به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین اول

نام و نام خانوادگی	حسام اسداله‌زاده – مسعود طهماسبی
شماره دانشجویی	810198346 – 810198429
تاریخ ارسال گزارش	۱۴۰۱.۰۸.۰۸

فهرست

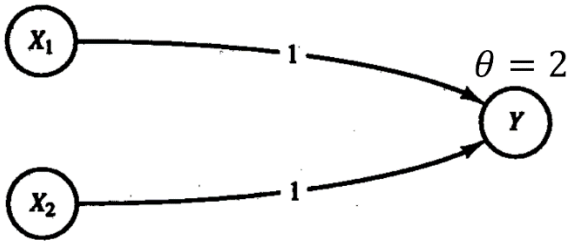
شکل‌ها

جدول‌ها

پاسخ 1. شبکه عصبی McCulloch-Pitts

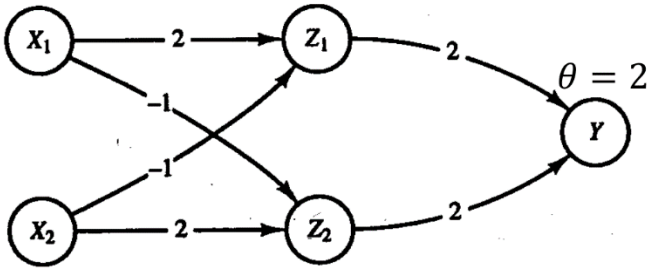
۱-۱. ضرب کننده باینری دو بیتی

الف)



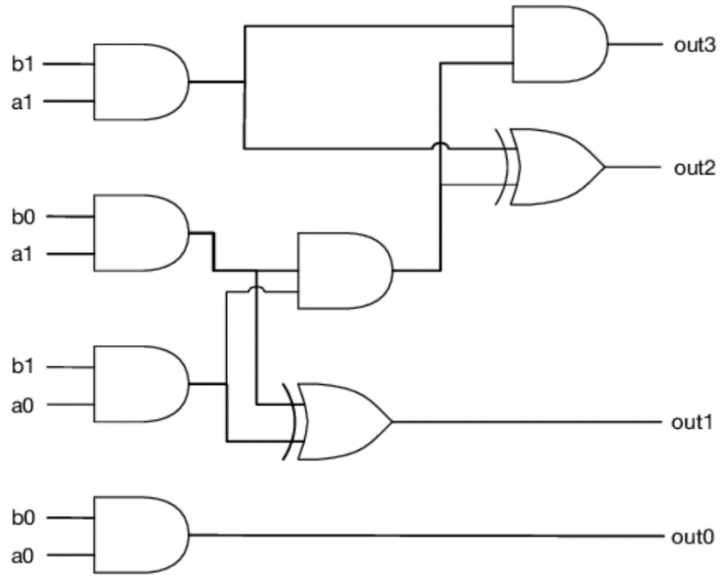
AND

شکل**1** . شبکه **AND**



XOR

شکل**2** . شبکه **XOR**



شکل**3** . مدار ضرب‌کننده دو بیت در دو بیت

در شکل1 و شکل2 شبکه‌های مربوط به gateهای AND و XOR نشان داده شده‌اند. شکل3 نیز مدار ضرب‌کننده دو بیت در دو بیت را نمایش می‌دهد که با استفاده از دو gate نمایش داده شده ساخته شده است.

نکته‌ی قابل توجه در پیاده‌سازی این شبکه آن است که به دلیل ارجحیت کوچک بودن threshold نسبت به تعداد نورون‌ها، از gateها با تعداد ورودی‌های بیشتر استفاده نشده است. به طور مثال می‌توانستیم خروجی out3 را با یک AND با ۴ ورودی پیاده‌سازی کنیم که نیازمند $\theta = 4$ می‌باشد. ولی ترجیح ما استفاده از $\theta = 2$ برای تمامی نورون‌های خروجی شبکه بود.

ب)

```python
# Mcculloch & Pitts Neuron - XOR
def XOR(x1 , x2):
  threshold = 2
  Z = np.array([0, 0])
  Z[0] = 2*x1 - x2
  Z[1] = 2*x2 - x1
  y1 , y2 , y = 0, 0, 0
  if Z[0] >= 2:
      y1 = 1
  if Z[1] >= 2:
      y2 = 1
  y =  2*y1 + 2*y2
  return 1 if y >= threshold else 0

# Mcculloch & Pitts Neuron - AND
def AND(x1, x2):
  threshold = 2
  y = x1 + x2
  return 1 if y >= threshold else 0
```

```
A  *  B -> result
00 * 00  = 0000
00 * 01  = 0000
00 * 10  = 0000
00 * 11  = 0000
01 * 00  = 0000
01 * 01  = 0001
01 * 10  = 0010
01 * 11  = 0011
10 * 00  = 0000
10 * 01  = 0010
10 * 10  = 0100
10 * 11  = 0110
11 * 00  = 0000
11 * 01  = 0011
11 * 10  = 0110
11 * 11  = 1001
```

شکل**4** . پیاده‌سازی نورون‌های **AND** و **XOR** و نتایج ضرب‌کننده

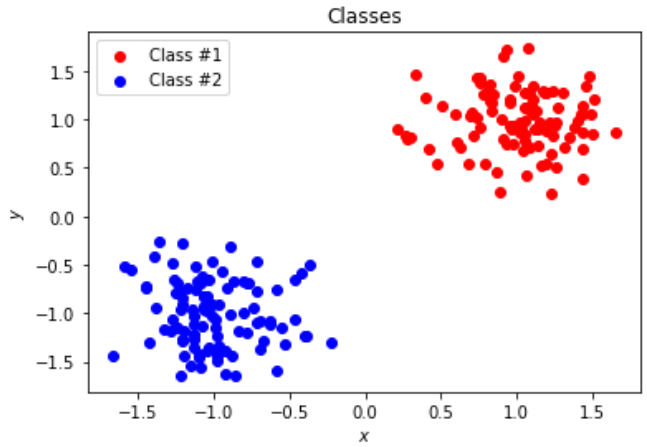
پاسخ ۲ – AdaLine & MadaLine

۲-۱. AdaLine

الف)

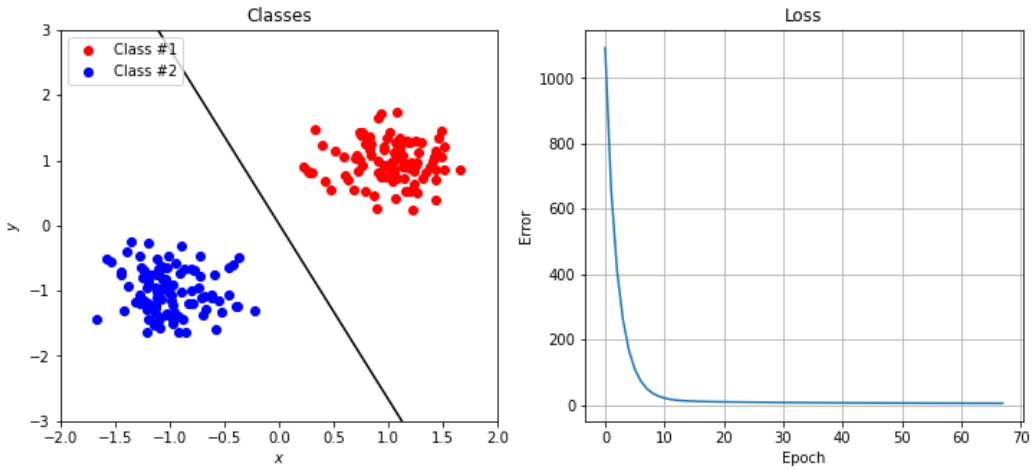
```
x1 = np.random.normal(1, 0.3, 100)
y1 = np.random.normal(1, 0.3, 100)

x2 = np.random.normal(-1, 0.3, 100)
y2 = np.random.normal(-1, 0.3, 100)
```



شکل5 . نمودار پراکندگی داده‌های بخش اول

ب)



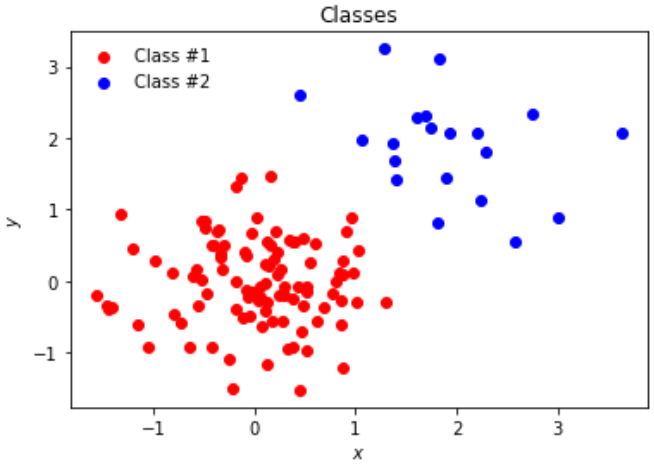
خط رسم‌شده به خوبی داده‌های دو دسته را از یکدیگر جدا کرده و margin مناسبی نیز از داده‌ها دارد. علت این موضوع نیز تقارن بین تعداد داده‌های دو دسته می‌باشد. چراکه می‌دانیم مادامی که دو دسته داده

از نظر تعداد تقارن داشته باشند[1] و به صورت خطی جداپذیر باشند، مدل AdaLine به خوبی می‌تواند دو دسته را از هم جدا کند.

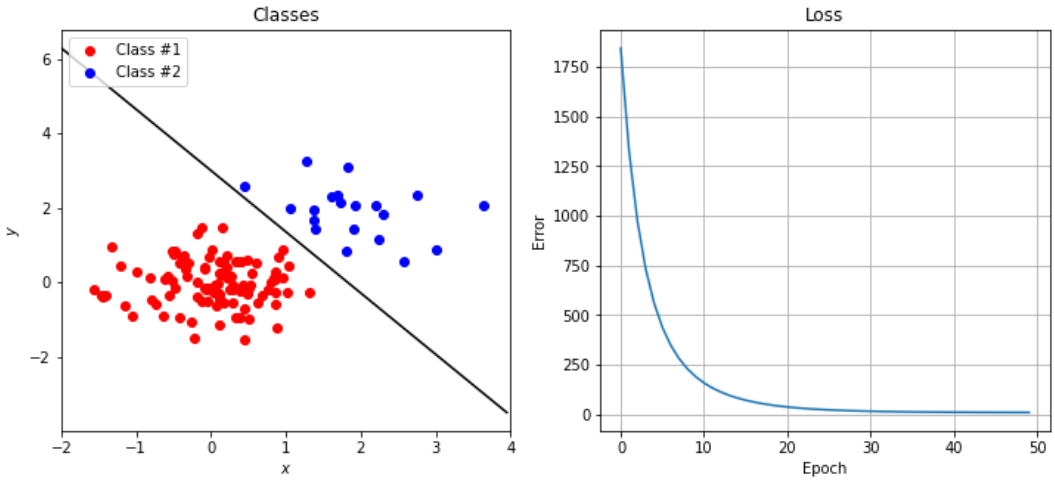
ج)

```
x1 = np.random.normal(0, 0.6, 100)
y1 = np.random.normal(0, 0.6, 100)

x2 = np.random.normal(2, 0.8, 20)
y2 = np.random.normal(2, 0.8, 20)
```



شکل **6** . نمودار پراکندگی داده‌های بخش دوم

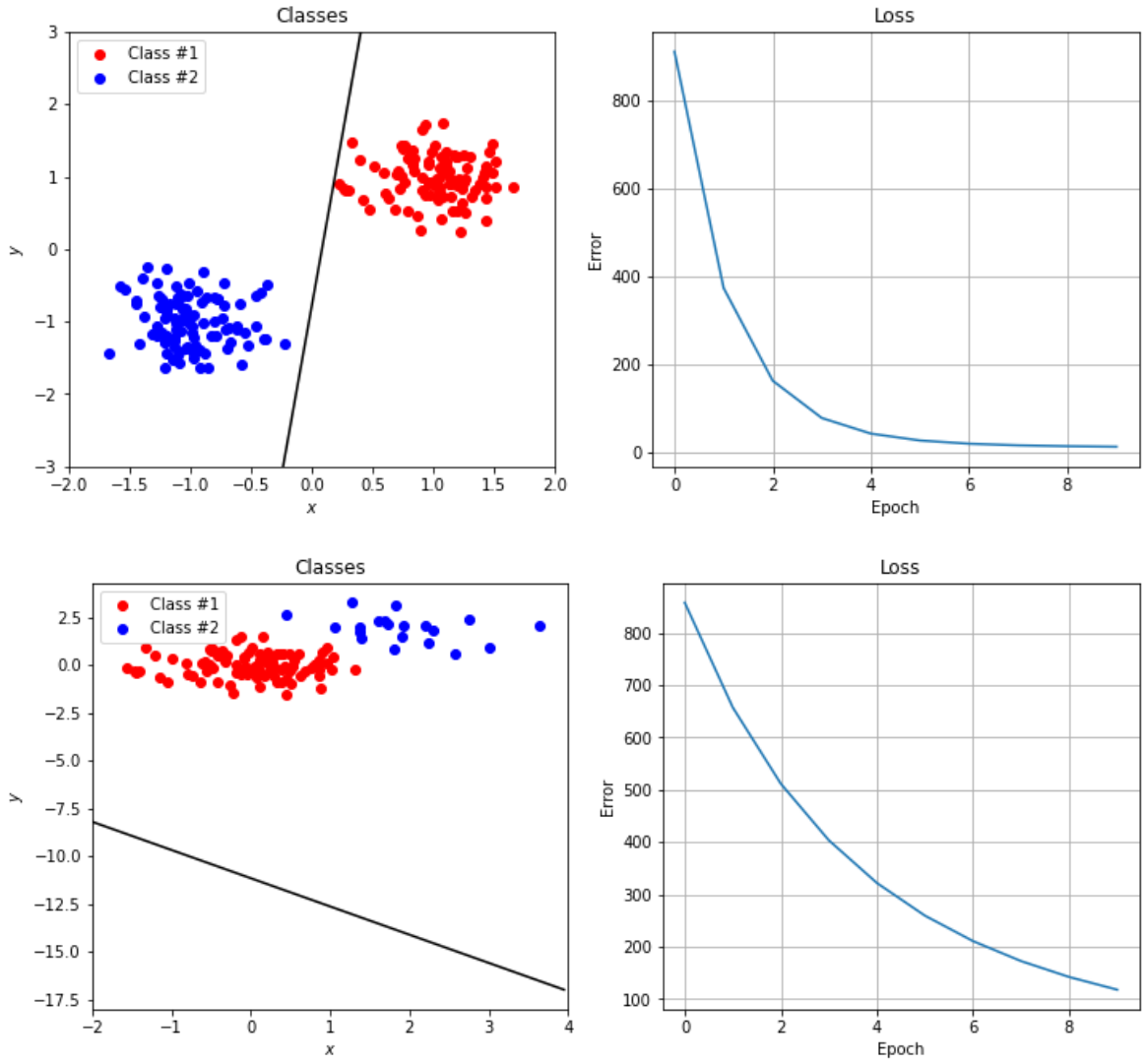


همانگونه که انتظار داشتیم به دلیل عدم تقارن بین تعداد داده‌های دو دسته، خط به دست آمده از خروجی مدل AdaLine، هرچند توانسته دو دسته را از هم تفکیک کند ولی margin مناسبی ندارد.

مدل AdaLine استفاده شده در پیاده‌سازی ما، از Stop Condition بر اساس حداکثر مقدار خطا استفاده می‌کند. از آنجایی که مدل برای بخش الف و بخش ب در تعداد epoch های متفاوتی به threshold

[1] Balanced

خطای مورد قبول رسیده، بهتر است برای مقایسه‌ی دو حالت، شرایط یکسانی را ایجاد کنیم؛ به همین منظور، Stop Condition را این بار بر اساس تعداد epoch تعیین می‌کنیم تا خروجی مدل در دو حالت را مقایسه کنیم. نتایج زیر، خروجی دو حالت را به ازای $epochs = 10$ نشان می‌دهد:



همانطور که مشاهده می‌شود، به ازای تعداد ایپاک یکسان، خروجی مدل برای داده‌های متقارن، توانسته دو دسته را از هم جدا کند ولی برای داده‌های نامتقارن نتوانسته این کار را انجام دهد.

MadaLine (۲-۲

الف)

هر دو روش MRI و MRII تشکیل یافته از تعدادی نورون AdaLine هستند و هر دو شباهت بالایی به یکدیگر دارند و تفاوت کمی با یکدیگر دارند. در MRI تنها وزن‌های لایه‌ی پنهان تغییر می‌کند و وزن‌های لایه‌ی آخر ثابت هستند و لایه‌ی آخر در واقع به نوعی عمل OR را انجام می‌دهد ولی در MRII وزن تمام لایه‌ها تغییر می‌کند. الگوریتمی که ما در این سوال استفاده می‌کنیم، الگوریتم MRI است. طبق متن کتاب، زمانی وزن‌ها و بایاس‌ها آپدیت می‌شوند که خطا رخ داده باشد، یعنی اگر خروجی مدل 1- باشد ولی target برابر 1 باشد، از آنجا که لایه‌ی آخر عمل OR را انجام می‌دهد، پس 1- شدن خروجی به معنی این است که خروجی تمام نورون‌های لایه‌ی مخفی، 1- شده است. در این صورت برای رسیدن به خروجی مطلوب، وزن‌های نورونی که از همه بیشتر به 0 نزدیک است را آپدیت می‌کنیم تا خروجی را به target برسانیم. همچنین اگر خروجی مدل 1 باشد ولی target برابر 1- باشد، باید وزن‌های تمام نورون‌های لایه‌ی مخفی با خروجی مثبت را آپدیت کنیم تا مقدار خروجی تمام این نورون‌ها کمتر از صفر شود و در نتیجه، OR خروجی نورون‌ها به 1- برسد. الگوریتم فوق به‌صورت زیر پیاده‌سازی می‌شود:

قدم صفر: در ابتدا مقادیر وزن‌ها و بایاس‌ها را به صورت رندوم اعداد کوچکی در نظر می‌گیریم.

قدم اول: stop condition را تعریف کرده و تا وقتی این شرط برقرار نشده قدم های 2 تا 8 را تکرار می‌کنیم.

قدم‌های بعدی در شکل صفحه‌ی بعد آورده شده است.

Step 2. For each bipolar training pair, s:t, do Steps 3–7.

Step 3. Set activations of input units:

$$x_i = s_i.$$

Step 4. Compute net input to each hidden ADALINE unit:

$$z\_in_1 = b_1 + x_1 w_{11} + x_2 w_{21},$$

$$z\_in_2 = b_2 + x_1 w_{12} + x_2 w_{22}.$$

Step 5. Determine output of each hidden ADALINE unit:

$$z_1 = f(z\_in_1),$$

$$z_2 = f(z\_in_2).$$

Step 6. Determine output of net:

$$y\_in = b_3 + z_1 v_1 + z_2 v_2;$$

$$y = f(y\_in).$$

Step 7. Determine error and update weights:
If $t = y$, no weight updates are performed.
Otherwise:
If $t = 1$, then update weights on Z_J, the unit whose net input is closest to 0,

$$b_J(\text{new}) = b_J(\text{old}) + \alpha(1 - z\_in_J),$$

$$w_{iJ}(\text{new}) = w_{iJ}(\text{old}) + \alpha(1 - z\_in_J)x_i;$$

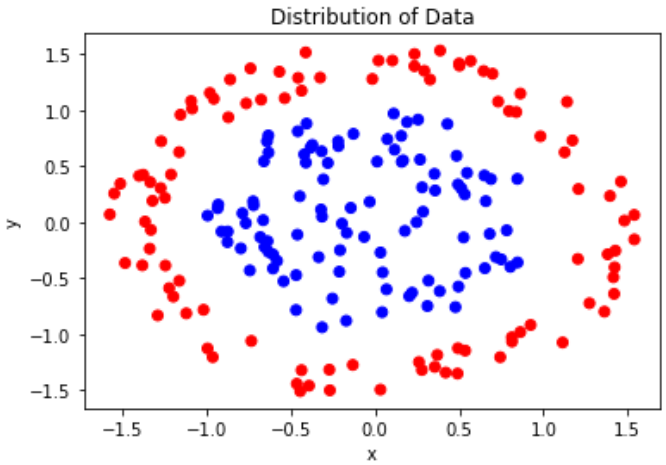
If $t = -1$, then update weights on all units Z_k that have positive net input,

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z\_in_k),$$

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z\_in_k)x_i.$$

Step 8. Test stopping condition.
If weight changes have stopped (or reached an acceptable level), or if a specified maximum number of weight update iterations (Step 2) have been performed, then stop; otherwise continue.

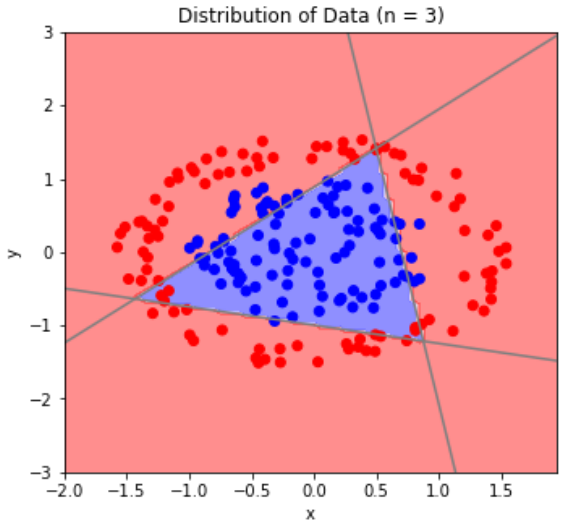
ب)

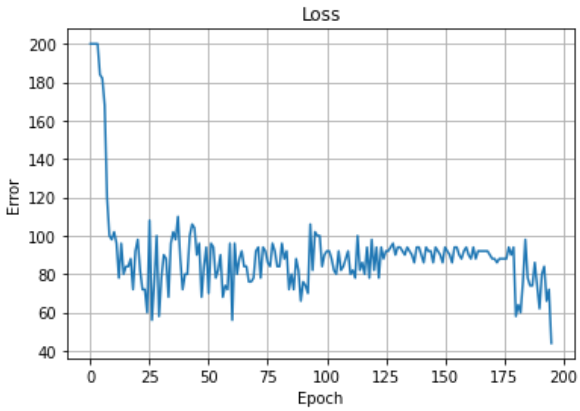


شکل 7 . پراکندگی داده‌های **MadaLine**

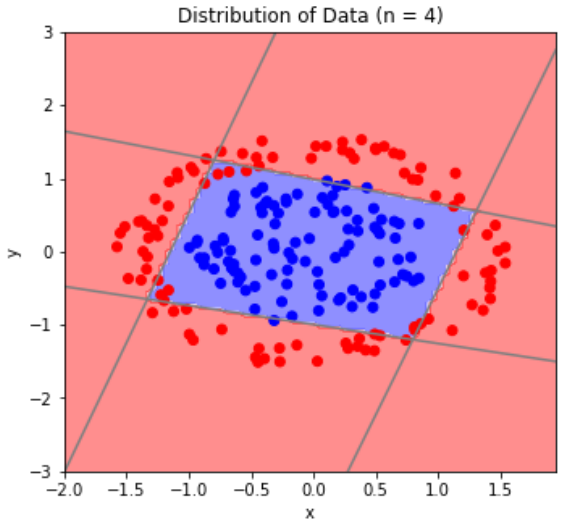
حال خروجی مدل MadaLine به ازای ۳، ۴ و ۸ نورون در لایه‌ی مخفی را نمایش می‌دهیم:

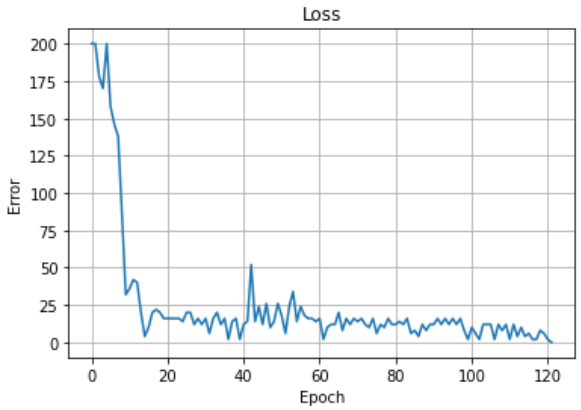
* ۳ نورون در لایه‌ی مخفی:



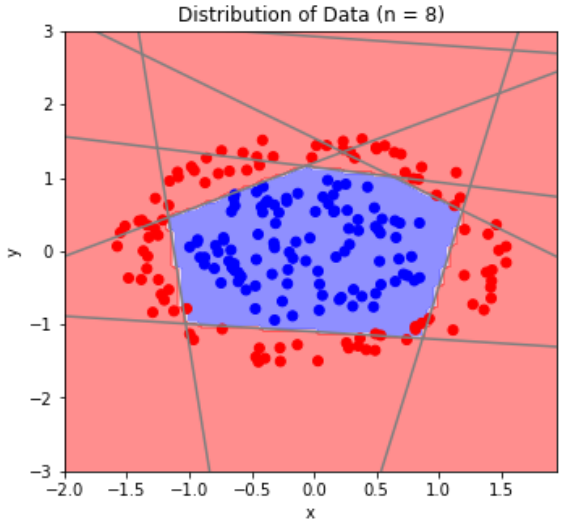


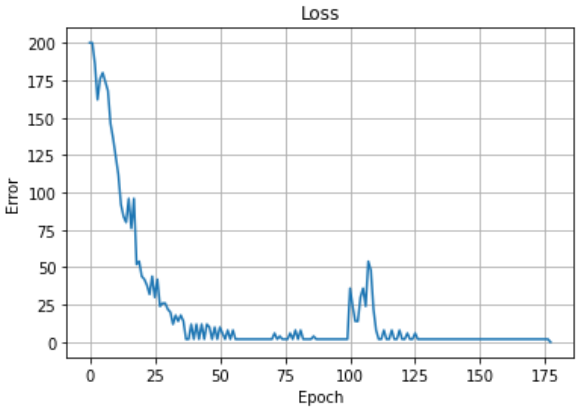
* ۴ نورون در لایه‌ی مخفی:





*** ۸ نورون در لایه‌ی مخفی:**





ج)

همانطور که در نمودارها نیز مشخص است، تعداد سه نورون توانایی جداسازی دو دسته داده را به خوبی ندارد و با وجود تعداد epoch بالایی که محاسبات انجام شده، همچنان خطای بالایی دارد و به دقت مناسبی دست نیافته است.

تعداد ۴ نورون عملکرد بهتری داشته و توانسته است با تعداد epoch نسبتا پایینی خطای کمتری داشته باشد و به دقت بالاتری برسد و توانسته بهتر دو دسته داده را جدا کند.

تعداد ۸ نورون نسبت به حالت‌های قبلی بهتر دو دسته را تفکیک کرده است. ۸ نورون توانایی رسم ۸ خط را دارند اما چون شبکه توانسته با یک شش ضلعی stop condition را ارضا کند، وزن‌های دو خط دیگر را آپدیت نکرده و از دو خط دیگر استفاده نکرده است. این حالت هر چند تعداد epoch بیشتری محاسبات را انجام داده ولی در عوض به دقت بالاتری رسیده است. تعداد epoch بیشتر نیز از نظر منطقی نیز درست بنظر می‌رسد چرا که در این حالت شبکه باید ضرایب ۸ خط را محاسبه کند و منطقی است که برای اینکار نیازمند محاسبات بیشتری نسبت به حالت‌های قبل باشد.

پاسخ ۳ – Restricted Boltzmann Machine

۳-۱. سیستم توصیه‌گر

.

پاسخ ۴ – MLP

۴-۱. Multi-Layer Perceptron

(A

```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21613 non-null  int64
 1   date           21613 non-null  object
 2   price          21613 non-null  float64
 3   bedrooms       21613 non-null  int64
 4   bathrooms      21613 non-null  float64
 5   sqft_living    21613 non-null  int64
 6   sqft_lot       21613 non-null  int64
 7   floors         21613 non-null  float64
 8   waterfront     21613 non-null  int64
 9   view           21613 non-null  int64
 10  condition      21613 non-null  int64
 11  grade          21613 non-null  int64
 12  sqft_above     21613 non-null  int64
 13  sqft_basement  21613 non-null  int64
 14  yr_built       21613 non-null  int64
 15  yr_renovated   21613 non-null  int64
 16  zipcode        21613 non-null  int64
 17  lat            21613 non-null  float64
 18  long           21613 non-null  float64
 19  sqft_living15  21613 non-null  int64
 20  sqft_lot15     21613 non-null  int64
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

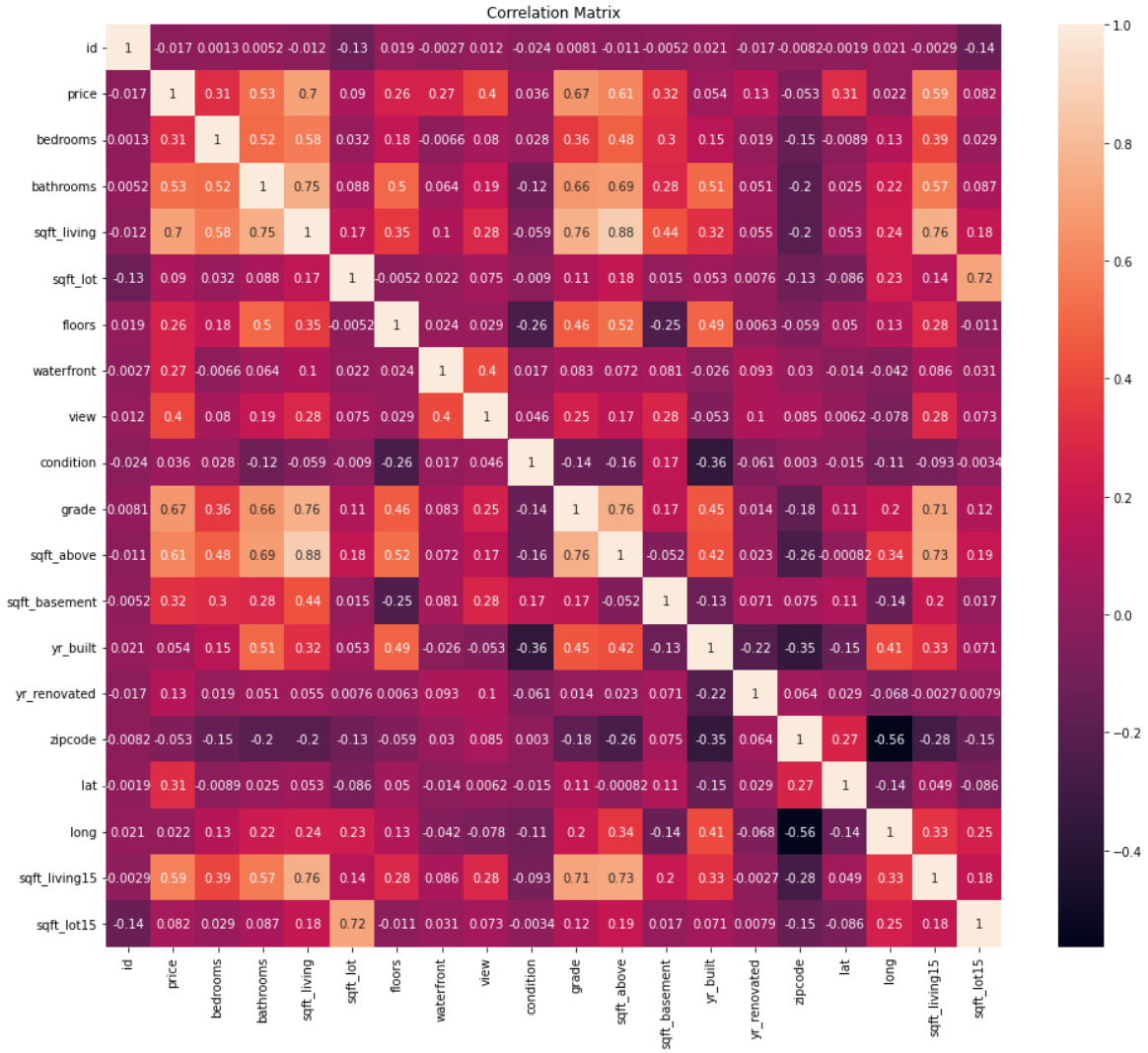
شکل **8** . خروجی **df.info()** برای دیتاست **houses**

(B

```
1 df.isna().sum()
```

```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

شکل **9** . تعداد داده‌های **Nan** برحسب هر ستون



شکل **10** . ماتریس همبستگی ویژگی‌ها

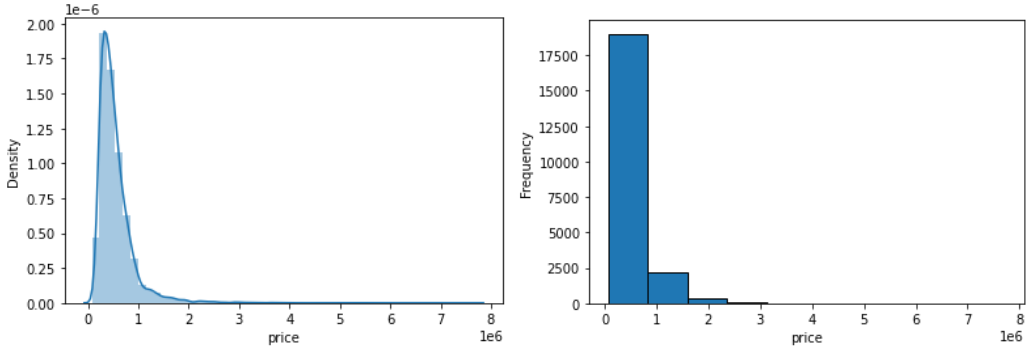
C) فیچر sqft_living بیشترین Correlation را با قیمت خانه‌ها دارد.

```
1 df.corr()['price'].sort_values(ascending=False)
```

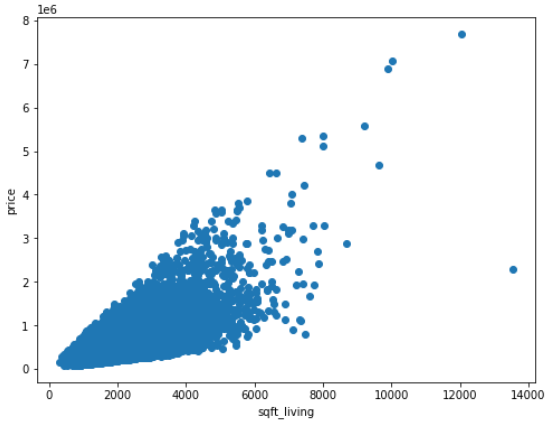
```
price            1.000000
sqft_living      0.702035
grade            0.667434
sqft_above       0.605567
sqft_living15    0.585379
bathrooms        0.525138
view             0.397293
sqft_basement    0.323816
bedrooms         0.308350
lat              0.307003
waterfront       0.266369
floors           0.256794
yr_renovated     0.126434
sqft_lot         0.089661
sqft_lot15       0.082447
yr_built         0.054012
condition        0.036362
long             0.021626
id              -0.016762
zipcode         -0.053203
Name: price, dtype: float64
```

شکل **11** . ترتیب همبستگی ویژگی‌های مختلف با قیمت خانه‌ها

D) نمودار توزیع قیمت و نمودار قیمت:



نمودار فیچری که Correlation زیادی با قیمت دارد:



E)

```
df['year'] = df['date'].str[:4]
df['year'] = df['year'].astype('int64')
df['month'] = df['date'].str[4:6]
df['month'] = df['month'].astype('int64')
df = df.drop(['date'], axis=1)
```

year	month
2014	10
2014	12
2015	2
2014	12
2015	2

شکل**12** . تبدیل ستون **date** به دو ستون **year** و **month**

F)

```
x_train, x_test, y_train, y_test = train_test_split(
                        df.drop(['price', 'id'], axis=1), df[['price']],
                        test_size=0.2, random_state=200)
```

شکل**13** . تقسیم داده‌ها به دو قسمت **train/test**

نکته‌ی قابل توجه در این قسمت آن است که ویژگی id صرفا برای برچسب‌گذاری و تفکیک داده‌ها استفاده می‌شود و ویژگی مناسبی برای پیش‌بینی قیمت خانه نیست و در نتیجه هنگام تفکیک داده‌های train و test، این ستون drop می‌شود و ۲۰ درصد داده‌ها به عنوان test جدا می‌شود.

(G

```
x_scaler = MinMaxScaler()
x_train = x_scaler.fit_transform(x_train)
x_test = x_scaler.transform(x_test)
```

شکل **14** . استفاده از **MinMaxScaler**

لازم به ذکر است که برای جلوگیری از data leakage، این scaler فقط روی داده‌های train فیت شده و داده‌های test را فقط transform می‌کند.

H) از یک MLP با ۴ لایه مخفی استفاده شده که به ترتیب ۱۲۸، ۶۴، ۳۲ و ۱۶ نورون دارند. همچنین در تمامی لایه‌ها از تابع فعالسازی ReLU استفاده شده است.

I) ساده‌ترین الگوریتم بهینه‌سازی، الگوریتم گرادیان کاهشی است که در روش mini-batch به صورت زیر وزن‌ها را به‌روزرسانی می‌کند:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$$

ما در این مسئله از دو الگوریتم بهینه‌سازی Adam و RMSProp استفاده می‌کنیم. الگوریتم Adam یک الگوریتم بهینه‌سازی است که می‌توان از آن به جای روش گرادیان کاهشی تصادفی[1] برای به‌روزرسانی وزن‌های شبکه استفاده کرد. الگوریتم آدام را می‌توان به عنوان ترکیبی از RMSprop و گرادیان کاهشی تصادفی با تکانه[2] در نظر گرفت. در RMSProp[3] نرخ‌های یادگیری هر پارامتر حفظ می‌شوند. این نرخ‌ها بر اساس میانگین مقادیر اخیر گرادیان‌های مربوط به وزن‌ها تطبیق داده شده‌اند. این بدان معنا است که الگوریتم RMSProp در مسائل برخط و ناپایدار (مانند مسائل noisy) به خوبی کار می‌کند. الگوریتم‌های RMSprop و Adam الگوریتم‌های بسیار مشابهی هستند که در شرایط مشابه، به خوبی عمل می‌کنند. اصلاح اختلاف معیار آدام به آن کمک می‌کند تا این الگوریتم با کم شدن تراکم گرادیان‌ها در پایان بهینه‌سازی کمی بهتر از الگوریتم RMSprop عمل کند.

حال به مقایسه دو تابع هزینه (زیان) مختلف برای مسئله رگرسیون خود می‌پردازیم:

[1] Stochastic Gradient Descent
[2] Momentum
[3] Root Mean Square Propagation

تابع هزینه میانگین مربعات (MSE[1] یا L2-Loss)

یکی از معروف‌ترین و معمول‌ترین توابع هزینه در تحلیل رگرسیونی، میانگین مربعات خطا است که به اختصار MSE نامیده می‌شود. این تابع هزینه، میانگین مربعات فاصله بین مقدار پیش‌بینی و واقعی را محاسبه می‌کند. شیوه و نحوه محاسبه آن در زیر دیده می‌شود.

$$MSE = \frac{\sum(y_i - \hat{y}_i)^2}{n}$$

تابع هزینه میانگین قدرمطلق خطا (MAE[2] یا L1-Loss):

این تابع هزینه، به مانند MSE از فاصله بین مقدار پیش‌بینی و واقعی به عنوان معیار استفاده کرده ولی جهت این تفاضل را در نظر نمی‌گیرد. بنابراین در محاسبه خطا MAE فقط میزان فاصله و نه جهت فاصله به کار می‌رود.

$$MAE = \frac{\sum|y_i - \hat{y}_i|}{n}$$

معمولا محاسبات و حل معادلات با مرتبه یا توان ۲ نسبت به توابعی که در آن از قدرمطلق استفاده شده، ساده‌تر است. به این ترتیب شاید به نظر برسد که استفاده از تابع هزینه MSE ارجح باشد. ولی تابع هزینه قدرمطلق نسبت به وجود «داده‌های پرت» (Outlier) مقاوم‌تر است.

توابع هزینه L1 نسبت به داده‌های پرت مقاوم بوده ولی مشتق آن‌ها پیوسته نیست. در نتیجه برای پیدا کردن کمینه به راحتی از روش‌های مشتق‌گیری نمی‌توان استفاده کرد. در مقابل توابع هزینه L2 نسبت به داده‌های پرت حساس بوده ولی محاسبات مربوط به پیدا کردن نقاط کمینه آن ساده‌تر است و به کمک روش‌های تحلیلی به دست می‌آیند.

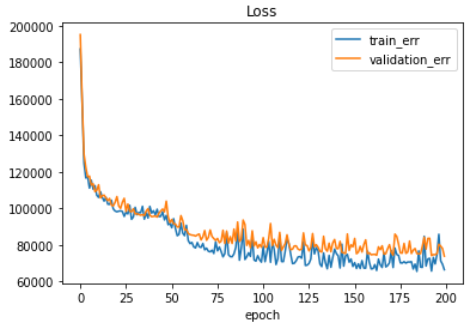
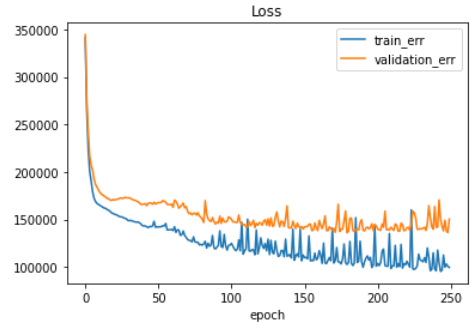
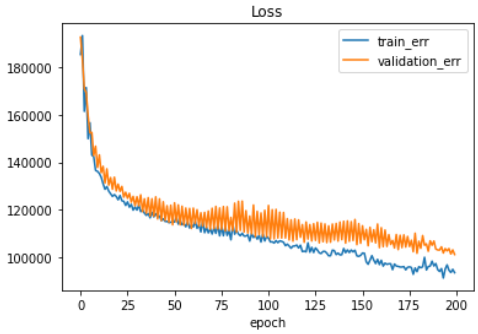
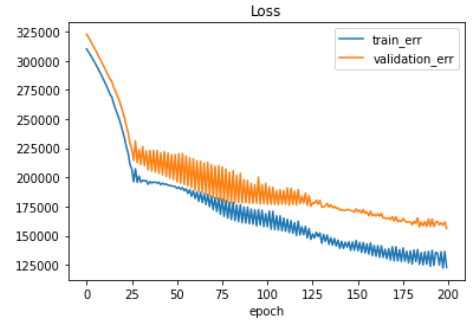
علاوه بر توابع هزینه فوق، تابع هزینه دیگری به نام Smooth L1 نیز داریم که نسبت به L2 کمتر تحت تاثیر داده‌های پرت است. همچنین، برعکس تابع هزینه L1، مشتق‌پذیر بوده و کمینه‌سازی آن به راحتی امکان پذیر است.

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & |y - \hat{y}| > \delta \end{cases}$$

[1] Mean Square Error
[2] Mean Absolute Error

(J

جدول 1. نمودارهای تغییرات **Loss** برای **Optimizer**ها و **Loss**های مختلف

Optimizer\ Loss Function	Smooth L1 Loss	L2 Loss
Adam		
RMSProp		

(K

با استفاده از الگوریتم بهینه‌سازی Adam و تابع هزینه SmoothL1Loss نتایج زیر به دست می‌آید:

```
Predicted             Y_true          diff
tensor(453602.3125)   tensor(469000.) tensor(15397.6875)
tensor(632424.5000)   tensor(527000.) tensor(-105424.5000)
tensor(287440.9688)   tensor(249000.) tensor(-38440.9688)
tensor(622859.8125)   tensor(682500.) tensor(59640.1875)
tensor(1215048.6250)  tensor(1005000.)        tensor(-210048.6250)
tensor(674790.6250)   tensor(645000.) tensor(-29790.6250)
tensor(421012.4375)   tensor(420000.) tensor(-1012.4375)
tensor(341734.7500)   tensor(304000.) tensor(-37734.7500)
tensor(387991.9062)   tensor(390000.) tensor(2008.0938)
tensor(350810.4375)   tensor(367000.) tensor(16189.5625)
```

همانگونه که مشخص است، در بازه‌ای که تعداد داده‌های train در آن زیاد بوده است، مدل دقت پیش‌بینی مناسبی دارد. اما در بازه‌هایی که تعداد داده‌ی آموزش در آن کم بوده، مدل دقت پایین‌تری دارد. دلیل این موضوع آن است که پراکندگی قیمت خانه‌های موجود در dataset بسیار زیاد است اما اکثریت داده‌ها در یک بازه‌ی کوچک‌تری قرار دارند. همین موضوع باعث می‌شود داده‌هایی که خارج این بازه‌ی کوچک‌تر قرار دارند، عملا داده‌ی دورافتاده (پرت) محسوب شوند.