به نام خدا

1928
K. N. Toosi University of Technology

دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده برق

مبانی سیستم های هوشمند

گزارش مینی پروژه شماره دو

[ابوالفضل ولی زاده لاکه]

[40010273]

استاد : آقای دکتر مهدی علیاری

دی 1403

فهرست مطالب

عنوان شماره صفحه

سوال اول بخش 1

اگر در یک مسئله‌ی طبقه‌بندی دوکلاسه، دو لایه‌ی انتهایی شبکه به ترتیب از فعال‌ساز **ReLU** و سپس **سیگموید** استفاده کنند، ممکن است مشکلاتی در عملکرد شبکه به وجود آید.

Relu:

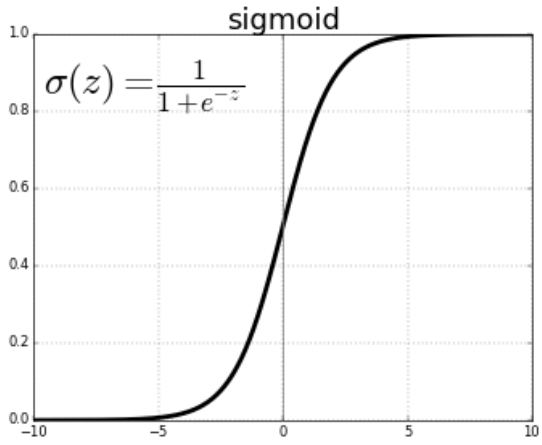
ReLU یک فعال‌ساز است که خروجی آن برابر است با $\max(0,x)$ این به این معنی است که اگر ورودی به این لایه منفی باشد، خروجی آن صفر خواهد شد. در غیر این صورت، خروجی برابر با مقدار ورودی باقی می‌ماند. ایراد این تابع فعال ساز این است که اگر ورودی به این لایه منفی باشد، سیگنال به صفر تبدیل شده و در لایه‌ی بعدی تأثیرگذار نخواهد بود. این ممکن است منجر به از دست دادن اطلاعات مفید شود.

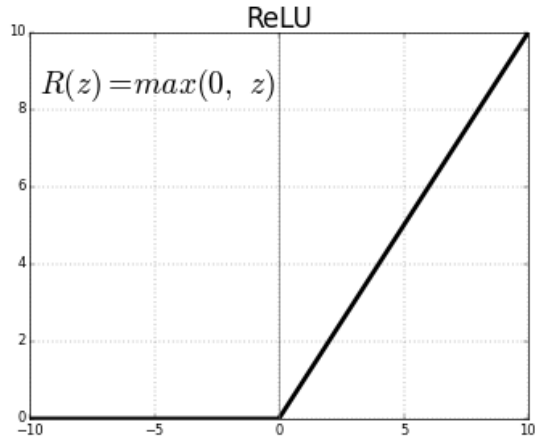
Sigmoid:

سیگموید خروجی خود را به محدوده [0,1] فشرده می‌کند و برای تولید احتمال مناسب است.

این فعال‌ساز معمولاً برای مسائل طبقه‌بندی دوکلاسه استفاده می‌شود.

Relu در لایه‌ی قبل از سیگموید ممکن است باعث صفر شدن بسیاری از ورودی‌های سیگموید شود. این باعث می‌شود که خروجی سیگموید برای این موارد نزدیک به 0.5 باشد. این رفتار ممکن است بر دقت شبکه در یادگیری **تأثیر منفی** بگذارد، زیرا مقادیر صفرشده‌ی ورودی به سیگموید می‌توانند باعث گمراهی در فرآیند به‌روزرسانی وزن‌ها شوند.



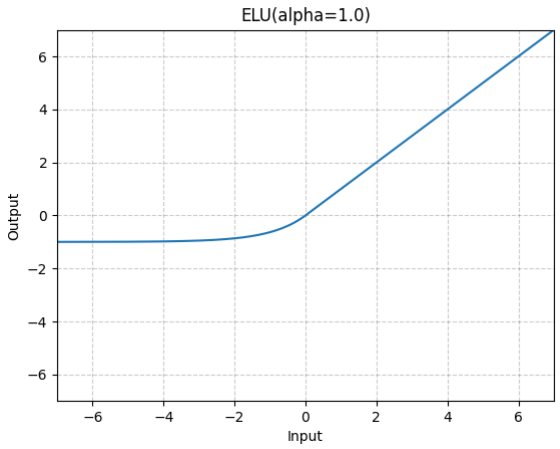


بخش 2

محاسبه گرادیان تابع جایگرین Relu به صورت زیر خواهد بود:

$$f'(x) = \begin{cases} 1 & x \geq 0 \\ \alpha e^x & x < 0 \end{cases}$$

این تابع ELU نام دارد و یک نسخه پیشرفته‌تر از ReLU است که مشکل خروجی صفر به ازای مقدار منفی را کاهش می‌دهد.



بخش 3

تعریف کلاس نورون مک‌کالوک-پیتس:

کلاس McCulloch_Pitts_neuron یک نورون ساده پیاده‌سازی می‌کند.

این نورون ورودی‌ها را می‌گیرد، حاصل ضرب داخلی بردار وزن‌ها و بردار ورودی را محاسبه می‌کند، و سپس خروجی باینری (0 یا 1) تولید می‌کند:

تعریف خطوط مثلث:

ناحیه محصور شده به صورت نامساوی‌های خطی تعریف شده‌اند که خطوط مثلث را تشکیل می دهند:

ناحیه $y > 0$

ناحیه $y < 2x - 2$

ناحیه $y < -2x + 6$

هر خط با یک نورون مدل‌سازی شده است که به عنوان یک تابع آستانه عمل می‌کند.

```python
#import libraries
import numpy as np
import matplotlib.pyplot as plt

# Define the McCulloch-Pitts neuron class
class McCulloch_Pitts_neuron:
    def __init__(self, weights, threshold):
        self.weights = weights
        self.threshold = threshold

    def model(self, x):
        return 1 if np.dot(self.weights, x) >= self.threshold else 0

# Define the function to determine if a point is inside the triangle
def Area(x, y):
    # Define neurons based on the lines of the triangle
    neur1 = McCulloch_Pitts_neuron([2, -1], 2)  # Line 1: 2x - y = 2
    neur2 = McCulloch_Pitts_neuron([-2, -1], -6)  # Line 2: -2x - y = -6
    neur3 = McCulloch_Pitts_neuron([0, 1], 0)  # Line 3: y = 0
    neur4 = McCulloch_Pitts_neuron([1, 1, 1], 3)  # Combine previous neurons' outputs

    # Calculate the output of each neuron
    z1 = neur1.model(np.array([x, y]))
    z2 = neur2.model(np.array([x, y]))
    z3 = neur3.model(np.array([x, y]))
    z4 = neur4.model(np.array([z1, z2, z3]))

    return z4
```

نورون 1: همانطور که در کامنت مشخص شده خط دوم را نمایندگی می کند.

نورون 2: همانطور که در کامنت مشخص شده خط سوم را نمایندگی می کند.

نورون 3: همانطور که در کامنت مشخص شده خط اول را نمایندگی می کند.

نورون 4: این نورون برای تصمیم گیری نهایی که آیا داده در داخل منطقه محصور شده می باشد یا نه قرار داده شده. نورون چهارم با وزن‌هایی برابر با [1,1,1] و آستانه 3 تعریف شده است و خروجی سه نورون قبلی را ترکیب می‌کند و فقط زمانی خروجی 1 می‌دهد که همه‌ی نورون‌های قبلی خروجی 1 داده باشند. این یعنی نقطه داخل مثلث است.

سپس داده های تصادفی را تولید می کنیم.

```python
# Generate random data points
num_points = 2000
x_values = np.random.uniform(0, 4, num_points)
y_values = np.random.uniform(-1, 3, num_points)
```

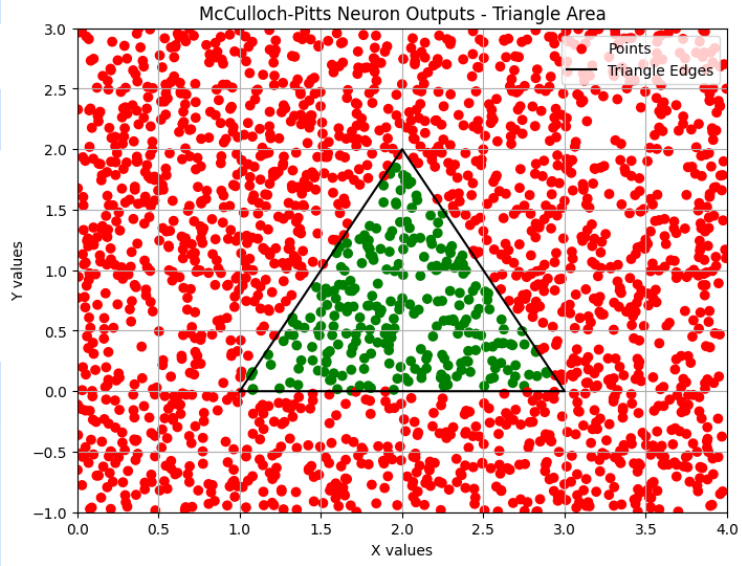
به کمک ابزار های هوشمند در یک خط کد شرایط داده رندوم تولید شده (داخل مثلث یا خارج مثلث بودن) بررسی می شود و رنگ داده بر حسب آن تغییر می کند.

```python
# Initialize color arrays based on the Area function output
colors = ['green' if Area(x, y) == 1 else 'red' for x, y in zip(x_values, y_values)]
```

در آخر هم به نمایش داده ها می پردازیم و محدوده مولفه های طول و عرض را با دقت به شکل تنظیم می کنیم:

```
# Plotting
plt.figure(figsize=(8, 6))
plt.scatter(x_values, y_values, c=colors, label='Points')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('McCulloch-Pitts Neuron Outputs - Triangle Area')

# Plotting lines with legends
x_triangle = [1, 3, 2, 1]
y_triangle = [0, 0, 2, 0]
plt.plot(x_triangle, y_triangle, color='black', linestyle='-', label='Triangle Edges')

plt.grid(True)
plt.xlim(0, 4)
plt.ylim(-1, 3)
plt.legend(loc='upper right')
plt.show()
```

در آخر نتایج مطابق نتایج مطلوب سوال بدست خواهد آمد:



اثر اضافه کردن تابع فعالساز:

در نحوه حل این سوال اضافه کردن تابع فعال ساز بی مورد می باشد. نیازی به اضافه کردن تابع فعالساز دیگری نمی باشد. در اثر اضافه کردن تابع های فعالسازی می توان موقعیت نقطه را به صورت احتمالاتی و وزن دار گزارش کرد که با چه احتمالی نقطه داخل محدوده است و با چه احتمالی خارج است و در واقع باعث می شود مرز مثلث حالت Gradient بگیرد نه مثل الان Strict که در این مسئله مطلوب ما نیست.

سوال دوم بخش 1

به کمک دستور نوشته شده در سایت Kaggle دیتاست را به محیط گوگل کولب import می کنیم و آنرا از حالت فشرده خارج می کنیم.

```
[1]  !kaggle datasets download aslkuscu/telecust1000t
```

```
Dataset URL: https://www.kaggle.com/datasets/aslkuscu/telecust1000t
License(s): apache-2.0
Downloading telecust1000t.zip to /content
  0% 0.00/10.3k [00:00<?, ?B/s]
100% 10.3k/10.3k [00:00<00:00, 18.0MB/s]
```

```
!unzip telecust1000t.zip -d /content/telecust1000t
```

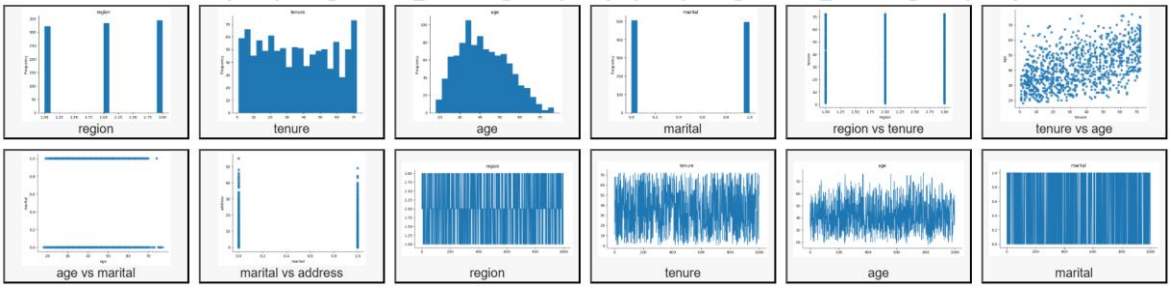
```
Archive:  telecust1000t.zip
  inflating: /content/telecust1000t/teleCust1000t.csv
```

سپس به کمک کتاب خانه Pandas اطلاعاتی را از دیتاست استخراج می کنیم.

```
import pandas as pd
df = pd.read_csv("/content/telecust1000t/teleCust1000t.csv")
df.head()
```

	region	tenure	age	marital	address	income	ed	employ	retire	gender	reside	custcat
0	2	13	44	1	9	64.0	4	5	0.0	0	2	1
1	3	11	33	1	7	136.0	5	5	0.0	0	6	4
2	3	68	52	1	24	116.0	1	29	0.0	1	2	3
3	2	33	33	0	12	33.0	2	0	0.0	1	1	1
4	2	23	30	1	9	30.0	1	2	0.0	0	4	3

نمودار های توصیه شده از جانب گوگل کولب هم بررسی می کنیم.



تا حدودی می توان از نمودار های رسم شده از پیوسته و یا گسسته بودن داده ها و ویژگی ها اطلاعات کسب کرد.

داده ها شامل ویژگی های زیر است:

- ✓ region: منطقه
- ✓ tenure: مدت عضویت
- ✓ age: سن
- ✓ marital: وضعیت تأهل
- ✓ address: آدرس
- ✓ income: درآمد
- ✓ ed: تحصیلات
- ✓ employ: اشتغال
- ✓ retire: بازنشستگی
- ✓ gender: جنسیت
- ✓ reside: محل سکونت
- ✓ custcat: دسته‌بندی مشتری (4 کلاس)

```
df.isna()
```

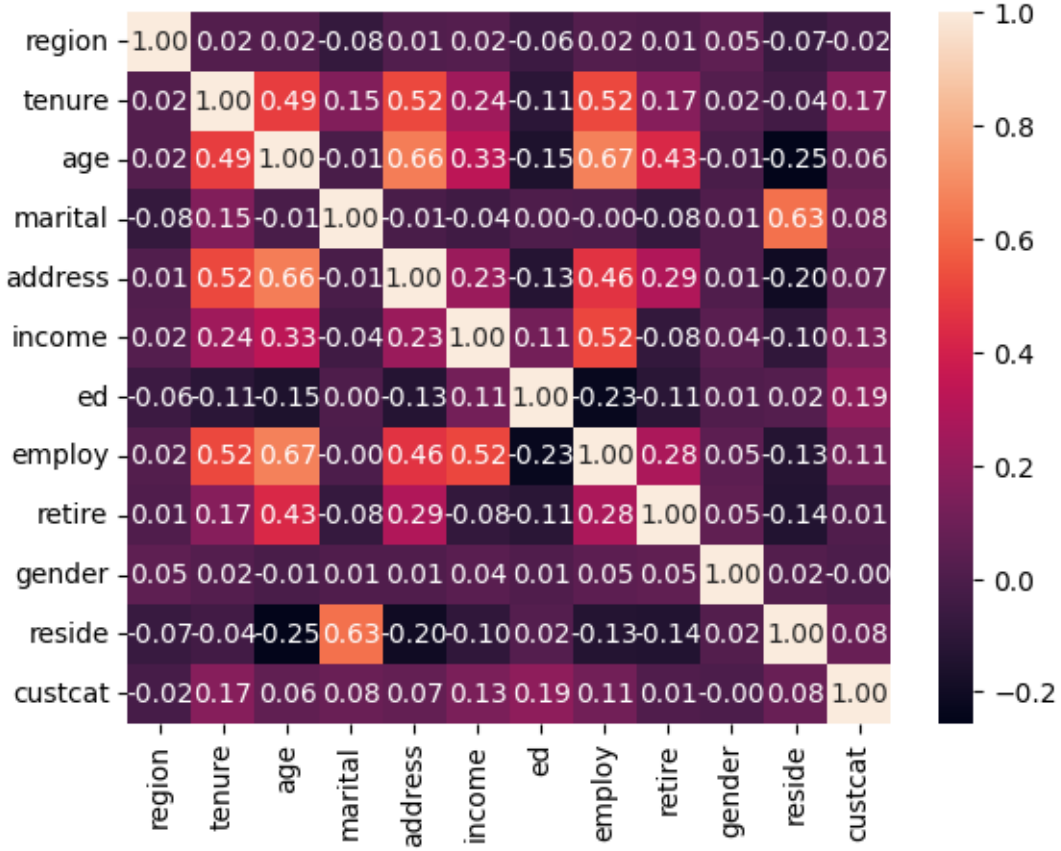
	region	tenure	age	marital	address	income	ed	employ	retire	gender	reside	custcat
0	False	False	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...
995	False	False	False	False	False	False	False	False	False	False	False	False
996	False	False	False	False	False	False	False	False	False	False	False	False
997	False	False	False	False	False	False	False	False	False	False	False	False
998	False	False	False	False	False	False	False	False	False	False	False	False
999	False	False	False	False	False	False	False	False	False	False	False	False

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 12 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   region    1000 non-null   int64  
 1   tenure    1000 non-null   int64  
 2   age       1000 non-null   int64  
 3   marital   1000 non-null   int64  
 4   address   1000 non-null   int64  
 5   income    1000 non-null   float64
 6   ed        1000 non-null   int64  
 7   employ    1000 non-null   int64  
 8   retire    1000 non-null   float64
 9   gender    1000 non-null   int64  
 10  reside    1000 non-null   int64  
 11  custcat   1000 non-null   int64  
dtypes: float64(2), int64(10)
memory usage: 93.9 KB
```

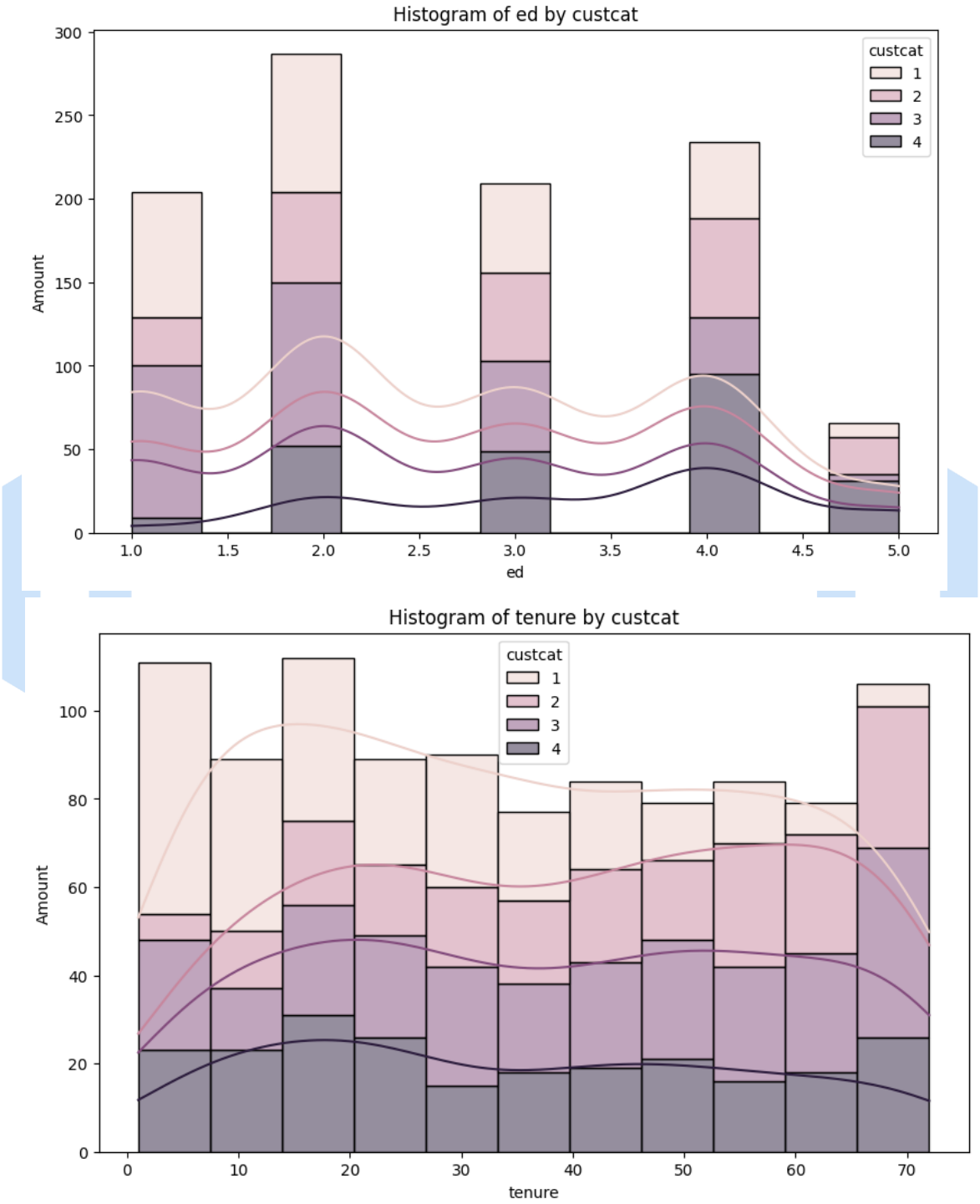
هیچ دیتای NULL در داده ها وجود ندارد. دو دیتا(بازنشستگی و درآمد) از تایپ float و بقیه همه int هستند.

سوال دوم بخش 2



همانطور که در دیتا دیده میشود دو ویژگی تحصیلات یا همان ed (0.19) و مدت عضویت Tenure (0.17) بیشترین همبستگی را با داده هدف دارند.

بنابراین نمودار همبستگی آنها را با ویژگی هدف (Custcat) رسم می کنیم.



سوال دوم بخش 3

حال داده ها را به کمک MinMaxScaler نرمالسازی می کنیم و به سه دسته آموزش، آزمون و اعتبارستجی تقسیم می کنیم.

داده ها را پیش و پس از نرمالسازی نمایش می دهیم همچنین نسبت تقسیم بندی داده ها برای مدل هم به نمایش می گذاریم.

```
Unnormalized Data:
   region  tenure  age  marital  address  income  ed  employ  retire  gender  \
0       2      13   44        1        9    64.0   4       5     0.0       0
1       3      11   33        1        7   136.0   5       5     0.0       0
2       3      68   52        1       24   116.0   1      29     0.0       1
3       2      33   33        0       12    33.0   2       0     0.0       1
4       2      23   30        1        9    30.0   1       2     0.0       0

   reside
0       2
1       6
2       2
3       1
4       4

Normalized Data:
   region    tenure       age  marital   address    income    ed    employ  \
0     0.5  0.169014  0.440678      1.0  0.163636  0.033153  0.75  0.106383
1     1.0  0.140845  0.254237      1.0  0.127273  0.076552  1.00  0.106383
2     1.0  0.943662  0.576271      1.0  0.436364  0.064497  0.00  0.617021
3     0.5  0.450704  0.254237      0.0  0.218182  0.014467  0.25  0.000000
4     0.5  0.309859  0.203390      1.0  0.163636  0.012658  0.00  0.042553

   retire  gender    reside
0     0.0     0.0  0.142857
1     0.0     0.0  0.714286
2     0.0     1.0  0.142857
3     0.0     1.0  0.000000
4     0.0     0.0  0.428571

Shapes of each split:
Training data: (700, 11), (700, 4)
Validation data: (150, 11), (150, 4)
Test data: (150, 11), (150, 4)
```

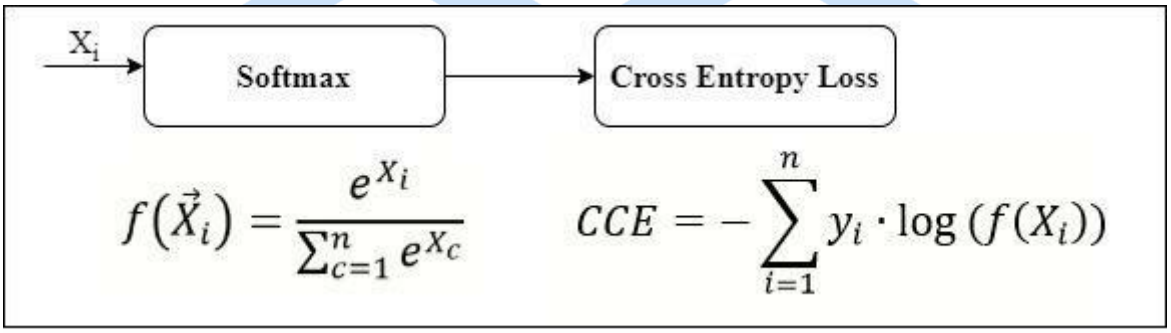
همان طور که مشاهده می شود تقسیم بندی داده ها به شرح زیر می باشد:

- داده های آموزش: 70 درصد
- داده های ولیدیشن: 15 درصد
- داده های تست: 15 درصد

سوال دوم بخش 4

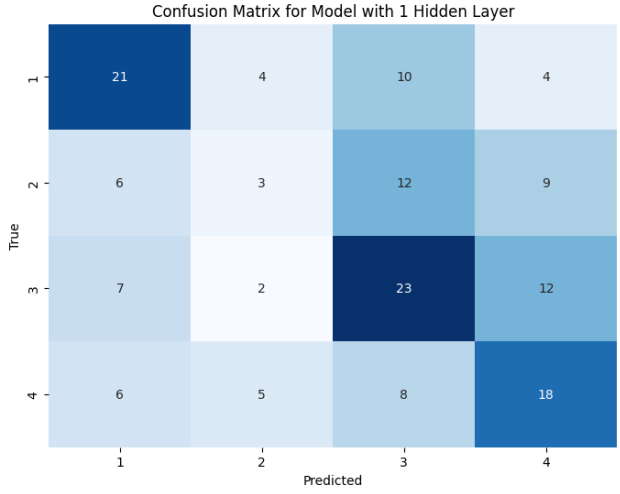
طبق گفته سوال دو مدل برای مسئله طراحی می کنیم.

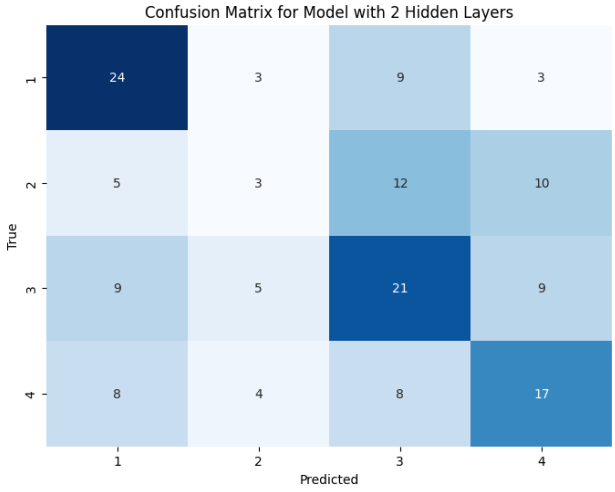
اولین مدل شامل یک لایه پنهان شامل 32 نورون و دومین مدل شامل دو لایه پنهان که هر دو شامل 32 نورون هستند، لایه خروجی شامل 4 نورون (که با تعداد کلاس‌ها یکیست) با تابع فعال‌سازی softmax پیاده سازی شده. بهینه‌ساز SGD با نرخ یادگیری 0.03 تنظیم شده است و تابع هزینه هم Categorial_Crossentropy در نظر گرفته شده.



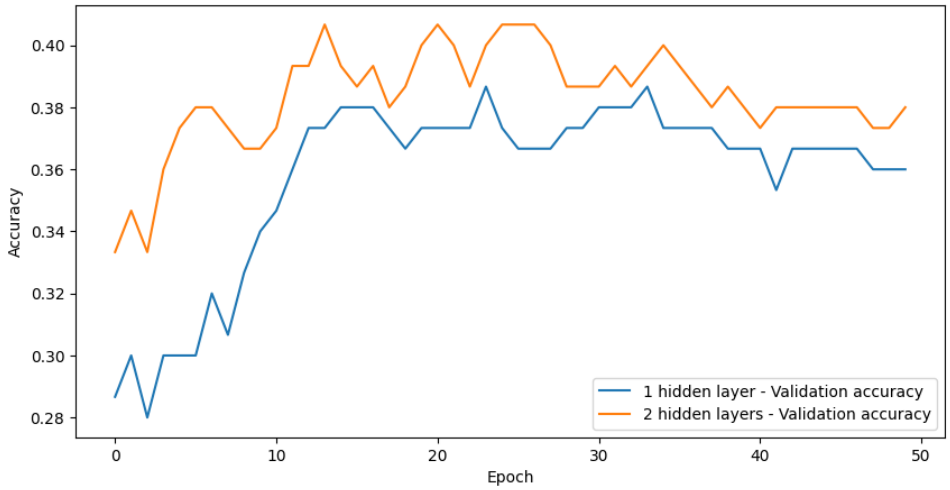
تابع train_and_evaluate مدل را آموزش می‌دهد و دقت نهایی و ماتریس سردرگمی را محاسبه می‌کند و در آخر هم پیش‌بینی‌های مدل را ارزیابی می‌کند.

نتایج مدل به شرح جداول زیر می باشد:





همچنین مقایسه دو مدل روی عملکرد بر روی داده های اعتبار سنجی در طول 50 ایپاک به نمایش زیر می باشد:

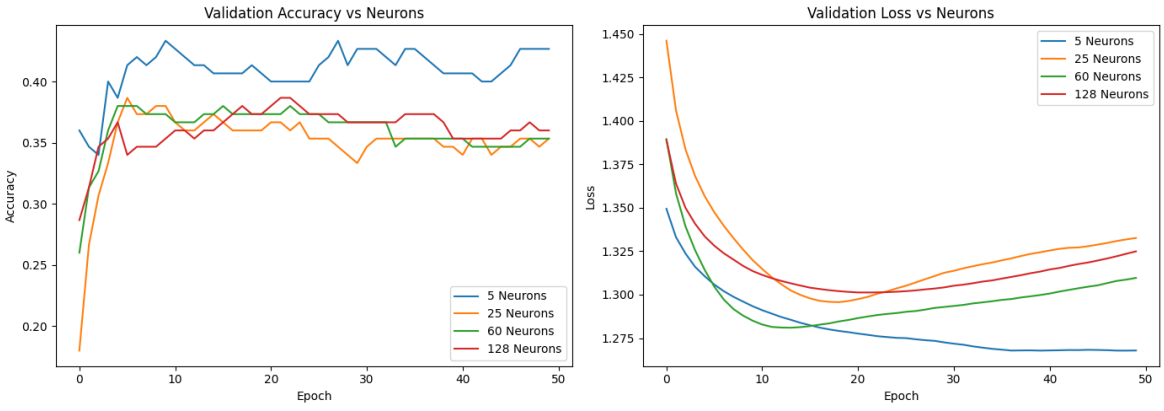


همانطور که مشاهده می شود عملکرد هر دو مدل آنچنان مطلوب نیست اما واضح است که عملکرد مدل با دو لایه پنهان بهتر از مدل با یک لایه پنهان بوده است.

❖ تاثیر افزایش نورون ها

مدل دو لایه انتخاب شده را در 4 حالت شامل 5، 25، 60، 128 نورونه بررسی می کنیم.

نتایج به صورت زیر می باشد:



همانطور که مشاهده می شود به نظر می آید مدل های با بیش از 5 نورون روی مدل overfit شده اند و خطای زیاد و دقت کم روی داده های اعتبارسنجی حاکی از این امر می باشد. پس مدل با 5 نورون در این قسمت بهترین مدل است.

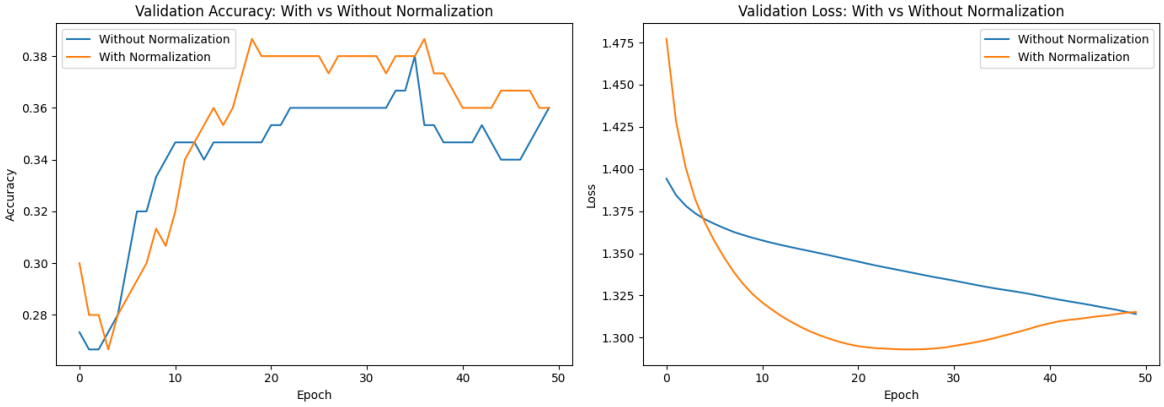
❖ تأثیر اضافه کردن لایه نرمال سازی دسته

Batch Normalization مقادیر ورودی هر لایه را نرمال‌سازی می‌کند و شامل مراحل زیر است:

- ✓ محاسبه میانگین و انحراف معیار برای هر دسته (Batch)
- ✓ نرمال‌سازی ورودی‌ها به توزیعی با میانگین صفر و واریانس یک.
- ✓ معرفی پارامترهای قابل یادگیری γ (مقیاس) و β (انتقال) برای حفظ انعطاف‌پذیری مدل.

$$y_i = \gamma x_i + \beta$$

این روش نرمال سازی را به مدل اعمال می کنیم و نمودار پیش و پس از اعمال را رسم می کنیم.



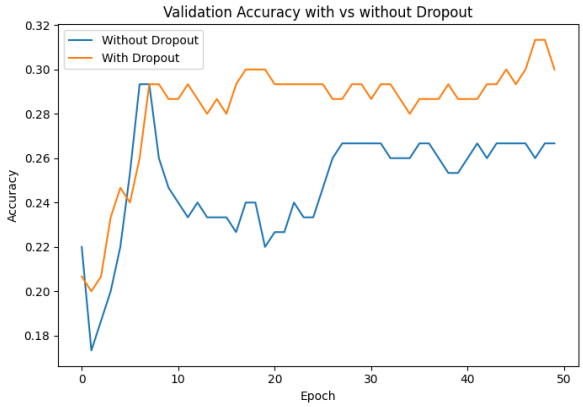
همانطور که مشاهده می شود دقت نمودار پس از نرمالسازی هم خطای کمتری دارد هم دقت بالاتری اما بنظر می آید پس از ایپاک 20 تا 30 به بعد مدل اندکی overfit شده که باعث افزایش مجدد خطا و کاهش دقت شده.

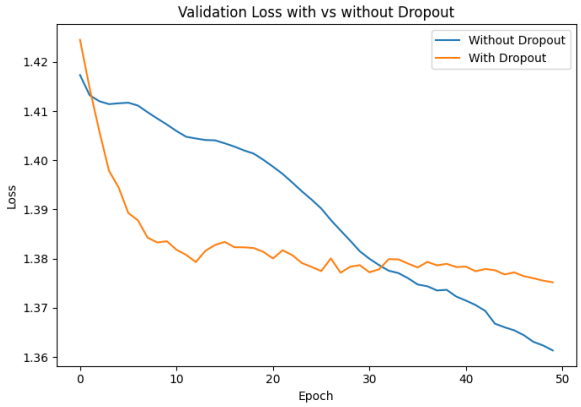
❖ تاثیر Dropout

Dropout یکی از تکنیک‌های معروف برای تنظیم‌سازی (Regularization) در شبکه‌های عصبی است که با کاهش احتمال بیش‌برازش (Overfitting) به بهبود عملکرد مدل کمک می‌کند.

Dropout در طی آموزش، به طور تصادفی تعدادی از نورون‌های یک لایه را غیرفعال (Drop) می‌کند. غیرفعال کردن به این معناست که اولا خروجی نورون برابر صفر می‌شود و ثانیا وزن‌های مربوط به نورون‌های غیرفعال در آن گام به‌روزرسانی نمی‌شوند.

نتایج افزودن Dropout به مدل به شرح زیر می باشد:



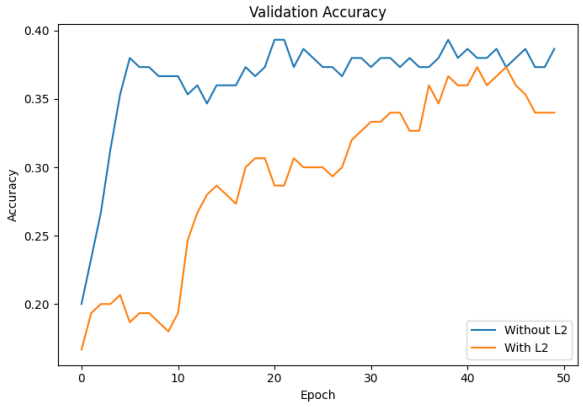


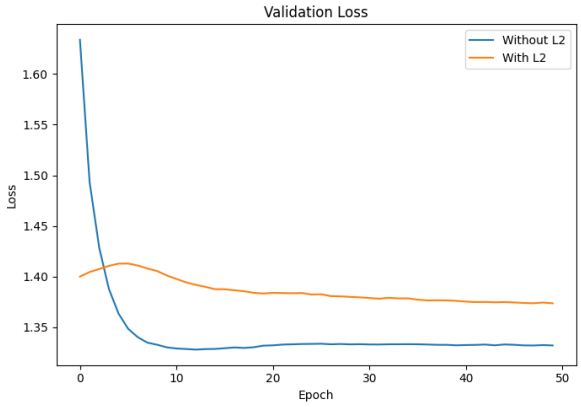
همانطور که مشاهده می شود افزودن Dropout تاثیر مثبتی در مدل داشته است. اما از آنجایی که در این مدل پیچیده تنها 1000 داده وجود دارد نمیتوان با دقت بالایی برای این مجموعه داده اعمال کرد و در مجموع این مدل نیز مانند مدل های پیشین عملکرد بهتری دارد اما همچنان مطلوب نیست.

❖ تاثیر L2-Regularization

تنظیم تناسب L2 به این صورت است که اگر مدل پیچیدگی زیادی داشته باشد یا داده‌های آموزشی نویزی باشد، ممکن است مدل بیش از حد داده‌های آموزشی را حفظ کند و به جای یادگیری الگوهای کلی، به خاطر بسپارد. این مسئله موجب کاهش عملکرد مدل بر روی داده‌های جدید (داده‌های آزمون) می‌شود. تنظیم تناسب L2 با محدودسازی وزن‌های مدل، به کاهش پیچیدگی آن کمک می‌کند. در این روش، یک جمله به تابع هزینه مدل افزوده می‌شود که شامل مجموع مربعات وزن‌هاست. فرمول کلی تابع هزینه جدید به این صورت است:

$$J(w) = L(w) + \lambda \sum_{i=1}^{n} {w_i}^2$$





با توجه به نمودار ها مدل بدون استفاده از L2 دارای خطای کمتر و عملکرد بهتر است.

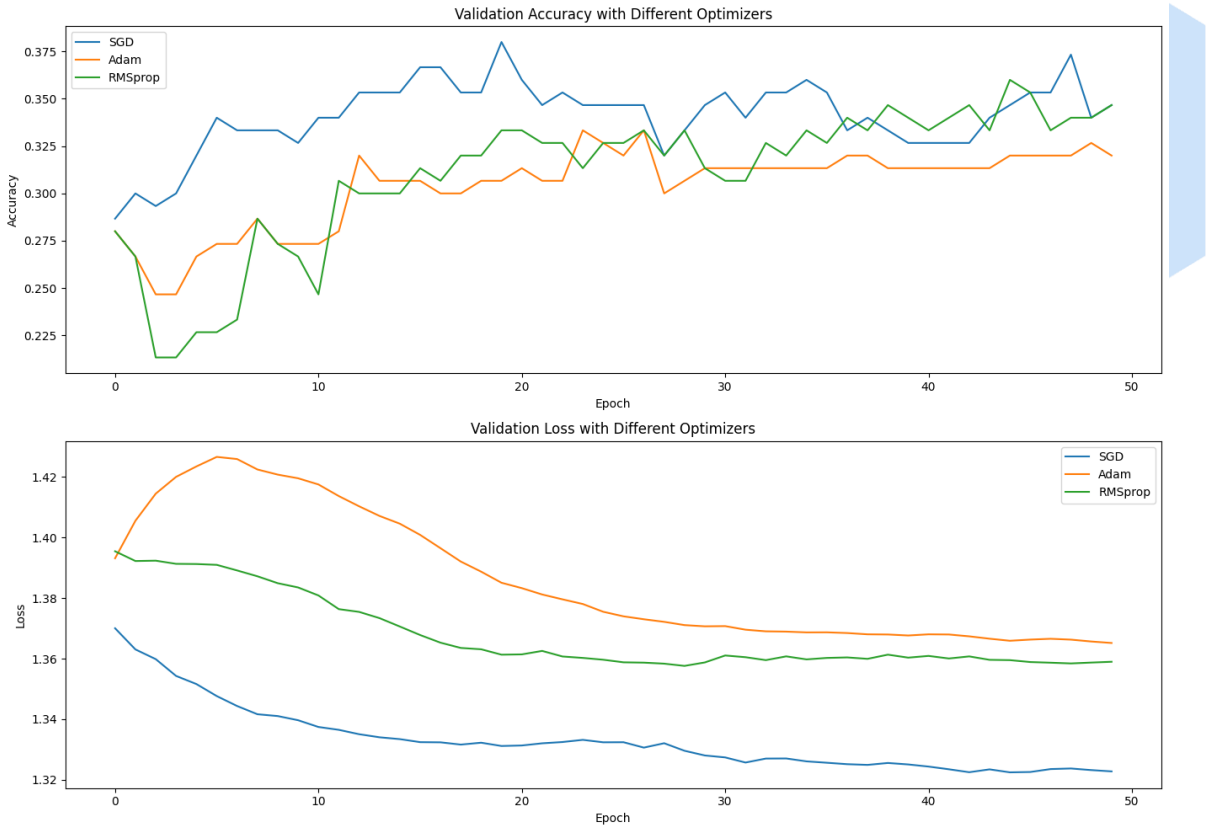
❖ تاثیر Adam و RMSProp

Adam یک روش بهینه‌سازی پیشرفته است که ترکیبی از دو تکنیک Momentum و RMSProp را ارائه می‌دهد .Momentum به کاهش نوسانات کمک می‌کند و RMSProp نرخ یادگیری را به‌صورت تطبیقی برای هر پارامتر تنظیم می‌کند.

Adam از دو میانگین متحرک استفاده می‌کند: یکی برای گرادیان‌ها که جهت کلی حرکت را مشخص می‌کند و دیگری برای مربع گرادیان‌ها که نرخ یادگیری را تنظیم می‌کند. این مقادیر در مراحل اولیه با اصلاح انحراف بهینه می‌شوند تا دقت تخمین‌ها افزایش یابد.

مزایای Adam شامل تنظیم خودکار نرخ یادگیری، همگرایی سریع و پایدار، و عملکرد خوب در داده‌های noisyو مسائل پیچیده است. این الگوریتم برای کاربردهای گسترده‌ای مناسب است و یکی از محبوب‌ترین روش‌های بهینه‌سازی در یادگیری عمیق به شمار می‌رود.

بررسی مدل به سه روش :



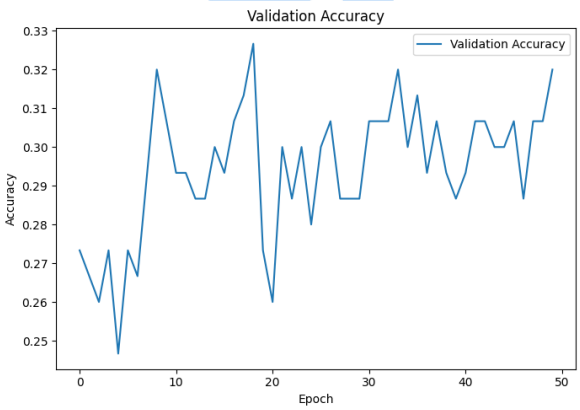
به نظر می آید که به ترتیب SGD ، Adam، و در نهایت RMSprop بهترین عملکرد ها را در مدل ما داشته اند.

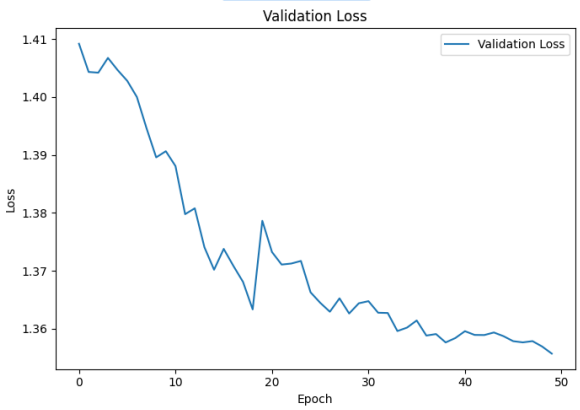
سوال دوم بخش 5

نتایج ارزیابی روی داده های تست به شرح زیر می باشد:

```
Test accuracy: 0.3533
Test loss: 1.3465

10 test data samples with predicted values:
Sample 1: True value: 2 Predicted value: 3
Sample 2: True value: 0 Predicted value: 3
Sample 3: True value: 1 Predicted value: 3
Sample 4: True value: 0 Predicted value: 3
Sample 5: True value: 1 Predicted value: 0
Sample 6: True value: 3 Predicted value: 3
Sample 7: True value: 1 Predicted value: 0
Sample 8: True value: 3 Predicted value: 0
Sample 9: True value: 0 Predicted value: 0
Sample 10: True value: 3 Predicted value: 3
```

همانطور که مشاهده می شود دقت آنچنان مطلوب نیست از 10 داده تقریبا 3 داده به درستی پیشبینی شده که این حاکی از مدل پیچیده و تعداد دیتا های محدود می باشد.



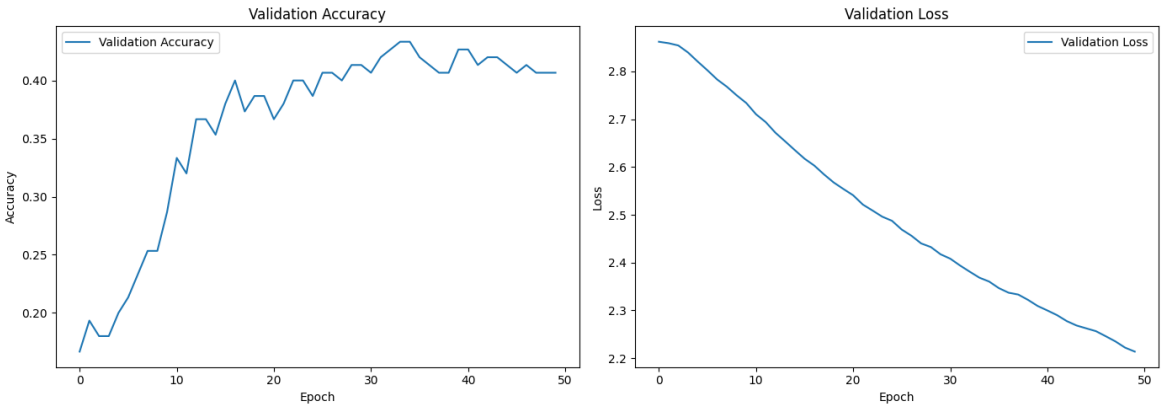


سوال دوم بخش 6

برای یافتن بهترین مدل برای این سوال به کمک الگوریتم های هوش مصنوعی کدی نوشته شد که تمام حالات را از جمله انتخاب از بین Optimizer ها یا تعداد لایه و حتی تعداد نورون ها بررسی کند و بهترین مدل را گزارش کند.

طبق این برنامه مدل با 64 نورون در سه لایه و بهینه ساز SGD و با استفاده از Batch Normalization، Dropoutبا نرخ 0.5، L2 Regularization با نرخ 0.01 توانست مدلی را با دقت 0.4333 درصد در داده های تست نتیجه بدهد.

نتایج آن به شرح نمودار زیر می باشد:




```
Best model configuration:
{'neurons': 64, 'layers': 3, 'optimizer_name': 'SGD', 'use_batch_norm': True, 'use_dropout': True, 'dropout_rate': 0.5, 'use_l2': True, 'l2_reg': 0.01}
Best validation accuracy: 0.4333
```

سوال سوم بخش 1

ابتدا به عملکرد تابع های آماده ای که در اختیارمان قرار داده شده می پردازیم و سپس به حل بخش های 2 و 3 می پردازیم.

تابع convertImageToBinary تابعی می باشد که داده عکسی را دریافت کرده و به آرایه باینری تبدیل می کند و آن را برمی گرداند. در این آرایه نقاطی که سفید در نظر گرفته می شوند با عدد 1- و نقاطی که سیاه لحاظ می شوند عدد 1 به آنها اختصاص می یابد.

در این تابع مقادیر عددی هر یک از سه رنگ RGB بدست می آید و با درنظر گرفتن مقدار آستانه آنرا سفید یا سیاه در نظر می گیرد. در آخر هم لیست رنگ پیکسل ها به عنوان خروجی تابع برگردانده می شود.

```
def convertImageToBinary(path):
    image = Image.open(path)
    draw = ImageDraw.Draw(image)
    width, height = image.size
    pix = image.load()
    factor = 100
    binary_representation = []

    for i in range(width):
        for j in range(height):
            red, green, blue = pix[i, j]
            total_intensity = red + green + blue

            if total_intensity > (((255 + factor) // 2) * 3):
                red, green, blue = 255, 255, 255
                binary_representation.append(-1)
            else:
                red, green, blue = 0, 0, 0
                binary_representation.append(1)

            draw.point((i, j), (red, green, blue))

    del draw
    return binary_representation
```

تابع generateNoisyImages تابعیست که به منظور تولید عکس های نویزی مورد استفاده قرار می گیرد. با حلقه موجود در این تابع برای هر 5 تصویر یک تصویر نویزی تولید می کند. این تابع آدرس عکس را به عنوان ورودی می گیرد و مقادیر باینری را به عنوان خروجی تحویل می دهد.

```
from PIL import Image, ImageDraw
import random

def generateNoisyImages():
    # List of image file paths
    image_paths = [
        "/content/1.jpg",
        "/content/2.jpg",
        "/content/3.jpg",
        "/content/4.jpg",
        "/content/5.jpg"
    ]

    for i, image_path in enumerate(image_paths, start=1):
        noisy_image_path = f"/content/noisy{i}.jpg"
        getNoisyBinaryImage(image_path, noisy_image_path)
        print(f"Noisy image for {image_path} generated and saved as {noisy_image_path}")
```

در تابع getNoisyBinaryImage عکسی را به عنوان ورودی دریافت می کند و به آن نویز اضافه می کند و در آدرس جدید آن را ذخیره می کند.

در این تابع به کمک حلقه for تو در تو روی هر یک از رنگ های RGB یک پیکسل به طور تصادفی نویز ایجاد می شود. پس از مطمئن شدن این که نویز در محدوده قابل تعریف می باشد(0-255) آنرا به تصویر اعمال می کند و تصویر نویزی را ذخیره سازی می کند.

ابتدا 5 داده عکسی حروف الفبا را به کمک دستور gdown وارد محیط گوگل کولب می کنیم و آن ها را نمایش می دهیم.

```
!pip install --upgrade --no-cache-dir gdown
!gdown 1QTi7dJtNAfFR5mG0rd8K3ZGvEIfSn_DS
!unzip PersianData.zip
```

نمایش داده های عکسی:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import glob

# Assuming the images are in the current directory and named with a ".jpg" extension
image_paths = glob.glob('*.jpg')

# Display each image with its dimensions
for img_path in image_paths:
    img = mpimg.imread(img_path)
    height, width, _ = img.shape

    plt.figure()
    plt.imshow(img)
    plt.axis('off')  # Hide the axes
    plt.title(f'Dimensions: {width} x {height} pixels')
    plt.show()
```

این داده ها تصویر هایی مربعی 96 پیکسلی از حروف الفبای فارسی شامل آ، و، ب، د و ج می باشند.

Dimensions: 96 x 96 pixels Dimensions: 96 x 96 pixels Dimensions: 96 x 96 pixels

سوال سوم بخش 2

حال به تعریف مدل همینگ می پردازیم.

به صورت یک کلاس که دارای 3 ویژگی تعداد نورون، وزن و بایاس می باشند تعریف می کنیم.

این کلاس شامل توابع آموزش و پیشبینی می باشند.

تابع آموزش بر حسب ورودی، وزن و بایاس را تنظیم می کند و تابع پیشبینی، نزدیک ترین عکس به تصویر ورودی را بر حسب نتیجه مدل بر می گرداند.

```
class HammingNetwork:
    def __init__(self, num_neurons):
        self.num_neurons = num_neurons
        self.weights = None
        self.bias = None

    def train(self, inputs):
        self.weights = np.array(inputs)
        self.bias = np.ones(self.num_neurons) * 0.5 * len(inputs[0]) * 3

    def predict(self, input_image):
        product = np.dot(self.weights, input_image)
        stored = product - self.bias
        return np.argmax(stored)
```

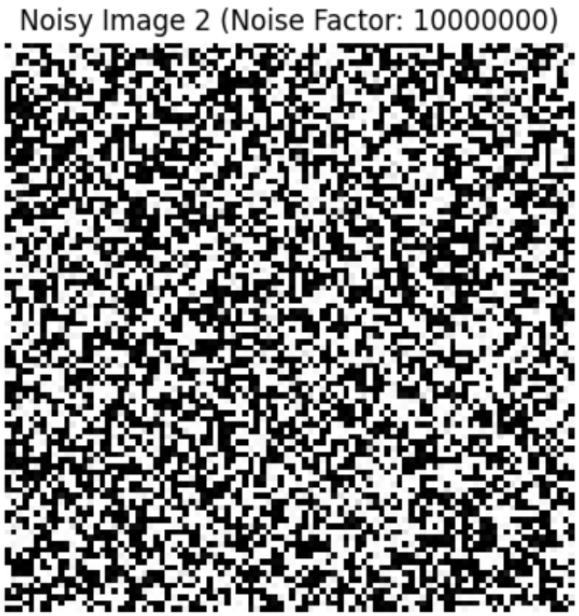
برای نمایش تصاویر عادی و نویزی طبق گفته سوال به صورت زیر عمل میکنیم تا با هر بار اضافه کردن نویز به تصویر آن را به ما نمایش بدهد:

```
def display_images(noisy_image_pil, predicted_image_pil, noise_factor, inde
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(noisy_image_pil)
    plt.title(f'Noisy Image {index+1} (Noise Factor: {noise_factor})')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.imshow(predicted_image_pil)
    plt.title(f'Predicted Image')
    plt.axis('off')

    plt.show()
```

متغییر noisy_image_pil همان تصویر حاوی نویز می باشد. noise_factor هم میزان نویز را در تصویر مشخص می کند.

در آخرین مرحله هم به آموزش شبکه می پردازیم و نویز را با پله های 500000 زیاد می کنیم تا اولین جایی که مدل دچار اختلال شود و عکس را اشتباه تشخیص دهد و پیشبینی کند.

نتایج مدل به شرح زیر می باشد:




```
Given Image: noisy2.jpg, Predicted Image Index: 3
Prediction error at noise factor 10000000 for image 2
The highest noise factor where the network was still accurate: 9500000
```

در این جا کد متوقف می شود چون که مدل قادر به تشخیص تصویر نمی باشد. آخرین عدد نویزی که مدل قادر به تشخیص عکس شد 9500000 می باشد. همان طور که مشاهده می شود تصویر دوم نویزی به مدل داده شد و شبکه تصویر سوم را پیش بینی کرده است.

سوال سوم بخش 3

تفاوت این بخش با بخش قبلی تنها در این است که در مسئله قبل نقاط سیاه و سفید به تصویر به صورت رندوم اضافه می شد. اما در این بخش نقاطی که سیاه هستند به صورت رندوم که با متغییر Change_rate کنترل می شوند به نقاط سفید تبدیل می شوند. این متغییر در ابتدا با نرخ 0.01 تعریف شده و با همین مقدار هر بار درصدی از شکل را سفید می کند.

تابع missing_point نیز مسئول تولید این نقاط سفید می باشد.

```python
# Improved function to change black/gray-like pixels to white gradually
def addingMissingPoint(input_path, output_path, change_rate):

    # Open the input image.
    image = Image.open(input_path).convert('RGB')
    draw = ImageDraw.Draw(image)
    width, height = image.size
    pix = image.load()

    missing_points = 0
    # Threshold to consider gray-like points as black
    threshold = 128

    black_pixels = [(i, j) for i in range(width) for j in range(height) if all(channel < threshold for channel in pix[i, j])]
    num_pixels_to_change = int(len(black_pixels) * change_rate)
    sampled_pixels = random.sample(black_pixels, num_pixels_to_change)

    for i, j in sampled_pixels:
        draw.point((i, j), (255, 255, 255))  # Change black/gray-like pixel to white
        missing_points += 1

    image.save(output_path, "JPEG")
    del draw

    return missing_points
```

این تابع ابتدا با استخراج RGB پس از باز کردن عکس و با در نظر گرفتن مقدار آستانه نقاطی را سفید می کند. همچنین متغییر missing_points نیز تعداد نقاط سفید شده را شمارش می کند.

در آخر این روند تا جایی ادامه پیدا می کند که عکس نویزی را به اشتباه تشخیص می دهد.

```
Given Image: noisy3.jpg, Predicted Image Index: 4
Prediction error. The image was image 3 but the network predicted image 4
The highest number of missing points where the network was still accurate: 88910
```

همانطور که مشاهده می شود با افزودن 88910 نقطه سفید به تصویر حرف ج مدل دیگر قادر به تشخیص آن نخواهد بود.

راه حل های پیشنهادی:

استفاده از شبکه‌های بازسازی تصویر:

شبکه‌های بازسازی یا همان **اتواینکودرها** برای یادگیری ویژگی‌های اصلی تصاویر طراحی می‌شوند. این شبکه‌ها با ورودی دادن تصاویر نویزی و آموزش برای بازسازی نسخه‌ی اصلی، می‌توانند به خوبی نویز را

حذف کنند. شما می‌توانید این روش را به عنوان یک پیش‌پردازش استفاده کنید و تصاویر نویزی را قبل از اینکه به شبکه اصلی بدهید، بازسازی کنید.

افزایش تنوع داده‌ها در آموزش:

یکی از دلایل حساس بودن شبکه‌ها به نویز، این است که شبکه با تنوع کافی از داده‌ها آموزش ندیده است. برای حل این مشکل:

- در مرحله‌ی آموزش، تصاویر را عمداً با انواع نویزها (نقاط سفید، لکه‌ها، نویز گوسی و ...) خراب کنید و شبکه را برای تشخیص آن‌ها آموزش دهید.
- این کار باعث می‌شود شبکه به جای تمرکز روی جزئیات کوچک و غیرضروری، روی ویژگی‌های کلی و مهم تصویر تمرکز کند.

معماری‌های مقاوم‌سازی شده در شبکه عصبی:

طراحی معماری شبکه به شکلی که نسبت به نویز مقاوم باشد اهمیت زیادی دارد. برای مثال:

- استفاده از لایه‌ها **Dropout** در حین آموزش کمک می‌کند شبکه توانایی خود را در مواجهه با تغییرات تصادفی افزایش دهد.
- طراحی شبکه با **ResNet یا DenseNet** باعث می‌شود مسیرهای متنوعی برای پردازش اطلاعات وجود داشته باشد و شبکه کمتر به جزئیات نویزی حساس شود.

تکنیک‌های افزایش داده:

با اعمال تغییرات تصادفی روی تصاویر اصلی مانند چرخش، برش، بزرگ‌نمایی و تغییر روشنایی، شبکه می‌تواند آموزش ببیند که ویژگی‌های اصلی تصویر در مقابل این تغییرات ثابت بماند. این روش مقاومت شبکه را در برابر نویز افزایش می‌دهد.

استفاده از شبکه‌های مبتنی بر تبدیل:

برخی از نویزها در حوزه‌ی زمان یا مکان مشخص می‌شوند. استفاده از تبدیل‌های **Wavelet** یا **Fourier** می‌تواند به تشخیص و حذف این نویزها کمک کند. مثلاً در تبدیل موجک، نویز اغلب در مقیاس‌های خاصی ظاهر می‌شود که می‌توان آن‌ها را حذف کرد.

استفاده از داده‌های مصنوعی با شدت نویز بالا:

تولید داده‌های مصنوعی با شدت‌های مختلف نویز و استفاده از آن‌ها در مراحل آموزش، شبکه را قادر می‌سازد که در مواجهه با نویزهای مشابه عملکرد بهتری داشته باشد.

سوال چهارم بخش 1

ابتدا داده ها را از کتاب خانه Scikit learn بارگزاری می کنیم و آنها را به مجموعه داده های آموزش و تست تقسیم می کنیم. آن ها را استاندارد سازی کرده (Normalization) و سپس به نمایش اطلاعات کلی مجموعه داده ها می پردازیم.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the California Housing dataset
housing = fetch_california_housing()
X, y = housing.data, housing.target

print("Dataset loaded successfully.")
print(f"Feature names: {housing.feature_names}")
print(f"First 5 rows of data:\n{X[:5]}")
print(f"First 5 target values:\n{y[:5]}")

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=73)

print("\nData split into training and testing sets.")
print(f"Training data shape: {X_train.shape}")
print(f"Testing data shape: {X_test.shape}")

# Standardize the dataset
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print("\nData standardized.")
```

نمایش اطلاعات کلی داده ها به شرح شکل زیر می باشد:

- نام ویژگی داده
- پنج سطر اول به همراه داده هدف آن ها یعنی قیمت متوسط خانه ها
- تعداد داده های آزمون و آموزش

```
Dataset loaded successfully.
Feature names: ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup', 'Latitude', 'Longitude']
First 5 rows of data:
[[ 8.32520000e+00  4.10000000e+01  6.98412698e+00  1.02380952e+00
   3.22000000e+02  2.55555556e+00  3.78800000e+01 -1.22230000e+02]
 [ 8.30140000e+00  2.10000000e+01  6.23813708e+00  9.71880492e-01
   2.40100000e+03  2.10984183e+00  3.78600000e+01 -1.22220000e+02]
 [ 7.25740000e+00  5.20000000e+01  8.28813559e+00  1.07344633e+00
   4.96000000e+02  2.80225989e+00  3.78500000e+01 -1.22240000e+02]
 [ 5.64310000e+00  5.20000000e+01  5.81735160e+00  1.07305936e+00
   5.58000000e+02  2.54794521e+00  3.78500000e+01 -1.22250000e+02]
 [ 3.84620000e+00  5.20000000e+01  6.28185328e+00  1.08108108e+00
   5.65000000e+02  2.18146718e+00  3.78500000e+01 -1.22250000e+02]]
First 5 target values:
[4.526 3.585 3.521 3.413 3.422]

Data split into training and testing sets.
Training data shape: (16512, 8)
Testing data shape: (4128, 8)

Data standardized.
```

سپس به تعریف لایه RBF می پردازیم:

```python
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.layers import Input
import tensorflow as tf

# Define RBF layer
class RBFLayer(tf.keras.layers.Layer):
    def __init__(self, units, gamma):
        super().__init__()
        self.units = units
        self.gamma = gamma

    def build(self, input_shape):
        self.centers = self.add_weight(
            shape=(self.units, input_shape[1]),
            initializer='random_normal',
            trainable=True
        )
        self.built = True

    def call(self, inputs):
        c = tf.expand_dims(self.centers, axis=0)
        h = tf.expand_dims(inputs, axis=1)
        return tf.exp(-self.gamma * tf.math.reduce_sum(tf.math.square(h - c), axis=2))
```

gamma تعیین‌کننده حساسیت لایه نسبت به فاصله داده‌ها است.

سوال چهارم بخش 2

حال به تعریف مدل RBF می پردازیم و از Adam برای بهینه سازی و از MSE برای محاسبه خطا استفاده می کنیم:

```
# Define the RBF model
rbf_model = Sequential()
rbf_model.add(Input(shape=(X_train.shape[1],)))
rbf_model.add(RBFLayer(10, 0.1))
rbf_model.add(Dense(1))

# Compile the model
rbf_model.compile(optimizer=Adam(learning_rate=0.01), loss='mse')

print("RBF model defined and compiled.")
print(rbf_model.summary())
```

مدل نهایی با دولایه که یکی RBF و یکی Dense می باشد تعریف شده است.

```
RBF model defined and compiled.
Model: "sequential"
```

Layer (type)	Output Shape	Param #
rbf_layer (RBFLayer)	(None, 10)	80
dense (Dense)	(None, 1)	11

```
Total params: 91 (364.00 B)
Trainable params: 91 (364.00 B)
Non-trainable params: 0 (0.00 B)
None
```

سوال چهارم بخش 4

حال به تعریف مدل شبکه عصبی کاملا متصل می پردازیم و از Adam برای بهینه سازی و از MSE برای محاسبه خطا استفاده می کنیم.

از تابع فعالساز Relu که مناسب این شبکه می باشد نیز استفاده می شود.

```
# Define the Dense model
dense_model = Sequential()
dense_model.add(Dense(64, input_shape=(X_train.shape[1],), activation='relu'))
dense_model.add(Dense(1))

# Compile the model
dense_model.compile(optimizer=Adam(learning_rate=0.01), loss='mse')

print("Dense model defined and compiled.")
print(dense_model.summary())
```

مدل نهایی با دولایه Dense تعریف شد.

```
Dense model defined and compiled.
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pas
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 64)	576
dense_2 (Dense)	(None, 1)	65

```
Total params: 641 (2.50 KB)
Trainable params: 641 (2.50 KB)
Non-trainable params: 0 (0.00 B)
```

سپس به آموزش مدل ها می پردازیم:

```
# Train the RBF model
history_rbf = rbf_model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2, verbose=1)

print("RBF model training complete.")
print(f"Training Loss: {history_rbf.history['loss']}")
print(f"Validation Loss: {history_rbf.history['val_loss']}")
```

سوال چهارم بخش5 و 6

میزان خطا را با داده های آزمون محاسبه کرده و نتایج مدل را گزارش می کنیم.

```
# Evaluate the RBF model
rbf_loss = rbf_model.evaluate(X_test, y_test, verbose=0)
print(f"RBF Model Test Loss: {rbf_loss:.4f}")
```

خطای محاسبه شده MSE را به صورت زیر گزارش می دهیم.

RBF Model Test Loss: 0.4152

حال به آموزش مدل متصل می پردازیم:

```
# Train the Dense model
history_dense = dense_model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2, verbose=1)

print("Dense model training complete.")
print(f"Training Loss: {history_dense.history['loss']}")
print(f"Validation Loss: {history_dense.history['val_loss']}")
```

میزان خطا را با داده های آزمون محاسبه کرده و نتایج مدل را گزارش می کنیم.

```
# Evaluate the Dense model
dense_loss = dense_model.evaluate(X_test, y_test, verbose=0)
print(f"Dense Model Test Loss: {dense_loss:.4f}")
```

خطای محاسبه شده MSE را به صورت زیر گزارش می دهیم.

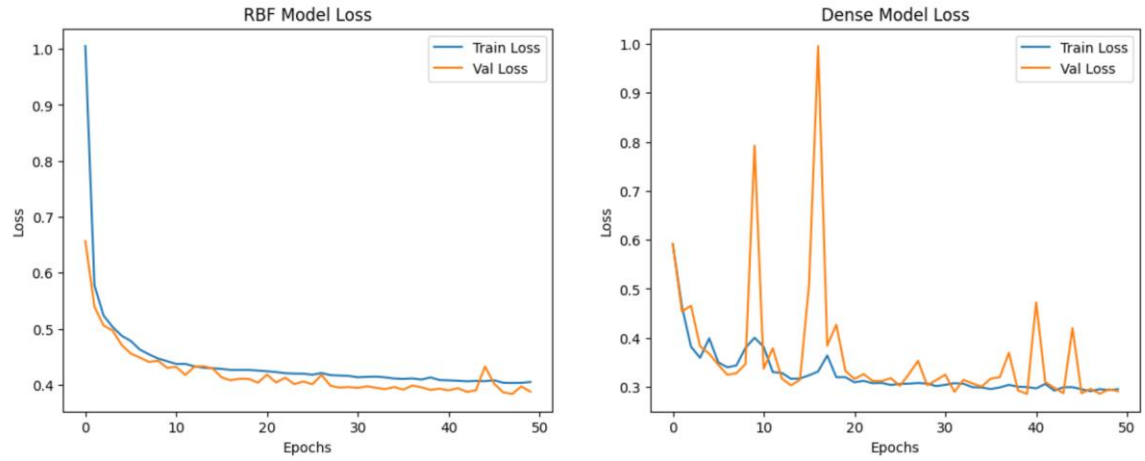
Dense Model Test Loss: 0.3038

سپس به نمایش نمودار خطا هر دو مدل و مقایسه آن ها می پردازیم:

```
# Summarizing the results
print("\nSummary of Results:")
print(f"RBF Model Test Loss: {rbf_loss:.4f}")
print(f"Dense Model Test Loss: {dense_loss:.4f}")

print("Comparing Training and Validation Losses for both models:")

# Plot the training and validation losses
import matplotlib.pyplot as plt

# RBF Model
plt.figure(figsize=(14, 5))
plt.subplot(1, 2, 1)
plt.plot(history_rbf.history['loss'], label='Training Loss')
plt.plot(history_rbf.history['val_loss'], label='Validation Loss')
plt.title('RBF Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Dense Model
plt.subplot(1, 2, 2)
plt.plot(history_dense.history['loss'], label='Training Loss')
plt.plot(history_dense.history['val_loss'], label='Validation Loss')
plt.title('Dense Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

در آخر نمودار های خطای MSE هر دو مدل را به تصویر می کشیم.

```
RBF Test Loss: 0.4152
Dense Test Loss: 0.3038
```



سوال چهارم بخش 7

همانطور که مشاهده می شود خطای نهایی مدل کاملا متصل Dense در آخر به خطای کمتر 0.3038 رسیده و مدل با یک لایه RBF به خطای 0.4152 رسیده که خطای بیشتری نسبت به مدل قبلی می باشد. اما نکته حائز اهمیت که مدل RBF را بهتر از مدل Dense می کند نمودار داده های Validation

می باشد که با توجه به نمودار عملکرد بهتر مدل RBF نسبت به مدل کاملا متصل به راحتی قابل مشاهده می باشد. نوسانات بسیار زیاد در نمودار مدل Dense در داده های اعتبارسنجی حاکی از این است که مدل بسیار خاص منظوره عمل می کند و نسبت به مدل RBF دارای Genralization کمتری می باشد. پس مدل RBF در این مسئله قطعا از مدل Dense بهتر است چون نسبت به داده هایی که تابحال ندیده است در کل عملکرد بهتری داشته است.