人工智能 BAKAGO

周劲光 5140309495

目录

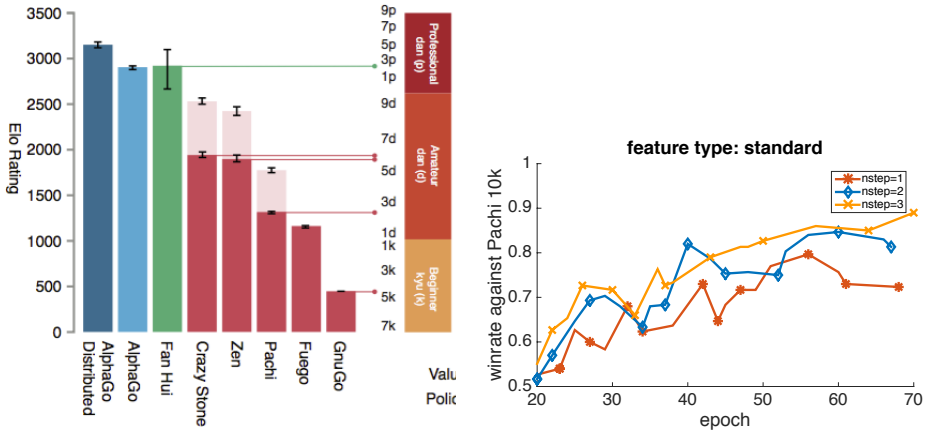
1 围棋背景

今年的人工智能可以说达到了一个大高潮，这股风潮由AlphaGo大战欧洲冠军以及胜过世界冠军引起。当今中国，无论是大学里的学生，商业场上的经理，还是路边的的士司机，都知道人工智能围棋大战人类高手这件事情，都饶有兴致，都能侃侃而谈。我们作为学计算机的学生，有必要跟上世界最前沿的脚步，对围棋这一智能进行了解和探索。

目前出现的围棋程序有（部分）：

- AlphaGo made by Google
- Master 出现在弈城围棋网，30s快棋打败柯洁，连胜亚洲棋手50余场，无败绩
- 刑天，出现在弈城围棋网上，国产神秘高手，被柯洁打败，但能胜大部分顶尖选手
- DeepZenGo 前身是传统围棋软件Zen，引入深度学习后强了很多
- DarkForest made by Yuandong Tian, Facebook
- Pachi 开源软件，业余级别

目前 DarkForest和Pachi 的源码已经公开，前者还公开了其粗调的model。



(a) 围棋软件水平　　(b) DarkForest对战Pachi 10k

图 1:

2 对围棋原理的理解

目前在围棋AI中，大量使用的方法是蒙特卡洛树搜索MCTS和最大置信区间算法UCT。对于传统AI程序，一般使用RAVE,AMAF的启发式方法和pattern的匹配方法来指导AI走棋。对于最新的人工智能AI，则大多采用由Deep learning训练出来的model来指导走棋。对于 tree policy，相应的model可用于减少MCTS树的宽度，对于default policy，model可用于减少MCTS树的深度。以下我们具体介绍。

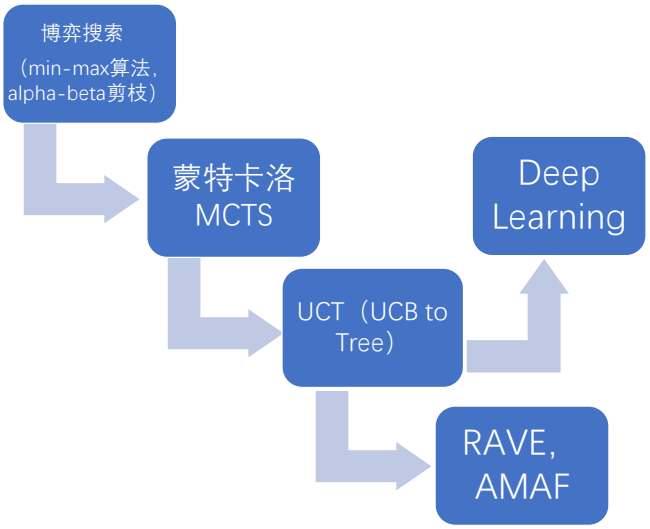


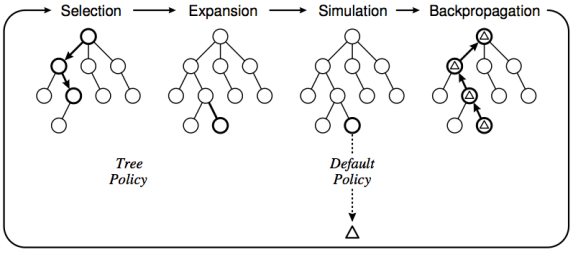
图 2: 围棋的基本方法

2.1 蒙特卡罗MCTS

课堂上我们学了博弈搜索，是专门用于棋类的搜索策略，Alpha-Beta搜索可以将其剪枝，但在围棋的搜索当中（五子棋也是），我们往往采用蒙特卡罗MCTS的搜索方法。这里用MCTS而不是固定的静态的规则的组合是因为：

- 围棋棋局情况相当复杂且规模庞大，单单采用一套固定的静态的规则去覆盖所有的情况必然会由于各种例外情况的出现而产生出各种不足。
- 围棋规则的描述不可能非常的准确。围棋中每个棋子的作用都差不多，而棋子的影响力更多的取决于它周围的环境，以及该棋子连接或距离较近的己方棋子和对方棋子都是决定其影响力的因素，这使得围棋中的很多情况甚至连职业棋手也很难精确的描述，而只能具体情况具体处理。
- 过度复杂的规则不仅会让机器处理起来效率低下，而且也会超出人们对系统实际运行效果的理解程度。而大量模拟往往易于实现。

因此，对于围棋，蒙特卡洛评估相比于静态评估方法具有明显的优势。它通过对当前局面下的每个的可选点进行大量的模拟来得出相应的胜负的统计特性，在简单情况下，胜率较高的点就可以认为是较好的点予以选择。



(a) MCTS四步法

算法 1：蒙特卡罗规划（MCTS approach）

function MCTSSEARCH(s_0)
以状态s_0创建根节点v_0;
while 尚未用完计算时长 **do**:
$v_l \leftarrow$TREEPOLICY(v_0);
$\Delta \leftarrow$DEFAULTPOLICY($s(v_l)$);
BACKUP(v_l, Δ);
end while
return a(BESTCHILD(v_0));

(b) MCTS伪代码

图 3:

不过，由于MCTS随机性很强，会影响做出正确评估的收敛速度，特别的，由于围棋的状态空间非常巨大，我们所能搜索的是状态空间的一个小部分，在没有知识的指导时树的扩展层数较少，不利于最优解的获取。所以人们想出了一个办法：能在随机策略中加入启发式知识，将加快收敛速度，更快找到更优的解。于是产生了UCT算法（Upper Confidence Bound Apply to Tree），即上限置信区间算法。

2.2 UCT算法

UCT算法来自于UCB，其背景是多臂老虎机模型。UCT算法是将UCB算法思想用于蒙特卡罗树的特定算法，其与只利用蒙特卡罗方法构建搜索树的方法的主要区别如下：

- 可落子点的选择不是随机的，而是根据UCB的信心上界值进行选择，如果可落子没有被访问，则其信心上限值为正无穷大，如果可落子点已经被访问过，则其信心上限索引值可以根据UCB算法给出的值来确定。在实际应用中，我们采用UCB给出的信心上限值，当我们需要在众多可下点中选取一个时，我们选择信心上限值最大的那一个。

- 当模拟结束后确定最终选择结果时，我们不再仅仅根据胜率做出判断，而是兼顾信心上限所给出的估计值。在实际应用中，选择方法也是多种多样的，一些策略中选择估计值最大的子节点为落子点，另外由于选择可落子点进行访问的过程已经兼顾了极小极大算法的思想，所以在另外一些策略中也经常直接选择那个被访问了最多次的可落子点。

UCT算法的伪代码如图4，该策略 综合考虑两个方面的因素：一是对尚未充分了解的节点的探索(exploration)，以尽早发现潜在价值高的节点并尽早排除潜在价值低的节点；二是对当前有较大希望的节点的利用(exploitation)，以尽可能地抓住机会创造对己方更有利的局势。其中**function** BESTCHILD中的公式，前一项代表到目前为止已经搜集到的知识的价值，后一项代表exploration，尚未充分探索过的节点需要继续探索的必要性。

算法 3：信心上限树算法（UCT）

function UCTSEARCH(s_0)
以状态s_0创建根节点v_0;
while 尚未用完计算时长 **do**:
$v_l \leftarrow$TREEPOLICY(v_0);
$\Delta \leftarrow$DEFAULTPOLICY(s(v_l));
BACKUP(v_l, Δ);
end while
return a(BESTCHILD($v_0, 0$));

function TREEPOLICY(v)
while 节点v不是终止节点 **do**:
if 节点v是可扩展的 **then**:
return EXPAND(v)
else:
$v \leftarrow$ BESTCHILD(v, c)
return v

function EXPAND(v)
选择行动$a \in A(state(v))$中尚未选择过的行动
向节点v添加子节点v'，使得$s(v')= f(s(v), a), a(v') = a$
return v'

function BESTCHILD(v, c)
return $argmax_{v' \in children\ of\ v} \left(\frac{Q(v')}{N(v')} + c\sqrt{\frac{2ln(N(v))}{N(v')}}\right)$

function DEFAULTPOLICY(s)
while s不是终止状态 **do**:
以等概率选择行动$a \in A(s)$
$s \leftarrow f(s, a)$
return 状态s的收益

function BACKUP(v, Δ)
while $v \neq NULL$ **do**:
$N(v) \leftarrow N(v) + 1$
$Q(v) \leftarrow Q(v) + \Delta$
$\Delta \leftarrow -\Delta$

图 4: UCT的伪代码

2.3 引入 Deep Learning 指导走棋

根据AlphaGo的论文，同时采用了Supervised Learning和Reinforcement Learning两种策略。DarkForestGo的论文中只提到了前者，但其开放的源代码里融入了后者的部分，不过效果如何，论文没有提，估计可能效果并没有那么明显。

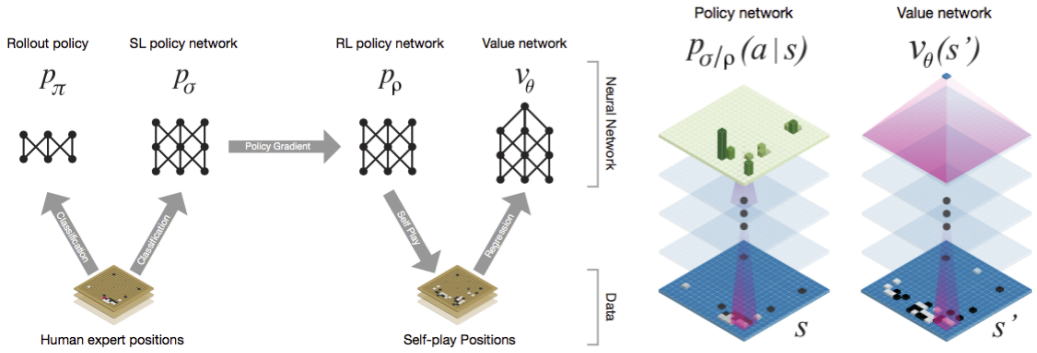
2.3.1 Supervised Learning

有监督学习Supervised Learning可以在三个地方使用。

- 一个是tree policy只要时间足够就可以继续扩展节点，扩展哪个节点呢？以前的方法大多是RAVE/AMAF+启发式，高级一点的用了pattern匹配，但现在我们可以用 Supervised Learning来告诉AI，扩展哪个节点是更有前途的。每一步深度，都有19 × 19个分叉，我们要走足够深，此时，Supervised Learning通过对人类棋局的学习尽可能模仿人类的下棋方法，然后告诉AI走哪个分支，哪些分支。

- 第二个是在default policy时，对于某状态，需要随机快速走子(fast rollout)来走完这盘棋。注意fast很重要，因为rollout的次数非常多，AlphaGo的一次快速走子是2微秒。没有Supervised Learning时，人们只能随机random进行快速走子，准确度不高。而Supervised Learning可以训练一个简单的走子网络，训练一些pattern让快速走子相比之前更有效。写到这里我一下子就想到徐雷老师的一句话，他说default policy时快速走子就像侦察兵，多派出一些侦察兵出去，看看情况如何。是的，Supervised Learning让这些侦察兵受更多的训练，让他们更靠谱。

- 第三个是在default policy时我们一般都需要快速走子走完全程，有没有办法不走呢？这个时间不就省了吗？事实是，可以有，Supervised Learning可以训练一个value network，直接评价某局面的得分。不过事实上`AlphaGo`采用了走和不走相结合的方式，用一个比例系数来组合这两者。



(a) (b)

图 5:

Supervised Learning的重点在policy network和rollout network。`AlphaGo`的论文里说，value network是最后加上的，对于结果的百分比有提升，但提升不大。

2.3.2 Reinforcement Learning

Reinforcement Learning是一个好东西，不过只有Google这样的土豪玩家才玩得起，Reinforcement Learning很难收敛，需要大量GPU和data。目前Reinforcement Learning的前沿还是被Deepmind和OpenAI所把持，小玩家无法进入。我们来谈谈Reinforcement Learning在围棋AI里发挥的作用。

- 提高policy network。在结构上，RL 策略网络和SL策略网络是一样的，它的权重也被初始化为SL网络的值。在训练时，让当前版本的策略网络和它的一个随机选择的先前版本相对抗。先前的版本组成一个对手池，从里随机选择可以让训练稳定下来，以防止当前策略的过拟合。对于非最后一步$t < T$，回报函数（reward function）r(s)被定义为0。从当前玩家在时间t 时的角度看，游戏的结果z_t是游戏结束时最终回报：+1 代表赢了，-1 代表输了。然后使用随机梯度上升法（stochastic gradient ascent）在每一时间步骤t 更新权重，以最大化期望的结果。
  效果极其显著（成本也极其巨大）。RL和SL对战，可赢80%的棋局，直接对当前局面用RL，不用搜索和pachi对战可赢85%的棋局。简直可怕。

- 提高value network。价值网络和策略网络的架构类似，但输出的是一个预测，而非概率分布。价值网络的权重是基于盘面-结果对（state-outcome pairs）(s, z)的回归来训练的，就是使用随机梯度降低法（stochastic gradient descent）来最小化预测值和相应结果之间的均方误差（MSE：mean squared error）。注意：使用整个比赛的数据来简单预测比赛会导致过拟合。问题在于相连的局势是强相关的，区别仅仅是一个子，但回归目标被整个游戏共享。用这种方法基于KGS 数据集进行训练，结果在测试集上，最小MSE 是0.37，而在训练集上，最小MSE 是0.19，表明价值网络记住了比赛的结果，而不是具有泛化到一个新盘面的能力。为了减轻这个问题，生成了包含3 千万个不同盘面的自我对抗数据集。每个盘面是从单独的比赛采样的。每个比赛由RL 策略网络自我对抗，直到比赛结束。基于这个数据集训练后，MSE 在训练数据集上是0.226，在测试数据集上是0.234，只是稍微的过拟合。
  价值网络的盘面评估也有相当的准确性，但运算量是rollout结果的15000 分之一，这是因为rollout要多次调用rollout network，而且rollout本身要调用多次取平均。AlphaGo是二者结合，估计也是想出于rollout次数的考虑。

注意，AlphaGo论文里说到，在AlphaGo里SL策略网络比更强的RL策略网络表现的更好，大概是因为是人类会选择一些更有前途的走子，而RL是为单步走子优化的。然而，从更强的RL策略网络导出的value network比从SL策略网络导出的要更好。

2.3.3 整体观

选择动作a_t,

$$a_t = \arg\max_a (Q(s_t, a) + u(s_t, a) \tag{1}$$

前者来自于value network和rollout的组合结果。

$$Q(s, a) = \frac{1}{N(s, a)} \sum \mathbf{1}(s, a, i) V(s_L^i) \tag{2}$$

$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L \tag{3}$$

后者来源于policy network，而且考虑到了探索，随分母增大而减小。所以会照顾到那些访问次数N比较小的节点。

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)} \tag{4}$$

一旦搜索完成，算法从根节点选择最多访问次数的儿子。为了有效结合MCTS和

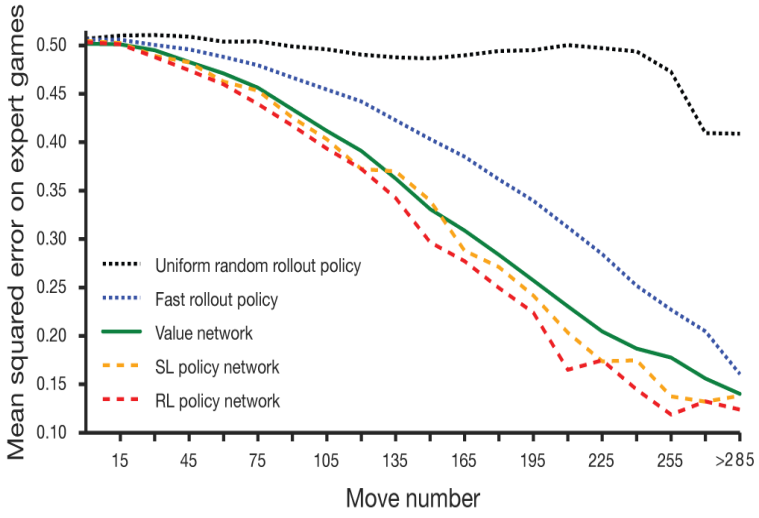


图 6:

神经网络，AlphaGo使用异步多线程进行搜索：在CPU上执行仿真，在GPU上并行地计算策略和价值网络。 AlphaGo的policy network更新异步地发生在的50 个GPU上。100 个步骤之前的梯度将被丢弃。3亿4千万个训练步骤总共大约花了3 周。reinforcement learning使用50 个GPU，在1 天的时间内用1 万个mini-batch（每个mini-batch 由128 局比赛组成）进行了训练。value network用5千万个mini-batch（每个mini-bach由32个盘面组成）进行了训练，使用了50个GPU，耗时1周。 AlphaGo的最终版本使用了40个搜索线程，48个CPU和8个GPU。也实现了一个分布式版本，利用了多台机器，40个搜索线程，1202个CPU和176个GPU。

3 从头造轮子：BAKAGO

3.1 项目基本情况

BakaGo是作为期末AI作业的一个围棋Bot，由四位小组成员共同开发。主入口是https://github.com/sjtu-ai-go/go-ai。名称来源：AlphaGo $\mapsto$ BetaGo(Github某项目，很弱) $\mapsto$ BataGo(拼写错误) $\mapsto$ BakaGo。因为是从头造的轮子，所以在学期初设计时，将其分拆为几个部分：

- libgo-common: 共用的基础代码（主要是Log）
- libgoboard：棋盘
- libfastrollout：输入棋盘，输出局面对双方谁更有利
- libuct：主算法库，输入棋盘，输出应下何处
- libpolicy-train：训练CNN
- libpolicy-server：CNN Server，接受Feature，返回Policy network对下在每一个点概率的评估
- libpolicy-grpc：原先准备采用gRPC作为main prog和CNN Server通信机制，但最后实际是存储.proto文件
- 第三方库： googletest, 单元测试。 spdlog，log。 Boost，主要是Program options。 gtplib，gtp协议处理。

我们AI下棋的步骤：
1.收到STDIN的genmove B命令，解析（go-ai, gtplib）。 2.派发给对应的engine处理(go-ai）。3.engine收到解析好的命令，调用libuct，传入当前棋盘状态。4.libuct传入棋盘，启动多个线程进行MCTS搜索(libuct)。 5.每个线程先从Root根据Tree Policy（通俗的来说就是有个UCT(node)函数，选取最大的孩子），一路下行，直到走到孩子不满的一个节点。 6.如果这个节点之前没有计算过candidate，且被访问超过16次，那么通过socket调用libpolicy-server，其通过CNN计算出当前局面状态下人类最可能下的若干个点，传回libuct，在节点上记录下这些candidate。如果访问不到16次，终止该次搜索。 7.之后，选取candidate中最可能的一个，加入ch。 8.利用libfastrollout评估这个ch对哪一方有利（Q值）。这一步是利用近似随机落子（有一个启发函数）下10盘，然后评估最终状态的情况取平均。然后Back propagation回树根，对黑方Q +=，对白方Q -=。 9.本次搜索结束。 10.在时间到后，选取Visit次数最大的一个根的孩子作为下的位置。

目前我们的模型主要优势在于选取Tree Policy扩展节点时，前期libpolicy-server的CNN能够给我们优质的初始点，所以大局上来看比较强。弱势主要在于：fast rollout是随机下子，效果差最终评估盘面优劣没有实现完整的点目机制CNN对局部状态的判断较差，容易在复杂互杀局面中被采用Pattern matching的bot整波带走经过初步的测试，我们的程序在单机上（i7 4710HQ, 850M）对DarkForest、GnuGo有很高的胜率；在服务器上（E5-2670，25线程，K8）对Pachi有一定胜率。

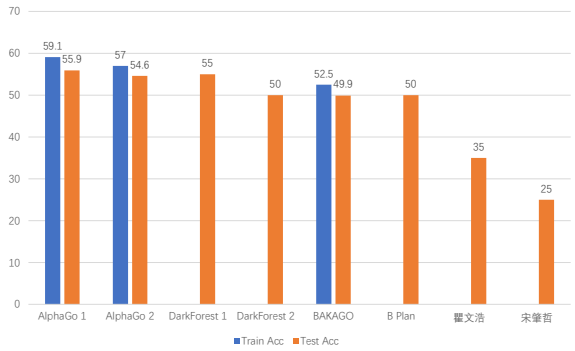
3.2 用Deep Learning指导走棋

这一部分是我在主导。我所做的工作如下：

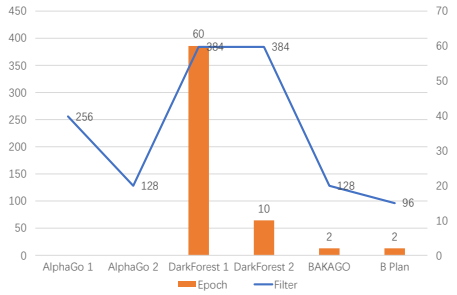
- 通读AlphaGo和Darkforest的paper，通读BetaGo的代码（一个开源的小轮子，准确率不到0.1），同时找来前人所写的UCT的项目以及gtp接口，仔细研究。
- 负责整个train的流程，推进项目。从前期的feature extracter, data loader, 到在小数据集上测试candidate model，再到放在服务器上跑，由单GPU到多GPU的支持。在中途出于对uct和启发式函数实现上的担忧，同时展开了我们的B计划，最终A，B计划都取得不错的结果。

组里有尚靖桓和我一起在做这方面的工作，尚靖桓负责feature extracter, 从棋盘中提取39维特征，还负责A计划BAKAGO后期在服务器上的监控和维护。对于data loader这部分，骆铮也帮忙对文件做了压缩和分chunk的处理。

先给出最终效果图7(a),7(b)。AlphaGo1和AlphaGo2的区别是前者filter为256，后者128。filter数目大体决定参数的多少。Darkforest1和Darkforest2的区别是Epoch不同，前者把整个数据集过了60遍而后者只过了10遍。这个遍数代表着收敛的速度，同等accuracy下遍数越少，收敛速度越快。（部分数据未公布）我们的BAKAGO和B plan由于训练时间不长，GPU有限，只过了整个数据不到2遍，就达到了不错的accuracy。



(a) Performance比较



(b) 参数比较

图 7: AlphaGo的accuracy最强55.9%。我们的BAKAGO的accuracy已接近第一梯队，考虑到我们只用2GPU，将全部数据只过了大概2遍，达到这个准确率是很让人吃惊的。假如继续训下去，精调参数，很可能达到更好的水平。B plan也表现良好。

以上是效果图，我们还是取得了自己比较满意的结果。关于train的过程，其实我也是第一次实战deep learning，我会介绍得比较naive，大神轻拍。具体思路和执行细节如下：

3.2.1 定方向

- 在项目开始的时候，搜寻信息可以帮助我们了解局势，确定方向。于是我们进行了分工，我看UCT相关论文，然后跟大伙交流。另外，考虑到我可能之后要训model，我必须对全盘有所了解，对model这块有比较仔细的了解。所以我还看了`AlphaGo`，`Darkforest`的论文，看了Yuandong Tian的一系列博文。我在github上搜到不少关于围棋AI的课程项目，深入看过brown 1.0的代码。

- 搜寻信息之后需要破局，要怎么切入。之后骆铮和我商量，考虑到各人的情况，我来负责model这块，他负责uct这块，其他人帮忙。我这学期在上Kai Yu的深度学习和Lei Xu的类脑智能，所以对learning这块还是有一定的了解。对于train models，我定的方向是:由小数据 $\mapsto$ 大数据，由existing model $\mapsto$ our new model，由少层 $\mapsto$ 多层（pretrain，有利于收敛），由支持单GPU $\mapsto$ 多GPU。

3.2.2 train的前奏

train往往不止于train本身，它必须有前期的准备工作：feature extracter, data loader。实际操作上我们是同时推进，没有明显的前后：由在小数据集上训练，发现问题，我们会考虑feature extracter，data loader是否需要进一步优化，优化完毕后再往大数据集上迁移。

- 我们第一步是利用了一个开源项目BetaGo（$acc < 0.10$）的feature extracter，只有7 features，7 features只能用于检验model的有效性（收敛速度，capacity，etc）。

- 在小数据 $\mapsto$ 大数据时，我们重写了feature extracter（尚靖桓，黄河）。考虑到`AlphaGo`抽取了40+ features，`Darkforest`抽取了20+ features，我们考虑特征的性价比（一次用时，实现难度），抽取了39 features。

- BetaGo本身自带data loader，但代码有歧义有错，而且完全不能支持big data的训练，会造成严重的I/O bound，带来巨大的存储空间(GB级别)。这部分我改了大量bug，添了注释，针对keras采用 fit-generator的方法来
处理数据，解决了内存占用巨大问题。骆铮将npy文件压缩成npz文件，解决了存储空间问题。尚靖桓调整了chunksize，一定程度上缓解了I/O bound的问题。

有必要讲一讲I/O bound的解决，这个问题困扰了我们很久。比较几种方式。

- 理想方式：一边训，一边调data进来，保证data充足。

- Torch，imagenet方式。data放在多个子文件夹中，多线程，每个线程去sample以保持选择class比较均匀。此法一次只加载相应thread所需的data。但是thread够多的话，就能源源不断。不过可能会出现on OSX, Apple has artificially limited the number of open file handles (so they can sell OSX server with a unlimited ulimit) which causes Lua's require system to die.

- Torch， darkforest方式。data同时有data index。用了torchnet库中的IndexedDataset和ParallelDatasetIterator来load数据，会比较消耗内存4-5G，但不会爆炸。BTW，你会发现darkforest的model入口和data入口特别难找。其中data入口被.bin被默认实现在了torchnet的源码里，所以代码里ctrl-f根本找不到(data).bin,也找不到torch.load命令,model入口也被隐藏得很深。torchnet源码里，IndexedDataset可以设置maxload数，默认是全部。读文件不采用load，而是专用的函数torch.DiskFile(indexfilename):binary()。

- Keras，我们的方式，先提好特征，存成npz文件。然后我们维护一个pool/chunk，从pool中fit-generate，pool用完了再走I/O。问题是chunksize很小的话会出现严重的I/O延时，具体体现在GPU utils一下子90%,一下子0%。所以需要每次都调chunksize，而调chunksize需要把所有zip文件重新处理一遍，需要几个小时。所以调chunksize是麻烦事情。而且线程需要够大。其实不能太怪chunksize，我们还有一点，是我的疏忽，我们没有做asynchronous training。train可以异步optimize，意思是在并行load数据时，可以顺便把梯度下降也给做了。这时梯度下降是异步的，但是没有关系，这被理论证明过也是可以的。

所以对于I/O bound问题的最好方式，我总结为以下技巧的组合。预计可以让GPU utils一直达到90%而非忽上忽下。

- .bin文件保存，而非.npz。.bin文件有torchnet的IndexedDataset的先天支持(虽然我们用的keras，keras的灵活性不如torch，不过假如之后有精力，可以转成torch框架)。

- 利用torchnet的ParallelDatasetIterator(keras有对应的fit-generator)，做Data Parallel，并且对model实现asynchronous training。

3.2.3 如何train一个好model（上）

这一部分侧重讲小数据作model测试。我先用约60w moves小数据在我的GTX960笔记本上初步检测model的performance，大数据迁移到张丽清老师实验室的服务器，我们用了2台k80。所有的model都在我的笔记本上先做效果测试，然后再迁移。

首先，什么样的小数据可以用来选择model？按常理来说deep learning需要big data，越大越好，小数据选择出的model迁移到大数据上的未必好，甚至小数据根本就训不出合格的model。这也是我最初所担心的。以下是我的考虑：

- 考虑到时间不多，GPU远不如Darkforest，我们必须用小数据做model可行性的测试。我们别无选择。

- 根据Computational Learning Theory的思想，数据量m超出某界限即可保证误差小于一定范围，如图8。我虽然不知到这个bound是多少，但也只能让我们的小数据尽可能的大，最终我们取了56w moves做training，4w可做validation。

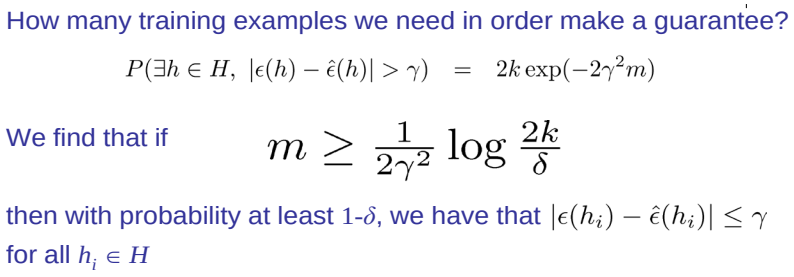
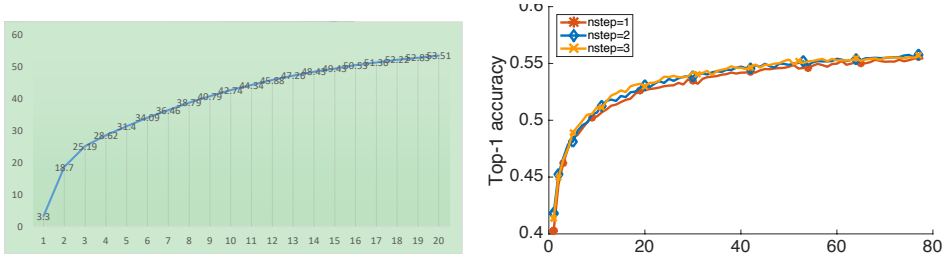


图 8:

- 图9(a)中，我测了Darkforest的小数据版本(56w, filter=32而非384)的training速度，发现与Darkforest的paper中的结果图趋势相一致，并且前20 epoch的accuracy惊人地接近。这可能是因为56w数据量选的刚刚好，而且围棋本身也具有高相关联性。



(a) darkforest 56w moves 32 filters (b) Darkforest paper

图 9: 横坐标为Epoch，左为小数据版本，右为Darkforest的paper中的结果图。我们的小数据版本在前20个epoch和darkforest的论文结果的前20个epoch惊人地相似。这证明了56w小数据可以来做model可行性检查。

这里列出小数据train model比较重要的几个因素:

- 数据量。我最开始取了比较少的数据，发现accuracy总是0.01以下，训练多个epoch也没用。这是因为model没有学到足够的信息所造成。之后解决了爆内存问题后，我取了将近60w moves做model的检测，好了很多，虽然60w仍然很小，还需要在big data上进行进一步检测。

- loss 的下降速度/accuracy的上升速度。指的是每经过几个epoch，会带来怎样的提升，这意味着model的收敛速度，在我们时间短而资源缺乏的环境下，收敛速度直接决定着项目能否完工以及效果好坏。

- accuracy limit。在小数据集检测时上，这项指标非常重要，它代表着一个model的capacity到底能有多大。在相同数量的参数的情况下，需优先选择那些capacity高的model。它预示着我们项目最终的upper bound。

- GPU数量和性能，train的时间，把整个数据集遍历的次数。

以下是我们的model原型及对比。先是policy network model,我们有三个版本（图10），再是rollout model，我们也有两个版本（图12）。我们的policy network model的performance对比如图11。

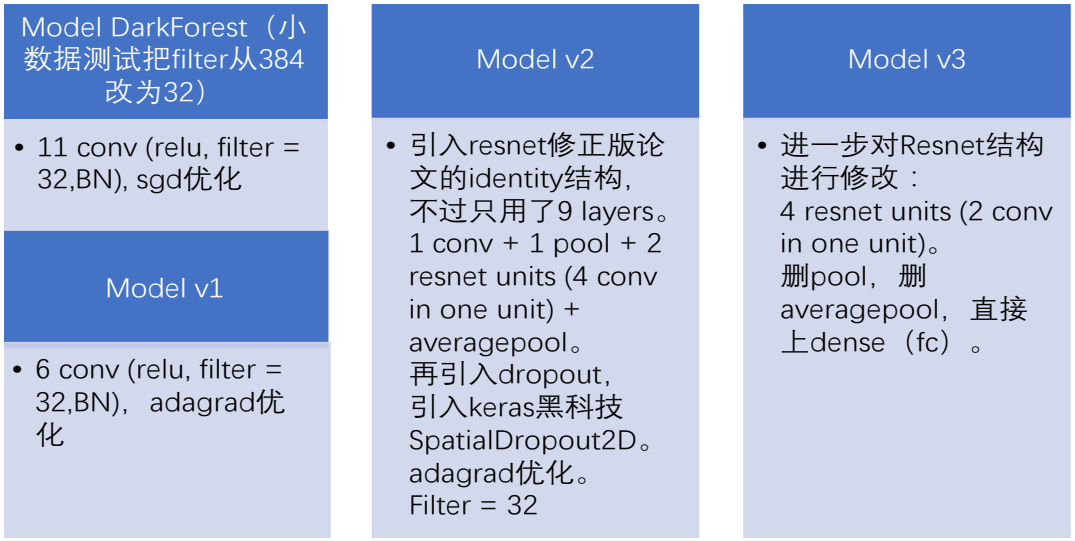


图 10: policy network model对比

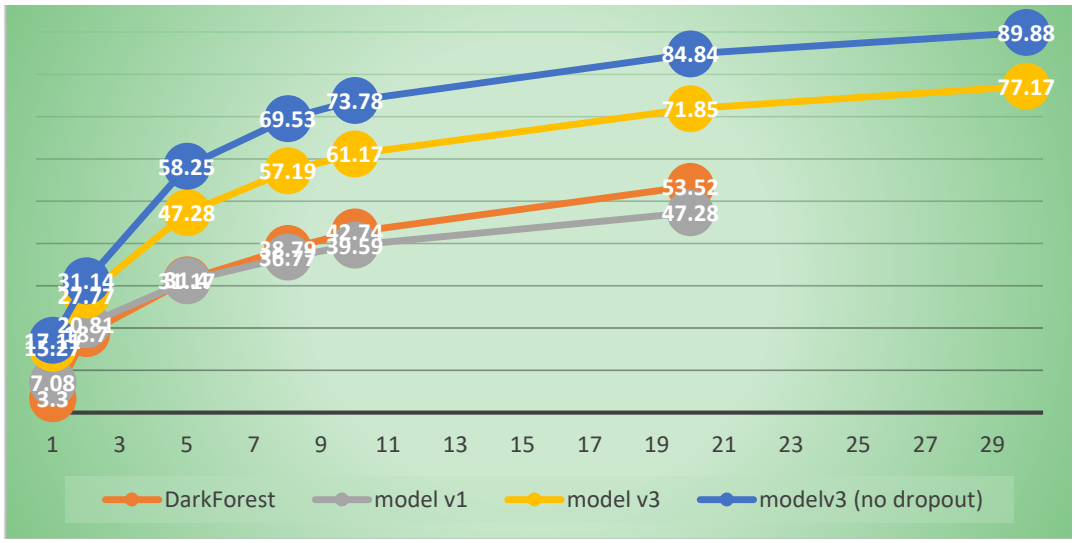


图 11: 56w数据集上 accuracy (y-axis) vs epoch (x-axis)。对于小数据难免到最后会过拟合，但我关注的是model的1.收敛速度，2.capacity。Model v1和DarkForest的区别是把11 conv改成6 conv，v1在epoch少时比DarkForest的acc高，但后劲不足，到20 epoch时已落后6个百分点。Model v3是残差的围棋改进版，从6 conv增加到9 conv，但依然不是DarkForest的11个，考虑到层数的性价比和model v3本身的capacity已经展示出了足够的优越性，我们取6，11之间的9。Model v3展示了较高的收敛速度(约为DarkForest的5倍快)和capacity，在迁移到大数据上训练时，我们时间有限，model v3的收敛速度拯救了我们。

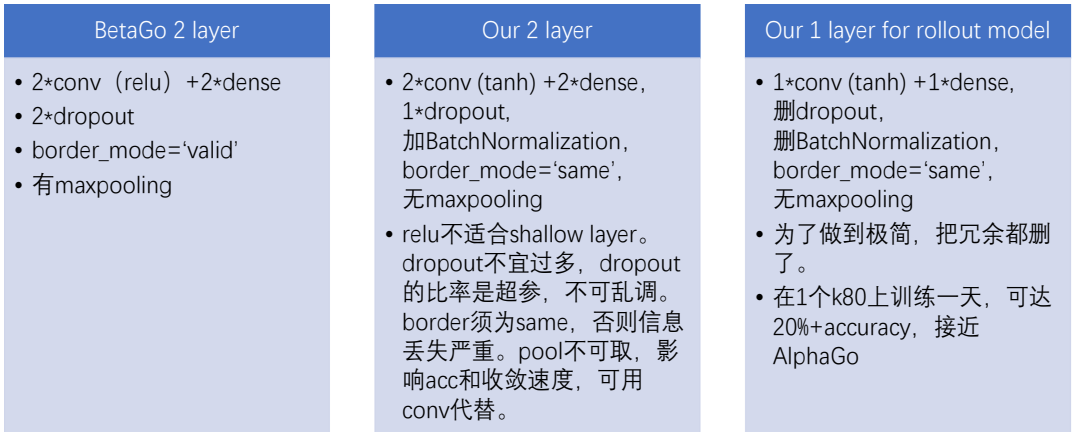


图 12: rollout model对比

在实验中，我发现围棋的训练和普通的deep learning最大的不同在于以下几点：

- 围棋中model的capacity一定要够大。Machine learning讲究降维，即使是deep learning也讲究“束腰”(Lei Xu老师语，指的是将input size减小，有必要时才通过deconv/upsampling来放大。)而且Kaiming He的原本的残差model也充满着降维的思想：pool，average pool，两种resnet unit交叉。但是围棋这边一定要保证capacity（这可能也是Yuandong Tian采用384 filter的原因之一，384远大于AlphaGo的128/256），经过实验，将所有的显式降维通通去掉了，因为丢失了大量信息，围棋本身信息就有限, 造成很差的performance。我们希望用all convolution layers达到比pooling更好的结果，我们采用了两种resnet unit中的一种：identity直接接过来。尤其是残差结束时用的一个averagepooling收尾，非常不适合围棋，删之。

- 围棋的输入与普通deep learning不同。普通deep learning是一张图丢进去，而围棋需要先提特征，将39张图丢进去。这是否可以成为提高deep learning performance的利器，甚至是deep learning和其他方法（如graphic model/metric learning/logic）结合的方法？想想应该很有可能从这里下手。通过显式增加输入层数的方法来训练。

Model部分吸收了最新的paper里的观点，列出如下。带!的项都对我们的训练有帮助，但是边际效用递减，所以只有尝试，并未采用。

- Deep Residual Learning for Image Recognition(Kaiming He 2015) && Identity Mappings in Deep Residual Networks(Kaiming He 2016) 残差网络及残差网络修改版。我们吸收其中的残差思想，对围棋AI进行了相应地修改。

- Striving for Simplicity: The All Convolutional Net(Jost Tobias Springenberg 2015)里面提到pooling如果加的不小心，会损伤性能，可只用conv足以。我们发现在围棋19*19棋盘上尤其如此(参见model v2 vs model v3)。

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (Sergey Ioffe 2015) BN的方法被我们所采用。

- Understanding the difficulty of training deepfeedforward neural networks. In International Conference on Artificial Intelligence and Statistics(Xavier Glorot 2010) && Caffe: Convolutional architecture for fast featureembedding(Yangqing Jia 2014)里提到的xavier初始化方法被我们采用。

- Deep Sparse Rectifier Neural Networks (Xavier Glorot 2011) && Dropout: A Simple Way to Prevent Neural Networks from Overfitting(Nitish Srivastava 2014)里面sparse的思想被我们借鉴。我们使用了dropout断裂节点间联系，用keras黑科技SpatialDroppout2D断裂map之间的联系。

! Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning(Christian Szegedy 2016)中提到的3*3 conv $\mapsto$ 3*1 + 1*3 + 1*1的思想（有点类似NIN）我们也有尝试，其中inception的module模块化的思想我们也有尝试。

! Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (Kaiming He 2015)里面提到pRelu的方法。

! Learning Deconvolution Network for Semantic Segmentation(Hyeonwoo Noh 2015)里面反卷积，上采样的方法被我们考虑过。

! The Microsoft 2016 Conversational Speech Recognition System(Weidong Xiong 2016)里提到deep固然重要，但wide的作用也在逐渐显露。将ensemble的方法显式用于CNN，是最新挖的坑。

3.2.4 如何train一个好model（下）

这一部分侧重讲把model迁移到服务器上用大数据跑。我先配好服务器相关环境，解决服务器cudnn，cuda版本和我们的keras的gpu不兼容问题，解决一系列麻烦的路径问题（不能sudo真麻烦，似乎有人搞乱了guest的环境），将单GPU版本改写成支持多GPU的版本（Data Parallel）。尚靖桓解决了BatchNormalization的版本不兼容问题；调整了chunk size，缓解了I/O bound；增加了“断点续传”功能。对于在大数据上的model我们的调整如下：

- filter size由30增加到128，为了增强model的capacity。

- 出于时间的考虑，我们没时间把整个数据过二三十遍，所以把dropout之类的删了，我们担心的是欠拟合，而不是过拟合。

- pretrain一层一层往上可获得好的参数分布。不过时间原因来不及采用。

- 减少全卷积层，原来我们flatten之后有3w节点，是通过3k节点再接到19*19=361节点上。训练到中途时，考虑到参数的体积问题，以及带来的训练时间消耗和forward一遍的时间消耗，我们利用去掉了3k节点的层，其余层参数不变。

- 我们的train accuracy达到0.45时，发现了两个特征提的不对，棋子离边框的L2距离被当成了L1距离。修改回来后，重新将特征灌入，发现准确率弹射起飞，达到0.52。即：用L1训出的参数对L2正则同样有效，甚至更有效。个人猜想是L1的sparse功能让x，y趋近0，某种意义上也满足了L2的要求。

后期对服务器情况的跟踪由尚靖桓完成，validation的结果也由尚靖桓实验获得。组里出于全盘考虑，黄河租了云服务器，我另外在darkforest源码下开展B计划（下一章详细说明）。

3.2.5 反思

我们的A计划截止比赛前训得比较成功。达到了0.52的训练准确率，0.50的测试准确率，能战胜开源的darkforest model。事实上，我们在小数据集上作了比较，无论是收敛速度，还是capacity，还是参数数量，我们的model都要明显优于darkforest。只是我们时间有限，对全部数据集只过了不到2遍，假如有充足的资源和时间，我们通力合作，相信能够得到更好结果。

A计划BAKAGO同时也有很多不足。我们认识到BAKAGO离高水准围棋AI差距较大。以下几点需要改进：

- 由于c和python通信延时较大，rollout model没有用进来，这是巨大的遗憾。这造成了我们用k80 10s vs 12s输给田晓亮michi/pachi(k80 10 vs 10胜，960 10 vs 10 败于田晓亮组)。我们的对杀短板需要补起来。
- 引入增强学习，用好增强学习。（宋肇哲组号称用了增强学习，不过他们用k80 vs 我们850,被我们狂虐约30目，之后他们偷偷加到2 GPU，第二局才稍胜我们1.5目。个人认为他们不太会用增强学习，反而为了晋级不择手段。最终重比，被我们2:0击败）增强学习是利器，不过用好也不易。
- Ponder Skill。目前darkforest有这功能：在对方走棋的10s内GPU utils仍然满载。这功能非常可怕，将GPU的利用率达到极致。我们目前没有这功能，能打赢darkforest，要是增加了，相信水平会更上一层楼。Ponder Skill的大致思想是通过预测对方的走子，然后展开最可能的节点，节省己方的时间。
- 多GPU并行，目前只做了data parallel，还可以做model parallel。
- I/O bound的处理。darkforest的GPU utils利用率一直高居0.9以上，而我们是忽上忽下。这部分需要上asynchronous training的方法。
- 收官加强。避免送子，控制劫争。这部分其实是deep learning AI共有的缺点，`AlphaGo`也有，不过最近的`Master`似乎克服了这一缺点，在与高手的劫争中毫不逊色。这部分值得研究。
- 选取特征精细化，data augmentation多样化。

4 B计划：改良darkforest model

B计划由我全权负责。B计划展开的原因是出于两个担忧：

- A计划需要根据大数据进行调参，而且不知道要训多久才能训好，而且服务器很紧张，能否在比赛前训出一个好的结果，能否打败现有的darkforest。
- A计划的uct，rollout能否跟上来，假如跟不上来我们就采用darkforest的uct部分。

于是，在18周周一晚，我们紧急启动B计划，第二天，黄河租了一台云服务器，我们连夜配了环境，（云服务器还有另外一项目的，是为了训rollout model，但因为通信延迟不可忍受，最后没有用进去。）B计划我打算采用和A计划类似的model，根据darkforest进行稍微修改，并增加其训练时的“断点续传”功能，最终在达到0.50的test accuracy时停止训练，因为云服务器太贵，再训性价比不高了。

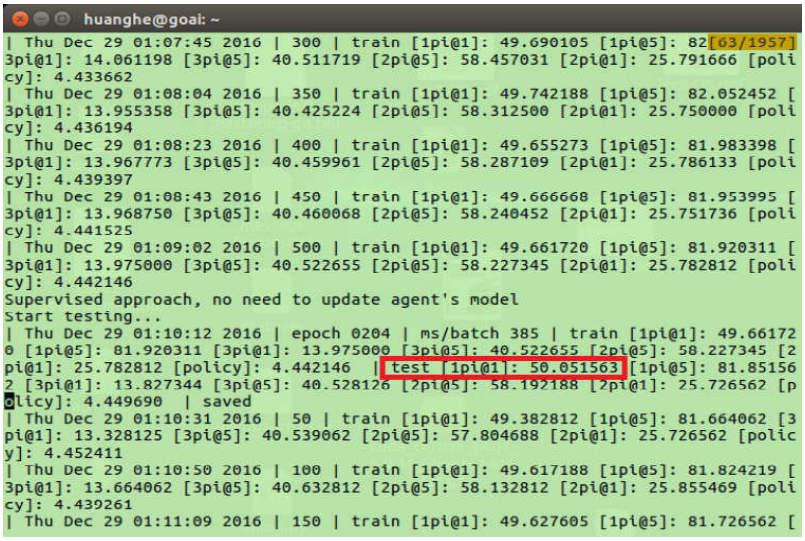


图 13: B计划结果：test accuracy: 0.50。值得指出的是，Darkforest的model大小有90多M，而B plan的model只有5.8M，相差近16倍。

B计划1台k80训练只有将近2天，将全部数据过完不到2遍（图中epoch已做了修改，不代表遍数），能达到这样的效果靠两个因素：

- resnet思想的引入使我们的model比darkforest的model要先进，收敛能力强。
- darkforest的优化方式是非常原始的sgd(我们A计划是adagrad)，原始sgd可以达到好的performance，但是精调和好的初值缺一不可。我是属于初始参数碰运气碰的好。之前我多次尝试，都是训练了一段时间还是0.01以下，就被我切掉。但是有一次我来试，有如神来之笔，才训了一会儿，就达到了0.2+以上准确率！我抓住这次机会训下去，中途有改过结构，但仍可利用不变层的参数继续训练，恢复速度也很快。

B计划在18周周四晚上和A计划BAKAGO比赛，结果是输在10目之内。当时A计划的acc比0.50差了几个百分点，竟然可以打赢B计划。我当时没有细想，后来思考其中原因，可能是因为B计划换了model，但uct的相关参数没有相应修改。不过A计划最后的acc也达到了0.5+，能打败原生的darkforest，所以我们仍决定由我们的A计划BAKAGO参与比赛。

5 model附录

Our rollout model 1 layer:

```
def one_layer_rollout( go_board_rows, go_board_cols,  input_channels, nb_classes):
nb_filters = 32  # for small data, if big: 96
nb_pool = 2  # size of pooling area for max pooling
nb_conv = 3  # convolution kernel size
l, w = 0, 0
model = Sequential()
#19*19
if K.image_dim_ordering() == 'tf':
model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same',input_shape=(go_board_rows,
   go_board_cols, input_channels))) #for tensorflow
else:
```

```
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same',input_shape=(input_channels, go_board_rows, go_board_cols)))
  model.add(Activation('tanh'))
  model.add(Flatten())
  model.add(Dense(nb_classes))
  model.add(Activation('softmax'))

```

Our 8-layer model (model v1):

```
def model_v1():
  nb_filters = 32  # number of convolutional filters to use
  nb_pool = 2  # size of pooling area for max pooling
  nb_conv = 3  # convolution kernel size
  l, w = 0, 0
  model = Sequential()
  #19*19
  if K.image_dim_ordering() == 'tf':
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same',input_shape=(go_board_rows, go_board_cols, input_channels))) #for tensorflow
  else:
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same',input_shape=(input_channels, go_board_rows, go_board_cols)))
  model.add(Activation('relu'))
  model.add(BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.9, weights=None, beta_init='zero', gamma_init='one'))
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same'))
  model.add(Activation('relu'))
  model.add(BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.9, weights=None, beta_init='zero', gamma_init='one'))
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same'))
  model.add(Activation('relu'))
  model.add(BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.9, weights=None, beta_init='zero', gamma_init='one'))
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same'))
  model.add(Activation('relu'))
  model.add(BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.9, weights=None, beta_init='zero', gamma_init='one'))
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same'))
  model.add(Activation('relu'))
  model.add(BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.9, weights=None, beta_init='zero', gamma_init='one'))
  model.add(Convolution2D(nb_filters, nb_conv, nb_conv, border_mode='same'))
  model.add(Activation('relu'))
  model.add(BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.9, weights=None, beta_init='zero', gamma_init='one'))
  model.add(Dropout(0.25))
  model.add(Flatten())
  model.add(Dense(256))
  model.add(Activation('relu'))
  model.add(Dropout(0.5))
  model.add(Dense(nb_classes))
  model.add(Activation('softmax'))

```

Our model v3, supporting multiple GPU:

```
def make_parallel(model, gpu_count):
  def get_slice(data, idx, parts):
    shape = tf.shape(data)
    size = tf.concat(0, [ shape[:1]/parts, shape[1:] ])
    stride = tf.concat(0, [ shape[:1]/parts, shape[1:]*0 ])
    start = stride * idx
    return tf.slice(data, start, size)

  outputs_all = []
  for i in xrange(len(model.outputs)):
  outputs_all.append([])

  #Place a copy of the model on each GPU, each getting a slice of the batch
  for i in xrange(gpu_count):
    with tf.device('/gpu:%d' % i):
      with tf.name_scope('tower_%d' % i) as scope:

        inputs = []
        #Slice each input into a piece for processing on this GPU
        for x in model.inputs:
          input_shape = tuple(x.get_shape().as_list())[1:]
          slice_n = Lambda(get_slice, output_shape=input_shape, arguments={'idx':i,'parts':gpu_count})(x)
          inputs.append(slice_n)
        outputs = model(inputs)

        if not isinstance(outputs, list):
        outputs = [outputs]

        #Save all the outputs for merging back together later
        for l in xrange(len(outputs)):
        outputs_all[l].append(outputs[l])

  # merge outputs on CPU
```

```
  with tf.device('/cpu:0'):
    merged = []
    for outputs in outputs_all:
      merged.append(merge(outputs, mode='concat', concat_axis=0))

    return Model(input=model.inputs, output=merged)

def _conv_bn_relu(nb_filter, nb_row, nb_col, subsample=(1, 1)):
  def f(input):
    conv = Convolution2D(nb_filter=nb_filter, nb_row=nb_row, nb_col=nb_col, subsample=subsample,
    init="he_normal", border_mode="same")(input)
    norm = BatchNormalization(mode=0, axis=CHANNEL_AXIS)(conv)
    return Activation("relu")(norm)
  return f

# Helper to build a BN -> relu -> conv block
# This is an improved scheme proposed in http://arxiv.org/pdf/1603.05027v2.pdf
def _bn_relu_conv(nb_filter, nb_row, nb_col, subsample=(1, 1)):
  def f(input):
    norm = BatchNormalization(mode=0, axis=CHANNEL_AXIS)(input)
    activation = Activation("relu")(norm)
    return Convolution2D(nb_filter=nb_filter, nb_row=nb_row, nb_col=nb_col, subsample=subsample,
    init="he_normal", border_mode="same")(activation)
  return f

def _shortcut(input, residual):
  stride_width = input._keras_shape[ROW_AXIS] // residual._keras_shape[ROW_AXIS]
  stride_height = input._keras_shape[COL_AXIS] // residual._keras_shape[COL_AXIS]
  equal_channels = residual._keras_shape[CHANNEL_AXIS] == input._keras_shape[CHANNEL_AXIS]

  shortcut = input
  # 1 X 1 conv if shape is different. Else identity.
  if stride_width > 1 or stride_height > 1 or not equal_channels:
    shortcut = Convolution2D(nb_filter=residual._keras_shape[CHANNEL_AXIS],
    nb_row=1, nb_col=1,
    subsample=(stride_width, stride_height),
    init="he_normal", border_mode="valid")(input)
  return merge([shortcut, residual], mode="sum")

# Builds a residual block with repeating bottleneck blocks.
def _residual_block(block_function, nb_filters, repetitions, is_first_layer=False):
  def f(input):
    for i in range(repetitions):
    init_subsample = (1, 1)
    #if i == 0 and not is_first_layer:
    #    init_subsample = (2, 2)
    input = block_function(nb_filters=nb_filters, init_subsample=init_subsample)(input)
    return input
  return f

def basic_block(nb_filters, init_subsample=(1, 1)):
  def f(input):
    conv1 = _bn_relu_conv(nb_filters, 3, 3, subsample=init_subsample)(input)
    residual = _bn_relu_conv(nb_filters, 3, 3)(conv1)
    return _shortcut(input, residual)
  return f

def handle_dim_ordering():
  global ROW_AXIS
  global COL_AXIS
  global CHANNEL_AXIS
  if K.image_dim_ordering() == 'tf':
  ROW_AXIS = 1
  COL_AXIS = 2
  CHANNEL_AXIS = 3
  else:
  CHANNEL_AXIS = 1
  ROW_AXIS = 2
  COL_AXIS = 3

class ResnetBuilder(object):
  @staticmethod
  def build(input_shape, num_outputs, block_fn, repetitions):
    handle_dim_ordering()
    if len(input_shape) != 3:
    raise Exception("Input shape should be a tuple (nb_channels, nb_rows, nb_cols)")

    # Permute dimension order if necessary
    if K.image_dim_ordering() == 'tf':
      input_shape = (input_shape[1], input_shape[2], input_shape[0])

    input = Input(shape=input_shape)
    #conv1 = _conv_bn_relu(nb_filter=64, nb_row=7, nb_col=7, subsample=(2, 2))(input)
    conv1 = _conv_bn_relu(nb_filter=128, nb_row=3, nb_col=3, subsample=(1, 1))(input)
    #pool1 = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), border_mode="same")(conv1)
    #block = pool1
    block = conv1
    nb_filters = 128#32#64
    for i, r in enumerate(repetitions):
      block = _residual_block(block_fn, nb_filters=nb_filters, repetitions=r, is_first_layer=i == 0)(
     block)
      #nb_filters *= 2
```

```
    flatten1 = Flatten()(block)
    dense = Dense(output_dim=num_outputs, init="he_normal", activation="softmax")(flatten1)
    model = Model(input=input, output=dense)
    return model

@staticmethod
def build_resnet_8(input_shape, num_outputs):
  return ResnetBuilder.build(input_shape, num_outputs, basic_block, [1, 1, 1, 1])

def fetch_model(go_board_rows, go_board_cols,  input_channels, nb_classes):
  model = ResnetBuilder.build_resnet_8((input_channels, go_board_rows, go_board_cols), nb_classes)#
  #model.compile(loss="categorical_crossentropy", optimizer="sgd")
  model.summary()
  return model
```