گزارش مینی پروژه ۲

هوش مصنوعی

دکتر علیاری

سینا حسن‌پور شماره دانشجویی: ۴۰۲۱۶۷۲۳

۳۱ اردیبهشت ۱۴۰۴

فهرست مطالب

مخزن گیت‌هاب

برای مشاهده و دانلود کدهای مربوط به این پروژه، می‌توانید به مخزن گیت‌هاب مراجعه کنید:

مخزن گیت‌هاب پروژه‌ها

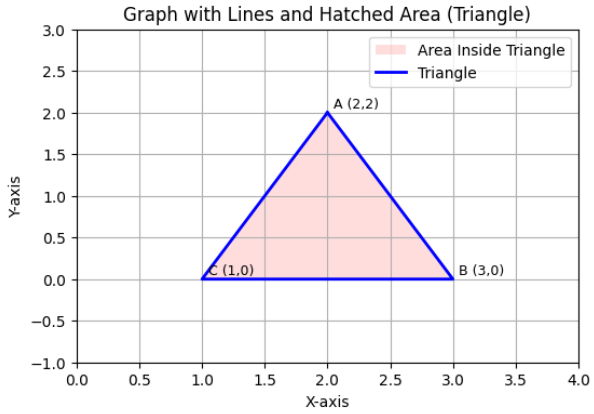
پیوست: لینک‌های اجرای آنلاین پروژه‌ها در Google Colab

برای مشاهده و اجرای مستقیم هر یک از پرسش‌ها در محیط Google Colab، می‌توانید از لینک‌های زیر استفاده کنید:

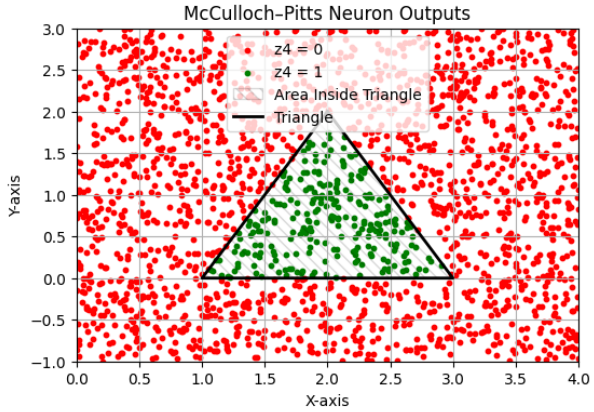
- **پرسش ۱:** مشاهده در Google Colab
- **پرسش ۲:** مشاهده در Google Colab
- **پرسش ۳:** مشاهده در Google Colab
- **پرسش ۴:** مشاهده در Google Colab

۱ پرسش یک

هدف این تمرین طراحی شبکه‌ای ساده مبتنی بر نورون McCulloch–Pitts است که بتواند نقاط داخل ناحیهٔ مثلثی مشخص‌شده را از سایر نواحی صفحه جدا کند. این شبکه باید بتواند برای نقاط داخل مثلث خروجی ۱ و برای سایر نقاط خروجی ۰ تولید کند. شکل‌های زیر، ناحیهٔ هدف و خروجی شبکه را نشان می‌دهند.



شکل ۱: (الف) ناحیه هاشورزدهٔ هدف – مثلث ABC



شکل ۲: (ب) خروجی شبکه مک‌کالاک–پیتس (نقاط سبز داخل مثلث)

مراحل و روش حل

۱. تعریف مسئله

در این تمرین، مسئله‌ی اصلی تشخیص ناحیه‌ای هندسی (یک مثلث) در فضای دوبعدی است. رأس‌های مثلث به صورت دقیق با مختصات مشخص $A = (2, 2), B = (3, 0), C = (1, 0)$ تعریف شده‌اند و هدف آن است که با طراحی یک شبکه‌ی ساده، یاد بگیریم که آیا یک نقطه داده‌شده درون این ناحیه مثلثی قرار دارد یا خیر. این نوع مسئله به‌طور خاص برای بررسی توانایی تفکیک ناحیه‌ای شبکه‌های عصبی ساده و نورون‌های منطقی مثل مک‌کالاک–پیتس بسیار مناسب است. بازه‌ای که نقاط در آن بررسی می‌شوند، به‌صورت $[0, 4] \times [-1, 3]$ در نظر گرفته شده تا هم داخل و هم خارج مثلث نمونه‌برداری شود.

۲. تولید داده

در این بخش، ابتدا مختصات دقیق رأس‌های مثلث وارد می‌شود. سپس با استفاده از تابع uniform از کتابخانه numpy.random، تعداد ۲۰۰۰ نقطه دوبعدی تصادفی در محدوده‌ی مشخص‌شده تولید می‌کنیم. این نقاط برای آموزش یا ارزیابی مدل‌ها استفاده می‌شوند. برای تشخیص اینکه آیا هر نقطه داخل مثلث قرار دارد یا نه، از روش هندسی استفاده شده که مبتنی بر ضرب‌های برداری بین بردارهای تشکیل‌دهنده اضلاع مثلث و نقطه‌ی مورد نظر است. این روش تضمین می‌کند که اگر علامت تمام ضرب‌های برداری (در مقایسه با جهت اضلاع مثلث) یکسان باشد، آن نقطه داخل مثلث است.

تولید داده‌های تصادفی و تعریف رأس‌های مثلث

```
A = np.array([2, 2])
B = np.array([3, 0])
C = np.array([1, 0])

N = 2000
rng = np.random.default_rng(42)
X = rng.uniform([0, -1], [4, 3], size=(N, 2))
```

۳. طراحی شبکهٔ مک‌کالاک–پیتس

برای هر لبهٔ مثلث، یک نورون تعریف شده است که ناحیه نیم‌صفحه‌ای مناسب را تشخیص می‌دهد. زمانی که هر سه نورون فعال باشند (یعنی نقطه مورد نظر در هر سه نیم‌صفحه قرار بگیرد)، خروجی نهایی ۱ خواهد بود.

تعریف شبکه مک‌کالاک–پیتس با سه نورون مرزی

```
def step(u):
    return (u >= 0).astype(int)

W = np.array([[-2, -1, 6],
              [ 0,  1, 0],
              [ 2, -1, -2]])

def mcp_network(X):
    X_ = np.c_[X, np.ones(len(X))]
    h  = step(X_ @ W.T)
    return (h.sum(axis=1) == 3).astype(int)

Y_pred_step = mcp_network(X)
```

در این مرحله، شبکه‌ای از نوع کلاسیک مک‌کالاک–پیتس پیاده‌سازی شده است. برای این کار، ابتدا سه معادله‌ی خطی مربوط به لبه‌های مثلث استخراج شده و به صورت وزن‌هایی برای نورون‌ها تعریف شده‌اند. هر نورون یک ناحیه‌ی نیم‌صفحه‌ای را شناسایی می‌کند و خروجی آن با استفاده از تابع پله‌ای (Heaviside Step Function) فعال می‌شود. سپس خروجی سه نورون بررسی می‌شود؛ تنها در صورتی که هر سه خروجی برابر ۱ باشند (یعنی نقطه داخل هر سه نیم‌صفحه باشد)، شبکه خروجی نهایی را برابر ۱ قرار می‌دهد. به این ترتیب، شبکه توانسته است ناحیه‌ی داخلی مثلث را به‌صورت کامل پوشش دهد.

۴. مقایسه با شبکهٔ عصبی سیگموییدی

برای بررسی قابلیت تعمیم‌پذیری مدل‌ها، یک شبکهٔ عصبی ساده با لایه پنهان ۴ نرونی و تابع فعال‌سازی سیگموید طراحی و آموزش داده شد.

مدل شبکه عصبی ساده با لایه پنهان

```
model = nn.Sequential(
    nn.Linear(2, 4),
    nn.Sigmoid(),
    nn.Linear(4, 1),
    nn.Sigmoid()
)
loss_fn = nn.BCELoss()
opt     = optim.SGD(model.parameters(), lr=0.1)

for epoch in range(400):
    opt.zero_grad()
    loss = loss_fn(model(X_t), Y_t)
    loss.backward(); opt.step()

with torch.no_grad():
    Y_pred_sig = (model(X_t) > 0.5).numpy().astype(int).flatten()
```

برای بررسی اینکه آیا یک مدل یادگیری می‌تواند همین مسئله را بدون اطلاعات هندسی حل کند، یک شبکه‌ی عصبی ساده طراحی شده است. این شبکه شامل یک لایه‌ی ورودی با ۲ نورون (مختصات x , y)، یک لایه‌ی پنهان با ۴ نورون، و یک نورون خروجی است. برای هر لایه، از تابع فعال‌ساز Sigmoid استفاده شده است تا مدل توانایی یادگیری تصمیم‌گیری‌های نرم soft decision را داشته باشد. از الگوریتم بهینه‌سازی SGD برای به‌روزرسانی وزن‌ها استفاده شده و تابع خطای BCELoss برای اندازه‌گیری دقت پیش‌بینی‌ها به‌کار رفته است. پس از ۴۰۰ مرحله آموزش، مدل قادر است به‌صورت تقریبی ناحیه مثلثی را از سایر نقاط تمایز دهد.

۵. دقت نهایی مدل‌ها

دقت هر مدل بر اساس درصد پیش‌بینی صحیح نسبت به برچسب‌های واقعی محاسبه شد:

- دقت شبکهٔ پله‌ای (دست‌ساز): **99.85%**
- دقت شبکهٔ سیگموییدی (آموزش‌دیده):**99.80%**

در این بخش عملکرد دو مدل مورد بررسی قرار گرفته است. مدل مک‌کالاک–پیتس که با منطق هندسی طراحی شده، دقت 99.85% را نشان می‌دهد که بسیار بالا و دقیق است. مدل عصبی نیز با وجود ساختار ساده و آموزش محدود، دقت 99.80% را کسب کرده که نشان‌دهنده‌ی قدرت یادگیری مدل در تشخیص الگوهای فضایی حتی بدون دانستن معادلات هندسی است. این نتیجه نشان می‌دهد که در مسائل ساده‌ی ناحیه‌بندی، حتی مدل‌های کوچک یادگیری نیز می‌توانند با تنظیم مناسب، عملکرد بسیار خوبی از خود نشان دهند.

نتیجه‌گیری

در این تمرین، توانایی مدل‌های ساده در تشخیص نواحی هندسی به کمک منطق و یادگیری بررسی شد. مدل مک‌کالاک–پیتس که بر پایه منطق خطی و هندسه تحلیلی طراحی شده بود، با استفاده از سه نورون و تابع پله‌ای توانست ناحیهٔ داخل مثلث را با دقت بسیار بالا (99.85%) تشخیص دهد. این مدل اگرچه ساختاری ساده و بدون قابلیت یادگیری دارد، اما در مسائل دارای مرزهای خطی ثابت، بسیار مؤثر و کاراست.

در مقابل، یک شبکهٔ عصبی با ساختار ساده شامل یک لایه پنهان و توابع فعال‌ساز سیگموید نیز پیاده‌سازی و آموزش داده شد. با وجود نداشتن اطلاعات هندسی، این مدل یاد گرفت که مرز ناحیهٔ مثلثی را به‌صورت تقریبی تشخیص دهد و دقت 99.80% را کسب کرد. این موفقیت نشان‌دهندهٔ قدرت یادگیری مدل‌های عصبی حتی در ساختارهای ساده و با تعداد کم نورون است.

نتایج حاصل از این پروژه نشان می‌دهد که در صورتی که ساختار مسئله به‌صورت تحلیلی مشخص باشد، می‌توان با طراحی منطقی ساده (مانند نورون‌های مک‌کالاک–پیتس) به پاسخ دقیقی رسید؛ اما اگر اطلاعات تحلیلی در دسترس نباشد یا مرزها پیچیده‌تر باشند، مدل‌های یادگیری ماشین گزینه‌ای قابل اعتماد و منعطف خواهند بود.

در نهایت، ترکیب دیدگاه هندسی با روش‌های یادگیری می‌تواند در طراحی سیستم‌های هوشمند دقیق‌تر و کارآمدتر نقش مؤثری ایفا کند. پیشنهاد می‌شود در ادامه، مدل‌ها بر روی نواحی غیرخطی و داده‌های نویزی نیز آزمایش شوند تا میزان تعمیم‌پذیری آن‌ها در شرایط پیچیده‌تر بررسی گردد.

۲ پرسش دو

۱.۱.۲

در این تمرین، از داده‌های ذخیره‌شده در فایل weather_prediction_dataset.csv استفاده شده است. این فایل شامل اطلاعات آب‌وهوایی چندین شهر در کشورهای مختلف است. با توجه به هدف پروژه که مشابه‌سازی مقاله A real-time collaborative machine learning based weather forecasting system with multiple predictor locations می‌باشد، تنها داده‌های مربوط به کشور فرانسه استخراج شده و سایر شهرها حذف شدند.

۲.۱.۲ استخراج داده‌های فرانسه و حذف سایر کشورها

در ادامه، ابتدا ستون‌هایی که نام آن‌ها با یکی از شهرهای فرانسه مانند TOURS، LYON، PARIS یا NANTES آغاز می‌شود، نگه داشته شدند. همچنین ستون DATE برای ثبت زمان نمونه‌ها نگه داشته شد تا در مرحله پیش‌پردازش مورد استفاده قرار گیرد.

کد فیلتر کردن داده‌های فرانسه در Colab

```python
import pandas as pd

# Load the dataset
df = pd.read_csv('weather_prediction_dataset.csv')

# Display all column names (for analysis)
print(df.columns.tolist())

# Select French cities
french_prefixes = ['TOURS_', 'LYON_', 'PARIS_', 'NANTES_']
french_cols = [col for col in df.columns if any(col.startswith(prefix) for
    prefix in french_prefixes)]

# Keep the DATE column too
french_cols.append('DATE')

# Filter the dataset
df_france = df[french_cols].copy()

# Convert DATE to datetime
df_france['DATE'] = pd.to_datetime(df_france['DATE'])

# Preview the filtered dataset
df_france.head()
```

در این مرحله، داده‌های مربوط به سایر کشورها حذف شده‌اند و تنها مقادیر مربوط به چهار شهر از فرانسه نگه‌داری می‌شوند. این انتخاب با هدف شبیه‌سازی معماری collaborative learning انجام گرفته که نیازمند داده از چند ناحیه جغرافیایی مجاور است.

۳.۱.۲ بازهٔ زمانی، نرمال‌سازی داده و ساخت پنجره‌های زمانی

داده‌های این پروژه مربوط به بازهٔ زمانی May 2021 هستند. برای آماده‌سازی داده‌ها جهت یادگیری مدل، ابتدا ستون تاریخ (DATE) جدا شده و سپس تمامی ویژگی‌های عددی با استفاده از روش MinMaxScaler نرمال‌سازی شده‌اند تا مقادیر در بازهٔ صفر تا یک قرار گیرند.

در مرحلهٔ بعد، طبق مقاله، از ساختار Sliding Window با طول پنجرهٔ ۵ روز استفاده شده است؛ به این صورت که ۵ روز متوالی به عنوان ورودی داده شده و مقدار پارامتر هدف در روز ششم به عنوان خروجی پیش‌بینی می‌شود.

کد مربوط به نرمال‌سازی و Window Sliding

```python
# Check time range of dataset
print("Date range:", df_france['DATE'].min(), "to", df_france['DATE'].max())

# Drop DATE column for normalization
df_features = df_france.drop(columns=['DATE'])

# Normalize features using MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df_features), columns=df_features.columns)

# Re-attach DATE column
df_scaled['DATE'] = df_france['DATE'].values

# Create sliding windows
import numpy as np

def create_sliding_windows(data, input_days=5, target_col='TOURS_temp_mean'):
    X, y = [], []
    for i in range(len(data) - input_days):
        input_window = data.iloc[i:i+input_days].drop(columns=['DATE']).values
        target_value = data.iloc[i+input_days][target_col]
        X.append(input_window)
        y.append(target_value)
    return np.array(X), np.array(y)

X, y = create_sliding_windows(df_scaled, input_days=5)

# Show final shapes
print("X shape:", X.shape)
print("y shape:", y.shape)
```

در این مرحله، مجموعه داده به شکل X: $(\mathrm{n},\ 5,\ \mathrm{m})$ و y: $(\mathrm{n},)$ آماده‌سازی شده که در آن n تعداد نمونه‌ها و m تعداد ویژگی‌ها برای هر روز می‌باشد. این داده‌ها در مرحله بعدی به مدل‌های یادگیری ماشین داده خواهند شد.

۴.۱.۲ ساخت پنجره‌های آموزش و آزمون

برای جداسازی مجموعه داده به دو بخش آموزش و آزمون، طبق الگوی مقاله، داده‌های مربوط به چند روز پایانی به عنوان دادهٔ آزمون انتخاب شده‌اند. در این پیاده‌سازی، ۴ روز آخر به‌همراه پنجرهٔ ورودی ۵ روزه برای پیش‌بینی آن، به‌عنوان دادهٔ آزمون در نظر گرفته شده‌اند. سایر داده‌ها به عنوان آموزش مورد استفاده قرار گرفته‌اند.

در این مرحله، sliding window به‌صورت هم‌پوشان اجرا شده است تا با استفاده از ورودی ۵ روزه، مقدار هدف در روز ششم پیش‌بینی شود. شکل نهایی داده‌ها به صورت $n \times 5 \times m$ برای ورودی و $n \times 1$ برای خروجی است.

کد جداسازی داده آموزش و آزمون و اعمال sliding window

```
# Split last 4 days + 5-day input window for testing
test_days = 4
window_size = 5
test_samples = test_days + window_size

df_test = df_scaled.tail(test_samples).reset_index(drop=True)
df_train = df_scaled.iloc[:-test_samples].reset_index(drop=True)

# Apply sliding window on both sets
X_train, y_train = create_sliding_windows(df_train, input_days=window_size)
X_test, y_test = create_sliding_windows(df_test, input_days=window_size)

# Final shapes
print("Train shape:", X_train.shape)
print("Test shape:", X_test.shape)
```

با اعمال این تقسیم‌بندی، داده‌ها برای آموزش شبکه عصبی و تحلیل در بخش‌های بعدی آماده شده‌اند.
ساختار داده‌ها به صورت ریاضی به‌صورت زیر تعریف می‌شود:

$$X_{\text{train}} \in \mathbb{R}^{n \times 5 \times m}, \quad y_{\text{train}} \in \mathbb{R}^{n}$$

۲.۲ آموزش مدل

۱.۲.۲ مفهوم Collaborative Machine Learning

در یادگیری ماشین کلاسیک، برای پیش‌بینی مقدار متغیری مانند دما، تنها از داده‌های تاریخی همان ناحیه استفاده می‌شود. اما در یادگیری مشارکتی یا Collaborative Learning، مدل از داده‌های چند ناحیهٔ مختلف (به‌صورت هم‌زمان) برای آموزش استفاده می‌کند.

در این پروژه، با استفاده از داده‌های آب‌وهوایی شهرهای فرانسه مانند TOURS, PARIS, LYON, NANTES، سعی شده است دمای یک شهر خاص (مثلاً TOURS) را نه تنها بر اساس داده‌های قبلی خود شهر، بلکه با کمک داده‌های شهرهای دیگر نیز پیش‌بینی کنیم.

این رویکرد باعث افزایش دقت مدل و کاهش خطا شده است، چراکه الگوهای همبسته در بین نواحی مختلف در مدل لحاظ می‌شوند. مقالهٔ مرجع نیز نشان می‌دهد که در حالت مشارکتی، میزان خطای میانگین مطلق درصدی (MAPE) تا ۵٪ کاهش یافته است.

۳.۲

در این مرحله، از مدل رگرسیون خطی چندمتغیره برای پیش‌بینی دمای آیندهٔ شهر TOURS استفاده شده است. داده‌های در دسترس فقط مربوط به همین شهر هستند، بنابراین مدل از داده‌های همین شهر (غیر مشارکتی) بهره می‌برد.

- دمای میانگین، حداقل و حداکثر TOURS
- رطوبت، فشار، و سرعت باد TOURS

برای هر نمونه، اطلاعات ۵ روز متوالی از ویژگی‌های بالا به‌عنوان ورودی به مدل داده شده و دمای روز ششم به‌عنوان خروجی پیش‌بینی شده است.

کد Google Colab برای MLR TOURS

```
selected_columns = [
    'TOURS_temp_mean', 'TOURS_temp_min', 'TOURS_temp_max',
    'TOURS_humidity', 'TOURS_pressure', 'TOURS_wind_speed'
]
...
X, y = create_xy(df_scaled)
...
model = LinearRegression()
model.fit(X_train, y_train)
```

در نهایت با استفاده از معیارهای MAE و MSE عملکرد مدل روی داده آزمون ارزیابی شد.

طراحی شبکه عصبی از صفر

در این بخش، یک شبکه عصبی ساده مطابق با ساختار شکل طراحی و آموزش داده شده است. این شبکه دارای یک لایهٔ پنهان و یک لایهٔ خروجی می‌باشد. از داده‌های شهر TOURS برای آموزش مدل استفاده شده است و هدف، پیش‌بینی دمای آیندهٔ این شهر بر اساس مقادیر قبلی متغیرهای جوی است.

معماری شبکه عصبی:

- ورودی: داده‌های ۵ روز گذشته شامل میانگین، حداقل و حداکثر دما، رطوبت، فشار و سرعت باد $\Rightarrow$ Input shape $= (30,)$
- لایهٔ پنهان: ۱۶ نورون با تابع فعال‌ساز ReLU
- لایهٔ خروجی: ۱ نورون با تابع فعال‌ساز Linear

پارامترهای آموزش:

- تابع خطا: Mean Squared Error (MSE)
- الگوریتم بهینه‌سازی: Stochastic Gradient Descent (SGD)
- تعداد epochs: ۲۰۰
- نرخ‌های یادگیری مختلف: 10^{-3}، 10^{-5}، 10^{-8}

برای هر نرخ یادگیری، مدل جداگانه‌ای آموزش داده شد. در طول آموزش، تغییرات تابع خطا برای مجموعه‌های آموزش و اعتبارسنجی ثبت شده و در نمودار مقایسه‌ای نمایش داده شده‌اند.

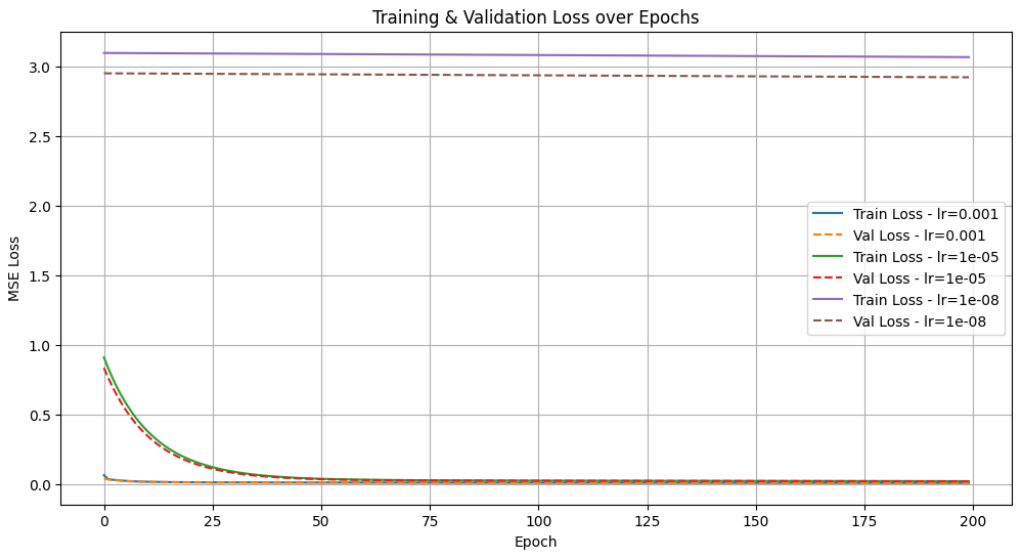
نتایج مدل‌ها:

MSE Test	MSE Train	Rate Learning
0.006868	0.007184	10^{-3}
0.019215	0.019522	10^{-5}
2.921629	3.066482	10^{-8}

جدول ۱: مقدار خطای MSE برای هر نرخ یادگیری پس از آموزش مدل

تحلیل نتایج:
همان‌طور که مشاهده می‌شود، شبکه با نرخ یادگیری 10^{-3} بهترین عملکرد را از لحاظ کمترین خطای آموزش و آزمون داشته است. در نرخ‌های پایین‌تر، سرعت همگرایی مدل به شدت کاهش یافته و در نرخ 10^{-8}، مدل تقریباً نتوانسته بهینه‌سازی مؤثری انجام دهد و در وضعیت اولیه باقی مانده است.

نمودار تغییرات تابع خطا:
در شکل ، نمودار تغییرات MSE برای مجموعه‌های آموزش و آزمون در هر سه حالت نرخ یادگیری نمایش داده شده است.



شکل ۳: مقایسه تغییرات خطای آموزش و آزمون برای نرخ‌های یادگیری مختلف

نتایج به خوبی نشان می‌دهند که انتخاب مناسب نرخ یادگیری نقش کلیدی در عملکرد نهایی شبکه دارد. نرخ خیلی بالا باعث نوسان و عدم همگرایی، و نرخ خیلی پایین باعث کندی شدید یادگیری یا گیر افتادن در مقدارهای اولیه می‌شود.

در ادامه، مدل‌های دیگر مانند MLP, CNN, KNN نیز برای همین مسئله آموزش داده خواهند شد و مقایسه‌ای جامع صورت خواهد گرفت.

۴.۲ تحلیل و ارزیابی مدل عصبی طراحی‌شده

با توجه به نتایج به‌دست‌آمده در بخش قبل، می‌توان عملکرد شبکه عصبی را به شکل زیر تحلیل کرد:

- شبکه با نرخ یادگیری 10^{-3} سریع‌ترین و دقیق‌ترین همگرایی را داشته و بهترین عملکرد نهایی را از خود نشان داده است.
- در نرخ 10^{-5}، مدل نیز به همگرایی رسیده اما با دقت پایین‌تر و سرعت کمتر.
- نرخ بسیار پایین 10^{-8} مانع از یادگیری مؤثر شده و مدل تقریباً در مقدار اولیه باقی مانده است.

نتیجه‌گیری کلی این است که انتخاب مناسب نرخ یادگیری نقش حیاتی در عملکرد شبکه دارد. نرخ بیش از حد بزرگ باعث نوسانات شدید و عدم همگرایی می‌شود و نرخ بیش از حد کوچک نیز فرآیند یادگیری را مختل می‌کند.

در ادامه، مدل‌های پیشرفته‌تری مانند MLP, CNN, KNN نیز بررسی و با این شبکه مقایسه خواهند شد.

۵.۲ مدل جدید (شبکه عصبی عمیق‌تر)

در این بخش، یک شبکه عصبی با ساختاری عمیق‌تر طراحی و آموزش داده شده است. هدف از این مرحله، بررسی تاثیر عمق مدل (افزایش تعداد لایه‌های پنهان) بر دقت پیش‌بینی و رفتار همگرایی مدل می‌باشد.

معماری شبکه عمیق:

- ورودی: داده‌های ۵ روز گذشته از ویژگی‌های جوی $\Rightarrow \text{Input shape} = (30,)$
- سه لایهٔ پنهان با تعداد نورون‌های ۶۴، ۳۲ و ۱۶ و تابع فعال‌ساز ReLU
- یک لایهٔ خروجی با یک نورون و تابع فعال‌ساز Linear

پارامترهای آموزش:

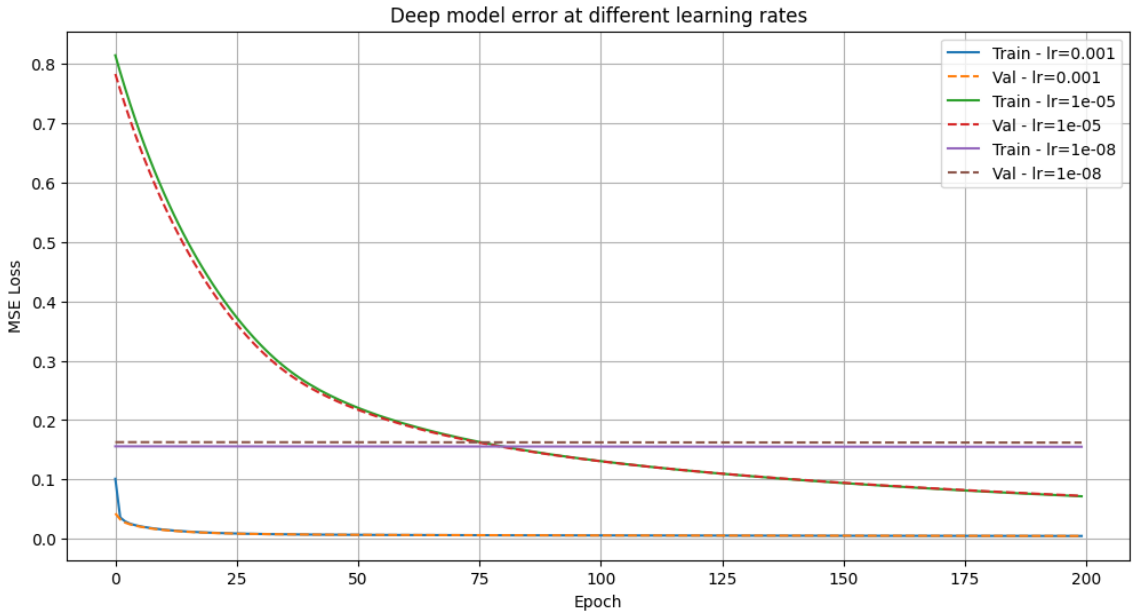
- تابع هزینه: MSE
- بهینه‌ساز: SGD
- تعداد دوره آموزش: ۲۰۰
- نرخ‌های یادگیری: 10^{-3}، 10^{-5}، 10^{-8}

نتایج عملکرد مدل عمیق:

MSE Test	MSE Train	Rate Learning
0.004884	0.004850	10^{-3}
0.072507	0.071538	10^{-5}
0.162131	0.155102	10^{-8}

جدول ۲: مقدار خطای MSE برای مدل عمیق در نرخ‌های یادگیری مختلف

نمودار خطای آموزش و اعتبارسنجی:



شکل ۴: مقایسه خطای مدل عمیق برای نرخ‌های یادگیری مختلف

تحلیل عملکرد:
نتایج حاصل از مدل عمیق نشان می‌دهد که افزایش عمق شبکه باعث بهبود عملکرد مدل در نرخ یادگیری مناسب شده است. شبکه عمیق با نرخ یادگیری 10^{-3} توانسته است بهترین دقت را در بین همه مدل‌ها داشته باشد و به سرعت به کمترین خطا همگرا شود.
در نرخ‌های پایین‌تر، مدل همچنان قادر به یادگیری است، اما با دقت کمتر و سرعت بسیار پایین‌تر. نرخ 10^{-8} همانند قبل، باعث کند شدن بیش از حد روند یادگیری شده و مدل نتوانسته بهینه‌سازی مناسبی انجام دهد.
در مجموع، می‌توان نتیجه گرفت که عمق بیشتر شبکه در صورتی مفید است که با نرخ یادگیری مناسب ترکیب شود؛ در غیر این صورت ممکن است حتی باعث افزایش خطا یا کندی یادگیری شود.

در مرحله بعد، مدل‌های دیگری مانند $\mathrm{KNN}, \mathrm{CNN}$ نیز برای این مسئله بررسی خواهند شد تا مقایسه نهایی بین الگوریتم‌ها انجام گیرد.

۶.۲ مقایسه مدل‌های مختلف یادگیری ماشین

در این بخش، مدل‌های یادگیری ماشین مختلف شامل CNN MLP KNN برای مسئله پیش‌بینی دمای شهر TOURS اجرا شده و نتایج آن‌ها با مدل‌های قبلی مقایسه می‌شود.

۱.۶.۲ مدل K-Nearest Neighbors (KNN)

مدل KNN یک الگوریتم ساده و غیرپارامتریک است که در مسائل رگرسیون نیز کاربرد دارد. این مدل از فاصلهٔ بین نمونهٔ ورودی و داده‌های آموزش برای پیش‌بینی خروجی استفاده می‌کند. در این پروژه، از نسخهٔ رگرسیون KNN برای پیش‌بینی دمای روز ششم استفاده شده است.
مشخصات مدل:

- تعداد همسایه‌ها: $k = 3$
- معیار فاصله: $\mathrm{Euclidean\ Distance}$
- ساختار ورودی: بردار تخت‌شده از ۵ روز متوالی ویژگی‌های TOURS (۶ ویژگی در هر روز)

نتایج مدل :KNN

شاخص خطا	مقدار
Train MSE	0.002338
Test MSE	0.005013

جدول ۳: نتایج مدل KNN برای پیش‌بینی دمای TOURS

تحلیل:
نتایج حاصل از اجرای مدل KNN نشان می‌دهد که این الگوریتم با وجود سادگی، توانسته است عملکرد قابل قبولی در پیش‌بینی دمای آیندهٔ TOURS ارائه دهد. خطای آموزش نسبتاً پایین است و مدل در آزمون نیز دقت مناسبی دارد. این موضوع نشان می‌دهد که ساختار زمانی داده و الگوهای محلی در این مسئله به‌خوبی با روش KNN قابل مدل‌سازی هستند.

در ادامه، مدل‌های دیگر مانند MLP نیز پیاده‌سازی شده و با این مدل مقایسه خواهند شد.
TOURS

۲.۶.۲ مدل Multi-Layer Perceptron (MLP)

مدل MLP یکی از رایج‌ترین ساختارهای شبکه عصبی پیش‌خور است که در بسیاری از مسائل پیش‌بینی و طبقه‌بندی کاربرد دارد. این مدل قابلیت یادگیری نگاشت‌های غیرخطی را با استفاده از چندین لایه پنهان فراهم می‌کند.

در این پروژه، مدل MLP برای پیش‌بینی دمای روز آینده در شهر TOURS پیاده‌سازی شده است.

مشخصات مدل:

- ورودی TOURS: بردار ۳۰ تایی شامل ۵ روز × ۶ ویژگی
- لایه‌های پنهان:
  - Dense(۶۴) با ReLU
  - Dense(۳۲) با ReLU
- لایه خروجی: Dense(۱) با Linear
- تابع خطا: MSE
- بهینه‌ساز: Adam با نرخ یادگیری 0.001
- تعداد دوره آموزش: 200 epochs

نتایج مدل MLP:

شاخص خطا	مقدار
Train MSE	0.002419
Test MSE	0.003005

جدول ۴: مقادیر خطای مدل MLP

تحلیل:
نتایج حاصل از مدل MLP نشان می‌دهد که این مدل توانسته است دقت خوبی در پیش‌بینی دمای آینده داشته باشد. دقت آن در حد مدل KNN و حتی بهتر از نسخه کلاسیک شبکه عصبی (۲.۳) و مدل MLR است. ترکیب ساختار شبکه با ReLU و استفاده از بهینه‌ساز Adam موجب شده تا مدل به‌سرعت همگرا شود و از overfitting نیز جلوگیری گردد.

در مرحله بعد، مدل CNN بررسی خواهد شد (در صورت امکان).

جمع‌بندی نهایی و مقایسه الگوریتم‌ها

در جدول زیر، خلاصه‌ای از نتایج مدل‌های پیاده‌سازی‌شده برای پیش‌بینی دمای شهر TOURS آورده شده است. برای هر مدل، مقدار خطای MSE بر روی داده‌های آموزش و آزمون درج شده است:

MSE Test	MSE Train	مدل
0.006868	0.007184	MLR
0.005013	0.002338	KNN
0.003005	0.002419	MLP
0.004884	0.004850	شبکهٔ عصبی عمیق (Deep NN)

جدول ۵: مقایسه نهایی عملکرد مدل‌های مختلف

تحلیل و نتیجه‌گیری نهایی:
با توجه به مقادیر جدول بالا، می‌توان نتیجه گرفت که:

- مدل MLP با ساختار مناسب و بهینه‌سازی Adam، کمترین خطای آزمون را به‌دست آورده است.
- مدل KNN نیز با وجود سادگی، عملکرد بسیار خوبی داشته و نزدیک به MLP ظاهر شده است.
- مدل MLR عملکرد مناسبی دارد اما نسبت به روش‌های یادگیری عمیق دقت پایین‌تری دارد.
- شبکه عصبی عمیق (Deep NN) عملکرد خوبی دارد ولی به اندازه MLP دقیق نبوده؛ علت می‌تواند به تنظیمات ثابت و تعداد پارامتر بیشتر نسبت داده شود.

در مجموع، مدل MLP را می‌توان به عنوان بهترین گزینه برای این مسئله (در قالب پیاده‌سازی‌های انجام‌شده) معرفی کرد.

در صورت در دسترس بودن داده‌های چندشهری واقعی (برای پیاده‌سازی کامل Collaborative Learning)، مدل‌های ترکیبی نیز می‌توانند به دقت بالاتری دست یابند.

۳ پرسش سه

در این بخش ابتدا تصاویر مربوط به پنج حرف از الفبای فارسی به‌عنوان داده‌های اصلی در نظر گرفته می‌شود. این تصاویر به‌صورت رنگی و با فرمت JPEG در اختیار هستند. هدف از این مرحله، پیش‌پردازش داده‌های تصویری و آماده‌سازی آن‌ها برای ورود به شبکه عصبی است. نخستین گام در این مسیر، تبدیل تصاویر رنگی به بردارهایی باینری است که تنها شامل مقادیر عددی ساده باشند.

در فرآیند تبدیل، تصویر به مقادیر عددی براساس روشنایی پیکسل‌ها تحلیل می‌شود؛ به این معنا که رنگ‌های روشن (پیکسل‌هایی که مجموع کانال‌های قرمز، سبز و آبی آن‌ها زیاد است) به مقدار 1- و رنگ‌های تیره (که مجموع RGB آن‌ها کم است) به مقدار 1 نگاشت می‌شوند. این عملیات باعث می‌شود داده‌های تصویری به‌شکل بردارهایی با مقادیر گسسته و قابل‌پردازش برای مدل تبدیل شوند.

تبدیل تصویر به بردار باینری

```python
def convertImageToBinary(path: str, factor: int=100, output_path : str = None):
    with Image.open(path) as img:
        image = img.convert("RGB")
    img_array = np.array(image)
    intensity = img_array.sum(axis=2)
    threshold = ((255 + factor) // 2) * 3
    binary_representation = np.where(intensity > threshold, -1, 1).flatten()
    if output_path:
      binarized_array = np.where(intensity > threshold, 255, 0)
      binarized_array = np.stack([binarized_array]*3, axis=-1)
      binarized_image = Image.fromarray(binarized_array.astype("uint8"), "RGB")
      binarized_image.save(output_path, 'JPEG')
    return binary_representation
```

تابع convertImageToBinary با استفاده از یک آستانه‌گذاری ساده روی شدت روشنایی کل پیکسل‌ها، تصویر را به برداری باینری تبدیل می‌کند که در آن مقادیر 1 نمایانگر نواحی تیره (حروف) و 1- نمایانگر نواحی روشن (پس‌زمینه) هستند. این آستانه براساس پارامتر factor تعیین می‌شود و مقدار پیش‌فرض آن ۱۰۰ در نظر گرفته شده است.

در صورت فعال بودن پارامتر output_path، نسخه‌ی باینری شده‌ی تصویر نیز به‌صورت فایل تصویری ذخیره می‌شود. خروجی نهایی تابع، یک بردار یک‌بعدی شامل مقادیر 1- و 1 است که نماینده‌ی تصویر اصلی در فضای ویژگی‌های ساده‌شده می‌باشد و مستقیماً قابل استفاده در مدل‌های یادگیری نظیر شبکه هامینگ است.

افزودن نویز به تصویر و ذخیره تصویر نویزی

```
def getNoisyBinaryImage(input_path, output_path, noise_factor=10000000,
    random_state=None):
    with Image.open(input_path) as img:
        image = img.convert("RGB")
    img_array = np.array(image)
    if random_state is not None:
        np.random.seed(random_state)
    rand = np.random.randint(-noise_factor, noise_factor, img.size[0:2])
    red = img_array[:, :, 0] + rand
    red[red > 255] = 255
    red[red < 0] = 0
    green = img_array[:, :, 1] + rand
    green[green > 255] = 255
    green[green < 0] = 0
    blue = img_array[:, :, 2] + rand
    blue[blue > 255] = 255
    blue[blue < 0] = 0
    noise_image = np.stack((red, green, blue), axis=2)
    noise_image = Image.fromarray(noise_image.astype("uint8"), "RGB")
    noise_image.save(output_path, "JPEG")
```

در فرایند یادگیری شبکه‌های عصبی، یکی از مسائل مهم بررسی میزان تاب‌آوری مدل در برابر داده‌های واقعی است که معمولاً با نویز همراه‌اند. در شرایط دنیای واقعی، داده‌های تصویری ممکن است به دلایل متعددی دچار اختلال شوند—از جمله نویز حسگر، فشرده‌سازی نامناسب تصویر، یا خطاهای انتقال. بنابراین، برای آنکه ارزیابی مدل دقیق‌تر و کاربردی‌تر باشد، باید نسخه‌هایی نویزی از داده‌ها نیز در فرآیند آموزش یا تست شبکه گنجانده شوند.

تابع getNoisyBinaryImage با هدف شبیه‌سازی چنین شرایطی طراحی شده است. این تابع یک تصویر رنگی را گرفته و به‌صورت کنترل‌شده به آن نویز اضافه می‌کند. سازوکار افزودن نویز بدین‌صورت است که به هر یک از کانال‌های رنگی تصویر (RGB) یک مقدار تصادفی افزوده می‌شود. این مقادیر از یک بازهٔ مشخص (وابسته به noise_factor) به‌صورت یکنواخت و تصادفی تولید می‌شوند. شدت نویز با این پارامتر قابل تنظیم است؛ هر چه noise_factor بزرگ‌تر باشد، تغییرات اعمال‌شده روی تصویر نیز بیشتر خواهد بود.

برای آنکه نویز به‌شکل بازتولیدپذیر باشد—یعنی اگر آزمایش دوباره تکرار شد، نتایج مشابهی حاصل شود—تابع از پارامتر random_state نیز پشتیبانی می‌کند. در صورت تعیین این مقدار، توزیع تصادفی ثابت باقی خواهد ماند و این امکان فراهم می‌شود که چندین بار با شرایط کاملاً مشابه آزمایش تکرار شود.

در پایان، مقادیر جدید RGB که ممکن است خارج از بازه مجاز پیکسل (۰ تا ۲۵۵) باشند، به‌کمک عملیات clip به این بازه محدود می‌شوند و تصویر نهایی با نویز ایجادشده، در قالب فایل JPEG ذخیره می‌شود. این خروجی‌ها به‌صورت داده‌های نویزی واقع‌گرایانه، مستقیماً قابل استفاده در ارزیابی شبکه هستند.

ساخت مجموعه‌ای از تصاویر نویزی

```
def generateNoisyImages(noise_factor=10000000, random_state = None):
    image_paths = ["1.jpg", "2.jpg", "3.jpg", "4.jpg", "5.jpg"]
    os.makedirs('noisy', exist_ok=True)
    for i, image_path in enumerate(image_paths, start=1):
        noisy_image_path = f"noisy/noisy{i}.jpg"
        getNoisyBinaryImage(image_path, noisy_image_path, noise_factor,
   random_state)
        print(f"Noisy image for {image_path} saved as {noisy_image_path}")
```

این تابع با هدف تولید مجموعه‌ای منظم از تصاویر نویزی طراحی شده است. برای ارزیابی دقیق عملکرد یک مدل یادگیری ماشین در شرایط واقعی، نیاز داریم مجموعه‌ای از ورودی‌های مختل‌شده توسط نویز داشته باشیم. به همین دلیل، در این مرحله از پروژه پنج تصویر اصلی (نمایانگر پنج حرف الفبای فارسی) به عنوان پایه انتخاب می‌شوند و روی هر کدام نویز مصنوعی اعمال می‌گردد.
سازوکار این تابع به این صورت است که ابتدا لیستی از مسیر تصاویر اولیه تعریف می‌شود. سپس با استفاده از حلقه‌ای تکرارشونده، مسیر جدیدی برای ذخیره‌سازی هر تصویر نویزی در پوشه‌ای به نام noisy تعریف شده و سپس نویز به تصویر افزوده می‌شود. تابع getNoisyBinaryImage که در بخش قبل توضیح داده شد، برای این منظور فراخوانی می‌شود.
از آنجا که ایجاد این تصاویر باید به‌صورت ساختاریافته و قابل‌کنترل باشد، دو پارامتر noise_factor و random_state به این تابع نیز منتقل می‌شوند. اولی شدت نویز را تعیین می‌کند و دومی امکان بازتولید نویز مشابه در دفعات مختلف را فراهم می‌سازد. در نهایت، پنج تصویر نویزی با نام‌هایی مشخص در پوشه‌ای مجزا ذخیره می‌شوند که در ادامه به‌عنوان دادهٔ ورودی مدل مورد استفاده قرار خواهند گرفت.

«پوشهٔ نویزی» یعنی چه؟ به‌منظور حفظ نظم و ساختار فایل‌ها در پروژه، نسخه‌های نویزی‌شدهٔ تصاویر اولیه در پوشه‌ای جداگانه به نام noisy ذخیره شده‌اند. این رویکرد باعث می‌شود تصاویر اصلی (بدون نویز) و تصاویر دست‌کاری‌شده (با نویز) از یکدیگر تفکیک شوند و فرآیندهای پردازش، تحلیل و ارزیابی شبکه به‌صورت هدفمند و بدون تداخل انجام گیرد.
ساخت پوشهٔ noisy در ابتدای اجرای تابع generateNoisyImages انجام می‌شود و در آن، فایل‌هایی مانند noisy1.jpg تا noisy5.jpg قرار می‌گیرند. این نام‌گذاری و دسته‌بندی به ما کمک می‌کند تا در مراحل بعدی مانند تست شبکه عصبی، به‌طور مستقیم به مجموعه تصاویر نویزی دسترسی داشته باشیم، بدون آنکه نگران دستکاری تصاویر اصلی باشیم.

۱.۲.۳ طراحی شبکه عصبی و بررسی تأثیر نویز

در این پروژه از یک شبکه عصبی ساده به نام Hamming Network استفاده شده است که بر پایهٔ محاسبهٔ شباهت بین بردار ورودی و بردارهای مرجع عمل می‌کند. این شبکه به‌جای استفاده از لایه‌های پنهان یا گرادیان، از مفهوم ضرب داخلی (dot product) برای تشخیص بیشترین تطابق بین ورودی و الگوهای آموزش‌دیده استفاده می‌کند.
ایده اصلی پشت این مدل آن است که اگر داده ورودی (حتی در حالت نویزی) به اندازه کافی به یکی از نمونه‌های مرجع نزدیک باشد، مقدار ضرب داخلی بین این دو بردار بیشتر از بقیه خواهد بود. از این طریق، شبکه می‌تواند نزدیک‌ترین الگو را انتخاب کرده و به عنوان پیش‌بینی نهایی خروجی دهد.
طراحی ساده و محاسبات سریع این شبکه باعث می‌شود برای پروژه‌هایی با داده‌های نسبتاً ثابت، نویز محدود و نیاز به تفسیر سریع، انتخاب مناسبی باشد. در این آزمایش، هدف بررسی عملکرد این شبکه در مواجهه با داده‌های نویزی است، تا میزان مقاومت آن در برابر اختلال بررسی شود.

تعریف کلاس شبکهٔ هامینگ

```
class HammingNetwork:
    def __init__(self, patterns):
        self.patterns = np.array([p for p in patterns])
        self.n_classes = len(patterns)

    def predict(self, x_noisy):
        similarities = np.dot(self.patterns, x_noisy)
        return self.patterns[np.argmax(similarities)]
```

کلاس HammingNetwork یک مدل ساده برای شناسایی الگوهاست که در آن نیازی به آموزش یا تنظیم وزن‌ها نیست. در ابتدا، فقط کافی است نمونه‌های مرجع (مثل تصاویر اصلی بدون نویز) به شبکه داده شوند. سپس در مرحلهٔ پیش‌بینی، بردار ورودی که ممکن است نویزی باشد، با تمام الگوهای ذخیره‌شده مقایسه می‌شود.

این مقایسه با استفاده از ضرب داخلی انجام می‌شود؛ یعنی میزان شباهت عددی بین بردار ورودی و هر الگوی مرجع محاسبه می‌گردد. الگویی که بالاترین مقدار شباهت را داشته باشد، به عنوان نتیجه نهایی انتخاب می‌شود.
دلیل استفاده از این مدل در پروژه، سادگی در پیاده‌سازی و سرعت در پردازش است. در عین حال، این شبکه می‌تواند در برابر نویزهای محدود، عملکرد مناسبی داشته باشد و برای بررسی اولیهٔ مقاومت نسبت به نویز، انتخاب قابل قبولی است.

آماده‌سازی مدل و داده‌های نویزی

```
images = [convertImageToBinary(f'{i}.jpg') for i in range(1,6)]
model = HammingNetwork(images)

generateNoisyImages(1100, 93)
paths = [f'noisy/noisy{i}.jpg' for i in range(1,6)]
noise = [convertImageToBinary(f'{i}') for i in paths]
```

در این مرحله، ابتدا تصاویر اصلی پنج حرف فارسی به کمک تابع convertImageToBinary به بردارهای باینری تبدیل می‌شوند. این بردارها به‌عنوان الگوهای مرجع، به شبکهٔ هامینگ داده می‌شوند تا پایهٔ مقایسه و تشخیص در مرحلهٔ پیش‌بینی باشند.

در ادامه، با استفاده از تابع generateNoisyImages نسخه‌هایی نویزی از همین تصاویر تولید می‌شود. شدت نویز با پارامتر noise_factor کنترل می‌شود و مقدار random_state نیز برای تولید نویز بازتولیدپذیر تعیین شده است. تصاویر نویزی در مسیر مشخصی ذخیره می‌شوند.

در پایان، داده‌های نویزی تولیدشده نیز با استفاده از همان تابع convertImageToBinary به بردارهای باینری تبدیل شده و برای تست شبکه آماده می‌شوند. خروجی این مرحله، دو مجموعهٔ منظم از داده‌ها (مرجع و نویزی) است که در مراحل بعدی مورد تحلیل قرار خواهند گرفت.

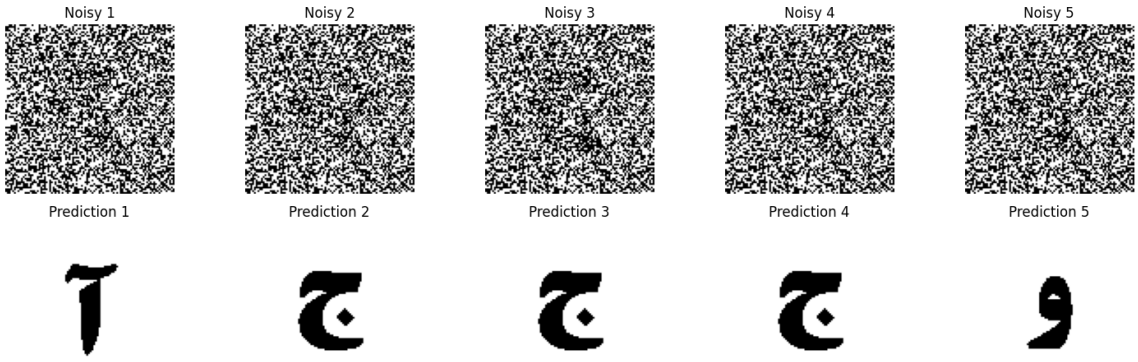
نمایش نتایج شناسایی روی تصاویر نویزی

```python
def BinarytoImage(binary_img):
    binary_img = np.array(binary_img)
    img = np.where(binary_img == -1, 255, 0)
    img = img.reshape(96,96)
    img = np.stack([img]*3, axis=2)
    return Image.fromarray(img.astype("uint8"), "RGB")

def show_images(noise, predictions):
    fig, axes = plt.subplots(2, 5, figsize=(15, 5))
    for i in range(5):
        axes[0, i].imshow(noise[i], cmap='gray')
        axes[0, i].set_title(f"Noisy {i+1}")
        axes[0, i].axis('off')

        axes[1, i].imshow(predictions[i], cmap='gray')
        axes[1, i].set_title(f"Prediction {i+1}")
        axes[1, i].axis('off')
    plt.tight_layout()
    plt.show()

per = [model.predict(noise[i]) for i in range(5)]
noise_img = [BinarytoImage(noise[i]) for i in range(5)]
per_img = [BinarytoImage(per[i]) for i in range(5)]
show_images(noise_img, per_img)
```



شکل ۵: عملکرد شبکه هامینگ روی تصاویر نویزی. ردیف اول: داده‌های نویزی پنج حرف «آ، ب، ج، د، و». ردیف دوم: پیش‌بینی‌های مدل.

تحلیل عملکرد: نتایج به‌دست‌آمده از اجرای شبکه روی تصاویر نویزی نشان می‌دهند که مدل در شرایطی با نویز کم تا متوسط، توانسته است اغلب حروف را به‌درستی شناسایی کند. با این حال، افزایش سطح نویز موجب کاهش دقت در پیش‌بینی‌ها شده و در برخی موارد، خروجی نهایی با حرف واقعی مطابقت نداشته است.

این موضوع بیانگر آن است که شبکه نسبت به نویز حساس است و تنها زمانی عملکرد مطلوب دارد که الگوی نویزی همچنان شباهت کافی به نمونه مرجع داشته باشد. بنابراین، هرچه شدت نویز بیشتر شود، احتمال تشخیص نادرست نیز افزایش می‌یابد.

تابع BinarytoImage برای بازسازی تصویر از بردار باینری استفاده می‌شود. در مرحلهٔ پیش‌پردازش، تصاویر به بردارهایی با مقادیر 1 (برای پیکسل‌های سیاه) و 1- (برای پیکسل‌های سفید) تبدیل شده‌اند. این تابع دقیقاً روند معکوس را انجام می‌دهد: ابتدا بردار باینری به یک آرایهٔ دوبعدی 96×96 تبدیل شده، سپس مقادیر 1- به عدد 255 (سفید) و مقادیر 1 به صفر (سیاه) نگاشت می‌شوند تا تصویر قابل نمایش باشد. در پایان، این آرایه به یک تصویر رنگی RGB با سه کانال یکسان تبدیل و به‌صورت شیء تصویر بازگردانده می‌شود.
تابع دوم، show_images وظیفهٔ نمایش گرافیکی نتایج پیش‌بینی را بر عهده دارد. این تابع از کتابخانه matplotlib برای رسم دو ردیف تصویر استفاده می‌کند: ردیف اول تصاویر نویزی ورودی و ردیف دوم پیش‌بینی‌های شبکه برای هر تصویر. در هر ستون از نمودار، تصویر نویزی و پیش‌بینی‌شدهٔ مربوط به یک حرف قرار داده می‌شوند تا امکان مقایسهٔ دیداری فراهم گردد. محورهای نمایش داده نمی‌شوند تا تمرکز روی شکل حروف باقی بماند.
در بخش انتهایی کد، ابتدا با استفاده از تابع model.predict برای هر تصویر نویزی یک خروجی پیش‌بینی‌شده تولید می‌شود. سپس هر دو مجموعه (ورودی و خروجی) با تابع BinarytoImage به تصاویر قابل مشاهده تبدیل شده و در نهایت با show_images نمایش داده می‌شوند. این فرآیند کمک می‌کند عملکرد شبکه در مواجهه با داده‌های نویزی به‌صورت بصری و ملموس تحلیل شود.

۱.۳.۳ تحلیل عملکرد شبکه در حضور Missing Point

در این بخش، عملکرد شبکه در شرایطی بررسی می‌شود که برخی از پیکسل‌های کلیدی تصاویر ورودی به‌صورت تصادفی حذف شده‌اند (Missing Point). این وضعیت می‌تواند نمایانگر خرابی یا دستکاری در داده‌های ورودی باشد. هدف از این مرحله، بررسی میزان تحمل شبکه نسبت به حذف بخشی از اطلاعات و ارائه راه‌حلی برای افزایش مقاومت آن است.

تابع حذف تصادفی درصدی از پیکسل‌های مشکی Missing Point

```
def missimage(input_path, output_path, percent=20, factor=100, random_state=None):
    with Image.open(input_path) as img:
        image = img.convert("RGB")

    img_array = np.array(image)
    intensity = img_array.sum(axis=2)
    threshold = ((255 + factor) // 2) * 3
    blackpix = np.where(intensity < threshold)
    number = round(len(blackpix[0]) * percent / 100)

    if random_state is not None:
        np.random.seed(random_state)

    pixels_to_flip = np.random.choice(range(len(blackpix[0])), size=number, replace=False)
    img_array[blackpix[0][pixels_to_flip], blackpix[1][pixels_to_flip]] = 255
    img = Image.fromarray(img_array.astype("uint8"), "RGB")
    img.save(output_path, "JPEG")
```

این تابع برای شبیه‌سازی حالت از بین رفتن بخشی از اطلاعات تصویری طراحی شده است. در آن، بخشی از پیکسل‌های مشکی که معرف بخش‌های اصلی حرف هستند، به‌صورت تصادفی به رنگ سفید تغییر داده می‌شوند. این تغییر باعث حذف بخشی از ساختار بصری حرف شده و شرایطی مشابه «نقاط گمشده» یا Missing Point را ایجاد می‌کند که در آن بخشی از داده‌های ورودی ناقص هستند.

تولید دسته‌ای از تصاویر Missing Point برای آزمون

```
def generatemissimage(percent=20, factor=100, random_state=None):
    image_paths = ["1.jpg", "2.jpg", "3.jpg", "4.jpg", "5.jpg"]
    os.makedirs('miss', exist_ok=True)
    for i, image_path in enumerate(image_paths, start=1):
        miss_image_path = f"miss/miss{i}.jpg"
        missimage(image_path, miss_image_path, percent, factor,
    random_state)
        print(f"Missing image for {image_path} saved as {miss_image_path}")
```

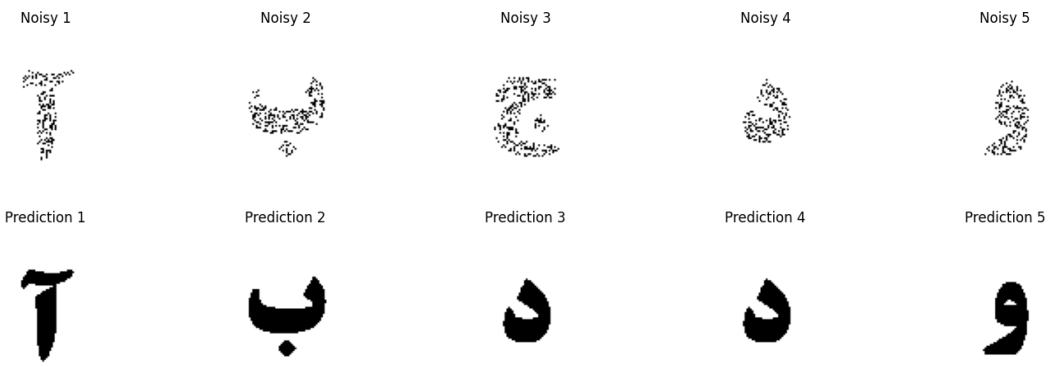
آزمایش: برای آزمایش شبکه، تصاویر با Missing Point تا سطح ۷۰٪ تولید شده و به مدل داده شدند.

ارزیابی نسخه ساده شبکه روی داده‌های Missing Point

```
generatemissimage(70, 100, 93)
paths = [f'miss/miss{i}.jpg' for i in range(1, 6)]
miss = [convertImageToBinary(i) for i in paths]

model = HammingNetwork(images)

per = [model.predict(miss[i]) for i in range(5)]
miss_img = [BinarytoImage(miss[i]) for i in range(5)]
per_img = [BinarytoImage(per[i]) for i in range(5)]
show_images(miss_img, per_img)
```



شکل ۶: عملکرد نسخه ساده شبکه هامینگ روی تصاویر دارای Missing Point. کاهش دقت در برخی حروف مانند «ج» و «د» مشهود است.

بازنویسی شبکه برای درک نقاط Missing

```
class HammingNetwork:
    def __init__(self, patterns):
        self.patterns = np.array([p for p in patterns])
        self.n_classes = len(patterns)

    def masked_dot(self, a, b, mask):
        return np.dot(a[mask], b[mask])

    def predict(self, x_noisy):
        mask = x_noisy != -1
        similarities = [self.masked_dot(p, x_noisy, mask) for p in self.
    patterns]
        return self.patterns[np.argmax(similarities)]
```

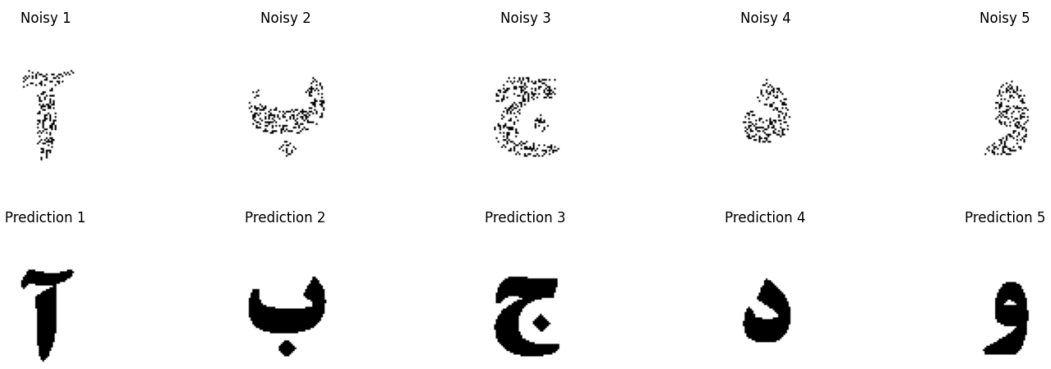
در نسخهٔ بازنویسی‌شدهٔ شبکه، از یک mask برای مشخص کردن نواحی معتبر تصویر استفاده می‌شود. در این ساختار، پیکسل‌هایی که مقدارشان برابر با 1- است به‌عنوان نقاط حذف‌شده (Missing Point) در نظر گرفته شده و در محاسبات نادیده گرفته می‌شوند. به این ترتیب، مقایسهٔ بردار ورودی با الگوهای مرجع تنها بر اساس پیکسل‌های واقعی انجام می‌شود. این روش باعث می‌شود شبکه بتواند در شرایطی که بخشی از تصویر ناقص است، همچنان تصمیم‌گیری دقیقی داشته باشد.

ارزیابی شبکه روی تصاویر Missing Point

```
generatemissimage(70, 100, 93)
paths = [f'miss/miss{i}.jpg' for i in range(1, 6)]
miss = [convertImageToBinary(i) for i in paths]

model = HammingNetwork(images)

per = [model.predict(miss[i]) for i in range(5)]
miss_img = [BinarytoImage(miss[i]) for i in range(5)]
per_img = [BinarytoImage(per[i]) for i in range(5)]
show_images(miss_img, per_img)
```



شکل ۷: عملکرد نسخه مقاوم‌شدهٔ شبکه (با استفاده از ماسک) در برابر داده‌های Missing Point. دقت شبکه نسبت به نسخه ساده بهبود یافته است.

تحلیل عملکرد: با افزایش درصد حذف پیکسل‌ها، عملکرد نسخه ساده شبکه به‌شدت افت می‌کند و توانایی تشخیص خود را از دست می‌دهد. اما نسخه بهبودیافته (با استفاده از mask) می‌تواند حتی با ۷۰٪ حذف اطلاعات، برخی حروف را به‌درستی شناسایی کند. این نشان‌دهنده اهمیت پردازش مقاوم به داده‌های ناقص است.

جمع‌بندی نهایی

در این پروژه، ابتدا داده‌های تصویری از پنج حرف فارسی استخراج شدند. سپس با تولید نسخه‌های نویزی و حذف نقطه‌ای، یک شبکه هامینگ ساده و نسخه مقاوم آن پیاده‌سازی شد. در آزمایش‌ها مشاهده شد که شبکه نسخه مقاوم به Missing Point عملکرد بسیار بهتری در شرایط حذف داده دارد.

نتیجه‌گیری کلی

در این پروژه، با هدف ارزیابی و تحلیل توانایی یک شبکه عصبی ساده در شناسایی داده‌های تصویری حروف فارسی در شرایط مختلف، سه مرحله‌ی اصلی طی شد:

ابتدا داده‌های تصویری پنج حرف فارسی (آ، ب، ج، د، و) به بردارهای باینری تبدیل شدند. سپس با اعمال نویز کنترل‌شده، نسخه‌های نویزی از این داده‌ها تولید گردید و به عنوان ورودی به شبکه داده شد. در مرحله دوم، یک شبکه عصبی از نوع هامینگ پیاده‌سازی شد که بر اساس شباهت بین بردار ورودی و بردارهای مرجع، حرف مربوطه را تشخیص می‌داد. نتایج نشان داد که شبکه در برابر مقدار کم نویز عملکرد قابل‌قبولی دارد ولی با افزایش نویز، دقت آن کاهش می‌یابد.

در گام سوم، مسئله‌ی Missing Point مطرح شد؛ به‌گونه‌ای که بخشی از اطلاعات تصویر به‌طور تصادفی حذف گردید. شبکه‌ی اولیه قادر به مدیریت این شرایط نبود و دقت آن به‌طور چشم‌گیری افت کرد. اما با بازنویسی تابع پیش‌بینی شبکه و استفاده از مکانیزم masking برای نادیده گرفتن پیکسل‌های نامشخص، شبکه توانست مقاومت بالاتری در برابر داده‌های ناقص از خود نشان دهد.

در مجموع، نتایج به‌دست‌آمده نشان می‌دهند که حتی مدل‌های ساده‌ای همچون شبکه هامینگ، اگر با طراحی مناسب همراه باشند، می‌توانند در شرایط نویزی و ناقص نیز عملکرد مؤثری داشته باشند. این پروژه اهمیت پیش‌پردازش داده و طراحی مقاوم به نقص در شبکه‌های عصبی را به‌خوبی نمایان ساخت.

۴ پرسش چهار

۱. خروجی شبکهٔ عصبی

هدف، محاسبه‌ی احتمال تعلق یک ورودی $X = (X_1, X_2)$ به کلاس مثبت (برچسب ۱) با استفاده از شبکهٔ عصبی داده‌شده است. این شبکه دارای دو نورون در لایهٔ پنهان و یک نورون خروجی است. تابع فعال‌ساز لایهٔ مخفی خطی است و تابع خروجی، سیگموئید است.

ابتدا خروجی نورون‌های لایهٔ مخفی را محاسبه می‌کنیم:

$$Z_1 = W_1X_1 + W_3X_2 + W_5$$
$$Z_2 = W_2X_1 + W_4X_2 + W_6$$

با اعمال تابع فعال‌ساز خطی $h(z) = c \cdot z$، خروجی نورون‌های لایهٔ پنهان به صورت زیر خواهد بود:

$$H_1 = c \cdot Z_1 \qquad H_2 = c \cdot Z_2$$

ورودی به نورون خروجی به‌صورت زیر محاسبه می‌شود:

$$Y' = W_7 + W_8H_1 + W_9H_2$$

و خروجی نهایی پس از اعمال تابع سیگموئید $g(z) = \frac{1}{1+e^{-z}}$ به صورت زیر خواهد بود:

$$P(Y = 1 \mid X, w) =$$

$$\frac{1}{1 + \exp\Big(- \big[W_7 + cW_8(W_1X_1 + W_3X_2 + W_5) + cW_9(W_2X_1 + W_4X_2 + W_6)\big]\Big)}$$

۲. مدل معادل بدون لایهٔ پنهان

با توجه به اینکه تابع فعال‌ساز لایهٔ مخفی خطی است، می‌توان لایهٔ مخفی را با ترکیب وزن‌ها در نورون خروجی از بین برد. در این حالت، شبکه به‌صورت یک پرسپترون ساده درخواهد آمد:

$$P(Y = 1 \mid X, w) =$$
$$\frac{1}{1 + \exp\Big(- \big[W_7 + cW_8W_1X_1 + cW_8W_3X_2 + cW_8W_5 + cW_9W_2X_1 + cW_9W_4X_2 + cW_9W_6\big]\Big)}$$

این فرم نشان می‌دهد که مدل نهایی صرفاً ترکیب خطی از ورودی‌ها X_1, X_2 است که از طریق وزن‌های ترکیبی جدید به شبکه وارد می‌شوند. بنابراین، این مدل معادل یک شبکه بدون لایهٔ پنهان است.

مرز تصمیم

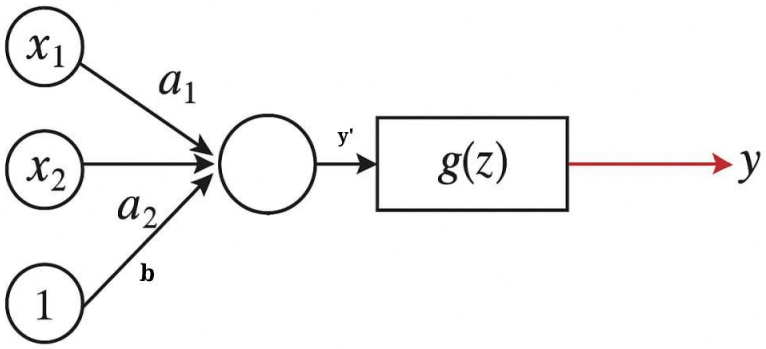
مرز تصمیم زمانی رخ می‌دهد که خروجی برابر با ۰٫۵ باشد. برای تابع سیگموئید، این به معنای صفر شدن عبارت در توان نمایی است:

$$P(Y = 1 \mid X, w) = 0.5 \quad \Leftrightarrow \quad Y' = 0$$

در نتیجه، رابطه‌ی مرز تصمیم به‌صورت زیر خواهد بود:

$$W_7 + cW_8(W_1X_1 + W_3X_2 + W_5) + cW_9(W_2X_1 + W_4X_2 + W_6) = 0$$

که این یک معادله‌ی خطی از مرتبهٔ اول در فضای X_1 و X_2 است و ناحیهٔ جداسازی بین دو کلاس را تعریف می‌کند.



شکل ۸: شبکه عصبی معادل بدون لایهٔ مخفی

تحلیل ساختار شبکه و اثبات معادل‌سازی بدون لایهٔ مخفی

در این بخش ساختار شبکهٔ عصبی نمایش داده‌شده در شکل ۵ فایل مینی پروژه را بررسی می‌کنیم. این شبکه شامل یک لایهٔ پنهان با دو نورون و یک لایهٔ خروجی است. فرض شده است که تابع فعال‌ساز لایهٔ پنهان یک تابع خطی از نوع $h(z) = c \cdot z$ است و خروجی نهایی از طریق تابع سیگموئید $g(z) = \frac{1}{1+e^{-z}}$ محاسبه می‌شود.

ابتدا خروجی نورون‌های لایهٔ پنهان را محاسبه می‌کنیم:

$$Z_1 = W_1 \cdot 1 + W_3 X_1 + W_5 X_2 \qquad Z_2 = W_2 \cdot 1 + W_4 X_1 + W_6 X_2$$

که در آن 1 ورودی بایاس است.

با اعمال تابع فعال‌ساز خطی:

$$H_1 = c \cdot Z_1 \qquad H_2 = c \cdot Z_2$$

اکنون مقدار ورودی به نورون خروجی به صورت زیر محاسبه می‌شود:

$$Y' = W_7 + W_8 H_1 + W_9 H_2$$

و خروجی نهایی با اعمال تابع سیگموئید به شکل زیر خواهد بود:

$$P(Y = 1 \mid X, w) = \frac{1}{1 + \exp\Big(-Y'\Big)}$$

اگر عبارات H_1 و H_2 را جای‌گذاری کنیم، رابطهٔ خروجی نهایی به صورت زیر بازنویسی می‌شود:

$$P(Y = 1 \mid X, w) = \frac{1}{1 + \exp\Big(-\big[W_7 + cW_8(W_1 + W_3 X_1 + W_5 X_2) + cW_9(W_2 + W_4 X_1 + W_6 X_2)\big]\Big)}$$

اثبات امکان معادل‌سازی

از آن‌جا که ترکیب خطی از ترکیب‌های خطی باز هم یک ترکیب خطی است، می‌توان رابطهٔ بالا را به فرم زیر ساده‌سازی کرد:

$$P(Y = 1 \mid X, w) = \frac{1}{1 + \exp\left(-(a_1 X_1 + a_2 X_2 + b)\right)}$$

که در آن:

$$a_1 = c(W_8 W_3 + W_9 W_4), \qquad a_2 = c(W_8 W_5 + W_9 W_6), \qquad b = W_7 + c(W_8 W_1 + W_9 W_2)$$

بنابراین، شبکهٔ فوق که دارای یک لایهٔ پنهان با تابع فعال‌ساز خطی است، معادلی کاملاً مشابه با یک شبکهٔ پرسپترون ساده (بدون لایهٔ پنهان) دارد.

مرز تصمیم

مرز تصمیم زمانی رخ می‌دهد که:

$$P(Y = 1 \mid X, w) = 0.5 \quad \Rightarrow \quad a_1 X_1 + a_2 X_2 + b = 0$$

که این معادله یک خط تصمیم خطی در فضای ویژگی‌هاست.

نتیجه‌گیری: در شبکه‌های عصبی، قدرت یادگیری و مدل‌سازی روابط غیرخطی مستقیماً وابسته به وجود توابع فعال‌ساز غیرخطی در لایه‌های میانی است. اگر لایهٔ پنهان از یک تابع فعال‌ساز خطی مانند $h(z) = c \cdot z$ استفاده کند، کل شبکه—صرف‌نظر از تعداد نورون‌های میانی—در نهایت رفتاری معادل با یک مدل خطی خواهد داشت.

به عبارت دیگر، ترکیب خطی چند تابع خطی، هنوز یک تابع خطی است. بنابراین، اضافه‌کردن لایهٔ پنهان با چنین تابعی صرفاً پیچیدگی محاسباتی شبکه را افزایش می‌دهد، بدون آنکه ظرفیت مدل در یادگیری الگوهای پیچیده‌تر تقویت شود.

در چنین حالتی، می‌توان شبکهٔ اولیه را به‌صورت کامل با یک پرسپترون ساده (تک‌نورونه، بدون لایهٔ مخفی) بازنویسی کرد، بی‌آنکه خللی در عملکرد شبکه ایجاد شود. این امر نشان می‌دهد که برای افزایش واقعی قدرت تفکیک و تعمیم‌پذیری شبکه، به‌کارگیری توابع فعال‌ساز غیرخطی (مانند ReLU، sigmoid یا tanh) در لایه‌های میانی نه‌تنها توصیه‌شده بلکه ضروری است.

از این تحلیل نتیجه می‌گیریم که:

- در غیاب غیرفخطی‌سازی در لایهٔ پنهان، شبکه دچار model underfitting می‌شود.
- این معادل‌سازی می‌تواند به‌عنوان تکنیکی برای architecture optimization جهت کاهش پیچیدگی محاسباتی مورد استفاده قرار گیرد.
- طراحی شبکه باید با درک عمیق از نقش هر مؤلفه صورت گیرد، نه صرفاً افزودن لایه‌ها بدون اثرگذار بودن آن‌ها.