

实验三：1神经网络

一、实验目的

- 1、理解并掌握神经网络基本要素：激活函数、反向传播、交叉熵
- 2、理解神经网络反向传播过程
- 3、了解利用 sklearn 构建神经网络过程。
- 4、学会使用 pytorch 搭建出卷积神经网络

二、实验内容

- 1、用 Python 实现常用激活函数。
- 2、用 sklearn 构建神经网络模型，并通过鸢尾花数据集中鸢尾花的 4 种属性与种类对神经网络模型进行训练。我们会调用你训练好的神经网络模型，来对未知的鸢尾花进行分类。
- 3、使用 pytorch 搭建出卷积神经网络并对手写数字进行识别

二、实验步骤

- 1、用 Python 实现常用激活函数。

神经网络是由一个个神经元也就是感知机组成，感知机数学模型如下：

$$f(x) = \text{sign}(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$
$$\text{sign} = \begin{cases} -1 & x < 0 \\ +1 & x \geq 0 \end{cases}$$

其中的 sign 函数被称为阶跃函数，是用来引入非线性因素的。而在神经网络里，称其为激活函数。

如果不用激活函数，每一层输出都是上层输入的线性函数，无论神经网络有多少层，输出都是输入的线性组合，这种情况就是最原始的感知机。只能对线性数据进行分类；如果使用的话，激活函数给神经元引入了非线性因素，使得神经网络可以任意逼近任何非线性函数，这样神经网络就可以应用到众多的非线性模型中，对非线性数据进行分类：

神经网络中经常使用的一个激活函数就是 sigmoid 函数，函数公式如下：

$$\text{sign} = \begin{cases} -1 & x < 0 \\ +1 & x \geq 0 \end{cases}$$

现在常使用 relu 激活函数，函数公式如下：

$$\text{relu}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

relu 函数在 x 大于等于 0 时，输出的是 x 本身，函数变化率恒为 1，这样就避免了梯度消失的情况。当 x 小于 0 时，输出为 0，即神经元不被激活，这样也符合人脑内并不是所有神经元同时被激活的情况。并且，relu 函数计算成本非常低，所以在神经网络过深时常使用 relu 函数作为激活函数。

```

1. #encoding=utf8
2.
3. def relu(x):
4.     '''
5.     x: 负无穷到正无穷的实数
6.     '''
7.     #***** Begin *****#
8.
9.     #***** End *****#

```

2、用 sklearn 构建神经网络模型，并通过鸢尾花数据集中鸢尾花的 4 种属性与种类对神经网络模型进行训练。我们会调用你训练好的神经网络模型，来对未知的鸢尾花进行分类。

鸢尾花数据集是一类多重变量分析的数据集。通过花萼长度，花萼宽度，花瓣长度，花瓣宽度 4 个属性预测鸢尾花卉属于(Setosa, Versicolour, Virginica)三个种类中的哪一类。

想要使用该数据集可以使用如下代码：

```

1. #获取训练数据
2. train_data = pd.read_csv('./step2/train_data.csv')
3. #获取训练标签
4. train_label = pd.read_csv('./step2/train_label.csv')
5. train_label = train_label['target']
6. #获取测试数据
7. test_data = pd.read_csv('./step2/test_data.csv')

```

数据集中部分数据与标签如下图所示：

sepal length	sepal width	petal length	petal width
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
5.7	2.8	4.1	1.3
6.2	3.4	5.4	2.3
5.1	2.5	3	1.1
7	3.2	4.7	1.4
6.1	2.6	5.6	1.4
7.6	3	6.6	2.1
5.2	4.1	1.5	0.1
6.2	2.2	4.5	1.5
7.3	2.9	6.3	1.8

target
0
0
1
2
1
1
2
2

神经网络的训练方法跟逻辑回归相似，也是使用梯度下降算法来更新模型的参数，既然要使用梯度下降算法，就要知道损失函数对参数的梯度。反向传播算法能够快速计算这些梯度。反向传播算法一共分为两部分：前向传播与反向传播。

前向传播指的是数据 x 从神经网络输入层，与当层的权重相乘，再加上当层的偏置，所得到的值经过激活函数激活后，再输入到下一层。最后，在输出层所得到的值经过 softmax 函数转化为网络对数据的预测。其中，输出层的 Z 值要经过 softmax 函数，转化为预测值。前向传播的主要目的就是得到预测值，再算出交叉熵损失函数。

交叉熵（cross entropy）是深度学习中常用的一个概念，一般用来求目标与预测值之间的差距。机器学习是用网络训练出来的分布 q 来表示真实分布 p ，此时当两者的交叉熵越小时，模型训练的结果越接近样本的真实分布，这也是交叉熵被用来作为损失函数的原因。

通过前向传播能够求出损失函数，而反向传播就是利用前向传播得到的损失函数对参数求梯度，即每个参数的偏导。

之所以称为反向传播，是因为我们在利用链式法则的时候对各个参数求偏导的传播顺序跟前向传播相反，如我们要求 loss 对 w_1 的偏导，则要先求出 loss 对 a

的偏导，再求出 a 对 z^2 的偏导，再求出 z^2 对 a^1 的偏导，再求出 a^1 对 z^1 的偏导，再求出 z^1 对 w^1 的偏导，然后全部相乘就得到 loss 对 w^1 的偏导了。公式如下：

$$\frac{\partial \text{loss}}{\partial w^1} = \frac{\partial \text{loss}}{\partial a^1} \cdot \frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}$$

所以反向传播的目的就是求出损失函数对各个参数的梯度。最后我们就可以用梯度下降算法来训练我们的神经网络模型了。

sklearn 中的神经网络

MLPClassifier 的构造函数中有四个常用的参数可以设置：

- **solver**: MLP 的求解方法 lbfgs 在小数据上表现较好，adam 较为鲁棒，sgd 在参数调整较优时会有最佳表现（分类效果与迭代次数）；sgd 标识随机梯度下降；
- **alpha**: 正则项系数，默认为 L2 正则化，具体参数需要调整；
- **hidden_layer_sizes**: hidden_layer_sizes=(3, 2) 设置隐藏层 size 为 2 层隐藏层，第一层 3 个神经元，第二层 2 个神经元。
- **max_iter**: 最大训练轮数。

和 sklearn 中其他分类器一样，MLPClassifier 类中的 fit 函数用于训练模型，fit 函数有两个向量输入：

- **X**: 大小为**[样本数量,特征数量]**的 ndarray，存放训练样本；
- **Y**: 值为整型，大小为**[样本数量]**的 ndarray，存放训练样本的分类标签。

MLPClassifier 类中的 predict 函数用于预测，返回预测标签，predict 函数有一个向量输入：

- **X**: 大小为**[样本数量,特征数量]**的 ndarray，存放预测样本。

MLPClassifier 的使用代码如下：

```
1. from sklearn.neural_network import MLPClassifier
2. mlp = MLPClassifier(solver='lbfgs', max_iter=10,
3.                     alpha=1e-5, hidden_layer_sizes=(3, 2))
4. mlp.fit(X_train, Y_train)
5. result = mlp.predict(X_test)
```

编程要求：

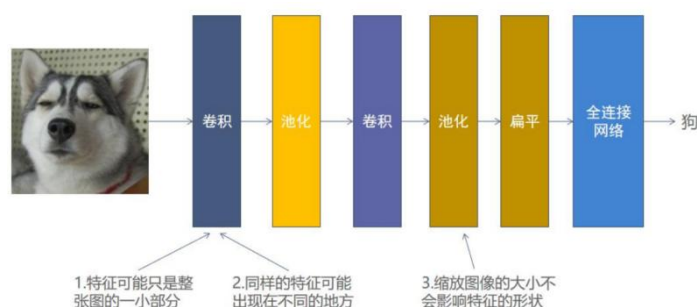
使用 `sklearn` 构建神经网络模型，利用训练集数据与训练标签对模型进行训练，然后使用训练好的模型对测试集数据进行预测，并将预测结果保存到 `./step2/result.csv` 中。保存格式如下：

result
0
1
1
1
0

```
1. #encoding=utf8
2. import os
3. if os.path.exists('./step2/result.csv'):
4.     os.remove('./step2/result.csv')
5. #***** Begin *****#
6.
7. #***** End *****#
```

3、使用 `pytorch` 搭建出卷积神经网络并对手写数字进行识别。

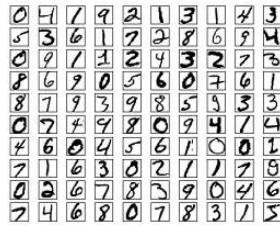
卷积神经网络是一种具有局部连接、权重共享等特性的深层前馈神经网络。将卷积，池化，全连接网络进行合理的组合，就能构建出属于自己的神经网络来识别图像中是猫还是狗。通常来说卷积，池化可以多叠加几层用来提取特征，然后接上一个全连接网络来进行分类。大致结构如下：



`pytorch` 构建卷积神经网络项目流程

数据集介绍与加载数据

本次使用数据集为 `mnist` 手写数字数据集，简单来讲就是如下的东西：



数据集分为训练集与测试集，训练集中一共有 60000 张图片，测试集中一共有 10000 张图片，每张图片大小为 28X28x1。图片标签为对应的数字，如 8 对应的 label 为 8，若使用 onehot 编码则对应的 label 为：[0,0,0,0,0,0,0,1,0]。

为节约计算时间，我们取训练集中的 6000 张图片用来训练，测试集中的 600 张进行测试。使用 pytorch 加载数据集方法如下：

```
1.  #加载数据
2.  import torchvision
3.  train_data = torchvision.datasets.MNIST(
4.      root='./step3/mnist/',
5.      train=True,                      # this is training data
6.      transform=torchvision.transforms.ToTensor(),  # Converts a
PIL.Image or numpy.ndarray to
7.      download=False,
8.  )
9.  #取 6000 个样本为训练集
10. train_data_tiny = []
11. for i in range(6000):
12.     train_data_tiny.append(train_data[i])
13. train_data = train_data_tiny
```

构建模型

加载好数据集，就需要构建卷积神经网络模型：

```
1.  #构建卷积神经网络模型
2.  class CNN(nn.Module):
3.      def __init__(self):
4.          super(CNN, self).__init__()
5.          self.conv1 = nn.Sequential(          # input shape (1, 28, 28)
6.              nn.Conv2d(
7.                  in_channels=1,              # input height
8.                  out_channels=16,            # n_filters
9.                  kernel_size=5,              # filter size
10.                 stride=1,                    # filter movement/step
```

```

11.         padding=2,                # if want same width and length
of this image after conv2d, padding=(kernel_size-1)/2 if stride=1
12.     ),                            # output shape (16, 28, 28)
13.     nn.ReLU(),                     # activation
14.     nn.MaxPool2d(kernel_size=2),   # choose max value in 2x2 area,
output shape (16, 14, 14)
15. )
16.     self.conv2 = nn.Sequential(     # input shape (16, 14, 14)
17.         nn.Conv2d(16, 32, 5, 1, 2), # output shape (32, 14, 14)
18.         nn.ReLU(),                 # activation
19.         nn.MaxPool2d(2),           # output shape (32, 7, 7)
20.     )
21.     self.out = nn.Linear(32 * 7 * 7, 10) # fully connected layer, output
10 classes
22.     def forward(self, x):
23.         x = self.conv1(x)
24.         x = self.conv2(x)
25.         x = x.view(x.size(0), -1)    # flatten the output of conv2
to (batch_size, 32 * 7 * 7)
26.         output = self.out(x)
27.         return output
28.     cnn = CNN()

```

需要指出的几个地方：

1. `class CNN` 需要继承 `Module`
2. 需要调用父类的构造方法：`super(CNN, self).__init__()`
3. 在 `Pytorch` 中激活函数 `Relu` 也算是一层 `layer`
4. 需要实现 `forward()` 方法，用于网络的前向传播，而反向传播只需要调用 `Variable.backward()` 即可。

定义好模型后还要构建优化器与损失函数：

`torch.optim` 是一个实现了各种优化算法的库。使用方法如下：

1. `#SGD` 表示使用随机梯度下降方法，`lr` 为学习率，`momentum` 为动量项系数
2. `optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)`
3. `#交叉熵损失函数`
4. `loss_func = nn.CrossEntropyLoss()`

训练模型

在定义好模型后，就可以根据反向传播计算出来的梯度，对模型参数进行更新，在 `pytorch` 中实现部分代码如下：

```

1.  #将梯度清零
2.  optimizer.zero_grad()
3.  #对损失函数进行反向传播
4.  loss.backward()
5.  #训练
6.  optimizer.step()

```

保存模型

在 pytorch 中使用 torch.save 保存模型，有两种方法，第一种：

保存整个模型和参数，方法如下：

```
torch.save(model, PATH)
```

第二种为官方推荐，只保存模型的参数，方法如下：

```
torch.save(model.state_dict(), PATH)
```

加载模型

对应两种保存模型的方法，加载模型也有两种方法，第一种如下：

```
model = torch.load(PATH)
```

第二种：

```

1.  #CNN()为你搭建的模型
2.  model = CNN()
3.  model.load_state_dict(torch.load(PATH))

```

如果要对加载的模型进行测试，需将模型切换为验证模式

```
model.eval()
```

```

1.  #mini_batch
2.  train_loader = Data.DataLoader(dataset=train_data, batch_size=64,
shuffle=True)

```

阅读代码中注释部分要求，补充实现 begin-end 间代码，并运行测试结果

```

1.  #encoding=utf8
2.  import torch
3.  import torch.nn as nn
4.  from torch.autograd import Variable
5.  import torch.utils.data as Data
6.  import torchvision
7.  import os

```

```

8.     if os.path.exists('./step3/cnn.pkl'):
9.         os.remove('./step3/cnn.pkl')
10.
11.     #加载数据
12.     train_data = torchvision.datasets.MNIST(
13.         root='./step3/mnist/',
14.         train=True,                                # this is training data
15.         transform=torchvision.transforms.ToTensor(), # Converts a
16.         download=False,
17.     )
18.     #取 6000 个样本为训练集
19.     train_data_tiny = []
20.
21.     for i in range(6000):
22.         train_data_tiny.append(train_data[i])
23.
24.     train_data = train_data_tiny
25.
26.     #***** Begin *****#
27.
28.
29.     #***** End *****#
30.     #保存模型
31.     torch.save(cnn.state_dict(), './step3/cnn.pkl')

```

四、实验报告要求

参见实验报告模板

实验三：2深度学习-CNN

一、实验目的

- 1、理解经典的卷积神经网络模型： LeNet 模型和 AlexNet 模型。
- 2、掌握基于 Pytorch 的经典的卷积神经网络模型的构建方法。

二、实验内容

- 1、编写基于 Pytorch 的 LeNet 模型。
- 2、编写基于 Pytorch 的 AlexNet 模型。

二、实验步骤

- 1、简单的卷积网络的搭建-LeNet 模型

LeNet 模型是一个早期用来识别手写数字图像的卷积神经网络。LeNet 的网络结构如下图所示：

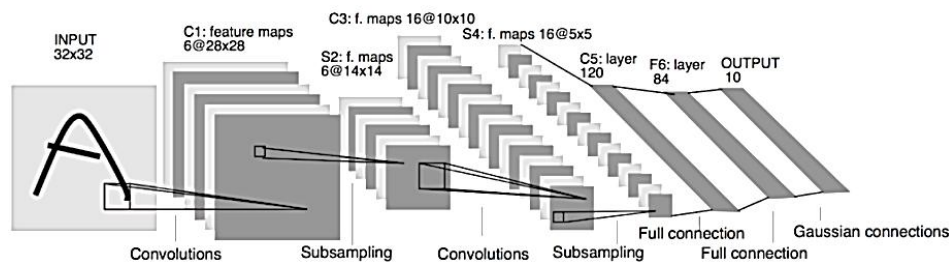


图 1 LeNet 模型结构图

在图中，INPUT 表示输入层，定义的输入图像大小为 32*32。

在 LeNet 的第一层中定义了 6 通道的卷积核，这里没有涉及填充和改变默认步幅，所以已知输入为 32*32，输出为 28*28，根据公式输入大小-卷积核大小+1=输出大小，可以得出卷积核的形状 5*5，为对于每一个卷积核可以训练的参数为 5*5+1(其中 1 为偏置项)，因此这一层一共可以训练的参数为(5*5+1)*6。第二层为池化层，输入为 28*28 的矩阵，池化层形状为 2*2，步幅为 2，原论文采样的方式为四个输入值相加，然后乘以一个可训练的参数，再加上偏置项，通过 Sigmoid 函数进行激活。第三层属于卷积层，输入为 S2 层中的 6 个通道的输出，卷积核大小为 5*5，但是输出通道数增加到 16，所以可以训练的参数为(5*5+1)*16。第四层是池化层，形状为 2*2，步幅为 2，原论文采样的方式为四个输入值相加，然后乘以一个可训练的参数，再加上偏置项，通过 Sigmoid 函数

进行激活。接下来三层是全连接层（ Fully Connected，简称 FC ），首先需要计算出上层的输出为 $16 \times 5 \times 5 = 400$ ，所以接下来三层的参数矩阵应为 400×120 ， 120×84 ， 84×10 。层之间的激活函数也采用的是 Sigmoid 函数。

Pytorch 中模型的搭建

Module 类是 nn 模块里提供的一个模型构造类，是所有神经网络模块的基类，可以继承它来定义我们想要的模型。下面展示一个简单的卷积网络的实例，可以在待会的实战中采用相同的方法来搭建：

```
1. import torch
2. from torch import nn    # 导入 nn 模块
3. class Sample(nn.Module):
4.     def __init__(self):
5.         super(Sample, self).__init__()
6.         self.conv = nn.Sequential(
7.             nn.Conv2d(1, 1, 5),    # 卷积层
8.             nn.Sigmoid(),    # 激活函数
9.             nn.MaxPool2d(2, 2),    # 最大池化层
10.        )
11.        self.fc = nn.Sequential(
12.            nn.Linear(14*14, 10),    # 全连接层
13.            nn.Sigmoid(),    # 激活函数
14.        )
15.
16.    def forward(self, img):    # 定义前向计算
17.        feature = self.conv(img)    # 卷积层
18.        output = self.fc(feature.view(img.shape[0], -1))    # 全连接层
19.        return output
```

在样例中，我们定义了一层卷积，一层池化，一层全连接层，也使用了激活函数，可以说在最简单的 LeNet 中需要用到的组件都有了使用。

将 Sample 的实例对象打印出来就可以清楚地看到上面代码中搭建出的模型的整体结构：

```
1. Sample(
2.   (conv): Sequential(
3.     (0): Conv2d(1, 1, kernel_size=(5, 5), stride=(1, 1))
4.     (1): Sigmoid()
5.     (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
        ceil_mode=False)
```

```

6.     )
7.     (fc): Sequential(
8.         (0): Linear(in_features=196, out_features=10, bias=True)
9.         (1): Sigmoid()
10.    )
11. )

```

阅读代码中注释部分要求，补充实现 begin-end 间代码，并运行测试结果。

```

1.  import torch
2.  from torch import nn
3.  class LeNet(nn.Module):
4.      def __init__(self):
5.          super(LeNet, self).__init__()
6.          '''
7.              这里搭建卷积层，需要按顺序定义卷积层、
8.              激活函数、最大池化层、卷积层、激活函数、最大池化层，
9.              具体形状见测试说明
10.         '''
11.         self.conv = nn.Sequential(
12.             ##### Begin #####
13.
14.
15.             ##### End #####
16.         )
17.         '''
18.             这里搭建全连接层，需要按顺序定义全连接层、
19.             激活函数、全连接层、激活函数、全连接层，
20.             具体形状见测试说明
21.         '''
22.         self.fc = nn.Sequential(
23.             ##### Begin #####
24.
25.             ##### End #####
26.         )
27.
28.     def forward(self, img):
29.         '''
30.             这里需要定义前向计算
31.         '''
32.         ##### Begin #####
33.
34.         ##### End #####

```

2、编写基于 Pytorch 的 AlexNet 模型。

AlexNet 模型的网络结构如下图所示：

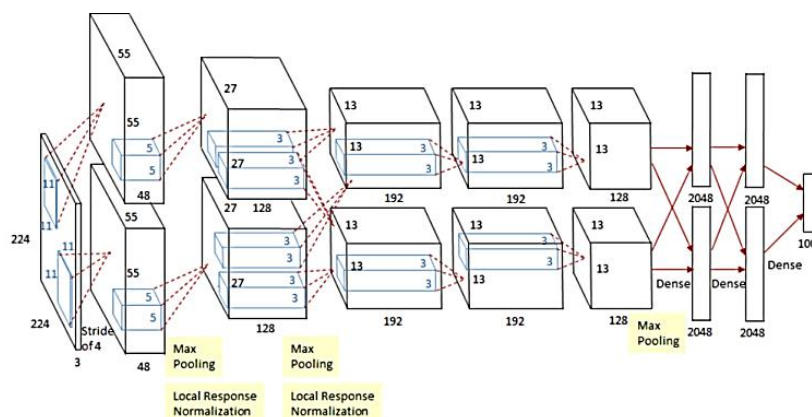


图 1 AlexNet 模型结构示意图

AlexNet 模型一共有八层，包含五个卷积层和三个全连接层，对于每一个卷积层，均包含了 ReLU 激活函数和局部响应归一化处理，接着进行了池化操作。

在模型设计时，通道数需要将图中上下两个部分的通道数相加来得到整个模型单层的通道数，例如第一层的通道数就是 $96=48+48$ 。从图中可以发现，输入的图像大小为 $224*224$ ，但是考虑到图像是由 RGB 组成的三通道，因此输入图像的形状是 $3*224*224$ 。(在 AlexNet 模型的实际处理过程中会通过预处理，图像的输入是 $3*227*227$)。

AlexNet 第一层中的卷积窗口形状是 $96*11*11$ 。第二层中的卷积窗口形状减小到 $5*5$ 但是通道数从 96 增加值 256，之后采用 $3*3$ 的卷积核，但是通道进一步增加至 384，然后继续一层 $384*384*3*3$ 的卷积层，之后是 $384*256*3*3$ 的最后一层卷积层。此外，第一、第二和第五个卷积层之后都使用了窗口形状为 $3*3$ 、步幅为 2 的最大池化层。接下来是两个输出个数为 4096 的全连接层。

同时 AlexNet 通过丢弃法来控制全连接层的模型复杂度，同时提高了模型的泛化能力。`nn.Dropout()` 就是 Pytorch 中用于丢弃法的模块，使用方法为 `nn.Dropout(0.5)`。

阅读代码中注释部分要求，补充实现 begin-end 间代码，并运行测试结果。

```
1. import torch
2. from torch import nn
3. class AlexNet(nn.Module):
4.     def __init__(self):
5.         super(AlexNet, self).__init__()
```

```

6.      '''
7.      这里搭建卷积层，需要按顺序定义卷积层、
8.      激活函数、最大池化层、卷积层、激活函数、
9.      最大池化层、卷积层、激活函数、卷积层、
10.     激活函数、卷积层、激活函数、最大池化层，
11.     具体形状见测试说明
12.     '''
13.     self.conv = nn.Sequential(
14.         ##### Begin #####
15.
16.         ##### End #####
17.     )
18.     '''
19.     这里搭建全连接层，需要按顺序定义
20.     全连接层、激活函数、丢弃法、
21.     全连接层、激活函数、丢弃法、全连接层，
22.     具体形状见测试说明
23.     '''
24.     self.fc = nn.Sequential(
25.         ##### Begin #####
26.
27.         ##### End #####
28.     )
29.
30.     def forward(self, img):
31.         '''
32.         这里需要定义前向计算
33.         '''
34.         ##### Begin #####
35.
36.         ##### End #####

```

四、实验报告要求

参见实验报告模板

实验三：3深度学习-RNN

一、实验目的

- 1、学习 RNN 循环神经网络的基本概念并构建单个 RNNCell。
- 2、了解 LSTM 的核心思想，创建 LSTM 网络的一个 LSTMCell 。
- 3、学习如何一次执行得到 RNNCell 多步调用 call 方法得到的结果，掌握构建堆叠 RNNCell 的方法。

二、实验内容

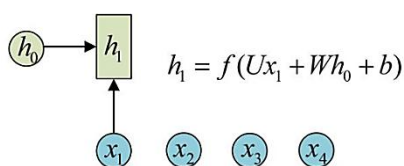
- 1、学习 RNN 循环神经网络的基本概念并构建单个 RNNCell。
- 2、了解 LSTM 的核心思想，根据教程指导创建 LSTM 网络的一个 LSTMCell 。
- 3、学习如何一次执行得到 RNNCell 多步调用 call 方法得到的结果，其次掌握构建堆叠 RNNCell 的方法。

三、实验步骤

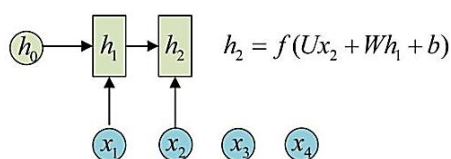
- 1、学习 RNN 循环神经网络的基本概念并构建单个 RNNCell。

如果要学习 TensorFlow 中的 RNN ，首先需要了解 RNNCell ，它是 TensorFlow 中实现 RNN 的基本单元，每个 RNNCell 都有一个 call 方法，使用方式是： $(output, next_state) = call(input, state)$ 。

借助图片来说可能更容易理解。假设我们有一个初始状态 h_0 ，还有输入 x_1 ，调用 $call(x_1, h_0)$ 后就可以得到 $(output_1, h_1)$ ：

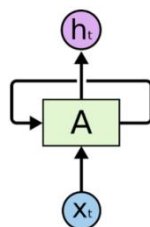


再调用一次 $call(x_2, h_1)$ 就可以得到 $(output_2, h_2)$ ：



也就是说，每调用一次 RNNCell 的 call 方法，就相当于在时间上“推进

了一步”，隐含层的 h_1 代表了第一个 RNNCell 的状态，这个状态记录了 X_1 输入后 RNNCell 积攒的有关先前输入序列的知识，当把 h_1 传给第二个 RNNCell，网络就完成了“经验”的传递工作，当我们把许许多多这样的 RNNCell 联结，就可以聚合成为下图所示的网络结构：



于是有关时间序列的经验传递再结合当前的输入就可以构成具有“循环”结构的 RNN (循环神经网络)。

单个 RNNCell 的构建

方法 `tf.nn.rnn_cell.BasicRNNCell()` 可以构造一个 `BasicRNNCell` 类的实例即一个最基本的 RNNCell，这个方法有一个参数 `num_units` 需要设置一下

有了 RNNCell 后，需要 `call` 方法看看这个网络能给什么样的输出，不过 `BasicRNNCell` 类里的这一调用方法叫做 `call()`。完整的过程如下：

```
cell=tf.nn.rnn_cell.BasicRNNCell(num_units=5)
```

新建一个输入值 x_1 。

```
x1=tf.placeholder(tf.float32,[batch_size,n_inputs])
```

`batch_size` 代表批量输入值的大小，`n_inputs` 代表单个输入值的维度

构建 `BasicRNNCell`，含有 `n_units` 个神经元

```
cell=tf.nn.rnn_cell.BasicRNNCell(num_units=n_units)
```

将初始状态初始化为全零

```
h0=cell.zero_state(batch_size=batch_size,dtype=tf.float32)
```

`output`，`h1` 分别代表了这个 RNNCell 的输出和当前状态。

```
output,h1=cell.__call__(x1,h0)
```

打印当前状态

```
print(h1)
```

编程要求：根据提示，在右侧编辑器的 `begin-end` 间补充代码，创建一个包含 `a` 个神经元，可以接受一个 `shape=[b,c]`，类型为 `float32` 的张量作为输入的

BasicRNNCell，并打印一次输入后该 BasicRNNCell 的输出。

测试说明：平台会对你编写的代码进行测试：

测试输入： 5 ， 4 ， 3 ；

预期输出：

```
1. 5
2. Tensor("basic_rnn_cell/Tanh:0", shape=(4, 5), dtype=float32)
```

测试输入： 151 ， 12 ， 122 ；

预期输出：

```
1. 151
2. Tensor("basic_rnn_cell/Tanh:0", shape=(12, 151), dtype=float32)
```

2、了解 LSTM 的核心思想，根据教程指导创建 LSTM 网络的一个 LSTMCell 。

Long Short Term 网络，一般叫做 LSTM，是一种特殊的 RNN 类型，可以学习长期依赖信息。LSTM 通过刻意的设计来避免长期依赖问题。记住长期的信息在实践中是 LSTM 的默认行为。

对于 BasicLSTMCell 类，LSTM 网络单元的情况有些许不同，因为 LSTM 可以看做有两个隐含状态 h 层和 c 层，对应的隐含层就是一个 Tuple，每个都是 (batch_size, state_size) 的形状。下面介绍如何创建一个 LSTMCell：

```
1. #首先创建一个输入张量，batch_size,n_inputs 自行指定
2. inputs = tf.placeholder(np.float32, [batch_size,n_inputs])

1. #接下来正式创建一个包含n_units 个神经元的BasicLSTMCell 实例
2. lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units=n_units)

1. # 通过zero_state 得到一个全0 的初始状态
2. h0 = lstm_cell.zero_state(batch_size=batch_size,dtype=np.float32)

1. #调用一次__call__()方法，得到输出和新的隐含状态
2. output, h1 = lstm_cell.__call__(inputs, h0)

1. print(h1.h) #打印新的h 层
2.
3. print(h1.c) #打印新的c 层
```

编程要求：

创建一个包含 a 个神经元，可以接受一个 $\text{shape}=[b, c]$ ，类型为 float32 的张量作为输入的 `BasicLSTMCell`，并打印一次输入后该 `BasicRNNCell` 的隐含状态的 h ， c 层。

```
1.      # -*- coding: utf-8 -*-
2.      import tensorflow as tf
3.
4.      # 参数 a 是 BasicRNNCell 所含的神经元数，参数 b 是 batch_size，参数 c 是单个
      input 的维数，shape = [ b , c ]
5.      def creatRNNCell(a,b,c):
6.          # 请在此添加代码 完成本关任务
7.          # ***** Begin *****#
8.
9.          # ***** End *****#
```

3、学习如何一次执行得到 `RNNCell` 多步调用 `call` 方法得到的结果，其次掌握构建堆叠 `RNNCell` 的方法。

基础的 `RNNCell` 有一个很明显的问题：对于单个的 `RNNCell`，我们使用它的 `call` 函数进行运算时，只是在序列时间上前进了一步。比如使用 x_1 、 h_0 得到 h_1 ，通过 x_2 、 h_1 得到 h_2 等。这样的话，如果我们的序列长度为 10，就要调用 10 次 `call` 函数，比较麻烦。对此，`TensorFlow` 提供了一个 `tf.nn.dynamic_rnn` 函数，使用该函数就相当于调用了 n 次 `call` 函数。即通过 $\{h_0, x_1, x_2, \dots, x_n\}$ 直接得 $\{h_1, h_2, \dots, h_n\}$ 。

具体来说，设我们输入数据的格式为 $(\text{batch_size}, \text{time_steps}, \text{input_size})$ ，其中 `time_steps` 表示序列本身的长度，如在 `Char RNN` 这一开源项目中，长度为 10 的句子对应的 `time_steps` 就等于 10。最后的 `input_size` 就表示输入数据单个序列单个时间维度上固有的长度。另外我们已经定义好了一个 `RNNCell`，调用该 `RNNCell` 的 `call` 函数 `time_steps` 次，对应的代码就是：

```
1.      # inputs: shape = (batch_size, time_steps, input_size)
2.
3.      # cell: RNNCell
4.
```

```

5.     # initial_state: shape = (batch_size, cell.state_size)。初始状态。一般可以取零矩阵
6.
7.     outputs, state = tf.nn.dynamic_rnn(cell, inputs,
initial_state=initial_state)

```

此时，得到的 `outputs` 就是 `time_steps` 步里所有的输出。它的形状为 `[batch_size, time_steps, cell.state_size]`。`state` 是最后一步的隐状态，它的形状为 `(batch_size, cell.state_size)`。

学习如何堆叠 RNNCell: MultiRNNCell

很多时候，单层 RNN 的能力有限，我们需要多层的 RNN。将 `x` 输入第一层 RNN 的后得到隐层状态 `h`，这个隐层状态就相当于第二层 RNN 的输入，第二层 RNN 的隐层状态又相当于第三层 RNN 的输入，以此类推。在 TensorFlow 中，可以使用 `tf.nn.rnn_cell.MultiRNNCell` 函数对 RNNCell 进行堆叠，相应的示例程序如下：

```

1.     import tensorflow as tf
2.
3.     import numpy as np
4.
5.
6.
7.     # 每调用一次这个函数就返回一个 BasicRNNCell
8.
9.     def get_a_cell():
10.         return tf.nn.rnn_cell.BasicRNNCell(num_units=128)
11.
12.     # 用 tf.nn.rnn_cell MultiRNNCell 创建 3 层 RNN
13.
14.     cell = tf.nn.rnn_cell.MultiRNNCell([get_a_cell() for _ in range(3)]) # 3
层 RNN
15.
16.     # 得到的 cell 实际也是 RNNCell 的子类
17.
18.     # 它的 state_size 是(128, 128, 128)
19.
20.     # (128, 128, 128)并不是 128x128x128 的意思
21.
22.     # 而是表示共有 3 个隐层状态，每个隐层状态的大小为 128
23.

```

```

24. print(cell.state_size) # (128, 128, 128)
25.
26. # 使用对应的 call 函数
27.
28. inputs = tf.placeholder(np.float32, shape=(32, 100)) # 32 是 batch_size,
100 是 input_size
29.
30. h0 = cell.zero_state(32, np.float32) # 通过 zero_state 得到一个全 0 的初始状态
31.
32. output, h1 = cell.call(inputs, h0)
33.
34. print(h1) # 新的隐层状态中含有 3 个 32x128 的向量

```

通过 MultiRNNCell 得到的 cell 并不是什么新鲜事物，它实际也是 RNNCell 的子类，因此也有 call 方法、state_size 和 output_size 属性。同样可以通过 tf.nn.dynamic_rnn 来一次运行多步。

编程要求

根据前面所学知识，在 begin-end 间尝试构建一个 a 层的 RNN，每一层由 state_size=b 的 RNNBasicCell 组成，初始化隐层状态为全零，输入一个 batch_size=c,time_steps=d,input_size=e 的输入序列，调用 tf.nn.dynamic_rnn 一次得到多步输入后的结果，并打印出最终的输出 output 的函数 MultiRNNCell_dynamic_call(a,b,c,d,e)。

```

1. #-*- coding: utf-8 -*-
2. import tensorflow as tf
3. import numpy as np
4.
5. # 参数 a 是 RNN 的层数，参数 b 是每个 BasicRNNCell 包含的神经元数即 state_size
6. # 参数 c 是输入序列的批量大小即 batch_size，参数 d 是时间序列的步长即 time_steps，参数 e 是单个输入 input 的维数即 input_size
7. def MultiRNNCell_dynamic_call(a,b,c,d,e):
8. # 用 tf.nn.rnn_cell MultiRNNCell 创建 a 层 RNN,并调用 tf.nn.dynamic_rnn
9. # 请在此添加代码 完成本关任务
10. # ***** Begin *****#
11.
12. # ***** End *****#

```

四、实验报告要求

参见实验报告模板