Compact CFR

Eric Jackson

Abstract

This paper describes a collection of ideas that allow large games of imperfect information to be solved with counterfactual regret minimization (CFR) using little memory. We replace the regret matching component of CFR with a simple approach known as “follow-the-leader”. This helps us quantize the regret values computed in CFR to a single byte. We also investigate not maintaining the accumulated strategy, which saves additional memory. Ultimately, our collection of techniques allows CFR to be run with only 1/16 of the memory required by classic approaches. We present experimental results on poker.

Counterfactual Regret Minimization

Counterfactual regret minimization (CFR) (Zinkevich et al. 2007) is a technique for solving large games that has become the dominant approach in the field of computer poker. It takes regret minimization techniques that have been traditionally applied to normal-form games and adapts them to work efficiently on extensive-form games like poker.

Regret minimization has been applied to compute solutions to normal-form games through a form of self-play. Over a series of iterations each player computes a strategy. The average of these strategies can be shown to converge to a Nash equilibrium. For a normal-form game, a strategy is just a probability distribution over the actions available to the player, and the particular regret minimization algorithm employed determines the probability of each action based on its current “regret”. Regret is a quantity that represents how much the player could have gained over previous iterations by playing that action as opposed to playing the actual sequence of actions that he did:

\[ R(a) = \sum_{t=1}^{T} (u(a) - u(a^t)) \]  

(1)

Here \( t \) is the iteration index, \( a^t \) is the action actually taken at iteration \( t \), and \( u \) is the utility function. We are often interested in the overall regret which is the regret of the best action we could have taken with the benefit of hindsight:

\[ R = \max_{a^*} \sum_{t=1}^{T} (u(a^*) - u(a^t)) \]  

(2)

The key insight of CFR is that we can adapt this approach to extensive-form games by applying regret minimization independently at each information set. This yields notions of the regret \( R(I) \) at an information set \( I \) and the regret \( R(I, a) \) of an action \( a \) at information set \( I \). (We’ll often omit the argument \( I \) if no confusion results.) In CFR on each iteration at each information set, a player computes the new distribution over the available actions using “regret matching” (Blackwell 1956). The probability assigned an action is proportional to that action’s positive regret. We designate the positive regret of an action \( a \) with \( R(a)^+ \):

\[ R(a)^+ = \begin{cases} R(a) & \text{if } R(a) > 0; \text{otherwise it is zero} \\ \end{cases} \]  

(3)

Regret matching specifies that probabilities are assigned in proportion to positive regret:

\[ \sigma^t(a) = R^{t-1}(a)^+ / \sum_{b \in A} R^{t-1}(b)^+ \]  

(4)

The original formulation of CFR is known as “Vanilla” CFR and requires a complete walk of the game tree on each iteration. Subsequent work has proposed a number of variants of CFR that incorporate sampling and converge much faster on large games. The particular form of sampling that we use is known as external sampling (Lanctot et al. 2008). When external sampling is applied to poker we sample from the chance events (the hole cards for each player and the community cards) and from the distribution over the opponent’s actions, but we fully evaluate each action of the target player. As with all of the sampling variants of CFR, updates are made to only a portion of the game tree on each iteration.

Based on work of Oskari Tammelin, many current implementations of external sampling use a variant known as pure external CFR which allows representing the parameters of the system (the regrets and the accumulated strategy) as integers rather than floating point values (Gibson 2014).

Follow-The-Leader

In standard CFR implementations, regret matching is performed at each information set to determine the next strategy. One very simple alternative to regret matching is known as “follow-the-leader” (see, e.g., (Cesa-Bianci and Lugosi 2006)). Under this scheme, the entire probability mass is assigned to the action with the highest regret. (Ties may be broken arbitrarily.) In the context of solving a two-person
zero-sum normal-form game, “follow-the-leader” is equivalent to fictitious play. Fictitious play is not a no-regret algorithm (it does not exhibit vanishing per-round regret); despite this, the average strategy learned by fictitious play does converge to a Nash equilibrium.

Our first idea is to apply “follow-the-leader” on a per-information-set basis (as in CFR) and within an external sampling implementation. By operating on each information set independently, we get something different from fictitious play which updates the entire strategy on each iteration to be a best-response to the opponent’s average strategy to date. Because “follow-the-leader” is not a no-regret algorithm, we lose the theoretical guarantees of CFR, and because it is not identical to fictitious play, results that apply to fictitious play do not pertain to us. Nevertheless, we forge ahead.

As was pointed out by an anonymous reviewer, a simple modification to “follow-the-leader” would allow us to recover the no-regret guarantee. This new algorithm known as “perturbed-follow-the-leader” was originally proposed in (Hannan 1957) and is more recently analyzed in (Kalai and Vempala 2005). In our context, the idea is to simply perturb the regrets with uniform random noise prior to selecting the best action.

In practice, we dispense with the random noise because we have achieved good results empirically with regular (unperturbed) “follow-the-leader”. But in future work we would like to evaluate whether we can achieve better results on any games of interest using “perturbed-follow-the-leader”.

One advantage of the “follow-the-leader” algorithm is that it allows us to represent regrets more simply. Since all that matters is what action has the highest regret, it suffices to store for each action the offset from the best regret. In other words, we represent the best action(s) with a regret offset of zero, while all other actions have a positive offset indicating how far their regret is below the regret of the best action. Since these offsets are non-negative, we can use an unsigned data type to represent them.

**Experiments**

We compared “follow-the-leader” to regret matching by looking at exploitability over time of a simple poker variant. “Exploitability” measures the worst-case loss of a strategy; i.e., how much it would lose (in expectation) to a best-response. We report results in units of milli-big-blinds per game (mbb/g) which is thousandths of a big blind per game (“game” here means a single hand of poker).

The game we employ is a variant of No-Limit Texas Holdem. In our variant, there are only two streets, with a maximum of two bets on each street and one bet size (full pot). A full 52-card deck is used and no card abstraction. Figure 1 shows the results. “Follow-the-leader” actually starts out ahead, with lower exploitability in the early iterations, but in later iterations, it is a bit worse, although only by a negligible amount.

**Regret Compression**

In order to build larger systems with a fixed memory budget, we explored the use of quantization in order to represent regrets with a single byte. The representation of regrets as unsigned quantities facilitates this compression.

We explored a variety of quantization functions with the property that regrets for better actions (i.e., values closer to zero) are represented with greater precision, while regrets for worse actions are represented more coarsely. We ended up with a step function that maps raw regret offsets from $0 \ldots 13140$ into one byte unsigned values (i.e., $0 \ldots 255$). Note that in doing so we are implicitly imposing a ceiling of 13140.

We can think of quantization as determining buckets of raw regret values that are equivalent in the sense that they map to the same compressed regret value. Taking this viewpoint, we can see that there is a danger which is that if the magnitude of regret increments is too small, a regret may get stuck in a certain bucket despite repeated small updates that in aggregate ought to drive it to a different bucket.

We address this with stochastic rounding. We use a compression function that maps the uncompressed regret value to a real value between 0 and 255. We then stochastically round to one of the two nearest whole values which becomes the compressed regret value. For example, we may map a raw value of 1300 to a real value of 174.4. With probability 0.6 we round that value to 174 and with probability 0.4, we round that value to 175.

The regret ceiling imposed by a compression scheme like this can be problematic when we move to games with a larger variance in payoffs. In a poker context, when we move from shallow-stacked limit games to deeper-stacked no-limit games, the variance of the counterfactual values increases and a regret cap increasingly becomes problematic. We have found it advantageous to scale all regret updates by a constant factor less than 1.0 (we used 0.05). As we did before, we employ stochastic rounding so that the scaled updates do not differ in expectation from the raw unscaled updates.

Other recent systems use compression to store regrets more compactly: e.g., CFR+ (Tammelin et al. 2015). One
difference in our system is that due to our use of external sampling we need fast random access to regrets. CFR+, which does a full traversal of the tree on each iteration can use a streaming compression technique with a variable-size encoding, whereas we must use a fixed-size encoding to enable random access.

Experiments
We evaluated whether regret compression would degrade performance by looking at exploitability over time when running CFR with and without regret compression. We used the same game that was employed in the previous experiment.

As you can see from figure 2, regret compression does cause some degradation in how long one must run CFR to reach a given level of exploitability. At the endpoint of this experiment, the compressed system had not fully asymptoted and was still improving.

Current Strategy
In normal CFR, in addition to the regrets, the accumulated strategy is maintained. Recall that it is the average strategy that is guaranteed to converge to equilibrium. Researchers have experimented previously with using only the current strategy (typically at the last iteration) despite the lack of guarantees. See, for example, (Tammelin et al. 2015). If we are willing to do this, we can dispense with maintaining the accumulated probability.

Some implementations have used purification of the strategies produced by CFR (Brown, Ganzfried, and Sandholm 2015). Purification maps a mixed strategy into a pure strategy by giving the entire probability mass at any information set to the action with the highest probability in the given mixed strategy. Purification is often employed because it produces systems that require less memory at runtime. For example, we can represent whether an action has probability 1 or 0 with a single bit. But it is also sometimes claimed that purification produces systems that perform better empirically.

Note that under the follow-the-leader approach, if we use the current strategy from the last iteration we get a pure strategy automatically because the current strategy is always a pure strategy. This will give us the desired runtime memory savings.

Experiments
We investigated the performance of the current strategy as compared to the normal average strategy. For this experiment we employed a two-street poker game with a single bet size and a maximum of four bets per street. Instead of exploitability, we focused on head-to-head performance against a baseline known good strategy. The baseline we used was a near-equilibrium strategy; it was computed with a perfect abstraction and measured to have exploitability of 2.1 mbb/g. The experimental system used a perfect-recall abstraction with 16,468 flop buckets.

The results are shown in figure 3. Note that the current strategy generally outperforms the average strategy throughout the entire run. The current strategy starts out with a small loss from a very early iteration, but doesn’t improve by much. The average strategy starts out rather poorly, but improves to almost the level of the current strategy.

For comparison, we show the exploitability (in the abstract game) of the average strategy.

Blending Checkpoints
A less radical alternative to using the current strategy is to checkpoint the regrets at several points in training and compute a mixed strategy that is the average of the current strategy at each checkpoint (or perhaps some subset of the more recent checkpoints). This still saves us from the burden of maintaining the accumulated strategy in memory. Checkpointing has a cost in terms of storage, although it is typical of large systems to maintain at least one checkpoint in order to guard against unanticipated system failures of various sorts.

The simplest way to combine multiple checkpoints is to compute an equally weighted average of the current strategies corresponding to the regrets at those checkpoints. Note, though, that this doesn’t mimic how we accumulate probabilities in CFR. In CFR, when we accumulate probabilities of actions, the probabilities are weighted by the “reach”
probabilities; i.e., the probability that the information set will be reached by the given player. In the case of perfect recall abstractions we can certainly do the right thing and weight by the reach probabilities, but this becomes prohibitively expensive in the case of imperfect recall abstractions that are now typically used for large systems.

If we elect to use multiple checkpoints, we may still wish to produce a purified strategy. We can create one in the normal way, or we can instead produce one via sampling. In the normal approach, if the mixed probabilities are 0.5, 0.3 and 0.2 we would always assign the entire probability mass to the first action in the purified strategy. In the sampling approach we instead sample from this distribution; we assign the entire probability mass to the first action with probability 0.5, to the second action with probability 0.3 and so forth. Although we still end up with purified strategies, we may end up with effectively more mixing as similar game states will tend to end up with different selected actions.

Experiments
We investigated the exploitability of systems built by computing a mixed strategy which is an equally weighted blend of the current strategies from the last \( N \) checkpoints. We used the same game as in our initial experiments. Checkpoints were computed every 800 million iterations. CFR was run for 40.8 billion iterations with 51 checkpoints. The exploitability of the baseline system in which the accumulated strategy was maintained was 10.14 mbb/g. The results are in figure 4. If we blend all 51 checkpoints, we achieve exploitability of 21.7 mbb/g.

Note that using only the last checkpoint yields a strategy that has very poor exploitability. Having said that, head-to-head performance may still be strong as we saw in the previous experiment, as long as we are competing against agents that do not adapt or do not adapt well.

Summary
Standard implementations of CFR require sixteen bytes of memory for each action at every information set. Regrets and accumulated probabilities are represented with double-precision floating-point values.

With pure external CFR the requirement is lessened to eight bytes as we can represent each of the needed quantities with four-byte integer values.

With the combination of approaches described above, we reduce our memory requirement to a single byte.

Distributed Implementation
We use a distributed implementation of external sampling that allows us to combine the memory and CPU of multiple machines, rather than being limited by the capacity of a single machine. Distributed implementations of CFR have been described in our earlier work (Jackson 2012, Jackson 2013), but our current system is most similar to that described in (Brown, Ganzfried, and Sandholm 2015).

To enable distributed processing we divide the game tree so that each machine can perform a portion of a single iteration of external sampling, operating as independently as possible from every other machine. Specifically, we divide the game tree into partitions based on public information, namely the cards on the flop. A designated master machine manages the preflop portion of the game. By partitioning the tree on public information we guarantee that no information set spans multiple partitions.

With this setup, interaction between the master machine and the workers is fairly minimal. At the beginning of an iteration, the master traverses the preflop portion of the game tree and tells all the workers which opponent hands reach which flop-initial betting states. Each worker can perform a standard external sampling iteration over the postflop portion of the game tree. At the end of the iteration, each worker sends back to the master the counterfactual value achieved by the target player’s preflop hand at each flop-initial state. The master completes the iteration for the preflop portion of the game tree.

In classic external sampling we sample a pair of private cards for each player (known as the “hole” cards) and a single set of public cards (known as the “board”). We traverse the betting tree considering each action for the target player while sampling only a single action for the opponent. With the classic approach we have encountered the problem that each iteration is too fast and the system ends up spending the bulk of its time on network overhead. To achieve better CPU utilization we increase the amount of work performed in each iteration in two ways: first by evaluating multiple preflop hands for each player, and second by sampling multiple boards. The number of boards and hands evaluated can be tuned to individual needs.

This system is being used to build Slumbot 2016, our entry to the 2016 Computer Poker Competition. For compute resources, we are using eleven r3.4xlarge Amazon EC2 instances, each having 122 GB of RAM. Our abstraction has \( 3.9 \times 10^{11} \) information sets and \( 9.7 \times 10^{11} \) information-set action pairs.

Experiments
We conducted experiments comparing a distributed implementation to a non-distributed implementation. For these experiments we looked at head-to-head performance. The
baseline system at a future iteration was used as the reference system and all other systems evaluated by head-to-head performance against that system.

For these experiments we used a larger no-limit poker game with 200 big blind stacks, four bet sizes and a card abstraction with 800 buckets on the river and 8000 buckets on the flop and turn. The computation was distributed across four machines.

Figure 5 shows the results. The distributed system lags the baseline system in the later iterations, but only by about 2 mbb/g.

References


Hannan, J. 1957. Approximation to bayes risk in repeated play. Contributions to the Theory of Games, III.


