

# 初步考察

## 拆轮子

Caffe的框架是给supervised learning准备的，要实现unsupervised的reinforcement learning(RL)只能暴力拆轮子。虽然没有文档给出如何使用Net类，但是观察Solver类的代码，配合已有的注释，就已经足够了。我们对Caffe的框架的改动主要有这些：

1. 输入方式。框架使用仅有输出而没有输入的Data layer作为一个Net的最底端，它负责从LevelDB/LMDB当中读取训练数据并传给之后的Layer。但这显然不太适用于RL，我们目前有两种备选方案，其一，用MemoryDataLayer，其二，直接设置Input blob然后把要输入的数据拷贝到其中。前者可能需要对MemoryDataLayer作更多的考察，因此我们倾向于后者。
2. Forward和Backward的分离。原来的框架以SoftmaxWithLoss之类的Layer直接作为最后一层，在Forward之后立刻Backward完成参数的调整。我们现在回来看论文的算法，它需要先把当前画面进行一次Forward获取当前的决策，添加进history，再从history随机抽取一个minibatch进行使用EuclideanLoss的Gradient Descent。这同时引起了batch size不一的问题，从Reshape的操作看来batch size是能随时更改的；这是个好消息。
3. 两个网络。论文使用了一种神奇的办法来增强算法的稳定性：用一个网络Q计算当前的决策和进行学习，但Loss是用另一个网络Q'计算的，并且每C步命 $Q' = Q$ ，也即Q'完全接收Q的权值。这严格来说不是什么重要的改动，但会使得我们的实现稍微麻烦一些，我们会考虑先实现不考虑这一因素的版本。

## 模拟器

我们采用Arcade Learning Environment作为游戏环境。它提供了便利的和模拟器交互的Interface，我们要做的工作事实上大部分也就是实现在Caffe和ALE之间的数据传输。这就有两个方面：

1. 图像的预处理和传输。论文中提到的图像数据的预处理主要包括三个部分：对于每一帧，和前一帧取较大的颜色值（由于机能的限制，Atari上有些物件是隔帧显示的）；对于每个像素，取灰度值，然后把灰度图像压缩到 $84 * 84$ 的大小；最后把最近的m帧堆到一块作为神经网络的输入（m取4，但论文同时指出m为3或5的时候算法也能得到满意的结果）。第一部分ALE已经做好了。第二部分需要我们拿到RGB数据，但观察ALEInterface#getScreen的返回值，发现其中包含的矩阵是char类型的。这是由于Atari年代久远，采用的是颜色集合较为固定（称为调色板）然后每个像素从调色板中取色的颜色显示方案。观察ALE提供的保存屏幕到PNG文件的类的代码可以发现theOSystem->colourPalette()可以得到调色板（其中theOSystem是ALEInterface的成员变量），设为palette，再对palette调用palette.getRGB(screen.getArray()[i \* dataWidth + j], r, g, b)就可以在r, g, b中得到RGB值，这里screen是getScreen返回的ALEScreen的对象。从RGB得到灰度值不是什么难事，但是最后一步如果在Caffe当中试图实现，就会显得有些怪异，因为这个“堆起来”不是图像尺寸的改变，也不是batch size的改变，因而只能在Blob的第二维，也就是Channel，来改变。而这一维通常是指图像到底是灰度的（Channel = 1）还是彩色的（Channel = 3，分别给RGB），因此强行让Channel = m然后塞m帧灰度图像进去不见得能保证效果。不过在有更好的方案之前也只能这样了。
2. 动作的执行。从网络里获取输出并不是什么难事，观察Layer的类别可以发现ArgmaxLayer可以用于从网络中提取它所选取的动作。而从Net的外部获取这一信息，只需要利用Net#output\_blobs()就可以了，这是因为ArgmaxLayer的输出没有Layer接收，因而会被视为输出blob。

# 一个测试

我们用简化的 $n$ -Bandit问题来测试我们的想法。在这个简化的问题中，我们的Agent将重复不断的被要求从n个选项中选择一个，然后得到一定的Reward。选择第*i*个选项得到的Reward是仅和*i*有关的常数（而不是通常采用的正态分布）。

## 网络

Agent所需要做的其实就是在为每个选项做一个估价，因此网络可以简单的设定为没有输入， $n$ 个输出，bias即是各个选项的估价值。但我们不清楚没有输入的InnerProductLayer是否能正常工作，因此我们没有采用的这个想法；更重要的是，我们需要测试实时塞数据给网络的做法的可行性。因此我们将InnerProductLayer设计为，仅有一个被赋予常值1的输入， $n$ 个不带bias的输出，也就是让权值来作为估价。在这一层之后，我们使用EuclideanLoss来学习，这一层接收IP层的输出，以及新观测到的值。这里我们发现了一个问题，新观测到的值只有一个，没法对全部的估价值做调整。所以我们把新观测到的值的向量设定为，基于目前的估价值向量（也就是IP层的输出），仅把刚刚采取的操作对应的下标元素改为新观测到的值。以后这一行为可能会修改以测试minibatch SGD。

## 学习

激动人心的拆轮子的时候到了。我们实际需要实现的学习策略比较麻烦（见下文RMSProp节），但作为简单的测试，我们考虑直接修改SGDSolver的代码来使用。具体说来，实际上需要修改的目前看起来只有Step函数，观察SGDSolver的定义，它现在就已经是一个基类了，我们可以放心的继承下来，然后修改Step函数的行为。

## 更多的细节

### RMSProp

论文给出的算法采用了RMSProp配合momentum进行学习。RMSProp是SGD的常见优化之一，参阅[这里](#)。相应的参数可以在论文的Hyperparameters表中查到：momentum是0.95，squared gradient momentum也是0.95，min squared gradient(也就是链接中的 $\mu$ )是0.01。我们惊喜的发现Caffe只有momentum没有RMSProp，不仅如此，我们还惊喜的发现试图这样做的Pull Request并没有成功。

所以我们对待这个情况的态度和“两个网络”的问题一样，如果最终能实现当然是好事，但早期的半成品多半是不会包含这个特性的。