

به نام خدا



Fundamentals of intelligent systems

Dr. aliyari

Mini project 2

Amin Elmi Ghiasi

https://github.com/AminElmiGhiasi/ML_course

Alireza Moradmand

Question1:

-۱

برای کلاس بندی دو راه پیشنهاد می شود:

۱) به اندازه کلاس ها نورون داشته اشیم و هر نورون احتمال تعلق هر کلاس را نشان دهیم

۲) یک نورون داشته باشیم و یرای هر کلاس بازه ای از مقدار را در به هر کلاس اخواص دهیم

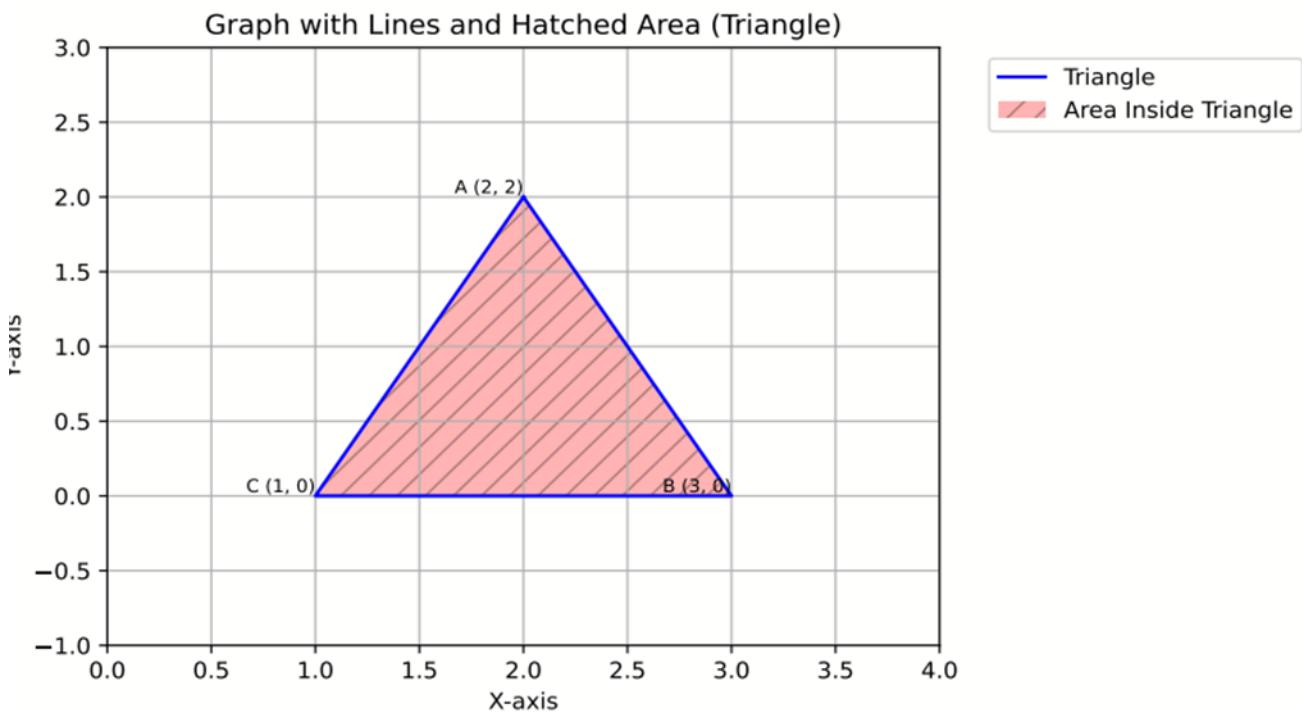
تابع ReLU برای هیچکدام مناسب نیست زیرا طرف مثبت آن کاملا باز است و در هر دو روش پیشنهادی بازه تابع فعالسازی باید بسته باشد تا هر کلاس بدون تبعیض کلاسیندی شوند. در مقابل سیگموئید نه تنها بازه بسته دارد بین ۰ و ۱ است که برای کلاسیندی بسیار مناسب است.

-۲

تابع فعال سازی بسیار کاربردی است که برای از بین بردن مشکل ناپدیدی گرادیان پیشنهاد شده و به خوبی این مشکل را حل کرد. با وجود این مزیت ها این تابع مقادیر منفی را از خود عبور نمی دهد و همه را صفر می کند در طول یادگیری ممکن است وزن ها طوری تغییر کنند که تعداد خاصی از نورون ها در خالت ورودی منفی و خروجی صفر گیر کنند به این پدیده مرگ نورون می گویند نورون های مرده تنها ظرفیت محاسباتی کامپیوتر را اشغال می کنند و تاثیری روی مدل ندارند. برای خل این مشکل توابعی همانند ReLU پیشنهاد شد تا اعداد منفی را تا حد جزئی از خود عبور دهند یکی از این توابع ELU است که نه تنها مزیت جلوگیری از مرگ نورون را دارد قابلیت تنظیم میزان عبور را نیز به کمک آلفا دارد (معادله زیر).

$$\text{ELU}(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

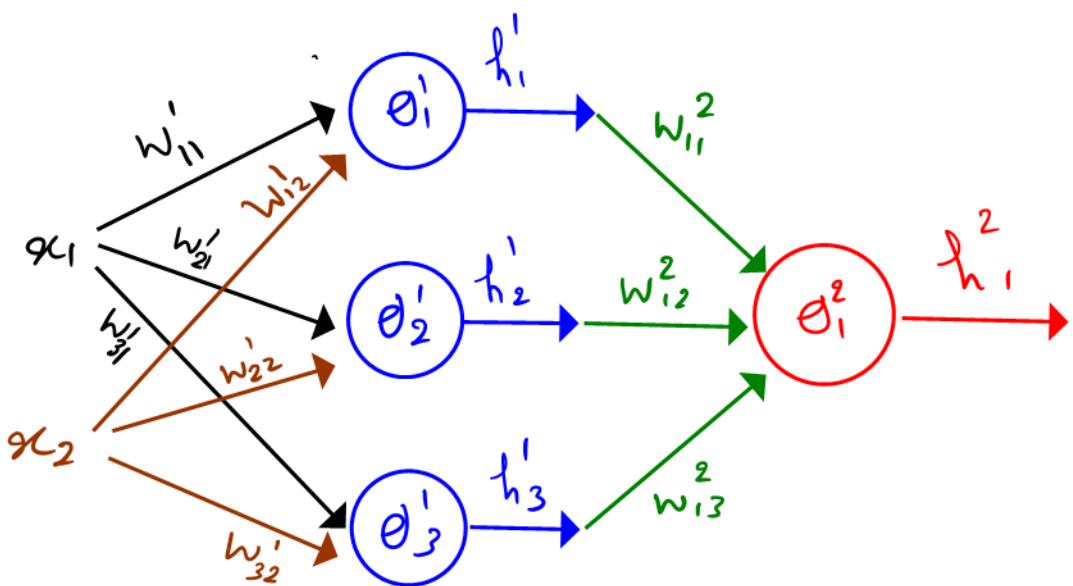
Question 3



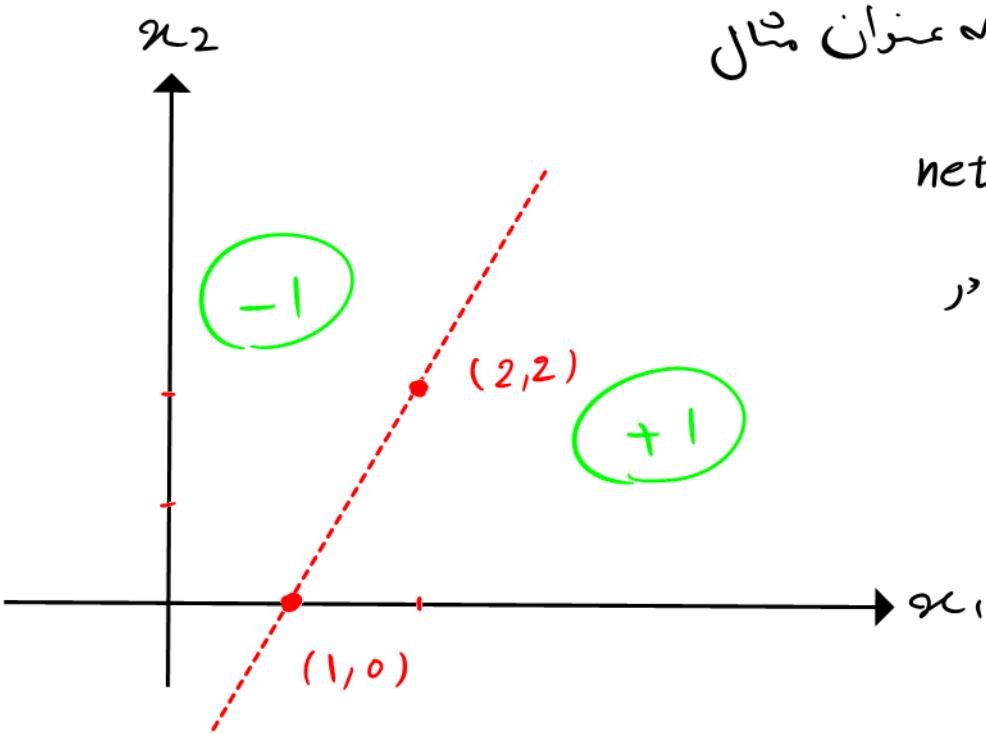
ساختار سلسیم ایکے با نرون classification برای انبعام McCollum-Pitts درین مسئلہ خواهی میں سودہ بھرت کے پہلے بائیک لایے جانی کے ساتھ نرون دار دخواہ ہو۔ ہر نرون لایے پہنچان درواعے عمل classification را برائی کی از افلاع ناصیہ میں تسلیں سے انبعامی دے۔ ہر ضلع این مسئلہ کے را می توانے سے خطر درنک کرنے کے صفحع دو بعدی طبقے دو ناصیہ تقسیم کی کہ و بائیک وزنہای سلسیم راطرزی معابس کئی کے خروجی ناصیہ مطلوب 1 و خروجی ناصیہ نامطلوب 1 سے سودہ۔

لایے خروجی سلسیم شرط درواعے دستار AND اسکے اگر خروجی ہم سے نرون 1 بائیک (کہ بہ معنی این لسکے کے نعم مواد نلم درون ناصیہ میں مدار دارد) در این بھرت خروجی این نرون نیز 1 خواہ ہے۔

رسکل زیر ساختهای این شبکه را می‌دانم



برای به سمت آوردن وزنها و threshold نزدیکی از سریع حل مسئله اصلی مبتدا و نامنفع مغلوب استفاده می‌کنم به عنوان مثال



$$\text{net} = w̑₁x₁ + w̑₂x₂ + b₁$$

به عنوان مثال نعم (2, 1) را در نظر گیرم. خروجی این نعم باشد +1 است.

با وجود بسیار خط:

$$2x₁ - x₂ + b₁ = 4 - 1 + b₁ = 3 + b = +1 \Rightarrow b = -2 \\ \Rightarrow \thetȃ₁ = +2$$

$$w̑₁ = +2$$

$$w̑₂ = -1$$

$$\thetȃ₁ = +2$$

درنتیج:

به همین ترتیب برای دو صفحه دلخواه سرعت محاسبه کنیم و وزنهای آن صورت ناس می‌آیند:

$$w_{21}^1 = -2$$

$$w_{31}^1 = 0$$

$$w_{22}^1 = -1$$

$$w_{32}^1 = 1$$

$$\theta_2^1 = -6$$

$$\theta_3^1 = 0$$

مهم می‌شود وزنهای نهون لایه خروجی سری باشد بتوانیم آن را در هر سه نرون
سود خروجی باشد بزرگتر یا مساوی صفر باشد.

$$\text{net} = w_{11}^2 h_1^1 + w_{12}^2 h_2^1 + w_{13}^2 h_3^1 + b_3$$

$$+1+1+1+b_3=0 \Rightarrow b_3=-3 \Rightarrow \theta_1^2=3$$

$$w_{11}^2 = 1$$

$$w_{12}^2 = 1$$

$$w_{13}^2 = 1$$

$$\theta_1^2 = 3$$

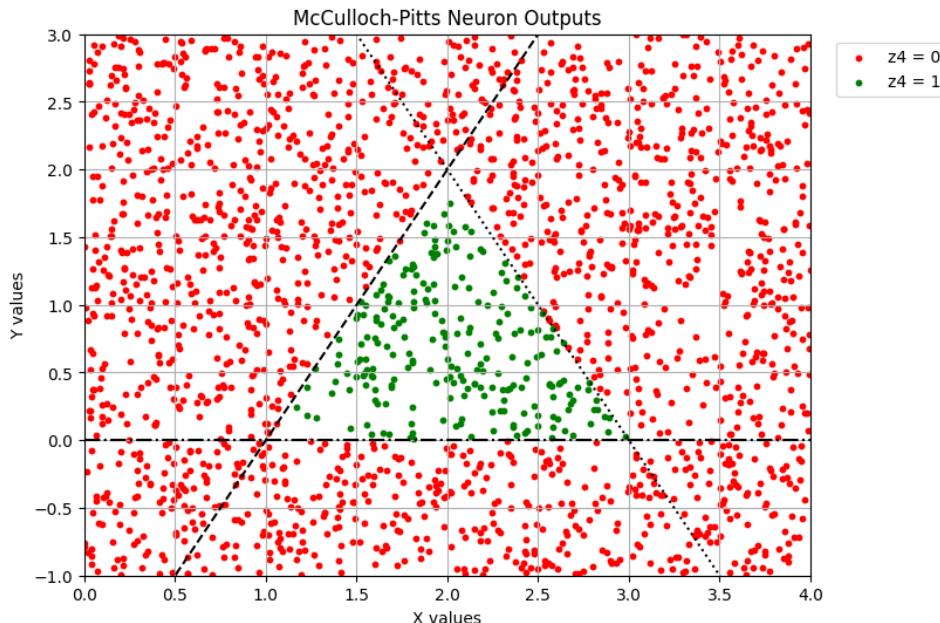
در نهایت نزدیکی مربوط به کلاس مراحلی McCulloch-Pitts را می‌توانیم کنیم.

```
#define model for dataset
def Area(x, y):
    neur1 = McCulloch_Pitts_neuron([2, -1], 2)
    neur2 = McCulloch_Pitts_neuron([-2, -1], -6)
    neur3 = McCulloch_Pitts_neuron([0, 1], 0)
    neur4 = McCulloch_Pitts_neuron([1, 1, 1], 3)

    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 list([z4])
```

نتیجه شبکه طراحی شده به وسیله نرون McCulloch-Pitts به صورت زیر خواهد شد. همانطور که از شکل مشخص است عمل classification به درستی انجام شده است.

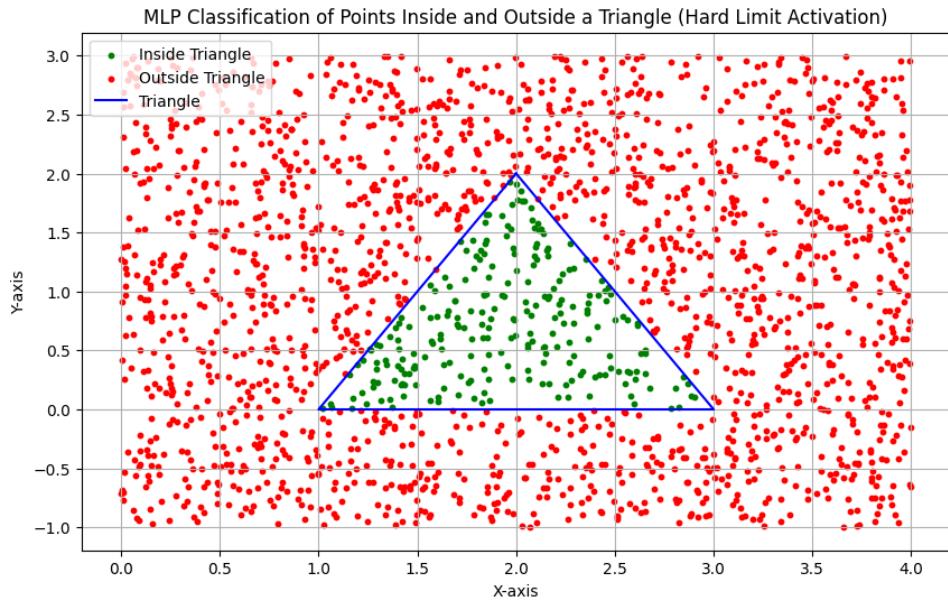


حال در این قسمت یک شبکه MLP با همان معماری شبکه McCulloch-Pitts به وسیله کتابخانه keras یعنی با یک لایه پنخان که سه نرون دارد و یک لایه خروجی طراحی می کنیم.

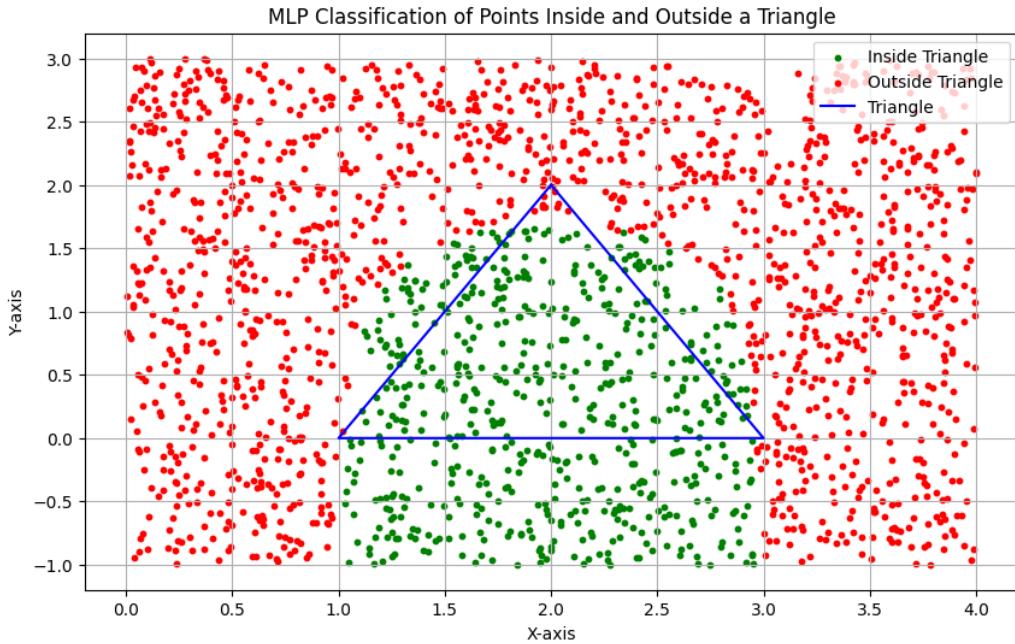
در این شبکه از activation function hardlim استفاده می کنیم و وزنها را نیز به صورت دستی دقیقا برابر با وزنها نرونها McCulloch-Pitts قرار می دهیم. یعنی وزنها این شبکه از طریق یادگیری تعیین نمی شود.

در شکل زیر نیز خروجی این شبکه را مشاهده می کنید که همانطور که مشخص است عمل classification را به درستی انجام داده است.

با توجه به نتایج این دو شبکه که بررسی شد می توان نتیجه گرفت که نرون پرسپترون با تابع فعال ساز تابع پله یا hardlim دقیقا مانند نرون McCulloch-Pitts عمل می کند.



در گام بعدی نیز سعی می کنیم یک شبکه MLP با همان ساختار اما این بار با تابع فعال ساز sigmoid و سعی می کنیم وزنهای این شبکه را نیز بدون فرآیند یادگیری و سعی و خطأ به صورت دستی تنظیم کنیم. بهترین نتیجه ای که از این شبکه بدست آمده است در شکل زیر قابل مشاهده است.



همانطور که مشخص است عمل classification در این شبکه با عملکرد مناسبی انجام نشده است بنابراین برای استفاده از تابع فعال ساز sigmoid حتما برای تنظیم وزنهای باید فرآیند یادگیری انجام شود.

Question2:

۱- به وسیله دستورات زیر اسن مجموعه داده را توسط کتابخانه pandas می خوانیم.

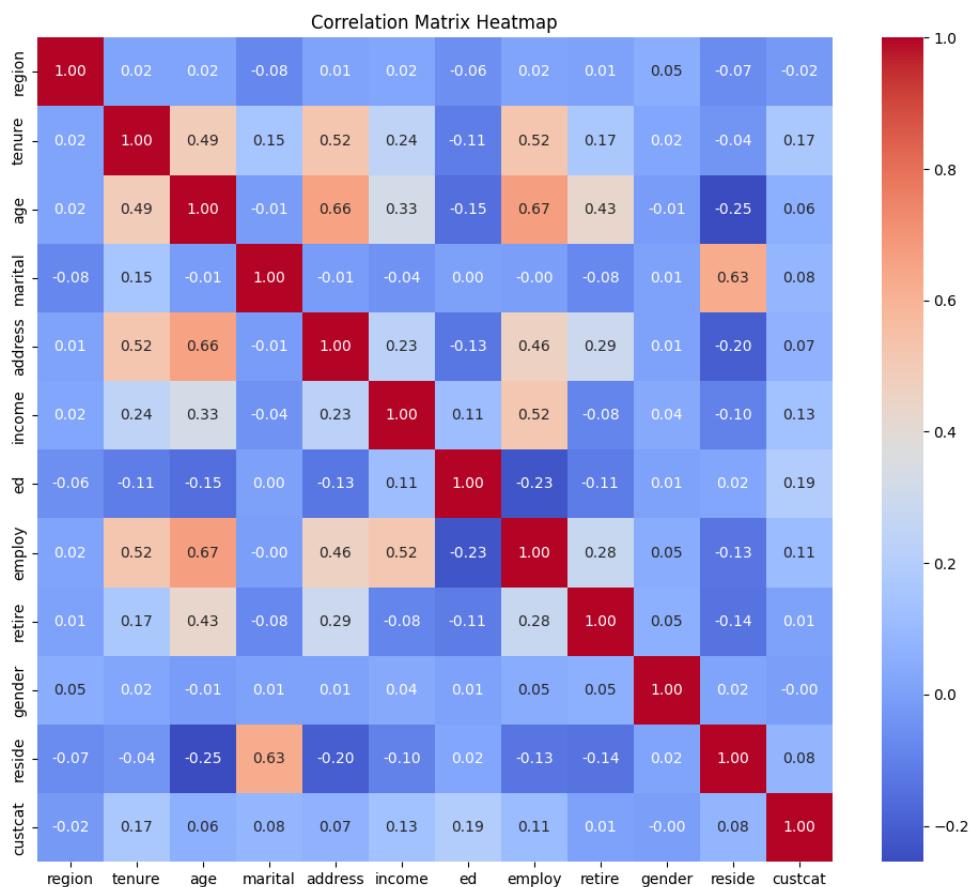
```
import pandas as pd

df = pd.read_csv('/content/telecust_data/teleCust1000t.csv')

df.head() # Display first few rows to verify
```

	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

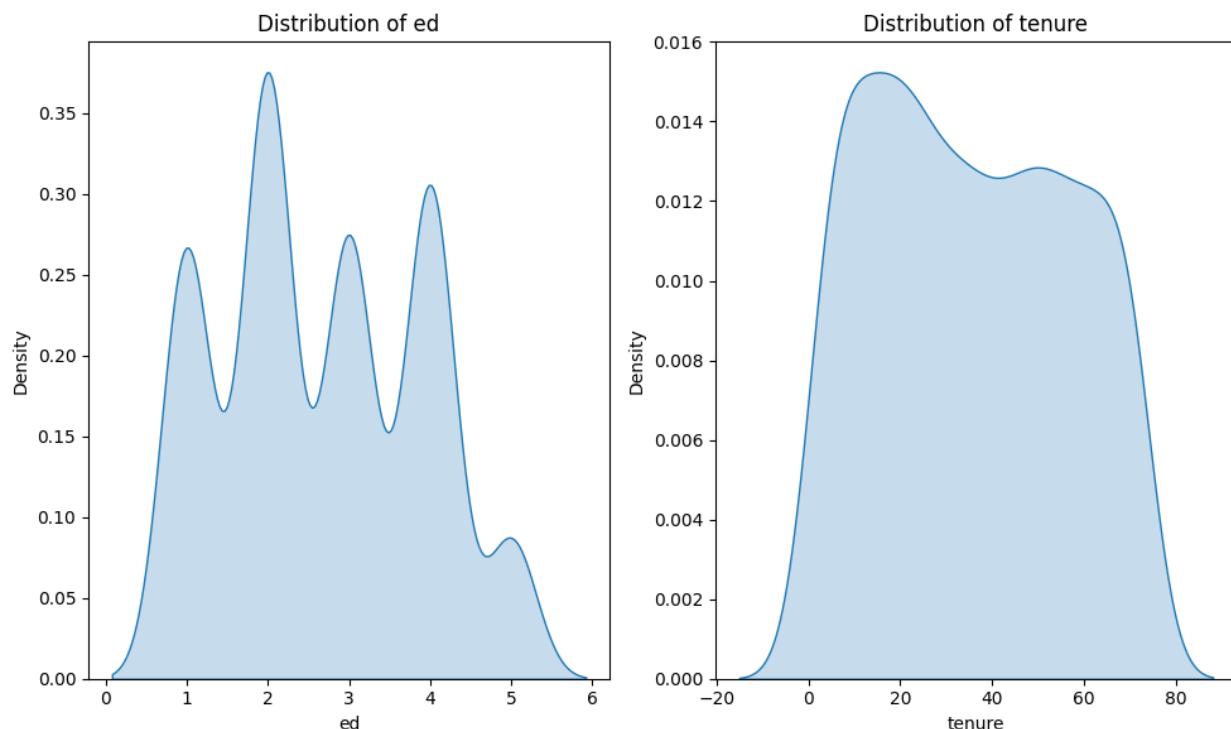
۲- هیت مپ مربوط به correlation داده های این مجموعه بدین صورت است.



همچنین ترتیب مقدار همبستگی داده ها با فیلد هدف نیز اینگونه است.

custcat	
custcat	1.000000
ed	0.193864
tenure	0.166691
income	0.134525
employ	0.110011
marital	0.083836
reside	0.082022
address	0.067913
age	0.056909
retire	0.008908
gender	-0.004966
region	-0.023771

همانطور که مشخص است بیشترین همبستگی با tenure و ed برقرار است و توزیع این دو داده نیز اینگونه است.



```

from sklearn.model_selection import train_test_split

X = df.drop('custcat', axis=1)
y = df['custcat']

# Split data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=42)

print("Training data shape:", X_train.shape)
print("Validation data shape:", X_val.shape)
print("Testing data shape:", X_test.shape)

Training data shape: (720, 11)
Validation data shape: (80, 11)
Testing data shape: (200, 11)

from sklearn.preprocessing import StandardScaler

# Create a MinMaxScaler object
scaler = StandardScaler()

# Transform the training, testing, and validation data
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
X_val = scaler.transform(X_val)

```

همچنین در این قسمت برای افزایش تعداد داده‌ها جهت بهبود عملکرد مدل تعداد داده‌ها را به وسیله اضافه کردن نویز، برای اینکار برای ایجاد تغییرات جزئی، نویز تصادفی کوچکی را به داده‌های موجود اضافه می‌کنیم و تعداد داده‌ها را بیشتر می‌کنیم.

با انجام این کار تعداد داده‌های train حدوداً ده برابر می‌شود.

```

Original data size: 720
Augmented data size: 7920

```

-۴- برای طراحی شبکه و انجام عمل classification بر روی این مجموعه دیتا ابتدا لازم است که بر روی داده خروجی one hot encoding انجام شود.

```

# Convert target variables to one-hot encoding
y_train = to_categorical(y_train-1, num_classes=4)
y_val = to_categorical(y_val-1, num_classes=4)
y_test = to_categorical(y_test-1, num_classes=4) # Added for consistency

```

حال در این مرحله یک شبکه عصبی MLP با یک لایه پنهان و در ابتدا با ۱۱ نرون (برابر با تعداد features مجموعه داده) ایجاد می کنیم. همچنین تعداد نرونهای لایه خروجی نیز چهار نرون خواهد بود زیرا مسئله مورد نظر یک classification با چهار کلاس است.

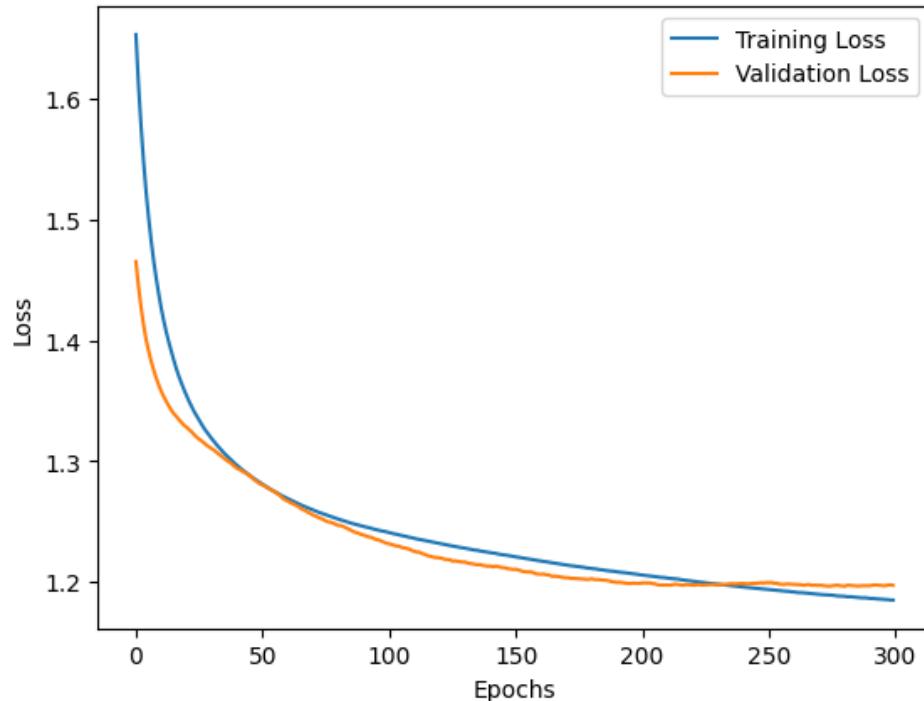
به عنوان تابع فعالساز در لایه پنهان از `relu` و در لایه خروجی از `softmax` استفاده میکنیم.

```
# one hidden layer network
model = Sequential()
model.add(Dense(11, activation='relu', input_shape=(X_train.shape[1],))) # Input layer with 11 neurons
model.add(Dense(4, activation='softmax')) # Output layer (assuming 4 classes)
```

با استفاده از SGD به عنوان بهینه ساز و در epoch ۳۰۰ و مقدار batch size ۵۰ مدل را آموزش می دهیم. نتایج مطابق جدول زیر است.

	Loss	Accuracy
Train	1.1984	43.88%
Validation	1.1971	43.75%
Test	1.2942	40%

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



حال تعداد نرونها را زیاد می کنیم تا تاثیر این مورد را بر روی شبکه با یک لایه پنهان مشاهده کینم.

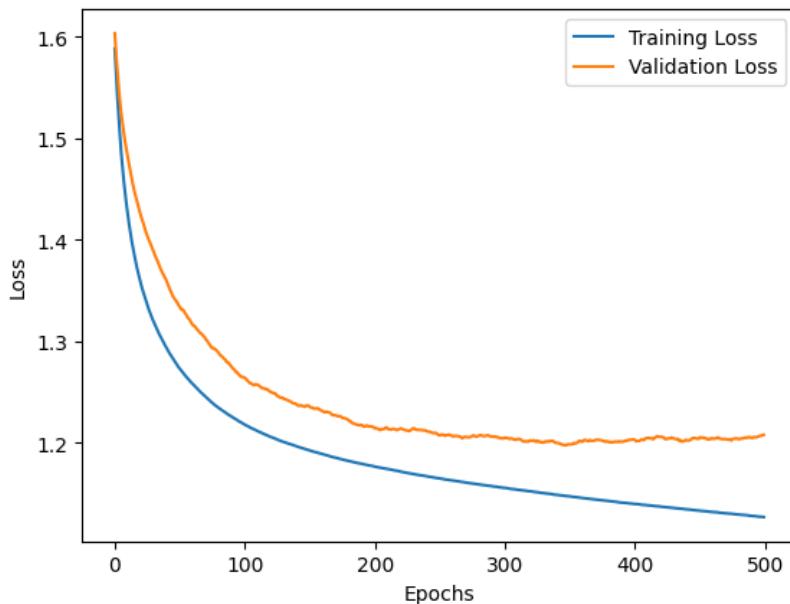
تعداد نرونهای لایه پنهان را به ۵۵ نرون افزایش می دهیم.

آموزش را در ۵۰۰ epoch و با batch-size ۱۰۰ انجام می دهیم.

	Loss	Accuracy
Train	1.14	49.9%
Validation	1.2078	42.5%
Test	1.2972	44%

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.

همنطور که مشخص است نتیجه شبکه با تعداد نرونهای ۴ درصد بهبود یافته است اما شبکه کمی به overfit شدن نیز متمایل شده است.



حال تعداد لایه های مخفی را به دو لایه مخفی افزایش می دهیم.

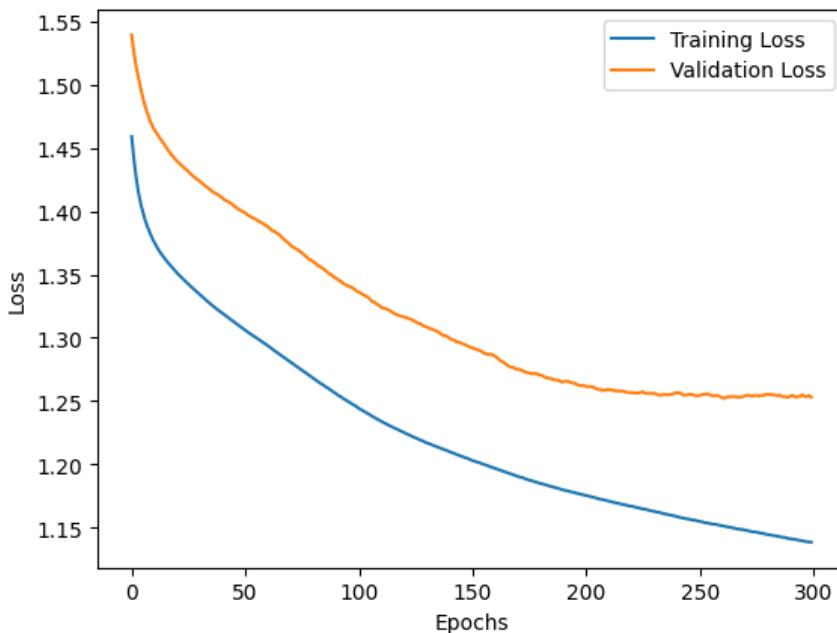
لایه پنهان اول با ۲۲ نرون و لایه پنهان دوم با ۸ نرون (دلیل انتخاب ۸ نرون برای لایه پنهان دوم این است که دو برابر تعداد خروجی ها یعنی ۴ کلاس مورد نظر خواهد بود). تابع فعال ساز نیز مانند حالت قبل برای لایه های میانی relu و برای لایه خروجی softmax خواهد بود.

با استفاده از SGD به عنوان بهینه ساز و در ۳۰۰ epoch و مقدار batch size ۵۰ مدل را آموزش می دهیم. نتایج مطابق جدول زیر است.

	Loss	Accuracy
Train	1.1206	47.18%
Validation	1.2532	46.25%
Test	1.3093	42%

نیجه این شبکه تک لایه پنهان با ۱۱ نرون بهتر است و همچنین تعداد نرونها کمتری نیز نسبت به شبکه تک لایه پنهان با ۵۵ نرون دارد.

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



افزایش تعداد نرون‌ها در یک لایه‌ی شبکه پرسپترون چندلایه (MLP) باعث افزایش ظرفیت مدل برای یادگیری و ذخیره اطلاعات پیچیده‌تر می‌شود. با تعداد بیشتری از نرون‌ها، شبکه می‌تواند الگوها و روابط غیرخطی بیشتری را در داده‌ها مدل‌سازی کند. با این حال، افزایش بیش از حد تعداد نرون‌ها ممکن است منجر به بیش‌برازش (overfitting) شود، به این معنا که مدل به جای یادگیری ویژگی‌های عمومی داده‌ها، نویز یا جزئیات خاص داده‌های آموزشی را یاد می‌گیرد. بنابراین، تنظیم تعداد نرون‌ها بهینه‌سازی مناسبی بین توانایی مدل و جلوگیری از پیچیدگی بیش‌ازحد نیازمند است.

همچنین افزایش تعداد لایه‌ها در شبکه MLP باعث افزایش عمق مدل و توانایی آن در یادگیری ویژگی‌های سلسله‌مراتبی می‌شود. لایه‌های بیشتر امکان استخراج ویژگی‌های سطح پایین در لایه‌های اولیه و ترکیب آن‌ها به ویژگی‌های سطح بالاتر در لایه‌های عمیق‌تر را فراهم می‌کنند. این موضوع به ویژه در مسائل پیچیده و داده‌های با ابعاد بالا بسیار مفید است. اما، افزایش تعداد لایه‌ها ممکن است مشکلاتی نظیر ناپایداری گرادیان

افزایش زمان آموزش، و نیاز به داده‌های آموزشی بیشتر برای جلوگیری از بیشبرازش ایجاد کند. برای مدیریت این چالش‌ها، تکنیک‌هایی مانند نرمال‌سازی دسته‌ای (batch normalization) و استفاده از معماری‌های پیشرفته‌تر ضروری هستند.

در این قسمت لایه نرمال‌ساز دسته (batch normalization layer) را به مدل اضافه می‌کنیم.

لایه نرمال‌سازی دسته‌ای (Batch Normalization) تکنیکی است که به منظور پایدارسازی فرآیند آموزش در شبکه‌های عصبی معرفی شده است. این روش با نرمال‌سازی خروجی‌های یک لایه (فعال‌سازی‌ها) برای هر دسته از داده‌ها (batch)، از تغییرات زیاد در توزیع داده‌ها بین لایه‌ها جلوگیری می‌کند. این تغییرات که به شیفت کواریانس داخلی (Internal Covariate Shift) معروف هستند، می‌توانند باعث کند شدن فرآیند آموزش شوند.

اضافه کردن این لایه به دلیل پایدارتر شدن گرادیان‌ها و یادگیری بهتر باعث افزایش سرعت آموزش می‌شود. نحوه کار این لایه بدین صورت است که خروجی هر لایه پس از اعمال شدن وزن‌ها و بایاس بر روی آن و قبل از ورود به تابع فعال‌ساز نرمالایز می‌شوند. بدینگونه که ابتدا میانگین و واریانس داده‌های دسته حساب می‌شود. فرض کنید \mathbf{x} خروجی لایه قبل است (وزنها و بایاس بر روی \mathbf{x} اعمال شده اند و به آن اصطلاحاً pre-activation می‌گویند):

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i, \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

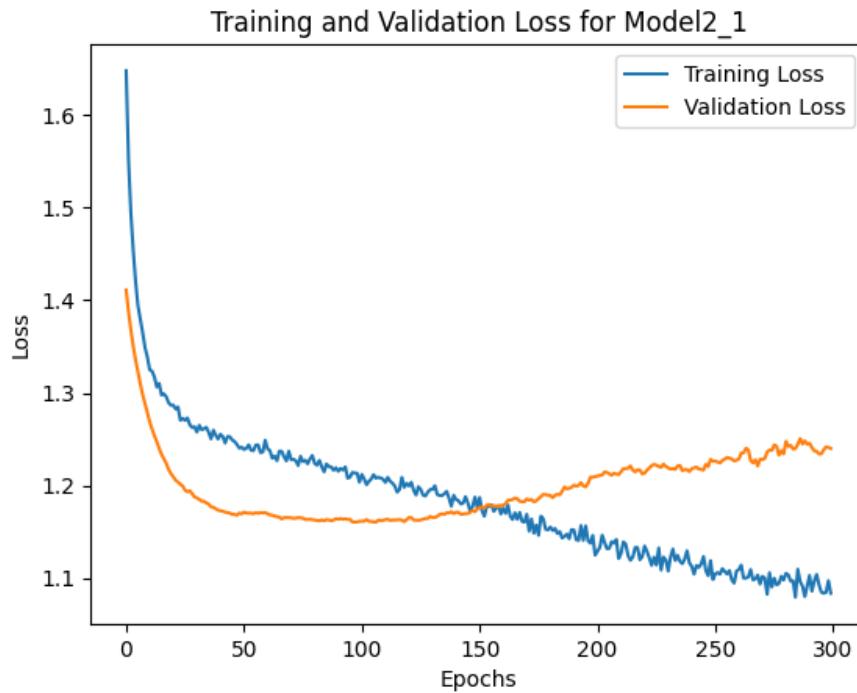
و سپس \mathbf{x} نرمالایز شده اینگونه محاسبه می‌شود.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

با اعمال لایه نرمال‌ساز به شبکه مرحله قبل (دو لایه پنهان) نتایج بدین صورت خواهد بود.

	Loss	Accuracy
Train	1.0714	50.99%
Validation	1.2399	40%
Test	1.4045	39.5%

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



همانطور که در توضیحات قبای گفته شد اضافه کردن لایه نرمالساز باعث افزایش سرعت آموزش می شود و با دقت در نمودار آموزش داده validation شبکه بدون لایه نرمالساز همگرایی در حدود 250 epoch رخ داده است اما با توجه به شکل بالا با اضافه کردن لایه نرمالساز همگرایی در حدود 50 epoch رخ داده است که یعنی آموزش بسیار سریعتر رخ داده و حتی بعد از آن مدل به سمت overfit شدن رفته است که این مورد لزوم استفاده از تکنیک هایی مثل dropout یا early stopping را نشان می دهد.

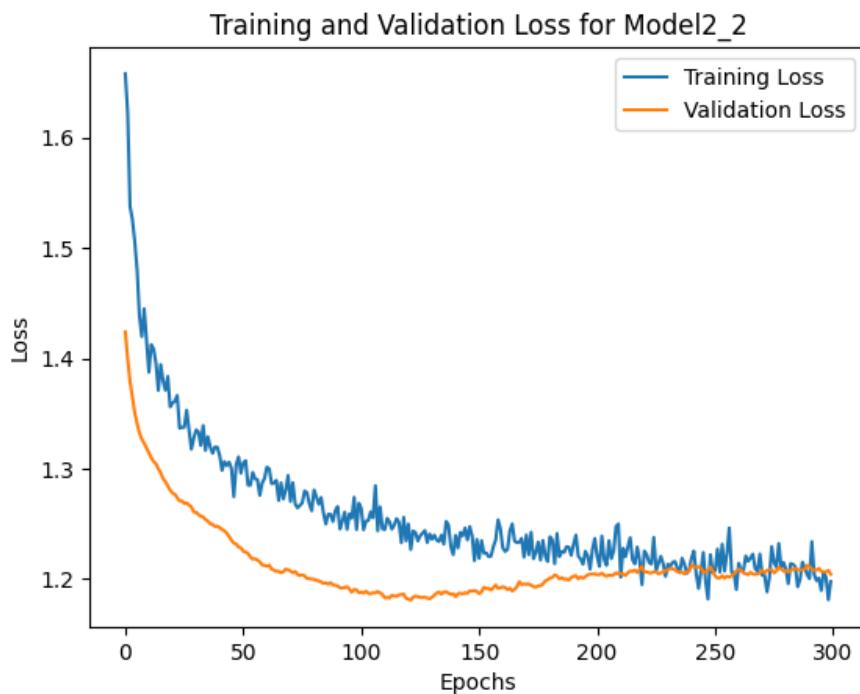
حال در این قسمت به اضافه کردن dropout به شبکه مان می پردازیم.

Dropout (overfitting) یک تکنیک منظم‌سازی (Regularization) است که برای کاهش بیش‌برازش (overfitting) در شبکه‌های عصبی استفاده می‌شود. در این روش، طی فرآیند آموزش، برخی از نرون‌ها به طور تصادفی با احتمال p غیرفعال (یا صفر) می‌شوند. این باعث می‌شود که شبکه از وابستگی بیش از حد به نرون‌های خاص جلوگیری کند و ویژگی‌های متنوع‌تری یاد بگیرد. هنگام پیش‌بینی (مرحله تست)، تمامی نرون‌ها فعال هستند و مقادیر آن‌ها با مقیاس $p-1$ تنظیم می‌شود.

پس از اعمال dropout به مدل بدین صورت که هر بار ۲۰ دصد نرون‌ها غیر فعال شوند. نتایج به صورت زیر خواهد بود.

	Loss	Accuracy
Train	1.1950	47.01%
Validation	1.2044	45%
Test	1.3136	40.5%

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



همانطور که مشخص است overfitting ای که در مرحله قبل با اضافه کردن لایه نرمالساز ایجاد شده بود با اعمال dropout تقریباً به صورت کامل از بین رفته است ولی همچنان سرعت آموزش بیشتر از حالت بدون لایه نرمالساز است و همگرایی داده validation در حدود 100 epoch انجام شده است.

در قسمت بعدی L2-Regularization را با نرخ 0.0001 به مدل اضافه می کنیم.

نیز (Weight Decay) یا وزن‌زدایی (Ridge Regression) که با نام‌های L2-Regularization شناخته می‌شود، تکنیکی برای جلوگیری از بیش‌برازش (Overfitting) در شبکه‌های عصبی است. این روش با اضافه کردن یک جمله منظم‌سازی به تابع هزینه، از بزرگ شدن مقادیر وزن‌ها جلوگیری می‌کند. هدف این است که مدل به جای تمرکز بر پیچیدگی بیش از حد، به دنبال راه حل‌های ساده‌تر باشد که توانایی تعمیم‌دهی بهتری دارند.

تابع هزینه کلی با تنظیم L2 به صورت زیر تعریف می‌شود:

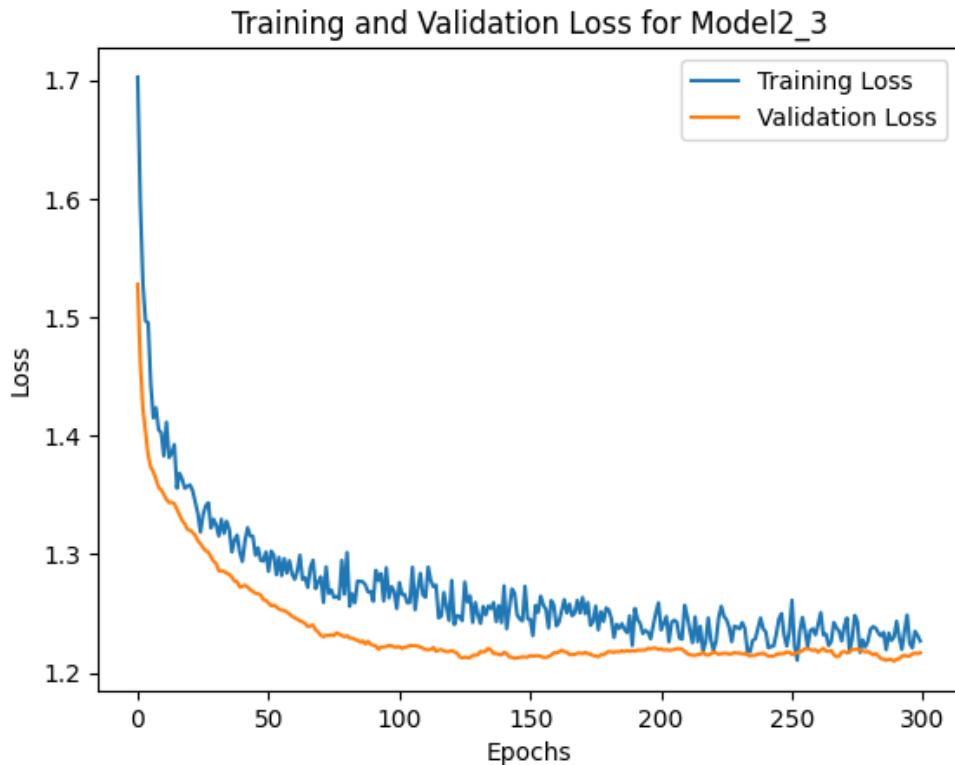
$$L_{\text{total}} = L_{\text{original}} + \lambda \cdot \frac{1}{2} \sum_i w_i^2$$

جمله $\lambda \cdot \frac{1}{2} \sum_i w_i^2$ وزن‌های بزرگ را با اضافه کردن مربع مقادیر آن‌ها به هزینه جریمه می‌کند.

در حین آموزش، وزن‌ها با استفاده از گرادیان نزولی به روزرسانی می‌شوند. با تنظیم L2، قانون به روزرسانی وزن‌ها به صورت زیر تغییر می‌کند:

	Loss	Accuracy
Train	1.2229	44.07%
Validation	1.2168	41.25%
Test	1.2994	38.5%

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



همانطور که از شکل مشخص است مدل به هیچ عنوان دچار overfit نشده است.

در این مرحله بهینه ساز را از SGD و ADAM تغییر می دهیم. در این مراحل از 12- dropout و regularization استفاده شده است.

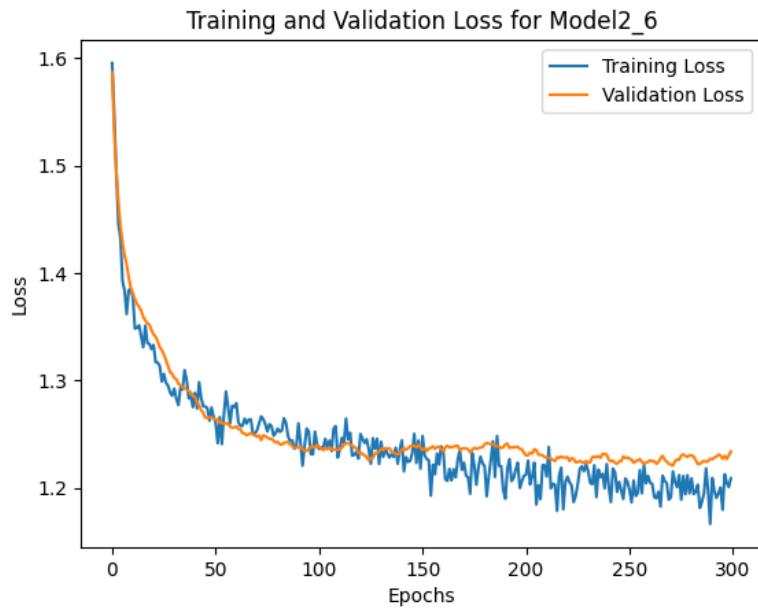
(Root Mean Square Propagation) یک الگوریتم بهینه سازی است که برای پایدارسازی و تسریع فرآیند یادگیری در شبکه های عصبی طراحی شده است. این الگوریتم با استفاده از میانگین متحرک مربع گرادیان ها، نرخ یادگیری را برای هر پارامتر به طور جداگانه تنظیم می کند. این ویژگی باعث می شود که نرخ یادگیری برای پارامترهایی که گرادیان های بزرگی دارند کاهش یابد و برای پارامترهایی با گرادیان های کوچک حفظ شود، که به جلوگیری از نوسانات زیاد و همگرا بی سریع تر کمک می کند RMSprop. به دلیل مدیریت مؤثر گرادیان ها و سازگاری با اهداف غیر ایستا، برای مسائل یادگیری عمیق، به ویژه شبکه های بازگشتی (RNNs)، بسیار مناسب است.

الگوریتم بهینه سازی Adam یکی از روش های پیشرفته و پر کاربرد در یادگیری عمیق است که با ترکیب مزایای الگوریتم های RMSProp و Momentum کار می کند. این الگوریتم با استفاده از میانگین متحرک گرادیان ها و مربع گرادیان ها، نرخ یادگیری را برای هر پارامتر به صورت تطبیقی تنظیم می کند Adam. با جلوگیری از نوسانات زیاد و تطبیق با تغییرات داده ها، به پایداری و سرعت بیشتر در فرآیند آموزش کمک می کند. این الگوریتم به دلیل عملکرد خوب در مسائل با داده های پراکنده یا نویزی، و همچنین کارایی بالا در شبکه های عصبی پیچیده، به یکی از روش های محبوب بهینه سازی تبدیل شده است.

ابتدا بهینه ساز را به RMSprop تغییر می دهیم و نتایج به صورت زیر خواهد بود.

	Loss	Accuracy
Train	1.1793	47.93%
Validation	1.2340	40%
Test	1.3105	40%

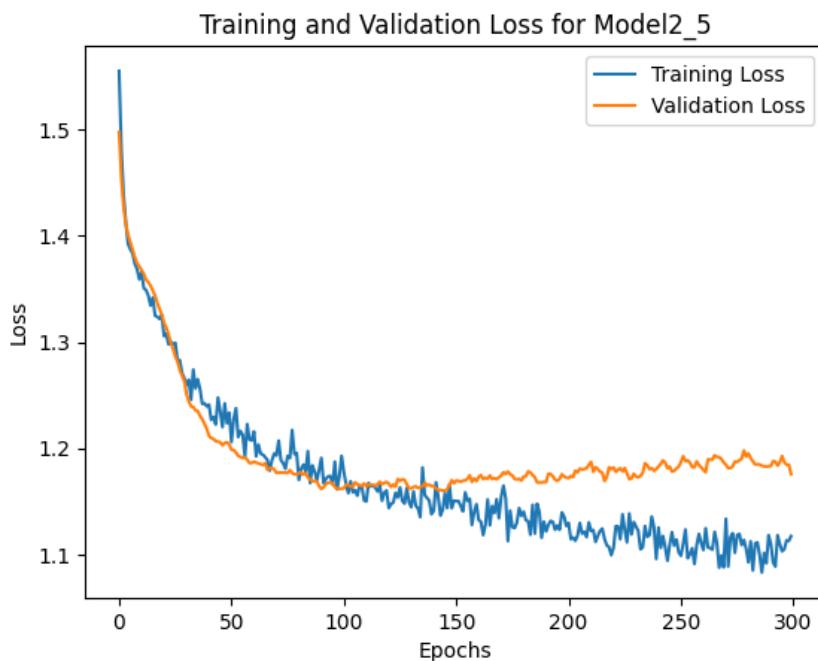
همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



نتایج اعمال بهینه ساز ADAM نیز بدینگونه خواهد بود.

	Loss	Accuracy
Train	1.1218	48.57%
Validation	1.1758	38.57%
Test	1.3570	40.5%

همچنین نمودار Loss برای داده train و test نیز به صورت زیر است.



۵- نتایج ارزیابی مدلها بر روی داده تست در قسمت قبل بیان شده است.

۶- راه های ترکیب دو مدل :

یادگیری جمعی(Ensemble Learning)

یادگیری جمعی شامل ترکیب چندین مدل برای بهبود دقت پیش‌بینی است. در روش **Bagging**، مدل‌ها با استفاده از زیرمجموعه‌های تصادفی از داده‌ها آموزش می‌بینند و خروجی‌ها با رأی‌گیری اکثریت (برای طبقه‌بندی) یا میانگین‌گیری (برای رگرسیون) ترکیب می‌شوند. در **Boosting**، مدل‌ها به صورت ترتیبی آموزش داده می‌شوند، به طوری که هر مدل خطاهای مدل قبلی را اصلاح می‌کند و خروجی‌ها با وزن‌دهی ترکیب می‌شوند.

مدل‌سازی پشت‌های (Stacking)

در این روش، چندین مدل پایه مانند دو MLP آموزش داده می‌شوند و خروجی‌های آن‌ها به عنوان ورودی به یک "متا-مدل" داده می‌شوند. متا-مدل یاد می‌گیرد که چگونه بهترین تصمیم را بر اساس پیش‌بینی‌های مدل‌های پایه بگیرد. این روش برای ترکیب مدل‌های مختلف و بهبود عملکرد کلی بسیار موثر است.

پردازش موازی(Parallel Processing)

در این روش، دو مدل به صورت موازی و با یک ورودی مشابه اجرا می‌شوند. سپس خروجی‌های آن‌ها با استفاده از روش‌هایی مانند میانگین‌گیری، جمع وزن‌دار، یا یک ترکیب یادگرفته‌شده ترکیب می‌شوند. این روش به بهبود دقت و پایداری مدل کمک می‌کند.

Question3:

:convertImageToBinary تابع توضیحات

این روش با تجزیه و تحلیل شدت هر پیکسل، یک تصویر را به یک نمایش باینری تبدیل می کند. شدت کل هر پیکسل (مجموع مقادیر RGB آن) را محاسبه می کند و آن را با آستانه ای که بر اساس نقطه میانی شدت پیکسل های ممکن با یک ضریب تنظیم قابل تنظیم تعیین می شود، مقایسه می کند. پیکسل هایی با شدت بالاتر از آستانه به عنوان سفید طبقه بندی می شوند و مقدار باینری ۱- به آن ها اختصاص می یابد، در حالی که پیکسل های زیر آستانه به عنوان سیاه طبقه بندی می شوند و به ۰ اختصاص می یابند. طبقه بندی باینری فقط با پیکسل های سیاه و سفید خواهد بود.

نحوه کار کد به این صورت است:

- عرض و ارتفاع تصویر را ارائه می دهنده image.size[1] و image.size[0].
- داده های پیکسل را برای تصویر بارگذاری می کند و با استفاده از مختصات pix = image.load()
- (y, x) به مقادیر پیکسلی منفرد دسترسی پیدا می کند.
- یک عامل آستانه (factor) تعریف شده است که بر مرز تصمیم گیری برای طبقه بندی یک پیکسل به عنوان "سفید" یا "سیاه" تأثیر می گذارد.
- برای هر پیکسل داریم:

 - مقادیر قرمز R، سبز G و آبی B استخراج می شوند
 - شدت کل پیکسل به صورت جمع مقادیر آبی سبز قرمز محاسبه می شود.
 - آستانه تصمیم گیری در مورد سیاه یا سفید بودن یک پیکسل به صورت $((255 + factor) // 2)$ می شود.
 - این مقدار نقطه میانی شدت پیکسل ممکن است (با مقداری تنظیم بر اساس factor).
 - اگر شدت کل یک پیکسل از آستانه فراتر رود پیکسل سفید در نظر گرفته می شود و مقدار باینری آن ۱- است. مقادیر RGB پیکسل روی (R, G, B) تنظیم می شود.
 - در غیر این صورت پیکسل سیاه در نظر گرفته می شود و مقدار باینری آن ۰ است. مقادیر RGB پیکسل روی $(0, 0, 0)$ تنظیم می شود.

توضیح تابع `getNoisyBinaryImage`

این روش با افزودن تغییرات تصادفی به مقادیر RGB هر پیکسل در محدوده مشخصی که توسط یک فاکتور نویز کنترل می شود، نسخه های نویزدار تصاویر را ایجاد می کند. برای هر پیکسل، یک مقدار نویز تصادفی تولید می شود و به اجزاء رنگ آن اضافه می شود، تا اطمینان حاصل شود که مقادیر در محدوده معتبر (۰-۲۵۵) باقی مانند. این فرآیند برای هر پیکسل در تصویر تکرار می شود و تصویر اصلاح شده به عنوان یک فایل جدید ذخیره می شود. نتیجه یک تصویر بصری پر سر و صدا است که می تواند برای اهداف آزمایش یا افزایش داده استفاده شود.

نحوه کار کد به این صورت است:

- داده های پیکسل از طریق `pix = image.load()` برای بازیابی مقادیر تک تک پیکسل ها قابل دسترسی است.
- یک ضریب نویز مشخص شده است `noise_factor = 200`، که محدوده نویز تصادفی اعمال شده به هر پیکسل را کنترل می کند.
- این تابع با استفاده از حلقه های تو در تو در عرض و ارتفاع هر پیکسل در تصویر حلقه می زند. و برای هر پیکسل:

 - یک مقدار نویز تصادفی با استفاده از `random.randint(-noise_factor, noise_factor)` تولید می شود.
 - این مقدار نویز به اجزای قرمز R، سبز G و آبی B پیکسل اضافه می شود.
 - از آنجایی که مقادیر RGB باید در محدوده [۰، ۲۵۵] باقی بمانند، هر پیکسلی که به آن نویز اضافه شده است به این صورت تنظیم می شود که اگر مقدار زیر ۰ باشد، روی ۰ تنظیم می شود. اگر مقدار بالاتر از ۲۵۵ باشد، روی ۲۵۵ تنظیم می شود.

۲- شبکه هاپفیلد نوعی recurrent artificial neural network است که توسط جان هاپفیلد در سال ۱۹۸۲ معرفی شد. این شبکه به عنوان یک سیستم حافظه آدرس پذیر محتوا با گره های آستانه باینری طراحی شده است. کاربرد اصلی شبکه های هاپفیلد در حافظه انجمانی (associative memory) است، جایی که الگوهای ذخیره می کنند و زمانی که با ورودی های نویز یا ناقص ارائه می شوند، آنها را به خاطر می آورند.

شبکه ها پفیلد از مجموعه ای از نورون های به هم پیوسته تشکیل شده است که هر نورون هم ورودی و هم خروجی است. همه نورون ها به طور کامل به هم متصل هستند، اما هیچ ارتباطی وجود ندارد (یعنی یک نورون به خودش متصل نیست).

شبکه ها پفیلد می تواند الگوها را با تنظیم وزن اتصالات خود ذخیره کند. هنگامی که ورودی مشابه یک الگوی ذخیره شده ارائه می شود، شبکه به نزدیک ترین الگوی ذخیره شده همگرا می شود و به عنوان یک تشخیص دهنده الگو یا یک بازیابی عمل می کند.

شبکه با به حداقل رساندن یک تابع انرژی عمل می کند. وضعیت شبکه در طول زمان برای رسیدن به یک حالت پایدار (حداقل محلی تابع انرژی) تکامل می یابد. این حالت های پایدار با الگوهای ذخیره شده مطابقت دارند.

نورون های شبکه حالت های خود را به صورت ناهمzman یا همزمان به روز می کنند. وضعیت هر نورون بر اساس تابع آستانه ای تعیین می شود که به مجموع وزنی ورودی های نورون های دیگر بستگی دارد.

نحوه کار شبکه به این صورت است:

- در طول مرحله آموزش، الگوها با به روز رسانی وزن بین نورون ها با استفاده از یک قانون یادگیری خاص، معمولاً قانون یادگیری Hebb، در شبکه کدگذاری می شوند.
- وزن ها متقارن هستند ($w_{ij} = w_{ji}$) و اطمینان می دهند که شبکه می تواند به عنوان یک سیستم دینامیکی با حالت های پایدار عمل کند.
- پس از آموزش، شبکه می تواند الگوها را بازیابی کند. هنگامی که یک الگوی ناقص یا پرنویز به عنوان ورودی ارائه می شود، شبکه به طور مکرر حالت های نورون را به روز می کند تا به یک حالت پایدار مطابق با نزدیک ترین الگوی ذخیره شده برسد.
- هنگامی که همه نورون ها در حالت های ثابت قرار می گیرند، شبکه به حالت پایدار می رسد. این ثبات نشان دهنده یک الگو یا حافظه ذخیره شده است.

این شبکه می تواند تا $0.15N$ الگوها را ذخیره کند که در آن N تعداد نورون ها است. فراتر از این، عملکرد شبکه به دلیل تداخل بین الگوها کاهش می یابد.

حالت هر نورون i دودویی است، معمولاً ۱ (فعال) یا ۰ (غیرفعال).

انرژی E شبکه ها پفیلد توسط رابطه زیر محاسبه می شود:

$$E = -\frac{1}{2} \sum_i \sum_j w_{ij} s_i s_j + \sum_i \theta_i s_i$$

w_{ij} : weights of connections

s_i : states of neurons

θ_i : biases of each neuron

شبکه برای به حداقل رساندن این تابع انرژی تکامل می یابد. حالت های پایدار با حداقل محلی E مطابقت دارد.

وضعیت نورون i بر اساس مجموع وزنی ورودی ها به روز می شود:

$$s_i(t+1) = \text{sign} \left(\sum_j w_{ij} s_j(t) - \theta_i \right)$$

این قانون به روزرسانی نورون ها را تضمین می کند تا انرژی کلی شبکه را کاهش دهد.

الگوها با استفاده از قانون Hebbian ذخیره می شوند:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^P \xi_i^\mu \xi_j^\mu$$

N : Number of neurons.

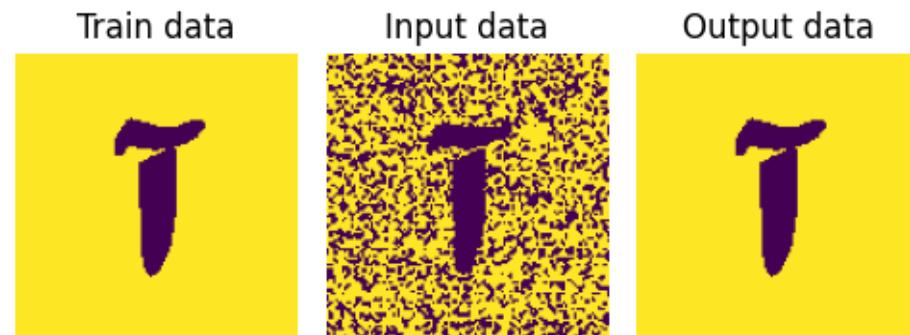
P : Number of patterns to store.

ξ_i^μ : State of neuron i in pattern μ .

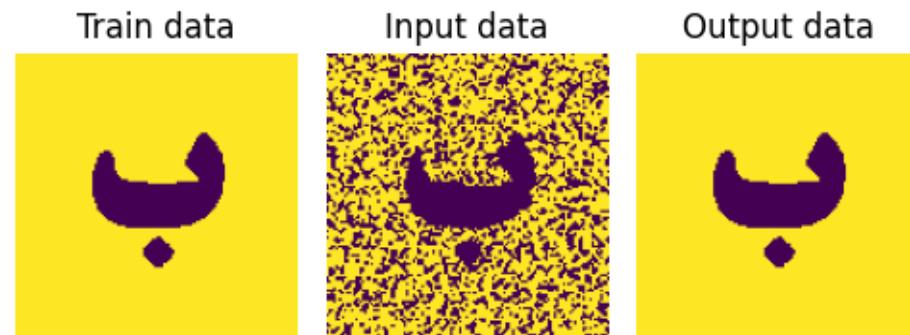
حال در این قسمت هر یک از عکسها را به صورت جداگانه به عنوان ورودی به شبکه ها پفیلد وارد می کنیم تا شبکه وزنها و بایاس هارا مطابق ورودی مورد نظر تعیین کند و پروفایل تابع انرژی مورد نظر را مطابق الگو هر عکس شکل دهد تا در ادامه با ارائه عکس نویز دار به شبکه بتواند با شروع از یک نقطه اولیه به سمت الگو بازشناخته شده همگرا شود.

نتایج آموزش و تست برای هر یک از حروف به صورت زیر است:

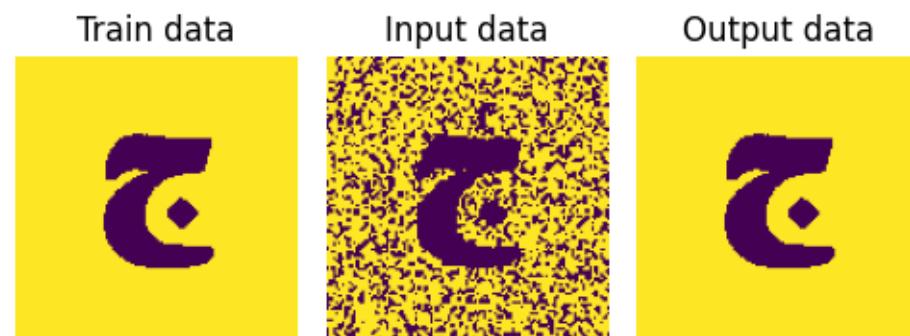
برای حرف آ:



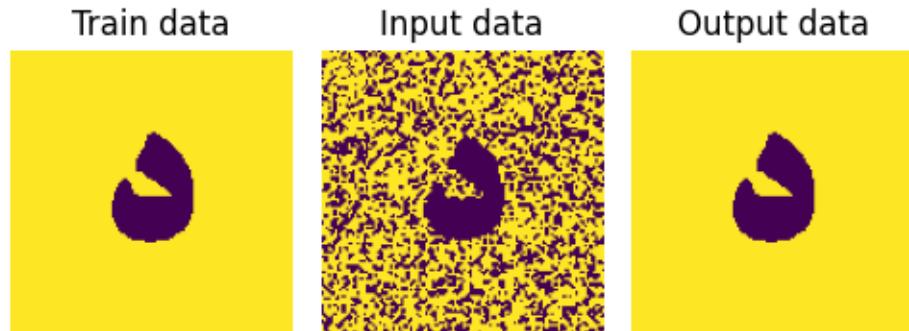
برای حرف ب:



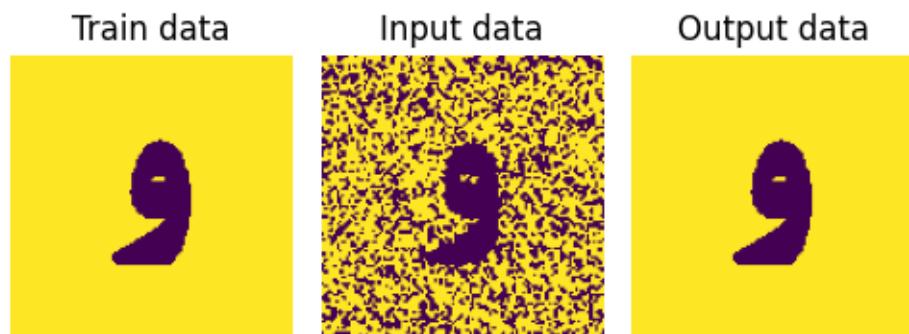
برای حرف ج:



برای حرف د:

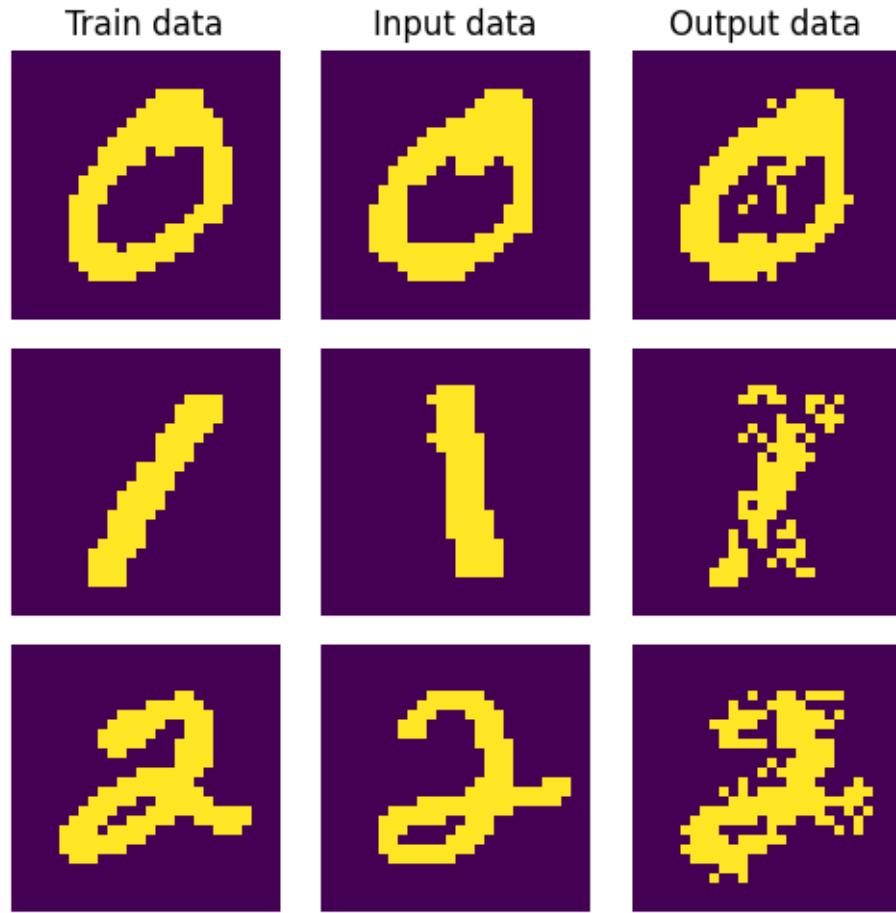


و در نهایت برای حرف و:



شبکه های فیلید استفاده شده در این قسمت به دلیل سادگی توانایی به خاطر سپردن الگوهای این حروف به صورت جمعی و با هم را ندارد و لازم است هر عکس به صورت جداگانه به آن آموزش داده شود اما اگر ورودی های ساده تری از حروف الفبا در آن استفاده کنیم توانایی به خاطر سپردن چند الگو به صورت همزمان را نیز دارد.

در اینجا به عنوان مثال از داده های تصویری سه عدد اول که از دیتابیس **mnist** آورده شده اند استفاده همی کنیم و همانطور که مشاهده می کنید شبکه میتواند چند الگوی ساده را بخاطر بسیار داد و آنها را بازیابی کند.



روش دوم در طراحی شبکه : Hopfield

در این قسمت یک رویکرد دیگر را نیز برای پیاده سازی شبکه ها پیشنهاد استفاده کرده ایم.

شرح کلی کد

1. HopfieldNetwork

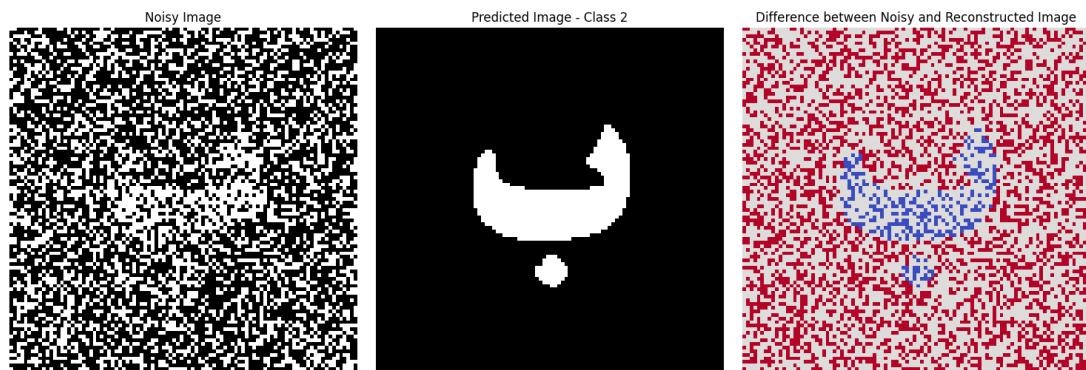
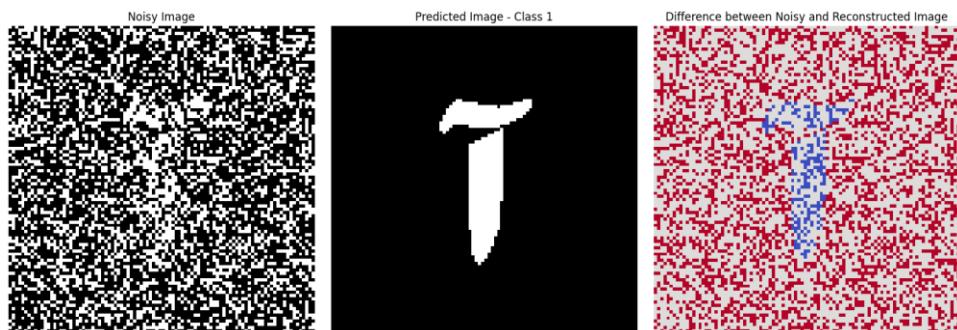
- مدیریت ماتریس وزن W و تعداد نورونها.
- آموزش: ماتریس وزن را با استفاده از الگوهای ورودی به روزرسانی می کند.
- پیش‌بینی: با استفاده از یک الگوی ورودی، حالات نورون‌ها را به‌طور تکراری به روزرسانی می کند تا به یک الگوی ذخیره‌شده همگرا شود.

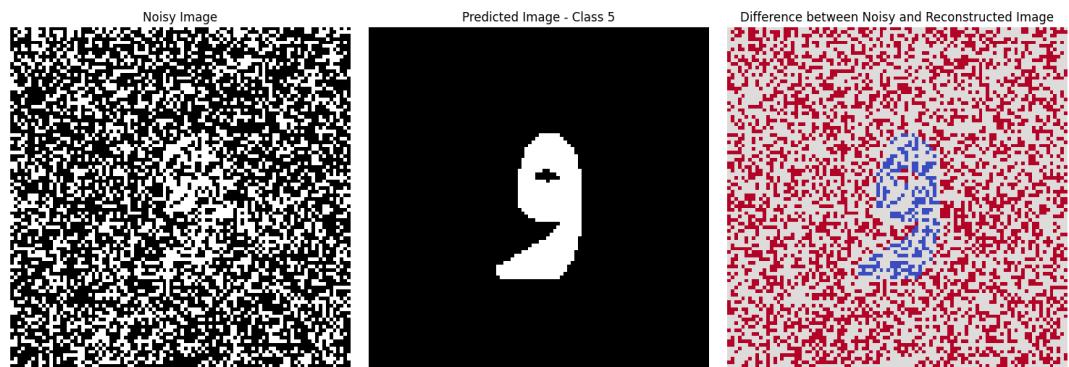
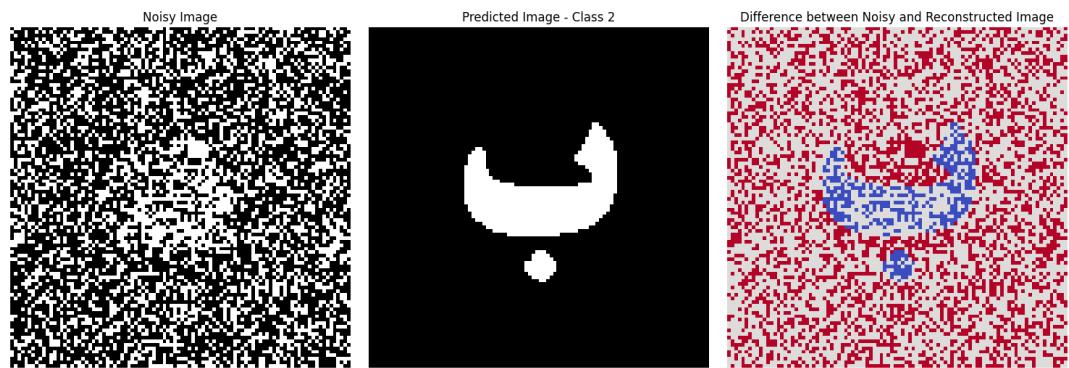
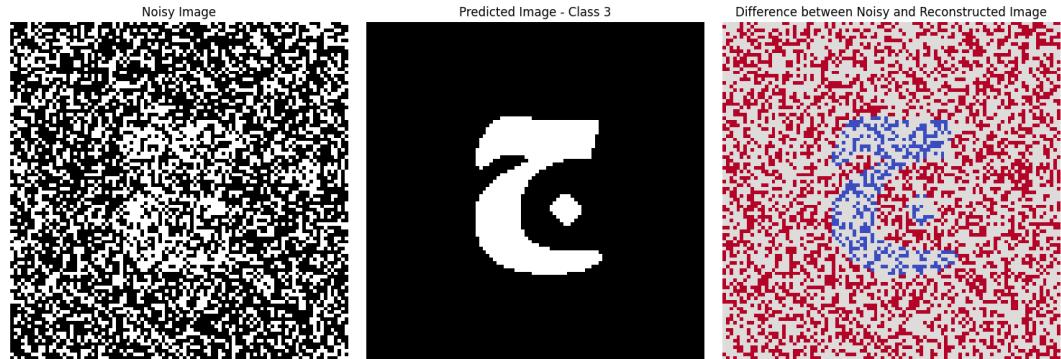
- محاسبه انرژی : انرژی یک الگو را محاسبه می کند که نشان دهنده پایداری الگو در مدل ها پفیلد است.

match_class .۲

- الگوی پیش بینی شده را با یکی از الگوهای آموزشی تطبیق می دهد و شباهت (فاصله همینگ) را محاسبه می کند.

پس از مرحله مدل سازی نویز تا حدی زیاد می شود تا اولین خطا در تشخیص رخ دهد این اتفاق در $\text{noise_factor} = 500$ رخ می دهد. در تصاویر زیر به ترتیب از چپ به راست تصویر ورودی تصویر ساخته شده و تغییراتی که شبکه بر روی تصویر اعمال کرده را می توانیم مشاهده کنیم. در این حالت شبکه نتوانست حرف "د" را به درستی تشخیص دهد (تصویر چهارم). از روی شکل هم می توان دید که شبکه در محدوده تصویر حرف تغییرات زیادی اعمال کرده برای مثال خود به خود نقطه را ساخته در صورتی که واضح است حرف "د" نقطه ندارد ولی سایر تصاویر تقریبا به صورت یکنواخت تغییر کردند.





توضیح عملکرد کد

این کد برای حذف پیکسل طراحی شده است و نتایج را به صورت تصویری نمایش می‌دهد. این کد از تولید کننده تصویر نویزی الهام گرفته شده اما بر تغییر انتخابی پیکسل های سیاه به سفید تمرکز دارد تا از بین رفتن داده پیکسل ها را شبیه سازی کند.

۱. عملکرد:

- تبدیل تصویر به باینری : مانند قبل پیکسل های تصویر را به فرمت باینری تبدیل می کند.
- ایجاد تصاویر با پیکسل های حذف شده : به صورت تصادفی پیکسل های سیاه را با سفید جایگزین می کند.
- نمایش تصاویر اصلی و تغییر یافته: تصویر اصلی و نسخه تغییر یافته آن را در کنار هم نمایش می دهد.

۲. فرآیند:

- لیستی از مسیر تصاویر ورودی دریافت می کند.
- حذف پیکسل ها را با درجات مختلف (مثالاً ۱۰٪، ۳۰٪ و غیره) اعمال می کند.
- تصاویر تغییر یافته را ذخیره می کند.
- مقایسه ای تصویری از تصویر اصلی و تغییر یافته ارائه می دهد.

Question4:

لایه تابع پایه شعاعی (RBF: Radial Basis Function) یک لایه در یک شبکه عصبی است که از توابع پایه شعاعی که معمولاً در تئوری تقریب استفاده می‌شود الهام گرفته شده است. لایه RBF معمولاً برای مسائلی مانند طبقه بندی، تقریب تابع و درون یابی استفاده می‌شود.

مشخصه تعیین کننده لایه RBF استفاده از تابع پایه شعاعی به عنوان تابع فعال سازی است. بر خلاف سایر توابع فعال سازی رایج (به عنوان مثال، ReLU، سیگموئید)، تابع پایه شعاعی به فاصله بین یک بردار ورودی و مجموعه‌ای از نقاط مرکزی (که به آنها centroids نیز می‌گویند) بستگی دارد. این باعث می‌شود که لایه RBF به ویژه در یادگیری ویژگی‌های محلی در فضای ورودی ماهر باشد.

هر نرون در لایه RBF :

- بردار مرکزی که نشان دهنده یک نقطه در فضای ورودی است دارد.
- تابع پایه شعاعی، معمولاً گاووسی، که میزان "نزدیک بودن" ورودی به مرکز را اندازه گیری می‌کند.
- وزنی که همراه با خروجی RBF برای کمک به پیش‌بینی شبکه استفاده می‌شود.

در عمل، لایه RBF فاصله بین بردار ورودی و هر مرکز را محاسبه می‌کند، این فاصله را از تابع پایه شعاعی عبور می‌دهد تا یک مقدار تولید کند، و نتایج را با استفاده از وزن‌ها برای تولید خروجی ترکیب می‌کند. این به شبکه اجازه می‌دهد تا ورودی‌ها را به خروجی‌ها بر اساس مجاورت نگاشت کند، و به ویژه برای مشکلاتی با الگوهای پیچیده و محلی مناسب است.

نحوه کار این شبکه به صورت زیر است:

هنگامی که یک ورودی ارائه می‌شود، لایه RBF فاصله آن را تا هر مرکز از پیش تعریف شده محاسبه می‌کند. مراکز اغلب در طول آموزش آموخته می‌شوند یا از قبل مشخص می‌شوند.

این فواصل با استفاده از تابع پایه شعاعی به مقادیر فعلی سازی تبدیل می‌شوند. رایج‌ترین انتخاب تابع گاووسی است که فعل سازی‌های بالاتر را به ورودی‌های نزدیک به مرکز و فعل سازی‌های کمتر را به ورودی‌های دورتر اختصاص می‌دهد.

خروجی‌های RBF به صورت خطی با استفاده از وزن‌ها ترکیب می‌شوند و نتیجه به لایه بعدی (به عنوان مثال، لایه خروجی یا پنهان) منتقل می‌شود.

توانایی لایه RBF برای تمرکز بر روی مناطق محلی فضای ورودی، آن را برای برنامه‌هایی مانند تشخیص الگو و پیش‌بینی سری‌های زمانی مفید می‌کند، جایی که تغییرات کوچک در ورودی‌ها می‌تواند به طور قابل توجهی بر خروجی‌ها تأثیر بگذارد.

خروجی یک نورون RBF را می‌توان به صورت ریاضی اینگونه بیان کرد:

$$y_i = \phi(\|x - c_i\|)$$

y_i is the output of the i -th RBF neuron.

$\phi(\cdot)$ is the radial basis function.

x is the input vector.

c_i is the center vector of the i -th RBF neuron.

$\|\cdot\|$ denotes a distance metric, typically the Euclidean distance.

برای RBF گاوی (متداول ترین انتخاب):

$$\phi(\|x - c_i\|) = \exp\left(-\frac{\|x - c_i\|^2}{2\sigma_i^2}\right)$$

σ_i گسترش یا عرض تابع گاوی است که میزان شدت کاهش آن با فاصله را کنترل می‌کند.

برای شبکه‌ای با نورون‌های N در لایه RBF، خروجی کلی به صورت زیر است:

$$y = \sum_{i=1}^N w_i \phi(\|x - c_i\|)$$

در واقع خروجی از جمع خروجی‌های نورونها که در وزنهای ضرب شده اند بدست می‌آید.

مراحل یادگیری این شبکه به این صورت است:

- تعیین مرکز (c_i): این مرکز را می‌توان با استفاده از روش‌هایی مانند خوشبندی k-means یا یادگیری در طول آموزش انتخاب کرد.

- تنظیم (σ_i): این را می‌توان در طول آموزش ثابت یا بهینه کرد.
- بهینه سازی وزنها (W_{ij}): این از طریق تکنیک‌های بهینه سازی استاندارد، اغلب با استفاده از گرادیان نزولی یا رگرسیون خطی انجام می‌شود.

در این قسمت برای طراحی شبکه عصبی با لایه RBF از کد آماده شده توسط گروه تدریسیاری که در گیت هاب دوره آموزشی موجود بوده است استفاده شده.

البته کد فراهم شده برای انجام عمل classification توسعه داده شده است و لازم بود تغییراتی در این کد اعمال شود به صورتی که بتوان عمل regression را به وسیله آن انجام داد.

در اینجا به صورت اجمالی به توضیح این کد می‌پردازیم:

شبکه RBF اصلاح شده برای رگرسیون ساختار اصلی مدل طبقه‌بندی اصلی را حفظ می‌کند و در عین حال اجزای کلیدی را برای رسیدگی به اهداف پیوسته تطبیق را بدین صورت تغییر می‌دهد:

انطباق لایه خروجی: لایه خروجی شبکه به جای تابع سیگموئیدی که در طبقه‌بندی استفاده می‌شود، از یک تابع فعال سازی خطی استفاده می‌کند. این تغییر برای پیش‌بینی مقادیر پیوسته ذاتی در وظایف رگرسیون ضروری است.

وزن و بایاس اولیه: وزن‌ها و بایاس‌ها با ابعاد مناسب برای خروجی رگرسیون منفرد ($nY = 1$) مقداردهی اولیه می‌شوند. این تنظیم تضمین می‌کند که شبکه به جای احتمالات کلاس چند بعدی، پیش‌بینی‌های اسکالر تولید می‌کند.

تابع تلفات و حلقه آموزشی: میانگین مربعات خطأ (MSE) به عنوان تنها معیار تلفات استفاده می‌شود و نیاز به اندازه گیری دقیق طبقه‌بندی را از بین می‌برد. حلقه آموزشی به طور مکرر وزن‌ها و bias‌ها را با استفاده از GDR به روز می‌کند و منحصرًا بر روی حداقل کردن MSE برای افزایش عملکرد رگرسیون تمرکز می‌کند.

مکانیسم پیش‌بینی: روش پیش‌بینی مقادیر پیوسته را با استفاده مستقیم از فعال‌سازی خطی لایه خروجی، خروجی می‌دهد و پیش‌بینی‌های عددی دقیق مورد نیاز برای کاربردهای رگرسیون را ارائه می‌دهد.

Center Initialization with K-Means Clustering (`fit_centers` Method):

```
def fit_centers (self, X:np.ndarray):
    self.KMN = cl.KMeans(n_clusters=self.nH, random_state=42)
    self.KMN.fit(X)
    self.C = self.KMN.cluster_centers_
```

از الگوریتم خوشبندی K-Means برای تعیین مراکز توابع پایه شعاعی در لایه پنهان استفاده می‌کند.

Training the Network (`fit` Method)

```
def fit (self, X:np.ndarray, Y:np.ndarray, nH:int, nEpoch:int=100, lr:float=1e-2):
    self.nX = X.shape[1]
    # For regression, output dimension is 1
    self.nY = 1
    self.nH = nH
    self.nEpoch = nEpoch
    self.lr = lr
    self.fit_centers(X)
    # Ensure Y is a 2D array with shape (N, 1)
    if Y.ndim == 1:
        Y = Y.reshape(-1, 1)
    self.fit_wb(X, Y)
```

این قسمت معماری شبکه را با تنظیم ابعاد ورودی (nX) ، ابعاد خروجی (nY) ، تعداد نورون‌های پنهان (nH) ، تعداد دوره‌های آموزشی ($nEpoch$) و نرخ یادگیری (lr) پیکربندی می‌کند.

فرآیند برازش مرکزی را آغاز می‌کند و متغیر هدف Y را برای آموزش آماده می‌کند.

Weight and Bias Training (`fit_wb` Method):

مراحل حلقه آموزش بدین صورت است:

برای هر جفت ورودی-هدف:

خروجی شبکه را با استفاده از وزن‌ها و بایاس‌های فعلی محاسبه می‌کند.

به روز رسانی وزن‌ها و سوگیری‌ها: وزن‌ها و سوگیری‌ها را بر اساس خطای و نرخ یادگیری با استفاده از قانون دلتای تعمیم یافته (GDR) تنظیم می‌کند.

```

def fit_wb (self, X:np.ndarray, Y:np.ndarray):
    D = self.get_distances(X)
    O1 = self.bf(D)
    # Initialize weights and biases for regression (nY=1)
    self.W = np.random.uniform(-1, +1, (self.nH, self.nY))
    self.B = np.random.uniform(-1, +1, (self.nY,))
    self.history = {'loss':[]}
    O2 = self.model(O1)
    E = round(met.mean_squared_error(Y, O2), 4)
    self.history['loss'].append(E)
    print(f'Epoch: {0} -- Loss: {E}')
    for I in range(self.nEpoch):
        for x, y in zip(O1, Y):
            # Forward pass
            o = self.model(x)
            # Compute error
            e = y - o # Shape: (1,)
            # For linear activation, derivative d = 1
            d = 1
            # Update weights and biases
            self.W += self.lr * np.outer(x, e * d) # Shape: (nH, 1)
            self.B += self.lr * (e * d) # Shape: (1,)
            # Compute loss after epoch
        O2 = self.model(O1)
        E = round(met.mean_squared_error(Y, O2), 4)
        self.history['loss'].append(E)
        print(f'Epoch: {I+1} -- Loss: {E}')

```

Distance Computation (get_distances Method)

```

def get_distances (self, X:np.ndarray):
    N = X.shape[0]
    D = np.zeros((N, self.nH))
    for i in range(N):
        for j in range(self.nH):
            D[i, j] = np.linalg.norm(X[i] - self.C[j], ord=2)
    return D

```

یک ماتریس فاصله D را ایجاد می کند که در آن هر ورودی i فاصله بین نمونه ورودی i و مرکز j ام را نشان می دهد.

Euclidean Distance:

$$\|x - c_j\| = \sqrt{\sum_{k=1}^d (x_k - c_{jk})^2}$$

Basis Function Transformation (bf Method):

```
def bf (self, D:np.ndarray, a:float=10):
    return np.exp(-a*np.power(D, 2))
```

ماتریس فاصله D را با استفاده از تابع پایه شعاعی گاووسی به فعال سازی تبدیل می کند. پارامتر a پهنهای (گسترش) تابع گاووسی را کنترل می کند.

Gaussian Radial Basis Function:

$$\phi(\|x - c_j\|) = \exp(-a\|x - c_j\|^2)$$

در نهایت خروجی نهایی و عمل prediction از ترکیب خطی خروجی لایه پنهان (RBF activation) با وزنهای مناسب بدست می آید.

```
def model (self, X:np.ndarray):
    Z = np.dot(X, self.W) + self.B # Shape: (nY,)
    # For regression, use linear activation
    O = Z
    return O
```

Linear Activation:

$$O = \mathbf{W}^T \phi + \mathbf{B}$$

نتایج مدل بالا بر روی مجموعه مورد نظر به قرار زیر خواهد بود:

Train loss(MSE)	Test loss(MSE)
1.3179	1.2937

نمودار loss داده آموزش به صورت زیر است:



همانطور که از نمودار مشخص است آموزش انجام شده است و مقدار خطای مدل نیز قابل قبول است.

روش دوم طراحی شبکه RBF :

به عنوان روش دوم برای طراحی شبکه RBF از یک پروژه github استفاده شده است که امکان ایجاد یک لایه RBF به عنوان لایه پنهایی و استفاده از کتابخانه keras برای توسعه باقی شبکه را می دهد.

در واقع ابتدا یک مدل بوسیله keras.models.Sequential() ایجاد می کنیم و سپس یک لایه RBF به شبکه خود اضافه می کنیم (model.add(rbfLayer)). که این لایه RBF توسط پروژه github مورد نظر ایجاد می شود.

سپس یک لایه خروجی نیز به مدل اضافه می کنیم.

با این روش می توان از متدهای آماده خود کتابخانه keras مثل model.fit استفاده کنیم.

```

kfold=KFold(n_splits=2)
histories=[]

for train_index, test_index in kfold.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

rbfLayer=rbfLayer.RBFLayer(40,initializer=kmeans_initializer.InitCentersKMeans(X_train), betas=1.0, input_shape=(X_train.shape[1],))

model=keras.models.Sequential()
model.add(rbfLayer)
model.add(layers.Dense(1))
model.compile(loss='mean_squared_error', optimizer=RMSprop())

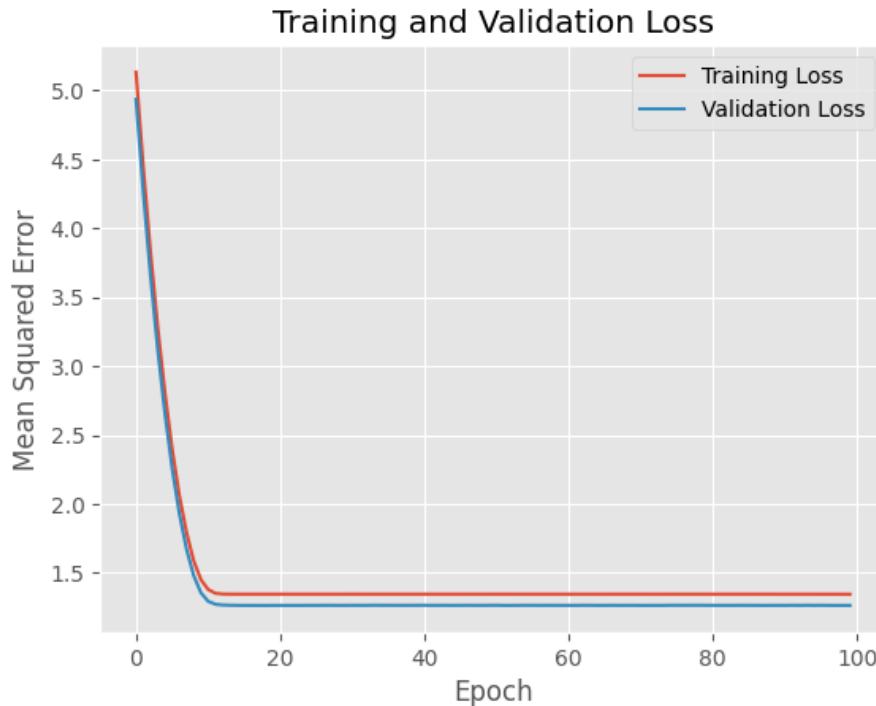
history=model.fit(X_train, y_train, batch_size=50, epochs=100, verbose=1, validation_split=0.1)
histories.append(history)

```

نتایج این مدل نیز اینگونه خواهد بود:

Train loss	Validation loss	Test loss
1.3454	1.2642	1.3258

نمودار loss داده آموزش به صورت زیر است:



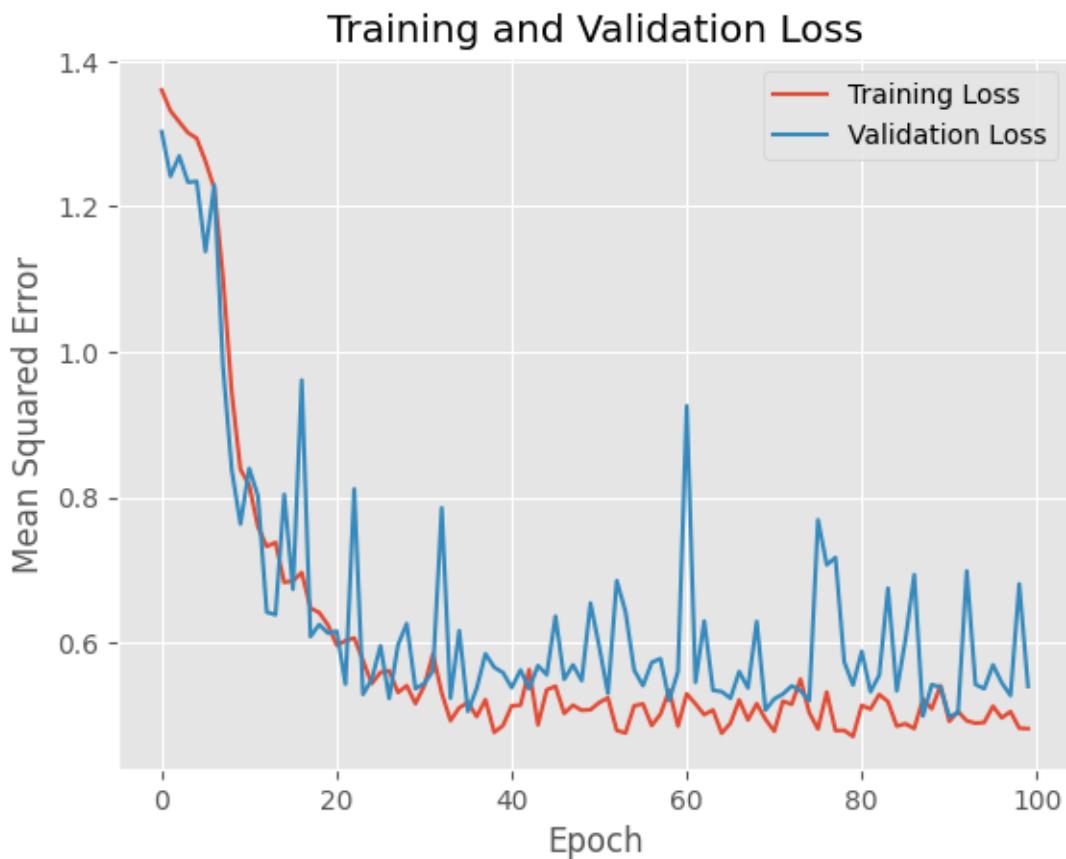
می توان مشاهده کرد که نتیجه این مدل نیز مشابه مدل توسعه داده شده قبلی و در حدود 1.3 است. یعنی نتایج هر دو مدل RBF تقریبا مشابه بدست آمدند.

حال در قسمت دوم یک شبکه عصبی MLP با یک لایه پنهان را برای انجام عمل regression روی این مجموعه داده طراحی می کنیم. برای این کار یک شبکه با یک لایه پنهان که ۲۰ نرون دارد استفاده می کنیم. به عنوان تابع فعالساز در لایه پنهان از tanh و در لایه خروجی چون عمل regression انجام می شود از linear استفاده می کنیم.

از adam به عنوان بهینه ساز و از mse به عنوان loss استفاده می کنیم.

Train loss	Validation loss	Test loss
0.4825	0.5406	0.6830

تصویر زیر نمودار loss برای داده train و test را نشان می دهد.



با توجه به نتایج بدست آمده از شبکه RBF و MLP مشخص است که شبکه پرسپترون عملکرد بهتری در انجام عمل regression برای این داده ها داشته است و مقدار loss تست آن تقریباً نصف شبکه RBF است.

MLP به دلیل توانایی آن در یادگیری الگوهای جهانی و روابط پیچیده و غیر خطی در داده‌ها، اغلب در وظایف رگرسیونی بهتر از شبکه RBF عمل می‌کند. MLP‌ها از لایه‌های کاملاً متصل با توابع فعال‌سازی استفاده می‌کنند که به آن‌ها اجازه می‌دهد تا توابع را در کل فضای ورودی تقریبی کنند، نه اینکه به پاسخ‌های محلی محدود شوند. این باعث می‌شود که آنها برای مشکلاتی که عملکرد هدف در طیف وسیعی از ورودی‌ها به آرامی تغییر می‌کند، انعطاف‌پذیرتر و مناسب‌تر باشند.

در مقابل، شبکه‌های RBF به توابع فعال‌سازی محلی که حول نقاط خاص متتمرکز شده‌اند، متکی هستند. در حالی که این می‌تواند برای گرفتن الگوهای موضعی موثر باشد، اما می‌تواند با فضاهای ورودی با ابعاد بالا یا زمانی که وظیفه رگرسیون نیاز به تعمیم فراتر از تأثیر تعداد محدودی از توابع پایه دارد، موثر واقع شود. علاوه بر این، شبکه‌های RBF ممکن است نیاز به تنظیم دقیق تعداد و محل قرارگیری مراکز داشته باشند، که می‌تواند منجر به عملکرد کمتر از حد مطلوب در کارهای رگرسیون پیچیده یا بزرگ در مقایسه با یادگیری خودکار ویژگی‌های MLP شود.