به نام خدا

پروژه برنامه نویسی ژنتیک

درس هوش مصنوعی

احمد زعفرانی

97105985

بهار 98-99

راهنمای اجرای کد:

- پس از اجرای برنامه، انتخاب کنید که تابع خاصی مد نظر دارید یا خیر. سپس تعداد نقاطی آموزشی ای را که مایل هستید برای آموزش الگوریتم از آنها استفاده شود را وارد کنید (لطفا برای عملکرد بهتر، تعداد نقاط بیشتری وارد کنید).

- با وارد کردن حرف "Y" می توانید تابع مورد نظر را وارد کنید. عملگر هایی که در این برنامه پشتیبانی می شوند عبارتند از:

Sin(x), cos(x), e^x, +, -, *, /, ^

همچنین عملوند ها عبارتند از: اعداد صحیح یا اعشاری و متغییر x

توجه: تابع ورودی باید یک متغییره باشد.

لطفا برای e^x و اعداد توان دار، به ترتیب از اسامی زیر استفاده کنید:

A**b, exp(x)

مثال:$f(x) = 2 - sin(e^x * 5^9)$ را بصورت زیر وارد کنید:

2–sin(exp(x)*5**9)

- اگر "N" را وارد کنید، از شما خواسته می شود تا مختصات هر نقطه را بصورت دو عدد که با یک فاصله از هم جدا شده اند، در خطوط جدا وارد کنید (ابتدا مختصه x، سپس مختصه y).

- برای دیدن بهتر نتایج الگوریتم، توصیه می کنیم خط 359 برنامه را uncomment کنید. این خط شماره نسل، جمعیت آن نسل و ضابطه و برازش بهترین فرد آن نسل را چاپ می کند.

گزارش کار:

1. اگر ورودی برنامه بصورت یک تابع باشد، تنها از آن برای ایجاد تعدادی نقطه آموزشی استفاده می شود و هیچ کاربرد دیگری در الگوریتم ندارد؛ به این صورت که n عدد تصادفی(در ورودی گرفته شد) غیر

تکراری بین -1000 تا 1000 تولید می کنیم، مقدار تابع را به ازای این مقادیر محاسبه می کنیم و بعد از جایگذاری آنها در تابع ارزیاب، fitness بدست آمده را به عنوان هدف الگوریتم منظور می کنیم. در صورتی که ورودی تعدادی نقطه باشد، همین پروسه روی آنها اجرا می شود.

2. علت فرمت خاص ورودی و خروجی، استفاده از تابع eval() در زبان پایتون برای تبدیل رشته ورودی به عبارت ریاضی است. به همین علت، ممکن است خطای SyntaxError دریافت شود و ناشی از عدم مطابقت فرمت رشته ی ورودی با پیشفرض توابع در زبان پایتون باشد.

3. تابع ارزیاب را اینگونه تعریف می کنیم:

اگر n نقطه p_1 تا p_n، نقاط ورودی ما به برنامه با مختصات (x_1, y_1) تا (x_n, y_n) باشند، هدف این الگوریتم را رسیدن به میانگین مختصه دوم این نقاط تعریف می کنیم. به عبارت دیگر:

$$goal = \frac{\sum_{i=1}^{n} y_i}{n}$$

حال فرض کنید تابعی مانند f(x)، یکی از افرادی باشد که می خواهیم برازش آن را بدست آوریم. تابع برازش را اینگونه تعریف می کنیم:

$$fitness(f) = -|\, goal - \frac{\sum_{i=1}^{n} f(x_i)}{n} \,|$$

که x_i ها همان مختصه اول p_1 تا p_n هستند. بنابراین، میزان برازندگی یک تابع، برابر با قدر مطلق تفاضل میانگین $f(x_i)$ از میانگین y_i ها است. علامت منفی برای حفظ شدن شرط تابع برازش است. توضیح آن که: بدترین تابع، یعنی تابعی که میانگین خروجی هایش از $goal$ فاصله ی بیشتری دارند، پس میزان برازندگی آن نیز منفی تر است.

4. جمعیت نسل اولیه به تعداد 500 برابر تعداد ورودی هاست. الگوریتم تا وقتی که جمعیت آخرین نسل تولید شده کمتر از 20 شود، به کار خود ادامه می دهد و پس از آن، بهترین فرد(فردی با بیشترین برازش) آخرین نسل را به عنوان خروجی الگوریتم ارائه می کند و برنامه با چاپ کردن نتایج خاتمه می یابد.

سیاست برنامه نویسی ژنتیک مورد استفاده:

نسل کنونی را درون آرایه ای به نام current_population ذخیره می کنیم و آن را مرتب می کنیم (بر اساس برازش).

- 10% بالا (دهک اول جامعه) را عینا درون نسل بعد کپی می کنیم.
- 5٪ پایین جامعه را نیز در نظر بگیرید. مقدار برازندگی بهترین فرد این گروه را x می نامیم.
- current_population را نصف می کنیم؛ سپس هر دفعه بصورت تصادفی یک عضو را از نیمه بالای جامعه و عضوی دیگر را از نیمه پایین جامعه انتخاب می کنیم و این دو عضو را، *به امید تولید فرزندان قوی تر و یک نسل با تنوع کمتر*، با هم crossover می کنیم تا دو فرزند تولید شود. اگر این کار را برای تمامی افراد current_population تکرار کنیم و فرزندان را درون آرایه ای به نام temp بریزیم، طول این آرایه برابر با طول current_population خواهد بود.
- Temp را مرتب کنید. سپس هر فرد درون temp را که برازشی کمتر از x دارد، در نظر بگیرید: این فرد را جهش می دهیم. *اگر این جهش منجر به بزرگتر شدن برازش او شود، او را نگه می داریم*؛ در غیراینصورت آن را نابود می کنیم!

سیاست فوق تضمین می کند ضعیف ترین فرد نسل جدید، از تمامی افراد 5٪ آخر نسل گذشته بهتر باشد، حتی اگر تعداد افراد یک نسل از نسل قبلی بیشتر باشد.[1]

5. اجرای سیاست فوق بسیار شگفت انگیز بود؛ جمعیت نسل ها ازدیاد می یافت! معضلی که ناشی از همگرایی افراد یک نسل به تابعی خاص بود و باعث ادامه الگوریتم تا بینهایت می شد.
راهکار اول، این بود که توابعی که در واقع اعداد ثابت بودند را از نسل ها حذف کنیم (برای مثال چنین تابعی: exp(sin(2+67))). که متاسفانه تاثیر زیادی نداشت.
ایده بعدی حذف افراد تکراری از دامنه نسل ها بود. این روش به طرز جالبی جمعیت نسل اول را بیش از 50٪ کاهش می داد ولی تاثیر چشم گیری روی جمعیت نسل های آتی نداشت و البته شرط خاتمه الگوریتم نیز ارضا می شد.

6. جزئیات کد:

[1] منابع زیر در اتخاذ این سیاست بسیار الهام بخش بودند:
http://downloads.hindawi.com/journals/ijap/2008/197849.pdf
https://www.cs.montana.edu/~bwall/cs580/introduction_to_gp.pdf

- برای تولید یک تابع تصادفی، نیازمند تولید یک درخت دودویی هستیم. کلاس Function در واقع یک درخت باینری است. در ابتدا لازم بود تا محدودیتی برای اندازه کروموزوم ها (تابع ها در این مسئله) لحاظ گردد؛ بنابراین یک حد بالا برای تعداد گره های درخت (حداکثر 10 گره) در نظر گرفتیم و تضمین می شود که مجموع عملگرها و عملوند های یک تابع هرگز بیشتر از 20 نمی شود (حتی در اثر crossover). برای مثال، عبارت زیر 6 گره دارد:

$$x^{(sin(x)-x)}$$

- تولید درخت دودویی تصادفی به روش زیر انجام می شود:

عددی تصادفی بین 2 تا 10 می آوریم و آن را به عنوان طول کروموزوم یا درخت عبارت در نظر می گیریم.

سپس عملگری تصادفی از میان مجموعه {+و-و*و/وsinوcosو توان و exp} انتخاب می کنیم. یک نمونه (object) هم از کلاس Node می سازیم و عملگر انتخابی را به عنوان مقدار این نود قرار می دهیم. از این پس به این گره ریشه اطلاق می کنیم.

حال متد generate_function را برای ریشه صدا می زنیم. وظیفه این تابع تولید یک زیر درخت برای ریشه است، بطوریکه اولا مجموع تمامی این گره ها برابر طول کروموزوم شود، و ثانیا پیمایش میان ترتیب(in order) این درخت در نهایت یک عبارت ریاضی معتبر را تولید کند.

روش کار generate_function هم به این صورت است که برای تولید هر گره بصورت تصادفی انتخاب می کند مقدار گره از چه نوعی باشد: binary, unary, x, constant

الف) binary: زیر درخت های چپ و راست را برای این گره تولید می کند (خودش را بصورت بازگشتی صدا می زند).

ب) unary: فقط زیر درخت راست را برای این گره تولید می کند (خودش را بصورت بازگشتی صدا می زند).

ج) x: node.value را برابر با رشته "x " قرار می دهد و return می کند (این گره یک برگ است).

د) constant: node.value را برابر با عددی رندوم بین -10 تا 10 قرار می دهد و return می کند (این گره یک برگ است).

توجه: برای جلوگیری از تولید توابع ثابت، برای برگ های درخت x را با احتمال 90٪ و اعداد ثابت را به احتمال 10٪ برای این گره انتخاب می کنیم.

- تابع calculate در کلاس Function، وظیفه جایگذاری x_i ها را در تابع تولید شده و محاسبه مقدار $f(x_i)$ را دارد. خطای متداول این تابع، overflow است (این سرریز ناشی از جا نشدن عدد تولید شده در نوع float یا تقسیم کردن بر صفر یا... است).

- برای ترکیب متقاطع دو فرد، تنها کافیست یک گره تصادفی از فرد اول و یک گره تصادفی دیگر از فرد دوم انتخاب کنیم و جای این دو گره و زیر درخت هایشان را با هم عوض کنیم. جهش یک فرد هم بسیار ساده انجام می شود: یک گره از درخت را بصورت تصادفی انتخاب می کنیم و مقدار آن را با توجه به مقدار قبلی تغییر می دهیم (برای مثال اگر مقدارش + است، یک "عملگر دودویی" تصادفی دیگر را به جای آن قرار می دهیم).

- اصلی ترین چالش برای تمامی موارد بالا احتمال بی معنی بودن عبارت ریاضی خروجی پس از تولید یک فرد، جهش یک فرد یا ترکیب متقاطع دو فرد است. برای رفع این مشکل، شرط خاتمه توابع مربوط به این اعمال منوط به عدم دریافت SyntaxError است. در غیر این صورت کل عملیات دوباره تکرار می شود. به همین علت ممکن است مثلا برای تولید یک درخت عبارت معتبر، تابع generate_function بارها صدا زده شود (هر چند سعی شده حتی الامکان حالت های خاص پیش بینی شوند و احتمال تولید نمونه های ناقص الخلقه! به حداقل برسد).

7. از ماژول های heapq , decimal که جزو پکیج های پیش فرض زبان پایتون هستند، بهره برده ایم. ماژول decimal برای تبدیل اعداد ممیز شناور به ممیز ثابت و کاهش دقت محاسبات تا 6 رقم اعشار استفاده می شود. Heapq هم با تبدیل یک آرایه به یک هرم دودویی (min-heap)، مرتب نگه داشتن آرایه و توابع پیش فرض برای پیدا کردن n% بالا یا پایین یک جامعه به کاهش زمان اجرای برنامه کمک می کند.

8. هم اکنون نتیجه چند آزمایش را برای شناخت بهتر این برنامه در جدول زیر جمع آوری کرده ایم:

این پاسخ را از کدام نسل پیدا کرده بود	تعداد نسل	زمان اجرا (ثانیه)	جواب الگوریتم	تابع ورودی
1	**27**	**25.095162**	**2+x**	**X+2**
4	**24**	**21.572487**	**X**2+x-84**	**x**2+10*x-28**
5	**21**	**36.322555**	**Sin(-77)+79**cos(x)**	**Sin(2**x)+10**
2	**16**	**6.507337**	**(cos(cos(x)))**x**	**x**6-x**5+x**4-2*x**2-12**

تجربه نشان داد هر چه تابع پیچیده تر باشد، دقت الگوریتم کاهش می یابد. از طرفی الگوریتم جواب های را در نسل های ابتدایی پیدا می کند و ادامه الگوریتم صرفا جهت کاهش جمعیت یک نسل است.

در جدول زیر تعداد نقاط آموزشی را 100 نقطه در نظر گرفتیم که به معنی افزایش جمعیت اولیه به 50000 تابع است:

تعداد نسل	زمان اجرا (ثانیه)	برازندگی پاسخ	تابع ورودی
29	685.702800	0.00001-	cos(2*x)-sin(x)/cos(x)
24	775.267890	0.00024-	cos(45*x)*x
14	91.514271	-2.63 * 10**75	exp(2*x-11)+sin(x)
0	32.838254	0	exp(x)/x+x**x

9. نتیجه گیری:

- این الگوریتم بار محاسباتی زیادی دارد. همچنین برای عملکرد بهتر آن تعداد نقاط اولیه بیشتری لازم است.
- جذابیت این الگوریتم این است که شما بدون انجام هیچ عملیات ریاضی پیشرفته یا رگرسیون یا ... به توابعی می رسید که بعضا شباهت زیادی به به ورودی دارند.
- برنامه ممکن است توابع با ضابطه طولانی تولید کند، ولی اکثرا با ساده کردن این توابع می توان به شباهت پاسخ و ورودی پی برد.

پایان