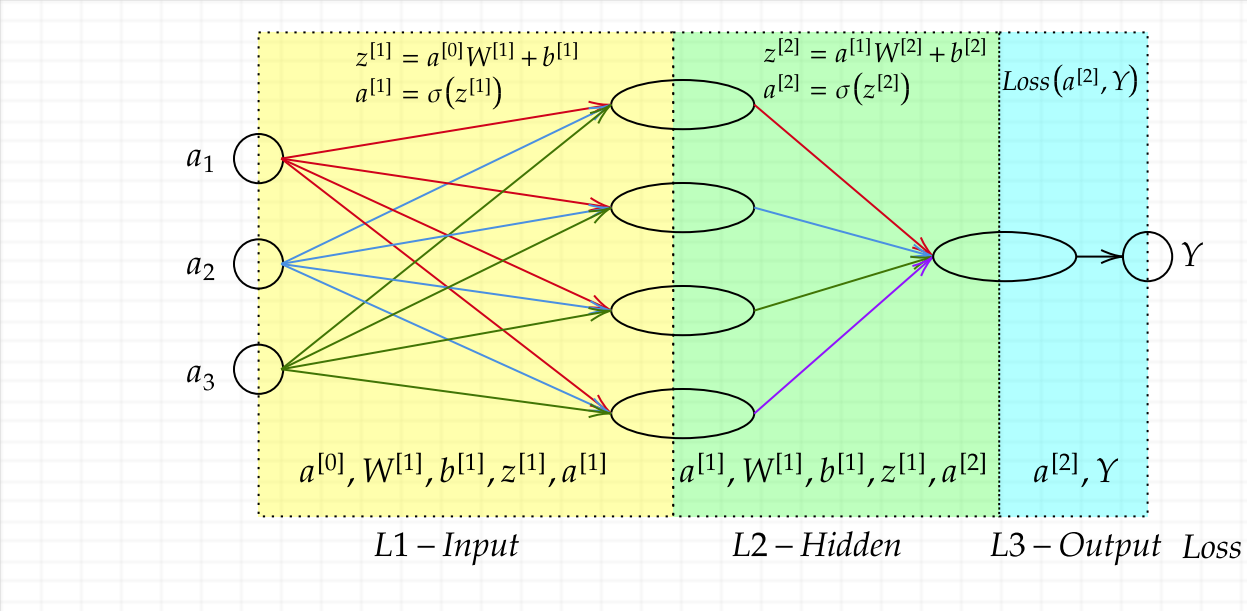
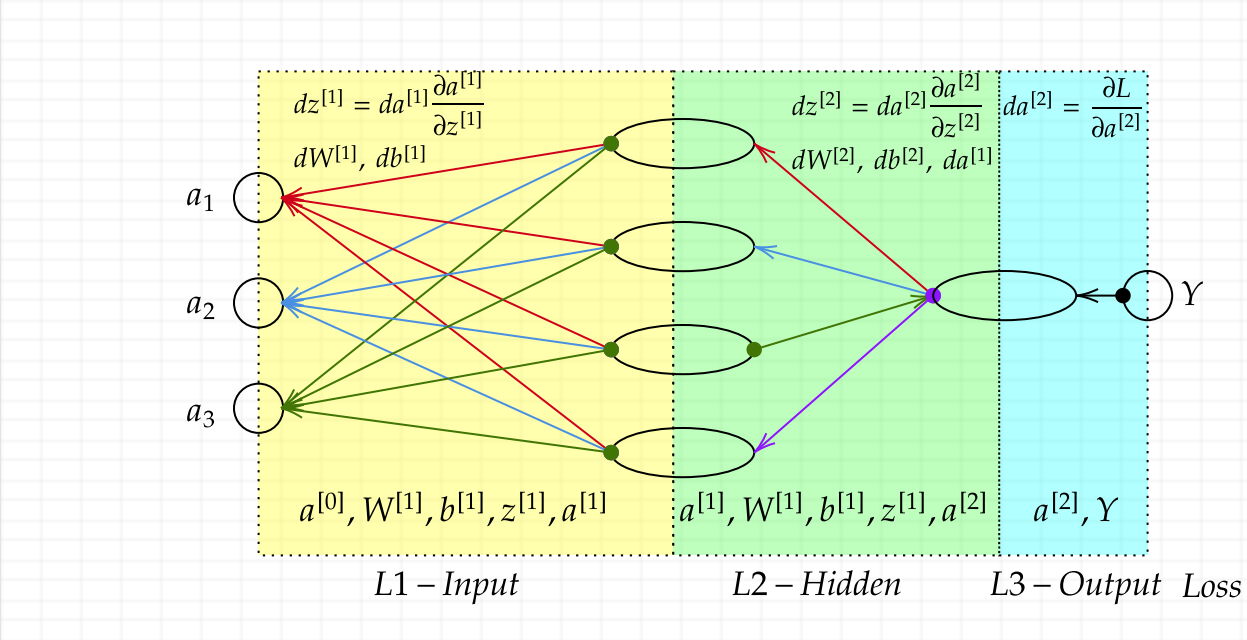


如上图所示，除去输出层比较特殊之外，普通的输入、隐藏层L_i 都会包含：输入数据$a^{[i-1]}$，权重$W^{[i]}$，常数项$b^{[i]}$，$z^{[i]} = a^{[i-1]}W^{[i]} + b^{[i]}$，输出到下一层的数据$a^{[i]} = \sigma(z^{[i]})$。而对于反向过程则有：



从上图我们看到，对于每一层L_i 来说，还会接收后面层L_{i+1} 传过来的梯度，并经过计算后传递给前面的层。于是我们可以提取每一层都包含公共的数据结构。

代码结构

```python
import numpy as np
import activation

''' Layer 数据结构
属性:
self.learning_rate  # 本层的学习率
self.a  # 前一层的输出作为本层的输入
self.weights  # 本层权重
self.b  # 本层常数项
self.activation  # 本层的激活函数，能够同时计算出本层的梯度

方法:
fp(a): 本层的正向过程，接收前一层的输出，计算后输出结果到下一层
bp(dNext): 本层的反向过程，接收后一层对本层输出a 的偏导，同时计算并输出本层对上一层输出的偏导数。并调节权重与常数项
randomW_B(row, col): 为本层生成随机的权重和常数矩阵
'''

class Layer(object):
  def __init__(self,
    weights: np.array,  # 权重矩阵
    b: np.array,  # 常数项
    learning_rate=0.01,
    act: activation.Activation = activation.ReLU(),  # 激活函数
  ) -> None:
    self.weights = weights
    self.b = b
    self.learning_rate = learning_rate
    self.activation = act

  def fp(self, a: np.array) -> np.array:
    self.a = a
    z = np.dot(a, self.weights) + self.b
    self.z = z
    return self.activation(self.z)

  def bp(self, dNext: np.array) -> np.array:  # 传入下一层的偏导
```

```
    dz = dNext*self.activation.gradient(self.z)
    dw = np.dot(self.a.T, dz)/len(self.a)
    self.weights -= self.learning_rate*dw
    db = np.mean(dz, axis=0)
    self.b -= self.learning_rate*db
    dPrev = np.dot(dz, self.weights.T)
    return dPrev

  def randomW_B(row, col):
    weight = 2 * np.random.random((row, col))-1
    b = 2 * np.random.random((1, col))-1
    return weight, b
```

调用流程

```
def main():
  np.random.seed(1)
  a0, y = loadData()

  # 初始化
  w0, b0 = layer.Layer.randomW_B(3, 4)
  w1, b1 = layer.Layer.randomW_B(4, 1)
  l0 = layer.Layer(
      w0,
      b0,
      learning_rate=0.3,
      act=activation.LeakyReLU()
  )
  l1 = layer.Layer(
      w1,
      b1,
      learning_rate=0.3,
      act=activation.Sigmoid()
  )

  # 真正的调用过程
  for i in range(10000):
    # 计算正向过程
```

```
    a1 = l0.fp(a0)
    a2 = l1.fp(a1)
    loss = Loss(a2, y)

    # 计算反向过程，由后至前反推一遍
    g_a2 = loss.gradient()
    g_a1 = l1.bp(g_a2)
    l0.bp(g_a1)

# 验证
s0 = [
      [0, 0, 0],
      [0, 0, 1],
      [0, 1, 0],
      [0, 1, 1],
      [1, 0, 0],
      [1, 0, 1],
      [1, 1, 0],
      [1, 1, 1]
     ]
s1 = l0.fp(s0)
s2 = l1.fp(s1)
print(s2)
```

其实这里应该还可以进一步封装，将所有的层存入一个数组或者链表，通过遍历或者是递归来自动执行正向过程和反向过程。