بسم الله الرحمن الرحیم

دانشجو: محمدرضا جنیدی جعفری ۹۹۲۵۲۵۳

درس مبانی سیستم های هوشمند

استاد: دکتر مهدی علیاری

مینی پروژه دوم

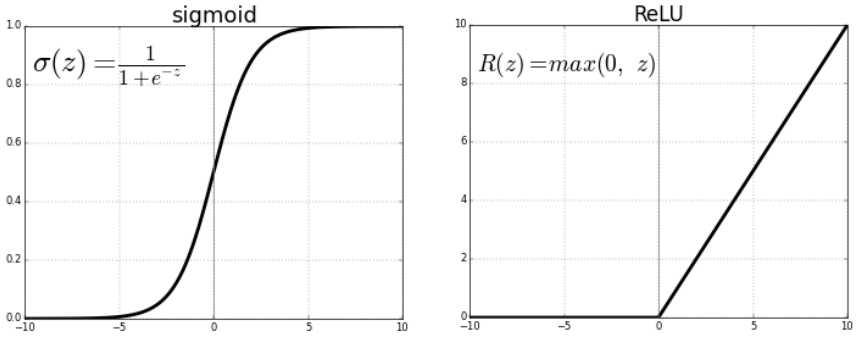
لینک مخزن گیت هاب - گوگل کولب سوال ۱ - گوگل کولب سوال ۲ – گوگل کولب سوال ۳ - گوگل کولب سوال ۴

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

۱-

۱-

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 ۰ 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.

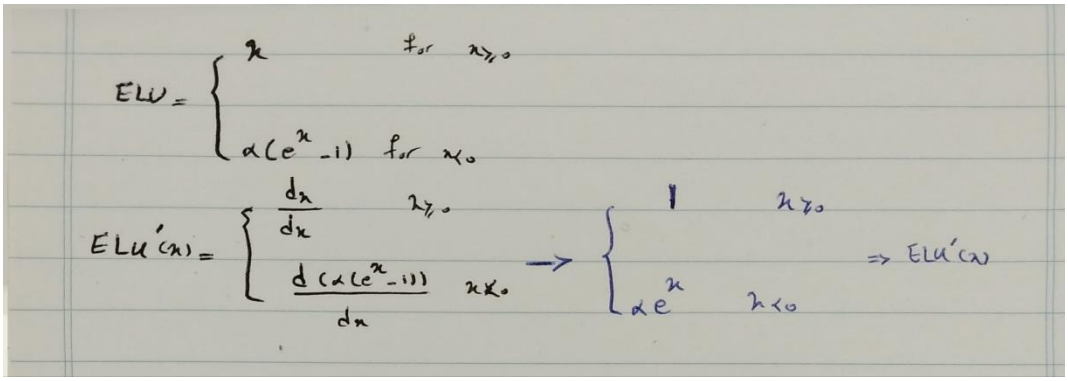
اگر این دو لایه پشت هم در دو لایه آخر یک مدل وجود داشته باشد:

- 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.

۲-

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.

۳-

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:

۱. **تعریف مثلث:**
   - مختصات سه رأس مثلث به صورت دستی تعریف شده‌اند.
۲. **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.
۳. **تولید نقاط تصادفی:**

- ۲۰۰۰ نقطه تصادفی در محدوده مشخص برای محورهای x و y تولید شده‌اند.

۴. **طبقه‌بندی نقاط:**

- نقاطی که داخل مثلث هستند، برچسب 1 می‌گیرند و نقاط خارج مثلث برچسب0.

۵. **نمایش نتایج:**

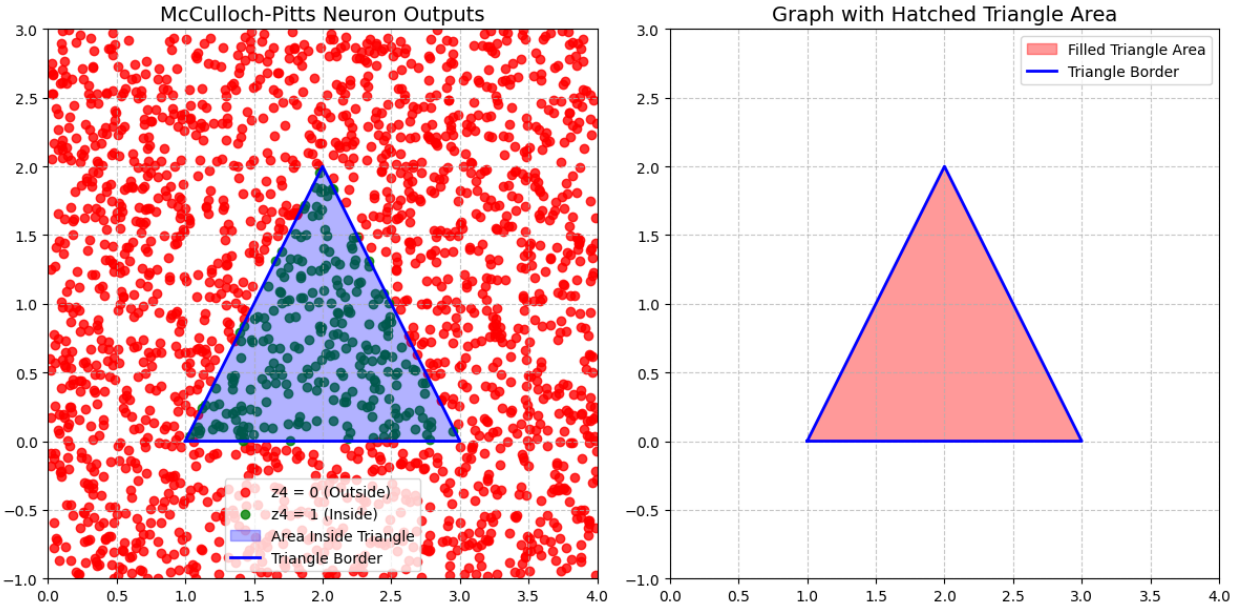
- نمودار اول: نقاط داخل مثلث (سبز) و خارج مثلث (قرمز) نمایش داده می‌شوند. مثلث با رنگ آبی پر شده است.
- نمودار دوم: مثلث به صورت جداگانه با رنگ قرمز و خطوط ضخیم‌تر نمایش داده می‌شود.

۶. **چاپ مختصات مثلث:**

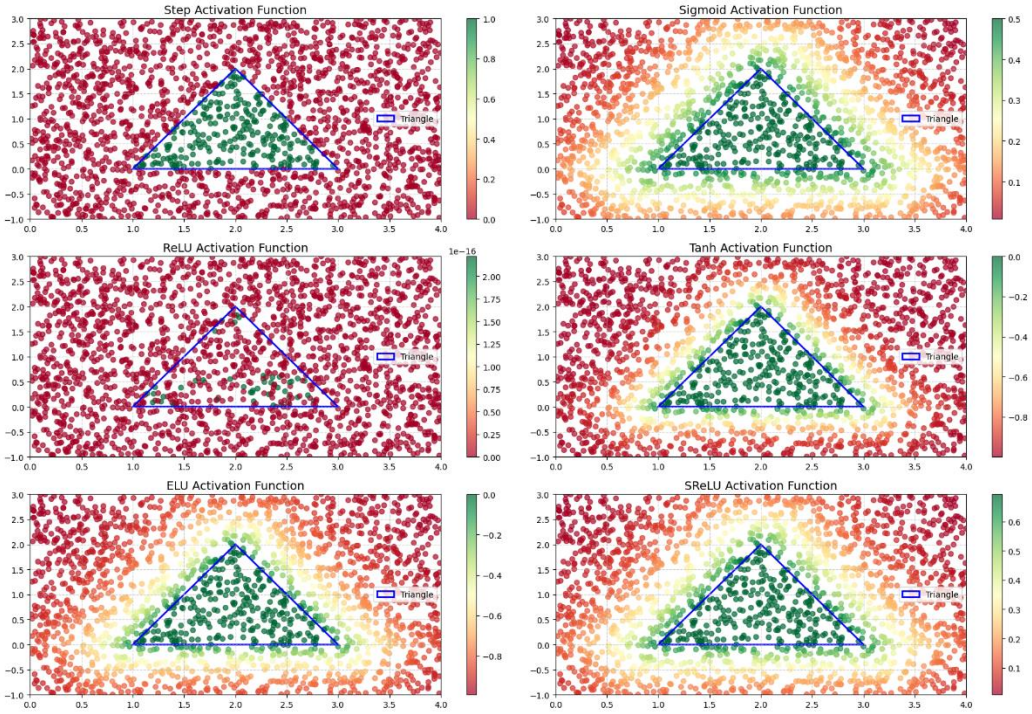
- مختصات رأس‌های مثلث برای مرجع چاپ می‌شوند.

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

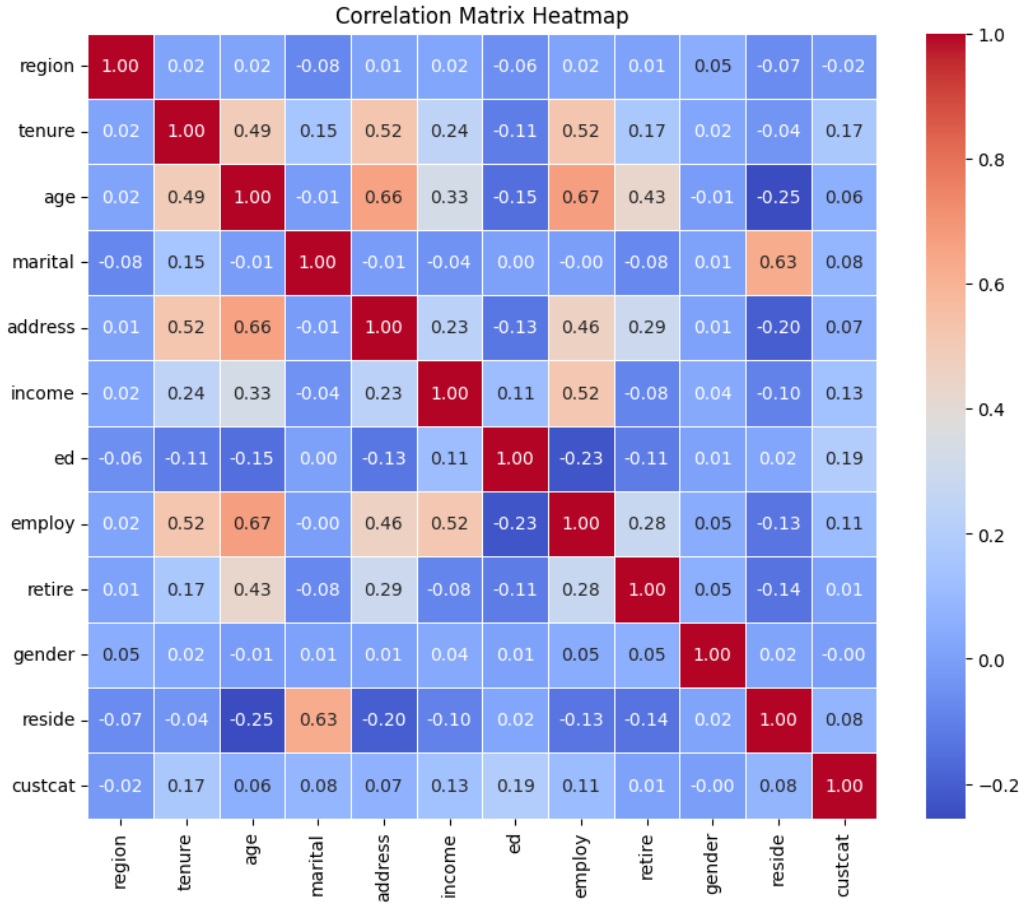
ReLU ،Tanh ،ELU:

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

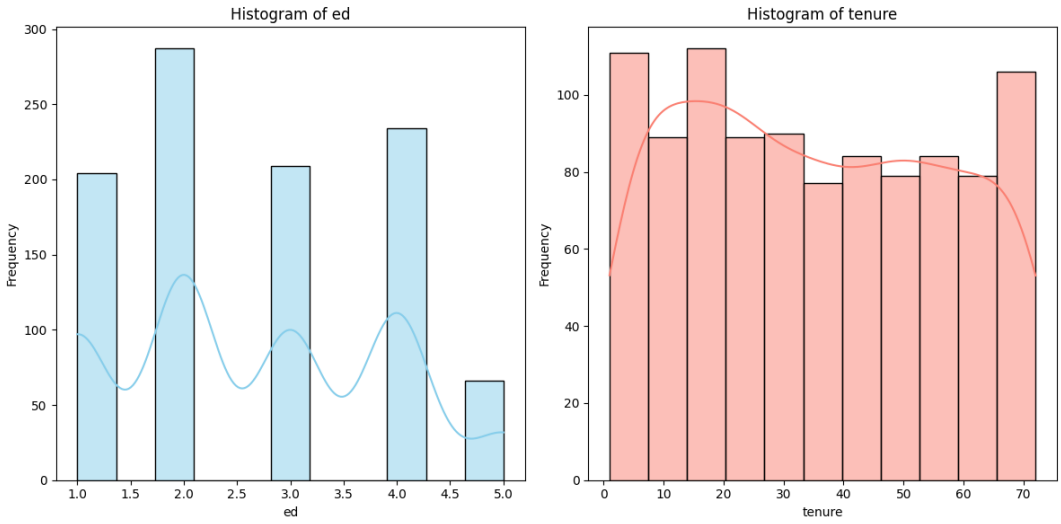
۲- داده ها را با استفاده از تابع پانداز فرامیخوانیم:

	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

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 ۱ تا ۴، به ۰ تا ۳ تبدیل می کنیم و داده ها را به ۶۵٪ برای آموزش، ۲۰٪ برای ارزیابی و ۱۵٪ 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:

	hidden_layers	neurons	batch_norm	train_loss	train_accuracy	val_loss	val_accuracy
0	1	64	False	1.271821	0.406593	1.273091	0.384977
1	1	64	True	1.387538	0.219780	1.389042	0.215962
2	2	64	False	1.406585	0.208791	1.391467	0.215962
3	2	64	True	1.387927	0.226060	1.385098	0.220657
4	1	128	False	1.393932	0.265306	1.394708	0.305164
5	1	128	True	1.391997	0.274725	1.382995	0.262911
6	2	128	False	1.381010	0.312402	1.377711	0.281690
7	2	128	True	1.368719	0.312402	1.379359	0.239437

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:

	hidden_layers	neurons	dropout_rate	train_loss	train_accuracy	val_loss	val_accuracy
0	1	64	0.0	1.270263	0.405024	1.282351	0.408451
1	1	64	0.3	1.276935	0.412873	1.277708	0.413146
2	1	64	0.5	1.273410	0.406593	1.284321	0.403756
3	2	128	0.0	1.235055	0.420722	1.244752	0.422535
4	2	128	0.3	1.237332	0.422292	1.242144	0.431925
5	2	128	0.5	1.279183	0.414443	1.291292	0.422535

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

	hidden_layers	neurons	dropout_rate	train_loss	train_accuracy	val_loss	val_accuracy
0	1	64	0.3	1.272440	0.416013	1.275747	0.417840
1	2	128	0.5	1.292215	0.397174	1.299869	0.394366

این دو مدل، فعلا بهترین نتایج را داشته اند.

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

	optimizer	hidden_layers	neurons	dropout_rate	train_loss	train_accuracy	val_loss	val_accuracy
0	Adam	1	64	0.3	1.175099	0.455259	1.210369	0.441315
1	Adam	2	128	0.5	1.159123	0.466248	1.201315	0.417840
2	RMSprop	1	64	0.3	1.181364	0.450549	1.206420	0.417840
3	RMSprop	2	128	0.5	1.161914	0.461538	1.200697	0.417840

و حالا، برای هر ۴ مدل به صورت تصادفی ۱۰ 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

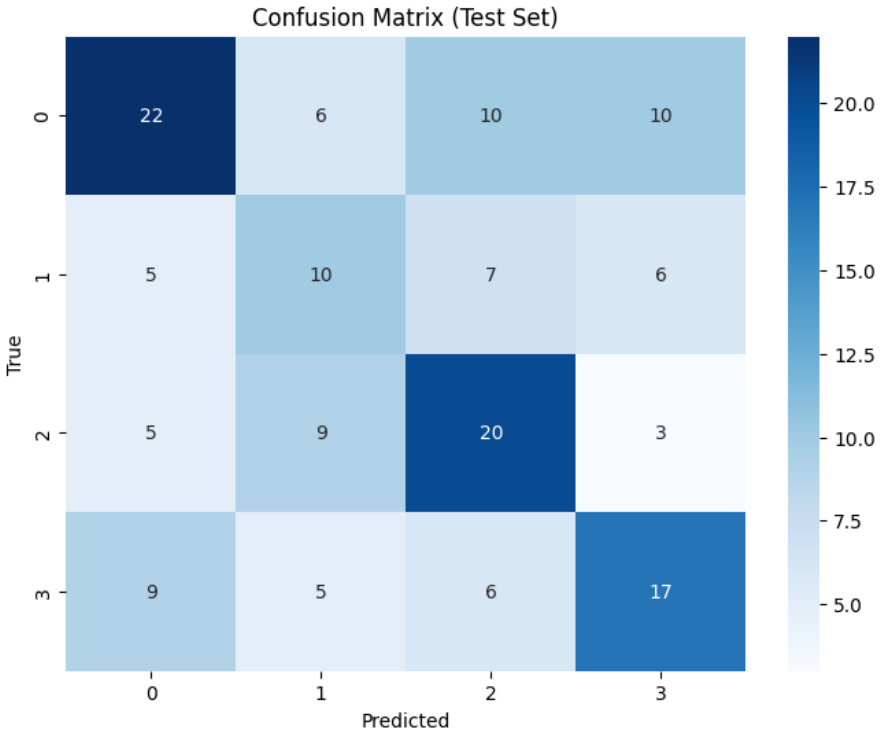
Train Loss: 1.1947, Train Accuracy: 0.4521

Validation Loss: 1.2162, Validation Accuracy: 0.4554

```
Ensemble Model Accuracy on Train Set: 0.4505

Classification Report for Ensemble Model (Train Set):
              precision    recall  f1-score   support

           0       0.47      0.57      0.51       161
           1       0.46      0.41      0.43       149
           2       0.45      0.40      0.43       183
           3       0.42      0.42      0.42       144

    accuracy                           0.45       637
   macro avg       0.45      0.45      0.45       637
weighted avg       0.45      0.45      0.45       637
```

```
Ensemble Model Accuracy on Test Set: 0.4400

Classification Report for Ensemble Model (Test Set):
              precision    recall  f1-score   support

           0       0.49      0.44      0.46        48
           1       0.31      0.32      0.32        28
           2       0.47      0.54      0.50        37
           3       0.46      0.43      0.44        37

    accuracy                           0.44       150
   macro avg       0.43      0.43      0.43       150
weighted avg       0.44      0.44      0.44       150
```

ماتریس هم بستگی به صورت زیر خواهد بود:


Confusion Matrix (Test Set)
True
Predicted
0
1
2
3
22
6
10
10
5
10
7
6
5
9
20
3
9
5
6
17
20.0
17.5
15.0
12.5
10.0
7.5
5.0

۳-

۱-

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 1تبدیل 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 ۰ تا ۲۵۵ 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:

۱- تبدیل به باینری:

```
2- from PIL import Image, ImageDraw
3-
4- def convertImageToBinary(path):
5-     image = Image.open(path)
6-     draw = ImageDraw.Draw(image)
7-     width, height = image.size
8-     pix = image.load()
9-     factor = 1000
10-          binary_representation = []
11-
12-          for i in range(width):
13-              for j in range(height):
14-                  red, green, blue = pix[i, j]
15-                  total_intensity = red + green + blue
16-
17-                  if total_intensity > (((255 + factor) // 2) * 3):
18-                      red, green, blue = 255, 255, 255
19-                      binary_representation.append(-1)
20-                  else:
21-                      red, green, blue = 0, 0, 0
22-                      binary_representation.append(1)
23-
24-                  draw.point((i, j), (red, green, blue))
25-
26-          del draw
27-          return binary_representation
```

۲- 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_distanceبرای 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 numpyدر آمده و سپس تفاوت‌های هر پیکسل با پیکسل‌های متناظر در تصویر دیگر محاسبه می‌شود.
- **شبکه عصبی همینگ :(Hamming Neural Network)** تابع hamming_neural_networkوظیفه 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_imagesبرای 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 numpyتبدیل 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_networkبرای 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 matplotlibنمایش 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.

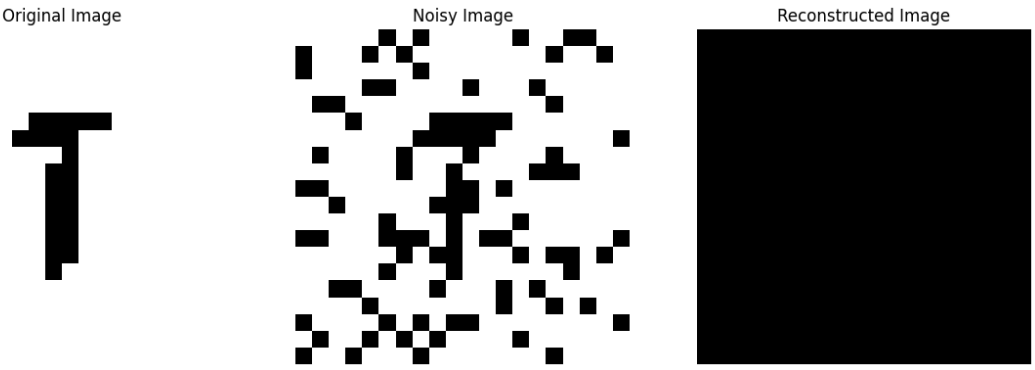
این کد به‌طور مؤثر از شبکه هافیلد برای یادگیری الگوهای باینری و بازسازی آن‌ها از ورودی‌های نویزی استفاده می‌کند. از آنجا که شبکه هافیلد قادر به یادگیری الگوهای باینری است و با استفاده از ویژگی‌های تعمیم‌دهی‌اش می‌تواند نویز را از بین ببرد، این روش برای شبیه‌سازی حافظه‌های مشابه انسانی در سیستم‌های عصبی 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()
```

Original Image

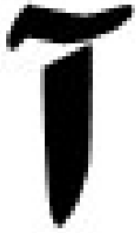


Image with Missing Points

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_unifo
rm',
                                       trainable=True)
        self.betas = self.add_weight(name='betas',
                                     shape=(self.units,),
                                     initializer=tf.keras.initia
lizers.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 ۱۰ 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:

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

```
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: ۶۴ نرون با فعال‌سازی ReLU
- لایه دوم: ۳۲ نرون با فعال‌سازی ReLU
- لایه سوم: یک نرون برای پیش‌بینی خروجی
- بهینه ساز و تابع خطا مانند قبل

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

مشاهده می شود که مدل اول که بر پایه RBF نوشته شده است پس از ۳۰۰ دور آموزش به خطای ۰.۳۹۱۰ رسیده اما مدل دوم با تعداد دور آموزش یک سوم برابر مدل اول، توانسته است به خطای ۰.۲۶۰۶ دست یابد. واضح است که مدل دوم که بر اساس Dense نوشته شده است بهتر عمل کرده است.