به نام خدا

پروژه سوم درس هوش مصنوعي – برنامه نویسی ژنتیک بهاره كاوسي نژاد – 99431217

ابتدا كتابخانه هاي لازم را import مي‌كنيم.

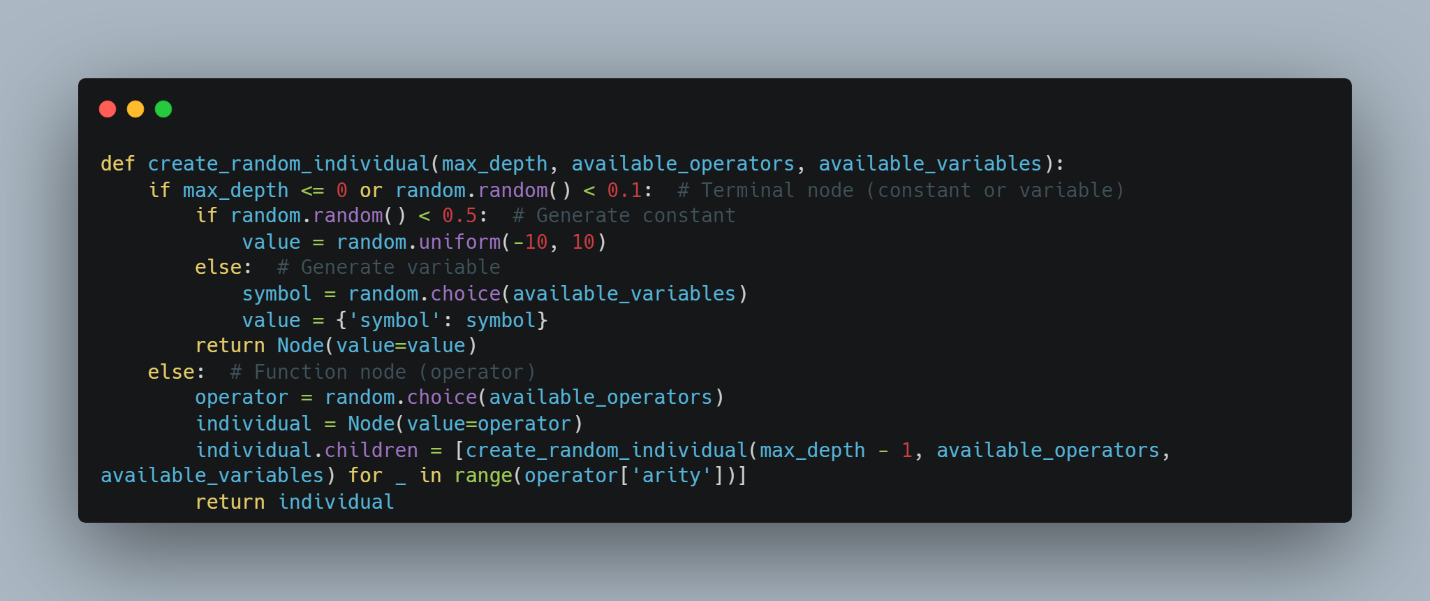


کلاس Node را تعریف می کنیم؛ از این کلاس برای نمایش عبارات ریاضی به صورت درختی استفاده می شود. این کلاس از بخش های زیر تشکیل شده است:

* در constructor این کلاس، سه attribute داریم: type، value و children
* در تابع evaluate مقدار گره کنونی و فرزندانش را تخمین می زنیم؛ بدین صورت که
  + اگر value گره کنونی برابر با None بود، مقدار 0 را برمیگرداند
  + اگر مقدار value گره کنونی یک رشته بود، بدین معنا است که این گره یک عملگر یا یک تابع است؛ این تابع به صورت بازگشتی و با تخمین زدن فرزندان، مقدار عملگر یا تابع مربوطه را به دست می آورد. به عنوان مثال اگر مقدار value، ‘+’ باشد، فرزند اول و دوم را با هم جمع می کند.
  + اگر مقدار value گره کنونی یک عدد (int یا float) باشد، همان مقدار عددی را بازمیگرداند.
  + اگر مقدار value یک دیکشنری (یعنی متغیر) باشد، نام متغیر را از دیکشنری گرفته و چک می کند که در لیست available\_variables قرار دارد یا خیر. اگر نام متغیر در این لیست موجود بود، value مربوطه را از لیست input\_values بازمیگرداند.
  + برای valueهای unknown مقدار 0 را برمیگرداند.
* تابع print\_function، یک رشته حاوی تابع نمایش داده شده توسط گره را برمیگرداند.
  + اگر value، None باشد، به این معنا است که به آن گره مقداری assign نشده است، پس رشته ‘0’ را برمیگرداند.
  + اگر value، یک رشته (یعنی عملگر یا تابع) باشد، به صورت بازگشتی print\_function را روی گره فررزند صدا می کند تا آنها را به صورت رشته به همراه مقدار value گره کنونی نمایش دهد.
  + اگر مقدار value گره کنونی یک عدد (int یا float) باشد، آن را به رشته تبدیل کرده و بازمیگرداند.
  + اگر مقدار value یک دیکشنری (یعنی متغیر) باشد، نام متغیر را از دیکشنری گرفته و آن را به شکل یک رشته بازمیگرداند.
  + برای valueهای unknown مقدار ‘0’ را برمیگرداند.



* تابع create\_random\_individual برای تولید random individuals با عمق های متفاوت برای برنامه نویسی ژنتیک استفاده می شود.
  + در ابتدا چک می کند که max\_depth ورودی، کمتر یا مساوی با 0 است یا یک عدد تصادفی بین 0 تا 1 از 0.1 کوچکتر است یا خیر. اگر هرکدام از این شروط درست باشند، بدین معنا است که یک گره پایانی (terminal node) باید تولید شود (constant or variable). در این صورت، به صورت تصادفی انتخاب می شود که گره پایانی constant باشد یا variable. اگر constant باشد، یک عدد float بین -10 تا 10 تولید می شود. در غیر این صورت (انتخاب variable)، یک symbol به صورت تصادفی از لیست available\_variables انتخاب می شود و یک دیکشنری با کلید ‘symbol’ و مقدار symbol انتخاب شده ساخته می شود. در نهایت، یک گره با value تعیین شده تشکیل می شود و این گره بازگردانده می شود.
  + اگر شرط ساختن گره پایانی برقرار نشود، بخش بعدی کد اجرا می شود مشخص می کند یک گره تابع (عملگر) باید تولید شود. این قسمت به صورت تصادفی یک عملگر از لیست available\_operators انتخاب می کند. یک گره جدید با value این عملگر انتخاب شده تولید می شود. فرزندان این گره به صورت تصادفی توسط تابع create\_random\_individual تولید می شوند که مقدار max\_depth در آنها کاهش یافته اما همان available\_operators و available\_variables را داریم. تعداد فرزندان با مقدار arity عملگر انتخاب شده تعیین می شود.



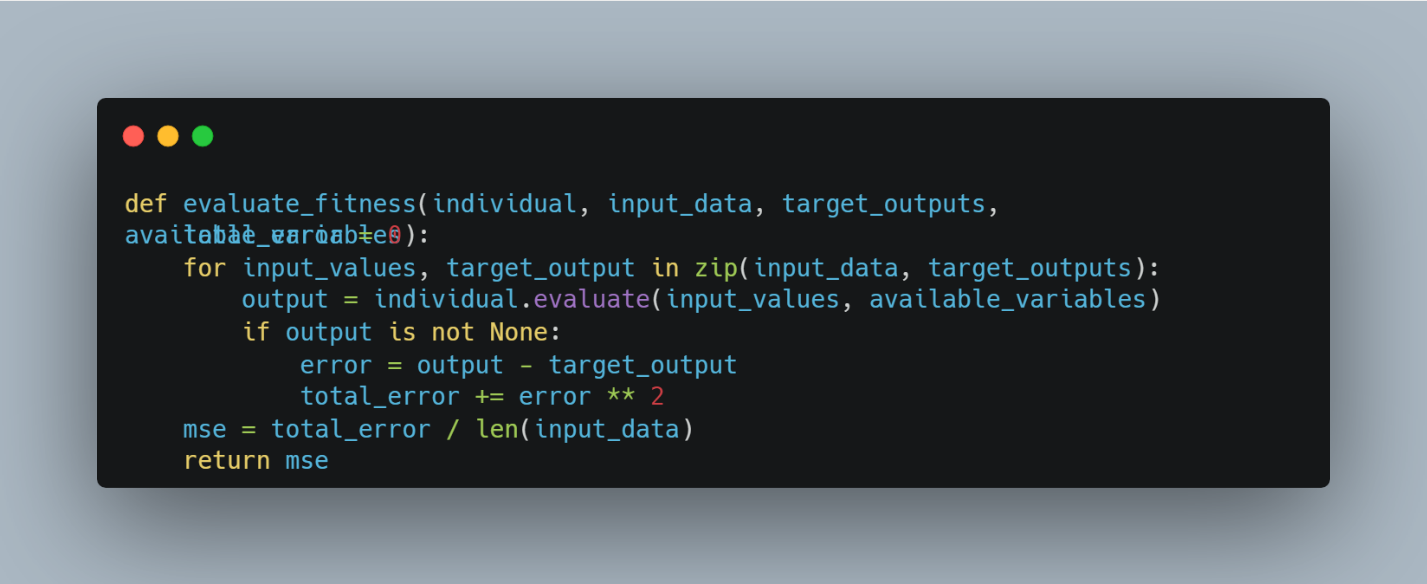
* تابع crossover، در برنامه نویسی ژنتیک برای ایجاد فرزند استفاده می شود. در ابتدا مقدار value فرزند برابر با value والد اول (عملگر یا تابع) قرار داده می شود.
* با استفاده از تابع zip یک پیمایش روی فرزندان والد اول و دوم انجام می شود و به صورت بازگشتی تابع crossover صدا زده می شود تا گره های فرزند ایجاد شده و در یک لیست ذخیره شوند.
* در انتها نیز گره فرزند ایجاد شده بازگردانده می شود.



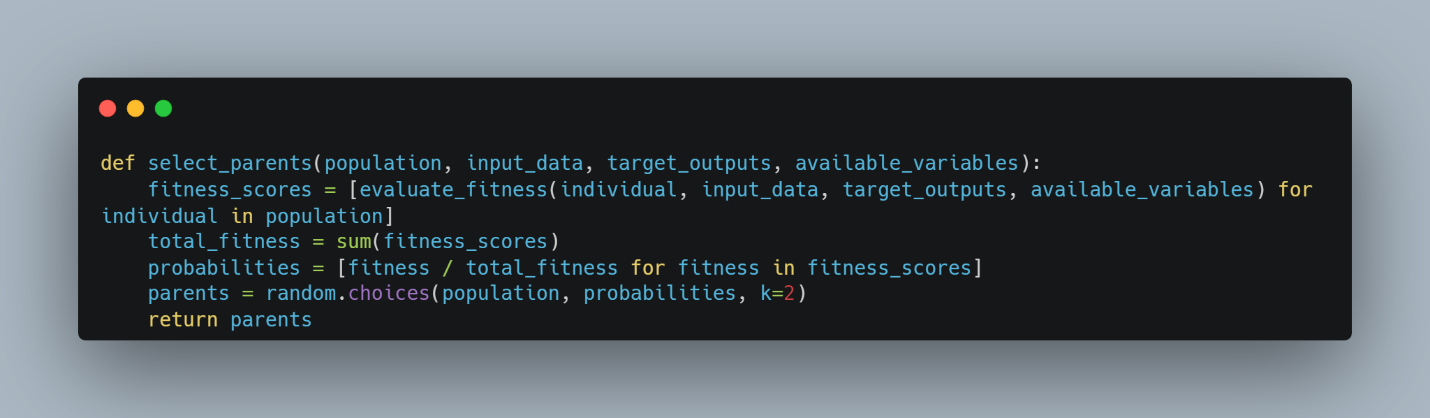
* تابع mutation در برنامه نویسی ژنتیک برای ارائه تغییرات تصادفی در یک جمعیت استفاده می شود.
* این تابع در ابتدا چک می کند که max\_depth از 0 بزرگتر باشد. در این صورت می توان جهش بیشتری روی یک گره اعمال کرد. در این صورت، با استفاده از enumerate روی تمامی گره های فرزند پیمایش می کند و در هر گره فرزند به صورت بازگشتی تابع mutation را صدا می زند و max\_depth را برای آن یک واحد کمتر در نظر میگیرد.
* اگر نتوان جهش بیشتری روی یک گره اعمال کرد، بدین معنا است که یک برگ باید جهش یابد. در ابتدا چک می کند که مقدار value یک رشته است یا خیر. اگر رشته بود، چک می کند که آیا در میان عملگرها و متغیرهای موجود می توان این رشته را پیدا کرد یا خیر. در صورت پیدا شدن، به صورت تصادفی از میان گزینه های موجود یک مقدار دیگر برای آن انتخاب می کند و در غیر این صورت، یک عدد تصادفی بین -10 تا 10 را به value آن assign می کند.
* اگر مقدار value رشته نبود، مستقیما یک عدد تصادفی بین -10 تا 10 را به عنوان value در نظر میگیرد.



* تابع evaluate\_fitness برای ارزیابی یک individual در برنامه نویسی ژنتیک استفاده می شود. این ارزیابی بر اساس mean squared error بین خروجی های تولید شده توسط individual و خروجی های هدف انجام می شود. هرچه MSE کمتر باشد، به معنای fitness و خروجی های بهتر است.
* در ابتدا مقدار اولیه total\_error صفر قرار داده می شود.
* با استفاده از تابع zip روی input\_values و target\_output پیمایش انجام می شود. لیست های input\_data و target\_outputs به صورت جفت جفت پیمایش می شوند.
* در مرحله بعدی ارزیابی انجام می شود و تابع evaluate برای هر گره صدا زده می شود و در output ذخیره می شود.
* اگر output، None نباشد (ارزیابی با موفقیت انجام شده باشد)، error به صورت اختلاف بین output و target\_output محاسبه می شود. توان دوم این مقدار به total\_error اضافه می شود.
* mse به صورت total\_error تقسیم بر طول input\_data محاسبه و بازگردانده می شود.



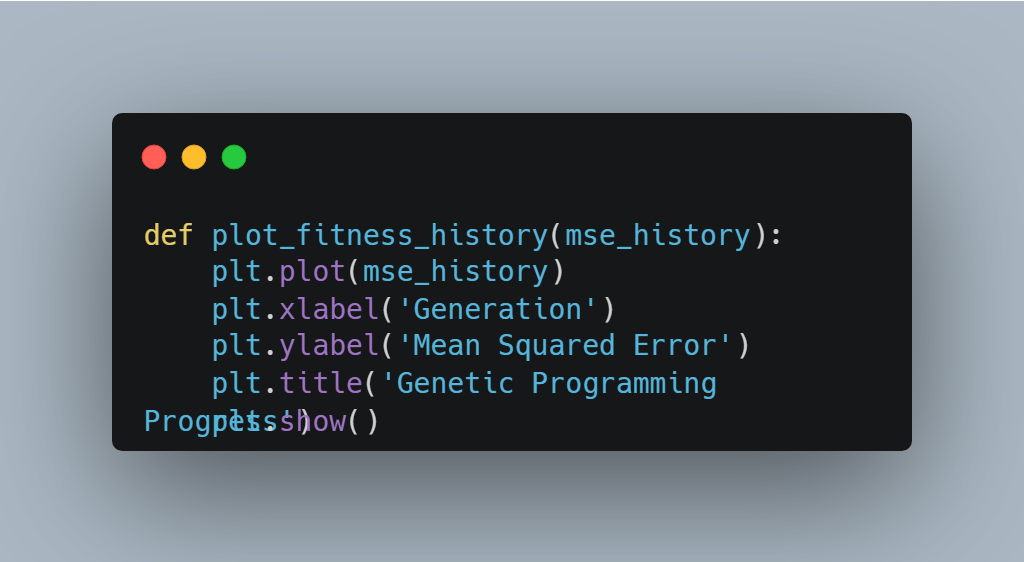
* تابع select\_parents برای انتخاب والدها از یک جمعیت برای operationهای ژنتیکی استفاده می شود.
* این تابع با استفاده از list comprehension برای محاسبه fitness score برای هر individual در جمعیت محاسبه می شود. این مقدار در یک لیست با نام fitness\_scores ذخیره می شود.
* با استفاده از تابع sum مقدار total\_fitness را محاسبه می کنیم.
* احتمال انتخاب هر individual به عنوان والد در probabilities به صورت fitness آن individual تقسیم بر total\_fitness به دست می آید.
* با استفاده از تابع random.choices دو والد از جمعیت انتخاب می کنیم و در لیست parents ذخیره می کنیم و آن را برمیگردانیم.



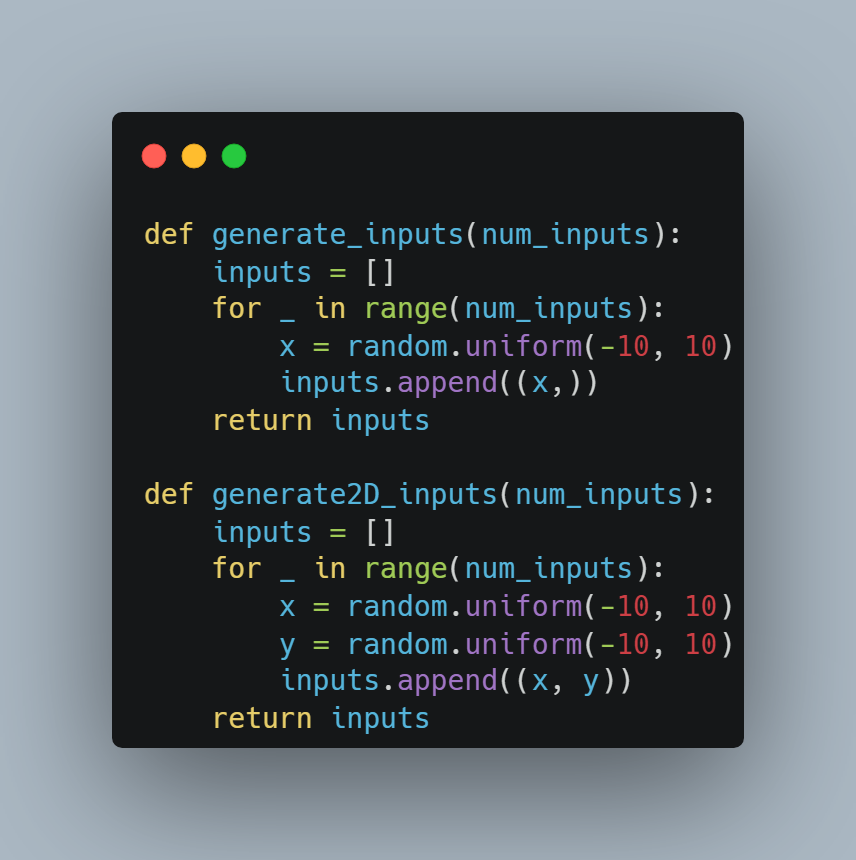
* این تابع برنامه نویسی ژنتیک را پیاده سازی می کند:
* با استفاده از تابع create\_random\_individual جمعیت اولیه را می سازیم و در population ذخیره می کنیم.
* متغیرهای best\_individual، best\_fitness، unchanged\_iterations، best\_fitness\_history و mse\_history را مقداردهی اولیه می کنیم.
* در یک حلقه for تا max\_generations پیش می رویم:
  + یک لیست به نام new\_population برای ذخیره offspring یا فرزندان نسل کنونی می سازیم.
  + در یک حلقه while، فرزندان را می سازیم. این حلقه تا زمانی ادامه میابد که اندازه جمعیت جدید با جمعیت اصلی برابر شود. با استفاده از تابع select\_parents دو والد انتخاب می کند و بعد از crossover و mutation، فرزند را به لیست new\_population اضافه می کند.
  + در مرحله بعد نسل جدید را جایگزین نسل قبلی می کنیم.
  + در یک حلقه for، fitness هر individual در جمعیت را با استفاده از تابع evaluate\_fitness تخمین می زند. مقدار fitness هر individual با مقدار best\_fitness کنونی مقایسه شده و اگر از آن بهتر (کمتر) بود، individual فعلی به new best individual جدید تبدیل می شود و fitness آن به new best fitness تبدیل می شود. همچنین شمارنده unchanged\_iterations به 0 ریست می شود. اگر fitness یک individual از best\_fitness بهتر نبود، یک واحد به unchanged\_iterations اضافه می شود.
  + مقدار best\_fitness (MSE) و best\_fitness کنونی در دو لیست best\_fitness\_history و mse\_history ذخیره می شوند تا بتوانیم تغییرات و پیشرفت fitness در طی نسل ها را بررسی کنیم.
  + اگر مقدار best\_fitness صفر بود، بدین معنا است که تابع هدف به بهترین صورت ممکن حدس زده شده است؛ پس از حلقه خارج می شویم.
  + اگر شمارنده unchanged\_iterations به unchanged\_threshold مشخص شده رسیده باشد، جهش ژنتیکی به best\_individual کنونی اعمال می شود تا variation ایجاد شود و احتمالا باعث بهبود fitness می شود. شمارنده unchanged\_iterations به صفر ریست می شود.
* تابع plot\_fitness\_history برای نمایش fitness (MSE) در طول نسل ها صدا زده می شود و best\_individual بازگردانده می شود.



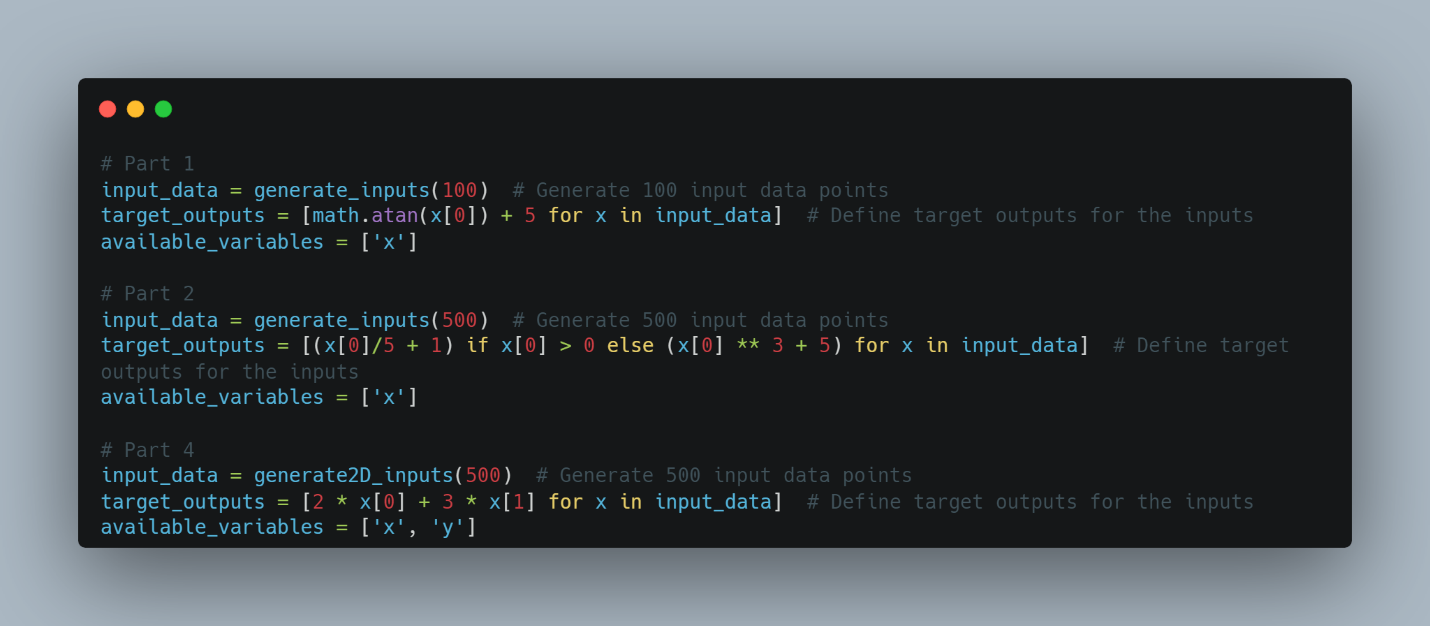
* تابع plot\_fitness\_history برای رسم mse ها استفاده می شود.



* از دو تابع generate\_inputs و generate2D\_inputs برای تولید ورودی ها بین -10 تا 10 استفاده می شود.



* در این قسمت ورودی های هر بخش پروژه مشخص شده است.



* در available\_operators عملگرهای ممکن با arity آنها را ذخیره می کنیم.



