中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

（期中作业）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 计算机科学与技术
小 组 成 员	17341067 江金昱　16327143 仲逊
时　　间 ：	2019　年　11　月　4　日

	文件名	文件描述
RNN	pure_rnn.py	包含用纯 RNN 训练的过程
	pure_rnn_model.py	基础的 RNN 模型
	lstm.py	包含基础的 LSTM 模型
	model.py	基础的 LSTM 模型
	lstm_with_word_embedding.py	加入了 word embedding 后的 LSTM
CNN	origin.py	经典的 CNN 模型
	origin_dropout.py	加入 dropout 后的模型
	origin_GAP.py	加入全局平均池化后的模型
	origin_GAP_dropout.py	加入 dropout 和全局平均池化后的模型

RNN 部分：

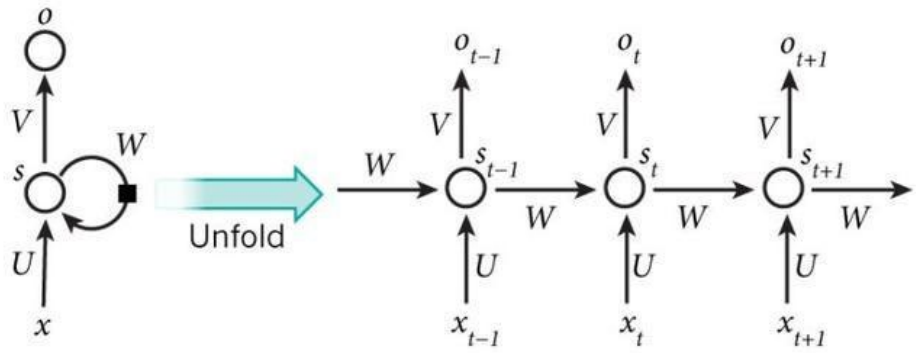
一、实验原理：

什么是 RNN?

RNN，中文名叫循环神经网络，因对序列数据建模的需要而产生。传统的神经网络的输出只与本次输入的数据有关，而 RNN 则可对前面时刻输入网络的信息进行记忆，找到过往时刻数据的相关性综合过往时刻的信息进行输出。RNN 适用于预测一些具有“时间序列”性质的数据，如 NLP 中的词语预测。

RNN 的基本结构：

一个基本的 RNN 结构如下图所示：



在t时刻，每个节点都有一个隐藏状态s_t，包含了该时刻和过往时刻的综合信息。每个节点都共享权重U, W, V。刚开始，我们会随机初始化一个隐层状态记为s_0，此后根据各个节点

的输入，我们可以计算出每个节点的状态s_t，公式为

$$s_t = f(W \cdot s_{t-1} + U \cdot x_t)$$

其中$f(\cdot)$是激活函数，通常可选的激活函数有$tanh, sigmoid, RELU$等。得到了状态s_t后，我们便可以根据以下公式计算该层节点的输出o_t：

$$o_t = \sigma(V \cdot h_t)$$

其中，σ是激活函数，根据不同的训练任务场景可选用不同的激活函数，如在分类任务中常选用$softmax$作为激活函数，$softmax$使得各个维度的总和为 1，可表示为每一维的概率。

由于每一层的结构十分相似再加上权重共享，我们可以将网络简化为上图中左半部分的结构。这也为我们的代码实现提供了方便，只要定义了一层的网络结构，训练的时候只要保存每一层的状态即可循环计算下一层的状态与输出。

RNN 的梯度更新：

我们需要一个损失函数来评估我们预测的结果，记为$E_t(y_t, \hat{y}_t)$，他表示真实值y_t与预测值 $\hat{y}_t$之间的误差，这里的y_t就是上图中的o_t。我们的目标是计算误差关于参数U、V、W的梯度，然后使用一定的参数更新公式（通常为$W = W - \alpha \Delta W$，α是学习率ΔW是梯度）来更新参数。梯度的计算公式分别为

$$\frac{\partial E}{\partial U} = \sum_t \frac{\partial E_t}{\partial U}、\ \frac{\partial E}{\partial V} = \sum_t \frac{\partial E_t}{\partial V}、\ \frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

以求E_t关于U、V、W的梯度为例子，为了求其梯度，需要运用求导的链式法则，即

$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial V}$$

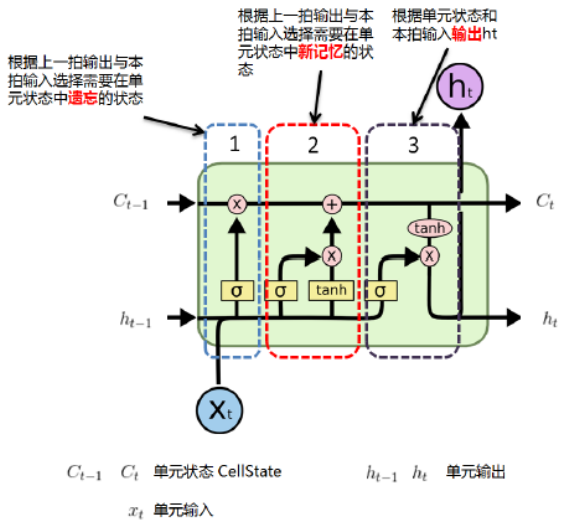
$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial s_t}\left(\prod_{m=k+1}^{t} \frac{\partial s_m}{\partial s_{m-1}}\right)\frac{\partial s_k}{\partial W}$$

$$\frac{\partial E_t}{\partial U} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial s_t}\left(\prod_{m=k+1}^{t} \frac{\partial s_m}{\partial s_{m-1}}\right)\frac{\partial s_k}{\partial U}$$

二、创新：

（1）LSTM

在 RNN 的基础上，引入 LSTM 结构，一个典型的 LSTM 结构如下图所示：



上图中省略了该层的输出y_t，因为y_t是通过该层的输出h_t乘以权重再通过激活函数确定的，与 RNN 一致就不再赘述了。LSTM 引入了“门机制”，有三种门，即“遗忘门”，“输入门”，“输出门”。

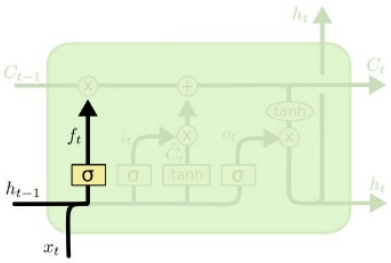
LSTM 相比于 RNN 有如下优点：

1. 因为最终的细胞状态C_t是由一系列系数连加得到的，一定程度上解决了梯度消失的问题。

2. LSTM 可选择性的记住过往重要的信息而遗忘次要的信息，比 RNN 多出来的细胞状态C_t使得模型在处理“长数据”上更有优势

遗忘门：

遗忘门的功能是以一定的概率控制是否遗忘上一层的细胞状态。

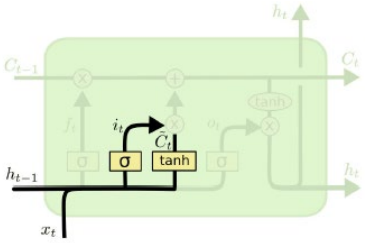


$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

最后输出的f_t向量在每一维上是 0 或者 1，再与C_{t-1}做点乘，即可控制C_{t-1}的保留程度。

输入门：

输入门的功能是控制当前生成的隐状态的保留程度



$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

$$\widetilde{C_t} = \tanh(W_C h_{t-1} + U_C x_t + b_C)$$

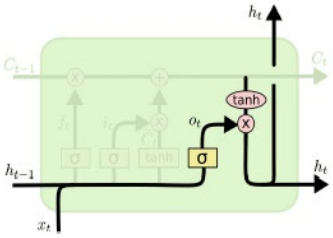
最后$i_t \cdot \widetilde{C_t}$表示该层产生的隐状态保留的程度

门状态更新：

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \widetilde{C_t}$$

C_t就是该层最终的细胞状态

输出门：



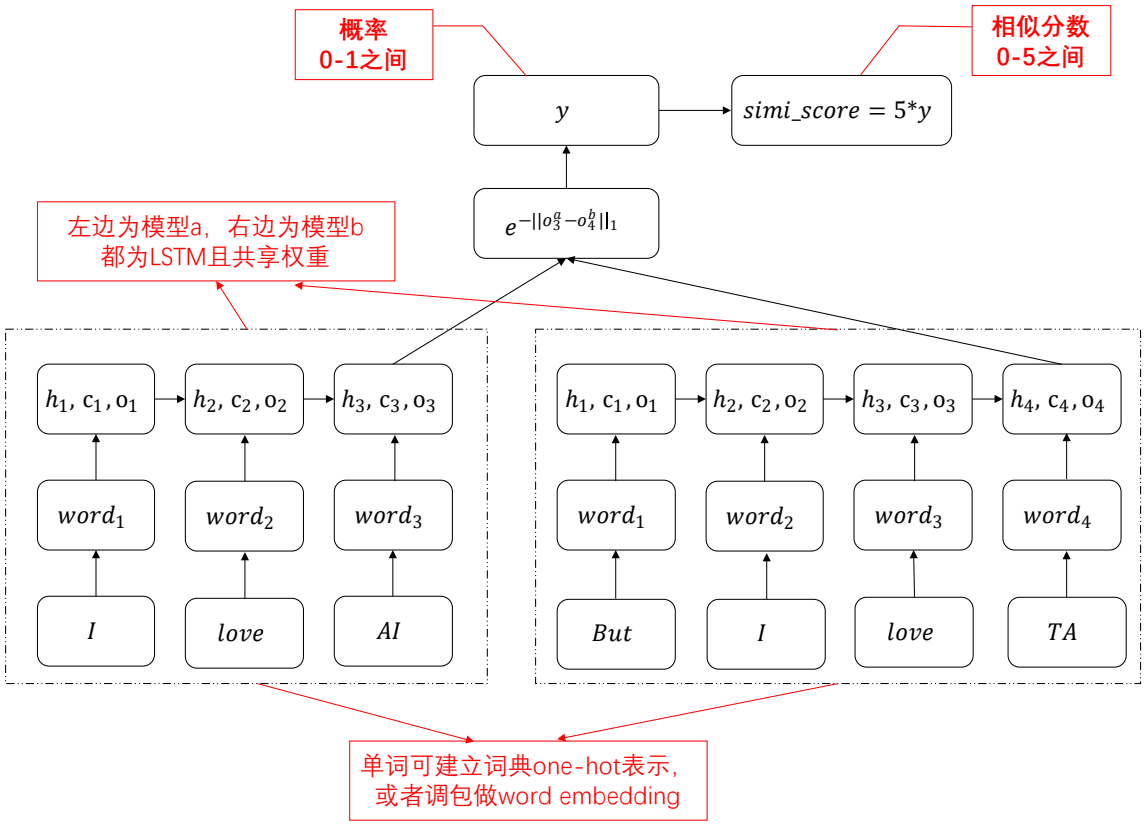
$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$

$$h_t = o_t \cdot \tanh(C_t)$$

（2）dropout

为了增强模型的泛化能力，引入了“Dropout”机制，即使得训练过程中神经元有一定机率失效，即不再参与训练的过程。该方法可在 pytorch 的 LSTM 函数的参数中 dropout 中设置。

三、网络结构及相应的代码展示：



注：以上模型借鉴了 Siamese 模型**[1]**

（1） 数据预处理：

```
# ------------------数据处理----------------------------------------
# 判断使用GPU 还是 CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Now using: "+str(device))
# 开始时间
prerocess_start = time.time()
# 加载数据，进行切词，去标点
train_data_raw = load_data('data/sts-train.txt')
dev_data_raw = load_data('data/sts-dev.txt')
# 获得词典
word_dict = get_word_dict(train_data_raw, dev_data_raw)
dict_len = len(word_dict) + 1
print("The length of the word_list is " + str(dict_len))
# 将单词表示成 one-hot 向量
train_data = get_word_vect(train_data_raw, word_dict)
dev_data = get_word_vect(dev_data_raw, word_dict)
print("Done preprocessing data, spent "+timeSince(prerocess_start))
# ---------------------以上是数据处理部分----------------------------
```

如果采用了 word embedding 则可使用 gensim 调用训练好的模型将单词直接表示成向量:

```
# 获取预训练好的模型
model = word2vec.Word2Vec.load('word2vec_corpus/word2vec.model')
```

```
train_data=get_word_vect(train_data_raw,word_dict,model)
dev_data=get_word_vect(dev_data_raw,word_dict,model)
```

(2) 模型设置

```
# ----------------------模型参数--------------------------
# 输入向量的特征维度
embed_size=dict_len
# 批大小
batch_size=1
# 隐藏状态维度
hidden_size=128
# 每一时刻 LSTM 堆叠的层数
num_layers=1
# dropout 概率
drop_out_prob=0
# 初始化模型
lstm = my_lstm(embed_size,batch_size,hidden_size,num_layers,drop_out_prob)
lstm=lstm.to(device)
# -----------以上是模型参数部分----------------------------
```

```
# LSTM 结构
class LSTMEncoder(nn.Module):
    # 初始化 LSTM 模型
    def __init__(self, embed_size,batch_size,hidden_size,num_layers,drop_out_prob):
        super(LSTMEncoder, self).__init__()
        self.embed_size = embed_size
        self.batch_size = batch_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.dropout = drop_out_prob
        self.lstm = nn.LSTM(input_size=self.embed_size,
hidden_size=self.hidden_size,num_layers=self.num_layers,dropout=self.dropout)
    # 初始化隐藏层状态以及细胞状态
    def initHiddenCell(self):
        rand_hidden =Variable(torch.randn( self.num_layers, self.batch_size,
self.hidden_size).to(device))
        rand_cell = Variable(torch.randn( self.num_layers, self.batch_size,
self.hidden_size).to(device))
        return rand_hidden, rand_cell
    # 前向传播
    def forward(self, input, hidden, cell):
        input = input.view(1, 1, -1)
```

```
        output, (hidden, cell) = self.lstm(input, (hidden, cell))
        return output, hidden, cell
```

（3） 优化器选择（权重更新方法）、损失函数选择

```
# 优化器（梯度更新）
optimizer = torch.optim.Adadelta(filter(lambda x: x.requires_grad,
lstm.parameters()), lr=learning_rate)
# 损失函数采用均方误差
criterion = nn.MSELoss()
criterion=criterion.to(device)
```

（4） 训练方法

封装了一次句子的训练过程，用的时候可以直接调用

```
# 使用LSTM对句子进行训练
class my_lstm(nn.Module):
    def __init__(self,embed_size,batch_size,hidden_size,num_layers,drop_out_prob):
        # 初始化
        super(my_lstm, self).__init__()
        self.encoder =
LSTMEncoder(embed_size,batch_size,hidden_size,num_layers,drop_out_prob)

    # 前向传播
    def forward(self, s1, s2):
        # 初始化隐层状态以及细胞状态
        h1, c1 = self.encoder.initHiddenCell()
        h2, c2 = self.encoder.initHiddenCell()
        # 一个一个把单词输入进去
        o1=0
        o2=0
        for i in range(len(s1)):
            o1, h1, c1 = self.encoder(s1[i], h1, c1)
        for j in range(len(s2)):
            o2, h2, c2 = self.encoder(s2[j], h2, c2)
        # 计算exp(-||o1_o2||_1)后还要乘以5映射到0-5
        output= 5*torch.exp(-torch.norm(o1-o2,p=1))
        return output
```

调用封装的类进行训练，每次都要执行清除梯度，然后计算输出与真实值之间的误差反向传播得到梯度后更新梯度

```
# 清除梯度
optimizer.zero_grad()
# 输出
output = lstm(sentence_tensor1, sentence_tensor2)
```

```
original_score = Variable(original_score)
original_score.to(device)
# 误差反向传播
loss = criterion(output, original_score)
loss.backward()
# 更新梯度
optimizer.step()
```

（5） 动态调整学习率

当本次训练误差比上次训练误差大的时候将学习率减半

```
# 动态调整学习率
if epoch >= 3 and all_train_mean_loss[-1] > all_train_mean_loss[-2]:
    learning_rate = learning_rate / 2
    print('!!!!!!!!' + str(learning_rate))
    for param_group in optimizer.param_groups:
        param_group['lr'] = learning_rate
```

（6） 模型保存

在验证集上验证，如果本次验证集的误差小于此前最好的误差则保存本次的模型

```
# 查看与最好的 loss 的差距
if np.mean(valid_loss) < best_loss:
    best_loss = np.mean(valid_loss)
    torch.save(lstm.state_dict(), "lstm_params.pkl")
```

四、结果分析：

以下结果使用 STS 中的 correlation.pl 在测试集(sts-test.txt)_上计算皮尔逊相关系数来作为最终验证模型的评测指标

在训练的过程中动态调整学习率，即当上次训练损失要大于上上次训练损失的时候，学习率减半。

***经尝试在经过参数调整后，经足量的训练轮数后区别不大，因此报告中仅展示模型间的效果区别。**

基础参数		
参数名	**解释**	**值**
Num_layers	Rnn 每一时刻堆叠的层数	1
Drop_out_prob	神经元失活的概率，仅当 num_layers>1 的时候有效	0

Hidden_size	隐藏状态向量的特征的维度	128
Embed_size	输入的特征向量的特征的维数	字典大小/embedding 后的大小
Learning_rate	学习率	初始化为 0.1，后经动态调整

其他基础设置	
损失函数	**均方误差（torch.nn.MSELoss）**
优化器（梯度更新方法）	**Adadelta**

1. 基础 RNN 模型

使用基础参数运行结果记录

Epoch（训练轮数）	训练集 MSE	DEV 验证集 MSE
1	4.470229	3.113677
5	2.682181	2.482660
10	2.452715	2.505848
20	2.505528	2.612407
30	2.560050	2.485946

采用在验证集上效果最好的模型在测试集上验证，

最终在测试集上的 Pearson 相关系数为：**0.2371**

可以看到模型在训练集上在 20-30epoch 时已经拟合了，而验证集的误差在 5 个 epoch 后变化已经不大，不论是偏差和方差都较大。

2. 引入 LSTM 机制后的模型

使用基础参数运行结果记录

Epoch（训练轮数）	训练集 MSE	DEV 验证集 MSE
1	2.788375	2.363721

5	1.943528	2.229069
10	1.657909	2.027339
20	1.248842	1.880080
30	0.947972	1.793984
85	0.475499	1.683241

采用在验证集上效果最好的模型在测试集上验证，

最终在测试集上的 Pearson 相关系数为：**0.48066**

可以看到引入 LSTM 后，在训练集上的拟合能力增强，在验证集上的泛化能力也增强，此时模型相对于 RNN 具有较低的方差和偏差。

3. 当 LSTM 多余一层时（体现在 pytorch 中 LSTM 模型参数 num_layers>1），引入 dropout 后的模型

由于堆叠层数的加深，模型容易出现过拟合，这里设置了 dropout=0.2

使用基础参数运行结果记录

Epoch（训练轮数）	训练集 MSE	DEV 验证集 MSE
1	2.551955	2.540359
5	2.095119	2.560954
10	1.991081	2.954000
20	1.787526	5.105361
30	1.5044667	3.979506
50	0.921095	3.580214
100	0.540754	2.595597

最终在测试集上的 Pearson 相关系数为：**0.03857**

可以看到 ，当加深网络层数，即使加入了 dropout，但是只是在训练集上的 MSE 减少了，验证集上的 MSE 不降反升，泛化效果较差，模型面临过拟合的问题，具有低偏差，高方差

4. 在输入数据上调用预训练的模型，将单词处理成向量矩阵(即引入 word embedding)

使用基础参数运行结果记录

Epoch（训练轮数）	训练集 MSE	DEV 验证集 MSE
1	2.966341	2.270921
5	1.793359	1.945135
10	1.557647	1.777588
20	1.334712	1.706778
30	1.236174	1.660273
50	1.176918	1.653160
100	1.173570	1.647994

最终在测试集上的 Pearson 相关系数为：**0.50225**

可以看到加入 embedding 后模型拟合较快，因为 embedding 在一定程度上完成了一部分的单词编码工作，因此能加速模型的训练。另外在最终结果上，加入了 embedding 的皮尔逊系数也是最高的。模型具有较低的偏差和方差。

CNN 部分：

一、实验原理：

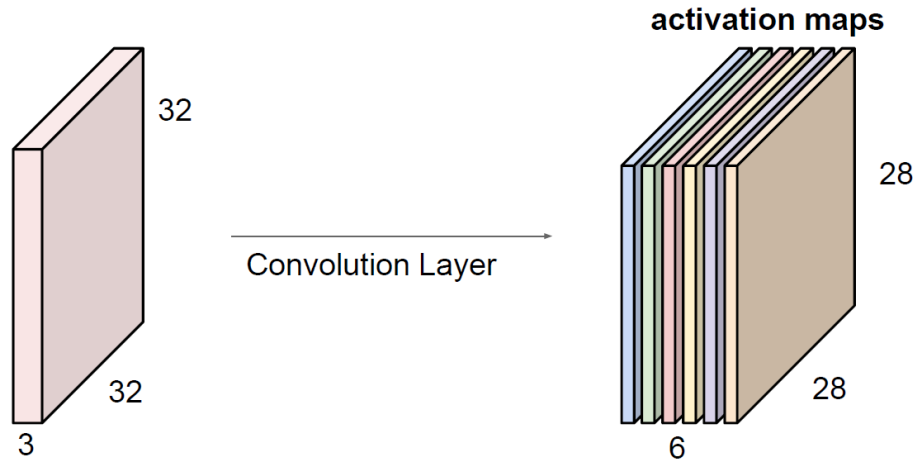
什么是 CNN?

Convolutional Neural Networks 中文名为卷积神经网络，是包含卷积操作的一种前馈神经网络，它的神经元能够响应一个窗口大小覆盖范围内的单元，因此在图像处理方面有着比较出色的表现。

卷积操作在信号与系统和数字图像处理课程中用的比较多，在数字图像处理中我们已经了解了多种滤波器，都是使用卷积操作来对图像进行特征提取，在传统的图像处理中，这样的滤波器都是人为设计好的，比如高斯平滑，锐化和边缘提取等等，滤波器窗口的每个位置值已经确定，而 CNN 的思想就来源于能否不人为设计，而是让网络自己学习滤波器的参数，以达到特征提取的效果

CNN 的基本结构：

CNN 由卷积层和池化层构成，一个卷积层结构如下图所示：

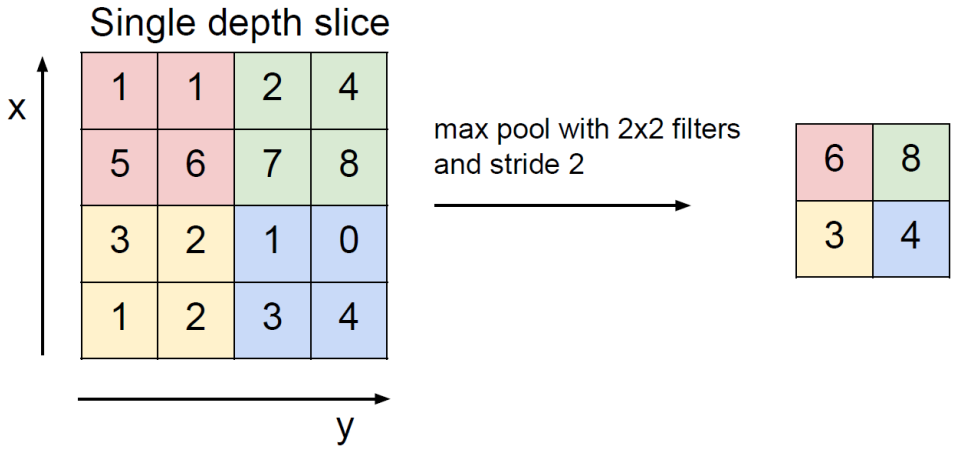


这里的示例采用的是 cifar-10 数据集的图片，图片大小是 32×32，彩色图片又分为 RGB 3 个通道（可以认为有 3 张灰度图），因此输入层为 32×32×3，经过卷积层提取特征后输出了 6 张大小为 28×28 的特征图片，即输出层为 28×28×6。

由此我们可以形式化定义结构，一个卷积层有D个大小为$M \times N$的特征图片输入（输入数据为$M \times N \times D$维），输出P张大小为$M' \times N'$的特征图片（输出数据为$M' \times N' \times P$维），一个卷积窗口叫做卷积核，共有$D \times P$个二维卷积核。那么计算第i个输出特征图Y_i的方式就是先用卷积核$W_{i,1}$，$W_{i,2}$，… $W_{i,D}$分别对输入特征图X_1，X_2，… X_D进行卷积，然后求和加上标量偏置项b_i得到Z_i，再通过激活函数得到输出Y_i：

$$Y_i = f(Z_i) = f(\sum_{d=1}^{D} W_{i,d} \otimes X_D + b_i)$$

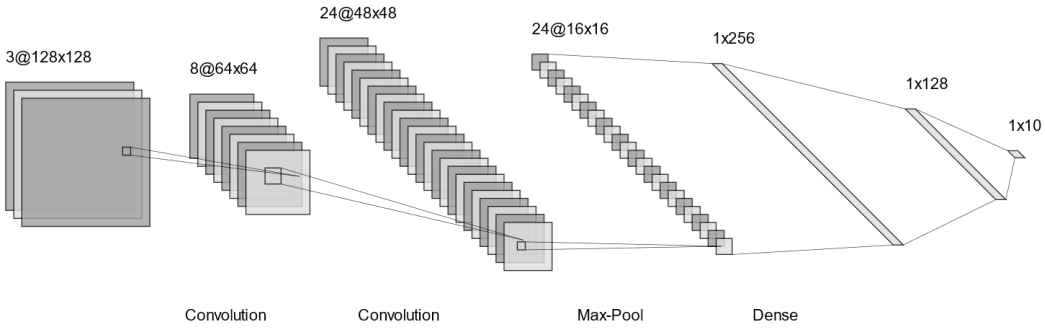
卷积层虽然可以显著减少网络中参数的量，但特征映射输出的维度仍然很高。因此又引入了池化层对输入特征映射组进行下采样，进一步的筛选特征。一个典型的池化层结构如下：



pooling 是为了提取一定区域的主要特征，并减少参数数量，防止模型过拟合，最常用到的池化类型是最大池化（max pooling），大小与步长均设置为 2，即在 2×2 的区域内选择

最大值作为这个区域的特征。此外也有均值池化（average pooling）等其他类型可以选用。池化层的可解释性并不好，这里可以将它理解为一种降采样，例如原本的高清图片很大，降低图片质量可以压缩图片大小（即此处的减少参数数量），但是图片的特征依然很明显。因此池化层能够起到保留显著特征，降低特征维度的作用。

CNN 的结构由卷积层，池化层和全连接层交替堆叠而成，一个典型的结构如下图所示：



我们将连续 M 个卷积层和 b 个池化层（M 通常设置为 2~5，b 为 0 或 1）称为一个卷积块，一个 CNN 中首先堆叠 N 个连续的卷积块，最后将特征图像向量化后，再连接 K 个全连接层（K 一般为 0~2）形成完整的网络结构。即前面的多个卷积块负责图像特征的提取，而最后的全连接层根据输入特征进行分类/回归。

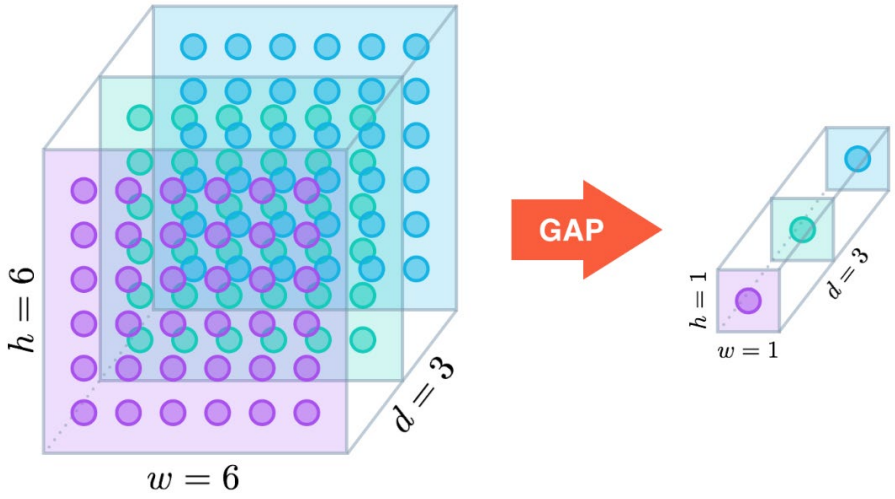
二、创新：

（1）全局平均池化（Global Average Pooling）

Golbal Average Pooling（GAP）在《Network in Network》（arXiv:1312.4400）中首先被提出，长期以来，最后的全连接层一直是 CNN 分类网络的标配。全连接层的作用就是将最后一层卷积得到的特征图拉伸成向量，然后对这个向量做乘法降低其维度，最后输入到 softmax 层中得到对应的每个类别的得分。全连接层的问题在于参数数量较多，因此除了训练比较慢以外还有可能产生过拟合。

既然全连接层目的是使特征图的维度减少，进而输入到 softmax 层分类，那么能否用有相似功能的池化层来替代它呢？这就是 GAP 提出的契机。

Network in Network 使用 GAP 来取代了最后的全连接层，直接实现了降维，更重要的是极大地减少了网络的参数(CNN 网络中占比最大的参数其实后面的全连接层)。Global average pooling 的结构如下图所示：



可以看出 GAP 将最后一层输出的特征图全局取平均得到一个特征点作为整个特征图的代表，大大减少了网络参数。尽管看起来这个操作有些草率，但它直接从最后的输出特征图的数量（即 channel 数）下手，相当于剔除了全连接层黑箱子操作的特征，直接赋予了每个 channel 实际的类别意义。需要注意的是，使用 GAP 有可能造成收敛变慢。

（2）dropout

为了提高模型的泛化能力减少过拟合现象，除了全局平均池化，还有 dropout 机制可以采用，即使得训练过程中神经元有一定机率失活，不再参与训练的过程。通俗地可以解释为网络训练时用较少的神经元，而在实际预测时有更多的神经元工作（训练时增加难度而考试时却比较容易），如此能够得到更准确的结果。该方法可在 pytorch 的 nn 包中直接设置。

（3）Batch Normalization

Batch Normalization（批量标准化，简称 BN）是近些年来深度学习优化中一个重要的手段。BN 的优点主要有：加速训练过程、可以使用较大的学习率、允许在深层网络中使用易导致梯度消失的激活函数和具有一定正则化效果四点。

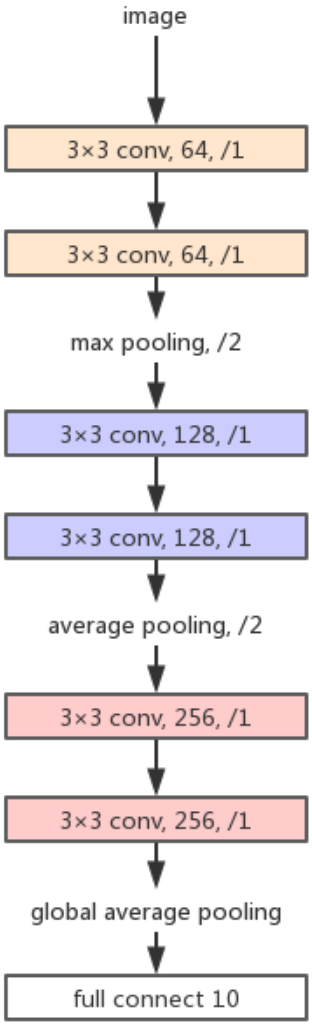
一次训练过程里面包含 m 个训练实例，其具体 BN 操作就是对于隐层内每个神经元的激活值来说，进行如下变换：

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

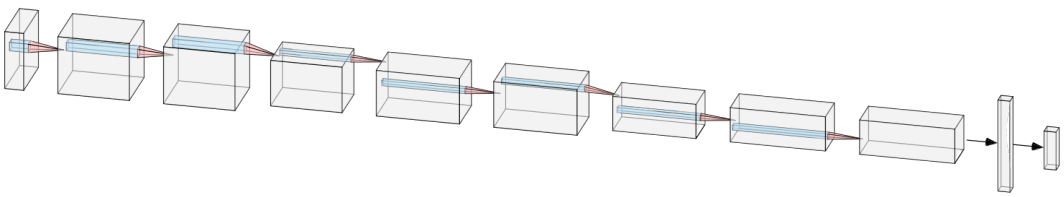
BN 方法也可以直接在 pytorch 的 nn 包中直接设置 BN 层。

三、网络结构及相应的代码展示：

自己搭建的 CNN 结构图如下所示：



上图采用的是 ResNet 风格的网络图，偏描述性，同时我制作了一张 AlexNet 风格的网络图如下，从中能够较为直观地看出特征图经过每一层后的变化情况，但由于篇幅所限，可能清晰度不佳。



注：ResNet 的网络结构图可以参照原文**[2]**

(1) 数据预处理：

```
# 加载数据集
batch_size=100
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
# train=True 加载训练集，否则为测试集，transform 为特定格式的数据变换
trainset = torchvision.datasets.CIFAR10(root='./', train=True,
                                        download=False, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./', train=False,
                                       download=False, transform=transform)
# windows 下注意线程问题，需要将 num_workers 改为默认值 0
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=0)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)
# cifar-10 数据集的类别
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

(2) 模型设置

```
# 定义网络结构
class Net(nn.Module):


    # 构造函数，定义了网络的基本结构
    def __init__(self):
        # 使用 Net 的父类的初始化方法，即运行 nn.Module 的初始化函数
        super(Net, self).__init__()
        # 卷积层 1：输入为图像(rgb3 通道图像)，输出为 64 张特征图，卷积核为 3x3，padding 为 1
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        # 卷积层 2：输入为 64 张特征图，输出为 64 张特征图，卷积核为 3x3，padding 为 1
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, 3, padding=1)
        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
        self.conv6 = nn.Conv2d(256, 256, 3, padding=1)
        self.maxpool = nn.MaxPool2d(2, 2)
        self.avgpool = nn.AvgPool2d(2, 2)
        # 用 global average pooling 代替最后的全连接层
        self.globalavgpool = nn.AvgPool2d(8, 8)
        # BatchNorm2d 常用于卷积网络中(防止梯度消失或爆炸)，设置的参数就是卷积的输出通道数
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
```

```
        self.bn3 = nn.BatchNorm2d(256)
        # drop 训练的常用技巧，训练时随机使一部分神经元失活，能够防止过拟合
        self.dropout50 = nn.Dropout(0.5)
        self.dropout10 = nn.Dropout(0.1)
        self.fc = nn.Linear(256, 10)

    # 前向传播过程，定义了各个网络层之间的连接方式
    def forward(self, x):
        # input x->卷积层1->激活函数ReLU->正规化BatchNorm
        x = self.bn1(F.relu(self.conv1(x)))
        x = self.bn1(F.relu(self.conv2(x)))
        x = self.maxpool(x)
        # x = self.dropout10(x)
        x = self.bn2(F.relu(self.conv3(x)))
        x = self.bn2(F.relu(self.conv4(x)))
        x = self.avgpool(x)
        # x = self.dropout10(x)
        x = self.bn3(F.relu(self.conv5(x)))
        x = self.bn3(F.relu(self.conv6(x)))
        # 用global average pooling 代替最后的全连接层
        x = self.globalavgpool(x)
        # x = self.dropout50(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

net = Net()
```

（3） 设置优化器、损失函数和运行设备

```
# 使用交叉熵损失函数
criterion = nn.CrossEntropyLoss()
# 优化器使用Adam，初始学习率为0.001
optimizer = optim.Adam(net.parameters(), lr=0.001)
# 指定运行的设备，多卡机器可以通过cuda:后的数字指定哪张卡
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
# 将网络中的数据转换为对应设备的类型(GPU 则需要转换成 cuda 数据类型，CPU 不需要)
net.to(device)
```

（4） 训练

```
# 训练计时
start = time.clock()
# 运行epoch 数
EPOCH_NUM = 100
# 准确率作图
accuracy_range = []
```

```
for epoch in range(EPOCH_NUM):
    running_loss = 0.
    # 获取索引和数据
    for i, data in enumerate(trainloader, 0):
        # data 包含数据和标签信息，分别赋值给 inputs 和 labels
        inputs, labels = data
        # 将数据类型转换为对应设备类型（用 GPU 则转为 cuda 类型，CPU 不需要）
        inputs, labels = inputs.to(device), labels.to(device)
        # 梯度清 0，因为反向传播过程中梯度会累加上一次循环的梯度
        optimizer.zero_grad()
        # 将训练数据数据输入网络得到输出
        outputs = net(inputs)
        # 使用交叉熵损失函数计算输出和 Label 的误差值
        loss = criterion(outputs, labels)
        # 反向传播计算梯度
        loss.backward()
        # 执行反向传播后，使用优化器更新参数
        optimizer.step()
        print('[epoch %d] loss: %.4f' % (epoch+1, (i+1)*batch_size, loss.item()))

print('%d epoch Finished Training'%EPOCH_NUM)
end = time.clock()
print("Training Time: ",str(end-start))

torch.save(net, 'cifar10_gap_nodrop.pkl')
```

（5） 预测并计算测试集准确率

```
correct = 0
total = 0
# 在计算图中会自动计算张量的梯度，此处为非训练过程，应指定无需累计梯度
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        # 将图片输入到网络中得到结果
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        # 统计正确数
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on test set: %d %%' % (100 * correct / total))
```

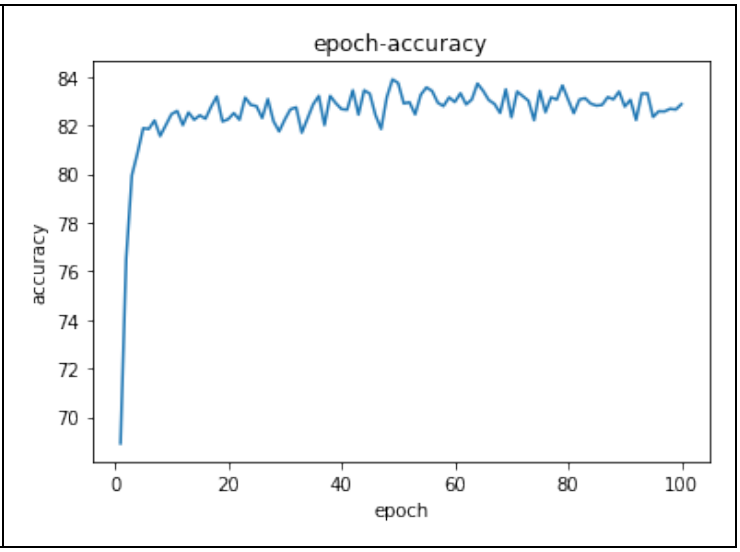
四、结果分析：

以下对比分析了自己搭建的 CNN 网络在经典模型（卷积块+全连接网络），经典模型+dropout，经典模型+全局平均池化和经典模型+全局平均池化+dropout 四种情况下的收敛状况，最终准确率，epoch-准确率的关系等结果。（Batch Normalization 是比较基本的结构，因此四种网络结构中都有使用）

由于网络结构并不复杂，30 个 epoch 时整个模型就基本稳定下来，但为了更多观察 epoch-准确率变化情况，我们将 epoch 设为 100

经典模型：

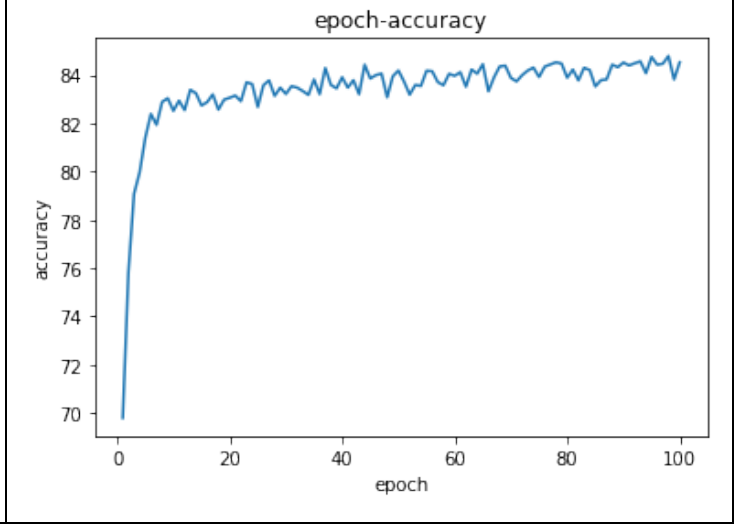
```
Accuracy of the network
on test set: 82 %
Training Time: 2511.1305
Accuracy of plane : 86 %
Accuracy of   car : 96 %
Accuracy of  bird : 73 %
Accuracy of   cat : 65 %
Accuracy of  deer : 77 %
Accuracy of   dog : 71 %
Accuracy of  frog : 90 %
Accuracy of horse : 84 %
Accuracy of  ship : 95 %
Accuracy of truck : 93 %
```



经典模型的收敛速度较快，在不到 10 个 epoch 时已经几近收敛，但效果比较一般，随着 epoch 的增长在测试集上的准确率主要在 81%-83%之间波动，最好的准确率为 83%。

经典模型+dropout

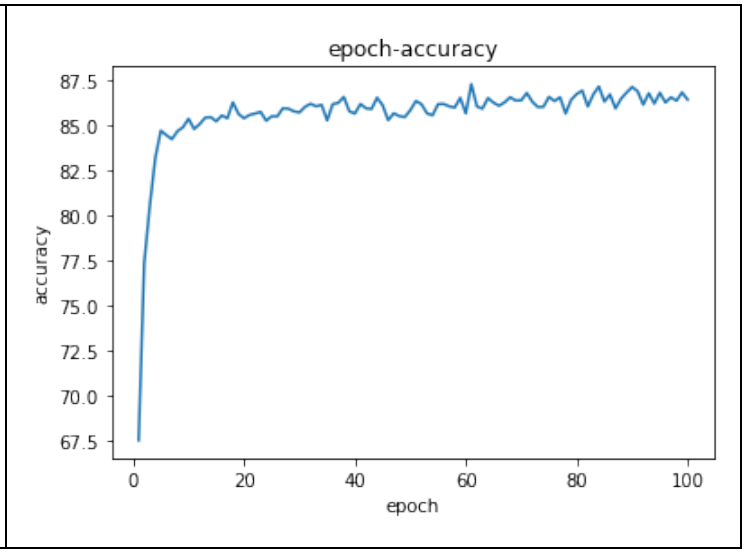
```
Accuracy of the network
on test set: 84 %
Training Time: 2746.8287
Accuracy of plane : 88 %
Accuracy of   car : 93 %
Accuracy of  bird : 89 %
Accuracy of   cat : 72 %
Accuracy of  deer : 81 %
Accuracy of   dog : 75 %
Accuracy of  frog : 92 %
Accuracy of horse : 87 %
Accuracy of  ship : 85 %
Accuracy of truck : 96 %
```



经典模型加上 dropout 之后明显结果比没有加 dropout 要好，稳定后在测试集上的准确率在 83%-85%之间，但缺点也能看出，收敛速度变慢了很多，且训练需要的时间也增加了。

经典模型+全局平均池化

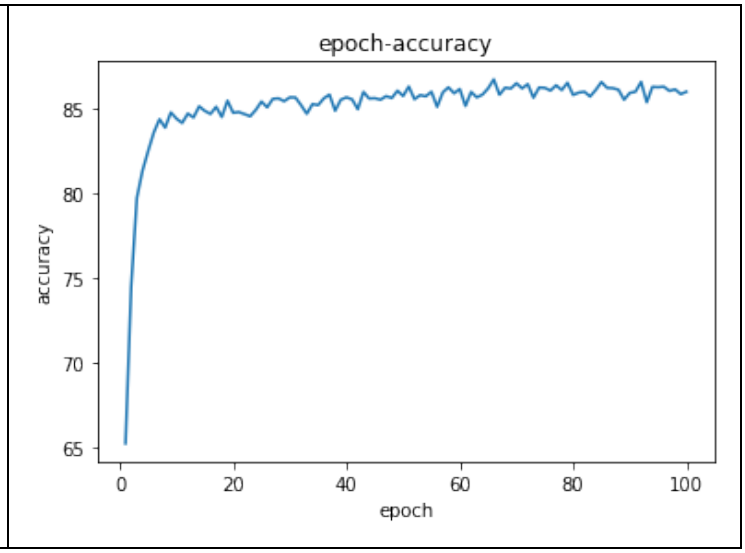
```
Accuracy of the network
on test set: 87 %
Training Time: 2495.2561
Accuracy of plane : 90 %
Accuracy of  car : 100 %
Accuracy of  bird : 76 %
Accuracy of  cat : 67 %
Accuracy of  deer : 87 %
Accuracy of  dog : 75 %
Accuracy of  frog : 83 %
Accuracy of horse : 87 %
Accuracy of  ship : 93 %
Accuracy of truck : 98 %
```



经典模型加上全局平均池化的效果最好，不仅训练时间短，收敛速度也非常快，最终得到的测试集准确率在 87%左右，可以看出 GAP 确实减少了网络参数，加快了收敛速度，并且使得 CNN 分类模型的效果更好。

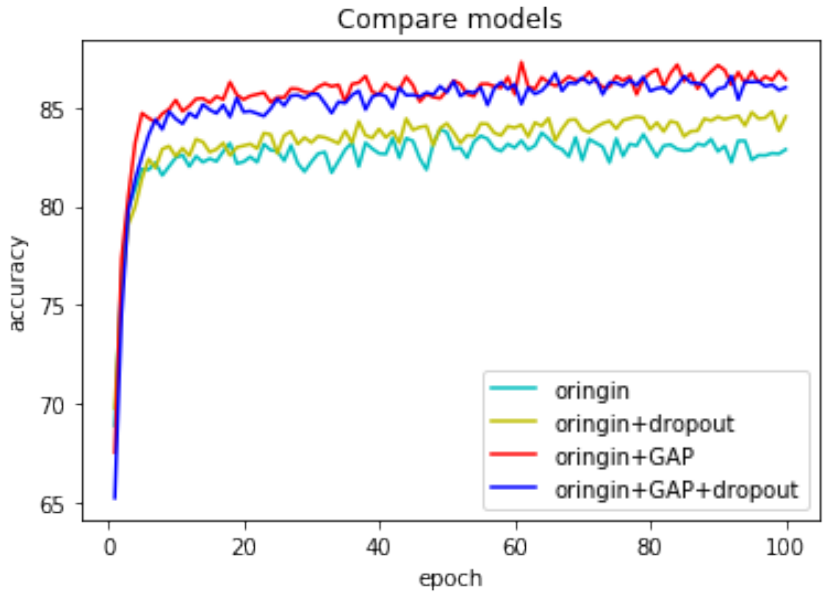
经典模型+全局平均池化+dropout

```
Accuracy of the network
on test set : 86 %
Training Time: 2606.0361
Accuracy of plane : 93 %
Accuracy of  car : 93 %
Accuracy of  bird : 73 %
Accuracy of  cat : 74 %
Accuracy of  deer : 79 %
Accuracy of  dog : 81 %
Accuracy of  frog : 88 %
Accuracy of horse : 93 %
Accuracy of  ship : 85 %
Accuracy of truck : 93 %
```



最后进行了一次在全局平均池化基础上加了 dropout 的尝试，最终结果并没有提升，反而比直接使用 GAP 的结果稍逊一筹，且收敛速度和训练时间都比 GAP 要差一些。

四种网络结构的对比图：



由于四种模型都有使用 Batch Normalization，起到一定的正则化效果，因此虽然最终在测试集上的准确率有差异，但在 100 个 epoch 中都没有出现过拟合的现象。

对比我们发现 dropout 的效果不佳，提升不大且训练耗时较长，收敛较慢。因此我们查找了资料了解到，随着深度学习的发展，Dropout 在现代 CNN 架构中，已经逐步被 BN（Batch Normalization）和 GAP（Global Average Pooling）取代。至于为何 Dropout 不再受青睐，原因主要有两个：一是 Dropout 在卷积层的正则效果有限。相比较于全连接层，卷积层的训练参数较少，激活函数也能很好地完成特征的空间转换，因此正则化效果在卷积层不明显；二则是 Dropout 也过时了，能发挥其作用的地方在全连接层，当代的 CNN 中，全连接层也在慢慢被 GAP 所取代，因为其不但能减低模型尺寸，还可以提升性能。因此在上面的四种模型结构中都使用了 BN，dropout 的作用就不大了。

除此以外我们还尝试使用了与经典架构有所改进的 ResNet 进行实验（使用的是 pytorch 的 torchvision 包中的官方实现），但由于算力实在不够，也没能花费过多的心思调整参数，最终最高也只达到了 92%的准确率，与《Deep Residual Learning for Image Recognition》**[2]** 中能够达到的 95%有所差距。

参考文献：

[1] Siamese Recurrent Architecture for learning Sentence Similarity

[https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12195/12023]

[2] Deep Residual Learning for Image Recognition

[https://arxiv.org/pdf/1512.03385.pdf]

[3] Network In Network

[https://arxiv.org/abs/1312.4400]