中山大学数据科学与计算机院本科生实验报告

(2020 秋季学期)

课程名称：人工智能 **任课老师：**饶洋辉

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340233 & 18340236	姓名	朱益澄 & 朱煜
Email	zhuy85@mail2.sysu.edu.cn zhuych9@mail2.sysu.edu.cn	完成日期	2020.11.11

实验六 CNN 与 RNN

1 实验原理

1.1 CNN

卷积神经网络主要用于图像分类与识别等任务上。在卷积网络出现之前，图像相关的任务诸多困难。首先，采用单纯的全连接层会引入非常多的参数。譬如以本次实验的数据集 cifar10 为例，图像的大小为 32*32*3，32*32 的像素点，3 个通道。如果有一个全连接层，以这 3072 个整数为输入，以 10 个数字为输出，那么我们需要整整 32*32*3*10 个待优化参数，超过了三万。事实上，这样的全连接层我们需要多个，参数量的大小是十分恐怖的，优化困难。其次，全连接层的表达能力也存在局限性，输出会收到尺度缩放、平移、旋转等操作的影响。而往往这些操作并不改变图像所蕴含的语义。比方说，有一只狗的图像，放大以后还是狗的图像。

相较于传统的神经网络，卷积神经网络引入了卷积层和池化层，能够很好地解决与图像相关的任务。

1.1.1 卷积层原理

卷积层采用了卷积操作，可以很好的提取图像的局部特征以供后续处理。

二维卷积操作的数学公式如下所示：

$$y_{i,j} = \sum_{u=1}^{m}\sum_{v=1}^{n} w_{u,v} \times x_{i-u+1,j-v+1}$$

举个例子，如果我们有如下的形如 x 和 w 的两个矩阵，他们卷积的结果就是 4。

$$x = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad w = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

在实际的使用中，我们采用点乘，也就是对应位置相乘并相加替代卷积操作，因为显然它们是等价的。

在卷积神经网络中的卷积层中，我们有输入 x 和卷积核 w。输入 x 可能是原始的图像，也可能是一个中间张量，是一个三维的张量。卷积核是网络中的待优化参数。卷积核的规模较小，常见的规模是 3*3 或者 5*5，也是三维的。为了实现卷积操作，我们需要对于图像的每一个部分反复的进行卷积操作，因为正如我们之前看到的，卷积操作的两个矩阵是需要规模相同。所以实际上我们是在用一个卷积核“扫描”整个输入，得到对应的输出。

假设我们采用了一个 5*5*3 卷积核，输入的大小为 32*32*3，那么卷积层的操作如图1所示：

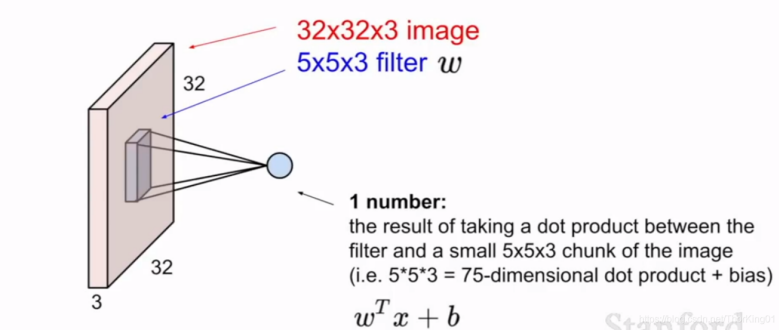

图 1: 卷积层的操作

卷积核反复扫描输入张量，每次作点乘，得到一个结果，然后移动到下一位置，循环往复。就这样，我们得到了规模为 28*28*1 的输出张量。

需要注意的是，在每次的卷积操作中，我们使用的都是同一个卷积核，也就是说，在图片的不同位置，卷积核是参数共享的。

如果我们需要输入有多个通道，就可以使用多个卷积核，每个卷积核分别可以得到一个通道。

当采用 5 个卷积核，每一移动的步数为 1 时，卷积层输出结果如图 2所示：

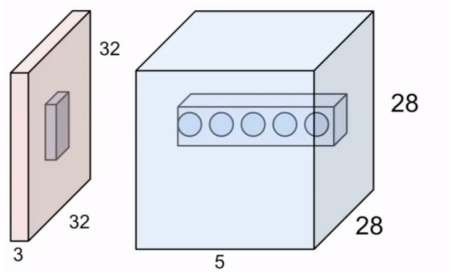

图 2: 卷积层的输出

可以看到，输出的前两个维度大小不变，通道数增加到了五，与卷积核数量相同。

对于卷积层而言，上述的卷积操作会使得输出规模减小。如果我们想要保持原有的规模，可以采用在输入的边界上加入值全为零的 padding。输入张量的规模与输出张量的规模的关系有如下公式：

$$output\_length = \frac{input\_length - conv\_kernel\_length + 2 * padding}{stride} + 1$$

所以，如果对于上述的例子我们想要获得 32*32 的输出，则要在各边增加大小为 2 的 padding。

1.1.2 池化层原理

池化层采用了池化操作，可以降低输入的规模，防止过拟合。

卷积层虽然可以显著减少网络中连接的数量，但特征映射组中的神经元个数并没有显著减少。如果后面接一个分类器，分类器的输入维数依然很高，很容易出现过拟合. 为了解决这个问题，可以在卷积层之后加上一个汇聚层，从而降低特征维数，避免过拟合。

池化操作会将输入划分为多个部分，这些部分可以重叠，也可以不重叠，常见的有将输入划分为多个 3*3 的部分。池化层对每个部分进行分别处理，输出单个的值。常见的池化操作是最大池，会对每个部分求出最大值。对于输入的一个通道，最大值池化操作如图 3所示：可以看到，对于 4*4 的输入，采用 2*2 的 kernel size，那么输出的规模为 2*2。每个区

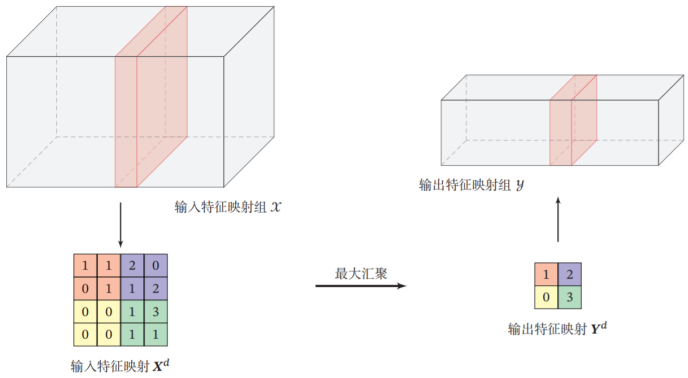

图 3: 最大值池化

域中，池化层会找出局部的最大值。

除去最大池化以外，还有其他种类的池化，如均值池化。均值池化会将一个区域内的值取平均值输出。

最大值池化在图像分类中最为广泛，因为其能很好的找到图像的边缘，转角等信息。通常，在卷积层以后会加上池化层，降低输出规模，减小参数量。

同时，池化层也能够很好的防止过拟合的发生。这是因为池化操作减小了参数数量。可以有效地减少神经元的数量，还可以使得网络对一些小的局部形态改变保持不变性，并拥有更大的感受野。

1.1.3 优化器选择

神经网络的训练是一个优化过程。我们采用的总体方法是梯度下降法。但是，对于复杂的问题，梯度下降法显现出了较大的局限性。

随机梯度下降 (SGD) 面临着几大问题：

(1) 震荡剧烈。如果各个参数对于 loss 的影响幅度不一，采用统一的学习率很可能使得那些与 loss 联系较大的参数难以优化到理想区域。这种情况如图 4所示：

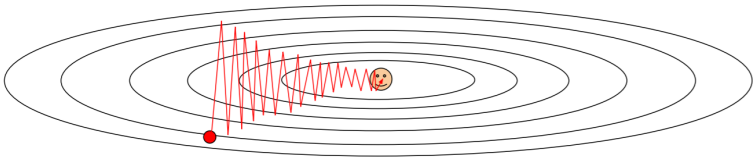

图 4: 震荡剧烈

(2) 无法处理鞍点和局部最优点。对于如下的鞍点和局部最优点，如果优化的过程中，参数落入其中，有可能难以从中脱离，从而陷入了局部最优，使得训练效果较差。

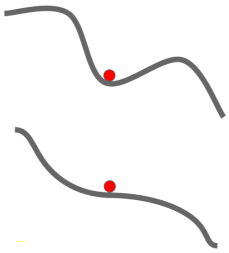

图 5: 局部最优

针对这两个问题，我们可以采用动量法。

对比传统的 SGD，动量法的公式如图 6所示：

我们仿照物理中的速度与力的关系，当前位置的梯度可以看作物体收到的力，实际更新的方向可以看作是物体当前运动的速度。每次我们会用梯度去更新当前的更新方向，然后再用这个更新方向去实际的更新参数。也就是说，参数的更新方向不仅收到当前梯度的影响，也会收到之前梯度的影响。

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

图 6: 动量法

采用了动量法以后，上述的困难都可以得到缓解。因为各个时间的梯度都会影响物体更新方向，所以震荡的问题会得到缓解。同样的，将参数的优化想象为一辆汽车在运动，当引入了动量法以后，我们的这辆汽车可以冲过参数空间中的鞍点和局部极小值点，能够接近于全局的最优点。

以动量法为基础，我们本次实验中采用的是 ADAM 优化器。实现伪代码如下所示：

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

ADAM 优化器以动量为基础，综合考虑了梯度的一阶矩估计和二阶矩估计，并且加上了偏移量修正，防止在训练初期训练过慢。用如上的公式，综合计算出了更新步长。具有实现简单，计算高效，对内存需求少，能自然自动调整学习率等优点。

ADAM 优化器可以在训练的过程中将梯度频繁正负变化的部分相互抵消，从而显著的减少震荡，加快了学习速率。

1.2 RNN

循环神经网络（Recurrent Neural Network，RNN）是一类具有短期记忆能力的神经网络。在循环神经网络中，神经元不但可以接受其他神经元的信息，也可以接受自身的信息，形成具有环路的网络结构，通过使用带自反馈的神经元，能够处理任意长度的时序数据，在本次实验中用以处理不等长的英文文本。

1.2.1 简单循环网络

给定一个输入序列 $x_{1:T} = (x_1, x_2, \cdots, x_t, , x_T)$，循环神经网络通过下面公式更新带反馈边的隐藏层的活性值 h_t：

$$h_t = f(h_{t-1}, x_t),$$

其中 $h_0 = 0, f(\cdot)$ 为一个非线性函数，可以是一个前馈网络。具体网络结构如图 7其中“延时器”记录神经元的最近一次（或几次）活性值。隐藏层的活性值?h_t 也称为状态或隐状态。

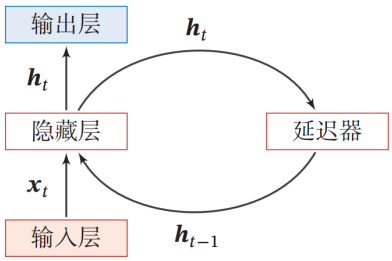

图 7: 循环神经网络

令向量 $x_t \in \mathbb{R}^{\mathbb{M}}$ 表示在时刻 t 时网络的输入，$h_t \in \mathbb{R}^{\mathbb{D}}$ 表示隐藏层状态（即隐藏层神经元活性值），则 h_t 不仅和当前时刻的输入 x_t 相关，也和上一个时刻的隐藏层状态 h_{t-1} 相关．简单循环网络在时刻 t 的更新公式为

$$z_t = Uh_{t-1} + Wx_t + b,$$

$$h_t = f(z_t)$$

其中 z_t 为隐藏层的净输入，$U \in \mathbb{R}^{D\times D}$ 为状态-状态权重矩阵，$W \in \mathbb{R}^{D\times M}$ 为状态-输入权重矩阵，$b \in \mathbb{R}^D$ 为偏置向量，$f(\cdot)$ 是非线性激活函数，通常为 Logistic 函数或 Tanh 函数。而神经网络的输出 y_t 最终表示为

$$h_t = f(Uh_{t-1} + Wx_t + b)$$

$$y_t = g(Vh_t)$$

其中 V 为网络参数，$g(\cdot)$ 是非线性激活函数，在多分类任务中多采用 Softmax 函数。

1.2.2 参数学习

循环神经网络的参数可以通过梯度下降方法来进行学习，以随机梯度下降为例，给定一个训练样本 (x, y)，其中 $x_{1:T} = (x_1, \cdots, x_T)$ 为长度是 T 的输入序列，$y_{1:T} = (y_1, \cdots, y_T)$

是长度为 T 的标签序列，即在每个时刻 t，都有一个监督信息 y_t，我们定义时刻 t 的损失函数为

$$L_t = L(y_t, g(h_t)),$$

其中 $g(h_t)$ 为第 t 时刻的输出，L 为可微分的损失函数，比如交叉熵，那么整个序列的损失函数为

$$L = \sum_{t=1}^{T} L_t.$$

整个序列的损失函数 L 关于参数 U 的梯度为

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial U}$$

随时间反向传播（BackPropagation Through Time，BPTT）算法的主要思想是通过类似前馈神经网络的错误反向传播算法来计算梯度。BPTT 算法将循环神经网络看作一个展开的多层前馈网络，如图 8。其中“每一层”对应循环网络中的“每个时刻”这样，循环神

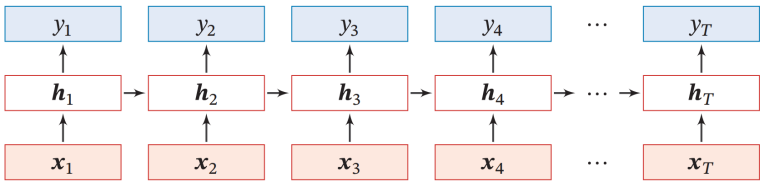

图 8: 按时间展开的循环神经网络

经网络就可以按照前馈网络中的反向传播算法计算参数梯度. 在“展开”的前馈网络中，所有层的参数是共享的，因此参数的真实梯度是所有“展开层”的参数梯度之和。

因为参数 U 和隐藏层在每个时刻 $k(1 \le k \le t)$ 的净输入 $z_k = Uh_{k-1} + Wx_k + b$ 有关，因此 t 时刻的损失函数 L_t 关于参数 u_{ij} 的梯度为：

$$\frac{\partial L_t}{\partial u_{ij}} = \sum_{k=1}^{t} \frac{\partial^+ z_k}{\partial u_{ij}} \frac{\partial L_t}{\partial z_k}$$

其中 $\frac{\partial^+ z_k}{\partial u_{ij}}$ 表示“直接”偏导数，即公式 $z_k = Uh_{k-1} + Wx_k + b$ 中保持 h_{k-1} 不变，对 u_{ij} 进行求偏导数，得到

$$\frac{\partial^+ z_k}{\partial u_{ij}} = [0, \cdots, [h_{k-1}]_j, \cdots, 0]],$$

其中 $[h_{k-1}]_j$ 为第 $k-1$ 时刻隐状态的第 j 维，根据上式可计算梯度进行参数的更新。

对每个训练样例，RNN 先将输入实例提供给输入层神经元，然后逐层将信号前传，直到产生输出层的结果；然后计算输出层的误差，最后 BPTT 算法来对连接权和偏置进行调整。循坏迭代该过程，直到满足某些停止条件（如：达最高迭代次数，误差小于一定值），伪代码如下

Algorithm 1 RNN

输入: 训练集 $D = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{m}$; 学习率 η

输出: 连接权值确定的简单循环网络

function $RNN(D, \eta)$

在 $(0, 1)$ 范围内随机初始化网络中的所有连接权值和偏置,$h_0 = 0$

repeat

for all$(\mathbf{x}_t, \mathbf{y}_t) \in D$ **do**

计算隐藏层的活性值 h_t

$$z_t = Uh_{t-1} + Wx_t + b,$$

$$h_t = f(z_t)$$

计算神经网络的输出 y_t

$$y_t = g(Vh_t)$$

计算 t 时刻的损失函数 $L_t = L(y_t, g(h_t))$

使用 BPTT 算法计算损失函数 L_t 关于参数的梯度

$$\frac{\partial L_t}{\partial u_{ij}} = \sum_{k=1}^{t} \frac{\partial^+ z_k}{\partial u_{ij}} \frac{\partial L_t}{\partial z_k}$$

$$\cdots$$

采用梯度下降更新参数

$$u_{ij} = u_{ij} - \eta \frac{\partial L_t}{\partial u_{ij}}$$

$$\cdots$$

end for

until 达到停止条件

end function

1.2.3 长短期记忆网络

长短期记忆网络（Long Short-Term Memory Network，LSTM）是循环神经网络的一个变体，可以有效地解决简单循环神经网络的梯度爆炸或消失问题。

LSTM 网络引入一个新的内部状态 $c_t \in R^D$，专门进行线性的循环信息传递，同时（非线性地）输出信息给隐藏层的外部状态 $h_t \in \mathbb{R}^D$，内部状态 c_t 通过下面公式计算：

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t,$$

$$h_t = o_t \odot \tanh(c_t),$$

其中 $f_t \in [0,1]^D$、$i_t \in [0,1]^D$ 和 $o_t \in [0,1]^D$ 为三个门来控制信息传递的路径；$\odot$ 为向量元素乘积；c_{t-1} 为上一时刻的记忆单元；$\tilde{c}_t \in \mathbb{R}^D$ 是通过非线性函数得到的候选状态：

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c).$$

LSTM 网络引入门控机制来控制信息传递的路径，三个“门”分别为输入门 i_t、遗忘门 f_t 和输出门 o_t．这三个门的作用为：

(1) 遗忘门 f_t 控制上一个时刻的内部状态 c_{t-1} 需要遗忘多少信息

(2) 输入门 i_t 控制当前时刻的候选状态 $\tilde{c}_t$ 有多少信息需要保存

(3) 输出门 o_t 控制当前时刻的内部状态 c_t 有多少信息需要输出给外部状态 h_t

三个门的计算方式为：

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i), \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f), \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o), \end{aligned}$$

LSTM 的结构单元如图 9, 其计算过程为：1）首先利用上一时刻的外部状态 h_{t-1} 和当前时刻的输入 x_t，计算出三个门，以及候选状态 $\tilde{c}_t$；2）结合遗忘门 f_t 和输入门 i_t 来更新记忆单元 c_t；3）结合输出门 o_t，将内部状态的信息传递给外部状态 h_t．

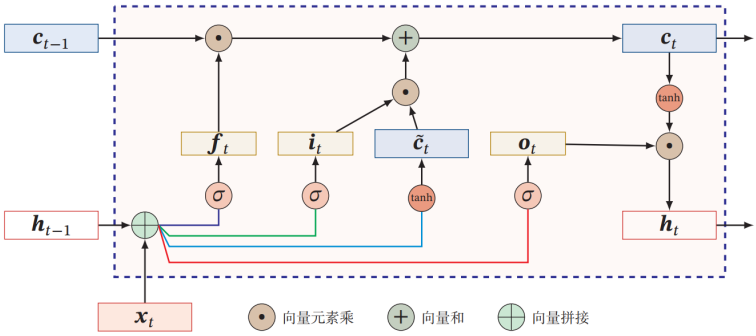

图 9: LSTM 的结构单元

2 实验过程

2.1 CNN

在本次实验中，采用了简单了 lenet，然后采用了进阶的 vgg16，最后作为创新，采用了较新的 resnet18。

2.1.1 lenet

lenet 最先在 1998 年被提出，被用于简单的字母识别上。网络结构十分简单，只有六个隐藏层，卷积层加池化层，卷积层加池化层，卷积层，全连接层接分类器。网络结构如图10所示：

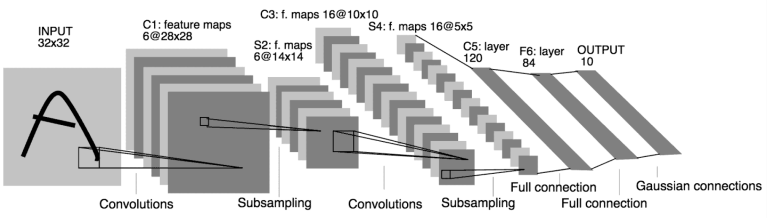

图 10: lenet

网络的输入为一个 32*32 的图像，在 C1 卷积层中，使用六个 5*5 的卷积核，将输入转化为六通道的 28*28 的张量。在 S2 池化层中，每四个元素被最大池化处理，将长和宽降低为原先的一半。接着是相似的卷积核池化操作，C3,S4 与 C1,S2 的定义完全一致。

C5 是一个卷积层，但是其采用了 120 个 5*5 的卷积核，与全连接类似。F6 是全连接层，然后是输出层。

我们对网络结构进行了微调，将 C5 这一卷积层变为全连接层，将全连接层的激活函数由 tanh 变为 relu，并将全连接层的输出规模变换为 10，直接接入输出层。输出层的高斯连接变为 softmax 分类器。如图 11

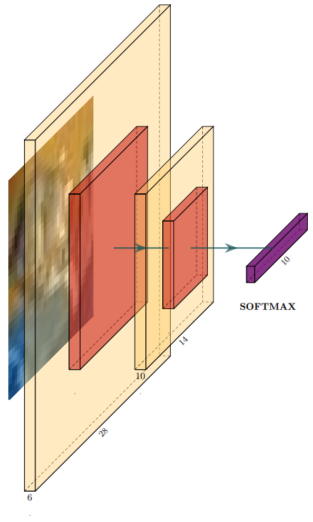

图 11: 网络结构

利用 pytorch，我们可以定义出这一网络。

```
class lenet(nn.Module):
    def __init__(self):
        super(lenet,self).__init__()# 初始化父类
        self.conv1 = nn.Conv2d(3,6,5)# 第一层卷积层
        self.conv2 = nn.Conv2d(6,16,5)# 第二层卷积层
        self.fc1 = nn.Linear(16*5*5,120)
        self.fc2 = nn.Linear(120,10)
    def forward(self,x):
        x = F.max_pool2d(F.relu(self.conv1(x)),(2,2))# 卷积 + 池化
        x = F.max_pool2d(F.relu(self.conv2(x)),2)# 卷积 + 池化
        x = x.view(x.size()[0],-1)# 展平向量到一维
        x = F.relu(self.fc1(x))# 第一层全连接 +relu
        x = self.fc2(x)# 第二层全连接
        return  x
    def initial_weight(self):# 初始化参数
        for m in self.modules():# 采用 xavier 初始化法，初始化每一层参数
            if isinstance(m, (nn.Conv2d, nn.Linear)):

                nn.init.xavier_uniform_(m.weight,gain=nn.init.calculate_gain('relu'))
```

这段代码有几个需要注意的点。

首先是 init 函数，pytorch 要求所有的网络都继承 nn.Module 模块，并在初始化函数中调用父类的初始化函数。然后我们初始化各层。对于卷积层，参数分别是输入通道数，输出通道数以及卷积核大小。我们按照之前的网络结构定义出两层。然后，我们定义了两个全连接层。

然后是前向传播函数。前向传播中值得注意的是，我们并没有将 softmax 写入网络定义中。在后向传播时，可以采用 ‘nn.CrossEntropyLoss()‘交叉熵损失函数直接利用全连接层的输出计算出损失。在后向传播进行预测时，不经过 softmax 处理并不会影响实际的预测值的相对大小。

我们定义了 ‘initail_weight‘函数，用于初始化网络参数，利用的方法时 xavier 初始化法。xavier 初始化法的动机是，在初始化后，使得每层的输入输入的方差尽可能相同。xavier 初始化指的是如下的分布：

$$w \sim U[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}]$$

可以证明，当 w 满足这种分布时，可以使得每层的输入与输出的方差彼此相同。pytorch 提供了相关的接口，只要调用 'nn.init.xavier_uniform_'，并传入使用的激活函数类型即可。

2.1.2 vgg16

vgg16 是牛津大学的 K Simonyan 与 A Zisserman 在 2014 年提出的网络结构，在 imagenet 分类竞赛中取得了相当好的准确率，网络结构的定义如 12所示：

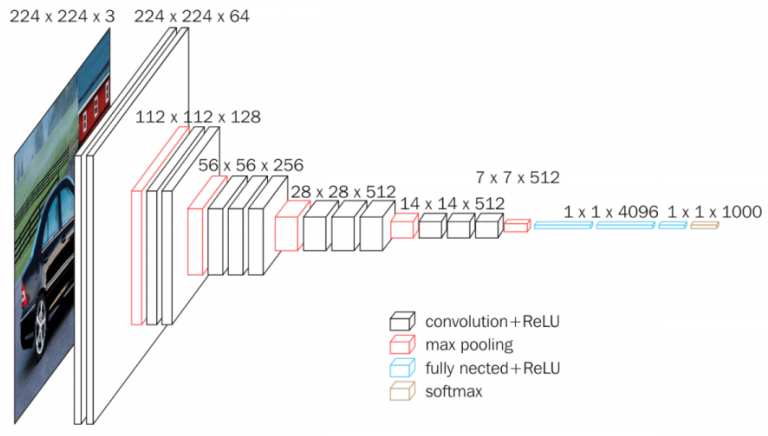

图 12: vgg16

vgg16 针对的是 224*224 的图片进行 1000 分类的任务。采用了多个卷积层与最大池化叠加的形式。原始的 vgg16 共有五大层，每层由 2-3 个卷积层加上 1 个池化层组成。第一大层和第二大层是两个卷积加上一个池化，后面的三大层是三个卷积加上一个池化。然后是三个连续的全连接层，最后是 softmax 分类器。

因为分类任务有所区别，我们对于网络结构进行了微调。

首先去除了众多的全连接层，仅仅保留一个。vgg16 采用众多的全连接的一个原因是所需要分类的种类较多，多层全连接层可以更好的解析 feature vector。但是在本次任务中，我们只需要对图像进行 10 分类，不需要这么多全连接层。事实上，vgg16 网络的绝大多数参数都集中在全连接层处，极大的限制了训练速度。经过实验，在本次任务中，去除了适量的全连接层可以在加速训练的同时，获得更好的实验效果。

此外，我们也适当减少了卷积池化层的数量。这是因为图像的大小降低了，不需要太多的卷积层以获得足够的感受野。在原网络结构中，当感受野降低到 7*7 时候，就进入全连接层。所以相应的，我们可以去除第四和五大层。当获得 8*8 的感受野时就直接结束卷积操作。经过实验，这可以提高分类的准确率。

我们定义的网络的结构如图 13所示：

利用 pytorch，我们可以定义 vgg16，如下所示：

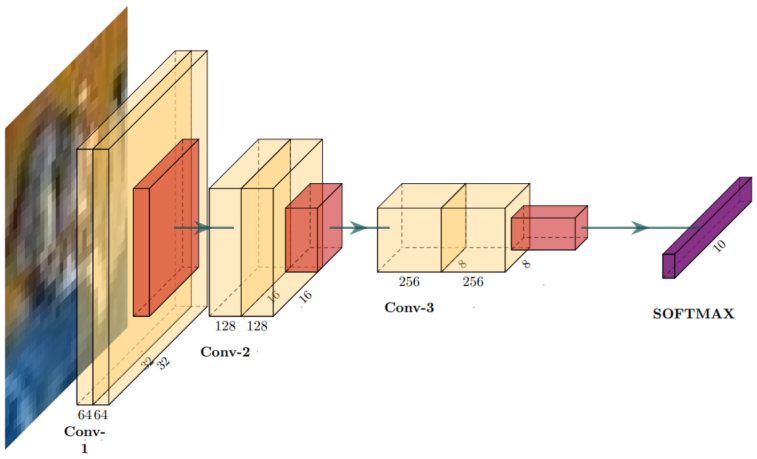

图 13: 网络的结构

```python
class VGG16(nn.Module):


    def __init__(self):
        super(VGG16, self).__init__()

        # 3 * 32 * 32
        self.conv1_1 = nn.Conv2d(3, 64, 3, padding=(1, 1)) # 64 * 32 *32
        self.conv1_2 = nn.Conv2d(64, 64, 3, padding=(1, 1)) # 64 * 32* 32
        self.maxpool1 = nn.MaxPool2d((2, 2)) # pooling 64 * 16 * 16

        self.conv2_1 = nn.Conv2d(64, 128, 3, padding=(1, 1)) # 128 * 16 *
        ↪  16
        self.conv2_2 = nn.Conv2d(128, 128, 3, padding=(1, 1)) # 128 * 16 *
        ↪  16
        self.maxpool2 = nn.MaxPool2d((2, 2)) # pooling 128 * 8 * 8

        self.conv3_1 = nn.Conv2d(128, 256, 3, padding=(1, 1)) # 256 * 8 * 8
        self.conv3_2 = nn.Conv2d(256, 256, 3, padding=(1, 1)) # 256 * 8 * 8
        self.conv3_3 = nn.Conv2d(256, 256, 3, padding=(1, 1)) # 256 * 8 * 8
        self.maxpool3 = nn.MaxPool2d((2, 2)) # pooling 256 * 4 * 4
```

```
20
21          self.fc1 = nn.Linear(256 * 4 * 4, 10)
22          self.dropout = nn.Dropout(p=0.5)
23
24      def forward(self, x):
25
26          # x.size(0) 即为 batch_size
27          in_size = x.size(0)
28
29          out = self.conv1_1(x) # 64 * 32 *32
30          out = F.relu(out)
31          out = self.conv1_2(out)# 64 * 32 *32
32          out = F.relu(out)
33          out = self.maxpool1(out)# 64 * 16 *16
34
35          out = self.conv2_1(out)# 128 * 16 * 16
36          out = F.relu(out)
37          out = self.conv2_2(out)# 128 * 16 * 16
38          out = F.relu(out)
39          out = self.maxpool2(out)# pooling 128 * 8 * 8
40
41          out = self.conv3_1(out)# 256 * 8 * 8
42          out = F.relu(out)
43          out = self.conv3_2(out)# 256 * 8 * 8
44          out = F.relu(out)
45          out = self.conv3_3(out)# 256 * 8 * 8
46          out = F.relu(out)
47          out = self.maxpool3(out)# pooling 256 * 4 * 4
48
49          # 展平
50          out = out.view(in_size, -1)
51          out=self.dropout(out)
52          out = self.fc1(out)
53
54          return out
55      def initial_weight(self):
```

```
56          for m in self.modules():
57              if isinstance(m, (nn.Conv2d, nn.Linear)):
58
                    ↪  nn.init.xavier_uniform_(m.weight,gain=nn.init.calculate_gain('relu'))
```

网络的定义的语法与 lenet 法类似，我们在注释中加入了前向传播过程中输入张量的规模变化。我们使用了 dropout，在 feature_vector 后加入了 dropout，可以有效地防止过拟合的发生。

2.2 创新点

在上述的实验中，一个显著的现象是，对于 vgg 网络，采用三大层卷积加池化时效果最好，层数过多时反而效果变差。当网络层次过深时，会出现梯度消失或爆炸现象的发生，导致过深的网络训练困难，效果不如更浅的网络。这就是所谓的退化现象。退化现象并不是由过拟合引起，而是由梯度下降法本身决定的。

针对退化现象，何凯明博士在 2015 年提出了深度残差网络，也即 resnet。resnet 基于一个简单的想法，如果神经网络的本质是函数估计，那么在已有的函数上加上恒等映射不会降低函数的表达能力。resnet 由许多的残差块组成，单一残差块如图 14所示：

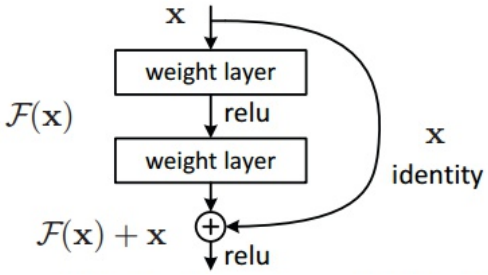

图 14: 单一残差块

在传统的网络中，我们希望每一个卷积层、池化层能够学习到一个函数，将输入映射到合适的输出。与之前的网络不同，在 resnet 中，每层卷积层或池化层学习的目标是输出与输入的差。我们将输入原封不动的加到了卷积池化等层次的后面。假设目标函数为 H(x)，那么我们的学习目标就是残差 H(x)-x。

这种做法的优势在于，如果网络发现一个残差块的存在是不必要的。那么它可以通过将残差块中的权重逼近于 0 的方式，将残差块输出的残差置零，那么这个残差块相当于就不存在了。如果残差块中学习到了有意义的信息，那么就可以提升网络的性能。

理论上而言，增加 resnet 的深度不会降低其原有的性能，这就是 resnet 的威力所在。

标准的 resnet18 的网络结构如图 15所示：

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of block

图 15: resnet18 的网络结构

我们采用的 resnet18 的网络结构与标准的 resnet18 完全一致，如图 16所示：

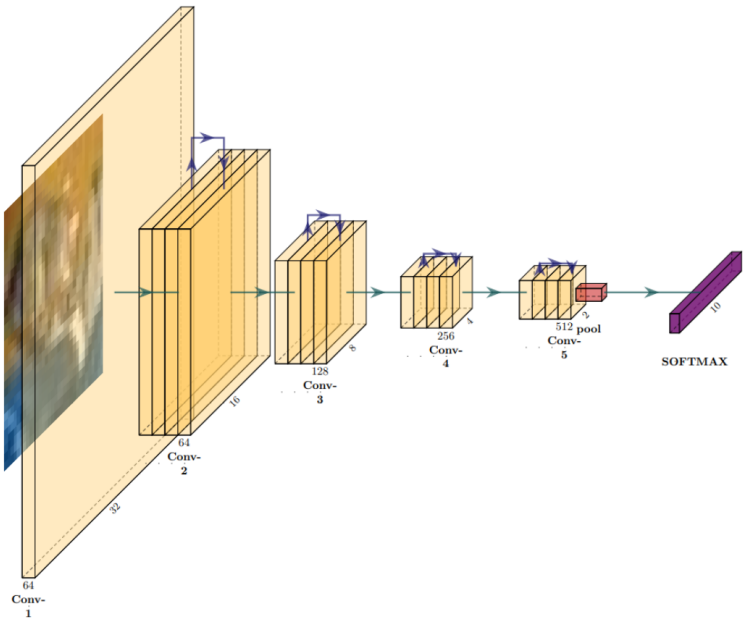

图 16: 采用的 resnet18 的网络结构

利用 pytorch，我们可以定义出 resnet18：

```
class ResidualBlock(nn.Module):# 单个残差块的定义
    def __init__(self, inchannel, outchannel, stride=1):# 输入参数：输入通道数，输出通道数，卷积层步数
        super(ResidualBlock, self).__init__()
```

```
4           self.left = nn.Sequential(# 定义残差模块：一个卷积层，一个 batch 层，
            ↪  一个卷积层，再加一个 batch 层
5               nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride,
                ↪  padding=1, bias=False),
6               nn.BatchNorm2d(outchannel),
7               nn.ReLU(inplace=True),
8               nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1,
                ↪  padding=1, bias=False),
9               nn.BatchNorm2d(outchannel)
10          )
11          self.shortcut = nn.Sequential()# 定义短路模块
12          if stride != 1 or inchannel != outchannel:# 如果输入输出通道数不一样，
            ↪  则需要进行下采样转化
13              self.shortcut = nn.Sequential(
14                  nn.Conv2d(inchannel, outchannel, kernel_size=1,
                    ↪  stride=stride, bias=False),
15                  nn.BatchNorm2d(outchannel)
16              )
17
18      def forward(self, x):# 前向传播，输出等于短路加上残差模块
19          out = self.left(x)
20          out += self.shortcut(x)
21          out = F.relu(out)
22          return out
23
24  class ResNet(nn.Module):
25      def __init__(self, ResidualBlock, num_classes=10):# 初始化 resnet
26          super(ResNet, self).__init__()
27          self.inchannel = 64# 输入通道初始化
28          self.conv1 = nn.Sequential(# 定义第一个卷积层
29              nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,
                ↪  bias=False),
30              nn.BatchNorm2d(64),
31              nn.ReLU(),
32          )
```

```
33          self.layer1 = self.make_layer(ResidualBlock, 64,  2, stride=1)# 定
            ↪ 义第一个残差块
34          self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride=2)# 定
            ↪ 义第二个残差块
35          self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride=2)# 定
            ↪ 义第三个残差块
36          self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride=2)# 定
            ↪ 义第四个残差块
37          self.fc = nn.Linear(512, num_classes)# 全连接输出
38
39      def make_layer(self, block, channels, num_blocks, stride):# 定义一层残差
        ↪ 块，需要通道数，每层的卷积块数目，步数
40          strides = [stride] + [1] * (num_blocks - 1)# 定义残差块内部的步数
41          layers = []
42          for stride in strides:
43              layers.append(block(self.inchannel, channels, stride))
44              self.inchannel = channels
45          return nn.Sequential(*layers)# 构建一层
46
47      def forward(self, x):# 前向传播
48          out = self.conv1(x)
49          out = self.layer1(out)
50          out = self.layer2(out)
51          out = self.layer3(out)
52          out = self.layer4(out)
53          out = F.avg_pool2d(out, 4)
54          out = out.view(out.size(0), -1)
55          out = self.fc(out)
56          return out
57
58
59  def ResNet18():
60      return ResNet(ResidualBlock)
```

这段代码参考了 pytorch 对于 resnet 的官方实现。对于 resnet18 而言，我们首先定义了一个卷积层。然后是连续的四个残差块。每个残差块包括四个卷积块和四个 batch_norm 块，

还有一个短路路径。实现的方法有两个需要注意的点。

首先，在残差块的思想中，理论上短路的部分会使得输入直接加到输出当中。但是在实现时，因为输入输出的大小以及通道数往往不一，输入无法直接叠加到输出当中，需要进行转化。我们的做法是利用 1*1 的卷积核进行下采样，通过设置合适的通道数和步长，可以保证下采样后的规模与输出相同。事实上，尽管这种做法并不能称为恒等映射，但是当卷积核为 1*1 时，几乎可以认为是线性变换。网络可以很简单的在训练过程中通过学习使得这个线性变化近似于恒等变换。

其次，我们采用了 batch_norm 块。Batch Normalization 可以使得每层的输入空间和目标空间的分布一致，减低学习难度。batch normalization 的算法如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

2.3 数据集处理

cifar10 数据集由 32*32 的彩色图片组成，共有十种类别，如马，牛等等。每一类有 6000 张图片，总共有 60000 张图片，其中 50000 张作为训练集合，10000 张作为测试集。

CIFAR-10 数据集被划分成了 5 个训练的 batch 和 1 个测试的 batch，每个 batch 均包含 10000 张图片。测试集 batch 的图片是从每个类别中随机挑选的 1000 张图片组成的，训练集 batch 以随机的顺序包含剩下的 50000 张图片。不过一些训练集 batch 可能出现包含某一类图片比其他类的图片数量多的情况。训练集 batch 包含来自每一类的 5000 张图片，一共 50000 张训练图片。

在本次实验中，我们直接采用 cifar10 为我们划分好的数据集，50000 张作为训练集，10000 张作为测试集。

2.3.1 dataset 类定义

我们可以从官网上下载 cifar10 数据集，利用官网提供的代码（如下），我们可以将数据集转换为字典：

```
def unpickle(file):
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
```

我们需要使用字典的两个键值对，第一个是 'data'，保存了数据本身，第二个是 'label'，保存了数据的标签对于一个 batch 而言，'data'键对应的数据是一个大小为 3072*10000 的 numpy 数组，每一行是一张图片，前 1024 个数字表示 R 通道，后面的 1024 个数字表示 G 通道，最后的 1024 个数字表示 B 通道。'label'是一个长度为 10000 的数组，每个数字代表对应位置的真实标签，取值从 0-9。

按照 pytorch 的语法，我们可以定义自己的 Dataset 类，用于训练时读取数据。定义如下所示：

```
class trainDataset(Dataset): # 创建自己的类: trainDataset, 这个类是继承的 torch.utils.data.Dataset
    def __init__(self, transform=None,target_transform=None, loader=default_loader): # 初始化一些需要传入的参数
        super(trainDataset,self).__init__()# 对继承自父类的属性进行初始化
        # 分别读入每一个 batch
        data_set =unpickle(root+'cifar-10-python/cifar-10-batches-py/data_batch_1')# 调用函数 unpickle 取出字典
        imgs = data_set[str.encode('data')]# 取出 data
        labels=np.array(data_set[str.encode('labels')])# 取出 label

        # 滴入后续的 batch 的数据, 与之前的数据拼接在一起
        data_set =unpickle(root+'cifar-10-python/cifar-10-batches-py/data_batch_2')
        imgs = np.concatenate((imgs,data_set[str.encode('data')]))

        labels=np.concatenate((labels,np.array(data_set[str.encode('labels')])))

```

```
14          data_set
            ↪  =unpickle(root+'cifar-10-python/cifar-10-batches-py/data_batch_3')
15          imgs = np.concatenate((imgs,data_set[str.encode('data')]))
16
            ↪  labels=np.concatenate((labels,np.array(data_set[str.encode('labels')])))
17
18          data_set
            ↪  =unpickle(root+'cifar-10-python/cifar-10-batches-py/data_batch_4')
19          imgs = np.concatenate((imgs,data_set[str.encode('data')]))
20
            ↪  labels=np.concatenate((labels,np.array(data_set[str.encode('labels')])))
21
22          data_set
            ↪  =unpickle(root+'cifar-10-python/cifar-10-batches-py/data_batch_5')
23          imgs = np.concatenate((imgs,data_set[str.encode('data')]))
24          imgs=imgs.astype('uint8')
25
            ↪  labels=np.concatenate((labels,np.array(data_set[str.encode('labels')])))
26          # 将数据 reshape，并调整维度，以满足图片的要求
27          self.imgs = imgs.reshape(50000,3,32,32).transpose(0,2,3,1)
28          self.labels=labels
29          self.transform = transform
30          self.target_transform = target_transform
31          self.loader = loader
32
33      def __getitem__(self, index):# 这个方法是必须要有的，用于按照索引读取每个
        ↪  元素的具体内容
34          temp_img=self.imgs[index]
35          temp_label=self.labels[index]
36          if self.transform is not None:
37              temp_img = self.transform(temp_img)
38          return temp_img,temp_label#return 回哪些内容，那么我们在训练时循环读
            ↪  取每个 batch 时，就能获得哪些内容
39      def __len__(self): # 这个函数也必须要写，它返回的是数据集的长度，也就是多
        ↪  少张图片，要和 loader 的长度作区分
40          return self.imgs.shape[0]
```

对于 dataset 类而言,我们必须要定义三个函数‘ ___init___‘,‘___getitem___‘ 以及‘___len___‘。

‘ ___init___‘负责初始化 dataset 类，从路径中读入数据集，并保存。在这段代码中，trainDataset 定义了训练集。所以，我们读入五个 batch 中的数据和标签，并全部保存。其中，数据保存在 numpy 数组 imgs 中，标签保存在 numpy 数组 labels 中。

‘___getitem___‘是必要的方法，负责按照索引返回数据。返回的数据就是我们在每次训练中具体可以读取得到的数据。在这个类中，我们返回图片和标签。

‘___len___‘负责返回数据集的长度，在本段训练中，就是类的 imgs 数组的第一个维度的长度。

在这段代码当中，一个值得注意的细节是我们对 imgs 数组处理。按照图像内部的定义，图像是一个像素，一个像素的保存的。也就是说，在图片文件的内部，对于第一个像素而言，会有连续的 RGB 三个通道的值描述第一个像素在 RGB 三个通道上的取值。然后是第二个像素，第三个像素。然而，数据集中的图片是三个通道分开的。所以，我们必须要予以处理，先利用 reshape 函数将每张图片从 3072 转换为 3*32*32，然后利用 transpose 函数进行转置，使得图片变为 32*32*3。在这一过程中，每个像素的 rgb 通道的值会被重排聚集。

这是由 reshape 的机制决定的。numpy 的 reshape 可以看作是一个取数 + 填充的过程从变换前的矩阵中取数，向变换后的标签填充。取数的标签从最右侧的维度开始递增，也就是说，如果我们利用双层循环和两个下标 i，j 从二维矩阵 a 中取数，若取出的数是 ‘a[i][j]‘，那么 i 在外层循环，j 在内层循环。存储的标签也是从最右侧标签开始递增，也就是说，如果我们利用双层循环和两个下标 i，j 向二维矩阵 a 中存数，若存入的位置是 ‘a[i][j]‘，那么 i 在外层循环，j 在内层循环。

对于 ‘imgs.reshape(50000,3,32,32)‘这句代码，imgs 在变换前是 50000*3072 的，在 reshape 的过程中，会先取尽 R 通道中的数，将他们分给每一个像素的 R 的值，然后再取 G 的值。RGB 三个通道被均匀打散，完成了向图片的转化。

2.3.2 数据增强

在图像分类的任务中，我们可以通过加大训练集难度的方式提高在测试集上的效果。这种做法的有效性可以类比于做题，平时联系做的题目很难，那么最后考试作简单的题目，成绩也自然会变好。而且这也可以认为是引入了新的数据，自然可以提高模型表现。

数据增强也可以减少过拟合。通过引入了噪声，破坏了原先的图片中可能存在的无关特征，加大了关键特征在模型判断中的比重，从而在未知的测试集上获得较好的效果。比如，如果训练集中所有的猫拍照的背景都有一个花瓶，那么模型可能会通过判断是否有花瓶来判断是否是猫。我们通过裁剪让花瓶在训练集中并不总是出现，那么模型在看到测试集中没有花瓶的猫的图片就会知道，因为猫本身才是判断依据，与花瓶无关，所以即使没有花瓶，这也是一只猫，从而提升了测试效果。

具体来说，我们采用的两种方法提高训练难度，随机裁剪和随机翻转。随即裁剪指的是在图像周围加上 0 的边，然后裁剪出与原先图片大小一致的图片。随机翻转指的是将一半的图片水平翻转，一半的图片不翻转。代码如下所：

```
transforms.RandomCrop(32, padding=4),  # 先四周填充 0，在吧图像随机裁剪成 32*32
transforms.RandomHorizontalFlip(),  # 图像一半的概率翻转，一半的概率不翻转
```

2.4 训练结果及分析

首先展示训练效果（采用效果最优的 resnet18 网络结合数据增强）：
对于测试集中的第一张图片图片（这是一只猫）：

网络给出如下预测：

类别	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
概率	2.8e-6	3.3e-6	2.5e-6	9.9e-1	3.5e-6	6.3e-6	2.9e-6	3.5e-6	1.8e-6	1.6e-6

对于测试集中的第二张图片图片（这是一艘船）：

网络给出如下预测。可见，训练效果优秀。

类别	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
概率	1.7e-5	2.1e-5	8.3e-6	5.3e-6	6.3e-6	5.5e-6	8.9e-6	5.4e-6	9.9e-1	9.1e-6

我们设定步数为 150 步，minibatch 为 128 张图片。采用 adma 优化器，betas 设置为 0.9 和 0.99，不采用学习率衰减。

下面对比不同网络结构，超参数以及训练方法的效果。

2.4.1 lenet 结果

我们将学习率设置为 0.01。lenet 训练的过程中，loss 变化以及验证集上的准确率变化如下所示：

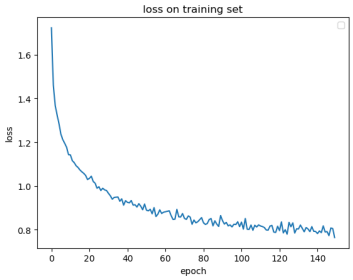

图 17: loss 变化

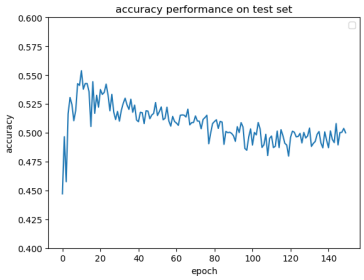

图 18: 验证集准确率

其中，测试集上的最优准确率出现在 epoch10，为 0.5537。150 步后的分类准确率为 0.499。

2.4.2 vgg16 结果

我们将学习率设置为 0.001。vgg16 训练的过程中，loss 变化以及验证集上的准确率变化如下图所示：

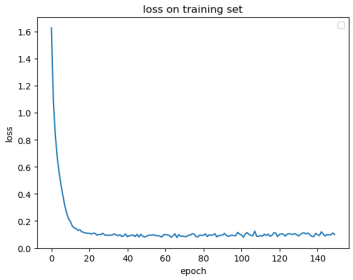

图 19: loss 变化

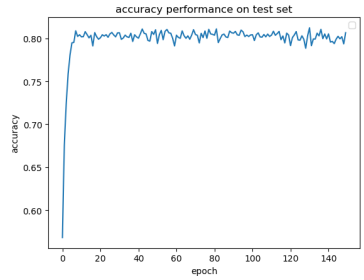

图 20: 验证集准确率

最优的测试集准确率出现在 epoch130，为 0.8123。150 步后的分类准确率为 0.8066。

2.4.3 resnet 结果

我们将学习率设置为 0.001。resnet 训练的过程中，loss 变化以及验证集上的准确率变化如下所示：

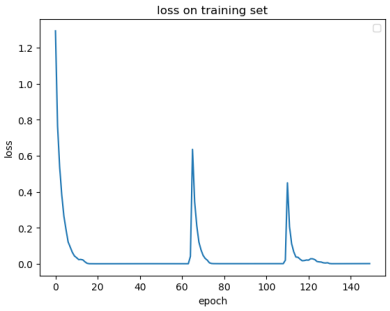

图 21: loss 变化

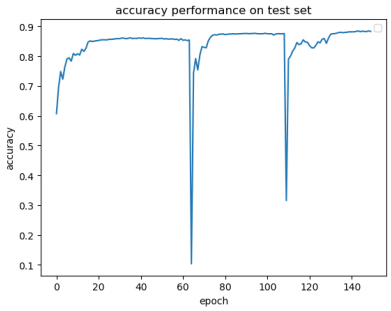

图 22: 验证集准确率

最优测试集准确率出现在 143 步，为 0.8847。150 步后的分类准确率为 0.8831。

2.4.4 不同网络结构的实验结果分析

lenet	55.37%
vgg16	81.23%
resnet18	88.47%

表 1: 各种网络结构准确率比较

可以看到，lenet 的分类准确率并不理想，这是因为模型的表达能力有限。模型迅速达到了最高的分类准确率，并且因为 lenet 本身没有防止过拟合的措施，出现了过拟合的现象。loss 稳定下降，但是测试集的准确率却显著的下降了。

究其根本，lenet 并不是一个用于彩色图像分类的网络。其被发明出来是为了完成识别手写数字这样的简单的任务。对于我们的复杂的彩色图像，lenet 显得力不从心。

vgg16 的准确率相较于 lenet 得到了显著提升，更深的网络具有更优秀的表达能力，能学习到更多有效的特征。同时，我们看到了 dropout 对于网络训练的效果。尽管训练了很多步，模型并没有出现明显的过拟合现象。

resnet18 获得了很好的效果，准确率相较于 vgg16 提升了将近 8%。resnet 模型性能较好的原因是容于理解的，我们在前文的原理部分已经解释过，resnet 通过引入残差块，可以构建非常深的网络而不发生明显的退化现象。

其中，resnet18 在训练的过程中出现了两次震荡，这是尝试走出局部最优点时的现象。

2.4.5 不同超参数

对于 resnet18 网络，我们尝试增大或减小学习率，观察训练效果：

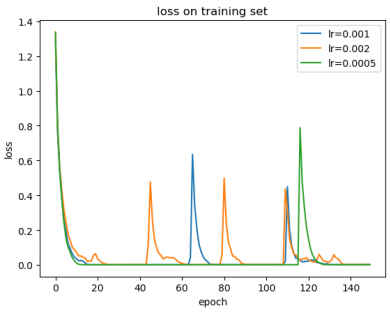

图 23: loss 变化

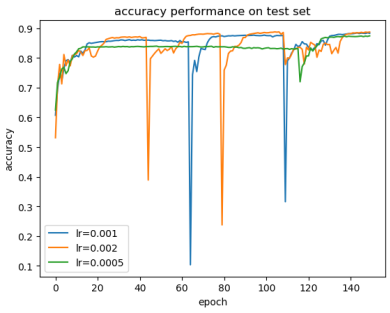

图 24: 验证集准确率

对于 resnet18 网络，我们尝试不同的 batch_size，训练效果如下：

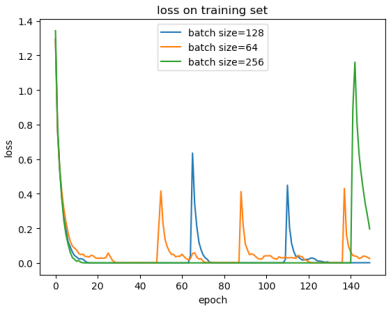

图 25: loss 变化

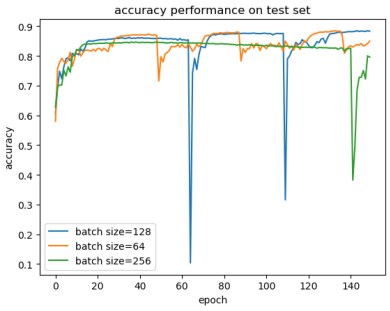

图 26: 验证集准确率

可以看到，当学习率变化时，最终的训练效果基本一致，这是因为 adam 优化器有强大的自适应学习率的能力。对于 batch size 而言，我们经过实验发现，选取 128 的 batchsize，效果是最优秀的。

2.4.6 数据增强效果

对于 resnet18 网络，我们尝试采用数据增强，进行随机截取和随机翻转，对比不采用数据增强的网络，训练效果如下：

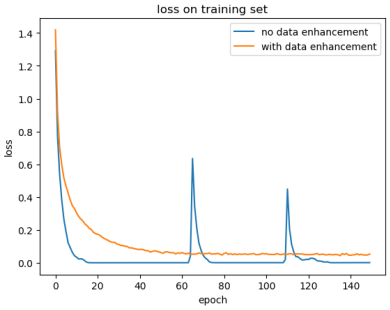

图 27: loss 变化

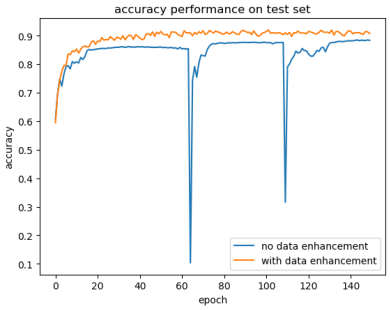

图 28: 验证集准确率

采用数据增强以后，最优的测试集准确率出现在 epoch 101，为 0.9198，相较于不采用数据增强，提高了 4.5% 左右。

一个有趣的事情是，采用了数据增强以后，loss 反而增大了。这背后的原因我们也在原理部分有所讲述。数据增强会起到防止过拟合的作用，可以阻止训练过程中对于无关的、易学习的特征的拟合。所以，loss 上升，accuracy 也上升，这正是防止过拟合起作用的标志。

2.5 RNN

本次实验使用 RNN 完成关键词提取任务，可将该任务转变成序列标注任务，将每个句子中的每个词进行标注，采用 BIO 标注，具体标签类型如表 2。实验中采用的模型结构为

表 2: 标注标签与含义

标签	含义
B	关键词短语的开始词
I	关键词短语的中间词
O	非关键词
<start>	句子开始
<end>	句子结尾
<pau>	填充词

BiLSTM+CRF，BiLSTM-CRF 模型主要包括两部分，即 BILSTM 层和 CRF 损失层，如图 29所示。对于一个输入句子，首先经过 embedding 层将每个词汇或者字符映射为一个词向量或者字符向量，然后传入 BiLSTM 层，获得句子的前向和后向向量，接着将前向和后向向量进行拼接作为当前词汇或字符的隐藏状态向量。隐藏状态向量作为 CRF 层的输入，经过 CRF 层得到输入句子经过模型得到的标签序列。在训练过程中采用梯度下降法对参数进行更新，以得到最吻合的标签序列。具体原理见创新点。

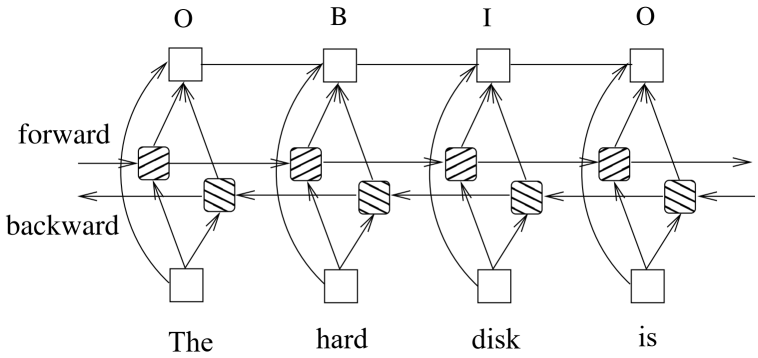

图 29: BiLSTM+CRF 模型

2.6 创新点

2.6.1 BiLSTM-CRF

以句子为输入，将一个含有 n 个词的句子（词的序列）记作

$$x = (x_1, x_2, \cdots, x_n)$$

其中 x_i 表示句子的第 i 个字在字典中的 id，进而可以得到每个词的词向量。$nima$ 模型的第一层是 embedding 层，利用预训练或随机初始化的 embededs 矩阵将句子中的每个词 x_i 映射成词向量 $\mathbf{x}_i \in \mathbb{R}^d$，d 是词向量维度大小。

模型的第二层是双向 LSTM 层，用以提取句子的特征向量，将句子每个词的词向量序列 $(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n)$ 作为双向 LSTM 各个时间步的输入，再将正向 LSTM 输出的隐状态序列 $(h_{1f}, h_{2f}, \cdots, h_{nf})$ 与反 LSTM 向 $(h_{1b}, h_{2b}, \cdots, h_{nb})$ 在各个时间输入的隐状态按输入拼接 $h_t = [h_{tf}; h_{tb}]$，得到完整的隐状态序列 $(h_1, h_2, \cdots, h_n)$。

在设置 dropout 后，接入一个线性层，将隐状态向量映射为 k 维，k 是标注集的标签数，从而得到自动提取的句子特征，记作矩阵 $P = (p_1, p_2, \cdots, p_n) \in \mathbb{R}^{n \times k}$。可以把 $p_i \in \mathbb{R}^k$ 的每一维 p_{ij} 看作字 x_i 分类到第 j 个标签的打分制，如果对 P 进行 $Softmax$，则结果为独立的 k 类分类，不考虑约束。

若不增加 CRF 层，可以选择分值最高的一个作为该词的标签。虽然根据分值得到句子中每个词的正确标签从而进行优化，但是不能保证标签满足标签之间的约束关系，如 I 标签可以在 B 标签或 I 标签后，而不能在 O 标签后，而在训练过程中可能会出现 OI 的序列，很显然这是错误的，因此需要增加 CRF 层来学习约束。CRF 层可以为最后预测的标签添加一些约束来保证预测的标签是合法的。在训练数据训练过程中，这些约束可以通过 CRF 层自动学习到。

模型的第三层是 CRF 层，进行句子的序列标注。CRF 层的参数是一个 $k \times k$ 的矩阵 T，T_{ij} 表示的是从第 i 个标签到第 j 个标签的转移得分，设句子的标签序列为 $y =$

$(y_1, y_2, \cdots, y_n)$，则模型对于句子 x 的标签 y 的打分为

$$score(x, y) = \sum_{i=1}^{n} P_{i,y_i} + \sum_{i=1}^{n+1} T_{y_{i-1},y_i}$$

得分由两部分得到，一部分是前两层得到的输出 p_i 决定，另一部分有 CRF 的转移矩阵 T 决定。进而可以利用 Softmax 得到归一化后的概率：

$$P(y|x) = \frac{e^{score(x,y)}}{\sum_{y'} e^{score(x,y')}}$$

模型训练时通过最小化负对数似然函数，对于一个训练样本 (x, y^x)，损失函数为

$$Loss = -log(P(y^x|x)) = -(score(x, y^x) - log(\sum_{y'} e^{score(x,y')}))$$

模型在预测过程（解码）时使用动态规划的 ViterBi 算法来求解最优路径：

$$y* = arg\,\max_{y'} score(x, y')$$

2.7 关键代码

本次实验采用 pytorch 深度学习架构实现，关键代码分数据处理、BiLSTM 和 CRF 三部分。

2.7.1 数据处理

实验中采用的数据集为 xml 格式，内容有每个句子与对应的关键词，将每个样例用元组 $(sentence : [word_1, word_2, \cdots, word_n], tags : [tag_1, tag_2, \cdots, tag_n])$，sentence 列表中为分词后的词序列，而 tags 保存对应的标签序列。需要根据每个句子分词后的结果得到标签序列，实现代码如下：

```
def GetSentenceTag(sentence_set, termset):
    # 返回的标签序列 初始化全为‘O’
    sentence_tag = ['O' for i in range(len(sentence_set))]
    for term in termset:
        # 对于每个关键词进行分词得到集合
        term_split = term.split()
        begin = term_split[0]
        # 关键短语的开始词标志为‘B’
        if begin in sentence_set:
            sentence_tag[sentence_set.index(begin)] = 'B'
```

```
        # 关键短语的其余词标志为‘I’
        for i in range(1, len(term_split)):
            in_t = term_split[i]
            if in_t in sentence_set:
                sentence_tag[sentence_set.index(in_t)] = 'I'
    return sentence_tag
```

为了能实现同时训练多个数据，需要将句子进行长度处理，通过截断或填充’<PAD>’词来使句子长度相同，同时‘<PAD>’词对应的标签为’<PAD>’，实现代码如下：

```
def TruncAndPad(sentence_set, sentence_tag, max_len):
    # 句子长度大于设定长度则截断
    if (len(sentence_set) >= max_len):
        return sentence_set[0: max_len], sentence_tag[0:max_len]
    # 句子长度小于设定长度则在句子后增加“<PAD>”
    else:
        sentence_set.extend(["<PAD>"] * (max_len - len(sentence_set)))
        sentence_tag.extend(["<PAD>"] * (max_len - len(sentence_tag)))
    return sentence_set, sentence_tag
```

根据以上两个函数实现数据的处理，在 xml 文件中进行读取保存到元组列表中，实现代码如下：

```
def LoadData(file):
    train_data = []
    DOMTree = parse(file)
    sentence_list = DOMTree.documentElement
    # 读取每个样例
    sentences = sentence_list.getElementsByTagName('sentence')
    for sentence in sentences:
        # 样例句子文本
        sentence_text = sentence.getElementsByTagName('text')[0]
        sentence_term = sentence.getElementsByTagName('aspectTerm')
        # 对句子进行分词
        sentence_set = sentence_text.childNodes[0].data.replace('.',
        ↪  '').replace('-', '').lower().split()
```

```
        # 关键词集合
        term_set = []
        for Term in sentence_term:
            term_set.append(Term.getAttribute('term'))
        # 根据关键词集合得到标签序列
        sentence_tag = GetSentenceTag(sentence_set, term_set)
        # 对句子进行截断或增长
        train_data.append(TruncAndPad(sentence_set, sentence_tag, MAX_LEN))
    return train_data
```

2.7.2 BiLSTM

使用 pytorch 中的 lstm 模型建立我们所需要的 BiLSTM 模型，并在类型中初始化 CRF 需要使用到的转移矩阵，并加入到训练参数中，实现代码如下：

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, drop_rate,
    ↪ vocab_size, tag_to_ix, word_weight):
        super(LSTM, self).__init__()
        self.lstm = nn.LSTM(
            input_size=input_size,          # 输出向量大小 也为词向量大小
            hidden_size=hidden_size // 2,   # 隐状态输出向量大小 双向则为
            ↪ 1/2
            num_layers=num_layers,          # 层数
            bidirectional=True,             # 双向
            batch_first=True)               # 是否 batch
        self.word_embeds = nn.Embedding(vocab_size, input_size)
        ↪ # 采用随机初始化的词向量
        # self.word_embeds = nn.Embedding.from_pretrained(word_weight)
        ↪ # 采用训练的词向量
        self.tag_to_ix = tag_to_ix
        self.tag_size = len(tag_to_ix)
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.hidden2tag = nn.Linear(hidden_size, self.tag_size)
        ↪ # 线性层从隐状态向量到标签得分向量
```

```
17          self.transitions = nn.Parameter(torch.randn(self.tag_size,
            ↪  self.tag_size))  # CRF 的转移矩阵表示从列序号对应标签转换到行序号
            ↪  对应标签
18          self.transitions.data[tag_to_ix[START_TAG], :] = -10000
            ↪  # 任意标签不能转移到 start 标签
19          self.transitions.data[:, tag_to_ix[END_TAG]] = -10000
            ↪  # end 标签不能转移到任意标签
20          self.hidden = self.HiddenInit()
            ↪  # 隐藏层初始化
```

2.7.3 CRF

根据损失函数 $-log(P(y^x|x))$，需要计算 $log(\sum_{y'} e^{score(x,y')})$，则需要计算 x 每一条可能的路径 y' 的分数，而采用前向算法，根据公式

$$log(\sum e^{log(\sum e^x)+y}) = log(\sum \sum e^{x+y})$$

得到

$$log(\sum e^{log(\sum e^{score(x_i,y_i)})+T_{i,i+1}+P_{i+1,y_{i+1}}}) = log(\sum \sum e^{score(x_i,y_i)+T_{i,i+1}+P_{i+1,y_{i+1}}})$$

则对于词 x_{i+1} 的路径分数的 $logsumexp$，可计算词 x_i 的路径分数的 $logsumexp$ 得到。实现代码如下：

```
1       def ForwardAlg(self, feats):
2           init_alphas = torch.full([feats.shape[0], self.tag_size],
            ↪  -10000.).cuda()
3           # 开始标签的转换得分为 0
4           init_alphas[:, self.tag_to_ix[START_TAG]] = 0.
5           # 输入的每个句子都进行前向算法
6           forward_var_list = []
7           forward_var_list.append(init_alphas)
8           # 每个句子从句首开始迭代
9           for feat_index in range(feats.shape[1]):
10              # 迭代到某一词的 logsumexp
11              tag_score_now = torch.stack([forward_var_list[feat_index]] *
                ↪  feats.shape[2]).transpose(0, 1)
12              feats_batch = torch.unsqueeze(feats[:, feat_index, :],
                ↪  1).transpose(1, 2)
```

```
            # 新词的所有转移路径
            next_tag_score = tag_score_now + feats_batch +
            ↪  torch.unsqueeze(self.transitions, 0)
            forward_var_list.append(torch.logsumexp(next_tag_score, dim=2))
        # 加上 end 标签的得分
        terminal_var = forward_var_list[-1] +
        ↪  self.transitions[self.tag_to_ix[END_TAG]].repeat([feats.shape[0],
        ↪  1])
        # 每个句子进行 logsumexp 得到最终结果
        alpha = torch.logsumexp(terminal_var, dim=1)
        return alpha
```

对于给定序列 tags 的得分，可简单地使用以上公式进行迭代计算，实现代码如下

```
    # 给定序列 tags 的得分
    def Score(self, feats, tags):
        score = torch.zeros(tags.shape[0]).cuda()
        tags = torch.cat([torch.full([tags.shape[0], 1],
        ↪  self.tag_to_ix[START_TAG]).long().cuda(), tags], dim=1)
        for i in range(feats.shape[1]):
            # 第 i 个词得到的 feats 二维张量
            feat = feats[:, i, :].cuda()
            score = score + \
                    self.transitions[tags[:, i + 1], tags[:, i]].cuda() +
                    ↪  feat[
                        range(feat.shape[0]), tags[:, i + 1]].cuda()
        score = score + self.transitions[self.tag_to_ix[END_TAG], tags[:,
        ↪  -1]]
        return score
```

根据以上实现的函数可得到训练需要的损失函数，实现代码如下

```
    def LossFuction(self, sentences, tags):
        feats = self.GetFeatsBatch(sentences)
        forward_score = self.ForwardAlg(feats)
        gold_score = self.Score(feats, tags)
```

```
        # 所有输出句子的误差和作为结果
        return torch.sum(forward_score - gold_score)
```

在如上的训练模型可采用 pytorch 的优化器 optim.SGD 进行训练，具体训练代码见源码。

3 实验结果及分析

3.1 参数调整

实验采用的训练参数具体如表 3,在设定的训练参数中将下载的训练集 Laptops_Train.xml 进行训练, 训练中 Loss 随迭代次数的变化如下图，可见随着迭代，损失函数在不断缩小，在

表 3: 训练参数-1

参数	大小
词向量维度 (输入向量维度)	100
BiLSTM 层输出向量维度	200
BiLSTM 层层数	1
Drop Out Rate	0.6
学习率	1e-5
句子统一长度	30
权重衰减率	1e-4

1500 次迭代后 Loss 下降到接近 4500，而在到达 1500 次迭代后 Loss 的下降速度变缓。

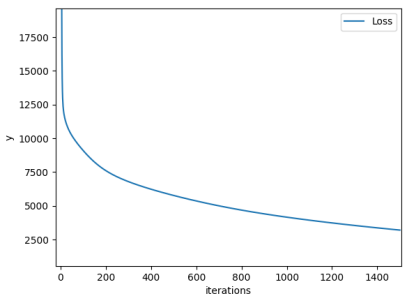

图 30: Loss 随迭代次数变化

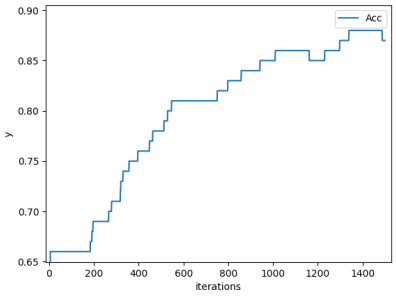

图 31: 验证集准确率随迭代次数变化

在此过程中评估验证集 laptops-trial.xml 的准确度，只有当句子的标注全部正确时，即

提取的关键词全部正确时该例子句子才判断为正确，正确率随迭代次数的变化如图**??**，在迭代次数为 1500 左右，可知虽然 Loss 在下降，但其在验证集上的准确率已保持在 86% 上下浮动。

在该参数下的继续训练，当迭代次数达到 5000 时，此时的 Loss 已下降到 800 以下，而准确率到达 94% 后开始下降至 92%，已出现过拟合现象。为了缓解该现象以提高在验证集的正确率，选择降低 Drop Out Rate 与权重衰减率，再进行训练，修改后的参数如表 4，当迭代次数达到 5000 时，此时的 Loss 已下降到 120 以下，而最终修改参数后得到的准确率稳定在 93%。考虑到验证集的代表性有限，该准确率已达到实验要求，不再进行调整。

表 4: 训练参数-2

参数	大小
词向量维度 (输入向量维度)	100
BiLSTM 层输出向量维度	200
BiLSTM 层层数	2
Drop Out Rate	0.3
学习率	1e-5
句子统一长度	30
权重衰减率	1e-5

3.2 词向量的比较

本次实验中词向量有三种训练方式：1）随机初始化，作为网络参数训练；2）将训练集与验证集中的句子使用 Word2Vec 模块进行训练得到词向量；3）使用已训练好的 Glove 词向量。以下对这三种方式的词向量实现进行分析比较。

在 BiLSTM 网络结构定义中，$\text{self.word\_embeds} = \text{nn.Embedding}(\text{vocab\_size}, \text{input\_size})$，会将输入的每个词随机初始化成一个词向量。而在训练过程中，会去更新词向量。这样的词向量实现方式在本次实验中无疑是最好的，因为最终训练得到的词向量是基于 Loss 函数进行更新的，能够最大程度地帮助提取关键词，关键词与关键词之间的相似度会更高，这也满足实验的目的。但这样的词向量脱离了文本，是在训练过程得到的，不能代表词语实际含义的相似度，这是与词向量的本质违背的。

如果使用训练集与验证集的句子子使用 Word2Vec 模块来生成词向量，生成的词向量可以表示词语实际含义的相似度，但由于训练生成词向量的语料较少，生成的词向量特征表示程度较差，因此训练效果会比较差。

由于 Glove 词向量是基于足够大的语料库训练得出的，因此在实验中是最符合词向量特征的，其训练效果相比较于第二种方式较好，但由于选取的词向量文件为 **glove.6B.100d**，

只有 40000 个词，在使用时存在有大量缺失的词向量，而这些词采用了随机初始化的方式，导致最终效果比不上第一种方式。三种实现方式的最终在验证集的准确率如表 5

表 5: 词向量实现的准确率比较

Embedding 训练	93%
Word2Vec 训练	84%
Glove	88%

4 参考文献

Very Deep Convolutional Networks for Large-Scale Image Recognition - arXiv:1409.1556 [cs.CV]

Deep Residual Learning for Image Recognition - arXiv:1512.03385 [cs.CV]

Bidirectional LSTM-CRF Models for Sequence Tagging - arXiv:1508.01991 [cs.CL]

5 分工

在实验前，我们详细讨论了 CNN 与 RNN 实现的大致思路，一同在网络上搜索了相关论文，以决定具体实现的网络架构与代码思路。最终，CNN 主要代码由朱益澄同学完成，RNN 主要代码由朱煜同学完成，实验报告共同完成。