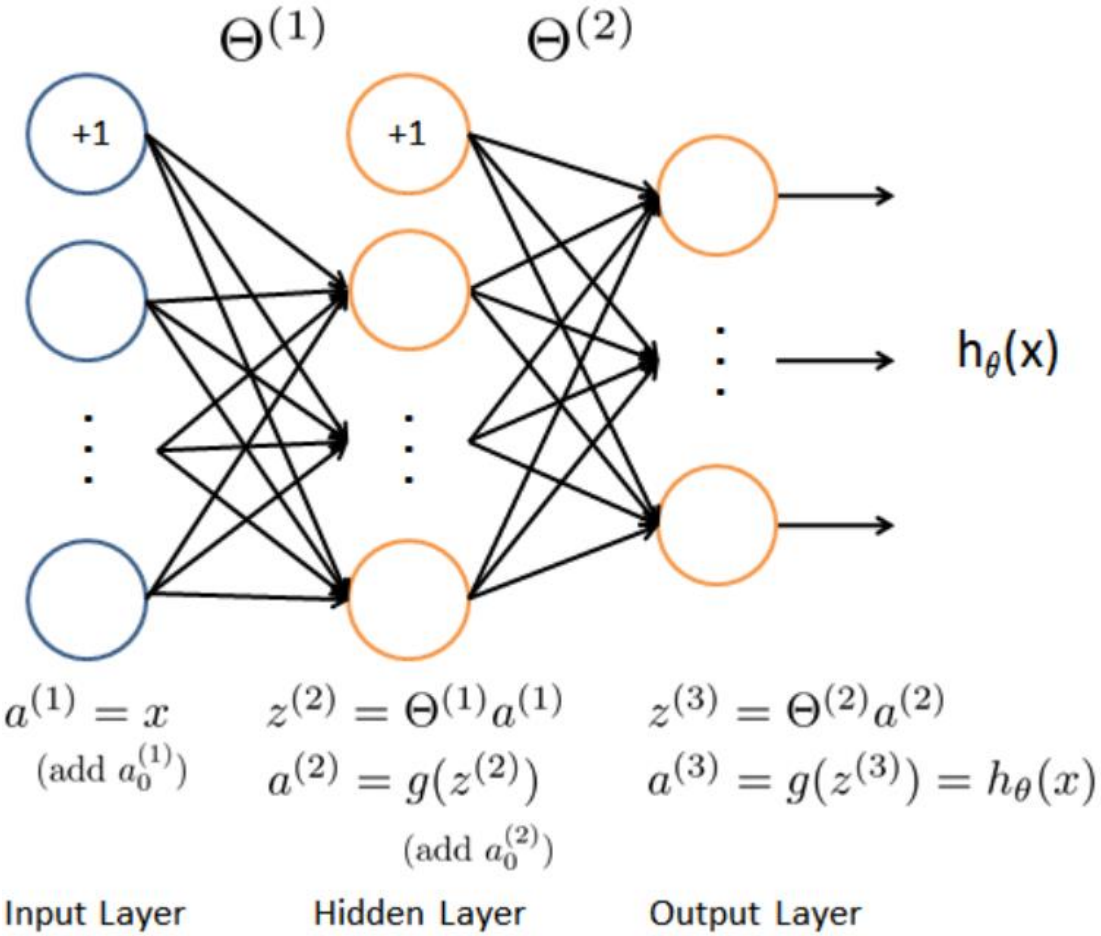
BP 说明文档

一、网络结构



1. 从左到右依次是输入层，隐藏层，输出层共三层的网络。向前传递的计算公式如图中所示。其中 Theta 表示层层之间的参数矩阵（包含了 bias），a 是每一层向下一层输入向量，z 表示每一层从上一层得到的输出向量，g 是激活函数。$h_\theta(x)$为最终的输出向量。
2. 图中标有+1 的节点表示 bias 节点，该节点值恒为 1，这样参数 Theta 中由该节点延伸出的线上的权重就是 bias，只是使得计算公式更统一一点。
3. 每层激活函数 g 选取为 sigmoid 函数。

二、参数调整算法

1. Loss function

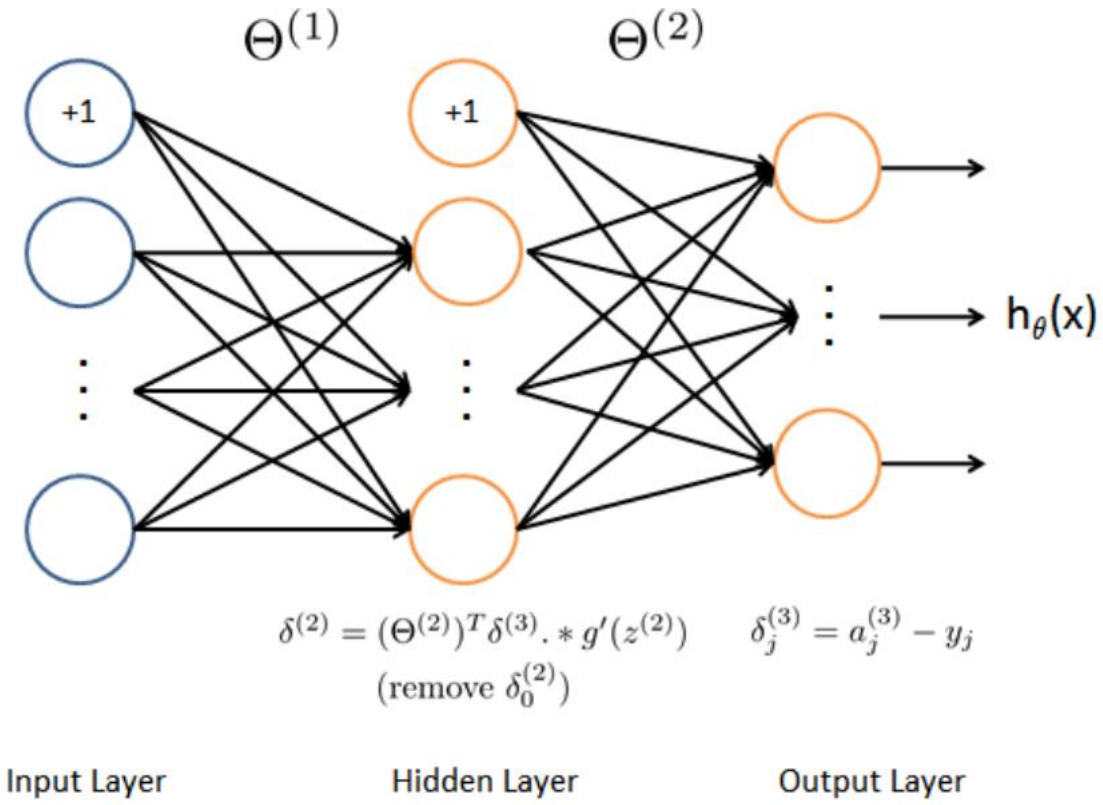
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

Loss function 的前一部分是误差项，后一部分是正则化项。

误差项使用的是 cross entropy。然后这是一个由二分类问题扩展而来的 loss function。即对于一个多分类问题，当成是每一个节点的二分类问题。每一个节点只需考虑自己，如果正确答案是自己，就向着使得自己变大的方向调整，反之向着使得自己变小的方向调整。这与 softmax 相比，它的导数形式与 softmax 是一致的，只是上述 $h_\theta(x)$的值由 sigmoid 得来而非 softmax，少考虑了输出节点之间的相互影响，但是其要快一点。

添加正则化项是为了防止过拟合。

2. 利用 back propagation 更新参数。



1) loss function 中误差项和正则化项是分开的，这极大方便了我们的计算。这里先计算误差项的梯度。
2) 图中 $\delta_j^{(l)}$ 表示第 l 层，第 j 各 unit 的误差。
3) 按层更新参数的方法网上有很多，下图是来自 coursera 上一门机器学习课程提供的公式。需要注意的是，我们前面应用了一些方法使得 bias 和权重参数 Theta 混在一起了。所以在更新的时候需要特别小心的注意到它

不能影响到第一层与第二层之间的参数。

Figure

2. For each output unit k in layer 3 (the output layer), set

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

where $y_k \in \{0, 1\}$ indicates whether the current training example belongs to class k ($y_k = 1$), or if it belongs to a different class ($y_k = 0$). You may find logical arrays helpful for this task (explained in the previous programming exercise).

3. For the hidden layer $l = 2$, set

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$. In Octave/MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta_2 = delta_2(2:end)`.

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$

5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

之后再加上正则化项的梯度，如下：

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \qquad \text{for } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \qquad \text{for } j \geq 1$$

注意 j=0 那一列是 bias，不包含在正则化项里的。

3. Mini-batch 方法

对于大的样本，如果每次都将所有样本的误差都累积取平均数再更新，那么更新速度非常慢。所以采用 mini-batch 方法，大小选取合适使得更新速度变快，而且方向也很准确。

三、LCDD 训练报告

1. 参数选择说明

1) 正则化的 λ 值一直是设为 0 的，因为该训练的训练集已经包含了测试集所有可能的情况，所以没必要防止过拟合。

2) mini batch 的 batch size 是设为 50 的，但是训练集的大小只有 10，所以每次请求 next batch 的时候返回是 10 个。这里 mini-batch 并没有发挥作用，也不需要发挥作用。

3) 隐藏层的大小，这个重头到尾都没有改过，一直是 18，因为老师上课用的就是 18。而且该网络很容易就收敛了，而且测试结果也非常好。也没必要去改了。

4) input layer size 是固定的，output layer size 选择了 one hot 模式表示输出，所以一共 10 种输出，output layer size 就是 10 了。

5) 所以最终在做这个训练的时候主要是在调整 learning rate α。当 cost function 一直减小到一定程度，就开始跳动变化，不再减小。此时会记录这个值。下次训练的时候让 α 在此值附近变小。所以最终如下图所示。

```
@Override
public double changeRate(double cost, double oldRate){
    if (cost < 0.0001) {// origin: 0.0175
        return -1; //表示结束训练
    }
    if (cost < 0.02) {
        return 0.0001;
    } else if (cost < 0.05) {
        return 0.001;
    } else if (cost < 0.1) {
        return 0.01;
    } else if (cost < 0.15) {
        return 0.1;
    }
    return oldRate;
}
```

最后训练到小于 0.0175 时就很难减小了。如下所示

```
Iteration:  49989|Cost: 0.017472384789086085|rate:1.0E-4
Iteration:  49990|Cost: 0.017472384729909623|rate:1.0E-4
Iteration:  49991|Cost: 0.017472384670733018|rate:1.0E-4
Iteration:  49992|Cost: 0.017472384611556268|rate:1.0E-4
Iteration:  49993|Cost: 0.017472384552379493|rate:1.0E-4
Iteration:  49994|Cost: 0.017472384493202837|rate:1.0E-4
Iteration:  49995|Cost: 0.017472384434026062|rate:1.0E-4
Iteration:  49996|Cost: 0.01747238437484931|rate: 1.0E-4
Iteration:  49997|Cost: 0.017472384315672763|rate:1.0E-4
Iteration:  49998|Cost: 0.017472384256496023|rate:1.0E-4
```

```
Iteration:  49999|Cost: 0.017472384197319408|rate:1.0E-4
test:0123456789
0123456789
```

但是此时测试结果已经很好了，也没必要纠结下去。

最终我们看一下，训练得到的参数中的一部分：

```
54.430531099081655 150.79814752146765 -213.7513358700602
-164.88143219752644 65.62336593348714 39.51853580284915 -
163.78693213817337 54.21726589379872
```

由这些参数分析，都是远离 0 的数，我们的按理是十分过拟合了。但是正如第一条所说，我们这样做是可以的。

四、sin 函数拟合训练。

1. 基本情况说明

这个我原来是想着把输入的数字近似用二进制表示，然后输出二进制表示，再转换成数字输出的，但是一直不能得到好的结果（其实在转换过程中已有很多近似了）。后来想到也许一个输入和一个输入就可以。于是尝试了一下，结果有所改进，但还是不佳。

究其原因，上述 loss function 在只有一个输出时就是一个二分类结果了。而这时后使用 cross entropy，实际上这个 loss function 会使得结果大于 0.5 的向 1 靠拢，小于 0.5 的向 0 靠拢。而我最后测试评估的时候，是使用平方差来衡量的，这结果也可想而知。

最后，用平常差来衡量测试结果误差肯定是好的模型。所以以上 loss function 也要改成平方差型的了。

此时，反向更新参数时 $\delta_k^{(l)}$ 就要改变了。其变为：

$$\delta_k^{(3)} = (a_k^{(3)} - y_k) \cdot a_k^{(3)}(1 - a_k^{(3)})$$

在一开始设计时，也没有为此改动留有好的设计，就如下实现

Loss function 如下所示，需要用什么，就注释掉另一个（-.-）

```
private double getOneExampleCost(double[] desiredOutput){
      double err = 0;
      for (int j = 0; j < outputLayerSize; j++) {
         // get error
//       double error = desiredOutput[j] - output[j];
//       err += error*error;
         double y = desiredOutput[j];
         double o = output[j];
         err += -y*Math.log(o)-(1-y)*Math.log(1-o);
      }
      return err;
}
```

反向更新参数时，根据上面的公式，我们高兴地发现只是多乘了一项，如下所示，想要使用 square error 时，只要加上//delta3 =… ；这一行即可。

```
 */
private void backPropagation(double[][] deltaTheta1, double[][] deltaTheta2, doub
    double[] delta3 = vecSubVec(output, desiredOutput);
    //delta3 = dotMulti(delta3, dotMulti(output, numSubVec(1, output)));//for squ

    //delta2 = Theta2_O'*delta3.*sigmoidGradient(z2);
    double[][] Theta2_1 = rmFirstColumn(Theta2);
    double[] delta2 = dotMulti(matrixMultiVec(transpose(Theta2_1), delta3),
            dotMulti(hidden, numSubVec(1, hidden)));
```

此外，就不需要改动了。（-.-）

2. 输入处理

根据上述说明，我们发现并没有改变激活函数 sigmoid，但是这个函数是不能产生负值的。根据这个特殊的 sin 函数，我没有选择修改激活函数，因为这样太麻烦了。上面对网络结构的修改已经很糟糕了。所以根据 sin 函数的性质，对于所有的负样本，都将其取负值，再作为输入，然后对输出再取个负号，注意这是在网络之外的改动，不要影响原来网络的更新。

3. 参数选择说明

1) 根据上述说明，我们选择了 output layer size 和 input layer size 都为 1.

2) 样本选取区间都给定了。所以选样本量当然是选的多点好了。训练的时候在区间内均分选了 1500 个点作为样本。

3) 根据上面样本选取的密集度，根本就不害怕过拟合了，所以 λ 只是象征性的选了一个非常小的值 0.0000000001

4) 隐藏层的大小为 20，试过 20-200 的隐藏层个数，但是影响不大。最终选了 20 个隐藏层。为了追求速度。

5) 训练的时候，mini-batch 的 size 就是 1500。因为这个网络并不大，所以这样也可以忍。但是如果不这样做的话，因为我是时时看着误差来决定网络的 learning rate，这样会有点影响 learning rate 的选取。因为上一个 batch 的误差与下一个 batch 的误差是不同的，可能正好有一个 batch 误差很小，就停止更新了。所以每次都 focus 在所有的样本上，使得其误差尽可能小就可以了。

6) 所以最终还是 focus 在 learning rate α 上。当 cost function 一直减小到一定程度，就开始跳动变化，不再减小。此时会记录这个值。下次训练的时候让 α 在此值附近变小。所以最终如下图所示。

```
@Override
public double changeRate(double cost, double oldRate){
    if (cost < 0.0000719503) {
        return -1;
    }
    else if (cost < 0.00007195033) {
        return 0.00001;
    }
    else if (cost < 0.000072) {
        return 0.001;
    }
    return oldRate;
}
```

最终 cost 小于 0.00007195033 时，就很难更新了，α 也非常小了。如下

```
Iteration:  9995|Cost:  7.195031219812563E-5|rate:1.0E-5
Iteration:  9996|Cost:  7.19503121980153E-5|rate:  1.0E-5
Iteration:  9997|Cost:  7.195031219790502E-5|rate:1.0E-5
Iteration:  9998|Cost:  7.1950312197795E-5|rate:  1.0E-5
Iteration:  9999|Cost:  7.19503121976849E-5|rate:  1.0E-5
0.7812429672031014/0.7835614085197449
-0.26820521531927854/-0.26617178320884705
-0.9915709507418619/-0.9985857605934143
0.503382590869247/0.5009148716926575
…
Average cost: 3.894464003736861E-5
```

在随机的 200 个测试样本上有-5 次方级别的平均平方误差，本来想训到-6 次方级别，但是达不到。

五、BP 网图片识别训练

1. 基本情况说明

与前面两个网络不同的是，这里数据集非常大，所以 mini-batch 就非常实用了，但是为了此效果，就不得不舍弃掉前面那种精准的根据 loss function 的大小改变 learning rate α 的大小的方法。因为对每一个 batch，其误差值是不同的。所以修改α的策略要有所改变。还有图片测试有一个验证集，防止过拟合就十分重要了。所以参数λ也是个重要的调整对象。还有这个又是一个分类问题，所以 loss function 也应当改回来了，前面对改动的两点已经加一说明了。

2. 输入处理

为了使得 BP 网能够训练二维的图片，所以决定将二维 28*28 的图片转换为一维 784*1 的向量作为输入。根据经验，我觉得输入在 0-1 之间会有较好的结果。所以会将 0-255 的输入值除以 255.0 再作为网络的输入。再有就是需要将输入的图片打乱顺序后再作为输入。

3. 参数选择说明

1) 根据上面的说明 input layer size 设为 784，output layer size 设为 8。

2) 隐藏层的大小设置过 25-128，维度变大后，网络速度迅速减慢，并且也没有训练出明显较好的结果，最后主要 focus 在隐藏层大小为 25 的训练上。

3) 与前面两个实验不同的是，这里参数λ的设置会影响结果。但主要影响在于λ过大会欠拟合，λ小的时候，影响不是很大，也许是网络太小的原因吧。

λ=0.1

```
train set accuracy: 56.22107969151671
validation set accuracy: 54.8125
```

λ=0.0

```
train set accuracy: 89.6401028277635
validation set accuracy: 85.89375
```

λ=0.01

train set accuracy: 88.34190231362467

validation set accuracy: 85.78125

4) 前面已经说过本次训练样本较大，所以要用mini-batch方法来加快训练速度。Batch size设置的是50。但是此时对每一个batch，其误差值是不同的，就不得不舍弃掉前面那种精准的根据loss function的大小改变learning rate α的大小的方法。基本上从一个α的训练结果中选取一个好的结果，然后选一个更小的α在此结果上接着训练。最终有一个较好的结果。

λ = 0.01

train set accuracy: 94.96143958868895

validation set accuracy: 90.125