مبانی سیستم‌های هوشمند

مینی پروژه شماره ۲

نام و نام خانوادگی	محمد خلیلی
شمارهٔ دانشجویی	۹۹۲۵۹۷۳
تاریخ	دی ۱۴۰۳
GitHub	proj۲_kntu
drive	drive
Telegram	@m۱۵khh

فهرست مطالب

فهرست تصاویر

۱ سوال اول

۱.۱ جایگزین برای Relu

در مسئله طبقه‌بندی دوکلاسه، اگر دو لایه انتهایی شبکه شما از فعال‌سازهای ReLU و سیگموید استفاده کنند، رفتار غیرمعمولی رخ می‌دهد که می‌تواند دقت شبکه را تحت تأثیر قرار دهد. در ادامه توضیح می‌دهیم:

۱. **ReLU** در لایه انتهایی: ReLU (Rectified Linear Unit) برای وظایفی مانند طبقه‌بندی مناسب نیست، زیرا مقادیر خروجی آن می‌توانند مقادیر مثبت و صفر باشند، و هیچ محدودیتی برای مقیاس خروجی وجود ندارد. برای طبقه‌بندی، خروجی به احتمال‌هایی بین ۰ و ۱ نیاز دارد تا بتوان آن‌ها را به کلاس‌ها نسبت داد. استفاده از ReLU در لایه انتهایی باعث می‌شود خروجی‌ها فاقد این خصوصیت باشند.

۲. سیگموید در لایه دوم: سیگموید مقدار ورودی خود را به مقادیر بین ۰ و ۱ نگاشت می‌کند، که برای مسئله طبقه‌بندی باینری مناسب است. اما اگر ورودی سیگموید از ReLU دریافت شود، ممکن است مشکل‌ساز شود:

- ReLU می‌تواند مقادیر بسیار بزرگ را تولید کند. وقتی این مقادیر به سیگموید وارد می‌شوند، سیگموید اشباع شده و خروجی به شدت نزدیک به ۱ خواهد بود. این امر باعث از دست دادن اطلاعات در فرآیند پیش‌بینی می‌شود.
- اگر مقدار ورودی ReLU صفر باشد (به دلیل خاصیت ReLU)، خروجی سیگموید به مقدار ۰.۵ نزدیک خواهد شد، که می‌تواند منجر به پیش‌بینی‌های نامطمئن شود.

۳. مشکل کلی: ترکیب ReLU و سیگموید در لایه‌های انتهایی به جای بهینه‌سازی خروجی احتمال‌محور، منجر به عدم تعادل در مقیاس مقادیر و پیش‌بینی‌های غیردقیق می‌شود.

راه‌حل پیشنهادی: به جای ReLU، از Softmax (در مسائل چندکلاسه) یا سیگموید (در مسائل دوکلاسه) به عنوان لایه انتهایی استفاده کنید. این فعال‌سازها خروجی را به احتمال‌هایی که مجموع آن‌ها ۱ است تبدیل می‌کنند، که برای وظایف طبقه‌بندی مناسب‌تر است.

۲.۱ ELU

جایگزینی ارائه‌شده برای ReLU، یعنی ELU (Exponential Linear Unit)، مزایای قابل‌توجهی نسبت به ReLU دارد. در ادامه جزئیات آورده شده است:

۱. محاسبه گرادیان ELU:

گرادیان ELU با توجه به دو حالت تعریف می‌شود:

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

۲. مزایای ELU نسبت به ReLU:

۱. رفع مشکل "خاموش شدن نورون‌ها" **(Dead Neurons)**: در ReLU، اگر مقدار ورودی منفی باشد، خروجی صفر می‌شود و گرادیان آن نیز صفر خواهد بود. این حالت می‌تواند منجر به خاموش شدن دائمی نورون‌ها شود. اما در ELU، حتی برای مقادیر منفی، خروجی و گرادیان صفر نیست (گرادیان برابر با αe^x است). این ویژگی به مدل کمک می‌کند تا یادگیری مؤثرتری داشته باشد.

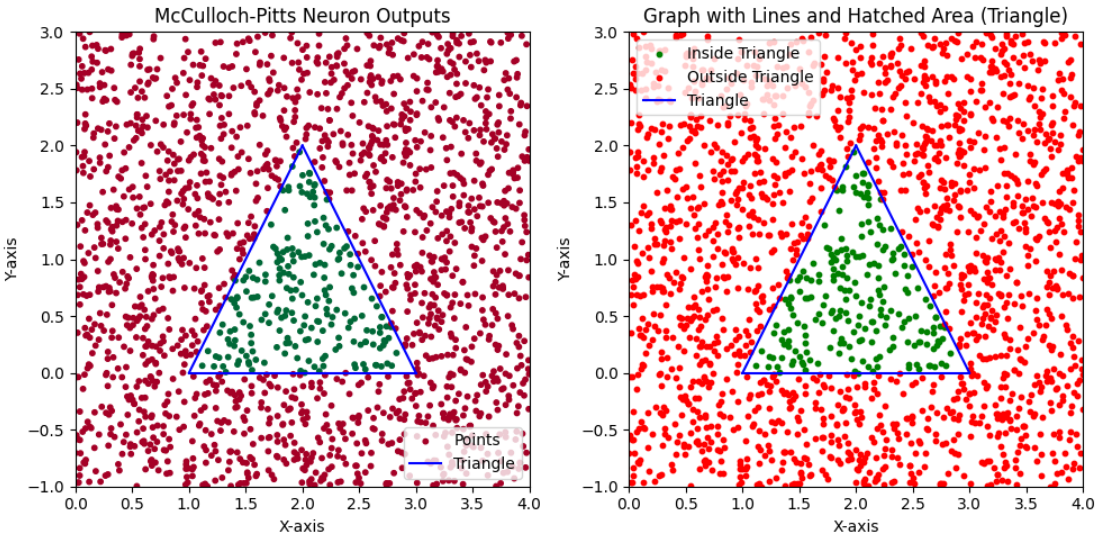
۲. تعادل در مقادیر میانگین خروجی: خروجی ELU در مقادیر منفی، به تدریج به $-\alpha$ نزدیک می‌شود. این امر باعث می‌شود مقادیر خروجی شبکه تعادلی نزدیک به صفر داشته باشند و یادگیری بهینه‌تری صورت گیرد. در مقابل، ReLU خروجی‌های نامتعادلی ایجاد می‌کند که ممکن است سرعت همگرایی را کاهش دهد.

۳. پیوستگی گرادیان: در ELU، گرادیان در تمام نقاط پیوسته است (حتی در $x = 0$). اما در ReLU، گرادیان در $x = 0$ ناپیوسته است، که می‌تواند در بهینه‌سازی مشکل‌ساز شود.

نتیجه‌گیری:

ELU جایگزینی مناسب برای ReLU است، به ویژه در مواردی که مشکل خاموش شدن نورون‌ها یا ناپیوستگی گرادیان وجود دارد. این فعال‌ساز عملکرد بهتری را در شبکه‌های عمیق ارائه می‌دهد و می‌تواند سرعت همگرایی را بهبود بخشد.

۳.۱ تفکیک ناحیه هاشورزدهٔ داخل مثلثی



شکل ۱: نمایش خروجی

به کمک یک Perceptron ساده یا نورون McCulloch-Pitts شبکه‌ای طراحی کنید که بتواند ناحیه هاشورزده داخل مربع شکل که در نمودار شکل (۱) نشان داده شده را از سایر نواحی تفکیک کند. پس از انجام مرحله طراحی شبکه (که می‌تواند به صورت دستی انجام شود، برنامه‌ای که در آن دفترچه که در کلاس برای McCulloch-Pitts آموزش دیده‌اید را به گونه‌ای توسعه دهید که ۴۰۰۰ نقطه رندوم تولید کند و آن‌ها را به عنوان ورودی به شبکه طراحی شده توسط شما بدهد و نقاطی که خروجی 1 تولید می‌کنند را با رنگ سبز و نقاطی که خروجی 0 تولید می‌کنند را با رنگ قرمز نشان دهد. خروجی تولیدشده توسط برنامه شما باید به صورتی که در شکل (ب) نشان داده شده است باشد (به محدوده عددی محورها دقت کنید). اثر اضافه کردن تابع فعال‌ساز مختلف به فرآیند تصمیم‌گیری را بررسی کنید.

دقت شود که همانطور که گفتم activation را قرار داد تا مثل sign 1 خروجی هر نورون یک مقدار یادگیری شود.

نورون چهارم (Neur4):

- خروجی سه نورون قبلی به عنوان ورودی به این نورون داده می‌شوند.
- این نورون دارای یک تابع فعال‌ساز (Activation Function) مانند Sign 1، ELU، ReLU یا Sigmoid است که روی مجموع وزنی ورودی‌ها اعمال می‌شود.

این نورون تصمیم نهایی را می‌گیرد که آیا نقطه داخل مثلث است یا خیر.

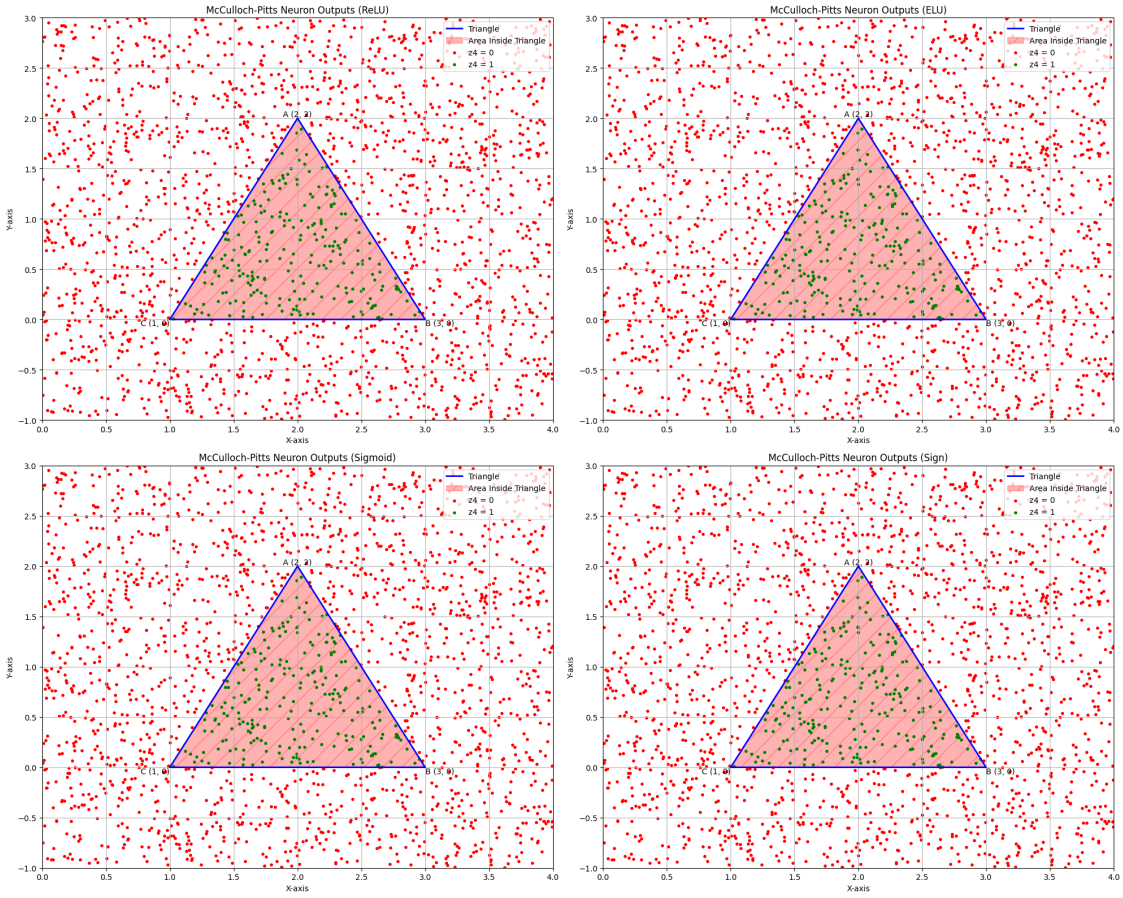
برای تنظیم وزن‌های این نورون ابتدا دقت کنیم که خروجی لایه‌های قبل اگر همه یک باشند داخل مثلث و اگر حتی یکی از خروجی‌ها صفر باشد بیرون مثلث است و خروجی اکتیویشن فانکشن تنها در صورت نزدیک ۵.۰ بودن داخل یک حساب میانبر می‌بینیم پس نیاز به یک تابع انتخابی که اطمینان کم باید پایین مناسب را قرار دهیم:

$$\sigma(x) = \begin{cases} \frac{1}{1-e^{-(W\times x-threshold)}} & \text{if} x = 3 \\ 0.5 \implies \text{threshold} = 2.99 \end{cases}$$

$$\text{relu}(x) = \begin{cases} x - \text{threshold}, & x - \text{threshold} \geq 0 \text{ if } x = 3 \\ 0, & x - \text{threshold} < 0 \end{cases} \implies 0.5 \implies \text{threshold} = 2.499$$

$$\text{ELU}(x) = \begin{cases} \alpha(e^{x-\text{threshold}} - 1), & x - \text{threshold} < 0 \\ x - \text{threshold}, & x - \text{threshold} \geq 0 \end{cases} \implies 0.5 \implies \text{threshold} = 2.499$$

$$\text{sign}(x) = \begin{cases} 1, & x - \text{threshold} \geq 0 \text{ if } x = 3 \\ 0, & x - \text{threshold} < 0 \end{cases} \implies 0.5 \implies \text{threshold} = 2.9999$$



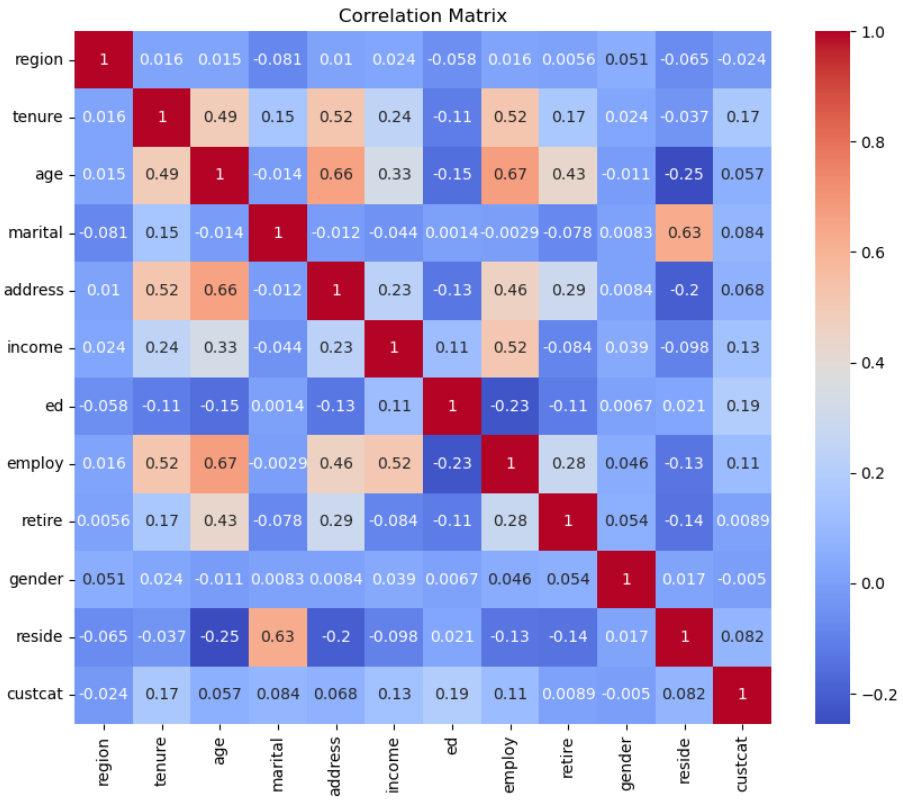
شکل ۲: نمایش خروجی نهایی

۲ سوال دوم

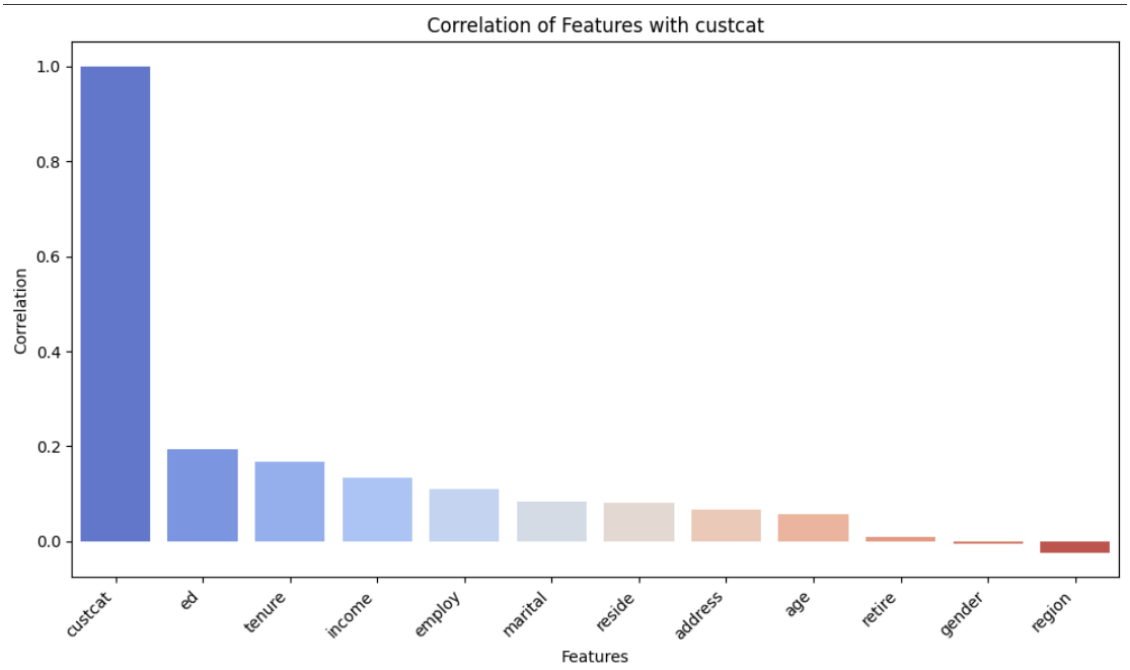
۱.۲ خواندن دیتاست

```python
import pandas as pd
df = pd.read_csv('teleCust1000t.csv')
```

۲.۲ نمایش heat map

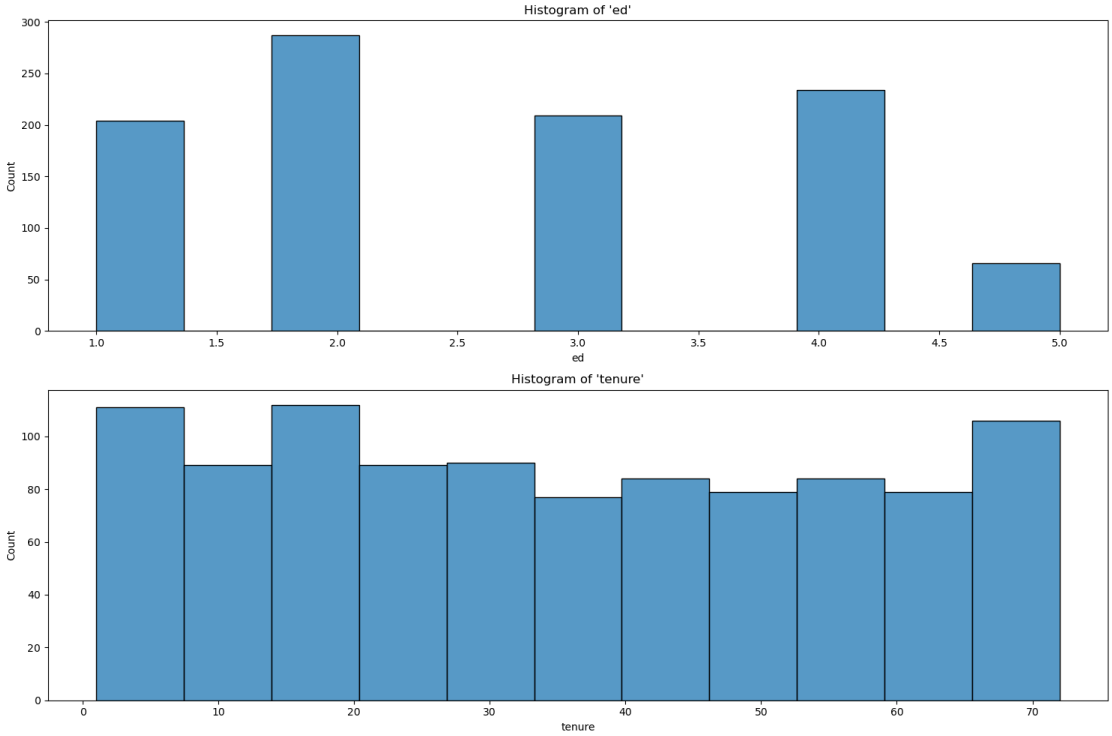


شکل ۳: نمایش heat map



شکل ۴: corrolation sort

همانطور که از شکل ۴ مشخص هست custact بیشترین corrolation با خودش دارد (کاملا منطقی) خب خواسته سوال قطعا این مورد نیست بیشترین corrolation مثبت با ed و tenure رادارد



شکل ۵: نمایش histogram با بیشترین corrolation

۳.۲ نرمالایز کردن داده

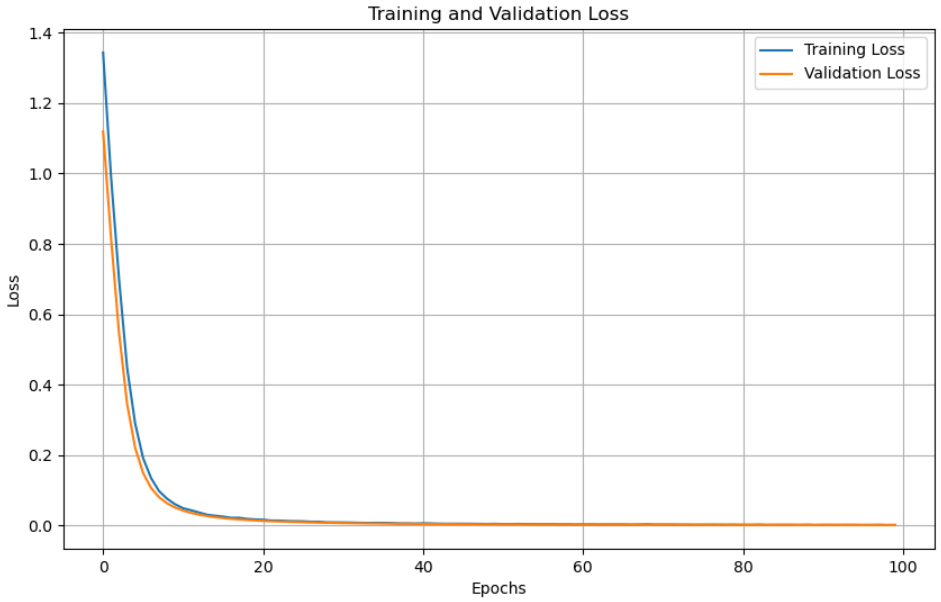
در کد پیاده سازی شد

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data.iloc[:, :-1])
labels = data.iloc[:, -1]
scaled_data = pd.DataFrame(scaled_data, columns=data.columns[:-1])


X_train, X_temp, y_train, y_temp = train_test_split(scaled_data, labels, test_size=0.3,
    random_state=73)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=73)

```

۴.۲ شبکه عصبی چندلایه

برای ارزیابی بهتر لیبل ها را one hot می‌کنیم

```
y_train_onehot = to_categorical(y_train, num_classes=4)
y_val_onehot = to_categorical(y_val, num_classes=4)
y_test_onehot = to_categorical(y_test, num_classes=4)

```



شکل ۶: نمودار train loss و test

۵.۲ نرمالایز کردن داده

در کد پیاده سازی شد

۳ سوال سوم

۱.۳ توضیح توابع استفاده شده

توضیحات کد تولید تصاویر نویزدار

این کد دو بخش اصلی دارد:

۱. تابع generateNoisyImages

۲. تابع getNoisyBinaryImage

در ادامه، مراحل عملکرد هر بخش توضیح داده شده است.

۱. تابع generateNoisyImages

این تابع وظایف زیر را انجام می‌دهد:

- یک لیست از مسیر تصاویر ورودی تعریف می‌کند:
- برای هر تصویر، تابع getNoisyBinaryImage فراخوانی می‌شود.
- مسیر ذخیره تصاویر نویزدار مشخص می‌شود، برای مثال:

"/content/noisy1.jpg"

- پس از پردازش هر تصویر، یک پیام در کنسول چاپ می‌شود تا اعلام کند تصویر نویزدار ذخیره شده است.

۲. تابع getNoisyBinaryImage

این تابع وظیفه افزودن نویز به تصویر و ذخیره آن را بر عهده دارد. مراحل عملکرد آن به شرح زیر است:

- تصویر ورودی با استفاده از کتابخانه Pillow باز می‌شود.
- یک ابزار رسم از طریق ImageDraw ایجاد می‌شود تا بتوان پیکسل‌های تصویر را تغییر داد.
- نویز تصادفی به هر پیکسل تصویر اضافه می‌شود. نویز از بازه:

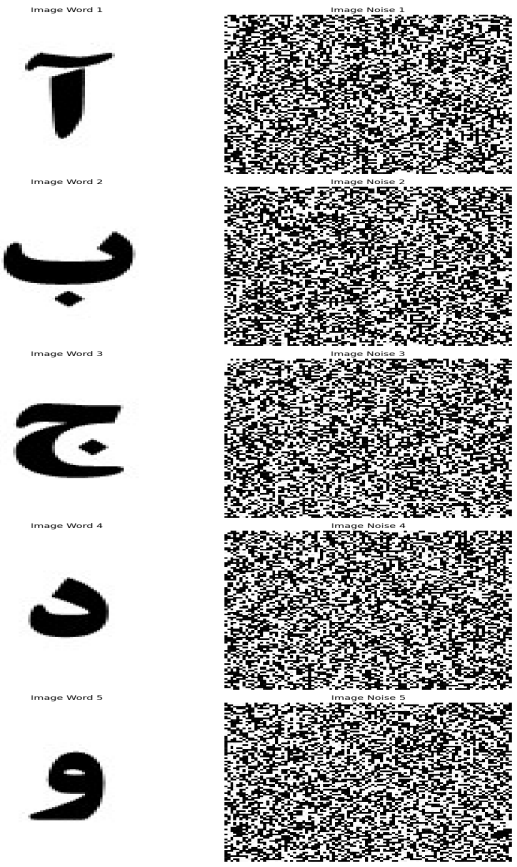
$$[-\text{factor noise}, +\text{factor noise}]$$

انتخاب می‌شود. در اینجا مقدار noise factor برابر با 10000000 تنظیم شده است.

- مقادیر RGB پس از اضافه شدن نویز بررسی می‌شوند تا خارج از بازه مجاز $[0, 255]$ نباشند. مقادیری که از این بازه خارج شوند، اصلاح می‌شوند:
  - اگر مقدار کمتر از 0 باشد، به 0 تنظیم می‌شود.
  - اگر مقدار بیشتر از 255 باشد، به 255 تنظیم می‌شود.
- تصویر نویزدار با فرمت JPEG در مسیر مشخص‌شده ذخیره می‌شود.

کاربردهای کد

- تولید مجموعه داده‌های نویزدار برای ارزیابی مدل‌های یادگیری ماشین.
- شبیه‌سازی شرایط واقعی با نویز برای بررسی الگوریتم‌های پردازش تصویر.
- تحلیل تأثیر نویز بر کیفیت تصویر یا دقت تشخیص.



شکل ۷: نمایش تصاویر کلمات به همراه نویز

۲.۳ طراحی شبکه عصبی

```
۱
۲ class HopfieldNetwork:
۳     def __init__(self, num_neurons):
۴         self.num_neurons = num_neurons
۵         self.weights = np.zeros((num_neurons, num_neurons))
۶
۷     def train(self, patterns):
۸         '''Train the network using Hebbian learning.'''
```

```
        for pattern in patterns:
            pattern = np.array(pattern).flatten()
            self.weights += np.outer(pattern, pattern)
        np.fill_diagonal(self.weights, 0)
        self.weights /= len(patterns)

    def recall(self, pattern, steps=10):
        '''Recall a pattern from the network.'''
        pattern = np.array(pattern).flatten()
        for _ in range(steps):
            for i in range(self.num_neurons):
                raw_value = np.dot(self.weights[i], pattern)
                pattern[i] = 1 if raw_value >= 0 else -1
        return pattern.reshape(int(np.sqrt(self.num_neurons)), -1)

    def energy(self, pattern):
        '''Calculate the energy of a given pattern.'''
        pattern = np.array(pattern).flatten()
        return -0.5 * pattern.T @ self.weights @ pattern


```

طراحی شبکه عصبی هاپفیلد و تحلیل عملکرد

۱. یک شبکه عصبی هاپفیلد با تعداد نورون برابر با تعداد پیکسل‌های تصویر (۱۰۰ نورون) طراحی شده است.

۲. شبکه با استفاده از قانون یادگیری هم‌آموزش دیده شده است.

۳. با افزایش سطح نویز، عملکرد شبکه تحلیل شده و نقاط شکست شناسایی شده‌اند.

نتایج آزمایش

۱. با نویز پایین (کمتر از ۳۰%)، شبکه به‌خوبی الگوی اصلی را بازسازی می‌کند.

۲. با افزایش نویز به سطوح بالاتر (بیش از ۵۰%)، عملکرد شبکه دچار افت شده و الگو به‌درستی بازسازی نمی‌شود.

در این تابع به‌صورت تصادفی درصد مشخصی از پیکسل‌ها به مقدار صفر تغییر می‌کنند.

۱. آزمایش‌ها نشان داده‌اند که با Point Missing کمتر از ۲۰%، شبکه به‌خوبی بازسازی انجام می‌دهد.

۲. با افزایش این مقدار به بیش از ۴۰%، بازسازی دچار اختلال می‌شود.

برای بهتر شدن می‌توان تعداد نورون‌ها را بیشتر کرد.

```
def add_point_missing(pattern, missing_ratio):
    '''Add point missing noise to the pattern.'''
    missing_pattern = pattern.copy()
    num_missing = int(missing_ratio * pattern.size)
```

```
    indices = np.random.choice(pattern.size, num_missing, replace=False)
    missing_pattern.flat[indices] = 0
    return missing_pattern
```

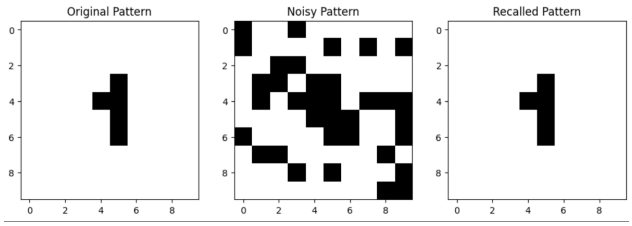
نتیجه‌گیری کلی

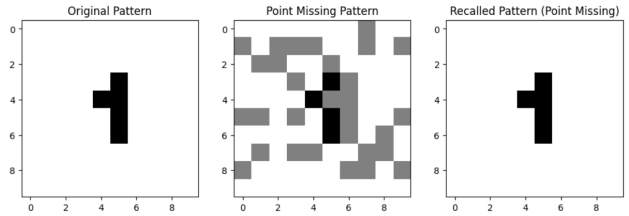
۱. شبکه هاپفیلد در بازسازی تصاویر نویزی و با نقاط از دست‌رفته عملکرد خوبی دارد.

۲. محدودیت‌هایی مانند سطوح بالای نویز و Point Missing شناسایی شده‌اند.

۳. راهکارهایی برای بهبود عملکرد پیشنهاد شده‌اند.

۳.۳ تولید Point Missing





۴ سوال چهارم

۱.۴ پیاده سازی layer RBF

برای پیاده سازی لایه layer RBF از کد زیر استفاده کردم

```
class RBFLayer(tf.keras.layers.Layer):
    def __init__(self, units, gamma=1.0, **kwargs):
        super(RBFLayer, self).__init__(**kwargs)
        self.units = units
        self.gamma = K.constant(gamma)

    def build(self, input_shape):
        self.centers = self.add_weight(name='centers',
```

```
                                        shape=(self.units, input_shape[-1]),
                                        initializer='uniform',
                                        trainable=True)
            self.built = True

        def call(self, inputs):
            diff = K.expand_dims(inputs, axis=1) - self.centers
            l2 = K.sum(K.square(diff), axis=-1)
            return K.exp(-self.gamma * l2)


```

۲.۴

در کد پیاده سازی شد

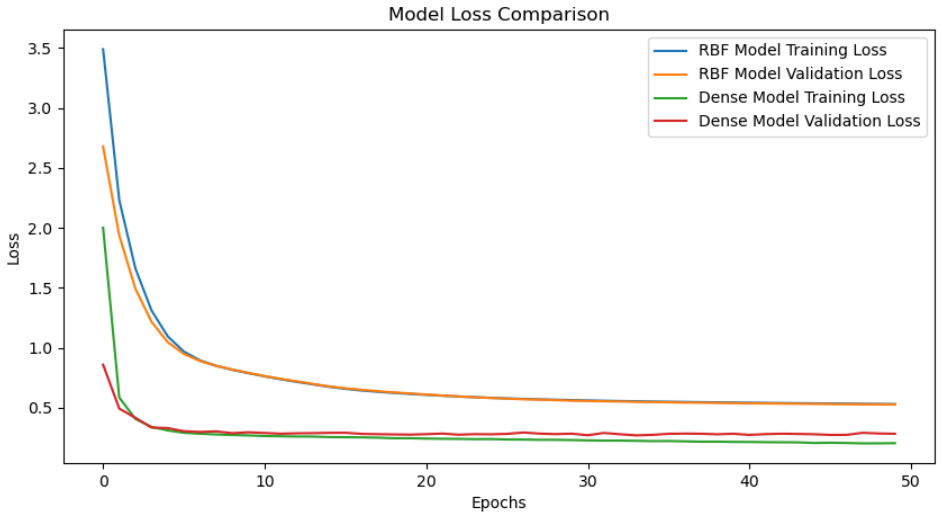
۴.۴ شبکه عصبی ساده با dense layer

```

# Build the Dense model
dense_model = Sequential([
    Input(shape=(X_train.shape[1],)),
    Dense(128, activation='relu'),  # First hidden layer
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(16, activation='sigmoid'),
    Dense(1)  # Output layer
])

# Compile the Dense model
dense_model.compile(optimizer='adam', loss='mse', metrics=['mse'])

# Train the Dense model
dense_history = dense_model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1,
    validation_data=(X_test, y_test))


```

۵.۴

در کد پیاده سازی شد

۶.۴ میزان loss دو لایه



شکل ۸: مقایسه loss برای مدل dense layer و RBF layer

همانطور که از شکل ۸ مشخص است، دقت لایه `dense` بهتر از لایه `rbf` است. همچنین، دقت `train` از عملکرد دقت `test` در لایه `dense` بهتر است. اما در لایه `rbf` ظاهراً مدل بهتر generalize شده و دقت `test` و `train` با هم برابر شده‌اند.

۷.۴ عملکرد دو مدل

لایه RBF به عنوان یک نوع لایه غیرخطی می‌تواند عملکرد خوبی در مدل‌های رگرسیونی برای شبیه‌سازی تابع‌های پیچیده داشته باشد. در این مدل از یک لایه RBF به عنوان لایه پنهان استفاده خواهیم کرد.

مدل Dense بهتر عمل کرده است، ممکن است به دلایل زیر باشد:

۱. ساده بودن داده‌ها: اگر داده‌ها شامل روابط پیچیده و غیرخطی نباشند، لایه‌های کاملاً متصل (Dense) که برای یادگیری روابط خطی طراحی شده‌اند می‌توانند عملکرد بهتری داشته باشند. مدل‌های با لایه‌های RBF معمولاً برای داده‌های پیچیده‌تر و غیرخطی عملکرد بهتری دارند، اما اگر داده‌ها ساده و رابطه‌شان خطی‌تر باشد، مدل‌های Dense می‌توانند سریع‌تر و مؤثرتر عمل کنند.

۲. تنظیمات مدل: ممکن است تنظیمات مدل RBF به درستی بهینه نشده باشد. مثلاً تعداد نورون‌ها یا پارامترهای دیگری که روی عملکرد مدل تأثیر می‌گذارند ممکن است بهینه نباشند.

۳. مقیاس‌دهی داده‌ها: برخی از مدل‌ها مانند RBF به مقیاس‌دهی داده‌ها حساس‌تر هستند. اگر داده‌ها به درستی مقیاس‌دهی نشده باشند، ممکن است عملکرد مدل RBF ضعیف‌تر از Dense شود.

۴. پیچیدگی مدل: مدل‌های Dense معمولاً کمتر حساس به مشکلات مقیاس و معمولاً سریع‌تر و دقیق‌تر در مدل‌سازی روابط ساده‌تر عمل می‌کنند.