Шахматные программы и влияние на них Deep Learning алгоритмов

Аржанцев Андрей, Даниэль Юлий, Савинский Всеволод

1 Немного о шахматах

Шахматы — одна из самых древних и самых популярных настольных игр, также это вид спорта, имеющих огромное количество фанатов и любителей по всему миру. Феномен этой игры заключается в том, что несмотря на кажущуюся до банальности простой систему правил, научиться играть в шахматы идеально человечеству, кажется, почти невозможно. Профессиональные шахматисты с раннего детства играют, изучают различные тактитки и аспекты игры, но за сотни лет людям так и не удалось достичь совершенства в стратегиях.

Когда появились компьютеры, почти сразу любители шахмат пытались использовать их для анализа игры — еще в 1957 году в СССР была создана первая программа, умеющая играть в полноценные шахматы. Постепенно программы совершенствовались, и в 1997 году суперкомпьютер Deep Blue смог победить многолетнего чемпиона мира Гарри Каспарова (отыгравшись за проигранный матч в 1994-ом). Примерно с этого времени люди уже стали неспособны противостоять лучшим шахматным программам.

Рис. 1: Партия Гарри Каспарова против суперкомпьютера Deep Blue

Несмотря на превосходство компьютеров над людьми, индустрия продолжала и продолжает развиваться — программы конкурируют уже не с людьми, а друг с другом. Некоторой революцией в мире шахматных движков стало появления программы AlphaZero от DeepMind. Программа, которая впервые начала масштабно использовать нейросети в своих алгоритмах, почти сразу после релиза в 2017 году смогла победить самый сильный на тот момент шахматный движок Stockfish, чем поразила все шахматное сообщество. На самом деле если раньше преимущество компьютеров сказывалось лишь в вычислительной мощности, дающей возможность перебирать миллионы последовательностей ходов за секунды, то теперь индустрия постепенно смещается в сторону deep learning алгоритмов, чтобы программы совмещали как преимущество в вычислительной мощности, так и преимущество в оценке позиции, полученной с помощью предобученных нейронных сетей.

Так, например, самый мощный и оптимизированный шахматный движок Stockfish, относительно недавно успешно добавил нейронную сеть в свою новую версию, об этом нововведении и самой нейросети будет первая часть.

Во второй же части мы разберем статью, в которой авторы предлагают улучшения уже для

другого шахматного движка — как раз нашумевшего AlphaZero. Если в первой части будет скорее полноценное описание нейронной сети, то во второй части мы также затронем второй важный аспект текущих версий лучших шахматных программ, а именно дерево перебора ходов и различные связанные с ним алгоритмы.

2 NNUE

2.1 Вступление

NNUE (Efficiently Updatable Neural Network) — нейронная сеть, придуманная в 2018 году для японской игры Сёги. В 2020 году её же использовали в Stockfish-е для улучшения оценки позиции в игре, в общем-то нейронная сеть оказалось применимой и для многих других настольных игр.

Добавление NNUE в Stockfish стало одним из главных нововведений за всю историю существования программы — теперь Stockfish перестала уступать таким движкам, как LeelaChess0 или AlphaZero в, как говорят, positional understating, то есть в глобальной оценке позиции на доске. Сравнивая версии Stockfish с NNUE и без нее, разработчики программы получили увеличение ELO на целых 80 пунктов (для сравнения — такой же разрыв в рейтинге у первого и пятого шахматистов мира, либо у второго и 29-ого). Также после нововведения Stockfish победил в главном турнире шахматных движков TCEC, одолев в финале LeelaChessZero со счетом 53:47.

Кажется, эта новая нейронная сеть должна быть какой-то очень сложной, раз Stockfish с ней стал настолько сильнее. На деле же это правда лишь отчасти — реализация NNUE внутри движка действительно задача тяжелая, но вот понимание принципа работы нейросети не требует от пользователей каких-то чрезмерных знаний в области deep learning.

2.2 Постановка задачи

Начнем с того, какую задачу вообще решает наша нейросеть. Тут все просто — на вход ей поступает позиция на шахматной доске и цвет фигур, делающих следующий ход, а на выходе ожидается число — оценка этой позиции, где 0, -10000, $+10000$ (такие границы используются в Stockfish) — ничья, заведомый проигрыш и выигрыш соответственно.

Желая реализовать нейросеть для решения такой задачи, основной проблемой является необходимость обучения на большом сэмпле — действительно, позиций в шахматах огромное количество, многие совсем не тривиальные для анализа. Поэтому необходимым требованием во всех таких шахматных нейросетях является скорость обучения, из-за которой зачастую приходится жертвовать точностью вычислений.

2.3 Описание нейросети

Бесхитростную версию можно описать очень просто: входные данные, несколько линейных слоев, после каждого слои с функций активации, результирующий слой — готово.

Линейные слои самые обычные: $output = W * input + b$. Единственным замечанием является то, что зачастую придется прибегать к разреженным векторам и матрицам, используя так называемые Sparse Linear Layer, что впрочем не является большой проблемой. Количество слоев для NNUE хочется иметь не очень большим (в Stockfish их 4 без учета параллельных слоев) — как оказалось, основная часть обучения заключается в прохождении первого большого слоя, тогда как последующие лишь немного улучшают точность, тогда как слишком много слоев могут эту точность и ухудшить из-за уже вычислительность неточности, ну и еще это все-таки ресурсозатратно.

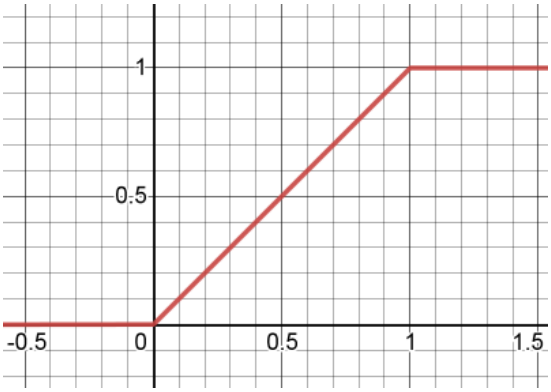


Рис. 2: ClippedRelu

Функции активации могут быть различные — авторы статьи в качестве примера приводят ClippedRelu: $y = \min(\max(x, 0), 1)$, как самую простую и быстровычислимую (она и используется в StockFish), хоть и не дифференцируемую в 0 и 1.

Альтернативный вариант — сигмоида $y = 1/(1+e^{-kx})$, но это довольно затратно, к тому же не применимо для целочисленных данных (позже будет подробно описано, как и почему мы работаем только с целыми числами). Поэтому вместо нее используют ее аппроксимацию, отнормированную так, чтобы значения были целыми.

$$\begin{cases} x > 0 & : \quad \left\lfloor \frac{(\min\{|x|,\ 127\} - 127)^2}{256} \right\rfloor \\ x <= 0 & : \quad 126 - \left\lfloor \frac{(\min\{|x|,\ 127\} - 127)^2}{256} \right\rfloor \end{cases}$$

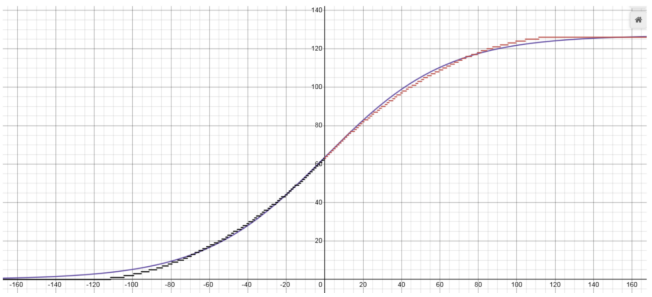


Рис. 3: Quantmoid4 - красная, Sigmoid - синяя

Иногда могут оказаться полезными pooling-слои, сжимающие данные в текущем слое, дабы не работать с огромными размерностями. Можно использовать, например average pooling или max pooling слои (из названия понятно, как они сжимают данные), в Stockfish-е же для сжатия используется Product Pooling — значения данных перемножаются и кладутся в выходной вектор (хорошо работает только для сжатия в 2 раза, то есть только перемножая по два числа). Несмотря на то, что такой слой в целом в ML применяется редко, но в конкретной задаче оказался наиболее подходящим.

2.4 Входные данные

Отдельно стоит сказать про преобразование входных данных — как нам превратить шахматную позицию в численный вектор. Простым, но эффективным решением является представление позиции в виде массива, проиндексированного всеми возможными тройками (координата клетки, фигура, цвет), а значения массива соответсвует наличию такой фигуры в такой клетке — 0 или 1. В итоге получается массив длины $6 \cdot 2 \cdot 64 = 768$. Можно заметить, что массив будет почти полностью забит нулями — как раз из-за этого мы и хотим использовать разреженные линейные слои.

К тому же такой подход помогает нам и с другой проблемой — представим две последовательные позиции в одной игре, они же почти не отличаются, тем не менее мы тратим много ресурсов, пересчитывая линейный слой для каждой из них отдельно. В нашем же подходе мы можем обучаться на сэмпле из последовательных позиций, не делая много лишних операций — будем запоминать не позиции, а последовательности ходов — тогда легко будет восстановить в каких местах поменялся входной вектор с 0 на 1 или с 1 на 0 (таких мест обычно 2 или 3, но есть один ход, меняющий наш вектор сразу в 4 местах — рокировка) и тогда из первоначального результата линейного слоя нам достаточно будет просто вычесть/добавить 2–4 столбца из матрицы W. В NNUE такая идея реализована с помощью аккумулятора — специальный класс, который запоминает ходы в игре, переделывает их в множество координат входного массива, которые менялись со временем на $+1$ и на -1, после чего считает выходные данные линейного слоя уже по этим координатам и предыдущему выходу. И небольшое замечание — в работе с числами с плавающей точкой такое последовательное вычитание и прибавление вместо честного подсчета приводит к значительным ошибкам в точности вычислений, которая накапливается с каждым следующим ходом — но мы это починим, когда дело дойдет до реализации нейросети исключительно в области целых чисел!

Альтернативный способ задать шахматную позицию чуть более сложный, но, как оказалось, более эффективный и являющийся типичным во многих движках — HalfKP. Суть похожая, но

теперь мы индексируем массив всевозможными четверками (позиция нашего короля [отсюда и KP], координата клетки, фигура, цвет) для всех фигур, отличных от короля. Как оказалось, ценную информацию несет не только расположение фигуры на доске, а также ее расположение относительно своего короля. Вторая идея в таком подходе — нумерация координат доски вертикально симметричная для разноцветных фигур, а также разделение первого линейного слоя для двух цветов. Действительно, раньше матрица W совершенно не должна была быть симметричной для двух цветов, теперь же мы будет оценивать одной и той же матрицей W ситуацию заведомо одинаково. Реализация вызывает несколько сложностей — нам теперь надо будет поддерживать сразу 2 аккумулятора для разных цветов, ходы королём станут чуть более затратными, после двух независимых линейных слоев потом еще соединять два слоя в один, но как показывает практика — это допустимые издержки.

2.5 Квантилизация

Квантилизация — еще одна интересная идея, которая используются в алгоритме. Она позволяет перейти от чисел с правающей точкой к целым числам и нужна для оптимизации скорости и улучшения точности работы нейронной сети, так как современные процессоры оптимизированны под работу именно с целыми числами, а точные вычисления с ними позволяют не накапливать ошибку от слоя к слою.

Во многих современных процессорах есть SIMD инструкции, которые позволяют работать сразу с несколькими интами одновременно. Например, если работать с 8-мибитными интами, то на некоторых процессорах можно обработать 64 операции с ними за раз. Поэтому обычно стараются использовать 8-мибитные или 16-тибитные инты.

Соответственно использовать ClippedRelu без изменений становится бессмысленно, так как для целых чисел ClippedRelu выдает лишь два значения. Поэтому сжимаемый интервал в ClippedRelu раздвигается до $0 \dots 127$, чтобы заодно можно было использовать значения в виде восьмибитных интов.

Теперь посмотрим, как будет работать линейный слой. Пусть у весов и входящего вектора коэффициенты масштабирования s_W и s_A соответственно. Так как это матричное умножение, рассмотрим одно из слагаемых суммы:

$$x \cdot w + b = y$$
$$((s_A \cdot x) \cdot (s_W \cdot w)) + (b \cdot s_A \cdot s_W) = (y \cdot s_A) \cdot s_W$$
$$(((s_A \cdot x) \cdot (s_W \cdot w)) + (b \cdot s_A \cdot s_W))/s_W = (y \cdot s_A)$$

Таким образом надо умножать bias на коэффициент $(s_A \cdot s_W)$, а чтобы получить необходимый $(y \cdot s_A)$ нужно поделить результат на s_W.

Это лишь примеры того, как можно уйти от использования чисел с правающей точкой. На гитхабе можно ознакомиться с работающим фрагментом кода, использующим эти оптимизации.

2.6 Процесс обучение

Теперь о том, как нейронная сеть у нас обучается. Для этого ей нужна какая-то обучающая выборка. Например, игры Stockfish-а с самим собой. Для каждой позиции у Stockfish-а уже есть какая-то своя функция оценки позиции, она нам в любом случае понадобится. Будем говорить, что оценка позиции X текущим Stockfish есть S_X, результат партии $F_X - 1$ победа, 0 поражение, 1/2 ничья.

Предварительно изучив данные об оценке различных позиций программой, было найдено, что функция распределения этой оценки имеет форму сигмоиды , поэтому чтобы попасть на интервал [0,1] самым честным будет заменить $S_X = sigmoid(S_X)$ и тоже самое сделать с оценкой, полученной в обучении (пусть T_X).

В качестве функции ошибки, которую мы будем оптимизировать, лучше всего будет взять кросс-энтропию, как это делается в задачах классификации (по сути у нас такая задача и есть), при этом стоит учитывать как ошибку в сравнении с результатом игры, так и ошибку в сравнении с текущей версией. Действительно, эти две ошибки хороши именно вместе, ведь именно случаи расхождения оценки позиции текущей версии и результата игры мы и хотим исправить. По итогу имеем такую формулу:

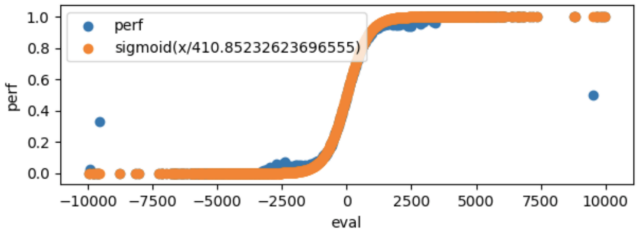


Рис. 4: оценка позиций Stockfish — синий, Sigmoid — оранжевый

$$S_X, T_X = sigmoid(S_X), sigmoid(T_X)$$

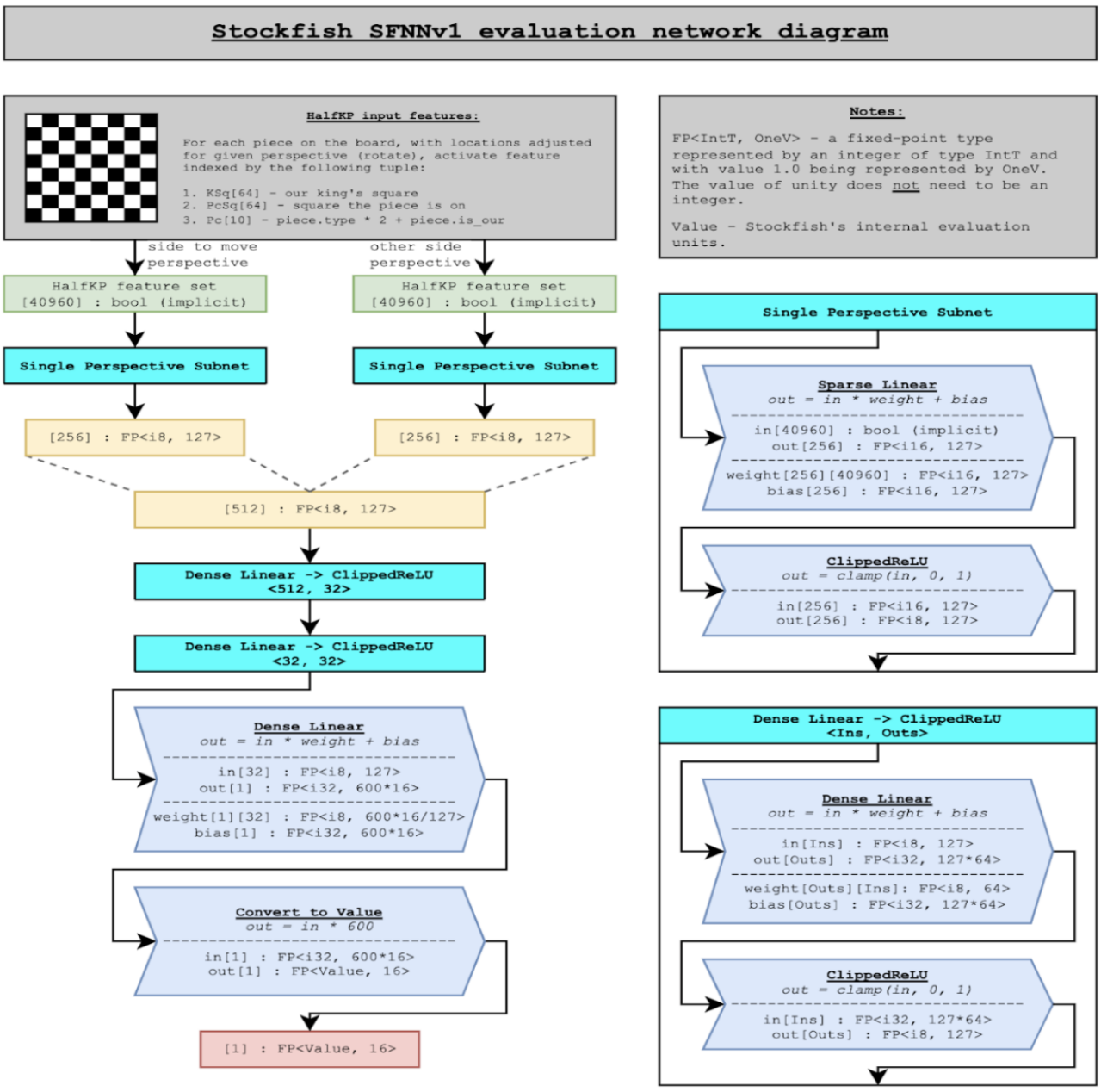
$$\begin{aligned} loss_{eval} = &(S_X \cdot \log(S_X + \varepsilon) + (1 - S_X) \cdot \log(1 - S_X + \varepsilon)) \\ &- (S_X \cdot \log(T_X + \varepsilon) + (1 - S_X) \cdot \log(1 - T_X + \varepsilon)) \end{aligned}$$

$$\begin{aligned} loss_{gameres} = &(F_X \cdot \log(S_X + \varepsilon) + (1 - F_X) \cdot \log(1 - S_X + \varepsilon)) \\ &- (F_X \cdot \log(T_X + \varepsilon) + (1 - F_X) \cdot \log(1 - T_X + \varepsilon)) \end{aligned}$$

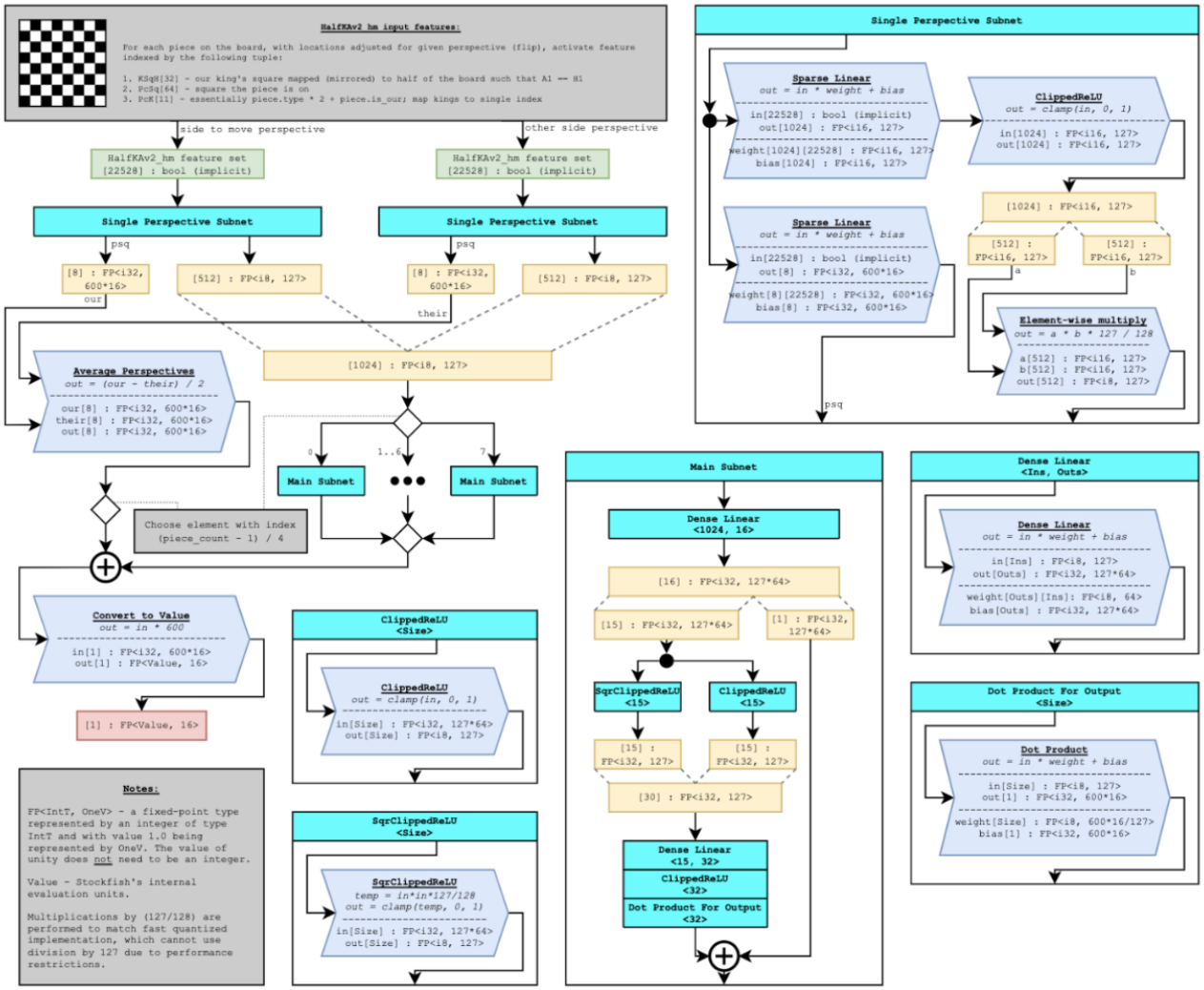
$$loss = \lambda \cdot loss_{eval} + (1 - \lambda) loss_{gameres}$$

2.7 Схемы

Собрав все вместе, мы как раз получим вот такую схему нейроести - такой она и была в 2020 году в релизе Stochfish12!



Впрочем с 2020 в Stockfish было еще много доработок и оптимизаций, схема на текущий момент выглядит так



3 Оптимизации алгоритма обхода графа игры для Alpha Zero

В этой части рассмотрим несколько оптимизаций алгоритма обхода графа игры для Alpha Zero, как уже было сказано ранее, шахматного движка, работающего на основе нейронной сети (как, впрочем, и все шахматные движки) и выпущенного DeepMind. Эти оптимизации были опубликованы в статье [2]. Какие-то из этих оптимизаций уже были предложены ранее, какие-то (например, 1) авторы предложили первыми, какие-то ранее уже предложенные оптимизации они соптимизировали еще сильнее (например, 2).

3.1 Оптимизация 1

Если раньше игру представляли как дерево, то алгоритм, предложенный авторами, представляет игру как ациклический ориентированный граф. Таким образом, в одну и ту же вершину графа можно прийти разными путями. Например, если белые первым ходом сделают ход `e2-e4`, черные пойдут `h7-h6`, а белые ответят `d2-d4`, позиция будет такая же, как и после последовательности ходов `d2-d4`, `h7-h6`, `e2-e4`. Значит, разными наборами ходов мы можем прийти в одну и ту же позицию. Получается, что представление игры как ациклического графа помогает нам сэкономить память на хранения дерева (с учётом того, что количество листьев увеличивается экспоненциально от уровня анализа, мы действительно будем экономить кучу памяти) и время на анализ позиций, которые мы уже проанализировали. Для выявления случаев совпадающих позиций авторы считают хэш позиций, чтобы сравнение не занимало слишком много времени. Для того, чтобы поддерживать актуальную информацию на нескольких родителях одной вершины, авторы предлагают специальную backpropagation технику.

3.2 Оптимизация 2

Авторы предлагают обновление terminal-solver (того куска анализа игры, который отвечает за решение позиций, в которых есть форсированная победа, ничья или поражение). Комментарий: эта часть статьи выглядит настолько очевидно, что очень странно, что до этого terminal-solver выглядели не так. Если есть ход, ведущий в достоверно проигрышную (для соперника) вершину, то мы запоминаем это и запоминаем, за сколько ходов мы выигрываем в этом варианте, и поддерживаем минимально возможное количество ходов с гарантированной победой. Если для всех детей достоверно известно, что они приводят к поражению, то поддерживаем максимально возможное количество ходов до поражения. Если же нет победы, а все дети достоверно приводят к поражению или ничьей (и есть хотя бы одна ничья), то мы выбираем ничью. Авторы говорят, что их алгоритм — первый, который гарантирует победу за минимальное количество ходов и первый, который поддерживает использование tablebase positions — позиций из специализированных баз данных, для которых уже посчитано, проигрышные они, выигрышные или ничейные.

3.3 Оптимизация 3

Самая интересная оптимизация, предлагаемая авторами (но уже придуманная другими людьми ранее) — элемент стохастического выбора анализируемых линий. Это будет реализовано следующим образом: с некоторой малой вероятностью мы решаем идти не в ту ветку, в которой по ранее посчитанной информации скорее всего будет содержаться наибольшая награда, а в какую-то другую ветку (выбираем explore вместо exploit, разведывать новое вместо того, чтобы использовать старое). Обычно такой метод используется на этапе reinforcement learning модели, чтобы модель смогла попробовать что-то другое и за счёт этого потенциально найти более хороший вариант и поменять параметры, чтобы в следующий раз лучше оценить эту позицию. Но авторы утверждают, что в ситуации игры с предобученной моделью такой подход

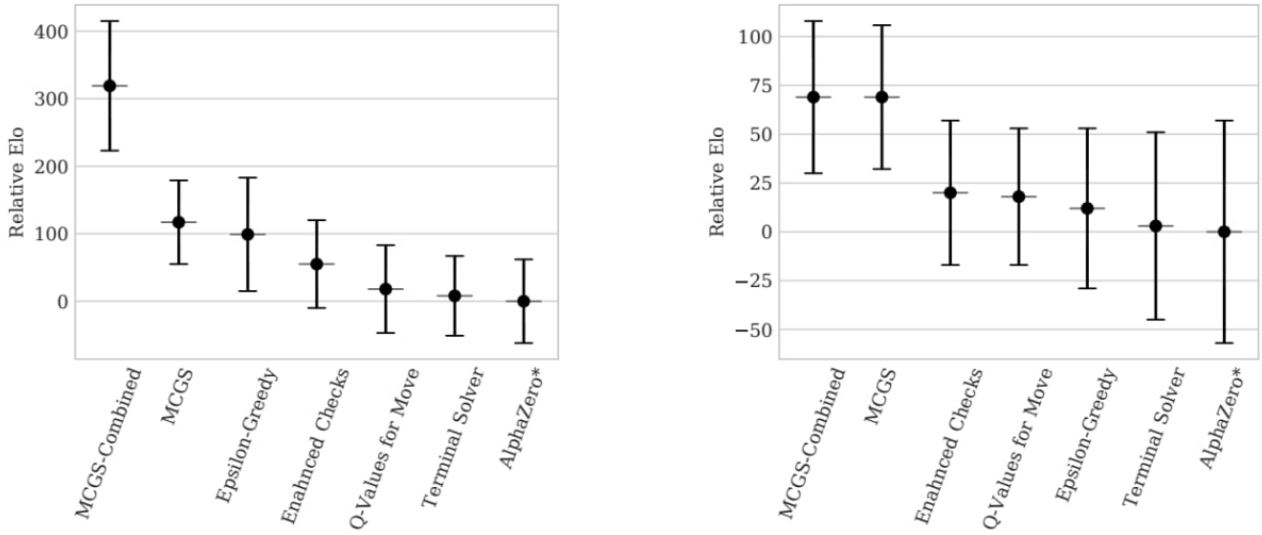
тоже может иметь смысл для преодоления ловушки локального максимума: на первый взгляд кажется, что ход `A` даёт меньшую награду, чем ход `B`, но если углубиться в ход `B`, оказывается, что in the long run мы получим более хорошую позицию.

3.4 Оптимизация 4

Довольно банальная оптимизация, которую часто советуют и людям, играющим в шахматы — при анализе позиции сначала рассматривать более «форсирующие» ходы, то есть ходы, после которых у соперника будет меньше «разумных» вариантов. Например, шахи — соперник будет обязан двинуть короля или заслониться, съедение чужой фигуры — скорее всего соперник будет пытаться съесть эту фигуру в ответ, так как большая часть фигур в шахматах уходит с поля в ходе разменов. Авторы статьи рассматривают эту оптимизацию только для шахов. Таким образом, ходы с шахами рассматриваются раньше, чем другие ходы.

3.5 Вывод

Все вышеуказанные оптимизации авторы применяют по отдельности и все вместе и смотрят, насколько улучшается уровень игры модели. На картинке можно увидеть, насколько улучшается elo-рейтинг alpha-zero, если добавить эти оптимизации.



Комментарий: здесь не была рассмотрена оптимизация Q-Values for Move, потому что её описание требует большого углубления в логику статьи и формулы выбора и не составляет особенного человеческого интереса.

4 Заключение

В ходе работы были рассмотрены различные оптимизации шахматных программ. Как мы поняли, новые идеи по их оптимизации все еще появляются, и эта область еще не исчерпала себя, несмотря на то, что первая победа компьютера над человечеством была 25 лет назад. То есть индустрия все еще развивается, скоро компухтеры захватят мир...

Список литературы

[1] Гитхаб про NNUE

[2] Статья про оптимизации для AlphaZero

[3] Stockfish NNUE wiki

[4] Википедия про компьютерные движки

5 Распределение обязанностей

Андрей Аржанцев – основная работа над темой NNUE (часть 1) и часть работы над отчётом.
Всеволод Савинский – часть работы над темой NNUE (часть 1) и сведение отчёта.
Юлий Даниэль – работа над темой оптимизаций обхода (часть 2) и финальные правки отчёта.