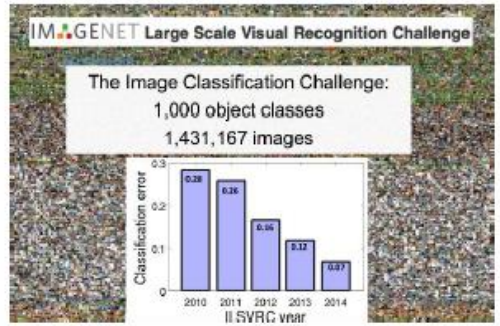
神经网络

2018 10 16 by xzn

视频中提到了深度学习可以做例如分类之类的任务，给 140 万张图片然后分到 1000 个类别中，如图 1.1（a）所示；在图像上也有应用，例如可以以图找图，对手机中的相似照片进行归类，如图 1.1（b）所示；还可以对图像中的物品进行识别，如图 1.1（c）所示；更有甚者还可以“模仿”某些画家的风格然后将某些图片“改成”该画家的风格，如图 1.1（d）所示；或者还可以“模仿”作家的写作风格进行文章创作，如图 1.1（e）所示。



（a）分类任务

（b）以图找图、对相似照片进行归类

（c）图像中物品识别

（d）“模仿”画家风格

每个人，闭上眼睛的时候，才能真正面对光明

他们在吱呀作响的船舷上，静静看着世界，没有痛苦的声音，碎裂的海洋里摇晃出阵阵沉默，吞噬过来。他们的躯体，一点，一点，逐渐暗淡在

你们虔诚的看看远方，视角起头，不经意间，目光划过你们的面庞，上面淡淡的循强印，那么坚强

尘世凡间

计算机中的神经网络是模仿人的生理结构神经网络所提出的，接下来我们将从神经网络的结构，功能具体实现以及一个具体事例来进行学习。

目前神经网络主要包括输入层，隐藏层和输出层，其中输入层和输出层是一定有的，一个神经网络可以没有隐藏层，而绝大多数复杂的功能都是在隐藏层中实现的，隐藏层的深度越深，神经网络的能力（capacity）就越大。如图 1.2 所示的神经网络，红色部分是输入层，紧接着是两个隐藏层，最后接了一个绿色的输出层。其中如果只有少量的隐层那么我们称之为浅层神经网络，增加更多的隐藏层后我们称之为深度神经网络。

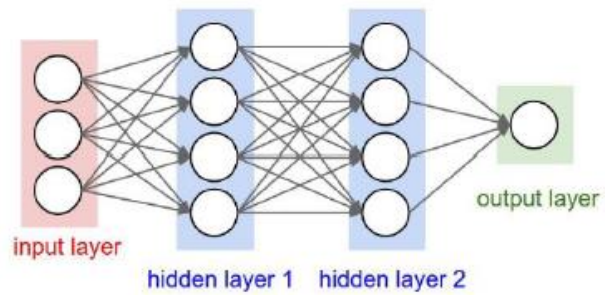


图 1.2 神经网络简单结构示意图

我们可以想到，对于每一个神经元，其与其他神经元的连接方式都是不同的，其余神经元对该神经元做出的响应也是不同的，例如，假设某一个神经元发出一个危险信号给周围的神经元，或许只有一个神经元会做出反应接着传递这个危险信号，而其他神经元可能会选择忽略甚至只对此信号做出较少反应；再举个例子，不同神经元之间的连接长度不同，相同的刺激经过不同的路径到达目的神经元之后的强度也会不同。上述事例说明了每个神经元与其相邻的神经元之间都是有不同的权重的，对于神经元之间的传递我们可以用如图 1.3 来进行示意

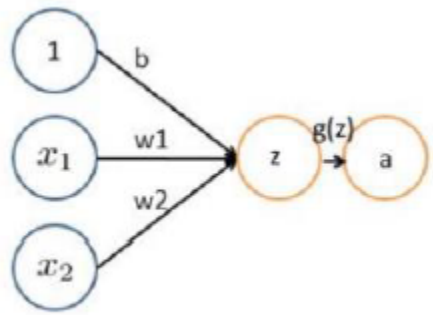


图 1.3 神经元间连接方式

可以看大对于每一个神经元 x，其与 z 都会有一个权重，对于 z 来说，最终会得到的输入为

$$z = w_1x_1 + w_2x_2 + b$$

常用的激活函数本节课我们只介绍两种，一种是 sigmoid 函数，一种是 sigmoid 的变形双 S 函数，如图 1.4 所示

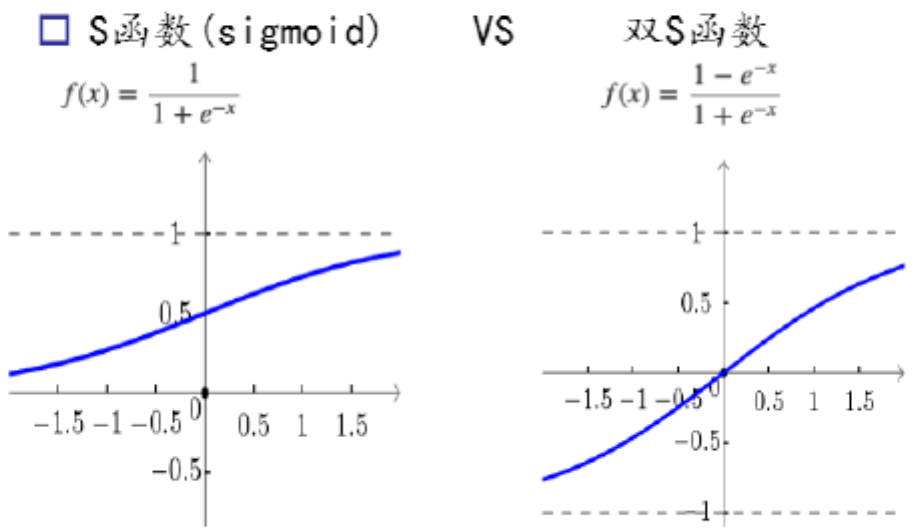


图 1.4 两种常见的激活函数

现在我们考虑一个分类问题，假设一个平面上存在可被一条线分隔开来的两个类别的点，如图 1.5 所示，根据 PLA 学习算法我们可以知道，机器通过学习是一定可以找到一条直线将这个线性可分集分成两部分

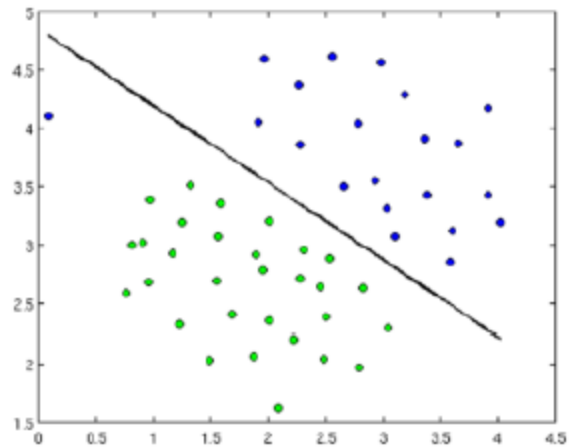


图 1.5 一个线性可分集

换句话说我们只需要用没有隐藏层的神经网络即可得到这样一条直线，假设直线的解析式为

$$z = k_1x_1 + k_2x_2 + b$$

那么我们构造一个包括一个输入和一个输出的神经网络即可，如图 1.6 所示

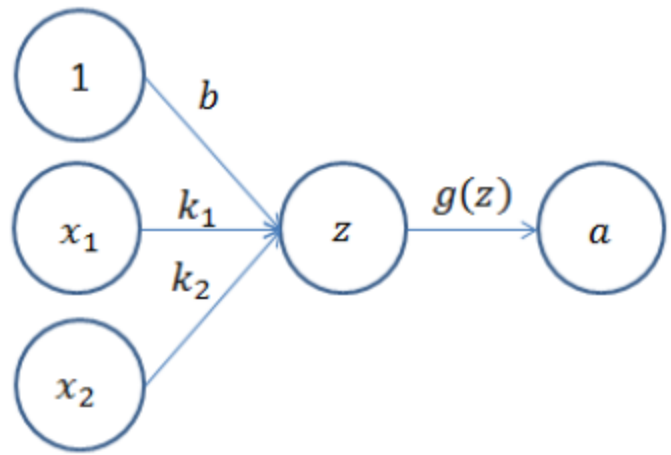
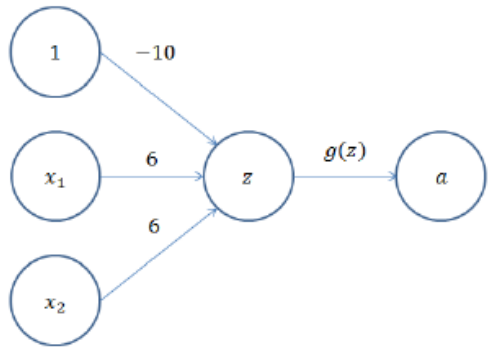


图 1.6 包含一个输入和一个输出的神经网络

这个神经网络激活函数我们采取 sigmoid 函数，最后我们判断当结果>=0.5 判断为一类，结果<0.5 判断为另一类即可。如此我们就用最简单的神经网络模拟了一个感知机。

1.4 神经网络实现逻辑与和逻辑或功能

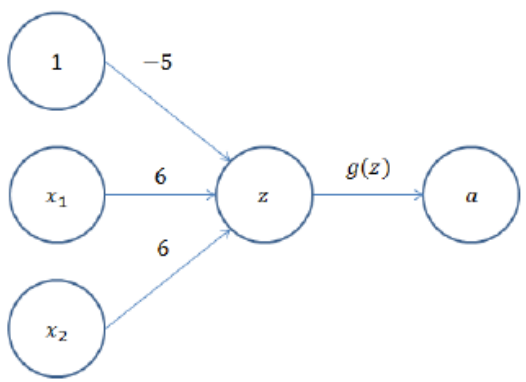
上一小节我们通过神经网络简单的实现了感知机的功能，这里我们要介绍如何用神经网络实现“逻辑与”和“逻辑或”，我们现在考虑如图 1.7 所示神经网络的输入与输出



x_1	x_2	a
0	0	0
0	1	0
1	0	0
1	1	1

图 1.7 神经网络实现“逻辑与”功能以及对应的输入输出

我们可以看到我们通过改变参数的值成功的实现了“逻辑与”的功能，相对应的“逻辑或”的功能也可以通过改变参数的值得到，如图 1.8 所示神经网络的输入与输出



x_1	x_2	a
0	0	0
0	1	1
1	0	1
1	1	1

图 1.8 神经网络实现“逻辑或”功能以及对应的输入输出

现在我们考虑线性不可分集，如图 1.9 所示的线性不可分集

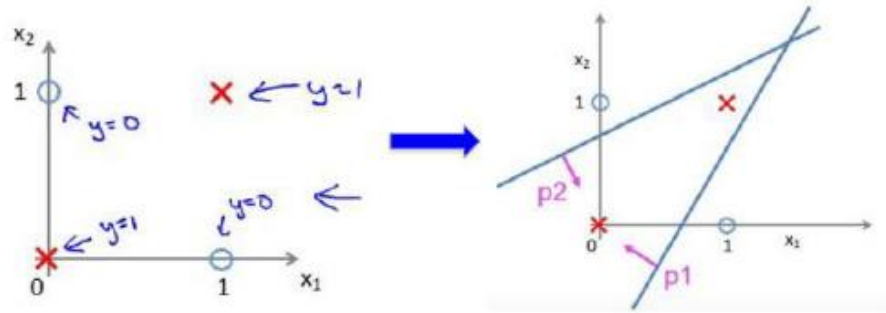


图 1.9 线性不可分集

假设我们定义落在直线一以下与直线二以上公共的区域的点记为一个类其余都当做另一个类，假设直线一与直线二有解析式

$$y_1 = k_1x_1 + k_2x_2 + b_1, y_2 = k_3x_1 + k_4x_2 + b_2$$

那么我们可以通过如下图 1.10 所示有一隐藏层的神经网络实现上述功能。

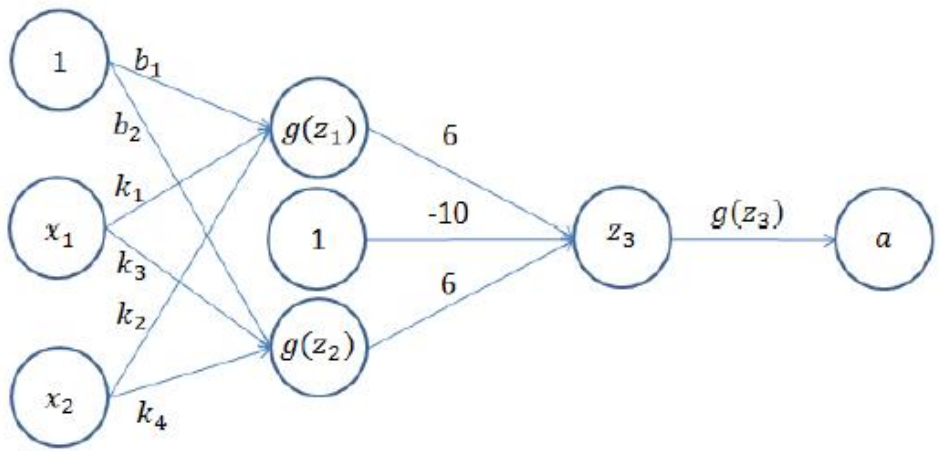


图 1.10 含有一层隐藏层的神经网络

图 1.11 一个较为复杂的分类问题

通过用非常多的直线模拟一个圆，继而围成一个圆域来进行分类问题。关于神经网络的隐藏层的层数我们还可以用图 1.12 来做一个理解

图 1.12 神经网络隐藏层的数量与其对应的能力

随着隐藏层的增加，我们可以很轻易的对上一层的结果进行一个组合，越深的层相当于在一个更高维度的一个角度对结果进行处理。

现在我们了解了通过增加隐藏层或者增加神经元的个数可以提高神经网络的能力，但是神经网络最关键的地方是其参数的取值，现在我们要考虑如何去更新神经网络的神经元的参数。这里我们使用 BP 算法（误差反向传播）。BP 算法大致思路如下，假设我们给定一组输入得到了一组输出，我们将这组输出与标准输出计算误差，然后使用随机梯度下降算法（SCD）想办法使得我们的输出与标准输出的误差最小，其中我们将误差函数对每一个需要更新的参数进行求导，然后按照一定的学习率来更新这个参数，经过一定次数的迭代或者当误差下降到一个可以接受的值时算法停止，这时候我们就得到了我们希望得到的参数。

现在我们来一步步详细解释，以三层的感知器为例，如图 1.14 所示

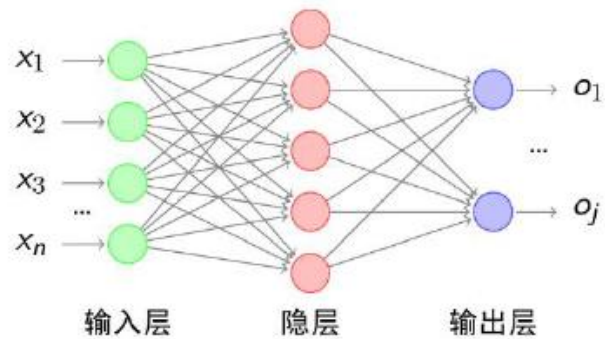


图 1.14 三层感知器

我们假设正确答案为 d，那么

输出层误差计算

$$\mathrm{E} = \frac{1}{2}(d-o)^2 = \frac{1}{2}\sum_{k=1}^{n}(d_k - o_k)^2$$

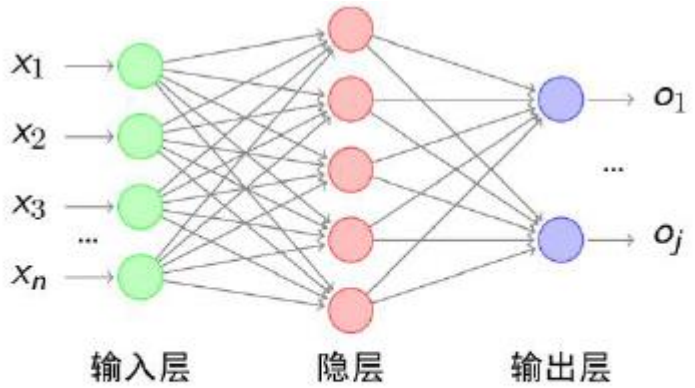


图 1.14 三层感知器

误差展开到隐藏层

$$\mathrm{E}=\frac{1}{2}\sum_{k=1}^{n}[d_k-f(net_k)]^2=\frac{1}{2}\sum_{k=1}^{n}[d_k-f(\sum_{j=0}^{m}\omega_{jk}y_j)]^2$$

误差展开到输入层

$$\mathrm{E}=\frac{1}{2}\sum_{k=1}^{n}[d_k-f(\sum_{j=0}^{m}\omega_{jk}f(net_j))]^2=\frac{1}{2}\sum_{k=1}^{n}[d_k-f(\sum_{j=0}^{m}\omega_{jk}f(\sum_{i=0}^{m}v_{ij}x_i)]^2$$

光看公式可能会晕，现在我们引入一个实例来进行计算

假设有如图 1.15 神经网络，对应的参数已在图中给出

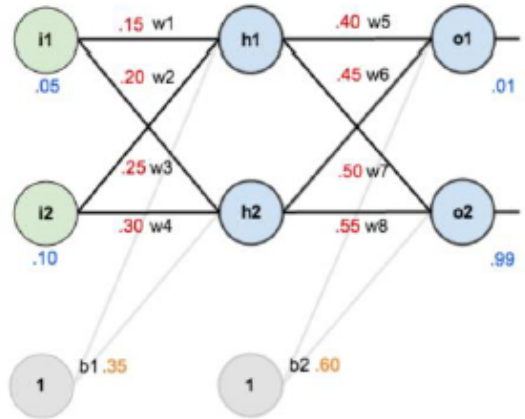


图 1.15 三层神经网络示意图

其中参数初始化为随机的，图中为了方便采取 0.05 为步长进行初始化

现在我们进行前向运算

$$net_{h1} = w_1 i_1 + w_2 i_2 + b_1 * 1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}} = \frac{1}{1 + e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$

$$net_{o1} = w_5 out_{h1} + w_6 out_{h2} + b_2 * 1 = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1$$

$$= 1.105905967$$

$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}} = \frac{1}{1 + e^{-1.105905967}} = 0.75136507$$

$$out_{o1} = 0.772928465$$

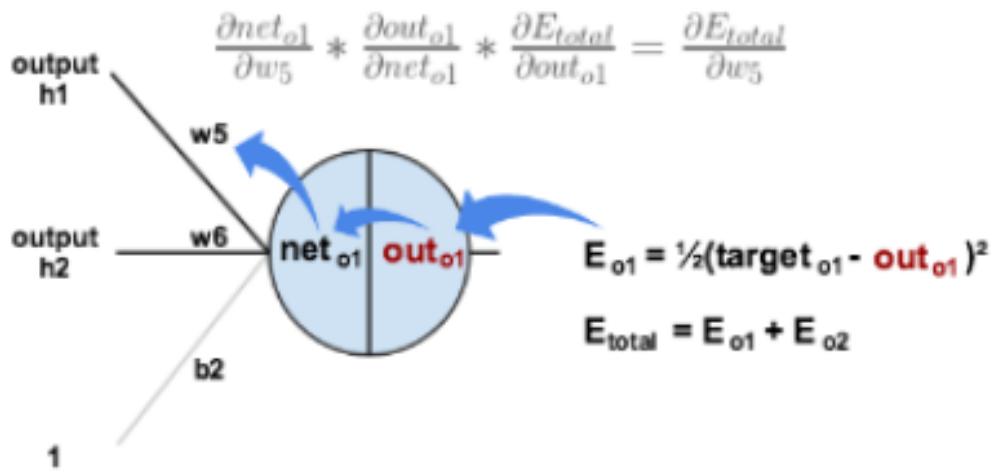
$$out_{o1} = 0.772928465$$

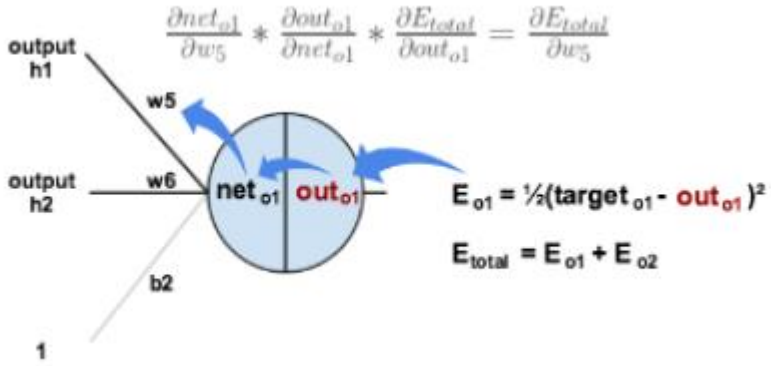
误差计算

$$E_{\text{total}} = \sum \frac{1}{2}(target - output)^2$$

$$E_{\text{total}} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

反向传播示意图见图 1.16 所示





$$E_{\text{total}} = \frac{1}{2}(target_{o1} - output_{o1})^2 + \frac{1}{2}(target_{o2} - output_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - output_{o1})^{2-1} * (-1) + 0 = output_{o1} - target_{o1}$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 0.75136507 - 0.01 = 0.74136507$$

$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

$$net_{o1} = w_5 out_{h1} + w_6 out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * {w_5}^{1-1} + 0 + 0 = out_{h1} = 0.593269992$$

$$\frac{\partial E_{total}}{\partial w_5} = (output_{o1} - target_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

$$= 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

$$\text{new } w_5 = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

$$\text{new } w_6 = 0.408666186$$

$$\text{new } w_7 = 0.511301270$$

$$\text{new } w_8 = 0.561370121$$

再往下的更新就不计算了越来越复杂。上式中需要注意η为学习率，即每次向最陡峭的位置走多大步。

卷积神经网络和我们上节课学的人工神经网络虽然都是神经网络，但是两者却不尽相同，一个较为完成的卷积神经网络可以见图 2.1 所示

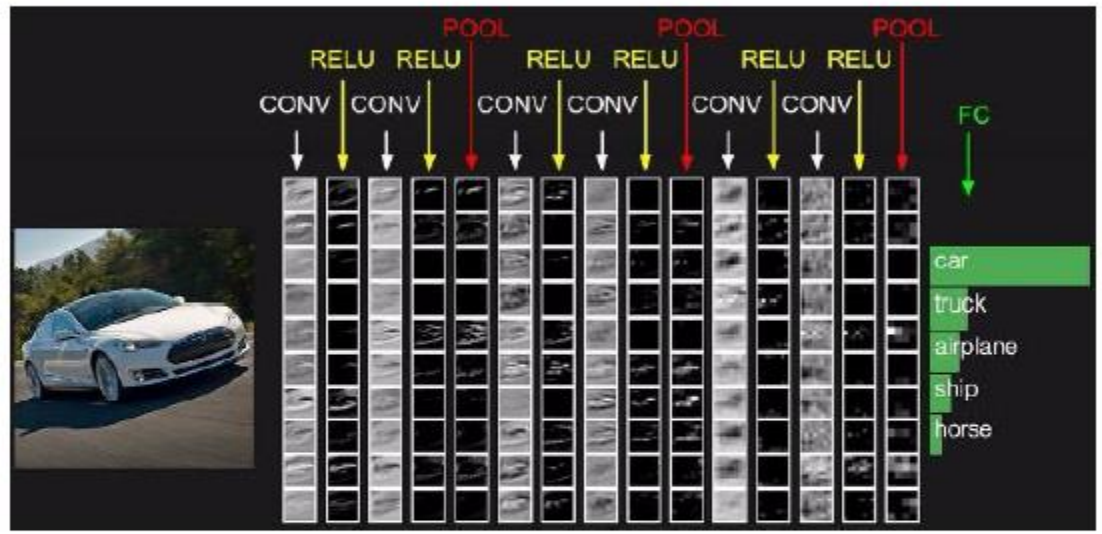


图 2.1 卷积神经网络结构示意图

我们可以看到一般一个完整的卷积神经网络主要包括输入层，输出层，CONV（卷积层），RELU（激活层），POOL（池化层）FC（全连接层）等，有些卷积神经网络还会具有 BN（Batch Normalization 层）。

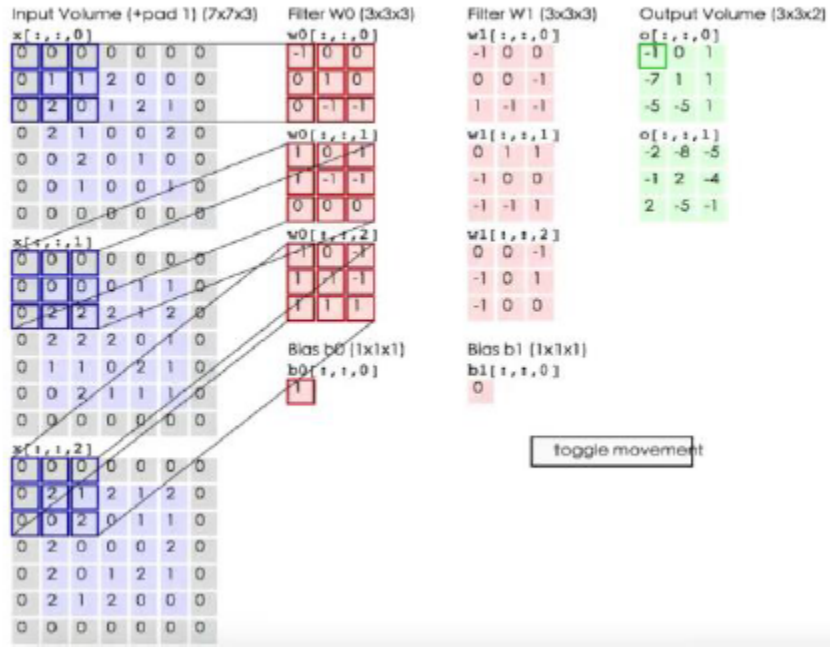


图 2.4 卷积神经网络计算过程 1[1]

由于图为三通道，故每个小朋友对于每一个通道都有一个自己的用于卷积计算的参数窗口，每一次将卷积窗口里面的值都与自己参数窗口里面的对应值相乘得到一个确定的数，然后三个通道的值再加上偏移值得到一个输出，例如上图中绿色矩阵第一个值的计算过程为

$$o = [0*(-1)+0*0+0*0+0*0+1*1+1*0+0*0+2*(-1)+0*(-1)]$$
$$+[0*1+0*0+0*(-1)+0*0+0*1+0*(-1)+0*0+2*0+2*0]$$
$$+[0*0+0*(-1)+0*(-1)+0*1+2*(-1)+1*(-1)+0*1+0*1$$
$$+2*1]+1=-1+0-1+1=-2+1=-1$$

接着我们移动这个窗口，由于规定了步长为 2，所以每次向左移动两个格子，那么此时如图 2.3 所示

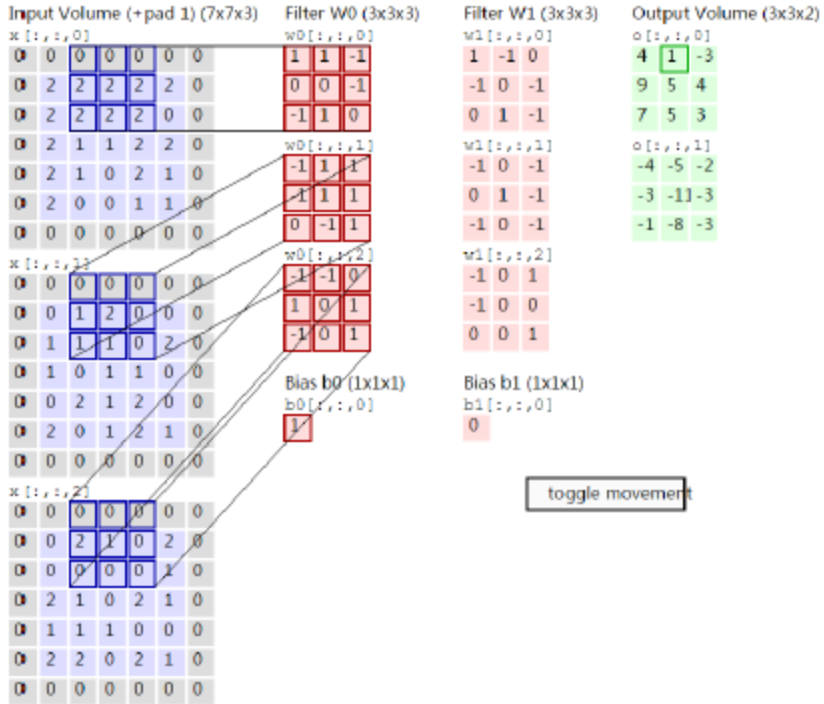


图 2.5 卷积神经网络计算过程 2

可以看到这时候所有窗口都向右便宜了两个格子，当窗口滑动到每一行最边缘之后，下一次窗口会先靠左，然后根据 depth 向下移动一定的格子，然后继续计算，例如当窗口滑动

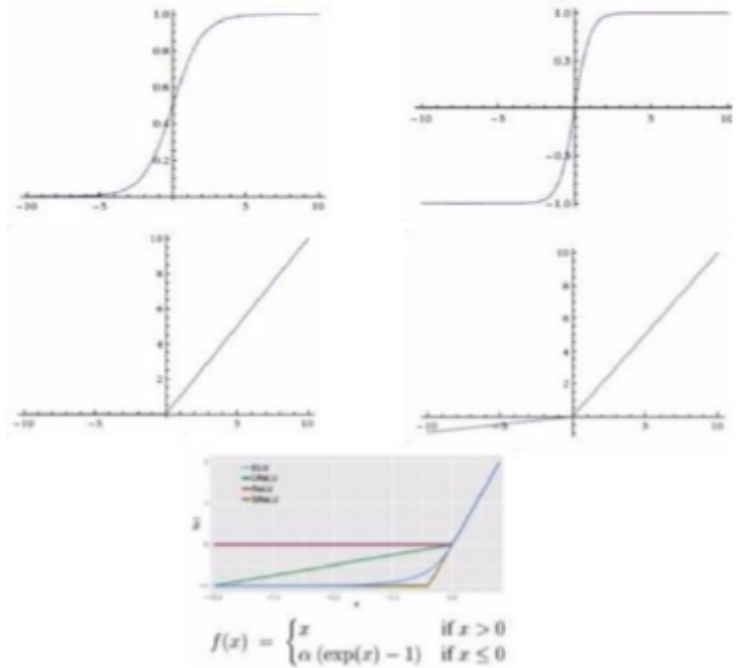


图 2.7 常见的五种激励函数

其中实际应用中 CNN 我们一般不用 sigmoid（显而易见，cnn 一般有较多的层数，如果使用 sigmoid 我们会发现，当有一层的输出比较大的时候，使用 sigmoid 作为激活函数会使得该处的倒数趋于 0，结合 sigmoid 图像也可以知道，这样由于链式法则，累乘的话会导致向后计算的时候在该层前面的层几乎都更新不动，也即到最后只有靠后面的全连接层一直在更新，前面的卷几层基本上都更新不了，为了避免这个问题，我们一般采用 Relu），对于 CNN 我们首先使用 RELU，如果效果太差，我们会使用 Leaky ReLu 或者 Maxout，某些情况下 Tanh 会有不错的结果，但是很少。（原因未知）

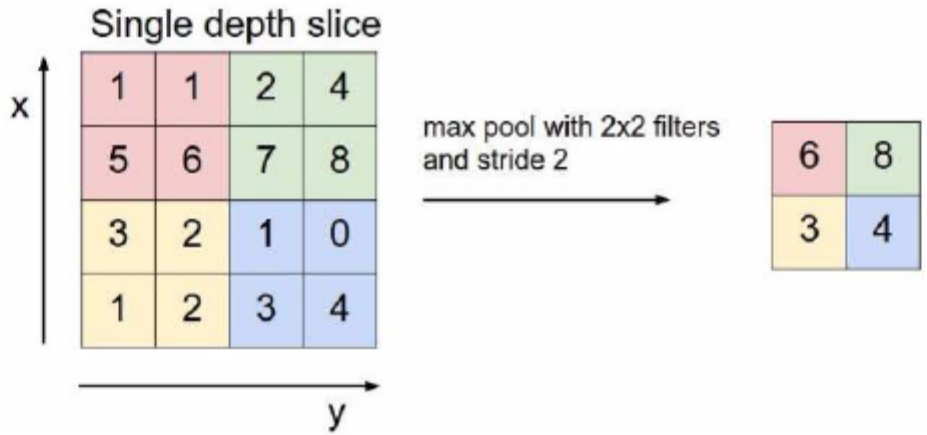


图 2.9 Max pooling 工作示意图

上图中我们选取的窗口大小为 2*2，步长为 2，意思就是每次我们将 4 个数据中最大的挑处来，然后以 2 为步长向左或者向下平移，上图中第一个 2*2 的方框内最大值为 6，所以结果为 6，向左平移两格后最大值为 8，所以结果为 8，窗口目前已达边缘，我们靠左并向下滑动两格，此时窗口最大值为 3，所以结果为三，最后一个窗口最大值为 4，所以结果为 4。虽然卷积神经网络也是将一个窗口的值通过一定的权重计算得到一个结果，但是 pooling 来得更直接更快。

Dropout 示意图如图 2.10 所示

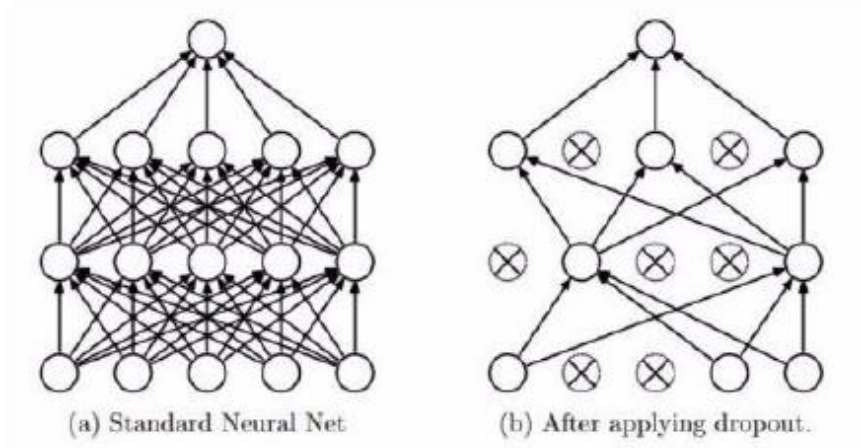


图 2.11 Dropout 示意图

此图我们可以这样理解每一次我们都关掉一部分感知器，得到一个新模型，最后做融合，这样不至于听一家之言，故可以减弱过拟合的影响。对于 Dropout 的理解还可以见图 2.11 所示

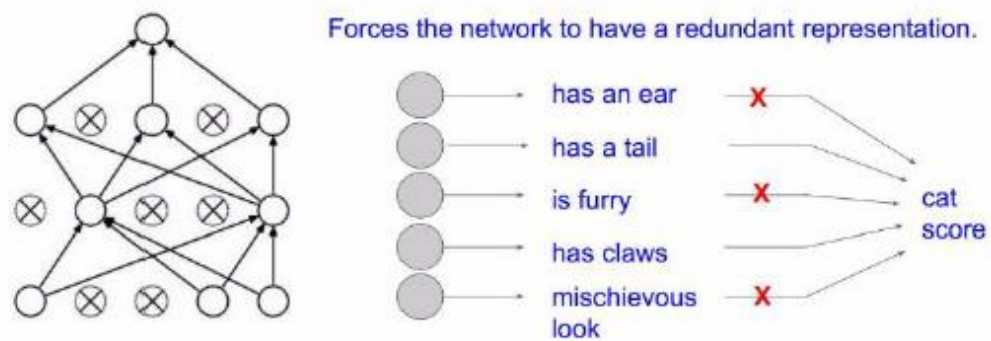


图 2.12 Dropout 示意图

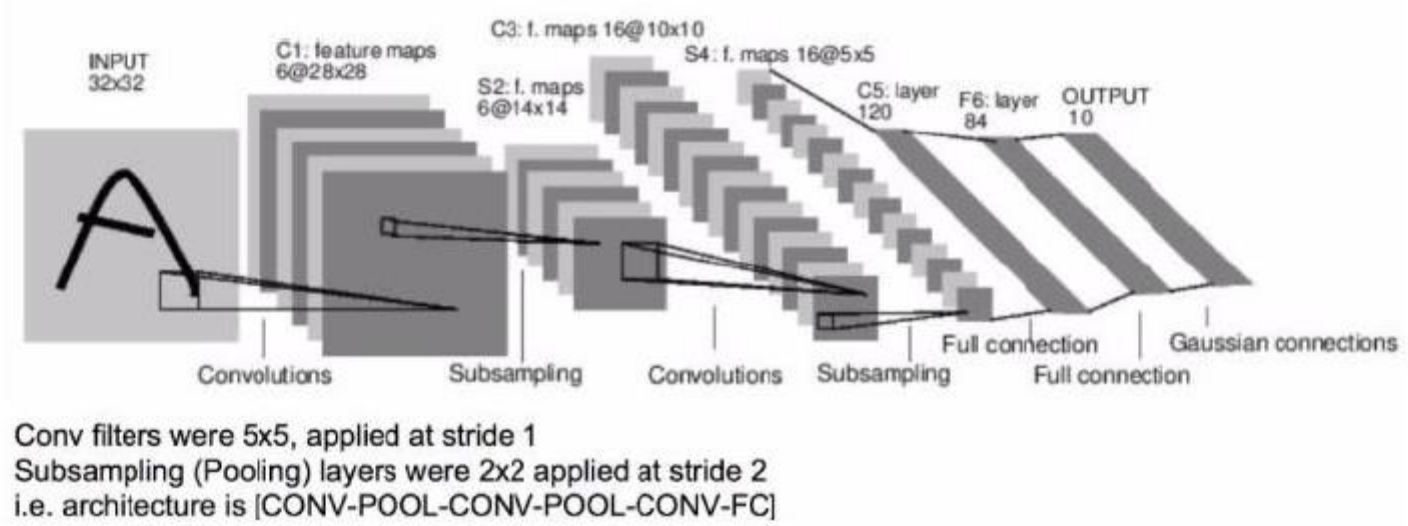


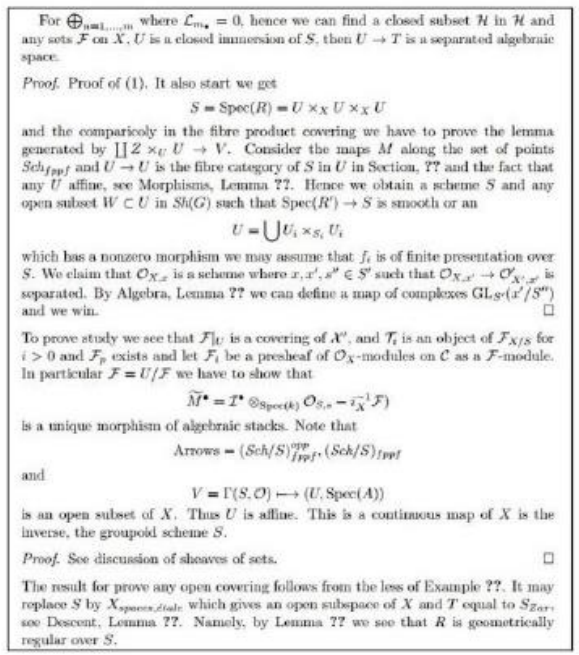
图 2.13 经典卷积神经网络 Lenet

该图像输入为 32*32，经过一个卷积层（以下讨论对于每一个小朋友），窗口为 5*5，步长为 1（会向左 28 步向下 28 步），故得到一个 28*28 的输出，经过一个 Subsampling（Pooling）窗口为 2*2，步长为 2（会向左走 14 步，向下 14 步），故得到一个 14*14 的输出，再来一个卷积层（会向左走 10 步，向下走 10 步），故得到一个 10*10 的输出，在经过一个池化层（向左走 5 步向下走 5 步），得到一个 5*5 的输出。由于 Lenet 第一个卷积层采用了六个小朋友进行观察，第二个卷积层采用了 16 个小朋友进行观察，所以现在得到结果是 16*5*5，然后紧跟着三个全连接层，分别是 120 个神经元，84 个神经元以及最后 10 个神经元。

刚进入这一章我们有必要来了解一下，为什么有了人工神经网络和复杂且功能强大的卷积神经网络后，人们还要提出一个新的循环神经网络（RNN）这一结构。一种结构之所以能被提出来，很大程度上是因为现有的结构还不能满足需求，所以在详细解说 RNN 之前，我们先来看看到底有什么场景与应用是人工神经网络与卷积神经网络所难以完成的。

一个很简单的例子就是机器翻译，也即输入一段话，给出一段话的翻译，这个是以往的人工神经网络与卷积神经网络所很难完成的任务，因为语言是有连续性的，很多时候一个单词的意思是由上下文语境共同决定的，所以不能简简单单的将每个单词对应的意思只是翻译出来，而且每个单词都有很多不同的意义，怎么选取。与图片或者简单的分类问题不同，比如一幅图上面有猫和狗，程序只需要识别出然后输出即可，但是如果遇到语言填空，比如问大天朝的官方语言是什么，这个就不是简简单单的只是对语言做特征提取就行了，你还需要预测接下来的答案。这个时候 RNN 就出现了，通过一定的机制使得其可以处理一段具有上下关联性的语句或其他。

目前 RNN 的应用包括，模仿论文如图 3.1 所示，通过给机器喂大量的论文，学习后开始学着去生成论文，还可以模仿 linux 内核代码来写程序，如图 3.2 所示，机器翻译如图 3.3 所示，看图说话和问答如图 3.4 所示



For $\bigoplus_{n=1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the compariccly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ??. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S''$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x''}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_{\overline{p}}$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - \overline{i}_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. □

The result for prove any open covering follows from the less of Example ??. It may replace S by $X_{spaces,\acute{e}tale}$ which gives an open subspace of X and T equal to $S_{\mathbb{Z}\alpha r}$, see Descent, Lemma ??. Namely, by Lemma ?? we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ over U compatible with the complex

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S. Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) *f is locally of finite type. Since $S = \mathrm{Spec}(R)$ and $Y = \mathrm{Spec}(R)$.*

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \sqcap U = \coprod_{i=1,\ldots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. □

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{\varepsilon_0} = \mathcal{F}_{K,\ldots,0}$.

Lemma 0.2. *Let X be a locally Noetherian scheme over S, $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq \mathfrak{p}$ is a subset of $\mathcal{J}_{n,0} \circ \overline{A}_2$ works.*

Lemma 0.3. *In Situation ??. Hence we may assume $\mathfrak{q}' = 0$.*

Proof. We will use the property we see that $\mathfrak{p}$ is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F-algebra where δ_{n+1} is a scheme over S. □

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

图 3.2 模仿写代码

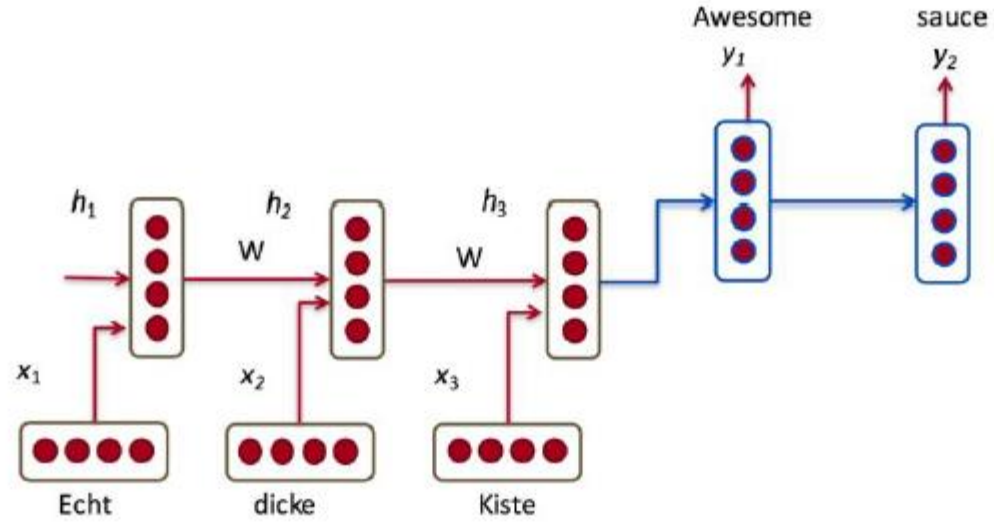


图 3.3 机器翻译

图 3.4 看图说话和问答

循环神经网络虽然结构与功能上与卷积神经网络不尽相同，但是其实神经网络也是包含卷积核的。接下来我们就要详细介绍简单循环神经网络的结构。

循环神经网络由于其有时序性，故其结构也是包含时序性的，我们可以将简单循环神经网络的结构按照时间顺序展开，如图 3.5 所示。

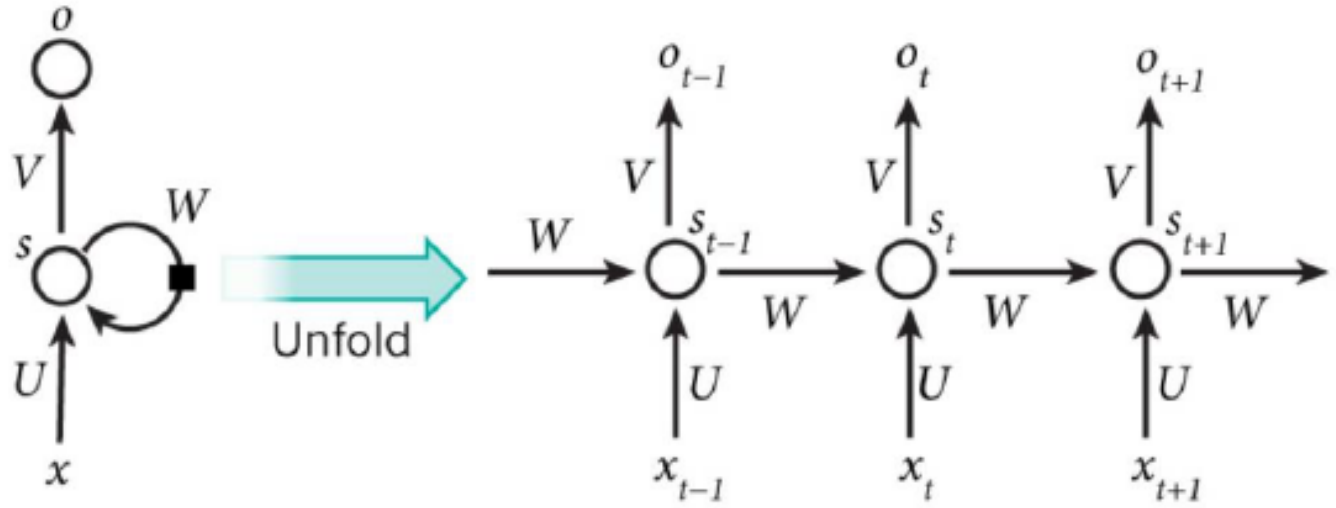
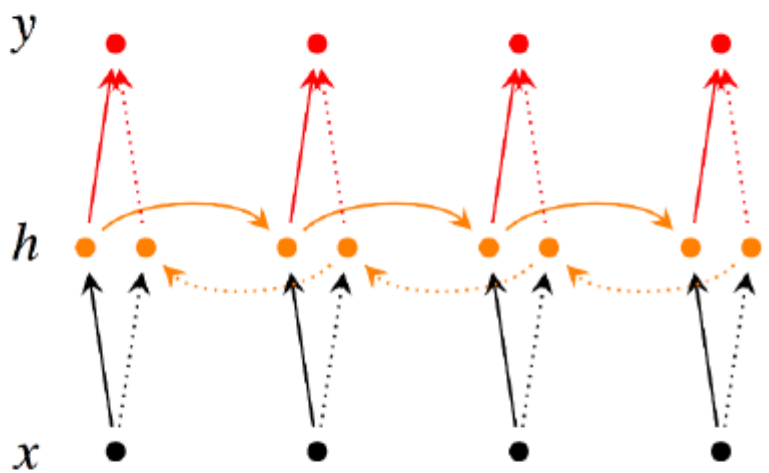


图 3.5 简单循环神经网络结构展开图

3.3.1 简单双向 RNN

为了使得我们可以从两个方向对数据进行处理，最简单的办法就是直接在加上另一个方向的简单 RNN 即可，示意图如图 3.7 所示



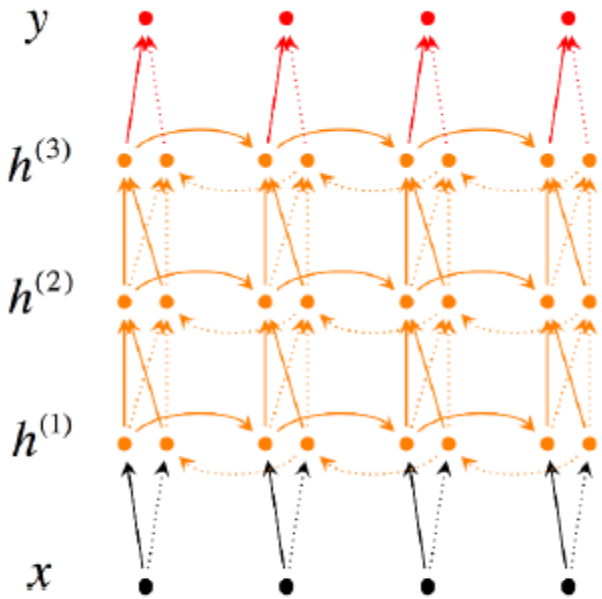
$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$

图 3.7 简单双向 RNN 展开示意图

简单双向 RNN 其实就是有两个细胞，每个细胞按照不同的方向对数据进行处理，然后将两个细胞的代谢产物（输出）按照一定的方式进行融合最后得到一个总的输出的过程。其中两个细胞之间是不进行通讯的，也即互不干扰，彼此独立，将两个细胞对应阶段的输出直接拼接然后进行一定的处理加上偏移函数之后经过一个激活函数就可以得到最终的输出。



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

图 3.8 深层双向 RNN 展开示意图

循环神经网络与卷积神经网络的参数更新思路是差不多的，都是通过使用 BP 反过来进行更新，但是由于循环神经网络 RNN 加入了时间，所以 BP 算法需要和时间进行结合，由此提出了 BPTT（Backpropagation Through Time）BPTT 主要是根据链式法则进行求导，过程是非常复杂的，这里简单介绍一下，不作深入研究。假设有如图 3.9 所示的 RNN，我们现在计算$\frac{\partial E_3}{\partial W}$的值，其中 W 是细胞每次代谢内部记忆体使用的卷积核，也即图中不同状态之间传递经过的卷积核。

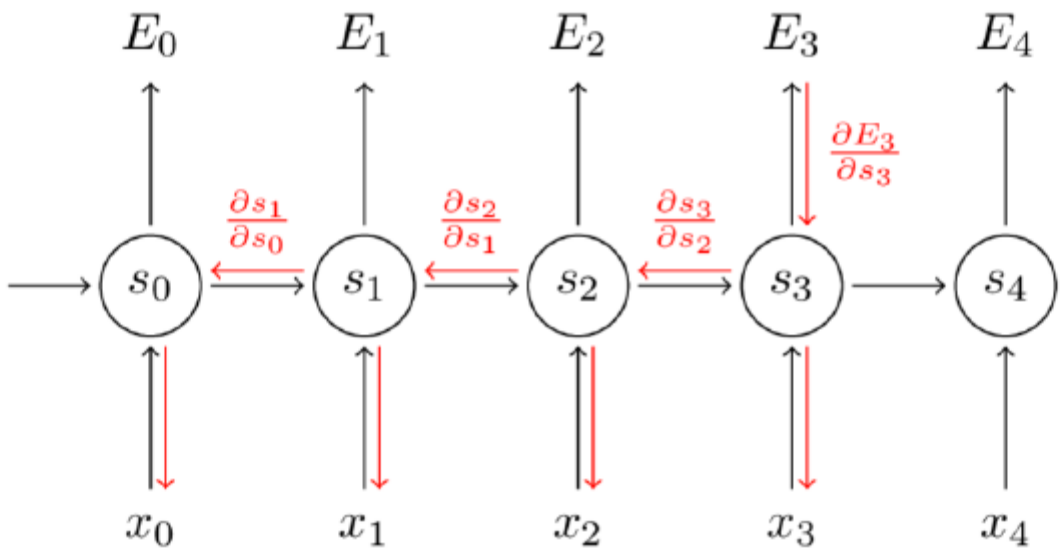


图 3.9 BPTT 算法演示

$$\frac{\partial E_3}{\partial W}=\frac{\partial E_3}{\partial S_3}\frac{\partial S_3}{\partial W}$$

比如看电影的时候，某些情节的伏笔可能会藏在很久以前的一些细节中，而随着时间间隔的不断增大，我们可以很轻易的发现 RNN 会逐渐丧失学习到连接如此远的信息的能力，也即有些信息在细胞经过很多轮代谢之后在记忆中基本上已经消失了。也就是说记忆的容量是有限的，一本书从头到尾一字不漏的去记，肯定离得越远的东西忘得越多。在这种情况下我们提出了 LSTM，实际上 LSTM 只是将原来简单 RNN 的细胞进行了改造，使得该被记住的信息会一直传递，不该记的会被“门”截断。原来的 RNN 的细胞简化图如图 3.10 所示，而现在改造后的 LSTM 细胞简化图如图 3.11 所示。这个细胞看起来可能很复杂，现在我们一步步来进行讲解：

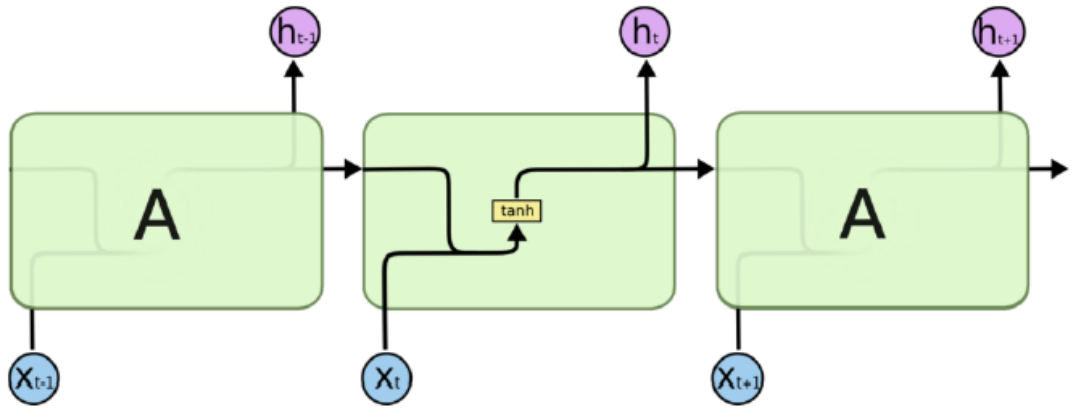
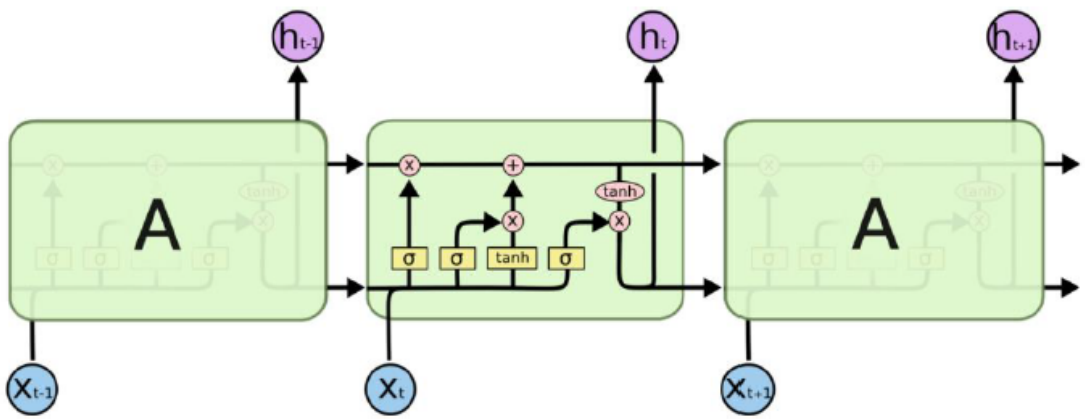


图 3.10 简单 RNN 细胞简化图



```python
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data", one_hot=True)

learning_rate = 0.01
train_iters = 200000
batch_size = 128
display_step = 10

n_input = 28*28
n_class = 10
dropout = 0.75

Weight = {
    # 权重有四个参数，第一个和第二个是filter的大小，第三个是输入的通道，第四个是输出的通道，这里进行详细解释，
    # 假设我们有32个通道，然后我们先用32个卷积核，每个核对应一个通道，那么每个核得到一个矩阵，然后将32个矩阵对
    # 应位置加到一起得到一个矩阵，接着如果偏置为2就将矩阵所有元素都+2这样得到一个feature map，将这个过程重复
    # 59遍得到了59个feature map，这就是32变成59的过程是吗？然后这个过程我们需要存储的卷积核的数量就是32*59个核
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),# 5x5 conv, 1 input, 32 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),# 5x5 conv, 32 inputs, 64 outputs
    'wf1': tf.Variable(tf.random_normal([7*7*64, 1024])),# fully connected, 7*7*64 inputs, 1024 outputs
    'out': tf.Variable(tf.random_normal([1024, n_class]))# 1024 inputs, 10 outputs (class prediction)
}

Bias = {
    'bc1':tf.Variable(tf.random_normal([32])),
    'bc2':tf.Variable(tf.random_normal([64])),
    'bf1':tf.Variable(tf.random_normal([1024])),
    'out':tf.Variable(tf.random_normal([n_class]))
}
def Conv2d(x, weights, biases, strides):
    conv2d = tf.nn.conv2d(x, weights, strides=[1, strides, strides, 1], padding='SAME')
    conv2d = tf.nn.bias_add(conv2d, biases)
    return tf.nn.relu(conv2d)
    # conv2d(input, filter, strides, padding)第一个是输入，第二个是过滤器，相当于卷积核

    # 第三个是步长，其中步长这样看(1,k,k,1)从最后一个参数开始看，相当于每次计算之后，将channel+1，即可
    # 也即再计算下一个通道的同样地方的卷积，然后再+1，直到计算完所有的通道后，使用倒数第二个参数，将宽度+k
    # 意味着此时我们会将框左移k个单位，然后继续更新最后一个参数，当移到最左边后开始使用第二个参数，将框向下
    # 移动k个单位，然后又开始更新最后一个参数，最后我们使用下一张图片在进行计算，所以如果我们最后一个参数
    # padding改为3的话，那么每次计算完成更新最后一个参数直接就就是channel+3，那么就回到了原点，那么此时会
    # 继续更新倒数第二个参数，也就是说到最后我们只计算了其中一个通道的卷积而忽略了其它两个通道的值

    # padding=VALID，那么经过5*5卷积核步长为1之后我们应该得到(28-4) × (28-4) 的矩阵
    # 如果padding=SAME那么就会进行填充使得结果还是28*28的矩阵

    # bias直接将对应位置相加，比如x是{[1,2],[3,4]}，bias=1,那么结果为{[2,3],[4,5]}
```

```python
def Max_pooling(x, kernel, strides):
    return tf.nn.max_pool(x, ksize=[1, kernel, kernel, 1],
                          strides=[1, strides, strides, 1], padding='SAME')

# Create model
def Conv_net(x, Weight, Bias, dropout):
    x = tf.reshape(x, shape=[-1, 28, 28, 1])# Reshape input picture
    # 需要将输入变成一个矩阵，其中reshape(input, shape),input就是输入的x，shape可以包括四个参数或更少的参数，这里解释四
    # 个参数的意思，第一个-1可以与[None, 784]中的None同意，就是不确定，可以为任意，第二个为height，第三个为width，第四个
    # 可以理解为通道的数量，也即最后x变成，任意张图片每个图片高与宽都为28位通道为1

    tensor = Conv2d(x, Weight['wc1'], Bias['bc1'], 1)#经过第一个卷积层
    tensor = Max_pooling(tensor, 2, 2)#经过第一个池化
    tensor = Conv2d(tensor, Weight['wc2'], Bias['bc2'], 1)#经过第二个卷积层
    tensor = Max_pooling(tensor, 2, 2)#经过第二个池化

    tensor = tf.reshape(tensor, [-1, Weight['wf1'].get_shape().as_list()[0]])# Reshape conv2 output to fit fully conne
    # tf.getshape()就是返回tensor的形状，例如tf.Variable(tf.float32, [a,b])就会返回(a,b)
    # tf.getshape().as_list()就是将(a,b)改变为[a,b]
    # tf.getshape().as_list()[0]就是取[a,b]第0位的值也就是返回a

    tensor = tf.add(tf.matmul(tensor, Weight['wf1']), Bias['bf1'])
    tensor = tf.nn.relu(tensor)

    tensor = tf.nn.dropout(tensor, dropout)# Apply Dropout

    out = tf.add(tf.matmul(tensor, Weight['out']), Bias['out'])
    return out

x = tf.placeholder(tf.float32, [None, n_input])
y = tf.placeholder(tf.float32, [None, n_class])
keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)

pred = Conv_net(x, Weight, Bias, keep_prob)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)

correct_pred = tf.equal(tf.arg_max(pred, 1), tf.arg_max(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    step = 1 # Keep training until reach max iterations
    while step * batch_size < train_iters:
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        sess.run(optimizer, feed_dict={x:batch_x, y:batch_y, keep_prob:dropout})
        if step % display_step == 0:
            loss, acc = sess.run([cost, accuracy], feed_dict={x:batch_x, y:batch_y, keep_prob:dropout})
            print "Iter " + str(step*batch_size) + ", Minibatch Loss= " + \
                  "{:.6f}".format(loss) + ", Training Accuracy= " + \
                  "{:.5f}".format(acc)
        step += 1
    print "Optimization Finished!"

    # Calculate accuracy for 256 mnist test images
    print "Testing Accuracy:", \
        sess.run(accuracy, feed_dict={x: mnist.test.images[:256],
                                      y: mnist.test.labels[:256],
                                      keep_prob: 1.})
```

```
Iter 181760, Minibatch Loss= 0.000000, Training Accuracy= 1.00000
Iter 183040, Minibatch Loss= 22.442045, Training Accuracy= 0.96875
Iter 184320, Minibatch Loss= 26.575214, Training Accuracy= 0.96875
Iter 185600, Minibatch Loss= 13.759210, Training Accuracy= 0.97656
Iter 186880, Minibatch Loss= 8.145270, Training Accuracy= 0.99219
Iter 188160, Minibatch Loss= 44.384930, Training Accuracy= 0.96094
Iter 189440, Minibatch Loss= 5.590534, Training Accuracy= 0.98438
Iter 190720, Minibatch Loss= 14.488055, Training Accuracy= 0.98438
Iter 192000, Minibatch Loss= 27.034473, Training Accuracy= 0.97656
Iter 193280, Minibatch Loss= 0.000000, Training Accuracy= 1.00000
Iter 194560, Minibatch Loss= 4.710227, Training Accuracy= 0.98438
Iter 195840, Minibatch Loss= 24.817835, Training Accuracy= 0.98438
Iter 197120, Minibatch Loss= 16.209532, Training Accuracy= 0.99219
Iter 198400, Minibatch Loss= 0.000000, Training Accuracy= 1.00000
Iter 199680, Minibatch Loss= 1.012459, Training Accuracy= 0.99219
Optimization Finished!
Testing Accuracy: 0.988281
```