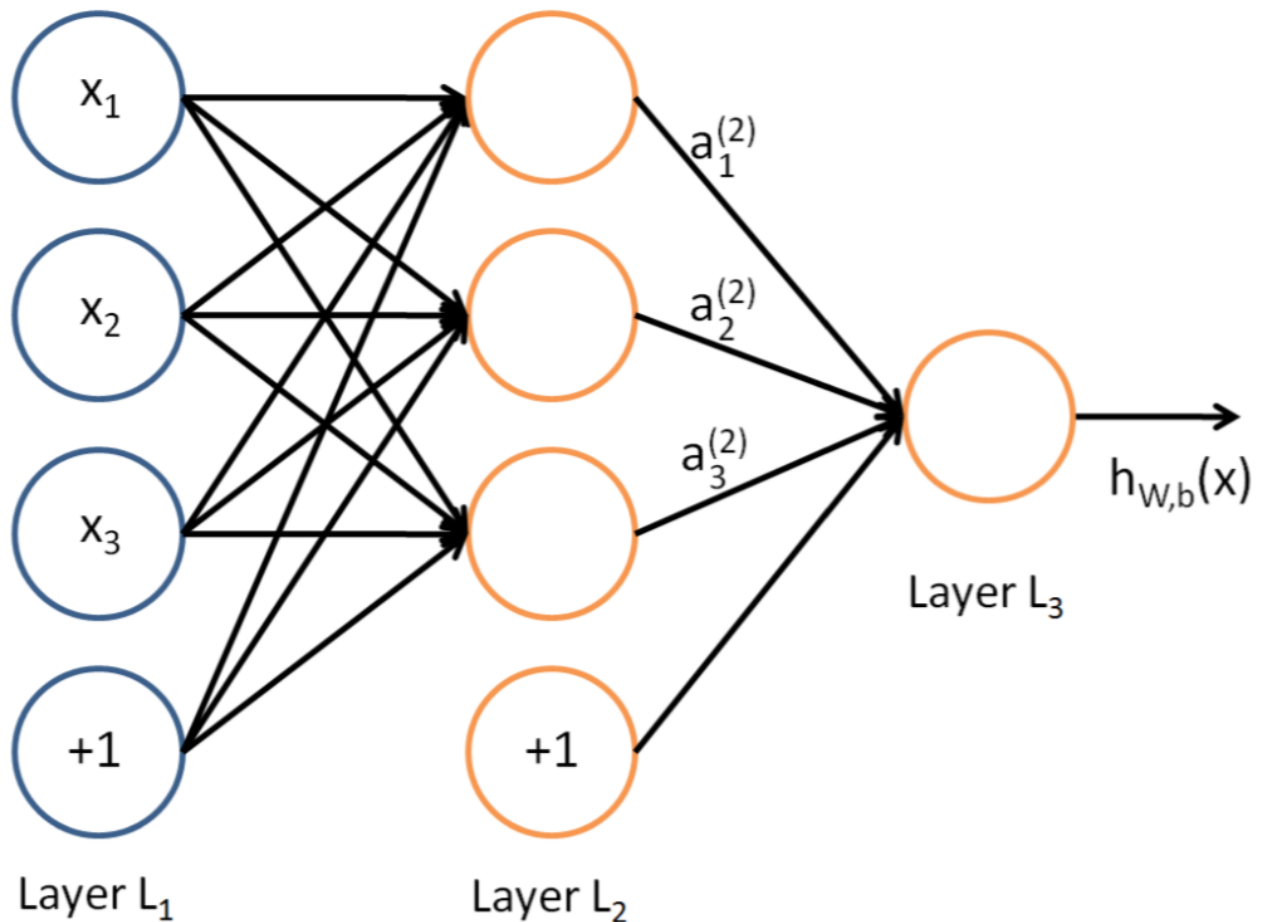


# 1.神经网络与BP神经网络

- 神经网络模型：



这是一个典型的三层神经网络模型，有输入层、隐含层和输出层，通常在隐含层引入一个非线性变换函数，例如sigmoid函数；通过该函数可以将模型变成一个非线性模型。

假设我们有一组数据 $\{X_1, X_2, \dots, X_n\}$ ，每一个 $X_i$ 代表一个 $N$ 维的数据，将数据输入神经网络模型后，得到一个结果 $y_j$ ； $y_j$ 可以是一个数(0或1)，也可以是一组向量。通常，这样得到结果往往是很难达到满意的结果，所以我们自然想到通过定义一个损失函数，通过损失函数来判断如何更新我们神经网络中各个链接的权重。

- BP(Backpropagation)

介绍神经网络模型之前需要再回顾共轭梯度法(Gradient Descent)。

给定一组参数： $\theta = \{w_1, w_2, \dots, b_1, b_2, \dots\}$ ，通过不断迭代更新参数：

$$\theta^0 \rightarrow \theta^1 \rightarrow \theta^2 \dots$$

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial w_1} \\ \frac{\partial L(\theta)}{\partial w_2} \\ \vdots \\ \frac{\partial L(\theta)}{\partial b_1} \\ \frac{\partial L(\theta)}{\partial b_2} \\ \vdots \end{bmatrix}$$

计算好每个参数的偏导数后，可以更新神经网络中的参数：

$$\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$$

$$\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$$

参考以下的神经网络模型，



令 $C$ 为损失函数，根据求导的链式法则：

$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w} \frac{\partial C}{\partial z}$$

其中， $\frac{\partial z}{\partial w}$ 是用来计算模型的前向传播， $\frac{\partial C}{\partial z}$ 用来计算模型的后向传播。观察以上模型可以发现：

$$\frac{\partial C}{\partial z} = \frac{\partial a}{\partial z} \frac{\partial C}{\partial a} = \sigma'(z) \frac{\partial C}{\partial a}$$

令 $w_3 = \frac{\partial z'}{\partial a}$ ， $w_4 = \frac{\partial z''}{\partial a}$ ，可以得到：

$$\frac{\partial C}{\partial z} = \sigma'(z) \left( w_3 \frac{\partial C}{\partial z'} + w_4 \frac{\partial C}{\partial z''} \right)$$

根据以上计算实例，可以明白BP神经网络的反向传播原理。

## 2. CNN(Convolutional Neural Network)

- 为什么使用CNN模型

CNN模型通常用来处理图像分类问题，那么对于一个多分类的问题，在原理是我们可以使用

传统的机器学习去完成的，比如多层感知机模型或者k-NN模型等。之所以使用CNN模型，是因为相比较于传统的机器学习模型，CNN模型不需要我们做特征工程。传统的机器学习模型需要对处理的对象做特征工程，当数据的维度非常大时，特征工程的难度是特性高的。所以，使用CNN模型，可以直接将数据输入进去，模型可以自己去学习判断哪些是好的特征。

- CNN模型的流程



这是一个CNN模型基本的计算流程。首先，输入一张图片，模型对图片做卷积运算(Convolution)，然后再进行池化操作(Max Pooling)，将这两步重复多次后，可以得到这张图片的特征向量并输入到搭建好的神经网络模型中。

- . 卷积运算(Convolution)

在卷积运算中，我们假设 $I$ 是一张二维图像，同时我们有一个二维的核 $K$ ，得到如下的映射：

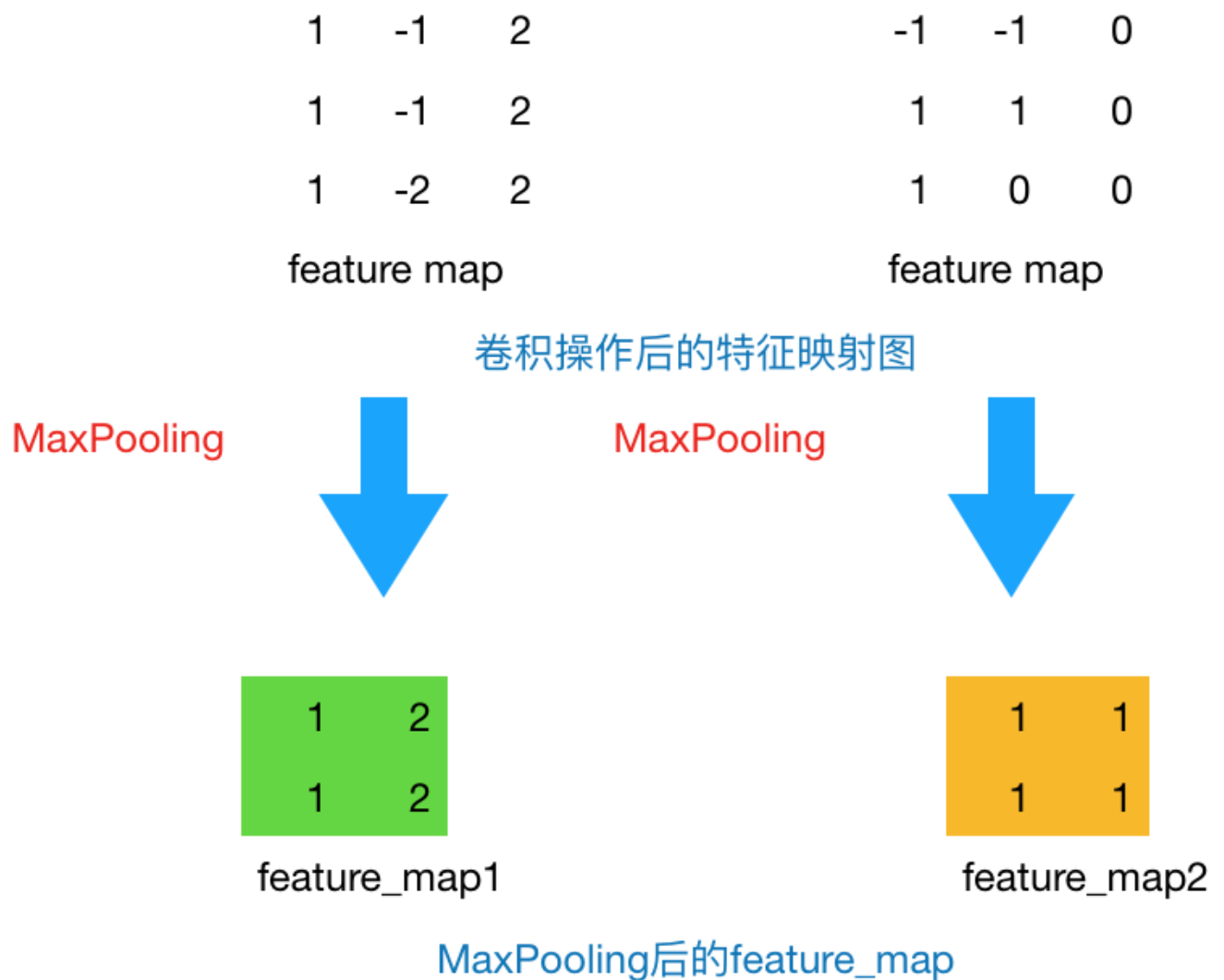
$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n)$$

举个例子，假设我们有一张4\*4的图片，以及卷积核1、卷积核2，通过以上定义的卷积运算，可以得到两张特征图(feature map)



- . 池化层(Max Pooling)

池化层的主要目的是通过降采样的方式，在不影响图像质量的情况下，压缩图片，减少参数。例如：

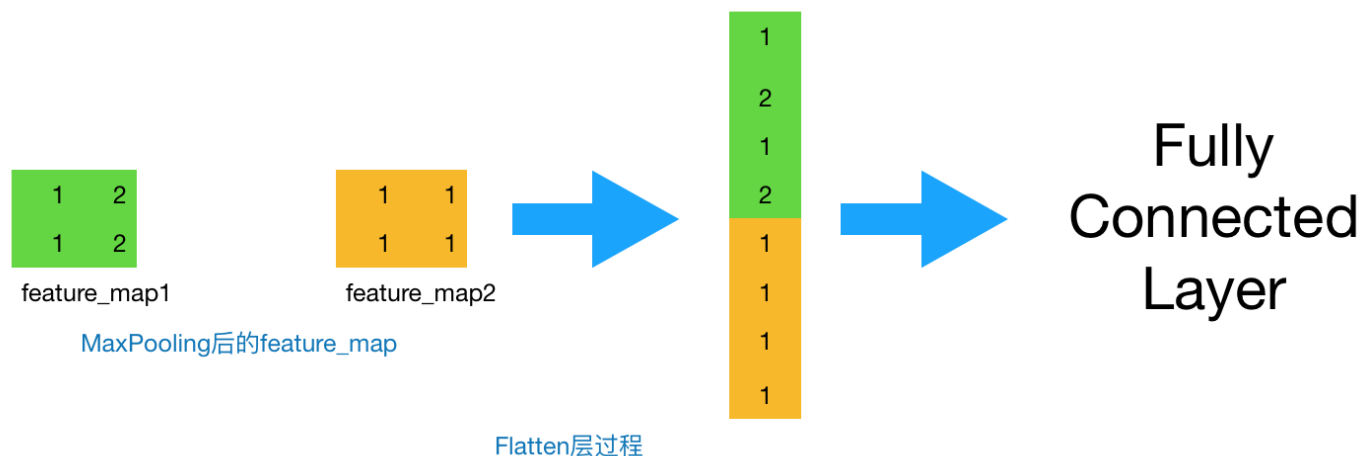


池化层的意义就是使用某一位置相邻的总体统计特征来代替网络在该位置的输出，或者说我们在提取特征时，只关系特征本身，而不是特征出现的位置。常用的池化函数有：相邻矩形区域平均值， $L^2$ 范数以及基于距中心像素距离的加权平均。

. Zero Padding和Flatten

因为图片在经过卷积运算和池化层后，会变得越来越大，所以通过补零(Zero Padding)，可以使得输出的图像和之前相比大小不变。

为了将得到的特征图输入到搭建好的神经网络模型中，需要将特征图平铺开(Flatten)，操作如下：



### 3.RNN(Recurrent Neural Network)

和能够有效处理空间数据的CNN模型不同，RNN是一种能够有效处理时序信息的模型，经常应用于语言模型、文本分析、机器翻译等。

- 语言模型

假设文本 $T$ 中的词汇以此看作 $w_1, w_2, \dots, w_T$ ，该语言模型序列的概率：

$$P(w_1, w_2, \dots, w_T) = \prod_{t=1}^T P(w_t | w_1, \dots, w_{t-1})$$

例如，一个含有4个词的文本序列的概率：

$$P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_4|w_1, w_2, w_3)$$

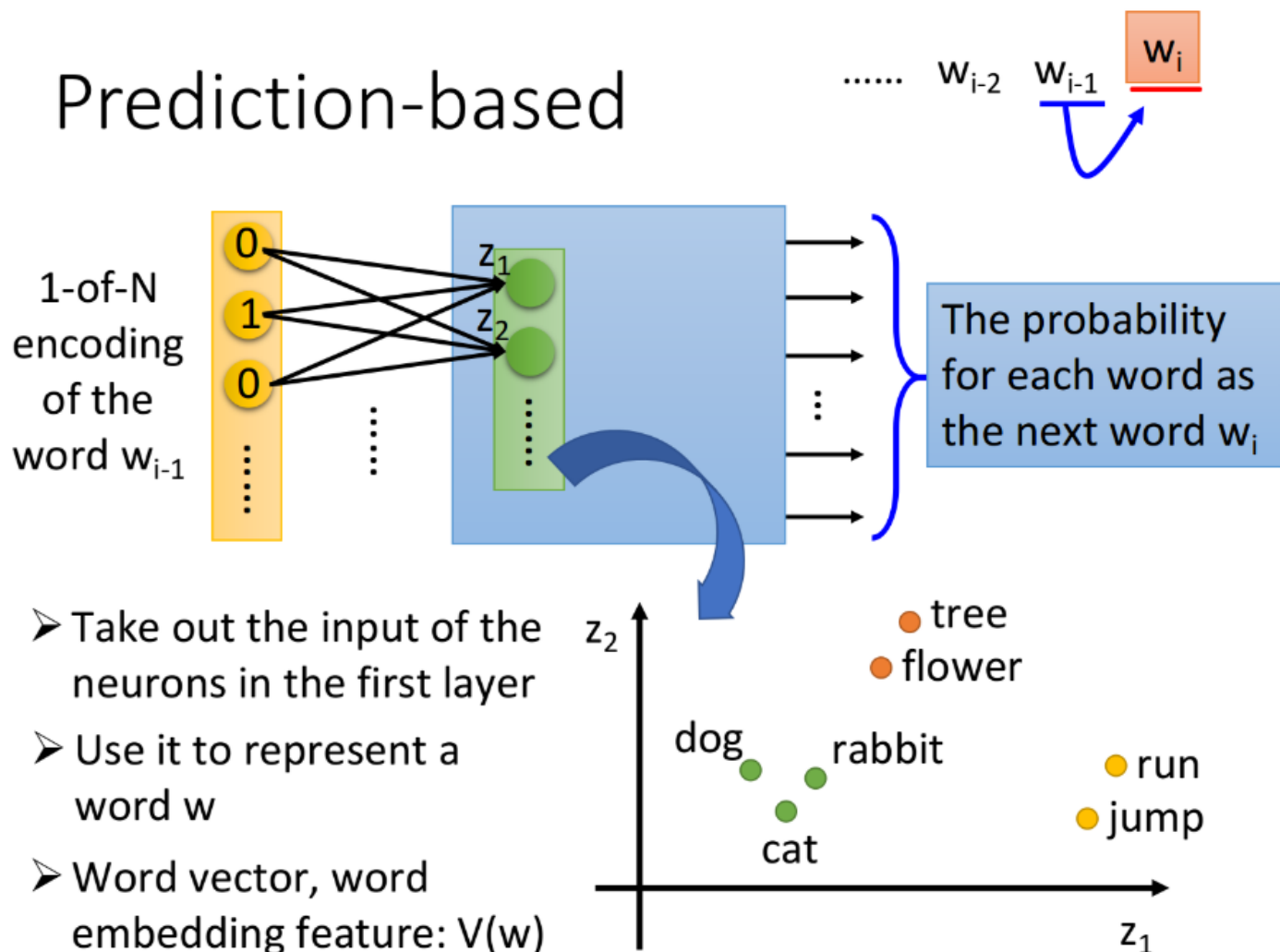
- Word Embedding

Word Embedding就是通过一种方法，让机器取阅读大量的文章，可以自动将一些语义相近的词汇映射到相近的高维空间去。

首先对词语进行1-of-N Encoding的方法进行描述，对每个词汇建立一个 $vector$ ，为了和一般的方法区分开来(1-of-N Encoding, clustering)，可以设计一个神经网络模型，它要做的就是根据当前的word $w_{i-1}$ ，来预测下一个可能出现的word $w_i$ 是什么。假设我们使用1-of-N encoding把 $w_{i-1}$ 表示成feature vector，它作为neural network的input，output的维数和input相等，只不过每一维都是小数，代表在1-of-N Encoding中该维为1其余维为0所对应的word会是下一个word $w_i$ 的概率。

假如将第一个隐藏层的输入 $z_1, z_2, \dots, z_m$ 拿出来，这个向量可以作为对应单词的另一种表示形式，将这个向量作在图上，可以发现相近的词向量具有相近的词性。

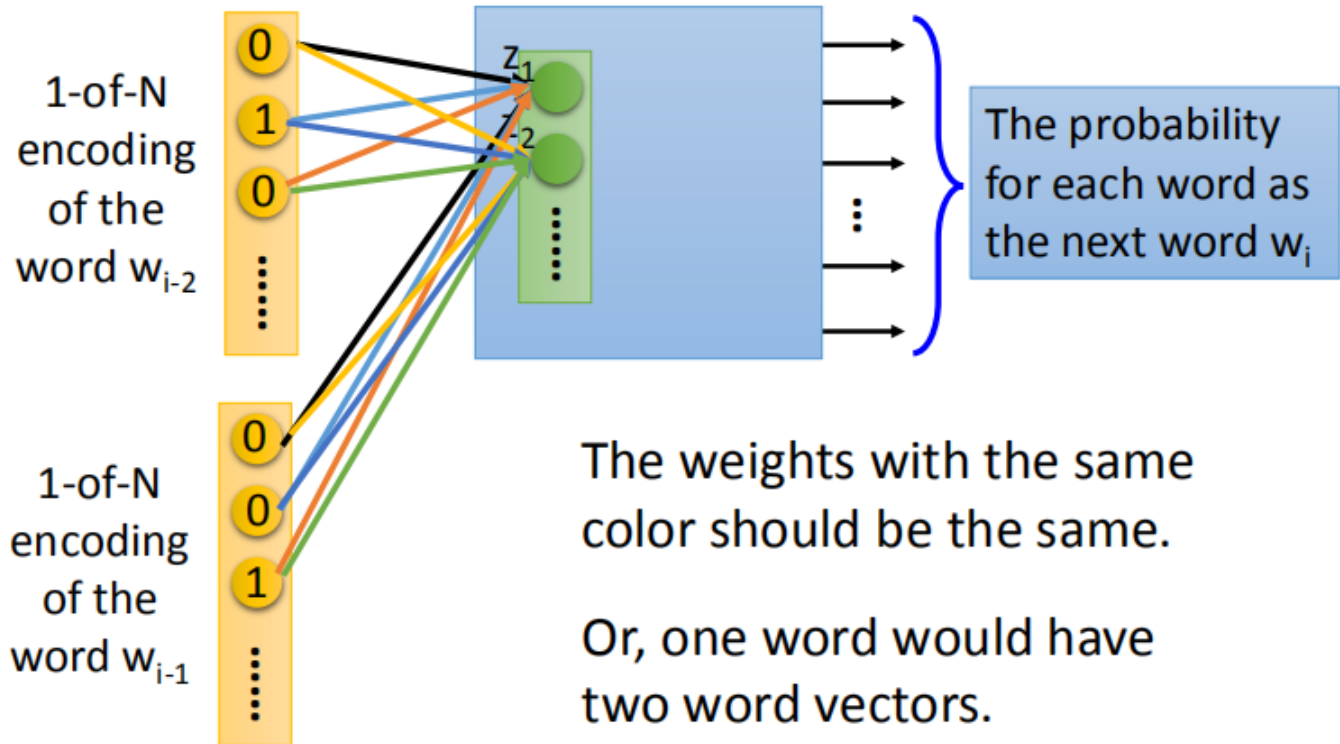
# Prediction-based



- 共享权重

因为仅仅只靠当前词汇去预测下一个词汇，那么模型的预测能力会很弱，我们可以通过共享权重的方法，通过多个词汇去预测下一个词汇。

# Prediction-based – Sharing Parameters



13

在具体操作时，可以通过如下运算，使得两个不同的词共享权重：

$$w_i = w_i - \eta \frac{\partial C}{\partial w_i} - \eta \frac{\partial C}{\partial w_i}$$

$$w_j = w_j - \eta \frac{\partial C}{\partial w_i} - \eta \frac{\partial C}{\partial w_i}$$

此时就可以保证 $w_i$ 和 $w_j$ 相等了。

## 4.LSTM(Long Short-Term Memory)

LSTM是一长短期神经网络算法，它具有记忆和遗忘功能，比如凭借临时内存，记忆一句话中较长一段时间前出现的词汇，并通过此预测下一个词出现的概率。下面给出一个典型的计算单元：





假设此时我们读取到的词为 $X_i$ ，这个词对应的 $vector$ 是 $Z_i$ ，将这个词向量分别从 $z, z_i, z_0, z_f$ 输入，并且假设其中使用的激活函数都是 $sigmoid$ 激活函数。 $z_f$ 控制模型是否将该模型记忆到内存中， $z_0$ 决定是否将计算的结果输出，作为下一个模型的输入。

## 5.实验(CNN)

- 实验数据

实验数据是来源于(<https://www.kaggle.com/c/digit-recognizer/data>)的从零到九的手绘数字的灰度图像。每个图像的高度为28像素，宽度为28像素，总计784像素。每个像素都有一个与之关联的像素值，表示该像素的亮度或暗度，数字越高表示像素越暗。此像素值是0到255之间的整数(含)训练数据集 (train.csv) 有785列。第一列称为“标签”，是用户绘制的数字。其余列包含关联图像的像素值。

- 模型搭建

首先读取数据，因为题目中明确说明没有数据的缺失，所以不再做任何预处理。读取数据后将数据的20%留作验证集，80%作为训练集。

```
data = pd.read_csv('../input/digit-recognizer/train.csv')
test = pd.read_csv('../input/digit-recognizer/test.csv')
train, valid = train_test_split(data, stratify=data.label, test_size=0.2)
```

导入好相应模块后，参考李宏毅老师搭建的神经网络模型，首先对原图像做卷积运算，并对每一批数据进行标准化处理，每次处理后都进行池化，最后得到神经网络的输入层有512个特征。



```

nn.Conv2d(1, 64, 3, 1, 1), # [32, 28, 28]
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0), # [64, 14, 14]

    nn.Conv2d(64, 128, 3, 1, 1), # [128, 14, 14]
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0), # [128, 7, 7]

    nn.Conv2d(128, 256, 3, 1, 1), # [256, 3, 3]
    nn.BatchNorm2d(256),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0), # [256, 1, 1]

    nn.Conv2d(256, 512, 3, 1, 1), # [512, 1, 1]
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0), # [512, 8, 8]

```

为了防止模型过拟合，再将特征输入全连接层时，采用 $p = 0.5$ 的*dropout*，采用 $LeRu()$ 作为模型的激活函数。

```

nn.Dropout(p = 0.5),
nn.Linear(512, 512),
nn.BatchNorm1d(512),
nn.ReLU(inplace=True),

nn.Dropout(p = 0.5),
nn.Linear(512, 512),
nn.BatchNorm1d(512),
nn.ReLU(inplace=True),

nn.Dropout(p = 0.5),
nn.Linear(512, 10),

```

接下来对模型进行训练，模型的学习率采用 $Adam$ ，损失函数定义为交叉熵损失函数。事实上，对于分类问题，一般习惯将交叉熵作为损失函数。

```

cnn = CNN().to(device)
# 训练
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
loss_func = nn.CrossEntropyLoss()

for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_dataloader):
        b_x = x.to(device)
        b_y = y.to(device)
        output = cnn(b_x)
        print
        loss = loss_func(output, b_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if step % 200 == 0:
        print('epoch:{}/{}, loss:{:.4f}'.format(epoch, EPOCH, loss))

cnn.eval()    #将模型的参数固定
with torch.no_grad():
    total = 0
    cor = 0
    for x, y in valid_dataloader:
        x = x.to(device)
        y = y.to(device)
        out = cnn(x)
        pred = torch.max(out, 1)[1]
        total += len(y)
        cor += (y == pred).sum().item()
    print('acc:{:.4f}'.format(cor/total))

```

将训练好的数据再测试集上进行测试，保存结果为csv文件并提交，得分的结果和损失曲线如下：

avatar

avatar

观察可以发现，虽然模型得到了较好的结果，模型计算的损失值在开始也很快地下降，但是在后面的迭代中，*loss*值一直有较大的波动，所以考虑减少卷积、池化的次数，观察是否可以改善模型。

重新搭建如下的神经网络模型：

```
nn.Conv2d(1,32,3,1,1),# (32, 28, 28)
nn.BatchNorm2d(32),
nn.ReLU(inplace=True),


nn.Conv2d(32, 32, 3, 1, 1), # (32, 28, 28)
nn.BatchNorm2d(32),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=2), # 池化 (32, 14, 14)

nn.Conv2d(32, 64, 3, 1, 1),# (64, 14, 14)
nn.BatchNorm2d(64),
nn.ReLU(inplace=True),

nn.Conv2d(64, 64, 3, 1, 1),# (64, 14, 14)
nn.BatchNorm2d(64),
nn.ReLU(inplace=True),
nn.MaxPool2d(2),# (64, 7, 7)
```

得到的实验结果如下：

Alt text

Alt text

可以发现得分较之前的模型，有了很大的提升，所以神经网络的模型并非越深越好，应该根据具体的问题来搭建合适的模型。