# به نام خدا

امیرمهدی نامجو

تمرین پیاده سازی اول درس هوش مصنوعی – استاد گرامی: جناب آقای دکتر عبدی

دانشگاه صنعتی شریف – دانشکده مهندسی کامپیوتر

ابتدا در مورد کدهایی که همراه سؤال است توضیح می‌دهم. توجه کنید که در فایل README.txt هم در مورد هر فایل و کار آن و اجرای کلی تمامی فایله ا توضیحاتی کوتاه به انگلیسی نوشته‌ام. توضیح مفصل‌تر در مورد کل برنامه این فایل PDF قرار دارد.

در کدهای همراه سؤال چهار فایل پایتون با نام‌های HillClimbing.py و MultiThread_HillClimbing.py و SimulatedAnnealing.py و MultiCore_HillClimbing.py به همراه نمونه txt داده شده قرار دارد. هر سه این‌ها مستقل از دیگری هستند و با اجرا به راحتی عملیات را روی فایل new_example.txt انجام می‌دهند. برای اجرا این فایل باید دقیقاً کنار این سه فایل پایتون باشد. اگر بخواهید فایل با نام دیگری بدهید، باید تابع initialize تغییر داده شود. اگر تابع قرار نیست تغییر کند، باید نمونه‌های جدید با همین نام در کنار فایل‌ها باشند. در هنگام اجرا سه عدد از کاربر خواسته می‌شود. عدد اول lower bound است (در مثال صورت سؤال منفی ۱۰۰۰)، عدد بعدی higher bound است (در مثال صورت سؤال ۱۰۰۰) و عدد آخر step. در مورد step الگوریتم‌های من همگی با step متغیر هستند و صرفاً چون سؤال گفته بود باید ورودی گرفته شود ورودی گرفته‌ام ولی رسماً از step ها استفاده‌ای نکرده‌ام و خودم آن‌ها را تغییر می‌دهم. در مورد lower bound و higher bound هم توجه کنید که این اعداد صرفاً تعیین کننده بازه اولیه ساخت یک state رندوم هستند و به نوعی step های اولیه هم برابر

$$(high - low) * 0.2$$

هستند که به تدریج تغییر می‌کنند؛ اما در کد تضمین نکرده‌ام که جواب نهایی از این بازه‌ها خارج نشود و بعضاً حرکات اولیه که بازه گسترده‌ای دارند ممکن است باعث شوند که جواب در این بازه قرار نگیرد، اما همچنان جواب درستی باشد؛ یعنی در کل از این دو عدد صرفاً برای تعیین بازه اولیه و step های اولیه استفاده شده است.

درون کد HillClimbing.py دو تابع اساسی هست. یکی variable_step_hill_climber و دیگری random_restart_variable_step_hill_climber

من به طور پیش‌فرض از تابع دوم استفاده می‌کنم که با توجه به روش random_restart سؤال را حل می‌کند و جواب‌های خوبی به دست می‌آورد. تابع اول فقط روی یک نقطه الگوریتم را اجرا می‌کند و در نتیجه ممکن است در یک مینیمم محلی گیر کرده و جواب خوبی به دست ندهد. با این وجود اگر می خواهید با تابع اول سؤال را تست کنید، کافی است خط ۱۹۴ از حالت کامنت در بیاید و خط ۱۹۶ کامنت شود. این موضوع را با یک کامنت در خود کد هم مشخص کرده‌ام.

درون کد MultiThread_HillClimbing.py با استفاده از تابع اول کد قبل یعنی variable_step_hill_climber و استفاده از Thread های پایتون، برای ۴ نقطه رندوم الگوریتم اجرا می‌شود. (این عدد با تغییر متغیر NUMBER_OF_TESTS در کد قابل تغییر است). در ابتدا برای ۲۰۰ نقطه گرفته بودم که زمان قابل توجهی طول می‌کشید. با این وجود به دلیل این که Thread های پایتون عموماً روی یک هسته CPU فراخوانی می‌شوند، این کد به طور کلی کندتر از بقیه است و برای ۲۰۰ نقطه حدود ۵ دقیقه اجرای آن طول می‌کشد؛ اما برای ۴ نقطه سریع است. البته امکان سریع‌تر کردن بسیار زیاد این کد با استفاده از کتابخانه multiprocessing که امکان استفاده از چند هسته را فراهم می‌کند وجود داشت و در کد بخش MultiCore_HillClimbing.py که در لحظات نزدیک ددلاین نوشته‌ام از آن استفاده کرده‌ام و از اجرای صحیح فایل MultiCore_HillClimbing.py بر روی همه سیستم ها مطمئن نیستم و صرفاً خودم توانستم روی Win10 64Bit که به آخرین نسخه آپدیت شده آن را اجرا کنم. همچنین امکان استفاده از کتابخانه‌های CUDA شرکت Nvidia نظیر numba و cudatoolkit برای انجام کل پردازش‌ها روی هسته‌های CUDA کارت گرافیک که تعداد بالایی دارند هم ممکن است ولی کار با این کتابخانه نسبتاً دشوار است و به طور کلی یادگیری کار کردن با آن از مبحث یک تمرین هوش مصنوعی فراتر است و به علاوه از آن جایی که من مطمئن نبودم که کامپیوتری که این کد روی آن تست خواهد شد کارت گرافیک شرکت NVIDIA را دارد یا نه، از این روش‌ها استفاده نکردم و صرفاً به کتابخانه معمول threading پایتون اتکا کردم و در فایل آخر هم از

multiprocessing استفاده کرده‌ام که روش بهتری است ولی به دلیل این که خیلی با آن کار نکرده‌ام در مورد پایدار بودن کد مطمئن نیستم.

حال در مورد روش‌های استفاده شده در حل سؤال توضیح می‌دهم.

در ابتدا در مورد نحوه انتخاب همسایه‌ها توضیح می‌دهم. در این سؤال همسایه‌ها به این صورت انتخاب شده‌اند که حالت فعلی به یک تابع به نام all_neighbours_generator داده می‌شود. این تابع علاوه بر این یک عدد به نام step هم می‌گیرد. با گرفتن این دو، اگر اندازه بردار داده شده به آن l باشد، تابع در مجموع 2l همسایه تولید می‌کند که در هر کدام یکی از متغیرها انتخاب شده و به اندازه یک step به بالا یا پایین رفته است. در نهایت از آن جایی که ممکن است در صورت وجود تعداد متغیر زیاد تعداد همسایه‌ها خیلی زیاد بشود، اگر این تعداد بیش از ۴۰ عدد بشود، تنها ۴۰ تا از آن‌ها به صورت رندوم انتخاب می‌شوند. این موضوع برای این است که در تعداد متغیر بالا، تولید و چک کردن همسایه‌ها (به خصوص در HillClimbing) خیلی طول نکشد. ضمن این که بدیهتا تعداد کل همسایه‌ها خیلی زیاد است و نمی‌توانم همه را تولید کنم و به ناچار با تغییر یک متغیر کنار آمده‌ام.

برای تابع هزینه از MSE یا Mean Square Error استفاده کرده‌ام. فرض کنید که مثلاً ما بردار X را به عنوان جواب پیدا کرده‌ایم. این بردار را یکی‌یکی در هر کدام از ضابطه‌ها جایگزین می‌کنیم و اگر m معادله داشته باشیم، به m عدد می‌رسیم که آن‌ها را $A_i$ می‌نامیم. از طرفی m عدد هم به عنوان سمت راست معادله داریم که آن‌ها را $B_i$ می‌نامیم و MSE را به این صورت تعریف می‌نماییم:

$$MSE = \frac{1}{m}\sum_{i=0}^{m-1}(A_i - B_i)^2$$

طبق درس آمار و احتمال معمولاً از این روش برای برآورد میزان خوب بودن تخمین استفاده می‌شود.

در ابتدا من صرفاً از طریق قدر مطلق اختلاف‌ها یعنی

$$Cost = \sum_{i=0}^{m-1}|A_i - B_i|$$

استفاده کرده بودم که نسبتاً جواب‌های خوبی می‌داد ولی با تغییر آن به MSE مشاهده کردم که دقت جواب‌ها تا حدی بهبود یافت و از این رو از این تابع استفاده کردم.

طبق این مطلب (لینک) هم چون MSE به اعداد خیلی پرت بار معنایی بیش‌تری می‌دهد و بیش‌تر جریمه می‌کند، معیار بهتری نسبت به قدر مطلق‌ها است.

معیار من هم برای دقت نهایی خود MSE و ریشه آن که RMSE گفته می‌شود بوده است. به علاوه حاصل جایگذاری اعداد در معادلات را هم نمایش می‌دهم و می‌توانید مشاهده کنید که با دقت بسیار بالایی به اعدادی که در سمت راست معادله قرار می‌گیرند نزدیک هستند.

در مورد روش‌های گفته شده، همان طور که در بالا هم گفته شد، برای فایل HillClimbing.py تابع variable_step_hill_climber روش hill climbing را صرفاً روی یک نقطه اولیه اجرا کرده و در نهایت احتمال خطا داشتن آن زیاد است. در این روش step ها هم متغیر هستند و اگر نتواند بین همه همسایگان، حالت بهتری بیابد، step را ۲ برابر کاهش می‌دهد. همچنین تا حدی جلو می‌رود که خطا از 0. 000000001 کمتر شود و پس از آن متوقف می‌شود. ضمن این که اگر مراحل کاهش step هم برابر یا بیش از ۳۰ مرحله بشود باز هم متوقف می‌شود. همچنین اگر تعداد کل مراحل از ۱۰۰۰۰۰۰۰۰ (یک میلیارد) هم بیش‌تر شود متوقف می‌شود هر چند بعید است که واقعاً به این حالت برسیم.

تابع random_restart_variable_step_hill_climber هم چنین روشی را پیاده سازی می‌کند ولی با این تفاوت که هر بار که تعداد مراحل کاهش step از بیش‌تر مساوی ۳۰ بشود، بهترین حالت فعلی را ذخیره کرده و دوباره از اول عملیات را restart می‌کند. این عمل تا حدی پیش می‌رود که تعداد کل مراحل ۱۰۰۰۰ را رد نکند و پس از آن کلاً به اتمام می‌رسد و بین همه بهترین حالت‌های هر دفعه، بهترین را پیدا می‌کند.

هر دوی این توابع در ورودی خود یک تابع hill_climber هم می‌گیرند که در اصل هسته اصلی hill_climbing است و به این صورت رفتار می‌کند که بین همسایه‌ها، تابع هزینه را محاسبه کرده و بهترین (کمترین هزینه) را انتخاب می‌کند و best_state گلوبال را آپدیت می‌کند. اگر هم کلاً هیچ کدام بهتر نباشد، false به معنی عدم تغییر بر می‌گرداند. توجه کنید که best_state هر کدام از random_restart ها در متغیر all_of_best_states ذخیره شده است و ربطی به این best_state ندارد.

در مورد فایل MultiThread_HillClimbing.py هم به طور همزمان ۴ نقطه همزمان بررسی می‌شوند. فقط باید توجه کرد که در این حالت از تابع variable_step_hill_climber استفاده می‌کنیم. هر چند نتیجه نهایی نسبتاً

خوب است اما تعداد نقطه‌های انتخابی را پایین در نظر گرفتم. در ابتدا حدود ۲۰۰ نقطه بودند که به دلیل اجرای همزمان ۲۰۰ نقطه روی یک هسته CPU زمان زیادی صرف می‌شود. اصول کلی این حالت مشابه بخش قبل است.

در مورد فایل MultiCore_HillClimbing.py این فایل از طریق پردازش موازی روی همه هسته‌های CPU با تستی که من کردم در حدود ۳۰ ثانیه ۵۰ پردازش موازی روی حالات رندوم مستقل را به خوبی انجام داد. با این وجود برای ۲۰۰ پردازش از نظر سیستمی با یکسری مشکل مواجه شد و یکسری خطاهای مربوط به عدم امکان اجرا داد. سی پی یو من Intel Core i7 8750H است که ۶ هسته فیزیکال پردازشی و ۱۲ پردازنده لاجیکال دارد. کد این بخش را من نزدیک به لحظات ددلاین نوشته‌ام و نمی‌توانم پایداری کامل آن را تضمین نمایم. هر چند کلاً سؤال هم پردازش چندهسته‌ای از ما نخواسته بود و من صرفاً برای یادگیری بیش‌تر از این کتابخانه استفاده کرده‌ام. اجرای این کد را صرفاً روی Win10 64bit با آخرین آپدیت تست کرده‌ام و مخصوصاً روی سیستم عامل‌های مبتنی بر UNIX نظیر اوبونتو به دلیل یکسری تفاوت در مورد هندل شدن اجرای چندهسته‌ای از روی سیستم عامل، نمی دانم رفتار درستی نشان خواهد داد یا نه.

در مورد فایل SimulatedAnnealing.py روش کار من به این صورت بوده که هر بار به روش قسمت‌های قبل همسایه تولید می‌کنم. بعد یکی‌یکی همسایه‌ها را بررسی می‌کنم. اگر بهتر بودند که انتخاب می‌شوند. اگر بهتر نبودند با احتمال $e^{(cost_{curr}-cost_{neighbour})/t}$ انتخاب می‌شوند. که cost ها به روش mse انتخاب شده و چون در این جا cost معیار بوده در اصل باید حالت فعلی را منهای همسایه‌اش بکنیم که عدد منفی به دست بیاید. چون حالت‌های بدتر به معنی cost بیش‌تر همسایه هستند و باید عدد صورت منفی بشود. برای تغییر t به این صورت است که من چهار پارامتر تعریف کرده‌ام. CONSTANT_FACTOR و T_start و T_end و alpha. در این جا T_ start = 100 * CONSTANT_FACTOR و

T_end = 0. 0000001* CONSTANT_FACTOR . در اصل CONSTAT_FACTOR=0.05 معیاری است برای این که در هنگام تقسیم شدن در توان عددهای کلی معنادار باشند و عددهای خیلی بزرگ یا کوچک به دست نیایند. alpha=0.9996 هم میزان کم شدن t در هر مرحله است و بعد از هر مرحله t=t*alpha می‌شود. خود t هم از T_start شروع کرده و تا زمانی که به T_end نرسد برنامه ادامه دارد. در طول کار بار عددهای CONSTAT_FACTOR و alpha و بازه زمانی بین هر بار تغییر alpha را عوض کرده‌ام تا به این اعداد رسیده‌ام. در alpha های کمتر (مثلاً ۰,۹ یا ۰,۹۹)و یا حالت‌هایی که alpha با بازه‌های زمانی زیاد بین مراحل عوض می‌شد

و یا زمانی که CONSTANT_FACTOR خیلی زیاد بود، عموماً جواب‌ها دقت خوبی نداشتند. این اعداد با تست حالت‌های مختلف به دست آمده‌اند و خصوصاً CONSTANT_FACTOR های بیش‌تر یا کمتر عموماً همگی مشکل زا بودند. برای alpha هم بیشتر کردن و نزدیک‌تر کردن آن به ۱ هر چند شاید تا حدی به دقت الگوریتم کمک کند ولی زمان اجرا را هم بیش از اندازه بالا می‌برد. زمان اجرای فعلی حدود 10 ثانیه (برای نمونه داده شده در سؤال) است که نسبتاً خوب است.

در روش من، در صورتی که بین همه همسایه‌های تولید شده به دلیل احتمال هیچ کدام انتخاب نشوند، step حدود ۱,۵ برابر کاهش یافته و دوباره همسایه‌های جدید تولید می‌شوند. یعنی در کل شرط کاهش step همین موضوع است که الگوریتم در یک نقطه گیر کند و احتمالات کوچک باشند و هیچ کدام از همسایگان هم بهتر نباشند.

در مجموع با تغییرات مختلف و تجربی سر این اعداد، به اعدادی که اکنون در کد هستند رسیده‌ام که هم از نظر زمان نتیجه خوبی بدهد و هم از نظر سرعت.

برای بررسی نتیجه حدودی توجه کنید که: (زمان‌ها براساس نمونه داده شده)

روش variable_step_hill_climber:

معمولاً خطای حدود 0. 000000001 تعیین شده را در زمان حدود ۱,۵ ثانیه می‌دهد. هر چند بعضاً من با مواردی مواجه شدم که خطای خوبی به دست نیامد اما این موارد نادرند.

روش random_restart_variable_step_hill_climber:

معمولاً خطای حدود 0. 000000001 تعیین شده را در زمان حدود ۹ ثانیه جواب می‌دهد. تقریباً مورد نادری که خطای خوبی نداشته باشد را پیدا نکردم.

روش MultiThread_HillClimbing که در اصل انجام روش variable_step_hill_climber روی همزمان تعدادی نقطه است:

برای ۴ نقطه در حدود ۴ ثانیه با دقت 0. 000000001 ولی برای ۲۰۰ نقطه زمان زیادی نزدیک به ۵ دقیقه. دلیل این موضوع این است که پایتون و کتابخانه Threading خیلی قادر به تقسیم روی هسته‌های مختلف پردازنده‌ها نیست. اگر این را با کتابخانه‌های مربوط به CUDA نوشته می‌شد، سرعت کار به شدت بالا می‌رفت که به دلیل

پیچیدگی کار با این کتابخانه و عدم اطمینان از این که کامپیوتری که کد روی آن اجرا می‌شود، کارت گرافیک NVIDIA دارد یا نه این کار را نکردم.

در روش MultiCore_HillClimbing.py که این هم با استفاده از variable_step_hill_climber همزمان روی تعدادی نقطه‌ها انجام می‌شود:

برای ۵۰ نقطه در حدود ۳۰ ثانیه با دقت 0. 000000001 می‌دهد. این فایل به دلیل استفاده واقعی از هسته‌های CPU به شدت از روش قبلی که با استفاده از ترد بود بهتر است. اما به هر حال چون اولین بار بود که از کتابخانه multiprocessing استفاده کردم، در مورد پایدار بودن کامل کد از نظر ثبات اجرا و امکان ادامه کار توسط سیستم عامل، به خصوص برای ورودی‌های بزرگ‌تر مطمئن نیستم. چون کد فعلی استفاده از CPU من را روی همه هسته‌ها به ۱۰۰ درصد می‌رساند.

برای روش SimulatedAnnealing با اعداد گفته شده در بالا:

در حدود ۱۰ ثانیه جوابی با همان دقت به دست می‌دهد. البته توجه کنید که با کمتر کردن T_end می‌توان دقت را بالاتر برد ولی زمان اجرا هم افزایش می‌یابد. البته با تغییر alpha به ۰,۹۹۹ تقریباً در زمان ۵ ثانیه‌ای به همان جواب می‌توان دست یافت. ولی باید توجه کرد که دیگر خیلی alpha را از این پایین‌تر نیاوریم. چون در این صورت با وجود کاهش زمان، MSE بالا رفته و یعنی خطا افزایش می‌یابد. در کل آن طور که من فهمیدم دقت و زمان به شدت به alpha وابسته هستند و همین طور تغییر در CONSTANT_FACTOR و زیاد یا کم کردن بیش از حد آن (به خصوص به خاطر تغییر در دمای شروع که میزان حرکات تصادفی خیلی غیر منطقی اولیه را مشخص می‌کند) هم به شدت مشکل زا می‌شود و الگوریتم نسبت به این موارد بسیار حساس است.

همچنین توجه کنید که در عبارات بالا منظور من از دقت یا خطا میزان MSE گزارش شده توسط تابع cost_function برای جواب نهایی است.

یک تابع به نام report_maker هم نوشته‌ام که کلاً گزارش مدت زمان اجرا و جواب نهایی و MSE و نتیجه جایگذاری جواب نهایی در عبارات را نمایش می‌دهد.

با مشاهداتی که داشتم و اعدادی که به طور تجربی به دست آوردم، تقریباً توانسته‌ام همه الگوریتم‌ها را تا حد خوبی از نظر زمانی و دقت نزدیک هم بیاوریم. اما اطمینان روش random_restart و SimulatedAnnealing با

توجه به اعداد تعیین شده و نتیجه‌ای که در نهایت به دست آوردم در چندین بار تست کردن، نسبتاً نتایج معتبرتری بود. ضمن این که روش MultiThread اگر واقعاً برای هسته‌های CUDA کارت گرافیک نوشته شود و روی تعداد تست بالا بر روی کارت گرافیک قوی نظیر کارت‌های سری GTX 1070 یا GTX 1080 یا RTX 2070 و RTX 2080 که تعداد هسته‌های CUDA بین ۲۰۰۰ تا ۴۵۰۰ دارند اجرا بشود، به شدت روش سریع و دقیقی خواهد بود اما اجرای آن روی یک هسته CPU از نظر زمانی به هیچ وجه به صرفه نیست.