Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

М.И. Гуисов

Применение нейронных сетей для оценки позиции в игре Го

Магистерская диссертация

Санкт-Петербург

2006

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данная работа содержит информацию о ходе исследования проблемы подсчета очков в игре Го, описание достигнутых результатов и созданного программного комплекса. Документ разбит на несколько разделов: описание правил игры, простейших элементов тактики, постановка исследовательской и алгоритмической задач, подробное описание примененных алгоритмов и технологий, и результаты исследования.

Кратко об истории Го

Го имеет несколько современных вариаций: *Вей-ци* в Китае, *Бадук* в Корее и *Иго* (или просто *Го*) в Японии. Правила этих разновидностей игры схожи, однако подсчёт очков в Китае и Японии несколько отличается. Согласно легенде, игра Го была изобретена в древнем Китае в 22-23 веках до нашей эры. За время своего существования игра не претерпела значительных изменений. Со временем менялись стили, предпочтения. Самые сильные изменения касались шаблона для начала игры и правил подсчета очков. До сих пор существует два основных варианта: *Инговский* (или китайский), в котором поле полностью заставляется камнями, и считаются выставленные камни, и японский, в котором очки складываются из окруженной территории и съеденных камней противника. Подробнее об истории возникновения игры и появления ее в России можно почитать на российском *Интернет*-ресурсе "*Го Библиотека*" [29].

Правила игры и основные тактические следствия

Правила игры просты. Для игры необходима доска, расчерченная сеткой из (условно) горизонтальных и вертикальных линий (традиционные размеры – 9x9, 13x13 и 19x19 линий), и два набора игровых фишек (двух цветов – черного и белого), традиционно называемых *камни*. Подойдет также электронное исполнение этих атрибутов игры (рис. 1). Два игрока делают ходы по очереди. *Ходом* считается постановка камня на одну из свободных точек – пересечение линий игрового поля (рис. 2). Внимание, камни ставятся не в квадраты сетки, а на пересечения её линий (в вершины "графа", который можно себе представить, глядя на сетку). В равной партии (без гандикапа) начинают черные (рис. 3), а белые получают компенсацию (яп. *коми*) в размере 6.5 очков. Половина очка гарантирует отсутствие ничьих.

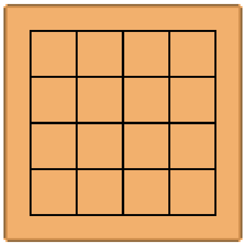


Рис. 1. Доска 5x5.

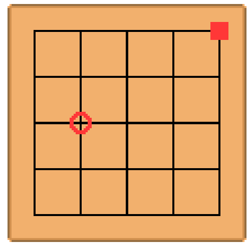


Рис. 2. Ходы делаются только в свободные пересечения линий игрового поля.

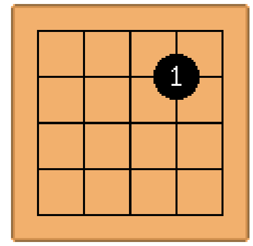


Рис. 3. Начинают черные.

Камни одного цвета, стоящие в соседних точках, условно объединяются в блоки. *Соседними точками* называются вершины воображаемого графа, соединенные ребром с данной точкой. Свободные соседние вершины блока называются его *степенями свободы* (яп. *даме*). У одиночного камня, стоящего не на краю доски, имеется четыре степени свободы, на краю – три, в самом углу – две. Если все соседние точки блока оказываются заняты камнями противника, блок считается *съеденным* и убирается с доски. Одной из локальных (тактических) целей игры является поедание камней противника (рис. 4-6). Блоки камней одного цвета, расположенные рядом друг с другом, условно объединяются в *группы*. Обычно игроки оценивают позицию в масштабе групп, а не отдельных блоков.

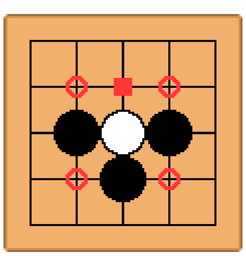


Рис. 4. Белый камень под ударом (яп. *атари*), у него всего одна степень свободы.

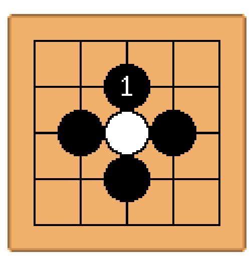


Рис. 5. Черные полностью окружают одинокий белый камень.

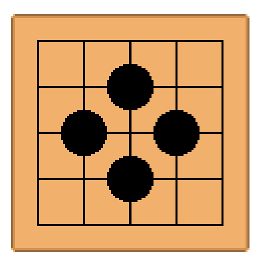


Рис. 6. Согласно правилам, белый камень убирается с доски и считается добычей черных.

Следующее правило запрещает *самоубийственные* ходы, в результате которых один или несколько ваших блоков лишаются всех степеней свободы (рис. 7). Однако если этим ходом вы съедаете камни противника, он не считается самоубийственным, поскольку у получившегося у вас блока появляются новые степени свободы на месте съеденных камней противника (рис. 8). Важнейшее тактическое следствие этого правила – возможность создания крепостей – блоков с двумя не соседними степенями свободы (рис. 9). Противник не сможет съесть такой блок при всем желании, поскольку ход в каждую из этих точек для него самоубийственный (для поедания вашего блока нужно занять обе точки).

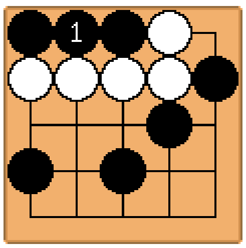


Рис. 7. Самоубийственные ходы запрещены: после хода в [1] у черного блока осталось ноль степеней свободы.

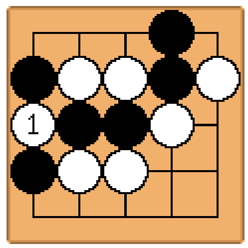


Рис.8. Белый ход в [1] разрешен правилами, поскольку два черных камня лишатся последней степени свободы.

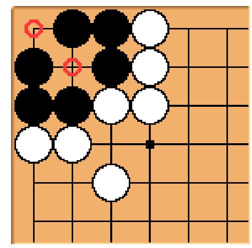


Рис. 9. Крепость, ходы в помеченные точки (*глаза*) запрещены как самоубийство.

Последнее правило запрещает повторение позиции. С этим правилом связана частая тактическая ситуация под названием *ко*. Стандартный паттерн (черный камень под ударом и одна степень свободы у белого камня после съедания) и начало *ко-борьбы* изображены на рис. 10-12.

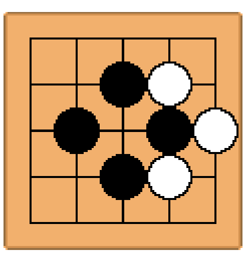


Рис. 10. Ко. Черный камень может стоить одно очко, а может означать и большую потерю.

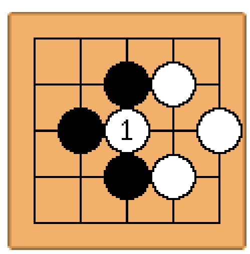


Рис. 11. Белые съедают черный камень – начинают (продолжают) ко-борьбу.

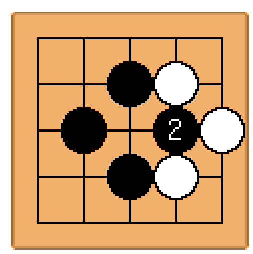


Рис. 12. В ответ черные не могут сразу съесть белый камень – это было бы повторением позиции.

Поскольку съесть белый камень черные не могут, вместо этого они делают ход в другой части поля, надеясь, что белые ответят (*ко-угроза*). И в случае ответа (позиция изменилась), уже съедают белый камень. *Ко-борьба* в значительной степени усложняет тактику игры, а с точки зрения программирования, глубину расчета тактических вариантов и сложность оценки игровой позиции. В том числе и статус группы часто зависит от исхода той или иной *ко-борьбы*.

Цель игры – набрать больше очков, чем противник. Очки приносят съеденные камни противника и огороженная вашими камнями территория. По ходу игры стоимость хода (потенциальные очки) уменьшается до нуля. Формально игра заканчивается, когда заканчиваются ходы неотрицательной ценности, и оба игрока пасуют (пропускают ход). Фактически обычно несколько раньше – когда игрок не видит больше ходов, приносящих очки. Добивать в своей территории обреченные камни противника не обязательно. Камни обречены, если невозможно (учитывая ответы противника) на их основе построить крепость, или соединить с другой крепостью.

Сложность игры

При общей простоте правил игра потрясающе глубокая и серьезная. В то время как изучение правил занимает 10-15 минут, мастерство растет с годами опыта. Причины этого – высокая комбинаторная сложность игры (которая также не позволяет "решить" игру простым перебором вариантов), обилие тактических принципов и интуитивных оценок для каждой ситуации. Условно игровой процесс делится на три стадии: *фусеки* (начало игры, время разбрасывать камни, создание стратегического плана игры), *тюбан* (время собирать камни, сражение между группами по всей доске, крупные обмены, вторжения и прочее) и *йосе* (доводка наметившихся территорий, точный подсчет стоимости каждого хода, в самом конце игры – чистая комбинаторика). В первой стадии прежде всего требуется стратегическое мышление, огромную роль играет опыт. Но без элементарного знания дебютов в углах доски (яп. *джосеки*) никакой задумки не получится. В *тюбане* важна интуиция, напор, умение считать варианты и находить многоцелевые ходы. В *йосе* важно уметь строить наилучшую последовательность отыгрыша, быстро и точно считать очки, которые принесет каждый ход, и важность этого хода. Огромную роль играет *инициатива*, в борьбе за нее многие игроки нередко оставляют одну не до конца стабилизировавшуюся ситуацию и переключаются на другую. Все вместе делает теорию игры необъятной.

Каждой стадии игры присущи характерные черты мышления игрока. В начале игры локальные ситуации чаще всего решаются при помощи стандартных комбинаций (например, дебюты в углу), хотя есть и очень сложные, с трудом поддающиеся изучению деревья вариантов. Игроки нечасто хотят усложнить игру в этот момент, и стараются переиграть противника стратегически. В *тюбане*, несмотря на множество стандартных технических комбинаций, часто приходится просчитывать варианты "по ходам", иногда – принимать стратегические решения. И, наконец, в *йосе* игра снова ведется стандартными комбинациями, а ключевая роль отводится оценке прибыли от каждой из них. В целом, чтобы получить звание мастера, игрок должен копить опыт (технические элементы, паттерны), обладать незаурядными комбинаторными способностями, и развивать игровую интуицию. Интуиция используется в игре сплошь и рядом, при оценке чистой и потенциальной территории, степени влияния одних групп на другие, при принятии стратегических решений и в сложных локальных ситуациях, когда просчитать все варианты не удается (или не хватает времени).

Концепция очков

Конечная цель игры – перевес в очках. Конкретные числа легко сравнивать, и поэтому при анализе позиции многие ее качества оцениваются в очках. Начиная от очевидного – чистой территории, в которую противник не может вторгнуться – к более сложным концепциям – например, оценка возможных потерь в данной локации от вторжения противника, или оценка текущей стоимости инициативы. В отличие от шахмат, где каждой фигуре присваивается определенный ранг и в оценке учитывается лишь свобода передвижения фигур, в Го камни одинаковы, а группа камней может оказаться независимой, а может и влиять на все поле. Важную роль играет статус камней – живые (несмотря на противодействие противника можно построить крепость, или соединиться с другой группой), мертвые (как ни пытайся, но если противник отвечает на каждый ход правильно – не оживешь) и критические (один ход решает, жить группе, или умереть). В середине игры ситуация может меняться. В какой-то момент противник может пойти на обмен, и мертвой группе удастся ожить. Бывает и наоборот, если игрок невнимательно (неверно) оценивает статус своей группы в тот момент, когда ее в действительности пора спасать.

В конце игры ситуация в большинстве игр (бывают исключения, особенно в партиях профессионалов) в значительной степени стабилизируется, появляется возможность более точно оценить потенциальные прибыли и потери во всех оставшихся ключевых точках. По окончании игры (оба игрока пасовали) практически любой наблюдатель в состоянии подсчитать, у кого очков больше. Практически любой, за исключением существующих компьютерных программ. Автору данной работы неизвестны играющие в Го программы, способные корректно подсчитать очки в 100% партий до того, как с доски убраны обреченные камни. Очевидно, если объяснить программе, какие камни обречены, она с легкостью подсчитает окруженную оставшимися камнями территорию. Основная проблема состоит именно в том, что пока не существует точного действующего алгоритма оценки статуса камней.

Постановка задачи

Подсчет очков при наличии точной оценки статусов групп – решенная задача. Можно воспользоваться стандартным алгоритмом класса "поиска пути" – распространяясь во всех направлениях от данной незанятой точки и считая камни границей области. Или можно подсчитать территорию при помощи морфологических преобразований изображения доски. Многие го-программы используют морфологические функции Зобриста – расширение и сжатие – для быстрой приблизительной оценки точной (хорошо очерченной) и потенциальной территории в процессе игры. К концу игры точность предсказания возрастает до 100%, но лишь при условии, если обреченные камни убраны с доски. Основная трудность – точно определить, какие камни мертвы.

Актуальность

В последнее десятилетие исследования в области программирования Го значительно активизировались, в основном благодаря крупному призу, назначенному за программу, которая сможет победить профессионального игрока. Несмотря на это, существующие программы с трудом могут соревноваться со средними любителями. Оценка позиции в игре Го – сложнейшая проблема в области создания / исследования искусственного интеллекта. Самая важная ее часть – оценка статуса группы камней – на данный момент алгоритмически нерешенная задача. Некоторые программы используют статический анализ различных свойств группы, и классифицирует ее по уровню живучести [13, 18, 20, 21, 22]. Другие используют брутальные методы активного локального перебора вариантов с целью определения вероятного статуса группы. В последнее время популярными становятся методы класса *Монте-Карло* – быстрый *поиск в глубину*, со случайным предсказателем ходов [4, 6]. В применении к оценке статуса группы это локальный поиск до момента, когда статус группы точно определим как *живая* или *мертвая*. Делается (например) 10.000 случайных поисков, и вычисляется живучесть группы как процент веток, закончившихся выживанием. Еще один метод оценки статусов групп – применение нейронных сетей. Позиция преобразуется определенным образом и не основе промежуточного представления нейронная сеть делает предсказание о статусе каждой группы. Исследованию этого вопроса посвящена данная работа.

Формулировка

Алгоритмическая задача. Исходные данные – игровая позиция, априори *финальная*. То есть, ситуация на доске стабильная, критических групп нет, доигрывать нигде не нужно (на самом деле даже профессиональные игры нередко не доигрываются до конца, поскольку счет точно известен за 10-15 ходов). Требуется найти все обреченные камни и удалить их с доски. Иными словами, требуется *построить алгоритм классификации блоков на живые и обреченные*. Конечность позиции несколько облегчает (и сужает) задачу определения статуса групп, поскольку, например, игроки не оставляют камни под ударом (с одной степенью свободы), если их можно спасти. С точки зрения предварительной обработки позиции, такие камни просто исключаются из оценки с предварительным статусом *мертвые*.

Исследовательская задача. Исходные данные – программный каркас (топологически анализирует позицию, выделяет численные характеристики отдельных блоков камней, групп, областей и т.д.), тренировочные и тестовые записи партий. Множество законченных партий с разметкой обреченных групп выбрано с сайта игрового портала http://kiseido.com/. Требования к позициям: *законченность* (игроки довели партию до конца, ни один не сдался), *точность экспертной разметки*, отсутствие альтернативных вариантов (игровое дерево – вырожденное). Требуется *построить наиболее эффективную нейронную сеть для оценки статуса групп*.

Пояснения

Два слова надо добавить о статусе группы. Группа считается *живой*, если несмотря на попытки противника ее захватить, игрок может построить крепость или соединиться с другой крепостью. Кроме того, есть еще одна возможность *оживить* камни – привести их в состояние *секи* с одной из групп противника. *Секи* это такая игровая ситуация, в которой две группы разных цветов окружены безусловно-живыми камнями противника и разделяют одну или несколько степеней свободы. Ход в общую точку приводит к тому, что у группы (сделавшей ход) остается только одна степень свободы, и соседствующая (в *секи*) группа противника ответным ходом ее съедает.

Теоретически, пространство характеристик групп четко разделимо по признаку живучести – используя полную информацию о группе, можно точно отнести ее к *живым, мертвым* или *критическим* (а в конце игры критических групп нет, только живые и мертвые). Иными словами, достаточно емкий перцептрон (многослойная нейронная сеть) должен быть в состоянии аппроксимировать функцию оценки статуса группы.

Полная информация о позиции – это ее представление (массив из 361 элемента, для доски 19x19), плюс определение отношения соседства между любыми двумя точками. В идеале, конечно, хотелось бы передавать нейронной сети позицию, и получать на выходе позицию с помеченными статусами. На практике, однако, есть проблема: подобное представление информации о позиции не является релевантным. Для оценки статуса используются производные величины, как например, количество *степеней свободы* у группы, количество *глаз*. Опытный игрок с одного взгляда определит статус группы с высокой точностью. При этом он руководствуется визуальной *формой* группы. Передать нейронной сети понятие формы очень нелегко, однако как показывают различные исследования в области статической оценки форм [13, 20], некоторые численные характеристики очень неплохо кореллируют с *живучестью формы*.

И, наконец, нельзя в общем случае оценивать статус группы без учета ее окружения. Нередко статус группы зависит от ближайших групп противника. Значит, нужно учитывать их характеристики. И здесь возникает другая проблема: количество характеристик не является постоянным среди групп (у одной группы может быть один глаз, но большой, а у другой – два поменьше). Однако нейронная сеть требует стандартизации входных данных – нельзя одним аргументом передавать различные характеристики от случая к случаю.

1. СХЕМА РЕШЕНИЯ

В данной главе описан примененный в работе математический аппарат, рассмотрены различные оптимизации нейронной сети, отдельное внимание уделено реализации нейронной сети и подготовке входных данных для нее. Структура программного комплекса *SEOG* (сокращение от Score End Of Game) приведена на рис. 13.

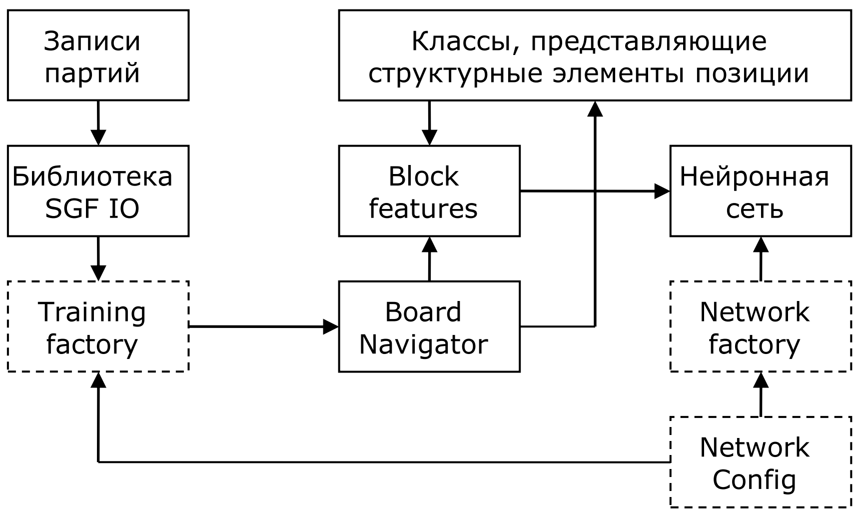


Рис. 13. Структура программного комплекса *SEOG*

Приведем кратко алгоритм работы программного комплекса.

1. Создаются управляющие объекты `network config`, `network factory` и `training factory`.

2. Создается набор нейронных сетей (*каскад*) – классификаторов, на практике от одной до пяти. Каждая следующая сеть уточняет результат предыдущей.

3. С жесткого диска последовательно считываются все записи партий, финальные позиции в которых используются для тренировки и тестирования сетей.

4. Выполняется обучение (совместно с тестированием) каскада сетей по эпохам (прежде чем меняются веса, осуществляется полный обход всех тренировочных вариантов):

- каждая финальная позиция из кэша структурно разбирается – выделяются основные элементы позиции (блоки, цепочки, регионы) и находятся их численные характеристики;

- каждый блок оценивается простым статическим алгоритмом, затем для всех блоков со статусом «под вопросом» строится вектор характеристик – получается обучающий (или тестовый пример) для нейронной сети;
- вычисляется ошибка сети на каждом примере, на тренировочных примерах происходит обратное распространение градиента ошибки по весам сети, затем единовременно выполняется изменение весов. Подробнее об алгоритмах обучения нейронной сети см. в параграфе 1.4.

Необходимо отметить, что классификация блоков может осуществляться при помощи нейронных сетей самых различных конфигураций. Обычно для решения задачи классификации создают нейронную сеть с количеством нейронов в выходном слое, равным количеству классов. Однако практика показала, что в случае всего двух классов достаточно использовать один выход, и рассматривать значение 0 как один класс (статус блока «мертв»), и 1 – как второй (статус блока «жив»). Кроме этого, как предлагается в работе [1], отдельно были исследованы возможности уточнения предсказания одной нейронной сети другой сетью. Такая конфигурация в данной работе называется *каскадом* нейронных сетей.

1. 1. Разбор записей партий

Для обучения нейронной сети необходимо тренировочное множество объектов – набор записей партий, отвечающих условиям задачи: иными словами, запись должна содержать финальную позицию, и предоставлять дополнительную информацию о статусах групп в ней. Создавать и маркировать записи вручную – весьма трудоемкий процесс, поэтому был выбран более простой путь. С сайта популярного игрового сервера kiseido были взяты порядка 15.000 партий игроков уровня 4-7 дана (партии мастеров среди любителей) в распространенном формате SGF (Smart game format [30]). Особенность данного сервера в том, что в конце каждой партии игроки отмечают мертвые группы, после чего система автоматически подсчитывает очки и сохраняет запись в базе данных вместе с разметкой. Чтобы отсеять партии, закончившиеся сдачей (не доведенные до этапа подсчета очков), и отфильтровать лишнюю мета-информацию о партиях (комментарии, варианты) была написана специальная утилита *selector*. В результате фильтрации, было получено множество из 10247 записей партий, содержащих только необходимую информацию (список ходов, разметку, результат). Это позволило использовать в программном комплексе SEOG упрощенную функцию разбора записи партии (библиотека SGF IO).

1. 2. Единичные управляющие объекты

Конфигурация каскада нейронных сетей и управление тренировочным процессом на верхнем уровне управляется тремя объектами (типа singleton): `training factory`, `net config` и `network factory`.

1. 2. 1. Объект `Training Factory`

Данный объект отвечает за кэширование характеристических векторов блоков для тренировки сети, координирует анализ свойств позиции, обучение набора нейронных сетей и их тестирование. В процессе разработки программного комплекса было испробовано несколько вариантов подачи информации для нейронной сети.

1. Последовательная загрузка и анализ позиций из множества файлов – самый медленный способ, из-за постоянного общения с жестким диском.

2. Предварительный загрузка и разбор всех обучающих примеров и кэширование свойств позиции – потребовало достаточно больших затрат памяти на хранение промежуточной информации, однако по скорости значительно превосходит первый вариант.

3. Кэширование финальных позиций, и структурный анализ непосредственно перед передачей нейронной сети быстрее первого метода и требует во много раз меньше памяти, чем второй.

4. И наконец, как выяснилось при работе с каскадами классификаторов, очень эффективно с точки зрения скорости обучения тренировать сперва один классификатор полностью, затем следующий, также полностью, и т.д. Одновременная тренировка всего каскада приводила к тому, что вторая нейронная сеть самую интересную (теоретически) информацию – предсказания первой сети – получала неточную и уделяла ей недостаточно внимания. Как следствие, стало возможным кэшировать прямо характеристические вектора блоков для последней сети, предварительно применив готовые классификаторы, и рассчитав динамические характеристики блоков.

Для удобного совмещения информации о динамике обучения сети, объект *training factory* рассматривает каждый десятый характеристический вектор, как тестовый. За один проход (эпоху обучения) мы получаем сразу две RMS ошибки (нормализованный квадрат отклонения значений классификатора от требуемых): для тренировочного и тестового множеств.

1. 2. 2. Объект `Network Config`

Данный объект сохраняет дополнительные параметры обучения сети, используемые объектом `training factory` при организации процесса.

1. Флаг `drop_solved` отвечает за откладывание позиции на данной эпохе обучения, если на предыдущей эпохе ошибка нейронной сети на этой позиции была очень мала. Эта оптимизация позволяет значительно ускорять обучение сети самым сложным примерам. Однако надо помнить, что пропуск *решенных* (верно классифицируемых) позиций влечет накопление ошибки на них.

2. Флаг `refresh_dropped` означает возврат отложенных позиций не следующем шаге (эпохе) обучения. Это позволяет держать ошибку классификации на решенных позициях на приемлемом уровне.

3. Еще один флаг, `start_on_half_training_set`, предназначен для балансировки количества отбрасываемых позиций. Достаточно тренированная сеть будет верно классифицировать больше 90% блоков, то есть около 80% партий могут быть отброшены. Данный флаг позволяет исключить ситуацию, когда на одном шаге сеть тренируется на всем множестве, а на следующем – только на малой части.

Кроме того, объект `network config` хранит список свойств блоков, подаваемых на вход нейронной сети. Для гибкой настройки конфигурации сети были созданы три основных набора характеристик.

1. 2. 3. Объект `Network Factory`

Данный объект предоставляет общий интерфейс к классифицирующему каскаду из нескольких сетей. Он сохраняет описания сетей в коллекции `networks`, позволяет запускать и останавливать тренировочный процесс, сохраняет результаты обучения, выводит ошибки на тренировочном и тестовом множествах в выходной файл.

1. 3. Класс `Board Navigator`

Навигатор финальной позиции – объект, инициирующий топологический разбор позиции и выделение характеристик структурных элементов (блоков / цепочек), и последовательно передающий вектора характеристик на вход нейронной сети.

1. 3. 1. Структурные элементы позиции

Нейронная сеть анализирует статусы блоков камней, основываясь на некотором наборе характеристик каждого блока (конкретный набор определяет топологию сети). Для нахождения большого количества характеристик программа должна иметь представление о структуре позиции. Для каждой позиции объект класса `board navigator` составляет списки различных структурных элементов – производит топологический разбор позиции. Кроме блоков (*blocks*), структурными элементами являются:

- цепочки блоков (*chains*) – блоки, соединенные хотя бы одной общей степенью свободы;
- пустые регионы (*empty spaces*) – топологически, эквивалент блока, но с присвоенным цветом `eEmpty`;
- окруженные цветом регионы (*color enclosed regions, CER*) – области, состоящие из камней одного цвета и пустых точек, ограниченные камнями противоположного цвета.

1. 3. 2. Характеристический вектор блока

Для наиболее точной классификации статуса блока необходимо предоставить максимум информации о его форме, окружающих дружественных и противостоящих блоках. Далеко не все представленные характеристики оказывают заметное влияние на точность предсказания нейронной сетью, однако вместе они позволяют добиться интересных результатов. Исследования в области статической оценки *формы глаз* группы показывают, что живучесть группы сильно коррелирует с размером и периметром глаза, с количеством *точек деления глаза* (*eye split points*, рис. 14). Иными словами, размер, периметр и количество точек деления региона – достаточно информативный набор характеристик, соответствующий в какой-то мере понятию *форма региона*.

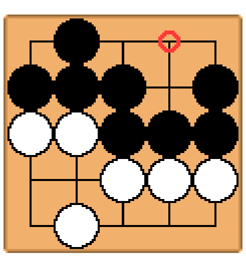


Рис. 14. Точка деления глаза.

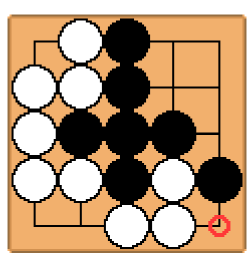


Рис. 15. Защищенная степень свободы – ход сюда ставит блок игрока в атари.

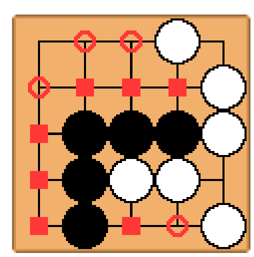


Рис. 16. Степени свободы черного блока.

Для каждого блока мы выделяем следующие статические характеристики.

1. *Размер* (size) – количество камней, его составляющих.
2. *Периметр* (perimeter) – количество виртуальных степеней свободы, в случае если блок расположить в одиночестве на доске бесконечного размера.
3. Количество *степеней свободы трех рангов* (liberties_1, _2, _3):
   - первого – это степени свободы, собственно, блока (квадратики на рис. 16);
   - второго – свободные соседние точки у степеней свободы первого ранга, не являющиеся сами степенью свободы первого ранга (кружки на рис. 16);
   - третьего – свободные соседние точки у степеней свободы второго ранга, не являющиеся сами степенью свободы первого или второго ранга (верхняя левая точка на рис. 16).
4. Количество *защищенных степеней свободы* (protected_liberties) – таких точек, ход противника в которые ставит его камень (или блок) в *атари* – оставляет одну степень свободы (рис. 15).
5. Количество *соединений с цепочкой* (chain_connections) – каждый блок является частью цепочки блоков, хотя бы из одного элемента.
6. Количество прилежащих блоков противника (opp_blocks_num).
7. *Локальное превосходство цвета* (local_majority) – количество камней цвета блока – количество камней противника на манхэттэновском расстоянии не превышающем двух (определяется как $dist(p_1,p_2) = (x_{p1} - x_{p2}) + (y_{p1} - y_{p2})$).
8. Расстояния до ближайшей границы доски по горизонтали (cm_x) и вертикали (cm_y).
9. Площадь прямоугольника, описанного вокруг блока (bb_square).

Кроме этого, к блоку прилегают один и более регионов, окруженных цветом блока. Каждый из этих регионов может быть *глазом* (пустой регион, не содержащий камней противоположного цвета), *частично-доступным регионом* (у блока есть одна или несколько степеней свободы в регионе) или *недоступным регионом* (все точки региона, являющиеся соседними по отношению к блоку, заняты камнями противника). К описанию блока добавляются следующие характеристики прилежащих к блоку глаз: количество (eye_num), суммарный размер (eye_sum_size), суммарный периметр (eye_sum_perimeter), суммарное количество точек деления (eye_sum_split_points).

Частично-доступные регионы (*partially accessible CER*) также добавляют блоку характеристики: количество свободных точек, доступных блоку (`pacer_sum_acc_size`), их суммарный периметр (`pacer_sum_acc_perimeter`), количество (`pacer_sum_int_size`) и периметр (`pacer_sum_int_perimeter`) остальных (недоступных) точек региона, количество их точек деления (`pacer_sum_int_split_points`).

Позднее была добавлена еще одна характеристика – *средний уровень функции давления* (или влияния) в регионе (`pacer_average_power`). В программном комплексе реализованы две функции, вычисляющие влияние камней.

1. Морфологическая функция Зобриста, построенная на *расширениях* (*dilations*) и *сжатиях* (*erosions*), как описана в [3].

2. Функция, основанная на распространении влияния от каждого камня по заданному шаблону, учитывающая предсказанный статус камня (актуально с точки зрения каскадной оценки статусов блоков и с точки зрения предварительного статического анализа позиции).

Далее, блок сам находится в регионе, окруженном цветом противника. Добавляем размеры региона (`opp_cer_size`) и его периметр (`opp_cer_perimeter`).

У цепочки, содержащий блок, выделяем следующие характеристики.

1. Количество составляющих цепочку блоков (`chain_blocks_num`).
2. Количество камней (`chain_size`) и периметр (`chain_perimeter`).
3. Количество степеней свободы у группы (`chain_liberties`) – не равно сумме количеств степеней свободы отдельных блоков, так как по определению группы, некоторые точки являются общими степенями свободы для двух и более блоков).
4. Количество прилежащих регионов, окруженных цветом группы (`chain_adj_cer_num`).
5. Количество глаз у группы, их суммарный размер, периметр, и точки деления.
6. Количество внешних степеней свободы (`chain_ext_opp_liberties`), общих с блоками противника.
7. Количество связующих блоки степеней свободы (`chain_split_liberties`).
8. Оценка прилежащей территории (`chain_estimated_adj_terr`) – еще одна морфологическая функция Зобриста. Отличие от предыдущей в количестве применяемых *расширений* и *сжатий* [3].
9. Количество частично-доступных регионов, размера не больше пяти (`chain_pacer5_num`), и среднее давление в них (`chain_pacer5_ave_power`). Аналогично для регионов, размера не больше девяти.

Однако одной информации о размерах и форме блока недостаточно. Важно также указать какие-либо характеристики окружающих блоков. Часто при анализе статуса блока, окончательный вывод делается исходя из слабости прилежащих блоков противника. Однако для многих блоков это будет нулевая информация (нет соседей), поэтому данные характеристики обладают ограниченной эффективностью в плане обучения сетей. Итак, в полный набор характеристик (для сетей с максимальным количеством входной информации) включены свойства следующих окружающих объектов.

1. Двух слабейших блоков противника, из прилежащих к данному блоку: размеры, периметр, количество степеней свободы, количество общих степеней свободы с блоком, количество степеней свободы, ход в которые ставит блок в *атари*, количество соединений блока со своей цепочкой, количество прилежащих глаз, их суммарный размер, периметр и количество точек раздела.

2. Цепочки противника, содержащей слабейший из прилежащих блоков: размер, периметр, количество глаз, их суммарный размер и периметр, количество точек раздела, количество связующих блоки степеней свободы.

3. Слабейшего блока противника, имеющего общие степени свободы с данной группой и сильнейшего соседнего блока своего цвета: список характеристик такой же как у слабейших соседей.

Последние из статических характеристик – количество степеней свободы блока (`terr_direct_liberties`), всех дружественных блоков (`terr_all_friend_liberties`) и всех блоков противника (`terr_all_opp_liberties`) в прилежащей к данному блоку территории (территории, определенной функцией Зобриста).

Кроме статических характеристик при обучении каскада нейронных сетей весьма эффективны оказываются динамические свойства блока и его окружения – предсказанный предыдущей сетью статус передается на вход следующей сети. Полный набор дополнительных характеристик для уточняющих сетей (всех после первой) состоит из предсказанных статусов рассматриваемого блока, и всех блоков, чьи характеристики входят в полное статическое описание блока.

1. 4. Нейронная сеть

Нейронная сеть – это математический инструмент класса *black box* (черный ящик), предназначенный для аппроксимации неизвестной функции, разделяющей пространство характеристик некоторого множества объектов на классы. Объекты описываются численными характеристиками, которые затем масштабируются (например по максимальному и минимальному значениям). Описание объекта – вектор характеристик, каждой из которых соответствует ось в пространстве характеристик. Каждый выход нейронной сети – это нелинейная функция от вектора характеристик, выделяющая часть пространства характеристик в отдельный класс. Важно отметить, что нейронная сеть, несмотря на заявленную цель – *аппроксимацию* классифицируюшего функционала, на самом деле лишь *интерполирует* ее по данному набору векторов тренировочного множества. Поэтому эффективность классификации *тренировочного множества* не является показателем качества, и оценка результата обучения нейронной сети производится на *тестовом множестве*.

Алгоритмически, нейронная сеть – это совокупность, состоящая из множества *нейронов,* связанных между собой определенным образом (схема связей называется *топологией сети*), *активационной функции*, и метода обучения сети. Каждый нейрон – это обработчик входных сигналов. Взвешенная сумма входных сигналов подается на вход активационной функции. Результат является выходом нейрона. В качестве активационной функции в данной работе использовалась только *сигмоида* (в общем случае выбор достаточно большой). В качестве основной топологии сети был выбран многослойный *перцептрон* (слои нейронов, выстроенные в ряд, каждый слой в качестве входа использует выход предыдущего слоя, а первый слой – входной вектор характеристик).

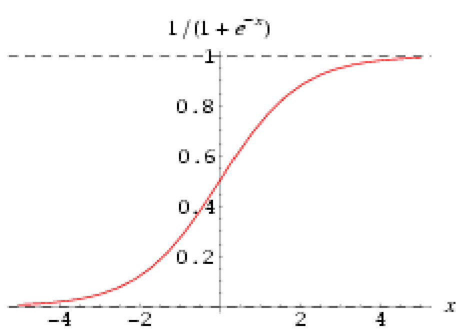


Рис. 17. Активационная функция – *сигмоида*.

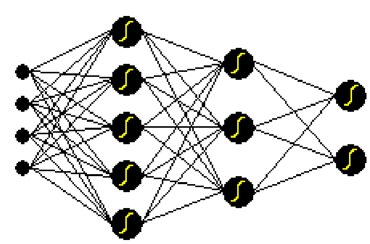


Рис. 18. Многослойный *перцептрон*.

1. 4. 1. Обучение нейронной сети

Говоря об обучении нейронной сети подразумевают подгонку весов входов в каждом нейроне. Для этого применяются мат. методы оптимизации, например, метод градиентного спуска (в простейшем варианте). Тренировочное множество обладает важным качеством – про каждый объект в нем известно, к какому классу он принадлежит. Поэтому, применив нейронную сеть на объект из тренировочного множества, можно найти величину ошибки классификации объекта. Метод *обратного распространения градиента ошибки* (*error gradient backpropagation*) позволяет вычислить составляющую градиента ошибки, соответствующую каждому весу. Как только известен градиент, можно применять методы поиска минимума ошибки. Стандартная сеть обратного распространения (*backpropagation network*) использует метод градиентного спуска. Скорость обучения данного класса сетей оставляет желать лучшего.

Основная проблема в методе градиентного спуска – контр-интуитивное изменение веса. Интуиция говорит, если градиент ошибки большой по модулю, надо искать локальный минимум маленькими шажками (иначе очередным шагом просто пролетим мимо минимума), а при градиенте в районе нуля можно и ускорить спуск. Однако метод действует ровно наоборот – величина изменения веса прямо пропорциональна модулю градиента ошибки.

Кроме того, все методы, основанные на градиентном спуске имеют тенденцию сходиться к локальному минимуму, что по сравнению со стохастическими методами, для многих задач классификации является разумной платой за скорость обучения.

1. 4. 2. Алгоритм *RPROP*

Для ускорения обучения нейронных сетей применяются различные адаптивные алгоритмы. В данной работе в качестве основы выбран алгоритм *RPROP* (*resilient backpropagation* – ленивое обратное распространение), один из лучших по скорости обучения в классе *локально-адаптивных* алгоритмов (этот алгоритм учитывает изменения локального градиента для выбора размера и направления шага). Метод обучения на его основе вычисляет изменение весов сети *по эпохам*: сначала нейронная сеть применяется на каждый объект тренировочного множества, вычисляется и распространяется обратно градиент ошибки, затем происходит однократное обновление весов (здесь видимо и кроется причина *ленивости* названия). Суть алгоритма заключается в следующем.

1. Величина изменения веса (далее, шаг) не зависит от значения модуля градиента, а вычисляется по простому правилу: если градиент с предыдущей эпохи обучения сохранил знак, шаг увеличивается; если градиент поменял знак – шаг уменьшается. Заданы максимальный и минимальный размеры шага.

2. Направление шага выбирается противоположным текущему градиенту ошибки.

3. В случае изменения направления градиента, производится откат предыдущего изменения веса (то есть движение по старому градиенту).

Иными словами, алгоритм *RPROP* учитывает при изменении веса только знак градиента ошибки по весу, адаптивно вычисляя размер шага. Для наглядности приведем псевдокод алгоритма адаптивного изменения веса.

```
для всех весов
{
   если ( градиент сохранил знак )
   {
      // увеличить шаг
      dWi(t) = min( max_delta, dWi(t-1) * n+ );

      // изменить вес на шаг, против текущего градиента
      Wi(t) = Wi(t-1) – sign( (dE/dWi)(t) ) * dWi(t);
   }
   если же ( градиент изменил знак )
   {
      // отменить изменение веса с предыдущей эпохи обучения
      Wi(t) = Wi(t-1) + sign( (dE/dWi)(t-1) ) * dWi(t-1);

      // уменьшить шаг
      dWi(t) = max( min_delta, dWi(t-1) * n- );

      // обнулить переменную, хранящую градиент
      (dE/dWi)(t-1) = 0;
   }
   иначе
   {
      // изменить вес на шаг, против текущего градиента
      Wi(t) = Wi(t-1) – sign( (dE/dWi)(t) ) * dWi(t);
   }
}
```

Здесь и далее $n_+ = 1.2$, $n_- = 0.5$, min_delta = 1E-6 и max_delta = 50.0 – константы со стандартными значениями [25].

1. 4. 3. Алгоритм *iRPROP+*

Сразу после появления, алгоритм *RPROP* завоевал популярность благодаря простоте и высокой скорости и эффективности обучения. Однако, нужно отметить, что данный алгоритм использует только информацию о локальном градиенте ошибки (точнее, только градиент ошибки по данному весу). В целом ряде работ были попытки улучшить скорость и эффективность обучения нейронной сети за счет учета глобального изменения ошибки. Одна из простейших модификаций базового алгоритма оказалась на редкость эффективной. Суть отличия – если изменился знак градиента, производить откат только в случае, если общая ошибка классификации увеличилась с предыдущего шага. Проиллюстрируем эту идею с помощью псевдокода.

```
для всех весов
{
   если ( градиент сохранил знак )
   {
      // увеличить шаг
      dWi(t) = min( max_delta, dWi(t-1) * n+ );

      // изменить вес на шаг, против текущего градиента
      Wi(t) = Wi(t-1) – sign( (dE/dWi)(t) ) * dWi(t);
   }
   если же ( градиент изменил знак )
   {
      // откат только если ошибка классификации увеличилась
      если ( E(t) > E(t-1) )
      {
         // отменить изменение веса с предыдущей эпохи обучения
         Wi(t) = Wi(t-1) + sign( (dE/dWi)(t-1) ) * dWi(t-1);
      }

      // уменьшить шаг
      dWi(t) = max( min_delta, dWi(t-1) * n- );

      // обнулить переменную, хранящую градиент
      (dE/dWi)(t-1) = 0;
   }
   иначе
   {
      // изменить вес на шаг, против текущего градиента
      Wi(t) = Wi(t-1) – sign( (dE/dWi)(t) ) * dWi(t);
   }
}
```

Данный алгоритм получил название *iRPROP+* (*improved RPROP with weight backtracking*). На некоторых тестовых задачах он обучает нейронную сеть быстрее базового алгоритма [26]. Кроме того, на отдельных задачах он сходится к лучшим значениям.

1. 4. 4. Алгоритм *SARPROP*

Несмотря на высокую скорость обучения, алгоритм *RPROP* страдает от общей для всех градиентных методов проблемы – градиент ошибки часто сходится к локальному минимуму. В работе [27] был предложен способ частичного решения этой проблемы, основанный на добавлении шума к размеру шага в случае когда он (размер шага) не превосходит некоторой величины. Как известно, основной плюс методов случайного поиска – сходимость сети к глобальному минимуму. Однако время обучения такой сети несравнимо больше, чем у простейших алгоритмов, основанных на градиенте ошибки.

SARPROP (*Simulated Annealing enchancement to RPROP*) или метод «*имитации отжига*» это попытка совместить случайный поиск с затухающими колебаниями и адаптивный выбор направления по градиенту. Если градиент ошибки с прошлой эпохи не поменял знак, работает базовый метод *RPROP* – шаг постепенно увеличивается. Шум добавляется только, когда градиент изменяет знак, в случае, если размер шага меньше порогового значения (формула 1).

$$P = k_2 * E^2(t), k_2 = 0.1 \quad (1)$$

Добавка dW_{add} определяется как случайная величина от нуля до единицы, с коэффициентом, экспоненциально убывающим с ростом числа пройденных эпох обучения (формула 2). Чем больше шум, тем большую *впадину* на графике ошибки по весу ищет метод. С уменьшением шума происходит более точная настройка весов. С некоторого момента добавочный шум вовсе перестает играть роль, и алгоритм превращается в базовый *RPROP*.

$$dW_{add} = k_3 * E(t) * r * 2^{-T*epoch}, k_3 = 3.0 \quad (2)$$

Здесь r – случайное число из интервала (0,1), $epoch$ – количество прошедших эпох обучения. Параметр T – температура метода – определяет скорость затухания величины добавочного шума. В программном комплексе есть возможность выбора из трех вариантов метода: с температурами 0.01, 0.015 и 0.02. В основной работе [27] предлагается несколько изменить способ вычисления градиента – добавить некоторую величину, также экспоненциально затухающую со временем (формула 3).

$$dE/dW_i(t)^{SARPROP} = dE/dW_i(t) - k_1 * W_i * 2^{-T*epoch}, k_1 = 0.01 \quad (3)$$

Цель этого изменения – ограничение размеров весов в начале обучения, до того как некоторые веса начнут расти бесконечно, для более полного исследования пространства весов в малых величинах. Тестирование показало, что данная оптимизация не оказывает влияния на обучаемость классификатора статуса блока, поэтому для уменьшения времени обучения сети, она не включена в текущую версию программного комплекса. Реализация *SARPROP* отличается в случае изменения знака градиента.

```
для всех весов
{
   если ( градиент сохранил знак )
   {
      // увеличить шаг
      dWi(t) = min( max_delta, dWi(t-1) * n+ );

      // изменить вес на шаг, против текущего градиента
      Wi(t) = Wi(t-1) – sign( (dE/dWi)(t) ) * dWi(t);
   }
   если же ( градиент изменил знак )
   {
      если ( dWi(t-1) < k2 * E^2(t) )
      {
         // уменьшить шаг и добавить случайную величину
         dWi(t) = dWi(t-1) * n- + k3 * E(t) * r * 2^(-T*epoch);
      }
      иначе
      {
         // уменьшить шаг
         dWi(t) = max( min_delta, dWi(t-1) * n- );
      }

      // обнулить переменную, хранящую градиент
      (dE/dWi)(t-1) = 0;
   }
   иначе
   {
      // изменить вес на шаг, против текущего градиента
      Wi(t) = Wi(t-1) – sign( (dE/dWi)(t) ) * dWi(t);
   }
}
```

1. 4. 5. Подробнее о реализации

Изначально нейронная сеть была реализована на *C++* в виде подключаемой библиотеки (*dll*) и с обширным использованием *STL*. Эта версия использовалась на протяжении большей части исследований. Однако на этапе оптимизации программного комплекса было принято решение интегрировать код библиотеки в проект (обращение к функции оценки статуса оказалось узким местом). Одновременно, контейнеры STL во многих случаях были заменены на стандартные массивы, что позволило значительно оптимизировать код на низком уровне.

2. ОБУЧЕНИЕ НЕЙРОСЕТИ

Данная глава посвящена сравнительному анализу характеристик нейронных сетей различной топологии и различных методов обучения, в применении к задаче классификации блоков. Оцениваются такие параметры, как скорость обучения сетей различных топологий, сходимость различных методов обучения (минимальная ошибка и количество эпох обучения до ее достижения). Сравниваются различные наборы характеристик блока, подаваемые на вход сети.

Отдельный параграф посвящен тренировке каскада классификаторов.

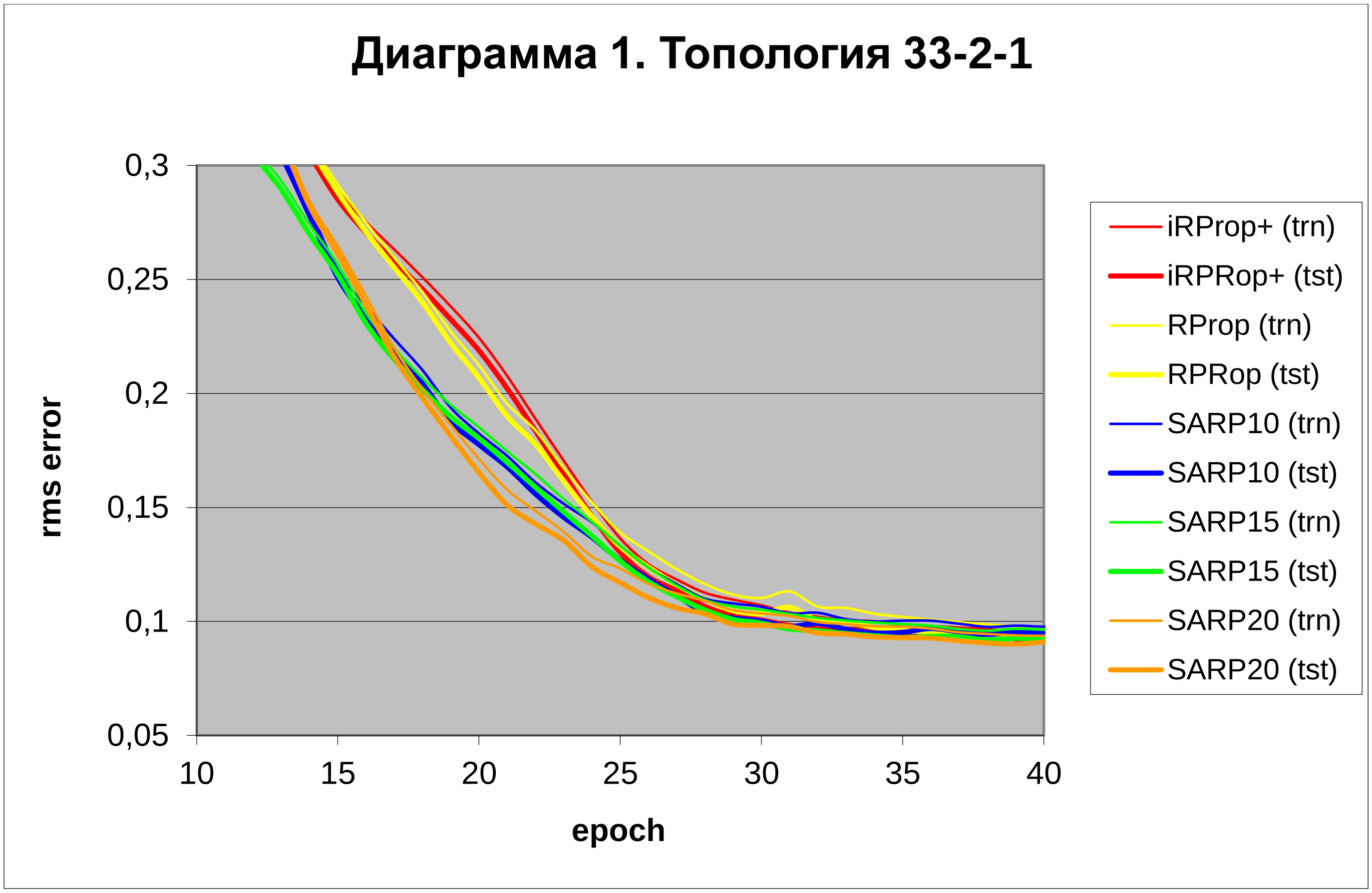
2. 1. Сравнение методов на малом множестве примеров

Итак, в нашем распоряжении пять методов обучения нейронных сетей: базовый *RPROP*, его (априори) оптимизированная по скорости сходимости версия *iRPROP+*, и три варианта *SARPROP* с различным параметром Т (0.01, 0.015 и 0.02). Кроме этого, на вход нейронной сети можно подать от одной до девяноста характеристик данного блока и его окружения. Есть возможность варьировать топологию скрытых слоев сети – их количество и размеры первого слоя (каждый последующий создается вдвое меньше предыдущего). И, наконец, можно выход одной сети использовать на входе в другую сеть – то есть тренировать каскад классификаторов. В работе [1] каскадное обучение заслужило отдельную похвалу автора.

В процессе отладки и тестирования программного комплекса были выявлены некоторые закономерности, позволившие несколько оптимизировать процесс исследования. В зависимости от топологии сети и объема входного тренировочного множества сильно менялось поведение методов «*имитации отжига*» (*SARPROP*). Все диаграммы в данном разделе построены по результатам обучения сетей на множестве из 37034 объектов (10% блоков – тестовые экземпляры).

2. 1. 1. Топология 33-2-1

Первая диаграмма построена на основе десяти независимых запусков процесса обучения для пяти классификаторов (пяти методов обучения) с двумя нейронами в скрытом слое.



В качестве ошибки выступает корень квадратный из нормализованной суммы квадратов отклонений результата классификации от верного. Приемлемый уровень ошибки был достигнут за 40 эпох. Диаграмма построена сглаживающими линиями по средним ошибкам за десять запусков тренировочного процесса (сеть каждый раз создавалась заново).

Окончательный результат тренировки приведен в табл. 1.

Таблица 1.

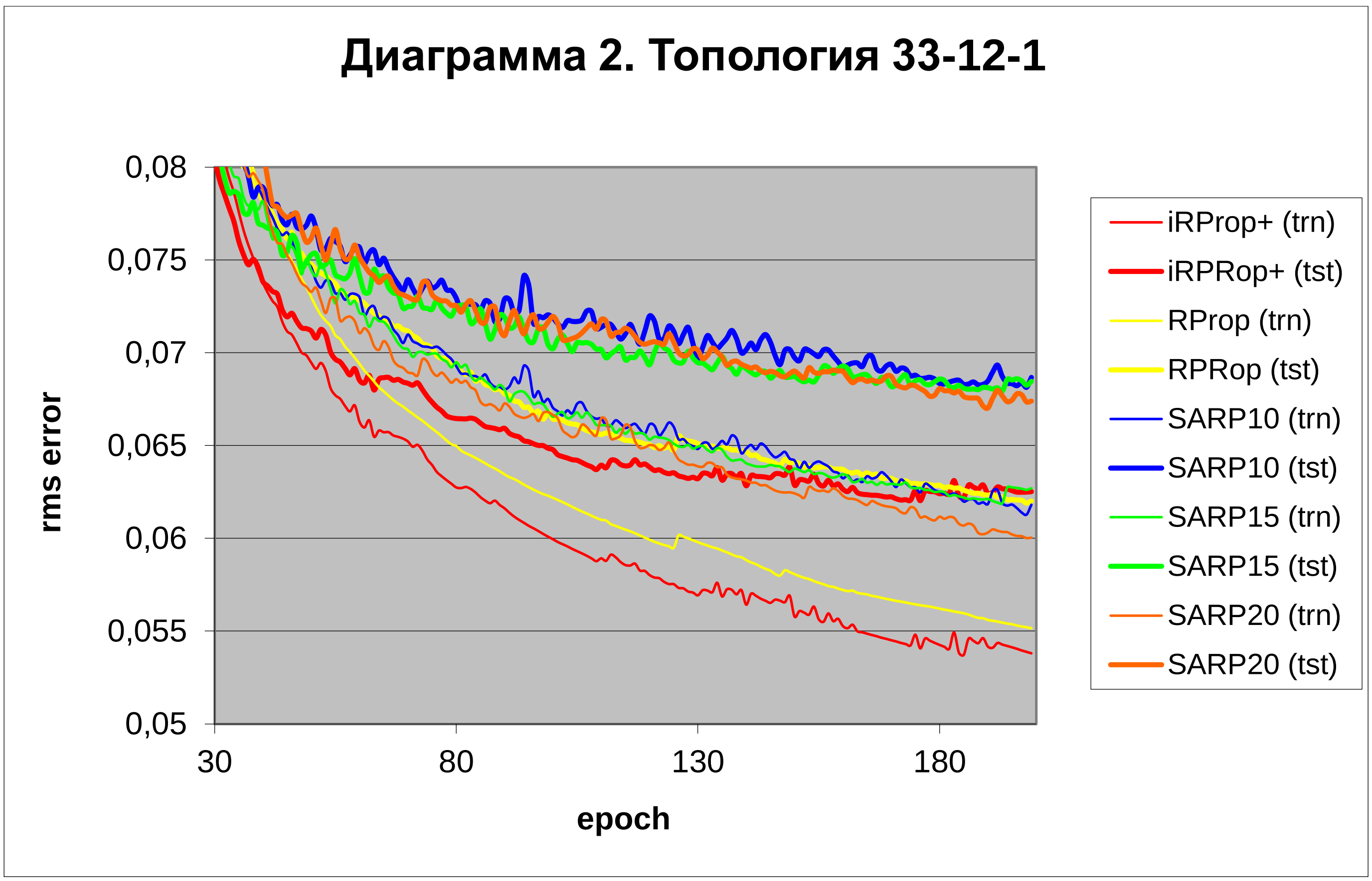
	Средняя тренировочная ошибка	Средняя тестовая ошибка	Среднее количество эпох обучения
iRPROP+	0,066112	0,066501	1601
RPROP	0,06355	0,064043	2282
SARPROP (T=0.010)	0,071659	0,072811	1344
SARPROP (T=0.015)	0,069932	0,071709	1558
SARPROP (T=0.020)	0,066833	0,069515	1828

Характерная особенность сетей всего с двумя нейронами в скрытом слое – очень низкая переобучаемость (что позволило натренировать набор сетей без лимита по количеству эпох, до того момента, пока ошибка на тренировочном множестве не перестала изменяться), сравнительно неплохой минимум ошибки на тестовом множестве и высочайшая скорость тренировки сети (из всех вариантов).

Из табл. 1 видно, что *iRPROP+* обладает лучшим соотношением скорости обучения и минимума тестовой ошибки. Из алгоритмов «имитации отжига» самым быстрым оказался алгоритм с температурой 0.01, однако у него и самая большая тестовая ошибка. Несмотря на заявленную сходимость к более глобальному минимуму, алгоритм *SARPROP* показал худшие результаты, нежели базовый алгоритм *RPROP*.

2. 1. 2. Топология 33-12-1

Следующая диаграмма соответствует классификаторам топологии 33-12-1, тренировочное и тестовое множества те же (1024 партии, 37034 блоков, 10% – тестовые экземпляры).



Данной топологии свойственно легкое переобучение, запоминание тренировочного множества. Классификаторы обучались в течении двухсот эпох. В табл. 2 приведены минимальные и окончательные усредненные ошибки для каждого метода.

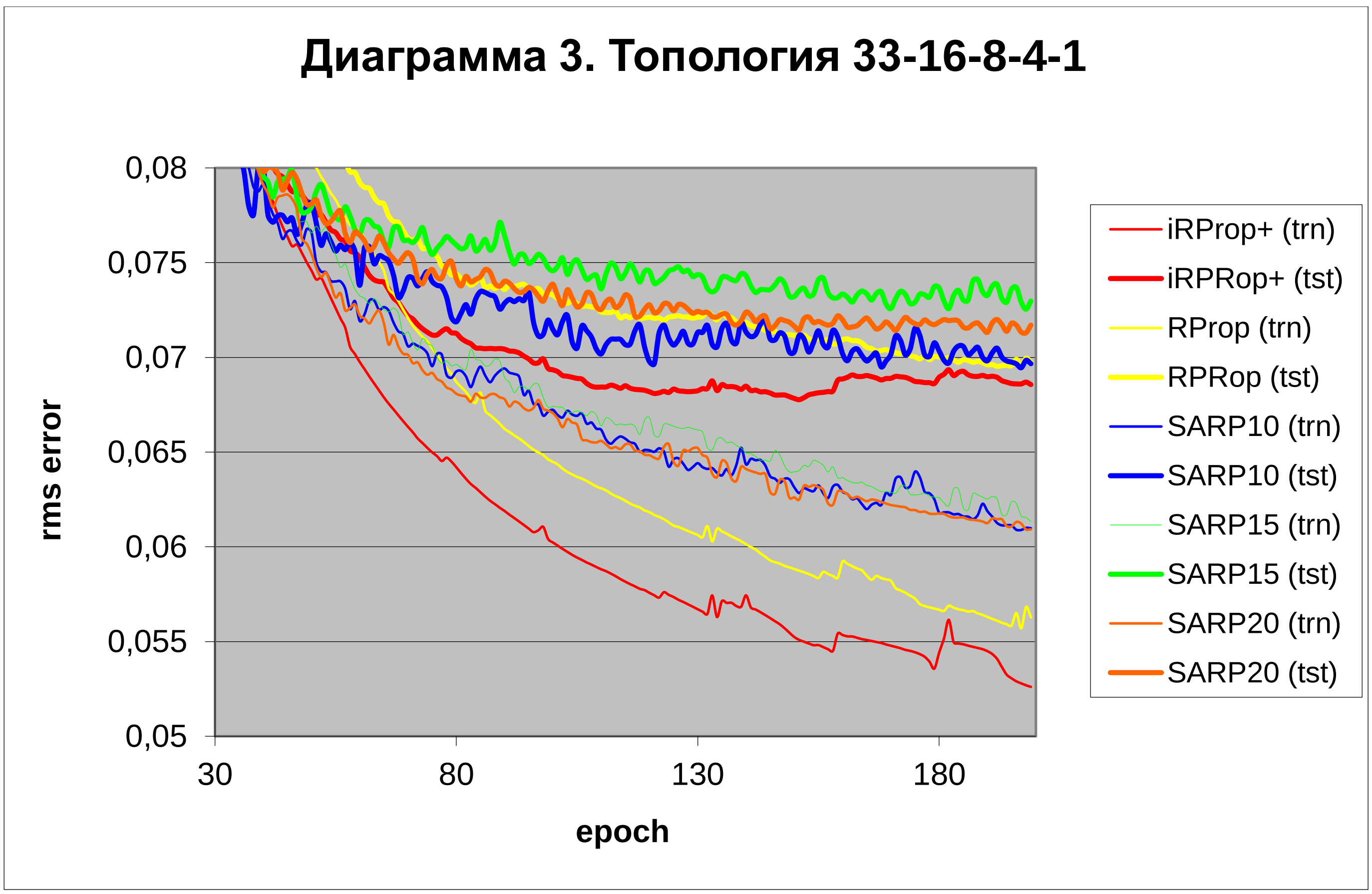
Учитывая, что количество весов в данной топологии нейронной сети увеличилось в шесть раз, по сравнению с предыдущей топологией (33-2-1), эффективность обучения выглядит лучше. За 200 эпох обучения, что по времени (благодаря кэшированию характеристических векторов) соответствует 1200 эпохам для 33-2-1, был достигнут лучший минимум тестовой ошибки, хотя и сильнее отличающийся от тренировочной ошибки.

Таблица 2.

	Тренировочная ошибка		Тестовая ошибка	
	минимальная	окончательная	минимальная	окончательная
iRPROP+	0,053715	0,053802	0,062031	0,062516
RPROP	0,055154	0,055154	0,0619	0,062014
SARPROP (T=0.010)	0,061279	0,061812	0,06814	0,068674
SARPROP (T=0.015)	0,061899	0,062699	0,067938	0,068462
SARPROP (T=0.020)	0,060015	0,060034	0,067023	0,067398

2. 1. 3. Топология 33-16-8-4-1

Для иллюстрации обучения многослойных перцептронов была построена диаграмма для перцептрона с топологией 33-16-8-4-1.



Приемлемый уровень ошибки достигается алгоритмом *SARPROP* всего за 30 эпох обучения, однако по результатам обучения (200 эпох) алгоритмы *RPROP* и *iRPROP+* показывают лучшие тестовые ошибки. Заметно, что абсолютный минимум достигнут алгоритмом *iRPROP+* за 150 эпох. В дальнейшем, по сравнению с 33-12-1 еще более растет переобучаемость сети – тренировочная и тестовая ошибки обучения отличаются более чем на 0.01. Одновременно ухудшилась средняя результативность обучения сети (табл. 3).

Таблица 3.

	Тренировочная ошибка		Тестовая ошибка	
	33-12-1	33-16-8-4-1	33-12-1	33-16-8-4-1
iRPROP+	0,053802	0,052609	0,062516	0,068566
RPROP	0,055154	0,056276	0,062014	0,069749
SARPROP (T=0.010)	0,061812	0,060984	0,068674	0,069668
SARPROP (T=0.015)	0,062699	0,06134	0,068462	0,072975
SARPROP (T=0.020)	0,060034	0,060922	0,067398	0,071705

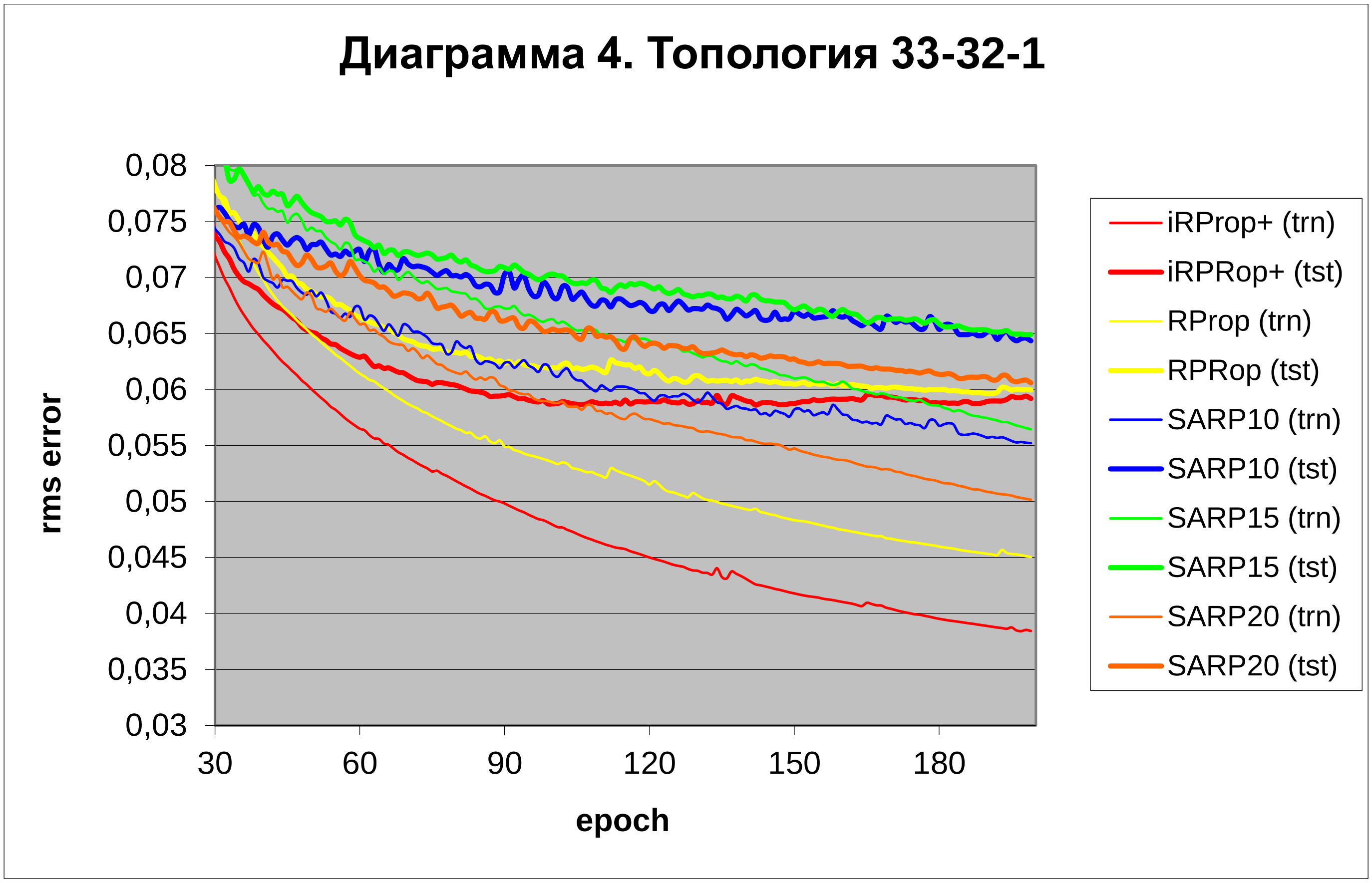
Обучаясь до момента сходимости в целом быстрее базового *RPROP*, алгоритм *iRPROP+* уступает ему в скорости достижения приемлемого результата (в начале обучения, первые 25-30 эпох) и в уровне финальной тестовой ошибки. Однако он чуть лучше запоминает тренировочное множество. Относительно данной топологии сети нужно отметить ее низкую эффективность в аппроксимации неизвестных входных данных, наравне с лучшей интерполяцией тренировочных. Помимо результатов данной диаграммы, в процессе отладки программного комплекса также было заметно отставание многослойных перцептронов в плане эффективности и скорости обучения. Поэтому в дальнейшем исследовались только однослойные классификаторы.

2. 1. 4. Топология 33-32-1

Четвертая диаграмма построена для сетей с топологией 33-32-1. Сразу надо отметить, что с ростом размера скрытого слоя сети все более склонны к запоминанию тренировочного множества. Однако заметно и снижение тестовой ошибки (см. табл. 4).

Таблица 4.

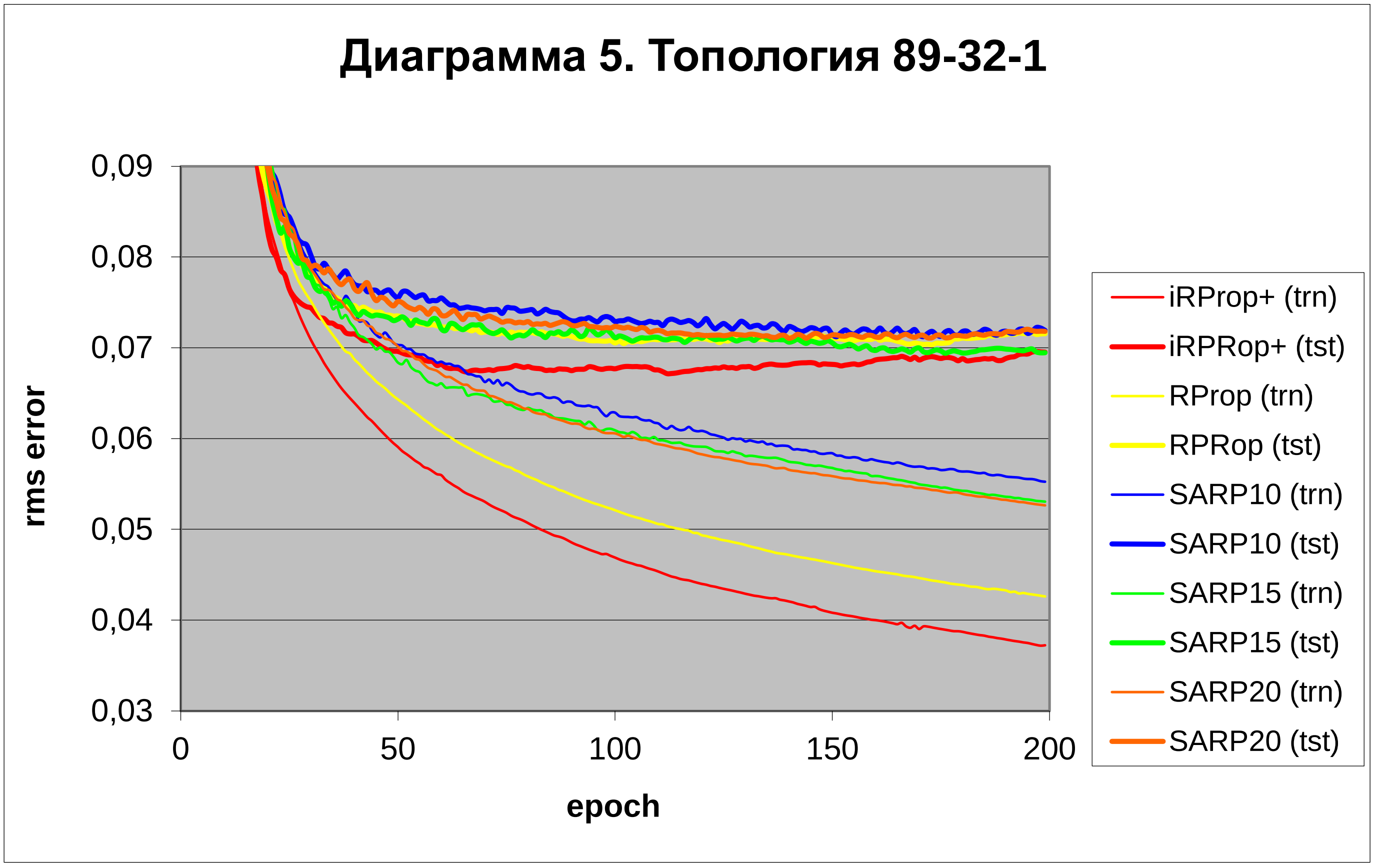
	Тренировочная ошибка		Тестовая ошибка	
	33-12-1	33-32-1	33-12-1	33-32-1
iRPROP+	0,053802	0,038431	0,062516	0,059191
RPROP	0,055154	0,045037	0,062014	0,059953
SARPROP (T=0.010)	0,061812	0,055212	0,068674	0,064346
SARPROP (T=0.015)	0,062699	0,056445	0,068462	0,064911
SARPROP (T=0.020)	0,060034	0,050157	0,067398	0,060596



На диаг. 4 отчетливо видно, что *iRPROP+* достиг минимума тестовой ошибки всего за сотню эпох обучения. Также заметно, что алгоритмам «имитации отжига» недостаточно для этого и двухсот эпох обучения. Поэтому несмотря на довольно низкую скорость обучения, набор сетей с данной топологией был обучен также и на полном множестве финальных позиций (результаты анализа см. в параграфе 2.2.3).

2. 1. 5. Топология 89-32-1

В процессе отладки и тестирования приложения были отмечены некоторые особенности обучения с большим размером характеристического вектора блока. Как было описано в параграфе 1.3.2, полный набор характеристик блока включает свойства многих окружающих его объектов. Изначально, в попытке обучить лучший классификатор, в программный комплекс добавлялись новые и новые характеристики, что однако не слишком заметно улучшало итоговый классификатор. Несмотря на более быстрое достижение минимума ошибки (до сотни эпох методом *iRPROP+*), минимальное значение ошибки с добавлением новых характеристик постепенно росло. Большинство новых качеств обладало слабо коррелировало с живучестью группы. В результате, поиск лучшего классификатора пошел по новому пути – было отобрано множество из наиболее релевантных характеристик (33 характеристики – на них и построены почти все диаграммы). Однако для наглядности была построена и диаграмма обучения классификаторов различными методами с использованием всех входных данных, предоставляемых программой (диаг. 5).



Из диаг. 5 видно, что алгоритм *iRPROP+* лучше всего проявляет свои качества на нейронных сетях с большим вектором входных данных. Практически все методы (за исключением *SARPROP* с параметром T = 0.015) успели обучить сеть до минимума ошибки за 200 эпох, однако лучшего результата как по скорости, так и по уровню конечной ошибки, достиг метод *iRPROP+*. Одна из сетей, обученная этим методом достигла минимума ошибки в 0,06 всего за 85 эпох обучения.

2. 1. 6. Резюме

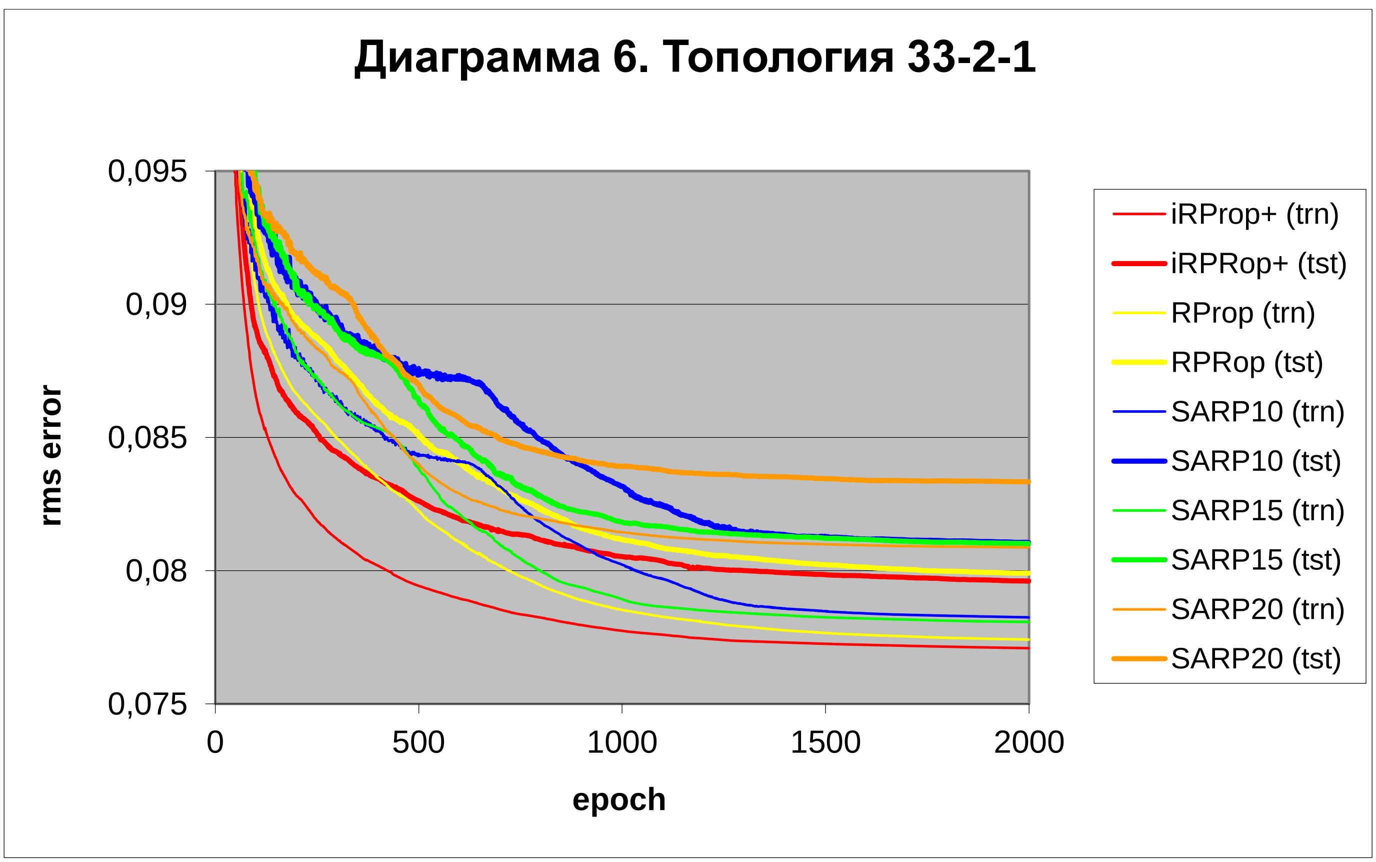
В результате исследования обучения классификаторов различных топологий на малом множестве примеров была отмечена низкая эффективность двух топологий: 33-16-8-4-1 (многослойный перцептрон) и 89-32-1 (использовавшая полный вектор входных данных – 89 характеристик). Нейронные сети 33-2-1, 33-12-1 и 33-32-1 были обучены на полном множестве финальных позиций для получения полной картины.

2. 2. Сравнение методов на полном множестве примеров

Как и в предыдущем параграфе, рассмотрим пять методов обучения нейронных сетей: *RPROP*, *iRPROP+* и *SARPROP* с параметром T = 0.01, 0.015 и 0.02. Полное множество входных данных состоит из 10247 финальных позиций, в которых статический анализатор выделяет 367726 блоков, подлежащих классификации. Для каждой топологии и каждого метода обучения строилось десять классификаторов.

2. 2. 1. Топология 33-2-1

Диаграмма 6 построена для самого быстрого в обучении (и применении) классификатора с двумя нейронами в скрытом слое. По сравнению с обучением на малом множестве уровень финальной ошибки оказался несколько выше. Обучение производилось до тех пор, пока ошибка на валидационном множестве не переставала снижаться – если ошибка была выше минимальной, достигнутой в течении 100 эпох, процесс обучения прерывался.

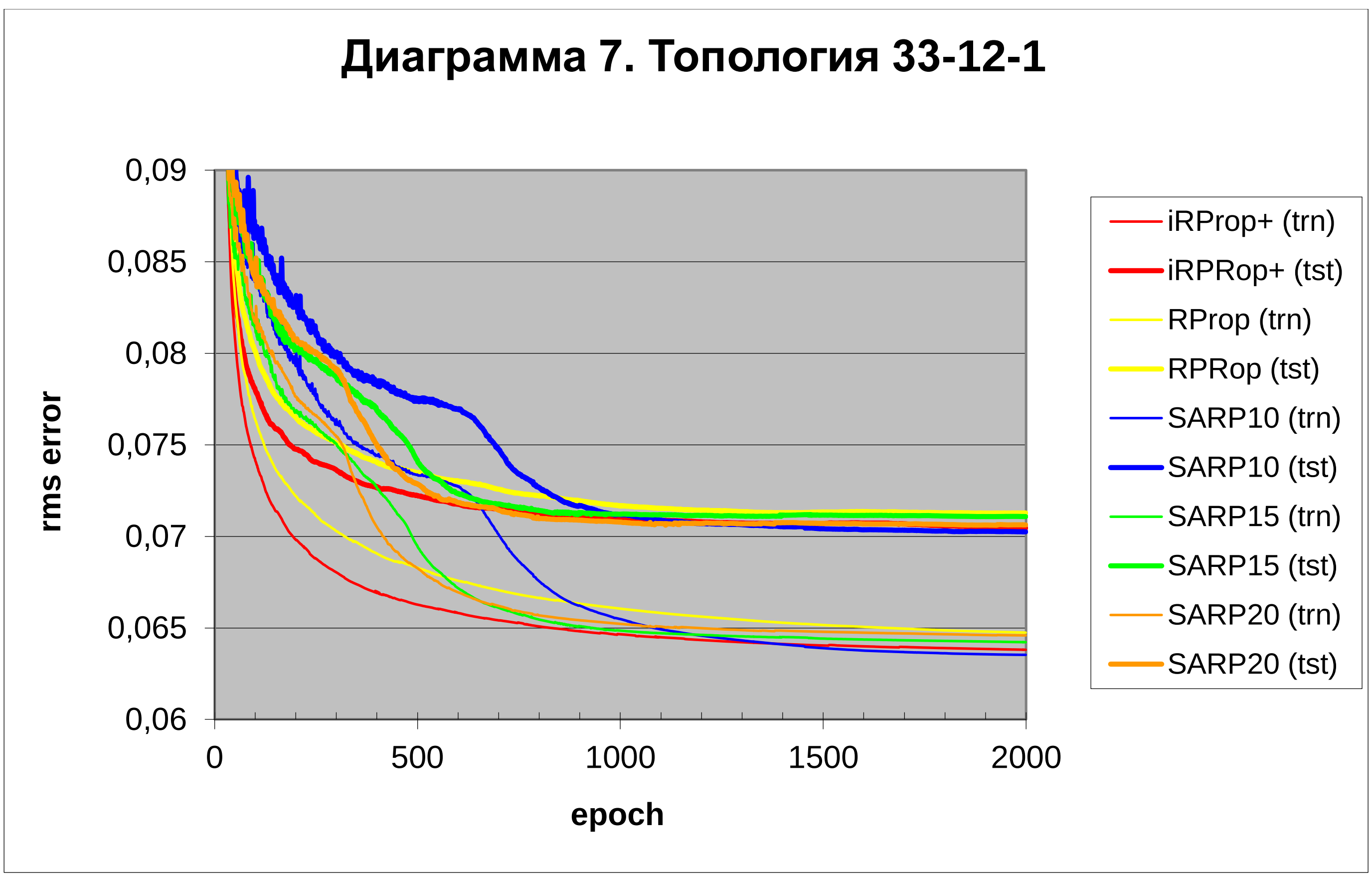


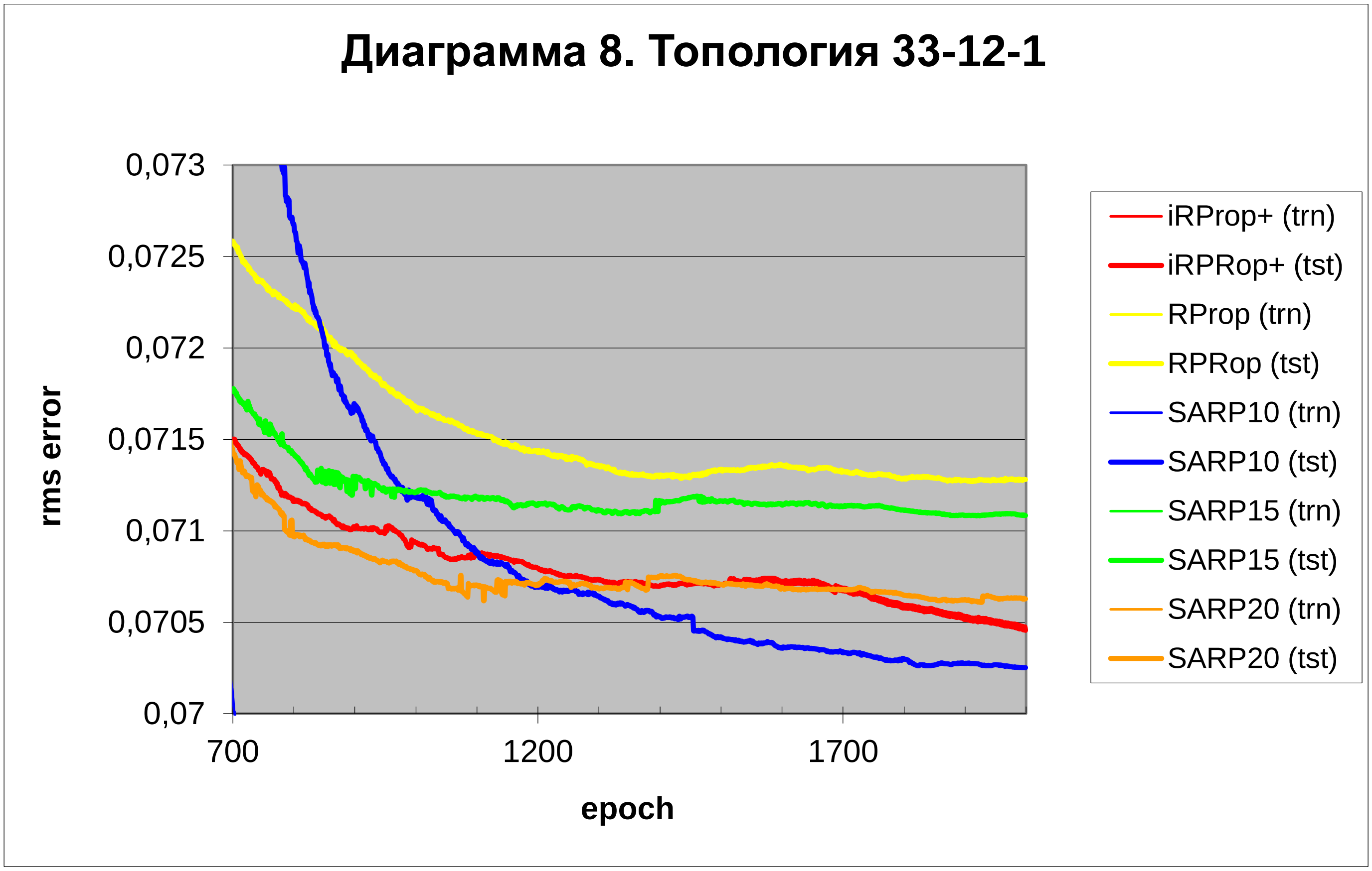
Самую высокую скорость обучения и лучший результат показал метод *iRPROP+*. У линии обучения методом *SARPROP* (с T = 0.01) на диаг. 6, стал заметен горб. Его происхождение обусловлено постепенным экспоненциальным уменьшением величины добавочного шума и порога шага ошибки, при котором шум добавляется. Начиная с некоторого момента метод становится практически идентичен базовому *RPROP*, за исключением отсутствия возврата в случае смены знака градиента (см. псевдокод метода *SARPROP* в параграфе 1.4.4).

Особенность большого количества входных экземпляров заключается в том, что нейронные сети одной топологии, обучавшиеся одним методом, показывали сильно различающуюся скорость обучения от одного запуска к другому. Иными словами, дисперсия количества эпох до окончания обучения оказалась очень высока. При среднем количестве в 2200 эпох (для метода *iRPROP+*), самая быстрая сеть обучилась за 718 эпох, а самая медленная – за 4805.

2. 2. 2. Топология 33-12-1

Диаграммы 7 и 8 построены для топологии 33-12-1 на полном множестве входных данных. Критерий остановки обучения – малое изменение валидационной ошибки.





Метод *RPROP* впервые отстал от всех остальных методов обучения по уровню финальной ошибки на тестовом и тренировочном множествах. У всех методов «имитации отжига» на диаг. 7 заметны горбы (следствие постепенного исчезания добавки шума).

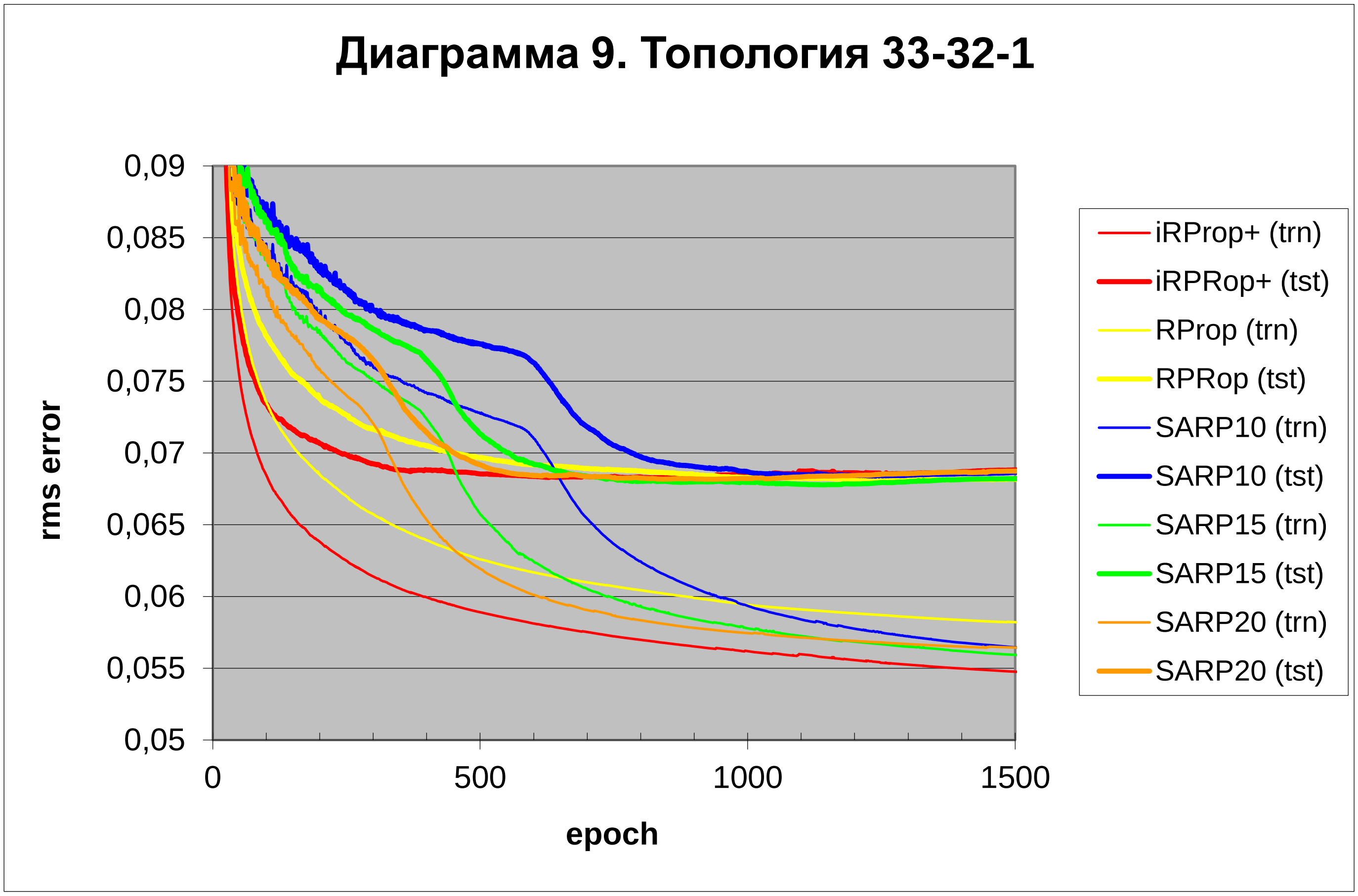
Таблица 5.

	Тренировочная ошибка		Тестовая ошибка		Среднее время обучения
	средняя конечная	минимальная	средняя конечная	минимальная	
iRPROP+	0.063397	0.062362	0.070372	0.068914	2764
RPROP	0.06446	0.062941	0.071167	0.070053	2263
SARPROP (T=0.010)	0.063477	**0.059786**	0.070229	**0.068234**	1815
SARPROP (T=0.015)	0.063859	0.060899	0.070952	0.068375	1873
SARPROP (T=0.020)	0.064539	0.063478	0.070636	0.06861	1609

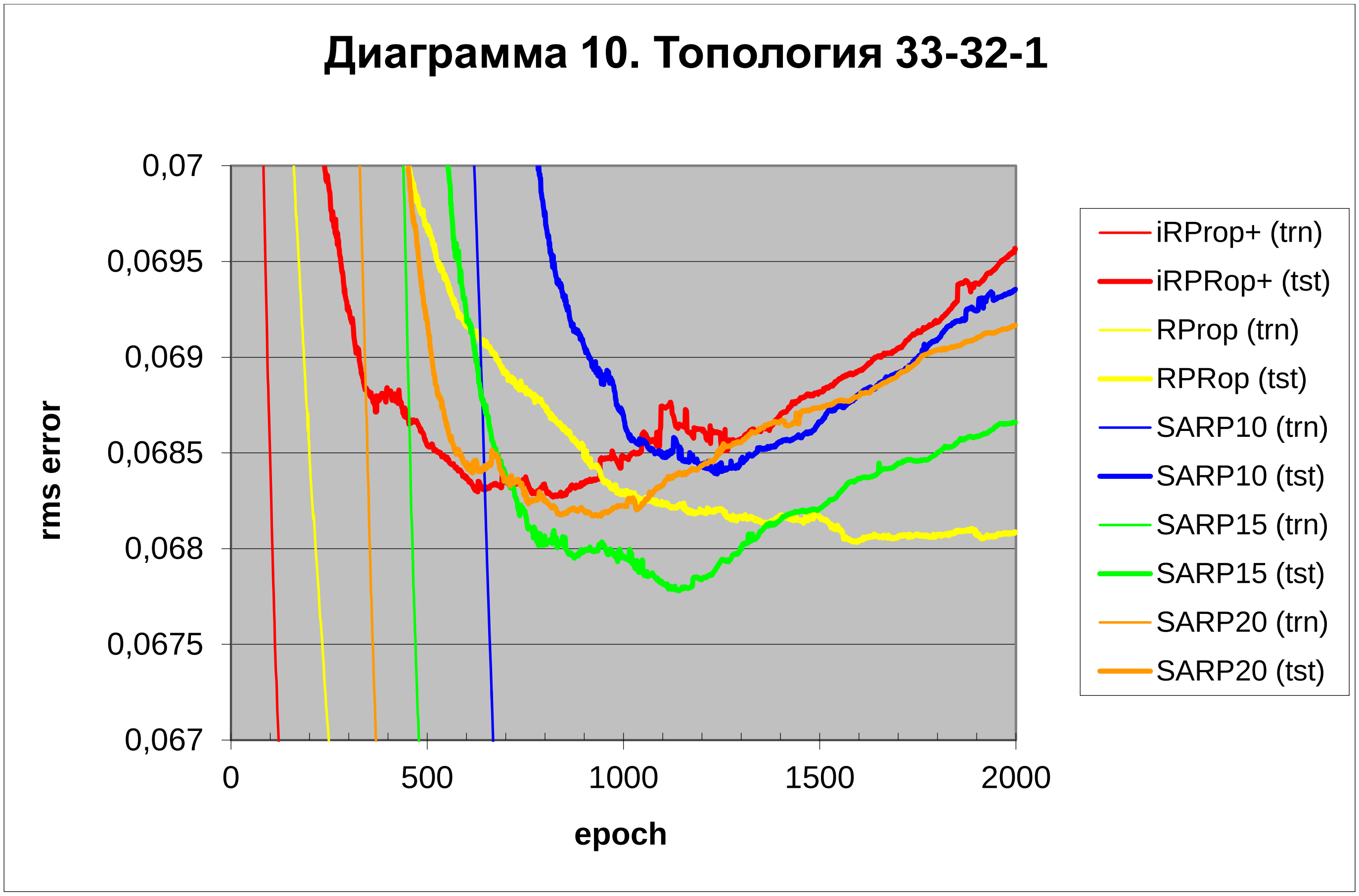
Из табл. 5 видно, что метод *SARPROP* (T = 0.01) показал лучший результат по уровню тестовой ошибки, однако на диаг. 7 он сильно отстает от остальных методов по времени достижения приемлемого уровня ошибки. Все методы показали лучший результат для данной топологии по сравнению с топологией 33-2-1.

2. 2. 3. Топология 33-32-1

Диаграммы 9 и 10 построены для топологии 33-32-1 на полном множестве входных данных. Критерий остановки обучения – малое изменение валидационной ошибки. Из-за размеров множества и количества нейронов в скрытом слое время построения диаграммы составило более 100 часов на процессоре Intel Core Duo с частотой 1,83 ГГц. Поэтому более емкие топологии на полном множестве не исследовались.



Метод обучения *iRPROP+* продемонстрировал максимальную скорость обучения, не достигнув, однако, лучшего результата (диаг. 10) – однако разница не составила даже одной тысячной. Учитывая общую низкую скорость обучения данной топологии, *iRPROP+* по праву может называться самым эффективным. Низкий параметр T = 0.01 (температура методов «имитации отжига») еще сильнее сдвинул линию графика *SARPROP* вправо, не улучшив, по сравнению с остальными вариантами, минимума ошибки.



2. 2. 4. Резюме

По результатам обучения классификаторов различных топологий на полном множестве входных данных наиболее эффективным признан метод *iRPROP+*. Учитывая довольно большую дисперсию как времени обучения, так и минимума ошибки у всех методов, метод *iRPROP+* позволяет в среднем быстрее прочих построить эффективный классификатор.

Поскольку область применения классификатора статуса блоков в конце партии не связана с расчетом большого количества вариантов, вполне допустимо, с практической точки зрения, использовать наиболее крупный классификатор 33-32-1. Можно было бы попробовать построить и более крупные классификаторы (и работа в этом направлении некоторое время велась), однако выигрыш составил совсем малую часть ошибки, и в то же время значительно увеличился срок обучения. Это сделало невозможным представить в работе диаграммы обучения более крупных классификаторов.

Надо отметить, что для комбинированного подхода, сочетающего расчет вариантов и применение нейронной сети, оптимально будет использовать классификатор с двумя нейронами в скрытом слое (поскольку неточность классификатора компенсируется расчетом вариантов, в то время как размеры сети прямо пропорционально влияют на скорость классификации).

2. 3. Каскады классификаторов

Основываясь на результатах, описанных в предыдущих параграфах, для тренировки каскада классификаторов была выбрана топология 33-32-1 и метод обучения *iRPROP+*. Каждая последующая сеть получала на вход дополнительные семь характеристик – предсказанные статусы блока, его цепочки, и соседних блоков. Динамические качества блоков, изменяющиеся после появления первой оценки, были рассчитаны заранее и кэшированы. Таким образом, как и в случае одной сети, за счет обучения только последнего слоя, удалось максимально ускорить процесс подачи данных на вход второго классификатора.

Несмотря на длительный процесс обучения, результаты оказались плохие – второй классификатор упорно не желал достичь результата предшественника. Применение каскада привело лишь к ухудшению оценки статуса.

Анализ свойств блоков, классифицированных неверно, показал, что большинство классификаторов верно оценивают статус блоков в приблизительно равной позиции. В случаях, когда один игрок полностью уничтожал все камни противника (что при игре на форе не редкость), классификатор неверно оценивал статусы обреченных блоков (такое ощущение, будто нейронная сеть не верит, что у игрока может быть съедено столько камней).

ЗАКЛЮЧЕНИЕ

В результате проведенной работы был построен комплекс программ для селекции партий в формате *SGF* и тренировки классификатора блоков на основе нейронной сети. Структурная схема комплекса имеет следующий вид.

1. База партий, выбранная с игрового сервера http://kiseido.com.
2. Утилита selector, проверяющая наличие результата в партии, отсутствие вариантов, и преобразующая партию к удобному виду (удаление комментариев, неинформативных тэгов).
3. Утилита trainer, обучающая нейронную сеть указанной топологии на основе некоторого набора партий. В процессе исследования настройка многочисленных параметров производилась в коде, и текущая версия «извне» (в виде аргументов к программе) получает только общие значения:
    - метод обучения (impro, rprop, sarp10, sarp15, sarp20);
    - количество сетей в каскаде;
    - количество нейронов в первом скрытом слое;
    - количество слоев (каждый последующий вдвое меньше предыдущего);
    - характеристический вектор (A ~ 33, B ~ 89);
    - количество партий для тренировки (нумерация с первой, 10% – тестовое множество);
    - количество сетей для построения;
    - номер первой сети для построения;
    - максимальное количество эпох обучения сети.
4. Утилита sortExcelData, облегчающая конвертирование множества выходных файлов в один csv файл, читаемый программой Microsoft Excel.
5. Утилита tester, подсчитывающая очки в финальной позиции указанной партии, при помощи указанной сети.

При помощи данного программного комплекса проведено ресурсоемкое исследование эффективности обучения классификаторов с различной топологией. В процессе исследования длительное время производилась настройка параметров методов обучения (типичные константы оказались не столь эффективны). Полученные результаты показали возможность обучения и работы классификатора всего на двух нейронах в скрытом слое. Наиболее эффективным с точки зрения критерия *скорость/результат* признан метод обучения *iRPROP+*. Его явное лидерство проявилось на большом объеме входных данных, в особенности для классификаторов с топологией 33-32-1.

Основные отличия проведенной работы от работы [1] (с нее все начиналось) включают: оптимизацию анализа позиции для доски 19x19, более полный список характеристик, подаваемых на вход классификатору, сравнительное исследование широкого спектра топологий нейронных сетей и продвинутых методов обучения. Лучшие классификаторы, построенные в процессе исследования верно определяют статус более чем у 99,5% блоков.

Для практической цели – классификации блоков в конце игры (а при наличии экспертной разметки и в середине игры) – на основании данной работы можно рекомендовать метод обучения классификаторов *iRPROP+*. Для однократной приблизительной оценки лучше всего использовать классификатор с достаточно большим количеством нейронов в скрытом слое. В то же время, для частой оценки статуса блока при использовании того или иного метода расчета вариантов, оптимальным будет применение наименьшего по размерам классификатора.

СПИСОК ЛИТЕРАТУРЫ

1. J.W.H.M. Uiterwijk, E.C.D. van der Werf, H.J. van den Herik. *Learning to score final positions in the game of Go* // In 10th Advances in Computer Games conference, pp. 143-158, 2003. [http://www.cs.unimaas.nl/~vanderwerf/pubdown/learning_to_score.pdf]
2. Bruno Bouzy, Tristan Cazenave. *Computer Go: an AI oriented survey* // Preprint for article in AI Journal. To be published, 2001. [http://www.math-info.univ-paris5.fr/~bouzy/CG-AISurvey.ps.gz]
3. Bruno Bouzy. *Mathematical morphology applied to computer go* // IJPRAI, 17(2), 2003. [http://www.math-info.univ-paris5.fr/~bouzy/publications/MyBouzy-IJPRAI.pdf]
4. Bruno Bouzy, Bernard Helmstetter. *Monte Carlo Go developments* // In Advances in Computer Games conference (ACG-10), Graz 2003, pp.159-174. Kluwer, 2003. [http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-helmstetter.pdf]
5. Bruno Bouzy, Guillaume Chaslot. *Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 Go* // In IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK, pp. 176-181, 2005. [http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-chaslot-cig05.pdf]
6. Bruno Bouzy, Guillaume Chaslot. *Monte-Carlo Go reinforcement learning experiments* // In IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA, 2006. [http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-chaslot-cig06.pdf]
7. Jay Burmeister, Janet Wiles. *The challenge of Go as a domain for AI research: A comparision between Go and chess* // In 3rd Australian and New Zealand Conference on Intelligent Information Systems, Perth, 1995. [http://www.itee.uq.edu.au/~janetw/Computer Go/go-vs-chess.pdf]
8. Jay Burmeister, Janet Wiles. *AI techniques used in computer Go*. In Fourth Conference of the Australasian Cognitive Science Society, Newcastle, 1997. [http://www.itee.uq.edu.au/~janetw/Computer Go/comp-go.AI.pdf]
9. David Al-Dabass, Richard Cant, Julian Churchill. *Using hard and soft artificial intelligence algorithms to simulate human Go playing techniques* // In International Journal of Simulation Systems, Science & Technology, 2(1):31-49, June 2001.

[http://ducati.doc.ntu.ac.uk/uksim/journal/Vol-2/No-1/RichardCant/Cant.pdf]

10. Heather Cook, Amanda Venghaus, and Peter Drake. *Machine learning applied to the game of Go* // Twelfth Regional Conference on Undergraduate Research, Murdock College Research Program. Conference poster, 2003.
[http://www.lclark.edu/~sciences/GOposter.pdf]
11. Fredrik A. Dahl. *Honte, a Go-playing program using neural nets* // In Fürnkranz and Kubat [FK99].
[http://www.ai.univie.ac.at/icml-99-ws-games/papers/dahl.ps.gz]
12. Paul Donnelly, Patrick Corr, Danny Crookes. *Evolving Go playing strategy in neural networks*, 1994.
[ftp://ftp-igs.joyjoy.net/Go/computer/egpsnn.ps.Z]
13. Peter Drake, Niku Schreiner, Brett Tomlin, and Loring Veenstra. *An efficient algorithm for eyespace classification in Go* // In International Conference on Artificial Intelligence. CREA Press, 2006.
[http://www.lclark.edu/~drake/go/icai2006-final-drake.pdf]
14. Reindert-Jan Ekker. *Reinforcement learning and games* // Master's thesis, Rijksuniversiteit Groningen, 2003.
[http://members.home.nl/r.j.ekker/afstudeer/tekst.pdf]
15. Markus Enzenberger. *Evaluation in Go by a neural network using soft segmentation* // In 10th Advances in Computer Games conference, pp. 97-108, 2003.
[http://www.markus-enzenberger.de/neurogo/neurogo3.pdf]
16. Christopher Fellows, Yuri Malitsky, Gregory Wojtaszczyk. *GoNN - incorporating a neural network into the game of Go* // In AI Project, CS 473, Cornell University, 2005.
[http://www.people.cornell.edu/pages/ynm2/Papers/Incorporating a Neural Net into the Game of Go - Final Report.pdf]
17. Dylan Shell, George Konidaris, Nir Oren. *Evolving neural networks for the Capture Game* // SAICSIT Postgraduate Symposium 2002, 2002.
[http://www-robotics.usc.edu/~dshell/res/evneurocapt.pdf]
18. Howard Landman. *Eyespace Values in Go* // Cambridge University Press, pp. 227-257, 1996.
[http://www.msri.org/publications/books/Book29/files/landman.pdf]
19. Byung-Doo Lee. *Life-and-death problem solver in go* // Technical Report CITR-TR-145, University of Auckland, 2004.
[http://www.citr.auckland.ac.nz/techreports/2004/CITR-TR-145.pdf]

20. Martin Müller. Playing it safe: *Recognizing secure territories in computer Go by using static rules and search* // In Game Programming Workshop in Japan '97, MATSUBARA, H. (ed.), Computer Shogi Association, Tokyo, Japan, 1997.
[http://www.cs.ualberta.ca/~mmueller/ps/gpw97.ps.gz]
21. Xiaozhen Niu. *Recognizing safe territories and stones in computer Go* // Master's thesis, Department of Computing Science, University of Alberta, 2004.
[http://www.cs.ualberta.ca/~games/go/seminar/notes/040225/thesis.pdf]
22. R. Vila, T. Cazenave. *When one eye is sufficient* // In Advance in Computer Games 10, Kluwer, 2003.
[http://www.ai.univ-paris8.fr/~cazenave/eyeLabelling.pdf]
23. J. W. H. M. Uiterwijk, E.C.D. van der Werf, H.J. van den Herik. *Learning to estimate potential territory in the game of Go* // In 4th International Conference on Computers and Games (CG'04), 2004.
[http://www.cs.unimaas.nl/~vanderwerf/pubdown/predicting_territory.pdf]
24. Shi-Jim Yen, Shun-Chin Hsu. *A positional-judgement system for computer Go* // In Advances in Computer Games, volume 9, pp. 313-326. Universiteit Maastricht, 2001.
[http://www.csie.ndhu.edu.tw/~sjyen/Papers/2001PJ.pdf]
25. Martin Riedmiller, Heinrich Braun. *A direct adaptive method for faster backpropagation learning: The RPROP algorithm* // In Proceedings of the IEEE International Conference on Neural Networks 1993 (ICNN 93), 1993.
[http://citeseer.ist.psu.edu/riedmiller93direct.html]
26. Christian Igel, Michael Hüsken. *Improving the Rprop Learning Algorithm*. // In H. Bothe and R. Rojas, editors, Second International Symposium on Neural Computation (NC 2000), pp. 115-121, ICSC Academic Press, 2000.
[http://citeseer.ist.psu.edu/igel00improving.html]
27. N.K. Treadgold, T.D. Gedeon. *The Sarprop Algorithm: A Simulated Annealing Enhancement To Resilient Back Propagation* // In Neural Networks, IEEE Transactions on Neural networks, volume 9, issue 4, pp. 662 - 668, 1998.
[http://citeseer.ist.psu.edu/157594.html]
28. K. Nicholas, N.K. Treadgold. *Constructive neural networks : generalization, convergence and architectures* // University of New South Wales, 1999.

[http://www.library.unsw.edu.au/~thesis/adt-NUN/public/adt-NUN20010223.135802/index.html]

29. *ГоБиблиотека* - крупный информационный го-проект в рунете. [http://rusgolib.iponweb.net]
30. Спецификация Smart Game Format, версия FF[4], онлайн источник. [http://red-bean.com/sgf/]

Большинство ссылок на англо-язычные документы, использованные в работе могут быть найдены в актуальной и постоянно пополняемой библиографии компьютерного Го, по адресу: http://www.markus-enzenberger.de/compgo_biblio/compgo_biblio.html.