自变量的更新规则为
$$v\to v'=v+\eta(-\nabla C)=v-\eta\nabla C.$$

神经网络的梯度下降

根据前面的规定，上一层的第 k 个神经元到下一层的第 j 个神经元的权重记为 w_{jk}，为了方便，以下称指向的层为目标层。设目标层为第 l 层，那么指向目标层的权重我们就记为 w_{jk}^l。这样，上一层对目标层的贡献可写为

$$a_j^l=\sigma\Big(\sum_k w_{jk}^l a_k^{l-1}+b_j^l\Big) \tag{2}$$

或写成矩阵形式

$$a^l=\sigma(w^l a^{l-1}+b^l),\quad l=2,3,\cdots \tag{3}$$

$a(x)$ 实际上就是上面公式的递推复合。

对 $C(w,b)$ 这种复杂非线性函数的优化问题，我们使用梯度下降法求解。为了方便，将变量 w,b 统一写为向量 $v=(v_1,v_2,\cdots,v_n)^T$，并记 $C(v)=C(w,b)$。给定初始值 v，梯度下降法如下更新 v：

$$v \to v' = v - \eta \nabla C, \tag{4}$$

这里，η 称为学习速率 (learning rate)，而

$$\nabla C = \left(\frac{\partial C}{\partial v_i} \right), \tag{5}$$

它与 v 的维数一致。

式 (4) 的分量形式为

$$v_k \to v'_k = v_k - \eta \frac{\partial C}{\partial v_k} \tag{6}$$

如果把 w 和 b 分别单独拉成向量，那么对应的梯度下降公式为

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}, \qquad b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{7}$$

随机梯度下降法 (SGD)

记

$$C_x(w, b) = \frac{\|a(x) - y(x)\|^2}{2}, \tag{8}$$

则

$$C(w, b) = \frac{1}{n} \sum_x C_x, \qquad \nabla C = \frac{1}{n} \sum_x \nabla C_x. \tag{9}$$

这表明，我们要对每个训练输入 x 计算梯度值 ∇C_x，之后再平均。对输入样本很大时，梯度的计算将会花费大量时间，使得学习速率很慢。

在实际计算中，我们通常随机选取若干个输入，记为 $X_1, X_2, \cdots, X_m$，用它们的梯度平均代替原来的梯度平均：

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x \approx \frac{1}{m} \sum\nolimits_{j=1}^{m} \nabla C_{X_j}. \tag{10}$$

上面随机抽取的数据称为**小批量数据 (a mini-batch)**. 此时的更新公式为

$$w_k \to w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}, \tag{11}$$

$$b_l \to b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}. \tag{12}$$

抽取小批量数据

mini-batches 的个数为

```
n = size(training_data_x, 2);  % number of training data
batch_num = fix(n/mini_batch_size);     % number of mini-batches
```

注意，training_data_x 的每列对应一个训练数据。SGD 是随机抽取训练数据，即随机排列 training_data_x 的列。在 MATLAB 中，我们可用 `randperm(n)` 生成 `1:n` 的随机排列，然后根据 mini_batch_size 提取数据，如下

```
kk = randperm(n);  % for shuffling the training data
for s = 1:batch_num

    % current mini-batch
    id = kk((s-1)*mini_batch_size+1 : s*mini_batch_size);
    mini_batch_x = training_data_x(:,id);
    mini_batch_y = training_data_y(:,id);

end
```

根据 SGD 的思想，神经网络每次训练的样本实际上为 mini-batch, 因此在循环中产生当前的 mini-batch，然后继续后面的操作。

前向传播

再回忆一下前向传播。先随机初始化神经网络的参数

```
sizes = [784, 15, 10];  % number of neurons on three layers
%% Initialize weights and biases
w2 = randn(sizes([2,1])); % weights from 1-layer to 2-layer
w3 = randn(sizes([3,2]));
b2 = randn(sizes(2),1);   % biases on 2-layer
b3 = randn(sizes(3),1);
```

为了更加清楚，我们假设隐藏层只有一层，从而权重矩阵只有从第 1 层到第 2 层的 w^2 和第 2 层到第 3 层的 w^3. 类似偏置只有 b^2 和 b^3. 要特别注意权重矩阵的阶。

有了初始参数，我们就可以利用当前的 mini-batch 进行前向传播，即计算权重和偏置的更新值:

```
% current mini-batch
id = kk((s-1)*mini_batch_size+1 : s*mini_batch_size);
mini_batch_x = training_data_x(:,id);
mini_batch_y = training_data_y(:,id);

% feedforward
a1 = mini_batch_x;
z2 = w2*a1 + b2;
a2 = sigmoid(z2);
z3 = w3*a2 + b3;
a3 = sigmoid(z3);
```

上面计算出的是样本中所有 x 的结果。feedforward 的这几行语句就是进行网络预测（假设权重和偏置已经训练好）。激活函数取为 sigmoid 函数，它及其导数的定义如下

```
%% Define activation functions
sigmoid = @(z) 1./(1+exp(-z));
sigmoid_prime = @(z) sigmoid(z).*(1-sigmoid(z));
```

参数更新

前向传播需要给定权重和偏置，后面利用梯度下降法更新这些参数，即式 $(11)-(12)$. 在该公式中，关键的是计算每个样本数据对应的损失函数的梯度。这就是后面要介绍的反向传播算法。

梯度的计算：反向传播算法

用神经元误差表示梯度

现在我们来导出梯度 $\frac{\partial C}{\partial w}$ 和 $\frac{\partial C}{\partial b}$ 的表达式。这里考虑一般的形式 (3). 网络的前向传递过程如下

$$a^1 \xrightarrow[b^2]{w^2} z^2 \xrightarrow{\sigma} a^2 \xrightarrow[b^3]{w^3} z^3 \xrightarrow{\sigma} \cdots z^L \xrightarrow{\sigma} a^L, \tag{13}$$

其中，

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, \qquad l = 2, 3, \cdots, L, \tag{14}$$

$$C(w,b) = \frac{1}{n} \sum_x C_x(w,b), \qquad C_x(w,b) = \frac{1}{2} \|a^L(x) - y(x)\|^2. \tag{15}$$

显然，对 C_x 的求导，本质上归结于 $a^L(x) = a^L(x; w, b)$ 的求导.

注意到 w_{jk}^l 在带权输入 z_j^l 中，于是由求导的链式法则

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} =: \delta_j^l a_k^{l-1}, \tag{16}$$

这里的 δ_j^l 称为 l 层上神经元 j 的误差，即损失函数对带权输入的偏导数。类似地，我们有

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} =: \delta_j^l. \tag{17}$$

可以看到，**神经元上的误差恰好为损失函数对偏置的偏导数。**

神经元误差的反向传播

现在的问题又归结为求神经元上的误差 $\delta_j^l = \frac{\partial C}{\partial z_j^l}$. 下面将导出前后两层神经元误差的关系式。根据前向传递图 (13)， z^{l+1} 是 z^l 的函数，具体写出来就是

$$z^{l+1} = w^{l+1} a^l + b^{l+1} = w^{l+1} \sigma(z^l) + b^{l+1}, \tag{18}$$

或写为分量形式

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}. \tag{19}$$

使用链式法则，我们有

$$\begin{aligned} \delta_j^l = \frac{\partial C}{\partial z_j^l} &= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l) \\ &= \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \end{aligned} \tag{20}$$

该式可写成向量形式为

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{21}$$

其中的 $\odot$ 表示向量的点乘或 Hadamard 乘积。可以看到，**神经元的误差是反向传播的。**

式 (21) 给出了神经元误差的反向传播迭代式，为了计算所有神经元的误差，我们需要给定 L **层神经元误差的值** δ^L **(即输出神经元误差) 作为迭代的初始值。**注意到 $a_j^L = \sigma(z_j^L)$, 我们有

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L), \tag{22}$$

此即

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{23}$$

综上，我们获得损失函数关于权重和偏置的梯度计算表达式：

- 权重和偏置的梯度

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}, \qquad \frac{\partial C}{\partial b_j^l} = \delta_j^l, \tag{24}$$

- 神经元误差的反向传播迭代式

$$\begin{cases} \delta^l &= ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \quad l = L-1, \cdots, 2, \\ \delta^L &= \nabla_a C \odot \sigma'(z^L). \end{cases} \tag{25}$$

注意，损失函数关于权重的梯度 $\frac{\partial C}{\partial w}$ 是一个矩阵，显然有

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T. \tag{26}$$

反向传播算法

1. 输入数据个数为 m 的小批量训练样本 (mini-batch).
2. 对每个训练数据 x：设置输入激活值 $a^{x,1}$，并执行如下步骤
   - 前向传播：对 $l = 2, \cdots, L$，计算 $z^{x,l} = w^l a^{x,l-1} + b^l$ 和 $a^{x,l} = \sigma(z^{x,l})$.
   - 输出误差：计算 $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
   - 反向传播：对 $l = L-1, \cdots, 2$, 计算 $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.
3. 梯度下降：对 $l = L, \cdots, 2$, 如下更加权重和偏置

$$w^l \to w^l - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T, \qquad b^l \to b^l - \frac{\eta}{m}\sum_x \delta^{x,l}. \tag{27}$$

反向传播的 MATLAB 实现

小批量训练样本数据 mini_batch_x 和 mini_batch_y 已经获得，并且我们已经给出了前向传播的过程。

接下来，我们要计算输出层的神经元误差 $\delta^{x,L}$. 注意到

$$C_x(w,b) = \frac{\|a(x) - y(x)\|^2}{2} = \frac{1}{2}\sum_{i=1}^p (a_i - y_i)^2, \tag{28}$$

其中的 $p = 10$ 为输出神经元个数。显然有

$$\nabla_a C_x = (a_1 - y_1, \cdots, a_p - y_p)^T = a - y, \tag{29}$$

于是 $\delta^{x,L}$ 如下计算（$L=3$）

```
% errors of output neurons
cost_derivative = a3 - mini_batch_y;
delta3 = cost_derivative.*sigmoid_prime(z3);
```

反向传播后，前面层 (除去第一层) 的神经元误差为

```
% backpropagation
delta2 = (w3'*delta3).*sigmoid_prime(z2);
```

最后利用梯度下降更新权重和偏置

```
% gradient descent: update weights and biases
m = size(mini_batch_x,2);
for i = 1:m   % loops of mini-batch data
    w2 = w2 - eta/m*delta2(:,i)*a1(:,i)';
    b2 = b2 - eta/m*delta2(:,i);
    w3 = w3 - eta/m*delta3(:,i)*a2(:,i)';
    b3 = b3 - eta/m*delta3(:,i);
end
```

迭代

对当前 mini-batch 计算出权重和偏置后，我们就可把它们当作初值，继续对后面的 mini-batch 执行上面的过程。当所有的 mini-batch 处理完后，我们称完成了参数的一代更新或网络的一代训练。对一代训练给出的参数，我们也可以把它们当作初值，重复所有过程。

神经网络的 MATLAB 程序

前面的讨论总结为如下代码

```
% main_mnist_process.m
%% Load MNIST
% name_data_x, name_data_y, where name = training, validation, test
load mnistdata
% parameters
sizes = [784, 15, 10];  % number of neurons on three layers
epochs = 10;
mini_batch_size = 10;
eta = 3;  % learning rate
n = size(training_data_x, 2);  % number of training data
batch_num = fix(n/mini_batch_size);     % number of mini-batches

%% Define activation functions
sigmoid = @(z) 1./(1+exp(-z));
sigmoid_prime = @(z) sigmoid(z).*(1-sigmoid(z));

%% Initialize weights and biases
w2 = randn(sizes([2,1])); % weights from 1-layer to 2-layer
```

```
w3 = randn(sizes([3,2]));
b2 = randn(sizes(2),1);   % biases on 2-layer
b3 = randn(sizes(3),1);

%% Train network with SGD
for ep = 1:epochs
    kk = randperm(n);  % for shuffling the training data
    for s = 1:batch_num

        % current mini-batch
        id = kk((s-1)*mini_batch_size+1 : s*mini_batch_size);
        mini_batch_x = training_data_x(:,id);
        mini_batch_y = training_data_y(:,id);

        % feedforward
        a1 = mini_batch_x;
        z2 = w2*a1 + b2;
        a2 = sigmoid(z2);
        z3 = w3*a2 + b3;
        a3 = sigmoid(z3);

        % errors of output neurons
        cost_derivative = a3 - mini_batch_y;
        delta3 = cost_derivative.*sigmoid_prime(z3);

        % backpropagation
        delta2 = (w3'*delta3).*sigmoid_prime(z2);

        % gradient descent: update weights and biases
        m = size(mini_batch_x,2);
        for i = 1:m   % loops of mini-batch data
            w2 = w2 - eta/m*delta2(:,i)*a1(:,i)';
            b2 = b2 - eta/m*delta2(:,i);
            w3 = w3 - eta/m*delta3(:,i)*a2(:,i)';
            b3 = b3 - eta/m*delta3(:,i);
        end

    end

    % evaluation of test_data
    a1 = test_data_x;
    z2 = w2*a1 + b2;
    a2 = sigmoid(z2);
    z3 = w3*a2 + b3;
    a3 = sigmoid(z3);
    [~,y_p] = max(a3,[],1);
    [~,y] = max(test_data_y,[],1);
    y_p = y_p'-1; y = y'-1;
    fprintf('Epoch %2d :   %d / %d \n', ep, sum(y_p==y), length(y));
end
```

在该程序中，每当完成一代训练，我们就对测试数据进行预测，并和真实值比较。程序运行过程中的输出结果如下（每次结果不同）：

```
Epoch  1 :   8811 / 10000
Epoch  2 :   8987 / 10000
Epoch  3 :   9161 / 10000
Epoch  4 :   9134 / 10000
Epoch  5 :   9198 / 10000
Epoch  6 :   9215 / 10000
Epoch  7 :   9210 / 10000
Epoch  8 :   9230 / 10000
Epoch  9 :   9253 / 10000
Epoch 10 :   9218 / 10000
```

可以看到，准确率达到 90%.

现在，若给定某些输入数据，你就可以在前面的代码后面添加前向传播的过程。例如，validation_data 的预测程序为

```
%% Recognize handwritten digits
a1 = validation_data_x;
z2 = w2*a1 + b2;
a2 = sigmoid(z2);
z3 = w3*a2 + b3;
a3 = sigmoid(z3);
[~,y_p] = max(a3,[],1);
[~,y] = max(validation_data_y,[],1);
y_p = y_p'-1; y = y'-1;
ratio = sum(y_p==y)/length(y);
fprintf('\n Recognize handwritten digits in validation_data \n');
fprintf(' Accuaracy = %.2f%% \n', ratio*100);
```

准确率为 91.91%.

程序扩展

隐藏层添加更多层的改动是直白的，后面编写面向对象的程序实现该功能。 Python 中习惯用面向对象的形式组织程序。