سوال ۱-

وزنهای گروه اول(w1) به شکل روبرو است: [0.2,0.4,0.6,0.8] و وزنهای گروه دوم(w2) به شکل روبرو: [0.9,0.7,0.5,0.3]

ابتدا با نقطه [1,0,0,0] شروع می‌کنیم. اختلاف این نقطه با گروه اول برابراست با: (1-0.2) + (0.4-0) + (0.6-0) + (0.8-0) = 2.6 ، اختلافها را به صورت قدرمطلقی حساب کردم.

برای گروه دوم این عدد برابر است با: (1-0.9)+(0.7-0)+(0.5-0)+(0.3-0) = 1.6

پس نورون یا گروه دوم برنده می‌شود، و در این مرحله، ورودی اول به این گروه تعلق خواهد گرفت. حال باید وزنها آپدیت شوند تا رابطه بین گروه برنده و ورودی داده شده، قوی‌تر شود. برای آپدیت وزنها از فرمول زیر استفاده میشود.

$$\Delta w_j = \eta h_{j,i(x)}(x - w_j)$$

که در آن، h همان همسایگی است که در این سوال برابر با صفر می‌گیریم. مقدار (x-w2) برابراست با: [0.1,-0.7,-0.5,-0.3] ، پس از ضرب نرخ یادگیری مقدار آپدیت هر وزن برابر خواهد بود با: [0.05,-0.35,-0.25,-0.15] ، پس مقدار w2 پس از آپدیت برابر است با: [0.95,0.35,0.25,0.15]

استیت شبکه تا به اینجا:

w1 = [0.2,0.4,0.6,0.8], w2 = [0.95,0.35,0.25,0.15]

حال ورودی بعدی یعنی [0,1,1,0] را به شبکه می‌دهیم.

اختلاف با گروه اول: (0.2-0) + (1-0.4) + (1-0.6) + (0.8-0) = 2

اختلاف با گروه دوم: (0.95-0) + (1-0.35) + (1-0.25) + (0.15-0) = 2.5

این بار گروه اول برنده شده و این ورودی در این دسته جای خواهد داشت.

مقدار (x-w1) برابراست با: [-0.2,0.6,0.4,-0.8] و پس از ضرب در نرخ یادگیری میشود: [-0.1,0.3,0.2,-0.4] ، پس از آپدیت، وزنهای گروه اول به شکل روبرو خواهد بود: [0.1,0.7,0.8,0.4]

استیت شبکه تا اینجا: w1=[0.1,0.7,0.8,0.4], w2=[0.95,0.35,0.25,0.15]

برای ورودیهای بعدی هم به همین شکل انجام میدهیم تا مشخص شود هر کدام به کدام گروه اختصاص خواهند داشت، البته باید این کار را چند بار تکرار کنیم تا این دو فضا کاملا از هم جدا شوند و نقشه ویژگی ما شکل بگیرد. البته اینجا کلا دو گروه داریم و خیلی چالشی نیست.

سوال ۲-

ماتریس وزنهای ما در این شبکه، یک ماتریس 4*4 می‌باشد.

ابتدا ورودی x1 = [1,1,1,1] را به مدل می‌دهیم و وزنها به صورت زیر محاسبه می‌شوند.

$$w_{12} = w_{21} = 1 * 1 = 1$$

$$w_{13} = w_{31} = 1 * 1 = 1$$

$$w_{14} = w_{41} = 1 * 1 = 1$$

$$w_{23} = w_{32} = 1 * 1 = 1$$

$$w_{24} = w_{42} = 1 * 1 = 1$$

$$w_{34} = w_{43} = 1 * 1 = 1$$

ماتریس وزنها پس از این معرفی این ورودی، به شکل زیر خواهد بود.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

حال ورودی بعدی یعنی x2=[-1,-1,-1,-1] را معرفی میکنیم و با فرمول زیر وزنها را آپدیت میکنیم.

$$w_{ij} = \sum_{k=1}^{P} x_i^k x_j^k$$

وزنها برای این مرحله برابراست با

$$w_{12} = w_{21} = w_{13} = w_{31} = w_{14} = w_{41} = w_{23} = w_{32} = w_{24} = w_{42} = w_{34} = w_{43} = -1 * -1 = 1$$

$$\begin{bmatrix} * & 1 & 1 & 1 \\ 1 & * & 1 & 1 \\ 1 & 1 & * & 1 \\ 1 & 1 & 1 & * \end{bmatrix}$$

حال این ماتریس باید با ماتریس مرحله قبل جمع بشود تا وزنها را پس از معرفی این دو ورودی داشته باشیم.

$$\begin{bmatrix} * & 2 & 2 & 2 \\ 2 & * & 2 & 2 \\ 2 & 2 & * & 2 \\ 2 & 2 & 2 & * \end{bmatrix}$$

معرفی ورودی x3=[-1,-1,1,1]:

$$w_{12} = w_{21} = -1 * -1 = 1$$

$$w_{13} = w_{31} = w_{14} = w_{41} = -1 * 1 = -1$$

$$w_{23} = w_{32} = w_{24} = w_{42} = -1 * 1 = -1$$

$$w_{34} = w_{43} = 1 * 1 = 1$$

$$\begin{bmatrix} * & 3 & 1 & 1 \\ 3 & * & 1 & 1 \\ 1 & 1 & * & 3 \\ 1 & 1 & 3 & * \end{bmatrix}$$

معرفی ورودی x4=[1,1,-1,-1]:

$$w_{12} = w_{21} = 1 * 1 = 1$$

$$w_{13} = w_{31} = w_{14} = w_{41} = 1 * -1 = -1$$

$$w_{23} = w_{32} = w_{24} = w_{42} = 1 * -1 = -1$$

$$w_{34} = w_{43} = -1 * -1 = 1$$

$$\begin{bmatrix} * & 4 & 0 & 0 \\ 4 & * & 0 & 0 \\ 0 & 0 & * & 4 \\ 0 & 0 & 4 & * \end{bmatrix}$$

پس وزنهای نهایی شبکه هاپفیلد ما به صورت زیر خواهد بود:

$$\begin{bmatrix} * & 4 & 0 & 0 \\ 4 & * & 0 & 0 \\ 0 & 0 & * & 4 \\ 0 & 0 & 4 & * \end{bmatrix}$$

حال نشان میدهیم که این پترن‌ها در شبکه ما به عنوان مینیمم انرژی ذخیره شده‌اند و پایدار می‌باشند. برای این کار کافیست هر ورودی را به شبکه بدهیم، اگر با یکبار تغییر مقادیر، خروجی برابر با همان ورودی بماند، یعنی در مینیمم انرژی هستیم و پترن ما توسط شبکه به عنوان یک مینیمم انرژی به حافظه سپرده شده است.

ورودی x1 را به شبکه میدهیم و threshold را برابر با صفر میگیریم. خروجی هر نورون به صورت زیر حساب می‌شود.

$$o1 = [1,1,1,1] * [*,4,0,0] = 4 \ \ which\ after\ threshold\ will\ be\ 1$$

$$o2 = [1,1,1,1] * [4,*,0,0] = 4\ which\ after\ threshold\ will\ be\ 1$$

$$o3 = [1,1,1,1] * [0,0,*,4] = 4\ which\ after\ threshold\ will\ be\ 1$$

$$o4 = [1,1,1,1] * [0,0,4,*] = 4\ which\ after\ threshold\ will\ be\ 1$$

در نتیجه پس از یک مرحله آپدیت، تغییری نداشتیم و در همین استیت خواهیم ماند، در نتیجه این استیت یک مینیمم انرژی است.

ورودی x2=[-1,-1,-1,-1]:

$$o1 = [-1,-1,-1,-1] * [*,4,0,0] = -4\ which\ after\ threshold\ will\ be -1$$

$$o2 = [-1,-1,-1,-1] * [4,*,0,0] = -4\ which\ after\ threshold\ will\ be -1$$

$$o3 = [-1,-1,-1,-1] * [0,0,*,4] = -4\ which\ after\ threshold\ will\ be -1$$

$$o4 = [-1,-1,-1,-1] * [0,0,4,*] = -4\ which\ after\ threshold\ will\ be -1$$

برای x2 نیز همانند x1 است.

ورودی x3=[-1,-1,1,1]:

$$o1 = [-1,-1,1,1] * [*,4,0,0] = -4\ which\ after\ threshold\ will\ be -1$$

$$o2 = [-1,-1,1,1] * [4,*,0,0] = -4\ which\ after\ threshold\ will\ be -1$$

$$o3 = [-1,-1,1,1] * [0,0,* ,4] = 4\ which\ after\ threshold\ will\ be\ 1$$

$$o4 = [-1,-1,1,1] * [0,0,4,*] = 4\ which\ after\ threshold\ will\ be\ 1$$

در نتیجه x3 نیز یک مینیمم انرژی است.

ورودی x4=[1,1,-1,-1]:

$$o1 = [1,1,-1,-1] * [* ,4,0,0] = 4\ which\ after\ threshold\ will\ be\ 1$$

$$o2 = [1,1,-1,-1] * [4,* ,0,0] = 4\ which\ after\ threshold\ will\ be\ 1$$

$$o3 = [1,1,-1,-1] * [0,0,* ,4] = -4\ which\ after\ threshold\ will\ be -1$$

$$o4 = [1,1,-1,-1] * [0,0,4,*] = -4\ which\ after\ threshold\ will\ be -1$$

و در نهایت x4 هم یک مینیمم انرژی است.

سوال ۳-

برای این سوال، مشکلی که پیش می‌آید این است که هر چه گستره اعدادی که به عنوان داده آموزشی به مدل میدهیم، زیاد شود، به اعداد کوچکتری مثل بازه صورت سوال یعنی [-3,3] اهمیت کمتری داده میشود، یعنی تابع ضرر Mean Square Error ترجیح میدهد، ورودیهای بزرگتر را اصلاح کند تا اینکه بتواند مقدار loss را کاهش دهد.

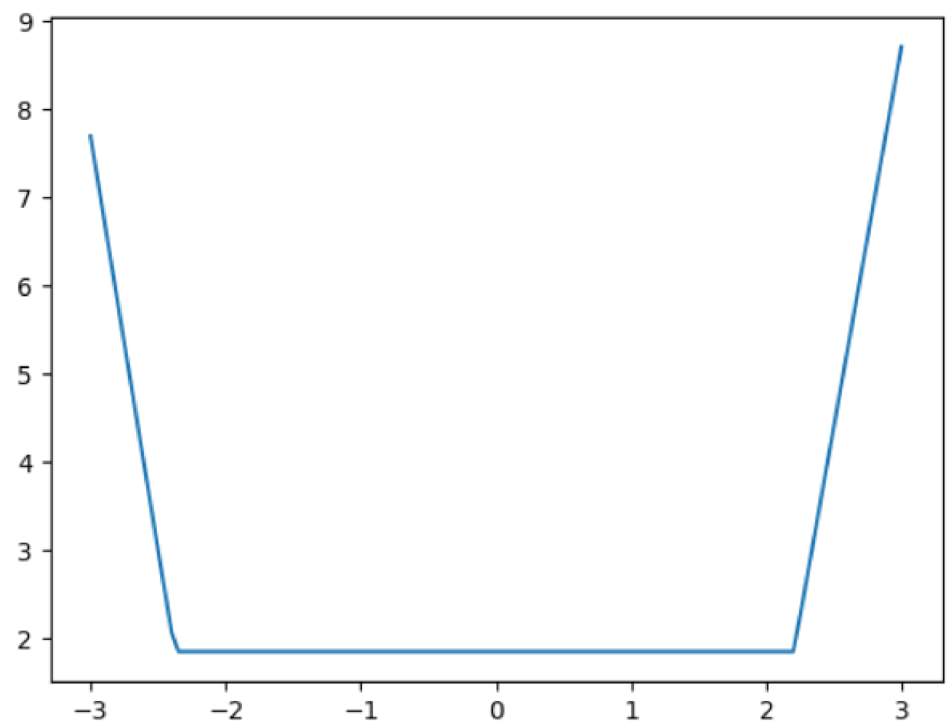
برای مثال یکبار مدل را با مقادیر زیر آموزش دادم:

```
x_train = []
y_train = []
bound = 20
step = 0.05
i = -bound
while(i <= bound):
  x_train.append(i)
  y_train.append(i**2)
  i += step
x_train = np.array(x_train)
y_train = np.array(y_train)
```

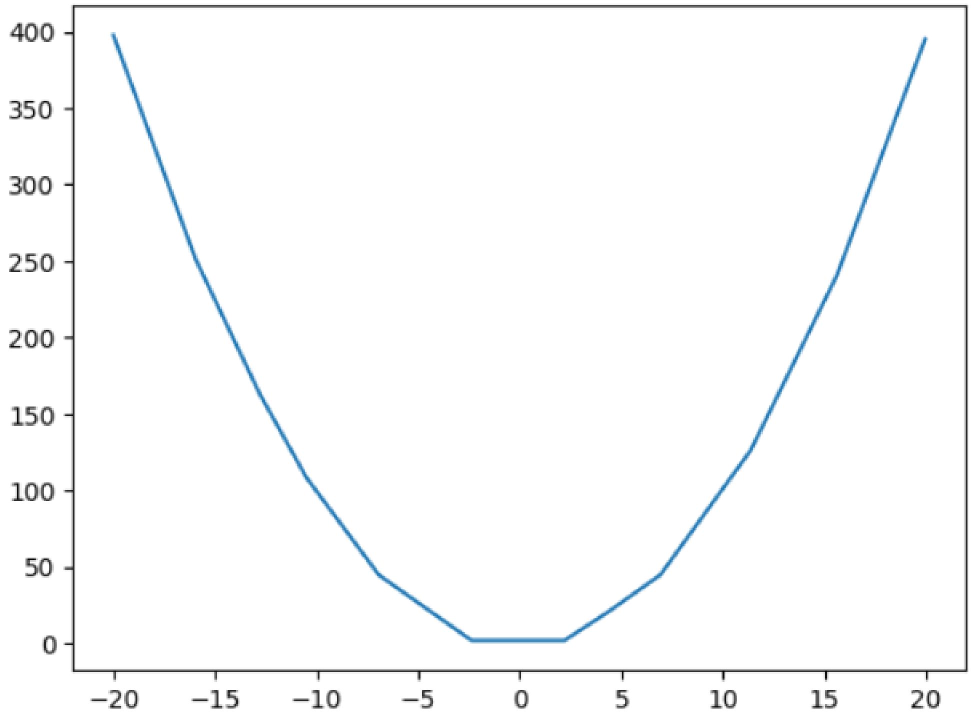
متغیر bound مشخص میکند از چه عددی در اعداد منفی تا چه عددی در اعداد مثبت، دیتاست داشته باشیم و متغیر step مشخص میکند با چه گام‌هایی نمونه برداری کنیم(مشابه دوره تناوب)

خروجی برای این بازه به صورت زیر است، زیرا مقدار زیادی عدد داریم که خیلی بزرگتر از اعداد این بازه هستند و عملا برای این بازه، مدل خیلی ضعیف عمل کرده و یادگیری نداشته.

بازه (-3,3):



بازه (-20,20)

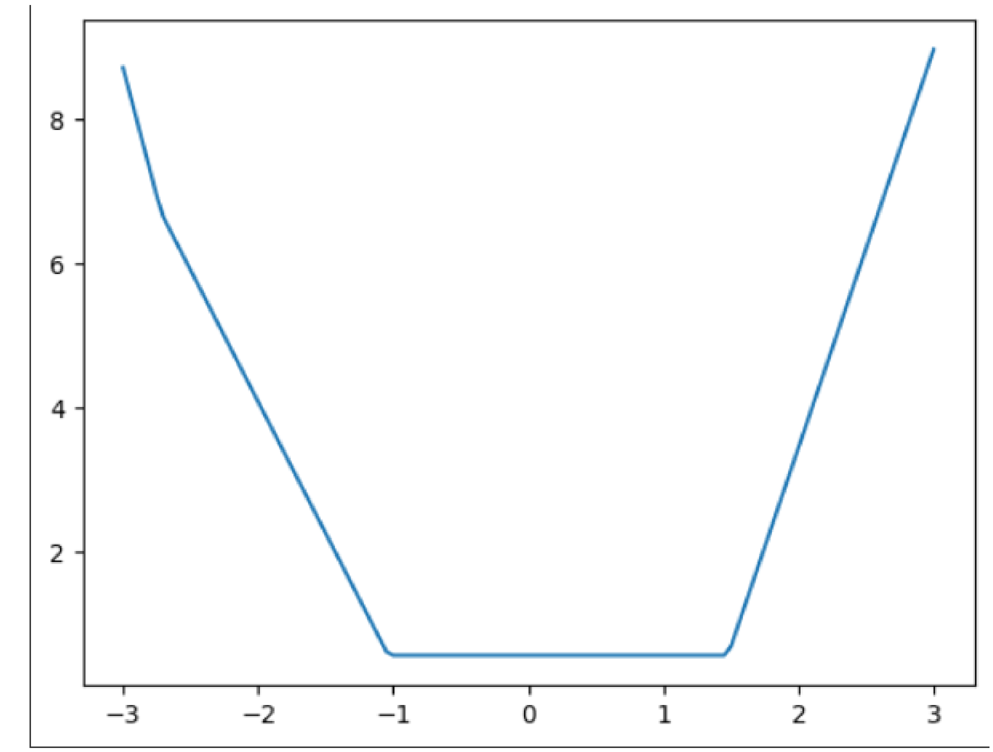


حال مقدار متغیر bound را از ۲۰ به ۱۰ تغییر می‌دهیم.

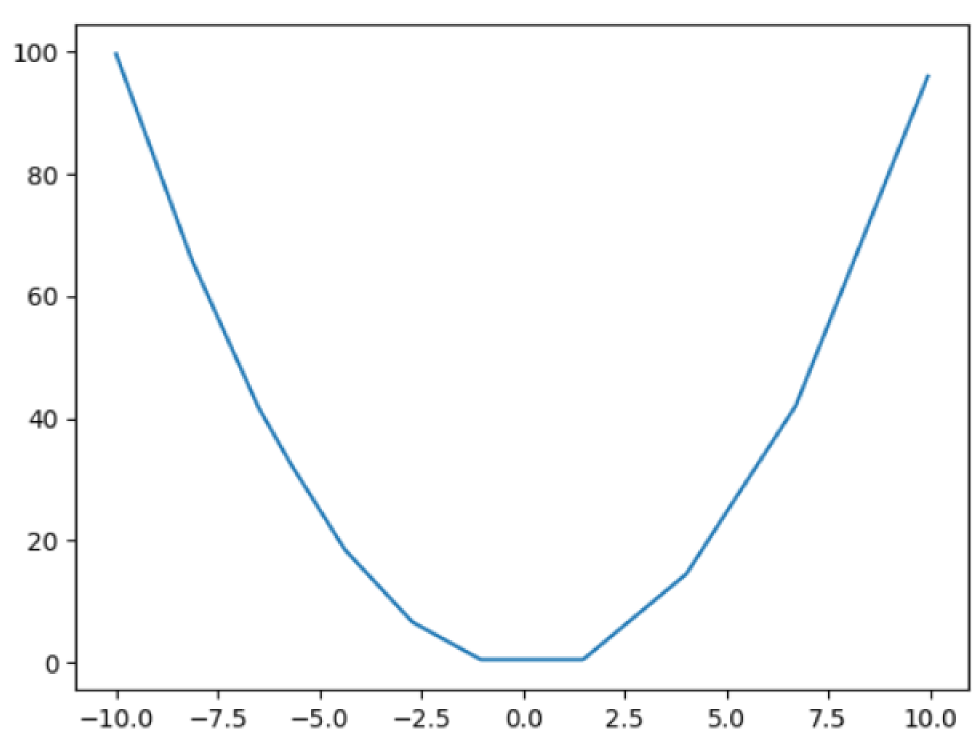
```
x_train = []
y_train = []
bound = 10
step = 0.05
i = -bound
while(i <= bound):
  x_train.append(i)
  y_train.append(i**2)
  i += step
x_train = np.array(x_train)
y_train = np.array(y_train)
```

می‌بینیم که خروجی این بازه بهتر شده است، خروجی قبلی بین ۲ و ۲- را صفر کرده بود ولی اینجا این بازه‌ی صفر شده کوچکتر شده است و به ۱ و ۱- رسیده.

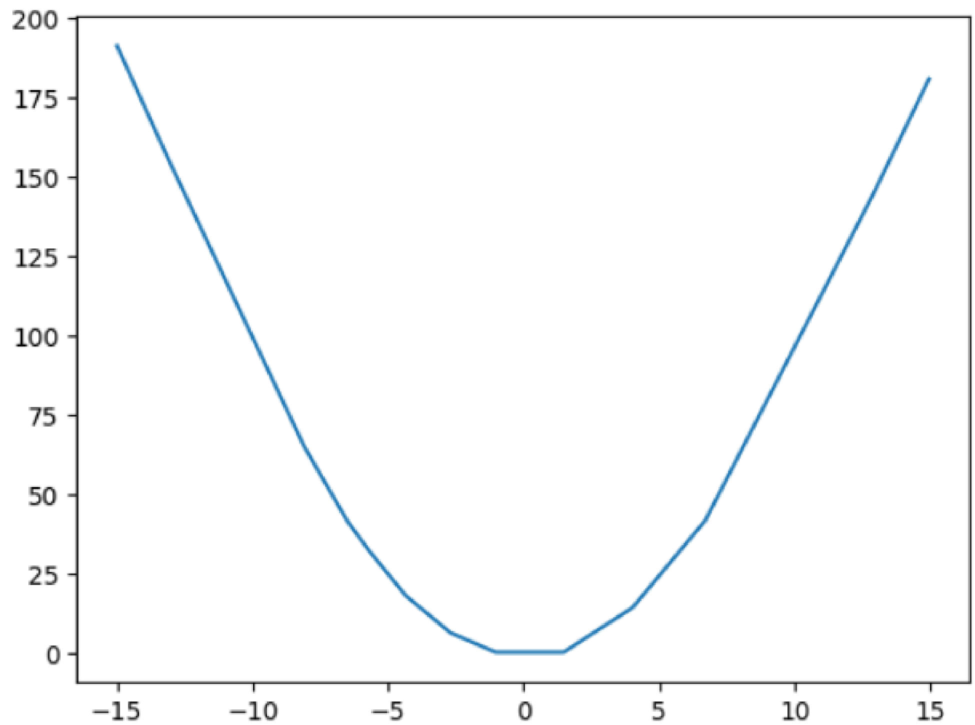
به ازای بازه (3,3-):



به ازای بازه (10,10-):



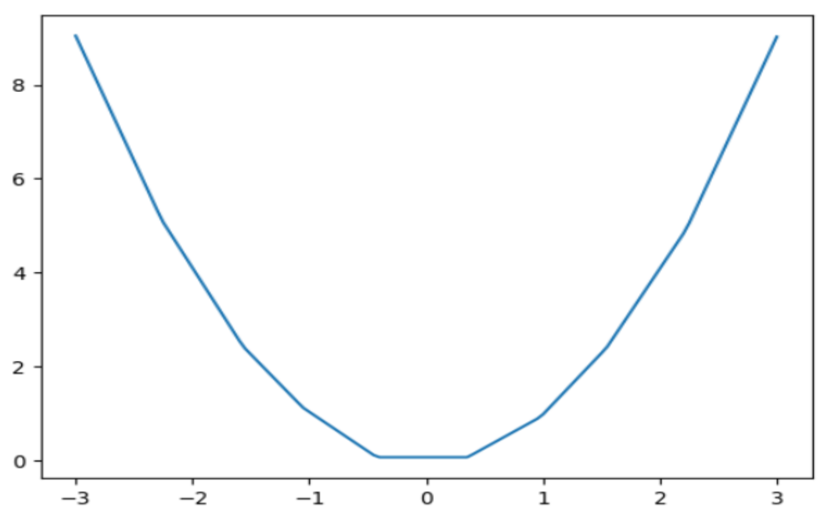
به ازای بازه (15,15-): در واقع میخواهیم ببینیم، مدل ما برای بازه خارج از بازه آموزش میتواند درست عمل کند یا نه



ولی خروجی خوبی نگرفتم، مثلا برای ورودی ۱۵ عدد ۱۸۱ رو خروجی میده و یه جورایی بعد از بازه آموزش به صورت خطی رشد میکنه به جای درجه دو.

حال مقدار متغیر bound را برابر با خود عدد ۳ گذاشتم و خروجی به شکل زیر شد:

بازه (3,3-):



از آزمایشات بالا نتیجه‌ای که میتوان گرفت، این است که اگر مقدار متغیر bound را زیاد بگیریم، مدل ما در اعداد بزرگتر بهتر میتواند generalization انجام دهد ولی باعث میشود برای اعداد کوچکتر مثل بازه درون صورت سوال عملکرد خوبی نداشته باشد.

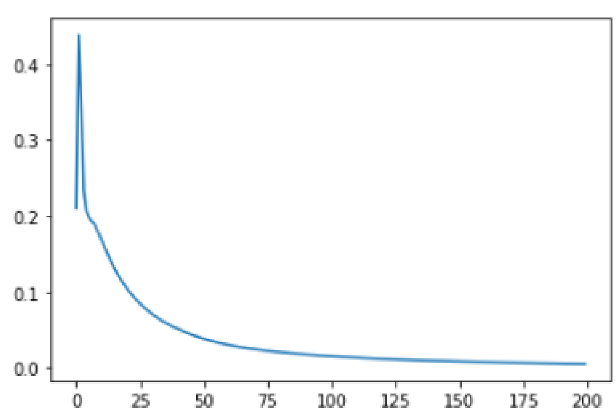
این بررسی‌ها رو با استفاده از keras انجام دادم چون میخواستم اولش یه تحلیل داشته باشم و با keras راحتتر بود، ولی مدل نهایی که زدم، فقط با استفاده از کتابخانه numpy هست. کد هر دو را قرار داده‌ام.

نتایج مربوط به کد Q3_numpy:

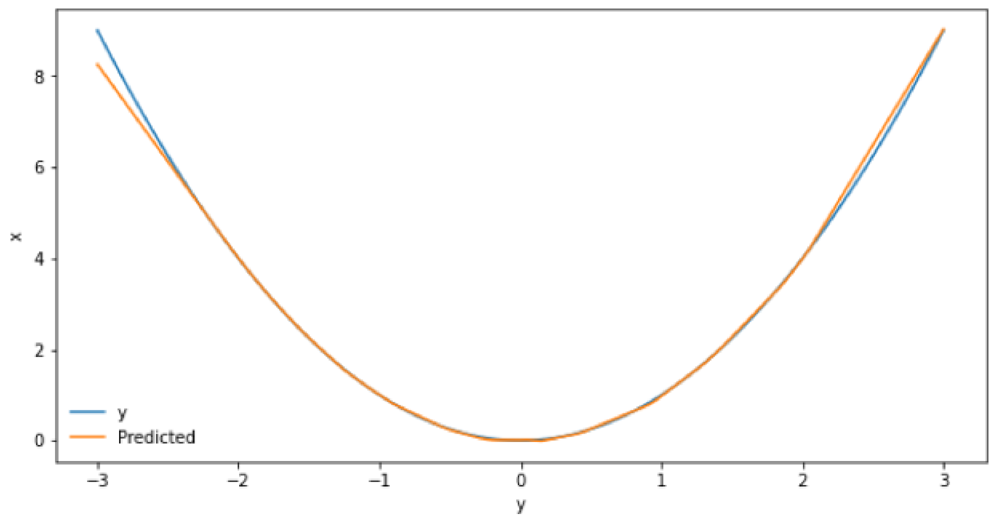
مقادیر ضرر در epochهای مختلف:

```
epoch 0: loss: 0.20990890054177835
epoch 10: loss: 0.1679737390897365
epoch 20: loss: 0.10398117574260332
epoch 30: loss: 0.07028856663946294
epoch 40: loss: 0.05121291306438501
epoch 50: loss: 0.038232912046512454
epoch 60: loss: 0.029909671061593715
epoch 70: loss: 0.024414606516472228
epoch 80: loss: 0.020490262720847575
epoch 90: loss: 0.017563537500930974
epoch 100: loss: 0.015290932182410788
epoch 110: loss: 0.013362437585735146
epoch 120: loss: 0.011804267224867755
epoch 130: loss: 0.010437148521299084
epoch 140: loss: 0.009279425230407996
epoch 150: loss: 0.008340357720543297
epoch 160: loss: 0.007512297737408902
epoch 170: loss: 0.006798034025341707
epoch 180: loss: 0.006191755280009034
epoch 190: loss: 0.005649277003141094
```

نمودار ضرر:



مقادیر پیش‌بینی شده و مقادیر واقعی در بازه (-3,3):



گزارش کد:

۱. **کلاس Sequential**:

- این کلاس برای ایجاد دنباله ای از لایه ها برای مدل شبکه عصبی استفاده می شود.

- `__init__(self, layers, learning_rate)`: کلاس Sequential را با لایه های داده شده و نرخ یادگیری مقدار دهی اولیه می کند.

- `forward_pass(self, inputs)`: یک گذر رو به جلو از تمام لایه های شبکه انجام می دهد.

- `backward_pass`: پس انتشار را از طریق شبکه انجام می دهد.

- `optimize`: تمام پارامترهای شبکه را بهینه می کند.

۲. **کلاس Dense**:

- این کلاس نشان دهنده یک لایه کاملا متصل (متراکم) در شبکه عصبی است.

- `__init__(self, num_previous_layer_units, num_units)`: کلاس Dense را با تعداد واحدهای لایه قبلی و لایه فعلی مقدار دهی اولیه می کند. همچنین وزن ها و سوگیری ها را مقداردهی اولیه می کند.

– `forward_pass(self, inputs)`: با استفاده از فرمول 'O = WX + b' از لایه عبور به جلو انجام می دهد.

– `backward_pass (خود، مشتقات)`: انتشار پس‌انداز را از این لایه انجام می‌دهد و مشتقات را برای وزن‌ها، بایاس‌ها و ورودی‌ها محاسبه می‌کند.

– 'optimize': پارامترهای این لایه را بر اساس آخرین مشتقات محاسبه شده در تابع 'backward_pass' بهینه می کند.

کلاس ReLU:

– این کلاس یک لایه تابع فعالسازی واحد خطی اصلاح شده (ReLU) را در شبکه عصبی نشان می دهد.

– `__init__(self)`: کلاس ReLU را راه اندازی می کند. ویژگی "trainable" روی "False" تنظیم شده است زیرا هیچ پارامتر قابل آموزش در لایه ReLU وجود ندارد.

– `forward_pass(self, inputs)`: یک پاس رو به جلو از لایه انجام می دهد. تابع ReLU را روی ورودی ها اعمال می کند و تمام مقادیر منفی را صفر می کند.

– `backward_pass(خود، مشتقات)`: پس انتشار را از این لایه انجام می دهد و مشتقات ورودی ها را محاسبه می کند. مشتقات ورودی های منفی روی صفر تنظیم می شوند، زیرا گرادیان ReLU برای ورودی های منفی صفر است.

Defining Dataset

```python
## Defining the dataset for y=x^2
x_train = []
y_train = []
bound = 3
step = 0.05
i = -bound
while(i <= bound):
    x_train.append(i)
    y_train.append(i**2)
    i += step
x_train = np.array(x_train)
y_train = np.array(y_train)
```

کد بالا یک مجموعه داده برای تابع y = x^2 تعریف می کند. دو لیست خالی «x_train» و «y_train» را برای ذخیره مقادیر x و y به ترتیب مقداردهی می‌کند. متغیر "bound" روی ۳ و "step" روی ۰٫۰۵ تنظیم شده است. حلقه while برای تولید مقادیر x از -۳ تا ۳ (شامل) در مراحل ۰٫۰۵ استفاده می شود. برای هر مقدار x، مقدار y مربوطه به عنوان مربع x محاسبه می شود. سپس این مقادیر x و y به ترتیب به «x_train» و «y_train» اضافه می‌شوند. در نهایت، «x_train» و «y_train» به آرایه‌های numpy تبدیل می‌شوند. این منجر به یک مجموعه داده می شود که تابع y = x^2 را در بازه [-۳، ۳] نشان می دهد.

Defining Model

```python
# defining the model using defined classes
model = Sequential(
    [
        Dense(1,60),
        ReLU(),
        Dense(60,1)
    ],
    0.01 # learning rate
)
```

این کد، با استفاده از موجودیتهای تعریف شده، یک مدل با یک لایه پنهان و با تعداد نورون ۶۰ ایجاد میکند. در لایه خروجی هم یک نورون قرار میدهیم، زیرا خروجی یک عدد است. مقدار نرخ یادگیری را هم برابر با 0.01 قرار دادم.

```python
## Training the model
num_epochs = 200
losses = []
for i in range(num_epochs):
    avg_loss = 0.0
    for j in range(len(x_train)):
        output = model.forward_pass([[x_train[j],],])
        ## Computing MSE error
        loss = (y_train[j] - output[0][0]) ** 2
        ## Computing the derivation of the loss for the last layer's output
        derivation = (output[0][0] - y_train[j])
        avg_loss += loss
        ## computing gradients of parameters
        model.backward_pass(np.array([derivation,]))
        ## updating weights using calculated gradients
        model.optimize()
    ## computing average loss on the whole dataset in each epoch
    avg_loss /= len(x_train)
    losses.append(avg_loss)
    if i % 10 == 0:
        print(f"epoch {i}: loss: {avg_loss}")
```

کد بالا در حال آموزش یک مدل بر روی مجموعه داده برای تابع y = x^2 است. تعداد دوره های آموزشی روی ۲۰۰ تنظیم شده است. برای هر دوره، مدل یک پاس رو به جلو انجام می دهد و ضرر میانگین مربعات خطا (MSE) را برای هر نمونه در مجموعه آموزشی محاسبه می کند. سپس مشتق از دست دادن با توجه به خروجی آخرین لایه محاسبه می شود. این مشتق برای محاسبه گرادیان پارامترها به روش backward_pass مدل منتقل می شود. سپس روش بهینه سازی برای به روز رسانی وزن های مدل با استفاده از گرادیان های محاسبه شده فراخوانی می شود. میانگین تلفات برای هر دوره محاسبه شده و در لیست تلفات ذخیره می شود. هر ۱۰ دوره، عدد دوره و میانگین تلفات چاپ می شود. این فرآیند به مدل کمک می کند تا تابع y = x^2 را در چندین تکرار بیاموزد.

سوال ۴-

در این مسئله، تنها یک pattern برای معرفی به شبکه داریم و انتظار داریم هر ورودی که به شبکه میدهیم، تنها به همین الگو همگرا شود.

در این شبکه ۶ نورون خواهیم داشت، در نتیجه ماتریس وزنها ابعاد 6*6 خواهد داشت و با فرمول زیر محاسبه خواهد شد:

$$w_{ij} = \sum_{k=1}^{P} x_i^k x_j^k$$

که در اینجا فقط یک الگو داریم یعنی P=1

وزنها به صورت زیر محاسبه میشوند:

$$w_{12} = w_{21} = w_{13} = w_{31} = w_{14} = w_{41} = w_{23} = w_{32} = w_{24} = w_{42} = w_{34} = w_{43} = 1 * 1 = 1$$

$$w_{25} = w_{52} = w_{26} = w_{62} = w_{35} = w_{53} = w_{36} = w_{63} = w_{45} = w_{54} = w_{46} = w_{64} = 1 * 0 = 0$$

$$w_{56} = w_{65} = 0 * 0 = 0$$

و ماتریس وزنها به صورت زیر خواهد بود:

$$\begin{bmatrix} * & 1 & 1 & 1 & 0 & 0 \\ 1 & * & 1 & 1 & 0 & 0 \\ 1 & 1 & * & 1 & 0 & 0 \\ 1 & 1 & 1 & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

حال اگر ورودی 010000 را به این شبکه بدهیم و threshold را صفر بگیریم، یعنی مقادیر بالای صفر را 1 و مقادیر کوچکتر یا مساوی صفر را 0 قرار بدهیم، خواهیم داشت:

نورون ششم	نورون پنجم	نورون چهارم	نورون سوم	نورون دوم	نورون اول	زمان
0	0	0	0	1	0	0
0	0	1	1	0	1	1
0	0	1	1	1	1	2
0	0	1	1	1	1	3

محاسبه مقدار خروجی نورون اول در زمان صفر:

$$[0,1,0,0,0,0] * [* ,1,1,1,0,0] = 1 \; After \; threshold \; will \; be \; 1$$

که علامت * بین دو لیست، همان ضرب نقطه‌ای و بعد از آن جمع مقادیر است. برای باقی نورونها نیز به همین شکل خروجی یا ورودی زمان بعدی محاسبه میشود.

طبق اثبات هاپفیلد، با هر بار آپدیت، به یک استیت با انرژی کمتر می‌رسیم و در اینجا میبینم که پس از دو آپدیت، به الگوی معرفی شده رسیدیم و در آپدیت بعدی، تغییری در استیت بوجود نیامد، بنابراین این استیت یک مینیمم انرژی در شبکه ما میباشد.

سوال ۵-

بررسی SOM:

برای مسئله TSP میتوان از Self Organizing Map به نوعی بهره برد. میتوان یک نقشه‌ی یک بعدی و با تعداد نورون به تعداد شهرهای درون مسئله در نظر گرفت و مراحل زیر را برای آن طی کرد.

۱- SOM را با مجموعه ای از گره ها، هر کدام دارای موقعیت تصادفی در فضای مسئله، مقداردهی می‌کنیم.

۲- برای هر شهر در TSP، نزدیکترین گره را در نقشه پیدا میکنیم و آن را به شهر نزدیک میکنیم که همان بحث تقویت کردن است. همچنین، گره های همسایه در SOM را به شهر نزدیکتر میکنیم، اما با مقدار کمتر، که با تعریف همسایگی میتوان این کار را انجام داد.

۳- مرحله ۲ را برای تعدادی تکرار تکرار میکنیم، به تدریج میزان جابجایی گره ها و اندازه همسایگی گره هایی که جابه جا می شوند را کاهش میدهیم.

۴- پس از آموزش، مسیر گره های SOM از طریق فضای مشکل، راه حلی برای TSP ارائه می دهد.

برای درک بهتر، عکس زیر کمک میکند.

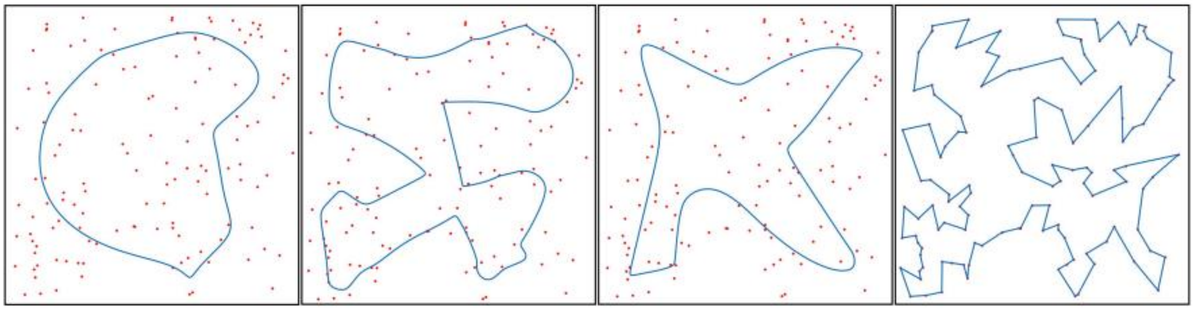


Figure 1. SOM algorithm iteration samples.

در چپ‌ترین عکس، نقاط قرمز شهرها هستند و نقاط آبی، مکان اولیه‌ی گره‌ها هستند که به صورت رندوم مقداردهی شده‌اند، عکسهای راستتر، پس از مقداری آموزش هستند و در نهایت راستترین عکس، پس از اتمام آموزش است و میبینیم که تقریبا هر گره روی یک شهر قرار گرفته است و گره‌های نزدیک به یکدیگر، دارای شهرهای نزدیک به یکدیگر هستند که این بخاطر تاثیر همسایگی در آموزش است. حال هر شهری را که انتخاب کنیم، متعلق به یک گره می‌باشد و کافیست از آن گره شروع کنیم و این حلقه را طی کنیم تا دوباره به همان نقطه مبدا برسیم. امیدواریم این مسیر، کوتاه‌ترین مسیر باشد، ولی باید بدانیم که تضمینی برای آن وجود ندارد.

اگرچه این روش می تواند راه حلی ارائه دهد، اما ممکن است همیشه راه حل بهینه را ارائه نکند. روش‌های دیگری مانند linear programming، branch and bound و الگوریتم‌های ژنتیک نیز معمولاً برای حل TSP استفاده می‌شوند.

رفرنس: https://arxiv.org/pdf/2201.07208.pdf

Enhanced Self-Organizing Map Solution for the Traveling Salesman Problem

بررسی Hopfield Network:

از شبکه هاپفیلد برای حل مسئله TSP در مقالات مختلف استفاده شده است. من از لینک زیر که به طور خلاصه توضیح داده است، استفاده میکنم.

https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_optimization_using_hopfield.htm

۱- ابتدا باید ساختار شبکه خود را مشخص کنیم. برای حل مسئله tsp ابتدا از یک نمایش ماتریسی برای نشان دادن مسیر طی شده برای برگشت به شهر مبدا استفاده میکنیم.

$$M = \begin{bmatrix} A: & 1 & 0 & 0 & 0 \\ B: & 0 & 1 & 0 & 0 \\ C: & 0 & 0 & 1 & 0 \\ D: & 0 & 0 & 0 & 1 \end{bmatrix}$$

برای مثال، در اینجا ۴ شهر A,B,C,D داریم و ماتریس بالا مسیر A-B-C-D-A را نمایش میدهد.

۲- حال تعداد نورونهای شبکه هاپفیلد را برابر با تعداد عناصر این ماتریس میگیریم. پس ۱۶ نورون خواهیم داشت که صفر و یکی هستند.

۳- حال باید تابع انرژی مناسب را به گونه‌ای انتخاب کنیم که مینیمم شدن تابع انرژی، هم‌ارز باشد با مینیمم شدن مسیر طی شده در مسئله tsp یا همان cost function

البته در تعریف تابع انرژی باید یک سری موارد دیگر نیز رعایت شود. مثلا از هر شهر باید یکبار رد شویم، پس درون ماتریس خروجی توسط شبکه، در هر سطر فقط یکبار عدد ۱ می‌آید و بقیه ۰ خواهند بود.

یا مورد دیگر، این است که در هر مرحله، در یک شهر هستیم و نمیتوانیم در هیچ شهر یا در چندین شهر باشیم، بنابراین در ماتریس ذکرشده، در هر ستون فقط یکبار عدد ۱ می‌آید و بقیه ۰ خواهند بود.

ترم تابع انرژی برای شرط مربوط به سطر:

$$\sum_{x=1}^{n} \left(1 - \sum_{j=1}^{n} M_{x,j} \right)^2$$

این عبارت زمانی مینیمم میشود که مقدار درون پرانتز صفر شود و این یعنی مقدار سیگمای داخلی ۱ شود که به معنای این است که فقط یکبار عدد ۱ در آن سطر داشته باشیم.

ترم تابع انرژی برای شرط مربوط به ستون:

$$\sum_{j=1}^{n}\left(1-\sum_{x=1}^{n}M_{x,j}\right)^2$$

توضیحات همانند ترم قبلی

ترم انرژی برای مینیمم کردن هزینه سفر یا cost function:

$$\sum_{i=1}^{n}C_{x,y}M_{x,i}(M_{y,i+1}+M_{y,i-1})$$

این ترم میخواهد بگوید که اگر مقدار cost بین دو شهر، زیاد بود، سعی کند آنها را در ماتریس خروجی، کنار هم قرار ندهد.

تعریف نهایی تابع انرژی برای این مسئله:

$$E=\frac{1}{2}\sum_{i=1}^{n}\sum_{x}\sum_{y\neq x}C_{x,y}M_{x,i}(M_{y,i+1}+M_{y,i-1})+$$
$$\left[\gamma_1\sum_{x}\left(1-\sum_{i}M_{x,i}\right)^2+\gamma_2\sum_{i}\left(1-\sum_{x}M_{x,i}\right)^2\right]$$

که γ1 و γ2 دو ثابت وزن هستند که میتوان با hyper-parameter tuning مقدار مناسب آنها را بدست آورد.

بررسی MLP:

پرسپترون چند لایه (MLP) یک کلاس از شبکه عصبی مصنوعی Feed Forward است. یک MLP حداقل از سه لایه گره تشکیل شده است: یک لایه ورودی، یک لایه پنهان و یک لایه خروجی. در حالی که MLP ها می توانند طیف گسترده ای از توابع را تقریب بزنند و در زمینه های مختلف مورد استفاده قرار می گیرند، معمولاً برای حل مشکل فروشنده دوره گرد (TSP) استفاده نمی شوند.

TSP یک مسئله بهینه سازی ترکیبی است و در حالی که شبکه های عصبی می توانند برای بهینه سازی استفاده شوند، بهترین ابزار برای این مشکل خاص نیستند. دلیل این امر این است که MLP ها معمولاً برای کارهایی مانند طبقه بندی یا رگرسیون استفاده می شوند، جایی که هدف یادگیری mapping از ورودی ها به خروجی ها بر اساس نمونه ورودی-خروجی است. در مقابل، TSP یک مسئله بهینه‌سازی است که در آن هدف یافتن بهترین ترتیب شهرها برای به حداقل رساندن مسافت کل سفر است. این شامل جست و جو در تمام ترتیب بندی های احتمالی شهرها می شود، که نوع دیگری از مشکل با آنچه که MLP ها معمولاً برای آن استفاده می شوند، است.

با این حال، تلاش هایی برای استفاده از تکنیک های یادگیری عمیق، از جمله معماری های شبیه MLP، برای حل TSP صورت گرفته است. برای مثال، یک رویکرد استفاده از یادگیری تقویتی عمیق (DRL) با تطبیق رویکردهای اخیری است که برای TSPهای معمولی به خوبی کار می‌کنند. این شامل استفاده از مدل‌های گراف مبتنی بر لایه‌های Multi-Head Attention(MHA) است که نوعی معماری شبیه MLP است. رویکرد دیگر طراحی یک راه حل شبکه عصبی است که فروشندگان، شهرها و انبارها را به عنوان سه مجموعه مختلف از کاردینالیته های متفاوت در نظر می گیرد. این شامل ترکیب عناصر از معماری های اخیر است که برای مجموعه ها توسعه داده شده اند، و همچنین عناصر شبکه های گراف(GAN).

در نتیجه، در حالی که MLP ها معمولا برای حل TSP استفاده نمی شوند، انواعی از معماری های مشابه MLP در ترکیب با تکنیک های دیگر مانند یادگیری تقویتی یا شبکه های گراف برای مقابله با مشکل استفاده شده است.