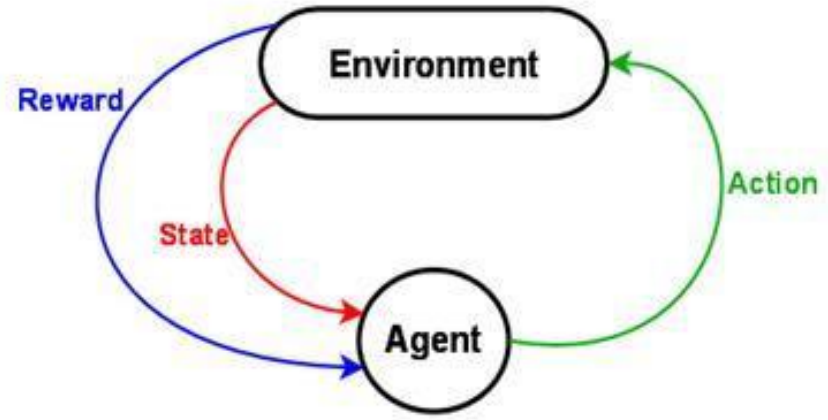
Q-Learning Documentation

问题

论文中主要处理了一个这样的问题，给出一个游戏，程序在不知道程序游戏规则，也不知道游戏目标或是程序不同的指令对游戏产生的影响的情况下，仅仅得到分数变化和游戏屏幕的图像作为输入的情况下学习游戏，并获得尽可能高的分数。



如图所示，程序 (图中 agent) 从游戏 (图中 Environment) 获得分数变化 (reward) 和屏幕图像 (state)，经过处理后向游戏发出指令 (action)，游戏获得指令后执行并继续向程序返回屏幕图像和分数变化，重复这一过程直到游戏结束为止。

Q-Learning 算法

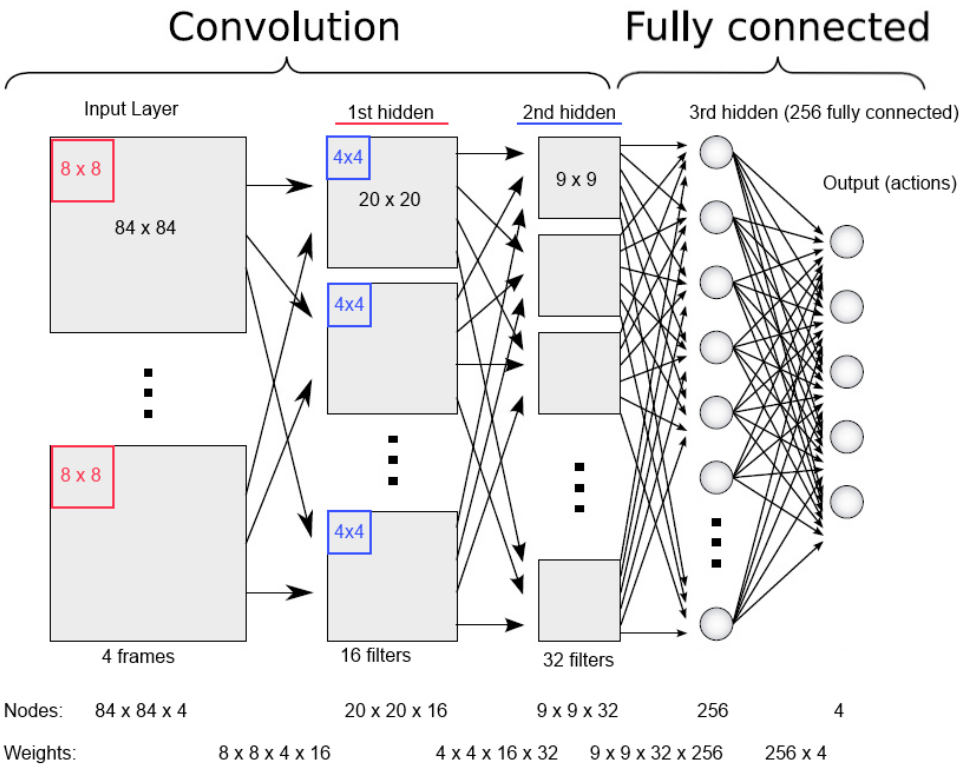
为了解决这个问题，论文使用了一种深度学习算法，论文中的深度学习算法称为 Q-Learning。论文在实现 Q-Learning 的过程中，使用了深度神经网络 (deep neural networks)。深度神经网络在实现过程中结合了深度学习 (reinforcement learning) 和人工神经网络 (artificial neural network)，论文中的这一深度神经网络称为 DQN (deep Q-network)。

我们可以认为程序的工作环境是一个由观察（游戏屏幕输出）、行动（程序发出指令）、和反馈（得分变化）组成的队列，而程序的目标是选择行动使得将来累积的反馈最大。一般地，我们可以使用深度卷积神经网络来得到反馈函数 $Q(s,a)$。同时可以根据下列的迭代式得到期望的反馈值。

$$Q^*(s,a) = \max_{\pi} E\left[\sum_{s \in state} \gamma^{t-s} r_s \mid s_t = s, a_t = a, \pi\right]$$

r_t 为 t 时刻的反馈，$\pi = P(a \mid b)$ ，π 在该问题中可以等价于决策，γ 为衰减常数。游戏过程中根据 $Q(s,a)$ 和 $Q^*(s,a)$ 不断更新神经网络 Q。

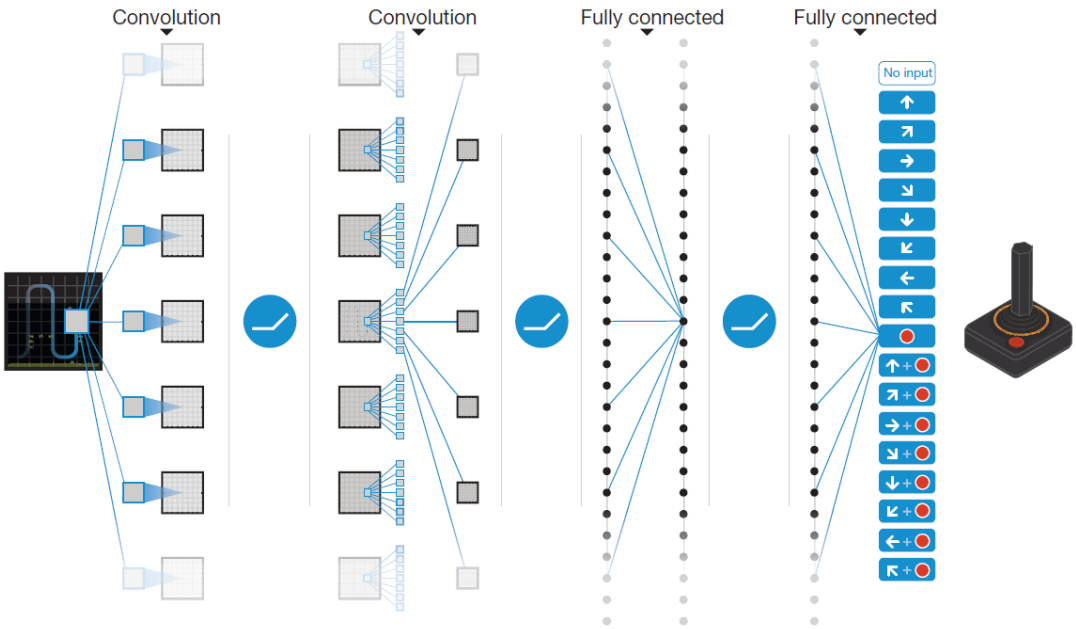
在程序中使用了一个神经网络 DQN。下图为文中所提出的 DQN 结构详细样例。输入部分由最新的若干幅画面组成，这样这个网络不仅仅能够看到最后的部分，而且能够看到一些这个游戏是如何变化的。输入经过三个后继的隐藏层，最终到达输出层。



在输出层有对每个指令都有对应的反馈，最终期望最高的反馈会被用来执行下一步。

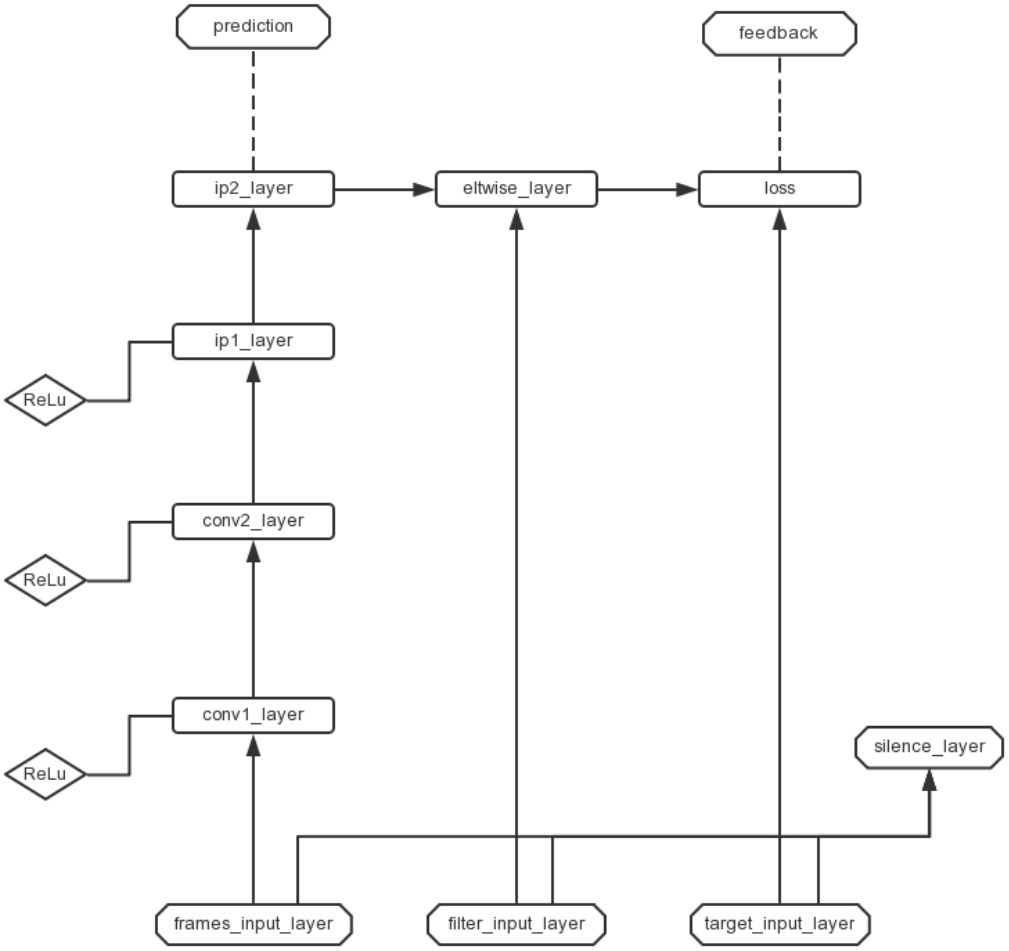
如何使用 caffe 实现 Q-Learning

在论文中，作者使用的卷积神经网络为下图所示：



图中为 6 层网络，除输入输出层以外，还有两个卷积层，两个隐藏层 (Fully connected，caffe 中的 InnerProduct 层)。

在这里我们使用 C++与 caffe 实现以上神经网络，由于在 caffe 中没有可以实现 reward 反馈的函数以及工具，所以需要在原网络中的输出层之后，再加入两层网络，分别为内积层与损失层。形成的神经网络如下图所示：



其中 frames_input_layer 为图形信息输入，silence_layer 接收输入中的空白 label，conv1_layer, conv2_layer, ip1_layer 分别对应为论文中 Convolution 层与 Fully connected 层，ReLu 为 ReLu 函数，ip2_layer 为输出层。在向 frames_input_layer 中传入数据后，可以使用 ForwardPrefilled 函数，前向传播，在 ip2_layer 处可以得到 Deep Q-Network 的预测结果，而 Q-Learning 中反向传播的反馈过程可以通过如下方式处理：

设 $Q(s, a)$ 为 Deep Q-Network 在游戏状态为 s，决策为 a 时的预测收益值。其中 $a \in LegalActions$。

则在状态 s 处预测向量为：

$$Q_a = Q(s,\ a)$$

决策向量为：

$$V_i = \begin{cases} 1 & \max\limits_{i \in LegalActions} \{Q_i\} \\ 0 & else \end{cases}$$

在这里可以将 V_i 为 lilter_input_layer 的输入，在 eltwise_layer 与预测结果做内积，获得极大预测收益，为标量如下：

$$\max_{a \in LegalActions} \{Q(s,a)\}$$

而状态 t 的反馈收益为 r_t。记录下所有状态的反馈 r_s，使用论文中的迭代式，如下：

$$Q^*(s_t,a) = \sum_{s \in state} \gamma^{t-s} r_s$$

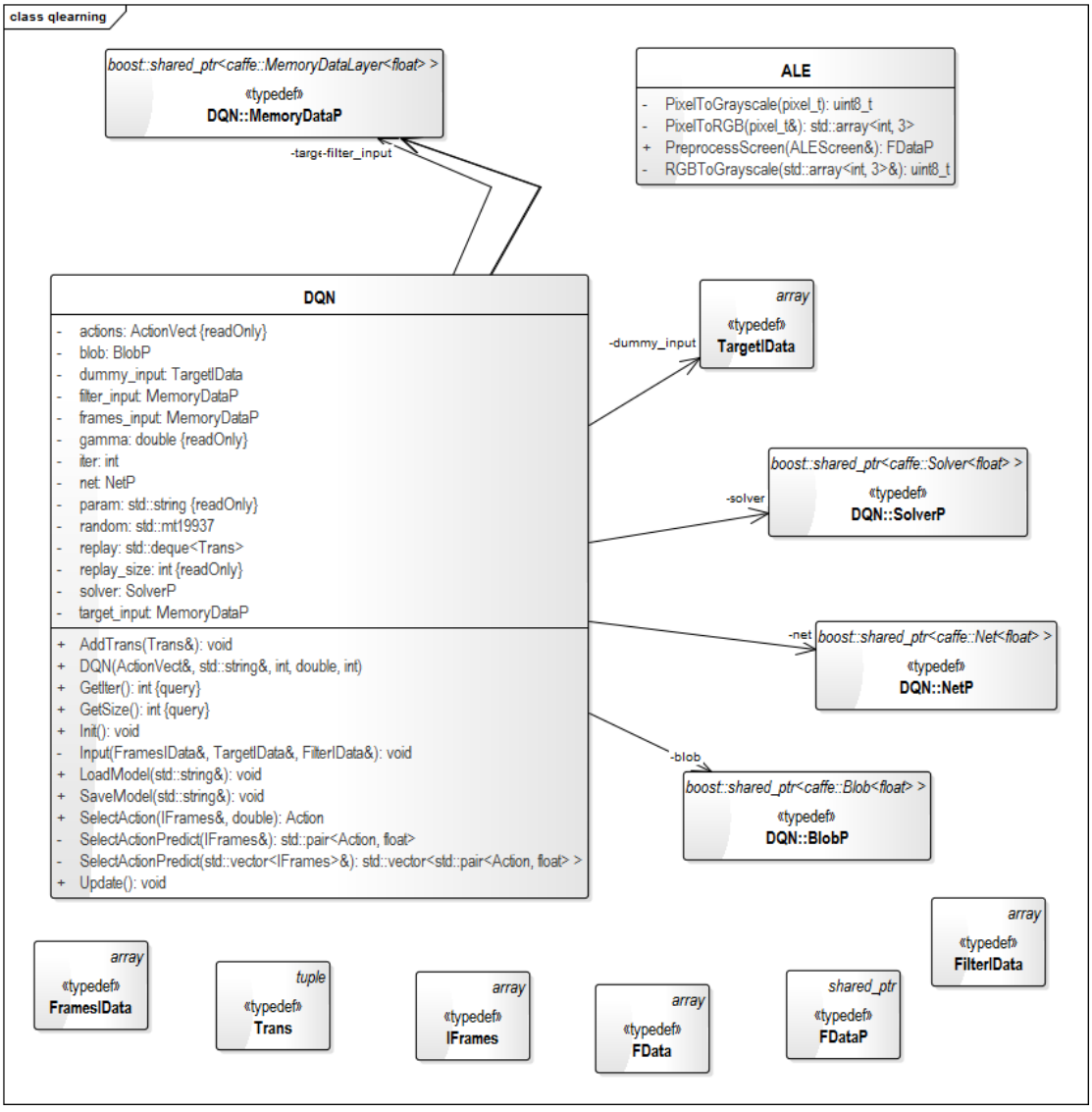
可获得 loss 层的目标值，与 eltwise_layer 处内积得到的结果一同传入 loss 层，使用 caffe 中 solver 自带函数 Step 进行反馈更新。

在这里，为了简化内存压力，使用近似式，考虑整个游戏过程，只有最后几部的操作影响较大，所以随机选取一些操作进行更新，而且γ较小，只需与最近一次操作迭代即可获得近似的目标值，可以加速学习的过程。

用户层架构

在 QLearning 这一项目中，用户层分为两个部分。一个是名为 qcli 的前端用户接口；另一个为 libqlearning 库。

libqlearning 库



其中 libqlearning 中包含两个类，DQN 类利用 Caffe 库，实现了一个 Deep Q-Network，完成 Q 学习的过程；ALE 类用于处理 Arcade Learning Environment 返回的截图数据，将其化为 DQN 接受的 FDataP 数据。

具体而言，程序向 DQN 类的实例提供神经网络定义、可选的操作列表 ActionVec，并每次提供最近的状态 FData 的指针 FDataP 输入序列 IFrames，通过对状态通过 SelectAction 分析决策，返回操作 Action，由程序执行。获得得分奖励，通过 AddTrans 将信息反馈给 DQN，使用 Update 方法，修改优化这一神经网络模型。在这一过程中，SelectAction 将根据参数 epsilon 指定的概率随机操作，保障网络模型可以得到优化。同时提供了 LoadModel 和 SaveModel 方法进行网络模型的读取与保存。

ALE 类中，有方法 PreprocessScreen，它接受 Arcade Learning Environment 库中，getScreen 返回的游戏截图 ALEScreen，将 NTSC 格式的颜色编码，通过 PixelToRGB 变为 RGB 格式，再通过 RGBToGrayscale 加权转化为能最好地区别颜色的灰度，再将其按点平均压缩为固定大小的，DQN 类的实例能够接受的 FData 状态类型。

在编译过程中，会产生或 libqlearning.a 静态链接库，以及 libqlearning.so 或者 libqlearning.dylib（OS X 下）动态链接库。结合 include 文件夹下的头文件，可以很方便地通过上述接口，通过神经网络定义及各类参数，实现任何 Arcade Learning Environment 游戏的 Q 学习。

qcli 前端

下面介绍 qcli 前端用户接口。qcli 是使用 libqlearning ，对其进行的一个简单封装。其功能为使用指定游戏的 rom 文件、神经网络定义，通过 libqlearning 训练 Q 网络模型。或是利用已训练的网络模型，对于指定游戏进

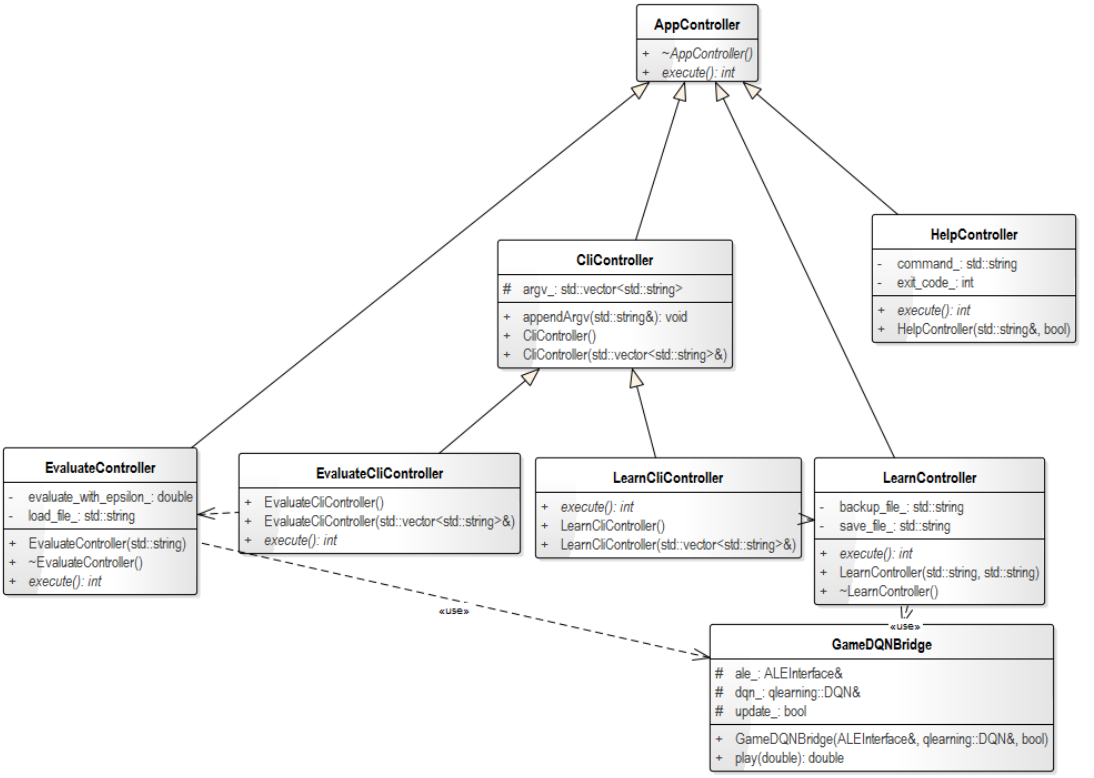
行决策。这一接口同时通过 Arcade Learning Environment 库的 GUI 功能，使得游戏过程能够展现出来。

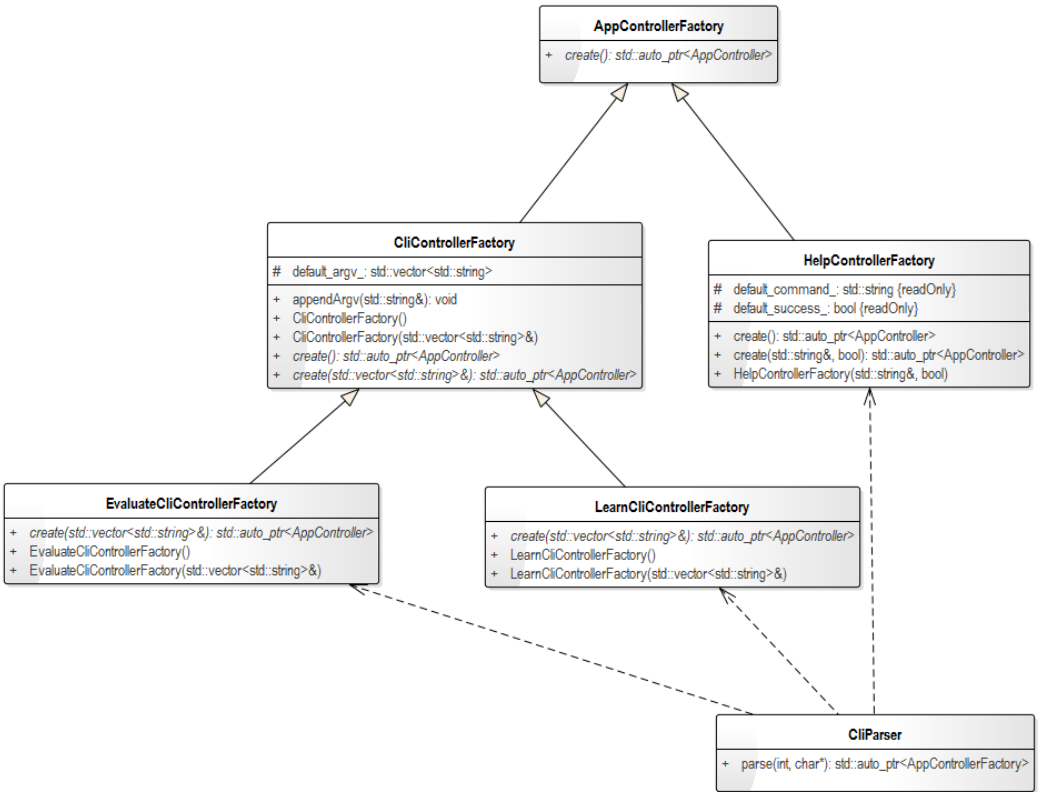
GameDQNBridge 是 Arcade Learning Environment 库以及 DQN 类之间的 Bridge。创建一个包含 ALE 游戏控制器、一个配置好了的 DQN 对象的实例，指定是否训练神经网络。此时只需执行 play 方法，它会重复利用 DQN 实例对于游戏控制器返回的状态进行决策，并向游戏控制器发送操作，直至游戏结束。在 update 设定为 true 时，会不断将游戏控制器返回的回放信息反馈给 DQN 实例，训练并修改这一神经网络。

qcli 包含了 AppController 抽象类、其各类实现以及工厂类，以及一个通过命令行参数，返回工厂实例的 CliParser 类。AppController 为简单的 Controller 模块模型实现，在调用 execute 时会被执行。CliController 抽象类为类似于 Strategy 设计模式，继承于 AppController，可以对其传入命令行参数处理。

LearnCliController、EvaluateCliController 继承于 CliController 虚类，它们可以处理命令行中传入的参数，将其转化为程序结构化数据，产生正确的 LearnController 或 EvaluateController 实例，或在命令行参数错误时产生 HelpController 实例。这一系列实例都继承于 AppController 抽象类，有着同样的虚函数接口，可以通过 execute 执行。

HelpController 的实例在执行虚函数 execute 时，将会输出程序的帮助信息，返回设定的返回值。而 LearnController 会对于设定的输出模型文件参数，通过上述的 GameDQNBridge 类的实例，训练神经网络模型，并进行保存。EvaluateController 会对于设定的输入模型文件参数，读取神经网络模型同样使用 GameDQNBridge 对于指定游戏进行决策，在开启 GUI 模式时展现游戏过程，最终输出游戏结果。


AppController
+ ~AppController()
+ execute(): int
CliController
argv_: std::vector<std::string>
+ appendArgv(std::string&): void
+ CliController()
+ CliController(std::vector<std::string>&)
HelpController
- command_: std::string
- exit_code_: int
+ execute(): int
+ HelpController(std::string&, bool)
EvaluateController
- evaluate_with_epsilon_: double
- load_file_: std::string
+ EvaluateController(std::string)
+ ~EvaluateController()
+ execute(): int
EvaluateCliController
+ EvaluateCliController()
+ EvaluateCliController(std::vector<std::string>&)
+ execute(): int
LearnCliController
+ execute(): int
+ LearnCliController()
+ LearnCliController(std::vector<std::string>&)
LearnController
- backup_file_: std::string
- save_file_: std::string
+ execute(): int
+ LearnController(std::string, std::string)
+ ~LearnController()
«use»
«use»
GameDQNBridge
ale_: ALEInterface&
dqn_: qlearning::DQN&
update_: bool
+ GameDQNBridge(ALEInterface&, qlearning::DQN&, bool)
+ play(double): double


AppControllerFactory
+ create(): std::auto_ptr<AppController>
CliControllerFactory
default_argv_: std::vector<std::string>
+ appendArgv(std::string&): void
+ CliControllerFactory()
+ CliControllerFactory(std::vector<std::string>&)
+ create(): std::auto_ptr<AppController>
+ create(std::vector<std::string>&): std::auto_ptr<AppController>
HelpControllerFactory
default_command_: std::string {readOnly}
default_success_: bool {readOnly}
+ create(): std::auto_ptr<AppController>
+ create(std::string&, bool): std::auto_ptr<AppController>
+ HelpControllerFactory(std::string&, bool)
EvaluateCliControllerFactory
+ create(std::vector<std::string>&): std::auto_ptr<AppController>
+ EvaluateCliControllerFactory()
+ EvaluateCliControllerFactory(std::vector<std::string>&)
LearnCliControllerFactory
+ create(std::vector<std::string>&): std::auto_ptr<AppController>
+ LearnCliControllerFactory()
+ LearnCliControllerFactory(std::vector<std::string>&)
CliParser
+ parse(int, char*): std::auto_ptr<AppControllerFactory>

成员分工

孙伟峻

阅读参考文献中的各类材料，仔细地学习了各论文、文档、代码中的思想，并进行了整理归纳。在组会中向小组成员详细地介绍了算法实现的原理及方法，使得项目开发能够顺利进行。在实现遇到困难时，发现了算法实现中的错误。并在参考文献中，一份代码处获得了利用随机取样过程状态，提高算法效率的灵感，提高了算法的效率，使得在给定的时间内，程序能够给出较优的结果。

冯冠宇

完成了 qlearning 库，实现了项目代码与 caffe 库还有 Arcade Learning Environment 库的交互。阅读了 caffe 的文档以及各种样例程序，学习了使用 caffe 搭建卷积神经网络的方法以及 caffe 的 C++ 接口定义，并实现了 DQN 类，将 caffe 进一步封装，供主程序调用。学习其他人实现的开源项目的基础上实现了 Q-Learning 算法在 caffe 环境中的具体实现方法。还在服务器中对程序进行测试，调整了常数，监视学习的成果以及进展。

刘家昌

仔细阅读项目中使用的各个库的文档，解决了各类依赖关系以及平台 bug，在组员的 Linux 以及 Mac OS X 中搭建了稳定的构建环境，使得各成员在空闲时间均可投入开发当中，而不受到平台的干扰。完成了项目的 cmake 构建文件，使得项目文件整洁可维护，产生了可以复用的动态运行时库，并能够在 caffe 支持的各个平台上顺利编译。完成了 qcli 前端，确定了项目的大致架构以及代码风格要求。对代码进行了性能调优，修改了其他成员的部分代码，加快了机器学习的速度。

参考文献

Human-level control through deep reinforcement learning

Playing Atari with Deep Reinforcement Learning

http://caffe.berkeleyvision.org/

http://www.arcadelearningenvironment.org/

https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner

https://github.com/muupan/dqn-in-the-caffe

http://www.infoq.com/cn/news/2014/10/deepmind/