人工智能实验

期中 project

组员：陈家豪 18308013

李骁达 18340095

目录

1 组员分工

学号	姓名	工作
18308013	陈家豪	主要完成了 CNN 工作
18340095	李骁达	主要完成了 RNN 工作

2 使用 CNN 完成图片分类任务

2.1 实验原理

2.1.1 图片分类问题

图片分类，顾名思义，是一个输入图像，输出对该图像内容分类的描述的问题。图片分类任务的完整流程如下：

(1) **输入**：输入是包含 N 个图像的集合，每个图像的标签是 K 种分类标签中的一种。这个集合称为训练集；

(2) **学习**：这一步的任务是使用训练集训练分类器，使其学习每个类别长什么样；

(3) **评价**：让分类器来预测它未曾见过的图像的分类标签，并以此来评价分类器的质量；

本次实验需要我们对图像分类数据集 CIFAR-10 训练一个模型，对其进行分类。CIFAR-10 数据集包含了 60000 张 32×32 的小图像。每张图像都有 10 种分类标签中的一种。毫无疑问，全连接神经网络可以用来完成这一任务，但其存在一些缺点，例如权重矩阵参数过多、会破坏图像场景特性等。因此，在计算机视觉领域，卷积神经网络成为了应用主流。

2.1.2 卷积神经网络（CNN）

相比于全连接神经网络，卷积神经网络依旧是层级网络，但对一些网络层的功能和形式进行修改。其中最关键的网络层分别是卷积层和池化层。

卷积层 在说明 CNN 具体原理前，我们需要理解什么是卷积。在数字图像处理中，一幅图像就是一个二维数据矩阵 $\mathbf{f}$，而滤波器就是一个小矩阵 $\mathbf{W}$。假设我们使用该滤波器对该图像进行卷积，则对每一个图像像素点，我们有：

$$f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} W(s,t) f(x+s, y+t) \tag{1}$$

在传统的数字图像处理中，卷积操作可以对图像进行局部区域的特征提取，也就是所谓的 “局部感受野”。

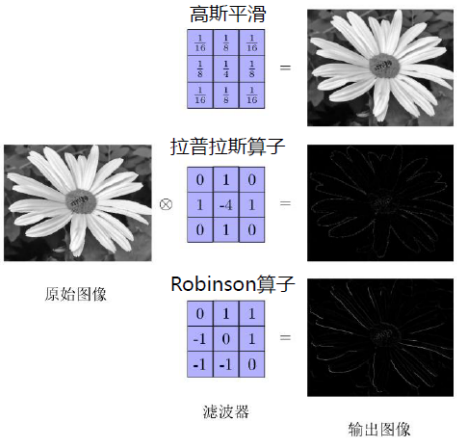


图 1: 滤波器对图像特征的提取

当我们把卷积的思想应用到神经网络中、让网络自行调整滤波器的参数时，我们就可以实现网络自动提取图像的特征进行学习。而这一层网络就是**卷积层**。其主体部分就是卷积核（也就是滤波器，或称为滑动窗口），在输入矩阵上不断滑动、进行卷积，输出计算结果。卷积层有以下几个参数：深度（depth）、步长（stride）、填充值（padding）等。

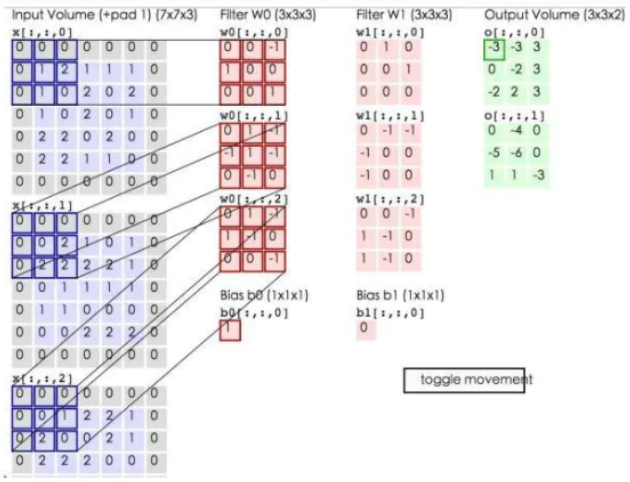


图 2: 卷积层运行示例 1

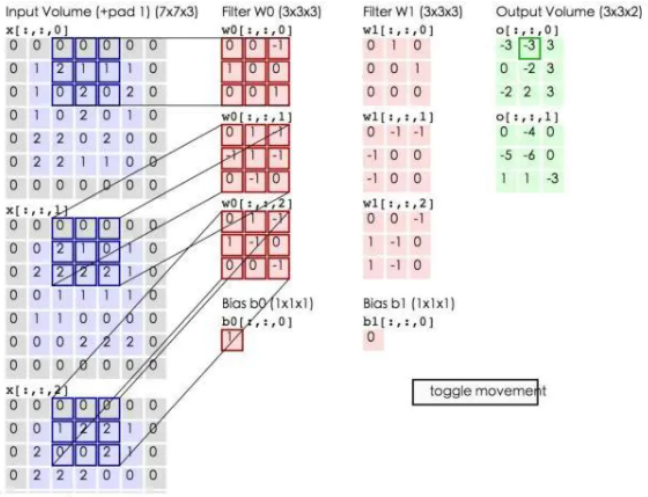


图 3: 卷积层运行示例 2

以图2和3为例，可见输入数据是一个蓝色矩阵，大小为 $5 \times 5 \times 3$，我们可以理解为输入就是一个大小为 5×5 的图像，有三个颜色通道。而该卷积层有两个卷积核，以粉色矩阵表示。每个卷积核对应输入数据的每个通道都有一个大小为 3×3 的矩阵（滤波器），因此一个卷积核的大小为 3×3（不包括偏置值）。此外，该卷积层的步长为 2，说明卷积核每次滑动两个单位进行计算。为了能够计算边缘值（如图2所示），我们需要对原图像进行填充，包围蓝色矩阵的灰色 0 即为我们填充的数值 0。具体而言，每个卷积核在相应通道上对输入数据进行卷积后，将各个通道的计算结果相加、再对所有数值加上同一个偏置值，即为输出结果，在图中以绿色矩阵表示。总而言之，图中一个 $5 \times 5 \times 3$ 的输入矩阵 I 在经过两个大小为 3×3 的卷积核卷积后，得到大小为 $3 \times 3 \times 2$ 的输出矩阵 O，其中 $O[:, :, 0]$ 是第一个卷积核的计算结果，$O[:, :, 1]$ 是第二个卷积核的计算结果。这层网络的参数数量即为 $(3 \times 3 \times 3 + 1) \times 2 = 56$。

除了计算卷积核对输入数据的卷积，卷积层还需要对计算结果进行非线性映射。有些资料会将这两个功能分离开来，前者称为卷积层、后者称为激励层。具体而言，对于图2和3的输出结果（绿色矩阵），

我们还需要使用激活函数进行处理，才能作为下一层网络的输入数据。CNN 一般使用 ReLU 作为激励函数，也有极少数情况使用 tanh。

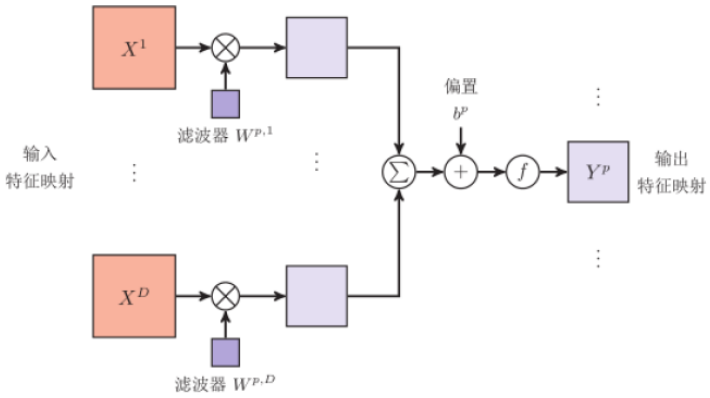


图 4: 卷积层的工作流程

池化层 除了卷积层以外，CNN 还有一种重要的网络结构，称为池化层。池化层一般位于连续的卷积层中间，作用是压缩数据和参数的量，减少过拟合。对于图像而言，池化层就是对图像进行 “压缩”。一方面，对图像进行压缩时去掉的信息往往是无关紧要的，池化层可以对信息进行筛选、留下具有尺度不变性的特征信息，而这些信息也最能表达图像的特征；另一方面，池化层可以在一定程度上防止过拟合，减少参数数量、提高计算速度、提高鲁棒性。

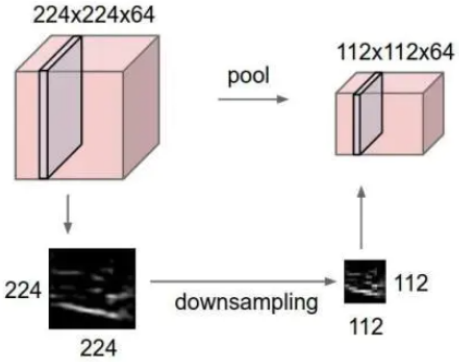


图 5: 对图像进行池化操作

池化层常用的方法有 Max pooling 和 Average pooling。而最常用的类型就是 max pooling。对于最大池化层，其与卷积层类似，就是使用一个滤波器、输出范围内的最大值作为输出结果。当我们设置大小和步长均为 2 时，就相当于将输入数据的特征高度和宽度都缩减一半。

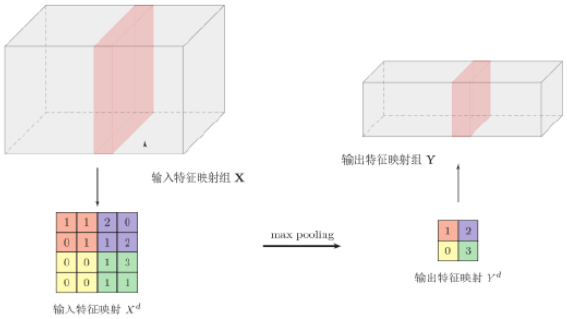


图 6: 最大池化层工作示意图

CNN 结构 CNN 一般由卷积层、池化层和全连接层交叉堆叠而成。对于一个典型的 CNN，其首先堆叠若干个卷积块（一个卷积块由若干层卷积层组成，最后一层可以设为池化层），然后再连接若干个全连接层，最后接上分类器实现。

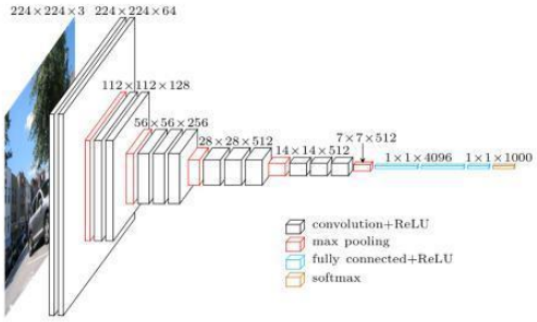


图 7: 一个典型的 CNN：VGG16 网络

CNN 反向传播 由上述内容可知，CNN 一般由卷积层、池化层和全连接层组成。在更新网络参数时，涉及到反向传播算法。对于全连接层，其反向传播与传统神经网络一致。对于卷积层，我们同样可以得到其前向传播的计算公式，这样的话我们就可以通过链式求导反向推出各个参数的梯度。而池化层比较复杂，虽然其没有参数，但我们需要用其还原误差，再将误差传播到上一层网络，这样才能让上一层网络得到正确的更新。以最大池化层为例，反向传播时就是需要将误差还原到做前向传播时得到最大值的位置，这个过程称为 unsample。

由于具体公式比较复杂，故我们没有列出。由于本次实验我们可以使用现成框架，故不需要具体实现。

2.2 创新点及其原理

2.2.1 数据增强

在训练网络时，我们常常会遇到数据不足的情况。一方面，一个先进的神经网络往往有数百万的参数。但我们的训练数据过少时，就会出现过拟合的现象，导致网络在测试集上的准确率下降；另一方面，神经网络的训练依赖于训练数据的质量，如果训练数据质量不够好，也会导致网络的准确率下降。因此，我们不仅要在有限的原始训练数据基础上得到更多的训练数据，还要保证、甚至提高训练数据的质量。这就是所谓的 “数据增强”。

对于不同类型的数据，数据增强的方式各有不同，而我们重点介绍图片分类中的数据增强。在训练网络的过程中，我们希望得到能够依据物体的特征进行识别、分类的网络。为了提高鲁棒性，被依据的物体特征应该具有不变性，即无论对图片进行任何微小改变，这一特征都不会受到影响。例如，即使对一张狗的图片进行任意的缩放、旋转、甚至改变狗在图片中的位置，我们都能识别出这里面有一条狗。基于上述前提，我们就可以实现对训练数据的数据增强。一般来说，数据增强有以下几种方式：

(1) 翻转：对图片进行水平或垂直翻转；

(2) 旋转：将图片进行一定角度的旋转。如果旋转后图像尺寸出现改变，就对图像进行缩放、剪切；

(3) 缩放：将图片进行缩放；

(4) 裁剪：对图像进行裁剪，然后将裁剪部分的调整回原始图片大小；

(5) 移位：将对象在图片内进行适当平移，使得神经网络可以看到图片的各个角落；

(6) 加上随机噪声；

除此以外，我们还有其他方式进行数据增强，例如使用条件 GAN 等。总而言之，通过数据增强，我们既可以增加训练数据的样本数量，也可以提高训练数据的质量、使得训练的网络更加鲁棒。

2.2.2 Dropout

在训练网络时，如果模型参数过多、训练样本数量过少，就会出现过拟合现象。为了解决这个问题，Hinton 在其论文《Improving neural networks by preventing co-adaptation of feature detectors》中提出了 Dropout。

Dropout 是指在深度学习网络的训练过程中，按照一定概率将一部分神经网络单元暂时从网络中丢弃，相当于从原始的网络中找到一个更 “瘦” 的网络。这样，每次训练都相当于对一个全新的网络进行训练。由于每次训练某个神经单元都会被强迫与其他被随机挑选出来的神经单元共同工作，因此神经单元之间的联合适应性被削弱，泛化能力得到增强。

当一个网络应用 Dropout 技术时，每轮训练过程中每个节点都可能以概率 p 去除。在下一轮训练开始前，被去除的节点都不会参与正向传播、反向传播、参数更新等计算；而在测试阶段，所有节点都会参与计算，但节点的权重参数会乘以 p，这样就相当于每轮训练出来的不同网络共同参与计算。

2.2.3 残差网络

在深度学习中，网络层数的增多往往会伴随着各种各样的问题，例如模型容易过拟合、计算资源大量消耗等，这些问题都可以通过各种方式进行解决。然而，无脑增加网络层数并不一定带来更好的结果。有时随着网络深度增加，网络反而会出现退化（degradation）现象，即一个深层网络的表现反而比不上浅层网络。

从信息论的角度上看，随着层数的加深，一个数据的信息在前向传播中会逐渐减少。那我们是否可以使用直接映射，让一些浅层网络的信息直接传递到深层网络呢？残差网络应运而生。残差网络是由一系列残差块组成的。

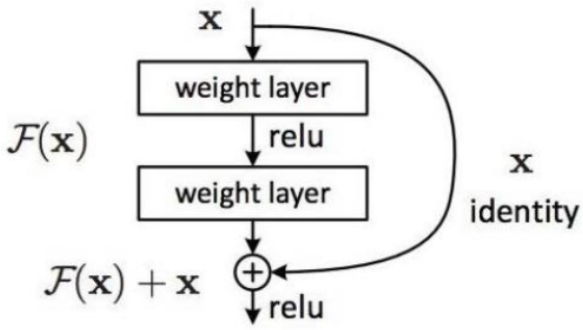


图 8: 残差块

残差块的结构如图8所示。可见，一个残差块可以表示为

$$x_{l+1} = x_l + \mathcal{F}(x_l) \tag{2}$$

残差块分为两部分：直接映射部分和残差部分。直接映射就是 x_l，当其与 $\mathcal{F}(x_l)$ 的 Feature Map 不一致、无法相加时，还需要通过卷积进行升降维；而 $\mathcal{F}(x_l)$ 就是残差部分。本质上，残差块是想实现恒等映射，相当于 $y = x$。因此，$y = H(x) = x + \mathcal{F}(x)$ 中的 $\mathcal{F}(x)$ 便对应残差概念。

以上是最简单的残差网络概念介绍。实际上，残差网络还有很多改进与优化，在此我们不再阐述。

2.3 网络结构

2.3.1 原始 CNN

我们实现的原始 CNN 网络如图9所示，我们实现了一个三层卷积层、两层池化层和四层全连接层的 CNN。

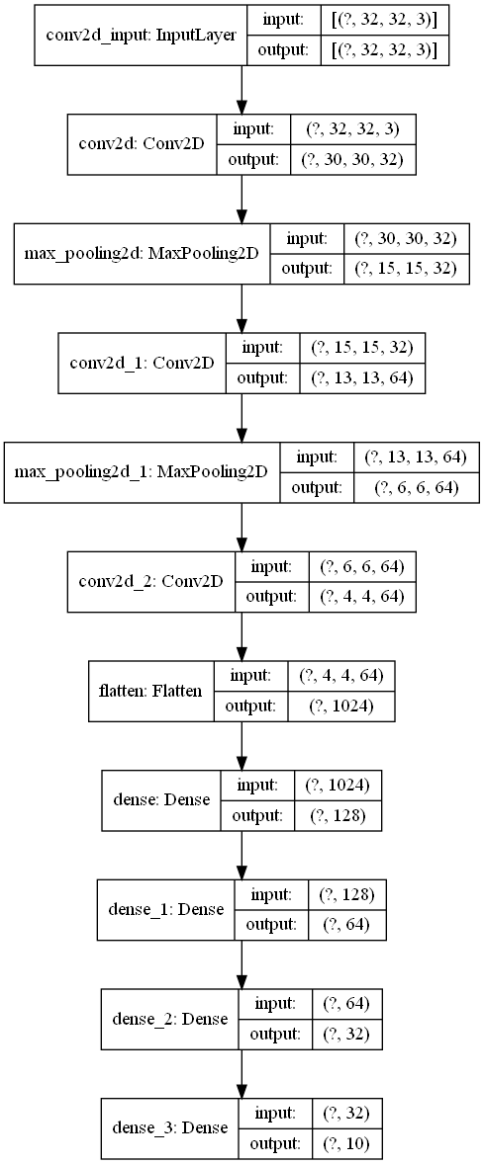


图 9: 原始 CNN 网络结构

2.3.2 原始 CNN+Dropout

我们实现的附带 Dropout 的原始 CNN 网络结构如图10所示。我们在第一、二层全连接层后面添加了概率为 50% 的 Dropout 层。

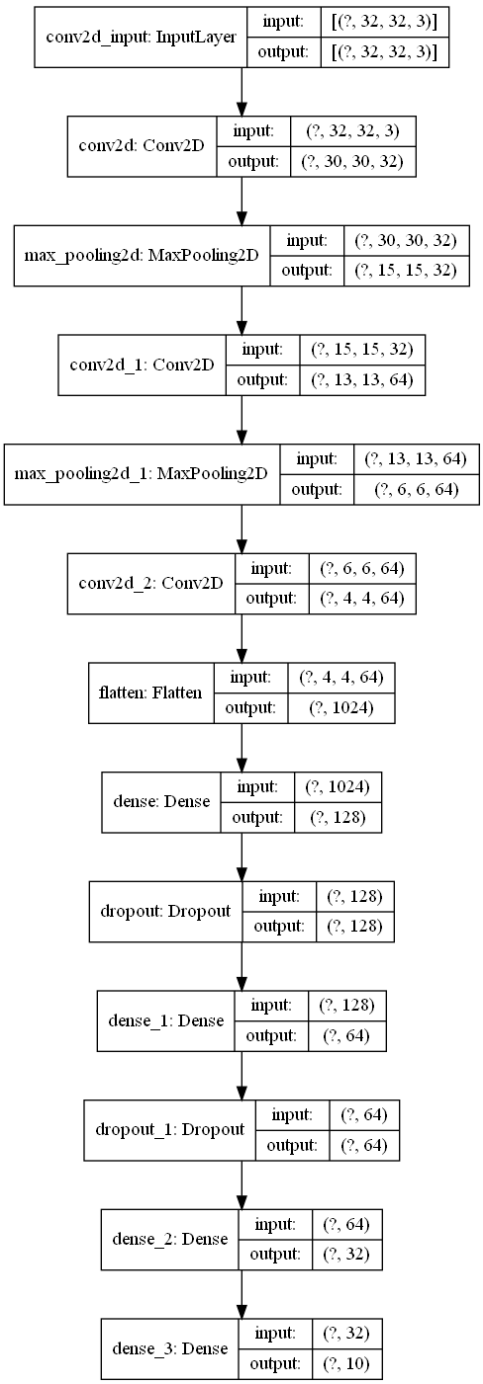


图 10: CNN+Dropout 网络结构

2.3.3 残差网络

我们实现的残差网络结构如图11，12，13和14所示。由于网络过长，故分四张图片显示。

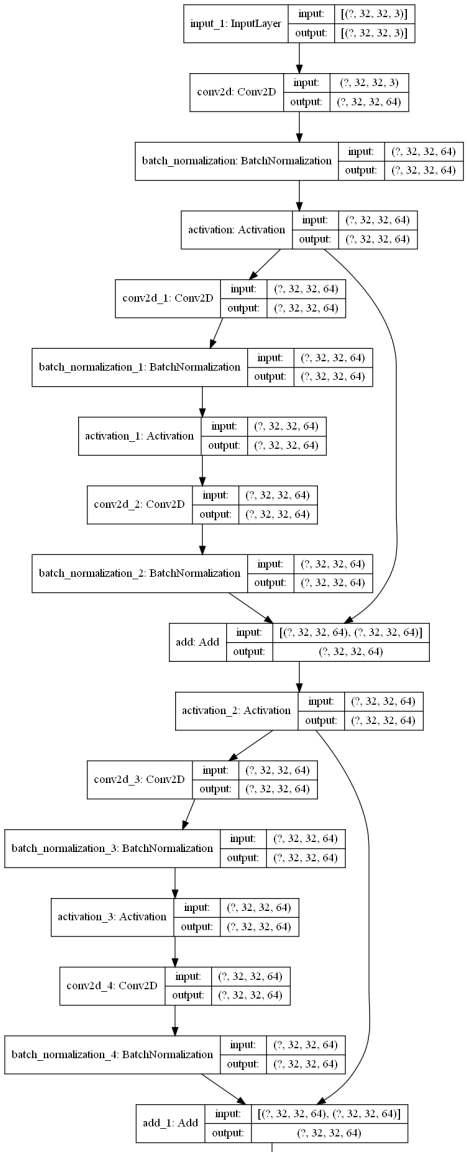


图 11: 残差网络结构 (1)

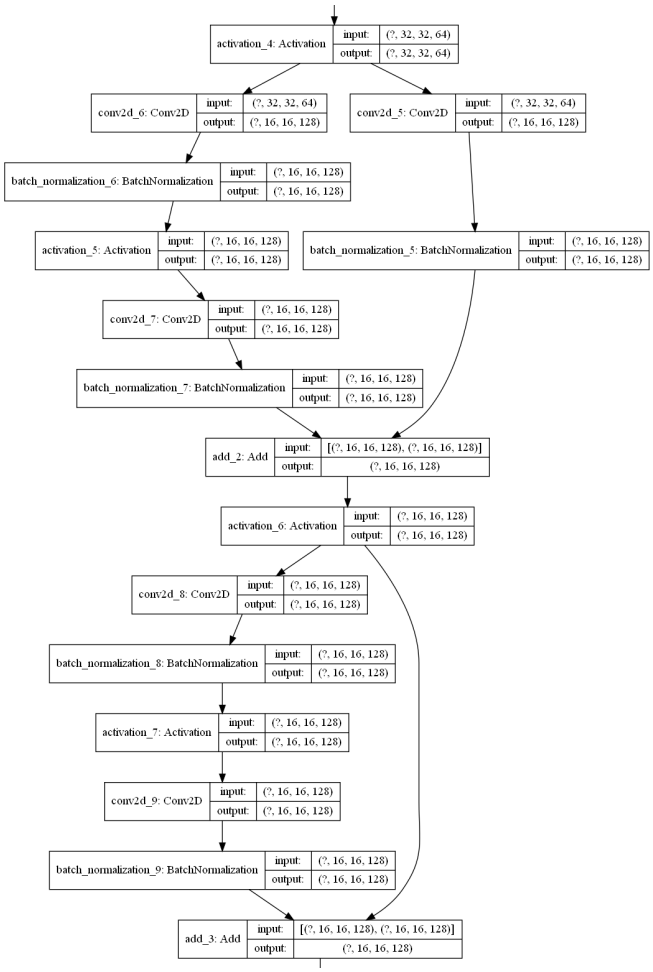


图 12: 残差网络结构 (2)

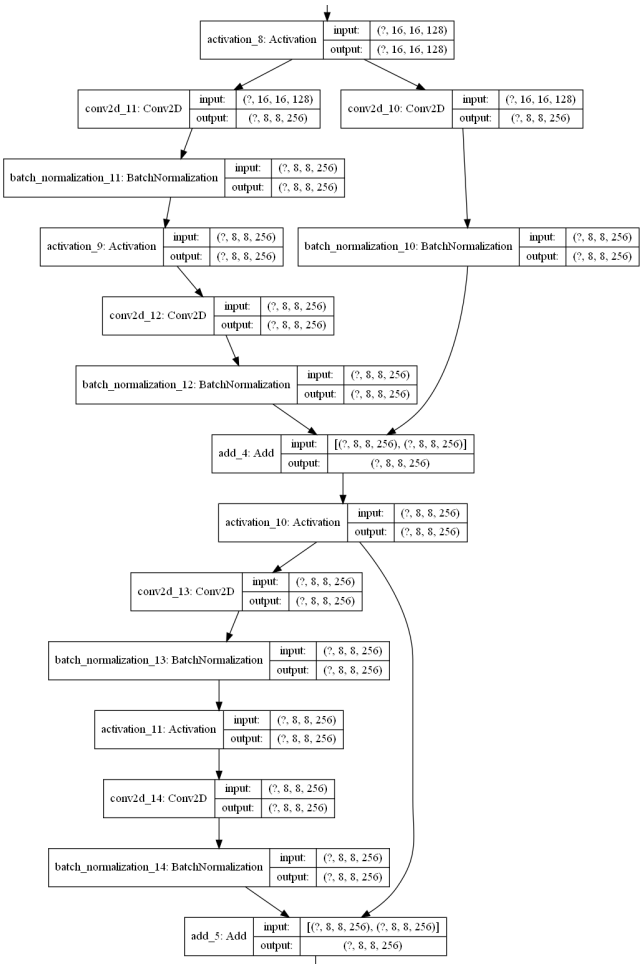


图 13: 残差网络结构 (3)

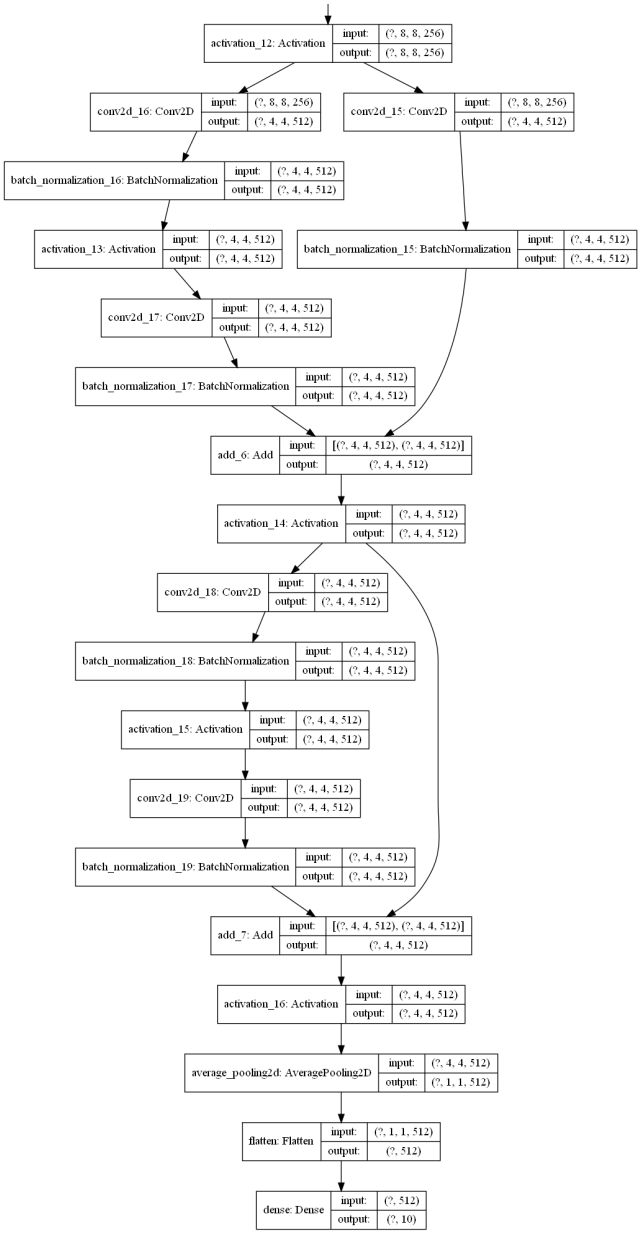


图 14: 残差网络结构 (4)

可见，我们实现了堆叠了八个残差块的网络，每个残差块都至少有两个卷积层、两个正则化层和两个激活层。

2.4 关键代码

我们使用 Keras 和 tensorflow 完成框架的搭建。

2.4.1 原始 CNN 模型代码

模型框架代码如下所示：

```
def build_model():
    # 实例化一个简单的CNN
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
```

```
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    # 在CNN上添加分类器
    model.add(layers.Flatten())  # 摊平成1D向量输入到密集连接分类器中
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))
    model.summary()
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy
        '])
    return model

def main():
    ......
    model = build_model()
    history = model.fit(train_images, train_labels, epochs=50, batch_size=64,
        validation_data=(val_images, val_labels))
    ......
```

可见，我们实现了一个三层卷积层、两层池化层和四层全连接层的原始 CNN 网络。

此外，实现 Dropout 的 CNN 代码如下所示：可见我们在第一、二层全连接层后面添加了概率为 0.5 的 Dropout 层：

```
def build_model():
    # 实例化一个简单的CNN
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    # 在CNN上添加分类器
    model.add(layers.Flatten())  # 摊平成1D向量输入到密集连接分类器中
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dropout(0.5));
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dropout(0.5));
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))
    model.summary()
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy
        '])
    return model
```

2.4.2 残差网络模型代码

我们实现残差网络模型的代码如下所示：

```
# 使用 batch normalization 的 Conv2D层，无激活函数
def con2d(x, filters, kernel_size, weight_decay=.0, strides=(1, 1)):
    layer = layers.Conv2D(filters=filters,
                          kernel_size=kernel_size,
                          strides=strides,
                          padding='same',
                          use_bias=False,
                          kernel_regularizer=l2(weight_decay)
                          )(x)
    layer = layers.BatchNormalization()(layer)
    return layer
# 在 conv2d 层上添加激活函数
def con2d_relu(x, filters, kernel_size, weight_decay=.0, strides=(1, 1)):
    layer = con2d(x, filters, kernel_size, weight_decay, strides)
    layer = layers.Activation('relu')(layer)
    return layer
# 残差单元，其中 feature_extend 参数用于确定是否要进行下采样，使 x 可以和 F(x) 相加
def residual_block(x, filters, kernel_size, weight_decay=.0, down_sample=True):
    if down_sample:
        shortcut = con2d(x, filters, kernel_size=1, strides=2)
        stride = 2
    else:
        shortcut = x
        stride = 1
    simple = con2d_relu(x, filters=filters, kernel_size=kernel_size,
                        weight_decay=weight_decay, strides=stride)
    simple = con2d(simple, filters = filters, kernel_size=kernel_size,
                   weight_decay=weight_decay, strides=1)
    output = layers.add([simple, shortcut])
    output = layers.Activation('relu')(output)
    return output
# 建立残差网络
def build_resnet18_model(class_num, input_shape, weight_decay=1e-4):
    input = layers.Input(shape=input_shape)
    x = input
    x = con2d_relu(x, filters=64, kernel_size=(3, 3), weight_decay=weight_decay, strides
        =(1, 1))
    # conv2
    x = residual_block(x, filters=64, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=False)
    x = residual_block(x, filters=64, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=False)
    # conv3
    x = residual_block(x, filters=128, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=True)
```

```
    x = residual_block(x, filters=128, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=False)
    # conv4
    x = residual_block(x, filters=256, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=True)
    x = residual_block(x, filters=256, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=False)
    # conv5
    x = residual_block(x, filters=512, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=True)
    x = residual_block(x, filters=512, kernel_size=(3, 3), weight_decay=weight_decay,
        down_sample=False)
    x = layers.AveragePooling2D(pool_size=(4, 4), padding='valid')(x)
    x = layers.Flatten()(x)
    x = layers.Dense(class_num, activation='softmax')(x)
    model = models.Model(input, x, name='ResNet18')
    return model
```

可见，我们首先创建了不带有激活函数的 con2d() 和带有激活函数的 con2d_relu()，然后以此为基础构建残差块函数 residual_block()。最后，我们将残差块进行堆叠，得到残差网络。

2.4.3 数据增强代码

数据增强代码如下所示，我们调用了 keras 的 ImageDataGenerator() 实现了数据的增强。

```
# 数据增强
train_datagen = ImageDataGenerator(rotation_range=40,        # 0~180，图像随机旋转的角度范
    围
                                   width_shift_range=0.2,    # 水平方向上平移的范围，相对于
                                       总宽度的比例
                                   height_shift_range=0.2,   # 垂直方向上平移的范围，相对于
                                       总高度的比例
                                   shear_range=0.2,          # 随机错切变换的角度
                                   zoom_range=0.2,           # 随机缩放的范围
                                   horizontal_flip=True,     # 随机将一半图像水平翻转
                                  )
train_gen = train_datagen.flow(train_images, train_labels, batch_size=128)
```

2.5 实验过程与结果分析

2.5.1 原始 CNN 及其相关改进

我们使用如图9所示的网络结构，设置 optimizer='rmsprop', loss='categorical_crossentropy', batch_size=64, epoch=100，使用原始训练集和验证集，训练结果如图15和16所示。

此外，我们还尝试使用对训练集进行数据增强处理。使用增强后的训练集训练网络，训练结果如图17和18所示。

我们也尝试使用 Dropout 改进了网络，网络结构如图10所示。使用原始数据集训练网络，训练结果如图19和20所示。

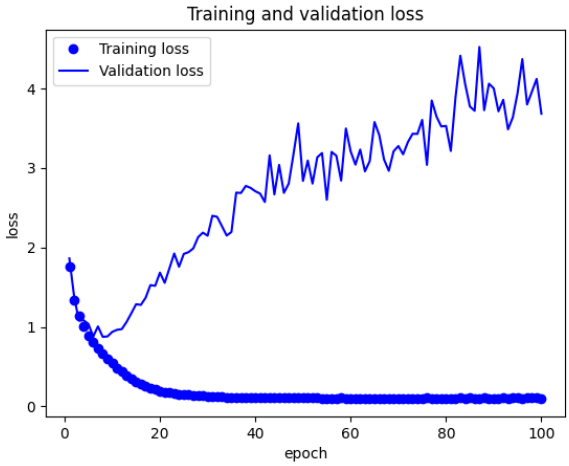


图 15: 原始 CNN 的训练损失值

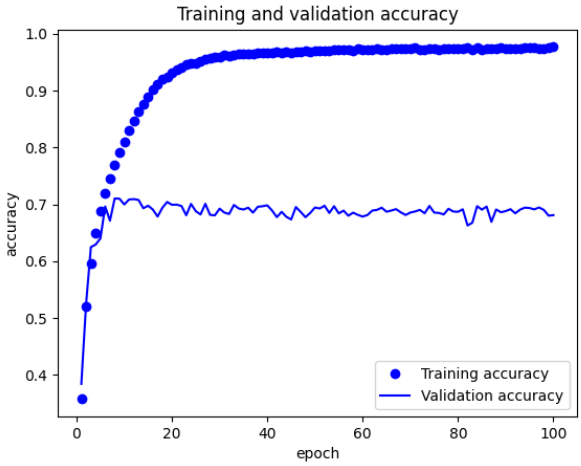


图 16: 原始 CNN 的训练准确率

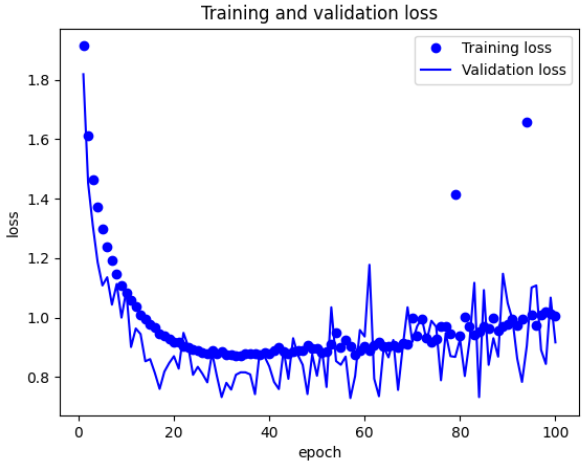


图 17: 使用数据增强的 CNN 训练损失值

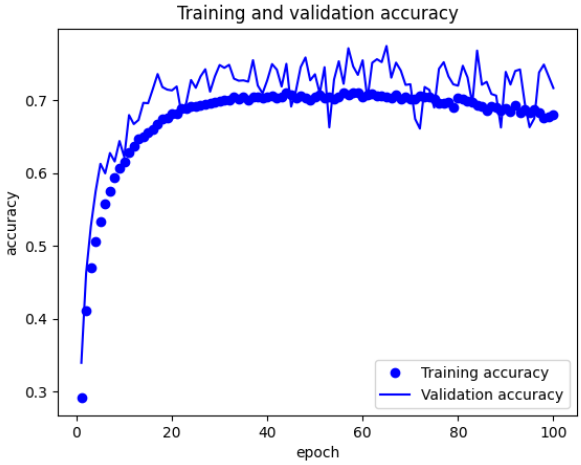


图 18: 使用数据增强的 CNN 训练准确率

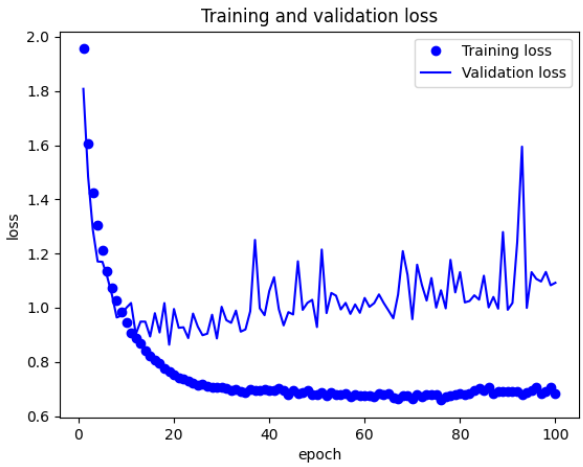


图 19: 使用 Dropout 的 CNN 训练损失值

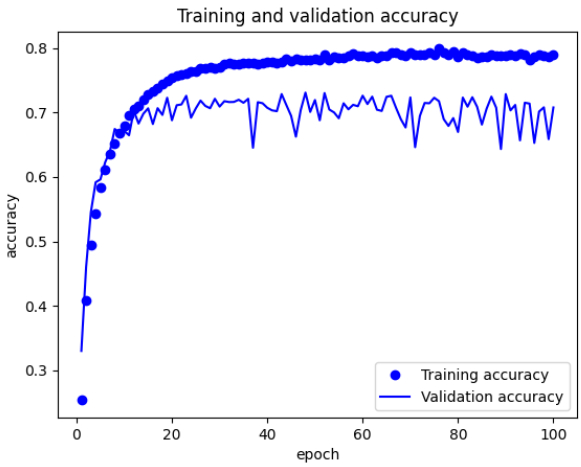


图 20: 使用 Dropout 的 CNN 训练准确率

对于原始 CNN，我们可以看到一开始训练时训练集和验证集的 loss 均下降，但在 epoch=10 后出现过拟合现象，validation loss 不降反升，此时验证集上的准确率为 **71%**。从准确率上看，模型在训练集上的准确率一直在上升，但验证集上的准确率在训练后期没有太多变化，在 **70%** 处抖动。我们选取 10 次迭代的模型在测试集上进行测试，loss 为 **0.9173**，准确率为 **0.7009**，与在验证集上的结果较为接近。

对于使用数据增强的 CNN，我们可以发现相比于原始 CNN，训练损失曲线和验证损失曲线贴合比较紧密，模型泛化能力得到了提高，这也是数据增强带来的结果。在前 30 次迭代时，模型在训练集和验证集上的 loss 不断下降，准确率不断提高，但在 30 次迭代后训练集和验证集的 loss 有所上升。该模型在验证集上的准确率在 **73%** 左右波动，最高可达 **76%**，相比原始 CNN 有所提高。我们选取 20 次迭代的模型在测试集上进行测试，loss 为 **0.7856**, 准确率为 **0.7295**。

对于使用 Dropout 的 CNN，通过图15和19的比较，我们可以发现使用 Dropout 后网络在第 30 轮迭代后才开始出现过拟合，且过拟合现象比原始 CNN 轻微很多。而使用 Dropout 的 CNN 在验证集上的准确率依然能保持在 **70%** 左右，与原始 CNN 相当。这些都说明 CNN 在使用 Dropout 后性能有了明显的提高，过拟合现象得到较大缓解。我们选取 20 次迭代的模型在测试集上进行测试，loss 为 **0.9105**, 准确率为 **0.7112**。

2.5.2 残差网络及其相关改进

我们使用如图11,12,13和14所示的残差网络结构,设置训练集和验证集的比例为 8:2,optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'], batch_size=64，epoch=200，使用原始训练集和验证集，训练结果如图21和22所示。为了对比残差网络给模型带来的影响，我们还额外建立了一个除了没有跳层连接、其他结构均与该残差网络一致的网络进行对照，训练结果如图23和24所示。

当然，我们也尝试对训练集进行数据增强后训练残差网络，最终训练结果如图25和26所示。

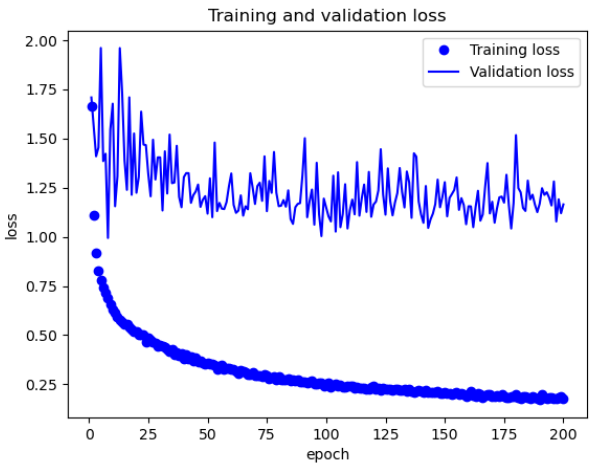


图 21: 残差网络的训练损失值

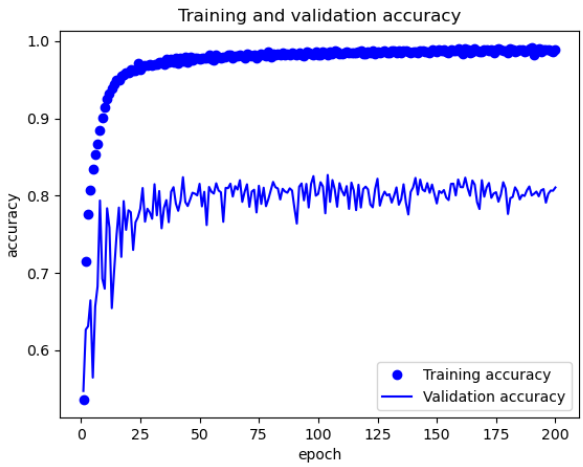


图 22: 残差网络的训练准确率

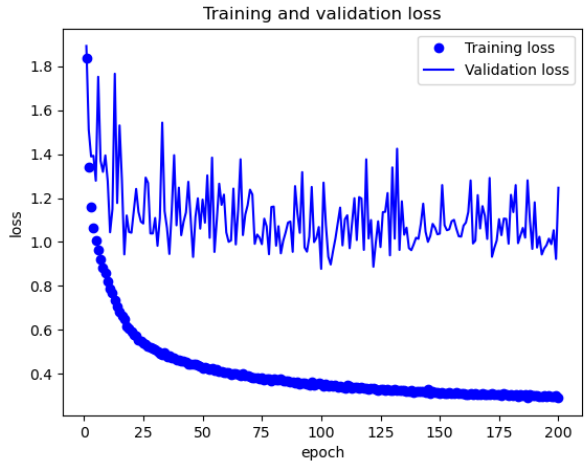


图 23: 未使用跳层连接的残差网络训练损失值

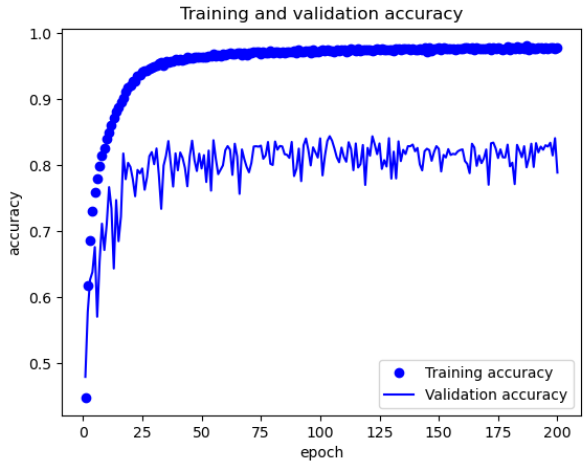


图 24: 未使用跳层连接的残差网络训练准确率

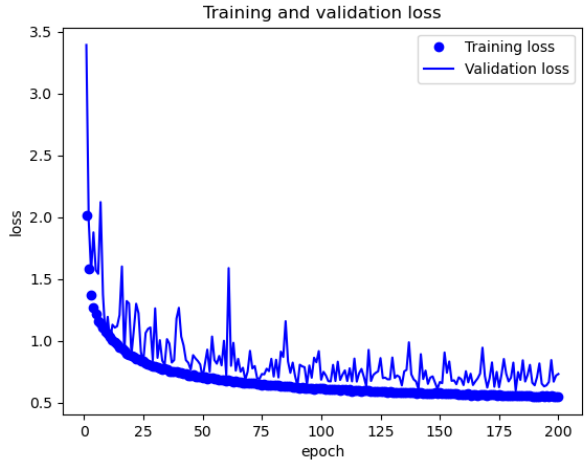


图 25: 使用数据增强的残差网络训练损失值

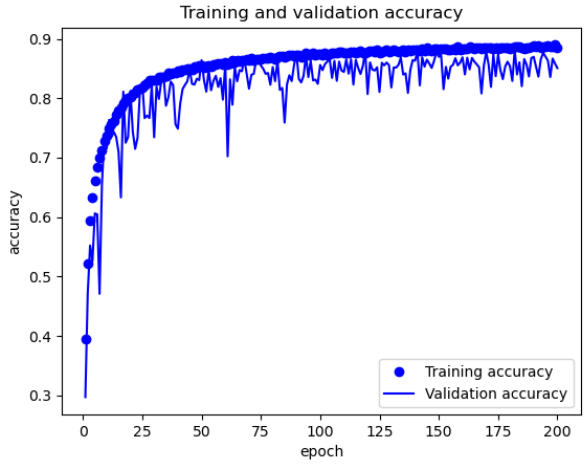


图 26: 使用数据增强的残差网络训练准确率

对于残差网络，其在验证集上的数据波动比较大，我们猜测可能是由于使用 mini batch 训练导致的结果。当迭代次数超过 50 次时，可以看到模型在验证集上出现了轻微的过拟合现象，此时模型在验证集上的准确率为 **80%**，此后一直在 80% 附近抖动。与2.5.1中的原始 CNN 相比，该残差网络的性能有了较大提高。

比较图21和25，22和26，我们可以判断残差网络中跳层连接的效果。在我们的模型上，跳层连接的存在与否对模型的性能影响不是很大，与理论结果存在一定的出入。硬要进行比较的话，反而不使用跳层连接的网络在验证集上的准确率高于使用跳层连接的网络。我们猜测这可能是我们的跳层连接路径设置不太合理导致的。

对于使用数据增强的残差网络，我们可以看到其与2.5.1的使用数据增强的原始 CNN 相似，验证集上的数据曲线与训练集上的数据曲线十分贴近，没有出现过拟合现象，这说明模型的泛化能力得到了提高。此外，模型在验证集上的准确率稳定在 **88%** 左右，相比未使用数据增强的模型有了较大的提升。

3 使用 RNN 完成关键词提取任务

3.1 实验原理

3.1.1 序列标注问题

本次实验要求我们完成关键词提取任务。所谓的关键词提取，就是给定一个句子，我们需要实现一个模型从中提取出关键词。在 NLP 中，有一类问题与该任务十分相似，就是序列标注问题。序列标注问题一般可以分为两类：

(1) 原始标注（Raw labeling）：每个元素都需要被标注为一个标签；

(2) 联合标注（Joint segmentation and labeling）：所有的分段被标注为同样的标签；

例如，对于句子 “I went to the Great Wall.”，如果我们想要实现命名实体识别（Named entity recognition，NER），就需要对里面的信息进行提取、标注。原始标注是对每个单词都打上标签，例如 “I” 就是 “人名”；而联合标注不仅识别单个元素，还要识别元素段，例如对整个短语 “Great Wall” 打上标签 “地名”。显然，关键词提取任务就是一个联合标注问题，我们需要识别里面的关键词并打上标签。

解决联合标注问题的最简单方法就是使用 BIO 标注法，将其转化为原始标注问题。BIO 标注法对每个元素进行如下标注：

(1) **B-X**：表明该元素所属片段为 X 类型，且其位于片段开头；

(2) **I-X**：表明该元素所属片段为 X 类型，但其不是片段开头；

(3) **O**：表明该元素不属于任何类型；

以 “I left home and went to the Great Wall.” 为例，如果我们要识别其中的地名，那么设 X 为地名短语，我们有：

I	left	home	and	went	to	the	Great	Wall.
O	**O**	**B-X**	**O**	**O**	**O**	**O**	**B-X**	**I-X**

显然，把 “地名” 替换成 “关键词”，我们就可以用 BIO 标注法完成关键词提取任务，将关键词提取任务转换为序列标注问题。

处理序列标注问题的常用模型有隐马尔可夫模型（HMM）、条件随机场（CRF）和循环神经网络（RNN）等。随着深度学习的流行，循环神经网络逐渐成为了主流方式。

3.1.2 循环神经网络（RNN）

RNN 是一种扩展的人工神经网络，适用于处理序列数据。对于传统的人工神经网络而言，其元素之间是相互独立的，输入和输出也是独立的，例如在图片分类问题中，网络会对不同的图片预测不同的类别，但不会考虑先后输入两张图片之间的联系。但对于序列数据而言，前后的样本数据在时序上存在相关性，我们需要通过某种方式保留对上一个数据的 “记忆”，才能输出正确的结果。而 RNN 就是一种用来捕捉样本间在时序上的相关性的网络。

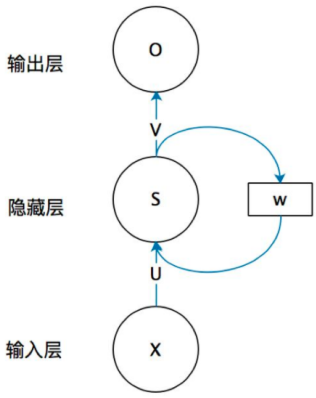


图 27: 一个简单的 RNN 结构

一个简单的 RNN 结构图如图27所示。其中左图为真实的 RNN，**X** 是输入层接收的输入向量；**U** 是输入层到隐藏层的权重矩阵；**S** 是隐藏层输出的输出向量，将作为输出层的输入数据；**V** 是隐藏层到输出层的权重矩阵，而 **O** 是输出层的输出向量。如果不考虑 **W** 所处节点的存在，那么该网络就是一个普通的全连接网络。而事实上，**W** 是一个权重矩阵，其与隐藏层上一次的输出值相乘后参与到当前隐藏层的计算。隐藏层当前的输出值不仅依赖于当前的输入数据 **X**，还依赖于上一次的输出数据。如图28所示，可见 t-1 时刻的隐藏层输出值也会参与到 t 时刻隐藏层输出值的计算。

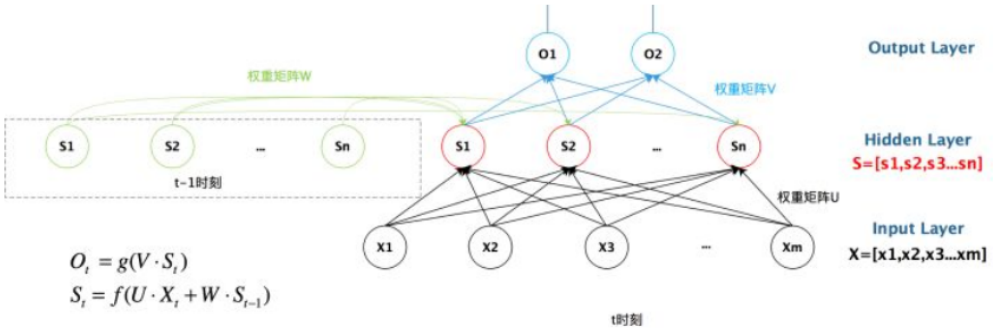


图 28: 一个具体的 RNN 结构

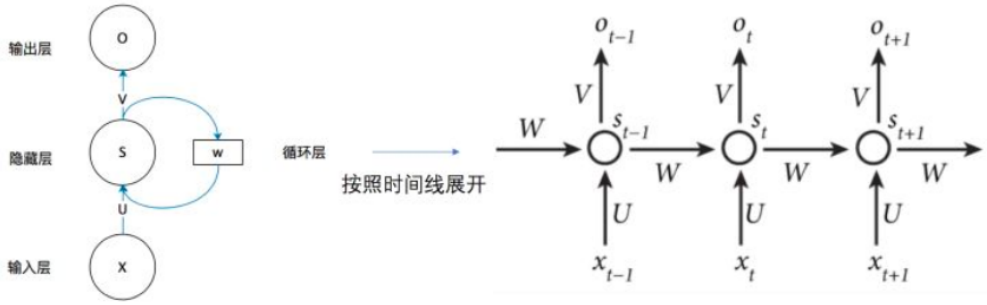


图 29: RNN 时间线展开图

如果我们将 RNN 按时间线展开，可以得到如图29所示的结构。在 t 时刻，网络接收到 X_t 后，与权重矩阵 U 相乘，而上一时刻隐藏层的输出 S_{t-1} 也会与其权重矩阵 W 相乘。两个相乘后的值共同参与对 S_t 的计算。在得到 S_t 后，其既可以计算出当前的输出层的输出值 O_t，也可以参与下一时刻的隐藏层

输出计算。用公式表达的话如下所示：

$$S_t = f(U \cdot X_t + W \cdot S_{t-1}) \tag{3}$$

$$O_t = g(V \cdot S_t) \tag{4}$$

在训练网络时，我们需要反向传播、对权重矩阵 U, V, W 进行更新。设式3和4即为每个时刻的隐状态 S_t 和输出 $\widehat{O_t}$，那么时序上的总损失为：

$$E(O, \widehat{O}) = \sum_t E_t(O_t, \widehat{O_t}) \tag{5}$$

因此，对于 V 的梯度，我们有：

$$\begin{aligned}\frac{\partial E}{\partial V} &= \sum_t \frac{\partial E_t}{\partial V} = \sum_t \frac{\partial E_t}{\partial \widehat{O_t}} \frac{\partial \widehat{O_t}}{\partial V} \\ &= \sum_t \frac{\partial E_t}{\partial \widehat{O_t}} \frac{\partial \widehat{O_t}}{\partial (V \cdot S_t)} \frac{\partial (V \cdot S_t)}{\partial V}\end{aligned}$$

对于 U 的梯度，我们有：

$$\begin{aligned}\frac{\partial E}{\partial U} &= \sum_t \frac{\partial E_t}{\partial U} = \sum_t \frac{\partial E_t}{\partial \widehat{O_t}} \frac{\partial \widehat{O_t}}{\partial U} \\ &= \sum_t \frac{\partial E_t}{\partial \widehat{O_t}} \frac{\partial \widehat{O_t}}{\partial S_t} \frac{\partial S_t}{\partial U}\end{aligned}$$

对于 W 的梯度，我们有：

$$\begin{aligned}\frac{\partial E}{\partial W} &= \sum_t \frac{\partial E_t}{\partial W} = \sum_t \frac{\partial E_t}{\partial \widehat{O_t}} \frac{\partial \widehat{O_t}}{\partial W} \\ &= \sum_t \frac{\partial E_t}{\partial \widehat{O_t}} \frac{\partial \widehat{O_t}}{\partial S_t} \frac{\partial S_t}{\partial W}\end{aligned}$$

3.2 创新点及其原理

3.2.1 LSTM

长短期记忆（Long Short-Term Memory, LSTM）网络是一种特殊的 RNN，其通过引入门控机制，解决了长序列训练过程中的梯度消失和梯度爆炸问题。如图31所示，其中黄色矩形是类似于激活函数操作，粉色圆圈是点操作，单箭头表示数据流向，箭头合并表示向量的合并，箭头分叉表示向量的拷贝。

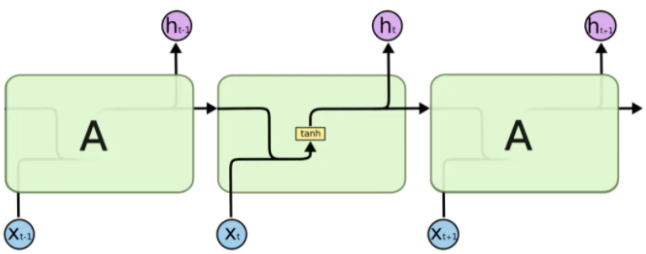


图 30: 传统 RNN 网络

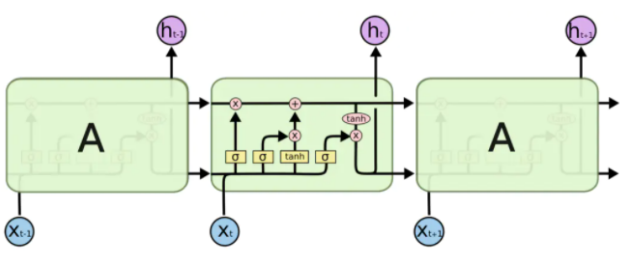


图 31: LSTM 网络

LSTM 的核心是细胞状态，在图中表示为 C_t。LSTM 通过一种被称为门的结构对细胞状态进行更新。在 LSTM 中有三种门：遗忘门、输入门和输出门。遗忘门如图32所示，其将当前时刻的输入 x_t 以及前一个时刻的输出 h_{t-1} 通过 sigmoid 激活函数输出一个 $0 \sim 1$ 之间的向量 f_t，用于表示对细胞状态 C_{t-1} 中信息的保留程度。

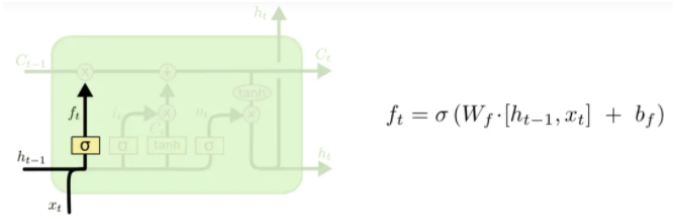


图 32: LSTM 遗忘门

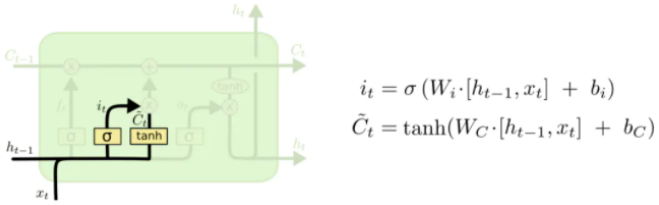


图 33: LSTM 输入门

输入门与遗忘门类似，其对 h_{t-1} 和 x_t 首先同样使用 sigmoid 激活函数处理，输出一个 $0 \sim 1$ 的向量 i_t 表示对新添加信息的保留程度，然后用 tanh 激活函数生成新添加信息 $\widetilde{C}_t$。

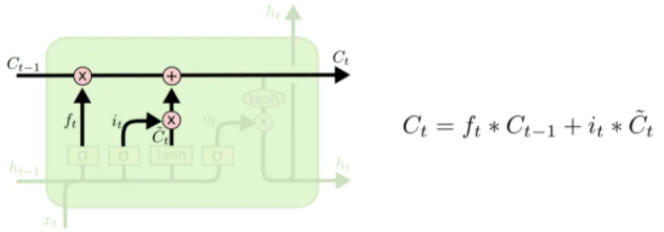


图 34: LSTM 细胞状态更新

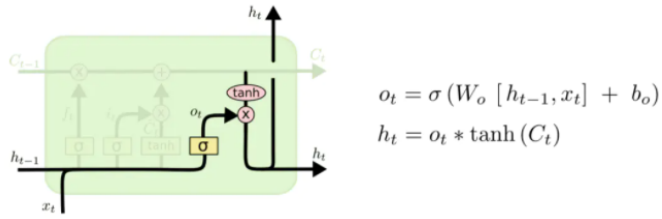


图 35: LSTM 输出门

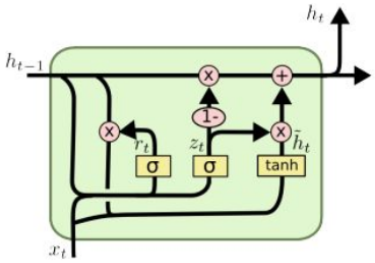
接下来就是对细胞状态进行更新。我们使用 f_t 与 C_{t-1} 点乘，“遗忘” 一部分信息后，再通过输入门添加 $\widetilde{C}_t$ 的部分信息，从而得到新细胞信息 C_t。

最后一部分操作是输出门。h_{t-1} 和 x_t 通过 sigmoid 激活函数得到控制输出的向量，然后 C_t 通过 tanh 激活函数得到 $-1 \sim 1$ 的向量，两个向量相乘即为该 RNN 单元的输出 h_t。

LSTM 的反向传播与 RNN 类似，不过涉及到的参数更多、运算更复杂，故不再阐述。总而言之，细胞状态的引入使得 LSTM 可以记忆到更 “久远” 的信息，有利于解决长依赖问题；此外，LSTM 也有利于缓解梯度消失问题。如今，随着技术发展，LSTM 也出现了许多变种，它们在 LSTM 的基础上对网络进行了部分改进。在此我们不再展开。

3.2.2 GRU

GRU（Gate Recurrent Unit）与 LSTM 一样，也是为了解决长期记忆和反向传播中的梯度问题而提出来的一种 RNN 网络结构。相较于 LSTM，GRU 结构更加简单，也有着很好的效果，故得到了广泛应用。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

图 36: GRU

如图36所示，GRU 同样有两个门控单元：重置门和更新门。重置门用于控制上一个状态的信息有多少内容参与到候选信息 $\widehat{h}_t$ 的计算。h_{t-1} 和 x_t 结合后通过 sigmoid 激活函数得到范围为 $0 \sim 1$ 的向量 r_t，然后 r_t 与上一状态的信息 h_{t-1} 相乘，通过 tanh 激活函数得到 $\widehat{h}_t$。

更新门则用于控制前一时刻的状态信息被带入当前状态的程度。同样，h_{t-1} 和 x_t 结合后通过 sigmoid 激活函数得到范围为 $0 \sim 1$ 的向量 z_t，而该向量将决定 h_{t-1} 和 $\widehat{h}_t$ 构成当前状态信息 h_t 的比例。

3.2.3 深层 RNN 与双向 RNN

基于普通的 RNN，人们“触类旁通”，提出了深层 RNN、双向 RNN、深层双向 RNN 等不同网络结构。

深层 RNN 可以视为单层 RNN 网络堆叠起来。深层 RNN 有多个隐藏层，上一个隐藏层的输出可以作为下一个隐藏层的输入。

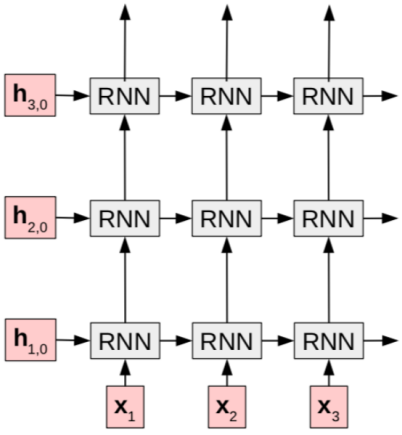


图 37: 一个三层 RNN

在经典 RNN 中，状态的传输是单向的、从前往后传播的。但有些问题不仅依赖于之前时刻的信息，也取决于未来时刻的信息。基于这种想法，人们提出了双向 RNN。

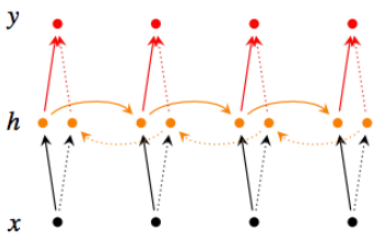


图 38: 双向 RNN

本质上，双向 RNN 就是两个单向 RNN 的结合。一个 RNN 进行正向计算，一个 RNN 进行反向计算，两个 RNN 的参数相互独立、互补共享，而输出层的结果由这两个单向 RNN 共同决定（例如将两个结果相加或拼接）。此外，将 RNN 更换为 LSTM、GRU，我们就可以得到双向 LSTM（BiLSTM）和双向 GRU（BiGRU）。

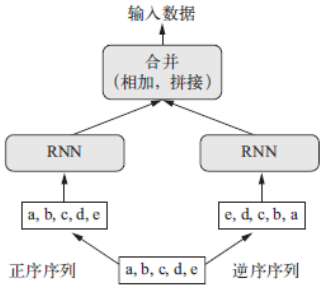


图 39: 双向 RNN 的另一种表示

基于深层 RNN 和双向 RNN，人们自然也会将它们组合起来（鸡尾酒乱炖大法好！），形成深层双向 RNN 网络，图40所示即为一种深层双向 RNN。

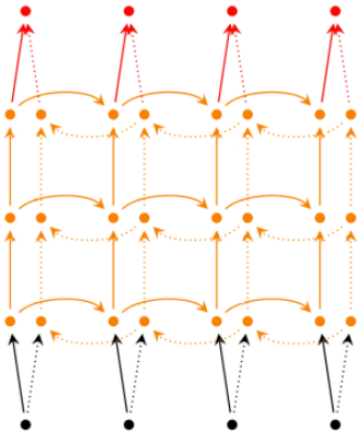


图 40: 一种深层双向 RNN

3.2.4 词嵌入

深度学习模型并不能处理原始文本，其只能处理数值张量。因此，文本向量化是数据预处理的必须过程。一般而言，文本向量化有两个步骤：分词和标记关联。分词就是将原始文本分解成一个个独立单元（字符或单词、短语），单元也被叫做标记（token）。而标记关联就是对标记进行编码，让每个标记与对应的数值向量相关联。编码方法有许多种，在第一次人工智能实验中我们就接触了 one-hot、tf-idf 等编码方式。而本次实验我们将使用一种新的编码方式，叫做词嵌入（word embedding）。

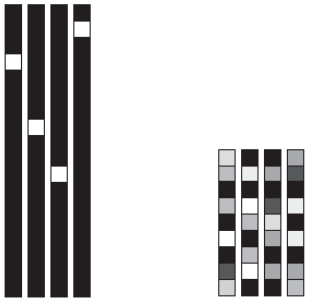


图 41: one-hot（左）与词嵌入（右）

相比于 one-hot 编码得到的二进制高维稀疏向量，词嵌入使用的词向量（word vector）是低维密集向量。不同于 one-hot 对单词的硬编码，词嵌入是从数据中学习得到词向量。词嵌入相比传统编码有以下优势：

(1) 维度更低、易于存储：如果有成千上万的单词，那么使用 one-hot 编码的话每个单词的对应向量维数也要成千上万，且维度会随着单词数量规模的增大而增大；相比之下词嵌入的向量维数更小且固定，一般为 256 ∼ 1024，不会随单词数量变化而变化；

(2) 词向量之间存在几何关系：传统编码只是单纯对单词进行标记，而词嵌入在将人类语言嵌入到低维空间时，可以通过训练使得含义相近的词被嵌入到更相似的词向量，这显然更有利于后续网络的训练；

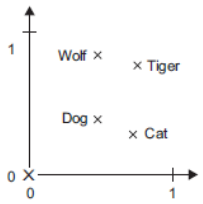


图 42: 词嵌入空间的简单示例

获取词嵌入的过程就是 word2vec。一般而言，获取词嵌入有两种方法：

(1) 在完成主任务的同时学习词嵌入；

(2) 在其他任务中预计算好词嵌入，然后再将其加载到模型中。

本次实验，我们将加载一个预先训练好的词嵌入模型 GloVe，并在其基础上进行调整学习，从而实现 word2vec。

3.2.5 CRF

随机场是由若干个位置组成的整体，当给每一个位置按照某种分布随机赋予一个值之后，其全体就叫做随机场。以词性标注为例，假如我们需要给包含十个词的句子进行词性标注，则每个词的词性都是从一个已知的词性集合中取选择的，给每个词选择完词性之后，就形成了一个随机场。条件随机场是随机场的特例，其基本思路是给定观察序列 X，输出标记序列 Y，通过计算 $P(Y|X)$ 来求解最优的标记序列。

使用单纯的 BiLSTM 来进行命名实体标注时，每个词都是选择所有标注中概率最大的一个进行标注，但是却没有考虑到相邻词之间的标注是否协调，这样就可能会出现一些明显错误的标注结果，比如说动词和动词相邻，机构名和人名相邻等等。在 BiLSTM 后面叠加 CRF 后可以完全避免这种不合逻辑的预测结果，因为 CRF 可以通过其特征函数对输入序列进行观察，学习到限定窗口大小之下各种标注之间的限制。把 CRF 用于关键词提取的任务中，也可以学习到关键词在一个句子中潜在的分布模式，以优化原始的 BiLSTM。

3.3 网络结构

3.3.1 原始双向 RNN/LSTM/GRU 网络结构

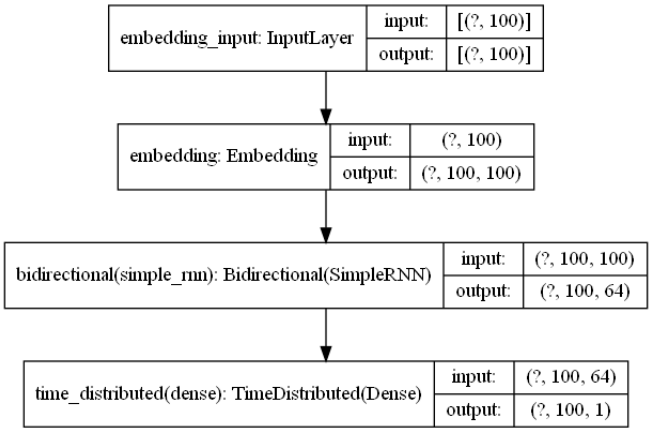


图 43: 双向 RNN 网络结构

我们实现的双向 RNN 网络结构图如图43所示。对于数据预处理，我们将每个句子扩充成 100 个单词/单位。首先我们使用 Embedding 层完成词嵌入工作，然后使用 units=32 的 RNN 组成双向 RNN 层。最后，我们使用 TimeDistribute 层在每个时间步上建立全连接，最终输出预测结果。

我们实现的双向 LSTM 网络结构相比于双向 RNN，我们只是将第三层的 simpleRNN 替换成了 LSTM。GRU 同理，故为节省报告空间，这两个网络结构我们没有贴图。

3.3.2 深度双向 LSTM 网络结构

在原始双向 LSTM 网络的基础上，我们实现了一个深度双向 LSTM。网络结构如图44所示。

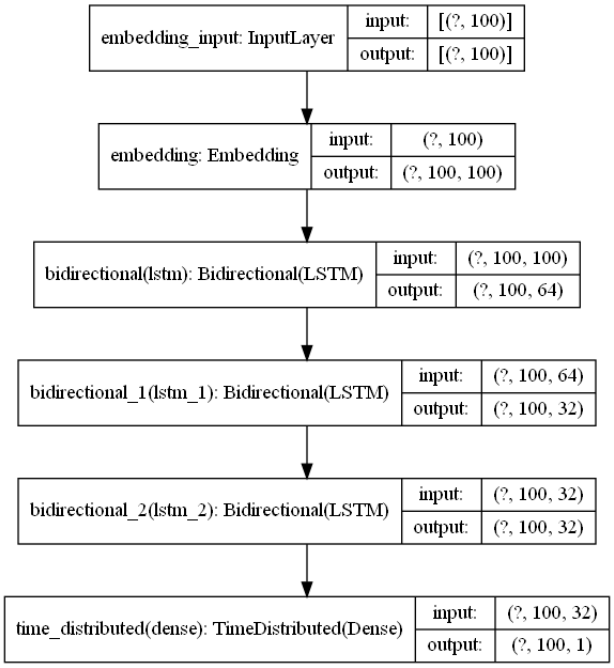


图 44: 深度双向 LSTM 网络结构

可见，我们搭建了三层双向 LSTM 层，最后对每个时间步使用全连接输出预测结果。

3.3.3 双向 LSTM+CRF 网络结构

在双向 LSTM 上，我们额外添加了 CRF 层，网络结构如图45所示。

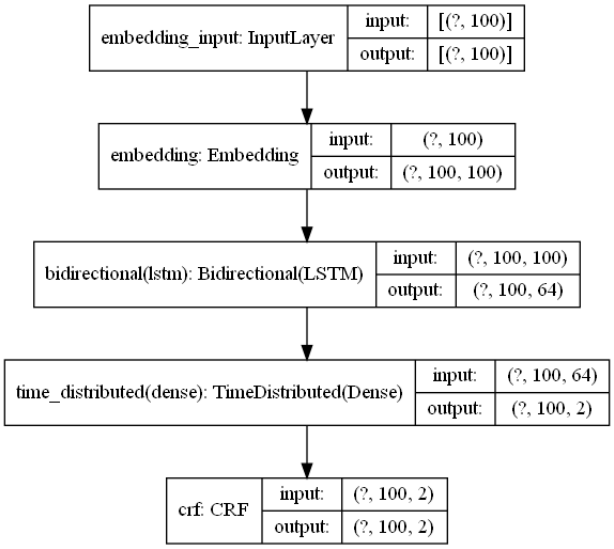


图 45: 双向 LSTM+CRF 网络结构

3.4 关键代码

在完成实验时，工作流程依次为读取数据、数据清洗与预处理、加载预训练词嵌入向量、构建模型与训练等内容。我们只展示模型构建代码，其余内容详见附录源码。

3.4.1 原始双向 RNN/LSTM/GRU

构建原始 RNN 模型代码如下所示：

```
def main():
    train_data, train_labels, embedding_matrix = my_load_data()
    train_labels = train_labels.reshape((train_labels.shape[0], train_labels.shape[1], 1))
    # 构建模型
    model = models.Sequential()
    model.add(layers.Embedding(max_words, Embedding_DIM, input_length=MAX_LEN))
    # 双向RNN
    model.add(layers.Bidirectional(layers.SimpleRNN(32, return_sequences=True)))
    model.add(layers.TimeDistributed(layers.Dense(1, activation='sigmoid')))
    # 加载预训练词嵌入向量glove
    model.layers[0].set_weights([embedding_matrix])
    model.layers[0].trainable = True
    model.compile('adam', loss='binary_crossentropy', metrics=['acc', f1])
    model.summary()
    # 训练
    history = model.fit(train_data, train_labels, batch_size=32, epochs=50,
          validation_split=0.1, verbose=1)
```

对于原始 LSTM 代码，我们只是将 simpleRNN 替换为 LSTM：

```
def main():
    train_data, train_labels, embedding_matrix = my_load_data()
    # 构建模型
    model = models.Sequential()
    model.add(layers.Embedding(max_words, Embedding_DIM, input_length=MAX_LEN))
    # 双向LSTM
    model.add(layers.Bidirectional(layers.LSTM(32, return_sequences=True)))
    model.add(layers.TimeDistributed(layers.Dense(1, activation='sigmoid')))
    model.compile('adam', loss='binary_crossentropy', metrics=['acc', f1])
    model.summary()
    ......
```

GRU 同理：

```
def main():
    ......
    # 双向GRU
    model.add(layers.Bidirectional(layers.GRU(32, return_sequences=True)))
    ......
```

3.4.2 深层双向 LSTM

构建深层双向 LSTM 模型代码如下所示：

```
def main():
    train_data, train_labels, embedding_matrix = my_load_data()
```

```
    # 构建模型
    model = models.Sequential()
    model.add(layers.Embedding(max_words, Embedding_DIM, input_length=MAX_LEN))
    model.add(layers.Bidirectional(layers.LSTM(32, return_sequences=True)))
    model.add(layers.Bidirectional(layers.LSTM(16, return_sequences=True)))
    model.add(layers.Bidirectional(layers.LSTM(16, return_sequences=True)))
    model.add(layers.TimeDistributed(layers.Dense(1, activation='sigmoid')))
    model.layers[0].set_weights([embedding_matrix])
    model.layers[0].trainable = True
    model.compile('adam', loss='binary_crossentropy', metrics=['acc', f1])
    model.summary()
    plot_model(model, show_shapes=True, to_file='model.png')
    history = model.fit(train_data, train_labels, batch_size=32, epochs=50,
          validation_split=0.1, verbose=1)
    ......
```

3.4.3 双向 LSTM+CRF

构建双向 LSTM+CRF 模型代码如下所示：

```
def main():
    ......
    # 构建模型
    model = models.Sequential()
    model.add(layers.Embedding(max_words, Embedding_DIM, input_length=MAX_LEN))
    model.add(layers.Bidirectional(layers.LSTM(32, return_sequences=True)))
    model.add(layers.TimeDistributed(layers.Dense(2, activation='softmax')))
    crf_layer = CRF(2)
    model.add(crf_layer)
    model.layers[0].set_weights([embedding_matrix])
    model.layers[0].trainable = True
    model.compile('adam', loss=crf_loss, metrics=[crf_viterbi_accuracy, f1, 'acc'])
    model.summary()
    ......
```

3.5 实验过程与结果分析

虽然我们把关键词提取任务视为一个使用 BIO 标注法的序列标注问题，但为了让任务更加简化，我们做出了以下假设：不同的关键词不会紧靠在一起。这样，我们就不需要 **I-X** 标记，这样就可以将问题简化为二分类问题：对于一个句子的每个单词，其 label 要么是 0（非关键词），要么是 1（关键词）。当出现连续的 1 时，说明这些单词属于同一个关键词。

此外，在后续实验中，我们默认加载预训练的词嵌入向量 GloVe，并设置 embedding 层是可训练的，以得到更好的实验结果。

3.5.1 原始双向 RNN/LSTM/GRU

我们使用一个原始的双向 RNN 网络完成该任务。网络结构如3.3.1的图43所示。设置 optimizer='adam', loss='binary_crossentropy', batch_size=32, epochs=50。划分训练集与验证集的比例为 9:1，最终训练结果如图46、47和48所示。

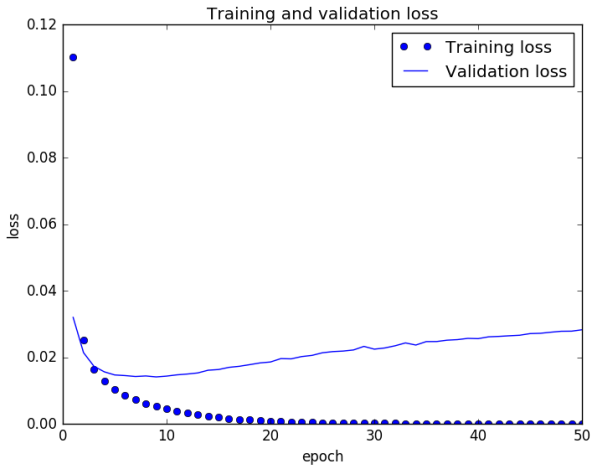


图 46: BiRNN 的训练损失

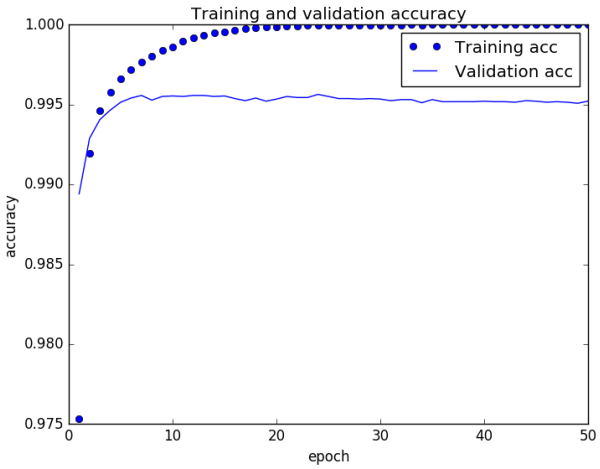


图 47: BiRNN 的训练准确率

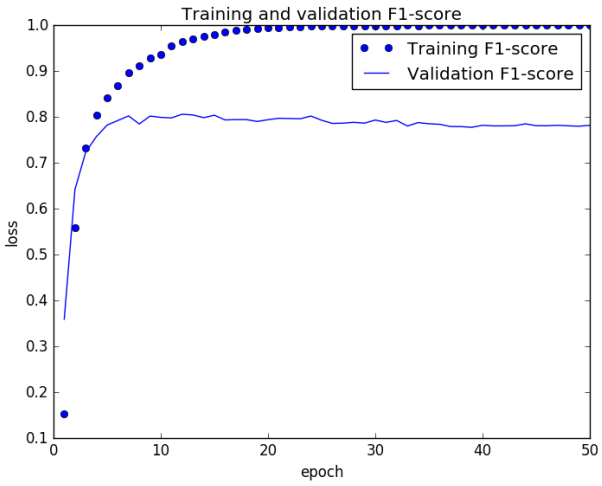


图 48: BiRNN 的训练 F1 score

可见，在第五轮迭代后模型出现过拟合现象。此外，在第 6 次迭代后准确率一直在 **99.5%** 以上。这是因为我们的数据存在不平衡，具体来说每个句子里面绝大部分单词都是非关键词、被标记为 0，只有一小部分甚至没有单词是关键词。因此，我们引入了统计学中的 F1 score 衡量模型的好坏，如图48所示。可见，模型在验证集上最好的 F1 score 达到 **80.8%**，一般在 79% 处波动。

将 simpleRNN 更换为 LSTM 和 GRU 后，其他参数指标均不变，最终训练结果如图49至54所示。

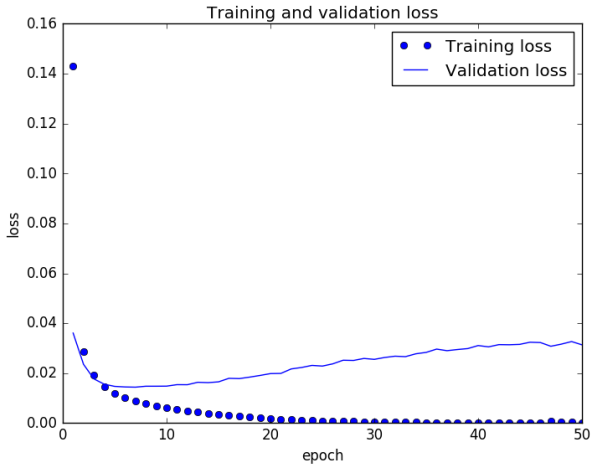


图 49: BiLSTM 的训练损失

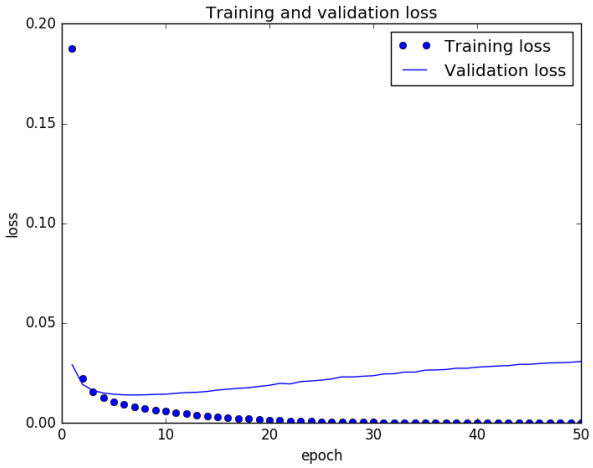


图 50: BiGRU 的训练损失

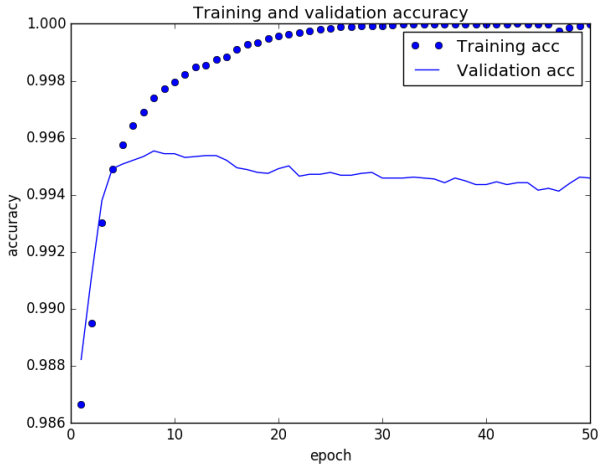


图 51: BiLSTM 的训练准确率

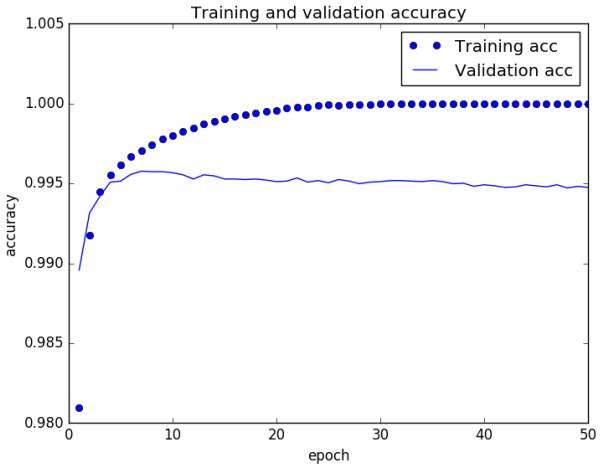


图 52: BiGRU 的训练准确率

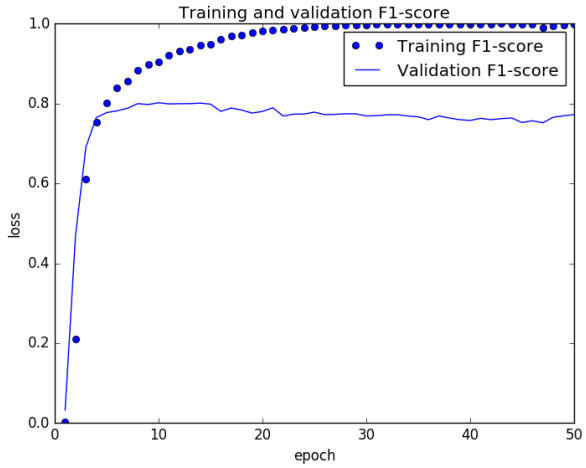


图 53: BiLSTM 的训练 F1 score

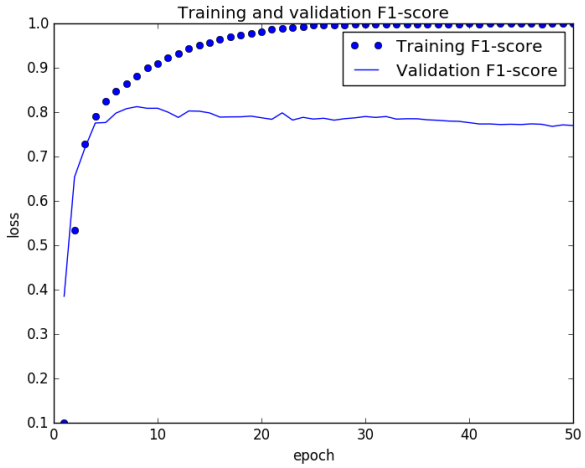


图 54: BiGRU 的训练 F1 score

可见，双向 LSTM、双向 GRU 和双向 RNN 差别不大，性能相似。相比较而言，双向 LSTM 和双向 GRU 在 F1 score 指标上比双向 RNN 稍好一点，都达到了 **80%**。此外，在相同迭代次数下，loss 值也有所下降。

3.5.2 深度双向 LSTM

我们使用3.3.2中的深度双向 LSTM，设置 optimizer='adam', loss='binary_crossentropy', batch_size=32, epochs=50。划分训练集与验证集的比例为 9:1，最终训练结果如图55、56和57所示。

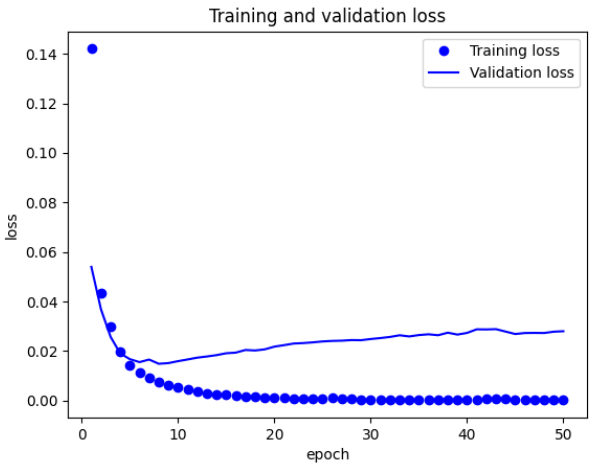


图 55: 深度 BiLSTM 的训练损失

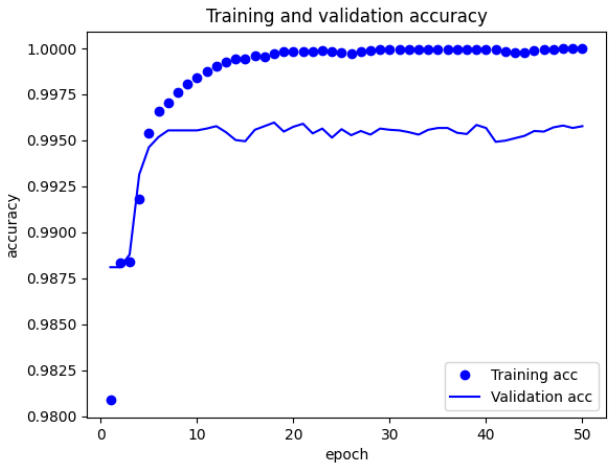


图 56: 深度 BiLSTM 的训练准确率

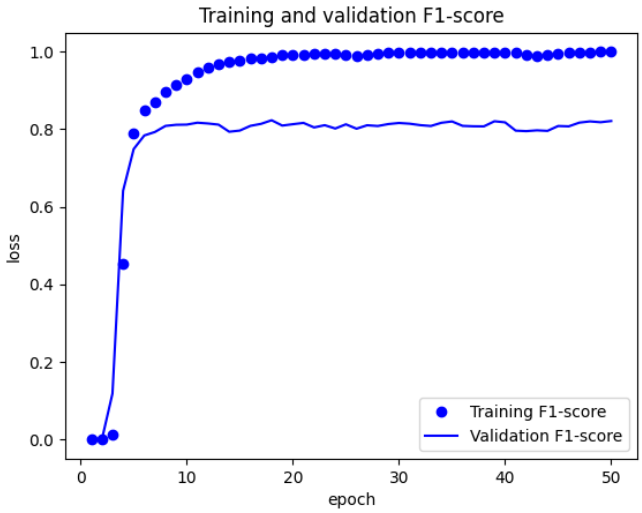


图 57: 深度 BiLSTM 的训练 F1 score

与单层双向 LSTM 的实验结果相比，两个网络在训练集和验证集上的 loss 曲线差别不大，均在第五次迭代左右开始出现过拟合现象。而深层双向 LSTM 网络在验证集上的准确率和 F1 score 均比单层双向 LSTM 要高，分别在 **99.6%** 和 **81%** 左右。这说明了该深层双向 LSTM 性能要比单层双向 LSTM 性能要好一些，但提升不是特别大。

3.5.3 双向 LSTM+CRF

我们使用3.3.3中的双向 LSTM+CRF，其中 CRF 层我们调用 keras 的拓展 keras_contrib 库。需要注意的是，该拓展中的 CRF 函数对 tensorflow 版本存在要求，版本号应在 1.13.1 以下。我们设置 optimizer='adam', loss=crf_loss, batch_size=32, epochs=25，训练结果如图58和59所示。

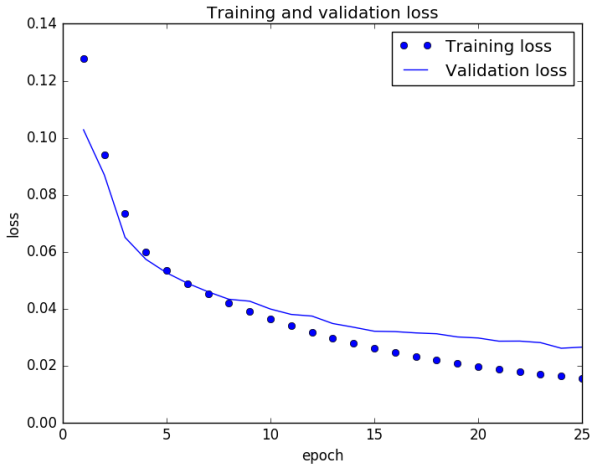


图 58: BiLSTM+CRF 的训练损失

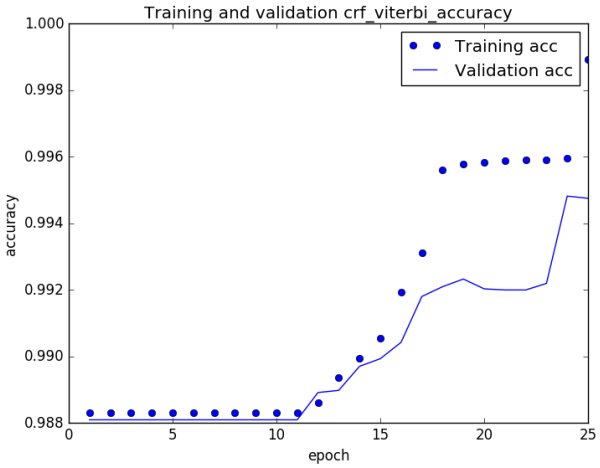


图 59: BiLSTM+CRF 的训练准确率

可见，双向 LSTM+CRF 在 25 轮迭代后验证集上的 loss 达到 **0.018**，准确率达到了 **99.6%**，而 F1 score 达到了 **98%**，性能表现均比前面的网络出色。

4 创新汇总

4.1 CNN

(1) 尝试了训练集的数据增强；

(2) 尝试了 Dropout 及性能比较；

(3) 实现了残差网络及性能比较；

4.2 RNN

(1) 使用了 LSTM，GRU 并实现性能比较；

(2) 尝试了深层/双向循环神经网络并进行性能比较；

(3) 使用了 GloVe 词嵌入模型；

(4) 尝试将 CRF 与神经网络结合使用；

5 实验感想

本次实验开始 “炼丹”，要求我们使用 CNN 和 RNN 完成图片分类任务和关键词提取任务。虽然本次实验允许我们调用现有库，但我们依然花了很多时间去寻找表现更好的模型以及调参。受到时间、经验和机器性能限制，我们未能得到性能十分优异的网络，但还是学了很多东西，取得了不错的效果。

希望我们能顺利完成下一次实验。