دانشکده مهندسی برق

تمرین سوم درس هوش مصنوعی

استاد درس:

جناب آقای دکتر علیاری

پریا ساعی ۴۰۱۱۹۱۶۳

ارشیا کلانتریان ۴۰۱۲۱۹۹۳

مریم سلطانی ۴۰۱۱۹۴۳۳

نیمسال دوم ۱۴۰۳_۱۴۰۴

لینک گوگل کولب:

https://colab.research.google.com/drive/13r35NxWulQkCMSUmkENPUO4XzoK7ze9v?usp=sharing

لینک گیت‌هاب:

مریم سلطانی: https://github.com/MaryamSoltani28/AI_2025

ارشیا کلانتریان: https://github.com/ARKAL-J04/MachineLearning2025

پریا ساعی: https://github.com/Paria-s/AI_4032

سوال۱)

هدف برنامه:

شبیه‌سازی یک نورون McCulloch-Pitts که بتواند نقاط داخل یک مثلث خاص را از نقاط خارج از آن تشخیص دهد. این مثلث با سه نقطه مشخص شده است:
$A(1,0),B(2,3),C(3,0)$

توضیح خط به خط کد:

```
import numpy as np
import matplotlib.pyplot as plt
```

ایمپورت دو کتابخانه مهم:

numpy: برای تولید اعداد تصادفی و محاسبات برداری.

matplotlib.pyplot: برای رسم نمودار داده‌ها و نمایش نقاط داخل/خارج مثلث.

```
lass 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
```

تعریف یک کلاس برای نورونMcCulloch-Pitts

هر نورون با دو پارامتر مشخص می‌شود:

weights: وزن‌های ورودی

threshold: آستانه تحریک نورون

تابع مدل نورون:

اگر مجموع وزن‌دار ورودی‌ها از آستانه بیشتر یا مساوی بود، خروجی ۱ است.

در غیر این‌صورت، خروجی ۰ خواهد بود.

```
# تابع تشخیص داخل مثلث
def is_inside_triangle(x, y):
    # ۳- :معادله ضلع چپx + y <= -3  <=>  3x - y >= 3
    n1 = McCulloch_Pitts_neuron(weights=[3, -1], threshold=3)

    # ۳ :معادله ضلع راستx + y <= 9
    n2 = McCulloch_Pitts_neuron(weights=[-3, -1], threshold=-9)

    # معادله پایین: y >= 0
    n3 = McCulloch_Pitts_neuron(weights=[0, 1], threshold=0)

    z1 = n1.model([x, y])
    z2 = n2.model([x, y])
    z3 = n3.model([x, y])
```

تابعی که بررسی می‌کند آیا نقطه (x,y) داخل مثلث مورد نظر قرار دارد یا نه.

نورون اول، معادله خط سمت چپ مثلث را مدل می‌کند.

خط مربوط به ضلع:AB

$$-3x+y\leq -3\Rightarrow 3x-y\geq 3$$

نورون دوم، خط سمت راست مثلث را مدل می‌کند.

خط مربوط به ضلع:BC

$$3x+y\leq 9\Rightarrow -3x-y\geq -9$$

نورون سوم شرط پایین مثلث را بررسی می‌کند. یعنی اینکه $y\geq 0$ باشد

خروجی سه نورون بالا برای نقطه داده‌شده (x,y) محاسبه می‌شود. این‌ها مشخص می‌کنند که آیا نقطه در ناحیه مورد نظر قرار دارد یا نه.

```
# خروجی نهایی
    output_neuron = McCulloch_Pitts_neuron([1, 1, 1], threshold=3)
```

```
    return output_neuron.model([z1, z2, z3])
```

یک نورون نهایی ساخته می‌شود که فقط وقتی خروجی‌اش 1 باشد که هر سه شرط بالا برقرار باشند. یعنی نقطه دقیقاً داخل مثلث باشد.

```
N = 1000
x_vals = np.random.uniform(0, 4, N)
y_vals = np.random.uniform(0, 4, N)
```

تولید ۱۰۰۰ نقطه تصادفی در بازه‌ی [0, 4] برای محور x و y این نقاط برای تست مدل ما هستند.

```
inside_x, inside_y = [], []
outside_x, outside_y = [], []
```

تعریف چهار لیست برای ذخیره‌سازی نقاطی که:

داخل مثلث هستند(inside)

بیرون از مثلث هستند(outside)

```
for i in range(N):
    x = x_vals[i]
    y = y_vals[i]
    if is_inside_triangle(x, y):
        inside_x.append(x)
        inside_y.append(y)
    else:
        outside_x.append(x)
        outside_y.append(y)
```

بررسی تک‌تک نقاط تولید شده:

اگر نقطه داخل مثلث بود، در لیست سبز ذخیره می‌شود.

اگر بیرون بود، در لیست قرمز ذخیره می‌شود.

```
# رسم
plt.figure(figsize=(8, 6))
plt.scatter(inside_x, inside_y, c='green', label='داخل مثلث (z=1)', s=10)
plt.scatter(outside_x, outside_y, c='red', label='خارج مثلث (z=0)', s=10)
```

رسم نقاط داخل مثلث با رنگ سبز، و خارج مثلث با رنگ قرمز.

```
# رسم مثلث
triangle_x = [1, 2, 3, 1]
triangle_y = [0, 3, 0, 0]
plt.fill(triangle_x[:-1], triangle_y[:-1], color='pink', alpha=0.3, label='ناحیه مثلث')

plt.plot(triangle_x, triangle_y, 'b-', linewidth=2, label='مرز مثلث')
```

رسم مرز مثلث با استفاده از نقاط رأس‌ها. این کار برای نمایش مرز تصمیم‌گیری مدل است.

```
plt.title("نورون McCulloch-Pitts برای ناحیه مثلثی")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.xlim(0, 4)
plt.ylim(0, 4)
plt.legend()
plt.show()
```

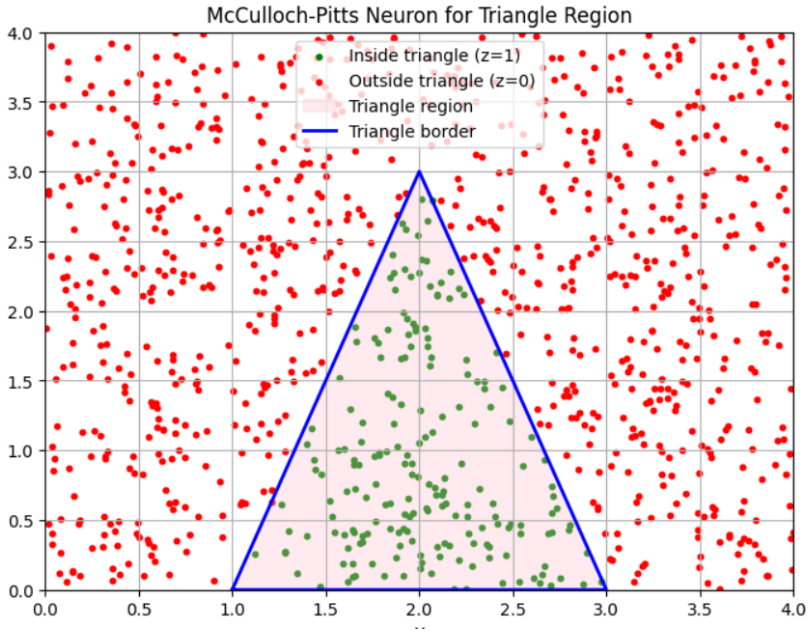
عنوان و برچسب‌های نمودار تنظیم می‌شود.

شبکه (grid) فعال می‌شود.

بازه محورهای x و y به [0, 4]محدود می‌شود.

نمایش نهایی نمودار انجام می‌گیرد.

تصویر خروجی:



مدل با موفقیت توانست نقاط داخل یک ناحیه مثلثی را با استفاده از سه نورون شرطی و یک نورون ترکیب‌کننده output neuron تشخیص دهد. این مثال نشان می‌دهد که چطور می‌توان از شبکه‌های ساده نورون‌های McCulloch-Pitts برای تعریف نواحی هندسی پیچیده استفاده کرد.

سوال۲)

(۲,۱,۱)

این مقاله از داده‌های آب و هوای جمع‌آوری‌شده از چهار منطقه مختلف در جزیره موریس برای پیش‌بینی شرایط آب و هوا در یک منطقه خاص استفاده می‌کند .

شامل پارامترهای زیر است:

- دما(°C)
- سرعت باد(m/s)
- جهت باد(°)
- فشار هوا(Pa)
- رطوبت(%)
- میزان ابری بودن آسمان(%)

داده‌ها در بازه زمانی ۱ تا ۳۱ می سال ۲۰۲۱ و با نرخ نمونه برداری ۴ نمونه در ساعت جمع آوری شده و برای هر منطقه ۲۹۷۶ نمونه وجود دارد.

منطقه‌ها : Curepipe، Vacoas، Quatres Bornes، Moka

فرآیند جمع‌آوری :داده‌ها هر ۱۵ دقیقه به‌روزرسانی شده و از طریق دستگاه‌های لبه‌ای (موبایل و دسکتاپ) دریافت و ذخیره شده‌اند .

(۲,۱,۲)

شهرهای 'MONTELIMAR', 'PERPIGNAN', 'TOURS' از شهرهای فرانسه هستند و باید آنها را جدا کنیم.

ابتدا فایل دیتا را آپلود می‌کنیم و آن را به دیتافرم تبدیل می‌کنیم:

```
!pip install --upgrade --no-cache-dir gdown
!gdown 1dXB660Jt-QwFMeHAmPAx3Im24vKT4maP
# Creating the dataframe
df = pd.read_csv('/content/weather_prediction_dataset.csv')
```

لیست شهرهای موجود در دیتاست را نمایش می‌دهیم:

```
city_columns = df.columns
city_names = sorted(set(col.split('_')[0] for col in city_columns if '_' in col))
```

```
# چاپ لیست شهرها
print(city_names)
```

شهرهای فرانسه را تعریف تعریف می‌کنیم. ردیف مربوط به نام شهرها را بررسی می‌کنیم و هر ستونی که عنوان آن جزو شهرهای فرانسه هست را جدا کرده و در یک دیتافرم جدید ذخیره می‌کنیم:

```
# تعریف شهرهای فرانسوی بر اساس تحلیل قبلی
french_cities = ['MONTELIMAR', 'PERPIGNAN', 'TOURS']

# فیلتر کردن فقط ستون‌هایی که مربوط به این شهرها هستند
filtered_columns = [col for col in df.columns if any(col.startswith(city + '_') for
city in french_cities)]

# ایجاد دیتافریم جدید فقط با شهرهای فرانسوی
df_french = df[filtered_columns]

# مشاهده شکل و نمونه‌ای از داده‌ها
print(df_french.shape)
print(df_french.head())
```

در ادامه ستون‌های حاوی اطلاعات زمانی را به دیتافرم ایجاد شده اضافه می‌کنیم:

```
# شناسایی ستون‌های زمانی
time_columns = [col for col in df.columns if col in ['DATE', 'MONTH']]

# ترکیب داده‌های شهرهای فرانسوی با اطلاعات زمانی
final_french_df = df[time_columns + filtered_columns]

# نمایش شکل نهایی داده‌ها
print(final_french_df.shape)
print(final_french_df.head())
```

(۲,۱,۳

برای بررسی بازه زمانی که دیتاها در آن جمع آوری شده‌اند، مینیمم و ماکسیمم ستون تاریخ را پیدا می‌کنیم:

```
# بررسی بازه زمانی با ستون DATE
start_date = df['DATE'].min()
end_date = df['DATE'].max()
print("Start Date:", start_date)
print("End Date:", end_date)
```

با توجه به مینیمم و ماکسیمم ستون تاریخ، مشاهده می‌شود داده‌ها از ۱,۱,۲۰۰۰ تا ۱,۱,۲۰۱۰ جمع‌آوری شده‌اند.

```
Start Date: 20000101
End Date: 20100101
```

همچنین تعداد نمونه‌ها برابر 3654 است.

(۲,۱,۴

ابتدا داده‌های زمانی را جدا کرده و باقی داده‌ها را نرمالیزه می‌کنیم و سپس ستون های مربوط به داده‌های زمانی را دوباره اضافه می‌کنیم:

```
# جدا کردن ستون‌های عددی برای نرمال‌سازی
data_columns = [col for col in final_french_df.columns if col not in ['DATE',
'MONTH']]
numeric_data = final_french_df[data_columns]

# نرمال‌سازی min-max
normalized_data = (numeric_data - numeric_data.min()) / (numeric_data.max() -
numeric_data.min())

# ترکیب مجدد با ستون‌های زمانی
preprocessed_df = final_french_df[['DATE', 'MONTH']].copy()
preprocessed_df = pd.concat([preprocessed_df, normalized_data], axis=1)
```

در این قسمت با در نظر گرفتن داده‌های پنج روز و دادن آن به مدل، پیش بینی روز بعدی انجام می‌شود.

ردیف هدف را انتخاب کرده و ردیف هایی که حاوی NAN هستند را حذف می‌کنیم.

دو لیست برای ویژگی‌ها و هدف ایجاد می‌کنیم. در ادامه یک حلقه می‌نویسیم این حلقه از ابتدای داده‌ها شروع می‌کند و به اندازه‌ای می‌چرخد که همیشه ۵ روز کامل برای ورودی و یک روز بعدی برای هدف در دسترس باشد.

window_data: شامل داده‌های ۵ روز اخیر است (از ردیف i تا i+5)

مقدار دمای بیشینه مربوط به روز ششم را به عنوان مقدار هدف (y) ذخیره می‌کنیم.

داده‌های date , month را حذف می‌کنیم زیرا در یادگیری مدل ما تاثیری ندارد.

ویژگی‌ها و هدف را در لیست‌هایی که ایجاد کردیم ذخیره می‌کنیم و سپس آنها را به آرایه نامپای تبدیل می‌کنیم تا بتوانیم در ادامه آنها را به مدل بدهیم.

```
# تنظیمات پنجره
```

```
window_size = 5
target_column = 'TOURS_temp_max'

# حذف ردیف‌های ناقص
preprocessed_df_clean = preprocessed_df.dropna().reset_index(drop=True)

# تعریف لیست‌های ویژگی‌ها و هدف
X = []
y = []

# پیمایش داده‌ها برای ایجاد sliding windows
for i in range(len(preprocessed_df_clean) - window_size):
    window_data = preprocessed_df_clean.iloc[i:i + window_size]
    next_day_target = preprocessed_df_clean.iloc[i + window_size][target_column]

    # ویژگی‌ها: داده‌های ۵ روز اخیر، بدون ستون‌های زمانی
    features = window_data.drop(columns=['DATE', 'MONTH']).values.flatten()

    X.append(features)
    y.append(next_day_target)

# تبدیل به آرایه‌های NumPy
X = np.array(X)
y = np.array(y)

# نمایش ابعاد نهایی برای ورودی مدل
print("X shape:", X.shape)
print("y shape:", y.shape)
```

ستون date را به یک رشته تبدیل می‌کنیم تا بتوانیم سال ۲۰۰۹ را جدا کنیم. همانند قسمت قبل لیست‌هایی برای ویژگی، هدف درست کرده همچنین یک لیست برای تاریخ درست می‌کنیم.

حلقه روی داده‌ها اجرا می‌شود تا هر بار یک پنجره ۵ روزه و روز ششم را بررسی کند. داده‌های ویندو را از داده‌های پنج روزه و داده هدف را از داده روز ششم استخراج می‌کنیم.

همانند بخش قبل داده‌های زمانی را حذف می‌کنیم و دیتافرم را به نامپای تبدیل می‌کنیم.

مقدار دمای بیشینه روز ششم به عنوان هدف مدل تعیین شده است.

تاریخ روز ششم ذخیره می‌شود تا در ادامه داده‌های مربوط به سال ۲۰۰۹ را فیلتر کنیم.

داده‌ها را به نامپای تبدیل می‌کنیم.

در ادامه dates بررسی می‌شود و تاریخ‌هایی که با ۲۰۰۹ شروع می‌شوند را به یک آرایه بولی تبدیل می‌کنیم تا مشخص کند هر تاریخ به ۲۰۰۹ مربوط است یا خیر و بتوانیم در ادامه از داده‌های مربوط به سال ۲۰۰۹ به عنوان داده تست استفاده کنیم.

```
# تبدیل ستون DATE به رشته برای فیلتر سال‌ها
preprocessed_df_clean['DATE'] = preprocessed_df_clean['DATE'].astype(str)

# تنظیمات
window_size = 5
target_column = 'TOURS_temp_max'

X = []
y = []
dates = []

# ایجاد sliding window با ثبت تاریخ هدف
for i in range(len(preprocessed_df_clean) - window_size):
    window = preprocessed_df_clean.iloc[i:i + window_size]
    target_row = preprocessed_df_clean.iloc[i + window_size]

    # ویژگی‌ها: ۵ روز × ۲۴ ویژگی (شکل: ۵×۲۴)
    X.append(window.drop(columns=['DATE', 'MONTH']).values)

    # هدف: دمای TOURS در روز ششم
    y.append(target_row[target_column])

    # ثبت تاریخ برای تشخیص train/test
    dates.append(target_row['DATE'])

# تبدیل به آرایه‌های NumPy
X = np.array(X)           # (نمونه‌ها، ۵، ۲۴)
y = np.array(y)           # (نمونه‌ها،)
dates = np.array(dates)   # (نمونه‌ها،)

# جدا کردن داده‌های مربوط به سال ۲۰۰۹ برای تست
test_mask = np.char.startswith(dates, '2009')
X_train = X[~test_mask]
y_train = y[~test_mask]
X_test = X[test_mask]
y_test = y[test_mask]

# نمایش ابعاد نهایی
print("X_train:", X_train.shape)
print("y_train:", y_train.shape)
print("X_test:", X_test.shape)
print("y_test:", y_test.shape)
```

(۲,۲,۱

(Collaborative Machine Learning) به حالتی گفته می‌شود که در آن داده‌های چند ناحیه یا منبع مختلف با هم ترکیب می‌شوند تا پیش‌بینی دقیق‌تری برای یک ناحیه هدف خاص انجام شود. به جای اینکه صرفاً از داده‌های یک ناحیه برای پیش‌بینی وضعیت همان ناحیه استفاده شود، در این روش داده‌های نواحی مجاور یا مرتبط نیز در مدل تاثیر دارند.

در این مقاله، با استفاده از یادگیری ماشین مشارکتی، سیستمی طراحی شده که بتواند پیش‌بینی آب‌وهوا را برای یک ناحیه خاص مثلاً شهر (Moka)با بهره‌گیری از داده‌های آب‌وهوایی نواحی دیگر مثلCurepipe ، Vacoasو Quatres Bornes انجام دهد.

مراحل استفاده از آن در مقاله:

۱- جمع‌آوری داده‌ها از چهار ناحیه مختلف در موریس از طریق.OpenWeather API

۲- تعریف معادلات مشارکتی برای الگوریتم‌های مختلف مثلMLR ، MPR، KNN، MLP، CNN، که در آن‌ها داده‌های چند ناحیه به عنوان ورودی مدل در نظر گرفته می‌شوند.

۳- افزایش دقت پیش‌بینی: نتایج تجربی نشان دادند که الگوریتم‌های مشارکتی، در مقایسه با الگوریتم‌های غیرمشارکتی، میانگین خطای مطلق پایین‌تری داشتند.

(۲,۳

ابتدا داده‌های ورودی آموزش (ویژگی‌ها) را به یک شیء tensor در PyTorch تبدیل می‌کنیم.

X_trainیک آرایه‌ی NumPy است.

reshape(X_train.shape[0], -1)باعث می‌شود هر نمونه به شکل یک بردار یک‌بعدی تبدیل شود (برای سازگاری با لایه ورودی شبکه عصبی).

dtype=torch.float32نوع داده را به عدد اعشاری ۳۲بیتی تنظیم می‌کند که استاندارد PyTorch برای داده‌های ورودی است.

در ادامه خروجی‌های آموزش (برچسب‌ها یا مقدار پیش‌بینی‌شونده) را به یک tensor تبدیل می‌کنیم و با استفاده از .view(-۱, ۱) شکل آن را به (تعداد نمونه‌ها، ۱) تغییر می‌دهیم تا با خروجی مدل سازگار باشد.

این کارها را برای داده‌های تست نیز انجام می‌دهیم.

```
X_train_tensor = torch.tensor(X_train.reshape(X_train.shape[0], -1),
dtype=torch.float32)
```

```
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
X_test_tensor = torch.tensor(X_test.reshape(X_test.shape[0], -1),
dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# تعریف مدل شبکه عصبی طبق شکل مقاله
class MLP(nn.Module):
    def __init__(self, input_size):
        super(MLP, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )
    def forward(self, x):
        return self.net(x)
```

یک تابع به نام train_model_final تعریف می‌کنیم که شبکه را با نرخ یادگیری مشخص (learning_rate) و تعداد epoch دلخواه (پیش‌فرض: ۲۰۰) آموزش دهد.

یک نمونه از شبکه عصبی MLP ساخته می‌شود. input_size=120 یعنی هر نمونه ورودی دارای ۱۲۰ ویژگی است.

تابع خطای مدل MSELoss است (برای مسائل رگرسیون). از بهینه‌ساز SGD (نزول گرادیان تصادفی) با نرخ یادگیری مشخص استفاده می‌کنیم.

با استفاده از DataLoader داده‌ها به شکل دسته‌ای (batch) بارگذاری می‌شوند. در آموزش، داده‌ها به صورت تصادفی (shuffle=True) مرتب می‌شوند ولی در آزمون اینطور نیست.

دو لیست خالی برای ذخیره‌سازی خطای آموزش و آزمون در طول زمان (هر epoch) می‌سازیم.

در ادامه مدل را در حالت train قرار می‌دهیم تا وزن‌ها به‌روزرسانی شوند. متغیر train_loss برای جمع کل خطای هر epochمقداردهی اولیه می‌شود.

در هر batch گرادیان‌ها صفر می‌شوند. مدل خروجی (y_pred) را پیش‌بینی می‌کند.

خطا با loss محاسبه می‌شود. با backward() گرادیان‌ها محاسبه می‌شوند.

با step() وزن‌ها به‌روزرسانی می‌شوند.

مقدار خطا به train_loss اضافه می‌شود.

میانگین خطای آموزش آن epoch در لیست train_losses ذخیره می‌شود.

مدل در مرحله تست قرار می‌گیرد.

به علت اینکه فقط می‌خواهیم تست انجام دهیم نیازی به ذخیره مشتق ها نیست و با torch.no_grad() گرادیان محاسبه نمی‌شود (برای صرفه‌جویی در حافظه و سرعت).

در هر batch، خروجی پیش‌بینی و خطا محاسبه می‌شود و به test_loss اضافه می‌شود.

میانگین خطای آزمون برای آن epoch در لیست test_losses ذخیره می‌شود.

در آخر، لیست کامل خطاهای آموزش و آزمون در طول epochs به صورت دو لیست بازگردانده می‌شود.

یک دیکشنری به نام final_results برای ذخیره نتایج ساخته می‌شود.

سه نرخ یادگیری مختلف برای تست شدن انتخاب شده‌اند: ۱، ۰,۰۰۱ و ۰,۰۰۰۰۰۰۰۱.

برای هر نرخ یادگیری: مدل آموزش داده می‌شود. لیست خطاهای آموزش و آزمون از تابع برمی‌گردد.

نتایج در دیکشنری final_results ذخیره می‌شوند.

```
# تابع آموزش برای یک نرخ یادگیری خاص
def train_model_final(learning_rate, epochs=200):
    model = MLP(input_size=120)
    criterion = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=64)

    train_losses = []
    test_losses = []

    for epoch in tqdm(range(epochs), desc=f"Training (lr={learning_rate})"):
        model.train()
        train_loss = 0
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        train_losses.append(train_loss / len(train_loader))

        model.eval()
```

```
        test_loss = 0
        with torch.no_grad():
            for X_batch, y_batch in test_loader:
                y_pred = model(X_batch)
                loss = criterion(y_pred, y_batch)
                test_loss += loss.item()
        test_losses.append(test_loss / len(test_loader))

    return train_losses, test_losses

# اجرای مدل برای هر سه نرخ یادگیری
final_results = {}
final_lrs = [1, 1e-3, 1e-8]
for lr in final_lrs:
    train_loss, test_loss = train_model_final(learning_rate=lr, epochs=200)
    final_results[lr] = (train_loss, test_loss)
```

در انتها نمودارهای train , test را برای نرخ یادگیری‌های مد نظر رسم می‌کنیم:

```
colors = {1: 'orange', 1e-3: 'green', 1e-8: 'brown'}

for lr in final_lrs:
    plt.figure(figsize=(6, 3))
    plt.plot(final_results[lr][0], color=colors[lr], label=f"Train - lr={lr}")
    plt.plot(final_results[lr][1], color=colors[lr], linestyle='--', label=f"Test  -
lr={lr}")
    plt.title(f"Loss Curve for Learning Rate = {lr}")
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

امتیازی)

داده‌های X_train و X_test به شکل دو بعدی ([تعداد نمونه, تعداد ویژگی‌ها]) بازسازی می‌کنیم.

y_train و y_test نیز به شکل [تعداد نمونه, ۱] درمی‌آیند تا با خروجی مدل هم‌خوانی داشته باشند.

سپس این داده‌ها را به TensorDataset تبدیل می‌کنیم.

مدل شبکه عصبی تعریف شده، از سه لایه تشکیل شده:

لایه ورودی: Linear(input_size → 64)

تابع فعال‌سازی: ReLU

لایه خروجی: Linear(۶۴ → ۱)

پارامترهای تابع آموزش مدل را مشخص می‌کنیم:

Learning_rate:نرخ یادگیری

epochs: تعداد تکرار کامل آموزش روی کل داده

درون تابع، تابع خطا و بهینه ساز را تعریف می‌کنیم.

داده‌های آموزش و تست جدید را با کمک DataLoader به صورت batchهایی کوچک آماده می‌کنیم تا در حلقه آموزش به مدل داده شوند.

در ادامه و در بخش آموزش،

مدل به حالت آموزش (train mode) می‌رود.

داده‌ها درbatch هایی با اندازه ۶۴ پردازش می‌شوند.

گرادیان‌ها صفر می‌شوند، پیش‌بینی انجام می‌شود، خطا محاسبه و backpropagation انجام می‌شود.

مجموع خطاها برای محاسبه میانگین در هر epoch جمع می‌شود.

مدل به حالت ارزیابی (evaluation) می‌رود.

گرادیان‌ها محاسبه نمی‌شوند (بهینه‌سازی متوقف می‌شود).

خطا روی داده تست محاسبه می‌شود.

در هر epoch، نمودار زنده‌ای از تغییرات خطای آموزش و تست رسم می‌شود. مدل با نرخ یادگیری ۰,۰۰۱ و به مدت ۱۰۰ تکرار آموزش داده می‌شود.

خروجی‌ها: دو لیست شامل مقادیر خطای آموزش و تست در هر epoch.

```
# قبلاً تعریف شده‌اند X_train ،y_train ،X_test ،y_test فرض بر این است که
X_train_tensor = torch.tensor(X_train.reshape(X_train.shape[0], -1),
dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
X_test_tensor = torch.tensor(X_test.reshape(X_test.shape[0], -1),
dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

class MLP(nn.Module):
    def __init__(self, input_size):
        super(MLP, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )
    def forward(self, x):
        return self.net(x)

def train_model_final(learning_rate, epochs=100):
    model = MLP(input_size=120)
    criterion = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=64)

    train_losses = []
    test_losses = []

    for epoch in range(epochs):
        model.train()
        train_loss = 0
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            loss.backward()
            optimizer.step()
```

```
            train_loss += loss.item()
        avg_train_loss = train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        model.eval()
        test_loss = 0
        with torch.no_grad():
            for X_batch, y_batch in test_loader:
                y_pred = model(X_batch)
                loss = criterion(y_pred, y_batch)
                test_loss += loss.item()
        avg_test_loss = test_loss / len(test_loader)
        test_losses.append(avg_test_loss)

        # نمودار زنده
        clear_output(wait=True)
        plt.figure(figsize=(8, 5))
        plt.plot(train_losses, label='Train Loss', color='blue')
        plt.plot(test_losses, label='Test Loss', color='orange')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title(f'Loss Curve (lr={learning_rate}) - Epoch {epoch+1}')
        plt.legend()
        plt.grid(True)
        plt.show()

    return train_losses, test_losses

# اجرای مدل با یک نرخ یادگیری
train_loss, test_loss = train_model_final(learning_rate=1e-3, epochs=100)
```

(۲,۴,۱

نمودار خطا را برای داده‌های آموزش و تست به ازای هر سه نرخ یادگیری رسم می‌کنیم:

نمودار اول: نرخ یادگیری ۱

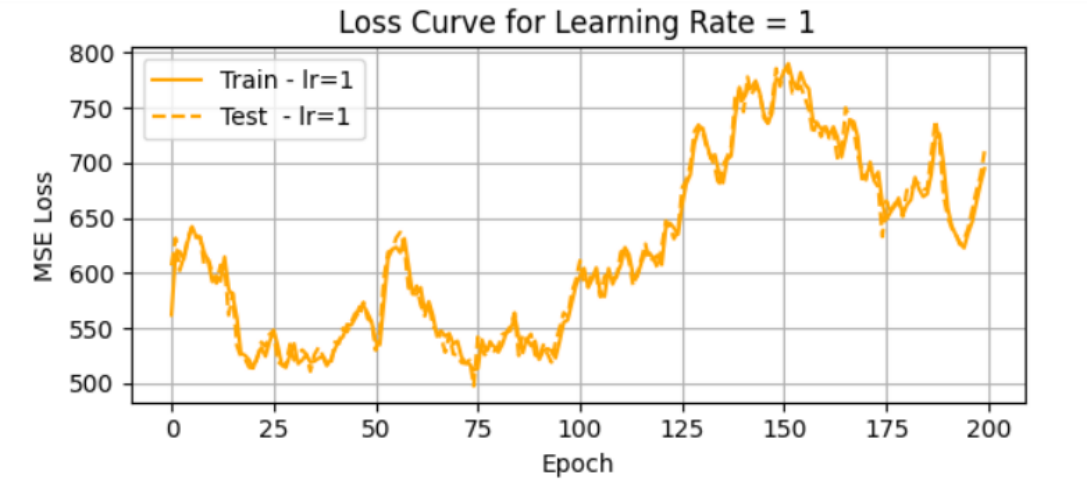
نرخ یادگیری بسیار زیاد است و باعث ناپایداری در آموزش شده. مدل نمی‌تواند به‌درستی مینیمم تابع هزینه را پیدا کند و در عوض بین نقاط مختلف نوسان دارد. مدل دچار واگرایی شده و بهینه‌سازی موفق نیست.

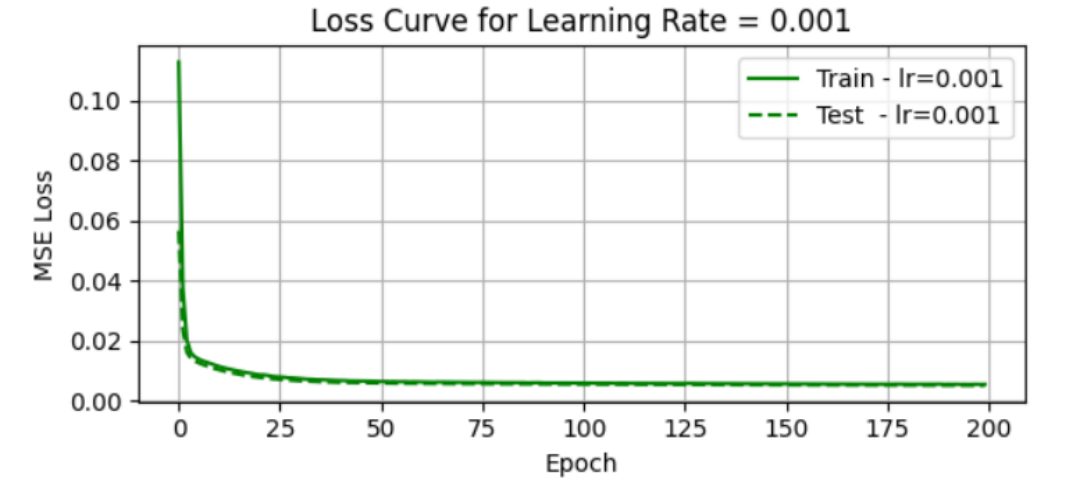
نمودار دوم: نرخ یادگیری ۰,۰۰۱

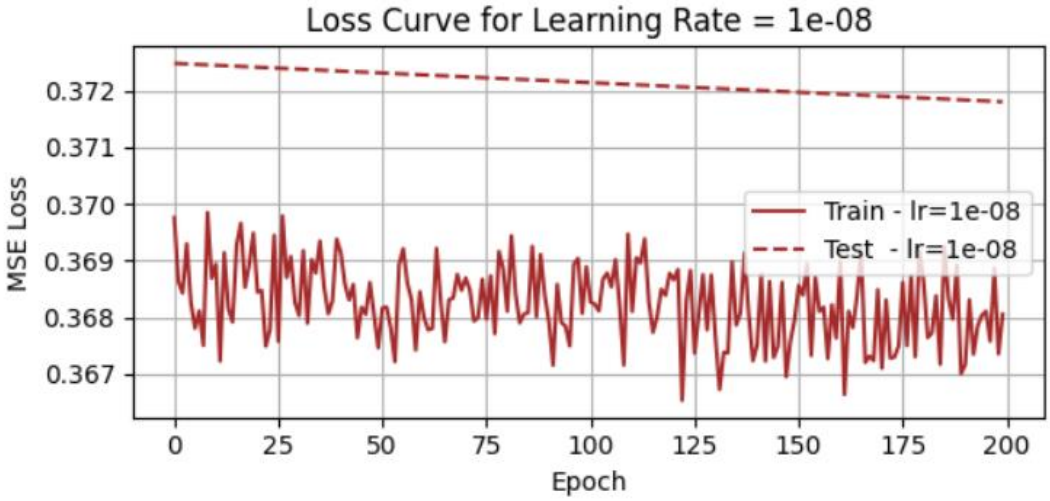
نرخ یادگیری مناسب انتخاب شده است. مدل به‌خوبی در حال یادگیری است و هم Train Loss و هم Test Loss به حداقل رسیده‌اند. یادگیری پایدار و مؤثر، نرخ یادگیری ایده‌آل برای این مسئله.

مودار سوم: نرخ یادگیری e-8۱

نرخ یادگیری بیش از حد کوچک است. مدل تقریباً هیچ چیزی یاد نمی‌گیرد و بهینه‌سازی به کندی یا اصلاً انجام نمی‌شود. مدل دچار ایستایی است و آموزش پیشرفت ندارد.







(۲,۵

ابتدا مدل را تعریف می‌کنیم:

این یک شبکه عصبی چندلایه عمیق (Deep MLP) است با ۳ لایه پنهان:

لایه اول: ۱۲۸ نورون

لایه دوم: ۶۴ نورون

لایه سوم: ۳۲ نورون

خروجی: یک نورون برای پیش‌بینی مقدار عددی

توابع فعال‌سازی: در بین هر لایه از تابع ReLU استفاده شده که رایج‌ترین و ساده‌ترین تابع فعال‌سازی در شبکه‌های عمیق است.

ساختار nn.Sequential باعث ساده‌سازی نوشتار مدل می‌شود و تمام لایه‌ها را پشت سر هم اجرا می‌کند.

در ادامه تابع آموزش مدل را تعریف می‌کنیم:

مدل: نمونه‌ای از کلاسDeepMLP

تابع خطا MSELoss :مناسب برای مسائل رگرسیون

بهینه‌ساز SGD :با نرخ یادگیری ۰,۰۰۱

داده‌ها به صورت mini-batch با اندازه ۶۴ نمونه بارگذاری می‌شوند.

shuffle=True برای داده‌های آموزش باعث تصادفی‌شدن در هر epoch می‌شود.

در حلقه آموزش داریم:

از tqdm برای نمایش نوار پیشرفت استفاده شده.

در هر:epoch مدل در حالت آموزش قرار می‌گیرد.

برای هر:batch خروجی مدل محاسبه می‌شود.

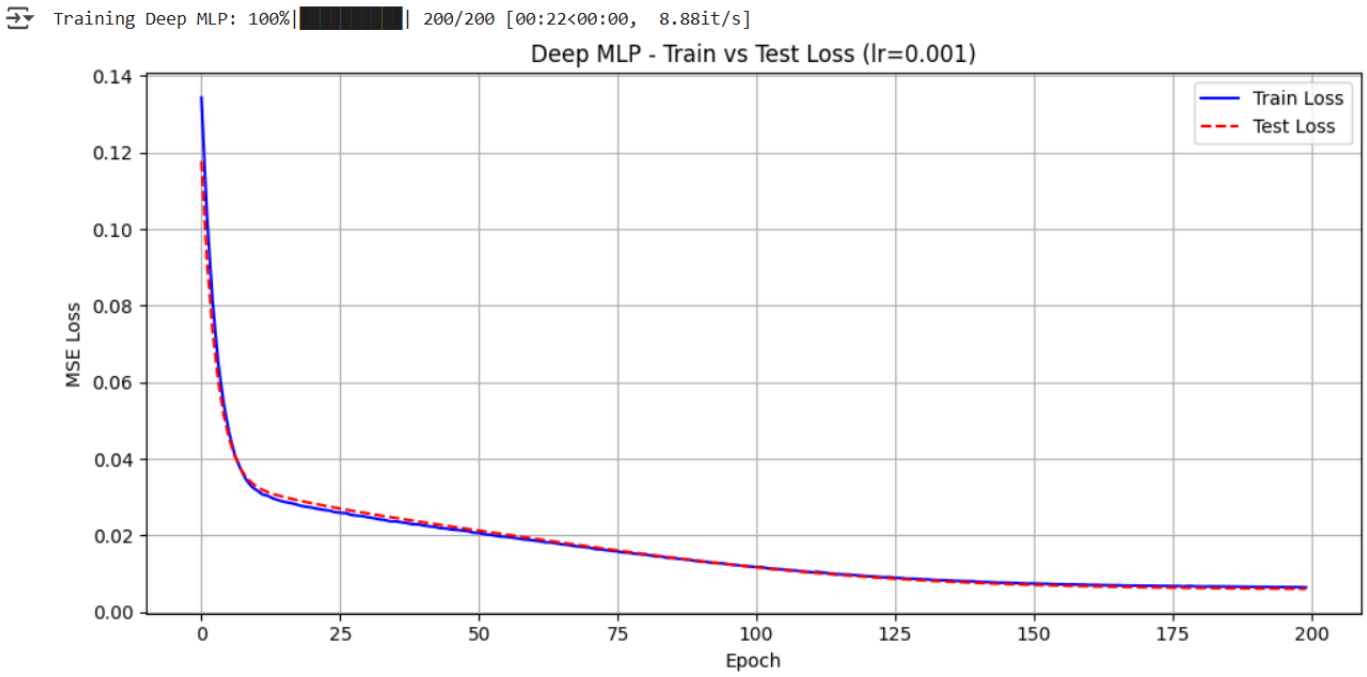
خطا محاسبه شده و گرادیان‌ها محاسبه می‌شوند.

پارامترها به‌روزرسانی می‌شوند.

مقدار خطای آموزش و تست به‌طور جداگانه ذخیره می‌شوند.

در آخر نمودار را رسم می‌کنیم:

این بخش، تغییرات خطای آموزش و تست را در طول ۲۰۰ epoch رسم می‌کنیم.



در این نمودار، خط آبی (Train Loss) و خط قرمز نقطه‌چین (Test Loss) به ترتیب میزان خطای آموزش و تست مدل را طی ۲۰۰ تکرار (epoch) نشان می‌دهند.

کاهش سریع اولیه خطا:

در ۲۰ epoch اول، کاهش بسیار سریعی در هر دو خطا دیده می‌شود که نشان‌دهنده یادگیری خوب مدل در مراحل ابتدایی است.

همگرایی یکنواخت و بدون نوسان: از حدود 50 epoch به بعد، هر دو خطا به‌صورت یکنواخت کاهش یافته و به سمت مقدار کوچکی نزدیک می‌شوند (حدود ۰,۰۱).

نبود نوسان شدید به معنی پایداری آموزش است.

مدل به‌خوبی تعمیم یافته و روی داده‌های دیده‌نشده نیز عملکرد مناسبی دارد.

کارایی نرخ یادگیری ۰,۰۰۱:

انتخاب این نرخ یادگیری باعث همگرایی مناسب، سریع و پایدار شده است.

نه بیش از حد سریع (که منجر به واگرایی شود)، و نه بیش از حد کند (که باعث تأخیر در یادگیری شود).

امتیازی)

ما می‌خواهیم تغییرات یک وزن خاص از مدل را در طول آموزش بررسی کنیم. بنابراین لیستی به نام weight_history تعریف کردیم که مقدار وزن را در هر epoch ذخیره می‌کند.

model.net[0]اشاره به لایه اول شبکه دارد

weight[0][0]انتخاب وزن مربوط به اولین نورون و اولین ورودی.

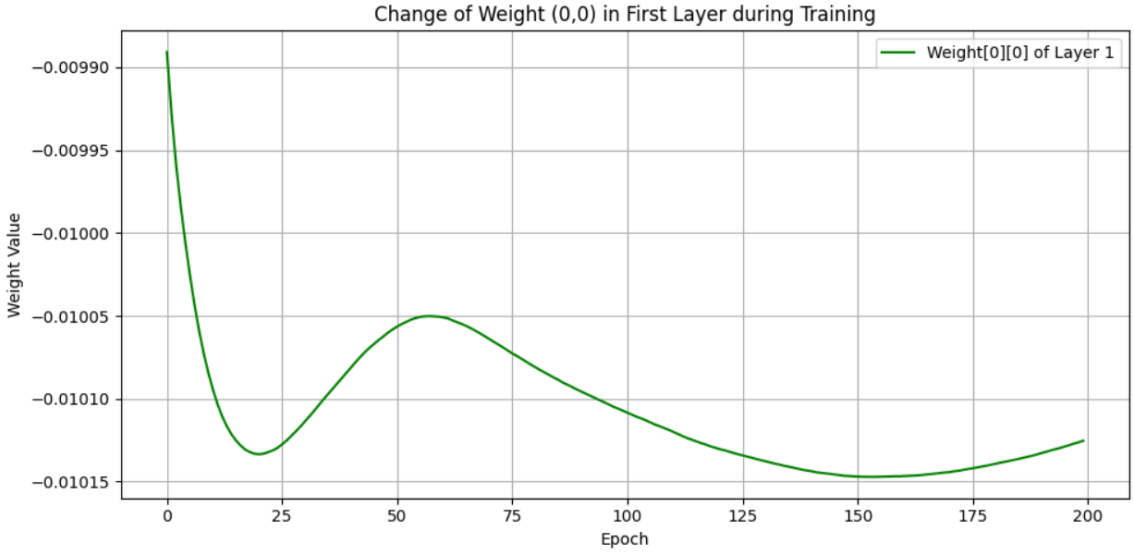
item()تبدیل مقدار وزن از Tensor به عدد.(float)

append(w)اضافه کردن مقدار این وزن به لیست .weight_history

این بخش بعد از مرحله آموزش هر epoch اجرا می‌شود تا مقدار به‌روز شده‌ی وزن را ذخیره کند.

در کنار مقادیر خطای آموزش و تست، حالا لیست weight_history را نیز باز می‌گردانیم تا بتوانیم آن را رسم و تحلیل کنیم.

بخش آخر کد با استفاده از matplotlib، نمودار تغییر مقدار وزن (۰،۰) از لایه اول را در طول ۲۰۰ epoch رسم می‌کند. این کار برای تحلیل اینکه آیا وزن‌ها به خوبی یاد می‌گیرند و همگرا می‌شوند مفید است.



نمودار نشان می‌دهد وزن (۰،۰) در لایه اول در ابتدا سریع کاهش یافته، سپس کمی نوسان داشته و در نهایت به سمت مقدار ثابتی همگرا شده است. این رفتار نشان‌دهنده یادگیری مؤثر و پایداری مدل در طول آموزش است.

سوال ۳)

3.1

هدف اصلی تابع ConvertImageToBinary این است که عکس های داده شده را از لحاظ مقادیر رنگی RGB بررسی کرده و بر اساس این مقادیر به پیکسل های عکس مقدار 1- و یا 1 بدهد که درصورتی که پیکسل روشن باشد آن را سفید درنظر گرفته و به آن مقدار 1- و در صورت تیره بودن آن را مشکی و به آن مقدار1 را اختصاص میدهد این کار باعث میشود تمام عکس به صورت لیستی از 1- , 1 مقدار پیدا کند که با دستور زیر انجام میشود

```
binary_representation = []
```

نحوه کارکرد تابع:

ابتدا تابع تصویر را باز میکند و مقادیر RGB را برای هر پیکسل استخراج میکند و جمع این مقادیر را محاسبه کرده و درصورت بیشتر بودن این جمع از حدی بیشتر، مقدار 1- یعنی سفید به آن پیکسل و درصورت کمتر شدن جمع از حدی، مقدار 1+ یعنی سیاه به آن پیکسل تعلق میگیرد و تصویر به صورت سیاه و سفید ساخته میشود

تابع بالا برای شبکه های عصبی همینگ و هاپفیلد که با مقادیر باینری یا 1,+1- کار میکنند مهم است و دلیل اینکه از 1,+1- به جای 0,+1 استفاده شد این است که در روند شبکه ای مانند هاپفیلد که بر اساس ضرب داخلی ماتریس های ساخته شده باید عدد خروجی معنادار باشد که تشخیص آن از مثبت یا منفی بودن عدد راحت تر است و میتوان شباهت ها را بهتر دید و اگر ضرب داخلی ماتریس ها 0 نتیجه داد یعنی به هم مربوط نبوده و خروجی معناداری ندارند

3.2

شبکه عصبی هاپفیلد یک شبکه بازگشتی بوده که همه نورون های داخل این شبکه به هم وصل هستند و روش تعیین وزن به صورت ضرب خارجی است

$$W = \sum_{i=1}^{n} x^{(i)} . (x^{(i)})^T$$

که در این رابطه هر x^i یک الگوی آموزشی باینری شامل 1,1- است . سعی میکند که خروجی درست را با یک ورودی ناقص بازیابی کند

از طرفی دیگر شبکه عصبی همینگ برای تشخیص بیشترین میزان شباهت الگوی آموزشی به ورودی است که به ازای هر الگوی آموزشی یک نورون داریم و وزن ها همان الگو هستند و خروجی این شبکه ضرب داخلی بین الگو و ورودی مربوطه است و در بین نورون ها، نورونی که بیشترین الگوی شباهت را دارد باقی مانده و بقیه حذف میشوند

حال برای ایجاد نویز در تصاویر داده شده ابتدا لازم است که عکس ها را در مسیر دسترسی قرار دهیم که برای این کار کد زیر را از قسمت کد های آماده برداریم

```
!pip install --upgrade --no-cache-dir gdown

!gdown 1QTi7dJtNAfFR5mG0rd8K3ZGvEIfSn_DS

!unzip PersianData.zip
```

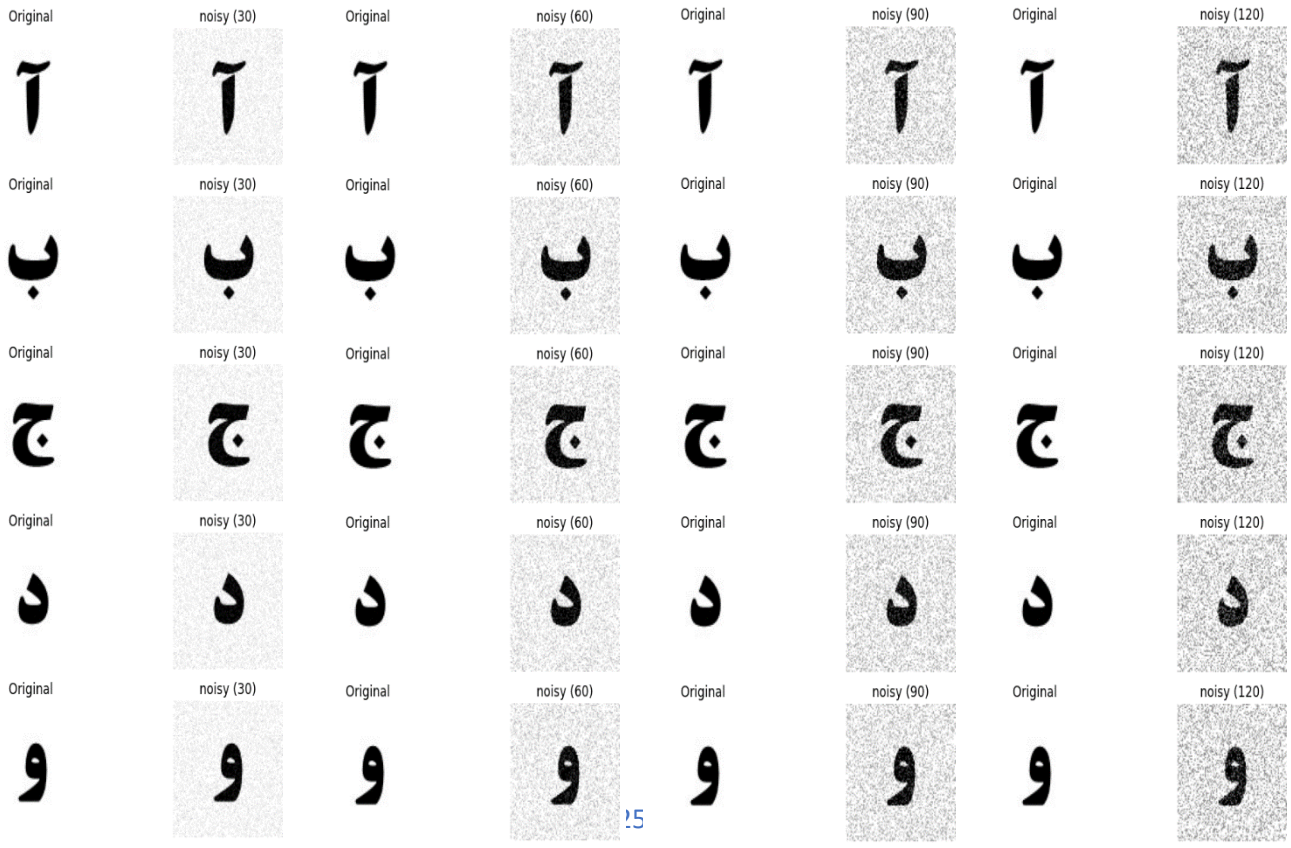
سپس کد های مربوطه به تبدیل باینری عکس و تابع ایجاد نویز را از قسمت کد های آماده نیز وارد میکنیم

حال برای ایجاد کردن نویز بدین صورت عمل میکنیم

ابتدا تصاویر مربوطه را به کمک تابع تبدیل به باینری به عکس به مقادیر پیکسلی **1,+1-** تبدیل میکنیم و روند این کار همانند روند RGB است که در بالاتر گفته شد بنابراین کد مورد نظر باید به صورت زیر باشد

```
patterns = [convertImageToBinary(p) for p in image_paths]
```

سپس باید شبکه هاپفیلد را آموزش دهیم و با استفاده از ()train و الگو های باینری وزن های شبکه، آموزش داده شوند و سپس نویز های مختلفی را برای تاثیر دادن به عکس تعریف میکنیم که به صورت زیر می باشد



3.3-

حالا قصد داریم که با داده های ورودی، خروجی دارای Missing point داشته باشیم و برای این کار تابعی باید بنویسیم

ابتدا در تابع ورودی ها را به صورت عکس و مقدار Missing point تعریف میکنیم. سپس تصویر را باز کرده و اندازه تصویر را استخراج کرده و به کمک دستور draw بر روس عکس تغییر اعمال میکنیم

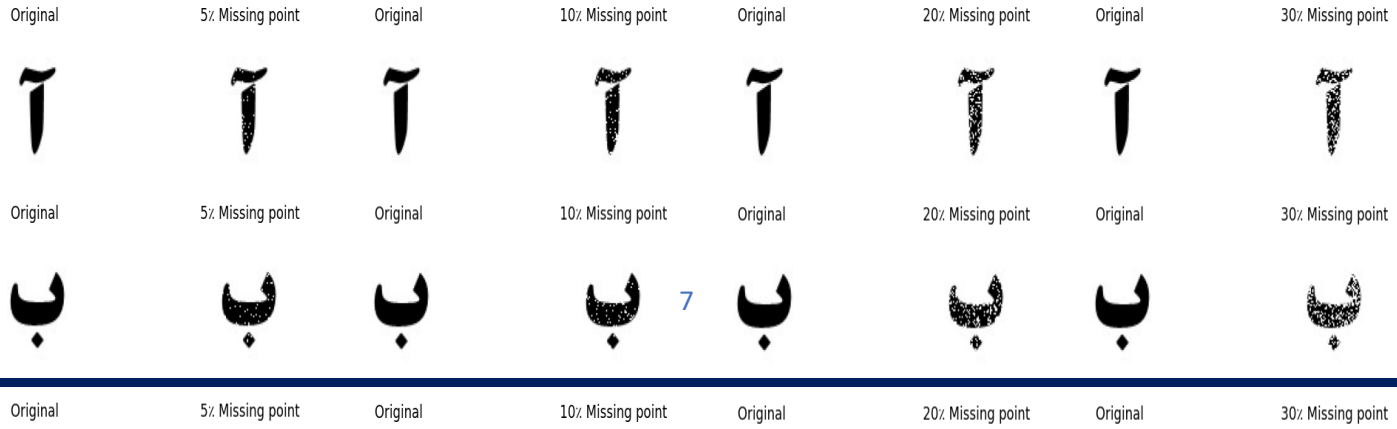
سپس تعداد پیکسل هایی که باید حذف شود را بر حسب درصدی از ابعاد عکس پیدا کرده ار بین مختصات های ممکن هر پیکسل، به طور تصادفی تعدادی از پیکسل ها را بر حسب مقدار درصد Missing point با خاکستری رنگ میکنیم بنابراین کد تابع به صورت زیر است:

```
def getPointMissingImage(input_path, missing_ratio=0.1):


    # باز کردن تصویر و ایجاد ابزار ترسیم
    image = Image.open(input_path)
    draw = ImageDraw.Draw(image)
    width, height = image.size
    pix = image.load()


    # تعداد پیکسل‌هایی که باید حذف شوند
    total_pixels = width * height
    missing_pixels_count = int(total_pixels * missing_ratio)


    # تولید مختصات تصادفی برای حذف
    missing_coords = random.sample([(x, y) for x in range(width) for y in
range(height)], missing_pixels_count)


    # اعمال حذف با رنگ سفید (یا خاکستری دلخواه)
    for x, y in missing_coords:
        draw.point((x, y), (255, 255, 255))  # یا مثلاً (۱۲۸،۱۲۸،۱۲۸) برای خاکستری


    del draw
    return image
```

حال تنها کافی است که ورودی ها را به این تابع بدهیم و آن ها را نمایش دهیم

```
# نسبت‌های مختلف حذف پیکسل
missing_ratios = [0.05, 0.1, 0.2, 0.3]


for ratio in missing_ratios:
    print(f"\n {int(ratio * 100)}٪missing points test ")
    fig, axs = plt.subplots(len(image_paths), 2, figsize=(7, 10))


    for i, path in enumerate(image_paths):
        # ساخت نسخه دارای نقاط حذف‌شده
        missing_img = getPointMissingImage(path, missing_ratio=ratio)
        missing_img.save(f"/content/point_missing_{i+1}_r{int(ratio*100)}.jpg")


        # نمایش
        axs[i, 0].imshow(Image.open(path))
        axs[i, 0].set_title("Original ")
        axs[i, 1].imshow(missing_img)
        axs[i, 1].set_title(f" {int(ratio*100)}٪Missing point")


        for ax in axs[i]:
            ax.axis("off")


    plt.tight_layout()
    plt.show()
```

نتیجه کد:

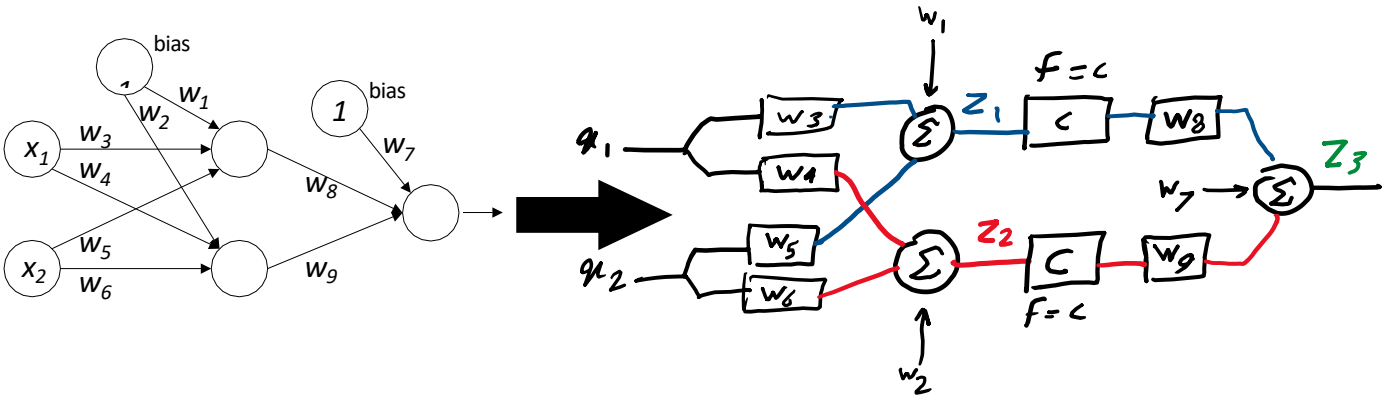


حال سوال این است که اگر میزان این Missing point ها زیاد شود اختلال را چگونه میتوان رفع کرد؟

در این صورت اگر بالای 40% یا 50% missing point داشته باشیم اطلاعات زیادی از دست میرود و بازسازی عکس به کمک شبکه عصبی هاپفیلد که بر اساس شباهت پیکسل ها کار میکند خیلی سخت و حتی ناممکن میشود

برای رفع این مشکل راه های مختلفی وجود دارد بر فرض مثال اگر پیکسل حذف شده مشخص باشد میتوان با میانگین گیری از پیکسل های اطراف، آن را تقریب زد همینطور میتوان از شبکه های عصبی قوی تری مانند Autoencoder و یا Denoising Autoencoder جهت بازسازی عکس استفاده کرد از شبکه عصبی هاپفیلد قوی تر هستند همینطور میتوان از تکنیک های training بهتر که به نویز مقاوم تر هستند استفاده کرد مانند Robust Training و یا Regularization

سوال ۴)



$$Z_1 = W_3X_1 + W_5X_2 + W_1$$
$$Z_2 = W_4X_1 + W_6X_2 + W_2$$
$$Z_3 = W_8Z_1C + W_9Z_2C + W_7$$
$$Z_3 = W_8(W_3X_1 + W_5X_2 + W_1)C + W_9(W_4X_1 + W_6X_2 + W_2)C + W_7$$
$$Y = \frac{1}{1 - e^{-Z_3}} = \frac{1}{1 - e^{-(W_8(W_3X_1+W_5X_2+W_1)C+W_9(W_4X_1+W_6X_2+W_2)C+W_7)}}$$

مرز تصمیم گیری:

$$Y = 0.5 \rightarrow \frac{1}{1 - e^{-Z_3}} = 0.5 \rightarrow Z_3 = 0$$

$$Z_3 = W_8(W_3X_1 + W_5X_2 + W_1)C + W_9(W_4X_1 + W_6X_2 + W_2)C + W_7 = 0$$

معادله خط و سیستم جایگزین پیشنهادی:

$$(CW_8W_3 + CW_9W_4)X_1 + (CW_8W_5 + CW_9W_6)X_2 + (W_1 + W_2)C + W_7 = 0$$

W'_1 W'_2 B'_1

شکل سیستم جدید: