

神经网络学习报告 (9.05-9.30)

高明琦 (PhD Applicant)

Email: im.mingqi@gmail.com, Web: <https://mingqigao.com/>

本次报告主要包括两个部分：斯坦福的 CS231n 课程学习以及全连接神经网络的实现。学习方式为观看[课程视频](#)，并结合相应的[课程笔记](#)。经过这几周的学习，我已经完成了神经网络部分的课程内容（笔记中的 Module 1 部分），并根据已掌握的知识对全连接神经网络做了实现。代码使用 Python 3.5 编写，数值计算通过 numpy 包完成。关于神经网络的定义借鉴了开源框架 PyTorch，但实现过程并没有使用任何开源框架。项目代码可见于我的 GitHub: <https://github.com/gaomingqi/NNforMNIST>。下面将具体介绍我在学习过程中所掌握的知识与体会。

目录

1	CS231n 课程学习	1
1.1	数据驱动的图片分类	2
1.2	线性分类器损失函数与最优化	3
1.3	反向传播与神经网络初步	5
1.4	神经网络训练细节	8
2	全连接神经网络实现	12
2.1	网络结构定义	13
2.2	实现细节	13
2.3	实验结果分析	17
2.4	总结	17

1. CS231n 课程学习

CS231n 的全称是 CS231n: Convolutional Neural Networks for Visual Recognition，即面向视觉识别的卷积神经网络，是斯坦福大学计算机视觉实验室推出的课程。随着计算机视觉技术在社会中的逐渐普及，其在信息检索、图像理解与自动驾驶等领域都得到了广泛应用。而这些应用的核心就是图像分类、定位、探测等视觉识别任务，因此我选择该课程作为理解深度学习、神经网络在计算机视觉领域应用的入门课程。接下来我会通过图像分类、线性分类器、反向传播、神经网络训练这四个部分介绍我的学习内容。

1.1. 数据驱动的形象分类

图像分类的任务，就是对一个给定图像以及已知的标签集合，预测其应属于哪个分类标签，或者给出其属于每个不同标签的可能性。对计算机来说，图像通常是以 3 维数组来表示的，数组元素的取值范围为 $[0, 255]$ 之间的整数，尺寸为 $W \times H \times 3$ ，其中 W 、 H 分别表示图像的宽、高，3 表示红、绿、蓝 3 个颜色通道。

虽然对人来说，在图像中进行视觉识别是非常简单的，但从计算机的角度来看就会遇到很多困难。因为描述相同目标的图像矩阵往往会因为视角变化、大小变化、形变、遮挡、光照条件、背景干扰等因素都具有很大的类内差别，因此对分类模型的稳定性要求较高。课程中介绍的分类模型是数据驱动的方法，即给计算机很多数据，然后实现学习算法，让计算机学习到每一类目标的特征。

课程首先给出的是最近邻分类器，最简单的分类器之一。对于图像分类，该方法会对逐像素地比较给定图像与每个训练图像，并统计相应的差异值之和。在比较之后，选择与给定图像差异值最小的训练图像的标签作为分类结果。常用的差异值定义是 d_1 距离与 d_2 距离，定义如下：

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|, \quad d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2} \quad (1)$$

其中， I_1 与 I_2 分别表示两个待比较图像的向量化表示， I_1^p 与 I_2^p 为这两个向量第 p 个元素的值。但这种方法在训练数据集出现异常值的情况下很容易分类错误，因此 K-近邻分类器被提出用于解决这个问题。该分类器会统计与给定图像最相近的 K 个训练图像的标签，并据此进行标签投票，选择数量最多的标签作为分类结果。虽然与最近邻相比，K-近邻分类器能获得比较鲁棒的分类结果，但在实际使用中效率不高。因为在分类过程中，需要存储所有的训练数据与标签用于统计差异值，这会极大的增加测试时时间与空间的需求。另外，逐像素值的比较对图像信息的使用不充分，仅获取了背景与颜色信息，忽略了更深层的语义特征，在存在干扰的情况下很难获得很好的效果，如光照变化、遮挡等，这些因素都会使图像的像素值发生改变，从而导致同类图像间的差异值增加。

在分类模型的定义过程中，会涉及到一些参数或方法的选择，如 K-近邻分类器中 K 的选择，或者 d_1 、 d_2 距离的选择等。这些选择被统称为超参数，课程中介绍了如何选择合适的超参数。即从训练集中提取一小部分数据作为验证集，在验证集上面选择表现最好的超参数进行训练与测试过程。而在训练集数量较小的情况下，课程介绍了交叉验证方法以补偿过小的验证集导致超参数设置不理想的问题。这种方法将训练集划分为多个子集合，除 1 个子集合用于验证外，其余均用于训练。然后通过轮流将其他子集合设置为验证集的方式进行多次验证，取综合表现最好的超参数作为验证结果。

总结: 通过本次学习我了解了图像分类的目的与意义, 并对简单的最近邻分类器、K-近邻分类器及其在应用领域的局限性有了深入的认识。虽然其算法很简便, 也有一定的效果, 但测试时的效率过低, 且对图像信息的挖掘不深入, 因此在实际的图像分类问题中基本不会被采用。此外, 我还学习到了设置超参数的方法, 以确保在训练与测试过程中分类模型的表现最优。

1.2. 线性分类器损失函数与最优化

在上一节介绍基于近邻的分类方法中, 我们了解到其测试过程需要存储全部的训练数据与标签, 且需要通过逐一对比才能得到分类结果。这种方式会在测试时造成极大的时间与空间消耗, 而在实际应用中, 对分类模型的实时性需求往往是很高的。因此课程在本节中介绍了一种性能更好、效率更高的方法来解决图像的分类问题, 该方法的设计思路可延伸到神经网络与卷积神经网络上。

这种方法即为线性分类。设图像矩阵被拉伸至一个 D 维向量 $x_i \in R^D$, 其中 $i = 1, 2, \dots, N$, N 表示训练数据集中图像的总数。对应着图像数据, 有分类标签 $y_i \in 1, 2, \dots, K$, K 为不同分类的数量。这样的话, 我们就能定义一个包含 N 个图像、 K 种类别的训练数据集。在线性分类器中, 定义了两个参数: 权重矩阵 W 与偏移量 b , 将图像数据 x_i 映射到一个长度为 K 的列向量。该映射定义为:

$$f(x_i, W, b) = Wx_i + b \quad (2)$$

其中 W 是一个大小为 $[K, D]$ 的矩阵, 被称为权重。 b 是一个大小为 $[K, 1]$ 的向量, 被称为偏移量, 因为该向量影响着输出值, 但不与图像数据产生关联。而输出值 $f(x_i, W, b)$ 是一个大小为 $[K, 1]$ 的向量, 被称为图像 x_i 在每一类上的得分。根据课程笔记中给出的图片, 可以更加深入的理解该映射的作用, 如图 1 所示。

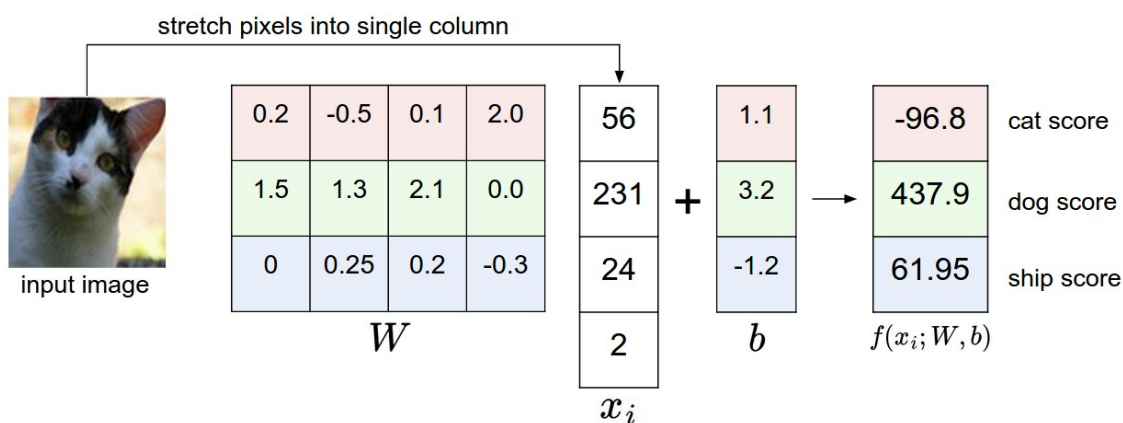


Figure 1. 线性分类器的图形解释.

从图中可以看到, 权重矩阵 W 的每一行都相当于一类目标的分类器。如对类别 dog, 其得分定义为 W 第二行的行向量与图像数据 x_i 的内积加上偏移量。而内积可以理解为对不同位置的不同像素值, 分类器会表现出喜好 (正值) 或者厌恶

(负值)。需要注意的是线性分类器的权重矩阵 W 与偏移量 b 是可学习的参数，在训练过程中，他们的值会不断被优化从而使正确类别的得分更高。这样的话，该方法在测试过程中仅需要保存这两个参数，并进行一次线性运算即可得到分类结果，因此在存储空间与运行时间上均要优于最近邻或 K-近邻的分类器。

虽然上述的分类器能够得到图像在每个类别上的得分，但难以衡量得分的好坏。因此课程引入了损失函数来表示对分类结果的不满意程度。即分类结果与真实结果差别越大，则损失函数越大，反正则越小。在课程中，Instructor 主要介绍了两种损失函数：多类 SVM 损失以及 Softmax 损失。

多类 SVM 损失. 首先给出该损失函数的定义，设 $s_j = f(x_i, W)_j$ 表示对第 j 个类别的得分，其中 $j = 1, 2, \dots, K$ 。对第 i 个数据，其多类 SVM 损失函数定义如下：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \quad (3)$$

其中 s_{y_i} 表示正确分类的得分， Δ 可理解为一个阈值，即只有正确分类得分比其他分类得分都大，且大于 Δ 时，损失函数才为 0，达到损失函数的最小值。因此多类 SVM 损失函数能鼓励分类器为正确类别计算出最大的得分，且得分还要比其他得分高出一定数值。

Softmax 损失. 在 SVM 损失函数中，映射 $f(x_i, W)$ 的输出被视为是每个类别的得分。但该评分对每个类别来说，其取值范围可能都是不同的，分析起来并不直观。而 Softmax 的得分是归一化的分类概率，定义为： $s_j = e^{f_{y_i}} / \sum_j e^{f_j}$ ，可以发现其得分被转化为了取值 0 到 1 之间的对数概率。相应的损失函数定义如下：

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \quad (4)$$

因此 Softmax 损失函数能鼓励分类器为正确类别计算出尽可能最大的对数概率。这一点与 SVM 不同，SVM 的关注点似乎只在于边界条件，即只要实现类别的有效区分就可以了，Softmax 则是尽量最大化正确类别的可能性。

虽然通过优化过程能得到一组使损失函数最小化的权重 W ，但这个 W 并不是唯一的，可能存在很多 W 都具有相同的效果。如对 W 进行 $\lambda > 1$ 的等比放大后的权重矩阵，并不会使损失值发生改变。因此需要一种方法能够为权重添加一些偏好，以得到唯一的权重矩阵。课程在这里介绍了正则化惩罚项 $R(W)$ ，最常用的方式是使用 L2 范式计算每个元素的平方和： $R(W) = \sum_k \sum_d W_{k,d}^2$ 。因此最终的损失函数通常是以如下形式定义的：

$$Loss = \frac{1}{N} \sum_{i=1}^N L(f(x_i, W)) + \lambda R(W) \quad (5)$$

因此分类损失函数与权重正则化会在优化过程中形成竞争，即在降低损失的

同时也减少 W 的值。在课程中 Instructor 说到正则项会使 W 的值越来越发散，而不是仅在几个位置具有较大的值。这样的话分类器就可以利用尽可能所有的输入信息，使更多的 evidence 逐渐积累从而进一步提升分类效果。

最优化. 接下来是有关优化的内容，由于本报告的分类是基于视频课程部分划分的，所以这里将列出 Instructor 紧接着线性分类器内容讲授的优化技巧。在优化过程中，可以将 W 看做一个多维空间的变量，在给定其初始值、公式 (5) 的定义与训练数据集 $\{x_i\}$ ，标签集 $\{y_i\}$ 后，如何寻找一个合适的 W 使损失函数达到最小。Instructor 首先介绍了随机搜索，即通过随机采样 W 的值来求解优化问题。接下来介绍的是梯度下降法，即计算当前 W 的梯度，通过乘一个负的步长来对当前 W 进行改变（因为只有梯度为负时 Loss 才会减少）。通过不断的迭代过程， W 就可以一步一步逐渐达到使 Loss 最小的值。关于求导过程，一维函数的导数定义如下：

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (6)$$

若对向量求导，其导数维度与原向量相同，每个维度的值对应着原向量同一维度的导数。因此若使用数值方式来计算梯度，就需要遍历每一个维度，然后使用公式 (6) 计算梯度值（很费时间，但在梯度检验时很有用，因为这种方法计算的梯度相对更加准确）。在现实中微积分是唯一计算梯度的方法。

Mini-batch 梯度下降. 虽然我们有完整的训练数据集，但实际的优化过程通常只选择其中一批样本数据来估算损失函数的梯度。因为这种方法的效率很高，能够计算多次梯度，同时也减少了 GPU 的负担。但由于使用了小批量采样，损失函数值的变化趋势可能是震荡着下降的，因为不同取样批所得到的梯度更新不同。不过这并不影响损失函数整体的下降趋势，因此 Mini-batch 采样对于实际的训练与测试过程是很实用且有效的方法。

总结. 经过本次学习，我了解了如何将图像分类问题定义为一个损失函数的最小化问题。主要分为三个步骤：首先，定义一个线性分类器，给定输入数据，获取每个类别的得分；其次，根据类别得分与给定的类别标签，定义损失函数；最后，通过梯度下降或者其他优化方法，更新参数使损失函数达到最小值。此外，我还掌握了两种常用的损失函数的定义及其特点，以及梯度下降法与 mini-batch 优化的具体操作方式。

1.3. 反向传播与神经网络初步

经过前面章节的学习，我了解到线性分类器的优化是能够通过梯度下降实现的。由于在后续的神经网络中，其损失函数通常是由多个可学习参数与训练数据构成的复合表达式，因此如何计算该式关于可学习参数的梯度，成为了优化过程的核心问题。课程在这里给出了反向传播方法，该方法能够利用链式法则递归地

计算表达式的梯度。

为便于理解反向传播，首先给出简单表达式的梯度计算。在损失函数中，常见的表达式有乘法、加法以及 $\max(x, y)$ 等。关于乘法，其梯度计算如下：

$$f(x, y) = xy \rightarrow \frac{df}{dx} = y, \quad \frac{df}{dy} = x \quad (7)$$

这里要介绍下梯度的意义：是当函数变量在某个值周围的极小区域变化时，所导致的函数值在变化方向上的变化率。因此，在上述梯度计算中，若 x 的导数大于 0，则说明如果将 x 的值变大一点，则整个表达式的值就会变大，反之，则会变小，且变化的量为 x 变化量的 $|\frac{df}{dx}|$ 倍。因此，函数关于每个变量的梯度表示了整个表达式对该变量的敏感程度。关于加法，其梯度计算如下：

$$f(x, y) = x + y \rightarrow \frac{df}{dx} = 1, \quad \frac{df}{dy} = 1 \quad (8)$$

上式说明了 x, y 的梯度在任何情况下都为 1，因此函数的变化率与函数变量的值是独立的。关于 $\max(x, y)$ ，其梯度表示为：

$$f(x, y) = \max(x, y) \rightarrow \frac{df}{dx} = 1(x \geq y), \quad \frac{df}{dy} = 1(y \geq x) \quad (9)$$

上式说明了如果变量 x 比 y 大，则其梯度为 1，反之为 0。这说明了函数在这里只对值大的变量敏感，值小的变量对函数输出没有效果。**注意**：关于 $x = y$ ，Instructor 说这种情况在实际操作中不大可能发生，但如果出现的话，随便选一个变量梯度为 1 就可以。

复杂表达式梯度：在给出简单表达式的梯度与解释后，这里使用一个例子来说明如何使用链式法则计算复杂表达式的梯度。课程笔记中给出了这样一个式子：

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \quad (10)$$

报告中给出了上述公式的计算流程图，如图-5所示。这里我按照笔记提供的图片做了一些修改，便于理解链式法则的流程。从图中可以看到公式 (10) 被拆分成多个简单表达式的组合，输入的变量值（绿色数字）在经过表达式层层递进后，即可得到最终的计算结果 $f(w, x)$ （最右节点）。

上述的计算过程即为前向传播，为得到函数 f 关于 w, x 的梯度，将计算方向反过来即可，即反向传播，不过此时的输入变成了 1（输出函数对本身求导）。反向传播的本质上和数学上求复杂函数导数的方法是相同的，如计算表达式 $f(x, y, z) =$

$(x^2 + y) \times z$ 关于 x, y 的梯度, 就需要先计算 f 关于 $(x^2 + y)$ 的梯度, 然后再计算 $(x^2 + y)$ 分别对 x, y 的梯度。

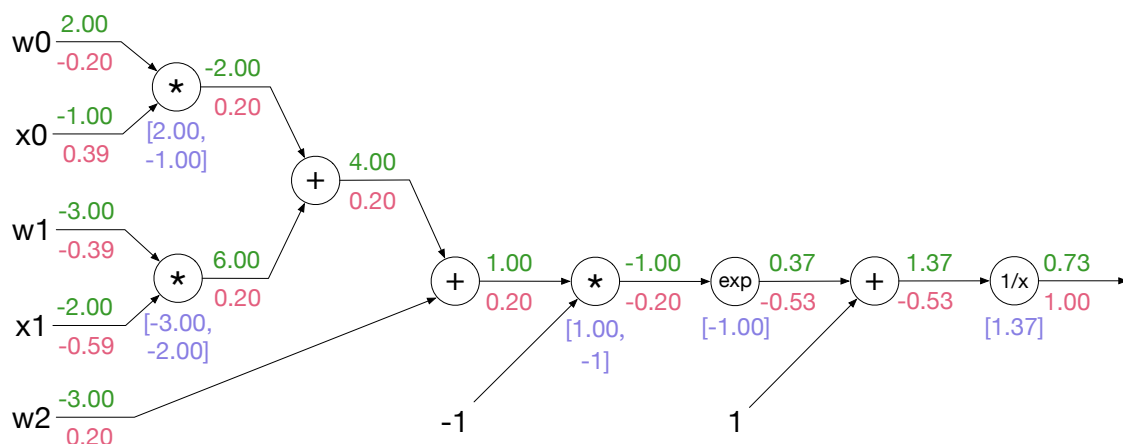


Figure 2. 链式法则计算流程图.

对每个简单表达式, 图中的红色数字表示反向传播过程中的输入变量。但其内部进行的运算并不是表达式本身描述的运算, 而是 [表达式本身的梯度] 乘 [上一个层表达式的梯度]。举个例子, 在单元 \exp 中, 进行的运算就是 $d(\exp(x))/dx \times (-0.53)$ 。这里就可以发现, 除了输入值之外, 计算还需要知道在前向传播时的输入变量值, 即 $x(-1.00)$ 。因此, 在前向传播过程中, 需要记录当时的输入变量 (这里使用紫色数字表示, 因为加法的梯度与输入无关, 因此不需要记录)。在已知 $f(w, x)$ 梯度本身 (1) 后, 同样经过层层递进, 就可以得到关于每个变量的梯度。

上述的内容是关于单个变量的梯度计算方法, 其所有的概念都适用于矩阵与向量操作。但在实际操作时需要注意维度和转置, 确保得到的梯度与原有矩阵具有相同维度即可。举个例子, 设有矩阵 $X, Z \in R^{a,b}$, $Y \in R^{b,c}$, $\max(X, Z)$ 关于 X 的梯度就是逐元素的进行梯度运算。但 $d(X \times Y)/dX$ 就需要考虑维度问题, 因为 $(X \times Y) \in R^{a,c}$, 因此上一层表达式的梯度也是维度为 $[a, c]$ 的矩阵, 为了保证对 X 的梯度维度与 X 本身相同, $d(X \times Y)/dX$ 应等于: 上一层梯度 $\times Y^T$ 。

神经网络初步. 截止目前, 我已经掌握了将图像分类转化到线性分类问题的方法, 同时也掌握了使用梯度最小化获取最优参数, 使损失函数最小化的方法。为进一步提升复杂问题的建模能力, 课程在线性分类器的基础上提出了神经网络方法。其本质可视为多个线性分类器的非线性组合, 如一个 2 层神经网络的评分函数:

$$s = W_2 \max(0, W_1 x) \quad (11)$$

根据上式可以看到, 从前的方法是输入数据在经过线性映射后直接得到每个类别的评分, 而现在则需要多经过 1 次隐含层的映射。举个例子, 若输入数据维度为 $[3072, 1]$, 隐含层参数 W_1 可能为 $[100, 3072]$, 输出层参数 W_2 可能为 $[10, 100]$ 。

Instructor 在课程中解释过隐含层的作用：可以理解为识别具有不同 appearance 但属于相同类别的样本数据（如不同朝向的汽车），其输出值只有满足这些条件后才会为正（如只识别正面），输出层则对这些不同情况下的样本数据进行汇总。因此神经网络方法可以适应目标的多样化，从而提升分类的效果。

关于隐含层的层数以及每一层的维度，这些都可视为是超参数，需要不断验证从而得到性能更好的模型。关于优化问题，与线性分类器的方法相同（反向传播，链式法则），只是这里的表达式变得更复杂了，变成了多个线性分类器的组合，由激活函数关联。理论上神经网络的层数越深性能应该会越好，但此时需要对权重进行适当的正则化操作，以免模型的过拟合。

总结. 经过本次学习，我了解了计算复杂表达式梯度的方法，即反向传播方法。需要注意的是在前向传播时需要记录当前的输入变量值，在反向传播过程需要用来计算局部梯度，还有就是矩阵表达式的梯度形式并不固定，只要注意维度保持即可。此外，我对神经网络的结构也有了初步了解。即由非线性激活函数链接的多个线性分类器，其作用是使模型能够适应目标的多样化表示。

1.4. 神经网络训练细节

截止目前，课程已经介绍了如何使用神经网络来定义图像分类问题，并给出了相应的优化方法。这里将主要讨论神经网络在训练过程中的一些细节问题，主要包含：激活函数、权重初始化、正则化以及参数更新方法。

激活函数. 不同线性分类器之间的非线性函数。具有多种不同形式，如 Sigmoid、tanh、ReLU、Maxout、ELU 等。下面将具体说明每个激活函数的特点。

1. Sigmoid. 该函数能够将输入值挤压到 $[0, 1]$ 的范围内。曾经比较常用，因为该函数可以很好的解释神经元的激活频率。但该函数在饱和时的梯度趋近为 0，会引发梯度消失的问题。在实际操作中，如果一个 Sigmoid 的神经元饱和了，根据反向传播原理，其趋近于 0 的梯度会被乘到后面的整条链路的神经元梯度上面。因此如果网络中很多神经元都饱和了，可能会导致大量梯度值趋近于 0，从而无法继续更新参数。另外一个问题是该函数输出不是关于原点对称的，而是 $[0, 1]$ 的实数。当这样的数据作为输入传递到下一层时，下一层的参数更新就会出现异常，若下一层有这样的线性映射： $f(\sum_i w_i x_i + b)$ ，若 x_i 都是大于 0 的实数，则该层关于 w_i 的梯度将要么全为正，要么全为负（正负取决于再下一层的梯度），从而导致网络的收敛速度变慢。最后一个问题就是 $\exp()$ 关于 x 的计算比较耗时。

2. tanh. 该函数是 Sigmoid 函数的一个变种，实现了原点中心化，但梯度消失与计算效率方面与 Sigmoid 相同。

3. ReLU, Rectified Linear Unit. 定义为 $\max(0, x)$ ，一种使网络收敛更快的激活函数。主要原因是：其在已激活区域的梯度不会饱和（梯度永远为 1）；运算效

率高，只是作比较而已，没有进行复杂的指数运算。通常被选为默认的激活函数。但没有实现原点中心化，且非激活部分无法进行反向传播。

4. Leaky ReLU. 定义为 $\max(0.01x, x)$ ，其基本思想是使激活函数分段线性，在保留 ReLU 优势的同时修补它的问题。在该函数任何区域内都不会出现饱和情况。

5. ELU. 将 $x \leq 0$ 的部分替换为指数函数，具有 Leaky ReLU 的优点，同时其输出更接近 0 均值。但加入了指数函数，使得计算效率降低了。

6. Maxout "Neuron". 定义为 $\max(w_1^T x + b_1, w_2^T x + b_2)$ ，没有 ReLU 的缺点，不会饱和，也不会有神经元失活，但参数量是加倍的。

Instructor 在课程中建议实际使用中尽量用 ReLU。但要注意学习率的选择，因为过大的学习率会导致神经元在一定范围内波动，从而会导致数据多样性不可逆的丢失。

权重初始化. 在训练网络之前，还需要做的是数据的预处理和权重初始化。对于数据预处理，Instructor 指出实际操作通常采用的方法是 0 均值化，原因在解释 Sigmoid 函数时有提到。关于权重初始化，课程中介绍了几种常见的方法。

1. 小随机数初始化. 在损失函数与优化过程的章节中，能够看到优化目标是找到能实现损失函数最小化的权重矩阵。因此初始权重可以考虑要接近 0 一些。但又不能等于 0，因为这样会造成梯度小时又不等于 0。课程首先介绍的是小随机数初始化，即使用高斯分布初始化，然后乘 0.01。但这种初始化方式不利于梯度的反向传播，因为小权重会造成输出的值很小，同样趋近于 0。而输出值与权重的梯度是成正比的（输出值是下一层输入，因此下一层权重梯度 = 输出值 \times 下下层反向传回来的梯度）。这样就会导致梯度不更新，网络难以收敛。

2. 校准方差 $1/\sqrt{n}$. 仍然是高斯分布初始化，但对加权做了些变化，不是固定值，而是使用高斯分布的值除每个神经元的输入值开根号，即 \sqrt{n} 。因此如果输入值很多，则权值很小，反之则很大。这种方法能够保证神经元的输入与输出数据之间具有近似同样的分布，这部分的证明在笔记中有提到。这种方法在 tanh 上效果很好，但在 ReLU 上输出数据的方差会随着层数加深逐渐减少。这是因为 ReLU 在激活是去除了小于 0 的输出，因而得到的数据分布是输入方差的一半。为此课程介绍了另外一种校准方差 $1/\sqrt{n/2}$ 。在激活后可以得到更好的数据分布。

3. Batch Normalization. 上述的方法是研究如何设计初始化过程，从而使每个网络的输出值具有与输入值相同的分布（即高斯分布）。Batch Normalization 的做法是在得到输出值之后，直接将其转化为高斯分布：

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}} \quad (12)$$

其中 $x^{(k)}$ 表示 mini-batch 中第 k 个维度的所有数据, $k = 1, 2, \dots, D$, E 、 Var 分别表示在 mini-batch 中的均值与方差。通常 batch normalization 层被放置在网络层与激活层中间。但有时激活层可能不需要输入数据是高斯分布的, 因此 BN 层就有了第二部分:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)} \quad (13)$$

这里相当于给网络一个自我调节的机会, 如果它还是喜欢原来的分布, 可以令 $\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$, $\beta^{(k)} = E[x^{(k)}]$, 再把输出数据调整回来。BN 的引入很好的解决了权重初始化的选择问题, 同时对防止梯度消失也有一定的作用。在实际操作中, 两个参数 γ 与 β 是可训练的, 但在训练过程中需要记录平均的 batch 均值与方差, 在测试时是不计算均值方差的, 甚至 batch size 可能只有 1, 所以要使用训练时记录这两个值。

在训练过程中, 还需要注意数据初始化, 网络结构选择等问题。确定了网络模型后, 为验证模型定义的准确性, 需要对其进行必要的合理性检查与完整性检查。在合理性检查中, 需要查看第一次前向传播得到的 loss。如果是随机初始化的参数, 那么最终得到的分类结果应该是均匀的。若使用 Softmax, 其值应在 $-\log(1/K)$ 左右, K 表示类别的数量。验证损失函数后, 可添加正则项然后再次检查输出的 Loss, 若损失值增加就说明模型在这里是没有问题的。在完整性检查中, 首先会提取训练集的小部分数据, 测试模型能够在小数据集上达到过拟合。一旦饱和, 就对数据集做适当扩充, 继续测试, 以找到能使损失函数减小的学习率。

超参数优化. 一般采用粗糙到精细的策略进行。首先利用少数 epochs 找到参数大概在什么区间内是 work 的; 接下来, 利用一定的训练时间在目标区间内随机查找, 从而确定最优的超参数选择。需要注意的是, 学习率与正则化系数等超参数最好在对数空间查找。之所以采用随机查找的方法, 是因为不同参数对损失函数下降或者准确率上升的影响可能是不同的, 因此网格法优化很难找到最优的超参数组合。得到学习率后需要注意的是, 对权重来说, 其本身的值与每次更新的增量值比例最好在 $1e-3$ 左右。若不满足这个条件, 可通过调整学习率做进一步的优化。

神经网络训练技巧. 神经网络的训练流程主要有四个步骤: 1. 在训练数据集中采样一批样本; 2. 对这些数据做前向传播, 得到 Loss; 3. 做反向传播, 计算参数的梯度; 4. 利用梯度更新参数。这里主要讨论参数的更新问题, 以及关于随机失活与模型融合的内容。

关于参数更新, 主要介绍了 6 种方法:

1. 随机梯度下降, SGD. 更新公式: $x \leftarrow x - \text{learning_rate} * dx$ 。更新速度较慢, 实际中很少用, 因为其在水平方向的梯度很小, 因此基本上只沿着梯度最大的方向运动, 导致 SGD 通常是以震荡的方式达到最小值。

2. 动量更新. 更新公式: $x \leftarrow x + v$, 其中 $v = m * v - \text{learning_rate} * dx$ 。该方法不直接使用计算得到的梯度做更新, 而是将梯度增加在变量 v 上。 m 是一个超参数, 位于 0-1 之间, 通常取 0.9。笔记中解释该值的意义与摩擦系数类似, 能够抑制下降的速度, 降低系统动能。在实际的 mini-batch 训练过程中, 可能每次更新时都可能会得到很多随机值, 这样的话经过反向传播就会得到带有噪声的梯度, 而动量法能够记忆连续几次更新时的梯度值, 因此将这些值的衰减和对稳定梯度方向具有促进的作用。

3. Nesterov 动量更新. 更新公式: $v_t = \mu v_{t-1} - \varepsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$, $\theta_t = \theta_{t-1} + v_t$ 。该方法不通过逐渐调整方向达到目标, 而是直接向目标方向做梯度的调整, 比普通动量法效果更好, 但计算起来比较复杂。

4. AdaGrad 更新. 更新公式: $x \leftarrow x - \text{learning_rate} * dx / (\text{sqrt}(\text{cache}) + 1e-7)$, $\text{cache} \leftarrow \text{cache} + dx ** 2$ 。与 SGD 相比, 增加了附加量来对学习率进行放缩, 参数空间的每个维度都有一个自己的学习率, 会根据梯度值的大小进行动态的变化, 这种效果会让水平方向梯度相对于垂直方向更新更快。Instructor 指出在神经网络中, 总是关注梯度减小最快的方向并不是很好的优化策略, AdaGrad 正是用来平衡这一点的。但如果训练时间过长, cache 的值会不断增加, 这样会导致学习率减少, 并影响网络收敛的速度。

5. RMSProp 更新. $\text{cache} = \text{decay_rate} * \text{cache} + (1 - \text{decay_rate}) * dx ** 2$, 加入了衰减率 decay_rate, 以免 cache 越来越大直至梯度更新停止。

6. Adam 更新. RMSProp 与动量结合: $x \leftarrow x - \text{learning_rate} * m / \text{sqrt}(v) + 1e-7$, 其中 $m = \text{beta1} * m + (1 - \text{beta1}) * dx$, $v = \text{beta2} * v + (1 - \text{beta2}) * dx ** 2$ 。在进行更新前, Adam 还需要做: $m /= 1 - \text{beta1} ** t$, $v /= 1 - \text{beta2} ** t$, 这是针对 m , v 在更新初期值为 0 的措施, 以免出现更新错误的情况。

正则化 (Dropout). 一种简单有效的正则化方法。在进行前向传播时, 网络中的神经元将以概率 p 被置为 0, 其中 p 是一个超参数, 通常为 50%。在训练过程阶段, 随机失活可被理解为将定义的神经网络划分为多个子网络, 每次更新只对子网络中的神经元进行改变。这种方法能够减少训练过程设计的变量数量, 同时减少网络过拟合的可能性。Instructor 对 Dropout 的一种解释是: Dropout 会强迫网络认为其对图像的表达是有冗余的, 这样网络在缺少某些特征的情况下仍能够得到理想的分类结果, 同时也不过分依赖于某些特征。在测试过程阶段, Dropout 是不进行的, 因此可视为将多个训练后的子网络进行了模型集成, 从而得到一个平均的预测结果。

总结. 经过本次学习, 我了解了要训练一个神经网络训练的基本流程, 以及需要注意的关键点都是什么。流程主要是: 样本数据采集; 前向传播; 反向传播以及梯度更新。在这过程中我掌握了图像数据的初始化方式 (通常是 0 均值化)、多种激活函数的定义、特点以及缺省的激活函数选择 (ReLU)、多种权值初始化方法的定义与特点以及多种参数更新方法的定义与特点。此外, 还有进一步补偿权重初始化不足的 Batch Normalization, 实现网络正则化的 Dropout 等。截止目前, 我已经完成了 CS231n 中神经网络的课程, 因此具有独立实现一个用来做图像分类的神经网络的能力。下面的章节将具体介绍我有关 MNIST 数字识别分类器的设计与实现细节。

2. 全连接神经网络实现

经过 CS231n 课程神经网络部分的学习, 我对神经网络在图像分类方面的问题定义、操作流程以及细节处理方面都有了比较全面的了解。为了检验我学习到的相关知识, 我使用 Python 实现了一个 3 层的全连接神经网络, 用来进行图像分类。其中数据集使用的是经典的 MNIST (<http://yann.lecun.com/exdb/mnist/>), 是一个有关手写数字识别的数据集, 包含 60000 条训练样本以及 10000 条测试样本。实验是在 MacOS 操作系统, CPU 2.7 GHz Intel Core i5 的平台下进行的, 网络结构为 3 层全连接网络, 学习率 0.01 (每 2epochs 做减半衰减, 共 10 个 epochs), 参数更新基于动量法 (动量为 0.9), batchsize 为 100。在上述设置中, 对测试数据的分类结果达到 97.33%, 训练时间 1158s。除训练与测试外, 为了直观的验证分类结果, 程序可以随机的显示 10 幅图像的原图及其类别划分, 如下图所示。后续章节会对网络结构的定义、实现细节做详细的分析, 同时也对实验结果做一些讨论。



Figure 3. 随机 10 幅 MNIST 数据集图像的分类结果.

关于使用. 上述神经网络的实现代码我已经上传到我的 Github 仓库:

<https://github.com/gaomingqi/NNforMNIST>。程序入口为 `main.py`，包含三个函数，`train()` 用于训练，得到的模型会保存在 `model` 文件夹；`test()` 用于计算测试集的分类准确率；`test10RandomImgs()` 随机分类 10 个样本，并使用图形界面显示。

2.1. 网络结构定义

网络结构图如下图所示：

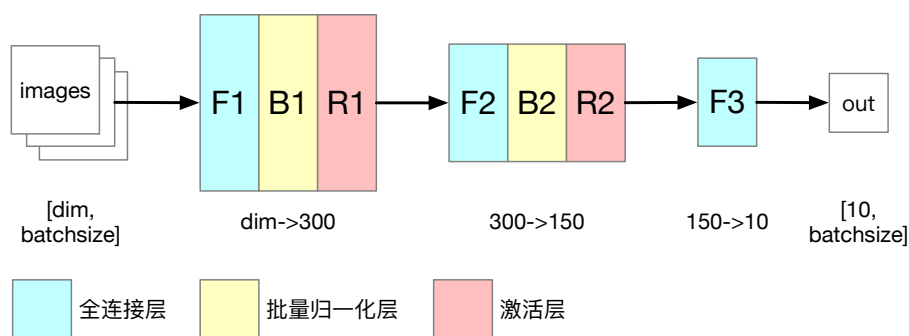


Figure 4. 网络结构图.

从图中可以看到，本次实验的神经网络主要由 3 个全连接层构成，全连接层主要负责对输入数据进行线性映射，使分类器对待分类数据的多样性进行建模。批量归一化层主要用来调整全连接层输出数据的分布，使网络对权重的不同初始化具有鲁棒性，同时在一定程度上避免梯度消失。激活层的作用是对线性分类器的结果进行“选择”，只有满足某些特定条件才能被“激活”，继续影响对输入数据的分类。图中的 `dim` 表示图像在经过向量化后的维度，“数字-> 数字”表示网络层对数据维度的变化。

根据上图的结构，我们使用 Python 对网络进行了定义，在文件 `network.py` 中可查看。首先，我们定义了三种层结构：全连接层（`Linear`），批量归一化层（`BN`），激活层（`ReLU`），定义出的代码截图如下图（5）所示。这些类均有自己的初始化函数（`__init__`）、前向传播函数（`forward`）、反向传播函数（`backward`）以及一个控制是否计算梯度的函数，在介绍反向传播时会提到他的作用。此外，全连接层与 `BN` 层还有一些 `get` 方法与 `set` 方法，用与在训练或者测试过程中，对网络层参数进行存取操作。在定义网络层后，我们借鉴 PyTorch 框架定义了网络结构，如图（6）所示：

根据神经网络的定义，可以看到我们三个全连接层，两个 `BN` 层，两个激活层。**注意：这里的网络层的变量名与图（4）中网络层的名称是对应的。**

2.2. 实现细节

上一节主要介绍了网络的整体结构，以及他们在代码中的表示等。对神经网络来说，我认为训练是其最最重要的功能，同时这里也是代码最多的模块。因此本节主要介绍关于网络的前向传播、反向传播以及参数更新的细节信息。

```

# 全连接层定义与实现
class Linear:
    # 初始化函数
    def __init__(self, in_dim, out_dim):
        :param in_dim: 输入数据维度
        :param out_dim: 输出数据维度
        # 使用高斯函数初始化权重分布
        self.W = np.random.randn(out_dim, in_dim)
        # grad 参数用来控制是否需要在前向传播中记录梯度
        self.grad = False
        # 记录全连接层的输入数据, 以及权重
        self.X = None
        self.WT = None

# BN 层定义与实现
class BN:
    # 初始化函数
    def __init__(self, dim):
        :param dim: 输入数据维度
        # gamma, beta: 调整输入数据范围的参数
        # 让模型自己选择是否再回到原有的分布
        self.gamma = np.ones([dim, 1])
        self.beta = np.zeros([dim, 1])
        # 记录训练过程中所有 Batch 的均值与方差
        self.running_mean = np.zeros([dim, 1])
        self.running_var = np.zeros([dim, 1])

# 激活层定义与实现 (ReLU)
class ReLU:
    # 初始化 ReLU 层
    def __init__(self):
        # grad 参数用来控制是否需要在前向传播中记录梯度
        self.grad = False
        # 记录输入数据
        self.input = None
        # 前向传播函数
    def forward(self, x):
        # 计算 Loss 对当前层输入的梯度
    def backward(self, top):

```

Figure 5. 网络层定义代码截图.

```

class NN:
    # 初始化函数
    def __init__(self):
        # 网络结构: X->F1->B1->R1->F2->B2->R2->F3->Out
        # 三个全连接层
        self.F1 = Linear(784, 300)
        self.F2 = Linear(300, 150)
        self.F3 = Linear(150, 10)
        # 两个激活层
        self.R1 = ReLU()
        self.R2 = ReLU()
        # 两个批归一化层
        self.B1 = BN(300)
        self.B2 = BN(150)

```

Figure 6. 神经网络定义代码截图.

前向传播. 为使这部分的说明更加清晰, 我首先给出这个网络的数据流图:

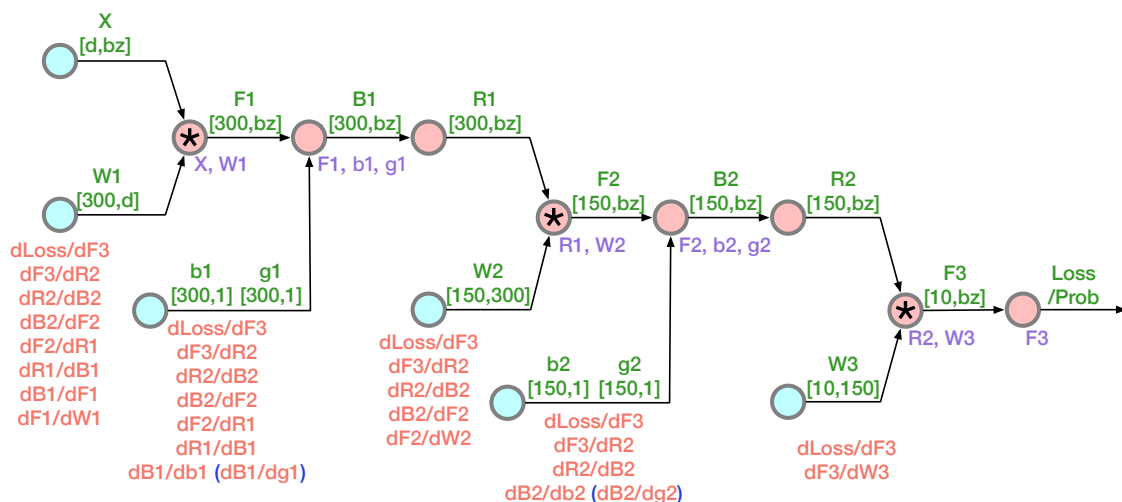
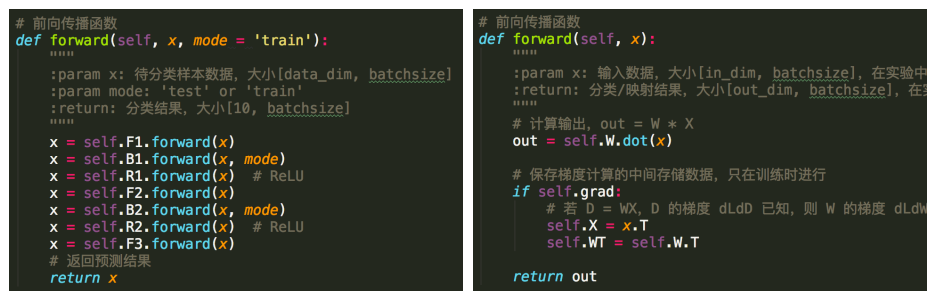


Figure 7. 本报告神经网络的数据流图.

图中的蓝色节点表示输入数据, 红色节点表示计算单元。这里我们只关注绿色文字的部分。前向传播的实现比较简单, 就是在得到数据后, 不断的与权重相乘 (W_1, W_2, W_3)、做元素级的归一化与激活即可得到最终的分概率, 或者进一步得到损失函数。需要注意的地方就是数据维度, 图中在每个输入输出值的下面都给出了相应的维度。为节省空间有些部分做了简写, d 表示数据样本的维度, bz 表示 batch size, Prob 表示 Softmax 的对数概率, 对测试来说, 计算到 Prob 就

可以了，若在训练阶段，则需要再计算损失函数的值 Loss。计算过程按照图中的数据流向即可得到前向传播的结果，本报告的网络前向传播代码以及部分网络层的前向传播代码截图如下：



```

# 前向传播函数
def forward(self, x, mode = 'train'):
    """
    :param x: 待分类样本数据, 大小[data_dim, batchsize]
    :param mode: 'test' or 'train'
    :return: 分类结果, 大小[10, batchsize]
    """
    x = self.F1.forward(x)
    x = self.B1.forward(x, mode)
    x = self.R1.forward(x) # ReLU
    x = self.F2.forward(x)
    x = self.B2.forward(x, mode)
    x = self.R2.forward(x) # ReLU
    x = self.F3.forward(x)
    # 返回预测结果
    return x

```

```

# 前向传播函数
def forward(self, x):
    """
    :param x: 输入数据, 大小[in_dim, batchsize], 在实验中
    :return: 分类/映射结果, 大小[out_dim, batchsize], 在实验中
    """
    # 计算输出, out = W * X
    out = self.W.dot(x)

    # 保存梯度计算的中间存储数据, 只在训练时进行
    if self.grad:
        # 若 D = WX, D 的梯度 dLdD 已知, 则 W 的梯度 dLdW
        self.X = x.T
        self.WT = self.W.T

    return out

```

Figure 8. 左：神经网络的前向传播代码截图；右：Linear 层的前向传播代码截图。

反向传播. 接下来是反向传播的实现，这一部分对网络学习效果的影响非常重大。在本报告中，需要更新的参数有：3 个权重矩阵 (W_1, W_2, W_3)，4 个 BN 层中调整数据分布的参数 ($\beta_1, \gamma_1, \beta_2, \gamma_2$)。根据反向传播的链式法则，就可以得到损失函数分别对这些参数的梯度。回到图 (7) 中，这时我们主要关注红色文字与紫色文字。其中红色文字就是我根据数据流图推导得到的，损失函数关于本报告参数的梯度计算堆栈。比如 W_1 ，损失函数关于 W_1 的梯度 $dLoss/dW_1$ 就等于 W_1 下面一系列局部梯度的乘积。虽然计算起来比较容易，但我们可以发现其中有些门单元的局部梯度需要前向传播时的输入数据。举个例子，在第一个计算门单元中，其关于 W_1 、 X 的梯度就分别与 X 、 W_1 两个输入数据相关。因此在前向传播过程应加入记录所需数据的功能，图 (7) 中的紫色文字即为相应计算门单元在前向传播时需要记录的数据。从图 (9) Linear 层的 forward 实现代码中，可以看到我的代码中加入了这个功能。在前向传播中，神经网络可以选择存储（训练时）以方便梯度计算，或不存储（测试时）以节约存储空间及其附加的计算消耗。

在得到局部梯度后，接下来就是将这些梯度按照链式法则累乘起来，计算损失函数关于可训练参数的梯度。具体的计算方式与前向传播相反，是自顶向下逐层计算的。下图将给出本报告神经网络的反向传播代码截图以及 Linear 层的 backward 代码截图。**注意：在神经网络反向传播的代码中，其变量表示的意义与图 (7) 中红色文字表示的意义是相同的。**

将 Linear 层的 forward 层、backward 层结合起来可以看到，局部梯度能够利用前向传播记录的输入数据计算出来，再乘上 Loss 对上一层的梯度值（变量 top），就可以得到 Loss 对本层输入数据的梯度了。

参数更新. 在得到损失函数关于网络参数的梯度后，通过使参数向负梯度的方向更新即可实现损失函数的最小化。在上一章节有介绍了多种更新方法，如随机梯度



Figure 9. 左：神经网络的反向传播代码截图；右：Linear 层的反向传播代码截图。

Table 1. 使用不同更新方法的神经网络在性能方面的对比结果.

网络结构	更新方法	验证准确率 (%)	测试准确率 (%)	训练时间 (s)
300, 150	Adam	92.53	94.90	4974
300, 150	Momentum	93.85	96.02	4583
300, 150	RMSProp	92.95	95.76	5506
300, 150	SGD	93.56	96.22	4473

下降，动量法，RMSProp 以及 Adam。本章节实现了这几种方法，并观察了它们对网络收敛时间以及准确率的影响。图（10）显示了这四种方法的验证准确率以及损失函数值随训练时间变化的趋势，表格（1）则比较了这些方法的验证准确率、测试准确率以及训练所需时间等信息。需要注意的是，这里我们将 batchsize 设为了 10，因此训练时间会比较久一些，这是我在实验初期设置的 batchsize，后续实验发现取 100 的效果更好，而且训练时间更少。

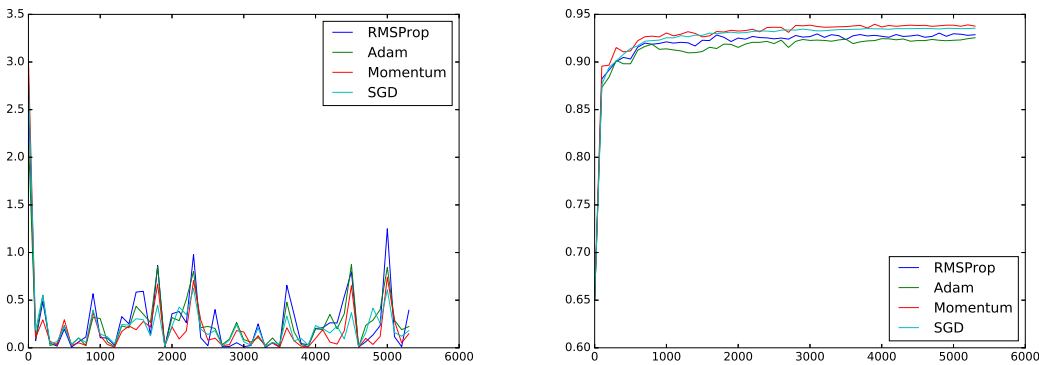


Figure 10. 左：损失函数随训练时间增加的变化趋势；右：验证准确率随训练时间增加的变化趋势。

从网络的收敛率与训练效果来看，并没有反映出 Adam、RMSProp 的优势，我觉得原因可能在超参数设置方面。因为时间关系，我并没有做超参数的交叉验证，因此实验中的网络结构，学习率，正则化系数等超参数很有可能不是最优的，从而导致更新方法的优势无法体现。但也有可能是 MNIST 数据集比较容易分类，无论

Table 2. 使用不同超参数的神经网络在性能方面的对比结果.

网络结构	batchsize	验证准确率 (%)	测试准确率 (%)	训练时间 (s)
300, 150	10	93.85	96.02	4583
300, 150	100	<u>97.40</u>	<u>97.33</u>	1158
300, 150	200	97.08	96.72	521
100, 30	100	94.76	94.03	490
500	100	<u>97.40</u>	96.98	1436
100	100	96.75	96.10	<u>414</u>

使用什么更新方法，都可以实现快速收敛。

2.3. 实验结果分析

由于时间的关系，这段时间只是完成了一个使用 Python 编写的神经网络，有些前面章节学到的训练技巧还没有放在这里实践。尤其是超参数选择部分，有关网络结构、学习率、batchsize 与正则化系数等都是依照测试时的经验得到，并没有进行全面的调优过程。这里将给出我在实验中实现过的一些超参数的性能对比，包括网络结构、batchsize 的选择，对比结果如表格 (2) 所示。虽然对比实验不是很全面，但还是能体现出一些问题。如网络参数的数量与分类准确率是呈正比的，参数越多，训练效果越好，但训练时间会长一些。此外，batchsize 的选择对训练时间与训练效果的影响也较大，其值的增加能够减少每次梯度更新中产生的噪声，因此对分类器的鲁棒性得到了一定提升。

2.4. 总结

经过 CS231n 课程第一部分即神经网络部分的学习，使我对神经网络的组成成分以及训练细节都有了更新更准确的认识。如激活层的作用、BN 层的功能与原理、更新函数以及权重初始化方法各自的特点等。同时也为接下来对卷积神经网络的深入学习打好了基础，因为二者除了全连接层与卷积层之间的区别，在激活函数、权重初始化以及更新方法等知识是相同的。而在编码的过程中，我学习到了很多课程中体会不到的知识。比如反向传播，在实际操作过程中遇到过链式法则如何使用的问题，最初的方案是先将图 (7) 中每个参数所需的局部梯度分别计算出来，再进行一次整体累乘。但如果局部梯度很多的话，计算过程就非常容易出错，而且代码的可读性也变差了。经过不断修改验证，才找到目前这种自顶向下的梯度求解方式，也体会到链式法则不是在最后一次累乘全部的局部梯度，而是逐步递进的计算损失函数对每一层的梯度。此外，经过这次的实现过程我也感受到了微积分知识的强大，体会到数学知识在计算机科学尤其是神经网络领域的重要作用。