Численная реализация алгоритмов селективного комбинирования разнородных представлений объектов в задачах распознавания образов

***Н. А. Разин*[1], *В. В. Моттль*[2]**
nrmanutd@gmail.com[1], vmottl@yandex.ru[2]
МФТИ(ГУ)[1,2]; ВЦ РАН[2]

В работе рассматриваются методы решения задач беспризнакового распознавания образов в предположении, что объекты попарно сравниваются при помощи произвольной действительной функции. Такой подход является гораздо более общим, чем традиционный метод потенциальных функций (кернелов), требующий положительной полуопределенности матрицы функции сравнения объектов. Важное преимущество предлагаемых алгоритмов перед существующими заключается в том, что они хорошо распараллеливаются на современных многопроцессорных вычислительных системах, что позволяет использовать мощные кластеры для быстрого решения задач с большим объемом данных. Полученное ускорение по сравнению с наивными реализациями алгоритмов доходит до 25 раз на сравнительно слабой по мощности видеокарте NVidia GeForce 310M.

Ключевые слова: *беспризнаковое распознавание образов, машина релевантных векторов, алгоритмы умножения матриц, численные алгоритмы решения систем линейных уравнений, параллелизуемые алгоритмы.*

Numerical algorithms for selective multimodal pattern recognition

***N. A. Razin*[1], *V. V. Mottl* [2]**
MIPT[1,2]; Computing Centre of the Russian Academy of Sciences[2]

The authors address the problem of regression estimation under the assumption that pair-wise comparison of objects is arbitrarily scored by real numbers. Such a linear embedding is much more general than the traditional kernel-based approach, which demands positive semidefiniteness of the matrix of object comparisons. The advantage of proposed algorithms is their good scalability at multiprocessor systems, leading to use powerful clusters for solving pattern recognition problems for large data. The resulting computational speed for algorithms tested on usual video card NVidia GeForce 310M is 25 times faster compared to naive algorithms implementation.

Keywords: *machine learning, svm, rvm, gpu, cuda, parallel computations.*

Введение

Построение стандартного классификатора SVM (support vector machine) [1] сводится к решению задачи квадратичного программирования. Как следует из [2], появляются неквадратичные задачи SVM, которые сводятся к решению выпуклой задачи оптимизации, но скорость работы стандартных методов становится узким горлышком при построении разделяющей гиперплоскости. Один из известных подходов к построению стандартного классификатора SVM — использовать эвристический алгоритм SMO (sequential minimal optimization) [3]. Во-первых, SMO заточен под решение именно квадратичной задачи, к которой сводится классический SVM, а значит неквадратичные задачи [2] выпуклой оптимизации этим алгоритмом решать нельзя. Во-вторых,

SMO делает большое количество итераций, поэтому на очень больших объемах данных его ускорить не удастся.

Существующие способы, оптимизирующие скорость построения классификатора SVM, не годятся по следующим причинам:

— способы [3], [4], [5], [6] заточены под классический SVM с квадратичной задачей;

— способы класса [4] делают очень много «легковесных» итераций. Естественно, если выборка большая, такие алгоритмы будут работать очень долго. Задействовать мощности современных кластеров в случае с SMO либо с другим аналогичным алгоритмом, делающим много итераций, невыгодно.

Решение критерия с квадратично-модульной регуляризацией аналитически выписать не удается, во-первых, из-за модуля, а, во-вторых, из-за линейных ограничений. Поэтому для поиска точки оптимума используется двойственная задача и итерационные методы решения задач математического программирования. Двойственная задача к критерию с квадратично-модульной регуляризацией является стандартной задачей квадратичного программирования с линейными ограничениями. Для ее решения можно применять следующие классические методы численной условной оптимизации:

— метод проекции градиента;

— метод условного градиента;

— метод внутренней точки (метод штрафных функций).

Каждый из перечисленных методов обладает своими преимуществами и недостатками. Нас в первую очередь интересует скорость работы этих методов и требуемая для их работы память. Метод проекции градиента и метод условного градиента требуют линейных расходов по памяти в зависимости от размерности задачи. Однако их слабая сторона — количество итераций, которые эти методы делают. Очень часто даже в задачах небольшой размерности ($N \sim 100$) эти методы выполняют тысячи итераций. Естественно, что как бы отдельная итерация ни была ускорена, все равно принципиально большой скорости достигнуть не получится в силу того, что этих итераций выполняется очень много.

Метод внутренней точки в этом смысле обладает гораздо более высокими показателями скорости. Даже на больших объемах данных ($N \sim 1000$) этот метод выполняет десятки итераций. Однако каждая итерация вычислительно оказывается тяжелой. Связано это с тем, что метод внутренней точки сводит исходную задачу математического программирования к задаче безусловной оптимизации, добавляя к целевой функции штрафные функции, штрафующие выход переменных оптимизации за пределы области допустимых ограничений. Сама задача безусловной оптимизации решается методом Ньютона, что означает, что на каждой итерации нужно перемножить матрицы размеров $N \times N$ и решить систему линейных уравнений размеров $N \times N$. Первая из этих операций очень хорошо параллелится. Вторая операция тоже может быть распараллелена, но менее эффективно. Метод внутренней точки требует много памяти для хранения матриц размеров $N \times N$, то есть потребление памяти в этом случае растет квадратично от размерности задачи.

В данной работе предлагается подход к реализации обучения SVM, основанный на методе внутренней точки [7]. Во-первых, этим методом можно решать произвольную задачу выпуклой оптимизации, целевая функция которой дважды непрерывно дифференцируема. Во-вторых, метод в среднем делает существенно меньшее количество итераций по сравнению с SMO и имеет отличный потенциал для ускорения через распараллеливание. Ясно, что квадратичное использование памяти делает этот метод неприменимым при работе с очень большим количеством данных.

В рамках текущей работы авторы не столкнулись с задачами такой размерности, при которой становится невозможным применять метод внутренней точки.

Выпуклый критерий обучения — дважды регуляризованная машина опорных векторов

В данной статье мы исходим из классической версии SVM, предназначенной для отбора признаков и названной в [8] дважды регуляризованной машиной SVM.

Как и в классической SVM, предполагается, что объекты реального мира $\omega \in \Omega$ описываются k признаками $\mathbf{x}(\omega) = (x_1(\omega) \ldots x_k(\omega))$, которых значительно больше, чем объектов N в обучающей выборке. Задача поиска оптимальной разделяющей гиперплоскости $\mathbf{a}^T\mathbf{x} + b = \sum_{l=1}^{k} a_l x_l + b \gtrless 0$ в $\mathbb{R}^k$ для заданной обучающей совокупности

$$\begin{cases} \sum\limits_{l=1}^{k} \left[\beta a_l^2 + \mu|a_l|\right] + \sum_{j=1}^{N} \delta_j \to \min\limits_{a_l,b,\delta_j}; \\ y_j(\sum\limits_{l=1}^{k} a_l x_{lj} + b) \geqslant 1 - \delta_j, j = 1, \ldots, N, \\ \delta_j \geqslant 0, j = 1, \ldots, N, \end{cases}$$

отличается от стандартной задачи обучения SVM более сложной регуляризацией, заключающейся в комбинировании норм L_1 и L_2 направляющего вектора с весами β, μ вместо чистой нормы L_2 в классическом случае.

Если $\mu = 0$, критерий превращается в классический SVM. Если же $\mu \to +\infty$, критерий становится сильно селективным, отбрасывая практически все признаки. Это означает, что, меняя параметр μ, можно менять характер обучения от сохранения всех признаков до исключения максимального их количества из решающего правила.

В отличие от классической машины SVM критерий (1) уже не является квадратичным, но остается выпуклым.

Машина RVM с произвольными функциями парного сравнения

Критерий обучения (1) остается полностью применимым в ситуации, когда представление объектов при помощи функцию сравнения $S(\omega', \omega'')$ более удобно, чем при помощи признакового описания $\mathbf{x}(\omega)$. Значения функции сравнения объекта $\omega \in \Omega$ на каждом из N объектов обучающей совокупности $x_l(\omega) = S(\omega_l, \omega)$ могут быть использованы как его вторичные признаки [9]. В этом случае мы получаем обобщенную версию RVM, которую следует назвать Relevance Object Machine, потому что нет никакой связи с представлением объектов при помощи векторов признаков.

Возможность представления объектов несколькими априори равновероятными функциями парного сравнения $S_i(\omega', \omega''), i = \overline{1, n}$ не влияет принципиально на критерий, кроме того, что количество вторичных признаков расширяется до nN:

$$\begin{cases} x_{il} = S_i(\omega_l, \omega) \text{для всех } \omega \in \Omega; \\ x_{il,j} = S_i(\omega_l, \omega_j) \text{для } j\text{-го объекта } \omega_j. \end{cases} \quad (1)$$

Прямое обобщение критерия (1) дает выпуклый критерий обучения, который отличается только количеством переменных:

$$\begin{cases} \sum\limits_{i=1}^{n} \sum\limits_{l=1}^{N} \left[\beta a_{il}^2 + \mu|a_{il}|\right] + \sum_{j=1}^{N} \delta_j \to \min\limits_{a_{il},b,\delta_j}; \\ y_j(\sum\limits_{i=1}^{n} \sum\limits_{l=1}^{N} a_{il} x_{il,j} + b) \geqslant 1 - \delta_j, j = \overline{1, N}; \\ \delta_j \geqslant 0, j = \overline{1, N}. \end{cases}$$

Двойственная запись критерия и разбиение множества вторичных признаков

Теорема 1. *Оптимальная гиперплоскость* $(a_{il}, i = \overline{1,n}, l = \overline{1,N}, b)$ *определяется равенствами*

$$\begin{cases} \hat{a}_{il} = \dfrac{1}{2\beta}(s_{il} + \mu), s_{il} < -\mu; \\ \hat{a}_{il} = 0, -\mu \leqslant s_{il} \leqslant \mu; \\ \hat{a}_{il} = \dfrac{1}{2\beta}(s_{il} - \mu), s_{il} > \mu; \end{cases} \tag{2}$$

$$\hat{b} = -\frac{\sum_{j:0<\hat{\lambda}_j<1} \hat{\lambda}_j \sum_{l,i:\hat{\lambda}_l>0,\hat{a}_{il}\neq 0} \hat{a}_{il} x_{il,j} + \sum_{j:\hat{\lambda}_j=1} y_j}{\sum_{j:0<\hat{\lambda}_j<1} \hat{\lambda}_j}, \tag{3}$$

где

$$s_{il} = \sum_{j:\hat{\lambda}_j>0} y_j \hat{\lambda}_j x_{il,j},$$

$\{j : 0 < \hat{\lambda}_j < 1\}$ *и* $\{j : \hat{\lambda}_j = 1\}$ *– подмножества ненулевых решений* $\{j : \hat{\lambda}_j > 0\}$ *двойственной задачи выпуклого программирования:*

$$\begin{cases} W(\lambda_1, \ldots, \lambda_N|\mu) = \sum_{j=1}^N \lambda_j - \dfrac{1}{4\beta} \sum_{i=1}^n \sum_{l=1}^N \left\{ \min \begin{bmatrix} \mu + \sum\limits_{j=1}^N y_j \lambda_j x_{il,j} \\ 0 \\ \mu - \sum\limits_{j=1}^N y_j \lambda_j x_{il,j} \end{bmatrix} \right\}^2 \to \max\limits_{\lambda_1,\ldots,\lambda_N}; \\ \sum_{j=1}^N y_j \lambda_j = 0; \\ 0 \leqslant \lambda_j \leqslant 1, \ j = \overline{1,N}. \end{cases} \tag{4}$$

Доказательство. Доказательство основано на анализе седловой точки Лагранжиана исходной задачи оптимизации (2), в которой числа $(\lambda_1, \ldots, \lambda_N)$ являются множителями Лагранжа при неравенствах $y_j(\sum_{i=1}^n \sum_{l=1}^N a_{il} x_{il,j} + b) - 1 + \delta_j \geqslant 0$. ■

Итерационный алгоритм решения двойственной задачи

Утверждение 1. *Функция* $W(\lambda)$*в задаче*(4) *является выпуклой.*

Доказательство. В силу того, что $W(\lambda)$ является суммой двух выпуклых функций: линейной по λ и кусочно-квадратичной по λ, то отсюда с очевидностью следует, что $W(\lambda)$ - выпуклая функция. ■

Избавимся от условия-равенства в задаче (4), чтобы она полностью соответствовала постановке вида (9). Рассмотрим новые переменные η и матрицу перехода Z, что $\lambda = \mathbf{Z}\eta$. Тогда матрица Z выглядит так:

$$Z = \begin{pmatrix} -\dfrac{y_2}{y_1} & -\dfrac{y_3}{y_1} & \cdots & -\dfrac{y_N}{y_1} \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{5}$$

Обозначим $\tilde{\mathbf{x}}_{il} = \{y_1 x_{il,1}, ..., y_N x_{il,N}\}^T$.

Утверждение 2. *Двойственная задача (4), записанная в переменных* η*, полностью соответствует постановке (9).*

Доказательство. Запишем задачу (4) в переменных η:

$$\begin{cases} W(\eta_1,\ldots,\eta_N|\mu) = \mathbf{1}^{\mathrm{T}} Z\eta - \frac{1}{4\beta}\sum_{i=1}^{n}\sum_{l=1}^{N}\left\{\min\begin{bmatrix}\mu+\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta\\ 0\\ \mu-\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta\end{bmatrix}\right\}^2 \to \max\limits_{\eta_1,\ldots,\eta_{N-1}}, \\ 0\leqslant (Z\eta)_j\leqslant 1,\ j=\overline{1,N} \end{cases} \tag{6}$$

Поскольку преобразование переменных линейно, то функция $W(\eta)$ остается выпуклой в силу теоремы 1. Ограничения-неравенства линейны, следовательно, вогнуты. ■

Утверждение 3. *Градиент и гессиан целевой функции $W(\eta)$ двойственной задачи 6 определяются выражениями:*

$$\nabla W(\eta) = \mathbf{Z}^{\mathrm{T}}\mathbf{1} - \frac{1}{2\beta}\sum_{i,l:|\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta|>\mu}\left[\mathbf{Z}^{\mathrm{T}}\tilde{\mathbf{x}}_{il}\left(\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta+\mu\,\mathrm{sign}(\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta)\right)\right] \tag{7}$$

$$\nabla^2 W(\eta) = \frac{1}{2\beta}\mathbf{Z}^{\mathrm{T}}\left[\sum_{i,l:|\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta|>\mu}\tilde{\mathbf{x}}_{il}\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\right]\mathbf{Z} \tag{8}$$

Доказательство. Заметим, что справедливо следующее равенство:

$$\min\begin{bmatrix}\mu+\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta\\ 0\\ \mu-\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta\end{bmatrix} = \min\begin{bmatrix}\mu-|\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta|\\ 0\end{bmatrix} = \min\begin{bmatrix}\mu-\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta\,\mathrm{sign}(\tilde{\mathbf{x}}_{il}^{\mathrm{T}}Z\eta)\\ 0\end{bmatrix}.$$

Подставляя это равенство в (6) и выполняя элементарные операции взятия первой и второй производных для функции многих переменных $W(\lambda)$, получаем требуемый результат. ■

Утверждения (2) и (3) дают нам возможность применить алгоритм [7] для решения двойственной задачи (6). После того, как найдены переменные η, мы выполняем обратное линейное преобразование к переменным λ. Решение исходной задачи получаем согласно теореме (1).

Алгоритм решения выпуклых задач селективного комбинирования потенциальных функций

Общая идея метода внутренней точки освещена в литературе [7], [10]. В данной работе будет использоваться модификация [7]. Исходная постановка задачи для этого метода такая:

$$\begin{cases}\min f(x);\\ c(x)\geqslant 0,\end{cases} \tag{9}$$

где $f(x) : R^n \to R$ — выпуклая функция, а $c(x) \geqslant 0$ означает, что каждая компонента $c_i : R^n \to R$ $(1 \leqslant i \leqslant m)$ неотрицательна и все c_i вогнуты. Штрафы выражаются логарифмическими функциями, а решение задачи безусловной минимизации ищется при помощи метода Ньютона.

Применяемый метод сначала сводит ту задачу, которую он решает, к двойственной, а потом оперирует в терминах этой двойственной задачи. Поэтому мы фактически будем решать задачу, двойственную к двойственной исходной задачи. После этого выписываются условия Каруша-Куна-Такера для новой двойственной задачи и полученная система решается итеративно при помощи метода Ньютона. Поскольку ограничений будет $2N$, то и двойственных переменных будет $2N$, а активных переменных останется $N-1$, потому что мы избавились от ограничения-равенства. Итого, переменных в двойственной задаче будет $3N-1$. Введем следующие обозначения:

— алгоритм оперирует значением функции $f(x)$, ее градиентом $\nabla f(x)$, гессианом $\partial^2 f(x)/(\partial x_i \partial x_j)$, значением ограничений $C(x)$ и их якобианом $\partial c_i(x)/\partial x_j$ в точке x;

— $N-1$ исходных переменных будем обозначать как и раньше: η.

— $2N$ двойственных переменных обозначим π;

— Ограничения-неравенства из (6) обозначим как вектор-столбец $\mathbf{c}(\eta) = \{c_1(\eta), ..., c_{2N}(\eta)\}^{\mathrm{T}}$, $\tilde{\mathbf{c}} = \{\underbrace{0, ..., 0}_{N}, \underbrace{1, ..., 1}_{N}\}^{\mathrm{T}}, \tilde{Z} = \left[\begin{smallmatrix} Z \\ -Z \end{smallmatrix}\right]$. Тогда

$$\mathbf{c}(\eta) = \tilde{Z}\eta + \tilde{\mathbf{c}}; \tag{10}$$

— $p = [\eta, \pi], \psi(p) = W(\eta) - \pi^T \mathbf{c}(\eta) - \mu \sum_{i=1}^{2N} \log(\pi_{(i)} c^2_{(i)}(\eta))$.

Теорема 2. *Алгоритм 1, построенный на базе модификации метода внутренней точки [7], корректно решает задачу (eqrefeq:svmmultimodalDoublyRegularizedSvm), т. е.:*

— *начальное приближение η^0 является внутренней точкой области, ограниченной условиями $c_{(l)}(\eta) > 0, l = \overline{1, 2N}$;*

— *градиент $\partial W(\eta)/(\partial \eta_i)$ и гессиан $\partial^2 W(\eta)/(\partial \eta_i \partial \eta_j)$ исходной функции $W(\eta)$ посчитаны корректно;*

— *якобиан ограничений $\partial c_{(i)}(\eta)/\partial \eta_j$ посчитан корректно;*

— *матрица вторых производных для каждого из ограничений $c_{(i)}(\eta)$ есть нулевая матрица.*

Доказательство. Напомним обозначения: условия-ограничения вычисляются согласно (10). Оценим элементы вектора $\mathbf{Z}\eta = (\lambda_1, ..., \lambda_N)^{\mathrm{T}}$:

$$\begin{cases} \lambda_1 = \sum_{i=1}^{N-1} -\frac{y_{i+1}}{y_1}\eta_i; \\ \lambda_i = \eta_{i-1}, i = 2, N. \end{cases} \tag{12}$$

Ясно, что для $i = 2, ..., N$ справедливо $0 < \lambda_i < 1$ в силу формулы (11). В силу предположения о том, что $n_{+1} \geqslant n_{-1}$ и $y_1 = -1$, имеем $\sum_{i=1}^{N-1} -(y_{i+1}/y_1)\eta_i = n_{+1}(3/(4n_{+1})) - (N - n_{+1})/(2(N - n_{+1})) = 1/4$, т.е. $0 < \lambda_1 < 1$. Совершенно аналогично доказывается, что $\mathbf{0} < -\mathbf{Z}\eta + \tilde{\mathbf{c}} < \mathbf{1}$. Остается заметить, что $\pi^0 = (1, ..., 1)^{\mathrm{T}} > \mathbf{0}^{\mathrm{T}}$, т. е. начальное приближение является внутренней точкой области, ограниченной условиями $c_{(l)}(\eta) > 0, l = \overline{1, 2N}$. Корректность градиента $\partial W(\eta)/\partial \eta_i$ и гессиана $\partial^2 W(\eta)/(\partial \eta_i \partial \eta_j)$ исходной функции $W(\eta)$ гарантируется **Утверждением** 3. Якобиан ограничений, поданных на вход, очевидным образом согласно (10) определяется как $\tilde{\mathbf{Z}}$. Матрица вторых производных $\partial^2 c_{(i)}(\eta)/(\partial \eta_i \partial \eta_j)$, очевидно, нулевая, так как все ограничения в рассматриваемом случае линейны. ■

Анализ сложности алгоритма решения выпуклых задач селективного комбинирования потенциальных функций

Предположим, что у нас в распоряжении имеется 1-процессорная вычислительная система. Это означает, что в один момент времени такая машина может выполнить одну элементарную арифметическую операций, т. е. сложение, вычитание, деление, умножение.

Пусть нам дана обучающая совокупность $\omega_j, j = \overline{1, N}$. Пусть нам заданы способы сравнения объектов $S_i(\omega', \omega''), i = \overline{1, n}$.

Каждый j-ый объект характеризуется вектором $\mathbf{x}_{il,j}, i = \overline{1, n}, l = \overline{1, N}$, т. е. nN числами. В итоге вся обучающая совокупность занимает память по порядку величины $O(nN^2)$. Перейдем к анализу времени работы алгоритма.

Алгоритм 1 Итерационный алгоритм решения задачи распознавания образов (2)

Вход: — обучающая выборка $x_{il,j}, i = \overline{1,n}, l = \overline{1,N}, j = \overline{1,N}, y_j = \pm 1$;

— $n_{+1} = |(j : y_j = 1)| \geqslant n_{-1} = |(j : y_j = -1)|, y_1 = -1$;

— регуляризующие коэффициенты $\beta > 0$, $\mu > 0$;

— параметры линейного поиска $0 < \nu < 1$, $0 < \alpha < 1$;

— параметры останова $\varepsilon_1 > 0, \varepsilon_2 > 0$.

Выход: $\hat{\mathbf{a}}_{il}, b$

1: начальное приближение $p^0 = [\eta^0, \pi^0]$:

$$\eta_i^0 = \begin{cases} 3/(4n_1), y_{i+1} = 1, \\ 1/(2(N - n_{+1})), y_{i+1} = -1 \end{cases} \tag{11}$$

$$\pi^0 = (1, \dots, 1)$$

2: **повторять**

3: $\tilde{\mathbf{X}}(\eta^s) = \mathbf{Z}^T \left[\sum_{i,l:|\tilde{\mathbf{x}}_{il}^T \mathbf{Z}\eta^s|>\mu} \tilde{\mathbf{x}}_{il}\tilde{\mathbf{x}}_{il}^T\right] \mathbf{Z}$

4: $\nabla W(\eta^s) = -\mathbf{1}^T\mathbf{Z} + \tilde{\mathbf{X}}(\eta^s)\eta^s$

5: $\nabla^2 W(\eta^s) = \tilde{\mathbf{X}}(\eta^s)$

6: $C(\eta^{\mathbf{s}}) = diag(\tilde{Z}\eta + \tilde{\mathbf{c}})$, $\Theta = diag(\pi_1^s, \dots, \pi_{2N}^s)$

7: $\rho^s = max(0, \frac{\|\eta^{s\,T}\tilde{Z}^T\pi^s\|}{2N})$

8: направление спуска $h^{s+1} = \left[d\eta^{s+1}, d\pi^{s+1}\right]$ вычисляем как решение системы линейный уравнений:

$$\begin{pmatrix} \nabla^2 W(\eta^s) & -\tilde{Z}^T \\ \Theta\tilde{Z} & C(\eta^s) \end{pmatrix} \begin{pmatrix} d\eta^{s+1} \\ d\pi^{s+1} \end{pmatrix} = \begin{pmatrix} -\nabla W(\eta^s) + \tilde{Z}\pi \\ \rho^s\mathbf{1}^T - C(\eta^s)\pi \end{pmatrix}$$

9: величину шага γ выбираем на основании алгоритма линейного поиска[10]: $\gamma_l = \nu^l\alpha$. Как только находится такое l, что $\psi(p^s + \gamma_l h^{s+1}) \leqslant \psi(p^s) + \omega\gamma_l\nabla\psi_\mu(p^s)^T h^{s+1}$, линейный поиск останавливается.

10: $\left[\eta^{s+1}, \pi^{s+1}\right] = [\eta^s, \pi^s] + \gamma_l h^{s+1} \in R^{3N-1}$.

11: **пока** $|W(\eta^s) - W(\eta^{s+1})| \geqslant \varepsilon_1$ **and** $|\nabla W(\eta^{s+1}) - \tilde{Z}^T\pi^{s+1}| \geqslant \varepsilon_2$ **and** $|C(\eta^{s+1})\pi^{s+1} - \rho^{s+1}\mathbf{1}^T| \geqslant \varepsilon_2$

12: $\lambda = \mathbf{Z}\eta^{\mathbf{s}}$

13: вычислить $\hat{\mathbf{a}}_{il}, b$ по формулам (2)

Утверждение 4. *Операция суммирования* $\sum_{i,l:|\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta^s|>\mu} \tilde{\mathbf{x}}_{il}\tilde{\mathbf{x}}_{il}^{\mathrm{T}}$ *эквивалента операции умножения матриц* $\mathbf{X}^*\mathbf{X}^{*\mathrm{T}}$, *где* $\mathbf{X}^* = (\mathbf{x}_{i_1,l_1}, ..., \mathbf{x}_{i_m,l_m})$, $\{(i_k, l_k) : |\tilde{\mathbf{x}}_{i_k l_k}^T \mathbf{Z}\eta^s| > \mu\}$.

Доказательство. Пусть $\mathbf{X}^{**} = \sum_{i,l:|\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta^s|>\mu} \tilde{\mathbf{x}}_{il}\tilde{\mathbf{x}}_{il}^{\mathrm{T}}$. Тогда $\mathbf{X}_{k,m}^{**} = \sum_{i,l=i_1,l_1}^{i_m,l_m} \mathbf{x}_{i,l,k}\mathbf{x}_{i,l,m}$, а это означает стандартную операцию умножения двух матриц, одна из которых состоит из векторов $\mathbf{x}_{i_1,l_1}, ..., \mathbf{x}_{i_m,l_m}$, а другая — ее транспонированная. ■

Сформулируем в терминах количества элементарных операций сложность каждого шага алгоритма 1:

1. Сначала необходимо сформировать матрицу объекты $\times$ признаки. Для этого нужно вычислить nN^2 чисел. Сложность, естественно, зависит от того, насколько долго вычисляются сами функции сравнения. Сложность этой операции $N^2\sum_{i=1}^n O(S_i)$, где $O(S_i)$ обозначает сложность вычисления i-ой функции парного сравнения.

2. Сложность вычисления $\tilde{\mathbf{x}}_{il,j}$ составляет $O(nN^2)$, так как сводится к выполнению $n \times N \times N$ операций умножения.

3. Начальное приближение, согласно шагу 1, генерирует $N-1$ число, используя каждый раз не более трех элементарных операций, так что сложность тут будет $O(N)$. Память такая же: $O(N)$.

4. В силу вида матрицы $\mathbf{Z}$ (5) операция умножения слева вектора на эту матрицу $\mathbf{x}^{\mathrm{T}}\mathbf{Z}$ эквивалентна сумме двух векторов: $(x_2, ..., x_N)^{\mathrm{T}} + x_1(-y_2/y_1, ..., -y_N/y_1)$, т.е. сложность получается $O(N)$.

5. В силу вида матрицы $\mathbf{Z}$ (5) операция умножения справа вектора на эту матрицу $\mathbf{Zx}$ эквивалентна умножению этого вектора на первый ряд этой матрицы и копированию элементов этого вектора в результирующий вектор, т. е. сложность получается $O(N)$.

6. В силу вида матрицы $\mathbf{Z}$ (5) операция умножения этой матрицы справа и слева на любую матрицу $\mathbf{Z}'$ сводится к умножению этой матрицы на первый ряд матрицы $\mathbf{Z}$ и копированию остальных элементов этой матрицы, т. е. сложность получается $O(N^2)$.

7. Анализ формулы шага 3 показывает, что на каждой итерации главного цикла необходимо знать для каждой пары i, l значение произведения $\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}$, которое никак не зависит от номера итерации и определяется только обучающей совокупностью. Поэтому можно один раз заранее вычислить все $\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}$ для всех i, l. Это занимает, с учетом вида матрицы $\mathbf{Z}$, $O(nN^2)$.

8. На шаге 3 необходимо для всех i, l вычислить произведение $\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta^s$, заранее предвычисленных векторов у нас nN, тогда в данном случае сложность будет $O(nN^2)$, так как каждый раз сложность умножения равняется $O(N)$.

9. Далее на шаге 3 необходимо вычислить сумму $\sum_{i,l:|\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta^s|>\mu} \tilde{\mathbf{x}}_{il}\tilde{\mathbf{x}}_{il}^{\mathrm{T}}$. Согласно **Утверждению** 4, это эквивалентно операции умножения матриц $nN \times k \times nN$, где $k = |i, l : |\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}\eta^s| > \mu|$, т. е. $k = 1, ..., nN$. То есть в худшем случае это $O(n^3N^3)$.

10. Шаг 4 ведет к сумме двух векторов, один из которых вычисляется за линейное время $O(N)$, а второй за квадратичное время $O(N^2)$.

11. Шаги 5–7 занимают линейное количество операций, т. е. $O(N)$

12. Решение системы линейных уравнений 8 делается в два этапа: сначала решается относительно $d\eta^{s+1}$ система с матрицей $(N-1) \times (N-1)$. Эту систему можно решить методом Гаусса за $O(N^3)$. На следующем этапе определяется уже $d\pi^{s+1}$, это делается за $O(N^2)$ операций.

13. Линейный поиск на шаге 9 выполняется за неопределенное количество операций, так как неизвестно заранее, на каком номере l функция $\psi(p^s)$ уменьшит свое значение. Допустим, что прошло l шагов. На каждом шаге требуется вычислить предыдущее значение $\psi(p^s)$, вычисленное на прошлой итерации, новое значение $\psi(p^s + \gamma_l h^{s+1})$ вычисляется, согласно формуле $\psi(p)$, за $O(nN)$, так как основные затраты тут пойдут на вычисление значения функции $W(\eta)$, остальные два слагаемых вычисляются за линейное время $O(N)$. Итого получается, что сложность шага 9 составляет $O(lnN)$.

14. Остальные шаги алгоритма 10–13 вычисляются за время $O(nN^2)$ за счет вычисления по формулам (2).

Количество операций до цикла 2 состоит из вычисления матрицы объекты × признаки, генерации начального приближения и предвычисления $\tilde{\mathbf{x}}_{il}^{\mathrm{T}}\mathbf{Z}$ для всех i, l: $O(nN^2) + O(N) + O(nN^2) = O(nN^2)$. Если предположить, что цикл 2 выполнялся m раз, а внутренний цикл линейного поиска выполнялся $l_1, ..., l_m$ раз соответственно, тогда сложность цикла 2 получается такой: $\sum_{i=1}^{m} \left[O(nN^2) + O(n^3N^3) + O(N^3) + O(l_i nN)\right] = O(mn^3N^3) + O(nN\sum_{i=1}^{m} l_i)$. Количество операций после алгоритма связано лишь с вычислением результата и равно $O(nN^2)$. В итоге общая сложность алгоритма, в предположении, что количество итераций равно m, а на каждой итерации выполнялся линейный поиск сложностью $l_i, i = 1, ..., m$,

$$\text{Complexity } = O(nN^2) + O(mn^3N^3) + O(nN\sum_{i=1}^{m} l_i) + O(nN^2) = \\ = O(mn^3N^3) + O(nN\sum_{i=1}^{m} l_i) \quad (13)$$

Соответственно, получается, что цикл 2 вносит максимальный вклад в количество операций алгоритма, так как на каждой итерации нужно выполнять умножение матрицы $nN \times k$ на саму себя транспонированную и решать систему линейных уравнений $N \times N$. Эти операции имеет смысл распараллелить. То, что алгоритм вынужден держать все nN^2 чисел, — особенность данного алгоритма и одновременно его самый главный недостаток, так как это существенно ограничивает объемы данных (количество объектов и количество функций парного сравнения), на которых можно было бы использовать данный алгоритм.

Модульная реализация распараллеливания

Согласно анализу сложности алгоритмов в разделах 8 наиболее трудоемкими операциями являются операции умножения двух матриц размеров $N \times N$ и решения системы линейных уравнений размеров $N \times N$. Именно эти две операции вносят максимальный вклад в итоговую сложность каждого из алгоритмов 1. Поэтому имеет смысл начать оптимизацию именно с этих операций. Согласно экспериментам, указанные выше алгоритмы тратят на умножение матриц и решение системы линейных уравнений 80% времени своей работы. Естественно, речь идет о реализации умножения матриц и решении системы линейных уравнений на однопроцессорной машине.

Умножение матриц

Обоснуем, почему распараллеливание умножения матриц лучше, чем существующие алгоритмы умножения. Кроме стандартного алгоритма умножения матриц $N \times N$ за время $O(N^3)$ известны также алгоритмы, работающие быстрее:

1. алгоритм Штрассена [11] умножает две матрицы размеров $N \times N$ за $O(N^{2.807})$;

2. алгоритм Копперсмита-Винограда [12], имеющий сложность умножения $O(N^{2.3755})$;

3. алгоритм Вильямс [13], немного обгоняющий алгоритм Копперсмита–Винограда, и имеющий сложность $O(N^{2.3727})$.

Представим, что у нас в распоряжении очень простая вычислительная система — видеокарта NVidia Geforce 310M, имеющая 16 GPU. Оценим, каких размеров должна быть матрица, чтобы алгоритм Штрассена дал выигрыш по сравнению с распараллеливанием на 16 ядер. Ответ прост: $N \geqslant 16^{1/(3-2.807)} \approx 1000000$. В оперативной памяти такая матрица занимала бы $8 \cdot 1000000^2/1024^2 \approx 8$ терабайт, что на текущий момент технически невозможно принципиально. Ясно, что при переходе к более мощным вычислительным системам выигрыш в скорости будет только увеличиваться при неизменных размерах матрицы. Что касается алгоритма Вильямс [13] и алгоритма Копперсмита–Винограда [12], то эти алгоритмы имеют очень большую константу

Алгоритм 2 Общая схема точных алгоритмов решения системы линейных уравнений $N \times N$

Вход: матрица A размеров $N \times N$, вектор b размеров $N \times 1$.
Выход: вектор $\mathbf{x}$ размеров $N \times 1 : A\mathbf{x} = \mathbf{b}$.
1: **для** $i = 1$ to N
;
2: обновить матрицу A, выполнив, в общем случае, для каждого метода(Гаусса-Жордана, разложение Холецкого и т.д.) $O(N^2)$ операций;
3: вычислить решение $\mathbf{x}$ на базе матрицы A, полученной после N итераций. Здесь сложность для каждого из методов не превосходит $O(N^2)$.

сложности, поэтому выигрывают у современных алгоритмов на матрицах, размеры которых превосходят современные технические возможности компьютеров. Поэтому в рамках данной работы параллельное умножение матриц на видеокарте — лучшее решение из всех возможных, исходя из количества арифметических операций. Если рассмотреть детали технической реализации, то необходимо учесть тот факт, что матрицу, загруженную в оперативную память компьютера, необходимо 1 раз скопировать в оперативную память видеокарты, что по порядку величины занимает $O(N^2)$ операций. Учитывая современные скорости передачи информации CPU $\rightarrow$ GPU и GPU $\rightarrow$ CPU [14], даже для матрицы 10000×10000 подобная операция копирования займет меньше секунды, что в разы меньше времени, которое тратится на операции умножения матриц на GPU. В рамках данной работы выполнено две реализации алгоритмов умножения матриц: для однопроцессорной машины(CPU) и для видеокарты(GPU). В качестве фреймворка, на базе которого выполнялась реализация алгоритмов, был взят OpenCL [15]. Это удобный современный фреймворк, сочетающий в себе возможность гибкой разработки на C++ и реализованный практически подо все современные видеокарты, в частности карты NVidia.

Решение системы линейных уравнений

Существующие способы решения системы линейных уравнений, матрицы которых не являются разреженными, делятся на точные(прямые) методы [16] и итерационные методы [16].

Точные методы решения системы линейных уравнений

Популярные точные методы для решения системы линейных уравнений $N \times N$:

1. метод Гаусса-Жордана [17], количество операций: $2N^3$;

2. решение системы линейных уравнений, используя разложение Холецкого [18], количество операций: $N^3/3$;

3. другие методы решения системы линейных уравнений через разложение матрицы в произведение более простых матриц [19]: LU-разложение, QR-разложение и т. д. Сложность таких методов по порядку величины составляет $O(N^3)$, однако эти методы работают дольше, чем через разложение Холецкого.

Каждый из перечисленных методов базируется на следующей схеме.

Согласно приведенному алгоритму 2 получается, что единственный вариант параллелизации заключается в распараллеливании каждой из N итераций. Здесь может быть два подхода. Первый подход заключается в том, чтобы на каждой из N итераций отправлять на видеокарту текущую матрицу, а потом забирать оттуда вычисленный результат. Такой подход имеет существенный недостаток: обновленную матрицу $N \times N$ придется копировать N раз во внутреннюю память карты. Уже при размерах $N = 10000$, согласно [14], на решение системы линейных уравнений потребуется ≈ 35 мин, что существенно дольше, чем просто решать систему линейных уравнений

через разложение Холецкого на однопроцессорной машине. Второй подход заключается в том, чтобы загрузить матрицу $N \times N$ на видеокарту целиком и проводить все вычисления на ней, однако даже в такой схеме не избежать N итераций, лежащих в основе данной схемы, что делает неэффективным подобный подход. Подтверждением являются попытки реализовать алгоритм разложения Холецкого на GPU [20], однако результаты пока слабые: алгоритм на видеокарте в 4 раза медленнее существующих аналогов, реализованных для однопроцессорной машины, причем потеря времени как раз уходит на копирование данных и прочие технические особенности существующих механизмов, обеспечивающих взаимодействие CPU и GPU. Итак, точные алгоритмы имеет смысл запускать на однопроцессорной машине, по крайней мере, в рамках технических ограничений, принятых в данной работе. Поэтому среди точных алгоритмов имеет смысл реализовать наиболее быстрый алгоритм решения системы линейных уравнений на базе разложения Холецкого [18].

Итерационные методы решения системы линейных уравнений

Наиболее популярные итерационные методы решения систем линейных уравнений:

1. метод Якоби [21];
2. метод Гаусса–Зейделя [21];
3. метод Релаксации [22];
4. метод сопряженных градиентов [10].

Для того чтобы первые три метода [21, 22] сходились, необходимо, чтобы спектральный радиус некоторой матрицы, зависящей от алгоритма, удовлетворял условию $\rho(\mathbf{B}) = \max\limits_{\lambda} |\lambda(\mathbf{B})| < 1$. Для каждого из алгоритмов экспериментально получен следующий факт: на задачах, которые рассматриваются в данной работе, спектральный радиус сходимости соответствующих матриц оказывается > 1, что приводит к тому, что итерационные алгоритмы не сходятся.

Метод сопряженных градиентов оказалось неэффективно использовать в прикладных задачах, рассмотренных в данной работе, поскольку этот метод почти всегда выполняет количество итераций, равное размерности матрицы (рис. 1). Поэтому в данной работе мы решили отказаться

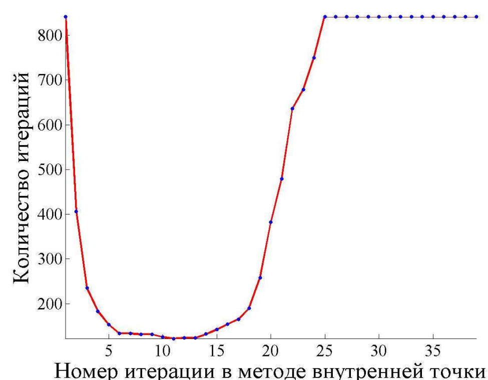

Рис. 1: Зависимость количества итераций, выполняемых методом сопряженных градиентов при решении системы линейных уравнений 842×842 от номера итерации в методе внутренней точки. Видно, что чем выше номер итерации, тем хуже работает метод, в середине пути выходя на плато и делая сложность алгоритма решения системы линейных уравнений $O(N^3)$

от итерационных методов решения системы линейных уравнений.

Алгоритм 3 Алгоритм умножения двух матриц $n \times N$ и $N \times m$

Вход: матрица A размеров $n \times N$, Матрица B размеров $N \times m$.
Выход: матрица $\mathbf{C}$ размеров $n \times m : \mathbf{C} = \mathbf{AB}$.

1: **для** $i = 1$ to n
2: **для** $j = 1$ to m
3: выбрать, какое из двух ядер свободно
4: обнулить текущую сумму для выбранного ядра: $S = 0$
5: запустить на потоке на выбранном ядре вычисление суммы:
6: **для** $k = 1$ to N
7: $S = S + A[i][k]A[k][j]$
8: записать результат $C[i][j] = S$ в соответствующую ячейку результирующей матрицы
9: освободить занятый поток
10: вернуть результат: заполненная матрица C

Реализация алгоритма умножения матриц

Современные методы реализации алгоритмов [23] легко позволяют вынести такие операции в отдельные модули, чтобы потом один и тот же алгоритм можно было запустить на ноутбуке, на десктопе и на вычислительном кластере, указав каждый раз разную реализацию операций умножения матриц и решения системы линейных уравнений. Для достижения гибкости подобного уровня в данной работе все алгоритмы будут реализованы на языке C++, matlab. Данные языки, во-первых, поддерживаются практически на всех современных вычислительных системах, а во-вторых, позволяют применять стандартные паттерны проектирования [23], позволяющие переиспользовать код и легко подменять версии операций умножения и решения систем линейных уравнений в зависимости от вычислительной системы. Следует отметить важную разницу между центральным процессором портативного компьютера (CPU) и видеокартой (GPU). Центральный процессор портативного компьютера оснащен несколькими ядрами (обычно 2–4), на которых можно параллельно исполнять несколько вычислительных операций. Видеокарта является своеобразным миникластером — она устроена таким образом, что имеет на борту большое количество микропроцессоров (GPU), которых обычно сотни(на мощных видеокартах), что дает существенное преимущество в вычислительных мощностях GPU по сравнению с CPU.

Параллельное умножение матриц на CPU

Эксперименты проводились на машине, оснащенной 2-ядерным процессором Intel Core i7. Естественно, даже на однопроцессорной машине можно выиграть в скорости умножения вдвое, если распараллелить умножение на два ядра. Алгоритм 3 иллюстрирует механизм использования нескольких ядер при работе с CPU.

Параллельное умножение матриц на видеокарте NVidia GeForce 310M

Особенность реализации OpenCL под видеокарты Nvidia заключается в том, что максимальная эффективность работы видеокарты достигается тогда, когда размеры матриц кратны 32 [24]. Естественно, что в реальной жизни размеры матриц, которые нужно перемножать, вовсе необязательно кратны 32. С другой стороны, так как память на видеокарте ограничена, то очень большие матрицы там умножать тоже нельзя. Поэтому обе исходных матрицы, которые требуется умножить, необходимо разбить на блоки так, чтобы размер каждого блока был максимальным для данной видеокарты и кратен 32. Естественно, что останутся такие блоки, один из размеров которых будет $0 < s < 32$. В силу малого размера такого блока, можно легко задействовать стандартный CPU для перемножения по алгоритму 3. Согласно спецификации для видеокарты NVidia GeForce 310M необходимо передавать матрицы размеров не более 1024×1024.

Из курса линейной алгебры [17] известно следующее

Алгоритм 4 Алгоритм умножения двух матриц $n \times N$ и $N \times m$, где $n \mid 1024, N \mid 1024, m \mid 1024$, на видеокарте

Вход: матрица A размеров $n \times N$, Матрица $\mathbf{B}$ размеров $N \times m$, $n \mid 1024, N \mid 1024, m \mid 1024$
Выход: матрица $\mathbf{C}$ размеров $n \times m : \mathbf{C} = \mathbf{AB}$
1: разбить исходные матрицы на блоки 1024×1024. Это можно сделать, так как размеры этих матриц кратны 1024.
2: $n = 1024\tilde{n}, N = 1024\tilde{N}, m = 1024\tilde{m}$.
3: **для** $k = 1$ to $\tilde{n}$
4: **для** $l = 1$ to $\tilde{m}$
5: инициализировать блок матрицы $\mathbf{C}[k][l]$ нулями
6: **для** $p = 1$ to $\tilde{N}$
7: вычислить результат умножения блока $\mathbf{R} = \mathbf{A}[k][p]\mathbf{B}[p][l]$ на GPU, используя стандартный алгоритм умножения матриц на GPU через shared memory [24]
8: $\mathbf{C}[k][l] = \mathbf{C}[k][l] + \mathbf{R}$
перейти к вычислению следующего блока матрицы $\mathbf{C}[k][l]$
9: вернуть результат: заполненная матрица $\mathbf{C}$

Утверждение 5. *Результатом умножения двух блочных матриц*

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & ... & \mathbf{A}_{1M} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{L1} & ... & \mathbf{A}_{LM} \end{pmatrix}$$

и

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & ... & \mathbf{B}_{1N} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{M1} & ... & \mathbf{B}_{MN} \end{pmatrix}$$

является матрица

$$\mathbf{C} = \begin{pmatrix} \sum_{k=1}^{M} \mathbf{A}_{1k}\mathbf{B}_{k1} & \cdots & \sum_{k=1}^{M} \mathbf{A}_{1k}\mathbf{B}_{kN} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{M} \mathbf{A}_{Lk}\mathbf{B}_{k1} & \cdots & \sum_{k=1}^{M} \mathbf{A}_{Lk}\mathbf{B}_{kN} \end{pmatrix}$$

при условии, что размеры всех блочных матриц $\mathbf{A}_{ik}, \mathbf{B}_{ik}$, а также матриц $\mathbf{A}, \mathbf{B}$, соответствуют правилам умножения матриц.

Алгоритм 4 описывает последовательность действий, необходимых для умножения двух матриц, когда их размеры кратны 1024. Алгоритм 5 обобщает ситуацию и описывает последовательность действий, необходимых для умножения матриц, размеры которых кратны 1024.

Оценим сложность алгоритма 4.

1. Шаги алгоритма 1–2 делаются за $O(1)$, так как это просто элементарные операции деления.
2. Шаг 5 заключается в том, что нужно выполнить инициализацию матрицы размерами $O(1024^2)$. Количество операций: 1024^2.
3. Шаг 7 заключается в том, чтобы скопировать два блока матрицы на устройство GPU - $O(1024^2)$, применить стандартный алгоритм умножения двух матриц [24] - $O(1024^3/k)$, где k – количество ядер на устройстве GPU, для видеокарты NVidia GeForce 310M это число равно 16.
4. Шаг 8 заключается в том, чтобы скопировать полученный результат умножения в финальную матрицу $\mathbf{C}$ — $O(1024^2)$.

Алгоритм 5 Алгоритм умножения двух матриц $n \times N$ и $N \times m$ на видеокарте

Вход: матрица A размеров $n \times N$, Матрица $\mathbf{B}$ размеров $N \times m$
Выход: матрица $\mathbf{C}$ размеров $n \times m : \mathbf{C} = \mathbf{AB}$
1: $n = 1024\tilde{n} + r_A, N = 1024\tilde{N} + r_{AB}, m = 1024\tilde{m} + r_B$
2: таким образом, мы получили разбиение двух исходных матриц на блоки:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}$$

размеры которых равны:

$$\begin{aligned} \mathbf{A}_{11} &= 1024\tilde{n} \times 1024\tilde{N}, \mathbf{A}_{12} = 1024\tilde{n} \times r_{AB}, \\ \mathbf{A}_{21} &= r_A \times 1024\tilde{N}, \mathbf{A}_{22} = r_A \times r_{AB} \\ \mathbf{B}_{11} &= 1024\tilde{N} \times 1024\tilde{m}, \mathbf{B}_{12} = 1024\tilde{m} \times r_B, \\ \mathbf{B}_{21} &= r_{AB} \times 1024\tilde{m}, \mathbf{B}_{22} = r_{AB} \times r_B \end{aligned}$$

3: произведение $A_{11}B_{11}$ вычисляем согласно алгоритму 4
4: все остальные произведения блоков $A_{ik} \times B_{lm}$ вычисляем по алгоритму 3
5: вернуть результат: заполненная матрица $\mathbf{C}$

Общее количество операций получается равным

$$\text{Complexity } = \tilde{n}\tilde{m}\left(1024^2 + \tilde{N}\left(1024^2 + 1024^3/16 + 1024^2\right)\right) = \\ = (66\tilde{N} + 1)nm = \left(\frac{33}{512} + \frac{1}{N}\right) nmN$$

Память, необходимая для работы данного алгоритма:

$$\frac{\left(8nN + 8Nm + 8 \times 2 \times 1024^2\right)}{1024^2} = 8\left[\tilde{N}\left(\tilde{n} + \tilde{m}\right) + 2\right] \text{ мегабайта.}$$

Если матрицы таковы, что $\tilde{n} = \tilde{m} = \tilde{N} = 10$, то памяти такому алгоритму потребуется 1602 мегабайта $\approx 1{,}5$ гигабайта.

Кроме умножения матриц $A_{11}B_{11}$ необходимо выполнить умножение остальных частей блочной системы:

$$\text{Complexity } = nmr_{AB} + nNr_B + nr_{AB}r_B + r_ANm + r_Ar_{AB}m + r_ANr_B + r_Ar_{AB}r_B \leqslant \\ \leqslant 1024\left(nm + nN + Nm\right) + 1024^2\left(n + m + N\right) + 1024^3 = O(nm + nN + Nm)$$

Общее количество операций в алгоритме умножения матриц 5 получается равным

$$\text{Complexity } \leqslant \left(\frac{33}{512} + \frac{1}{N}\right) nmN + 1024\left(nm + nN + Nm\right) + 1024^2\left(n + m + N\right) + 1024^3$$

Очевидно, что алгоритм 5 требует столько же памяти, сколько и алгоритм 4.

Еще раз обратим внимание на алгоритм 1. Согласно шагу 3 и **Утверждению** 4, единственная операция умножения матриц в этом алгоритме, которая никак не упрощается, заключается в вычислении произведения $\mathbf{X}^*\mathbf{X}^{*\mathrm{T}}$, где $\mathbf{X}^*$ — некоторая матрица размеров $nN \times k$, $k = 1, ..., nN$. В этом случае очевидно, как применить алгоритм 5 к вычислению этого произведения: достаточно взять матрицу $\mathbf{X}^*$, транспонировать ее и применить алгоритм 5. Неоптимальность такого подхода заключается в том, что придется выделить память для того, чтобы хранить транспонированную матрицу $\mathbf{X}^*$, а затем скопировать туда транспонированную матрицу $\mathbf{X}^*$. Очевидно, что этого можно избежать, правильно организовав обращение к одной и той же области памяти,

Алгоритм 6 Алгоритм умножения $\mathbf{AA}^{\mathrm{T}}$ на видеокарте

Вход: матрица A размеров $n \times N$
Выход: матрица $\mathbf{C}$ размеров $n \times n : \mathbf{C} = \mathbf{AA}^T$
1: $\mathbf{C}$ = результат работы алгоритма 5 на матрицах $\mathbf{A}, \mathbf{A}^T$

где хранится матрица $\mathbf{X}^*$. Приведем алгоритм, выполняющий умножение $\mathbf{XX}^{\mathrm{T}}$ по поданной на вход матрице $\mathbf{X}$.

По количеству операций алгоритм 6 ничем не отличается от алгоритма 5, поэтому общее количество операций у этого алгоритма:

$$Complexity \leqslant \left(\frac{33}{512} + \frac{1}{N}\right) nmN + 1024\,(nm + nN + Nm) + \\ + 1024^2\,(n + m + N) + 1024^3$$

Память, необходимая для работы алгоритма 6, вдвое меньше памяти, требуемой для работы алгоритма 5, потому что не нужно хранить вторую матрицу:

$$\text{Memory } = \left(8nN + 8 \times 2 \times 1024^2\right)/1024^2 = 8\left[\tilde{N}\tilde{n} + 2\right] \text{ мегабайта}$$

Ускорение, полученное за счет распараллеливания умножения матриц

Проанализируем выигрыш, даваемый алгоритмом умножения матриц на GPU 5, по сравнению с алгоритмом умножения матриц на CPU 3. Для проведения экспериментов использовался ноутбук Sony Vaio, оснащенный процессором Core i7 и имеющий 4096 мегабайт оперативной памяти. Видеокарта NVIDIA GeForce 310M, имеющая 512 мегабайт памяти и 16 независимых GPU. Для каждого из чисел $\{2^n, n = \overline{5, 11}\}$ 100 раз генерируется две случайных матрицы, измеряется время работы каждого из двух алгоритмов 5 и 3, а потом усредняется для каждого размера матрицы. Результаты экспериментов можно видеть на рис. 2. Видно, что выигрыш при пере-

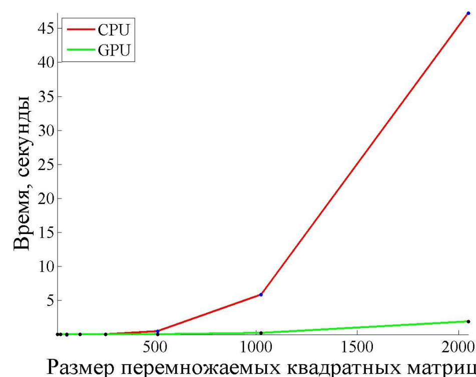

Рис. 2: Среднее время работы двух разных реализаций алгоритма умножения матриц — на видеокарте и на однопроцессорной машине — в зависимости от размера умножаемых матриц

множении матриц больших размеров доходит до 25 раз. Это связано не только с тем, что на видеокарте большее количество ядер, чем на центральном процессоре, но еще и с тем, что планировщик параллельных вычислений на видеокарте работает лучше и оптимальнее распределяет нагрузку, чем аналогичный планировщик на CPU.

Применение предложенного алгоритма на реальных задачах

Предложенный в работе алгоритм 1, использующий параллельный алгоритм умножения матриц 6 и решающий систему линейных уравнений на основе метода разложения Холецкого [18], успешно применен в задаче предсказания вторичной структуры белка в работе[25]. Математически строгие алгоритмы генерации и отбора признаков позволяют получать более высокое качество классификации. В работе [25] применен метод, основанный на RVM, к предсказанию вторичной структуры белка. Важной особенностью данного метода является то, что он позволяет автоматически отбирать наиболее информативные признаки из общего множества признаков. Средняя точность распознавания strand'ов оказалась равной примерно 75%, что находится на уровне уже существующих алгоритмов классификации. Особенно интересно то, что не было замечено переобучения, несмотря на то, что размерность векторов вторичных признаков в несколько раз выше размера обучающей совокупности. Разработанные алгоритмы позволяют существенно уменьшить количество аминокислотных фрагментов, необходимых для предсказания вторичной структуры белка с хорошей точностью. Предложенный в работе алгоритм 1 позволил ускорить этап обучения в задаче предсказания вторичной структуры белка [25] в 25 раз в сравнении с наивной реализацией данного алгоритма на однопроцессорной машине.

Заключение

В работе приведен численный алгоритм селективного комбинирования разнородных представлений объектов в задачах распознавания образов. Основное преимущество данного алгоритма перед существующими аналогами заключается в том, что он делает десятки итераций, каждая из которых хорошо распараллеливается. Данный алгоритм был реализован на видеокарте, что дало ускорение в 25 раз в сравнении с наивной реализацией такого же алгоритма на видеокарте. Недостатком данного алгоритма является то, что он требует квадратичного роста памяти в зависимости от количества объектов обучения, что существенно ограничивает применение данного алгоритма для обучения распознаванию на очень больших объемах данных(число объектов $N \gg 10^5$).

Литература

[1] V. Vapnik. *Statistical Learning Theory.* John-Wiley & Sons Inc., 1998.

[2] Татарчук А. И., Урлов Е. Н., and Моттль В. В. Метод опорных потенциальных функций в задаче селективного комбинирования разнородной информации при обучении распознаванию образов.

[3] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998. MSR-TR-98-14.

[4] Edgar Osuna, Robert Freund, and Federico Girosi. An improved training algorithm for support vector machines.

[5] Luca Zanni, Thomas Serafini, and Gaetano Zanghirati. Parallel software for training large scale support vector machines on multiprocessor systems. *Journal of Machine Learning Research.*

[6] Edward Y. Chang, Kaihua Zhu, Hao Wang, and Hongjie Bai. Psvm: Parallelizing support vector machines on distributed computers.

[7] Paul Armand, Jean-Charles Gilbert, and Sophie Jan-Jegou. A feasible bfgs interior point algorithm for solving strongly convex minimization problems.

[8] L. Wang, J. Zhu, and H. Zou. The doubly regularized support vector machine. *Statistica Sinicia*, 16:589–615, 2006.

[9] Robert P. W. Duin, Elżbieta Pekalska, and Dick de Ridder. Relational discriminant analysis. *Pattern Recogn. Lett.*, 20(11-13):1175–1181, Nov 1999.

[10] Сухарев А. Г., Тимохов А. В., and Федоров В. В. *Курс методов оптимизации. Учеб. пособие.* ФИЗМАТЛИТ, М., 2-е edition, 2005.

[11] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematatik*, 13(4):354 – 356, 1969.

[12] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990.

[13] Virginia Vassilevska Williams. Breaking the coppersmith-winograd barrier. unpublished manuscript, 2011.

[14] Gpu memory transfer.

[15] Opencl: The open standard for parallel programming of heterogeneous systems, 2008.

[16] Dennis S. Bernstein. *Matrix Mathematics: Theory, Facts, and Formulas (Second Edition)*. Princeton University Press, 41 William Street, Princeton, New Jersey 08540, 2009.

[17] Беклемишев Д. В. *Курс аналитической геометрии и линейной алгебры: Учеб. для вузов*. ФИЗМАТЛИТ, М., 11-е, испр. edition, 2006.

[18] Dariusz Dereniowski and Marek Kubale. Cholesky factorization of matrices in parallel and ranking of graphs. In *5th International Conference on Parallel Processing and Applied Mathematics. Lecture Notes on Computer Science 3019*, pages 985 —- 992. Springer-Verlag, 2004.

[19] Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. 2000.

[20] Opencl accelerated cholesky factorization, 2011.

[21] Davod Khojasteh Salkuyeh. Generalized jacobi and gauss-seidel methods for solving linear system of equations. *NUMERICAL MATHEMATICS, A Journal of Chinese Universities (English Series)*, 16(2):164–170, 2007.

[22] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, USA, 1994.

[24] Nvidia opencl programming guide version 2.3, 2009.

[25] Nikolay Razin, Dmitry Sungurov, Vadim Mottl, Ivan Torshin, Valentina Sulimova, and Oleg Seredin. Multi–modal relevance vector machines for protein secondary structure prediction. In *PRIB-2012 proceedings*. Springer, 2012.