مهم: در تمامی مراحل کد نویسی، مقدار random\_state = 53 قرار گرفته شده است.

بسم الله الرحمن الرحیم

دانشجو: محمدرضا جنیدی جعفری 9925253

درس مبانی سیستم های هوشمند

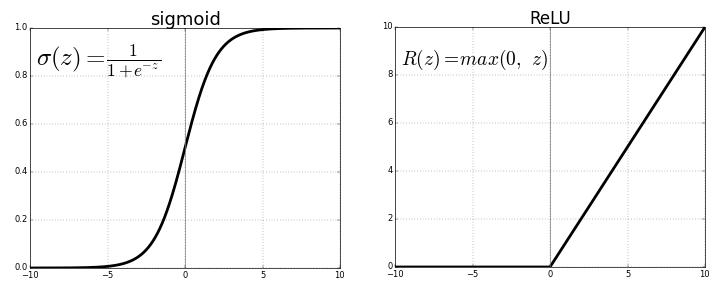
استاد: دکتر مهدی علیاری

مینی پروژه دوم

[لینک مخزن گیت هاب](https://github.com/mrjoneidi/FIS) - [گوگل کولب سوال 1](https://colab.research.google.com/drive/1FXQRz-ORQQj8WMYfarw7ubucuhWPwN7k?usp=sharing) - [گوگل کولب سوال 2](https://colab.research.google.com/drive/1osbVDYjkeqv8TiBotIHu4j6vW2b4n-W2?usp=sharing) – [گوگل کولب سوال 3](https://colab.research.google.com/drive/1djD-g9jd2N57ZJw5nVDI73wx8w4YOif7?usp=sharing) - [گوگل کولب سوال 4](https://colab.research.google.com/drive/1hHF-vqtMRX_tiqyeNYapUFTnE34Mqoy6?usp=sharing)



بهتر است قبل از نظر دادن نگاه کلی به نمودار توابع سیگموید و ReLU بیاندازیم:



همانطور که مشاهده می شود، تابع ReLU ارزش تمامی واحد های کمتر از صفر را صفر در نظر می گیرد و در مقابل تمامی واحدهای بزرگتر از 0 را به همان ارزش خود در نظر می گیرد. تابع سیگموید مقادیر ورودی را به بازه[0,1] نگاشت می‌کند، که این بازه را می‌توان به‌عنوان احتمال تفسیر کرد.

اگر این دو لایه پشت هم در دو لایه آخر یک مدل وجود داشته باشد:

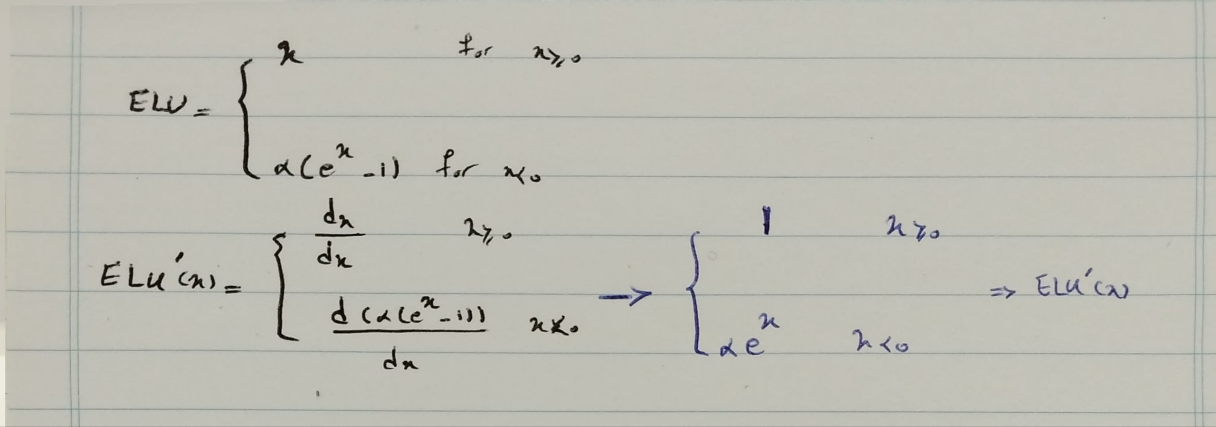
* هر مقدار مثبت از خروجی ReLU به سیگموید ارسال می‌شود و سیگموید آن را به احتمال بین[0,1] نگاشت می‌کند.
* اگر خروجی ReLU صفر باشد، سیگموید مقدار 0.5 را برمی‌گرداند.

این رفتار می‌تواند مشکل‌ساز باشد، زیرا خروجی صفر از ReLU برای سیگموید معنای خاصی ندارد و باعث ابهام در پیش‌بینی‌ها می‌شود.

مشکل بعدی که ممکن است رخ دهد، ایجاد مشکل در گرادیان یا بهینه سازی است:

گرادیان‌های تولیدشده برای پارامترهای مدل ممکن است ناسازگار شوند، زیرا ReLU و سیگموید رفتارهای کاملاً متفاوتی دارند. این امر می‌تواند به یادگیری ناکارآمد یا گیر افتادن در مقادیر نامناسب منجر شود.

در نهایت برداشت من این است که مدل در پیشبینی احتمال خطای بالایی خواهد داشت.

1. 

در زیر دو مزیت ELU نسبت به ReLU آورده شده است:

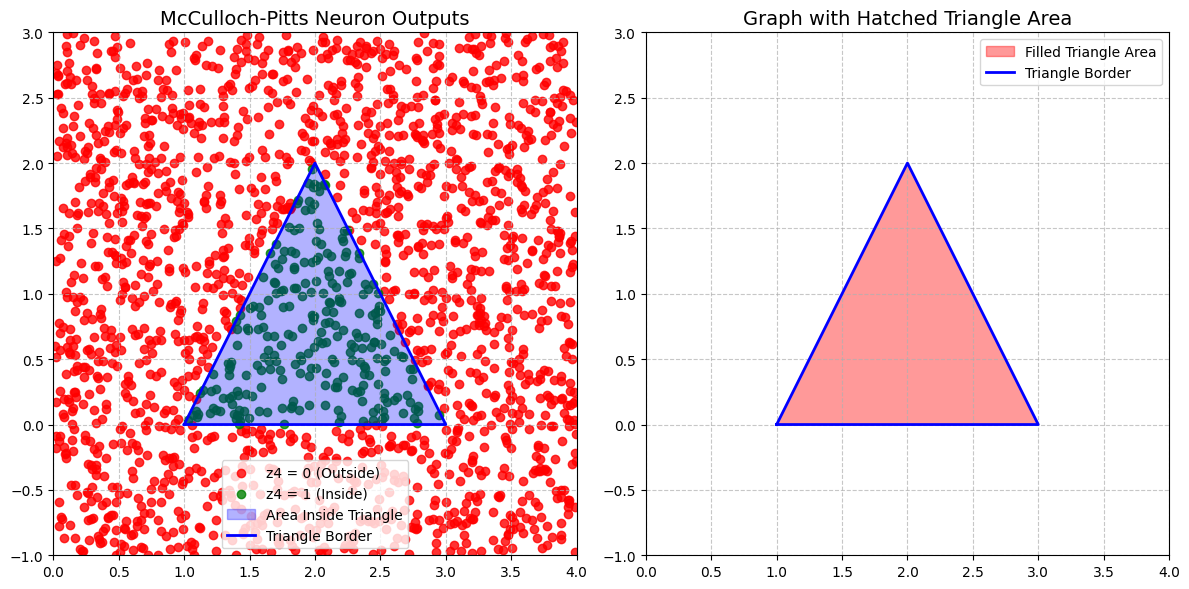
* در ReLU، مقادیر منفی ورودی به صفر تبدیل می‌شوند، که می‌تواند باعث "مرده شدن نورون‌ها" (Dead Neurons) شود. این یعنی نورون‌های شبکه دیگر به گرادیان حساس نیستند و یادگیری متوقف می‌شود. اما در ELU، مقادیر منفی به صورت غیرخطی و نرم به مقادیری نزدیک به صفر نگاشت می‌شوند، که امکان یادگیری را حفظ می‌کند.
* ELU به دلیل قسمت نمایی خود، خروجی‌های منفی تولید می‌کند که میانگین خروجی نورون‌ها را به سمت صفر نگه می‌دارد. این رفتار شبیه به نرمال‌سازی خروجی است و به تسریع یادگیری کمک می‌کند. در ReLU، خروجی‌های صفر یا مثبت هستند، که ممکن است میانگین خروجی‌ها را از صفر دور کند.



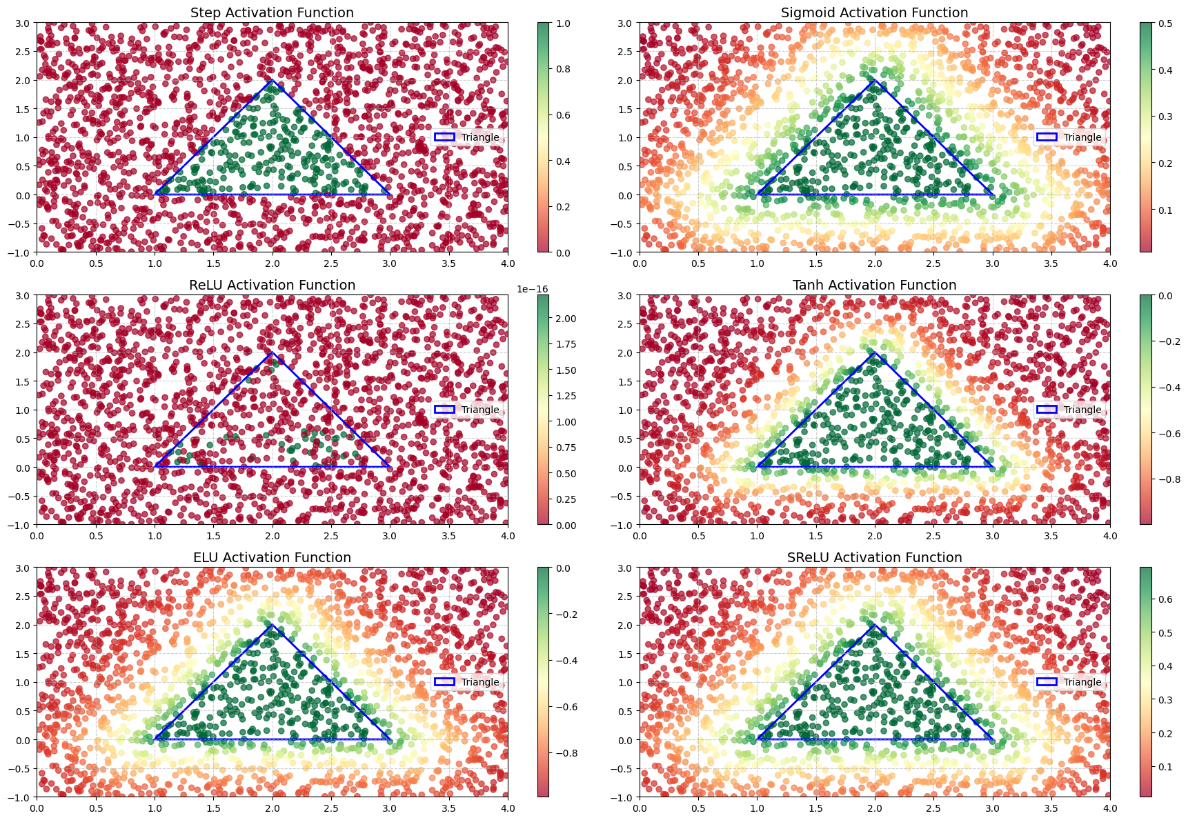
کد من، با استفاده از روش **مختصات باری‌سنتریک**، نقاط تصادفی تولید شده در یک بازه مشخص را بررسی می‌کند تا تعیین کند آیا این نقاط داخل یا خارج یک مثلث تعریف‌شده قرار دارند. این فرایند شامل مراحل زیر است:

1. **تعریف مثلث**:
   * مختصات سه رأس مثلث به صورت دستی تعریف شده‌اند.
2. **بررسی نقاط با مختصات باری‌سنتریک**:
   * با محاسبه مساحت مثلث اصلی و سه زیرمثلث (که توسط نقطه و دو رأس مثلث ساخته می‌شوند)، مختصات باری‌سنتریک نقطه محاسبه می‌شود.
   * اگر مجموع مختصات باری‌سنتریک s+t+u برابر ۱ باشد و هر یک در بازه [0,1] باشند، نقطه داخل مثلث است.
3. **تولید نقاط تصادفی**:
   * ۲۰۰۰ نقطه تصادفی در محدوده مشخص برای محورهای x و y تولید شده‌اند.
4. **طبقه‌بندی نقاط**:
   * نقاطی که داخل مثلث هستند، برچسب 1 می‌گیرند و نقاط خارج مثلث برچسب.0
5. **نمایش نتایج**:
   * نمودار اول: نقاط داخل مثلث (سبز) و خارج مثلث (قرمز) نمایش داده می‌شوند. مثلث با رنگ آبی پر شده است.
   * نمودار دوم: مثلث به صورت جداگانه با رنگ قرمز و خطوط ضخیم‌تر نمایش داده می‌شود.
6. **چاپ مختصات مثلث**:
   * مختصات رأس‌های مثلث برای مرجع چاپ می‌شوند.

**نتیجه کلی:**

* کد نشان می‌دهد که چگونه می‌توان با استفاده از مختصات باری‌سنتریک و هندسه ساده، نقاط داخل یا خارج یک مثلث را شناسایی و به صورت گرافیکی نمایش داد.
* نقاط و مثلث به طور واضح و قابل فهم طبقه‌بندی و ترسیم شده‌اند.

در زیر اثر تابع فعال ساز بررسی می شود:



**Step Activation**:

* بهترین عملکرد را در طبقه‌بندی نقاط داخل مثلث داشت
* اما خروجی آن فقط باینری (۰ یا ۱) است

**SReLU**:

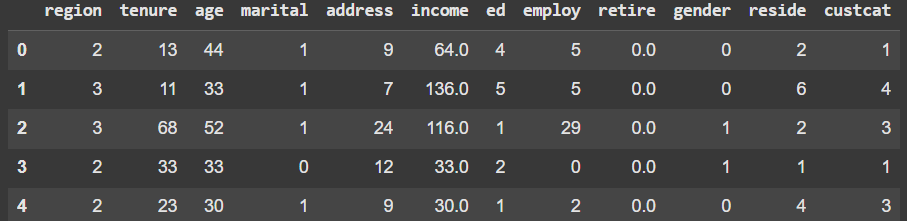
* بهترین عملکرد را برای طبقه‌بندی پیوسته داشت
* توانست نقاط بیشتری را با خروجی منطقی شناسایی کند

**Sigmoid**:

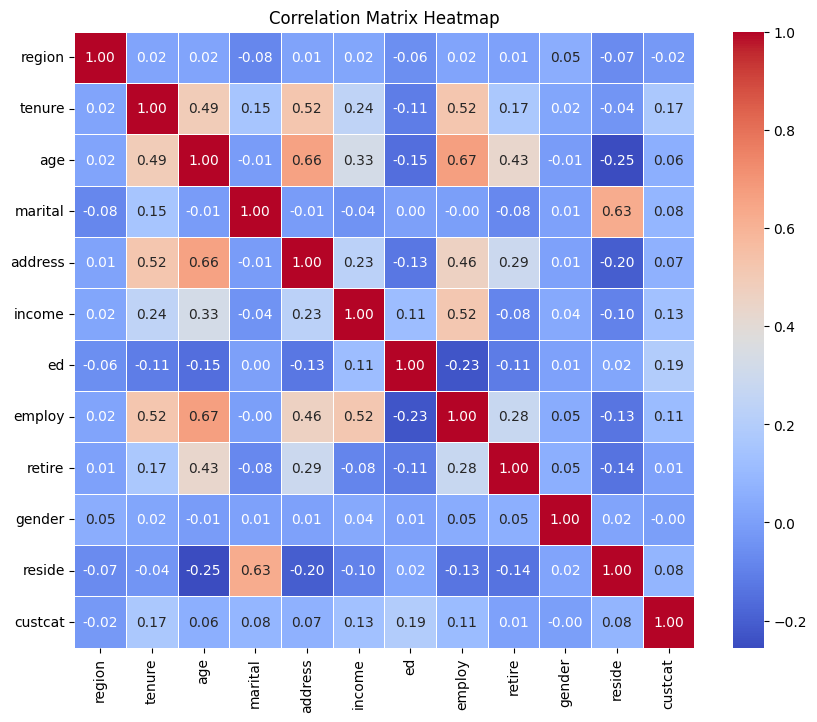
* عملکرد نسبتا ضعیف در شناسایی نقاط داخل مثلث داشت

**ReLU، Tanh، ELU**:

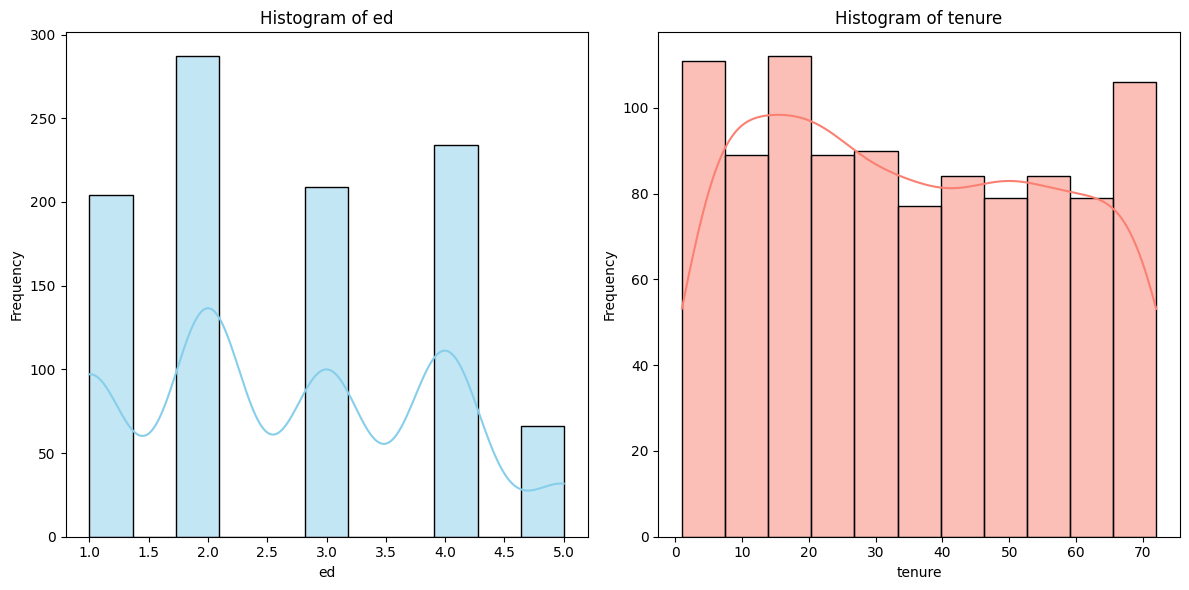
* به دلیل ماهیت توابع و مقیاس تصمیم‌گیری، هیچ نقطه‌ای را داخل مثلث شناسایی نکردند

1. داده ها را با استفاده از تابع پانداز فرامیخوانیم:

هیت مپ زیر نشان دهنده ی همبستگی میان ویژگی هاست. با توجه به نمودار زیر، دو ویژگی ed و tenure بیشترین همبستگی را به داده هدف ما دارند.



دو ویژگی با بیشترین همبستگی به داده هدف:



در ادامه کد، کلاس های داده ی هدف را از 1 تا 4، به 0 تا 3 تبدیل می کنیم و داده ها را به 65% برای آموزش، 20% برای ارزیابی و 15% برای داده تست تقسیم می کنیم.

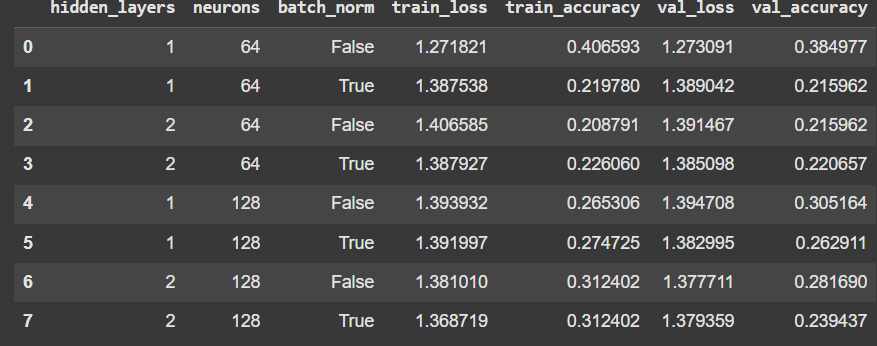
در مرحله بعد با تنظیمات زیر اقدام به آموزش مدل میکنیم:

neurons\_cases = [64, 128]  # Two cases for the number of neurons

hidden\_layers\_cases = [1, 2]  # One hidden layer vs two hidden layers

batch\_norm\_cases = [False, True]  # With and without batch normalization

که نتایج آن به صورت زیر است:



از میان مدل های بالا، بهترین مدل از تک لایه و دو لایه( در مجموع دو مدل) را برای ادامه مسیر انتخاب میکنیم: ( معیار مقدار خطا بود)

کانفیگ دو مدل:

# Configurations for the best models

configurations = [

    {"neurons": 64, "hidden\_layers": 1},  # Best model with 1 hidden layer

    {"neurons": 128, "hidden\_layers": 2}  # Best model with 2 hidden layers

]

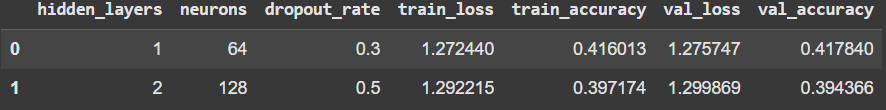
به ازای مقادیر زیر برای dropout دوباره مدل ها را آموزش می دهیم:

dropout\_rates = [0.0, 0.3, 0.5]

نتایج آن به صورت زیر است:

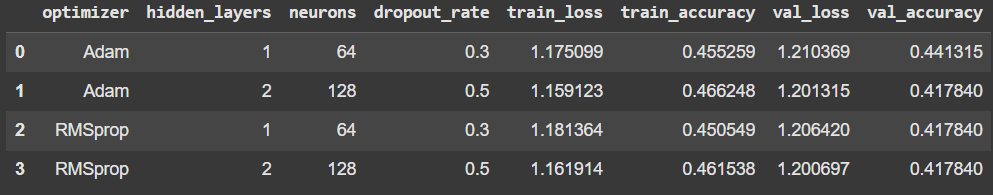


باز مثل قبل، دو مدل بهتر را انتخاب می کنیم و regularization را اعمال میکنیم:



این دو مدل، فعلا بهترین نتایج را داشته اند.

در مرحله آخر، بهینه ساز adam و RMSprob را اعمال میکنیم:



و حالا، برای هر 4 مدل به صورت تصادفی 10 نمونه را تست میگیریم:

==============================================================================

Results for Model with Optimizer: Adam, Hidden Layers: 1, Neurons: 64, Dropout Rate: 0.3

==============================================================================

Index True Label Predicted Label Prediction Probabilities

--------------------------------------------------------------------------------

8 0 0 [0.4827, 0.1231, 0.2415, 0.1527]

116 1 2 [0.1747, 0.1395, 0.6014, 0.0845]

73 3 3 [0.1003, 0.2282, 0.2337, 0.4378]

100 3 2 [0.1266, 0.2153, 0.5065, 0.1516]

22 2 0 [0.3993, 0.1247, 0.3966, 0.0794]

50 1 0 [0.3883, 0.1392, 0.2399, 0.2326]

102 0 0 [0.5797, 0.0370, 0.3327, 0.0506]

71 0 3 [0.2035, 0.1862, 0.2365, 0.3738]

69 0 0 [0.5008, 0.0552, 0.3918, 0.0521]

121 1 1 [0.0592, 0.3882, 0.1744, 0.3782]

==============================================================================

Retraining Model with 2 hidden layers, 128 neurons, Dropout Rate: 0.5, Optimizer: Adam

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 54ms/step

==============================================================================

Results for Model with Optimizer: Adam, Hidden Layers: 2, Neurons: 128, Dropout Rate: 0.5

==============================================================================

Index True Label Predicted Label Prediction Probabilities

--------------------------------------------------------------------------------

8 0 0 [0.4481, 0.1327, 0.2690, 0.1501]

116 1 2 [0.2157, 0.0754, 0.6565, 0.0524]

73 3 3 [0.0353, 0.2689, 0.1409, 0.5549]

100 3 2 [0.1420, 0.2401, 0.4541, 0.1638]

22 2 0 [0.5020, 0.0786, 0.3546, 0.0647]

50 1 0 [0.3800, 0.1266, 0.2587, 0.2347]

102 0 0 [0.5638, 0.0244, 0.3783, 0.0335]

71 0 3 [0.2086, 0.1435, 0.2294, 0.4184]

69 0 0 [0.5529, 0.0215, 0.4010, 0.0246]

121 1 1 [0.0392, 0.4237, 0.1847, 0.3524]

==============================================================================

Retraining Model with 1 hidden layers, 64 neurons, Dropout Rate: 0.3, Optimizer: RMSprop

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 72ms/step

==============================================================================

Results for Model with Optimizer: RMSprop, Hidden Layers: 1, Neurons: 64, Dropout Rate: 0.3

==============================================================================

Index True Label Predicted Label Prediction Probabilities

--------------------------------------------------------------------------------

8 0 0 [0.4622, 0.1323, 0.2511, 0.1545]

116 1 2 [0.1765, 0.1889, 0.5393, 0.0953]

73 3 3 [0.1320, 0.1917, 0.2353, 0.4410]

100 3 2 [0.1722, 0.1261, 0.5942, 0.1075]

22 2 0 [0.4226, 0.0978, 0.4149, 0.0647]

50 1 0 [0.3981, 0.1510, 0.2108, 0.2402]

102 0 0 [0.5804, 0.0562, 0.3000, 0.0634]

71 0 3 [0.2545, 0.1537, 0.2245, 0.3673]

69 0 0 [0.4723, 0.0558, 0.4087, 0.0632]

121 1 1 [0.0579, 0.3900, 0.1702, 0.3820]

==============================================================================

Retraining Model with 2 hidden layers, 128 neurons, Dropout Rate: 0.5, Optimizer: RMSprop

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 56ms/step

==============================================================================

Results for Model with Optimizer: RMSprop, Hidden Layers: 2, Neurons: 128, Dropout Rate: 0.5

==============================================================================

Index True Label Predicted Label Prediction Probabilities

--------------------------------------------------------------------------------

8 0 0 [0.5188, 0.1003, 0.2850, 0.0959]

116 1 2 [0.1960, 0.0883, 0.6555, 0.0602]

73 3 3 [0.0453, 0.2462, 0.1472, 0.5613]

100 3 2 [0.1000, 0.1642, 0.6110, 0.1248]

22 2 0 [0.4903, 0.0641, 0.3964, 0.0492]

50 1 0 [0.4050, 0.1281, 0.2400, 0.2268]

102 0 0 [0.5477, 0.0292, 0.3867, 0.0364]

71 0 3 [0.1731, 0.1588, 0.2102, 0.4579]

69 0 2 [0.4567, 0.0373, 0.4735, 0.0325]

121 1 1 [0.0367, 0.4424, 0.1903, 0.3307]

==============================================================================

در نهایت، یک مدل ساده دنس را پیاده سازی می کنیم و نتایج را با حالت قبل مقایسه می کنیم:

import numpy as np

from tensorflow.keras.utils import to\_categorical

from sklearn.metrics import accuracy\_score, classification\_report

# Function to create a model (reused from previous steps)

def create\_model(neurons, hidden\_layers, dropout\_rate, optimizer\_name):

    model = Sequential()

    model.add(Dense(neurons, activation='relu', input\_dim=X\_train.shape[1]))

    model.add(Dropout(dropout\_rate))

    # Add additional hidden layers

    for \_ in range(hidden\_layers - 1):

        model.add(Dense(neurons, activation='relu'))

        model.add(Dropout(dropout\_rate))

    model.add(Dense(4, activation='softmax'))  # Output layer for 4 classes

    # Optimizer

    if optimizer\_name == 'Adam':

        optimizer = Adam(learning\_rate=0.001)

    elif optimizer\_name == 'RMSprop':

        optimizer = RMSprop(learning\_rate=0.001)

    # Compile model

    model.compile(optimizer=optimizer,

                  loss='sparse\_categorical\_crossentropy',

                  metrics=['accuracy'])

    return model

# Best model configurations

best\_models = [

    {"neurons": 64, "hidden\_layers": 1, "dropout\_rate": 0.3, "optimizer": "Adam"},  # Best 1-layer model

    {"neurons": 128, "hidden\_layers": 2, "dropout\_rate": 0.5, "optimizer": "RMSprop"}  # Best 2-layer model

]

# Retrain the models and get predictions

test\_predictions = []

train\_predictions = []

for config in best\_models:

    neurons = config['neurons']

    hidden\_layers = config['hidden\_layers']

    dropout\_rate = config['dropout\_rate']

    optimizer\_name = config['optimizer']

    print(f"Training Model: {hidden\_layers} Hidden Layers, {neurons} Neurons, Dropout Rate: {dropout\_rate}, Optimizer: {optimizer\_name}")

    # Create and train the model

    model = create\_model(neurons=neurons, hidden\_layers=hidden\_layers, dropout\_rate=dropout\_rate, optimizer\_name=optimizer\_name)

    history = model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_val, y\_val), verbose=0)

    # Evaluate train and validation loss and accuracy

    train\_loss, train\_accuracy = model.evaluate(X\_train, y\_train, verbose=0)

    val\_loss, val\_accuracy = model.evaluate(X\_val, y\_val, verbose=0)

    print(f"Train Loss: {train\_loss:.4f}, Train Accuracy: {train\_accuracy:.4f}")

    print(f"Validation Loss: {val\_loss:.4f}, Validation Accuracy: {val\_accuracy:.4f}")

    # Get probabilities on the train and test sets

    train\_probs = model.predict(X\_train)

    test\_probs = model.predict(X\_test)

    train\_predictions.append(train\_probs)

    test\_predictions.append(test\_probs)

# Combine predictions using soft voting

ensemble\_train\_predictions = np.mean(train\_predictions, axis=0)  # Average probabilities for train

ensemble\_test\_predictions = np.mean(test\_predictions, axis=0)  # Average probabilities for test

# Convert probabilities to class labels

ensemble\_train\_labels = np.argmax(ensemble\_train\_predictions, axis=1)

ensemble\_test\_labels = np.argmax(ensemble\_test\_predictions, axis=1)

# Evaluate the ensemble model on the train set

ensemble\_train\_accuracy = accuracy\_score(y\_train, ensemble\_train\_labels)

print(f"\nEnsemble Model Accuracy on Train Set: {ensemble\_train\_accuracy:.4f}")

# Train classification report

print("\nClassification Report for Ensemble Model (Train Set):")

print(classification\_report(y\_train, ensemble\_train\_labels))

# Evaluate the ensemble model on the test set

ensemble\_test\_accuracy = accuracy\_score(y\_test, ensemble\_test\_labels)

print(f"\nEnsemble Model Accuracy on Test Set: {ensemble\_test\_accuracy:.4f}")

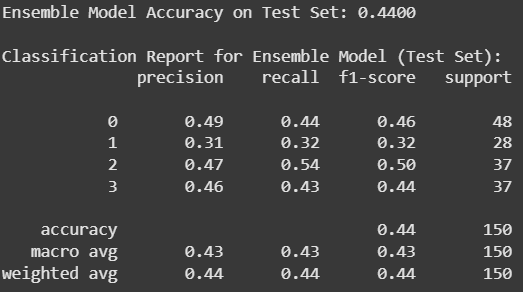
# Test classification report

print("\nClassification Report for Ensemble Model (Test Set):")

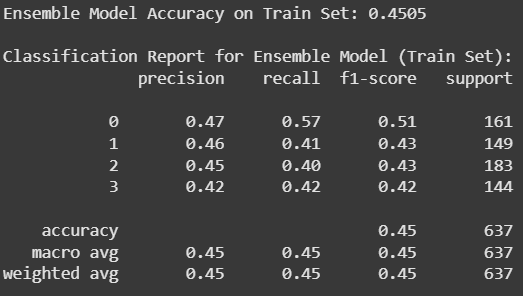
print(classification\_report(y\_test, ensemble\_test\_labels))

که نتایج به صورت زیر بود:

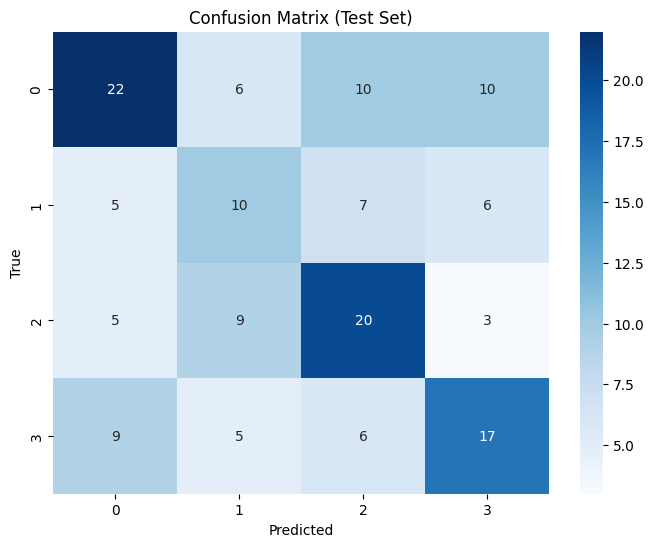
Training Model: 1 Hidden Layers, 64 Neurons, Dropout Rate: 0.3, Optimizer: Adam

Train Loss: 1.1947, Train Accuracy: 0.4521

Validation Loss: 1.2162, Validation Accuracy: 0.4554



ماتریس هم بستگی به صورت زیر خواهد بود:





تابع تبدیل به باینری:

این تابع یک تصویر رنگی را به یک آرایه باینری تبدیل می‌کند که در آن هر پیکسل به یکی از دو مقدار زیر تبدیل می‌شود:

* -1 برای پیکسل‌های سفید (پیکسل‌های با شدت نور بالا)
* 1 برای پیکسل‌های سیاه (پیکسل‌های با شدت نور پایین)

عملکرد:

* تصویر از مسیر مشخص‌شده بارگذاری و به فرمت RGB تبدیل می‌شود.
* هر پیکسل تصویر بررسی می‌شود:
  + مجموع مقادیر RGB (شدت نور) محاسبه می‌شود.
  + اگر شدت نور پیکسل بیشتر از یک آستانه (threshold) باشد، آن پیکسل به سفید و مقدار -1 تبدیل می‌شود.
  + در غیر این صورت، پیکسل به سیاه و مقدار 1 تبدیل می‌شود.
* مقادیر باینری حاصل به‌صورت یک آرایه ذخیره می‌شوند و در خروجی برگردانده می‌شوند.

چالش استفاده از این تابع، مقدار بهینه factor(threshold) است. در زیر چند نمونه از روش هایی که می توان از آنها استفاده کرد بیان شده است (کد های آن به صورت کامنت در دفترچه کد موجود است):

* استفاده از روش دستی
* روش اوتسو (Otsu's Method):

روش اوتسو به طور خودکار مقدار آستانه بهینه را برای تصویر پیدا می‌کند. این روش مبتنی بر یافتن کمینه‌ی واریانس درون‌کلاسی بین پیکسل‌های روشن و تاریک است.

* استفاده از آستانه تطبیقی:

این روش آستانه را به‌صورت محلی بر اساس شدت نور در نواحی مختلف تصویر محاسبه می‌کند. مناسب برای تصاویری است که شدت نور در نقاط مختلف متفاوت است.

* تجربه عملی و کنتراست تصویر

تابع تولید نویز:

این کد تصاویر را می‌گیرد، به هر پیکسل نویز تصادفی اضافه می‌کند و تصاویر جدید را با نویز ذخیره می‌کند. در مراحل زیر کار می‌کند:

* تصاویر ورودی را از لیستی می‌خواند.
* به هر پیکسل تصویر یک مقدار نویز تصادفی اضافه می‌کند.
* مطمئن می‌شود که مقادیر رنگ (RGB) بین 0 تا 255 باقی بمانند.
* تصویر تغییر داده‌شده را در یک فایل جدید ذخیره می‌کند.

کاربرد آن شبیه‌سازی تصاویر نویزی برای آموزش مدل‌های یادگیری ماشین، آزمایش الگوریتم‌های پردازش تصویر یا بررسی مقاومت سیستم‌ها در برابر نویز است.

تغییراتی در توابع دادم و نتایج به صورت زیر است:

1. تبدیل به باینری:
2. from PIL import Image, ImageDraw
3. def convertImageToBinary(path):
4. image = Image.open(path)
5. draw = ImageDraw.Draw(image)
6. width, height = image.size
7. pix = image.load()
8. factor = 1000
9. binary\_representation = []
10. for i in range(width):
11. for j in range(height):
12. red, green, blue = pix[i, j]
13. total\_intensity = red + green + blue
14. if total\_intensity > (((255 + factor) // 2) \* 3):
15. red, green, blue = 255, 255, 255
16. binary\_representation.append(-1)
17. else:
18. red, green, blue = 0, 0, 0
19. binary\_representation.append(1)
20. draw.point((i, j), (red, green, blue))
21. del draw
22. return binary\_representation

2- تولید عکس با نویز:

from PIL import Image, ImageDraw

import random

def generateNoisyImages():

    image\_paths = [

        "/content/1.jpg",

        "/content/2.jpg",

        "/content/3.jpg",

        "/content/4.jpg",

        "/content/5.jpg"

    ]

    for i, image\_path in enumerate(image\_paths, start=1):

        noisy\_image\_path = f"/content/noisy{i}.jpg"

        getNoisyBinaryImage(image\_path, noisy\_image\_path)

        print(f"Noisy image for {image\_path} generated and saved as {noisy\_image\_path}")

def getNoisyBinaryImage(input\_path, output\_path):

    image = Image.open(input\_path)

    draw = ImageDraw.Draw(image)

    width, height = image.size

    pix = image.load()

    noise\_factor = 1000

    for i in range(width):

        for j in range(height):

            rand = random.randint(-noise\_factor, noise\_factor)

            red = pix[i, j][0] + rand

            green = pix[i, j][1] + rand

            blue = pix[i, j][2] + rand

            if red < 0: red = 0

            if green < 0: green = 0

            if blue < 0: blue = 0

            if red > 255: red = 255

            if green > 255: green = 255

            if blue > 255: blue = 255

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

    image.save(output\_path, "JPEG")

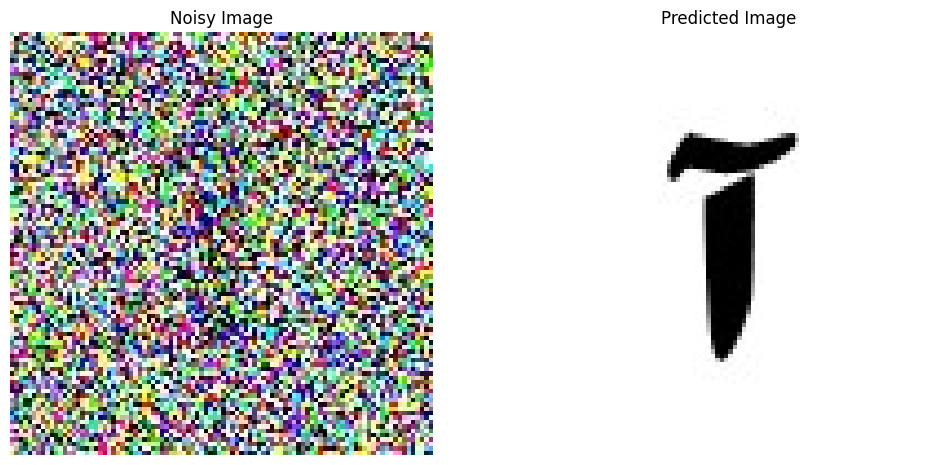
    del draw

generateNoisyImages()

* **محاسبه فاصله همینگ:** تابع hamming\_distance برای محاسبه فاصله همینگ بین دو تصویر باینری طراحی شده است. این تابع از عملیات مقایسه‌ای استفاده می‌کند تا تفاوت‌های بین دو تصویر را شمارش کند. در واقع، فاصله همینگ تعداد بیت‌های متفاوت میان دو رشته باینری را نشان می‌دهد. در اینجا تصاویر به صورت آرایه‌های numpy در آمده و سپس تفاوت‌های هر پیکسل با پیکسل‌های متناظر در تصویر دیگر محاسبه می‌شود.
* **شبکه عصبی همینگ (Hamming Neural Network):** تابع hamming\_neural\_network وظیفه پیش‌بینی تصویر اصلی از میان مجموعه‌ای از تصاویر اصلی را بر عهده دارد. برای این کار، فاصله همینگ بین تصویر نویزی و هر یک از تصاویر اصلی محاسبه می‌شود و تصویری که کمترین فاصله را با تصویر نویزی دارد، به عنوان تصویر پیش‌بینی‌شده انتخاب می‌شود.
* **نمودار تصاویر:** تابع plot\_images برای نمایش تصاویری که به پیش‌بینی تبدیل شده‌اند، طراحی شده است. این تابع دو تصویر را به طور همزمان در کنار یکدیگر به نمایش می‌گذارد: تصویر نویزی و تصویر پیش‌بینی‌شده.
  + **گام اول - بارگذاری تصاویر:** در ابتدا، مسیرهای تصاویر اصلی و نویزی تعریف شده‌اند. سپس با استفاده از کتابخانه PIL، این تصاویر بارگذاری شده و به آرایه‌های numpy تبدیل می‌شوند تا بتوانند در محاسبات فاصله همینگ مورد استفاده قرار گیرند.
  + **گام دوم - پیش‌بینی تصویر اصلی:** تصویر نویزی از مجموعه تصاویر نویزی انتخاب می‌شود و تابع hamming\_neural\_network برای پیش‌بینی تصویر اصلی مناسب از بین تصاویر موجود اجرا می‌شود. این تابع از روش فاصله همینگ برای مقایسه تصاویر استفاده می‌کند و تصویر با کمترین تفاوت (کمترین فاصله همینگ) را به عنوان تصویر پیش‌بینی‌شده انتخاب می‌کند.
  + **گام سوم - نمایش تصاویر:** در نهایت، تصویر نویزی و تصویر پیش‌بینی‌شده در کنار یکدیگر با استفاده از matplotlib نمایش داده می‌شوند. این نمودار به کاربر این امکان را می‌دهد که تصویر نویزی را مشاهده کرده و تطبیق آن با تصویر اصلی پیش‌بینی‌شده را مقایسه کند.
* import numpy as np
* import matplotlib.pyplot as plt
* from PIL import Image
* def hamming\_distance(image1, image2):
* """Calculate the Hamming distance between two binary images."""
* return np.sum(image1 != image2)
* def hamming\_neural\_network(noisy\_image, original\_images):
* """Predict the original image from a noisy image using Hamming distance."""
* min\_distance = float('inf')
* predicted\_image = None
* for original in original\_images:
* distance = hamming\_distance(noisy\_image.flatten(), original.flatten())  # Flatten for comparison
* if distance < min\_distance:
* min\_distance = distance
* predicted\_image = original
* return predicted\_image
* def plot\_images(noisy\_image, predicted\_image):
* """Plot the noisy and predicted images side by side."""
* plt.figure(figsize=(12, 6))
* plt.subplot(1, 2, 1)
* plt.title("Noisy Image")
* plt.imshow(noisy\_image)
* plt.axis('off')
* plt.subplot(1, 2, 2)
* plt.title("Predicted Image")
* plt.imshow(predicted\_image)
* plt.axis('off')
* plt.show()
* # Step 1: Set paths for original images and generated noisy images
* original\_image\_paths = [
* "/content/1.jpg",
* "/content/2.jpg",
* "/content/3.jpg",
* "/content/4.jpg",
* "/content/5.jpg"
* ]
* # Assuming noisy images have been saved as follows
* noisy\_image\_paths = [
* "/content/noisy1.jpg",
* "/content/noisy2.jpg",
* "/content/noisy3.jpg",
* "/content/noisy4.jpg",
* "/content/noisy5.jpg"
* ]
* # Load original images for prediction
* original\_images = [np.array(Image.open(path).convert('RGB')) for path in original\_image\_paths]
* # Load noisy images for testing
* noisy\_images = [np.array(Image.open(path).convert('RGB')) for path in noisy\_image\_paths]
* # Step 2: Use the first noisy image for prediction
* noisy\_image = noisy\_images[0]
* predicted\_image = hamming\_neural\_network(noisy\_image, original\_images)
* # Step 3: Plot the noisy image and its predicted original image
* plot\_images(noisy\_image, predicted\_image)

این کد به طور کلی یک روش ساده برای شبیه‌سازی پیش‌بینی تصاویر با استفاده از فاصله همینگ است. هرچند این روش معمولاً در مسائل پیچیده‌تر پردازش تصویر کاربرد ندارد، اما برای مقایسه شباهت‌های تصاویر باینری و تشخیص شباهت‌ها در تصاویر نویزی می‌تواند مفید باشد.

نتیجه:



کد زیر پیاده‌سازی یک **شبکه هافیلد (Hopfield Network)** است که برای شبیه‌سازی حافظه و بازسازی تصاویر نویزی استفاده می‌شود. شبکه هافیلد یک شبکه عصبی بازگشتی است که برای ذخیره‌سازی الگوهای باینری و بازیابی آن‌ها از روی تصاویر نویزی طراحی شده است. در اینجا، این شبکه برای بازسازی تصاویر نویزی از مجموعه‌ای از تصاویر اصلی آموزش داده می‌شود. کد شامل بخش‌هایی است که برای بارگذاری و پردازش تصاویر، آموزش شبکه هافیلد، و بازسازی تصاویر نویزی استفاده می‌شود.

این کد به‌طور مؤثر از شبکه هافیلد برای یادگیری الگوهای باینری و بازسازی آن‌ها از ورودی‌های نویزی استفاده می‌کند. از آنجا که شبکه هافیلد قادر به یادگیری الگوهای باینری است و با استفاده از ویژگی‌های تعمیم‌دهی‌اش می‌تواند نویز را از بین ببرد، این روش برای شبیه‌سازی حافظه‌های مشابه انسانی در سیستم‌های عصبی استفاده می‌شود. این شبکه می‌تواند برای شبیه‌سازی شناسایی الگو در بسیاری از مسائل کاربردی مانند تصحیح خطا و بازسازی تصاویر نویزی مورد استفاده قرار گیرد.

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

# Class implementation for the Hopfield Network

class HopfieldNetwork:

    def \_\_init\_\_(self, n\_units):

        self.weights = np.zeros((n\_units, n\_units))

    def train(self, patterns):

        """Train the Hopfield network with binary patterns."""

        for pattern in patterns:

            # Make sure the pattern is bipolar: -1 for 0, +1 for 1

            bipolar\_pattern = np.where(pattern == 0, -1, 1)

            self.weights += np.outer(bipolar\_pattern, bipolar\_pattern)

        # Set the diagonal to zero to avoid self-connections

        np.fill\_diagonal(self.weights, 0)

    def predict(self, input\_pattern, max\_iterations=5):

        """Predict the output based on an input pattern."""

        current\_pattern = np.where(input\_pattern == 0, -1, 1)  # Convert to bipolar

        for \_ in range(max\_iterations):

            for i in range(len(current\_pattern)):

                # Calculate the net input

                net\_input = np.dot(self.weights[i], current\_pattern)

                # Update the state based on the net input

                current\_pattern[i] = 1 if net\_input > 0 else 0

        return np.where(current\_pattern == -1, 0, 1)  # Convert back to binary

def load\_and\_binarize\_images(image\_paths, size=(20, 20)):

    """Load and binarize images to a specified size."""

    images = []

    for path in image\_paths:

        img = Image.open(path).convert('L').resize(size)  # Convert to grayscale and resize

        img\_array = np.array(img)

        # Binarize the image: 0 for Dark, 1 for Light

        binarized\_image = np.where(img\_array > 128, 1, 0)  # Threshold at 128

        images.append(binarized\_image.flatten())  # Flatten for the Hopfield network

    return images

def add\_noise(image, noise\_level=0.1):

    """Add noise to a binary image."""

    noisy\_image = image.copy()

    num\_flips = int(noise\_level \* noisy\_image.size)  # Total pixels to flip

    indices = np.random.choice(np.arange(noisy\_image.size), num\_flips, replace=False)

    noisy\_image.ravel()[indices] = 1 - noisy\_image.ravel()[indices]  # Flip 0 to 1 and 1 to 0

    return noisy\_image

def plot\_images(original\_image, noisy\_image, reconstructed\_image):

    """Plot the original, noisy, and reconstructed images side by side."""

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)

    plt.title("Original Image")

    plt.imshow(original\_image.reshape(20, 20), cmap='gray')

    plt.axis('off')

    plt.subplot(1, 3, 2)

    plt.title("Noisy Image")

    plt.imshow(noisy\_image.reshape(20, 20), cmap='gray')

    plt.axis('off')

    plt.subplot(1, 3, 3)

    plt.title("Reconstructed Image")

    plt.imshow(reconstructed\_image.reshape(20, 20), cmap='gray')

    plt.axis('off')

    plt.show()

# Set paths for original images (modify as needed)

original\_image\_paths = [

    "/content/1.jpg",

    "/content/2.jpg",

    "/content/3.jpg",

    "/content/4.jpg",

    "/content/5.jpg"

]

# Load and binarize original images

original\_patterns = load\_and\_binarize\_images(original\_image\_paths)

# Train the Hopfield network with original patterns

hopfield\_network = HopfieldNetwork(n\_units=len(original\_patterns[0]))

hopfield\_network.train(original\_patterns)

# Create a noisy version of one of the original images

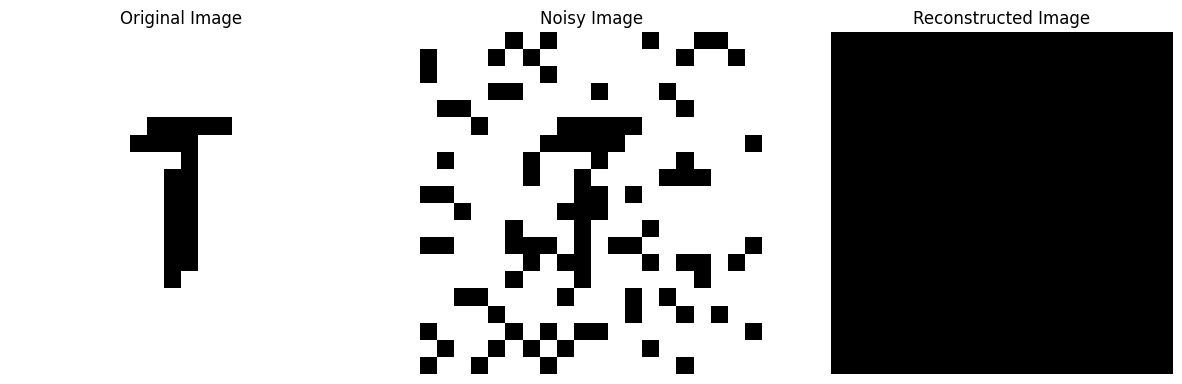
noisy\_image = add\_noise(original\_patterns[0], noise\_level=0.2)  # Change noise level as needed

# Predict/reconstruct the image from the noisy input

reconstructed\_image = hopfield\_network.predict(noisy\_image)

# Plot images

plot\_images(original\_patterns[0], noisy\_image, reconstructed\_image)



در آخر کد زیر، به‌طور کلی برای ایجاد تصاویری طراحی شده است که در آن‌ها برخی از نقاط به‌طور تصادفی حذف شده‌اند و سپس این تصاویر با تصاویر اصلی مقایسه می‌شوند. این کار می‌تواند برای شبیه‌سازی حذف داده‌ها در تصاویر یا آزمایش روش‌های ترمیم تصویر (Image Inpainting) مفید باشد. در اینجا جزئیات عملکرد کد را توضیح می‌دهم:

from PIL import Image, ImageDraw

import random

import matplotlib.pyplot as plt

def generateImagesWithMissingPoints():

    # List of image file paths

    image\_paths = [

        "/content/1.jpg",

        "/content/2.jpg",

        "/content/3.jpg",

        "/content/4.jpg",

        "/content/5.jpg"

    ]

    for i, image\_path in enumerate(image\_paths, start=1):

        output\_path = f"/content/missing\_points{i}.jpg"

        createImageWithMissingPoints(image\_path, output\_path)

        print(f"Image with missing points for {image\_path} generated and saved as {output\_path}")

    # Plot one of the images with missing points

    plotImageWithMissingPoints(image\_paths[0], f"/content/missing\_points1.jpg")

def createImageWithMissingPoints(input\_path, output\_path):

    """

    Create an image with random missing points and save it as a new file.

    Args:

        input\_path (str): The file path to the input image.

        output\_path (str): The file path to save the modified image.

    """

    # Open the input image

    image = Image.open(input\_path)

    # Create a drawing tool for manipulating the image

    draw = ImageDraw.Draw(image)

    # Determine the image's width and height in pixels

    width, height = image.size

    # Define the number of points to remove (as a percentage of total pixels)

    missing\_points\_count = int(0.99 \* width \* height)  # Remove 5% of pixels

    for \_ in range(missing\_points\_count):

        # Randomly select a pixel to remove

        x = random.randint(0, width - 1)

        y = random.randint(0, height - 1)

        # Set the pixel to white (255, 255, 255)

        draw.point((x, y), (255, 255, 255))

    # Save the modified image as a file

    image.save(output\_path, "JPEG")

    # Clean up the drawing tool

    del draw

def plotImageWithMissingPoints(original\_path, modified\_path):

    """

    Plot the original and modified images side by side for comparison.

    Args:

        original\_path (str): The file path to the original image.

        modified\_path (str): The file path to the modified image.

    """

    original\_image = Image.open(original\_path)

    modified\_image = Image.open(modified\_path)

    # Plot the images

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)

    plt.title("Original Image")

    plt.imshow(original\_image)

    plt.axis("off")

    plt.subplot(1, 2, 2)

    plt.title("Image with Missing Points")

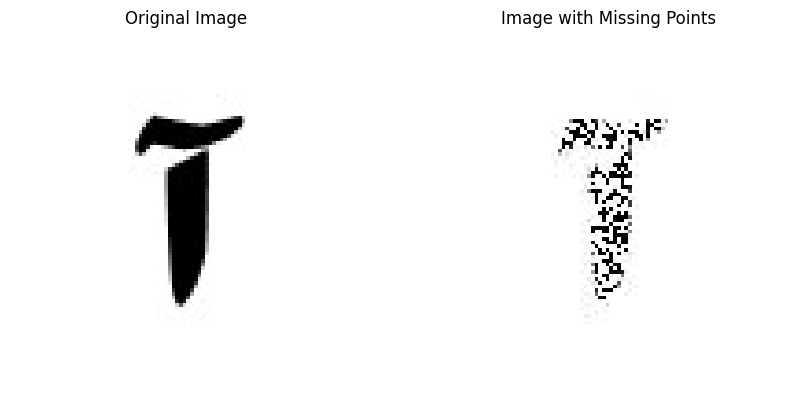
    plt.imshow(modified\_image)

    plt.axis("off")

    plt.show()

# Generate images with missing points and plot one of them

generateImagesWithMissingPoints()



4-

در خواندن داده و تقسیم بندی آن به داده آموزشی و آزمایشی جای بحث نیست. در ادامه لایه RBF را تعریف می کنم:

class RBFLayer(tf.keras.layers.Layer):

    def \_\_init\_\_(self, units, gamma=0.1):

        super(RBFLayer, self).\_\_init\_\_()

        self.units = units

        self.gamma = gamma

    def build(self, input\_shape):

        self.centers = self.add\_weight(name='centers',

                                       shape=(self.units, input\_shape[-1]),

                                       initializer='random\_uniform',

                                       trainable=True)

        self.betas = self.add\_weight(name='betas',

                                     shape=(self.units,),

                                     initializer=tf.keras.initializers.Constant(1.0),

                                     trainable=True)

    def call(self, inputs):

        C = tf.expand\_dims(self.centers, axis=0)

        X = tf.expand\_dims(inputs, axis=1)

        distances = tf.reduce\_sum((X - C) \*\* 2, axis=-1)

        return tf.exp(-self.gamma \* distances)

* **شرح:** این بخش یک لایه RBF (Radial Basis Function) تعریف می‌کند که به‌صورت سفارشی طراحی شده است.
* Units: تعداد نرون‌های RBF
* Gamma: پارامتری که میزان حساسیت RBF را تنظیم می‌کند.
* Centers: مراکز توابع RBF به‌صورت تصادفی مقداردهی می‌شوند.
* betas: وزن‌های RBF که کنترل پهنای توابع RBF را دارند.
* rbf\_model = Sequential([
* Input(shape=(X.shape[1],)),
* RBFLayer(units=10, gamma=0.1),
* Dense(1)
* ])

این یک لایه با 10 نورون است که تعداد ویژگی های آن، تمامی ویژگی های موجود در داده است.

rbf\_model.compile(optimizer=Adam(), loss=MeanSquaredError())

در سلول بالا از بهینه ساز Adam و تابع خطای MSE استفاده کردیم. سپس مدل RBF را به چند نحو آموزش دادیم:

|  |  |
| --- | --- |
| Loss | RBF |
| 0.4538 | Epochs = 50 , batch = 32 |
| 0.4131 | Epochs = 50 , batch = 64 |
| 0.4164 | Epochs = 100 , batch = 32 |
| 0.4073 | Epochs = 100 , batch = 64 |
| 0.3986 | Epochs = 200 , batch = 64 |
| **0.3910** | Epochs = 300 , batch = 64 |

simple\_model = Sequential([

    Input(shape=(X.shape[1],)),

    Dense(64, activation='relu'),

    Dense(32, activation='relu'),

    Dense(1)

])

simple\_model.compile(optimizer=Adam(), loss=MeanSquaredError())

مدل شامل سه لایه Dense است:

* لایه اول: 64 نرون با فعال‌سازی ReLU
* لایه دوم: 32 نرون با فعال‌سازی ReLU
* لایه سوم: یک نرون برای پیش‌بینی خروجی
* بهینه ساز و تابع خطا مانند قبل

|  |  |
| --- | --- |
| Loss | Dense |
| 0.2660 | Epochs = 50 , batch = 32 |
| 0.2720 | Epochs = 50 , batch = 64 |
| **0.2606** | Epochs = 100 , batch = 32 |
| 0.2644 | Epochs = 100 , batch = 64 |

مشاهده می شود که مدل اول که بر پایه RBF نوشته شده است پس از300 دور آموزش به خطای 0.3910 رسیده اما مدل دوم با تعداد دور آموزش یک سوم برابر مدل اول، توانسته است به خطای 0.2606 دست یابد. واضح است که مدل دوم که بر اساس Dense نوشته شده است بهتر عمل کرده است.