Author:UsanoCoCr Date:2023.4.19 仅为北京大学人工智能基础复习使用

目录

全局搜索

全局搜索模型的描述

全局搜索可以用来解决一定的问题，描述一个问题模型一般有以下几个要素： 1.初始状态state0 2.可选择动作action_set 3.状态转移函数next_state=state0+action 4.目标状态state_goal 5.花费函数cost=cost(state,action)

图搜索与树搜索

一般情况下，搜索分为树搜索和图搜索两种，树搜索是指搜索的过程中不会出现环，图搜索是指搜索的过程中可能出现环，这两种搜索的区别在于**图搜索需要记录已经搜索过的节点**，以避免重复搜索，而树搜索不需要记录已经搜索过的节点，因为树搜索不会出现环，所以不需要记录已经搜索过的节点。

下面给出树搜索和图搜索的伪代码：

▶ 树搜索

```
void tree_search(state0,action_set,next_state,cost,state_goal)
{
    if(state0==state_goal)//判断是否到达目标状态
        return state0;
    for(action in action_set)//遍历所有可选择的动作
```

```
    {
        next_state=state0+action;//计算下一个状态
        tree_search(next_state,action_set,next_state,cost,state_goal);//递归搜索
    }
}
```

▶ 图搜索

```
void graph_search(state0,action_set,next_state,cost,state_goal)
{
    explored=empty;//初始化explored
    if(state0==state_goal)//判断是否到达目标状态
        return state0;
    for(action in action_set)//遍历所有可选择的动作
    {
        next_state=state0+action;//计算下一个状态
        if(next_state not in explored)//判断下一个状态是否已经搜索过
        {
            graph_search(next_state,action_set,next_state,cost,state_goal);//递归
搜索
            explored.add(next_state);//将下一个状态加入已搜索状态集合
        }
    }
}
```

上述树搜索/图搜索有可能陷入死循环，在这种情况下我们需要加入判定条件，当搜索的次数超过一定次数时，就停止搜索，这样就可以避免陷入死循环。

搜索算法的评价标准

搜索算法的评价标准一般有以下几个： 1.搜索的时间复杂度 2.搜索的空间复杂度 3.搜索的完备性：当解存在的时候，算法是否一定能够找到解？ 4.搜索的最优性：当解存在的时候，算法是否一定能够找到最优解？

在搜索问题中，时间复杂度经常用搜索树展开的节点数目表示 空间复杂度经常用同一时间需要储存的最大节点数目来估计

无信息搜索

宽度优先搜索bfs

宽度优先搜索的时间复杂度为O(b^d),空间复杂度为O(b^d) 其中b为分支因子（一个节点有多少个子节点），d为搜索深度 在使用c++实现bfs时，一般会使用queue结构，闭节点集合一般使用set结构，这样可以避免重复搜索。

▶ 宽度优先搜索bfs

```
void bfs(state0,action_set,next_state,cost,state_goal)
{
```

```
    queue=empty;//初始化queue
    explored=empty;//初始化explored
    queue.push(state0);//将初始状态加入queue
    while(queue not empty)//当queue不为空时
    {
        state=queue.pop();//从queue中取出一个状态
        if(state==state_goal)//判断是否到达目标状态
            return state;
        for(action in action_set)//遍历所有可选择的动作
        {
            next_state=state+action;//计算下一个状态
            if(next_state not in explored)//判断下一个状态是否已经搜索过
            {
                queue.push(next_state);//将下一个状态加入queue
                explored.add(next_state);//将下一个状态加入已搜索状态集合
            }
        }
    }
}
```

深度优先搜索dfs

深度优先搜索的时间复杂度为O(b^m),空间复杂度为O(bm) 其中b为分支因子（一个节点有多少个子节点），m为搜索树的最大深度 在使用c++实现dfs时，一般会使用stack结构 深度优先搜索在内存受限的情况下可以搜的更深，所以**在内存受限的情况下，深度优先搜索比宽度优先搜索更好。**

▶ 深度优先搜索dfs

```
void dfs(state0,action_set,next_state,cost,state_goal)
{
    stack=empty;//初始化stack
    explored=empty;//初始化explored
    stack.push(state0);//将初始状态加入stack
    while(stack not empty)//当stack不为空时
    {
        state=stack.pop();//从stack中取出一个状态
        if(state==state_goal)//判断是否到达目标状态
            return state;
        for(action in action_set)//遍历所有可选择的动作
        {
            next_state=state+action;//计算下一个状态
            if(next_state not in explored)//判断下一个状态是否已经搜索过
            {
                stack.push(next_state);//将下一个状态加入stack
                explored.add(next_state);//将下一个状态加入已搜索状态集合
            }
        }
    }
}
```

深度受限的深度优先搜索

dfs因为没有存储过搜索过的点，所以可能会陷入死循环 深度受限的深度优先搜索的时间复杂度为O(b^l),空间复杂度为O(bl) 其中b为分支因子（一个节点有多少个子节点），l为限制层数

▶ 深度受限的深度优先搜索

```
void graph_search(state0,action_set,next_state,cost,state_goal,limit)
{
    if(state0==state_goal)
        return state0;
    if(limit==0)//判断是否到达限制层数
        return cutoff;
    cutoff_occurred=false;//判断是否陷入死循环
    for(action in action_set)
    {
        next_state=state0+action;

result=graph_search(next_state,action_set,next_state,cost,state_goal,limit-1);//递
归搜索
        if(result==cutoff)
            cutoff_occurred=true;
        else if(result!=failure)
            return result;
    }
    if(cutoff_occurred)
        return cutoff;
    else//程序陷入死循环
        return failure;
}
```

深度受限搜索引出了迭代加深算法：在深度受限的深度优先搜索中，我们可以将限制层数从0开始逐渐增加，直到找到解为止。

双向搜索

双向搜索的时间复杂度为O($b^{d/2}$),空间复杂度为O($b^{d/2}$)

一致代价搜索

一致代价搜索在c++实现中使用了优先队列的结构，通过节点到起点的cost来排列，当终点从优先队列中输出时，即找到最优解

下面是一致代价搜索的伪代码：

▶ 一致代价搜索

```
void uniform_cost_search(state0,action_set,next_state,cost,state_goal)
{
    priority_queue=empty;//初始化优先队列
```

```
    explored=empty;//初始化explored
    priority_queue.push(state0);//将初始状态加入优先队列
    while(priority_queue not empty)//当优先队列不为空时
    {
        state=priority_queue.pop();//从优先队列中取出一个状态
        if(state==state_goal)//判断是否到达目标状态
            return state;
        for(action in action_set)//遍历所有可选择的动作
        {
            next_state=state+action;//计算下一个状态
            if(next_state not in explored)//判断下一个状态是否已经搜索过
            {
                priority_queue.push(next_state);//将下一个状态加入优先队列
                explored.add(next_state);//将下一个状态加入已搜索状态集合
            }
        }
    }
}
```

有信息搜索

有信息搜索在搜索时会使用启发式函数f **可采纳性：f不会高估到达目标的代价，即f(n)<=c(n,n_goal) 一致性：对启发式函数，如果从状态n1到n2的代价为c，那么f(n1)<=c+f(n2)**

贪婪最佳搜索

有信息搜索在搜索时会使用启发式函数h(n)，同样使用了优先队列的结构，通过节点到终点的启发式函数值来排列

A*搜索

A搜索在c++实现中使用了优先队列的结构，通过节点到起点的cost加上节点到终点的启发式函数值来排列 即：f(n)=g(n)+h(n) A搜索的实际效果取决于启发式函数的好坏 **如果启发式函数f(n)是可采纳的，那么A*搜索是最优的（即一定可以找到最优解）**

估值函数是乐观的，A*就是最优的

常见的启发式函数

- 曼哈顿距离
- 欧拉距离
- 对角线距离
- 最大距离
- 最小距离

启发式函数不必完全接近真实值，有时我们可以松弛启发性函数的值

局部搜索

爬山算法

爬山算法每次会从当前状态移动到相邻节点中最好的一个，算法会在山峰/山谷处停下，算法并不储存一棵搜索树，只存储当前节点和估值函数

下面给出爬山算法的伪代码：

▶ 爬山算法

```
void hill_climbing(state0,action_set,next_state,cost,state_goal)
{
    state=state0;//初始化当前状态
    while(true)//循环
    {
        next_state=best_next_state(state,action_set);//找到最好的邻居节点
        if(next_state.value<=state.value)//如果所有邻居节点的值都比当前节点的值小，说明已经到达山顶
            return state;
        state=next_state;//更新当前状态
    }
}
```

爬山算法可能存在的问题：

- 陷入局部最优：到达局部极值点就僵住不动了，可能达不到全局最优解

解决方法： 1.随机平移：在局部最优点附近随机选择一个点，然后继续搜索 2.随机重启：在整个搜索空间中随机选择一个点，然后继续搜索 针对上述可能存在的问题，爬山法衍生出了很多变种

随机爬山法

随机爬山法在爬山算法的基础上，每次从当前状态的所有更优的邻居节点中随机选择一个节点作为下一个状态，随机爬山法收敛的更慢，但是往往可以找到更优的解

第一选择爬山法

第一选择爬山法每次随机找一个邻居，如果邻居估值更优，则直接跳到邻居，否则继续找下一个邻居，直到找到一个更优的邻居，或者所有邻居都不比当前状态更优 此方法**不需要计算所有邻居节点的估值，在一个状态有特别多个邻居节点的时候，可以大大减少计算量**

随机重启爬山法

上述爬山法变种虽然可以提高爬山法的效率，但是也不具有完备性，有可能会陷在局部极值 随机重启爬山法保证了爬山过程中，如果找不到解，则从一个随机位置再次开始爬山，直到找到解为止

下面给出随机重启爬山法的伪代码：

▶ 随机重启爬山法

```
void hill_climbing(state0,action_set,next_state,cost,state_goal)
{
    state=state0;//初始化当前状态
```

```
    while(true)//循环
    {
        next_state=best_next_state(state,action_set);//找到最好的邻居节点
        if(next_state.value<=state.value)//如果所有邻居节点的值都比当前节点的值小，说明已经到达山顶
            return state;
        state=next_state;//更新当前状态
    }
}

void random_restart_hill_climbing(state0,action_set,next_state,cost,state_goal)
{
    while(true)//循环
    {
        state=hill_climbing(state0,action_set,next_state,cost,state_goal);//调用爬山算法
        if(state==state_goal)//判断是否到达目标状态
            return state;
        state0=random_state();//从随机位置开始爬山
    }
}
//在使用随机重启爬山法时，可以设置一个最大迭代次数，如果超过最大迭代次数还没有找到解，则返回当前找到的最优解
```

模拟退火算法

模拟退火算法设置了一个概率函数p，用来决定是否接受一个更差的解，**当前节点有概率p(t0)跳到一个比他更差的邻居节点。p的值随着迭代次数的增加而减小**，模拟退火算法的伪代码如下：

▶ 模拟退火算法

```
void simulated_annealing(state0,action_set,next_state,cost,state_goal)
{
    state=state0;//初始化当前状态
    t0=100;//初始温度
    while(true)//循环
    {
        next_state=random_next_state(state,action_set);//随机选择一个邻居节点
        if(next_state.value<=state.value)//邻居节点更差
        {
            if(p(t0)>rand())//如果概率大于随机数，则接受更差的解
                state=next_state;
        }
        else//邻居节点更优
            state=next_state;
        t0=t0-1;//温度减小
    }
}
```

局部束搜索

局部束搜索在同一时刻保留了k个状态，每个状态生成b个后继，如果k*b个后继中存在全局最优解，则直接返回，否则，从*kb个后继中选择k个最优的后继，作为下一时刻的k个状态，继续搜索

局部束搜索和随机重启爬山法的区别：

- 随机重启爬山法每次搜索是独立的，而**局部束搜索将k个后继节点作为一个整体，每次搜索在这k个后继节点中进行挑选**

遗传算法

遗传算法模拟了生物有性生殖、变异、自然选择的过程，通过不断的迭代，最终得到一个最优解 在遗传算法中会使用到函数： 1.random_state()：随机生成一个状态 2.reproduce(state1,state2)：将两个状态交配 3.mutate(state)：对状态进行变异 4.evaluate(state)：评估状态的优劣

遗传算法的伪代码如下：

▶ 遗传算法

```
auto reproduce(state1,state2)
{
    new_state=empty;
    c=rand();//随机选择一个数
    new_state.add(substring(state1,0,c));//将state1的前c个元素加入新状态
    new_state.add(substring(state2,c+1,size-1));//将state2的后size()-c个元素加入新状态
    return new_state;
}

void genetic_algorithm(state0,action_set,next_state,cost,state_goal)
{
    population_size=100;//种群大小
    population=init_population(population_size);//初始化种群
    while(true)//循环
    {
        new_population=empty;
        for(i=0;i<population_size;i++)
        {
            x=random_state();//随机选择一个状态
            y=random_state();//随机选择一个状态
            z=reproduce(x,y);//交配
            if (rand()<0.1)//有10%的概率进行变异
                z=mutate(z);
            new_population.add(z);//将新生成的状态加入种群
        }
        population=new_population;//更新种群
    }
    best_state=best_state(population);//找到最优解
}
```

梯度下降法

梯度下降法在实现上和爬山法非常相似，梯度下降即连续集里的爬山算法

对抗搜索

ai基础课所讲的对抗搜索主要运用在双人零和完全信息游戏中，在这一类游戏中，我们可以通过搜索来描述游戏模型：

- 初始状态：游戏开始时的状态
- Players：轮到哪个玩家进行操作
- Actions：每个玩家可以进行的操作
- 状态转移模型：每个玩家进行操作后，游戏的状态如何转移
- 终止条件：游戏何时结束
- 效用函数：评估游戏结束后两位玩家的得分

极大极小值搜索 Minimax

Minimax算法中双方作为理性决策者，都会选择让自己收益最大的节点 Minimax算法的时间复杂度是O(b^m)，空间复杂度是O(bm)， 其中b为每个节点的分支数，m为搜索深度

Minimax算法的伪代码如下：

▶ Minimax算法

```
auto minimax(state,player)
{
    if(state is terminal)//判断是否为终止状态
        return utility(state);//返回效用值
    if(player==MAX)//如果是MAX玩家
    {
        v=-inf;//v为最大值
        for each action in actions(state)//遍历所有的操作
        {
            v=max(v,minimax(result(state,action),MIN));//更新v
        }
        return v;
    }
    else//如果是MIN玩家
    {
        v=inf;//v为最小值
        for each action in actions(state)//遍历所有的操作
        {
            v=min(v,minimax(result(state,action),MAX));//更新v
        }
        return v;
    }
}
```

Alpha-Beta剪枝

Alpha-Beta剪枝是对Minimax算法的优化，它可以减少搜索的节点数，从而减少搜索的时间 Alpha-Beta剪枝剪去的节点一般拥有这样的特点：

- 当该节点为最大节点时，节点上方的路径要取最小值，下方的路径要取最大值，当节点的子节点传上来一个value=v，且v大于节点的兄弟节点的value=β时，我们就不用展开这个节点了，因为节点上方的取最小值路径一定会选择兄弟节点的路径
- 当该节点为最小节点时，节点上方的路径要取最大值，下方的路径要取最小值，当节点的子节点传上来一个value=v，且v小于节点的兄弟节点的value=α时，我们就不用展开这个节点了，因为节点上方取最大值的路径一定会选择兄弟节点的路径

Alpha-Beta剪枝算法的伪代码如下：

▶ Alpha-Beta剪枝算法

```
auto alphabeta(state,player,alpha,beta)
{
    if(state is terminal)//判断是否为终止状态
        return utility(state);//返回效用值

    //注意：这里的最大值最小值节点是三层中的最上层
    function max_value(state,alpha,beta)
    {
        v=-inf;//v为最大值
        for each action in actions(state)//遍历所有的操作
        {
            v=max(v,min_value(result(child_state,action),alpha,beta));//更新v
            if(v>=beta)//如果v大于beta
                return v;//返回v
            alpha=max(alpha,v);//更新alpha
        }
        return v;
    }

    function min_value(state,alpha,beta)
    {
        v=inf;//v为最小值
        for each action in actions(state)//遍历所有的操作
        {
            v=min(v,max_value(result(child_state,action),alpha,beta));//更新v
            if(v<=alpha)//如果v小于alpha
                return v;//返回v
            beta=min(beta,v);//更新beta
        }
        return v;
    }
}
```

不完美的实时决策

不完美的实时决策是指在决策过程中，我们并不知道游戏的终止条件，因此我们需要**在决策过程中不断的评估当前的状态，使用启发式函数来辅助搜索** 当到了一定的时间/深度，则停止搜索，返回当前的最优解，这种方法是一种蒙特卡洛树搜索的近似方法

蒙特卡洛树搜索 MCTS

蒙特卡洛搜索是在一定的预设条件（时间、内存、迭代次数）下不断地迭代进行更新，并利用统计学原理找到近似最优解的一种方法 **蒙特卡洛的每次迭代包括四个步骤：选择、扩张、模拟、反向传播**

选择

选择是指从根节点开始，不断地选择下一个节点，直到遇到一个未被访问的节点，或者遇到一个终止节点 选择部分的代码是：

▶ 选择

```
function select(node)
{
    while(node is not terminal)//如果节点不是终止节点
    {
        if(node is fully expanded)//如果节点已经被扩张
        {
            node=best_child(node,ucb1) //选择最优子节点
            return node
        }
        else//如果节点未被扩张
        {
            expand(node)//扩张节点
        }
    }
}
```

扩张

扩张是指从当前节点中选择未被访问的子节点，将其加入到当前节点的子节点中 扩张部分的代码是：

▶ 扩张

```
function expand(node)
{
    if(node is not fully expanded)//如果节点未被扩张
    {
        action=untried_actions(node)//获取未被访问的子节点
        child_node=child_node(node,action)//获取子节点
        add_child(node,child_node)//将子节点加入到当前节点的子节点中
    }
}
```

模拟

模拟是指从当前节点开始，不断地随机选择一个子节点，直到遇到一个终止节点，然后返回该终止节点的效用值 有时我们可以选择随机模拟直到终局来获得效用值，也可以通过启发式函数来获得效用值 在评估上，人们人造了一个函数来反映节点的好坏： *$UCB = Vi + C\sqrt{(ln(n)/N_i)}$*

反向传播

反向传播是指从当前节点开始，不断地更新父节点的效用值，直到遇到根节点 反向传播部分的代码是：

▶ 反向传播

```
function backpropagate(node,value)
{
    while(node is not root)//如果节点不是根节点
    {
        node.visits++//节点的访问次数加一
        node.value+=value//节点的效用值加上value
        node=node.parent//节点指向父节点
    }
    //其中，value = UCB(node)
}
```

总结

此时，我们可以写出蒙特卡洛搜索的完整版伪代码：

▶ 蒙特卡洛搜索

```
function monte_carlo_tree_search(state)
{
    root=initialise(state)//初始化根节点
    while(时间未到)//如果时间未到
    {
        node=select(root)//选择节点
        value=simulate(node)//模拟节点
        backpropagate(node,value)//反向传播
    }
    return best_child(root,ucb1)//返回最优子节点
}
```

强化学习

强化学习在游戏中也有很广泛的应用，我们可以构建问题模型：

- 初始状态S0
- 当前玩家C

- 动作A
- 状态转移函数P
- 终止状态ST
- 奖励函数R

注意：状态转移和奖励都不一定是确定性的，可以是一个概率分布 在强化学习中，我们要寻找一个最优策略π，使得从初始状态S0开始，不断地执行策略π，最终达到终止状态ST，期间的奖励函数R最大化

强化学习的环境

累积收益

在计算强化学习某一路径的总价值时，我们会使用累积收益G G的计算公式是： *$G_t = R_{t+1} + γ*R_{t+2} + ... + γ^{T-t+1}R_T$ 其中，$γ$是折扣因子，$γ$越大，表示对未来的奖励越看重，$γ$越小，表示对未来的奖励越不看重

状态价值

状态价值是指在某一状态s下，执行某一策略π，从该状态开始，不断地执行策略π，最终达到终止状态ST，期间的奖励函数R的期望值 状态价值的计算公式是： *$V_{π}(s) = E_{π}[G_t|S_t=s]$ = $E_{π}[R_{t+1} + γ*R_{t+2} + ... + γ^{T-t+1}R_T|S_t=s]$

动作价值

动作价值是指在某一状态s下，执行某一策略π，从该状态开始，执行某一动作a，不断地执行策略π，最终达到终止状态ST，期间的奖励函数R的期望值 动作价值的计算公式是： *$Q_{π}(s,a) = E_{π}[G_t|S_t=s,A_t=a]$ = $E_{π}[R_{t+1} + γ*R_{t+2} + ... + γ^{T-t+1}R_T|S_t=s,A_t=a]$

状态价值和动作价值可以通过数学的方式建立联系： *$V_{π}(s) = Σ_a π(a|s)Q_{π}(s,a)$ 同理，有： **$Q_{π}(s,a) = Σ_{s'} Σ_r p(s',r|s,a)(r + γV_{π}(s'))$ 上述式子即为贝尔曼方程，它的意思是：在状态s下，执行动作a，得到状态s'和奖励r，那么在状态s下，执行动作a的价值就是 奖励r + 折扣因子*状态s'产生的价值依概率的期望值**

寻找最优策略的方法

贪心策略

贪心策略是指在每一步都选择当前状态下价值最大的动作，即： **$a = argmax_a Q_{π}(s,a)$**

ε-贪心策略

ε-贪心策略是指在每一步都有一定的概率选择随机动作，即： **$a = argmax_a Q_{π}(s,a)$ with probability 1-ε $a = random$ with probability ε**

乐观初值贪心

乐观初值贪心将所有节点动作价值的初始值设的很大，这样可以使得在初始阶段，鼓励没有被访问到的节点被访问，从而使得更多节点的价值有机会被更新，从而使得最终的策略更加优秀

UCB

估值大的节点，我们应该去访问；访问次数少的节点，我们应该去访问 将当前的估值和访问次数结合起来，得到UCB值，UCB值越大，表示该节点的价值越大，越有可能是最优节点 UCB公式是： *$UCB = Q_{π}(s,a) + c\sqrt{(ln(N)/n)}$*

强化学习的算法

贝尔曼方程

贝尔曼方程分为贝尔曼期望方程和贝尔曼最优方程 贝尔曼期望方程是指： **$Q_{π}(s,a) = Σ_{s'} Σ_r p(s',r|s,a)(r + γV_{π}(s'))$** 这个式子在介绍动作价值和状态价值的关联性时已经介绍过，下面将介绍贝尔曼最优方程： **$Q_{π}(s,a) = Σ_{s'} Σ_r p(s',r|s,a)(r + γmax_{a'}Q_{π}(s',a'))$**

在强化学习中，**贝尔曼期望方程对应着策略迭代，贝尔曼最优方程对应着值迭代**

策略迭代

策略迭代算法一共分为三个步骤：

- 策略估值
- 策略提升
- 策略迭代

策略估值

策略估值的目的是根据当前的策略，利用贝尔曼期望方程，计算出每个状态的值函数$v_π$，即： **$v_π(s) = Σ_aπ(a|s) Σ_{s'} Σ_r p(s',r|s,a)(r + γv_π(s'))$** 同时拥有迭代更新规则： **$v_π(s) = Σ_aπ(a|s) Σ_{s'} Σ_r p(s',r|s,a)(r + γv_k(s'))$** 其中，$v_π(s)$对应的是策略π下准确稳定的状态值函数，$v_k(s)$对应的是策略π下第k次迭代的状态值函数 策略估值在计算状态值函数时，是利用当前策略π，计算出下一次迭代的状态值函数v_{k+1}，然后再利用v_{k+1}计算出v_{k+2}，以此类推，直到v_{k+1}和v_k的差值小于某个阈值，即$v_{k+1} - v_k < \theta$，则停止迭代，得到$v_π$，即为策略π下准确稳定的状态值函数 下面是策略估值的伪代码：

▶ 策略估值

```
function policy_evaluation(π,θ)
    V = 0
    while True
        delta = 0
        for s in S
            v = V[s]
            V[s] = Σ_aπ(a|s) Σ_{s'} Σ_r p(s',r|s,a)*(r + γ*V[s'])
            delta = max(delta, abs(v - V[s]))
        if delta < θ
            break
    return V
```

策略提升

策略提升的目的是根据当前的状态值函数$v_π$，计算出每个状态下的最优动作价值函数$q_π$，即：**$q_π(s,a) = Σ_{s'} Σ_r p(s',r|s,a)(r + γv_π(s'))$** 之后，我们可以通过贪心选择动作产生新的策略，即：**$π'(s) = argmax_a q_π(s,a)$**

策略迭代

策略迭代的目的是不断迭代策略估值和策略提升，直到策略不再发生变化，策略迭代其实就相当于是策略估值和策略提升的结合，它的伪代码如下：

▶ 策略迭代

```
function policy_iteration(θ)
    1. initialize π arbitrarily

    2. policy evaluation
    while True
        delta = 0
        for s in S
            v = V[s]
            V[s] = Σ_{s'} Σ_r p(s',r|s,π(a))*(r + γ*V[s'])
            delta = max(delta, abs(v - V[s]))
        if delta < θ
            break

    3. policy improvement
    while True
        policy_stable = True
        for s in S
            old_action = π(s)
            π(s) = argmax_a Σ_{s'} Σ_r p(s',r|s,a)*(r + γ*V[s'])
            if π(s) != old_action
                policy_stable = False
        if policy_stable
            break
    return π
```

策略迭代的终止条件是策略不再发生变化，也就是贝尔曼最优方程中的最优动作价值函数$q_π(s,a)$不再发生变化。策略迭代的伪代码中，策略估值和策略提升是交替进行的。

值迭代

值迭代算法利用的是贝尔曼最优方程，它的伪代码如下：

▶ 值迭代

```
function value_iteration(θ)
    V = 0
    while True
        delta = 0
        for s in S
```

```
            v = V[s]
            V[s] = max_a Σ_{s'} Σ_r p(s',r|s,a)*(r + γ*V[s'])
            delta = max(delta, abs(v - V[s]))
        if delta < θ
            break
    return V
```

值迭代更新后的V[s]，是当前状态下所有动作的最优动作价值函数的最大值。

广义策略迭代 GPI

策略迭代和值迭代可以用来选择动作产生新策略，这起到了一种拮抗的作用，在GPI中，我们可以同时使用策略迭代和值迭代得到稳定的解决方案，同时引入协同学习的思想，使得策略迭代和值迭代的效果更好。

TD学习

TD学习和蒙特卡洛算法在实际操作层面上有一些差异：

- TD在有或者没有结果的情况下均可以学习(如不必等一盘象棋结束)；MC则必须等待结束得到结果。TD胜！
- TD在更新状态价值时使用的是TD 目标值，即基于即时奖励和下一状态的预估价值来替代当前状态在状态序列结束时可能得到的收获，是当前状态价值的有偏估计；而MC则使用实际的收获来更新状态价值，是某一策略下状态价值的无偏估计，MC胜！
- 虽然TD得到的价值是有偏估计，但是其方差却比MC得到的方差要低，且对初始值敏感，通常比MC法更加高效。TD胜！

TD学习的伪代码如下：

▶ TD学习

```
function TD(θ)
    V = 0
    while True
        delta = 0
        for s in S
            v = V[s]
            V[s] = V[s] + α*(r + γ*V[s'] - V[s])
            delta = max(delta, abs(v - V[s]))
        if delta < θ
            break
    return V
```

在强化学习的应用中，有两种时序差分算法，分别是SARSA算法和Q-learning算法，它们都可以被运用到实践中。

SARSA算法

- SARSA算法是一种基于值的强化学习算法，它的全称是State-Action-Reward-State-Action，即状态-动作-奖励-状态-动作。

- SARSA算法的核心思想是：在每一步更新Q值时，都**使用ε-贪心最优的动作来更新Q值**，即$Q(s,a)←Q(s,a)+α(r+γQ(s',a')-Q(s,a))$

SARSA算法的伪代码如下：

▶ SARSA算法

```
Initialize Q(s,a),∀s∈S,a∈A(s),Q(terminal-state,·)=0
Repeat (for each episode):
    Initialize s
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g., ε-greedy)
        Q(s,a)←Q(s,a)+α(r+γQ(s',a')-Q(s,a))
        s←s'; a←a';
    until s is terminal
```

SARSA通过策略给定最优的第二步来估计当前最优的第一步，如果第二步的效果很好，那么Q值就会被更新为更大的值，反之则会被更新为更小的值。

Q-learning算法

- Q-learning算法是一种基于值的强化学习算法，它的全称是Quality-Learning，即质量学习。
- Q-learning算法的核心思想是：在每一步更新Q值时，都**使用下一状态下的最大动作来更新Q值**，即:$Q(s,a)←Q(s,a)+α(r+γ*argmax_aQ(s',a)-Q(s,a))$

Q-learning算法的伪代码如下：

▶ Q-learning算法

```
Initialize Q(s,a),∀s∈S,a∈A(s),Q(terminal-state,·)=0
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q (e.g., ε-greedy)
        Take action a, observe r, s'
        Q(s,a)←Q(s,a)+α(r+γ*argmax_aQ(s',a)-Q(s,a))
        s←s'
    until s is terminal
```

Q-learning通过策略给定最优（不进行贪心了！）的第二步来估计当前最优的第一步，如果第二步的效果很好，那么Q值就会被更新为更大的值，反之则会被更新为更小的值

策略梯度定理

对于片段性问题，定义策略π的性能为$J(θ)=v_{πθ}(S_0)$ 在这里，θ是策略π的参数，v是状态价值函数，S_0是初始状态。θ的参数往往通过神经网络进行设定。 则策略梯度定理告诉我们，策略π的梯度为：

$∇_θJ(θ)=E_π[∑_a∇_θπ(a_t|s_t)Q_π(s_t,a_t)]$ 其中，$Q_π(s_t,a_t)$是在状态s_t下采取动作a_t的价值，$∇_θπ(a_t|s_t)$是在状态s_t下采取动作a_t的概率的梯度。

蒙特卡洛策略梯度算法

将策略梯度定理中的公式转化为基于采样的、期望相等的近似公式 则状态S_t的价值变化率为：$\frac{G_t}{π(a_t|s_t,θ)}*∇_θπ(a_t|s_t,θ)$ 其中G是历史回报，π是策略，θ是策略的参数，a_t是在状态S_t下采取的动作。 状态S_t的价值变化率乘以衰减率γ，则可以生成一个序列，得到S_0的价值变化率：$∇J(θ)=E_π[∑_tγ^t\frac{G_t}{π(a_t|s_t,θ)}*∇_θπ(a_t|s_t,θ)]$ 则可以得到蒙特卡洛策略梯度算法的伪代码：

▶ 蒙特卡洛策略梯度算法

```
Initialize θ
Repeat (for each episode):
    Generate an episode S_0,A_0,R_1,...,S_T-1,A_T-1,R_T following π(·|·,θ)
    For each step of episode t=0,1,...,T-1:
        G←∑_(k=t+1)^T*γ^(k-t-1)*R_k
        θ←θ+αγ^tG∇_θlnπ(A_t|S_t,θ)
```

蒙特卡洛策略梯度算法 with baseline

在蒙特卡洛策略梯度算法中，我们可以**引入一个baseline，即$b(S_t)$，来减少方差，使得算法更加稳定。** 在策略梯度定理中减去baseline之后，原定理变为：$∇_θJ(θ)=E_π[∑_a∇_θπ(a_t|s_t)(Q_π(s_t,a_t)-b(s_t))]$ 一般令$b(S_t)=v(S_t,w)$，其中**另一个神经网络W的参数为w**，v是状态价值函数，S_t是状态。 则蒙特卡洛策略梯度算法 with baseline的伪代码如下：

▶ 蒙特卡洛策略梯度算法*

```
Initialize θ,w
Repeat (for each episode):
    Generate an episode S_0,A_0,R_1,...,S_T-1,A_T-1,R_T following π(·|·,θ)
    For each step of episode t=0,1,...,T-1:
        G←∑_(k=t+1)^T*γ^(k-t-1)*R_k
        δ←G-v(S_t,w)
        θ←θ+αγ^t*δ*∇_θlnπ(A_t|S_t,θ)
        w←w+αγ^t*δ*∇_wv(S_t,w)
```

Actor-Critic算法

将蒙特卡洛策略梯度算法 with baseline中的蒙特卡洛算法替换为时序差分算法，即将全部return替换为单步return，可以获得单步Actor-Critic算法。 其中：

- Actor是按照策略梯度算法的策略$π(A_t|S_t,θ)$决定下一步的行动
- Critic是利用状态价值函数$v(S_t,w)$来评价策略的好坏

Actor-Critic算法的伪代码如下：

▶ Actor-Critic算法

```
Initialize θ,w,I=1
Repeat (for each episode):
    Generate an episode S_0,A_0,R_1,...,S_T-1,A_T-1,R_T following π(·|·,θ)
    while S_t is not terminal:
        δ←R_t+γ*v(S_t+1,w)-v(S_t,w)
        θ←θ+α^I*δ*∇_θlnπ(A_t|S_t,θ)
        w←w+β*δ*∇_wv(S_t,w)
        I←γ*I
        S_t←S_t+1
```

Actor-Critic相比于蒙特卡洛平均梯度算法with baseline的改进是，w网络不仅仅用于计算baseline，还用于计算下一步的状态价值函数,分化出了一个完整的Critic网络。

深度学习基础

线性回归

线性回归在解决回归问题时，利用最小二乘法，可以求解出最优的参数，使得预测值与真实值的差值最小，大概可以理解为函数： **$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b$** 其中，$w_1,w_2,...,w_n$是参数，$b$是偏置，$x_1,x_2,...,x_n$是特征，$y$是预测值 **偏置的意义是，可以使输出值上下浮动，以更好地适应输入值** 但是，**当样本量很少时，最小二乘解所需的A^TA的逆矩阵可能不存在，这时候就需要用到梯度下降法**，梯度下降法的伪代码如下:

▶ 梯度下降法

```
function gradient_descent(θ)
    initialize w arbitrarily
    η = learning rate
    while True
        w = w - η*∇wL(w)
        if ∇wL(w) < θ
            break
    return w
```

但是在梯度下降之后，我们获得的参数往往会拥有较高的自由度，得到的曲线可能不够平滑，这时候就需要用到正则化，**正则化的目的是为了防止过拟合**，正则化的方法有很多，比如L1正则化和L2正则化，反别对应着Lasso回归和Ridge回归。

Lasso回归

Lasso回归的目的是为了防止过拟合，它指的是：**$λΣ_{i=1}^n|w_i|$** 经过Lasso回归处理后的数据，会使**更多的参数变为0。**

Ridge回归

Ridge回归的目的是为了防止过拟合，它指的是：**$λΣ_{i=1}^n(w_i)^2$** 经过Ridge回归处理后的数据，会使**参数的值变小**。

最大似然估计

利用已知的样本结果信息，反推最具有可能导致这些样本出现的参数值，这就是最大似然估计。

逻辑回归

逻辑回归是用于分类问题的，它的目的是为了将数据分为两类，比如说，我们可以将数据分为正样本和负样本。

逻辑回归的损失函数

熵(Entropy)是无损编码事件信息中的最小平均编码长度，代表惊异程度，也就是说，熵越大，惊异程度越大，也就是说，熵越大，不确定性越大。熵的计算公式为： **$H(X) = -Σ_{i=1}^n p_i*log(p_i)$** 在逻辑回归中，我们使用的是交叉熵损失函数(Cross Entropy)，用于度量两个概率分布间的差异性信息，它的计算公式为：**$H(p,q) = -Σ_{i=1}^n p_i*log(q_i)$** 更一般的情况下，我们可以使用KL散度(Kullback-Leibler Divergence)来计算两个概率分布之间的差异性信息，它的计算公式为： **$D_{KL}(p||q) = Σ_{i=1}^n p_i*log(\frac{p_i}{q_i})$**

则PQ的KL散度=PQ的交叉熵-P的熵 当KL散度=0时，PQ的交叉熵=P的熵，也就是说，当两个概率分布完全一致时，KL散度为0。 对于线性模型，样本数据的分布P已经确定，即P=1，所以KL散度=PQ的交叉熵=$-Σ_{i}logq_i$

二项逻辑回归的激活函数

二项逻辑回归的回归函数是sigmoid函数，它的计算公式为： **$f(x) = \frac{1}{1+e^{-x}}$** 根据sigmoid的特点，可以得到在神经网络调节参数w时的推导过程：

设真实的分布为y_{real},预测的分布为y_{pred},则有： $H(p,q) = -Σ_{i=1}^n p_i*log(q_i)$ $=Σ_{i=1}^n[y_{real_i}*log(y_{pred_i}) + (1-y_{real_i})*log(1-y_{pred_i})]$ 我们要求损失函数最小，所以损失函数应当向着最小的方向梯度下降，于是我们要对损失函数求导，即： $\frac{\partial H(p,q)}{\partial y_{pred_i}} = Σ_{n}-(y_{real}-y_{pred})x_i^n$

于是，参数调节的方式是： $w_i = w_i - η*Σ_{n}-(y_{real}-y_{pred})x_i^n$

多项逻辑回归的激活函数

多项逻辑回归的回归函数是softmax函数，它的计算公式为： **$f(x) = \frac{e^x}{Σ_{i=1}^n e^x}$** softmax回归的导数推导过程在此省略，可以参考这篇文章。

补充：反向传播

在神经网络构建中，利用sigmoid函数和MSE均方误差，它的反向传播过程一般是这么进行的： *$W = W + η(y_{real}-y_{pred})y_{pred}(1-y_{pred})X$* 其中，W是参数矩阵，X是输入矩阵，y_pred是输出张量，y_real是真实张量。

将反向传播的算法一般化，则有： *$W = W + η\frac{\partial{L}}{\partial{W}}$* 其中，L是损失函数，W是参数矩阵。 这就是反向传播的基本过程。

当然，在$Y=WX+b$，设激活函数为$A=f(Y)$，损失函数为$L(A,\hat{Y})$(而$\hat{Y}$是常数)时，我们可以使用链式法则来推导上述的式子： **$\frac{\partial{A}}{\partial{Y}}=f'(Y)$ $\frac{\partial{L}}{\partial{A}}=L'(A)$ $\frac{\partial{Y}}{\partial{W}}=X$ $\frac{\partial{Y}}{\partial(b)}=1$** 所以，*$\frac{\partial{L}}{\partial{Y}}=\frac{\partial{L}}{\partial{A}}*\frac{\partial{A}}{\partial{Y}}=L'(A)f'(Y)$ **$\frac{\partial{L}}{\partial{W}}=\frac{\partial{L}}{\partial{Y}}\frac{\partial{Y}}{\partial{W}}=L'(A)f'(Y)X$ **$\frac{\partial{L}}{\partial{b}}=\frac{\partial{L}}{\partial{Y}}\frac{\partial{Y}}{\partial{b}}=L'(A)f'(Y)1$* 这样，我们就知道了W和b反向传播的更新公式。

聚类问题

K最近邻算法 KNN

在解决聚类问题时，传统的方法是使用K最近邻算法，即：**对于一个新的样本，我们找到与它最近的K个样本，然后将它归为这K个样本中出现次数最多的那一类。** KNN拥有比较明显的缺点，那就是**计算量大**，因为我们需要计算**每个**样本与其他样本的距离，并且根据最相近的K个样本来判断它的类别。 KNN算法的伪代码如下：

▶ KNN算法

```
function KNN(x)
    for i = 1 to n
        d[i] = distance(x, x[i])
    sort d[1...n]
    return majority_vote(d[1...k])
```

K均值算法 K-means

保存每个节点的坐标并进行计算要求非常大的计算量，所以诞生了K-means算法，它的思想是：将所有的点分为K组，每组的中心点作为该组的代表，然后**将所有的点分配到最近的中心点所在的组中，然后重新计算每组的中心点，重复这个过程，直到中心点不再发生变化。** K-means算法的伪代码如下：

▶ K-means算法

```
function K-means(x)
    randomly initialize k cluster centers
    while True
        for i = 1 to n
            c[i] = argmin_c distance(x[i], c)
        for c = 1 to k
            c = mean(x[i] where c[i] = c)
        if no change
            break
    return c
```

但是，**聚类算法对初始化种子的选择非常敏感，同时有些种子可能会导致算法收敛的很慢**，为了达到很好的效果，K-means算法一般会要求：

- 初始化种子的选择要尽可能的分散

- 尝试不同的初始化种子

激活函数

Sigmoid函数和Softmax函数的具体讨论已经在上面的内容中进行了，这里将介绍其它种类的激活函数。

ReLU函数

ReLU函数的计算公式为： **$f(x) = max(0, x)$** ReLU函数的优点是：

- **计算速度快，不容易出现梯度消失**

但是它的缺点是：

- 当神经元的输入为负数时，它的输出将永远为0，这样就会导致神经元的参数永远无法更新。

由于ReLU函数的特点，所以ReLU函数经常被用于作为隐藏层的激活函数，而不是作为输出层的激活函数。

Leaky ReLU函数

Leaky ReLU函数的计算公式为： **$f(x) = max(αx, x)$** 其中，α是一个很小的常数，在网络的训练过程中，α的值是可以被学习的。

Tanh函数

Tanh函数的计算公式为： **$f(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}}$** Tanh函数的优点是：

- **输出值在-1到1之间，所以通常可以用来做回归任务**

损失函数

损失函数可以用来量化预测的输出和真实输出的误差，可以设定优化神经网络参数的目标（是损失函数尽可能地小），并且可以用来评估模型的好坏。

均方误差 MSE

均方误差的计算公式为： **$MSE = \frac{1}{n}\sum_{i=1}^n(y_{real}-y_{pred})^2$** 均方误差对应的是L2范数，它的特点是使得误差的平方和最小化，这样可以使得误差的值更加集中在0附近，MSE对应着嵴回归和欧拉距离。

平均绝对误差 MAE

平均绝对误差的计算公式为： **$MAE = \frac{1}{n}\sum_{i=1}^n|y_{real}-y_{pred}|$** 平均绝对误差对应的是L1范数，它的特点是使得误差的绝对值和最小化，这样可以使得误差的值更加集中在0附近，MAE对应着L1范数和曼哈顿距离。

优化（超多优化方法！）

在多层感知机中，进行不断的反向传播过程中，很有可能出现梯度消失的情况，这样就会导致神经网络的参数无法更新。梯度消失问题的一个解决方法是使用ReLU函数进行激活函数。 当然，还有一些其他的优化方法，比如：

- 随机梯度下降 每次更新不用所有的样本，而是随机选择一个batch的样本进行更新。
- 自适应学习率 在训练过程中，随着训练的进行，学习率会逐渐减小，这样可以使得模型在训练的后期更加稳定。
- 动量 动量的思想是：在更新参数的时候，不仅仅考虑当前的梯度，还要考虑之前的梯度，这样可以使得梯度的变化更加平滑。
- 正则化 正则化的思想是：在损失函数中加入一些惩罚项，使得模型的参数更加稳定，从而避免过拟合。
- 数据增强 数据增强的思想是：在训练过程中，对原始的数据进行一些变换，比如旋转、翻转、裁剪等，这样可以使得模型对于数据的变化更加鲁棒。
- 权重衰减 权重衰减的思想是：在损失函数中加入权重的L1范数，即$L_{total}=L+||W||$这样可以使得模型的参数更加稳定。
- Dropout Dropout的思想是：在训练过程中，随机的将一些神经元的输出置为0，这样可以使得模型的参数更加稳定。

卷积神经网络 CNN

卷积神经网络需要的参数有：

- 卷积核filter/kernel：卷积核的大小和个数
- 步长stride：卷积核在输入上滑动的步长
- 填充padding：在输入的边缘填充0的个数

在进行卷积操作时，每一个卷积核都要在input的一个channel上滑动一次，最后生成的许多个卷积后的图形相加才得到output的一层，也就是说，卷积有 $filter_shape=filter_height*filter_width*in_channels*out_channels$ 其中，in_channels是输入的通道数，out_channels是输出的通道数。

感受野

感受野的概念是指：输出的某一层的神经元对于输入的某一层的神经元的影响区域。感受野的计算公式为：
$receptive_field = receptive_field_preview + (filtersize-1)*\prod_{i=1}^{k-1}stride_size$

对于神经网络来说，感受野越大，那么它对于输入的信息的获取就越多，这样就可以使得模型的表达能力更强。所以引入了空洞卷积的概念。 空洞卷积新增了一个卷积网络参数，即空洞率dilation，空洞卷积的取样区域是分离的，这样可以有效的增大感受野。

池化层

池化层的作用是：对输入的特征图进行降维，这样可以减少参数的数量，从而减少计算量，同时可以防止过拟合。 朴素的池化算法包括：最大池化、平均池化、随机池化等。池化算法和卷积算法在原理上是类似的，都是在一个filter_size中选择合适的stride和padding，然后在输入上滑动，最后得到输出。

空间金字塔池化

空间金字塔池化的思想是：在不同的尺度下进行池化，然后将不同尺度下的池化结果进行拼接，这样可以使得模型对于不同尺度的信息都能够获取到。 也就是说，空间金字塔池化可以获得很多个不同尺度的特征图，然后再将这些特征图分别进入全连接层，最后将这些全连接层的输出进行拼接，这样就可以得到最终的输出。

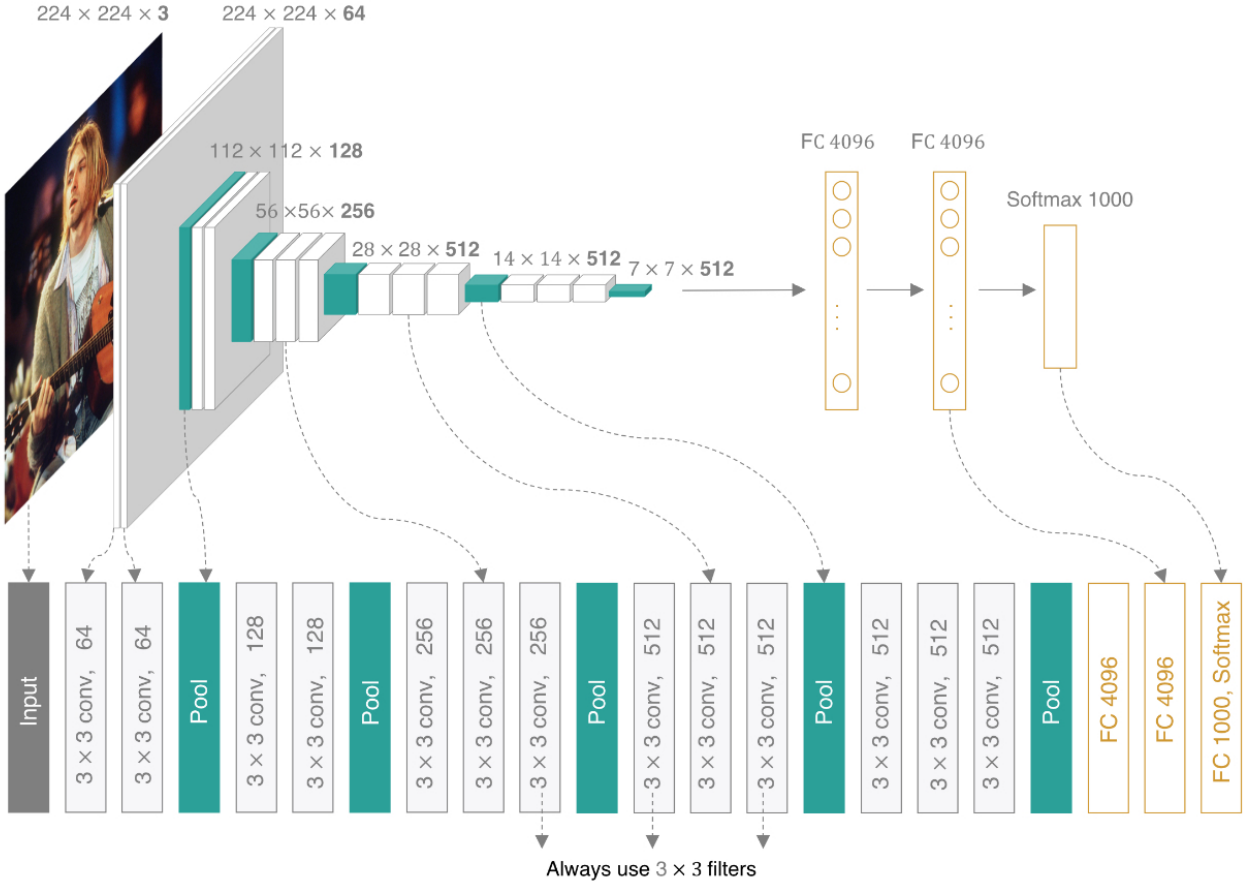
现代卷积神经网络

VGG16

VGG16的特点是：**使用小卷积核代替大卷积核**。它的好处是

- 减少计算量：小卷积核的参数数量更少，对于获取相同大小的感受野来说，虽然小卷积核会导致更多的卷积层，但是总体的参数数量仍然更少。
- 增加非线性：小卷积核的卷积层更多，这样可以使得模型的表达能力更强。
- **BottleNeck：使用1*1的卷积核进行降维，降维之后的数据通过3*3的卷积核进行卷积，最后再通过1*1的卷积核进行升维，这样参数量会非常大程度的减小。**（在ResNet中，便运用到了BottleNeck的结构）

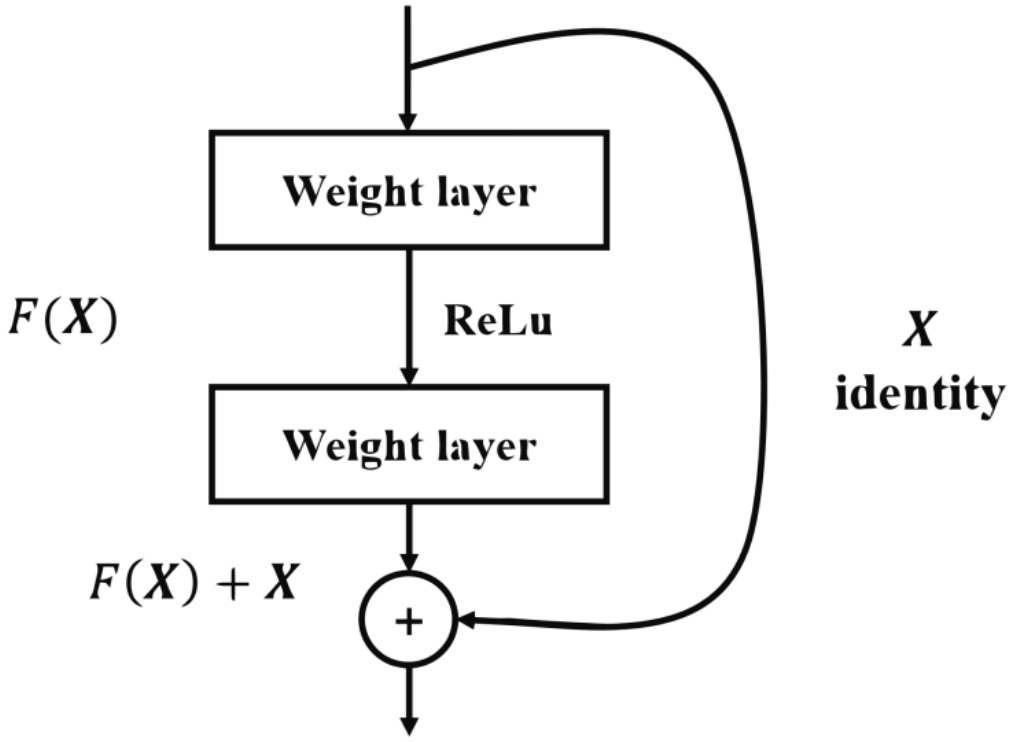
VGG16的网络结构如下图所示：



ResNet

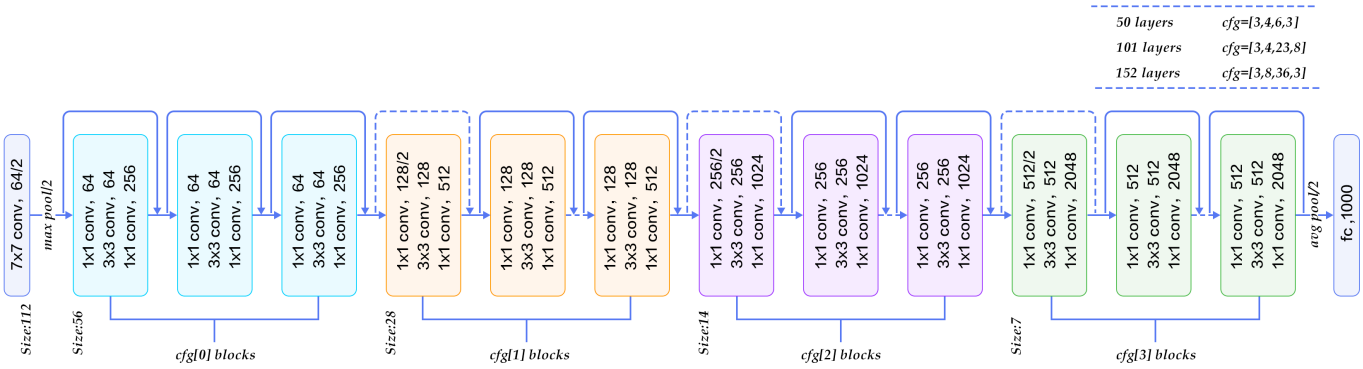
ResNet的特点是：**使用残差结构**。深层神经网络往往会出现梯度消失的问题，残差结构使得反向传播的过程中出现了一条捷径，这样可以使得梯度的传播更加顺畅，有效缓解梯度消失的问题。 ResNet网络一般由几层卷积

层和激活函数可以构成一个残差块，多个残差块可以构成一个残差网络。残差块的结构如下图所示：



当然，残差块的设计还可以进一步的精进，比如**在残差块中加入BN层。** BN层的基本意思就是把每个通过的位置的数值正则化到均值为0，方差为1的分布上，这样可以使得模型的训练更加稳定。**正则化不会影响模型的表达能力**的原因是，起作用的不是数值的绝对数值，而是数值的相对大小，所以正则化之后，模型的表达能力不会发生改变。

ResNet的网络结构如下图所示：



SqueezeNet

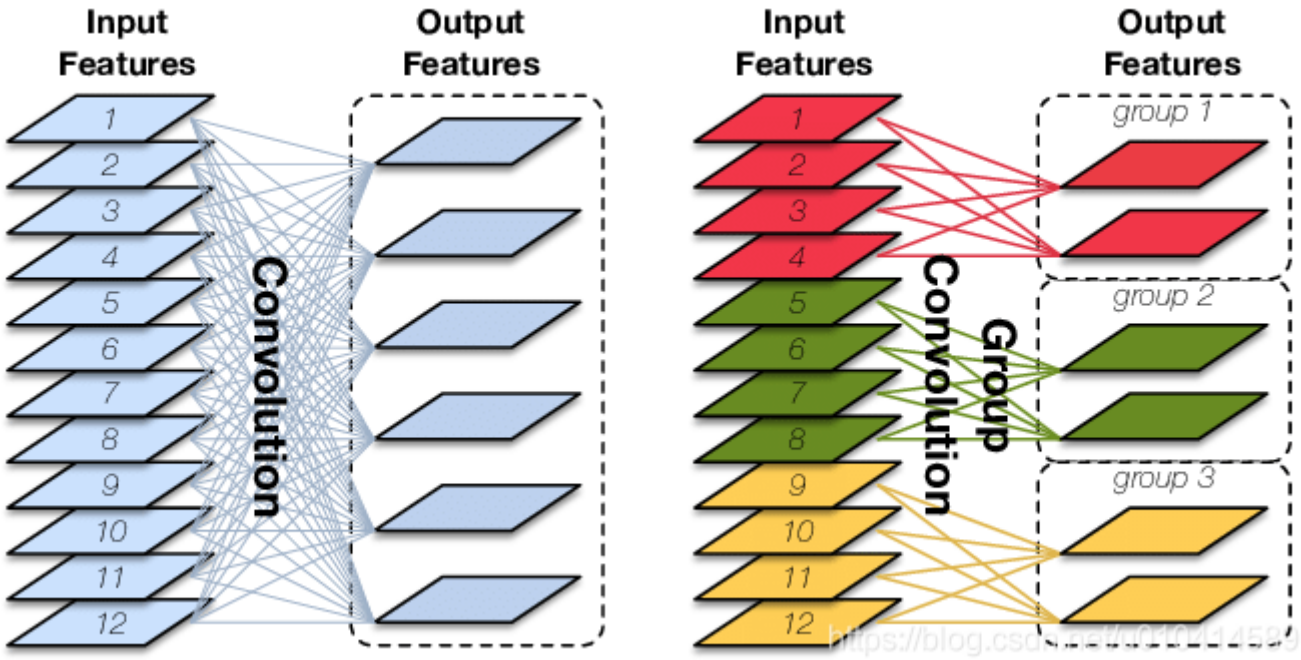
SqueezeNet的特点是：**使用1*1的卷积核代替3*3的卷积核。** SqueezeNet的网络结构如下图所示：
SqueezeNet 在倒数第二步中，通过1*1的卷积核和3*3的卷积核进行卷积，并将得到的两个64通道的卷积叠加到一起，获得了128通道的卷积并与input相加得到output。

MobileNet

MobileNet的特点是：**使用深度可分离卷积。** 深度可分离卷积包括两个步骤，首先是使用平凡的卷积核逐一对应输入channel进行卷积，最后使用1*1的卷积核进行卷积合并。这样可以使得参数数量大大减少，从而减少计算量。 MobileNet的网络结构如下图所示： MobileNet

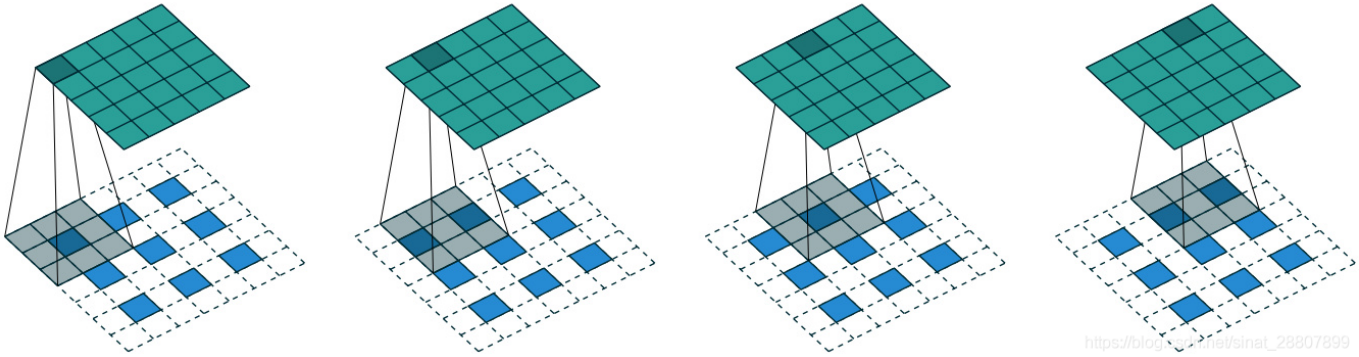
ShuffleNet

ShuffleNet的特点是：**使用分组卷积。** 和遗传算法有异曲同工之妙，ShuffleNet使用了分组卷积，在卷积之后进行了打乱操作，再将不同组的信息合并，这样可以使得不同组的信息进行交流，从而使得模型的表达能力更强。 ShuffleNet的网络结构如下图所示：



反卷积

反卷积是在原特征图上补padding或空洞填充，然后再进行卷积，这样可以使得特征图的尺寸变大，从而使得模型的感受野变大。 用图像来直观理解反卷积，如下图所示：



卷积神经网络的应用

目标检测

在评估目标检测的效果时，引入了交并比的概念，即目标检测的图像和真实图像分为四个部分：

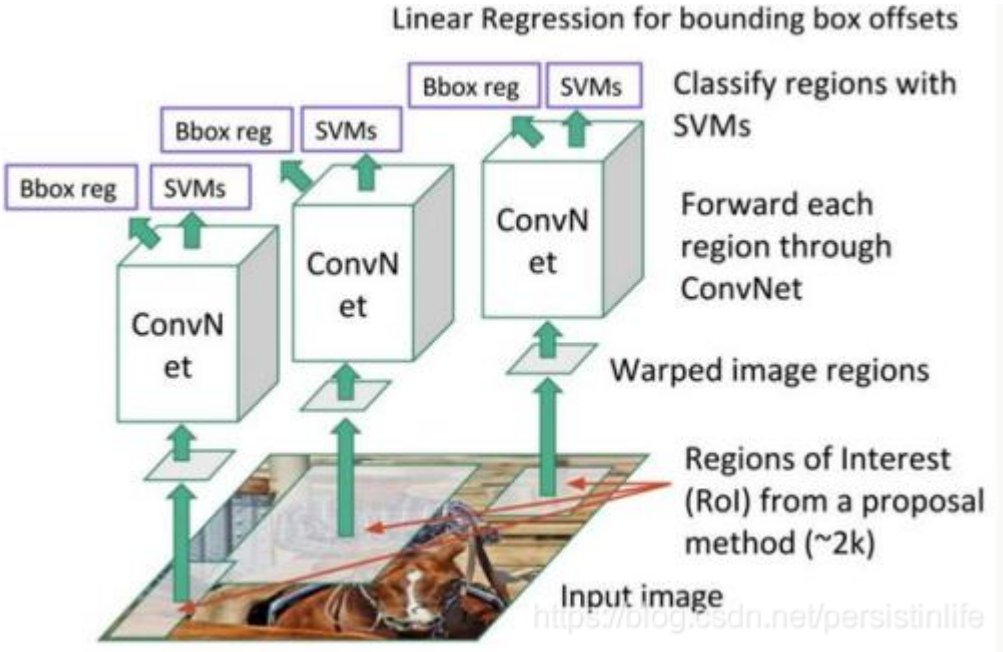
- 真阳性（True Positive，TP）：检测到的目标和真实目标的交集
- 假阳性（False Positive，FP）：检测到的目标和真实目标的差集
- 假阴性（False Negative，FN）：未检测到的目标和真实目标的差集
- 真阴性（True Negative，TN）：未检测到的目标和真实目标的交集

而目标检测的平均精确度AP为： **$Precision=\frac{TP}{TP+FP}$ $Recall=\frac{TP}{TP+FN}$** 当Precision和Recall的值同时最大，即乘积最大时，目标检测的效果就越好。

R-CNN

R-CNN总共经过了4个步骤：

1. 使用选择性搜索（Selective Search）算法从图像中提取出若干个候选区域
2. 将所有的候选区域进行缩放，使得所有的候选区域的尺寸都一致
3. 使用VGG网络对所有的候选区域进行特征提取
4. 通过回归获取边界框的位置 R-CNN网络还使用了非极大值抑制（Non-Maximum Suppression，NMS）算法，将重叠的边界框进行合并，保留最佳的边界框。 下面是R-CNN的网络结构：

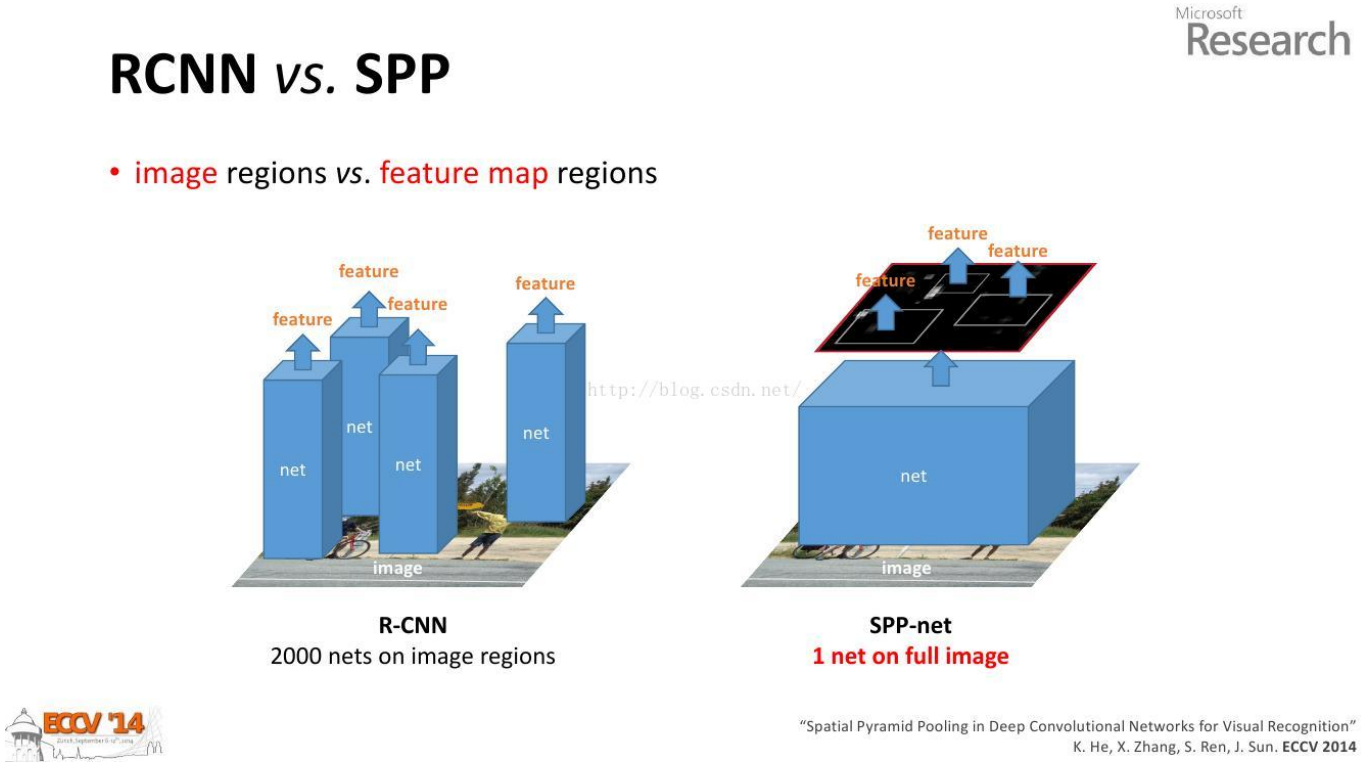


然而R-CNN具有以下四个局限性： 1.选择性搜索的速度慢 2.候选区域的尺寸在算法中会变化 3.每个区域被单独输入到VGG中，时间消耗长 4.非端到端的训练方式

SPPNet

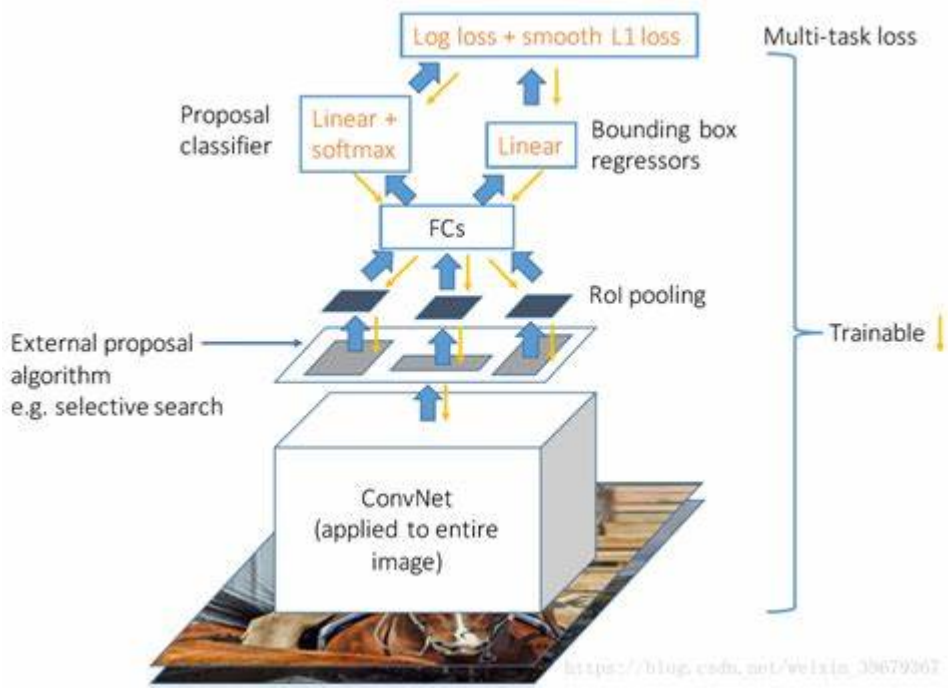
R-CNN拥有局限性**候选区域的尺寸在算法中会变化、每个区域被单独输入到VGG中，时间消耗长。**SPPNet通过引入空间金字塔池化（Spatial Pyramid Pooling，SPP）层解决了这两个问题。SPP层的作用是将不同尺寸的特征图池化到固定尺寸的特征图上，这样可以使得不同尺寸的特征图都可以输入到特征提取器CNN中，从而使得

候选区域的尺寸不会变化，同时也可以减少计算量。 对比SPPNet和R-CNN可以发现：



Fast R-CNN

Fast R-CNN提出了感兴趣区域池化层，简化了空间金字塔池化，同时使用了近似的端到端训练，**分类器、边界框回归器和特征提取器共享卷积层**，从而使得训练速度更快。而在SPP中，得到的特征图是逐个输入到分类器和边界框回归器中，这样会导致训练速度变慢。 下面是Fast R-CNN的网络结构：



Faster R-CNN

Faster R-CNN的特点是引入了**区域生成网络（Region Proposal Network，RPN）**，RPN的作用是生成候选区域，这样就不需要使用选择性搜索算法了，从而使得训练速度更快。 下面是Faster R-CNN的网络结构：

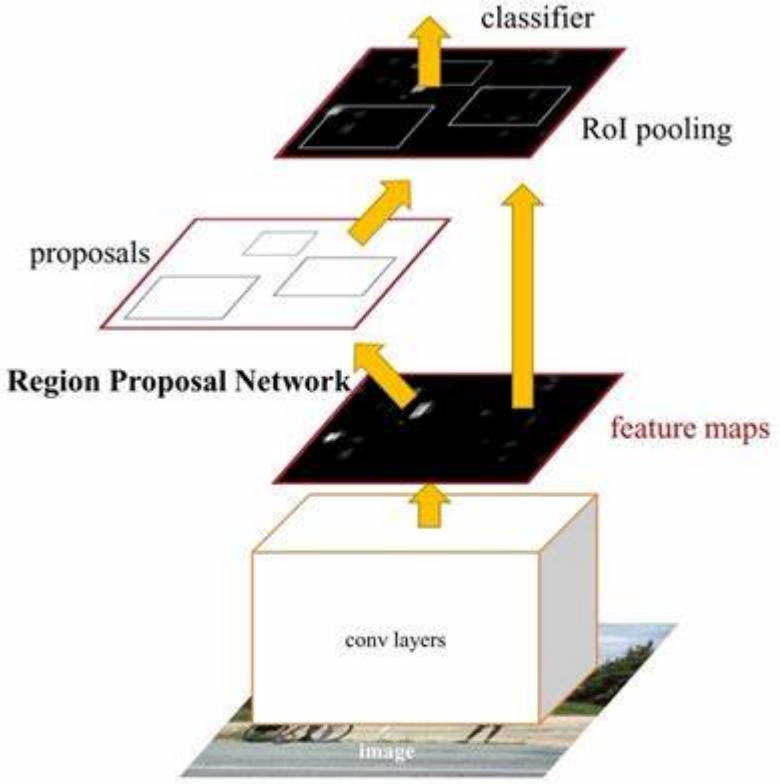


Figure 2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the 'attention' of this unified network.

YOLO

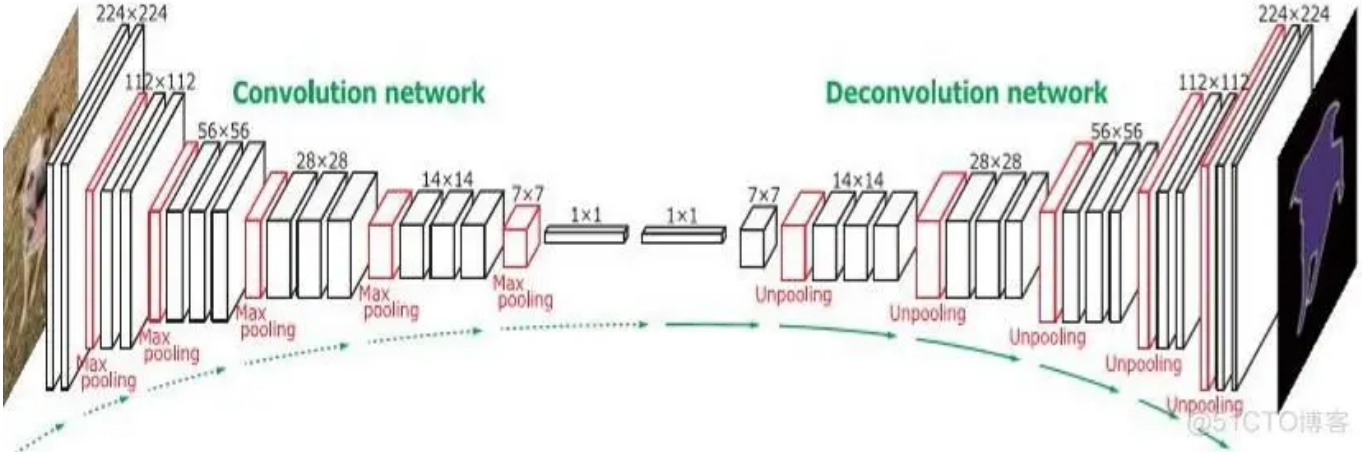
在YOLO算法中，一个图像被划分成了一个7*7的网格，每个网格都会预测得到物体的置信度。YOLO的问题是：**算法对小物体的检测准确率过低。** 在YOLO V2中，神经网络会给网格单元预定义多个提议区域候选并执行YOLO算法，同时将分辨率从7*7提高到了13*13。

总结R-CNN算法和YOLO算法，谷歌提出了新的SSD算法，SSD算法采用了**不同的特征区域划分**（又有8*8，又有4*4），并且通过不同的卷积层获取多个feature maps，提高了检测的准确性。

图像分割

SegNet

图像分割使用了全卷积神经网络FCN，仅使用卷积和池化操作，所以可以接受任意大小的输入图像。 下面给出使用FCN解决图像分割问题的网络结构；



但是，当神经网络过深时，**可能会失去一些低级别的特征**，所以在实际解决问题时，**使用了跳跃连接Skip Connection的思想**，即下图的蓝色连线：

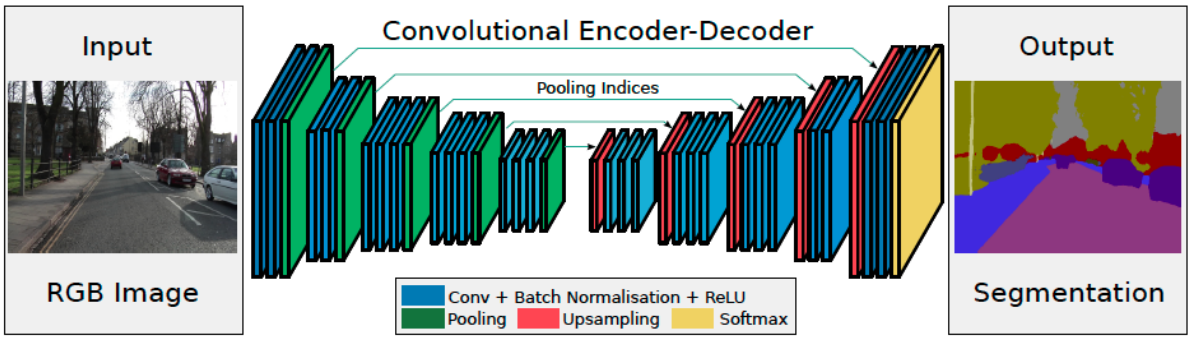


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

当然，先使用R-CNN再通过神经网络来分割图像，也不是不可以，即Mask R-CNN。

Tips：在图像分割时，为了避免在边界上丢失信息，我们可以使用损失加权的技巧，即：

- 边缘加权：增加边缘的权重，使损失对边缘更加敏感
- 平衡加权：根据小物体的大小增加权重，平衡大物体和小物体之间的权重

姿态估计

姿态估计问题有两种解决思路：

- 自顶向下：对每个人进行检测，得到边框，再使用姿态估计算法（缺点：可能无法检测到人、时间问题）
- 自底向上：首先检测所有关键点，再分配给不同的人（缺点：难以将关键点分配给不同的人）

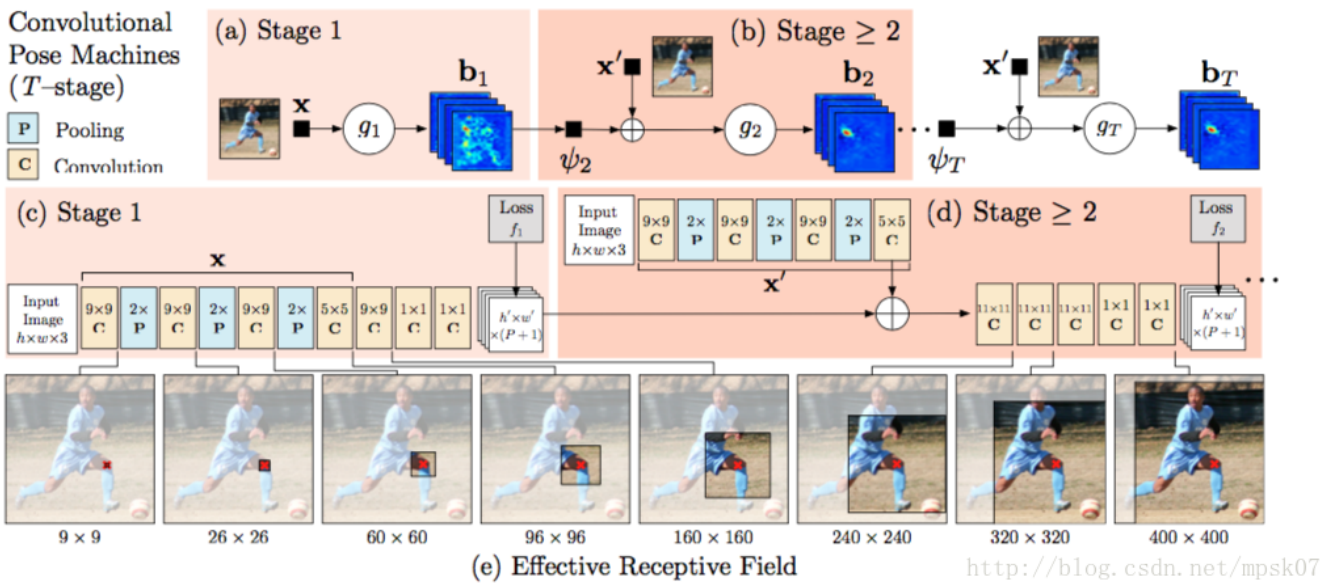
CPM

CPM算法是一种自顶向下的姿态估计算法，其核心思想是**使用多个卷积神经网络来预测关键点**，并且使用**多尺度的输入**，从而提高了算法的准确性。CPM算法的步骤是：

1. 使用目标检测算法检测出人体的边框
2. 将人的图像输入到VGG中，得到特征图

3. 将特征图输入到CNN中，得到关键点的预测结果
4. 将关键点的预测结果输入到CNN中，得到更加精确的关键点预测结果

用于姿态估计的CPM网络结构如下：



上述多阶段的方法拥有更大的感受野，同时可以精细的调整关键点的估计。 注意：CPM算法同时可以使用残差网络ResNet来提高算法的准确性。

OpenPose

OpenPose算法是一种自底向上的姿态估计算法，其核心思想是**使用多个卷积神经网络来预测关键点**，并且使用**多尺度的输入**，从而提高了算法的准确性。OpenPose算法的步骤是：

1. 将人的图像输入到VGG中，得到特征图
2. 将特征图输入到CNN中，得到关键点的预测结果
3. 将关键点的预测结果输入到CNN中，得到更加精确的关键点预测结果
4. 通过CNN网络，将不同的关键点的组合分配给不同的人

人脸识别

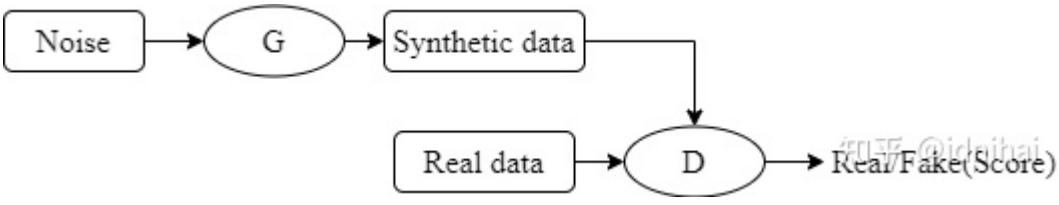
人脸识别包括closed-set和open-set两种情况：

- closed-set：简单的分类问题，即给定一个人脸，判断这个人脸属于哪个人
- open-set：给定一个人脸，判断这个人脸属于哪个人，如果不属于任何一个人，则判断为unknown

生成对抗网络 GAN

Vanilla GAN

Vanilla GAN的网络结构如下：



其中G是生成器，D是

判别器，G和D都是卷积神经网络，G的输入是随机噪声，D的输入是真实图像和G生成的图像。

GAN的数学原理

G的任务是：**生成和真实图像尽可能相似的图像**，D的任务是：**判断输入的图像是真实图像还是G生成的图像。**
于是，我们根据上述目标，可以设计出G和D的损失函数L_G和L_D：

- $L_G = -E[\log D(G(z))]$
- $L_D = -E[\log D(x)] - E[\log (1 - D(G(z)))]$ 其中z是随机噪声Noise，x是真实图像Real data。

那么，最优的G*和D*一定满足：

- $G* = \arg\min_G E[\log (1 - D(G(z)))]$（D(real)=1）
- $D* = \arg\max_D E[\log D(x)] + E[\log (1 - D(G(z)))]$

此时，对于函数$V(D,G)=E[\log D(x)] + E[\log (1 - D(G(z)))]$，D想要把V最大化为0，G想要把他最小化为$-\infty$，则： $V(D,G) = E[\log D(x)] + E[\log (1 - D(G(z)))]$ $= \int_{x} p_{data}(x) \log D(x) dx + \int_{z} p_{z}(z) \log (1 - D(G(z))) dz$ 此时我们只观察D，将G看作常数，则： $V(D,G) = \int_{x} p_{data}(x) \log D(x) dx + \int_{x} p_{g}(x) \log (1 - D(x)) dx$ 将函数对D求偏导，则： $\frac{\partial V}{\partial D} = \int_{x} p_{data}(x) \frac{1}{D(x)}+ p_{g}(x) \frac{1}{1 - D(x)} dx$ 于是： $V(G,D)' = \frac{p_{data}}{D(x)} - \frac{p_{g}}{1 - D(x)} =0$ 所以可以得到D的最优解为： $D^* = \frac{p_{data}(x)}{p_{data}(x) + p_{g}(x)}$

将D的最优解代入到V中，得到： $C(G)=E[\log \frac{p_{data}(x)}{p_{data}(x)+p_{g}(x)}] + E[\log \frac{p_{g}(x)}{p_{data}(x)+p_{g}(x)}]$ 此时再引入JS散度的概念，JS散度定义为： $JS(p||q) = \frac{1}{2}KL(p||m) + \frac{1}{2}KL(q||m)$ **与KL散度对比，JS散度中的p和q地位是对称的，而KL散度中的p是基准，q是待比较的分布。** 于是，我们可以得到： $C(G) = E[\log \frac{p_{data}(x)}{p_{g}(x)}] + E[\log \frac{p_{g}(x)}{p_{data}(x)+p_{g}(x)}] + E[\log \frac{p_{g}(x)}{p_{data}(x)}] + E[\log \frac{p_{data}(x)}{p_{data}(x)+p_{g}(x)}]$ $= 2*JS(p_{data}||p_{g}) + 2\log \frac{1}{2}$ $= 2*JS(p_{data}||p_{g}) - \log 4$

关于GAN网络基于Pytorch的实现，可以参考：

- Pytorch实现GAN

DC-GAN

DC-GAN的技巧是：发挥了卷积神经网络的优势，将生成器和判别器都改成了卷积神经网络，从而提高了GAN的效果。 DC-GAN的网络结构如下：

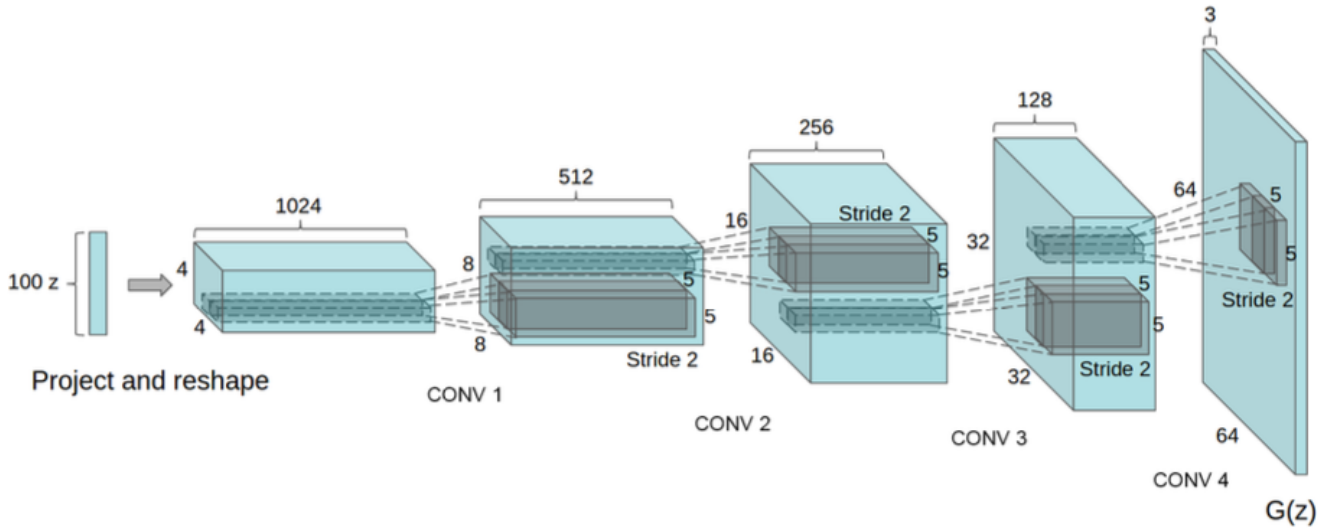
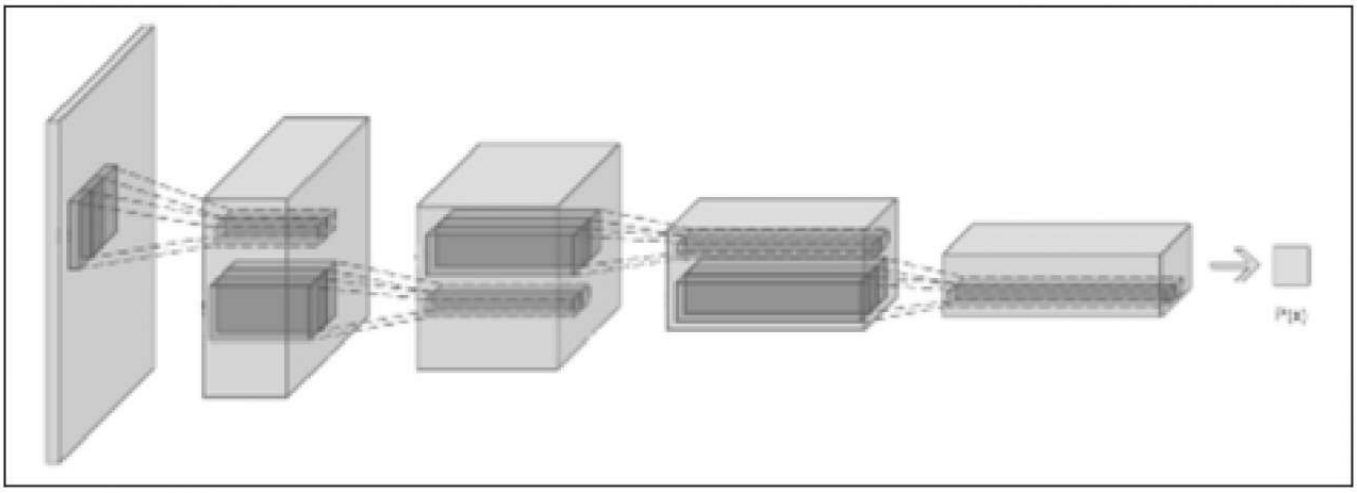
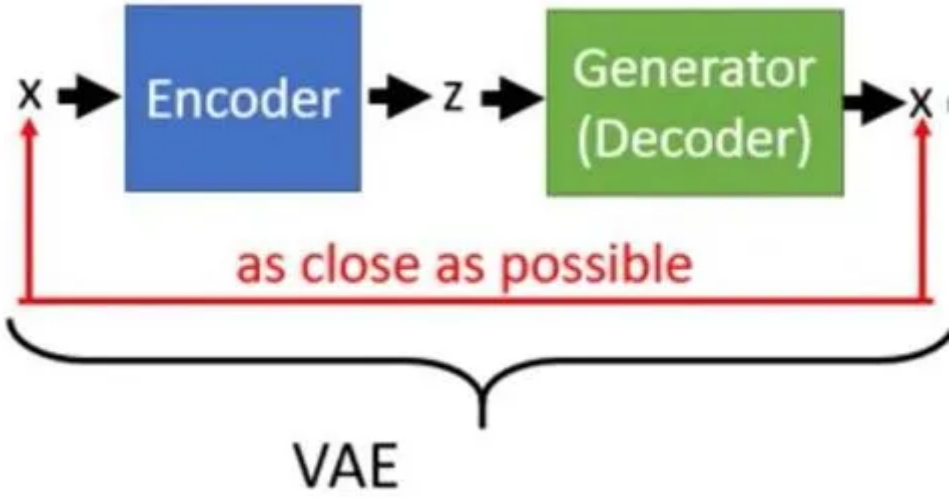


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.



VAE



VAE的网络结构如下： VAE给了我们一个新的思路：我们不仅可以将噪音转化为图像，还可以将图像转化为噪音（Encoder） 这又导致了GAN出现了井喷似的发展。

多种GAN网络

- Conditional GAN
- BiGAN BiGAN利用了联合分布的思想，对比的是分布$p(\hat{X},Z)$和$p(X,\hat{Z})$的相似度，其中X是真实图像，Z是噪声，$\hat{X}$是G生成的图像，$\hat{Z}$是D生成的噪声。
- coGAN
- CycleGAN CycleGAN将解码器和生成器组成了一个联合结构$G_{A2B}，又将生成器和解码器组成了一个联合结构G_{B2A}$，这样就可以实现图像的互相转换。 此时，判别器D对比的是分布$p(X_B,\hat{X_A})$和$p(\hat{X_B},X_A)$的相似度，其中X_A是真实图像A，X_B是真实图像B，$\hat{X_A}$是G_{B2A}生成的图像A，$\hat{X_B}$是G_{A2B}生成的图像B。

BiGAN、CycleGAN都可以很好地解决图像->图像的问题，但是对于文本->图像的问题，就得利用Conditional GAN了。

笔者后记

笔者写到这里，已经是2023年4月27日的凌晨2点了。前后一周的时间，1400行代码，我完整的浏览了一遍北京大学人工智能基础课程的前两个月的知识点。确实AI基础的调参、炼丹非常nt，作业的评分方式非常nt，课程的知识点设置非常混乱，老师讲的也有一些模糊不清，不过这些知识点本身是非常有趣且有意义的。 恰逢今天我们班的班长发了一条pyq，以感慨如果世界上没有内卷、没有恶心的评分，那些平日里拿着钩戟长锸的知识点，再看来也一定会变得平和一些。我认为这句话是非常正确的，因为知识点本身是非常有趣的，只是我们在学习的过程中，被一些不必要的东西所干扰了。和别人针锋相对、为0.01争上个头破血流，未免显得斤斤计较。大学生活应当是自由的、是快乐的，如果一个人真心认为自己的大学生活很无趣，那么无论拥有多好的成绩，从我的视角出发，我都认为他的大学生活是很失败的。 其实AI基础这门课也是这样，学习的过程是一个人的潜心修炼，有时候当然可以在瀑布中对着幽静的山谷大喊一声“煞笔ai基础，我测你的吗”，但是最终还是要回到水流之下——水帘洞之外只能出猴子猴孙，水帘洞之内才出齐天大圣。